

**课程设计报告**

**题目：** **基于SAT的数独游戏求解程序**

**课程名称： 程序设计综合课程设计**

**专业班级： CS1706**

**学 号： xxx**

**姓 名： xxx**

**指导教师： xxx**

**报告日期： 2019年2月26日**

**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：**将数独游戏[5]问题转化为SAT问题[6-8]，并集成到上面的求解器进行问题求解，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[6-8]。(15%)

**参考文献**

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

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

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

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

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

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

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

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

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

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

[10] Robert Ganian and Stefan Szeider. Community Structure Inspired Algorithms for SAT and #SAT. International Conference on Theory and Applications of Satisfiability Testing(SAT 2015),223-237360

**目录**

[任务书…………………………………………………………………………………I](#_Toc3030599)

[1 引言………………………………………………………………………………2](#_Toc3030600)

[1.1 课题背景与意义 2](#_Toc3030601)

[1.1.1数独游戏背景 2](#_Toc3030602)

[1.1.2 SAT问题简介 2](#_Toc3030603)

[1.2 国内外研究现状 3](#_Toc3030604)

[1.3 课程设计的主要研究工作 4](#_Toc3030605)

[2 系统需求分析与总体设计…………………………………………………………5](#_Toc3030606)

[2.1 系统需求分析 5](#_Toc3030607)

[2.2 系统总体设计 5](#_Toc3030608)

[3 系统详细设计………………………………………………………………………8](#_Toc3030609)

[3.1 数据结构定义 8](#_Toc3030610)

[3.2 主要算法设计 9](#_Toc3030611)

[3.2.1 CNF文件处理 9](#_Toc3030612)

[3.2.2 DPLL算法 9](#_Toc3030613)

[3.2.3 SuDoKu问题转化SAT问题 11](#_Toc3030614)

[3.2.4挖洞法生成唯一解数独棋盘 12](#_Toc3030615)

[4 系统实现与测试………………………………………………………………..…14](#_Toc3030616)

[4.1 系统实现 14](#_Toc3030617)

[4.1.1系统功能 14](#_Toc3030618)

[4.1.2数据结构 14](#_Toc3030619)

[4.1.3函数声明 16](#_Toc3030620)

[4.2 系统测试 16](#_Toc3030621)

[5 总结与展望………………………………………………………………………..26](#_Toc3030622)

[5.1 全文总结 26](#_Toc3030623)

[5.1 工作展望 26](#_Toc3030624)

[6 体会………………………………………………………………………………..29](#_Toc3030625)

# 1 引言

## 1.1 课题背景与意义

数独（Sudoku）是一个放置难题，同时也是运用纸和笔进行演算和推理的逻辑游戏，从发展至今，由于它具有开发智力，锻炼逻辑思维的优点，被大家所认识以及喜爱。数独游戏在世界上广受欢迎，但进入中国的时间并不长，但是得到了中国广泛群体的热爱，发展迅猛，许多数独俱乐部的诞生、数独游戏活动被大量的举办。如今在数独游戏协会、众多的民间数独社团和广大爱好者的共同努力下，将会更大程度上促进数独游戏在中国的推广与普及。

SAT 问题在人工智能领域的重要地位，使得许多学者都在 SAT 问题求解领域做了大量的研究，可满足性问题进而也成为了国内外研究的热点问题，并在算法研究和技术实现上取得了较大的突破，这也推动了形式验证和人工智能等领域的发展。在SAT 求解器被越来越多地应用到各种实际问题领域的今天，探寻解决 SAT 问题的高效算法仍然是一个吸引人并且极具挑战性的研究方向。

### 1.1.1数独游戏发展背景

数独游戏的前身为“九宫格”，最早起源于中国东汉末年。十八世纪末，瑞士数学家莱昂哈德·欧文发明了一种叫做“拉丁方块”的游戏，数独类游戏经过一系列的发展，终于在2004年真正的为世界所知晓。

数独游戏的规则很简单，在给定的一个由3×3子网格组成的9×9网格区域中，其中在某些单元格中给出了一些数字，目的是在网格中的每个单元格中输入数字1到9，以便每个行，列和子网格只包含1到9不重复的数字。

### 1.1.2 SAT问题简介

命题逻辑公式的可满足性问题（SAT），是数理逻辑、自动推理、计算机科学、集成电路的设计验证和人工智能等领域中的基本问题，是一个典型的NP完全问题。SAT 问题在计算复杂性理论中具有非常重要的地位，设计并实现解决该类问题的 高效算法意义重大。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。

## 1.2 国内外研究现状

最经典的求解 SAT 问题的完备算法是 DPLL 算法，它是由 Davis 和 Putnam 等 人在 1960 年提出，其它的完备算法大都是在 DPLL 算法的基础上衍生出来的， 是对 DPLL 算法的改进。由于 SAT 问题本身的特性使得其最坏情况下的时间复杂 度是指数级别，最初这使得许多的研究者望而却步。而后，S.A.Cook 在 1971年证明了SAT 问题是 NP 完全问题，这更加削弱了许多学者研究 SAT 问题的兴趣，从而导致了SAT 问题在很长的一段时间里都没有得到较好的重视，发展非常缓慢， 研究成果较少。如今每年可满足性理论和应用方面的国际会议都会组织一次 SAT 竞赛以求找到一组最快的 SAT 求解器，而且会详细展示一系列的高效求解器的性能。

Bart Selman和Henry Kautz分别于1997年和2003年在人工智能第五届国际合作会议上提出了 SAT 问题面临的十大挑战性问题，并在 2001 年和 2007 年先后 对当时的可满足性问题现状进行了全面的阐述和总结。这十大挑战性问题的提出 对于 SAT 基准问题的理论研究和算法改进都起到了强有力的推动作用。

这使得越来越多的人开始关注并研究 SAT 问题，所以这段时间也涌现出了众多新的高效的 SAT 算法如 MINISAT、SATO、CHAFF、POSIT和 GRASP等，SAT 算法的研究成果显著，求解算法也越来越多地应用到了实际问题领域。这些 新兴的算法大都是基于 DPLL 算法的改进算法，改进的方面包括：采用新的数据结构、新的变量决策策略或者新的快速的算法实现方案。国内也涌现出了许多高效的求解算法，如 1998 年作者梁东敏提出了改进的子句加权 WSAT 算法，2000 年金人超和黄文奇提出的并行 Solar 算法，2002 年作者张德富在文献中，提出模拟退火算法。

尽管 SAT 算法已经取得了举足轻重的改进，但是仍有一些问题没有得到高效的解决，已经解决的问题可能还存在更好的求解算法，因此研究并实现高效率的 求解算法仍是当前要解决的中心问题之一。

## 1.3 课程设计的主要研究工作

本实验主要学习命题逻辑可满足性问题的相关理论知识，基于 DPLL 的求解器所采用的关键技术及其算法框架进行了研究与实现。

具体来说，本文的主要研究内容如下：

1.对 SAT 问题的研究背景、意义及现状进行了简要总结，学习了命题逻辑可满足性问题的基本理论知识以及挖洞法生成唯一解数独棋盘的基本理论知识。

2.对基于DPLL的SAT 算法进行了深入研究，并仔细分析了 DPLL 算法的实现，对该算法中所使用到的数据结构和一些关键技术给予总结和分析。

3.实现基于DPLL算法实现的SAT求解器，并对系统的算法以及数据结构进行优化，并给出求解对应的SAT问题所计算出来的优化率。

4.将改进后的算法应用到现实中的实际问题—数独游戏。通过编码转换为 CNF 公式的形式后，输入到改进后的 DPLL 求解器中进行求解，并利用挖洞法生成具有唯一解数独棋盘。

# 2 系统需求分析与总体设计

## 2.1 系统需求分析

数独游戏在近些年来得到高速的发展以及广泛的传播和认可，但是对于一些数独游戏的求解时间问题仍旧存在。实验基于数独问题转化为SAT问题的思路，设计一个高效的SAT求解器，用于实现数独游戏生成与求解时间的优化，系统通过随机生成的棋盘进行求解，得到一个完整的数独解，据此通过挖洞法得到一个具有唯一解的数独棋盘，设计最终实现数独游戏可玩并且具有简单的交互性。

## 2.2 系统总体设计

实验设计的系统具分为两个板块，分别是基于DPLL算法的SAT求解器设计板块和数独游戏生成与交互板块。对于这两大板块，各自具有一些小模块如下所示。

1.基于DPLL算法的SAT求解器板块：CNF文件的读取、CNF文件输出校验、DPLL算法辅助函数设计、DPLL算法和SAT算例结果输出。

2.数独游戏生成与交互板块：初始化数独游戏格局、数独规约为SAT化简和预处理、数独游戏求解结果处理、数独游戏操作和输出数独对应的CNF文件。

基于设计问题和实验要求设计得到上述的两大板块以及各自具有的小模块，整个系统程序流程图见下图1-1。

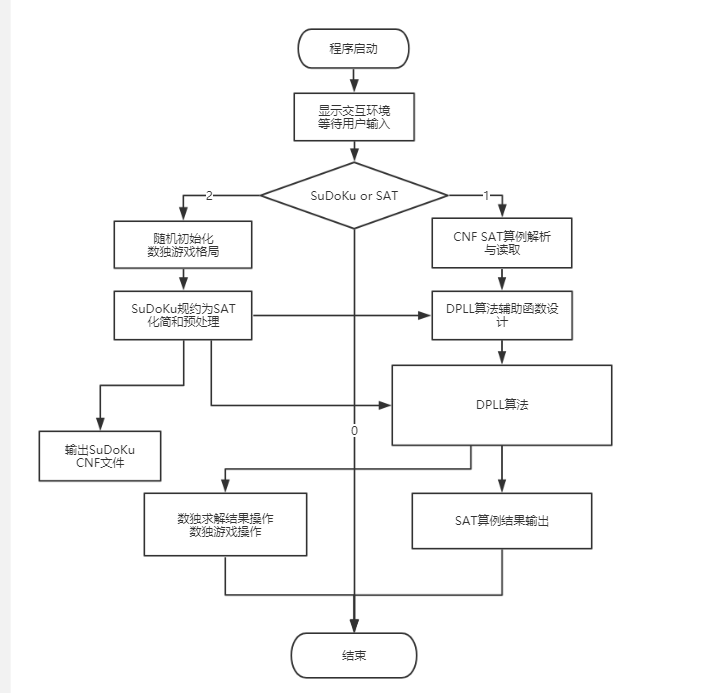


图2-1 实验系统程序流程图

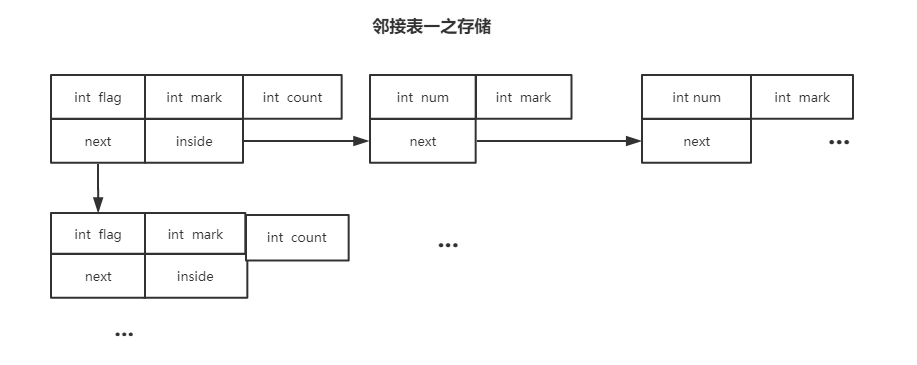
# 3 系统详细设计

## 3.1 数据结构定义

实验中需要读取CNF文件，按照文件中行列顺序完整的读入程序中，所以使用的物理结构为邻接表，分别为存储邻接表和索引邻接表两种不同的邻接表。

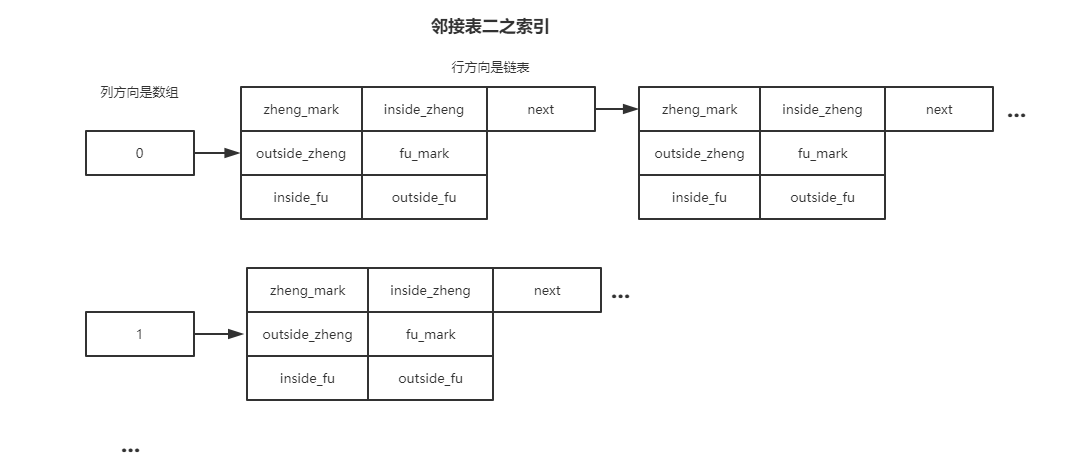
存储邻接表定义：

存储邻接表分为横竖两个方向的单链表，其中横轴方向的单链表用于存储从CNF文件中读入的公式即每行出现的变元。竖轴方向的单链表存入CNF文件中每行的公式。其中行结点的int类型的num用于记录存入的数，int类型的mark用于标记该结点的值是否被删除。竖轴方向上结点的int类型count记录当前横轴方向链表存入的数的数量，int类型的mark记录当前横轴方向的子句是否被删除，int类型的flag记录是因为哪个值被删除。存储邻接表的物理结构如下图3-1所示。

