

**课程设计报告**

**题目：基于SAT的百分号数独游戏求解程序**

**课程名称：程序设计综合课程设计**

**专业班级： 计科2407**

**学 号： U202414852**

**姓 名： 唐俊杰**

**指导教师： 李丹**

**报告日期： 2025年9月12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：**将数双独游戏[5]问题转化为SAT问题[6-8]，并集成到上面的求解器进行数独游戏求解，游戏可玩，具有一定的/简单的交互性。应用问题归约为SAT问题的具体方法可参考文献[3]与[6-8]。(15%)

* **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>

Twodoku： https://en.grandgames.net/multisudoku/twodoku

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

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

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

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

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

[10] 薛源海，蒋彪彬，李永卓. 基于“挖洞”思想的数独游戏生成算法. 数学的实践与认识,2009,39(21):1-7

[11] 黄祖贤. 数独游戏的问题生成及求解算法优化. 安徽工业大学学报(自然科学版), 2015,32(2):187-191

**目录**

[1引言 5](#_Toc210115088)

[1.1课题背景与意义 5](#_Toc210115089)

[1.2国内外研究现状 6](#_Toc210115090)

[1.3课程设计的主要研究工作 6](#_Toc210115091)

[2系统需求分析与总体设计 8](#_Toc210115092)

[2.1系统需求分析 8](#_Toc210115093)

[2.1.1 SAT求解器 8](#_Toc210115094)

[2.1.2 百分号数独游戏 8](#_Toc210115095)

[2.2系统总体设计 9](#_Toc210115096)

[3系统详细设计 11](#_Toc210115097)

[3.1有关数据结构的定义 11](#_Toc210115098)

[3.1.1数据对象、数据项与数据类型 11](#_Toc210115099)

[3.1.2数据间的关联 12](#_Toc210115100)

[3.2主要算法设计 15](#_Toc210115101)

[3.2.1核心DPLL算法 15](#_Toc210115102)

[3.2.2CNF解析模块 18](#_Toc210115103)

[3.2.3百分号数独游戏模块 20](#_Toc210115104)

[4系统实现与测试 26](#_Toc210115105)

[4.1系统实现 26](#_Toc210115106)

[4.1.1系统开发环境 26](#_Toc210115107)

[4.1.2数据类型定义 26](#_Toc210115108)

[4.1.3函数说明 26](#_Toc210115109)

[4.2系统测试 30](#_Toc210115110)

[4.2.1 功能演示菜单展示 31](#_Toc210115111)

[4.2.2 SAT问题模块测试 31](#_Toc210115112)

[4.2.3百分号数独游戏 37](#_Toc210115113)

[5总结与展望 41](#_Toc210115114)

[5.1全文总结 41](#_Toc210115115)

[5.2工作展望 41](#_Toc210115116)

[参考文献 43](#_Toc210115117)

[附录 45](#_Toc210115118)

**1引言**

**1.1课题背景与意义**

可满足性问题（Satisfiability Problem）即SAT问题，是对一个以合取范式（Conjunctive Normal Form，常简称CNF）的形式给出的命题逻辑公式进行判断，以找出是否存在一个真值指派，使得该命题逻辑公式的值为真。SAT问题看似简单，但它却是计算机领域和人工智能领域所要研究的中心问题，被称为理论计算机科学和数理逻辑中的第一问题，在硬件验证、人工智能、电子设计自动化、自动化推理、组合等式检测等领域具有非常重要的理论和实践意义。

现实生产和生活中存在着大量的NP完全问题，寻找高效快速的算法来解决该类问题是计算机理论研究和实际应用领域中的重要工作。由于SAT问题是NP完全问题，它如果能够得到高效解决，那么一定可以高效地解决所有其它NP完全问题，这是因为所有的NP完全问题都能在多项式时间内进行相互转化，即所有的NP完全问题都能够在多项式时间内转换为可满足性问题。所以，如果能找到求解SAT问题的有效算法，那么所有其它的NP完全问题也都可以解决了，设计和实现更高效的求解算法意义重大。

SAT问题的应用领域非常广泛，例如在数学研究和应用领域，它能用来解决旅行商（Traveling Salesman Problem，TSP）和逻辑算术问题；在计算机和人工智能（Artificial Intelligence）领域中，它能解CSP（约束满足问题）问题、语义信息的处理和逻辑编程等问题；在计算机辅助设计领域中，它能很好的解决任务规划与设计、三维物体识别等问题。许多的实际问题如人工智能、积木世界规划问题、数据库检索、Jobshop排工问题、超大规模集成电路设计和图着色都可转换为SAT问题进行求解。

本课题研究基于DPLL算法的SAT求解器并寻找优化策略，其具有重要的理论和实际意义。首先，DPLL算法作为SAT求解的基础算法，其性能优化直接影响到SAT问题的解决效率。通过深入研究DPLL算法，可以揭示其在处理大规模SAT实例时的表现，为改进算法提供理论支持。其次，DPLL算法的研究和应用不仅推动了SAT求解技术的进步，还为相关领域的算法研究和工程应用提供了宝贵的经验。尤其是在硬件设计验证、自动推理等实际应用中，DPLL算法的有效实现能够显著提高问题解决的效率和准确性。

## 1.2国内外研究现状

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。

但是1996年以后，很多国家都相继举办了一些SAT竞赛和研讨会，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[1]、SATO[2]、CHAFF[3]、POSIT[4]和GRASP[5]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[6]中，提出模拟退火算法。2022年，华中科技大学计算机学院何琨教授团队提出将随机游走策略与决策树模型相结合的创新方法，使求解器在面对具有不同特征的算例时采用不同的随机游走策略辅助搜索，很好地提升了SAT求解器的鲁棒性。

**1.3课程设计的主要研究工作**

本次课设主要研究SAT问题的理论知识，并学习基于DPLL算法对SAT进行求解和算法优化，在此基础上，设法将百分号数独游戏转化为SAT问题并调用上面的求解器进行求解。具体而言，课程设计的主要研究工作包括：

（1）研究学习SAT问题的概念、背景、研究意义和研究现状，学习理论知识。

（2）了解DPLL算法的基本原理和处理策略，分析涉及的数据结构，划分功能、人机交互需求与数据文件读写等，设计并实现一个基于DPLL算法的SAT求解器。

（3）通过测试不同规模和复杂度的SAT样例，评估DPLL算法求解的准确性和速度。基于测试结果，对算法和程序进行优化，提出改进策略，以提高求解效率。

（4）从完整合法填充开始，基于挖洞法自动生成合规的百分号数独。并将百分号数独游戏规约为SAT问题，并把它表示成CNF公式的形式。将构建的CNF公式输入到SAT求解器中，利用DPLL算法求解以确定数独的解。

2系统需求分析与总体设计

**2.1系统需求分析**

## 2.1.1 SAT求解器

SAT求解器应具有以下功能：

（1）输入功能：用户输入cnf测试文件路径，程序通过读取内容存储到一定的数据结构中。程序应该具备检查错误输入的能力，并提醒用户正确输入，例如防止在没有读取文件的情况下求解，同时还应该具备能再次输入不同文件进行检测的能力。

（2）输出功能：程序能够将cnf算例文件求解的结果和运行时间保存到同名.res文件中，并将运行结果打印到屏幕，包括SAT/UNSAT、可满足时各变量的赋值状态、运行时间、优化率。

（3）DPLL求解时间：程序通过调用DPLL函数，对读入的cnf测试集进行求解，用户可以选择是否运行优化版本

（4）交互功能：程序应能在屏幕上打印合理布局用于交互，直观显示每一块的功能，提供良好的用户体验。

## 2.1.2 百分号数独游戏

百分号数独游戏应具有以下功能：

（1）生成数独：程序应能将数独约束规则规约成一个cnf文件，转化成解SAT问题，生成一个随机数独，然后基于挖洞法随机生成一个有唯一解的数独并打印。

（2）游玩数独：用户可以通过输入横纵坐标以及值来选择如何填入数独。程序应该具有检查错误的功能，例如用户输入的坐标是否在有效范围内、所选格子是否已经有数填入、输入是否违反了数独游戏规则。初次之外，程序应具备修改所填值的功能。

（3）输出正确答案：程序应能将初始生成的数独输出，显示正确答案。

（4）交互功能：程序应能在屏幕上打印合理布局用于交互，直观显示每一块的功能以及漂亮的数独格局，提供良好的用户体验。

**2.2系统总体设计**

本系统要求实现一个具有交互功能的SAT求解器和百分号数独游戏。为实现上述功能并遵循模块化的设计原则，本系统被划分为以下几个核心模块，源代码也依据模块进行组织：

（1）主控与显示模块（display.cpp，SAT.cpp）

功能：作为程序的入口和总调度中心，提供顶层菜单（选择进入SAT求解器或数独游戏），处理用户输入。调用其它模块完成相应的功能，并控制界面的清屏与刷新。

核心函数：main() Display() PrintMenu()

（2）CNF解析模块(cnfparser.cpp)

功能：负责读取和解析用于测试的.cnf文件，将文件内容转化为内部的链表数据结构。销毁检测完成的.cnf文件，释放空间。打印读入的.cnf文件内容。

核心函数：ReadFile() DestroyCnf() PrintCnf()

（3）DPLL求解模块(solver.cpp)

功能：实现DPLL的核心算法逻辑，包括单子句规则、分支变量选择策略、递归求解和结果保存。

核心函数：DPLL() Simplify() FindUnitClause() ChooseLiteral\_1()

ChooseLiteral\_2() ChooseLiteral\_3() SaveResult()

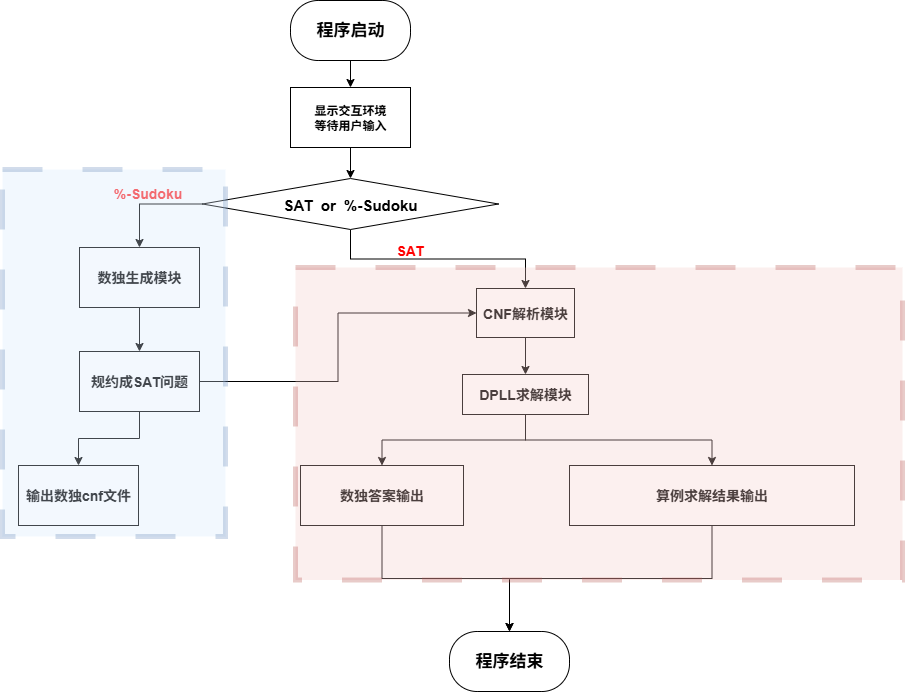
（4）百分号数独模块(%-Sudoku.cpp)

功能：集成了百分号数独的所有相关功能，包括数独的生成、游戏逻辑、用户交互、从数独到SAT问题的规约以及游戏求解和验错模块。

核心函数：Sudoku() Generate\_Sudoku() HasUniqueSolution() WriteToFile() Play\_Sudoku()

（5）头文件与数据结构定义(SAT.hpp)

功能：定义了贯穿整个项目的数据结构，如literalNode, clauseNode, cnfNode,以及所有函数的声明和全局常量的定义，是各模块之间交互接口。



**图 1 程序框架流程图**

**3系统详细设计**

## 3.1有关数据结构的定义

该系统主要处理两类核心数据：一类是用于表示和求解SAT（布尔可满足性问题）的逻辑范式结构，另一类是用于“百分号数独”游戏的游戏板和状态数据。

### 3.1.1数据对象、数据项与数据类型

系统中用于表示合取范式（CNF）的核心数据结构是动态的、基于指针的链式结构。这种设计非常适合CNF范式的特点，因为不同的算例其变量数、子句数以及每个子句中包含的文字数量都可能不同，采用链表可以灵活地表示这种不固定的结构。

（1）文字节点

1. typedef struct literalNode {
2. int literal;              *// 文字(变元)*
3. struct literalNode \*next; *// 指向下一个文字*
4. } literalNode, \*literalList;

（2）子句节点

1. typedef struct clauseNode {
2. literalList head;        *// 指向子句中的第一个文字*
3. struct clauseNode \*next; *// 指向下一个子句*
4. } clauseNode, \*clauseList;

（3）CNF文件

1. typedef struct cnfNode {
2. clauseList root; *// 指向CNF的第一个子句*
3. int boolCount;   *// 布尔变元个数*
4. int clauseCount; *// 子句个数*
5. } cnfNode, \*CNF;

以下表格为数据类型、数据项及详细描述的总结：

**表 1 数据项、数据类型及详细描述表**

|  |  |  |  |
| --- | --- | --- | --- |
| 结构名称 | 数据项 | 数据类型 | 详细描述 |
| **literalNode** | **literal** | **int** | **该字段存储一个布尔文字的整数表示。正数 x 代表变量 x，负数 -x 代表其否定 ¬x。** |
|  | **next** | **struct literalNode \*** | **指向同一个子句中下一个文字节点的指针。多个literalNode通过next指针串联，形成一个单向链表，代表一个完整的析取子句。** |
| **clauseNode** | **head** | **literalList** | **指向该子句文字链表的头节点。通过这个指针可以遍历子句中的所有文字。** |
|  | **next** | **struct clauseNode \*** | **指向CNF公式中下一个子句节点的指针。多个clauseNode通过next指针串联，形成代表整个CNF公式的单向链表。** |
| **cnfNode** | **root** | **clauseList** | **指向整个公式的子句链表的头节点，是访问所有子句的入口点。** |
|  | **boolCount** | **int** | **存储从CNF文件头部（p cnf行）读取的布尔变量总数。** |
|  | **clauseCount** | **int** | **存储从CNF文件头部读取的子句总数。** |

### 3.1.2数据间的关联

程序的核心数据关联是一个典型的**层次化链表结构**，这种结构完全是在 cnfparser.cpp 的 ReadFile 函数中动态构建起来的。为节省篇幅，这里只展示部分关键代码，完整代码见附录。

（1）顶层入口创建（display.cpp）:

1. CNF cnf = (CNF)malloc(sizeof(cnfNode));
2. cnf->root = NULL;

此时，cnf 作为一个容器被创建，但其root指针为空，表示还没有任何子句。

(2) 链表动态构建 (cnfparser.cpp):

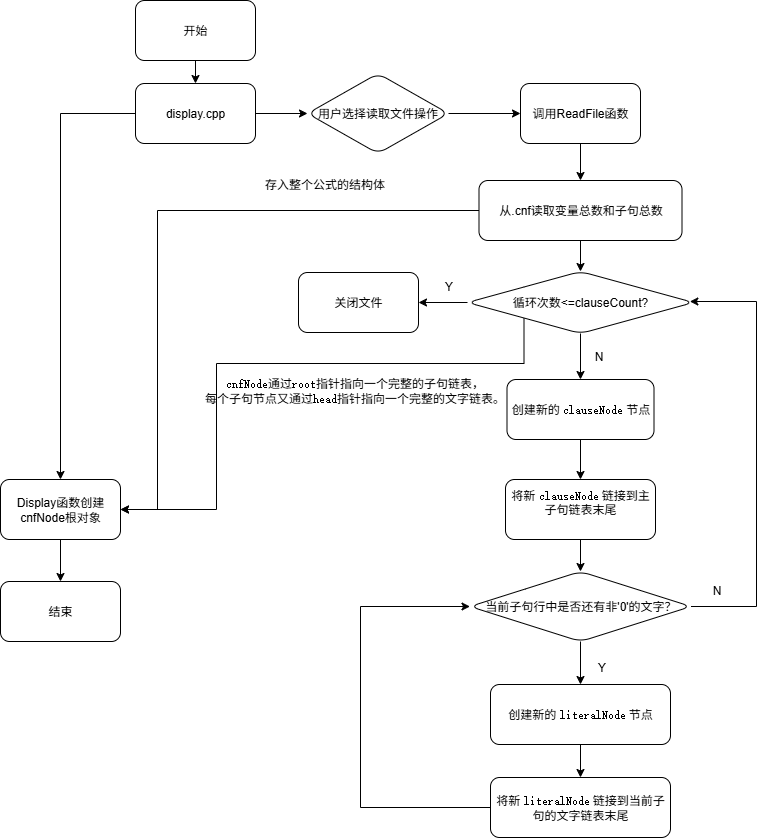
·**构建子句链表（外层循环）：**ReadFile 函数中使用一个 for 循环，根据读取到的 clauseCount 来创建相应数量的子句节点。

1. clauseList lastClause = NULL; *// 用于链接子句的尾指针*
2. for (int i = 0; i < cnf->clauseCount; i++) {
3. clauseList newClause = (clauseList)malloc(sizeof(clauseNode)); *// 1. 创建新的子句节点*
4. ...
5. if (cnf->root == NULL) *// 如果是第一个子句*
6. cnf->root = newClause; *// 直接挂在根节点上*
7. else
8. lastClause->next = newClause; *// 否则链接到上一个子句的末尾*
9. lastClause = newClause; *// 2. 更新尾指针*
10. }

·**构建文字链表 (内层循环)**: 在上述for循环的内部，有一个while循环负责读取每个子句中的所有文字，直到遇到0为止。

1. newClause->head = NULL;
2. literalList lastLiteral = NULL; *// 用于链接文字的尾指针*
3. ...
4. fscanf(fp, "%d", &number);
5. while (number != 0) {
6. literalList newLiteral = (literalList)malloc(sizeof(literalNode)); *// 3. 创建新的文字节点*
7. newLiteral->literal = number;
8. ...
9. if (newClause->head == NULL) *// 如果是该子句的第一个文字*
10. newClause->head = newLiteral; *// 直接挂在子句节点的head上*
11. else
12. lastLiteral->next = newLiteral; *// 否则链接到上一个文字末尾*
13. lastLiteral = newLiteral; *// 4. 更新尾指针*
14. fscanf(fp, "%d", &number);
15. }

下图为直观展示这多种数据间的关联的流程图：



**图 2 数据关联图**

## 3.2主要算法设计

由于各模块设计的函数过多，这里仅对各个模块中主要函数的实现进行介绍，源码位于附录中。

### 3.2.1核心DPLL算法

**1.主函数status DPLL(CNF cnf, bool value[], int flag)：**

（1）输入：

CNF cnf: 一个cnfNode结构体指针，包含了指向当前待求解CNF公式的指针以及变量和子句总数等元数据。

bool value[]: 一个布尔数组，用于存储求解过程中和最终结果中每个变量的真值（TRUE/FALSE）。

int flag: 一个整型标志，用于选择在分裂阶段使用哪种启发式策略来挑选变量 (1, 2, 3分别对应不同的策略)

（2）输出：status (整型): 如果公式可满足，返回OK (1)；如果不可满足，返回ERROR (0)

（3）算法思想描述: 该函数采用递归和回溯的搜索策略来解决SAT问题。其核心思想是通过**单子句传播**规则来处理所有确定性的赋值，从而快速化简问题、缩小搜索空间。当没有单子句可供传播时，算法进入**分裂**阶段，通过一个启发式策略选择一个未赋值的变量进行“猜测”，并对这个变量的两种可能取值（真或假）分别进行递归探索。如果一个分支导致了冲突，则回溯到上一个决策点，尝试另一个分支。

（4）算法处理步骤：

**单子句传播**：函数首先进入一个while循环，该循环会持续进行，直到公式

中不存在单子句为止。

步骤 1a: 调用FindUnitClause(cnf->root)查找是否存在单子句。如果找到，返回该单子句中的文字unitLiteral。

步骤 1b: 如果找到了unitLiteral，则根据其正负确定对应变量的真值，并存入value数组。

步骤 1c: 调用Simplify(cnf->root, unitLiteral)函数，根据unitLiteral的真值对整个CNF公式进行化简。

步骤 1d: 化简后，立即检查两个终止条件：

调用Satisfy(cnf->root)，如果返回OK（公式为空），说明已找到解，直接返回OK。调用EmptyClause(cnf->root)，如果返回TRUE（存在空子句），说明当前路径出现矛盾，直接返回ERROR。

