華中科技大學

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：$[(t-t_o)/t]*100\%$,其中 t 为未对 DPLL 优化时求解基准算例的执行时间，t_o 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%)

口 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

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、SATO、CHAFF、POSIT 和 GRASZP 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**错误!未找到引用源。**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。

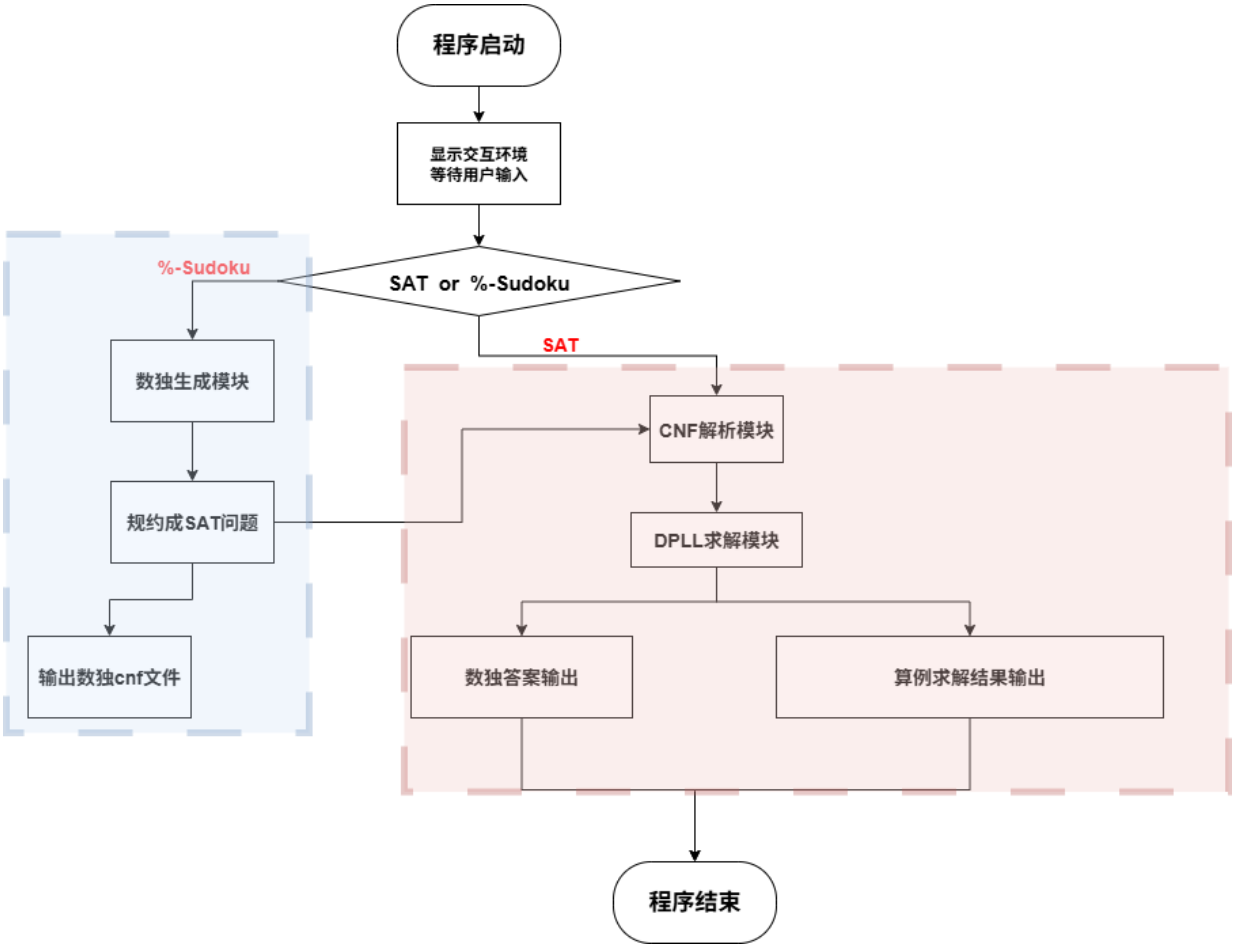

程序启动
显示交互环境
等待用户输入
%-Sudoku
SAT or %-Sudoku
SAT
数独生成模块
规约成SAT问题
输出数独cnf文件
CNF解析模块
DPLL求解模块
数独答案输出
算例求解结果输出
程序结束

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

```
typedef struct literalNode {
   int literal;             // 文字(变元)
   struct literalNode *next; // 指向下一个文字
} literalNode, *literalList;
```

（2）子句节点

```
typedef struct clauseNode {
   literalList head;        // 指向子句中的第一个文字
   struct clauseNode *next; // 指向下一个子句
} clauseNode, *clauseList;
```

（3）CNF 文件

```
typedef struct cnfNode {
   clauseList root; // 指向 CNF 的第一个子句
   int boolCount;   // 布尔变元个数
   int clauseCount; // 子句个数
} 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。

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

• **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。

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

下图为直观展示这多种数据间的关联的流程图：

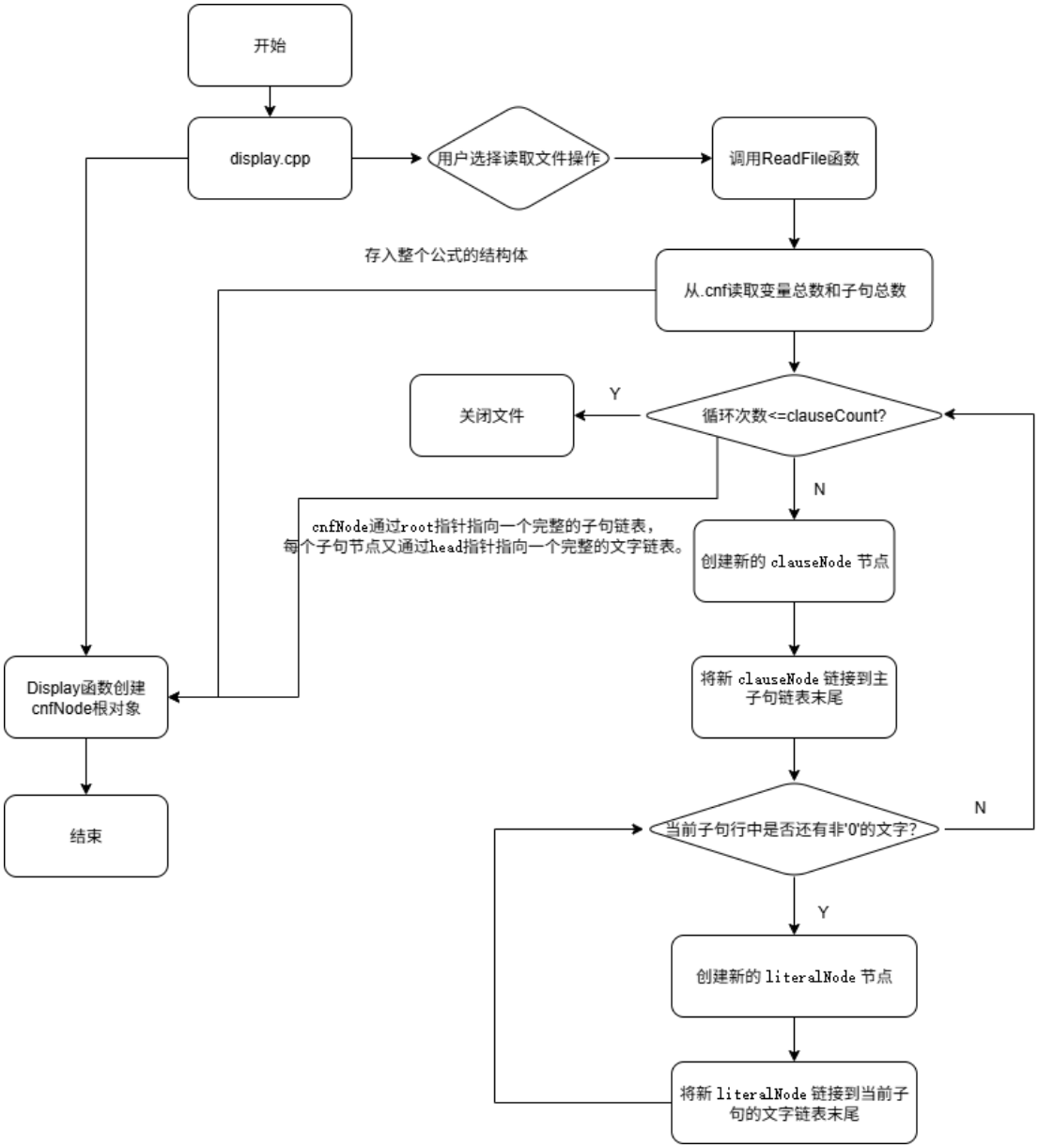

开始
display.cpp
用户选择读取文件操作
调用ReadFile函数
存入整个公式的结构体
从.cnf读取变量总数和子句总数
关闭文件
Y
循环次数<=clauseCount?
N
cnfNode通过root指针指向一个完整的子句链表，
每个子句节点又通过head指针指向一个完整的文字链表。
创建新的 clauseNode 节点
Display函数创建
cnfNode根对象
将新 clauseNode 链接到主
子句链表末尾
结束
当前子句行中是否还有非'0'的文字?
N
Y
创建新的 literalNode 节点
将新 literalNode 链接到当前子
句的文字链表末尾

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2^n种可能的变量赋值组合，因此时间复杂度为 O(2^n)，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：$O(L_{total})$，其中L_{total}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：$O(1)$，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：$O(L_{total})$，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：：$O(n)$，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：$O(L_{total})$其中L_{total}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：$O(L_{total})$，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S_1），算法会进入关键的第二阶段。它会根据解S_1动态地构建一个特殊的“阻塞子句”。这个子句是解S_1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S_1。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S_1之后再也找不到任何其他解了。这反过来证明了S_1是唯一的。如果第二次求解成功，意味着找到了一个不同于S_1的新解S_1。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$O(2^k)$，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$2O(2^k)$，仍然是$O(2^k)$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：$O(SIZE^2)$。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：$O(SIZE)$。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.
7. /*子句节点&链表*/
8. typedef struct clauseNode {
9.   literalList head;        // 指向子句中的第一个文字
10.  struct clauseNode *next; // 指向下一个子句
11.} clauseNode, *clauseList;
12.
13./*CNF 文件*/
14.typedef struct cnfNode {
15.  clauseList root; // 指向 CNF 的第一个子句
16.  int boolCount;   // 布尔变元个数
17.  int clauseCount; // 子句个数
18.} cnfNode, *CNF;
```