 **图3-1 存储邻接表物理结构示意图**

索引邻接表定义：

索引邻接表用于指向例如变元出现的所有的结点以及该结点对应的头结点。其中int类型的zheng\_mark标记该结点指向的数为正，int类型的fu\_mark标记该结点指向的数为负。其中结构成员inside\_zheng指向当前正变元下标出现的结点，成员outside\_zheng指向当前正变元出现的结点的头结点。结构成员inside\_fu指向当前负变元出现的结点,成员outside\_fu指向当前负变元出现结点的头结点。索引邻接表的物理结构如下图3-2所示。



**图3-2 索引邻接表物理结构示意图**

## 3.2 主要算法设计

### 3.2.1 CNF文件处理

1.读CNF文件到邻接表

建立一个读取CNF文件函数status ReadFile(headnode \*Head, bignode G,char filename[60])，通过动态分配空间和调用fread()函数从文件中读取数据，最后将CNF文件中的子句存入邻接表中。

2.保存为CNF文件

建立一个生成CNF文件函数status Save\_File(headnode Head)，通过使用while()循环遍历邻接表，使用fprintf()函数将邻接表中的数据存入一个新建立的CNF文件中。

3.输出读取的CNF文件

建立一个输出邻接表读入数据的函数void Traverse(headnode Head)，使用while()遍历整个邻接表，根据将读入的数据按照CNF文件中保存形式输出。

### 3.2.2 DPLL算法

1. 删除单子句

建立一个删除单子句函数status DeleteSingle(bignode G, int flag)。首先，在索引表中找到要删除数字flag的绝对值所对应的结构数组下标。其次，使用while循环遍历对结构数组位置为flag绝对值所指向的单链表。最后，通过已有的删除规则，选择将单一节点的mark设置为0或者是将该结点在存储邻接表的头结点的mark设置为0，达到删除效果。

2.恢复单子句

建立一个恢复单子句函数status Recovery(bignode G, int flag)。首先，在索引表中找到要恢复数字flag绝对值所对应的结构数组下标。其次，使用while循环遍历对结构数组位置为flag绝对值所指向的单链表。最后，通过已有的恢复规则，选择将单一节点的mark设置修改为1或者是将该结点在存储邻接表的头结点的mark设置为1，达到恢复效果。

3.查找下一个假设变元

建立一个查找下一个可以假设为真的变元函数status FindCount(HeadNode \*Head)。首先，通过使用while()循环遍历存储邻接表，判断是否能够找到单子句，找到单子句则返回对应的值。如果没有找到单子句，选择出现次数最多的变元或者是按顺序选取可以假设变元作为下一个假设为真的变元。

4.判断是否为空集

建立一个判断空集的函数status JudgeGroup(HeadNode \*Head)，通过使用while()循环遍历存储邻接表判断是否头结点的mark是否都被赋值为0，即是否都表示为空。如果头结点的mark都被赋值为0，则return OK,否则返回FALSE。

5.判断是否出现空子句

建立一个判断是否出现单子句函数status FindEmptyClause(HeadNode \*Head)。通过使用while()循环遍历存储邻接表和通过判断头结点q->count为1和q->mark为1等两个条件是来判断是否存在空子句，出现了return OK,否则return FALSE。

6.变元出现次数排序

建立一个统计并排序出现次数最多的变元函数void Strategy(headnode Head, int mark\_1[N], int mark\_2[N])。首先，使用while()循环遍历存储CNF文件的索引邻接表，使用mark\_1数组记录对应数组位序数字正负值出现的次数。其次，通过选择排序算法，将每一次遍历mark\_1数组得出的出现次数最多的位序存入数组mark\_2，完成统计任务。

7. DPLL算法

建立一个DPLL算法主函数status Dpll\_youhua(HeadNode \*Head, int now\_count, bignode G)。首先，判断当前传入的求解集合是否为空，为空则return OK。若不为空，再判断是否存在空子句，若为空子句就恢复前一步删除的变元，return FALSE。若不为空，则继续调用DPLL函数，出现回溯情况，就将当前变元的相反数带入。若最后都不成立，则return FALSE,完成回溯。

### 3.2.3 SuDoKu问题转化SAT问题

1.匹配棋盘

建立一个匹配棋盘函数void Match(int arr[][9], int a[][10][10])。首先，将建立的三维数组初始化，其中，三维数组的三个位序上的数值分别表示行、列和可能的赋值。其次，遍历二维数独数组，判断当前可能填入的数字，如可能填入，对应三维数组为序按照从小到大的次序赋值。最后，将最大的次序赋值给variable，return OK。

2.读取数据到邻接表

建立一个读取函数status Read\_SuDoKu(headnode \*Head, bignode G, int mid)，将SuDoKu转换成SAT命题公式后的值存入新的存储和索引邻接表。

3.匹配SAT结果

建立一个匹配SAT结果的函数void ShuChu\_match(int shuzu[][9])。结合DPLL算法给出的解以及三维标记数组来确定对应数组位置上可能填入的值，然后输出对应填写完毕的数独棋盘进行校验。

4.判断数独填入是否正确

建立一个判断数独填入是否正确的函数status SuDoKu\_Judge(int arr[][9], int row, int col, int n)。通过传入的数组以及对应位置信息和值，结合数独行、列和小方块内不重复规则来判断填入的数是否符合游戏规则。符合规则return OK,否则return FALSE。

5.生成数独函数

建立一个生成数独函数status Create\_sudoku(headnode \*H, bignode F,int arr[9][9])。首先，掉用随机生成函数ranf()来生成对应的行、列和对应位置的值。其次，结合判断函数判断当前生成数据是否有效，有效填入，继续填写。最后控制生成有效数据的个数。

6.将数独问题转化为SAT问题

建立将数独问题转化成SAT问题的主函数status Create\_sudoku(headnode \*H, bignode F,int arr[9][9])。其次建立八个将数独问题转化为SAT问题的辅助函数，这些函数分别依据每空取值组合、每空取值保证、每行取值组合、每行取值保证、每列取值组合、每列取值保证、每小方块取值组合和每小方块取值保证共八个部分，将数独问题转换成对应的SAT问题的CNF文件和存入当前的空邻接表中和索引表中。

### 3.2.4挖洞法生成唯一解

1.判断是否有唯一解

建立一个调用DPLL算法判断挖洞法生成的数独棋盘是否有解的函数status Juge\_Hole(int arr[][9], int n, int m, int k, headnode H, bignode F)。首先，判断从数字1到9且与传入的位置不同的数同时该数满足数独填入规则。其次，将该值带入要判断是否可以挖洞的位置，然后调用DPLL算法来检验填入其他不同的数后数独棋盘是否有解。若有解则恢复当前位置，return FALSE表示当前挖出的位置不存在唯一解。否则 return OK表示该位置可以挖洞。

2.挖洞法基于递归思想

建立一个基于回溯法的挖洞函数status Dig\_Hole(int arr[][9],int n, int m, int k, headnode H, bignode F)。首先，判断挖洞的个数是否达标，挖洞个数达标则return OK。其此，若没有挖洞数量达标，继续判断当前所挖的洞是否会满足存在唯一解这一个条件，不满足存在唯一解的要求则复原并且return FALSE。接着，如果当前挖的洞满足唯一解情况，再次调用当前的挖洞函数，如果挖洞失败出现回溯情况，那就根据所设置的while()循环的次数在该层上继续调用挖洞函数，争取最大次数的挖洞。

3.找到下一个要挖的洞

建立一个找到下一个要挖的洞的位置函数int Delete\_Num(int arr[][9])。首先，使用两个for()循环，判断奇数行和偶数行来决定第二个for循环，总体思想是走一个S形的寻找路线。通过返回一个三位数来表示删除的位置信息。

# 4 系统实现与测试

## 4.1 系统实现

系统实现的开发工具是Visual Studio 2017，数据结构算法的实现以及程序测试的结果具体如下。

### 4.1.1系统功能

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。达成如下6个功能：

1.输入输出功能：实现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：实现读取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；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。

3.基于DPLL的求解器：基于DPLL算法框架，实现SAT算例的求解。

4.时间性能的测量：实现基于相应的时间处理函数，记录DPLL过程执行时间，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。

6.SAT应用：将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，具有简单的交互性。

### 4.1.2数据结构

系统的实现所用的数据元素类型定义、宏定义和结构定义分别如下所示。

1.数据元素类型的定义

数据结构的表示（存储结构）用类型定义（typedef）描述。

typedef int status；

2.宏定义

预定义常量和类型，用于判断函数结果的状态。

#define FALSE -1

#define OK 1

#define OVERFLOW -2

#define N 600

3.存储邻接表行定义

创建一个status类型的结构，结构名为EdgeNode，结构体指针为\*edgenode，其成员分别为 int num、int mark和struct EdgeNode \*next。

typedef struct EdgeNode {

int num;

int mark;

struct EdgeNode \*next;

}EdgeNode, \*edgenode;

4.存储邻接表列定义

创建一个status类型的结构，结构名为HeadNode，结构体指针为\* headnode，其成员分别为 int count、int mark、int flag、struct EdgeNode \*inside和struct HeadNode \*next。

typedef struct HeadNode {

int count;

int mark;

int flag;

struct HeadNode \*next;

struct EdgeNode \*inside;

}HeadNode, \*headnode;

5.索引邻接表行定义

创建一个status类型的结构，结构名为SerchNode，结构体数组为 SerchList[N+1]，其成员分别为 int zheng\_mark、int fu\_mark、struct EdgeNode \*inside\_zheng、struct HeadNode \*outside\_zheng、struct EdgeNode \*inside\_fu、struct HeadNode \*outside\_fu和struct SerchNode\*next。

typedef struct SerchNode {

int zheng\_mark;

int fu\_mark;

struct EdgeNode \*inside\_zheng;

struct HeadNode \*outside\_zheng;

struct EdgeNode \*inside\_fu;

struct HeadNode \*outside\_fu;

struct SerchNode\*next;

}SerchNode, SerchList[N+1];

6.索引邻接表列定义

创建一个status类型的结构，结构名为BigNode，结构体指针为\*bignode，其成员分别为 int mark和SerchList serchlist。

typedef struct {

SerchList serchlist;

int mark;

} BigNode, \*bignode;