**分裂决策：**如果单子句传播的while循环正常结束（即没有单子句了），但公式既未满足也未产生冲突，则需要进行分支。

步骤2a:根据传入的flag参数，调用对应的启发式函数（如ChooseLiteral\_3(cnf)）来选择一个最佳的未赋值文字literal进行分裂。

**递归与回溯**:

步骤 3a (分支一：假设literal为真):调用CopyCnf(cnf->root)创建当前公式的一个深拷贝，保存不变。将文字literal作为一个新的单子句添加到这个拷贝的公式头部。对这个修改后的拷贝进行递归调用DPLL。如果递归调用返回OK，说明此分支成功，直接将OK作为结果返回。

步骤 3b (分支二：假设literal为假):如果分支一返回ERROR，说明假设literal为真行不通。程序会销毁之前创建的拷贝，实现回溯。将文字literal的否定形式-literal作为一个新的单子句添加到原始公式的头部。对这个修改后的原始公式进行第二次递归调用DPLL。第二次递归调用的返回值（无论是OK还是ERROR）将作为当前函数调用的最终结果返回。

（5）时间复杂度：在最坏情况下，算法需要遍历所有种可能的变量赋值组合，因此时间复杂度为O()，其中n是变量的数量。单子句传播和启发式策略能够显著剪枝，在许多实际算例中远达不到这个上界。

（6）空间复杂度：O(n⋅m)，其中n是变量数量，m是子句数量 。空间开销主要来自两个方面：

递归栈深度: 最深可达n层 。

公式拷贝: 在每个分裂点，都需要调用CopyCnf复制整个公式，其大小与子句和文字总数成正比。

**2.核心辅助函数Simplify(clauseList &cL, int literal):**

（1）输入：

clauseList &cL: 对子句链表头指针的引用，允许函数直接修改链表结构。

int literal: 需要进行传播的单子句文字。

（2）输出：void: 函数没有返回值，直接在传入的链表上进行修改。

（3）算法思想描述：根据布尔逻辑化简规则，当一个文字L被确定为真时，任何包含L的子句都变为真，可以从公式中移除；任何包含¬L的子句中，¬L这一项变为假，可以被删除。该函数精确地实现了这两个操作。

（4）算法处理步骤：

**遍历子句链表**: 使用pre和p两个指针遍历整个子句链表，pre指向前一个节点，p指向当前节点，便于删除操作。

**遍历文字链表**: 对于每个子句p，再使用lpre和q两个指针遍历其内部的文字链表。

**判断与操作**:

情况一 (子句满足): 如果在文字链表中发现q->literal == literal，说明当前子句p已被满足。此时，通过修改pre->next指针将p从子句链表中移除，并调用DestroyClause(p)彻底释放该子句及其所有文字节点的内存。

情况二 (文字为假): 如果发现q->literal == -literal，说明q这个文字节点为假。此时，仅将q从当前子句的文字链表中移除，并释放q节点的内存。

情况三 (无关文字): 如果既不满足情况一也不满足情况二，则继续遍历下一个文字。

（5）时间复杂度：，其中是当前公式中所有文字的总数。在最坏情况下，函数需要遍历每一个子句中的每一个文字一次。

（6）空间复杂度：，函数执行的是原地修改，除了几个指针变量外，不使用额外的、与输入规模相关的存储空间。

**3.启发式策略函数(优化后算法)int ChooseLiteral\_3(CNF cnf)：**

（1）输入：CNF cnf: 当前待求解的CNF公式结构体指针。

（2）输出：int: 经过启发式策略选出的最适合进行分裂的文字。

（3）算法思想描述：该函数实现了MOMs(Maximum Occurrences in clauses of Minimum size)启发式策略的一个简化版本。其核心思想是，最短的子句是最“脆弱”的，它们最容易因为变量赋值而产生冲突或新的单子句。因此，选择一个在这些最短子句中出现最频繁的文字进行分裂，可以最大化每次决策的影响力，从而期望能更快地剪枝搜索树。

（4）算法处理步骤：

寻找最短子句: 遍历一遍所有的子句，计算每个子句的长度（包含的文字数），找到当前公式中的最短子句长度minSize，并用一个指针temp记录下找到的第一个最短子句。

统计文字频率: 遍历指针temp所指向的那个最短子句中的所有文字。使用一个count数组来统计每个文字出现的次数（在代码中，只统计了temp指向的这一个最短子句里的文字）。

找出最频繁文字: 遍历count数组，找到计数值最大的那个文字。

返回结果: 返回上一步中找到的最频繁的文字。

（5）时间复杂度：，主要开销在于第一步为了寻找最短子句需要遍历所有文字。

（6）空间复杂度：：，其中n是变量总数。主要开销来自于需要一个大小与变量总数成正比的count数组来存储每个文字的出现频率。

（7）相比未优化策略的优点：与ChooseLiteral\_1的盲目选择不同，ChooseLiteral\_3首先会扫描整个公式，找到最短的子句。这些短子句是问题的“瓶颈”或“最薄弱环节”。对这些子句中的变量进行赋值，最有可能产生立竿见影的效果——要么满足该子句，要么使其成为新的单子句，从而触发一连串的单子句传播，快速化简问题。通过做出更“聪明”的决策，算法能够更快地走向正确的解路径，或者更快地发现当前路径是错误的。无论是哪种情况，都能显著减小需要探索的搜索树规模，从而避免了大量无效的计算和回溯。

### 3.2.2CNF解析模块

**核心函数status ReadFile(CNF &cnf, char fileName[])：**

（1）输入：

CNF &cnf: 一个对cnfNode指针的引用。函数将把解析出的数据填充到这个结构体所指向的内存中。

char fileName[]: 包含待解析CNF公式的文件名。

（2）输出：status (整型): 如果文件成功读取并解析，返回OK

（3）算法思想描述：该函数遵循标准的DIMACS CNF文件格式规范，通过顺序文件读取和解析，动态地在内存中构建起一个层次化的链表结构。它首先跳过注释，读取关键的元数据（变量数和子句数），然后通过嵌套循环，逐一创建子句节点和文字节点，并将它们正确地链接起来，最终形成完整的公式表示。

（4）算法处理步骤：

**打开文件:** 使用fopen尝试打开fileName。如果失败，程序会进入一个while循环，提示用户重新输入文件名，直到成功打开文件为止。

**跳过注释行:** 进入一个while循环，使用getc逐字符读取。如果行首字符为'c'，则继续读取并丢弃该行的所有后续字符，直到换行符'\n'为止。

**解析p cnf行:** 跳过注释后，程序预期会遇到格式行。它跳过固定的 "p cnf " 字符串，然后使用fscanf读取两个关键的整数，分别存入cnf->boolCount和cnf->clauseCount。

**初始化:** 声明一个尾指针lastClause = NULL，它将用于高效地在子句链表的末尾添加新节点，避免每次都从头遍历。

**遍历并创建子句 (外层循环):** 进入一个for循环，其次数由cnf->clauseCount决定。

步骤 5a: 在循环体内，首先使用malloc为新的子句分配一个clauseNode节点 (newClause)。

步骤 5b: 声明并初始化用于文字链表的尾指针lastLiteral = NULL。

**遍历并创建文字 (内层循环):** 进入一个while循环，该循环通过fscanf读取整数，直到读取的数字为0（DIMACS格式中子句的结束符）时停止。

步骤 6a: 在循环体内，使用malloc为新的文字分配一个literalNode节点 (newLiteral)。

步骤 6b: 将读取到的数字存入newLiteral->literal。

步骤 6c (链接文字): 判断当前子句的头指针newClause->head是否为空。如果为空，则这是第一个文字，令newClause->head = newLiteral；否则，将新节点链接到尾指针之后 lastLiteral->next = newLiteral。

步骤 6d: 更新文字链表的尾指针 lastLiteral = newLiteral。

**链接子句:** 当内层while循环结束后，一个完整的子句就构建完毕了。

步骤 7a: 判断整个公式的根指针cnf->root是否为空。如果为空，则这是第一个子句，令cnf->root = newClause；否则，将新子句链接到主链表的尾部 lastClause->next = newClause。

步骤 7b: 更新主子句链表的尾指针 lastClause = newClause。

**收尾:** 当外层for循环结束后，整个文件解析完毕。调用fclose关闭文件指针，并返回OK。

（5）时间复杂度：其中是文件中所有文字（非零整数）的总数。算法需要顺序读取并处理文件中的每一个有效数字一次。

（6）空间复杂度：，同样与文件中文字总数成正比。程序为每一个子句和每一个文字都分配了独立的内存节点，因此其空间开销直接反映了所表示问题的大小。

### 3.2.3百分号数独游戏模块

**1.数独生成核心函数status Generate\_Sudoku(int board[SIZE + 1][SIZE + 1],int newBoard[SIZE + 1][SIZE + 1],int newBoard2[SIZE + 1][SIZE + 1],bool isFixed[SIZE + 1][SIZE + 1], int num,bool value[SIZE \* SIZE \* SIZE + 1])**

（1）输入:

int board[SIZE + 1][SIZE + 1] 等多个二维数组: 用于存储最终生成的初始谜题、答案和供玩家操作的棋盘。

bool isFixed[SIZE + 1][SIZE + 1]: 用于标记哪些格子是固定提示数的数组。

int num: 用户期望的提示数数量。

bool value[]: 一个足够大的数组，用于接收DPLL求解器返回的729个变量的解。

（2）输出：status (整型): 如果成功生成谜题，返回OK。

（3）算法思想描述：该函数采用“生成后挖洞”的策略。它不依赖于预存的题库，而是动态地、随机地创建一个完整的、合法的数独终局。然后，它通过一个试探性的过程，随机地从终局中移除数字（挖洞），并利用SAT求解器作为验证工具，确保每次挖洞后谜题仍然有解，直到达到用户指定的提示数数量。

（4）算法处理步骤：

**初始化**: 使用当前时间time(NULL)作为随机数种子。清空所有棋盘数组。

**播种 (Seeding)**: 为了引入随机性并确保初始约束，调用Fill\_Box函数，用随机打乱的数字1-9填充两个特殊的“窗口”区域。

**生成完整终局**:

步骤 3a: 调用WriteToFile函数，将这个仅填充了两个窗口的“半成品”棋盘转化为一个CNF问题文件 (Sudoku.cnf)。

步骤 3b: 调用ReadFile解析刚刚生成的Sudoku.cnf文件。

步骤 3c: 调用DPLL求解器来求解这个问题。由于初始约束很少，求解器会找到一个满足所有数独规则的完整填充方案。

步骤 3d: 将DPLL返回的value解数组解码，填充到board二维数组中，至此获得了一个随机的、合法的数独终局。

容错: 如果DPLL求解失败（理论上可能，但概率极小），程序会通过goto START语句跳转回步骤1，重新开始整个生成过程。

**挖洞：**

步骤 4a: 进入一个while循环，目标是挖去81 - num个洞。

步骤 4b: 在循环中，随机选择一个格子的坐标(row, col)。

步骤 4c: 如果该格子尚未被挖，则先保存其数字temp = board[row][col]，然后将其置零，完成一次“试探性”挖洞。

步骤 4d (验证): 调用HasUniqueSolution函数，该函数会再次将挖洞后的棋盘转化为CNF并求解，以验证谜题是否仍然有解。

步骤 4e (决策): 如果HasUniqueSolution返回OK，说明挖洞成功，正式确认此次挖洞。如果返回ERROR，说明这个洞使得谜题无解，必须撤销操作，将temp恢复到board[row][col]中。

（5）时间复杂度：非常高，且难以精确表达。它由多次调用DPLL求解器主导，而DPLL本身是指数级的。生成终局需要一次DPLL调用，挖洞过程中的HasUniqueSolution函数在最坏情况下可能需要调用数十次DPLL。因此，这是一个非常耗时的操作。

（6）空间复杂度：主要由调用DPLL求解器时的空间开销决定，即O(n⋅m)。此外，还需要存储数个棋盘数组，空间为O(SIZE2)，这部分是常数。

**2.数独规约模块status WriteToFile(int board[SIZE + 1][SIZE + 1], int num, char name[])**

（1）输入：

int board[SIZE + 1][SIZE + 1]: 包含当前数独状态（有提示数）的二维数组。

int num: 棋盘上提示数的个数，用于精确计算总子句数并写入文件头。

char name[]: 输出的.cnf文件名。

（2）输出：status (整型): 文件写入成功返回OK

（3）算法思想描述：该函数通过一系列大规模的、高度结构化的嵌套循环，系统性地将百分号数独的所有规则（格子、行、列、九宫格、撇对角线、窗口）以及棋盘上的提示数，逐条翻译成等价的布尔逻辑子句，并以CNF标准格式写入到输出文件中。

（4）算法处理步骤：

**计算子句总数并写入文件头**: 函数首先精确计算出所有规则将产生的子句总数，然后打开文件，写入p cnf 729 <总子句数>的文件头。

**写入提示数约束**: 遍历board数组，对于每一个非零的格子（提示数），将其转化为一个单子句并写入文件。这是最优先处理的，因为单子句对DPLL求解器化简最有效。

**写入格子约束**: 通过两层循环遍历81个格子，再内嵌两层循环，生成确保“每个格子有且仅有一个数字”的子句。

**写入行列/九宫格/对角线/窗口约束**: 同样使用多重嵌套循环，为每一行、每一列、每个九宫格、撇对角线和两个窗口，分别生成确保“数字1-9有且仅出现一次”的约束子句。

**关闭文件**: 所有子句写入完毕后，关闭文件指针。

（5）时间复杂度：对于固定的9x9数独，这是一个常数时间操作，即O(1)。虽然内部有大量循环（甚至高达六重循环），但循环的次数是固定的（例如9\*9\*9\*9），不随外部输入变化

（6）空间复杂度：O(1)。函数在执行过程中只使用有限的几个循环变量，它将生成的子句直接流式写入文件，而不在内存中构建完整的数据结构。

**2.唯一解验证函数status HasUniqueSolution(int board[SIZE + 1][SIZE + 1],bool isFixed[SIZE + 1][SIZE + 1],int solution[SIZE + 1][SIZE + 1])**

（1）输入：

int board[SIZE + 1][SIZE + 1]: 经过一次“试探性”挖洞后的数独棋盘。

bool isFixed[SIZE + 1][SIZE + 1]: 固定的提示数标记数组。

int solution[SIZE + 1][SIZE + 1]: 一个输出参数，如果找到解，则将解填充到此数组中。

（2）输出：status (整型): 如果该棋盘**至少有一个解**，返回OK (1)；如果无解，返回ERROR。

（3）算法思想描述：第一阶段 (存在性验证): 算法首先尝试求解一次谜题。这一步的目的是确认谜题是否至少有一个解。如果连一个解都找不到，谜题本身就是无效的，可以直接得出结论。第二阶段 (唯一性验证): 如果第一阶段成功找到了一个解（称之为），算法会进入关键的第二阶段。它会根据解动态地构建一个特殊的“阻塞子句”。这个子句是解的逻辑否定，当把它加入到原始问题中时，相当于增加了一个约束：找到的任何新解都不能等于。然后，算法会带着这个新约束，对问题进行第二次求解。如果第二次求解失败，意味着在排除了 之后再也找不到任何其他解了。这反过来证明了是唯一的。如果第二次求解成功，意味着找到了一个不同于的新解。这证明了谜题的解不唯一。

（4）算法处理步骤：

**阶段一：首次求解**

步骤1a: 调用 WriteToFile 将当前 board 的状态转化为一个临时的CNF文件。

步骤1b: 调用 ReadFile 解析该文件到内存结构 cnf1 中。

步骤1c: 调用 DPLL 对 cnf1 进行求解。

步骤1d (无解判断): 如果 DPLL 返回 ERROR，说明该谜题无解。函数立即清理资源（释放cnf1，删除临时文件）并返回 FALSE。

步骤1e (保存第一个解): 如果 DPLL 返回 OK，说明找到了一个解。函数会用一个循环将 value1 数组中的解完整地复制到 firstSolution 数组中，以备后用。同时，将这个解解码并填充到solution输出参数中。

步骤1f: 清理第一次求解所用的 cnf1 结构体。

**阶段二：构建阻塞子句并二次求解**

步骤2a (构建阻塞子句):动态创建一个 clauseNode 节点 (blockingClause)。进入一个 for 循环，遍历 firstSolution 数组中的全部729个变量。对于每个变量 i，根据其在 firstSolution 中的真值，向 blockingClause 中添加一个与之相反的文字。如果 firstSolution[i] 为 TRUE，则添加文字 -i；反之则添加文字 i。循环结束后，blockingClause 成为一个包含729个文字的超长子句，它在逻辑上排除了第一个解。

步骤2b (准备二次求解):重新调用 ReadFile 读取同一个临时CNF文件，将其加载到一个新的、干净的内存结构 cnf2 中。将 blockingClause 链接到 cnf2 的子句链表头部，并将 cnf2 的子句总数加一。

步骤2c (二次求解): 再次调用 DPLL 求解这个被增强了约束的 cnf2。

**判断结果并返回**

步骤3a (唯一性判断): 检查第二次 DPLL 调用的结果 result2。如果 result2 == ERROR，证明解是唯一的，将最终结果 finalResult 设为 TRUE。如果 result2 == OK，证明存在其他解，将 finalResult 设为 FALSE。

步骤3b (最终清理): 调用 DestroyCnf 释放 cnf2（此操作会一并释放 blockingClause），并删除临时CNF文件。

步骤3c: 返回 finalResult。

（5）时间复杂度：函数的性能开销主要由两次调用 DPLL 求解器决定。单次 DPLL 调用的最坏时间复杂度为指数级，即，其中k是未定变量的数量（对于数独问题，k 最大为729）。因此，该函数的总时间复杂度约为，仍然是级别。

（6）空间复杂度：空间开销同样由 DPLL 求解器主导，其在递归过程中需要存储CNF公式的副本和递归栈，复杂度为 O(n⋅m)，其中n为变量数，m为子句数。函数自身还需要额外的空间来存储第一个解 (firstSolution 数组，大小为 O(n)) 和阻塞子句（大小也为 O(n)）。由于两次 DPLL 调用是顺序执行的，它们的空间开销不会叠加。因此，总的空间复杂度与单次 DPLL 调用相同，即 O(n⋅m)。

**3.检验玩家结果正确性模块status Is\_Valid(int board[SIZE + 1][SIZE + 1], int row, int col, int v)**

（1）输入：int board[SIZE + 1][SIZE + 1]: 一个完整的、被填满的数独棋盘。

（2）输出：status (整型): 如果棋盘完全符合所有百分号数独的规则，返回TRUE (1)；否则返回FALSE (0)。

（3）算法思想描述：该函数通过暴力检查的方式，系统性地验证一个完整的棋盘是否满足所有规则。它为每一个待检查的单元（行、列、九宫格、对角线、窗口）使用一个临时的布尔数组used[]作为哈希集合，来高效地检测是否存在重复数字。

（4）算法处理步骤：****

**检查所有行**: 外层循环遍历9行。对每一行，初始化used数组为全false。然后内层循环遍历该行所有单元格，如果遇到一个数字在used数组中已被标记，则说明有重复，立即返回FALSE。否则，将该数字在used数组中标记为true。

**检查所有列**: 逻辑同上，但内外层循环交换，按列进行检查。

**检查所有九宫格**: 通过步长为3的双重循环遍历9个九宫格的起始点。对每个九宫格，初始化used数组并检查内部是否有重复数字。

**检查撇对角线**: 初始化used数组，遍历对角线上的9个格子，检查是否有重复。

**检查两个窗口**: 对两个特殊的3x3窗口区域，分别初始化used数组并检查内部是否有重复。

（5）时间复杂度：。函数需要遍历棋盘上的每个格子数次（一次为行检查，一次为列检查，一次为九宫格检查等）。由于SIZE是常量9，这是一个非常快速的常数时间操作。

（6）空间复杂度：。每次检查一个单元时，需要一个大小为SIZE+1的used布尔数组。由于SIZE是常量9，这也是一个固定的常数空间开销。

**4系统实现与测试**

**4.1系统实现**

### 4.1.1系统开发环境

（1）操作系统版本：Microsoft Windows 11 家庭中文版 X64

（2）平台工具：Visual Studio 2022

### 4.1.2数据类型定义

1. */\*文字节点&链表\*/*
2. typedef struct literalNode {
3. int literal;              *// 文字(变元)*
4. struct literalNode \*next; *// 指向下一个文字*
5. } literalNode, \*literalList;
6. */\*子句节点&链表\*/*
7. typedef struct clauseNode {
8. literalList head;        *// 指向子句中的第一个文字*
9. struct clauseNode \*next; *// 指向下一个子句*
10. } clauseNode, \*clauseList;
11. */\*CNF文件\*/*
12. typedef struct cnfNode {
13. clauseList root; *// 指向CNF的第一个子句*
14. int boolCount;   *// 布尔变元个数*
15. int clauseCount; *// 子句个数*
16. } cnfNode, \*CNF;

### 4.1.3函数说明

（1）主控与显示模块

|  |  |
| --- | --- |
| 函数名 | 函数说明 |
|  | 程序的标准入口点。它的唯一作用是调用DisPlay()函数，从而将程序的控制权交给主交互界面。 |
|  | 系统的核心控制循环和交互界面。它初始化cnfNode结构，在一个while循环中向用户展示菜单选项，并根据用户的输入调用相应的功能函数。 |
| ***PrintMenu()*** | 一个简单的辅助函数，负责在终端打印主菜单的文本内容，使DisPlay()函数的主体逻辑更清晰。 |