4.1.3 函数说明

（1）主控与显示模块

函数名	函数说明
main()	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。
Display()	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：

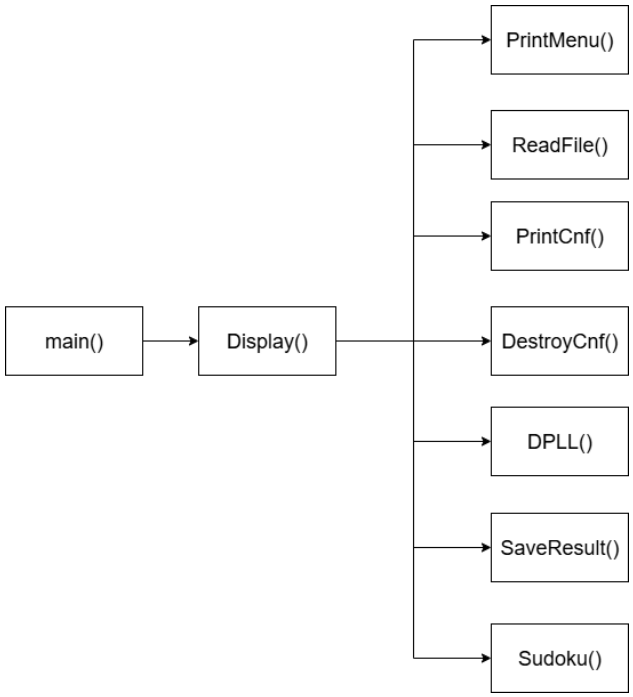


图 3 主控与显示模块函数调用关系图

（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$Display()$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：