### 4.1.3函数声明

整个系统分为四个大板块，分别是CNF文件存储板块、DPLL算法板块、数独问题转换SAT问题板块和挖洞法生成唯一解数独板块。

1.CNF文件存储板块

通过设计对应文件存储函数来完成CNF文件的存取存取任务：

（1）读取CNF文件函数status ReadFile(headnode \*Head, bignode G,char filename[60])。

（2）保存CNF文件声明函数：status Save\_File(headnode Head)。

（3）存CNF文件小前提函数：void check(headnode Head)。

（4）输出CNF文件函数：void Traverse(headnode Head)。

（5）清除邻接表函数：void Realse\_Link(headnode \* Head)。

2.DPLL算法板块

通过SAT问题求解规则设计了DPLL算法主函数以及对应的辅助函

（1）标记函数，确定接下来可选取的假设真值情况：void Strategy(headnode Head, int mark\_1[N], int mark\_2[N])。

（2）删除单子句函数：status DeleteSingle(bignode G, int flag)。

（3）恢复单子句函数：status Recovery(bignode G, int flag)。

（4）寻找假设的真值函数：status FindCount(HeadNode \*Head)。

（5）判断是否为空集函数：status JudgeGroup(HeadNode \*Head)。

（6）寻找空子句函数：status FindEmptyClause(HeadNode \*Head)。

（7）DPLL算法的主函数，基于递归思想status Dpll\_youhua(HeadNode \*Head, int now\_count, bignode G)。

DPLL算法板块的流程图如下图4-1所示。

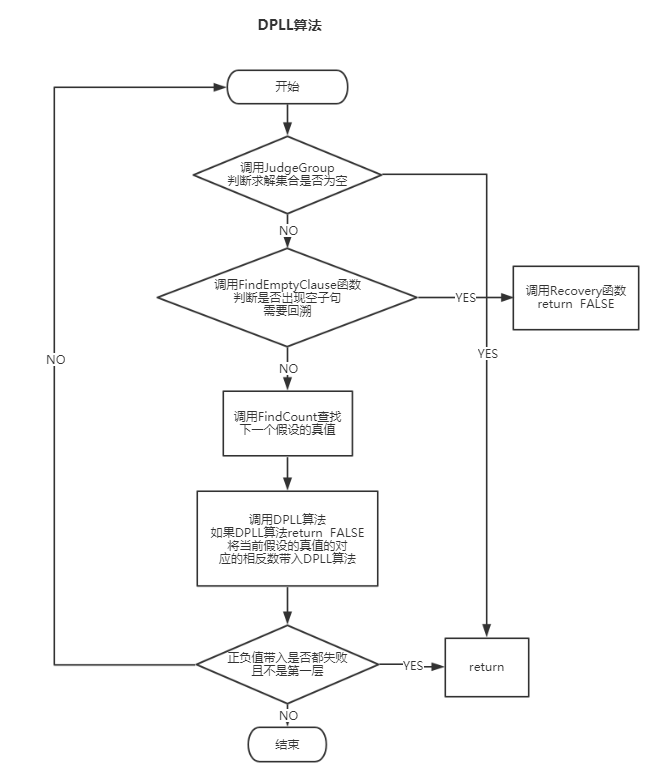


图4-1 DPLL算法求解板块函数调用情况

3.数独问题转换成SAT问题板块

依据数独棋盘情况，结合数独问题转换成SAT问题的规则，设计相应的辅助函数为：

（1）输出数独棋盘：void Print\_SuDoKu(int arr[][9])。

（2）匹配棋盘：void ShuChu\_match(int shuzu[][9])。

（3）调用读取函数：status Read\_SuDoKu(headnode \*Head, bignode G, int mid)。

（4）读入邻接表之初始化邻接表操作：status Isinit\_SuDoKu(headnode \*Head, bignode G)。

（5）数独之判断填入的数字是否正确：status SuDoKu\_Judge(int arr[][9], int row, int col, int n)。

（6）匹配函数：void Match(int arr[][9], int a[][10][10])。

（7）数独转换成函数：void Tranlate(int arr[][9], headnode \*H, bignode F)。

（8）调用数独的函数：status Create\_sudoku(headnode \*H, bignode F, int arr[9][9])。

其中依据数独转换成SAT问题的理论设计了八个辅助转换函数，依次为：

（1）每空取值组合 ：void Create\_Cnf\_Position(int arr[][9], int a[][10][10], headnode \*H, bignode G)。

（2）每行取值组合：void Create\_Cnf\_Row(int arr[][9], int a[][10][10], headnode \*H, bignode G)。

（3）每列取值组合：void Create\_Cnf\_Col(int arr[][9], int a[][10][10], headnode \*H, bignode G)。

（4）每小方块取值组合：void Create\_Cnf\_Block(int arr[][9], int a[][10][10], headnode \*H, bignode G)。

（5）每空取值保证：void Insure\_Cnf\_Position(int arr[][9], int a[][10][10], headnode \*H, bignode G)。

（6）每行取值保证：void Insure\_Cnf\_Row(int arr[][9], int a[][10][10], headnode \*H, bignode G)。

（7）每列取值保证：void Insure\_Cnf\_Col(int arr[][9], int a[][10][10], headnode \*H, bignode G)。

（8）每小方块取值保证：void Insure\_Cnf\_Block(int arr[][9], int a[][10][10], headnode \*H, bignode G)。

4.挖洞法生成唯一解板块

依据挖洞法原理设计了基于递归思想的挖洞法，其中所使用到的函数如下：

（1）挖洞法主函数：status Dig\_Hole(int arr[][9], int n, int m, int k, headnode H, bignode F)。

（2）判断是否为唯一解：status Juge\_Hole(int arr[][9], int n, int m, int k, headnode H, bignode F)。

（3）找到下一个符合要求的洞：int Delete\_Num(int arr[][9])。

挖洞算法板块的流程图如下图4-2所示。

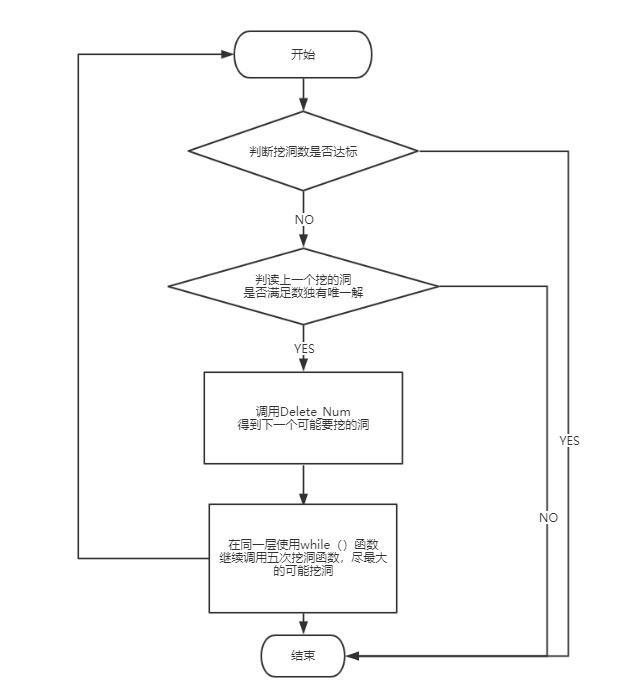


图4-2 挖洞法算法函数调用情况

## 4.2 系统测试

1.交互主菜单测试

测试交互主菜单的功能。测试内容分别为：①根据菜单编号正确输入。②未根据菜单编号的错误输入即输入单个字符。③没有根据菜单编号的输入错误即输入越界。

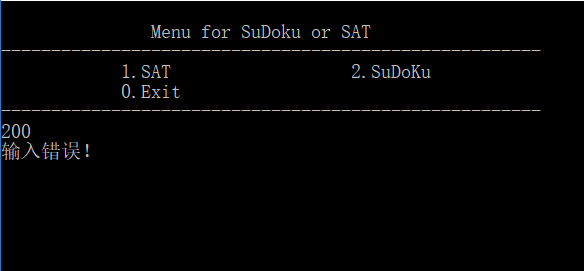
分析：通过swtich函数来选择不同的操作部分的展示，层层嵌套，根据不同的输入情况进行报错，从而达到更加人性化的人机交互。

表4-1测试数据样例表

|  |  |  |
| --- | --- | --- |
| 测试内容 | 测试用例 | 理论结果 |
| ① | 1 | 进入SAT板块 |
| ② | A | 输入错误 |
| ③ | 200 | 输入错误 |

测试结果见图4-3。其中，测试内容①、②和③的测试结果分别见图2-4（a）、图4-3（b）和图4-3（c）。交互菜单操单流程图见图4-4。

|  |  |
| --- | --- |
| （a） | （b） |



（c）

图4-3交互主菜单测试结果

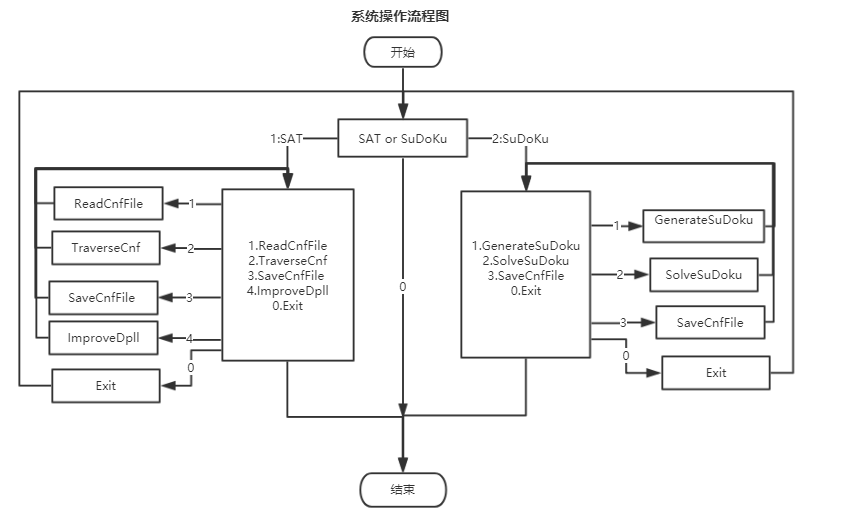


图4-4交互主菜单操作流程图

2.读取CNF算例文件测试

测试读取CNF算例文件，并输出验证是否读取正确。测试内容分别为：①读取CNF算例文件，并输出验证是否读取正确。

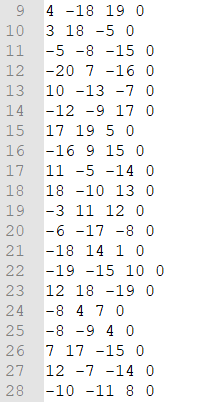
分析：使用fscanf()函数读取CNF文件中的数据，按照其存储形式存入邻接表中。对于输出CNF文件的函数部分，使用两个while循环遍历存储邻接表，输出对应的值即可。

表4-2测试数据样例表

|  |  |  |
| --- | --- | --- |
| 测试内容 | 测试用例 | 理论结果 |
| ① | sat-20.cnf | 读取成功 |

测试结果见图4-5。

|  |  |
| --- | --- |
| （a） | （b） |



（c）

图4-5读取CNF算例文件测试结果

3.基于DPLL的SAT求解器测试

测试基于DPLL的SAT求解器的功能以及优化过的SAT求解器的优化率。测试内容分别为：①CNF文件有解。②CNF文件无解。③测试三十组SAT算例，给出优化结构和算法前、优化结构和决策算法后的运行时间和优化率。

分析：未优化前使用一张邻接表来进行操作，同时没有优化选取策略，结果随着变元数量的增加，进而算法回溯次数的增多，时间的花销非常大。通过优化数据结构和变元选取策略即新建一张索引表和根据变元的绝对值出现的次数的多少作为在没有单子句的情况下，选择新的假设值的依据。经过优化后，求解时间大幅度减少。在实验过程中发现，对于数独问题的求解，结合单子句规则和按顺序选取假设变元的方法求解对应的SAT问题的效率更为明显。

表4-3测试数据样例表

|  |  |  |
| --- | --- | --- |
| 测试内容 | 测试用例 | 理论结果 |
| ① | sat-20.cnf | 有解 |
| ② | u-problem7-50.cnf | 无解 |