下图为该模块中各函数的调用关系：

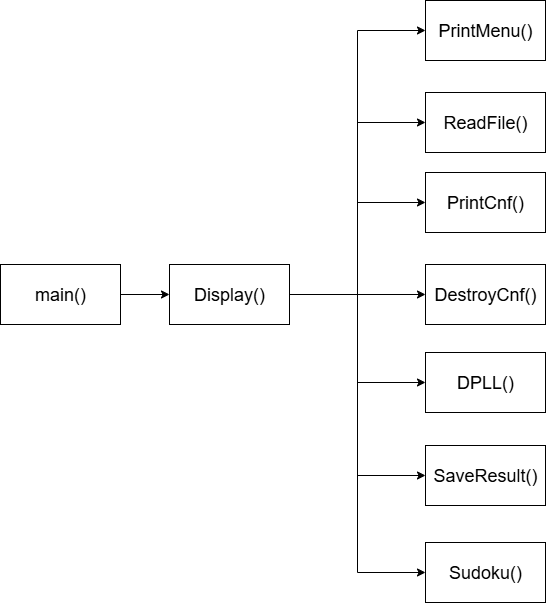


图 3 主控与显示模块函数调用关系图

（2）CNF解析模块

|  |  |
| --- | --- |
| 函数名 | 函数说明 |
| ***ReadFile()*** | 核心功能函数，负责打开并解析符合DIMACS CNF格式的文件，然后在内存中动态地创建并链接所有clauseNode和literalNode，构建起完整的层次化链表数据结构。 |
| ***PrintCnf()*** | 一个验证和调试函数。它遍历由ReadFile()创建的内存数据结构，并将所有子句和文字打印到控制台，方便用户核对文件是否被正确解析。在一个while循环中向用户展示菜单选项，并根据用户的输入调用相应的功能函数。 |
| ***DestroyCnf()*** | 内存管理函数。它负责彻底释放一个CNF公式所占用的所有动态内存，通过遍历并释放每一个literalNode和clauseNode来防止内存泄漏。 |

这些函数都是被按需调用。

（3）DPLL求解模块

|  |  |
| --- | --- |
| 函数名 | 函数说明 |
| ***DPLL()*** | 核心求解引擎，实现了递归回溯的DPLL算法。它通过“单子句传播”和“分裂策略”来寻找问题的解。 |
| ***Simplify()*** | DPLL的关键辅助函数，负责根据一个确定的文字（单子句）来化简整个CNF公式。 |
| ***FindUnitClause()*** | 在CNF公式中查找并返回第一个找到的单子句中的文字。 |
| ***ChooseLiteral\_3()*** | 优化的启发式策略函数。在需要进行“猜测”时，它会选择最短子句中出现次数最多的文字来进行分裂。 |
| ***CopyCnf()*** | 深度复制函数。在DPLL进行分支递归前，用它来创建一个当前CNF公式的完整副本，以保存现场，便于后续的回溯。 |
| ***SaveResult()*** | 文件输出函数，负责将求解结果（SAT/UNSAT）、解（各变量的真值）以及运行时间按照指定格式写入到.res文件中。 |

下图为该模块中各函数的调用关系：

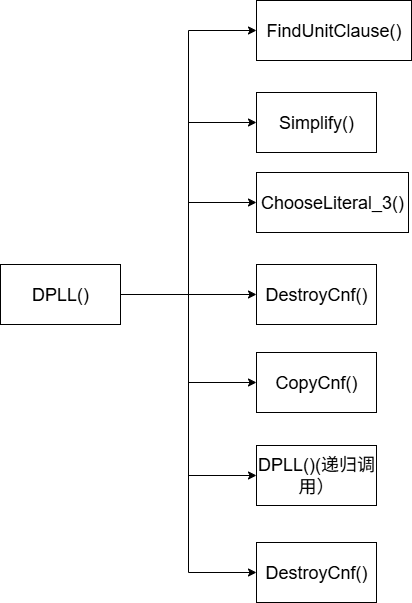
**

图 4 DPLL求解模块函数调用关系图

（4）百分号数独模块：

|  |  |
| --- | --- |
| 函数名 | 函数说明 |
| ***Sudoku()*** | 数独子系统的入口和主交互循环，功能类似于DisPlay()，但提供的是数独相关的菜单选项。 |
| ***Generate\_Sudoku()*** | 负责创建一个新的、随机的、有解的百分号数独谜题。它通过先生成一个完整终局再“挖洞”的方式来实现。 |
| ***WriteToFile()*** | 数独到SAT的“编译器”。它将一个数独棋盘的布局和所有游戏规则转化为等价的CNF逻辑公式，并写入文件。 |
| ***HasUniqueSolution()*** | 唯一性验证函数。它通过“求解-阻塞-再求解”的两阶段SAT求解过程，来严格判断一个数独谜题是否有唯一解。 |
| ***Play\_Sudoku()*** | 负责处理玩家的交互式游戏过程，接收玩家输入，更新棋盘并进行实时检查。 |
| ***Is\_Valid()*** | 实时检查函数，用于判断玩家尝试的一步填数操作是否违反了任何数独规则。 |
| ***Validate\_Sudoku()*** | 全盘验证函数，用于检查一个被完全填满的棋盘是否是百分号数独的一个正确的解。 |

下图为该模块中各函数的调用关系：



图 5 百分号数独模块函数调用关系图

**4.2系统测试**

软件测试是确保软件质量、可靠性和性能的关键步骤，是软件开发生命周期中不可或缺的一环。通过系统化的测试，可以发现并修复程序中潜在的缺陷、逻辑错误和性能瓶颈，从而验证软件是否满足其设计目标和用户需求。接下来，我将选取系统中三个技术水平和重要性最高的关键模块，进行详细的测试过程描述。

### 4.2.1 功能演示菜单展示

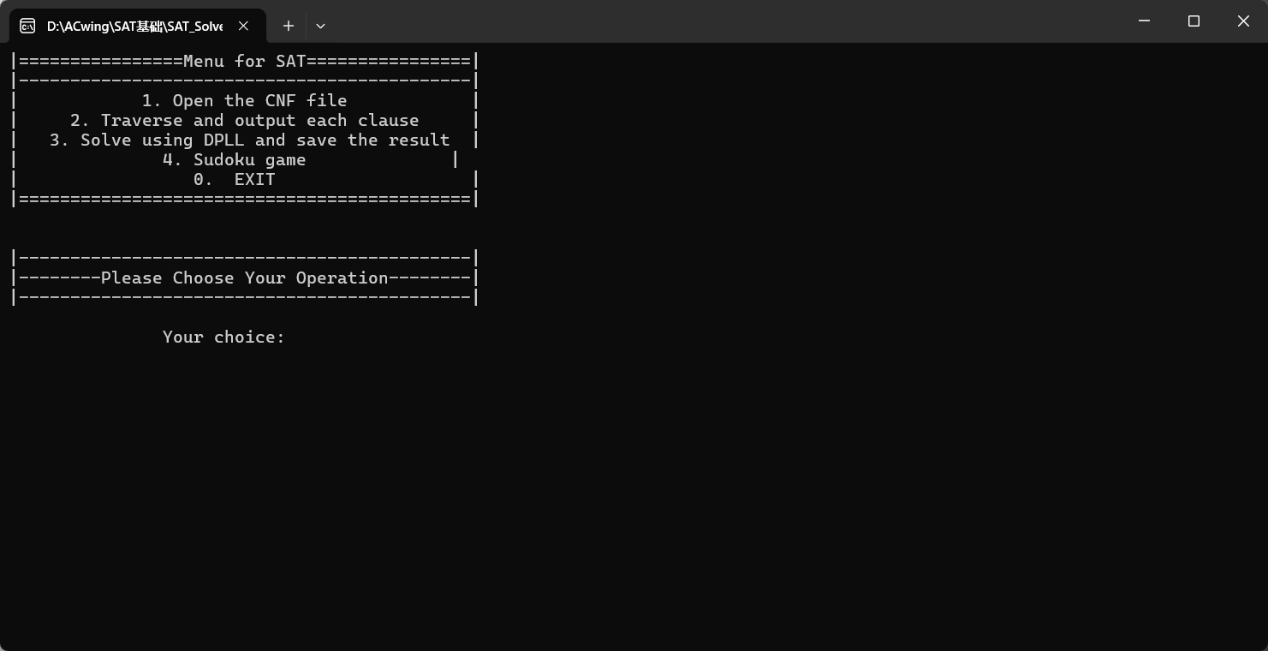


图 6功能演示菜单展示

### 4.2.2 SAT问题模块测试

（1）模块功能与设计目标

**模块功能**: 该模块是整个项目的逻辑核心。其主要功能是接收一个以内存数据结构表示的CNF范式，通过 DPLL() 函数判断该范式的可满足性。如果可满足（SAT），则返回成功状态并给出一组使范式为真的变量赋值；如果不可满足（UNSAT），则返回失败状态。

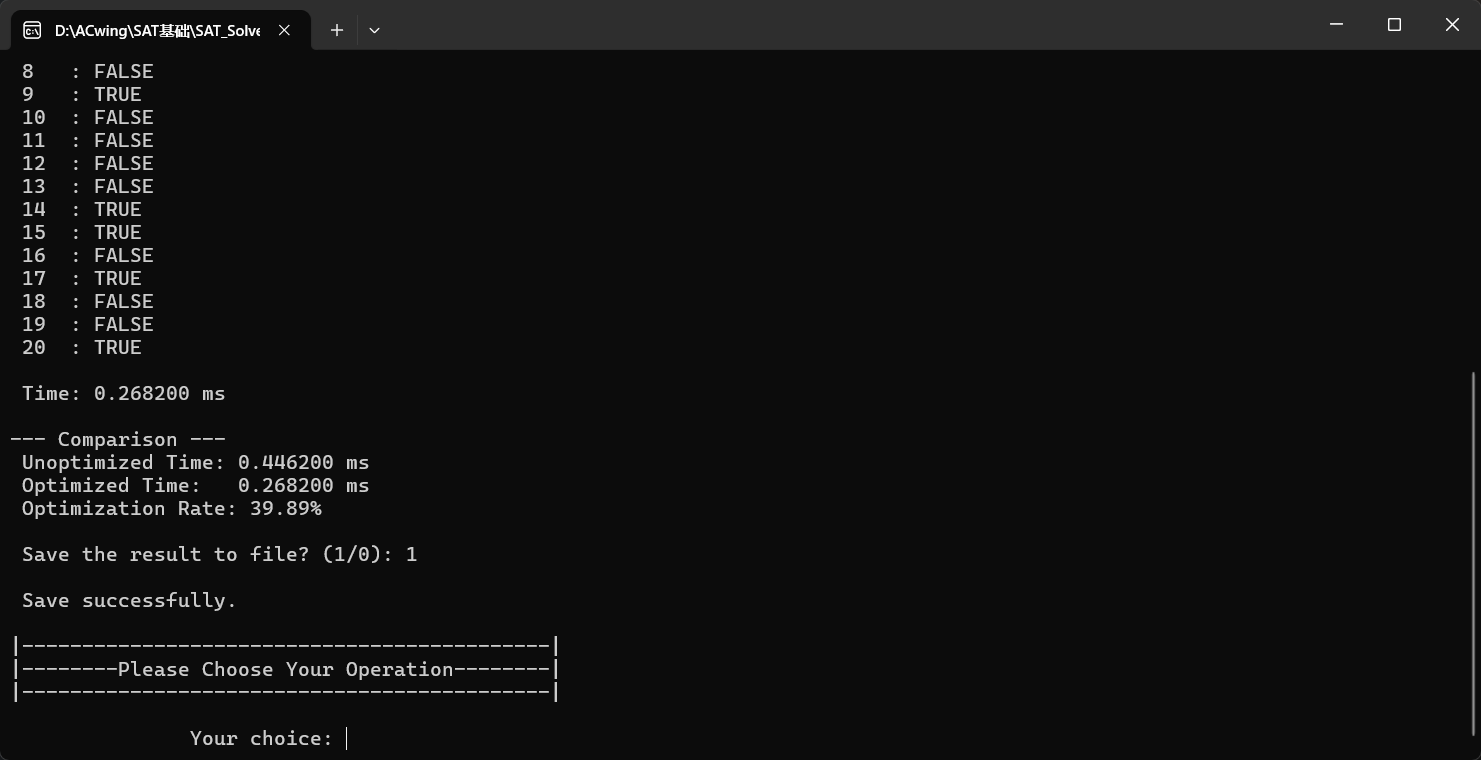
**设计目标**: 正确、高效地实现带有“单子句传播”和“分裂回溯”的DPLL算法。特别是，要通过优化的启发式策略 (ChooseLiteral\_3) 显著提升求解性能，并能够处理中等规模的SAT算例。

（2）测试数据选取与测试大纲

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 测试类型 | 满足算例 | | | 不满足算例 | | |
| L | M | S | L | M | S |
| **性能测试** | 5个 | 5个 | 5个 | 1个 | 1个 | 1个 |
| **功能测试** | 1个 | | | 1个 | | |

（3）功能测试：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算例名称 | 变元数 | 子句数 | 子句数与变元数比值 | 预期结果 | 实际结果 |  |  | 优化率 |
| sat-20.cnf | 20 | 91 | 4.55 | 可满足 | 可满足 | 0.446 | 0.268 | 39.89% |
| unsat-5cnf-30.cnf | 30 | 420 | 14 | 不满足 | 不满足 | 207.57 | 82.70 | 60.16% |



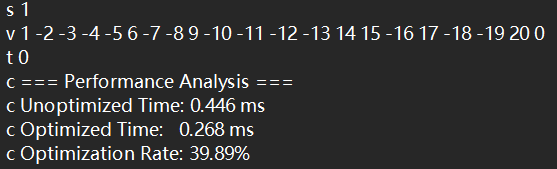
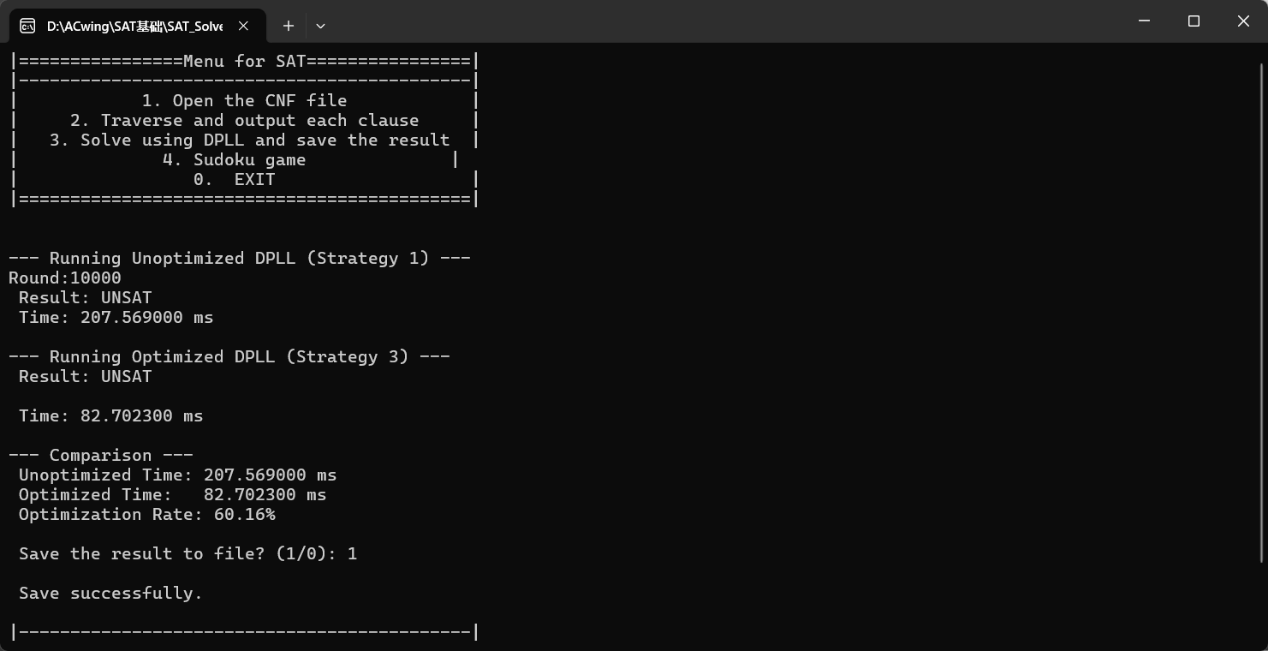


图 7 功能测试1运行结果和答案保存



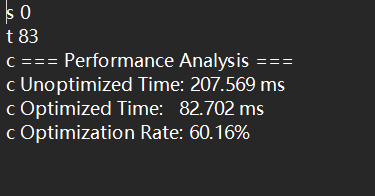
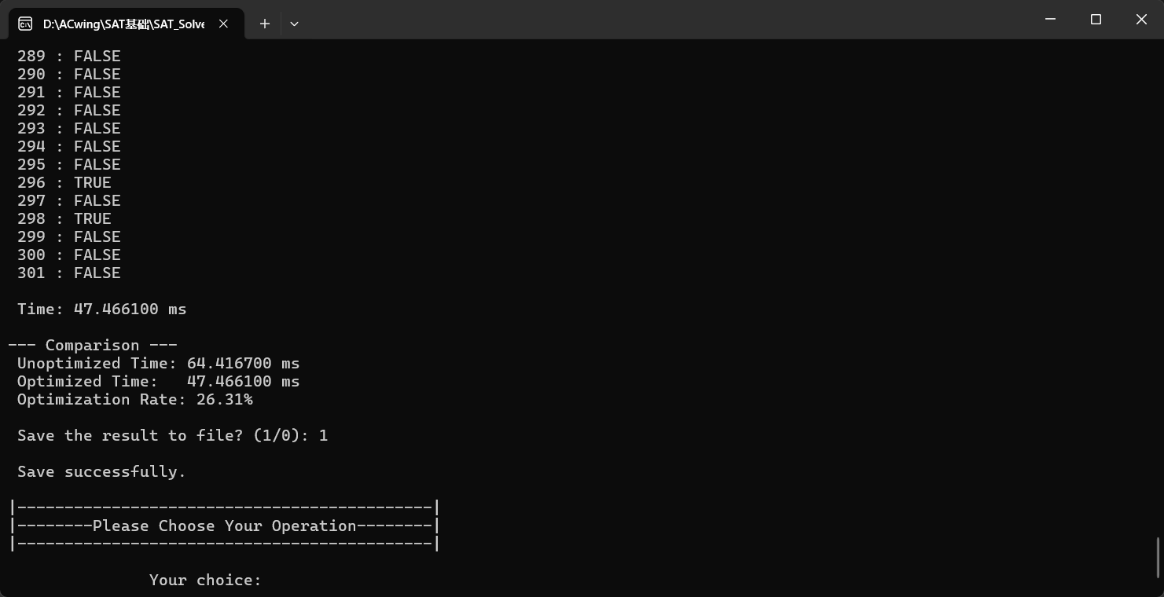


图 8 功能测试2运行和答案保存

（4）性能测试

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算例名称 | 变元数 | 子句数 | 子句数与变元数比值 | 预期结果 | 实际结果 |  |  | 优化率 |
| problem3-100. | 100 | 340 | 3.4 | 可满足 | 可满足 | 17.63 | 22.11 | -25.38% |
| sud00001.cnf | 301 | 2780 | 9.24 | 可满足 | 可满足 | 76.30 | 47.14 | 38.21% |
| 2.cnf | 1075 | 3152 | 2.93 | 可满足 | 可满足 | / | 3316  50.72 | / |
| 1.cnf | 200 | 1200 | 6 | 可满足 | 可满足 | 37.48 | 114.75 | -206.09% |
| 3.cnf | 301 | 2780 | 9.24 | 可满足 | 可满足 | 64.42 | 47.47 | 26.31% |
| 4（unsatisfied）.cnf | 512 | 9685 | 18.92 | 不满足 | 不满足 | 4983.47 | 3734.44 | 25.06% |
| 5.cnf | 20 | 1532 | 76.6 | 可满足 | 可满足 | 104.07 | 70.21 | 32.54% |
| 6.cnf | 265 | 5666 | 21.38 | 可满足 | 可满足 | 89360.68 | 5326.99 | 94.04% |
| 8（unsatisfied）.cnf | 238 | 634 | 2.66 | 不满足 | 不满足 | / | 94523.56 | / |
| 11（unsatisfied）.cnf | 60 | 936 | 10.4 | 不满足 | 不满足 | / | 78196.92 | / |
| problem9-100.cnf | 100 | 200 | 2 | 满足 | 满足 | 18.44 | 6.88 | 62.68% |
| problem11-100.cnf | 100 | 600 | 6 | 满足 | 满足 | 12.82 | 15.72 | -22.55% |
| problem8-50.cnf | 50 | 300 | 6 | 满足 | 满足 | 4.21 | 3.82 | 9.06% |
| sud00009.cnf | 303 | 2857 | 9.43 | 满足 | 满足 | 192.15 | 16.145 | 91.6% |
| eh-dp04s04.shuffled-1075.cnf | 1075 | 3152 | 2.93 | 满足 | 满足 | / | 320891.31 | / |