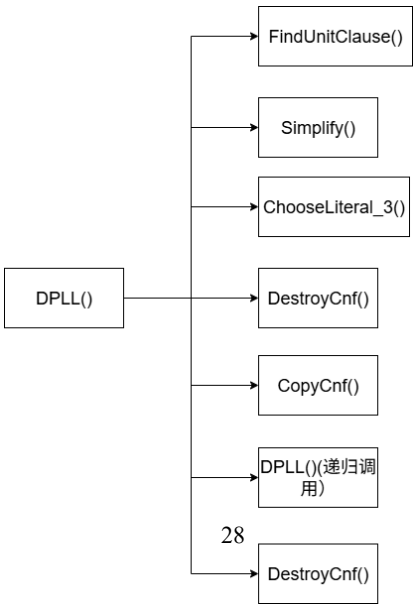


图 4 DPLL 求解模块函数调用关系图

（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

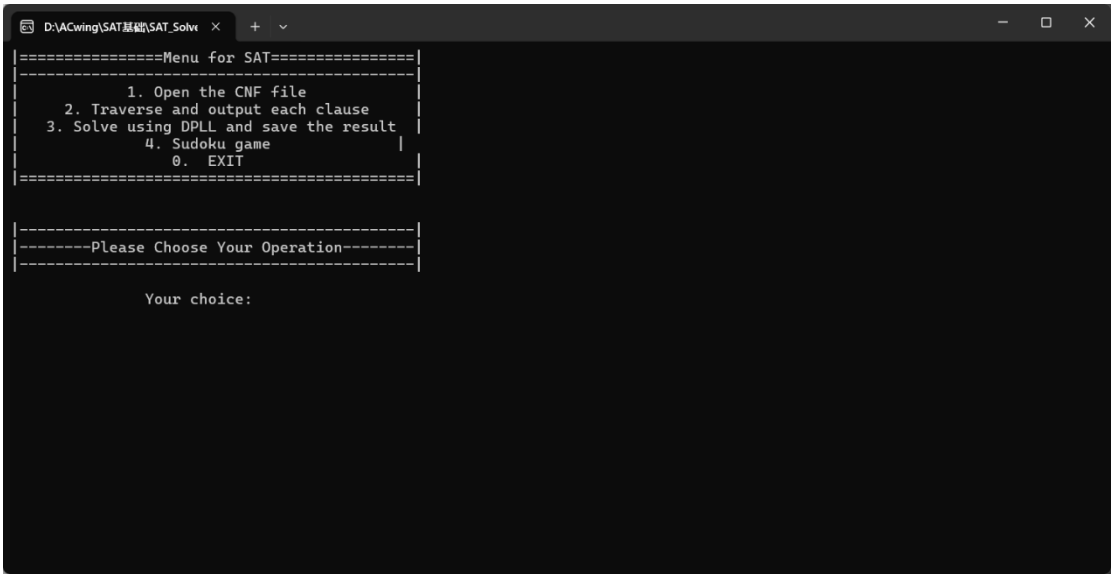


图 6 功能演示菜单展示

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。

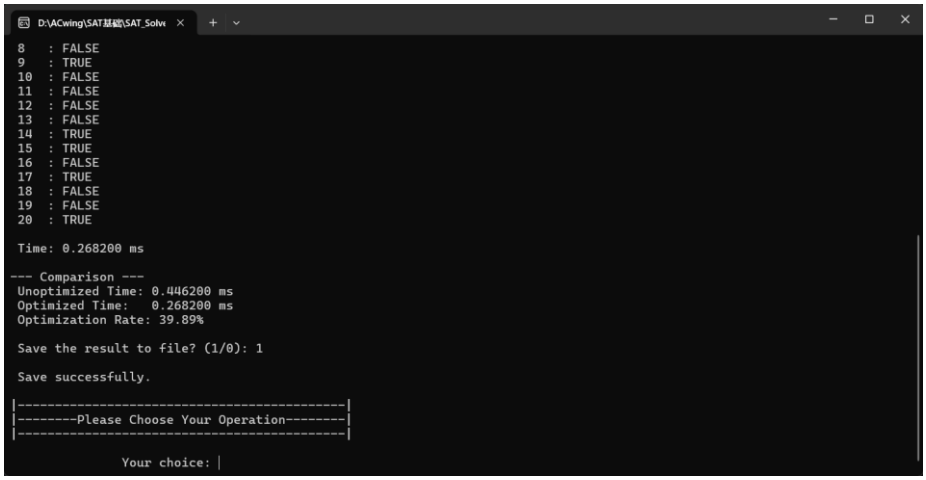
（2）测试数据选取与测试大纲

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	预期结果	实际结果	t_0/ms	t/ms	优化率

			变元数比值					
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%



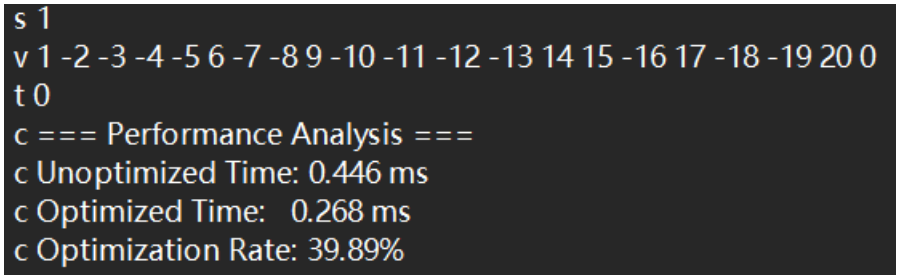


图 7 功能测试 1 运行结果和答案保存

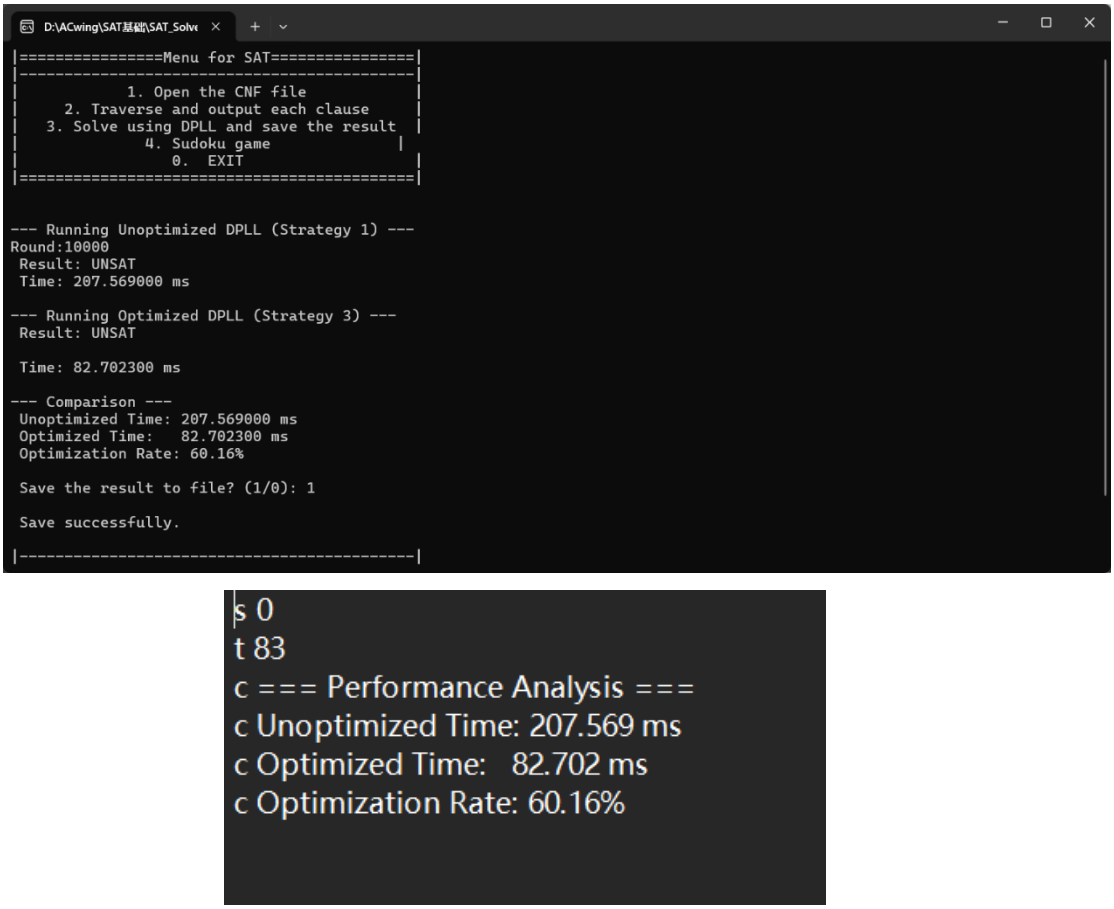


图 8 功能测试 2 运行和答案保存

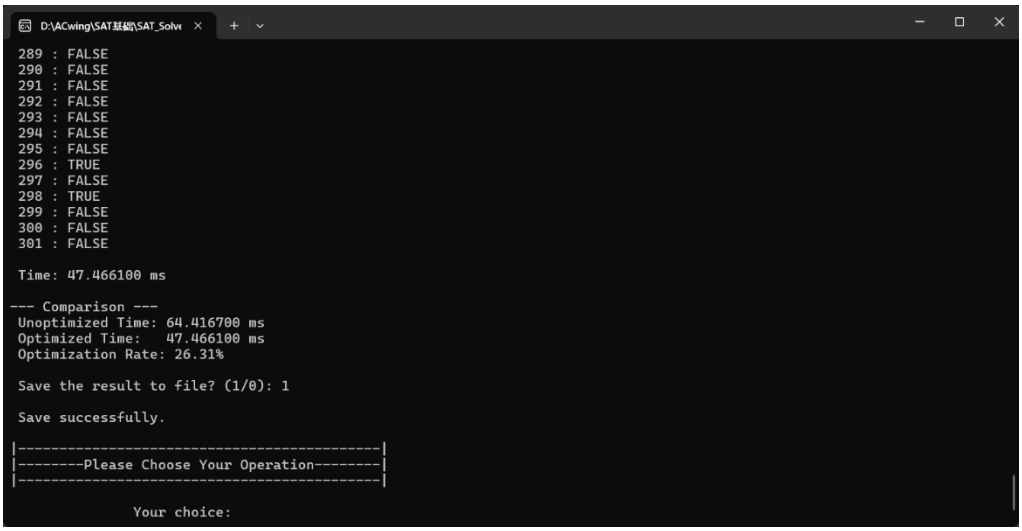
（4）性能测试

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	t_0/ms	t/ms	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	可		3316	

