

**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： 计算机1808**

**学 号： U201814719**

**姓 名： 马忠平**

**指导教师： 卢萍**

**报告日期： 2020.3.28**

**计算机科学与技术学院**

# 任 务 书

**设计内容**

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

**设计要求**

要求具有如下功能：

1. **输入输出功能：**包括程序执行参数的输入，SAT算例cnf文件的读取，执行结果的输出与文件保存等。(15%)
2. **公式解析与验证：**读取cnf算例文件，解析文件，基于一定的物理结构，建立公式的内部表示；并实现对解析正确性的验证功能，即遍历内部结构逐行输出与显示每个子句，与输入算例对比可人工判断解析功能的正确性。数据结构的设计可参考文献[1-3]。(15%)
3. **DPLL过程：**基于DPLL算法框架，实现SAT算例的求解。(35%)
4. **时间性能的测量：**基于相应的时间处理函数（参考time.h），记录DPLL过程执行时间（以毫秒为单位），并作为输出信息的一部分。(5%)
5. **程序优化：**对基本DPLL的实现进行存储结构、分支变元选取策略[1-3]等某一方面进行优化设计与实现，提供较明确的性能优化率结果。优化率的计算公式为：[(t-to)/t]\*100%,其中t 为未对DPLL优化时求解基准算例的执行时间，to则为优化DPLL实现时求解同一算例的执行时间。(15%)
6. **SAT应用：**将二进制数独游戏[5，6]问题转化为SAT问题[6]，并集成到上面的求解器进行问题求解，游戏可玩，具有一定的/简单的交互性。应用问题归约为SAT问题的具体方法可参考文献[3]与[6-9]。(15%)

**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](#_Toc36478129)

[1 引言 4](#_Toc36478130)

[1.1课题背景与意义 4](#_Toc36478131)

[1.1数独游戏发展背景 4](#_Toc36478132)

[1.1.2 SAT问题简介 4](#_Toc36478133)

[1.2 国内外研究现状 5](#_Toc36478134)

[1.3 课程设计的主要研究工作 6](#_Toc36478135)

[2 系统需求分析与总体设计 7](#_Toc36478136)

[2.1 系统需求分析 7](#_Toc36478137)

[2.2 系统总体设计 7](#_Toc36478138)

[3系统详细设计 9](#_Toc36478139)

[3.1 有关数据结构的定义 9](#_Toc36478140)

[3.2 主要算法设计 12](#_Toc36478141)

[3.2.1 CNF文件处理 12](#_Toc36478142)

[3.2.2 DPLL算法 14](#_Toc36478143)

[3.2.3 SoDoKu问题约束条件转化为cnf文件格式 19](#_Toc36478144)

[3.2.4 挖洞法生成唯一解 21](#_Toc36478145)

[4 系统实现与测试 22](#_Toc36478146)

[4.1 系统实现 22](#_Toc36478147)

[4.1.1系统功能 22](#_Toc36478148)

[4.1.2 数据结构定义 23](#_Toc36478149)

[4.1.3相关函数 25](#_Toc36478150)

[4.2 系统测试 29](#_Toc36478151)

[4.2.1.交互式主菜单测试 30](#_Toc36478152)

[4.2.2 SAT求解器测试 31](#_Toc36478153)

[4.2.3二进制数独测试 35](#_Toc36478154)

[5 总结与展望 40](#_Toc36478155)

[5.1 全文总结 40](#_Toc36478156)

[5.1 工作展望 41](#_Toc36478157)

[6 体会 42](#_Toc36478158)

[参考文献 44](#_Toc36478159)

[附录 45](#_Toc36478160)

# 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

实验设计的系统分为两个模块，分别是基于DPLL算法的SAT求解器模块和数独游戏生成模块。这两大板块之中各自具有一些小的模块。

1. 基于DPLL算法的SAT求解器：读取CNF文件并进行基于DPLL算法的求解（求解方式为三种，分别对应于不同的分支变元选择方法）、清空当前读入cnf文件后创建的文件结构、将当前数据结构输出为CNF文件。
2. 数独游戏生成模块：以CNF文件形式输出不同阶数的数独约束条件、挖洞法生成具有唯一解的数独地图、以cnf文件的形式输出当前数独地图。

整个系统程序流程图如图1-1所示。



图2- 1 程序流程图

# 3系统详细设计

## 3.1 有关数据结构的定义

实验中需要读取CNF文件，考虑到CNF公式的数据特性和逻辑关系，且为了有效支持回溯，选择邻接表的作为逻辑结构。

邻接表定义如下：

邻接表由头结点和横、纵两个方向的单链表组成，其中，头结点用于储存子句集的基本信息，横向的单链表用于储存从CNF文件中读取到的子句中出现的变元，纵向的链表作为子句公式的头结点储存子句的基本信息。

头结点（类型名：cnfFile）的数据项有：

1. int类型的numOfVariable，用于储存子句集涉及到的文字变元的个数。
2. int类型的numOfChildScentence，用于储存子句集的字句个数。
3. childSentence\*指针类型的sentence，指向储存子句集的邻接表。

纵结点（类型名：childSentence）的数据项有：

1. int类型的numOfWord，表示当前子句中变元文字的个数。
2. int类型的mark，用于标记在执行DPLL算法的过程中，该结点是否被删除，若该结点未被删除，该结点值为0，bei删除时其值等于单子句规则或分裂策略中导致该结点被删除的变元。比如，在单子句规则中，查找到单子句L，删除某一含有L的子句时，将其mark节点置为L。该结点主要用于支持DPLL算法的回溯。
3. bool类型的isCunzai，创建结点时该变量置为true。该结点用于挖洞生成具有唯一解的数独地图时，标记该结点是否被挖空。
4. childSentence\*指针类型的nextSen，用于链接各结点形成链表，指向下一个子句。
5. word\*指针类型的words，指向该子句的文字变元。

横结点（类型名：word）的数据项有

1. int类型的variable，储存该结点表示的文字变元。
2. bool类型的mark，用于标记该结点是否在执行DPLL算法的过程中因为分裂策略而被删除。
3. word\*指针类型的nextword，指向下一个文字变元结点。

邻接表的逻辑结构如下图3-1所示：



图 3- 1 储存邻接表的逻辑结构示意图

为提高算法的效率，使用链栈结构实现非递归的DPLL算法。栈的结点分为栈控制节点和链栈结点，定义分别如下：

栈控制结点（类型名stack）数据项有：

1. stackNode\*指针类型的top，指向栈顶
2. stackNode\*指针类型的tail，指向栈底

链栈结点（类型名stackNode）数据项有：

1. int类型的setTrueWord，表示在进行一次DPLL单子句规则或分裂策略中假定为真的变元。
2. bool类型的methodMark，标记该次化简使用的花间方法。使用单子句规则则将其置为false，使用分裂策略则将其置为true，表示当前子句集处理状态存在分支。（事实上，在使用分裂规则时，若选取的变元在子句集中仅以肯定的或者仅以否定的形式出现时，该变量也被置为false，这样做可以在回溯时避免进入不必要的分支）。
3. stackNode\*指针类型的 nextNode，指向紧随其后的栈结点。

在使用单子句规则或者分裂策略化简子句集时，将化简方式和选用的变元存入栈中，用于支持回溯。链栈的逻辑结构如图3-2所示。



图 3- 2 链栈的逻辑结构示意图

在分裂策略算法中，为优化变元选取策略，定义了一个统计cnf中出现的各文字变元的次数的结构。该结构由一个头结点和一个长度为N+1(N为变元的个数) 数组组成。头结点用于标记当前子句集中出现的最多的文字变元以及出现的次数，数组中储存变元出现的次数，数组元素的下标为变元文字，如文字变元L的相关信息储存在下标为L的数组元素中，下标为0的数组元素保留不使用。

头结点（类型名为headCount）数据项有：

1. int类型的max\_variable，用于标记当前子句集中出现次数最多的文字（该值恒为正数）。
2. int类型的sum，记录max\_variable出现的总次数。
3. bool类型的containPo，用于标记文字max\_variable是否在子句集中以肯定的方式出现过。
4. bool类型的containNe，用于标记文字max\_variable是否在子句集中以否定的方式出现过。（使用containPo和contianNe的意义详见上页：链栈结点数据项（b））。
5. count\*类型的count，指向统计文字出现次数的数组。

数组元素结点（类型名为count）数据项有：

1. int类型的posi，记录子句集中文字变元等于当前数组下标的文字出现的肯定的次数。
2. int类型的nega，记录子句集中文字变元等于当前数组下标的文字出现的否定的次数

在读取文件创建邻接表和化简子句集的过程中，对各个文字出现的次数进行统计和维护。分裂策略选取变元时，可直接从该结构的头结点中获取。统计数组的逻辑结构如图3-3所示。



图 3- 3 统计数组的逻辑结构示意图

## 3.2 主要算法设计

在头文件BinaryPuzzle1.h中宏定义了函数的状态表示参数

#define TRUE 1

#define FALSE 0

#define ERROR -1

#define OVERFLOW -2

#define OK 2

并添加typedef int Status语句，使用Status作为状态参量的类型。

本程序保留了两种选取分支变元的算法（两种选择方式的不同将会在之后说明，时候也将说明为什么同时将两种算法保留），两种方法涉及到的略有差别但功能大致相同的函数，通过在其后添加后缀“\_1”进行区分，如readFile函数和readFile\_1函数，两者功能大致相同，仅仅只是后者加入了对统计文字出现次数的统计数组的处理和维护。

### 3.2.1 CNF文件处理

1.读取cnf文件创建储存邻接表

程序同时保留了两种分支策略的变元选取方法（分别为优化前的方法和优化后的方法），故涉及到两种不同的文件读取算法。

第一种方法为读取文件创建储存邻接表，但不对统计数组做任何操作。该算法由两个函数实现，Status readFile(pCnfFile pcnf)和Status readAndCreate(pCnfFile pcnf, FILE\*file)（之所以将文件读取的算法分成两个函数进行实现，是为之后生成二进制地图时读取配置文件时的调用提供方便）。其中，pCnfFile为指向储存邻接表的指针，file为文件指针，在readFile中使用文件指针打开cnf文件，调用readAndCreate函数。在函数中使用fgets函数读取文件注释，然后使用fscanf函数读取各子句的文字变元，使用尾插法分配空间创建链表。算法流程的文字描述如下。

1. 以只读方式打开文件，打开成功，则执行b；打开文件失败则终止函数的执行，返回ERROR
2. 使用fget读取一行的首字母，判断该首字母是否为‘c’（注释行）或者‘p’（信息行），若为‘c’执行c，若为‘p’，执行d
3. 使用fgets函数读取注释行的内容，并将其打印在屏幕窗口上。跳转执行b
4. 使用fscanf函数读取信息行，并创建头结点，将子句集信息（子句个数，涉及到的变元个数）存入到头结点中。
5. 读入一行的首个文字，创建子句结点，若首个文字结点不为‘0’，则创建文字结点，储存当前的文字变元，创建链表。依次读取后面的变元，使用尾插法创建结点并赋值，直到读取到‘0’
6. 重复执行e，直到遇到文件结尾
7. 关闭文件，完成储存结构的建立，返回OK。

第二种方法与第一种方法基本相同，只是增加了对统计数组的创建和维护操作，该方式涉及到的函数有Status readFile\_1(pCnfFile pcnf,pHeadCount &\_phc)，和Status readAndCreate\_1(pCnfFile pcnf, FILE\*file,pHeadCount &\_phc)，在上述两个函数中调用操作统计数组的函数进行变元的统计。对统计数组的维护操作涉及到的函数有

1. void addToCount(pHeadCount phc, int variable)，将变元variable添加到统计数组中。判断变元variable的正负，将其对用的数组元素对应数据项+1，更新头结点的信息。
2. void delFormCount(pHeadCount phc, int variable, pCnfFile pcnf)，将变元variable从统计数组中删除一个。判断variable的正负，将其对应数组的对应数据项-1，更新头结点信息。
3. void resetCount(pCnfFile pcnf, pHeadCount phc)，重置当前cnf文件结构的变元统计。

2.保存CNF文件

创建一个生成CNF文件的函数Status writeSoDoKufile(pCnfFile pcnf)，该函数在生成数独cnf文件和生成当前储存结构的文件时均可调用。遍历整个储存结构，按照cnf文件的格式依次在文件中写入子句集的信息和各子句，该函数可用于查验创建出的储存结构的正确性。

1. 输出res文件

定义一个输出res文件的函数Status writeRes(char\* fileName,bool isSuccess , int sumTime,int \*answer,int n)，其中，fileName用于储存当前文件的文件名，isSuccsee用于标识当前储存结构中的子句集是否有解，sumTime是求解子句集时花费的时间，n为当前文件结构的解的路径的长度。该函数将储存于数组answer中的求解路径按照格式写入到res文件中。

### 3.2.2 DPLL算法

1.栈操作（涉及到回溯算法）

本程序通过栈进行回溯实现非递归的DPLL算法，涉及到一些基础的栈操作算法。

void initStack(pstack &sta)，初始化栈

void addStack(pstack sta, int word, bool methodMark)，将用于化简的变元word，以方法methodMark入栈，若使用的单子句化简规则，则将methodMark设置为false，若使用分支策略，则将其置为true，表示当前状态存在分支。

Status outStack(pstack sta, int &word, bool &methodMark)，出栈

bool cheakStack(pstack sta)，检测空栈

上述为栈的简单基本操作，接下来是本程序较为特殊的栈操作。

Status recovery(pstack sta, pCnfFile pcnf)函数，根据栈元素进行回溯的函数。获取栈中的函数，将其回溯到上一个分支处，并跳转至另一条分支。其流程如下：

1. 判断当前栈是否为空，若为空，返回ERROR，表示已经遍历完全部的分支，未找到解，即当前子句集无解；若不为空，则将栈顶元素出栈。
2. 得到栈顶元素的setTrueWord数据项，即上一次化简时选中的变元，根据该变元恢复储存结构中的相应的被删除的子句和文字。某一子句被删除时，其mark被设置为导致该子句被删除的文字，若setTrueWord与某子句的mark相同，则将mark置为0，表示该子句未被删除。
3. 若得到的栈顶元素的methodMark数据项为false，表示当前回溯没有碰到分支，跳转执行步骤a，若为true，表示遇到分支，执行d
4. 将setTrueWord\*（-1）作为选中的变元，对当前子句集结构进行化简（关于化简算法稍后说明），并将当前操作压栈，即调用addStack(sta,(-1)\*setTrueWord,false)函数，此处将methodMark置为false表示该处分支结构的前一个分支已经完成了遍历，下一次回溯时将不会进入前一个分支。
5. 回溯完成，返回OK。

调用一次recovery的结果是将当前子句集结构回溯至分支存在的地方，并跳转至另一条分支。例如，如图3-4所示，若当前结构进入S12状态，无解，开始回溯，调用一次recovery函数，可将储存结果回溯至S状态，再转入S2分支，且由于将methodMark置为false，则表明回溯之后，删除了S1所在的分支，下一次回溯将不再考虑该分支。



图 3- 4 举例说明调用recovery后的结果

和recovery对应的还有一个Status recovery\_1(pstack sta, pCnfFile pcnf,pHeadCount phc)函数，该函数功能与recovery相同，只是在函数中增加了维护统计变元的数组的操作。