由于篇幅限制，这里只展示部分运行结果，分别是三个满足算例和一个不满足算例。由于未优化求解时间过长，故对于大数据集只进行优化算法结果展示：



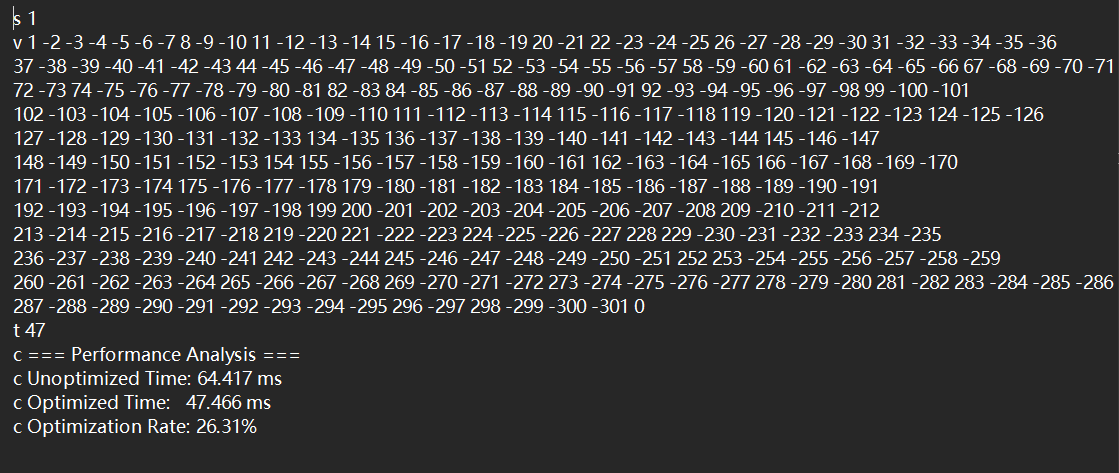
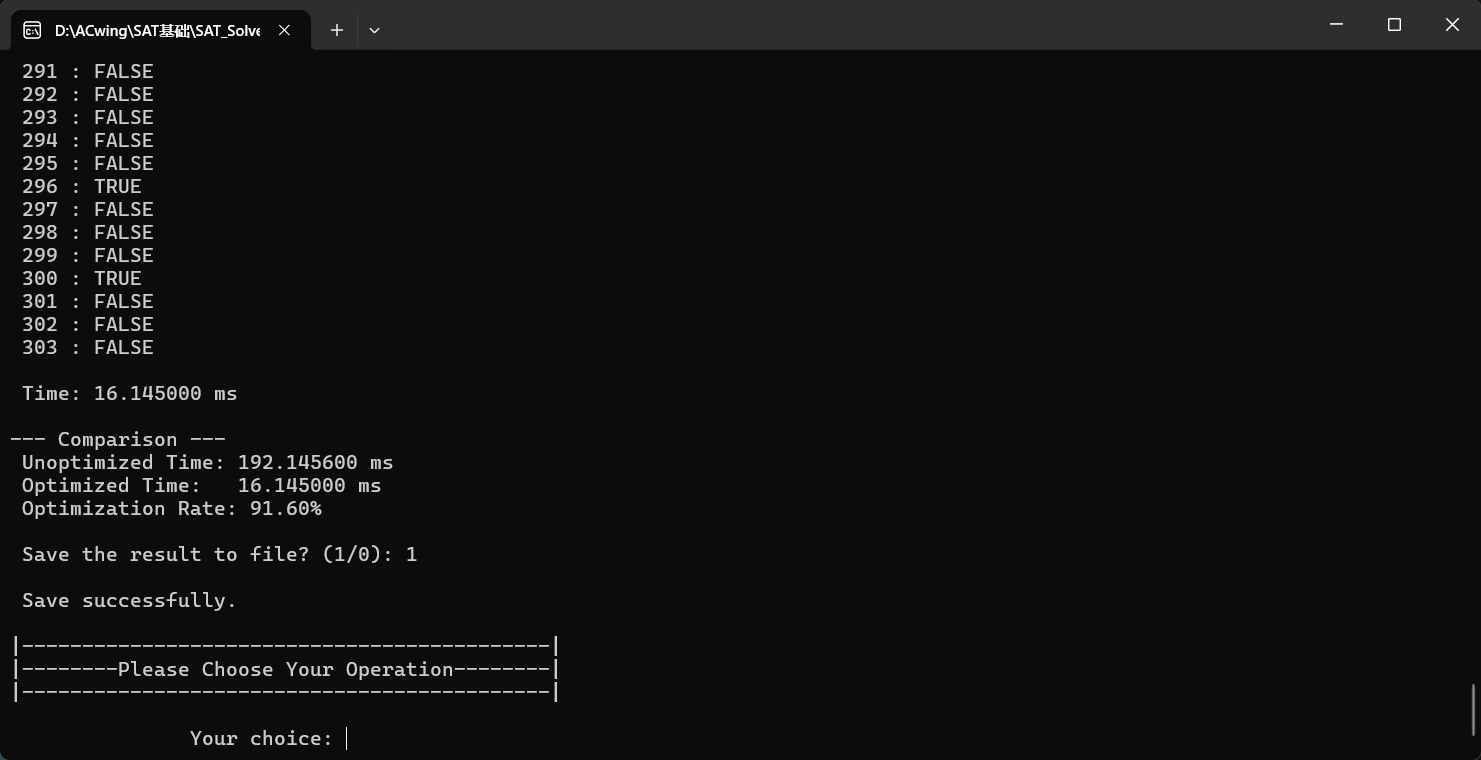


图 9 性能测试3.cnf运行结果和答案保存



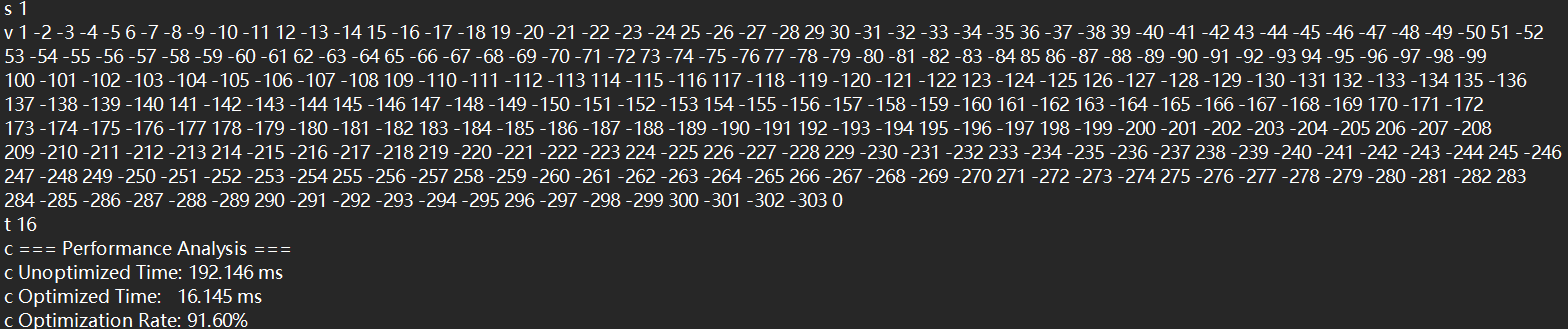
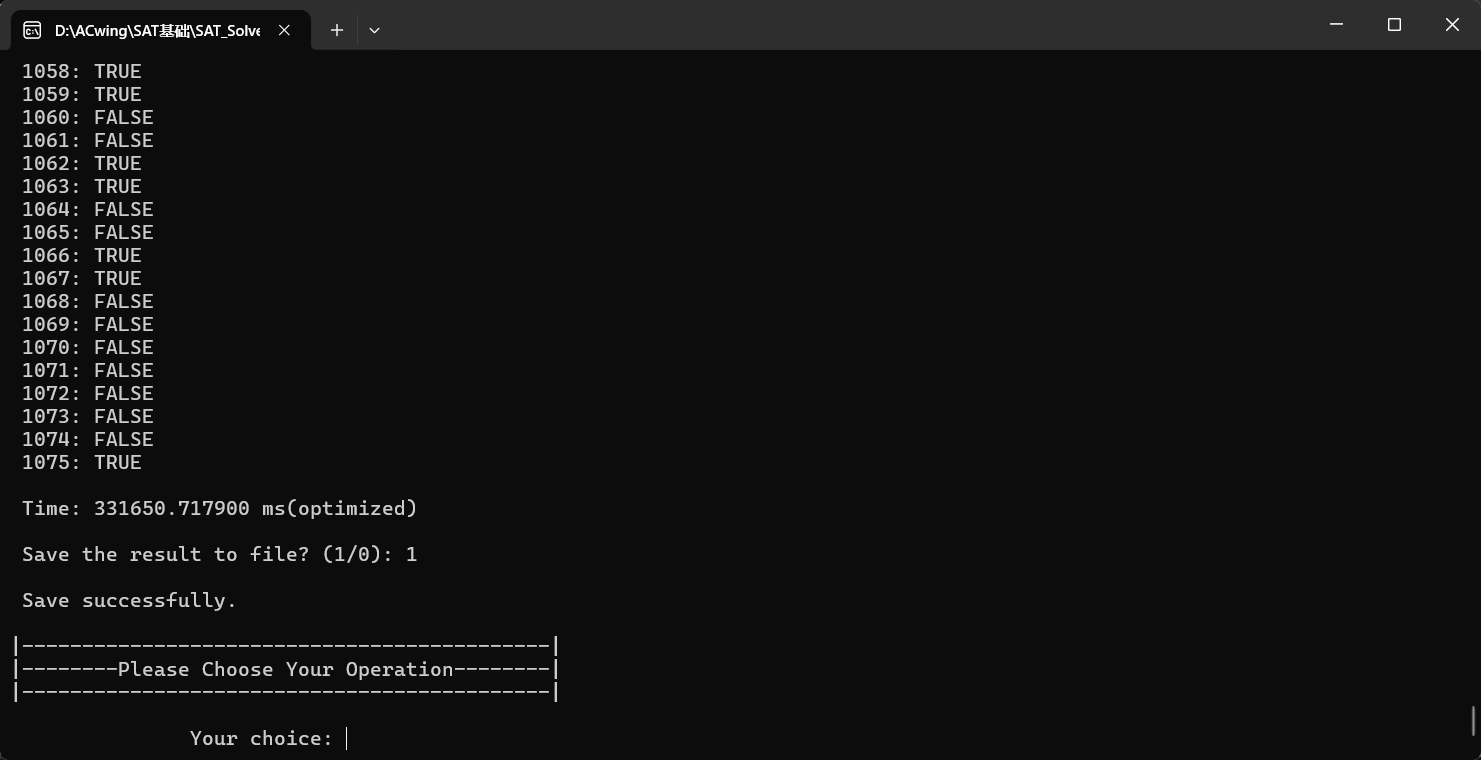