				满足	满足	/	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：



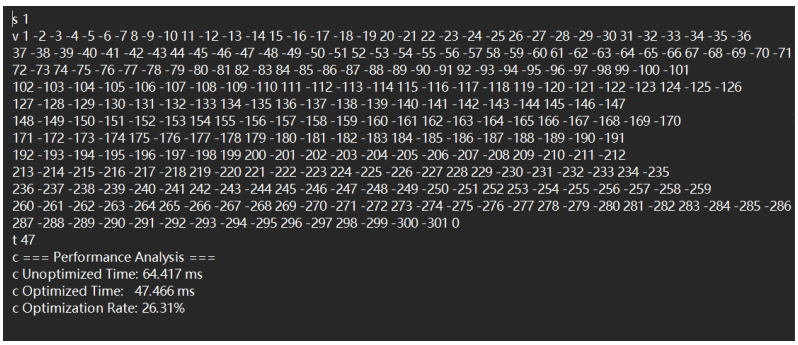


图 9 性能测试 3.cnf 运行结果和答案保存

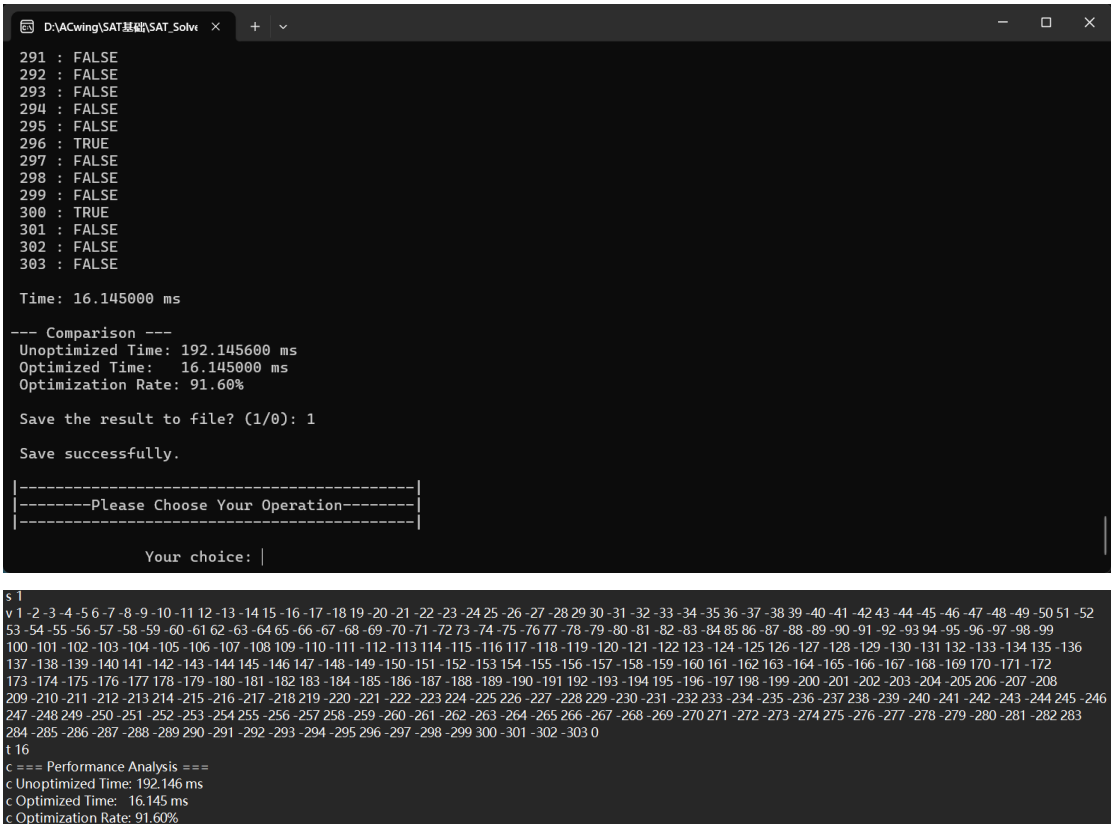
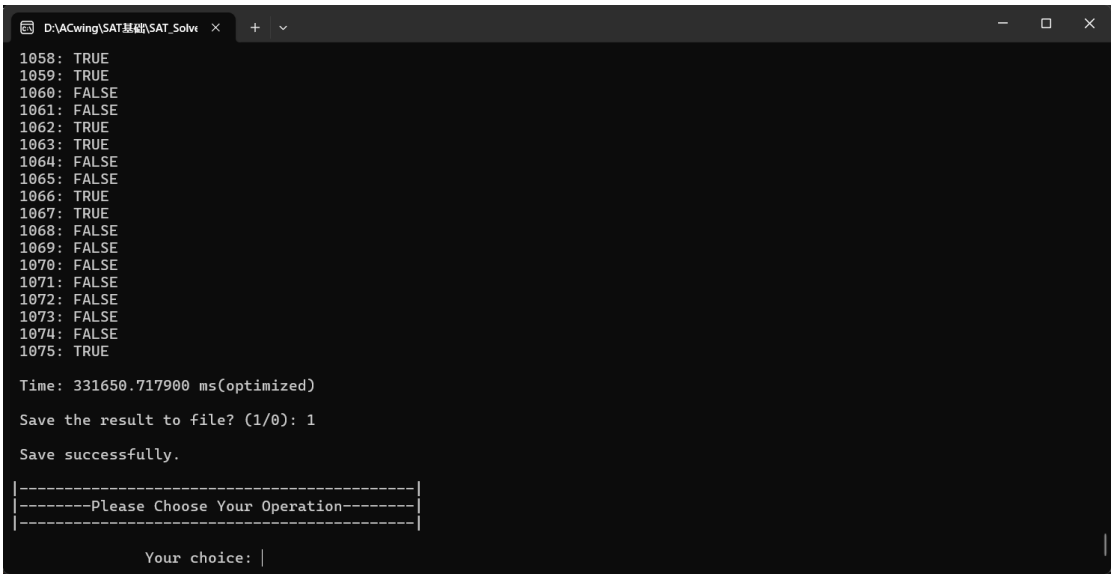