另一算法为销毁当前的栈结构并输出求解路径，该算法包含两个函数;

void destroyStack(pstack sta, bool isSuccess, int \* answer, int &index);

void destroy(pStackNode psn, pstack sta, int &i, bool isSuccess, int \* answer, int &index);

destroyStack为外层函数，sta为指向当前栈结构的指针，isSuccess用于标识是否求解成功，若求解成功，将isSuccess设置为true，在销毁栈的同时，输出求解路径；若求解未成功，将isSuccess设置为false，此时仅销毁栈。

事实上，求解成功时也可以将isSuccess设置为false，此时将不会输出求解路径，作为替代输出，将由指针answer指向的数组储存求解路径。index作为数组操作的索引下标使用。

destroy函数为一递归函数，设置一栈结点指针，每一次调用，栈结点向栈底方向移动，递归出口为结点指针指向栈底。在函数中，先递归调用，再输出当前指针指向的栈结点的setTrueWord，或者将其填入数组answer，这样，可以做到从栈底到栈顶输出结点储存的文字变元，即为按求解的选用变元的顺序输出求解的路径。

2.查找单子句

查找单子句涉及到一个函数——int findSingal(pCnfFile pcnf)，通过遍历当前的储存结构，找到单子句，并将其变元作为函数的返回值。在本程序的储存结构中，单子句的标志是子句结点的mark数据项为0（表示当前子句未被删除）且numOfWord数据项为1（表示当前子句只含有一个文字变元，该值会在化简过程中的删除文字阶段进行更新，其值为子句的mark标志位true的文字结点的个数）。

1. 化简子句集

该算法涉及到的函数为Status simplyS(pCnfFile pcnf, int word)以及与之对应的Status simplyS\_1(pCnfFile pcnf, int word,pHeadCount phc)。考虑到单子句规则和分裂策略化简具有一定的共通性，编写了一个可以同时使用于两个算法的化简函数。该函数的执行结果为删除子句集中含有文字word的子句，并且删除每条子句中含有的 –word文字。simplyS\_1加入了对统计数组的维护操作。simplyS的具体流程描述如下：

1. 传入参数word
2. 遍历储存结构的子句，将含有word的子句结点的mark数据项置为word。（simplyS\_1中另有统计数组的维护操作）
3. 遍历储存结构的子句的每一个文字，将-word的文字结点的mark置为false，并将其对应子句集的numOfWord数据项-1.（simplyS\_1中另有统计数组的维护操作）
4. 化简完毕，返回OK。
5. 分裂策略中的变元选取

此处为两种算法的本质区别所在，在以下的描述中，我将两种选取策略分别称作规则1和规则2。

（1）规则1在读取文件时并未创建任何统计用的结构，选取变元的规则为遍历数组，选用第一次同时出现的肯定（正值）和否定（负值）的文字，将其正值作为返回值；若未出现在储存结构中既出现肯定或者否定的文字，则将最小（统计顺序）的一个文字作为返回值，并通知DPLL函数化简压栈时无分支，将methodMark置为false。

该规则涉及到的函数为int seclect(pCnfFile pcnf, int \*count, bool &mark)，其中，pcnf为指向当前储存结构的指针，count为在DPLL中分配的用于统计的数组（该数组在其他时候并未有任何操作，只用于该函数中记录变元出现，之所以在DPLL算法中分配而不是在该函数中分配是防止频繁的内存分配和销毁可能会影响到算法的时间效率）。mark用于标记使用规则一选中的变元是否同时出现肯定和否定，用于通知DPLL函数化简压栈时是否存在分支。

其流程描述如下：

1. 传入参数pcnf,count(DPLL函数中分配的长度为N+1的数组，N为当前储存结构涉及到的文字的种类数)，mark。
2. 将count中元素初始化为0
3. 将子句结点指针pcs指向第一个子句结点
4. 判断pcs结点的mark是否为0，为0则执行e
5. 将文字结点指针pw指向pcs的第一个子结点
6. 判断pw指向的结点的mark是否为true，若为true执行g，否则执行h
7. 取结点的文字变元variable，若variable为正，而count[variable]为负，则选用variable为选中的文字，返回variable，mark置为true，函数结束；若variable为正，count[variable]非负，则count[variable]++；若variable为负而count[-variable]为正，返回wariable，mark置为true函数结束；若variable为负，count[variable]非正，count[variable]—
8. pw沿着链表后移一个单位，若移动之后pw非空，执行f，若为空，执行i
9. pcs沿着链表后移，若移动之后pcs非空，执行d，否则执行j
10. 完整访问储存结构之后函数未结束，表明未出现同时存在肯定和否定的变元，将mark置为false
11. 遍历count数组，找到第一个非0的元素count[i]，若count[i]为正，返回i，若count[i]为负，返回-i，函数结束

（2）规则2在读取文件创建储存结构和化简子句集的过程中，创建和维护了一个统计数组，用统计数组结构的数据作为变元先择的参照。由于在创建储存结构和处理子句集的过程中已经对统计用的数组结构进行了维护，此处只需要查看头结点中的数据，将其返回给DPLL函数即可。该规则涉及到的函数为int select\_1(pCnfFile pcnf, pHeadCount phc, bool &mark)，phc为指向统计结构头结点的指针。mark的值由头结点的containPo及contianNe决定，两者同时为true时，mark置为true，否则mark置为false。

关于两种规则算法效率的分析：

理论上来说，规则2选用分裂变元的方法优先对较为复杂的变元进行了剪枝处理，其效率应该更高。

但是，在实际实验测试中，我发现，在处理一些cnf文件时，规则1的效率更高，规则2选取变元的执行效率反而较低。在测试了大量的cnf文件之后，我发现，在处理变元数量较少、涉及到的子句数量较少时，规则1的时间效率更高，而在处理变元和子句数量达到一定量级的cnf文件时，规则2的优势更加明显。

经过多次测试，猜测其原因如下：

单就分支变元的选择来说，规则2优先剪去了复杂的变元分支，使得求解SAT时化简的步骤大幅度减少；但是，该方法需要在创建储存结构和化简子句集时，需要一定的计算资源去创建和处理统计文字变元出现次数的数组结构。在cnf文件涉及到的变元较少、子句较少时，优化分支变元文字选取方式这一方法对DPLL算法时间效率提升的贡献有限。使用规则2时，优化变元选取所带来的效率的提升不能抵扣处理用于统计的数组结构所消耗的效率。相反，规则1选取变元时并未消耗许多计算资源，其分支策略选取规则虽然较差，但在算例复杂度较小时，规则2对分支策略的优化体现出来的优异性就不是很明显。这就表现为在cnf算例的复杂度较低的情况下，规则1的时间效率更高。在cnf文件算例复杂度较高、变元较多、子句数量较多时，规则2分支策略选取变元时带来的优异性更加明显，足以抵消处理统计数组结构所消耗的时间资源，这时就表现为规则2的效率更高。

为了同时兼顾复杂度较高的和较低的cnf文件求解效率，提升求解的平均时间，我在将两种方法集成到程序中之后，选择另外编写一种同时采用两种规则的SAT求解方法。但因为程序无法在事先判断cnf算例的复杂程度，无法在事先决定选择的规则。我最终选择使用求解算例所用的时间来判断算例的复杂程度。

我后来决定采取如下的方式进行cnf算例的求解：先尝试使用规则1对算例进行求解，每一轮化简之后记录使用该规则后消耗的总时间，若求解的总时间超过了一定时间（记为t0），则证明当前的算例复杂度较高，此时重置储存结构，创建统计数组结构，采用规则2进行DPLL求解。

通过比较两种规则分别求解不同复杂度算例时花费的时间，我最终将t0确定为20000ms。

1. DPLL函数主体

同样的，两种变元选取规则对应于两个DPLL函数，如下所示

Status DPLL(pCnfFile pcnf, pstack &stack,int beginTime,bool change)，stack为实现非递归的数组，beginTime用于记录开始的时间，change被用来标记是否在时间超过20000ms之后更换使用的规则。

Status DPLL\_1(pCnfFile pcnf, pstack &stack,pHeadCount phc)，phc为指向统计数组结构的指针。

两个函数流程大同小异，只是在细节的处理上有少许差异。函数DPLL需要在开始的时候分配一个长度为pcnf->numOfVariable的数组空间并在调用完成后销毁该空间，DPLL\_1则没有此操作，此外，另一不同之处在于DPLL\_1调用的是添加后缀”\_1”的适合于规则2的方法。

两个函数的基本流程如下：

1. 传入参数，pcnf为指向子句集储存结构的指针
2. 调用函数查找panf中的单子句
3. 若找到单子句，使用化简规则基于找到的单子句化简当前子句集，若未找到单子句，执行f
4. 若子句集为空，返回TRUE；若子句集含有空子句，执行e；否则，执行 b
5. 若当前栈为空，返回FALSE，否则，使用当前的栈，调用recovery函数进行回溯，回溯完成后，跳转至b
6. 调用函数选择分裂策略中使用的变元
7. 根据得到的f中变元对子句集进行化简，化简完毕后跳转至b

### 3.2.3 SoDoKu问题约束条件转化为cnf文件格式

本程序将数独问题转化为SAT问题，将其4、6、8阶数独的约束条件分别输出至cnf文件之中，在后面的数独地图的生成和处理过程中，通过读入文件的内容结合已有条件转化为适合于DPLL算法进行求解的储存格式。

本程序并未采用附加变元的思想，对于4，6，8阶数独地图，均使用按行序的方式进行布尔变元的变元编码。如图3-5所示。



图 3- 5 变元编码图

对于二进制数独问题，有三个条件对其进行限制：

1. 横向或者纵向的连续三个数不能相同。该限制条件容易理解，且易于输出为cnf文件。只需要将每一行及每一列的连续三个同时取正和同时取负分别作为一条子句输出至cnf文件中，并以0结尾即可。该算法用两个嵌套的二层循环即可实现。
2. 同一行或同一列的0，1个数相同。换言之，同一行的任意五个变元的值不能相同。将该条件输出为cnf文件时，需要先生成1，2，3…n中取出(n/2+1)个数的所有不同的选取方式（n为阶数），共C (n,n/2+1)种，再根据不同的组合方式将每一行、每一列对应的变元分别同时设置为正和同时设置为负。

生成不同的组合数时，我选择使用二进制串来表示其形式，0表示不选取，1表示选取。我将一个数从0递增至2n-1，将这之中的每一个数转化为二进制形式，统计二进制形式中出现的1的个数，若出现的个数为n/2+1，则将该二进制数转化为二进制串并存放入事先设定好的数组中。最后可以从数组中得到所有的组合方式。

1. 约束条件3为任意两行输出结果不相同。

以八阶为例，对指导书中给出的表达式进行化简

¬{[(51∧71)∨(¬51∧¬71)]∧[(52∧72)∨(¬52∧¬72)]∧…[(58∧78)∨(¬58∧¬78)]} (3.1)

将¬拿进括号最里面，整理得：

{[(51∨71)∧(¬51∨¬71)]∨[(52∨72)∧(¬52∨¬72)]∨…∨[(58∨78)∧(¬58∨¬78)]}（3.2）

将公式3.2完全展开有一些困难，但我们很容易能发现展开过程中的规律。类比于乘法分配律，可以发现，将公式3.2转化为主合取范式，其每一个极大项，总包含(51∨71)与(¬51∨¬71)二者之一，(52∨72)与(¬52∨¬72)二者之一……(58∨78)与(¬58∨¬78)二者之一，也就是说，前面所列出的八组式子中每组选出一个，全部进行析取，得到的就是最终的每一个极大项的形式。换言之，将公式3.2化简得到的CNF范式就是前面列出的八组式子每组选出一项组成的析取式。每一组有两种不同的选择方式，一个形如3.2的八阶公式，可展开为28条不同的CNF范式，而8行取2行有C（8，2）种不同组合，同时考虑到列具有同样的限制，一个8阶二进制数独的限制条件三，可转化为2\*C(8,2)\*28条CNF范式。类比可得，一个n阶二进制数独的限制条件3，可以转化为2\*C(n,2)\*2n条CNF范式。我们很难通过手写这些式子，但可以通过编程将上述CNF范式打印出来。

这里同样需要用到C(n,2) 种组合方式，按照与（2）种相同的思路即可得到组合的不同形式；将0 ~ 2n-1转化为二进制串，可以得到2n种不同的选择方式，二进制串种，1表示选择形如(51∨71)这样的取正的分式，0表示取负的分式。在n行种使用组合方式选出组合，再使用选择方式选择不同的分式，可打印出全部的限制条件3。

该板块涉及到的函数有：

void printCondition1(FILE \* file, int n)，打印条件1，n表示阶数，在printCondition中调用。

void printCondition2(FILE\*file, int n)，打印条件2，在printCondition中调用。

void printCondition3(FILE\*file, int n)，打印条件3，在printCondition中调用。

void printCondition(void)，打印所有的条件到cnf文件中。

bool\* generateCombination(int n, int &len)，生成C(n,n/2+1)种不同的组合方式，用二维bool数组保存。

bool\* generateZuhe(int n, int &len)，生成条件三种的不同选取组合的方式，用二维bool数组储存。

bool\* printCn2(int n, int &len)，生成条件三种的C(n,2)种不同的选取方式，用二维bool数组保存。

### 3.2.4 挖洞法生成唯一解

1.生成二进制数独地图

该函数函数头为void generateSoDuKuMap(pCnfFile pcnf, int n)，其中n表示阶数。函数的流程如下：

1. 创建长度为n\*n的数组
2. 其初始化数组，并在数组种随机填入2~4个0或者1作为生成地图的初始条件，保证生成的地图的随机性
3. 读入配置文件，创建储存结构，将初始条件作为单子句结点添加到储存结构中
4. 调用DPLL函数对当前的储存结构进行求解，若无解跳转至b，若有解，执行e
5. 根据DPLL求解的路径，得到一个填满的二进制数独地图
6. 读取想要尝试挖空的数目
7. 挖洞法生成具有唯一解的二进制数独地图
8. 将生成的二进制数独地图打印至屏幕
9. 输出cnf文件

2.挖洞法

挖洞法可由已经填好的二进制数都地图反向生成具有唯一解的地图，挖洞法挖洞的路径为answer数组中储存的元素的顺序，即使用DPLL算法求解的路径。使用该路径更能使生成的数度地图具有随机性，而且可避免出现得到的数独地图的空格主要集中在一个地方的情况。其具体流程如下：

1. 设置一个指向地图某个方块的指针，指向第一个方块
2. 若当前方块为1，将其更换为0；若为0，更换为1。并根据更换改变储存结构的相应位置
3. 使用DPLL求解，若无解，证明该挖去该方块不会导致多解的出现，挖空该方块；若有解，证明挖空该方块会导致多解，恢复该方块原来的值
4. 将指针移向下一格。若指针到达结尾，停止该函数，返回结果。否则执行b

3.以cnf格式输出当前二进制数独地图文件

遍历创建好的储存结构，输出为cnf文件。

# 4 系统实现与测试

## 4.1 系统实现

使用的开发工具为Visual Studio 2019，实验环境条件：i7 6700 2.6Ghz,16G内存

### 4.1.1系统功能

该系统为一个基于DPLL算法的SAT求解器，对于给行的算例进行求解，输出求解结果，统计求解时间。具体功能如下：

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 数据结构定义

系统实现使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. 数据元素类型定义

数据结构的表示（存储结构）用类型定义（typedef）描述。

typedef int status；

1. 宏定义

预定义常量和类型，用于判断函数结果的状态

#define TRUE 1

#define FALSE 0

#define ERROR -1

#define OVERFLOW -2

#define OK 2

#define ADDR ". \\textFile\\" //文件输出的路径

1. 储存邻接表定义

定义一个结构体作为邻接表的头结点，储存当前结构的基本信息。结构名为cnfFile，结构体指针\*pCnfFile,具体定义及其成员如下所示：

typedef struct cnfFile {

int numOfVariable;//涉及到的文字的个数

int numOfchildSentence;

struct childSentence\* sentence;