图 10 性能测试sud00009.cnf运行结果和答案保存



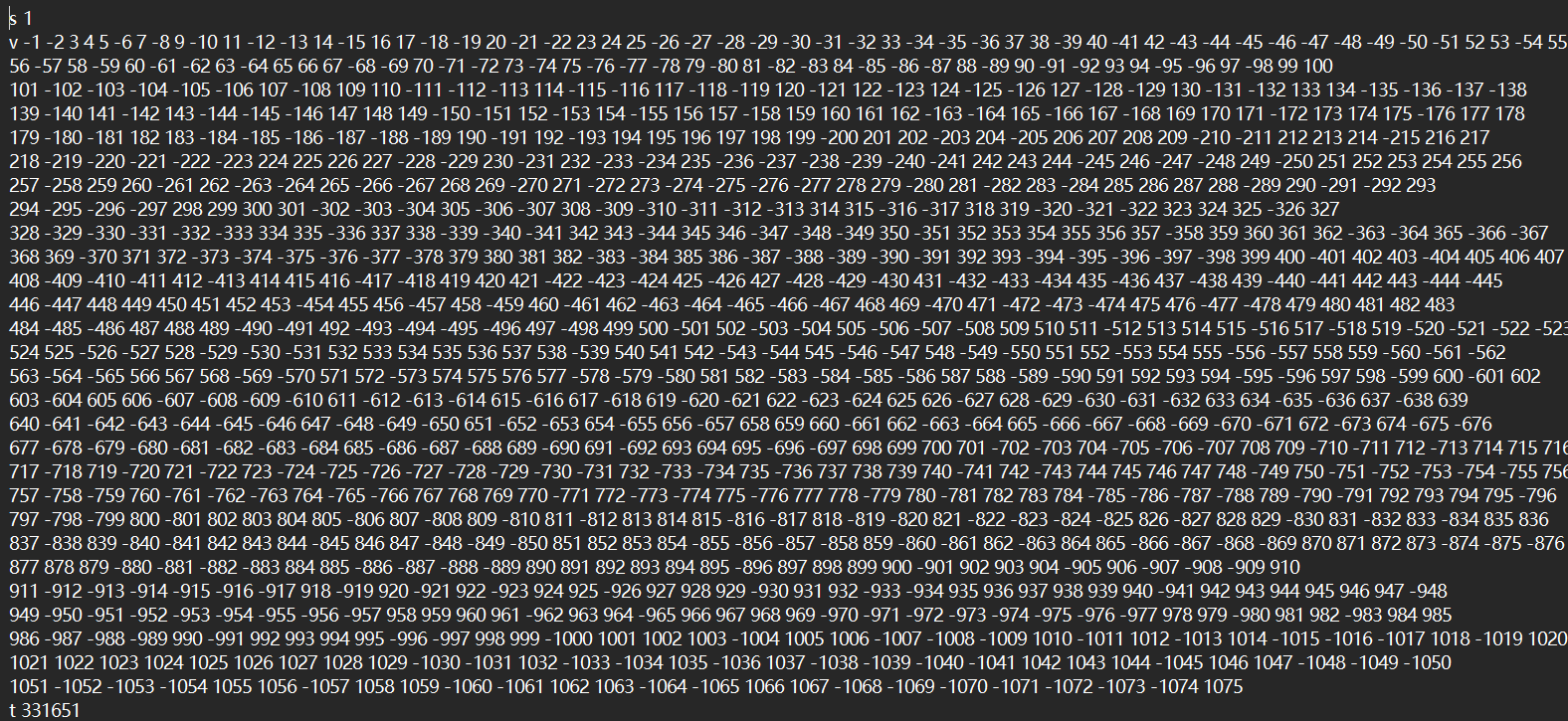


图 9 性能测试2.cnf运行结果和答案保存

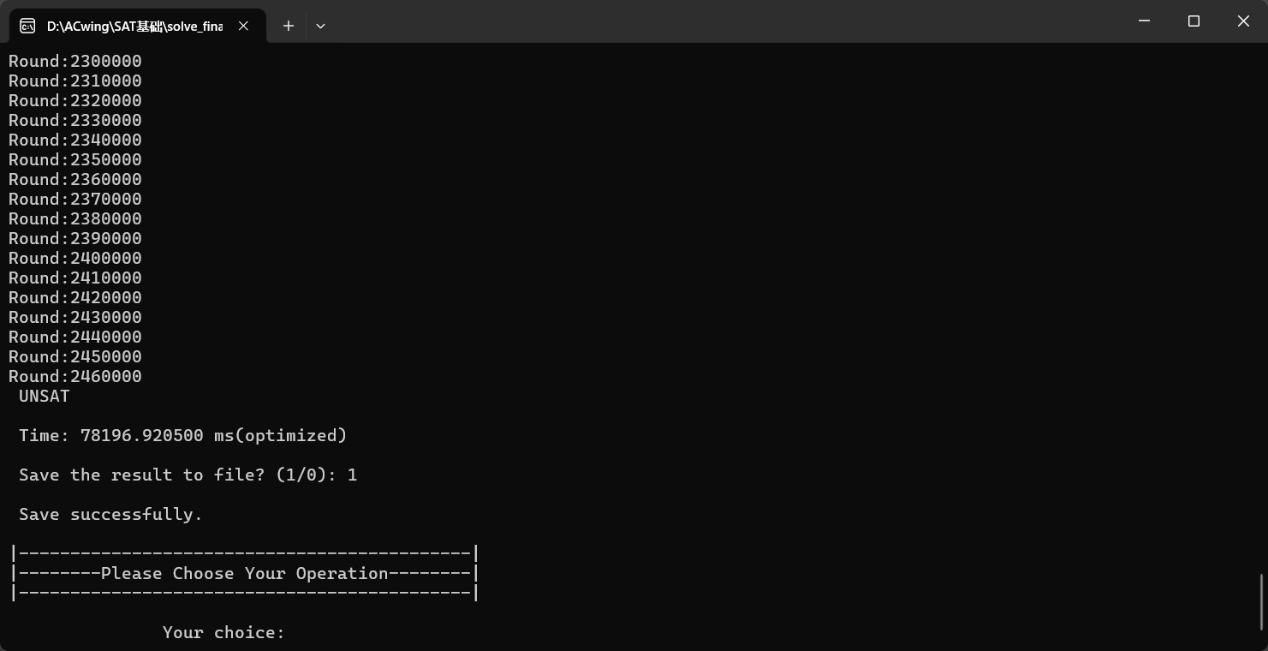




图 11 性能测试11(unsatisfied).cnf运行结果和答案保存

### 4.2.3百分号数独游戏

（1）模块功能与设计目标

**模块功能**: 该模块是SAT求解器的核心应用。主要功能是通过Generate\_Sudoku()函数，根据用户指定的提示数，动态生成一个随机的、符合百分号数独所有规则、且**有唯一解**的谜题。

**设计目标**: 实现一个鲁棒的“生成后挖洞”算法。其关键在于正确使用修改后的HasUniqueSolution()函数，在挖洞的每一步都严格保证谜题解的唯一性，从而产出高质量的数独谜题。

（2）数独生成验证

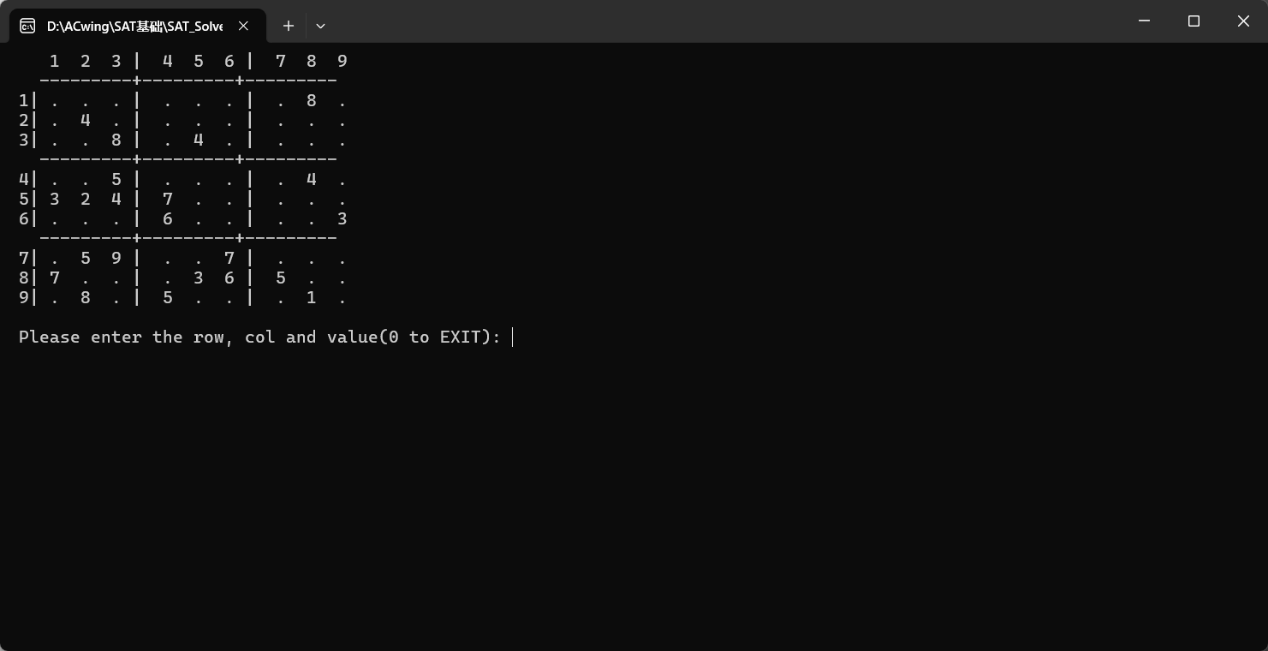


图 12 数独游戏开局随机生成数独

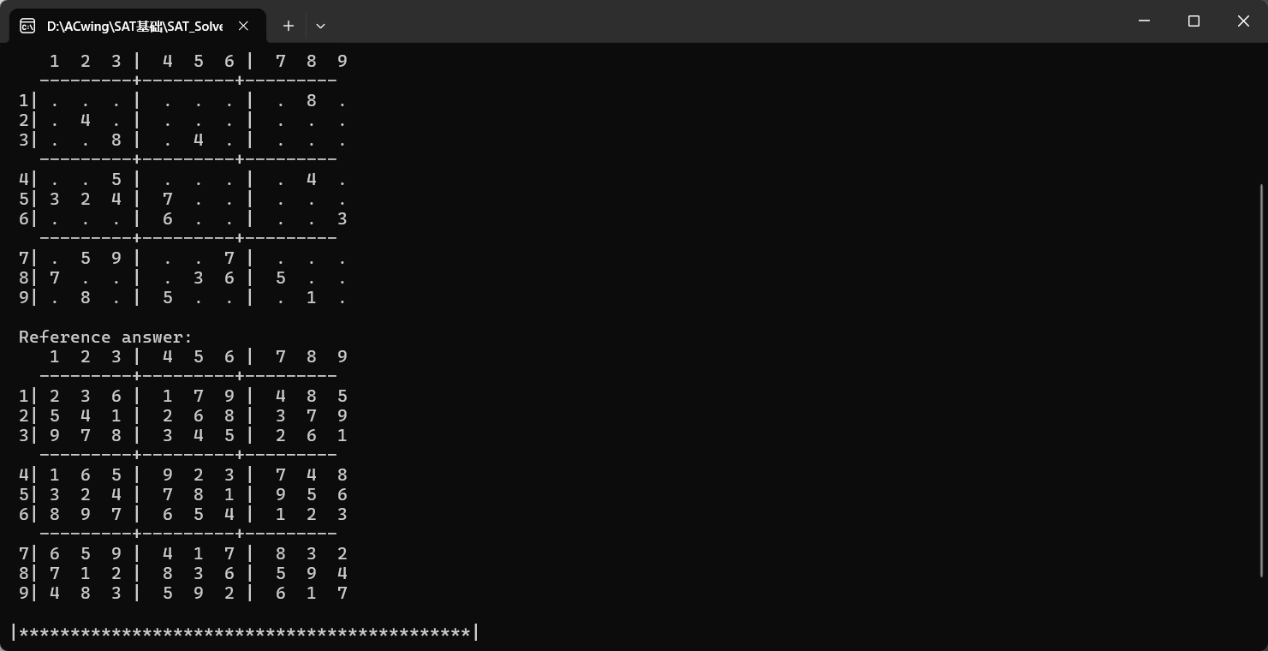


图 13 输出正确答案

（3）数独交互测试

**填入的位置已有数字：**

输入2 2 1

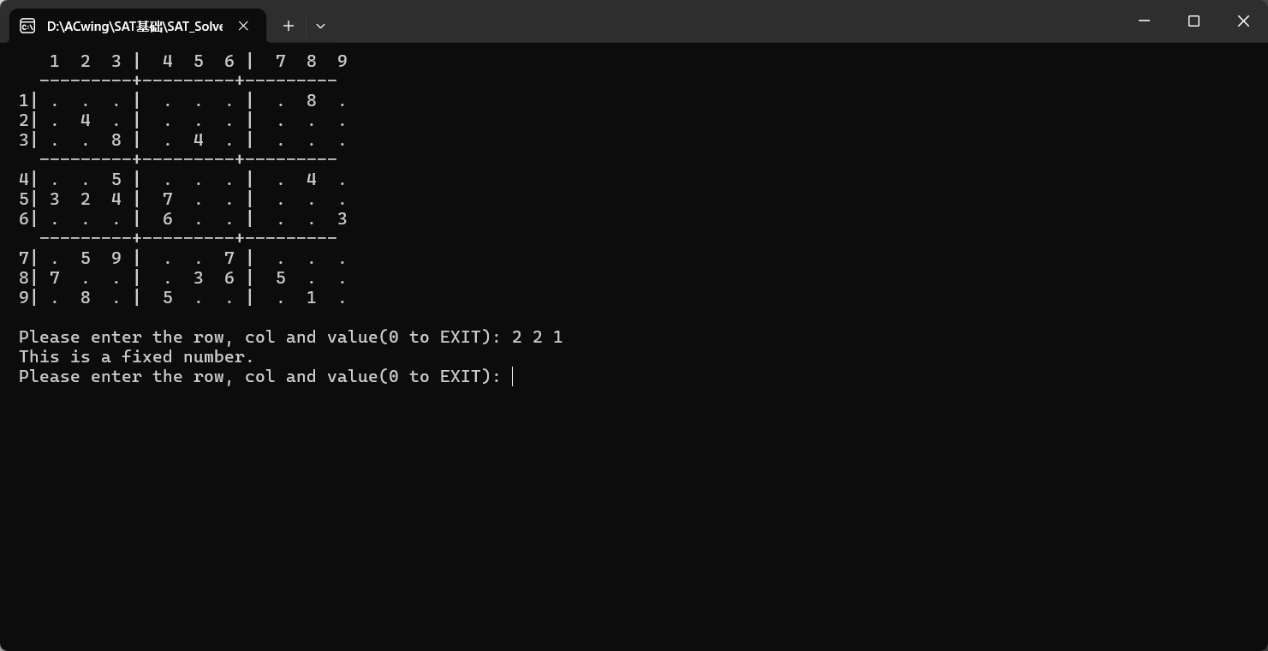


图 14 数独交互测试1

**填入数字违反规则：**

输入1 1 4

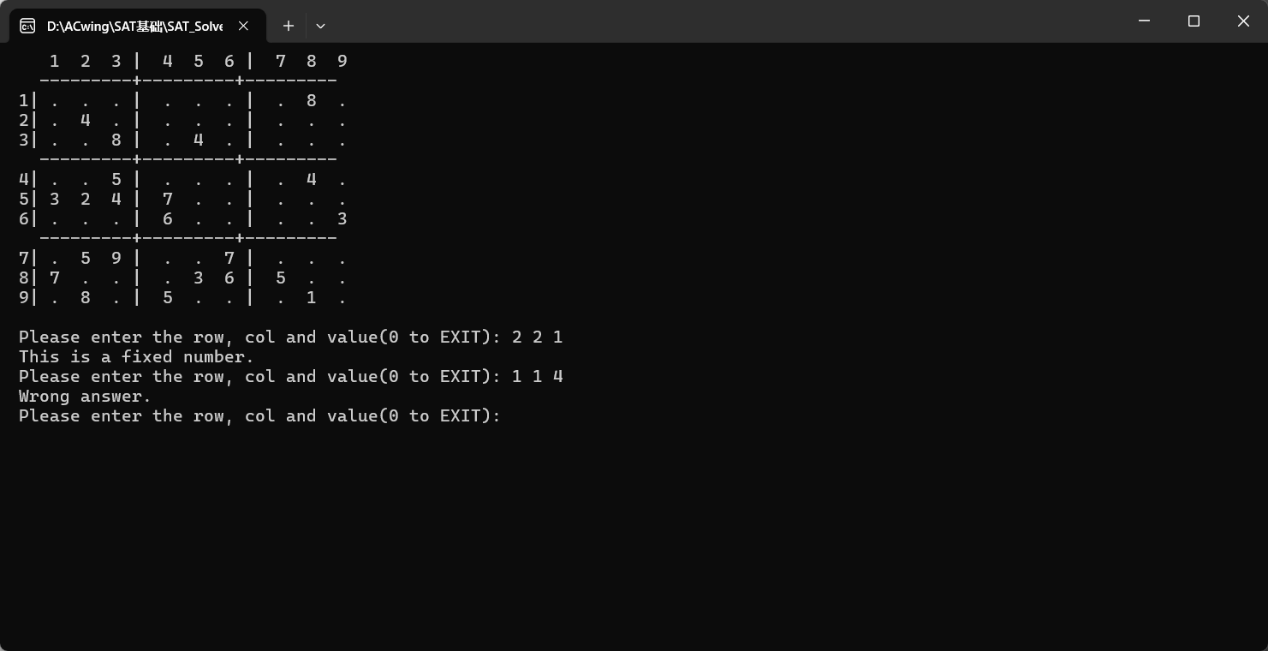


图 15 数独交互测试2

**填入正确数字：**

输入1 1 1

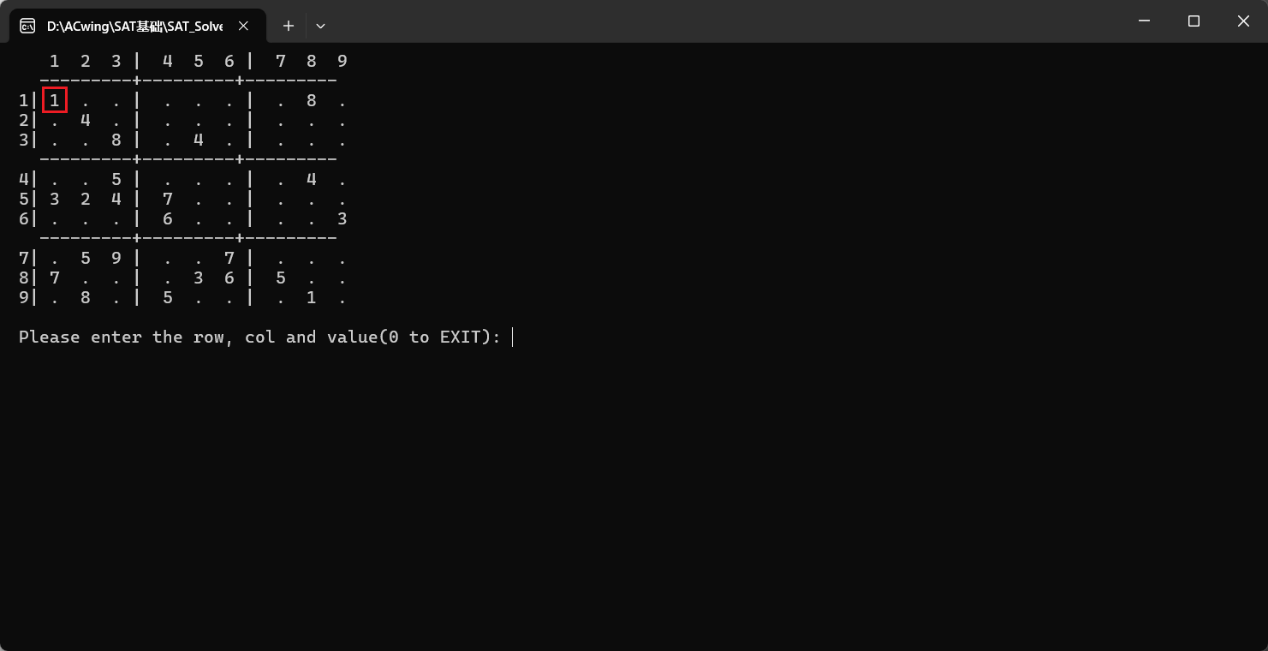


图 16 数独交互测试3

经测试，各类操作均在程序考虑范围之类，用户拥有良好的交互体验

**5总结与展望**

**5.1全文总结**

本次课程设计历时数周，通过对SAT（布尔可满足性问题）的深入学习和实践，我成功地设计并实现了一个基于DPLL算法的SAT求解器，并将其应用于解决百分号数独问题。在这次经历中，我不仅将《数据结构》和《C语言程序设计》的理论知识应用到实践中，还首次完成了对一个完整多文件项目的独立开发、调试与测试。我的主要工作总结如下：

（1）**DPLL算法的实现与优化**：我深入学习了DPLL算法的核心思想，包括单子句传播和分裂回溯 。在此基础上，我独立设计了程序的链式数据结构，并完整实现了递归求解的DPLL函数。特别地，我实现了多种分支启发式策略，并通过性能测试验证了优化的MOMs策略，相比朴素策略在求解复杂算例时具有压倒性的性能优势，优化率可达90%以上 。

（2）**百分号数独问题的SAT建模与转化**：我将SAT求解器作为核心引擎，探索了其在实际问题中的应用 。我详细分析了百分号数独的所有约束规则——包括行、列、九宫格、撇对角线以及两个特殊窗口——并设计实现了WriteToFile函数。该函数如同一个“编译器”，能将一个具体的数独棋盘布局及其所有规则，自动地、精确地转化为求解器可以处理的标准CNF范式文件 。

（3）**具备唯一解验证的数独生成器的设计与实现**：为了实现一个高质量的数独应用，我设计并实现了基于“生成终局后挖洞”的Generate\_Sudoku算法 。在此过程中，我发现并修正了原验证逻辑的缺陷，通过“求解-阻塞-再求解”的两阶段SAT方法，构建了HasUniqueSolution函数，使其能够严格保证最终生成的数独谜题具有**唯一解**。这不仅提升了数独应用本身的专业性，也加深了我对利用SAT求解器进行复杂属性验证的理解。

**5.2工作展望**

本次课程设计是我进入大学以来接触的第一个完整、复杂的多模块项目。它不仅是一次编程任务，更是一座桥梁，连接了课堂上的理论知识与软件开发的真实世界。在完成项目的过程中，我遇到了前所未有的挑战，也收获了宝贵的经验。基于这些经验总结，在今后的学习和研究中，我将围绕以下几个方面重点努力：

（1）**深化软件工程实践，提升项目构建与管理能力**：在本次项目中，我初次体验了从零开始搭建一个多文件系统。从最初面对头文件包含关系混乱、在不同模块间传递数据，到后来遭遇并解决棘手的“undefined reference”链接错误，我深刻认识到，一个成功的软件项目远不止是算法的实现。因此，我计划系统性地学习软件工程方法论，包括学习使用Makefile或CMake等构建系统来自动化编译过程，研究软件设计模式来编写更具可复用性和可扩展性的代码，并深入理解模块化与接口设计的思想，为未来参与更大型、更复杂的项目开发打下坚实的基础。

（2）**拓展算法视野，探索更广泛的问题求解范式**：通过亲手实现递归回溯的DPLL算法，我对“搜索”与“剪枝”这两个概念有了远超书本的直观理解。这激发了我对算法设计与分析的浓厚兴趣。未来，我将不仅仅满足于解决一个SAT问题，而是希望探索更广泛的问题求解范式。我期待在后续的“算法设计与分析”等课程中，深入学习**动态规划、贪心算法、网络流**等经典算法思想，并对NP完全理论有更深的理解，尝试去分析和解决其他有趣的组合优化问题，从而建立起一个更全面、更强大的算法知识体系。

（3）**融合面向对象思想，学习现代C++编程范式**：本项目主要采用了C风格的结构体和指针进行开发，虽然功能得以实现，但在数据封装和代码维护性上仍有提升空间。我注意到，面向对象编程（OOP）思想能够通过类（Class）更好地将数据与操作封装在一起，使系统结构更清晰。因此，我计划将本项目作为一个起点，用C++的类对其进行重构，并深入学习**继承、多态**等核心概念。同时，我将积极学习并实践现代C++（如C++11/17/20）的新特性，例如使用**智能指针**来避免内存泄漏，利用**STL标准模板库**来代替手动实现的数据结构，从而编写出更安全、更高效、更具现代风格的代码。

**参考文献**

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

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

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

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

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

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

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

[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] 薛源海，蒋彪彬，李永卓. 基于“挖洞”思想的数独游戏生成算法. 数学的实践与认识,2009,39(21):1-7

[11] 黄祖贤. 数独游戏的问题生成及求解算法优化. 安徽工业大学学报(自然科学版), 2015,32(2):187-191

[12]〔%数独〕百分号数独简介.

https://www.toutiao.com/article/7272556634214285859/?log\_from=1df6028568d0f\_1745483666191

[13] Percent Sudoku: Sudoku variants.

<http://forum.enjoysudoku.com/percent-sudoku-t38353.html#p296358>

**附录**

**SAT.hpp**

1. #pragma once
2. */\*头文件\*/*
3. #include <malloc.h>
4. #include <math.h>
5. #include <stdbool.h>
6. #include <stdio.h>
7. #include <stdlib.h>
8. #include <string.h>
9. #include <time.h>
10. #include <windows.h>
11. #include <winnt.h>
12. */\*常量\*/*
13. #define OK 1
14. #define ERROR 0
15. #define TRUE 1
16. #define FALSE 0
17. #define SIZE 9
18. extern long long cnt;
19. */\*状态码\*/*
20. typedef int status;
21. */\*文字节点&链表\*/*
22. typedef struct literalNode {
23. int literal;              *// 文字(变元)*
24. struct literalNode \*next; *// 指向下一个文字*
25. } literalNode, \*literalList;
26. */\*子句节点&链表\*/*
27. typedef struct clauseNode {
28. literalList head;        *// 指向子句中的第一个文字*
29. struct clauseNode \*next; *// 指向下一个子句*
30. } clauseNode, \*clauseList;
31. */\*CNF文件\*/*
32. typedef struct cnfNode {
33. clauseList root; *// 指向CNF的第一个子句*
34. int boolCount;   *// 布尔变元个数*
35. int clauseCount; *// 子句个数*
36. } cnfNode, \*CNF;
37. */\*函数声明\*/*
38. void DisPlay();                               *// 主交互界面*
39. void PrintMenu();                             *// 打印菜单*
40. status ReadFile(CNF &cnf, char fileName[]);   *// 读取文件并解析cnf*
41. status DestroyCnf(clauseList &cL);            *// 销毁当前解析的cnf*
42. status PrintCnf(CNF cnf);                     *// 打印cnf*
43. status DPLL(CNF cnf, bool value[], int flag); *// DPLL算法*
44. status IsUnitClause(literalList l);           *// 判断是否为单子句*
45. int FindUnitClause(clauseList cL);    *// 找到单子句并返回该文字*
46. status DestroyClause(clauseList &cL); *// 销毁子句*
47. int ChooseLiteral\_1(CNF cnf); *// 选择一个未赋值的文字(未优化)*
48. int ChooseLiteral\_2(CNF cnf); *// 选择一个未赋值的文字(优化)*
49. int ChooseLiteral\_3(CNF cnf); *// 改进2*
50. void Simplify(clauseList &cL, int literal); *// 根据选择的文字化简*
51. status Satisfy(clauseList cL);              *// 判断文字是否满足*
52. status EmptyClause(clauseList cL);          *// 判断是否有空子句*
53. clauseList CopyCnf(clauseList cL);          *// 复制cnf*
54. status SaveResult(int result, double time, double time\_, bool value[],
55. char fileName[], int boolCount); *// 保存求解结果*
56. void Sudoku();                                     *// X数独*
57. void PrintMenu\_X();                                *// 打印X数独菜单*
58. status Generate\_Sudoku(int board[SIZE + 1][SIZE + 1],
59. int newBoard[SIZE + 1][SIZE + 1],
60. int newBoard2[SIZE + 1][SIZE + 1],
61. bool isFixed[SIZE + 1][SIZE + 1], int num,
62. bool value[SIZE \* SIZE \* SIZE + 1]); *// 生成数独*
63. status Is\_Valid(int board[SIZE + 1][SIZE + 1], int row, int col,
64. int v); *// 判断board[row][col]是否可以填入*
65. void Print\_Sudoku(int board[SIZE + 1][SIZE + 1]); *// 打印数独*
66. void Play\_Sudoku(int board[SIZE + 1][SIZE + 1],
67. bool isFixed[SIZE + 1][SIZE + 1]); *// 玩数独*
68. status WriteToFile(int board[SIZE + 1][SIZE + 1], int num,
69. char name[]); *// 将数独约束条件写入文件*
70. status Slove(int board[SIZE + 1][SIZE + 1],
71. bool value[SIZE \* SIZE \* SIZE + 1]); *// 求解数独*
72. status Fill\_Box(int board[SIZE + 1][SIZE + 1], int newBoard[SIZE + 1][SIZE + 1],
73. int newBoard2[SIZE + 1][SIZE + 1], int rowStart,
74. int colStart);  *// 填充3x3的宫格*
75. void Shuffle(int arr[], int n); *// 打乱数组*
76. status Validate\_Sudoku(
77. int board[SIZE + 1][SIZE + 1]); *// 用于验证数独是否满足所有约束条件*
78. status
79. HasUniqueSolution(int board[SIZE + 1][SIZE + 1],
80. bool isFixed[SIZE + 1][SIZE + 1],
81. int solution[SIZE + 1][SIZE + 1]); *// 用于检查数独是否有唯一解*