图 10 性能测试 sud00009.cnf 运行结果和答案保存



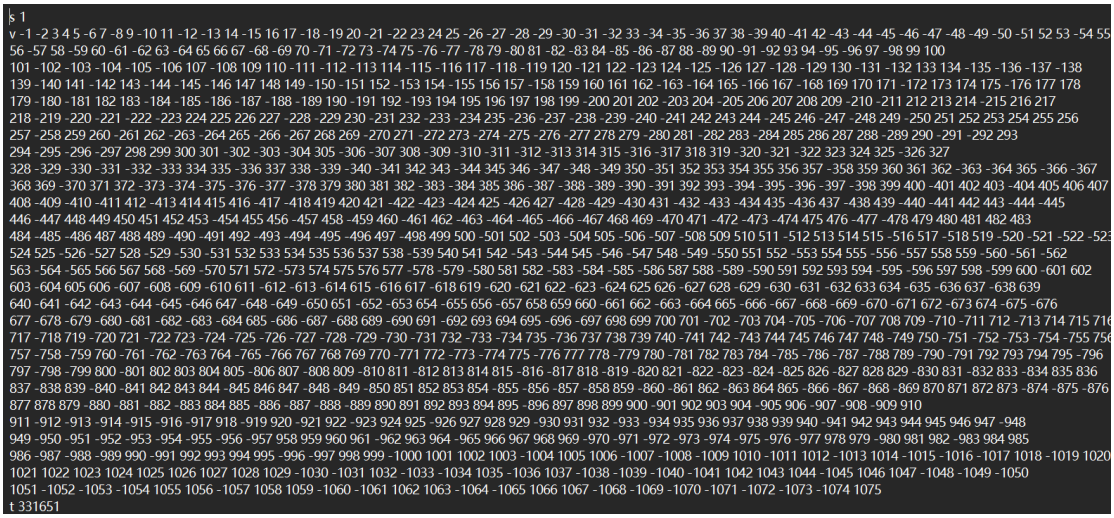


图 9 性能测试 2.cnf 运行结果和答案保存

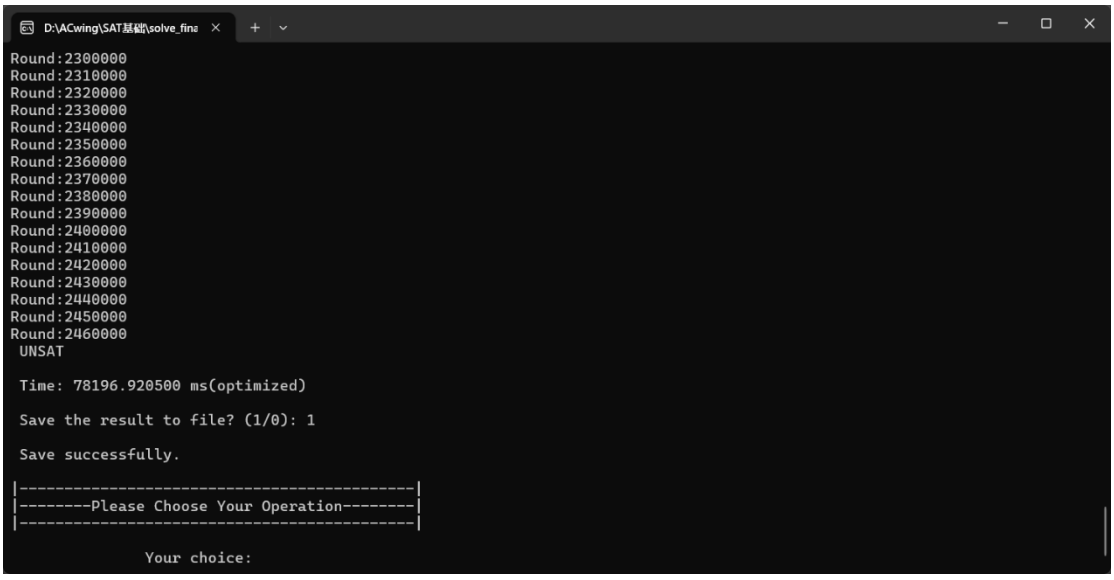


图 11 性能测试 11(unsatisfied).cnf 运行结果和答案保存

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

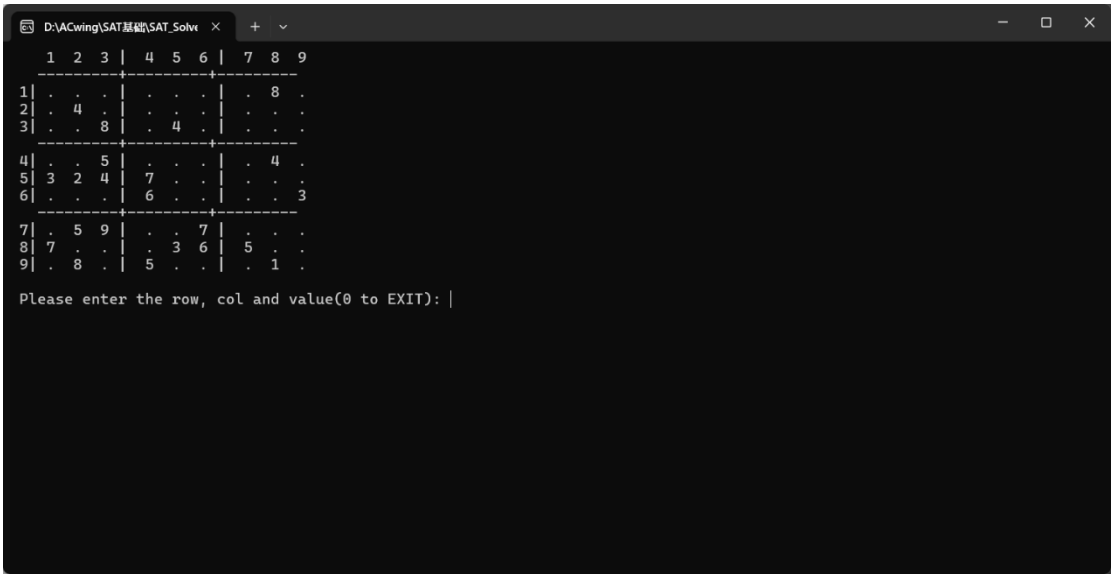


图 12 数独游戏开局随机生成数独

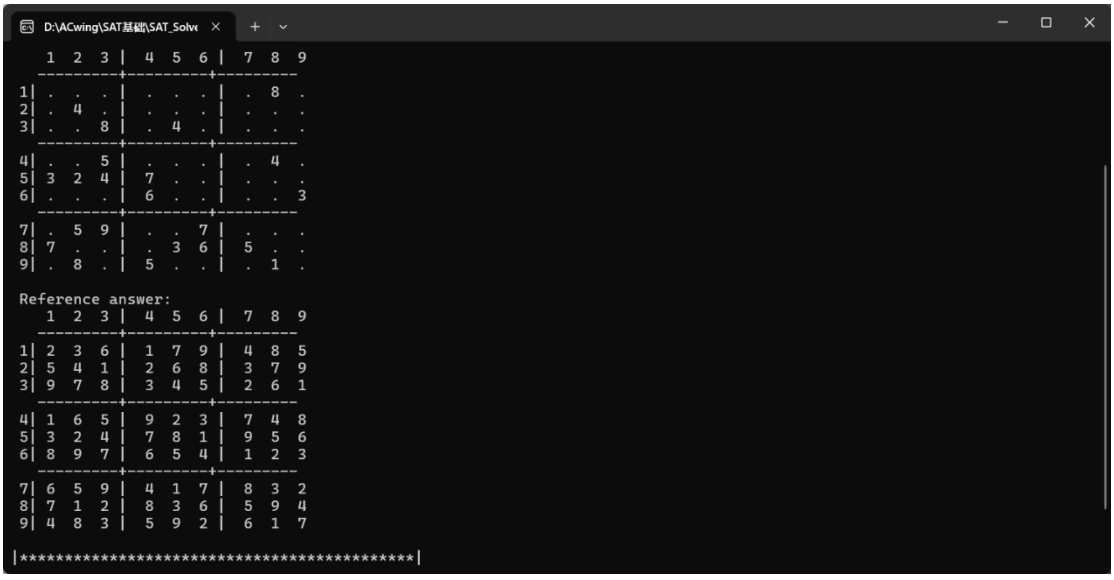


图 13 输出正确答案

（3）数独交互测试

填入的位置已有数字：

输入 2 2 1

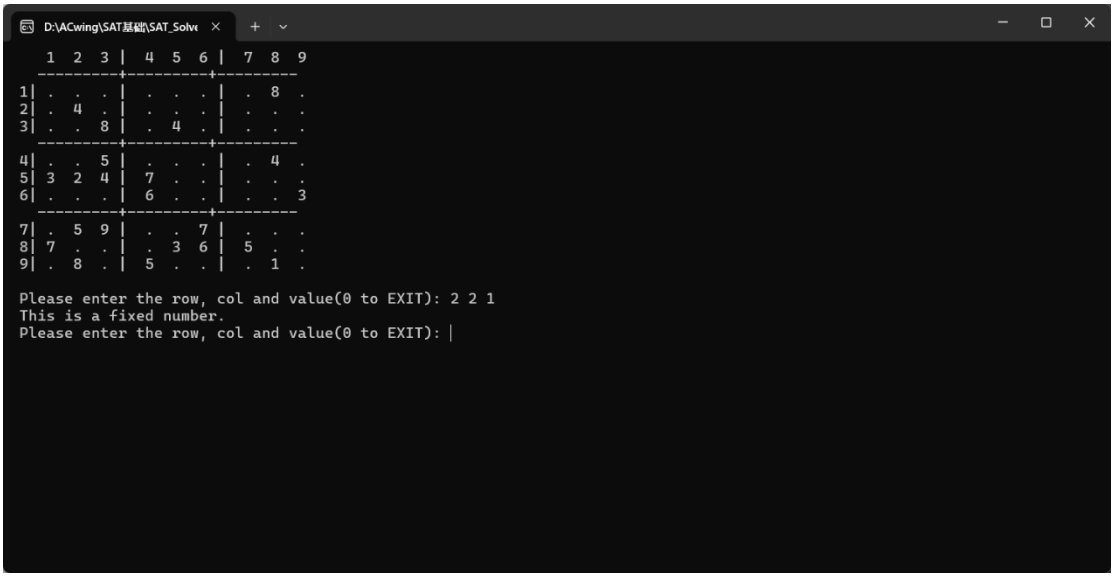


图 14 数独交互测试 1

填入数字违反规则：

输入 1 1 4

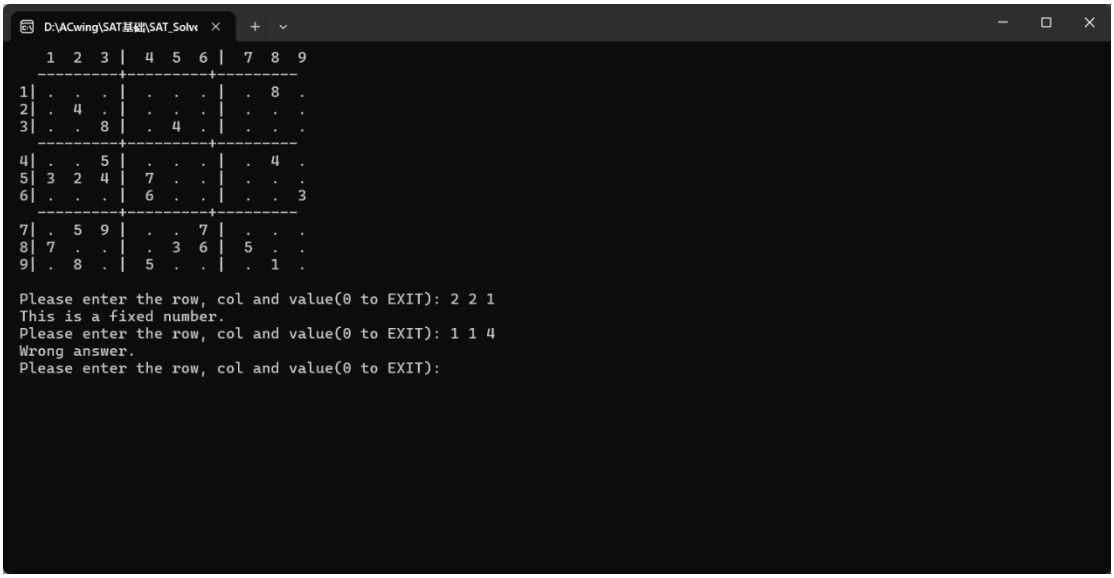


图 15 数独交互测试 2

填入正确数字：

输入 1 1 1

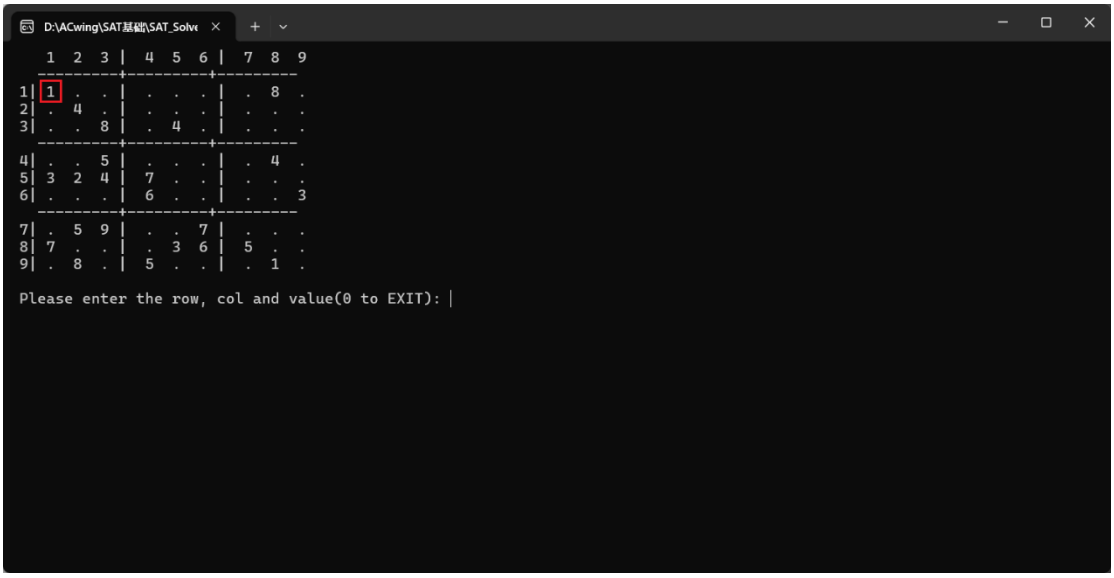


图 16 数独交互测试 3

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

```
#pragma once

/*头文件*/
#include <malloc.h>
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <windows.h>
#include <winnt.h>

/*常量*/
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0
#define SIZE 9

extern long long cnt;
/*状态码*/
typedef int status;

/*文字节点&链表*/
typedef struct literalNode {
  int literal;              // 文字(变元)
  struct literalNode *next; // 指向下一个文字
} literalNode, *literalList;

/*子句节点&链表*/
typedef struct clauseNode {
  literalList head;        // 指向子句中的第一个文字
  struct clauseNode *next; // 指向下一个子句
} clauseNode, *clauseList;

/*CNF 文件*/
```

```
38.typedef struct cnfNode {
39.  clauseList root; // 指向CNF的第一个子句
40.  int boolCount;   // 布尔变元个数
41.  int clauseCount; // 子句个数
42.} cnfNode, *CNF;
43.
44./*函数声明*/
45.void DisPlay();                                  // 主交互界面
46.void PrintMenu();                                // 打印菜单
47.status ReadFile(CNF &cnf, char fileName[]);   // 读取文件并解
   析cnf
48.status DestroyCnf(clauseList &cL);            // 销毁当前解析的
   cnf
49.status PrintCnf(CNF cnf);                       // 打印cnf
50.status DPLL(CNF cnf, bool value[], int flag); // DPLL算法
51.status IsUnitClause(literalList l);           // 判断是否为单子
   句
52.int FindUnitClause(clauseList cL);    // 找到单子句并返回该文
   字
53.status DestroyClause(clauseList &cL); // 销毁子句