}cnfFile, \*pCnfFile;

定义一个结构体作为邻接表的纵结点，储存子句集的子句，结构名为childSentence，结构体指针\*pChildSentence,具体定义及其成员如下所示：

typedef struct childSentence {

struct childSentence \* nextSen;//指向另一个子句

struct word\* words; /\*指向文字，当此项为空时，说明为空子句\*/

int numOfword; //该子句中文字的个数

int mark; /\*标记该节点是否被删除，当mark为0时，表示该节点存在，否则该节点的值等于分裂规则或者单子句规则中导致该字句被删除的文字\*/

bool isCunzai; /\*仅用于生成随机数独地图时，标记当前空格是否被删除掉\*/

}childSentence, \*pChildSentence;

定义一个结构体作为邻接表的横结点，储存各个变元。结构名为word，结构体指针\*pWord,具体定义及其成员如下所示：

typedef struct word {

int varibale; //文字变元

bool mark; //标记，用于标记该节点是被删除

struct word \*nextword; //单链表指向下一个文字

}word, \*pWord;

1. 栈定义

栈管理结点：

typedef struct stack {

struct stackNode \*top;

struct stackNode \*tail;

}\*pstack;

栈结点：

typedef struct stackNode {

int setTrueWord; /\*在单子句规则选中的单子句文字，或者在分裂规则中选中的假定为真的文字\*/

bool methodMark; /\*标记该次化简使用所使用的规则，false--单子句规则，使用回溯时直接恢复即可，true--分裂规则使用回溯时往下深入\*/

stackNode\* nextNode;

}stackNode, \*pStackNode;

1. 统计数组结构定义

typedef struct count {

int posi; //储存当前结构中存在的正的文字的个数

int nega; //当前文字负的个数

}count,\*pCount;

typedef struct headCount {

int max\_variable; //当前结构中个数最多的文字变量

int sum; //文字最多的文字变量的总数

bool containPo; //标记当前数量最多的文字是否存在正数

bool cintainNe; //标记当前结构中是否存在负数

pCount count; //指向统计数组

}headCount,\*pHeadCount;

### 4.1.3相关函数

整个程序分为三大模块，分别是DPLL算法模块，唯一解的二进制数独生成模块和主函数模块，其中，主函数模块的调用详情将放到另外两个模块中列出。

1. DPLL算法模块

通过设计文件储存结构，管理文件储存结构，执行DPLL算法，打印DPLL文件等来实现系统的基于DPLL算法的SAT求解的基础功能。

1. 栈基本操作函数：

初始化栈：void initStack(pstack &sta)

入栈：void addStack(pstack sta, int word, bool methodMark)

出栈：Status outStack(pstack sta, int &word, bool &methodMark)

检测空栈：bool cheakStack(pstack sta)

销毁栈并递归输出路径的函数：void destroyStack(pstack sta, bool isSuccess, int \* answer, int &index)

1. 根据栈顶元素进行回溯：Status recovery(pstack sta, pCnfFile pcnf)
2. 单子句化简函数：Status simplyS(pCnfFile pcnf, int word)
3. 检测子句中是否含有目标文字的函数：bool checkWord(pChildSentence pcs, int word)
4. 选择分裂变元的函数：int seclect(pCnfFile pcnf, int \*count, bool &mark)
5. 寻找单子句的函数：int findSingal(pCnfFile pcnf)
6. 检测是否存在空子句的函数：bool checkNull(pCnfFile pcnf)
7. DPLL算法主函数：Status DPLL(pCnfFile pcnf, pstack &stack,int beginTime,bool change)
8. 文件读取函数：Status readFile(pCnfFile pcnf,char \*pFilename)
9. 根据读取的文件创建储存结构的函数：Status readAndCreate(pCnfFile pcnf, FILE\*file)
10. 销毁当前储存结构的函数：void destroyCNF(pCnfFile pcnf)
11. 销毁储存结构中的子句的函数：void destroyChildScen(pChildSentence pcs)
12. 完成求解后重置储存结构到港读入文件的状态的函数：void resetPcnf(pCnfFile pcnf)
13. 输出res文件的函数：Status writeRes(char\* fileName,bool isSuccess , int sumTime,int \*answer,int n)

DPLL模块算法的主要函数调用流程图如图4-1、4-2所示



图 4- 1 DPLL模块主要函数的调用流程图-DPLL



图 4- 2 DPLL模块主要函数调用流程图-recovery

DPLL函数和其他函数主要是main函数进行调用，main函数中执行一次完整的DPLL算法具体调用流程如下

1. 调用readFile函数读取文件，readFile函数调用readAndCreate函数创建储存结构。
2. 调用DPLL算法，对当前储存结构中的子句集进行求解
3. 调用destroyStack函数，销毁栈并输出求解路径；在该函数中调用destroy函数递归销毁栈。
4. 调用writeRes函数输出res文件
5. 调用resetPcnf函数重置求解过程中更改的储存结构
6. 根据输入调用destroyCNF函数销毁当前的储存结构；该函数中调用destroyChildScen销毁子句结构。

使用另一种规则时，调用函数的方式基本一致。

1. 唯一解的二进制数独生成模块

该模块通过将二进制数独求解规约化为SAT求解，挖洞生成具有唯一解的二进制数独，打印二进制数独等实现SAT求解器在二进制数度求解和生成中的应用。

1. 生成数独地图的函数：void generateSoDuKuMap(pCnfFile pcnf, int n)
2. 输出cnf文件的函数：Status writeSoDoKufile(pCnfFile pcnf)
3. 打印数独地图的函数：void printMap(int \*digHole, int \*answer, int n)
4. 挖洞法挖空地图方格的函数：void digMap(pCnfFile pcnf, int \*digHole, int numOfHole, int n)
5. 更改结点变元，进行DPLL求解的函数：Status changeAndDPLL(pCnfFile pcnf, int num, int n)
6. 查找目标单子句，返回指针的函数：pChildSentence findNum(pCnfFile pcnf, int num, bool &bol)
7. 添加结点至当前储存结构的函数：void addNodeToPcnf(pCnfFile pcnf, int word)
8. 打印限制条件的函数：void printCondition(void)
9. 输出限制条件为cnf文件的函数：void printCondition1(FILE \* file, int n)
10. 输出条件一CNF范式的函数：void printCondition2(FILE\*file, int n)
11. 输出条件二CNF范式的函数：void printCondition3(FILE\*file, int n)
12. 输出条件三CNF范式的函数：bool\* generateCombination(int n, int &len)
13. 统计bol数组中true的个数的函数：int countNumOfTrue(bool bol[], int n)
14. 生成n中取n/2+1个的不同组合方式的函数：bool\* generateCombination(int n, int &len)
15. 生成2n中不同选择方式的函数：bool\* generateZuhe(int n, int &len)
16. 生成n个中取2个不同组合方式的函数：bool\* printCn2(int n, int &len)

该模块涉及到许多DPLL算法的调用，主要函数调用流程图如图4-3、4-4所示



图 4- 3 主要函数调用流程图-printCondition



图 4- 4 主要函数调用流程图-GenerateSoDoKuMap函数

Main函数中根据用户输入对printCondition函数、GenerateSoDoKuMap进行调用。

## 4.2 系统测试

常用的软件测试基本思路为：

1. 以合法的交互输入测试程序的功能，多轮测试，检验程序功能实现程度，程序功能的完备性，对不同合法数据处理是否具有百分之百的正确性。
2. 以非法的形式的输入测试程序遇到分法输入或非法数据时能否正常执行并输出错误提示信息，测试程序的健壮性。

### 4.2.1.交互式主菜单测试

交互式主菜单的功能：通过检测不同的用户输入跳转至不同的程序模块，设置程序出口。

设计目标：

（1）根据用户输入正常跳转至对应的程序模块或者退出程序。

（2）检测到非法输入时，提示输入错误，并等待用户的重新输入。

测试数据样例：测试数据样例如表4-1所示：

表4-1测试数据样例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试编号 | 测试用例 | 用例说明 | 理论结果 |
| 1 | 1 | 合法输入，测试跳转功能 | 跳转进入SAT菜单 |
| 2 | 2 | 合法输入，测试跳转功能 | 跳转进入数独菜单 |
| 3 | 0 | 合法输入，测试程序退出 | 程序退出 |
| 4 | A | 非法输入，字符型 | 提示重新输入 |
| 5 | 3 | 非法输入，越界 | 提示重新输入 |

程序测试：

（1）测试编号1：运行结果如图4-5所示，成功跳转至SAT求解器菜单，符合理论结果

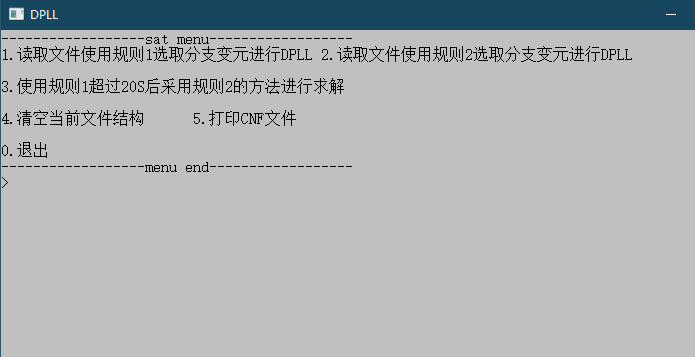


图 4- 5 交互式主菜单1号测试用例结果

（2）测试编号2： 运行结果如图4-6所示，程序正常跳转至二进制数独菜单，符合理论结果

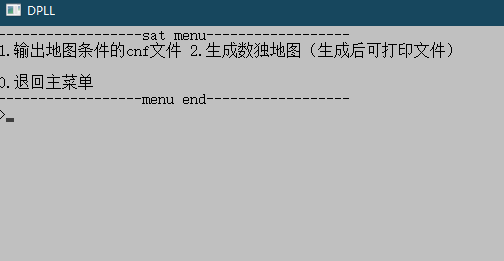


图 4- 6 交互式菜单2号测试用例结果

（3）测试编号3： 程序正常退出，符合理论结果

（4）测试编号4： 运行结果如图4-7所示，程序提示输入格式错误，提示用户重新输入，符合理论结果

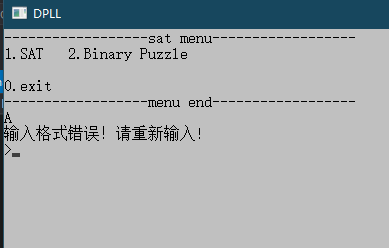


图 4- 7 交互式菜单4号测试用例运行结果

（5）测试编号5： 运行结果如图4-8所示，符合理论结果

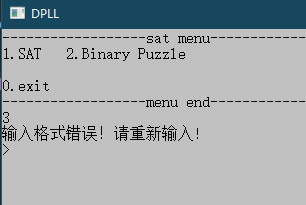


图 4- 8 交互式菜单5号测试用例运行结果

### 4.2.2 SAT求解器测试

1.读取CNF文件并进行DPLL求解的测试（基础功能测试）

基础功能：读取CNF文件，并通过DPLL求解得到求解路径并打印输出res文件，具有一定的交互功能。

设计目标：

1. 能够读取存在的CNF文件
2. 对不存在的文件，提示文件不存在并等待用户的再次输入
3. 正确打印求解路径和求解时间
4. 输出格式正确的res文件

测试用例：测试用例如表4-2所示（选项1，2用于算法效率比较之用，此处之测试选项3）：

表4-2 测试样例

|  |  |  |  |
| --- | --- | --- | --- |
| 编号 | 测试用例 | 说明 | 理论结果 |
| 1 | problem2-50.cnf | 合法输入，能求解的算例 | 提示有解，输出求解路径，输出求解时间，输出res文件 |
| 2 | tst\_v10\_c100.cnf | 合法输入，不满足算例 | 提示无解，输出求解时间，输出res文件 |
| 3 | 0000.cnf | 非法输入，文件不存在 | 提示文件不存在 |

测试结果：

1. 测试编号1：运行结果如图4-9所示，符合预期。

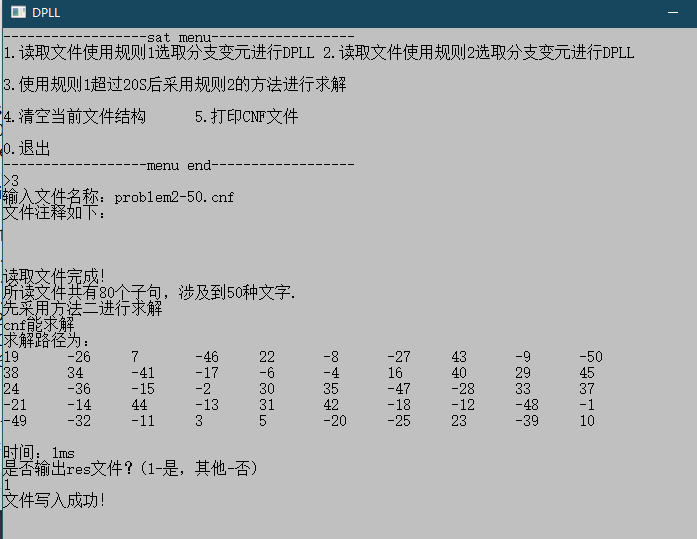


图 4- 9 SAT可行性测试-编号1

输出的res文件如图4-10所示：

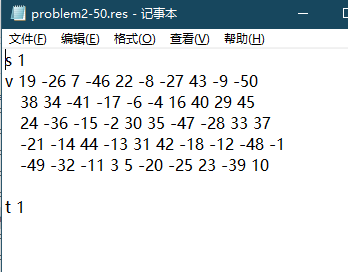


图 4- 10 测试编号1输出的res文件

（2）测试编号2：运行结果如图4-11所示，输出的res文件如图4-12所示，符合预期。

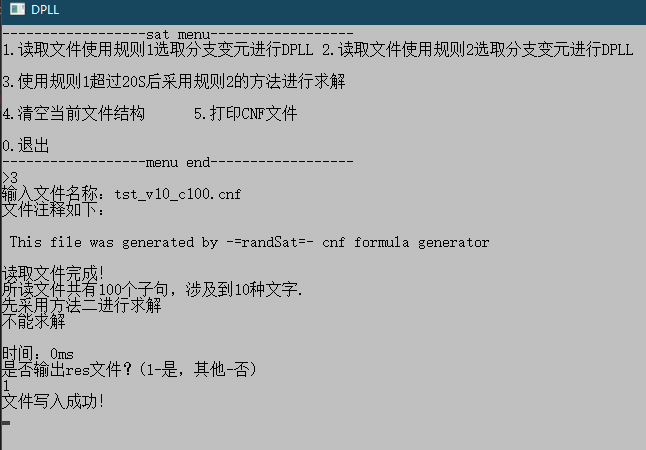


图 4- 11 SAT可行性测试-编号2

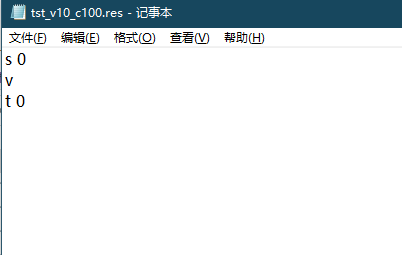


图 4- 12 测试编号2输出的文件

1. 测试编号3：测试结果如图4-13所示，符合预期。

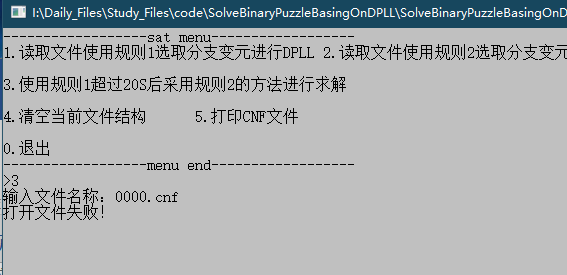


图 4- 13 SAT可行性测试-编号3

2. 读取CNF文件并进行DPLL求解的测试（性能测试）

测试基于DPLL算法的SAT求解器执行效率，同时测试采用选项１，２，３的方法进行测试，测试内容为①CNF文件有解。②CNF文件无解。测试多组算例，分别记录测试时间．测试结果如表4-3所示