测试结果见图4-6。其中，测试内容①和②的测试结果分别见图4-6（a）和图4-6（b）。测试内容③的测试结果见表格4-4。

表4-4 30个测试样例结果表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 编号 | 测试用例 | 变元数 | 公式数 | 优化前需要时间（ms） | 优化后需要时间（ms） | 优化率（%） |
| 1 | sat-20.cnf | 20 | 91 | 1 | 0 | 100 |
| 2 | unsat-5cnf-30.cnf | 30 | 420 | 584 | 122 | 79.11 |
| 3 | ais10.cnf | 181 | 3151 | 35562 | 4462 | 87.73 |
| 4 | sud00009.cnf | 303 | 2851 | 30 | 8 | 73.33 |
| 5 | tst\_v10\_c100.cnf | 10 | 100 | 2 | 0 | 100 |
| 6 | tst\_v25\_c100.cnf | 25 | 100 | 1 | 0 | 100 |
| 7 | tst\_v50\_c500.cnf | 50 | 500 | 21 | 0 | 100 |
| 8 | tst\_v100\_c160.cnf | 100 | 160 | 5 | 1 | 80 |
| 9 | tst\_v100\_c400.cnf | 100 | 400 | 6 | 1 | 83.33 |
| 10 | tst\_v200\_c210.cnf | 200 | 210 | 4 | 1 | 75 |
| 11 | tst\_v100\_c425.cnf | 100 | 425 | 797 | 1 | 99.99 |
| 12 | tst\_v200\_c220.cnf | 200 | 220 | 5 | 1 | 80 |
| 13 | ais6.cnf | 61 | 581 | 22 | 12 | 45.45 |
| 14 | ais8.cnf | 113 | 1520 | 870 | 20 | 97.7 |
| 15 | problem1-20.cnf | 20 | 91 | 2 | 0 | 100 |
| 16 | problem2-50.cnf | 50 | 80 | 1071 | 28 | 97.39 |
| 17 | problem3-100.cnf | 100 | 340 | 17755 | 364 | 97.95 |
| 18 | problem6-50.cnf | 50 | 100 | 625 | 111 | 82.24 |
| 19 | problem8-50.cnf | 50 | 300 | 20 | 2 | 90 |
| 20 | problem9-100.cnf | 100 | 200 | 123507 | 48041 | 61.11 |
| 21 | problem11-100.cnf | 100 | 600 | 521 | 54 | 89.64 |
| 22 | sud00001.cnf | 301 | 2780 | 153 | 30 | 80.39 |
| 23 | sud00012.cnf | 232 | 1901 | 881 | 112 | 98.75 |
| 24 | sud00021.cnf | 308 | 2911 | 1264 | 156 | 98.73 |
| 25 | sud00079.cnf | 301 | 2810 | 126 | 24 | 80.95 |
| 26 | sud00082.cnf | 224 | 1762 | 735 | 105 | 85.71 |
| 27 | sud00861.cnf | 297 | 2921 | 417 | 71 | 82.97 |
| 28 | u-problem7-50.cnf | 50 | 100 | 1748 | 319 | 81.75 |
| 29 | tst\_v10\_c100.cnf | 10 | 100 | 6 | 1 | 83.33 |
| 30 | u-problem10-100.cnf | 100 | 200 | 468940 | 60246 | 87.15 |

|  |  |
| --- | --- |
| （a） | （b） |

图4-6基于DPLL的SAT求解器测试

4.数独游戏转换成SAT问题测试

测试数独游戏转换成SAT问题测试。测试内容为：通过使用随机函数生成数独棋盘，将数独问题转换成SAT问题带入SAT问题求解器，最后得到其中一组数独解。

分析：简而言之，依据每空取值组合、每空取值保证、每行取值组合、每行取值保证、每列取值组合、每列取值保证、每小方块取值组合和每小方块取值保证等八个转换原则，将数独问题转换成对应的SAT问题的CNF文件和存入当前的空邻接表中和索引表中。

测试结果见图4-7。

|  |  |
| --- | --- |
| （a） | （b） |

图4-7数独游戏转换成SAT问题的测试结果

5.挖洞法生成唯一解数独棋盘

测试挖洞法生成唯一解数独棋盘。测试内容为：通过一个已知的数独的解，通过挖洞法挖出指定洞数的数独棋盘。

分析：挖洞法生成唯一解的数独棋盘的难点在于：如何判断当前数独棋盘是具有唯一解。解决的方法是：挖掉当前位置的数时，先判断：当其他1-9不同于原来位置上的数填入时，是否满足数独游戏规则。若一步满足，接着将满足该步的数填入数独棋盘，将数独问题转换为SAT问题，调用SAT问题求解器求解，如有解则表明当前挖掉此洞后不具有唯一解，接着继续挖下一个洞。

测试结果见图4-8。

|  |  |
| --- | --- |
| （a） | （b） |

图4-8挖洞法生成唯一解数独棋盘

# 5 总结与展望

## 5.1 全文总结

本次实验圆满完成了基于SAT的数独游戏求解系统的设计与实现，主要完成的工作如下：

1.查阅相关资料，了解和学习当前SAT问题的知识体系以及应用范围，掌握SAT求解器的算法内核与具体实现策略。同时，学习数独问题转换为SAT问题的一些知识体系和架构，为接下来的课程设计实验部分的开展奠定了坚实的理论基础。

2.设计并实现系统的输入输出板块的功能，使系统实现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。

3.设计并实现对读取的CNF文件进行遍历输出的功能，即遍历内部结构逐行输出与显示每个子句达到检验的目的。

4.基于DPLL算法框架，设计SAT求解器并实现SAT求解器正确求解CNF文件中提供的SAT算例。

5.对基本DPLL算法部分进行存储结构优化即建立一张用于存储的邻接表和一张用于索引的邻接表。对于变元选取策进行优化即按照变元出现次数进行选取结合单子句规则。

6.实现利用随机生成函数，创建一个初始的数独棋盘，将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，得到数独解。利用已知的数独解，通过挖洞法游戏挖出一个具有唯一解的数独棋盘，提供给用户，同时数独游戏具有简单的交互性。

## 5.1 工作展望

基于 DPLL 的SAT 求解器在求解性能上有了一定的改善，但是还有许多地方有待改进。现今的许多求解器在求解许多大规模的 SAT 实例时往往需要先对 CNF 公式进行预处理，经过预处理化简后导出一个简化的 CNF 公式，然后再用算法对简化后的公式进行求解，这样效率会更高一些，下面列出了几个将来的工作重点和研究方向：

1.为算法加入合适的预处理器，如引入高级前向推理机制，对 CNF 公式进行预处理或化简，以便能更高效地进行求解。

2.可以尝试将本文中的算法应用到更大规模的实际问题中，比如硬件验证、自动化推理等领域。

3.算法采用的数据结构和具体实现有待改进，可以引入更巧妙的数据结构来表示文字和子句信息，也可以优化代码实现以增强程序的内聚性。

4.可以在 DPLL 的求解过程中寻求更高效的启发式变量决策策略，以减少算 法的回溯次数和搜索空间。

5.可以通过对命题逻辑的可满足性问题的逻辑结构进行分析，采用一种基于 解方程组的 SAT 问题求解思路。即先将合取范式转换为等价的方程组形式，然后 再对方程组进行求解，进而得到对应 SAT 问题的解

# 6 实验体会

本次课程设计实验圆满结束，在整个实验过程中，学习到了很多的东西，也有很多的收获，但也看出来自身存在的一些不足之处。这次的课程设计实验，对于我来说，有很大的难度，对实验内容知识体系的不了解，编码能力不强，所以刚开始看到课设设计书时，有些无从下手。之后通过查阅相关文献，认真去学习自己的知识盲点，在程序设计中，能够举一反三，专心致志，一点点去解决难题，完成课程设计任务。

在学习的过程中，认真了解SAT问题，DPLL算法，以及DPLL算法的优化策略。到之后的数独问题转化为SAT问题的研究，自己的阅读和学习能力得到了很大的提高。

接触了这个课设计题目：基于SAT的数独游戏求解。知道了SAT 问题在计算复杂性理论中具有非常重要的地位，设计并实现解决该类问题的高效算法的意义重大。其中，SAT问题的求解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。

如下是对自己实验过程的出现的一些对应问题个人心得以及总结。

1. 在DPLL算法部分，刚开始没有理解好相关的知识点，所以在刚开始使用迭代法实现时，出现了很多的BUG。最后重新阅读相关的文献，理解好整个算法流程，在使用非递归实现后，改成了递归方式实现SAT求解器，圆满的完成了DPLL求解器的设计与实现。

2. 将数独游戏问题转换成CNF文件格式的时候，遇到了很大的难题，整整一天没有任何进展，心态有点受影响。最后通过结合资料以及将数独棋盘位置上的信息转化为一个三位数去替代进CNF文件，从而发现了其中的转换规律。再通过三维数组的辅助作用，成功的实现这部分的功能，感觉受到了极大的鼓舞。

3. 在通过挖洞法生成唯一解数独棋盘时，由于没有考虑清楚如何判定是否有唯一解，所以导致了思路想法全部错了，在之后确定好判断唯一解的方法后，才得以正常进行。在整个部分，自己最初判断唯一解的方法过于片面，没有考虑其他的情况，所以在之后返工改程序，浪费了很多的时间。

# 参考文献

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

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

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

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

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

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

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

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

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

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

[10] Robert Ganian and Stefan Szeider. Community Structure Inspired Algorithms for SAT and #SAT. International Conference on Theory and Applications of Satisfiability Testing(SAT 2015),223-237360

# 附录 课程设计源程序

课程设计源程序代码分为两个部分，分别是优化前和优化后的源程序。

### 优化前源程序代码

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include <time.h>

#pragma warning(disable:4996)

#define FALSE -1

#define OK 1

#define OVERFLOW -2

#define N 350

typedef int datatype;

typedef int status;

int variable, sentences;

int mark[350];

/\*小边集合\*/

typedef struct Edgenode {

int num;

int mark;

struct Edgenode \*next;

}Edgenode, \*edgenode;

/\*大边集合\*/

typedef struct Headnode {

int count;

int mark;

int flag;

struct Headnode \*next;

struct Edgenode \*inside;

}Headnode, \*headnode;

/\*cnf文件处理板块\*/

/\*读取cnf文件\*/

status ReadFile(headnode \*Head)

{

int mid,counnt=0;

char ch;

FILE \*fp;//文件指针

char filename[70];

printf("input file name: ");

scanf("%s", filename);

if ((fp = fopen(filename, "r")) == NULL)

{

printf("file open error！\n ");

return FALSE;

}

else

{

printf("\n所读取的cnf文件中的注释内容：\n");

while (fread(&ch, sizeof(char), 1, fp))

{

if (ch != 'p') printf("%c", ch);

else break;

}

for (int i = 0; i<5; i++)

fread(&ch, sizeof(char), 1, fp);

fscanf(fp, "%d", &variable);

fscanf(fp, "%d", &sentences);

counnt = sentences;

printf("cnf文件中的文字数：%d 公式数： %d\n", variable, sentences);

//读取公式部分

Edgenode \*p = NULL, \*t = NULL;

Headnode \*r = NULL, \*q = NULL;

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

q = r;

(\*Head) = r;

r->next = NULL;

r->inside = NULL;

r->count = 0;

r->mark = 0;

r->flag = 0;

while (fscanf(fp, "%d", &mid) != EOF && counnt)

{

if (mid != 0)

{

//读入变元

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

p->next = NULL;

p->num = mid;

p->mark = 1;

if (q->inside == NULL)

{

q->inside = p;

}

else

{

t = q->inside;

while (t->next)

t = t->next;

t->next = p;

}

q->count++;

}

else

{

//读入下一个公式

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

r->next = NULL;

r->inside = NULL;

r->count = 0;

r->mark = 0;

r->flag = 0;

if (q->inside != NULL) q->mark = 1;

q->next = r;

q = r;

counnt--;

}

}

}

return OK;

}

/\*输出cnf文件\*/

status Traverse(headnode Head)

{

int mark = 0;

Edgenode \*p;

Headnode \*q;

q = Head;

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

while (q)

{

if (q->mark == 1)

{

printf("%2d :", q->count);

p = q->inside;

while (p)

{

if (p->mark == 1)

printf("%4d", p->num);

p = p->next;

}

}

if (q->mark == 1) printf("\n");

q = q->next;

}

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

return OK;

}

/\*标记函数，确定假设的真值情况\*/

void Judgement(headnode Head, int mark[N])

{

int temp;

Edgenode \*p;

Headnode \*q;

q = Head;

while (q)

{

if (q->mark == 1)

{

p = q->inside;

while (p)

{

//设置标记，为零表示cnf文件中出现过

if (p->num > 0) mark[p->num] = 0;

else

{

temp = p->num;

mark[-temp] = 0;

};

p = p->next;

}

}

q = q->next;