**display.cpp**

1. */\*----------------------------display----------------------------\*/*
2. #include "SAT.hpp"
3. */\**
4. @ 函数名称: DisPlay
5. @ 函数功能: 交互界面
6. @ 返回值: void
7. \*/
8. void DisPlay() {
9. bool \*value = NULL; *// 存储文字的真值*
10. CNF cnf = (CNF)malloc(sizeof(cnfNode));
11. cnf->root = NULL;
12. char fileName[100]; *// 文件名*
13. PrintMenu();        *// 打印菜单*
14. int op = 1;
15. while (op) {
16. printf("\n|--------------------------------------------|\n");
17. printf("|--------Please Choose Your Operation--------|\n");
18. printf("|--------------------------------------------|\n\n");
19. printf("               Your choice: ");
20. scanf("%d", &op);
21. system("cls"); *// 每次进来清屏*
22. PrintMenu();   *// 打印菜单*
23. switch (op) {
24. case 1: {
25. if (cnf->root != NULL) *// 如果已经打开了CNF文件*
26. {
27. printf(" The CNF has been read.\n");
28. printf(" Do you want to read another? (1/0): ");
29. int choice;
30. scanf("%d", &choice);
31. if (choice == 0)
32. break;
33. else *// 重新读取*
34. {
35. cnt = 0;
36. DestroyCnf(cnf->root); *// 销毁当前解析的CNF*
37. }
38. }
39. printf(" Please input the file name: ");
40. scanf("%s", fileName);
41. if (ReadFile(cnf, fileName) == OK) *// 读取文件并解析CNF*
42. printf(" Read successfully.\n");
43. else
44. printf(" Read failed.\n");
45. break;
46. }
47. case 2: {
48. if (cnf->root == NULL) *// 如果没有打开CNF文件*
49. printf(" You haven't open the CNF file.\n");
50. else
51. PrintCnf(cnf); *// 打印CNF文件*
52. break;
53. }
54. *// case 3: {*
55. *//   if (cnf->root == NULL) // 如果没有打开CNF文件*
56. *//   {*
57. *//     printf(" You haven't open the CNF file.\n");*
58. *//     break;*
59. *//   } else {*
60. *//     CNF newCnf = (CNF)malloc(sizeof(cnfNode));*
61. *//     newCnf->root = CopyCnf(cnf->root); // 复制CNF*
62. *//     newCnf->boolCount = cnf->boolCount;*
63. *//     newCnf->clauseCount = cnf->clauseCount;*
64. *//     value = (bool \*)malloc(sizeof(bool) \* (cnf->boolCount + 1));*
65. *//     for (int i = 1; i <= cnf->boolCount; i++)*
66. *//       value[i] = FALSE;                     // 初始化为FALSE*
67. *//     LARGE\_INTEGER frequency, frequency\_;    // 计时器频率*
68. *//     LARGE\_INTEGER start, start\_, end, end\_; // 设置时间变量*
69. *//     double time = 0, time\_;*
70. *//     int result;*
71. *//     /\**
72. *//     // 未优化的时间*
73. *//     QueryPerformanceFrequency(&frequency);*
74. *//     QueryPerformanceCounter(&start); // 计时开始;*
75. *//     result = DPLL(newCnf, value, 1);*
76. *//     QueryPerformanceCounter(&end); //*
77. *//     time = (double)(end.QuadPart - start.QuadPart) /*
78. *//            frequency.QuadPart; // 计算运行时间*
79. *//     // 输出SAT结果*
80. *//     if (result == OK) // SAT*
81. *//     {*
82. *//       printf(" SAT\n\n");*
83. *//       // 输出文字的真值*
84. *//       for (int i = 1; i <= cnf->boolCount; i++) {*
85. *//         if (value[i] == TRUE)*
86. *//           printf(" %-4d: TRUE\n", i);*
87. *//         else*
88. *//           printf(" %-4d: FALSE\n", i);*
89. *//       }*
90. *//     } else // UNSAT*
91. *//       printf(" UNSAT\n");*
92. *//     // 输出优化前的时间*
93. *//     printf("\n Time: %lf ms(not optimized)\n", time \* 1000);*
94. *//     \*/*
95. *//     // 直接使用优化后的DPLL*
96. *//     int ch = 1; // 直接设置为1，表示使用优化*
97. *//     // 是否优化*
98. *//     // int ch;*
99. *//     printf("\n Do you want to optimize the algorithm? (1/0): ");*
100. *//     scanf("%d", &ch);*
101. *//     if (ch == 0) {*
102. *//       time\_ = 0;*
103. *//     } else {*
104. *//       // 重置value数组*
105. *//       for (int i = 1; i <= cnf->boolCount; i++)*
106. *//         value[i] = FALSE;*
107. *//       QueryPerformanceFrequency(&frequency\_);*
108. *//       QueryPerformanceCounter(&start\_); // 计时开始;*
109. *//       result = DPLL(newCnf, value, 3);  // 使用优化后的DPLL*
110. *//       QueryPerformanceCounter(&end\_);   // 结束*
111. *//       time\_ = (double)(end\_.QuadPart - start\_.QuadPart) /*
112. *//               frequency\_.QuadPart; // 计算运行时间*
113. *//       // 输出SAT结果*
114. *//       if (result == OK) // SAT*
115. *//       {*
116. *//         printf(" SAT\n\n");*
117. *//         // 输出文字的真值*
118. *//         for (int i = 1; i <= cnf->boolCount; i++) {*
119. *//           if (value[i] == TRUE)*
120. *//             printf(" %-4d: TRUE\n", i);*
121. *//           else*
122. *//             printf(" %-4d: FALSE\n", i);*
123. *//         }*
124. *//       } else // UNSAT*
125. *//         printf(" UNSAT\n");*
126. *//       printf("\n Time: %lf ms(optimized)\n", time\_ \* 1000);*
127. *//     }*
128. *//     // 是否保存*
129. *//     printf("\n Save the result to file? (1/0): ");*
130. *//     int choice;*
131. *//     scanf("%d", &choice);*
132. *//     printf("\n");*
133. *//     if (choice == 1) {*
134. *//       // 保存求解结果，将未优化时间设为0（因为我们只运行了优化版本）*
135. *//       if (SaveResult(result, 1, time\_, value, fileName, cnf->boolCount))*
136. *//         printf(" Save successfully.\n");*
137. *//       else*
138. *//         printf(" Save failed.\n");*
139. *//     }*
140. *//     // 释放内存*
141. *//     DestroyCnf(newCnf->root);*
142. *//     free(newCnf);*
143. *//   }*
144. *//   break;*
145. *// }*
146. *// display.cpp -> DisPlay() -> switch(op)*
147. case 3: {
148. if (cnf->root == NULL) *// 如果没有打开CNF文件*
149. {
150. printf(" You haven't open the CNF file.\n");
151. break;
152. } else {
153. *// --- 初始化 ---*
154. value = (bool \*)malloc(sizeof(bool) \* (cnf->boolCount + 1));
155. LARGE\_INTEGER frequency;    *// 计时器频率*
156. LARGE\_INTEGER start, end;   *// 时间变量*
157. double time\_unoptimized = 0, time\_optimized = 0;
158. int result\_unoptimized, result\_optimized;
159. printf("\n--- Running Unoptimized DPLL (Strategy 1) ---\n");
161. *// --- 1. 进行未优化的求解 ---*
162. *// 为未优化求解创建独立的CNF副本*
163. CNF cnf\_unoptimized = (CNF)malloc(sizeof(cnfNode));
164. cnf\_unoptimized->root = CopyCnf(cnf->root);
165. cnf\_unoptimized->boolCount = cnf->boolCount;
166. cnf\_unoptimized->clauseCount = cnf->clauseCount;
168. *// 初始化value数组*
169. for (int i = 1; i <= cnf->boolCount; i++) value[i] = FALSE;
170. QueryPerformanceFrequency(&frequency);
171. QueryPerformanceCounter(&start); *// 计时开始*
172. result\_unoptimized = DPLL(cnf\_unoptimized, value, 1); *// 使用 flag = 1*
173. QueryPerformanceCounter(&end); *// 计时结束*
174. time\_unoptimized = (double)(end.QuadPart - start.QuadPart) / frequency.QuadPart; *// 计算运行时间*
176. if (result\_unoptimized == OK) printf(" Result: SAT\n");
177. else printf(" Result: UNSAT\n");
178. printf(" Time: %lf ms\n", time\_unoptimized \* 1000);
180. DestroyCnf(cnf\_unoptimized->root); *// 销毁副本*
181. free(cnf\_unoptimized);
182. printf("\n--- Running Optimized DPLL (Strategy 3) ---\n");
183. *// --- 2. 进行优化后的求解 ---*
184. *// 为优化求解创建独立的CNF副本*
185. CNF cnf\_optimized = (CNF)malloc(sizeof(cnfNode));
186. cnf\_optimized->root = CopyCnf(cnf->root);
187. cnf\_optimized->boolCount = cnf->boolCount;
188. cnf\_optimized->clauseCount = cnf->clauseCount;
189. *// 重新初始化value数组*
190. for (int i = 1; i <= cnf->boolCount; i++) value[i] = FALSE;
191. QueryPerformanceFrequency(&frequency);
192. QueryPerformanceCounter(&start); *// 计时开始*
193. result\_optimized = DPLL(cnf\_optimized, value, 3); *// 使用 flag = 3*
194. QueryPerformanceCounter(&end); *// 计时结束*
195. time\_optimized = (double)(end.QuadPart - start.QuadPart) / frequency.QuadPart; *// 计算运行时间*
196. if (result\_optimized == OK) {
197. printf(" Result: SAT\n\n");
198. *// 打印优化后的解*
199. for (int i = 1; i <= cnf->boolCount; i++) {
200. if (value[i] == TRUE) printf(" %-4d: TRUE\n", i);
201. else printf(" %-4d: FALSE\n", i);
202. }
203. } else {
204. printf(" Result: UNSAT\n");
205. }
206. printf("\n Time: %lf ms\n", time\_optimized \* 1000);
207. DestroyCnf(cnf\_optimized->root); *// 销毁副本*
208. free(cnf\_optimized);
210. *// --- 3. 计算并输出优化率 ---*
211. printf("\n--- Comparison ---\n");
212. printf(" Unoptimized Time: %lf ms\n", time\_unoptimized \* 1000);
213. printf(" Optimized Time:   %lf ms\n", time\_optimized \* 1000);
214. if (time\_unoptimized > 0) {
215. double optimization\_rate = ((time\_unoptimized - time\_optimized) / time\_unoptimized) \* 100;
216. printf(" Optimization Rate: %.2lf%%\n", optimization\_rate);
217. } else {
218. printf(" Optimization Rate: N/A (unoptimized time is zero)\n");
219. }
220. *// --- 4. 保存结果 ---*
221. printf("\n Save the result to file? (1/0): ");
222. int choice;
223. scanf("%d", &choice);
224. printf("\n");
225. if (choice == 1) {
226. *// 保存优化后的求解结果，并传入两个时间用于记录*
227. if (SaveResult(result\_optimized, time\_unoptimized, time\_optimized, value, fileName, cnf->boolCount))
228. printf(" Save successfully.\n");
229. else
230. printf(" Save failed.\n");
231. }
232. *// 释放value数组*
233. free(value);
234. value = NULL;
235. }
236. break;
237. }
238. case 4: {
239. Sudoku();    *// X数独界面*
240. PrintMenu(); *// 跳转回来时重新打印菜单*
241. break;
242. }
243. case 0: {
244. printf(" Exit successfully.\n");
245. return; *// 退出*
246. }
247. default: {
248. printf(" Invalid input.\n"); *// 无效输入*
249. break;
250. }
251. }
252. }
253. if (cnf->root != NULL)
254. DestroyCnf(cnf->root); *// 退出时销毁CNF*
255. free(cnf);
256. return;
257. }
258. *// 打印菜单*
259. void PrintMenu() {
260. printf("|================Menu for SAT================|\n");
261. printf("|--------------------------------------------|\n");
262. printf("|            1. Open the CNF file            |\n");
263. printf("|     2. Traverse and output each clause     |\n");
264. printf("|   3. Solve using DPLL and save the result  |\n");
265. printf("|              4. Sudoku game              |\n");
266. printf("|                 0.  EXIT                   |\n");
267. printf("|============================================|\n\n");
268. }

**Cnfparser.cpp**

1. */\*----------------------------cnfparser----------------------------\*/*
2. #include "SAT.hpp"
3. *// 用文件指针fp打开用户指定的文件，并读取文件内容保存到给定参数中*
4. status ReadFile(CNF &cnf, char fileName[]) {
5. FILE \*fp = fopen(fileName, "r");
6. while (fp == NULL) {
7. printf(" File not found, please input again: ");
8. scanf("%s", fileName);
9. fp = fopen(fileName, "r");
10. }
11. char ch;
12. *// 跳过注释*
13. while ((ch = getc(fp)) == 'c') {
14. while ((ch = getc(fp)) != '\n')
15. continue;
16. }
17. *// 跳过p cnf*
18. getc(fp);
19. getc(fp);
20. getc(fp);
21. getc(fp);
22. fscanf(fp, "%d%d", &cnf->boolCount,
23. &cnf->clauseCount); *// 读取布尔变元个数和子句个数 // 初始化布尔变元个数*
24. cnf->root = NULL;             *// 初始化CNF*
25. clauseList lastClause = NULL; *// 用于记录上一个子句*
26. for (int i = 0; i < cnf->clauseCount; i++) {
27. *// 读取子句*
28. clauseList newClause = (clauseList)malloc(sizeof(clauseNode));
29. newClause->head = NULL;
30. newClause->next = NULL;
31. literalList lastLiteral = NULL; *// 用于记录上一个文字*
32. int number;                     *// 读取文字*
33. fscanf(fp, "%d", &number);
34. while (number != 0) {
35. literalList newLiteral = (literalList)malloc(sizeof(literalNode));
36. newLiteral->literal = number;
37. newLiteral->next = NULL;
38. if (newClause->head == NULL) *// 如果是第一个文字*
39. newClause->head = newLiteral;
40. else
41. lastLiteral->next = newLiteral;
42. lastLiteral = newLiteral; *// 更新上一个文字*
43. fscanf(fp, "%d", &number);
44. }
45. if (cnf->root == NULL) *// 如果是第一个子句*
46. cnf->root = newClause;
47. else
48. lastClause->next = newClause;
49. lastClause = newClause; *// 更新上一个子句*
50. }
51. fclose(fp);
52. return OK;
53. }
54. *// 销毁给定的CNF文件*
55. status DestroyCnf(clauseList &cL) {
56. while (cL != NULL) {
57. clauseList tempClause = cL;
58. cL = cL->next;                    *// 指向下一个子句*
59. literalList p = tempClause->head; *// 指向当前子句的第一个文字*
60. while (p != NULL) {
61. literalList tempLiteral = p;
62. p = p->next;
63. free(tempLiteral); *// 释放文字*
64. }
65. free(tempClause); *// 释放子句*
66. }
67. cL = NULL; *// cL指向NULL*
68. return OK;
69. }
70. *// 打印给定的CNF文件*
71. status PrintCnf(CNF cnf) {
72. clauseList p = cnf->root;
73. if (p == NULL) *// 如果没有子句*
74. {
75. printf(" No clauses.\n");
76. return ERROR;
77. }
78. printf("  The CNF is:\n");
79. printf("  boolCount:%d\n", cnf->boolCount);     *// 打印布尔变元个数*
80. printf("  clauseCount:%d\n", cnf->clauseCount); *// 打印子句个数*
81. while (p) {
82. literalList q = p->head; *// 指向子句的第一个文字*
83. printf("  ");
84. while (q) {
85. printf("%-5d", q->literal); *// 打印文字*
86. q = q->next;
87. }
88. printf("0\n"); *// 子句结束*
89. p = p->next;   *// 指向下一个子句*
90. }
91. return OK;
92. }