表4-3 SAT效率测试

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 编号 | 测试用例 | 变元数 | 用例说明 | 规则1需要的时间/ms | 规则2需要的时间/ms | 规则3需要的时间/ms |
| 1 | problem3-100.cnf | 100 | 可满足的S算例 | 556 | 589 | 269 |
| 2 | problem2-50.cnf | 50 | 可满足的S算例 | 1 | 39 | 1 |
| 3 | problem6-50.cnf | 50 | 可满足的S算例 | 0 | 73 | 0 |
| 4 | problem11-100.cnf | 100 | 可满足的S算例 | 65 | 27 | 63 |
| 5 | tst\_v10\_c100.cnf | 100 | 可满足的S算例 | 2 | 42 | 2 |
| 6 | bart17.shuffled-231.cnf | 231 | 可满足的M算例 | 506 | 17 | 508 |
| 7 | sud00001.cnf | 301 | 可满足的M算例 | 108 | 9522 | 106 |
| 8 | sud00009.cnf | 303 | 可满足的M算例 | 36 | 1757 | 36 |
| 9 | sud00021.cnf | 308 | 可满足的M算例 | 757 | 171 | 756 |
| 10 | sud00079.cnf | 317 | 可满足的M算例 | 92 | 383 | 98 |
| 11 | sud00082.cnf | 224 | 可满足的M算例 | 502 | 419 | 504 |
| 12 | sud00012.cnf | 232 | 可满足的M算例 | 524 | 178 | 536 |
| 13 | bart17.shuffled-231.cnf | 231 | 可满足的M算例 | 510 | 13 | 510 |
| 14 | tst\_v200\_c220.cnf | 200 | 可满足的M算例 | 1 | 1 | 1 |
| 15 | tst\_v200\_c210.cnf | 200 | 可满足的M算例 | 1 | 1 | 1 |
| 16 | u-5cnf\_3500\_3500\_30f1.  shuffled-30.cnf | 30 | 不可满足的算例 | 335 | 359 | 334 |
| 17 | u-problem10-100.cnf | 100 | 不可满足的算例 | 131 | >  10000 | 131 |
| 18 | qg4-08.cnf | 512 | 不可满足的算例 | 53781 | 237252 | 53782 |

说明：程序求解cnf算例效率较低，求解L算例花费时间过长，

### 4.2.3二进制数独测试

1.数独约束条件输出为cnf文件测试

该模块将数独问题的约束条件转化为SAT问题并输出为cnf文件,供之后生成二进制棋盘的时候调用.

其交互板块如图4-14所示:

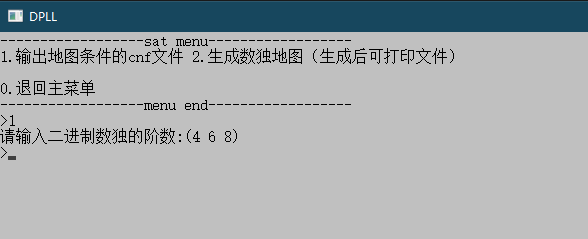


图 4- 14 输出二进制数独约束条件的cnf文件功能的交互窗口

输出的输出的4、6、8阶二进制数独的约束条件的cnf文件的部分内容如图4-15,4-16,4-17所示.

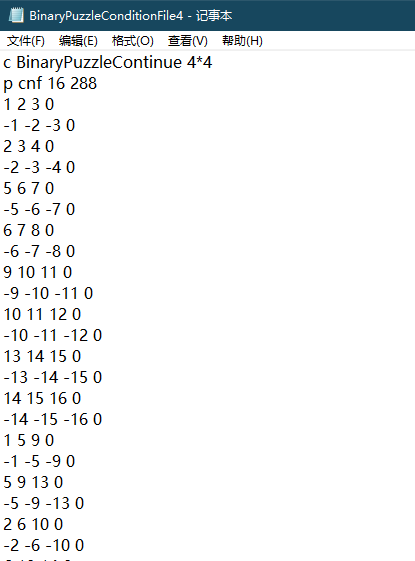


图 4- 15 4阶二进制数独约束条件生成的cnf文件

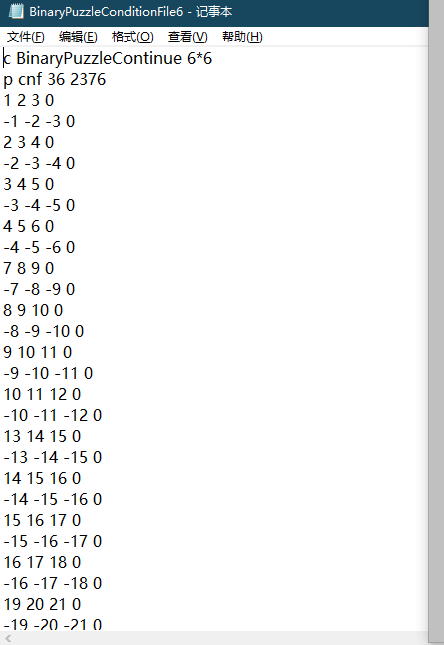


图 4- 16 6阶二进制数独约束条件生成的cnf文件

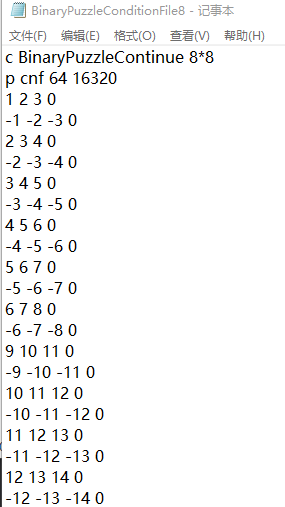


图 4- 17 8阶二进制数都约束条件生成的二进制文件

2.生成具有唯一解的二进制数独地图

此模块的测试主要为功能性测试

设计目标:

1. 基本目标，根据输入的阶数和挖洞的个数生成具有唯一解的二进制数独地图
2. 打印生成的二进制数独地图，并求解
3. 将二进制数独输出为cnf文件
4. 当配置文件不存在时，做出提示，并结束该模块，返回菜单．

程序测试：

(1)当配置文件不存在时，尝试生成数独，测试结果如图4-18所示，符合预期。

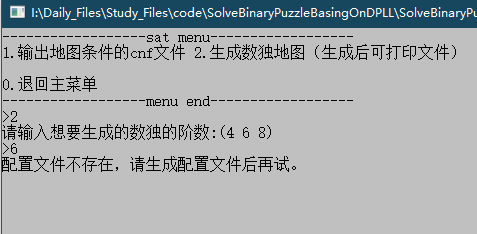


图 4- 18 程序测试-在缺少配置文件时尝试生成数独地图

(2)挖洞生成具有唯一解的二进制数独地图并输出至cnf文件,交互界面如图4-19所示,打印的数独地图如图4-20所示,生成的cnf文件的部分内容如图4-21所示

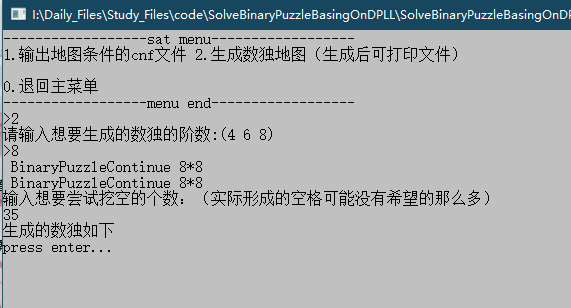


图 4- 19 生成数度地图时的交互界面

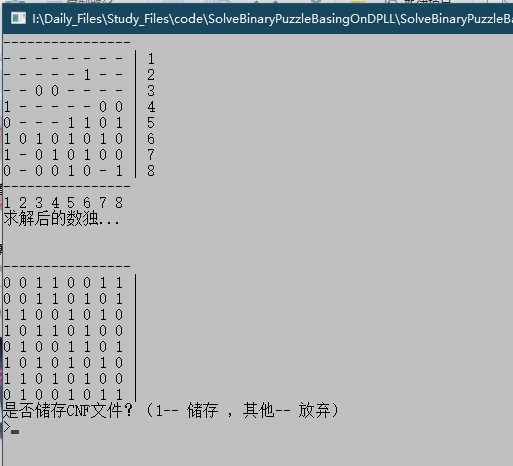


图 4- 20 系统测试-打印出生成的数独地图和其解

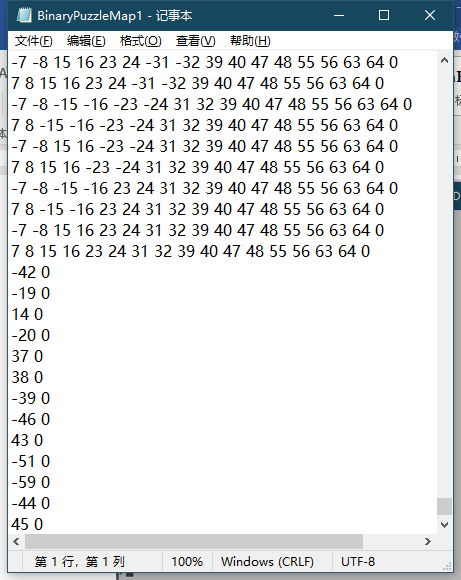


图 4- 21 生成的cnf文件

# 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.设计并实现了系统的物理储存结构，设计了栈回溯算法，为后面非递归实现DPLL算法打下基础。

3.设计并实现系统的输入输出功能，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。

4.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。

5.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。

6.对基本DPLL算法分支变元选择部分进行了优化，利用实时维护的统计数组，选用出现次数最多的变元，优先减去复杂的文字分支，减少了求解SAT问题时的化简步骤。

7.查阅资料并自己动手化简和计算相关公式，得到了4、6、8阶二进制数独约束条件的规律形式，并编写代码将其输出为cnf格式文件。

8.利用随机设定的若干初始值，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SAT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。设计算法输出二进制地图为cnf文件，以供查阅和调用。

## 5.1 工作展望

对程序进行优化时，分支策略的优化大大减少了求解过程中需要化简和回溯的步骤，但由于统计数组大量的资源耗费，最后提升的效率并不十分明显，在处理大型算例时，仍显得很吃力。在今后的学习生活中，希望能对这一系统进行改进提高程序效率。之后要做的工作如下：

1.设计更优的数据结构，使储存结构能更高效的支持DPLL算法执行过程中的化简和回溯。

2.添加文字节点的索引结构，串接相同的文字变元，用该结构替代本程序中使用的统计数组结构，降低统计变元个数的资源消耗，以弥补此次优化带来的缺陷，提高算法的执行效率。

3.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，进一步优化算法思路，以减少算法的回溯次数和搜索空间。

4.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。

5.在完成上述优化后，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。

6.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

7.对二进制数独游戏进行改写，创建可视化的交互图形界面，引入鼠标操作，提高游戏性。

8.尝试将十进制数独游戏求解转化为SAT可满足性问题，并利用DPLL算法进行求解，用多种方式生成数独地图，并包装成可玩性较高的游戏。

# 6 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的二进制数独游戏求解。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，刚开始没有进行高效的结构设计，直接开始了代码的编写，采用递归算法，程序中有着大量的内存分配和释放的语句。由于自己没有理性DPLL算法的逻辑，到导致编写代码时逻辑混乱，出现了很多访问越界的问题，最后导致free函数调用出错，程序退出。最后我放弃了原来的代码，重新认真阅读了指导书和相关文献，设计了更优的储存结构，并设计了栈结构，实现了非递归的DPLL实现。

2. 将数独游戏问题约束条件转换成CNF范式格式的时候，遇到了很大的难题，整整一天没有任何进展。最后通过尝试动手化简找到了转化的规律，并编写程序成功的实现这部分的功能，感觉受到了极大的鼓舞。

3. 由于前面吃过了没有思路却直接开始代码编写导致全盘放弃的亏，这次在编写生成数独地图的代码之前，我查阅了大量的资料，最后选择了便于实现的挖洞法生成唯一的数独地图，在笔记本上进行了算法的演算，理清了算法实现的思路之后，我才开始相关功能模块的编写。这是我意识到，在编程时，决不能在没有理清算法思路的时候贸然开始，决不能以走一步看一步的心态去编写程序。而是要有大局观，在开始编写第一行代码之前弄懂程序需求、理清程序思路、设计较好的数据结构、基本把握优化的方向之后，再进行代码的编写。这样可以大量减少调试和修改代码时需要做的工作。

4.事实上，编写程序之前，尤其是在写“数独游戏问题约束条件转换成CNF范式格式”这一模块的时候，我没有一点信心。复杂的逻辑关系搞得我我晕脑胀。但后来，我尝试着忽略程序实现的细节，使用中文大段大段的注释中写出了程序实现的基本步骤和思路，然后对实现的功能进行分块，然后分模块编写代码并进行封装。这样，我将一个看似巨大的工程，转化为了一个个小的模块，并根据封装的思路去控制代码的细节。这样，我就能在设计好了程序大框架的基础上，仅仅将目光着眼于某一个小的细节。由于已经对函数进行了封装处理，这样把每一个小模块的功能完整的实现之后，不用了担心代码块与代码块之间出现调用和转换异常的问题。

总而言之，此次试验我收获到了很多东西，除了专业知识方面，还有编程能力、编程习惯、编程思路的提升。这些收获，或许将伴随着我的一生。或许未来的某一天，回顾现在的生活，我会庆幸自己认真完成了这一次的课程设计。

# 参考文献

[1]□王静康,张凤宝,夏淑倩等.论化工本科专业国际认证与国内认证的“实质性”.高等工程教育研究,2014,5:1-4

[2]□Stone J A, Howard L P. A simple technique for observing periodic nonlinearities in Michelson interferometers. Precision Engineering,1998,22(4):220-232

[3]□朱印红,袁衍明.Dreamweaver完美网页设计——技术入门篇.(第一版).北京:中国电力出版社,2006:19～20

[4]□Lewis S L. Physics and chemistry of the solar system.北京:北京大学出版社,2014.1～2

[5]□陈剑.上博简《民之父母》“而得既塞於四海矣”句解释[EB/OL］.简帛研究网站，http://www.bamboosilk.org/Wssf/2003/chenjian03.htm．2003-01-18