}

}

/\*发现单子句\*/

status FindSingle(Headnode \*Head, int \* flag)

{

Edgenode \*p = NULL;

Headnode \*q = NULL;

q = Head;

while (q)

{

p = q->inside;

if (q->count == 1 && q->mark == 1)//记录变元数量

{

while (p)

{

if (p->mark == 1)

{

\*flag = p->num;

return OK;

}

p = p->next;

}

}

q = q->next;

}

return FALSE;

}

/\*删除单子句\*/

status DeleteSingle(Headnode \*Head, int flag)

{

Edgenode \*p = NULL;

Headnode \*q = NULL;

q = Head;

while (q)

{

p = q->inside;

while (p)

{

//删除单子句

if (q->mark == 1 && p->num == flag)

{

q->mark = 0;

q->flag = flag;

break;

}

//删除变元

if (q->mark == 1 && p->num == -flag&&p->mark != 0 && q->count != 0)

{

p->mark = 0;

q->count--;

}

p = p->next;

}

q = q->next;

}

return OK;

}

/\*恢复单子句\*/

status Recovery(Headnode \*Head, int flag)

{

Edgenode \*p = NULL;

Headnode \*q = NULL;

q = Head;

while (q)

{

p = q->inside;

while (p)

{

//恢复单子句

if (p->num == flag && q->mark == 0 && q->flag == flag)

{

q->mark = 1;

q->flag = 0;

break;

}

//恢复变元

else if (p->num == -flag && p->mark == 0 && q->mark == 1)

{

p->mark = 1;

q->count++;

}

p = p->next;

}

q = q->next;

}

return OK;

}

/\*寻找空子句\*/

status FindemptyClause(Headnode \*Head)

{

Headnode \*q = NULL;

q = Head;

while (q)

{

//p = q->inside;

if (q->count == 0 && q->mark == 1)

return OK;

q = q->next;

}

return FALSE;

}

/\*判断s是否为空集\*/

status JudgeGroup(Headnode \*Head)

{

Headnode \*q = NULL;

q = Head;

while (q)

{

if (q->mark != 0) return FALSE;

q = q->next;

}

return OK;

}

/\*寻找假设的真值\*/

int FindCount(Headnode \*Head)

{

int flag;

//先找单子句

if (FindSingle(Head, &flag) == OK)

{

if (flag > 0 && mark[flag] == 0)

{

//printf("11111当前假设的真值为:%d\n", flag);

mark[flag] = 1;

return flag;

}

if (flag < 0 && mark[-flag] == 0)

{

//printf("2222当前假设的真值为:%d\n", flag);

mark[-flag] = -1;

return flag;

}

}

else

{

for (int i = 1; i <= variable; i++)

{

if (mark[i] == 0)

{

//printf("333当前假设的真值为:%d\n", i);

mark[i] = 1;

return i;

}

}

}

return 0;

}

/\*dpll函数版块\*/

status Dpll(Headnode \*Head, int now\_count)

{

int next\_count;

//判断是否为空

if (JudgeGroup(Head) == OK) return OK;

else

{

//检查是否出现空子句，出现就修复然后return false

if (FindemptyClause(Head) == OK)

{

//printf("需要回溯的数值：%2d\n", now\_count);

Recovery(Head, now\_count);

//traverse(head);

if (now\_count>0)

mark[now\_count] = 0;

else mark[-now\_count] = 0;

return FALSE;

}

else

{

//不出现即找一个合适的真值

//测试当前假设真值是否成立

next\_count = FindCount(Head);

if (next\_count == 0)return FALSE;

DeleteSingle(Head, next\_count);

if (Dpll(Head, next\_count) == OK) return OK;

else

{

Recovery(Head, next\_count);

//printf("当前检验对应负值：%2d\n", -next\_count);

DeleteSingle(Head, -next\_count);

//Traverse(Head);

if (Dpll(Head, -next\_count) == OK)

{

//printf("当前检验对应负值：%2d\n", -next\_count);

//Traverse(Head);

if (next\_count>0)

mark[next\_count] = -1;

else mark[-next\_count] = 1;

return OK;

}

else

{//负数也测试失败，则标记该假设，返回上一个假设

Recovery(Head, -next\_count);

//printf("正负检验失败，退出该值并消去该值\n");

//traverse(head);

if (next\_count>0)

mark[next\_count] = 0;

else mark[-next\_count] = 0;

return FALSE;

}

//还不成立这个数的标记清空，退回到上一个数；

}

}

}

}

/\*主函数\*/

int main(void)

{

headnode T;

Headnode Arr;

T = &Arr;

int op = 1;

int t1, t2, flag = 0;

while (op) {

system("cls");

printf("\n menu for suduku or sat \n");

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

printf(" 1.readfile 3.dpll \n");

printf(" 2.printfile 0.exit\n");

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

if (scanf("%d", &op))

{

switch (op) {

case 1:

if (ReadFile(&T) == OK)

printf("cnf文件成功读取！\n");

else printf("cnf文件读取失败！\n");

getchar(); getchar();

break;

case 2:

if (Traverse(T) == OK)

printf("cnf文件输出成功！\n");

else printf("cnf文件输出失败！\n");

getchar(); getchar();

break;

case 3:

for (int i = 1; i <= variable; i++)

mark[i] = 2;

Judgement(T, mark);

t1 = clock();

if (Dpll(T, 1) == OK)

{

flag = 1;

printf("dpll算法处理成功，文件有解！\n");

}

else printf("dpll算法处理失败，文件无解！\n");

t2 = clock();

printf("\n时间：%dms\n", (t2 - t1) );

if (flag == 1)

{

printf("cnf文件的解为：\n");

for (int i = 1; i <= variable; i++)

{

if (mark[i] == 1)printf("%4d ", i);

else if (mark[i] == -1)printf("%4d ", -i);

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

}

}

getchar(); getchar();

break;

case 0:

break;

default:

printf("输入错误！\n");

getchar(); getchar();

break;

}//end of switch

}

else {

printf("输入错误！\n");

getchar(); getchar(); getchar();

}

}//end of while

printf("欢迎下次再使用本系统！\n");

system("pause");

return 0;

}//end of main()

### 优化后源程序代码

实验源代码分为头文件：SuDoKu.h、子文件：main.c、DPLL.c、File.c、SuDoKu.c和DigHole.c共计五个部分，具体部分代码如下：

**SuDuKo.h（头函数）:**

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

#pragma warning(disable:4996)

#define FALSE -1

#define OK 1

#define OVERFLOW -2

#define N 600

typedef int status;

int variable , sentences ;//记录变元数和公式数

int FLAG;//选择策略

int mark\_ture[N];//保存正确的解

int mark\_1[N];//存储变元出现的次数

int mark\_2[N];//存储变元的下标

int a[10][10][10];//三维数组

int counnt ;//记录挖洞数

int hole[9][9];//记录挖过的洞

int shuzu[9][9];//记录棋盘

//邻接表一之存储

typedef struct EdgeNode {//小边集合

int num;

int mark;

struct EdgeNode \*next;

}EdgeNode, \*edgenode;

typedef struct HeadNode {//大边集合

int count;

int mark;

int flag;

struct HeadNode \*next;

struct EdgeNode \*inside;

}HeadNode, \*headnode;

//邻接表二之索引

typedef struct SerchNode {//用于直接删除，不需要遍历

int zheng\_mark;//标记正变元

int fu\_mark;//标记负变元

struct EdgeNode \*inside\_zheng;//指向正变元

struct HeadNode \*outside\_zheng;

struct EdgeNode \*inside\_fu;//指向负变元

struct HeadNode \*outside\_fu;

struct SerchNode\*next;

}SerchNode, SerchList[N+1];

typedef struct {//用于删除合适的假设真值

SerchList serchlist;

int mark;

} BigNode, \*bignode;

// cnf文件读取板块优化后

//读取cnf文件

status ReadFile(headnode \*Head, bignode G,char filename[60]);

//保存cnf文件

status Save\_File(headnode Head);

//存cnf文件小前提

void check(headnode Head);

//输出cnf文件

void Traverse(headnode Head);

//清空链表

void Realse\_Link(headnode \* Head);

// cnf文件处理板块优化后

//标记函数，确定假设的真值情况

void Strategy(headnode Head, int mark\_1[N], int mark\_2[N]);

//删除单子句

status DeleteSingle(bignode G, int flag);

//恢复单子句

status Recovery(bignode G, int flag);

//判断是否为空集

status JudgeGroup(HeadNode \*Head);

//寻找空子句

status FindEmptyClause(HeadNode \*Head);

//寻找假设的真值

status FindCount(HeadNode \*Head);

//Dpll算法的主函数

status Dpll\_youhua(HeadNode \*Head, int now\_count, bignode G);

// 数独函数部分

//输出数独棋盘

void Print\_SuDoKu(int arr[][9]);

//匹配棋盘

void ShuChu\_match(int shuzu[][9]);

//调用读取函数

status Read\_SuDoKu(headnode \*Head, bignode G, int mid);

//读入邻接表之初始化邻接表操作

status Isinit\_SuDoKu(headnode \*Head, bignode G);

//数独之判断填入的数字是否正确

status SuDoKu\_Judge(int arr[][9], int row, int col, int n);

//匹配函数

void Match(int arr[][9], int a[][10][10]);

//每空取值组合

void Create\_Cnf\_Position(int arr[][9], int a[][10][10], headnode \*H, bignode G);

//每行取值组合

void Create\_Cnf\_Row(int arr[][9], int a[][10][10], headnode \*H, bignode G);

//每列取值组合

void Create\_Cnf\_Col(int arr[][9], int a[][10][10], headnode \*H, bignode G);

//每小方块取值组合

void Create\_Cnf\_Block(int arr[][9], int a[][10][10], headnode \*H, bignode G);

//每空取值保证

void Insure\_Cnf\_Position(int arr[][9], int a[][10][10], headnode \*H, bignode G);

//每行取值保证

void Insure\_Cnf\_Row(int arr[][9], int a[][10][10], headnode \*H, bignode G);

//每列取值保证

void Insure\_Cnf\_Col(int arr[][9], int a[][10][10], headnode \*H, bignode G);

//每小方块取值保证

void Insure\_Cnf\_Block(int arr[][9], int a[][10][10], headnode \*H, bignode G);

//数独转换成函数

void Tranlate(int arr[][9], headnode \*H, bignode F);

//调用数独的函数

status Create\_sudoku(headnode \*H, bignode F, int arr[9][9]);

// 挖洞法

//调用dpll算法

status resolve\_sudoku(int arr[][9], headnode H, bignode F);

//挖洞主函数

status Dig\_Hole(int arr[][9], int n, int m, int k, headnode H, bignode F);

//判断是否为唯一解

status Juge\_Hole(int arr[][9], int n, int m, int k, headnode H, bignode F);

//找到下一个洞

int Delete\_Num(int arr[][9]);

**main.c（主函数部分代码）：**

#include "SuDoKu.h"

//主函数部分

int main(void)

{

int fuben[9][9];

for (int i = 0; i < 9; i++)//初始化数独棋盘以及其副本

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

{

shuzu[i][j] = 0;

fuben[i][j] = 0;

}

headnode T;

HeadNode ARR;

T = &ARR;

BigNode arr;

bignode G;

G = &arr;

FILE \*fp;//文件指针

char filename[70],filename\_fuben[70];

int op = 1,Selet\_Sudoku = 1,Selet\_Sat=1;

int t1, t2,t3, isinit = 0, flag = 0, Flag = 0,sign=0,sigg=1;

while (op) {

system("cls");

printf("\n Menu for SuDoku or SAT \n");

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

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

printf(" 0.Exit \n");

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

if (scanf("%d", &op))

{

switch (op) {

case 1:

while (Selet\_Sat) {

system("cls");

printf("\n Menu for SAT \n");

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

printf(" 1.ReadCnfFile 2.TraverseCnf \n");

printf(" 3.SaveCnfFile 4.ImproveDpll \n");

printf(" 0.Exit \n");

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

if (scanf("%d", &Selet\_Sat))

switch (Selet\_Sat) {

case 1:

printf("input file name: ");

scanf("%s", filename);

strcpy(filename\_fuben, filename);//保存副本

if(isinit == 1)Realse\_Link(&T);

if (ReadFile(&T, G, filename) == OK)

{

isinit = 1;

printf("cnf文件读取成功！\n");

}

else printf("cnf文件读取失败！\n");

getchar(); getchar();

break;

case 2:

if (isinit == 1)

{

Traverse(T);

printf("cnf文件输出成功！\n");

}

else printf("cnf文件输出失败！\n");

getchar(); getchar();

break;

case 3:

if (isinit == 1)

{

if (Save\_File(T) == OK)

printf("保存为cnf文件成功！\n");

else printf("保存为cnf文件失败！\n");

}

else printf("保存为cnf文件失败！\n");

getchar(); getchar();

case 4:

if (isinit == 1)

{

for (int i = 1; i < variable + 1; i++)

mark\_ture[i] = 0;

printf("请选择你需要的策略：\n");

printf("1：选择出现次数多！ 2:暴力求解!\n");

scanf("%d", &Flag);

if (Flag == 1)//选择策略

{

FLAG = 1;

Strategy(T, mark\_1, mark\_2);

t1 = clock();

flag = Dpll\_youhua(T, 1, G);

t2 = clock();

t3 = (t2 - t1);

printf("时间：%4d ms\n", (t2 - t1));

}

else if (Flag == 2)

{

FLAG = 2;

t1 = clock();

flag = Dpll\_youhua(T, 1, G);

t2 = clock();

t3 = (t2 - t1);

printf("时间：%4d ms\n\n", t3);

}

if (flag == 1)

{

for (int i = 0; i < 70; i++)

{

if (filename\_fuben[i] == '.')