**solver.cpp**

1. #include "SAT.hpp"
2. long long cnt = 0;
3. *// 判断是否为单子句*
4. status IsUnitClause(literalList l) {
5. if (l != NULL && l->next == NULL) *// 只有一个文字*
6. return TRUE;
7. else
8. return FALSE;
9. }
10. *// 找到单子句并返回该文字*
11. int FindUnitClause(clauseList cL) {
12. clauseList p = cL;
13. while (p) {
14. if (IsUnitClause(p->head)) *// 是单子句*
15. return p->head->literal; *// 返回该文字*
16. p = p->next;
17. }
18. return 0;
19. }
20. *// 销毁子句*
21. status DestroyClause(clauseList &cL) {
22. literalList p = cL->head;
23. while (p) {
24. literalList temp = p;
25. p = p->next; *// 指向下一个文字*
26. free(temp);  *// 释放文字*
27. }
28. free(cL);  *// 释放子句*
29. cL = NULL; *// cL指向NULL*
30. return OK;
31. }
32. *// 根据选择的文字化简*
33. void Simplify(clauseList &cL, int literal) {
34. clauseList pre = NULL, p = cL; *// pre指向前一个子句*
35. while (p != NULL) {
36. bool clauseDeleted = false;           *// 是否删除子句*
37. literalList lpre = NULL, q = p->head; *// lpre指向前一个文字*
38. while (q != NULL) {
39. if (q->literal == literal) *// 删除该子句*
40. {
41. if (pre == NULL) *// 删除的是第一个子句*
42. cL = p->next;
43. else *// 删除的不是第一个子句*
44. pre->next = p->next;
45. DestroyClause(p);                   *// 销毁该子句*
46. p = (pre == NULL) ? cL : pre->next; *// 指向下一个子句*
47. clauseDeleted = true;               *// 子句已删除*
48. break;
49. } else if (q->literal == -literal) *// 删除该文字*
50. {
51. if (lpre == NULL) *// 删除的是第一个文字*
52. p->head = q->next;
53. else *// 删除的不是第一个文字*
54. lpre->next = q->next;
55. free(q);                                   *// 释放该文字*
56. q = (lpre == NULL) ? p->head : lpre->next; *// 指向下一个文字*
57. } else                                       *// 未删除*
58. {
59. lpre = q;
60. q = q->next;
61. }
62. }
63. if (!clauseDeleted) *// 子句未删除*
64. {
65. pre = p;
66. p = p->next;
67. }
68. }
69. }
70. *// 复制cnf*
71. clauseList CopyCnf(clauseList cL) {
72. *// 初始化新的CNF*
73. clauseList newCnf = (clauseList)malloc(sizeof(clauseNode));
74. clauseList lpa, lpb;  *// lpa指向新的子句,lpb指向旧的子句*
75. literalList tpa, tpb; *// tpa指向新的文字,tpb指向旧的文字*
76. newCnf->head = (literalList)malloc(sizeof(literalNode));
77. newCnf->next = NULL;
78. newCnf->head->next = NULL;
79. for (lpb = cL, lpa = newCnf; lpb != NULL; lpb = lpb->next, lpa = lpa->next) {
80. for (tpb = lpb->head, tpa = lpa->head; tpb != NULL;
81. tpb = tpb->next, tpa = tpa->next) {
82. tpa->literal = tpb->literal;
83. tpa->next = (literalList)malloc(sizeof(literalNode));
84. tpa->next->next = NULL;
85. if (tpb->next == NULL) *// 旧的子句中的文字已经复制完*
86. {
87. free(tpa->next);
88. tpa->next = NULL;
89. }
90. }
91. lpa->next = (clauseList)malloc(sizeof(clauseNode));
92. lpa->next->head = (literalList)malloc(sizeof(literalNode));
93. lpa->next->next = NULL;
94. lpa->next->head->next = NULL;
95. if (lpb->next == NULL) *// 旧的CNF中的子句已经复制完*
96. {
97. free(lpa->next->head);
98. free(lpa->next);
99. lpa->next = NULL;
100. }
101. }
102. return newCnf;
103. }
104. *// 选择文字(第一个文字)*
105. int ChooseLiteral\_1(CNF cnf) { return cnf->root->head->literal; }
106. *//(没有单子句时的策略)选择文字(出现次数最多的文字)*
107. int ChooseLiteral\_2(CNF cnf) {
108. clauseList lp = cnf->root;
109. literalList dp;
110. int \*count, MaxWord,
111. max; *// count记录每个文字出现次数,MaxWord记录出现最多次数的文字*
112. count = (int \*)malloc(sizeof(int) \* (cnf->boolCount \* 2 + 1));
113. for (int i = 0; i <= cnf->boolCount \* 2; i++)
114. count[i] = 0; *// 初始化*
115. *// 计算子句中各文字出现次数*
116. for (lp = cnf->root; lp != NULL; lp = lp->next) {
117. for (dp = lp->head; dp != NULL; dp = dp->next) {
118. if (dp->literal > 0) *// 正文字*
119. count[dp->literal]++;
120. else
121. count[cnf->boolCount - dp->literal]++; *// 负文字*
122. }
123. }
124. max = 0;
125. *// 找到出现次数最多的正文字*
126. for (int i = 1; i <= cnf->boolCount; i++) {
127. if (max < count[i]) {
128. max = count[i];
129. MaxWord = i;
130. }
131. }
132. if (max == 0) {
133. *// 若没有出现正文字,找到出现次数最多的负文字*
134. for (int i = cnf->boolCount + 1; i <= cnf->boolCount \* 2; i++) {
135. if (max < count[i]) {
136. max = count[i];
137. MaxWord = cnf->boolCount - i;
138. }
139. }
140. }
141. free(count);
142. return MaxWord;
143. }
144. *// 选择最短子句中出现次数最多的文字*
145. int ChooseLiteral\_3(CNF cnf) {
146. clauseList p = cnf->root;
147. int \*count = (int \*)calloc(cnf->boolCount \* 2 + 1, sizeof(int));
148. int minSize = INT\_MAX; *// 初始化为大于可能的最大子句长度*
149. int literal = 0;
150. clauseList temp = NULL;
151. *// 遍历子句，找到最小子句并统计其文字*
152. while (p != NULL) {
153. literalList q = p->head;
154. int clauseSize = 0;
155. while (q != NULL) {
156. clauseSize++;
157. q = q->next;
158. }
159. if (clauseSize < minSize) {
160. minSize = clauseSize; *// 更新最小子句大小*
161. temp = p;
162. }
163. p = p->next;
164. }
165. *// 遍历子句，统计最小子句中各文字出现次数*
166. literalList q = temp->head;
167. while (q != NULL) {
168. count[q->literal + cnf->boolCount]++;
169. q = q->next;
170. }
171. *// 找到最频繁的文字*
172. int maxCount = 0;
173. for (int i = 0; i < cnf->boolCount \* 2 + 1; i++) {
174. if (count[i] > maxCount) {
175. maxCount = count[i];
176. literal = i - cnf->boolCount;
177. }
178. }
179. free(count);
180. return literal;
181. }
182. *// 是否满足*
183. status Satisfy(clauseList cL) {
184. if (cL == NULL)
185. return OK;
186. else
187. return ERROR;
188. }
189. *// 是否有空子句*
190. status EmptyClause(clauseList cL) {
191. clauseList p = cL;
192. while (p) {
193. if (p->head == NULL) *// 空子句，返回UNSAT*
194. return TRUE;
195. p = p->next;
196. }
197. return FALSE;
198. }
199. *// DPLL算法求解SAT问题*
200. status DPLL(CNF cnf, bool value[], int flag) {
201. */\*1.单子句规则\*/*
202. int unitLiteral = FindUnitClause(cnf->root); *// 找单子句*
203. while (unitLiteral != 0) {
204. cnt++;
205. if (cnt % 10000 == 0 && cnt != 0)
206. printf("Round:%d\n", cnt);
207. value[abs(unitLiteral)] = (unitLiteral > 0) ? TRUE : FALSE;
208. Simplify(cnf->root, unitLiteral); *// 删句子(true)或者删文字(false)*
209. *// 终止条件*
210. if (Satisfy(cnf->root) == OK)
211. return OK;
212. if (EmptyClause(cnf->root) == TRUE)
213. return ERROR;
214. unitLiteral = FindUnitClause(cnf->root);
215. }
216. */\*2.选择一个未赋值的文字\*/*
217. int literal;
218. if (flag == 1)
219. literal = ChooseLiteral\_1(cnf); *// 未优化*
220. else if (flag == 2)
221. literal = ChooseLiteral\_2(cnf); *// 优化*
222. else
223. literal = ChooseLiteral\_3(cnf);
224. */\*3.将该文字赋值为真，递归求解\*/*
225. CNF newCnf = (CNF)malloc(sizeof(cnfNode));
226. newCnf->root = CopyCnf(cnf->root); *// 复制CNF*
227. newCnf->boolCount = cnf->boolCount;
228. newCnf->clauseCount = cnf->clauseCount;
229. clauseList p = (clauseList)malloc(sizeof(clauseNode));
230. p->head = (literalList)malloc(sizeof(literalNode));
231. p->head->literal = literal;
232. p->head->next = NULL;
233. p->next = newCnf->root;
234. newCnf->root = p; *// 插入到表头*
235. if (DPLL(newCnf, value, flag) == 1)
236. return 1; *// 在第一分支中搜索*
237. DestroyCnf(newCnf->root);
238. */\*4.将该文字赋值为假，递归求解\*/*
239. clauseList q = (clauseList)malloc(sizeof(clauseNode));
240. q->head = (literalList)malloc(sizeof(literalNode));
241. q->head->literal = -literal;
242. q->head->next = NULL;
243. q->next = cnf->root;
244. cnf->root = q;                      *// 插入到表头*
245. status re = DPLL(cnf, value, flag); *// 回溯到执行分支策略的初态进入另一分支*
246. *// DestroyCnf(cL);*
247. return re;
248. }
249. *// 保存求解结果*
250. *// status SaveResult(int result, double time, double time\_, bool value[],*
251. *//                   char fileName[], int boolCount) {*
252. *//   FILE \*fp;*
253. *//   char name[100];*
254. *//   for (int i = 0; fileName[i] != '\0'; i++) {*
255. *//     // 修改拓展名.res*
256. *//     if (fileName[i] == '.' && fileName[i + 4] == '\0') {*
257. *//       name[i] = '.';*
258. *//       name[i + 1] = 'r';*
259. *//       name[i + 2] = 'e';*
260. *//       name[i + 3] = 's';*
261. *//       name[i + 4] = '\0';*
262. *//       break;*
263. *//     }*
264. *//     name[i] = fileName[i];*
265. *//   }*
266. *//   if (fopen\_s(&fp, name, "w")) {*
267. *//     printf(" Fail!\n");*
268. *//     return ERROR;*
269. *//   }*
270. *//   // 按照要求格式输出*
271. *//   fprintf(fp, "s %d\n", result); // 求解结果*
272. *//   if (result == 1) {*
273. *//     fprintf(fp, "v");*
274. *//     // 确保输出所有变元的取值*
275. *//     for (int i = 1; i <= boolCount; i++) {*
276. *//       // 检查变元是否被赋值*
277. *//       if (value[i] == TRUE || value[i] == FALSE) {*
278. *//         if (value[i] == TRUE)*
279. *//           fprintf(fp, " %d", i);*
280. *//         else*
281. *//           fprintf(fp, " %d", -i);*
282. *//       } else {*
283. *//         // 如果变元没有被赋值，输出一个默认值（例如 TRUE）*
284. *//         fprintf(fp, " %d", i);*
285. *//       }*
286. *//     }*
287. *//     fprintf(fp, "\n");*
288. *//   }*
289. *//   // 输出优化后的执行时间（毫秒）*
290. *//   fprintf(fp, "t %.0f\n", time\_ \* 1000);*
291. *//   // 保留未优化时间和优化率信息（作为注释）*
292. *//   if (time != 0) {*
293. *//     fprintf(fp, "c 未优化时间: %.0fms\n", time \* 1000);*
294. *//     double optimization\_rate = ((time - time\_) / time) \* 100;*
295. *//     fprintf(fp, "c 优化率: %.2lf%%\n", optimization\_rate);*
296. *//   }*
297. *//   fclose(fp);*
298. *//   return OK;*
299. *// }*
300. status SaveResult(int result, double time\_unoptimized, double time\_optimized,
301. bool value[], char fileName[], int boolCount) {
302. FILE \*fp;
303. char name[100];
304. int len = strlen(fileName);
305. *// 健壮地生成.res文件名，替换.cnf后缀*
306. strcpy\_s(name, sizeof(name), fileName);
307. if (len > 4 && strcmp(name + len - 4, ".cnf") == 0) {
308. name[len - 3] = 'r';
309. name[len - 2] = 'e';
310. name[len - 1] = 's';
311. } else {
312. *// 如果原始文件名不以.cnf结尾，则直接追加.res*
313. strcat\_s(name, sizeof(name), ".res");
314. }
315. *// 使用fopen\_s安全地打开文件*
316. if (fopen\_s(&fp, name, "w")) {
317. printf(" Fail to open result file!\n");
318. return ERROR;
319. }
320. *// 1. 写入求解结果 (s 1 or s 0)，1代表可满足，0代表不可满足 [cite: 151, 152]*
321. fprintf(fp, "s %d\n", result);
322. *// 2. 如果可满足，写入每个变元的赋值序列 (v ...) [cite: 151, 153]*
323. if (result == OK) {
324. fprintf(fp, "v");
325. for (int i = 1; i <= boolCount; i++) {
326. *// 根据value数组的真值，输出正整数或负整数*
327. if (value[i] == TRUE) {
328. fprintf(fp, " %d", i);
329. } else {
330. fprintf(fp, " %d", -i);
331. }
332. }
333. fprintf(fp, " 0\n"); *// 按照惯例，以0作为赋值行的结束标志*
334. }
335. *// 3. 写入优化后的求解时间，以毫秒为单位 (t ...) [cite: 151, 154]*
336. fprintf(fp, "t %.0f\n", time\_optimized \* 1000);
337. *// 4. 以注释形式写入性能对比的详细信息，便于分析*
338. fprintf(fp, "c === Performance Analysis ===\n");
339. fprintf(fp, "c Unoptimized Time: %.3f ms\n", time\_unoptimized \* 1000);
340. fprintf(fp, "c Optimized Time:   %.3f ms\n", time\_optimized \* 1000);
342. *// 增加一个检查，防止未优化时间为0时出现除零错误*
343. if (time\_unoptimized > 1e-9) {
344. double optimization\_rate = ((time\_unoptimized - time\_optimized) / time\_unoptimized) \* 100;
345. fprintf(fp, "c Optimization Rate: %.2f%%\n", optimization\_rate);
346. } else {
347. fprintf(fp, "c Optimization Rate: N/A (Unoptimized time is zero)\n");
348. }
349. fclose(fp);
350. return OK;
351. }

**%-Sudoku.cpp**

#include "SAT.hpp"