# 附录

程序源代码分成4个cpp文件和一个头文件，头文件用于宏定义、结构体的定义和各个函数的声明，以供cpp文件中的函数正常调用。cpp文件有：

BinaryPuzzleMain.cpp --main函数所在的位置

DPLLMethod1.cpp --SAT求解器模块

addMethods.cpp --对程序进行优化和集成时添加的函数，作为SAT求解器的补充

SoDoKuKurt1.cpp --数独转化为SAT问题和生成数独地图的模块

程序源代码如下:

1.BinaryPuzzleMain1.cpp

#include "BinaryPuzzle1.h"

int main(void)

{

//设置窗口

HWND hwnd = GetForegroundWindow();

SetWindowTextA(hwnd, "DPLL");

int \_x = 800, \_y = 800;

SetWindowPos(hwnd, HWND\_TOPMOST, 0, 0, \_x, \_y, NULL); //设置窗口顶置

//SetWindowRgn(hwnd, CreateRectRgn(30, 30, \_x + 1300, \_y + 1300),true); //设置窗口大小

MoveWindow(hwnd, 800, 0, \_x, \_y, true);

//更改窗口颜色

system("color 70");

char filename[150] ; //文件名

int \_operate = 0;

int \_state = 0; //记录菜单界面 0--主界面 1--DPLL界面 2--数独界面

int beginTime, overTime;

pstack stack = NULL;

cnfFile cnf;

pHeadCount phc = NULL;

Status status = 0;

cnf.numOfchildSentence = 0;

cnf.numOfVariable = 0;

cnf.sentence = NULL;

pCnfFile pcnf = &cnf;

int index = 0;

int\* answer = NULL;

//读取文件创建结构

//\*\*\*\*\*\*\*\*\*\*\*\*\*

//菜单

while (1)

{

if (\_state == 0)//主菜单

{

system("CLS");

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

printf("1.SAT\t2.Binary Puzzle\n\n0.exit\n");

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

while (1)

{

status = scanf(" %d", &\_operate);

if (\_operate >= 0 && \_operate <= 2&&status == 1)

break;

else

{

printf("输入格式错误！请重新输入!\n");

getchar();

printf(">");

}

}

switch (\_operate)

{

case 1:

\_state = 1;

break;

case 2:

\_state = 2;

break;

default:

printf("欢迎使用！\n");

getchar();

getchar();

return 0;

}

}

else if (\_state == 1) //DPLL菜单

{

system("CLS");

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

printf("1.读取文件使用规则1选取分支变元进行DPLL\t2.读取文件使用规则2选取分支变元进行DPLL\n\n3.使用规则1超过20S后采用规则2的方法进行求解\n\n4.清空当前文件结构\t5.打印CNF文件\n\n0.退出\n");

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

printf(">");

while (1)

{

status = scanf(" %d", &\_operate);

if (\_operate >= 0 && \_operate <= 5 && status == 1)

break;

else

{

printf("输入格式错误！请重新输入!\n");

getchar();

printf(">");

}

}

switch (\_operate)

{

case 2:

//读文件

if (pcnf->sentence != NULL)

{

printf("当前cnf文件结构非空，请清空后再读入。\n");

getchar();

getchar();

break;

}

status = readFile\_1(pcnf,phc,filename);

if (status == OK)

printf("读取文件完成！\n所读文件共有%d个子句，涉及到%d种文字.\n", pcnf->numOfchildSentence, pcnf->numOfVariable);

else

{

printf("打开文件失败！");

getchar();

getchar();

continue;

}

if (pcnf->sentence == NULL)

{

printf("当前文件结构为空，请读入cnf文件后再试！\n");

}

printf("开始进行DPLL求解！\n");

index = 0;

beginTime = clock();

status = DPLL\_1(pcnf, stack, phc);

overTime = clock();

if (status == TRUE) {

//销毁栈并输出路径

printf("cnf能求解\n求解路径为：\n");

answer = (int\*)malloc(sizeof(int) \* (pcnf->numOfVariable));

index = 0;

destroyStack(stack, true, answer, index);

printf("\n时间：%dms", overTime - beginTime);

writeRes(filename,true, overTime - beginTime,answer,pcnf->numOfVariable);

free(answer);

answer = NULL;

}

else

{

destroyStack(stack, false, NULL, index);

printf("不能求解\n");

printf("\n时间：%dms", overTime - beginTime);

writeRes(filename, false, overTime - beginTime, NULL, pcnf->numOfVariable);

}

resetPcnf(pcnf);

resetCount(pcnf,phc);

getchar();

getchar();

break;

case 3:

//在使用第二种剪枝方法超过15s后，换用第一种方法进行求解

if (pcnf->sentence != NULL)

{

printf("当前cnf文件结构非空，请清空后再读入。\n");

getchar();

getchar();

break;

}

//读文件，考虑到使用两种方法，采用方法一的方式读取文件

status = readFile\_1(pcnf, phc, filename);

if (status == OK)

printf("读取文件完成！\n所读文件共有%d个子句，涉及到%d种文字.\n", pcnf->numOfchildSentence, pcnf->numOfVariable);

else

{

printf("打开文件失败！");

getchar();

getchar();

continue;

}

index = 0;

printf("先采用方法二进行求解\n");

beginTime = clock();

status = DPLL(pcnf, stack,beginTime,true);

overTime = clock();

if (status == ERROR)

{

//换用方法一进行求解

printf("20s stop!\n换用方法一进行求解\n");

resetPcnf(pcnf);

destroyStack(stack,false,NULL,index);

index = 0;

status = DPLL\_1(pcnf,stack,phc);

overTime = clock();

if (status == TRUE) {

//销毁栈并输出路径

printf("cnf能求解\n求解路径为：\n");

answer = (int\*)malloc(sizeof(int) \* (pcnf->numOfVariable));

index = 0;

destroyStack(stack, true, answer, index);

printf("\n时间：%dms", overTime - beginTime);

writeRes(filename, true, overTime - beginTime, answer, index);

free(answer);

answer = NULL;

}

else

{

destroyStack(stack, false, NULL, index);

printf("不能求解\n");

printf("\n时间：%dms", overTime - beginTime);

writeRes(filename, false, overTime - beginTime, NULL, index);

}

}

else if (status == TRUE) {

overTime = clock();

//销毁栈并输出路径

printf("cnf能求解\n求解路径为：\n");

answer = (int\*)malloc(sizeof(int) \* (pcnf->numOfVariable));

index = 0;

destroyStack(stack, true, answer, index);

printf("\n时间：%dms", overTime - beginTime);

writeRes(filename, true, overTime - beginTime, answer, index);

free(answer);

answer = NULL;

}

else

{

overTime = clock();

destroyStack(stack, false, NULL, index);

printf("不能求解\n");

printf("\n时间：%dms", overTime - beginTime);

writeRes(filename, false, overTime - beginTime, NULL, index);

}

getchar();

getchar();

resetPcnf(pcnf);

break;

case 4:

//销毁当前存在的结构

if (pcnf->sentence == NULL)

{

printf("当前程序的cnf文件结构为空！");

getchar();

printf("未读入cnf文件！\n");

}

else

{

//销毁

destroyCNF(pcnf);

//销毁count结构

if (phc != NULL)

{

free(phc->count);

free(phc);

phc = NULL;

}

printf("销毁成功!\n");

getchar();

getchar();

}

break;

case 1:

//DPLL方法一

//读入文件

//读文件

if (pcnf->sentence != NULL)

{

printf("当前cnf文件结构非空，请清空后再读入。\n");

getchar();

getchar();

break;

}

status = readFile(pcnf, filename);

if (status == OK)

printf("读取文件完成！\n所读文件共有%d个子句，涉及到%d种文字.\n", pcnf->numOfchildSentence, pcnf->numOfVariable);

else

{

printf("打开文件失败！");

getchar();

getchar();

continue;

}

if (pcnf->sentence == NULL)

{

printf("当前文件结构为空，请读入cnf文件后再试！\n");

}

printf("开始进行DPLL求解！\n");

index = 0;

beginTime = clock();

status = DPLL(pcnf, stack,0,false);

overTime = clock();

if (status == TRUE) {

//销毁栈并输出路径

printf("cnf能求解\n求解路径为：\n");

answer = (int\*)malloc(sizeof(int) \* (pcnf->numOfVariable));

index = 0;

destroyStack(stack, true, answer, index);

printf("\n时间：%dms", overTime - beginTime);

writeRes(filename, true, overTime - beginTime, answer, index);

free(answer);

answer = NULL;

}

else

{

destroyStack(stack, false, NULL, index);

printf("不能求解\n");

printf("\n时间：%dms", overTime - beginTime);

writeRes(filename, false, overTime - beginTime, NULL, index);

}

resetPcnf(pcnf);

getchar();

getchar();

break;

case 5:

//打印CNF文件

if (pcnf == NULL)

{

printf("当前CNF结构为空，不能输出CNF文件!\n");

}

else

{

writeSoDoKufile(pcnf);

printf("文件写入成功！\n");

}

getchar();

getchar();

break;

default:

//退出

printf("退回主菜单，同时清空当前CNF结构！\n");

if (pcnf != NULL)

destroyCNF(pcnf);

getchar();

getchar();

\_state = 0;

}

}

else if (\_state == 2) //BinaryPuzzle菜单

{

system("CLS");

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

printf("1.输出地图条件的cnf文件\t2.生成数独地图（生成后可打印文件）\n\n0.退回主菜单\n");

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

printf(">");

while (1)

{

status = scanf(" %d", &\_operate);

if (\_operate >= 0 && \_operate <= 2 && status == 1)

break;

else

{

printf("输入格式错误！请重新输入!\n");

getchar();

printf(">");

}

}

switch (\_operate)

{

case 2:

printf("请输入想要生成的数独的阶数:(4 6 8)\n>");

scanf("%d", &\_operate);

if (!(\_operate == 4 || \_operate == 6 || \_operate == 8))

{

printf("输入错误！");

getchar();

getchar();

}

else

{

generateSoDuKuMap(pcnf, \_operate);

}

break;

case 1: //打印配置文件

printConfition();

getchar();

getchar();

break;

case 0:

\_state = 0;

printf("退回主菜单");

getchar();

}

}

}

}

2.DPLLMethod1.cpp

#include "BinaryPuzzle1.h"

//初始化栈

void initStack(pstack &sta) {

if (sta != NULL) return; //栈非空，不能初始化

sta = (pstack)malloc(sizeof(stack));

sta->tail = (stackNode\*)malloc(sizeof(stackNode)); //设置栈底

sta->top = sta->tail;

sta->tail->nextNode = NULL;

}

//入栈

void addStack(pstack sta, int word, bool methodMark)

{

pStackNode psn = (pStackNode)malloc(sizeof(stackNode));

if (psn == NULL) exit(-4);

psn->methodMark = methodMark;

psn->setTrueWord = word;

psn->nextNode = sta->top;

sta->top = psn;

}

//出栈,返回两个值，word&methodMark,栈为空时返回ERROR，否则返回OK

Status outStack(pstack sta, int &word, bool &methodMark)

{

if (sta->tail == sta->top)

return ERROR;

pStackNode psn = sta->top;

methodMark = psn->methodMark;

word = psn->setTrueWord;

sta->top = sta->top->nextNode;

free(psn);

return OK;

}

//检测空栈

bool cheakStack(pstack sta)

{

if (sta->tail == sta->top) return true;

return false;

}

//根据栈元素进行回溯

Status recovery(pstack sta, pCnfFile pcnf)

{

//获取栈中的元素，并将其回溯为上一次的结构

int word = 0;

bool methodMark = false;

Status status;

pChildSentence pcs = NULL;

pWord pw = NULL;

while (1)//回溯直到找到分支为止

{

status = outStack(sta, word, methodMark);

if (status == ERROR) //空栈

return ERROR;

if (word == 0) exit(-6);

//回溯

pcs = pcnf->sentence;

while (pcs != NULL)

{

//重建因为word而被删除掉的子句（将mark置为0）

if (pcs->mark == word)

{

pcs->mark = 0;

pcnf->numOfchildSentence++;

}

//重建因为word而被删掉的-word文字

else if (pcs->mark == 0)

{

pw = pcs->words;

while (pw != NULL)

{

if (pw->mark == false && pw->varibale + word == 0)

{

pw->mark = true;

pcs->numOfword++;

}

pw = pw->nextword;

}

}

pcs = pcs->nextSen;

}

if (methodMark == true)

break;

}

//假设-word为真，利用规则化简集合，此处入栈时当作单子句规则处理，之后回溯到这一步就不用再次遇见分支

//入栈

addStack(sta, (-1)\*word, false);

simplyS(pcnf, (-1)\*word);

return OK;

}

//化简规则，单子句中的化简/分裂规则中的化简

Status simplyS(pCnfFile pcnf, int word)

{

pChildSentence pcs = pcnf->sentence;

pWord pw = NULL;

//删除含有word的子句

while (pcs != NULL)

{

if (pcs->mark == 0 && (checkWord(pcs, word)))

{

pcs->mark = word;

pcnf->numOfchildSentence--;

}

pcs = pcs->nextSen;

}

//分裂规则,删除含有-word的文字

pcs = pcnf->sentence;

while (pcs != NULL)

{

if (pcs->mark == 0)

{

//遍历该子句，查找word

pw = pcs->words;

while (pw != NULL)

{

if (pw->varibale + word == 0 && pw->mark == true)

{

//删除该文字

pw->mark = false;

pcs->numOfword--;

}

pw = pw->nextword;

}

}

pcs = pcs->nextSen;

}

return OK;

}

//检测子句中是否含有目标文字

bool checkWord(pChildSentence pcs, int word)

{

pWord pw = pcs->words;

while (pw != NULL)

{

if (pw->varibale == word && pw->mark == true)

return true;

pw = pw->nextword;

}

return false;

}

//选择分裂规则中使用的文字，选择结构中同时出现原文字、非文字的文字

int seclect(pCnfFile pcnf, int \*count, bool &mark)

{

//定义数组

//遍历数组，统计出现的各文字出现的次数

//初始化

for (int i = 0; i <= pcnf->numOfVariable; i++)

count[i] = 0;

pChildSentence pcs = NULL;

pWord pw = NULL;

pcs = pcnf->sentence;

mark = true;

while (pcs != NULL)

{

if (pcs->isCunzai != true || pcs->mark != 0)

{

pcs = pcs->nextSen;

continue;

}

pw = pcs->words;

while (pw != NULL)

{

if (pw->mark == true)

{

if (pw->varibale > 0)

{

if (count[pw->varibale] < 0)

{

return pw->varibale;

}

else

count[pw->varibale]++;

}

else

{

if (count[-1 \* pw->varibale] > 0)

{

return pw->varibale;

}

else

count[-1 \* pw->varibale]--;

}

}

pw = pw->nextword;

}

pcs = pcs->nextSen;

}

//未找到同时出现正负的文字，返回数组中首个出现过的数

mark = false;

for (int i = 1; i <= pcnf->numOfVariable; i++)

{

if (count[i] < 0)

{

return (-1)\*i;

}

else if (count[i] > 0)

{

return i;

}

}

for (int i = 1; i <= pcnf->numOfVariable; i++) {

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

}

exit(-99);

}

//寻找单子句，成功找到则返回单子句的文字，否则返回0

int findSingal(pCnfFile pcnf)

{

pChildSentence pcs = pcnf->sentence;

pWord pw = NULL;

while (pcs != NULL)