{

filename\_fuben[i + 1] = 'r';

filename\_fuben[i + 2] = 'e';

filename\_fuben[i + 3] = 's';

break;

}

}

fp = fopen(filename\_fuben, "wb");

fprintf(fp, "%d\n", 1);

printf("cnf文件的解为：\n");

for (int i = 1; i < variable + 1; i++)

{

if (mark\_ture[i] == 1)

{

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

printf("%6d", i);

}

else if (mark\_ture[i] == -1)

{

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

printf("%6d", -i);

}

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

}

fprintf(fp, "\n%d\n", t3);

printf("\ncnf的解输出成功！\n");

fclose(fp);

}

else

{

for (int i = 0; i < 70; i++)

{

if (filename\_fuben[i] == '.')

{

filename\_fuben[i + 1] = 'r';

filename\_fuben[i + 2] = 'e';

filename\_fuben[i + 3] = 's';

break;

}

}

fp = fopen(filename\_fuben, "wb");

fprintf(fp, "%d\n", 0);

fprintf(fp, "%d\n", t3);

fclose(fp);

printf("dpll算法处理失败，cnf文件无解！\n");

}

flag = 0;

isinit = 0;

}

else printf("操作失败！\n");

getchar(); getchar();

break;

case 0:

break;

default:

printf("输入错误！\n");

getchar(); getchar();

break;

}

}

getchar(); getchar();

break;

case 2:

while (Selet\_Sudoku)

{

system("cls");

printf("\n Menu for Sudoku \n");

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

printf(" 1.GenerateSuDoku 2.SolveSuDoku \n");

printf(" 3.SaveCnfFile 0.Exit \n");

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

if (scanf("%d", &Selet\_Sudoku))

switch (Selet\_Sudoku) {

case 1:

if (Create\_sudoku(&T, G, shuzu) == OK)

{

for (int i = 1; i < variable + 1; i++)

mark\_ture[i] = 0;

FLAG = 2;

Dpll\_youhua(T, 1, G);

ShuChu\_match(shuzu);

printf("求解后的棋盘为：\n");

Print\_SuDoKu(shuzu);

isinit = 1;

sign = 1;

printf("数独生成成功！\n");

}

else printf("数独生成失败！\n");

getchar(); getchar();

break;

case 2:

if (sign == 1)

{//保存副本

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

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

{

fuben[i][j] = shuzu[i][j];

hole[i][j] = 0;

}

sigg = 1;

while (sigg == 1)//判断输入的洞的数量是否合理

{

printf("请输入你想挖的洞的数量(结合自己游戏水平)（20-56）：");

scanf("%d", &counnt);

if (counnt <= 56 && counnt >= 20)

sigg = 0;

else sigg = 1;

}

if (Dig\_Hole(fuben, 1, 1, fuben[0][0],T,G) == OK)

printf("挖洞成功！\n");

else printf("挖洞失败！\n");

int n,m,k,t=1;

while (t == 1)//玩游戏的交互过程

{

int i = 1;

system("cls");

printf("\n 请开始你的数独游戏！ \n");

Print\_SuDoKu(fuben);

printf(" 1:填写数字 2：删除数字 \n");

scanf("%d", &i);

if (i == 1)

{

printf("请输入你想填入的位置：（例如：1 2 ）\n");

scanf("%d%d", &n, &m);

printf("请选择你想填入的数值（1-9）：");

scanf("%d", &k);

if (m >= 1 && m <= 9 && n >= 1 && n <= 9 && k >= 1 && k <= 9)

{

if (fuben[n - 1][m - 1] != 0) continue;

else fuben[n - 1][m - 1] = k;

}

else

{

printf("输入错误！请正确输入！\n");

continue;

}

}

else if (i == 2)

{

printf("请输入你想删除的位置：（例如：1 2 ）\n");

scanf("%d%d", &n, &m);

if (m >= 1 && m <= 9 && n >= 1 && n <= 9 )

{

if (fuben[n - 1][m - 1] == 0) continue;

else fuben[n - 1][m - 1] = 0;

}

else

{

printf("输入错误！请正确输入！\n");

continue;

}

}

printf("输入1表示继续填写，输入0表示放弃填写并查看答案！：");

scanf("%d", &t);

}

if (t == 0)//将对应文件存成.res文件，默认有解就存

{

int flag = 0;

for (int i = 0; i < 9; i++)//判断用户的填入是否正确

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

if (fuben[i][j] != shuzu[i][j]) flag = 1;

if (flag == 1)

{

printf("你填写的数独错误！");

printf("数独正确的解为：\n");

}

else printf("你填入的解正确！\n");

Print\_SuDoKu(shuzu);//输出正确的数独解

system("pause");

}

}

else printf("操作失败！\n");

break;

case 3:

if (isinit == 1)

{

check(T);

if (Save\_File(T) == OK)

printf("数据存入文件(Cnf)成功！\n");

else printf("数据存入文件(Cnf)失败！\n");

}

else printf("操作失败！\n");

getchar(); getchar();

break;

case 0:

break;

default:

printf("输入错误！\n");

getchar(); getchar();

break;

}

}

getchar(); getchar();

break;

case 0:

break;

default:

printf("输入错误！\n");

getchar(); getchar();

break;

}//end of switch

}

else

{

printf("输入错误！\n");

getchar(); getchar(); getchar();

}

}//end of while

printf("欢迎下次再使用本系统！\n");

system("pause");

return 0;

}//end of main()

**File.c(文件存储部分函数)：**

#include "SuDoKu.h"

/\*cnf文件读取板块\*/

//读取cnf文件

status ReadFile(headnode \*Head, bignode G,char filename[60])

{

int mid,countt=0;

char ch;

FILE \*fp;//文件指针

if ((fp = fopen(filename, "r")) == NULL)

{

printf("File open error！\n ");

return FALSE;

}

else

{

printf("\n所读取的cnf文件中的注释内容：\n");

while (fread(&ch, sizeof(char), 1, fp))

{

if (ch != 'p') printf("%c", ch);

else break;

}

for (int i = 0; i < 5; i++)

fread(&ch, sizeof(char), 1, fp);

fscanf(fp, "%d", &variable);

fscanf(fp, "%d", &sentences);

countt = sentences;

printf("cnf文件中的文字数：%d 公式数： %d\n", variable, sentences);

//读取公式部分

EdgeNode \*p = NULL, \*t = NULL;

HeadNode \*r = NULL, \*q = NULL;

SerchNode \*n = NULL, \*x = NULL;

//初始化索引表

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

{

G->mark = 0;

G->serchlist[i].fu\_mark = 0;

G->serchlist[i].zheng\_mark = 0;

G->serchlist[i].inside\_fu = NULL;

G->serchlist[i].inside\_zheng = NULL;

G->serchlist[i].next = NULL;

G->serchlist[i].outside\_fu = NULL;

G->serchlist[i].outside\_zheng = NULL;

}

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

q = r;

(\*Head) = r;

r->next = NULL;

r->inside = NULL;

r->count = 0;

r->mark = 0;

r->flag = 0;

while (fscanf(fp, "%d", &mid) != EOF&& countt )

{

if (mid != 0)

{

//读入变元

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

p->next = NULL;

p->num = mid;

p->mark = 1;

if (q->inside == NULL) q->inside = p;

else

{

t = q->inside;

while (t->next)

t = t->next;

t->next = p;

}

q->count++;

//索引表的建立

n = (SerchNode\*)malloc(sizeof(SerchNode));

if (mid > 0)

{

n->zheng\_mark = 1;

n->inside\_zheng = p;

n->outside\_zheng = q;

n->inside\_fu = NULL;

n->outside\_fu = NULL;

n->fu\_mark = 0;

n->next = NULL;

x = G->serchlist[mid].next;

if (!x) G->serchlist[mid].next = n;

else

{

while (x->next)

x = x->next;

x->next = n;

}

}

else

{

n->zheng\_mark = 0;

n->next = NULL;

n->inside\_zheng = NULL;

n->outside\_zheng = NULL;

n->inside\_fu = p;

n->outside\_fu = q;

n->fu\_mark = 1;

x = G->serchlist[-mid].next;

if (!x) G->serchlist[-mid].next = n;

else

{

while (x->next)

x = x->next;

x->next = n;

}

}

}

else

{

//读入下一个公式

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

r->next = NULL;

r->inside = NULL;

r->count = 0;

r->mark = 0;

r->flag = 0;

if (q->inside != NULL) q->mark = 1;

q->next = r;

q = r;

countt--;

}

}

}

return OK;

}

//输出cnf文件

void Traverse(headnode Head)

{

EdgeNode \*p = NULL;

HeadNode \*q = NULL;

q = Head;

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

while (q)

{

if (q->next != NULL)printf("%3d : ", q->count);

p = q->inside;

while (p)

{

printf("%6d", p->num);

p = p->next;

}

if (q->mark == 1) printf("\n");

q = q->next;

}

}

//存cnf文件小前提

void check(headnode Head)

{

EdgeNode \*p = NULL;

HeadNode \*q = NULL;

q = Head;

while (q)

{

sentences++;

q = q->next;

}

}

//保存cnf文件

status Save\_File(headnode Head)

{

EdgeNode \*p = NULL;

HeadNode \*q = NULL;

FILE \*fp;//文件111指针

char filename[60];

printf("input file name: ");

scanf("%s", filename);

if ((fp = fopen(filename, "wb")) == NULL)

{

printf("File open error！\n ");

return FALSE;

}

else

{

fprintf(fp, "%s", "p cnf ");

fprintf(fp, "%d ", variable);

fprintf(fp, "%d \n", sentences);

q = Head;

while (q)

{

p = q->inside;

while (p)

{

fprintf(fp, "%d ", p->num);

p = p->next;

}

fprintf(fp, "%d \n", 0);

q = q->next;

}

fclose(fp);

}

return OK;

}

//消除链表

void Realse\_Link(headnode \* Head)

{

EdgeNode \*p = NULL, \*r = NULL;

HeadNode \*q = NULL, \*t = NULL;

SerchNode \*n = NULL, \*x = NULL;

q = \*Head;

while (q)

{

p = q->inside;

while (p)

{

r = p->next;

free(p);

p = r;

}

q->inside = NULL;

q = q->next;

}

q = \*Head;

while (q)

{

t = q->next;

free(q);

q = t;

}

\*Head = NULL;

}

**DPLL.c（DPLL算法部分函数模块）**

#include "SuDoKu.h"

//标记函数，找到出现次数最多的数

void Strategy(headnode Head, int mark\_1[N], int mark\_2[N])

{

int temp = 0, t = 0;

//初始化

for (int i = 1; i <= N; i++)

{

mark\_1[i] = 0;

mark\_2[i] = 0;

}

//选择出现次数最到的

EdgeNode \*p;

HeadNode \*q;

q = Head;

while (q)

{

if (q->mark == 1)

{

p = q->inside;

while (p)

{

//设置标记，为零表示cnf文件中出现过

mark\_1[abs(p->num)]++;

p = p->next;