54.int ChooseLiteral_1(CNF cnf); // 选择一个未赋值的文字(未优化)
55.int ChooseLiteral_2(CNF cnf); // 选择一个未赋值的文字(优化)
56.int ChooseLiteral_3(CNF cnf); // 改进2
57.void Simplify(clauseList &cL, int literal); // 根据选择的文字
   化简
58.status Satisfy(clauseList cL);              // 判断文字是否满足
59.status EmptyClause(clauseList cL);          // 判断是否有空子句
60.clauseList CopyCnf(clauseList cL);          // 复制cnf
61.status SaveResult(int result, double time, double time_, bo
   ol value[],
62.                  char fileName[], int boolCount); // 保存求
   解结果
63.void Sudoku();                                    // X数独
64.void PrintMenu_X();                               // 打印X数独
   菜单
65.status Generate_Sudoku(int board[SIZE + 1][SIZE + 1],
66.                       int newBoard[SIZE + 1][SIZE + 1],
67.                       int newBoard2[SIZE + 1][SIZE + 1],
68.                       bool isFixed[SIZE + 1][SIZE + 1], in
   t num,
69.                       bool value[SIZE * SIZE * SIZE + 1]);
    // 生成数独
70.status Is_Valid(int board[SIZE + 1][SIZE + 1], int row, int
    col,
```

```
                int v); // 判断board[row][col]是否可以填入
void Print_Sudoku(int board[SIZE + 1][SIZE + 1]); // 打印数独
void Play_Sudoku(int board[SIZE + 1][SIZE + 1],
                 bool isFixed[SIZE + 1][SIZE + 1]); // 玩数独
status WriteToFile(int board[SIZE + 1][SIZE + 1], int num,
                   char name[]); // 将数独约束条件写入文件
status Slove(int board[SIZE + 1][SIZE + 1],
             bool value[SIZE * SIZE * SIZE + 1]); // 求解数独
status Fill_Box(int board[SIZE + 1][SIZE + 1], int newBoard
[SIZE + 1][SIZE + 1],
                int newBoard2[SIZE + 1][SIZE + 1], int rowS
tart,
                int colStart);  // 填充3x3的宫格
void Shuffle(int arr[], int n); // 打乱数组
status Validate_Sudoku(
    int board[SIZE + 1][SIZE + 1]); // 用于验证数独是否满足所有
约束条件
status
HasUniqueSolution(int board[SIZE + 1][SIZE + 1],
                  bool isFixed[SIZE + 1][SIZE + 1],
                  int solution[SIZE + 1][SIZE + 1]); // 用于
检查数独是否有唯一解
```