{

if (pcs->mark == 0 && pcs->numOfword == 1)

{

pw = pcs->words;

while (pw != NULL)

{

if (pw->mark == true)

return pw->varibale;

pw = pw->nextword;

}

}

pcs = pcs->nextSen;

}

//未找到单子句

return 0;

}

//检测S中是否存在空子句

bool checkNull(pCnfFile pcnf)

{

pChildSentence pcs = pcnf->sentence;

while (pcs != NULL)

{

if (pcs->mark == 0 && pcs->numOfword == 0)

return true;

pcs = pcs->nextSen;

}

return false;

}

Status DPLL(pCnfFile pcnf, pstack &stack,int beginTime,bool change)

{

int cur\_time;

int \* count;

count = (int\*)malloc(sizeof(int)\*(1 + pcnf->numOfVariable));

pstack sta = NULL;

initStack(sta);

stack = sta;

bool mark = false;

bool status;

if (sta == NULL) exit(-1);

int cur\_word = 0;

do {

while (1)

{

if (pcnf->numOfchildSentence == 0) //集合为空

{

free(count);

return TRUE;

}

status = false;

if ((status = checkNull(pcnf)) == true) //S中存在空子句，开始回溯

break;

cur\_word = findSingal(pcnf);

if (cur\_word == 0) //无单子句

break;

//化简单子句

addStack(sta, cur\_word, false);

simplyS(pcnf, cur\_word);

}

if (change == true)

{

cur\_time = clock();

if (cur\_time - beginTime >= 20000)//20s

{

free(count);

return ERROR;

}

}

if (status == true) //存在空子句，回溯

{

if (recovery(sta, pcnf) == ERROR)

{

free(count);

return FALSE;

}

}

else {

cur\_word = seclect(pcnf, count, mark);

if (cur\_word == 0) exit(-10);

addStack(sta, cur\_word, mark);

simplyS(pcnf, cur\_word);

}

} while (cheakStack(sta) == FALSE);

free(count);

return FALSE;

}

//文件读取函数

Status readFile(pCnfFile pcnf,char \*pFilename)

{

char filename[100], \*ch, cch;

char addrFilename[150] = ADDR;

printf("输入文件名称：");

scanf("%s", filename);

//拼接地址+文件名

int i = 0;

while (addrFilename[i] != '\0') i++;

for (int j = 0; filename[j] != '\0'; j++)

addrFilename[i++] = filename[j];

addrFilename[i] = '\0';

strcpy(pFilename,addrFilename);

FILE\* file;

//打开文件

if (!(file = fopen(addrFilename, "r")))

{

return ERROR;

}

readAndCreate(pcnf, file);

fclose(file);

//创建结点完成

//pcnf文件完成创建

return OK;

}

//读文件创建结构

Status readAndCreate(pCnfFile pcnf, FILE\*file)

{

char buffer[100]; //字符串缓冲区

int index = 0; //字符索引指针

char cch, \*ch;

//按行读取文件

while (1)

{

cch = fgetc(file);

if (cch == 'c') //注释行,打印到窗口中

{

fgets(buffer, 100, file);

//打印到窗口中

printf("%s", buffer);

}

else if (cch == 'p') //信息行，停止打印

{

fgets(buffer, 6, file);//跳过前缀读取

break;

}

}

fscanf(file, "%d %d", &(pcnf->numOfVariable), &(pcnf->numOfchildSentence));

//读取文件并创建子句

int cur\_variable = 1;

//读取首个子句

pChildSentence pcs = NULL;

pWord pw = NULL;

//读取首个变元

fscanf(file, "%d ", &cur\_variable);

//创建首个子句储存空间

pcnf->sentence = (pChildSentence)malloc(sizeof(childSentence));

//初始化

pcs = pcnf->sentence;

pcs->isCunzai = true;

pcs->mark = 0;

pcs->nextSen = NULL;

pcs->numOfword = 0;

pcs->words = NULL;

if (cur\_variable != 0)

{

//创建首个子句的变元文字

pcs->words = (pWord)malloc(sizeof(word));

//初始化和赋值

pw = pcs->words;

pw->varibale = cur\_variable;

pw->nextword = NULL;

pw->mark = true;

pcs->numOfword++;

//继续读取，创建文字

while (1)

{

fscanf\_s(file, "%d ", &cur\_variable);

if (cur\_variable == 0)

break;

pw->nextword = (pWord)malloc(sizeof(word));

//赋值

pw = pw->nextword;

pw->nextword = NULL;

pw->mark = true;

pw->varibale = cur\_variable;

pcs->numOfword++;

}

}

while (fscanf(file, "%d ", &cur\_variable) != EOF)

{

pcs->nextSen = (pChildSentence)malloc(sizeof(childSentence));

//初始化

pcs = pcs->nextSen;

pcs->isCunzai = true;

pcs->nextSen = NULL;

pcs->mark = 0;

pcs->numOfword = 0;

pcs->words = NULL;

//第一个元素就是0，空子句

if (cur\_variable == 0) continue;

//创建首个文字结点

pcs->words = (pWord)malloc(sizeof(word));

//赋值、初始化

pw = pcs->words;

pw->varibale = cur\_variable;

pw->mark = true;

pw->nextword = NULL;

pcs->numOfword++;

//读取后面的文字并创建结点

while (1)

{

fscanf\_s(file, "%d ", &cur\_variable);

if (cur\_variable == 0)

break;

pw->nextword = (pWord)malloc(sizeof(word));

//赋值

pw = pw->nextword;

pw->nextword = NULL;

pw->varibale = cur\_variable;

pw->mark = true;

pcs->numOfword++;

}

}

return OK;

}

//销毁栈并递归输出路径

void destroyStack(pstack sta, bool isSuccess, int \* answer, int &index)

{

//成功时输出路径，未成功时不输出

int count = 0;

pStackNode psn = sta->top;

destroy(psn, sta, count, isSuccess, answer, index);

return;

}

void destroy(pStackNode psn, pstack sta, int &i, bool isSuccess, int \*answer, int &index)

{

int word = 0;

if (psn == sta->tail)

free(psn);

else

{

word = psn->setTrueWord;

destroy(psn->nextNode, sta, i, isSuccess, answer, index);

if (answer != NULL)

answer[index++] = word;

if (isSuccess == true)

{

printf("%d\t", word);

i++;

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

}

}

return;

}

//销毁当前的cnf结构

void destroyCNF(pCnfFile pcnf)

{

//销毁子句

pChildSentence pcs = pcnf->sentence;

pChildSentence pcs\_l = pcs;

/\*while (pcs!=NULL)

{

pcs\_l = pcs->nextSen;

destroyChildScen(pcs);

pcs = pcs\_l;

}\*/ //意料之外的bug

pcnf->numOfchildSentence = 0;

pcnf->numOfVariable = 0;

pcnf->sentence = NULL;

return;

}

//销毁子句

void destroyChildScen(pChildSentence pcs)

{

pWord pw = pcs->words;

pWord pw1;

while (1)

{

if (pw == NULL)

break;

pw1 = pw->nextword;

free(pw);

pw = pw1;

}

free(pcs);

return;

}

//test

//完成求解后，重置pcnf的mask

void resetPcnf(pCnfFile pcnf)

{

pChildSentence pcs = pcnf->sentence;

pWord pw = NULL;

while (pcs != NULL)

{

if (pcs->mark != 0 && pcs->isCunzai == true)

{

pcs->mark = 0;

pcnf->numOfchildSentence++;

}

pw = pcs->words;

while (pw != NULL)

{

if (pw->mark == false)

{

pw->mark = true;

pcs->numOfword++;

}

pw = pw->nextword;

}

pcs = pcs->nextSen;

}

}

//输出res文件

Status writeRes(char\* fileName,bool isSuccess , int sumTime,int \*answer,int n)

{

//将文件名后缀改成res

int i = 0;

printf("\n是否输出res文件？(1-是，其他-否)\n");

scanf(" %d", &i);

if (i != 1) return OK;

i = 0;

while (fileName[i] != '\0') i++;//找到结尾

fileName[i - 3] = '\0';

strcat(fileName, "res");//改变文件名完毕

FILE\* fp = NULL;

fp = fopen(fileName,"w");//打开文件

if (fp == NULL) exit(-1);

if (isSuccess == true)

fprintf(fp, "s 1\n");

else

fprintf(fp, "s 0\n");

printf("文件写入成功！\n");

fprintf(fp, "v");

if (isSuccess == true)

{

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

{

fprintf(fp," %d", answer[i]);

if ((i + 1) % 10 == 0)

fprintf(fp,"\n ");

}

}

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

fclose(fp);

return OK;

}

3.addMethods.cpp

#include"BinaryPuzzle1.h"

/\*\*

\*改变储存结构实现DPLL，提高效率改进方法中改进的函数

\*/

//在count结构中添加变量variable

void addToCount(pHeadCount phc, int variable)

{

int cur\_vari = 0;

if (variable > 0)

{

cur\_vari = variable;

phc->count[variable].posi++;

}

else if (variable < 0)

{

cur\_vari = (-1)\*variable;

phc->count[-variable].nega++;

}

else {

printf("addToCount error!");

getchar();

getchar();

exit(-1);

}

//更改当前结构中的最多的变量

if (phc->count[cur\_vari].posi + phc->count[cur\_vari].nega > phc->sum)

{

phc->max\_variable = cur\_vari;

phc->sum = phc->count[cur\_vari].posi + phc->count[cur\_vari].nega;

if (phc->count[cur\_vari].posi > 0)

phc->containPo = true;

else

phc->containPo = false;

if (phc->count[cur\_vari].nega > 0)

phc->cintainNe = true;

else

phc->cintainNe = false;

}

}

//从count结构中删除variable

void delFormCount(pHeadCount phc, int variable, pCnfFile pcnf)

{

//从结构中删除一个

int cur\_vari = 0;

if (variable > 0)

{

cur\_vari = variable;

phc->count[variable].posi--;

}

else if (variable < 0)

{

cur\_vari = (-1)\*variable;

phc->count[cur\_vari].nega--;

}

else {

printf("delFromCount error!");

getchar();

getchar();

exit(-1);

}

//更新当前结构中的最多的变量

if (phc->max\_variable == cur\_vari)

{

phc->sum--;

if (phc->count[cur\_vari].nega == 0)

phc->cintainNe = false;

if (phc->count[cur\_vari].posi == 0)

phc->containPo = false;

for (int i = 1; i <= pcnf->numOfVariable; i++)

{

//遍历，判断是否存在超过sum的变量存在

if (phc->count[i].nega + phc->count[i].posi > phc->sum)

{

phc->sum = phc->count[i].nega + phc->count[i].posi > phc->sum;

phc->max\_variable = i;

if (phc->count[i].nega == 0)

phc->cintainNe = false;

else phc->cintainNe = true;

if (phc->count[i].posi == 0)

phc->containPo = false;

else phc->containPo = true;

}

}

}

}

//文件读取函数改进--添加统计count的操作入口

Status readFile\_1(pCnfFile pcnf,pHeadCount &\_phc,char\*pFilename)

{

char filename[200], \*ch, cch;

char addrFilename[100] = ADDR;

printf("输入文件名称：");

scanf("%s", filename);

//拼接地址+文件名

int i = 0;

while (addrFilename[i] != '\0') i++;

for (int j = 0; filename[j] != '\0'; j++)

addrFilename[i++] = filename[j];

addrFilename[i] = '\0';

strcpy(pFilename, addrFilename);

FILE\* file;

//打开文件

if (!(file = fopen(addrFilename, "r")))

{

return ERROR;

}

readAndCreate\_1(pcnf, file,\_phc);

fclose(file);

//创建结点完成

//pcnf文件完成创建

return OK;

}

//读文件创建结构

Status readAndCreate\_1(pCnfFile pcnf, FILE\*file,pHeadCount &\_phc)

{

char buffer[100]; //字符串缓冲区

int index = 0; //字符索引指针

char cch, \*ch;

pHeadCount phc = (pHeadCount)malloc(sizeof(headCount));

//初始化

phc->cintainNe = false;

phc->containPo = false;

phc->max\_variable = 0;

phc->sum = 0;

\_phc = phc;

printf("文件注释如下：\n");

//按行读取文件

while (1)

{

cch = fgetc(file);

if (cch == 'c') //注释行,打印到窗口中

{

fgets(buffer,100,file);

//打印到窗口中

printf("%s", buffer);

}

else if (cch == 'p') //信息行，停止打印

{

fgets(buffer, 6, file);//跳过前缀读取

break;

}

}

fscanf(file, "%d %d", &(pcnf->numOfVariable), &(pcnf->numOfchildSentence));

//创建数组结构

phc->count = (pCount)malloc(sizeof(count)\*(pcnf->numOfVariable + 1));

//初始化

for (int i = 0; i <= pcnf->numOfVariable; i++)

{

phc->count[i].nega = 0;

phc->count[i].posi = 0;

}

//读取文件并创建子句

int cur\_variable = 1;

//读取首个子句

pChildSentence pcs = NULL;

pWord pw = NULL;

//读取首个变元

fscanf(file, "%d ", &cur\_variable);

//创建首个子句储存空间

pcnf->sentence = (pChildSentence)malloc(sizeof(childSentence));

//初始化

pcs = pcnf->sentence;

pcs->isCunzai = true;

pcs->mark = 0;

pcs->nextSen = NULL;

pcs->numOfword = 0;

pcs->words = NULL;

if (cur\_variable != 0)

{

//创建首个子句的变元文字

pcs->words = (pWord)malloc(sizeof(word));

//初始化和赋值

pw = pcs->words;

pw->varibale = cur\_variable;

pw->nextword = NULL;

pw->mark = true;

pcs->numOfword++;

//储存到count中去

addToCount(phc,cur\_variable);

//继续读取，创建文字

while (1)

{

fscanf\_s(file, "%d ", &cur\_variable);

if (cur\_variable == 0)

break;

pw->nextword = (pWord)malloc(sizeof(word));

//赋值

pw = pw->nextword;

pw->nextword = NULL;

pw->mark = true;

pw->varibale = cur\_variable;

pcs->numOfword++;

//储存到count中去

addToCount(phc, cur\_variable);

}

}

while (fscanf(file, "%d ", &cur\_variable) != EOF)

{

pcs->nextSen = (pChildSentence)malloc(sizeof(childSentence));

//初始化

pcs = pcs->nextSen;

pcs->isCunzai = true;

pcs->nextSen = NULL;

pcs->mark = 0;

pcs->numOfword = 0;

pcs->words = NULL;

//第一个元素就是0，空子句

if (cur\_variable == 0) continue;

//创建首个文字结点

pcs->words = (pWord)malloc(sizeof(word));

//赋值、初始化

pw = pcs->words;

pw->varibale = cur\_variable;

pw->mark = true;

pw->nextword = NULL;

pcs->numOfword++;

//储存到count中去

addToCount(phc, cur\_variable);

//读取后面的文字并创建结点

while (1)

{

fscanf\_s(file, "%d ", &cur\_variable);

if (cur\_variable == 0)

break;

pw->nextword = (pWord)malloc(sizeof(word));

//赋值

pw = pw->nextword;

pw->nextword = NULL;

pw->varibale = cur\_variable;

pw->mark = true;

pcs->numOfword++;

//储存到count中去

addToCount(phc, cur\_variable);

}

}

return OK;

}

//DPLL改写

Status DPLL\_1(pCnfFile pcnf, pstack &stack,pHeadCount phc)