}

}

q = q->next;

}

//排序

for (int i = 1; i <= variable; i++)

{

temp = 0;

t = 0;

for (int j = 1; j <= variable; j++)

{

if (mark\_1[j] > temp)

{

t = j;

temp = mark\_1[j];

}

}

mark\_2[i] = t;

mark\_1[t] = 0;

}

}

//删除单子句

status DeleteSingle(bignode G, int flag)

{

SerchNode \*p;

if (flag > 0)

{//传进来为正

p = G->serchlist[flag].next;

while (p)

{

//为负，删除变元

if (p->fu\_mark == 1 && p->outside\_fu->mark == 1 && p->inside\_fu->mark == 1 && p->outside\_fu->count != 0)

{

p->inside\_fu->mark = 0;

p->outside\_fu->count--;

}

if (p->zheng\_mark == 1 && p->outside\_zheng->mark == 1)

{//为正，删除子句

p->outside\_zheng->mark = 0;

p->outside\_zheng->flag = flag;

}

p = p->next;

}

}

else

{//传进来为负

p = G->serchlist[-flag].next;

while (p)

{

//为正，删除变元

if (p->zheng\_mark == 1 && p->outside\_zheng->mark == 1 && p->inside\_zheng->mark == 1 && p->outside\_zheng->count != 0)

{

p->inside\_zheng->mark = 0;

p->outside\_zheng->count--;

}

if (p->fu\_mark == 1 && p->outside\_fu->mark == 1)

{//为负，删除子句

p->outside\_fu->mark = 0;

p->outside\_fu->flag = flag;

}

p = p->next;

}

}

return OK;

}

//恢复单子句

status Recovery(bignode G, int flag)

{

SerchNode \*p;

if (flag > 0)

{

p = G->serchlist[flag].next;

while (p)

{

//为负，恢复变元

if (p->fu\_mark == 1 && p->outside\_fu->mark == 1 && p->inside\_fu->mark == 0)

{

p->inside\_fu->mark = 1;

p->outside\_fu->count++;

}

else if (p->zheng\_mark == 1 && p->outside\_zheng->mark == 0 && p->outside\_zheng->flag == flag)

{//为正，恢复子句

p->outside\_zheng->mark = 1;

p->outside\_zheng->flag = 0;

}

p = p->next;

}

}

else

{//传进来是负数

p = G->serchlist[-flag].next;

while (p)

{

//为正，恢复变元

if (p->zheng\_mark == 1 && p->outside\_zheng->mark == 1 && p->inside\_zheng->mark == 0)

{

p->inside\_zheng->mark = 1;

p->outside\_zheng->count++;

}

else if (p->fu\_mark == 1 && p->outside\_fu->mark == 0 && p->outside\_fu->flag == flag)

{//为负，恢复子句

p->outside\_fu->mark = 1;

p->outside\_fu->flag = 0;

}

p = p->next;

}

}

return OK;

}

//寻找空子句

status FindEmptyClause(HeadNode \*Head)

{

HeadNode \*q = NULL;

q = Head;

while (q)

{

if (q->count == 0 && q->mark == 1)

return OK;

q = q->next;

}

return FALSE;

}

//判断是否为空集

status JudgeGroup(HeadNode \*Head)

{

HeadNode \*q = NULL;

q = Head;

while (q)

{

if (q->mark != 0) return FALSE;

q = q->next;

}

return OK;

}

//寻找假设的真值

status FindCount(HeadNode \*Head)

{

int flag = 0;

//先找单子句

EdgeNode \*p = NULL;

HeadNode \*q = NULL;

q = Head;

while (q)

{

p = q->inside;

if (q->count == 1 && q->mark == 1)//记录变元数量

{

while (p)

{

if (p->mark == 1)

{

flag = p->num;

goto next\_num;

}

p = p->next;

}

}

q = q->next;

}

next\_num:

{

if (flag > 0 && mark\_ture[flag] == 0)

{

mark\_ture[flag] = 1;

return flag;

}

if (flag < 0 && mark\_ture[-flag] == 0)

{

mark\_ture[-flag] = -1;

return flag;

}

}

if (FLAG == 1)

{

for (int i = 1; i <= variable; i++)

{

flag = mark\_2[i];

if (flag != 0 && mark\_ture[flag] == 0)

{

mark\_ture[flag] = 1;

return flag;

}

}

}

else if (FLAG == 2)

{

for (int i = 1; i <= variable; i++)

{

if (mark\_ture[i] == 0)

{

mark\_ture[i] = 1;

return i;

}

}

}

return 0;

}

//Dpll算法的主函数

status Dpll\_youhua(HeadNode \*Head, int now\_count, bignode G)

{

int next\_count;

if (JudgeGroup(Head) == OK) return OK;//判断是否为空

else

{

if (FindEmptyClause(Head) == OK)//检查是否出现空子句

{//出现空子句就恢复空子句并回溯

Recovery(G, now\_count);

mark\_ture[abs(now\_count)] = 0;

return FALSE;

}

else

{ //找一个合适的真值,并测试

next\_count = FindCount(Head);

if (next\_count == 0)return FALSE;

DeleteSingle(G, next\_count);

if (Dpll\_youhua(Head, next\_count, G) == OK) return OK;

else

{//将测试失败的数的负值带入检测

Recovery(G, next\_count);

DeleteSingle(G, -next\_count);

if (Dpll\_youhua(Head, -next\_count, G) == OK)

{

if (next\_count>0) mark\_ture[next\_count] = -1;

else mark\_ture[-next\_count] = 1;

return OK;

}

else

{//负数也测试失败，则标记该假设，返回上一个假设

Recovery(G, -next\_count);

mark\_ture[abs(next\_count)] = 0;

return FALSE;

}

}

}

}

}

//匹配棋盘

void ShuChu\_match(int shuzu[][9])

{

for (int t = 1; t < variable + 1; t++)

{

if (mark\_ture[t] == 1)

{

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

for (int j = 1; j <= 9; j++)

for (int k = 1; k <= 9; k++)

if (a[i][j][k] == t)

shuzu[i - 1][j - 1] = k;

}

}

}

**SuDoKu.c（数独部分板块函数）：**

#include "SuDoKu.h"

//数独函数部分

//输出数独棋盘

void Print\_SuDoKu(int arr[][9])

{

int i, j;

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

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

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

if (j % 3 == 0) printf("|");

if (arr[i][j] != 0) printf("%2d ", arr[i][j]);

else printf(" \_ ");

if (j == 8) printf("|\n");

}

if ((i + 1) % 3 == 0) printf("-------------------------------\n");

}

}

//调用读取函数

status Read\_SuDoKu(headnode \*Head, bignode G, int mid)

{

//读取公式部分

EdgeNode \*p = NULL, \*t = NULL;

HeadNode \*r = NULL, \*q = NULL;

SerchNode \*n = NULL, \*x = NULL;

q = \*Head;

while (q->next)

{

q = q->next;

}

if (mid != 0)

{

//读入变元

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

p->next = NULL;

p->num = mid;

p->mark = 1;

if (q->inside == NULL) q->inside = p;

else

{

t = q->inside;

while (t->next)

t = t->next;

t->next = p;

}

q->count++;

//索引表的建立

n = (SerchNode\*)malloc(sizeof(SerchNode));

if (mid > 0)

{

n->zheng\_mark = 1;

n->inside\_zheng = p;

n->outside\_zheng = q;

n->inside\_fu = NULL;

n->outside\_fu = NULL;

n->fu\_mark = 0;

n->next = NULL;

x = G->serchlist[mid].next;

if (!x) G->serchlist[mid].next = n;

else

{

while (x->next)

x = x->next;

x->next = n;

}

}

else

{

n->zheng\_mark = 0;

n->next = NULL;

n->inside\_zheng = NULL;

n->outside\_zheng = NULL;

n->inside\_fu = p;

n->outside\_fu = q;

n->fu\_mark = 1;

x = G->serchlist[-mid].next;

if (!x) G->serchlist[-mid].next = n;

else

{

while (x->next)

x = x->next;

x->next = n;

}

}

}

else

{

//读入下一个公式

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

r->next = NULL;

r->inside = NULL;

r->count = 0;

r->mark = 0;

r->flag = 0;

if (q->inside != NULL) q->mark = 1;

q->next = r;

q = r;

}

return OK;

}

//读入邻接表之初始化邻接表操作

status Isinit\_SuDoKu(headnode \*Head, bignode G)

{

HeadNode \*r = NULL, \*q = NULL;

//初始化索引表

for (int i = 1; i < N; i++)

{

G->mark = 0;

G->serchlist[i].fu\_mark = 0;

G->serchlist[i].zheng\_mark = 0;

G->serchlist[i].inside\_fu = NULL;

G->serchlist[i].inside\_zheng = NULL;

G->serchlist[i].next = NULL;

G->serchlist[i].outside\_fu = NULL;

G->serchlist[i].outside\_zheng = NULL;

}

//初始化存储表

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

q = r;

(\*Head) = r;

r->next = NULL;

r->inside = NULL;

r->count = 0;

r->mark = 0;

r->flag = 0;

return OK;

}

//数独之判断填入的数字是否正确

status SuDoKu\_Judge(int arr[][9], int row, int col, int n)

{

int i, j;

for (i = 0; i < 9; i++) {// 判断行重复

if (arr[row][i] == n) {

return FALSE;

}

}

for (j = 0; j < 9; j++) {// 判断列重复

if (arr[j][col] == n) {

return FALSE;

}

}

// 判断所在小九宫格重复

int x = (row / 3) \* 3;

int y = (col / 3) \* 3;

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

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

if (arr[i][j] == n) return FALSE;

return OK;

}

//匹配函数

void Match(int arr[][9], int a[][10][10])

{

int order = 1;

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

{

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

{

if (arr[i][j] == 0)

{

for (int k = 1; k <= 9; k++)

{

if (SuDoKu\_Judge(arr, i, j, k) == OK)

{

a[i + 1][j + 1][k] = order;

order++;

}

}

}

}

}

variable = order - 1;

}

//每空取值组合

void Create\_Cnf\_Position(int arr[][9], int a[][10][10], headnode \*H, bignode G)

{

int i, j, k, t;

int part\_1[10];

for (int i = 1; i <= 9; i++)//初始化

part\_1[i] = 0;

for (i = 0; i < 9; i++) //遍历数独列表

{

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

{

if (arr[i][j] == 0)//找到空格

{

for (k = 1; k <= 9; k++)//找出可能的取值

if (arr[i][j] == 0 && SuDoKu\_Judge(arr, i, j, k) == OK)

part\_1[k] = 1;

//输出可能的两两取值

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

if (part\_1[k] == 1)

{

for (t = k + 1; t <= 9; t++)

{

if (part\_1[t] == 1)

{

Read\_SuDoKu(H, G, -a[i + 1][j + 1][k]);

Read\_SuDoKu(H, G, -a[i + 1][j + 1][t]);

Read\_SuDoKu(H, G, 0);

}

}

part\_1[k] = 0;

}

}

}

}

}

//每行取值组合

void Create\_Cnf\_Row(int arr[][9], int a[][10][10], headnode \*H, bignode G)

{

int i, j, k, t;

int part\_2[10], sign\_2[10];

for (int i = 1; i <= 9; i++)//初始化

{

part\_2[i] = 0;

sign\_2[i] = 1;//记录每列还需要填的数

}

for (i = 0; i<9; i++) //遍历数独列表

{

for (j = 0; j < 9; j++)//找到可能要补的数

{

if (arr[i][j] != 0)

sign\_2[arr[i][j]] = 0;

}

for (t = 1; t <= 9; t++)//找到可能补数x的位置

{

if (sign\_2[t] == 1)//需要补数t，查找可能补t的位置

{

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

if (arr[i][k - 1] == 0 && SuDoKu\_Judge(arr, i, k - 1, t) == OK)

part\_2[k] = 1;

for (k = 1; k <= 9; k++)//全部输出

{

if (part\_2[k] == 1)

{

for (int r = k + 1; r <= 9; r++)

{

if (part\_2[r] == 1)

{

Read\_SuDoKu(H, G, -a[i + 1][k][t]);

Read\_SuDoKu(H, G, -a[i + 1][r][t]);

Read\_SuDoKu(H, G, 0);

}

}

part\_2[k] = 0;

}

}

}

sign\_2[t] = 1;