display.cpp

```
/*----------------------------display-----------------------
------*/

#include "SAT.hpp"

/*
 @ 函数名称: DisPlay
 @ 函数功能: 交互界面
 @ 返回值: void
 */
void DisPlay() {
  bool *value = NULL; // 存储文字的真值
  CNF cnf = (CNF)malloc(sizeof(cnfNode));
  cnf->root = NULL;
  char fileName[100]; // 文件名
  PrintMenu();        // 打印菜单
  int op = 1;
```

```
17.  while (op) {
18.    printf("\n|-------------------------------------------
   |\n");
19.    printf("|--------Please Choose Your Operation--------|\
   n");
20.    printf("|-----------------------------------------|\
   n\n");
21.    printf("              Your choice: ");
22.    scanf("%d", &op);
23.    system("cls"); // 每次进来清屏
24.    PrintMenu();   // 打印菜单
25.    switch (op) {
26.    case 1: {
27.      if (cnf->root != NULL) // 如果已经打开了CNF 文件
28.      {
29.        printf(" The CNF has been read.\n");
30.        printf(" Do you want to read another? (1/0): ");
31.        int choice;
32.        scanf("%d", &choice);
33.        if (choice == 0)
34.          break;
35.        else // 重新读取
36.        {
37.          cnt = 0;
38.          DestroyCnf(cnf->root); // 销毁当前解析的CNF
39.        }
40.      }
41.      printf(" Please input the file name: ");
42.      scanf("%s", fileName);
43.      if (ReadFile(cnf, fileName) == OK) // 读取文件并解析CNF
44.        printf(" Read successfully.\n");
45.      else
46.        printf(" Read failed.\n");
47.      break;
48.    }
49.    case 2: {
50.      if (cnf->root == NULL) // 如果没有打开CNF 文件
51.        printf(" You haven't open the CNF file.\n");
52.      else
53.        PrintCnf(cnf); // 打印CNF 文件
54.      break;
55.    }
56.    // case 3: {
57.    //   if (cnf->root == NULL) // 如果没有打开CNF 文件
```

```
58.     //   {
59.     //     printf(" You haven't open the CNF file.\n");
60.     //     break;
61.     //   } else {
62.     //     CNF newCnf = (CNF)malloc(sizeof(cnfNode));
63.     //     newCnf->root = CopyCnf(cnf->root); // 复制CNF
64.     //     newCnf->boolCount = cnf->boolCount;
65.     //     newCnf->clauseCount = cnf->clauseCount;
66.     //     value = (bool *)malloc(sizeof(bool) * (cnf->boolC
    ount + 1));
67.     //     for (int i = 1; i <= cnf->boolCount; i++)
68.     //       value[i] = FALSE;                    // 初始化为
    FALSE
69.     //     LARGE_INTEGER frequency, frequency_;    // 计时器频
    率
70.     //     LARGE_INTEGER start, start_, end, end_; // 设置时间
    变量
71.     //     double time = 0, time_;
72.     //     int result;
73.     //     /*
74.     //     // 未优化的时间
75.     //     QueryPerformanceFrequency(&frequency);
76.     //     QueryPerformanceCounter(&start); // 计时开始;
77.     //     result = DPLL(newCnf, value, 1);
78.     //     QueryPerformanceCounter(&end); //
79.     //     time = (double)(end.QuadPart - start.QuadPart) /
80.     //           frequency.QuadPart; // 计算运行时间
81.     //     // 输出SAT结果
82.     //     if (result == OK) // SAT
83.     //     {
84.     //       printf(" SAT\n\n");
85.     //       // 输出文字的真值
86.     //       for (int i = 1; i <= cnf->boolCount; i++) {
87.     //         if (value[i] == TRUE)
88.     //           printf(" %-4d: TRUE\n", i);
89.     //         else
90.     //           printf(" %-4d: FALSE\n", i);
91.     //       }
92.     //     } else // UNSAT
93.     //       printf(" UNSAT\n");
94.     //     // 输出优化前的时间
95.     //     printf("\n Time: %lf ms(not optimized)\n", time *
     1000);
96.     //     */
```

```
97.     //     // 直接使用优化后的DPLL
98.     //     int ch = 1; // 直接设置为1，表示使用优化
99.
100.      //     // 是否优化
101.      //     // int ch;
102.      //     printf("\n Do you want to optimize the algorit
    hm? (1/0): ");
103.      //     scanf("%d", &ch);
104.
105.      //     if (ch == 0) {
106.      //       time_ = 0;
107.      //     } else {
108.      //       // 重置value 数组
109.      //       for (int i = 1; i <= cnf->boolCount; i++)
110.      //         value[i] = FALSE;
111.
112.      //       QueryPerformanceFrequency(&frequency_);
113.      //       QueryPerformanceCounter(&start_); // 计时开始;
114.      //       result = DPLL(newCnf, value, 3);  // 使用优化后
    的DPLL
115.      //       QueryPerformanceCounter(&end_);   // 结束
116.      //       time_ = (double)(end_.QuadPart - start_.QuadP
    art) /
117.      //               frequency_.QuadPart; // 计算运行时间
118.
119.      //       // 输出SAT 结果
120.      //       if (result == OK) // SAT
121.      //       {
122.      //         printf(" SAT\n\n");
123.      //         // 输出文字的真值
124.      //         for (int i = 1; i <= cnf->boolCount; i++) {
125.      //           if (value[i] == TRUE)
126.      //             printf(" %-4d: TRUE\n", i);
127.      //           else
128.      //             printf(" %-4d: FALSE\n", i);
129.      //         }
130.      //       } else // UNSAT
131.      //         printf(" UNSAT\n");
132.
133.      //       printf("\n Time: %lf ms(optimized)\n", time_
    * 1000);
134.      //     }
135.
136.      //     // 是否保存
```

```
137.      //     printf("\n Save the result to file? (1/0): ");
138.      //     int choice;
139.      //     scanf("%d", &choice);
140.      //     printf("\n");
141.      //     if (choice == 1) {
142.      //       // 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）
143.      //       if (SaveResult(result, 1, time_, value, fileName, cnf->boolCount))
144.      //         printf(" Save successfully.\n");
145.      //       else
146.      //         printf(" Save failed.\n");
147.      //     }
148.
149.      //     // 释放内存
150.      //     DestroyCnf(newCnf->root);
151.      //     free(newCnf);
152.      //   }
153.      //   break;
154.      // }
155.      // display.cpp -> DisPlay() -> switch(op)
156.      case 3: {
157.        if (cnf->root == NULL) // 如果没有打开CNF文件
158.        {
159.          printf(" You haven't open the CNF file.\n");
160.          break;
161.        } else {
162.          // --- 初始化 ---
163.          value = (bool *)malloc(sizeof(bool) * (cnf->boolCount + 1));
164.          LARGE_INTEGER frequency;    // 计时器频率
165.          LARGE_INTEGER start, end;   // 时间变量
166.          double time_unoptimized = 0, time_optimized = 0;
167.          int result_unoptimized, result_optimized;
168.
169.          printf("\n--- Running Unoptimized DPLL (Strategy 1) ---\n");
170.
171.          // --- 1. 进行未优化的求解 ---
172.          // 为未优化求解创建独立的CNF副本
173.          CNF cnf_unoptimized = (CNF)malloc(sizeof(cnfNode));
174.          cnf_unoptimized->root = CopyCnf(cnf->root);
175.          cnf_unoptimized->boolCount = cnf->boolCount;
176.          cnf_unoptimized->clauseCount = cnf->clauseCount;
```

```
177.
178.          // 初始化 value 数组
179.          for (int i = 1; i <= cnf->boolCount; i++) value[i
   ] = FALSE;
180.
181.          QueryPerformanceFrequency(&frequency);
182.          QueryPerformanceCounter(&start); // 计时开始
183.          result_unoptimized = DPLL(cnf_unoptimized, value,
    1); // 使用 flag = 1
184.          QueryPerformanceCounter(&end); // 计时结束
185.          time_unoptimized = (double)(end.QuadPart - start.
   QuadPart) / frequency.QuadPart; // 计算运行时间
186.
187.          if (result_unoptimized == OK) printf(" Result: SA
   T\n");
188.          else printf(" Result: UNSAT\n");
189.          printf(" Time: %lf ms\n", time_unoptimized * 1000);
190.
191.          DestroyCnf(cnf_unoptimized->root); // 销毁副本
192.          free(cnf_unoptimized);
193.
194.          printf("\n--- Running Optimized DPLL (Strategy 3)
    ---\n");
195.
196.          // --- 2. 进行优化后的求解 ---
197.          // 为优化求解创建独立的 CNF 副本
198.          CNF cnf_optimized = (CNF)malloc(sizeof(cnfNode));
199.          cnf_optimized->root = CopyCnf(cnf->root);
200.          cnf_optimized->boolCount = cnf->boolCount;
201.          cnf_optimized->clauseCount = cnf->clauseCount;
202.
203.          // 重新初始化 value 数组
204.          for (int i = 1; i <= cnf->boolCount; i++) value[i
   ] = FALSE;
205.
206.          QueryPerformanceFrequency(&frequency);
207.          QueryPerformanceCounter(&start); // 计时开始
208.          result_optimized = DPLL(cnf_optimized, value, 3);
    // 使用 flag = 3
209.          QueryPerformanceCounter(&end); // 计时结束
210.          time_optimized = (double)(end.QuadPart - start.Qu
   adPart) / frequency.QuadPart; // 计算运行时间
211.
212.          if (result_optimized == OK) {
```

```
213.             printf(" Result: SAT\n\n");
214.             // 打印优化后的解
215.             for (int i = 1; i <= cnf->boolCount; i++) {
216.                 if (value[i] == TRUE) printf(" %-4d: TRUE
    \n", i);
217.                 else printf(" %-4d: FALSE\n", i);
218.             }
219.         } else {
220.             printf(" Result: UNSAT\n");
221.         }
222.         printf("\n Time: %lf ms\n", time_optimized * 1000);
223.
224.         DestroyCnf(cnf_optimized->root); // 销毁副本
225.         free(cnf_optimized);
226.
227.         // --- 3. 计算并输出优化率 ---
228.         printf("\n--- Comparison ---\n");
229.         printf(" Unoptimized Time: %lf ms\n", time_unopti
    mized * 1000);
230.         printf(" Optimized Time:   %lf ms\n", time_optimi
    zed * 1000);
231.         if (time_unoptimized > 0) {
232.             double optimization_rate = ((time_unoptimized
     - time_optimized) / time_unoptimized) * 100;
233.             printf(" Optimization Rate: %.2lf%%\n", optim
    ization_rate);
234.         } else {
235.             printf(" Optimization Rate: N/A (unoptimized
    time is zero)\n");
236.         }
237.
238.         // --- 4. 保存结果 ---
239.         printf("\n Save the result to file? (1/0): ");
240.         int choice;
241.         scanf("%d", &choice);
242.         printf("\n");
243.         if (choice == 1) {
244.             // 保存优化后的求解结果，并传入两个时间用于记录
245.             if (SaveResult(result_optimized, time_unoptim
    ized, time_optimized, value, fileName, cnf->boolCount))
246.                 printf(" Save successfully.\n");
247.             else
248.                 printf(" Save failed.\n");
249.         }
```

```
250.
251.           // 释放 value 数组
252.           free(value);
253.           value = NULL;
254.         }
255.         break;
256.       }
257.       case 4: {
258.         Sudoku();    // X 数独界面
259.         PrintMenu(); // 跳转回来时重新打印菜单
260.         break;
261.       }
262.       case 0: {
263.         printf(" Exit successfully.\n");
264.         return; // 退出
265.       }
266.       default: {
267.         printf(" Invalid input.\n"); // 无效输入
268.         break;
269.       }
270.       }
271.     }
272.     if (cnf->root != NULL)
273.       DestroyCnf(cnf->root); // 退出时销毁 CNF
274.     free(cnf);
275.     return;
276.   }
277.
278.   // 打印菜单
279.   void PrintMenu() {
280.     printf("|================Menu for SAT================|\
    n");
281.     printf("|--------------------------------------------|\
    n");
282.     printf("|           1. Open the CNF file             |\n");
283.     printf("|     2. Traverse and output each clause     |\n");
284.     printf("|   3. Solve using DPLL and save the result  |\n");
285.     printf("|              4. Sudoku game                |\n");
286.     printf("|                 0.  EXIT                   |\n")
    ;
287.     printf("|============================================|\
    n\n");
288.   }
```