{

int cur\_time;

pstack sta = NULL;

initStack(sta);

stack = sta;

bool mark = false;

bool status;

if (sta == NULL) exit(-1);

int cur\_word = 0;

do {

while (1)

{

if (pcnf->numOfchildSentence == 0) //集合为空

{

return TRUE;

}

status = false;

if ((status = checkNull(pcnf)) == true) //S中存在空子句，开始回溯

break;

cur\_word = findSingal(pcnf);

if (cur\_word == 0) //无单子句

break;

//化简单子句

addStack(sta, cur\_word, false);

simplyS\_1(pcnf, cur\_word, phc);

}

if (status == true) //存在空子句，回溯

{

if (recovery\_1(sta, pcnf,phc) == ERROR)

{

return FALSE;

}

}

else {

//选取变元，进行分裂操作

cur\_word = select\_1(pcnf, phc, mark);

if (cur\_word == 0) exit(-2);

addStack(sta, cur\_word, mark);

simplyS\_1(pcnf, cur\_word,phc);

}

} while (cheakStack(sta) == FALSE);

return FALSE;

}

//化简单子句函数改写

Status simplyS\_1(pCnfFile pcnf, int word,pHeadCount phc)

{

pChildSentence pcs = pcnf->sentence;

pWord pw = NULL;

//删除含有word的子句

while (pcs != NULL)

{

if (pcs->mark == 0 && (checkWord(pcs, word)))

{

pcs->mark = word;

pcnf->numOfchildSentence--;

pw = pcs->words;

while (pw)

{

//从count结构中删除一个记录

if (pw->mark == true)

{

delFormCount(phc,pw->varibale,pcnf);

}

pw = pw->nextword;

}

}

pcs = pcs->nextSen;

}

//分裂规则,删除含有-word的文字

pcs = pcnf->sentence;

while (pcs != NULL)

{

if (pcs->mark == 0)

{

//遍历该子句，查找word

pw = pcs->words;

while (pw != NULL)

{

if (pw->varibale + word == 0 && pw->mark == true)

{

//删除该文字

pw->mark = false;

pcs->numOfword--;

//从count结构中删除一个记录

delFormCount(phc, pw->varibale, pcnf);

}

pw = pw->nextword;

}

}

pcs = pcs->nextSen;

}

return OK;

}

//选取变元改写

int select\_1(pCnfFile pcnf, pHeadCount phc, bool &mark)

{

int res = 0;

//得到结构中数量最多的文字

if (phc->containPo == true && phc->cintainNe == true)

{

mark = true;

res = phc->max\_variable;

}

else if (phc->cintainNe == true && phc->containPo == false)

{

mark = false;

res = (-1)\*phc->max\_variable;

}

else if (phc->containPo == true && phc->cintainNe == false)

{

mark = false;

res = phc->max\_variable;

}

else

{

printf("select\_1 error!");

getchar();

getchar();

exit(-1);

}

return res;

}

//recovery函数改写

//根据栈元素进行回溯

Status recovery\_1(pstack sta, pCnfFile pcnf,pHeadCount phc)

{

//获取栈中的元素，并将其回溯为上一次的结构

int word = 0;

bool methodMark = false;

Status status;

pChildSentence pcs = NULL;

pWord pw = NULL;

while (1)//回溯直到找到分支为止

{

status = outStack(sta, word, methodMark);

if (status == ERROR) //空栈

return ERROR;

if (word == 0) exit(-6);

//回溯

pcs = pcnf->sentence;

while (pcs != NULL)

{

//重建因为word而被删除掉的子句（将mark置为0）

if (pcs->mark == word)

{

pcs->mark = 0;

pcnf->numOfchildSentence++;

pw = pcs->words;

while (pw)

{

if (pw->mark == true)

{

addToCount(phc,pw->varibale);

}

pw = pw->nextword;

}

}

//重建因为word而被删掉的-word文字

else if (pcs->mark == 0)

{

pw = pcs->words;

while (pw != NULL)

{

if (pw->mark == false && pw->varibale + word == 0)

{

pw->mark = true;

pcs->numOfword++;

addToCount(phc,pw->varibale);

}

pw = pw->nextword;

}

}

pcs = pcs->nextSen;

}

if (methodMark == true)

break;

}

//假设-word为真，利用规则化简集合，此处入栈时当作单子句规则处理，之后回溯到这一步就不用再次遇见分支

//入栈

addStack(sta, (-1)\*word, false);

simplyS\_1(pcnf, (-1)\*word, phc);

return OK;

}

//count数据恢复和重置pcnf标记

void resetCount(pCnfFile pcnf, pHeadCount phc)

{

//初始化

phc->cintainNe = false;

phc->containPo = false;

phc->max\_variable = 0;

phc->sum = 0;

for (int i = 1; i <= pcnf->numOfVariable; i++)

{

phc->count[i].nega = 0;

phc->count[i].posi = 0;

}

pChildSentence pcs = pcnf->sentence;

pWord pw = NULL;

while (pcs != NULL)

{

pw = pcs->words;

while (pw != NULL)

{

addToCount(phc,pw->varibale);

pw = pw->nextword;

}

pcs = pcs->nextSen;

}

}

4.SoDoKuKurt1.cpp

#include "BinaryPuzzle1.h"

//生成数读地图

//使用挖洞法生成数独地图

//首先随机在数独中填入一些空格，对数独进行求解，若有解，以求解完毕的数独为基础，使用挖洞法生成数独，若无解，则重新填入

//4阶数组--随机生成2个位置的数

//6阶数组--随机生成3个位置的数

//8阶数组--随机生成4个位置的数

void generateSoDuKuMap(pCnfFile pcnf, int n) //n表示阶数

{

if (pcnf->sentence != NULL)

{

printf("当前pcnf文件结构非空，请清空后再试！\n");

getchar();

getchar();

return;

}

bool\* map = (bool\*)malloc(sizeof(bool)\*n\*n); //储存当前数独地图

int numOfBlank = 0; //生成随机个数

int location = 0; //位置

//随机得到将要填入的个数

if (n == 4)

{

numOfBlank = 2;

}

else if (n == 6)

{

numOfBlank = 3;

}

else if (n == 8)

{

numOfBlank = 4;

}

else exit(-1); //不符合操作

//用pcnf读取数独配置文件，配置文件不存在时提醒创建配置文件

char addrFilename[200] = ADDR;

strcat(addrFilename, "BinaryPuzzleConditionFile");

char ch[2] = "4";

ch[0] = n + '0';

strcat(addrFilename, ch);

FILE\* file;

pstack sta = NULL;

int \*saveAnwser = (int \*)malloc(sizeof(int)\*(n\*n));

int num; //用于接收随机数，判断随机的数的正负

int index = 0;

//

while (1)

{

index = 0;

//初始化

for (int i = 0; i < n\*n; i++)

{

saveAnwser[i] = 0;

}

file = fopen(addrFilename, "r");

if (file == NULL)//打开文件失败

{

printf("配置文件不存在，请生成配置文件后再试。\n");

getchar();

getchar();

return;

}

//读取文件

readAndCreate(pcnf, file);

fclose(file);

//在随机位置填入随机数

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

{

//生成随机位置

location = rand() % (n\*n) + 1;

//将随机位置存入到pcnf中去

num = rand() % 2;

if (num == 1) location = (-1)\*location;

addNodeToPcnf(pcnf, location);

}

if (DPLL(pcnf, sta,0,false) == TRUE) break; //成功生成解

//销毁pcnf结构，重新读取

destroyCNF(pcnf);

destroyStack(sta, false, NULL, index);

sta = NULL;

pcnf = NULL;

}

//将结果储存到数组中去

destroyStack(sta, false, saveAnwser, index);

//重新生成pcnf结构

destroyCNF(pcnf);

file = fopen(addrFilename, "r");

if (file == NULL)//打开文件失败

{

printf("配置文件不存在，请生成配置文件后再试。\n");

getchar();

getchar();

return;

}

//读取文件

readAndCreate(pcnf, file);

fclose(file);

//将求得的解存入cnf

for (int i = 0; i < pcnf->numOfVariable; i++)

{

if (saveAnwser[i] == 0) exit(-1);

addNodeToPcnf(pcnf, saveAnwser[i]);

}

//复制当前数组作为挖洞的依据

int \*digHole = (int\*)malloc(sizeof(int)\*(n\*n));

for (int i = 0; i < (n\*n); i++)

{

digHole[i] = saveAnwser[i];

}

printf("输入想要尝试挖空的个数：（实际形成的空格可能没有希望的那么多）\n");

scanf("%d", &numOfBlank);

digMap(pcnf, digHole, numOfBlank, n);

//生成成功

//打印数独地图

printMap(digHole, saveAnwser, n);

printf("是否储存CNF文件？（1-- 储存 ，其他-- 放弃）\n>");

int op = -1;

scanf("%d", &op);

if (op == 1)

{

writeSoDoKufile(pcnf);

}

//清空当前cnf文件结构

destroyCNF(pcnf);

return;

}

//写入文件

Status writeSoDoKufile(pCnfFile pcnf)

{

char saveName[30];

char addr[100] = ADDR;

printf("请输入文件名：\n>");

scanf("%s", saveName);

FILE \*file = fopen(strcat(addr, saveName), "w");

if (file == NULL)

{

printf("打开文件失败！");

getchar();

getchar();

exit(-1);

}

fprintf(file, "p cnf %d %d\n", pcnf->numOfVariable, pcnf->numOfchildSentence);

pChildSentence pcs = pcnf->sentence;

pWord pw = NULL;

while (pcs != NULL)

{

if (pcs->isCunzai == true)

{

pw = pcs->words;

while (pw != NULL)

{

if (pw->mark == true)

fprintf(file, "%d ", pw->varibale);

pw = pw->nextword;

}

fprintf(file, "0\n");

}

pcs = pcs->nextSen;

}

fclose(file);

return OK;

}

void printMap(int \*digHole, int \*answer, int n)

{

int \*map = (int\*)malloc(sizeof(int)\*(n\*n + 1));

//-1表示空格

//初始化

//for (int i = 0; i < n\*n; i++)

//{

// printf("%d ", digHole[i]);

//}

//printf("\n");

//for (int i = 0; i < n\*n; i++)

//{

// printf("%d ", answer[i]);

//}

//printf("\n");

for (int i = 0; i < n\*n + 1; i++)

{

map[i] = -1;

}

//打印未求解的map

for (int i = 0; i < n\*n; i++)

{

//复制数组

if (digHole[i] != 0)

{

if (digHole[i] > 0)

{

if (map[digHole[i]] != -1)

{

printf("ERROR!");

exit(-1);

}

map[digHole[i]] = 1;

}

else

{

if (map[-digHole[i]] != -1)

{

printf("ERROR!");

exit(-1);

}

map[-digHole[i]] = 0;

}

}

}

printf("生成的数独如下\n");

printf("press enter...\n");

getchar();

getchar();

system("cls");

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

printf("--");

printf("\n");

for (int i = 1; i <= n \* n; i++)

{

if (map[i] == 1)

printf("1 ");

else if (map[i] == 0)

printf("0 ");

else if (map[i] == -1)

printf("- ");

else

{

printf("error!");

getchar();

exit(-1);

}

if ((i % n) == 0) printf("|%2d\n", i / n);

}

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

printf("--");

printf("\n");

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

printf("%-2d", i + 1);

printf("\n");

printf("求解后的数独...\n");

getchar();

getchar();

//打印求解的后的数独

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

printf("--");

printf("\n");

//初始化

for (int i = 0; i < n\*n + 1; i++)

{

map[i] = -1;

}

/\*for (int i = 0; i < n\*n + 1; i++)

{

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

}

printf("\n");\*/

for (int i = 0; i < n\*n; i++)

{

//复制数组

if (answer[i] == 0)

{

printf("error!");

getchar();

exit(-1);

}

if (answer[i] < 0)

{

if (map[-answer[i]] != -1)

{

printf("error!");

getchar();

exit(-1);

}

map[-answer[i]] = 0;

}

else

{

if (map[answer[i]] != -1)

{

printf("error!");

getchar();

exit(-1);

}

map[answer[i]] = 1;

}

}

for (int i = 1; i <= n \* n; i++)

{

if (map[i] == 1)

printf("1 ");

else if (map[i] == 0)

printf("0 ");

else

{

printf("error!");

getchar();

exit(-1);

}

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

}

}

//挖洞生成数独地图,传入生成的空的个数,n表示阶数

void digMap(pCnfFile pcnf, int \*digHole, int numOfHole, int n)

{

int num = 1;

int i = 1;

Status status;

//从首个位置开始，将其更改，查看是否有解

while (i <= numOfHole)

{

num = digHole[i];

status = changeAndDPLL(pcnf, num, n);

if (status == FALSE) //DPLL失败，可删除该结点

{

//在数组digHole中将该结点的值删除

for (int i = 0; i < n\*n; i++)

{

if (digHole[i] == num || digHole[i] == (-1)\*num)

{

digHole[i] = 0;

break;

}

}

}

i++;

if (i >= n \* n)

break; //达到了最大位置处,不能删掉num这么多个点

}

}

//将编号为num的位置设为其相反数，查看是否有解,n为阶数

Status changeAndDPLL(pCnfFile pcnf, int num, int n)

{

Status status; //接受DPLL函数返回值

//查找num或-num

pstack sta = NULL;

pChildSentence pcs = NULL;

int changeNum = 0;

bool bol = false;

pcs = findNum(pcnf, num, bol);

if (pcs == NULL)

{

printf("ERROR");

exit(-1);

}

//将pcnf中的num结点更改为-num

if (bol == true) changeNum = (-1)\*num;

else changeNum = num;

pcs->words->varibale = changeNum;

//DPLL检测是否有解

status = DPLL(pcnf, sta,0,false);

//将结点恢复原状

int \_index = 0;

if (bol == false) pcs->words->varibale = (-1)\*num;

else pcs->words->varibale = num;

destroyStack(sta, false, NULL, \_index);

//有解则证明该结点不可删除，返回TRUE

if (status == TRUE)

{

//将cnf结构重置

resetPcnf(pcnf);

return TRUE;

}

//无解将结点的mark置为-(n\*n+1),且将isCunzai置为false ，说明该结点已经被删除，返回FALSE

if (status == FALSE)

{

pcs->isCunzai = false;

pcs->mark = n \* n + 1;

pcnf->numOfchildSentence--;

//将cnf结构重置

resetPcnf(pcnf);

return FALSE;

}

}

//查找pcnf中为单子句num或者-num的结点，找到后返回指向该子句的指针，未找到则返回NULL,用bol记录该位置的数的正负

pChildSentence findNum(pCnfFile pcnf, int num, bool &bol)

{

pChildSentence pcs = pcnf->sentence;

pWord pw = NULL;

while (pcs != NULL)

{

if (pcs->mark == 0 && pcs->numOfword == 1) //单子句

{

pw = pcs->words;

if (pw->varibale == num)

{

bol = true;

return pcs;

}

if (pw->varibale == (-1)\*num)

{

bol = false;

return pcs;

}

}

pcs = pcs->nextSen;

}

exit(-1);

}

void addNodeToPcnf(pCnfFile pcnf, int word)

{

pChildSentence pcs = pcnf->sentence;

while (1)

{

if (pcs->nextSen == NULL)

break;

pcs = pcs->nextSen;