1. void Sudoku() {
2. system("cls");
3. PrintMenu\_X();
4. int num;                            *// 提示数的个数*
5. bool isFixed[SIZE + 1][SIZE + 1];   *// 记录是否为提示数字*
6. int board[SIZE + 1][SIZE + 1];      *// 生成的初始数独*
7. int newBoard[SIZE + 1][SIZE + 1];   *// 用来玩的数独*
8. int newBoard2[SIZE + 1][SIZE + 1];  *// 保存答案的数独*
9. bool value[SIZE \* SIZE \* SIZE + 1]; *// 记录DPLL的结果*
10. for (int i = 1; i <= SIZE \* SIZE \* SIZE; i++)
11. value[i] = FALSE;
12. int op = 1;   *// 操作*
13. int flag = 0; *// 是否生成数独*
14. while (op) {
15. printf("\n|\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*|\n");
16. printf("|--------Please Choose Your Operation--------|\n");
17. printf("|\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*|\n\n");
18. printf("               Your choice: ");
19. scanf("%d", &op);
20. system("cls");
21. PrintMenu\_X();
22. switch (op) {
23. case 1: {
24. printf(" Please enter the number of prompts(>=17): ");
25. scanf("%d", &num);
26. while (num < 17 || num > 81) *// 提示数的个数必须大于等于25小于等于81*
27. {
28. printf(" Invalid input, please enter again: ");
29. scanf("%d", &num);
30. }
31. if (Generate\_Sudoku(board, newBoard, newBoard2, isFixed, num, value)) {
32. printf(" Generate successfully.\n");
33. flag = 1; *// 生成成功*
34. } else
35. printf(" Generate failed.\n");
36. break;
37. }
38. case 2: {
39. if (flag) {
40. Play\_Sudoku(newBoard, isFixed);
41. PrintMenu\_X(); *// 每次玩完跳转回来重新打印菜单*
42. } else
43. printf(" Please generate the Sudoku first.\n");
44. break;
45. }
46. case 3: {
47. if (flag) {
48. printf(" Original Sudoku:\n");
49. Print\_Sudoku(board); *// 打印原始数独*
50. printf("\n");
51. if (Slove(newBoard2, value)) *// 求解数独*
52. {
53. printf(" Reference answer:\n");
54. Print\_Sudoku(newBoard2); *// 打印答案*
55. } else
56. printf(" No answer.\n"); *// 无解*
57. } else
58. printf(" Please generate the Sudoku first.\n");
59. break;
60. }
61. case 0: {
62. system("cls"); *// 退出时清屏*
63. break;
64. }
65. default: {
66. printf(" Invalid input.\n");
67. break;
68. }
69. }
70. }
71. }
72. *// 打印X数独菜单*
73. void PrintMenu\_X() {
74. printf("|\*\*\*\*\*\*\*\*\*\*\*\*\*\*Menu for X-Sudoku(Percent Sudoku)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*|\n");
75. printf("|--------------------------------------------------------------|\n");
76. printf("|              1. Generate a X-Sudoku                         |\n");
77. printf("|              2. Play the X-Sudoku                           |\n");
78. printf("|              3. Reference answer                            |\n");
79. printf("|                    0. EXIT                                  |\n");
80. printf(
81. "|\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*|\n\n");
82. }
83. *// 生成数独*
84. status Generate\_Sudoku(int board[SIZE + 1][SIZE + 1],
85. int newBoard[SIZE + 1][SIZE + 1],
86. int newBoard2[SIZE + 1][SIZE + 1],
87. bool isFixed[SIZE + 1][SIZE + 1], int num,
88. bool value[SIZE \* SIZE \* SIZE + 1]) {
89. char name[100] = "Sudoku.cnf"; *// 文件名*
90. START:
91. srand(time(NULL));
92. *// 初始化棋盘*
93. for (int i = 1; i <= SIZE; i++)
94. for (int j = 1; j <= SIZE; j++) {
95. board[i][j] = 0;
96. newBoard[i][j] = 0;
97. newBoard2[i][j] = 0;
98. isFixed[i][j] = FALSE; *// 初始化为非固定数字*
99. }
100. *// 先填充两个窗口*
101. int windows[2][2] = {{2, 2}, {6, 6}};
102. for (int w = 0; w < 2; w++) {
103. int startRow = windows[w][0];
104. int startCol = windows[w][1];
105. Fill\_Box(board, newBoard, newBoard2, startRow, startCol);
106. }
107. WriteToFile(board, 27, name); *// 将数独约束条件写入文件*
108. CNF p = (CNF)malloc(sizeof(cnfNode));
109. p->root = NULL;
110. if (ReadFile(p, name) != OK) {
111. printf("Failed to read CNF file\n");
112. return ERROR;
113. }
114. for (int i = 1; i <= SIZE \* SIZE \* SIZE; i++)
115. value[i] = FALSE;
116. if (DPLL(p, value, 3) == ERROR) { *// 求解数独*
117. printf("DPLL failed, regenerating...\n");
118. DestroyCnf(p->root);
119. free(p);
120. goto START;
121. }
122. *// 将DPLL的结果填入数独*
123. for (int i = 1; i <= SIZE \* SIZE \* SIZE; i++) {
124. if (value[i] == TRUE) {
125. int row = (i - 1) / (SIZE \* SIZE) + 1;
126. int col = (i - 1) / SIZE % SIZE + 1;
127. int v = (i - 1) % SIZE + 1;
128. board[row][col] = v;
129. newBoard[row][col] = v;
130. newBoard2[row][col] = v;
131. }
132. }
133. *// 验证数独是否满足所有约束*
134. if (!Validate\_Sudoku(board)) {
135. printf(
136. "Generated Sudoku doesn't satisfy all constraints, regenerating...\n");
137. DestroyCnf(p->root);
138. free(p);
139. goto START;
140. }
141. *// 标记所有格子为固定（提示数）*
142. for (int i = 1; i <= SIZE; i++) {
143. for (int j = 1; j <= SIZE; j++) {
144. isFixed[i][j] = TRUE;
145. }
146. }
147. *// 挖洞,剩下num个提示数*
148. int remove = 81 - num;
149. int attempts = 0;
150. int maxAttempts = 1000; *// 防止无限循环*
151. while (remove > 0 && attempts < maxAttempts) {
152. int row = rand() % SIZE + 1;
153. int col = rand() % SIZE + 1;
154. *// 检查是否在窗口中*
155. bool inWindow = false;
156. for (int w = 0; w < 2; w++) {
157. int startRow = windows[w][0];
158. int startCol = windows[w][1];
159. if (row >= startRow && row < startRow + 3 && col >= startCol &&
160. col < startCol + 3) {
161. inWindow = true;
162. break;
163. }
164. }
165. *// 确保每个窗口至少保留3个数字*
166. if (inWindow) {
167. int windowCount = 0;
168. for (int w = 0; w < 2; w++) {
169. int startRow = windows[w][0];
170. int startCol = windows[w][1];
171. if (row >= startRow && row < startRow + 3 && col >= startCol &&
172. col < startCol + 3) {
173. *// 计算当前窗口中的提示数数量*
174. windowCount = 0;
175. for (int i = startRow; i < startRow + 3; i++) {
176. for (int j = startCol; j < startCol + 3; j++) {
177. if (isFixed[i][j])
178. windowCount++;
179. }
180. }
181. break;
182. }
183. }
184. *// 如果窗口中的提示数已经少于3个，不再挖洞*
185. if (windowCount <= 3) {
186. attempts++;
187. continue;
188. }
189. }
190. *// 挖洞*
191. if (isFixed[row][col]) {
192. int temp = board[row][col];
193. board[row][col] = 0;
194. newBoard[row][col] = 0;
195. isFixed[row][col] = FALSE;
196. *// 检查挖洞后是否仍有唯一解*
197. if (!HasUniqueSolution(board, isFixed, newBoard2)) {
198. *// 如果没有唯一解，恢复这个洞*
199. board[row][col] = temp;
200. newBoard[row][col] = temp;
201. isFixed[row][col] = TRUE;
202. } else {
203. remove--;
204. }
205. }
206. attempts++;
207. }
208. *// 如果还有洞需要挖，但已达到最大尝试次数*
209. if (remove > 0) {
210. printf(" Warning: Could not remove all requested holes while maintaining "
211. "window constraints.\n");
212. }
213. DestroyCnf(p->root);
214. free(p);
215. return OK;
216. }
217. *// 验证数独是否满足所有约束*
218. status Validate\_Sudoku(int board[SIZE + 1][SIZE + 1]) {
219. *// 检查行*
220. for (int i = 1; i <= SIZE; i++) {
221. bool used[SIZE + 1] = {false};
222. for (int j = 1; j <= SIZE; j++) {
223. if (board[i][j] != 0) {
224. if (used[board[i][j]])
225. return FALSE;
226. used[board[i][j]] = true;
227. }
228. }
229. }
230. *// 检查列*
231. for (int j = 1; j <= SIZE; j++) {
232. bool used[SIZE + 1] = {false};
233. for (int i = 1; i <= SIZE; i++) {
234. if (board[i][j] != 0) {
235. if (used[board[i][j]])
236. return FALSE;
237. used[board[i][j]] = true;
238. }
239. }
240. }
241. *// 检查3x3宫格*
242. for (int boxRow = 1; boxRow <= SIZE; boxRow += 3) {
243. for (int boxCol = 1; boxCol <= SIZE; boxCol += 3) {
244. bool used[SIZE + 1] = {false};
245. for (int i = boxRow; i < boxRow + 3; i++) {
246. for (int j = boxCol; j < boxCol + 3; j++) {
247. if (board[i][j] != 0) {
248. if (used[board[i][j]])
249. return FALSE;
250. used[board[i][j]] = true;
251. }
252. }
253. }
254. }
255. }
256. *// 检查副对角线*
257. bool usedDiag[SIZE + 1] = {false};
258. for (int i = 1; i <= SIZE; i++) {
259. int j = SIZE + 1 - i;
260. if (board[i][j] != 0) {
261. if (usedDiag[board[i][j]])
262. return FALSE;
263. usedDiag[board[i][j]] = true;
264. }
265. }
266. *// 检查两个窗口*
267. int windows[2][2] = {{2, 2}, {6, 6}};
268. for (int w = 0; w < 2; w++) {
269. int startRow = windows[w][0];
270. int startCol = windows[w][1];
271. bool used[SIZE + 1] = {false};
272. for (int i = startRow; i < startRow + 3; i++) {
273. for (int j = startCol; j < startCol + 3; j++) {
274. if (board[i][j] != 0) {
275. if (used[board[i][j]])
276. return FALSE;
277. used[board[i][j]] = true;
278. }
279. }
280. }
281. }
282. return TRUE;
283. }
284. *// // 检查数独是否有唯一解*
285. *// status HasUniqueSolution(int board[SIZE + 1][SIZE + 1],*
286. *//                          bool isFixed[SIZE + 1][SIZE + 1],*
287. *//                          int solution[SIZE + 1][SIZE + 1]) {*
288. *//   // 创建临时数独副本*
289. *//   int tempBoard[SIZE + 1][SIZE + 1];*
290. *//   for (int i = 1; i <= SIZE; i++) {*
291. *//     for (int j = 1; j <= SIZE; j++) {*
292. *//       tempBoard[i][j] = board[i][j];*
293. *//     }*
294. *//   }*
295. *//   // 使用DPLL求解数独*
296. *//   char name[100] = "temp\_Sudoku.cnf";*
297. *//   WriteToFile(tempBoard, 0, name);*
298. *//   CNF p = (CNF)malloc(sizeof(cnfNode));*
299. *//   p->root = NULL;*
300. *//   if (ReadFile(p, name) != OK) {*
301. *//     printf("Failed to read temporary CNF file\n");*
302. *//     return FALSE;*
303. *//   }*
304. *//   bool value[SIZE \* SIZE \* SIZE + 1];*
305. *//   for (int i = 1; i <= SIZE \* SIZE \* SIZE; i++)*
306. *//     value[i] = FALSE;*
307. *//   status result = DPLL(p, value, 3);*
308. *//   if (result == OK) {*
309. *//     // 将解填入solution数组*
310. *//     for (int i = 1; i <= SIZE \* SIZE \* SIZE; i++) {*
311. *//       if (value[i] == TRUE) {*
312. *//         int row = (i - 1) / (SIZE \* SIZE) + 1;*
313. *//         int col = (i - 1) / SIZE % SIZE + 1;*
314. *//         int v = (i - 1) % SIZE + 1;*
315. *//         solution[row][col] = v;*
316. *//       }*
317. *//     }*
318. *//   }*
319. *//   DestroyCnf(p->root);*
320. *//   free(p);*
321. *//   remove(name); // 删除临时文件*
322. *//   return result == OK;*
323. *// }*
324. *// [%-Sudoku.cpp]*
325. *// 请用这个新版本替换原有的 HasUniqueSolution 函数*
326. status HasUniqueSolution(int board[SIZE + 1][SIZE + 1],
327. bool isFixed[SIZE + 1][SIZE + 1],
328. int solution[SIZE + 1][SIZE + 1]) {
329. char name[100] = "temp\_Sudoku\_uniqueness\_check.cnf";
330. bool firstSolution[SIZE \* SIZE \* SIZE + 1];
331. status finalResult = FALSE; *// 最终返回结果，默认为非唯一解*
332. *// ==================== 阶段一：第一次求解，寻找第一个解 ====================*
333. CNF cnf1 = (CNF)malloc(sizeof(cnfNode));
334. cnf1->root = NULL;
335. bool value1[SIZE \* SIZE \* SIZE + 1];
336. for (int i = 1; i <= SIZE \* SIZE \* SIZE; i++) value1[i] = FALSE;
337. WriteToFile(board, 0, name); *// 将当前棋盘转化为CNF*
338. if (ReadFile(cnf1, name) != OK) {
339. printf("Error: Failed to read temp CNF for first solve.\n");
340. free(cnf1);
341. remove(name);
342. return FALSE; *// 文件读取失败，无法验证*
343. }
344. status result1 = DPLL(cnf1, value1, 3); *// 调用DPLL求解*
345. if (result1 == ERROR) {
346. *// 如果第一次求解就失败了，说明谜题根本无解，自然谈不上唯一解*
347. DestroyCnf(cnf1->root);
348. free(cnf1);
349. remove(name);
350. return FALSE;
351. }
352. *// 保存第一个解*
353. for (int i = 1; i <= SIZE \* SIZE \* SIZE; i++) {
354. firstSolution[i] = value1[i];
355. }
357. *// 如果找到了解，将其解码并存入solution输出参数*
358. for (int i = 1; i <= SIZE \* SIZE \* SIZE; i++) {
359. if (firstSolution[i] == TRUE) {
360. int row = (i - 1) / (SIZE \* SIZE) + 1;
361. int col = (i - 1) / SIZE % SIZE + 1;
362. int v = (i - 1) % SIZE + 1;
363. solution[row][col] = v;
364. }
365. }
366. DestroyCnf(cnf1->root); *// 释放第一次求解的CNF结构*
367. free(cnf1);
368. *// ==================== 阶段二：增加“阻塞子句”，尝试寻找第二个解 ====================*
369. *// 构建一个“阻塞子句”，这个子句是第一个解的逻辑否定*
370. clauseList blockingClause = (clauseList)malloc(sizeof(clauseNode));
371. blockingClause->head = NULL;
372. blockingClause->next = NULL;
373. literalList lastLiteral = NULL;
374. for (int i = 1; i <= SIZE \* SIZE \* SIZE; i++) {
375. literalList newLiteral = (literalList)malloc(sizeof(literalNode));
376. *// 如果第一解中变量i为真，阻塞子句中就加入-i；反之则加入i*
377. newLiteral->literal = (firstSolution[i] == TRUE) ? -i : i;
378. newLiteral->next = NULL;
379. if (blockingClause->head == NULL) {
380. blockingClause->head = newLiteral;
381. } else {
382. lastLiteral->next = newLiteral;
383. }
384. lastLiteral = newLiteral;
385. }
386. *// 重新读取原始CNF问题，并加入阻塞子句*
387. CNF cnf2 = (CNF)malloc(sizeof(cnfNode));
388. cnf2->root = NULL;
389. bool value2[SIZE \* SIZE \* SIZE + 1]; *// 第二次求解的结果数组（虽然内容不重要）*
390. if (ReadFile(cnf2, name) != OK) {
391. printf("Error: Failed to read temp CNF for second solve.\n");
392. DestroyClause(blockingClause); *// 清理创建的阻塞子句*
393. free(cnf2);
394. remove(name);
395. return FALSE; *// 文件读取失败*
396. }
397. *// 将阻塞子句添加到CNF公式的头部*
398. blockingClause->next = cnf2->root;
399. cnf2->root = blockingClause;
400. cnf2->clauseCount++;
401. status result2 = DPLL(cnf2, value2, 3); *// 进行第二次求解*
402. if (result2 == ERROR) {
403. *// 如果在排除了第一个解之后，问题变为“不可满足”，*
404. *// 这恰好证明了第一个解是唯一的。*
405. finalResult = TRUE;
406. } else {
407. *// 如果还能找到另一个解，说明解不唯一。*
408. finalResult = FALSE;
409. }
410. DestroyCnf(cnf2->root); *// 释放第二次求解的CNF结构 (这会同时释放阻塞子句)*
411. free(cnf2);
412. remove(name); *// 删除临时文件*
413. return finalResult;
414. }
415. *// 判断board[row][col]是否可以填入v*
416. status Is\_Valid(int board[SIZE + 1][SIZE + 1], int row, int col, int v) {
417. *// 检查行*
418. for (int j = 1; j <= SIZE; j++) {
419. if (j != col && board[row][j] == v) {
420. return FALSE;
421. }
422. }
423. *// 检查列*
424. for (int i = 1; i <= SIZE; i++) {
425. if (i != row && board[i][col] == v) {
426. return FALSE;
427. }
428. }
429. *// 检查3x3宫格*
430. int startRow = ((row - 1) / 3) \* 3 + 1;
431. int startCol = ((col - 1) / 3) \* 3 + 1;
432. for (int i = startRow; i < startRow + 3; i++) {
433. for (int j = startCol; j < startCol + 3; j++) {
434. if ((i != row || j != col) && board[i][j] == v) {
435. return FALSE;
436. }
437. }
438. }
439. *// 检查副对角线*
440. if (row + col == SIZE + 1) {
441. for (int i = 1; i <= SIZE; i++) {
442. int j = SIZE + 1 - i;
443. if (i != row && board[i][j] == v) {
444. return FALSE;
445. }
446. }
447. }
448. *// 检查两个窗口*
449. int windows[2][2] = {{2, 2}, {6, 6}};
450. for (int w = 0; w < 2; w++) {
451. int startRow = windows[w][0];
452. int startCol = windows[w][1];
453. *// 检查当前单元格是否在这个窗口中*
454. if (row >= startRow && row < startRow + 3 && col >= startCol &&
455. col < startCol + 3) {
456. *// 检查窗口内是否有重复数字*
457. for (int i = startRow; i < startRow + 3; i++) {
458. for (int j = startCol; j < startCol + 3; j++) {
459. if ((i != row || j != col) && board[i][j] == v) {
460. return FALSE;
461. }
462. }
463. }
464. break; *// 找到所属窗口后跳出循环*
465. }
466. }
467. return TRUE;
468. }
469. *// 打印数独*
470. void Print\_Sudoku(int board[SIZE + 1][SIZE + 1]) {
471. printf("   ");
472. for (int j = 1; j <= SIZE; j++) {
473. printf("%2d ", j);
474. if (j % 3 == 0 && j != SIZE)
475. printf("| ");
476. }
477. printf("\n");
478. printf("   ");
479. for (int j = 1; j <= SIZE; j++) {
480. printf("---");
481. if (j % 3 == 0 && j != SIZE)
482. printf("+");
483. }
484. printf("\n");
485. for (int i = 1; i <= SIZE; i++) {
486. printf("%2d|", i);
487. for (int j = 1; j <= SIZE; j++) {
488. if (board[i][j] == 0) *// 未填入*
489. printf(" . ");
490. else *// 已填入*
491. printf("%2d ", board[i][j]);
492. if (j % 3 == 0 && j != SIZE) *// 每3列打印一个竖线*
493. printf("| ");
494. }
495. printf("\n");
496. if (i % 3 == 0 && i != SIZE) { *// 每3行打印一个横线*
497. printf("   ");
498. for (int j = 1; j <= SIZE; j++) {
499. printf("---");
500. if (j % 3 == 0 && j != SIZE)
501. printf("+");
502. }
503. printf("\n");
504. }
505. }
506. }
507. *// 玩数独的交互界面*
508. void Play\_Sudoku(int board[SIZE + 1][SIZE + 1],
509. bool isFixed[SIZE + 1][SIZE + 1]) {
510. system("cls");       *// 清屏*
511. Print\_Sudoku(board); *// 打印初始数独*
512. printf("\n");
513. while (1) {
514. int row, col, v;
515. printf(" Please enter the row, col and value(0 to EXIT): ");
516. scanf("%d", &row);
517. if (row == 0) *// 退出*
518. {
519. system("cls");
520. return;
521. }
522. scanf("%d%d", &col, &v);
523. if (row < 1 || row > SIZE || col < 1 || col > SIZE || v < 1 ||
524. v > SIZE) *// 输入不合法*
525. {
526. printf(" Invalid input.\n");
527. continue;
528. }
529. if (isFixed[row][col]) *// 是提示数*
530. {
531. printf(" This is a fixed number.\n");
532. continue;
533. }
534. if (!Is\_Valid(board, row, col, v)) *// 不符合数独规则*
535. {
536. printf(" Wrong answer.\n");
537. continue;
538. } else *// 符合数独规则*
539. {
540. board[row][col] = v;
541. system("cls");
542. Print\_Sudoku(board); *// 打印新数独*
543. printf("\n");
544. *// 检查是否完成*
545. int complete = 1;
546. for (int i = 1; i <= SIZE; i++) {
547. for (int j = 1; j <= SIZE; j++) {
548. if (board[i][j] == 0) {
549. complete = 0;
550. break;
551. }
552. }
553. if (!complete)
554. break;
555. }
556. if (complete) {
557. if (Validate\_Sudoku(board)) {
558. printf(" Congratulations! You solved the Sudoku!\n");
559. } else {
560. printf(" Sorry, your solution is incorrect.\n");
561. }
562. return;
563. }
564. }
565. }
566. }
567. *// 将数独约束条件写入文件*
568. status WriteToFile(int board[SIZE + 1][SIZE + 1], int num, char name[]) {
569. FILE \*fp;
570. if (fopen\_s(&fp, name, "w")) {
571. printf(" Fail!\n");
572. return ERROR;
573. }
574. *// 计算总子句数*
575. int clauseCount = num; *// 提示数子句*
576. *// 每个格子必须填入一个数字: 81个子句*
577. clauseCount += 81;
578. *// 每个格子不能填入两个数字: C(9,2)\*81 = 36\*81 = 2916个子句*
579. clauseCount += 2916;
580. *// 行约束: 每行必须包含1-9: 9\*9 = 81个子句*
581. clauseCount += 81;
582. *// 行约束: 每行不能有重复数字: C(9,2)\*9\*9 = 36\*81 = 2916个子句*
583. clauseCount += 2916;
584. *// 列约束: 每列必须包含1-9: 9\*9 = 81个子句*
585. clauseCount += 81;
586. *// 列约束: 每列不能有重复数字: C(9,2)\*9\*9 = 36\*81 = 2916个子句*
587. clauseCount += 2916;
588. *// 3x3宫格约束: 每个宫格必须包含1-9: 9\*9 = 81个子句*
589. clauseCount += 81;
590. *// 3x3宫格约束: 每个宫格不能有重复数字: C(9,2)\*9\*9 = 36\*81 = 2916个子句*
591. clauseCount += 2916;
592. *// 副对角线约束: 必须包含1-9: 9个子句*
593. clauseCount += 9;
594. *// 副对角线约束: 不能有重复数字: C(9,2) = 36个子句*
595. clauseCount += 36;
596. *// 窗口约束: 两个窗口各必须包含1-9: 2\*9 = 18个子句*
597. clauseCount += 18;
598. *// 窗口约束: 两个窗口各不能有重复数字: 2\*C(9,2) = 2\*36 = 72个子句*
599. clauseCount += 72;
600. fprintf(fp, "c %s\n", name);
601. fprintf(fp, "p cnf 729 %d\n", clauseCount);
602. */\*提示数约束(写在前面,便于单子句规则进行)\*/*
603. for (int i = 1; i <= SIZE; i++) {
604. for (int j = 1; j <= SIZE; j++) {
605. if (board[i][j] != 0) {
606. fprintf(fp, "%d 0\n",
607. (i - 1) \* SIZE \* SIZE + (j - 1) \* SIZE + board[i][j]);
608. }
609. }
610. }
611. */\*每个格子的约束\*/*
612. *// 每个格子必须填入一个数字*
613. for (int i = 1; i <= SIZE; i++) {
614. for (int j = 1; j <= SIZE; j++) {
615. for (int k = 1; k <= SIZE; k++) {
616. fprintf(fp, "%d ", (i - 1) \* SIZE \* SIZE + (j - 1) \* SIZE + k);
617. }
618. fprintf(fp, "0\n");
619. }
620. }
621. *// 每个格子不能填入两个数字*
622. for (int i = 1; i <= SIZE; i++) {
623. for (int j = 1; j <= SIZE; j++) {
624. for (int k = 1; k <= SIZE; k++) {
625. for (int l = k + 1; l <= SIZE; l++) {
626. fprintf(fp, "%d %d 0\n",
627. -((i - 1) \* SIZE \* SIZE + (j - 1) \* SIZE + k),
628. -((i - 1) \* SIZE \* SIZE + (j - 1) \* SIZE + l));
629. }
630. }
631. }
632. }
633. */\*行约束\*/*
634. *// 每一行必须填入1-9*
635. for (int i = 1; i <= SIZE; i++) {
636. for (int k = 1; k <= SIZE; k++) {
637. for (int j = 1; j <= SIZE; j++) {
638. fprintf(fp, "%d ", (i - 1) \* SIZE \* SIZE + (j - 1) \* SIZE + k);
639. }
640. fprintf(fp, "0\n");
641. }
642. }
643. *// 每一行不能填入两个相同的数字*
644. for (int i = 1; i <= SIZE; i++) {
645. for (int k = 1; k <= SIZE; k++) {
646. for (int j = 1; j <= SIZE; j++) {
647. for (int l = j + 1; l <= SIZE; l++) {
648. fprintf(fp, "%d %d 0\n",
649. -((i - 1) \* SIZE \* SIZE + (j - 1) \* SIZE + k),
650. -((i - 1) \* SIZE \* SIZE + (l - 1) \* SIZE + k));
651. }
652. }
653. }
654. }
655. */\*列约束\*/*
656. *// 每一列必须填入1-9*
657. for (int j = 1; j <= SIZE; j++) {
658. for (int k = 1; k <= SIZE; k++) {
659. for (int i = 1; i <= SIZE; i++) {
660. fprintf(fp, "%d ", (i - 1) \* SIZE \* SIZE + (j - 1) \* SIZE + k);
661. }
662. fprintf(fp, "0\n");
663. }
664. }
665. *// 每一列不能填入两个相同的数字*
666. for (int j = 1; j <= SIZE; j++) {
667. for (int k = 1; k <= SIZE; k++) {
668. for (int i = 1; i <= SIZE; i++) {
669. for (int l = i + 1; l <= SIZE; l++) {
670. fprintf(fp, "%d %d 0\n",
671. -((i - 1) \* SIZE \* SIZE + (j - 1) \* SIZE + k),
672. -((l - 1) \* SIZE \* SIZE + (j - 1) \* SIZE + k));
673. }
674. }
675. }
676. }
677. */\*3x3宫格约束\*/*
678. *// 每个3x3宫格必须填入1-9*
679. for (int boxRow = 1; boxRow <= SIZE; boxRow += 3) {
680. for (int boxCol = 1; boxCol <= SIZE; boxCol += 3) {
681. for (int k = 1; k <= SIZE; k++) {
682. for (int i = boxRow; i < boxRow + 3; i++) {
683. for (int j = boxCol; j < boxCol + 3; j++) {
684. fprintf(fp, "%d ", (i - 1) \* SIZE \* SIZE + (j - 1) \* SIZE + k);
685. }
686. }
687. fprintf(fp, "0\n");
688. }
689. }
690. }
691. *// 每个3x3宫格不能填入两个相同的数字*
692. for (int boxRow = 1; boxRow <= SIZE; boxRow += 3) {
693. for (int boxCol = 1; boxCol <= SIZE; boxCol += 3) {
694. for (int k = 1; k <= SIZE; k++) {
695. for (int i = boxRow; i < boxRow + 3; i++) {
696. for (int j = boxCol; j < boxCol + 3; j++) {
697. for (int p = i; p < boxRow + 3; p++) {
698. for (int q = (p == i ? j + 1 : boxCol); q < boxCol + 3; q++) {
699. fprintf(fp, "%d %d 0\n",
700. -((i - 1) \* SIZE \* SIZE + (j - 1) \* SIZE + k),
701. -((p - 1) \* SIZE \* SIZE + (q - 1) \* SIZE + k));
702. }
703. }
704. }
705. }
706. }
707. }
708. }
709. */\*副对角线约束\*/*
710. *// 副对角线必须包含1-9*
711. for (int k = 1; k <= SIZE; k++) {
712. for (int i = 1; i <= SIZE; i++) {
713. int j = SIZE + 1 - i;
714. fprintf(fp, "%d ", (i - 1) \* SIZE \* SIZE + (j - 1) \* SIZE + k);
715. }
716. fprintf(fp, "0\n");
717. }
718. *// 副对角线不能有重复数字*
719. for (int k = 1; k <= SIZE; k++) {
720. for (int i = 1; i <= SIZE; i++) {
721. int j = SIZE + 1 - i;
722. for (int p = i + 1; p <= SIZE; p++) {
723. int q = SIZE + 1 - p;
724. fprintf(fp, "%d %d 0\n", -((i - 1) \* SIZE \* SIZE + (j - 1) \* SIZE + k),
725. -((p - 1) \* SIZE \* SIZE + (q - 1) \* SIZE + k));
726. }
727. }
728. }
729. */\*两个窗口约束\*/*
730. int windows[2][2] = {{2, 2}, {6, 6}};
731. for (int w = 0; w < 2; w++) {
732. int startRow = windows[w][0];
733. int startCol = windows[w][1];
734. *// 窗口必须包含1-9*
735. for (int k = 1; k <= SIZE; k++) {
736. for (int i = startRow; i < startRow + 3; i++) {
737. for (int j = startCol; j < startCol + 3; j++) {
738. fprintf(fp, "%d ", (i - 1) \* SIZE \* SIZE + (j - 1) \* SIZE + k);
739. }
740. }
741. fprintf(fp, "0\n");
742. }
743. *// 窗口内不能有重复数字*
744. for (int k = 1; k <= SIZE; k++) {
745. for (int i = startRow; i < startRow + 3; i++) {
746. for (int j = startCol; j < startCol + 3; j++) {
747. for (int p = i; p < startRow + 3; p++) {
748. for (int q = (p == i ? j + 1 : startCol); q < startCol + 3; q++) {
749. fprintf(fp, "%d %d 0\n",
750. -((i - 1) \* SIZE \* SIZE + (j - 1) \* SIZE + k),
751. -((p - 1) \* SIZE \* SIZE + (q - 1) \* SIZE + k));
752. }
753. }
754. }
755. }
756. }
757. }
758. fclose(fp);
759. return OK;
760. }
761. *// DPLL求解数独*
762. status Slove(int board[SIZE + 1][SIZE + 1],
763. bool value[SIZE \* SIZE \* SIZE + 1]) {
764. for (int i = 1; i <= SIZE \* SIZE \* SIZE; i++) {
765. if (value[i] == TRUE) {
766. int row = (i - 1) / (SIZE \* SIZE) + 1;
767. int col = (i - 1) / SIZE % SIZE + 1;
768. int v = (i - 1) % SIZE + 1;
769. board[row][col] = v;
770. }
771. }
772. return OK;
773. }
774. *// 填充3x3的宫格*
775. status Fill\_Box(int board[SIZE + 1][SIZE + 1], int newBoard[SIZE + 1][SIZE + 1],
776. int newBoard2[SIZE + 1][SIZE + 1], int rowStart, int colStart) {
777. int numbers[SIZE];
778. for (int i = 0; i < SIZE; i++) {
779. numbers[i] = i + 1;
780. }
781. Shuffle(numbers, SIZE);
782. int index = 0;
783. for (int i = rowStart; i < rowStart + 3; i++) {
784. for (int j = colStart; j < colStart + 3; j++) {
785. if (board[i][j] == 0) {
786. board[i][j] = numbers[index];
787. newBoard[i][j] = numbers[index];
788. newBoard2[i][j] = numbers[index];
789. index++;
790. }
791. }
792. }
793. return OK;
794. }
795. *// 打乱数组顺序*
796. void Shuffle(int arr[], int n) {
797. srand(time(NULL)); *// 用时间做种子*
798. *// 每次从后面的数中随机选一个数与前面的数交换*
799. for (int i = n - 1; i > 0; i--) {
800. int j = rand() % (i + 1);
801. int temp = arr[i];
802. arr[i] = arr[j];
803. arr[j] = temp;
804. }