}

}

}

//每列取值组合

void Create\_Cnf\_Col(int arr[][9], int a[][10][10], headnode \*H, bignode G)

{

int i, j, k, t;

int part\_3[10], sign\_3[10];

for (int i = 1; i <= 9; i++)//初始化

{

part\_3[i] = 0;

sign\_3[i] = 1;//记录每列还需要填的数

}

for (j = 0; j<9; j++) //遍历数独列表

{

for (i = 0; i < 9; i++)//找到可能要补的数

{

if (arr[i][j] != 0)

sign\_3[arr[i][j]] = 0;

}

for (t = 1; t <= 9; t++)//找到可能补数x的位置

{

if (sign\_3[t] == 1)//需要补数t，查找可能补t的位置

{

for (k = 1; k <= 9; k++)//

if (arr[k - 1][j] == 0 && SuDoKu\_Judge(arr, k - 1, j, t) == OK)

part\_3[k] = 1;

for (k = 1; k <= 9; k++)//全部输出

{

if (part\_3[k] == 1)

{

for (int r = k + 1; r <= 9; r++)

{

if (part\_3[r] == 1)

{

Read\_SuDoKu(H, G, -a[k][j + 1][t]);

Read\_SuDoKu(H, G, -a[r][j + 1][t]);

Read\_SuDoKu(H, G, 0);

}

}

part\_3[k] = 0;

}

}

}

sign\_3[t] = 1;

}

}

}

//每小方块取值组合

void Create\_Cnf\_Block(int arr[][9], int a[][10][10], headnode \*H, bignode G)

{

int i, j, k, t, n;

int part\_4[10], sign\_4[10];

for (int i = 1; i <= 9; i++)//初始化

{

sign\_4[i] = 1;//记录每个小方块还需要填的数

part\_4[i] = 0;//记录可以填的位置

}

for (i = 0; i < 9; i += 3)

{

for (j = 0; j < 9; j += 3)

{

for (k = i; k < i + 3; k++)//找到小方块中可能填充的数字

{

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

{

if (arr[k][t] != 0)

sign\_4[arr[k][t]] = 0;

}

}

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

{

if (sign\_4[n] == 1)//接下来找出可能填入k的位置

{

for (k = i; k < i + 3; k++)//找到可能放入的位置

{

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

if (arr[k][t] == 0 && SuDoKu\_Judge(arr, k, t, n) == OK)

{

if (k % 3 == 0) part\_4[(t % 3) + 1] = 1;

else if (k % 3 == 1)part\_4[4 + (t % 3)] = 1;

else part\_4[7 + (t % 3)] = 1;

}

}

for (k = 1; k <= 9; k++)//全部输出

if (part\_4[k] == 1)

{

for (t = k + 1; t <= 9; t++)

if (part\_4[t] == 1)

{

if (k >= 1 && k <= 3)

Read\_SuDoKu(H, G, -a[i + 1][j + k][n]);

else if (k >= 4 && k <= 6)

Read\_SuDoKu(H, G, -a[i + 2][j + k - 3][n]);

else

Read\_SuDoKu(H, G, -a[i + 3][j + k - 6][n]);

if (t >= 1 && t <= 3)

Read\_SuDoKu(H, G, -a[i + 1][j + t][n]);

else if (t >= 4 && t <= 6)

Read\_SuDoKu(H, G, -a[i + 2][j + t - 3][n]);

else

Read\_SuDoKu(H, G, -a[i + 3][j + t - 6][n]);

Read\_SuDoKu(H, G, 0);

}

part\_4[k] = 0;

}

}

sign\_4[n] = 1;

}

}

}

}

//每空取值保证

void Insure\_Cnf\_Position(int arr[][9], int a[][10][10], headnode \*H, bignode G)

{

int i, j, k;

int part\_5[10];

for (int i = 1; i <= 9; i++)//初始化

part\_5[i] = 0;

for (i = 0; i < 9; i++) //遍历数独列表

{

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

{

if (arr[i][j] == 0)//找到空格

{

for (k = 1; k <= 9; k++)//找出可能的取值

if (arr[i][j] == 0 && SuDoKu\_Judge(arr, i, j, k) == OK)

part\_5[k] = 1;

//输出可能的取值

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

{

if (part\_5[k] == 1)

{

Read\_SuDoKu(H, G, a[i + 1][j + 1][k]);

}

part\_5[k] = 0;

}

Read\_SuDoKu(H, G, 0);

}

}

}

}

//每行取值保证

void Insure\_Cnf\_Row(int arr[][9], int a[][10][10], headnode \*H, bignode G)

{

int i, j, k, t;

int part\_6[10], sign\_6[10];

for (int i = 1; i <= 9; i++)//初始化

{

part\_6[i] = 0;

sign\_6[i] = 1;//记录每列还需要填的数

}

for (i = 0; i<9; i++) //遍历数独列表

{

for (j = 0; j < 9; j++)//找到可能要补的数

{

if (arr[i][j] != 0)

sign\_6[arr[i][j]] = 0;

}

for (t = 1; t <= 9; t++)//找到可能补数x的位置

{

if (sign\_6[t] == 1)//需要补数t，查找可能补t的位置

{

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

if (arr[i][k - 1] == 0 && SuDoKu\_Judge(arr, i, k - 1, t) == OK)

part\_6[k] = 1;

for (k = 1; k <= 9; k++)//全部输出

{

if (part\_6[k] == 1)

{

Read\_SuDoKu(H, G, a[i + 1][k][t]);

}

part\_6[k] = 0;

}

Read\_SuDoKu(H, G, 0);

}

sign\_6[t] = 1;

}

}

}

//每列取值保证

void Insure\_Cnf\_Col(int arr[][9], int a[][10][10], headnode \*H, bignode G)

{

int i, j, k, t;

int part\_7[10], sign\_7[10];

for (int i = 1; i <= 9; i++)//初始化

{

part\_7[i] = 0;

sign\_7[i] = 1;//记录每列还需要填的数

}

for (j = 0; j < 9; j++) //遍历数独列表

{

for (i = 0; i < 9; i++)//找到可能要补的数

{

if (arr[i][j] != 0)

sign\_7[arr[i][j]] = 0;

}

for (t = 1; t <= 9; t++)//找到可能补数x的位置

{

if (sign\_7[t] == 1)//需要补数t，查找可能补t的位置

{

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

if (arr[k - 1][j] == 0 && SuDoKu\_Judge(arr, k - 1, j, t) == OK)

part\_7[k] = 1;

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

{

if (part\_7[k] == 1)

{

Read\_SuDoKu(H, G, a[k][j + 1][t]);

}

part\_7[k] = 0;

}

Read\_SuDoKu(H, G, 0);

}

sign\_7[t] = 1;

}

}

}

//每小方块取值保证

void Insure\_Cnf\_Block(int arr[][9], int a[][10][10], headnode \*H, bignode G)

{

int i, j, k, t, n;

int part\_8[10][10], sign\_8[10];

for (int i = 1; i <= 9; i++)//初始化

sign\_8[i] = 1;//记录每个小方块还需要填的数

for (i = 0; i < 9; i++)//初始化

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

part\_8[i][j] = 0;

for (i = 0; i < 9; i += 3)

for (j = 0; j < 9; j += 3)

{

for (k = i; k < i + 3; k++)//找到小方块中可能填充的数字

{

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

{

if (arr[k][t] != 0)

sign\_8[arr[k][t]] = 0;

}

}

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

{

if (sign\_8[n] == 1)//接下来找出可能填入k的位置

{

for (k = i; k < i + 3; k++)//找到可能放入的位置

{

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

{

if (arr[k][t] == 0 && SuDoKu\_Judge(arr, k, t, n) == OK)

part\_8[k][t] = 1;

}

}

for (k = i; k < i + 3; k++)//全部输出

{

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

{

if (part\_8[k][t] == 1)

{

Read\_SuDoKu(H, G, a[k + 1][t + 1][n]);

}

part\_8[k][t] = 0;

}

}

Read\_SuDoKu(H, G, 0);

}

sign\_8[n] = 1;

}

}

}

//数独转换成函数

void Tranlate(int arr[9][9], headnode \*H, bignode F)

{

for (int i = 0; i < 10; i++)

for (int j = 0; j < 10; j++)

for (int k = 0; k < 10; k++)

a[i][j][k] = 0;

Match(arr, a);

Isinit\_SuDoKu(H, F);

Create\_Cnf\_Position(arr, a, H, F);

Create\_Cnf\_Row(arr, a, H, F);

Create\_Cnf\_Col(arr, a, H, F);

Create\_Cnf\_Block(arr, a, H, F);

Insure\_Cnf\_Position(arr, a, H, F);

Insure\_Cnf\_Row(arr, a, H, F);

Insure\_Cnf\_Col(arr, a, H, F);

Insure\_Cnf\_Block(arr, a, H, F);

}

//调用数独的函数

status Create\_sudoku(headnode \*H, bignode F, int arr[9][9])

{

int n, m, k, count = 0;

srand((int)time(0));

printf("正在创建数独棋盘！\n");

while (count <= 17)

{

n = rand() % 9;

m = rand() % 9;

k = (rand() % 9) + 1;

if (arr[n][m] == 0 && SuDoKu\_Judge(arr, n, m, k) == OK)

{

count++;

arr[n][m] = k;

}

}

Print\_SuDoKu(arr);

Tranlate(arr, H, F);

return OK;

}

**DigHole.c（挖洞法生成唯一解数独棋盘板块函数）：**

#include "SuDoKu.h"

//找到下一个要删除的点

int Delete\_Num(int arr[][9])

{

int k = 0;

for (int i = 0; i < 9; i++)//初始化

if (i % 2 == 0)

{

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

{

if (arr[i][j] != 0 && hole[i][j] == 0)

{

k = arr[i][j];

arr[i][j] = 0;

hole[i][j] = 1;

return (i + 1) \* 100 + (j + 1) \* 10 + k;

}

}

}

else

{

for (int j = 8; j >=0; j--)

if (arr[i][j] != 0 && hole[i][j] == 0)

{

k = arr[i][j];

arr[i][j] = 0;

hole[i][j] = 1;

return (i + 1) \* 100 + (j + 1) \* 10 + k;

}

}

return 0;

}

//判断洞是否可以挖

status Juge\_Hole(int arr[][9], int n, int m, int k, headnode H, bignode F)

{

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

if (i != k && SuDoKu\_Judge(arr, n, m, i)==OK)//判断填入的数是否符合数独规则

{

arr[n][m] = i;

if (resolve\_sudoku(arr, H, F) == OK)//判断是否有唯一解

{

arr[n][m] = k;

hole[n][m] = 1;

return FALSE;

}

arr[n][m] = k;

}

arr[n][m] = 0;

hole[n][m] = 1;

return OK;

}

//挖洞法主函数

status Dig\_Hole(int arr[][9],int n, int m, int k, headnode H, bignode F)

{

counnt--;

int next\_hole=0,sig=4;

if (counnt == 0) return OK;

else

{

if (Juge\_Hole(arr, n-1, m-1, k,H,F) == FALSE )//判断传入的是否有其他解

{//没解复原

arr[n-1][m-1] = k;

hole[n-1][m-1] = 1;

counnt++;

return FALSE;

}

else

{

next\_hole = Delete\_Num(arr);//找到下一个要删除的数

if (next\_hole == 0)return OK;

if (Dig\_Hole(arr, next\_hole / 100, (next\_hole / 10) % 10, next\_hole % 10, H, F) == OK)

return OK;

else

{

while (sig--)

{

next\_hole = Delete\_Num(arr);//找到下一个要删除的数

if (next\_hole == 0)return OK;

if (Dig\_Hole(arr, next\_hole / 100, (next\_hole / 10) % 10, next\_hole % 10, H, F) == OK)

return OK;

else return FALSE;

}

return FALSE;

}

}

}

}

//调用dpll判断是否有解

status resolve\_sudoku(int arr[][9], headnode H, bignode F)

{

int c[9][9];

for (int i = 0; i < 9; i++)//初始化

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

c[i][j] = 0;

for (int i = 0; i < 9; i++)//复制

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

c[i][j] = arr[i][j];

Tranlate(c, &H, F);

for (int i = 1; i < variable + 1; i++)

mark\_ture[i] = 0;

FLAG = 2;

if (Dpll\_youhua(H, 1, F) == OK) return OK;

else return FALSE;

}