}

//找到最后一个子句的位置

pcs->nextSen = (pChildSentence)malloc(sizeof(childSentence));

if (pcs->nextSen == NULL) exit(-1);

pcs = pcs->nextSen;

pcs->isCunzai = true;

pcs->mark = 0;

pcs->numOfword = 1;

pcs->nextSen = NULL;

pcs->words = (pWord)malloc(sizeof(word));

if (pcs->words == NULL) exit(-1);

pcs->words->mark = true;

pcs->words->nextword = NULL;

pcs->words->varibale = word;

pcnf->numOfchildSentence++;

return;

}

//(1)限制1，同行、同列不能连续三个相同

void printCondition1(FILE \* file, int n)

{

//同行

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

{

for (int j = 0; j < n - 2; j++)

{

fprintf(file, "%d %d %d 0\n", n \* i + j + 1, n \* i + j + 2, n \* i + j + 3);

fprintf(file, "%d %d %d 0\n", -1 \* (n \* i + j + 1), -1 \* (n \* i + j + 2), -1 \* (n \* i + j + 3));

}

}

//同列

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

{

for (int j = 0; j < n - 2; j++)

{

fprintf(file, "%d %d %d 0\n", i + n \* j + 1, i + n \* j + 1 + n, i + n \* j + 1 + n + n);

fprintf(file, "%d %d %d 0\n", -1 \* (i + n \* j + 1), -1 \* (i + n \* j + 1 + n), -1 \* (i + n \* j + 1 + n + n));

}

}

return;

}

//(2)限制2，同一行或同一列0，1个数相同，任意n/2 +1个不相同

void printCondition2(FILE\*file, int n)

{

int len = 0;

int cur\_index;

//生成1~n中的n/2+1的所有组合数，用数组保存

bool \*bol = generateCombination(n, len);

//通过数组选择组合数

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

{

//编号为i的n个bool组成的组合数首个bool的下标为：n\*i

//同一行

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

{

//从上往下的n行

//编号为j的行开始的位置编号为：1+n\*j

//正

cur\_index = 1 + n \* j;

for (int k = n \* i; k < n\*i + n; k++, cur\_index++)

{

if (bol[k] == true)

{

fprintf(file, "%d ", cur\_index);

}

}

fprintf(file, "0\n");

//负

cur\_index = 1 + n \* j;

for (int k = n \* i; k < n\*i + n; k++, cur\_index++)

{

if (bol[k] == true)

{

fprintf(file, "%d ", -cur\_index);

}

}

fprintf(file, "0\n");

}

//同一列

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

{

//编号为j的列开始的首个位置为j+1，每次增加n

//正

cur\_index = j + 1;

for (int k = n \* i; k < n\*i + n; k++, cur\_index += n)

{

if (bol[k] == true)

{

fprintf(file, "%d ", cur\_index);

}

}

fprintf(file, "0\n");

cur\_index = j + 1;

for (int k = n \* i; k < n\*i + n; k++, cur\_index += n)

{

if (bol[k] == true)

{

fprintf(file, "%d ", -cur\_index);

}

}

fprintf(file, "0\n");

}

}

return;

}

//（3）限制3，任意两行不相同

void printCondition3(FILE\*file, int n)

{

//得到C（n,2）

int sumSeclectLine = 0; //选择两行的不同选择的总数

bool \*sceLine = printCn2(n, sumSeclectLine);

int line1 = 0, line2 = 0; //当前操作的两行的首编号

int row1 = 0, row2 = 0; //当前操作的列的首编号

//得到不同的配合数

int sumOfzuhe = 0;

bool \*zuhe = generateZuhe(n, sumOfzuhe);

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

{

//编号为i的一组c(n,2)组合开始的位置为:n\*i

//行选择，列选择

line1 = 0; line2 = 0;//初始化

for (int j = n \* i; j < n\*i + n; j++)

{

if (sceLine[j] == true)

{

if (line1 == 0)

line1 = (j - n \* i)\*n + 1;

else

line2 = (j - n \* i)\*n + 1;

}

}

//完成行选择

//对于选择的行进行操作

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

{

//标号为k的zuhe开始的坐标为k\*n,结束于k\*(n+1)-1

for (int v = k \* n; v < k\*n + n; v++)

{

if (zuhe[v] == true)

{

fprintf(file, "%d %d ", line1 + v - k \* n, line2 + v - k \* n);

}

else

{

fprintf(file, "%d %d ", -(line1 + v - k \* n), -(line2 + v - k \* n));

}

}

fprintf(file, "0\n");

}

//根据行设置列

row1 = (line1 / n) + 1;

row2 = (line2 / n) + 1;

//对于选择的列进行操作

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

{

//标号为k的zuhe开始的坐标为k\*n,结束于k\*(n+1)-1

for (int v = k \* n; v < k\*n + n; v++)

{

if (zuhe[v] == true)

{

fprintf(file, "%d %d ", row1 + n \* (v - k \* n), row2 + n \* (v - k \* n));

}

else

{

fprintf(file, "%d %d ", -(row1 + n \* (v - k \* n)), -(row2 + n \* (v - k \* n)));

}

}

fprintf(file, "0\n");

}

}

return;

}

void printConfition(void)

{

char\* name = (char\*)malloc(sizeof(char) \* 30);

int n = 0; //数独的阶数

int maxLen = 1;

int sumOfChildScentence = 0; //子句总数

printf("请输入二进制数独的阶数:(4 6 8)\n>");

getchar();

do

{

scanf("%d", &n);

} while (n != 4 && n != 6 && n != 8);

if (n == 4) sumOfChildScentence = 288;

else if (n == 6) sumOfChildScentence = 2376;

else if (n == 8) sumOfChildScentence = 16320;

else exit(-1);

char addrFilename[200] = ADDR;

strcat(addrFilename, "BinaryPuzzleConditionFile");

char ch[2] = "4";

ch[0] = n + '0';

strcat(addrFilename, ch);

FILE\* file;

//打开文件

if (!(file = fopen(addrFilename, "w")))

{

exit(-3);

}

fprintf(file, "c BinaryPuzzleContinue %d\*%d\n", n, n);

fprintf(file, "p cnf %d %d\n", n\*n, sumOfChildScentence);

printCondition1(file, n);

printCondition2(file, n);

printCondition3(file, n);

fclose(file);

printf("文件%s已存入\n", addrFilename);

return;

}

//生成1~n中的n/2+1的所有组合数，用数组保存

bool\* generateCombination(int n, int &len)

{

if (n == 8)

len = 56;

else if (n == 6)

len = 15;

else if (n == 4)

len = 4;

else

return NULL;//错误情况

bool \*combi = (bool\*)malloc(sizeof(bool) \* n\*len); //n == 8

bool buffer[8]; //缓冲区

int cur\_num; //当前操作数

//组合数

int length = 1;

int curLen = 0;

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

length \*= 2; //n位二进制数最长的情况

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

{

cur\_num = i;

//重置buffer

for (int j = 0; j < 8; j++)

buffer[j] = false;

//除二取余法，求二进制码

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

{

if (cur\_num == 0) break; //当前操作数已经为0，无需继续储存下去

if (cur\_num % 2 == 1)

buffer[j] = true;

cur\_num /= 2;

}

//求得二进制码，判断二进制码中有几个true，有 (n/2)+1个1则取用存入目标数组中去

if (countNumOfTrue(buffer, n) == (n / 2) + 1)

{

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

{

combi[curLen \* n + j] = buffer[j];

}

curLen++; //当前位置

if (curLen > len)

exit(-6); //溢出，错误情况

}

}

return combi;

}

int countNumOfTrue(bool bol[], int n)

{

int sum = 0;

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

{

if (bol[i] == true)

sum++;

}

return sum;

}

//生成00000000~11111111中的所有数

bool\* generateZuhe(int n, int &len)

{

int length = 1;

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

length \*= 2;

len = length;

int cur\_num;

//创建储存空间

bool \*zuHe = (bool\*)malloc(sizeof(bool) \* n \* length);

//初始化

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

zuHe[i] = false;

//除二取余法列出00000000~11111111中的所有数

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

{

//除二取余法

cur\_num = i;

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

{

if (cur\_num == 0) break;

if (cur\_num % 2 == 1)

zuHe[j + n \* i] = true; //第i个数存储的位置

cur\_num /= 2;

}

}

return zuHe;

}

bool\* printCn2(int n, int &len)

{

int cur\_num = 0;

len = (n\*(n - 1)) / 2;

int curLen = 0;

bool \*bol = (bool\*)malloc(sizeof(bool)\*n\*len);

int maxLen = 1;

bool buffer[8];

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

{

maxLen \*= 2;

}

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

{

cur\_num = i;

//重置buffer

for (int j = 0; j < 8; j++)

buffer[j] = false;

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

{

if (cur\_num == 0) break;

if (cur\_num % 2 == 1)

buffer[j] = true;

cur\_num = cur\_num / 2;

}

if (countNumOfTrue(buffer, n) == 2)

{

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

{

bol[curLen \* n + j] = buffer[j];

}

curLen++;

if (curLen > len)

exit(-6); //溢出，错误情况

}

}

return bol;

}

5.头文件BinaryPuzzle1.h

#define \_CRT\_SECURE\_NO\_WARNINGS

#include<stdlib.h>

#include<stdio.h>

#include<string.h>

#include<time.h>

#include<windows.h>

//宏定义区域

#define TRUE 1

#define FALSE 0

#define ERROR -1

#define OVERFLOW -2

#define OK 2

#define ADDR "C:\\Users\\18147\\Desktop\\textFile\\"

typedef int Status;

typedef struct word;

typedef struct childSentence;

typedef struct cnfFile;

//用于统计cnf中文字总数的结构

typedef struct count {

int posi; //储存当前结构中存在的正的文字的个数

int nega; //当前文字负的个数

}count,\*pCount;

//count的头结点

typedef struct headCount {

int max\_variable; //当前结构中个数最多的文字变量

int sum; //文字最多的文字变量的总数

bool containPo; //标记当前数量最多的文字是否存在正数

bool cintainNe; //标记当前结构中是否存在负数

pCount count; //指向统计数组

}headCount,\*pHeadCount;

//总 结点cnf文件，基本信息

typedef struct cnfFile {

int numOfVariable;//涉及到的文字的个数

int numOfchildSentence;

struct childSentence\* sentence;

}cnfFile, \*pCnfFile;

//子句结点

typedef struct childSentence {

struct childSentence \* nextSen;//指向另一个子句

struct word\* words; //指向文字，当此项为空时，说明为空子句

int numOfword; //该子句中文字的个数

int mark; //标记该节点是否被删除，当mark为0时，表示该节点存在，否则该节点的值等于分裂规则或者单子句规则中导致该字句被删除的文字

bool isCunzai; //仅用于生成随机数独地图时，标记当前空格是否被删除掉

}childSentence, \*pChildSentence;

//文字结点

typedef struct word {

int varibale; //文字变元

bool mark; //标记，用于标记该节点是被删除

struct word \*nextword; //单链表指向下一个文字

}word, \*pWord;

//链栈，用于非递归实现DPLL回溯

typedef struct stack {

struct stackNode \*top;

struct stackNode \*tail;

}\*pstack;

typedef struct stackNode {

int setTrueWord; //在单子句规则选中的单子句文字，或者在分裂规则中选中的假定为真的文字

bool methodMark; //标记该次化简使用所使用的规则，false--单子句规则，使用回溯时直接恢复即可，true--分裂规则使用回溯时往下深入

stackNode\* nextNode;

}stackNode, \*pStackNode;

//单子句规则查找字句的结构,数组下标作为变量

//初始化栈

void initStack(pstack &sta);

//入栈

void addStack(pstack sta, int word, bool methodMark);

//出栈

Status outStack(pstack sta, int &word, bool &methodMark);

//检测空栈

bool cheakStack(pstack sta);

//根据栈元素进行回溯

Status recovery(pstack sta, pCnfFile pcnf);

//化简规则，单子句中的化简/分裂规则中的化简

Status simplyS(pCnfFile pcnf, int word);

//检测子句中是否含有目标文字

bool checkWord(pChildSentence pcs, int word);

//选择分裂规则中使用的文字，选择结构中同时出现原文字、非文字的文字

int seclect(pCnfFile pcnf, int \*count, bool &mark);

//寻找单子句，成功找到则返回单子句的文字，否则返回0

int findSingal(pCnfFile pcnf);

//检测S中是否存在空子句

bool checkNull(pCnfFile pcnf);

//DPLL算法主体

Status DPLL(pCnfFile pcnf, pstack &stack,int beginTime,bool change);

//文件读取函数

Status readFile(pCnfFile pcnf,char \*pFilename);

//读文件创建结构

Status readAndCreate(pCnfFile pcnf, FILE\*file);

//销毁栈并递归输出路径

void destroyStack(pstack sta, bool isSuccess, int \* answer, int &index);

void destroy(pStackNode psn, pstack sta, int &i, bool isSuccess, int \* answer, int &index);

//销毁当前的cnf结构

void destroyCNF(pCnfFile pcnf);

//销毁子句

void destroyChildScen(pChildSentence pcs);

//C（n,2）组合数

bool\* printCn2(int n, int &len);

//生成1~n中的n/2+1的所有组合数，用数组保存

bool\* generateCombination(int n, int &len);

int countNumOfTrue(bool bol[], int n);

//生成00000000~11111111中的所有数按位形成的bool数组

bool\* generateZuhe(int n, int &len);

//(1)限制1，同行、同列不能连续三个相同

void printCondition1(FILE \* file, int n);

//(2)限制2，同一行或同一列0，1个数相同，任意n/2 +1个不相同

void printCondition2(FILE\*file, int n);

//（3）限制3，任意两行不相同

void printCondition3(FILE\*file, int n);

void printConfition(void);

void addNodeToPcnf(pCnfFile pcnf, int word);

//完成求解后，重置pcnf的mask

void resetPcnf(pCnfFile pcnf);

//生成数读地图

void generateSoDuKuMap(pCnfFile pcnf, int n);

//写入文件

Status writeSoDoKufile(pCnfFile pcnf);

void printMap(int \*digHole, int \*answer, int n);

//挖洞生成数独地图,传入生成的空的个数,n表示阶数

void digMap(pCnfFile pcnf, int \*digHole, int numOfHole, int n);

//将编号为num的位置设为其相反数，查看是否有解,n为阶数

Status changeAndDPLL(pCnfFile pcnf, int num, int n);

pChildSentence findNum(pCnfFile pcnf, int num, bool &bol);

//在count结构中添加变量variable

void addToCount(pHeadCount phc, int variable);

//从count结构中删除variable

void delFormCount(pHeadCount phc, int variable, pCnfFile pcnf);

//文件读取函数改进--添加统计count的操作入口

Status readFile\_1(pCnfFile pcnf, pHeadCount &phc,char \* pFilename);

//读文件创建结构

Status readAndCreate\_1(pCnfFile pcnf, FILE\*file, pHeadCount &phc);

//选取变元改写

int select\_1(pCnfFile pcnf, pHeadCount phc, bool &mark);

//化简单子句改写

Status simplyS\_1(pCnfFile pcnf, int word, pHeadCount phc);

//DPLL改写

Status DPLL\_1(pCnfFile pcnf, pstack &stack, pHeadCount phc);

//recovery函数改写

//根据栈元素进行回溯

Status recovery\_1(pstack sta, pCnfFile pcnf, pHeadCount phc);

void resetCount(pCnfFile pcnf, pHeadCount phc);

Status writeRes(char\* fileName, bool isSuccess, int sumTime, int\* answer, int n);