Cnfparser.cpp

```
1. /*----------------------------cnfparser-------------------------
   --------*/
2.
3. #include "SAT.hpp"
4.
5. // 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
6.
7. status ReadFile(CNF &cnf, char fileName[]) {
8.   FILE *fp = fopen(fileName, "r");
9.   while (fp == NULL) {
10.    printf(" File not found, please input again: ");
11.    scanf("%s", fileName);
12.    fp = fopen(fileName, "r");
13.  }
14.  char ch;
15.  // 跳过注释
16.  while ((ch = getc(fp)) == 'c') {
17.    while ((ch = getc(fp)) != '\n')
18.      continue;
19.  }
20.  // 跳过 p cnf
21.  getc(fp);
22.  getc(fp);
23.  getc(fp);
24.  getc(fp);
25.  fscanf(fp, "%d%d", &cnf->boolCount,
26.          &cnf->clauseCount); // 读取布尔变元个数和子句个
   数 // 初始化布尔变元个数
27.  cnf->root = NULL;              // 初始化 CNF
28.  clauseList lastClause = NULL; // 用于记录上一个子句
29.  for (int i = 0; i < cnf->clauseCount; i++) {
30.    // 读取子句
31.    clauseList newClause = (clauseList)malloc(sizeof(clause
   Node));
32.    newClause->head = NULL;
33.    newClause->next = NULL;
34.    literalList lastLiteral = NULL; // 用于记录上一个文字
35.    int number;                     // 读取文字
36.    fscanf(fp, "%d", &number);
37.    while (number != 0) {
```

```
38.       literalList newLiteral = (literalList)malloc(sizeof(l
   iteralNode));
39.       newLiteral->literal = number;
40.       newLiteral->next = NULL;
41.       if (newClause->head == NULL) // 如果是第一个文字
42.         newClause->head = newLiteral;
43.       else
44.         lastLiteral->next = newLiteral;
45.       lastLiteral = newLiteral; // 更新上一个文字
46.       fscanf(fp, "%d", &number);
47.     }
48.     if (cnf->root == NULL) // 如果是第一个子句
49.       cnf->root = newClause;
50.     else
51.       lastClause->next = newClause;
52.     lastClause = newClause; // 更新上一个子句
53.   }
54.   fclose(fp);
55.   return OK;
56.}
57.
58.// 销毁给定的CNF 文件
59.
60.status DestroyCnf(clauseList &cL) {
61.   while (cL != NULL) {
62.     clauseList tempClause = cL;
63.     cL = cL->next;                    // 指向下一个子句
64.     literalList p = tempClause->head; // 指向当前子句的第一个
   文字
65.     while (p != NULL) {
66.       literalList tempLiteral = p;
67.       p = p->next;
68.       free(tempLiteral); // 释放文字
69.     }
70.     free(tempClause); // 释放子句
71.   }
72.   cL = NULL; // cL 指向NULL
73.   return OK;
74.}
75.
76.// 打印给定的CNF 文件
77.
78.status PrintCnf(CNF cnf) {
79.   clauseList p = cnf->root;
```

```
80.   if (p == NULL) // 如果没有子句
81.   {
82.     printf(" No clauses.\n");
83.     return ERROR;
84.   }
85.   printf("  The CNF is:\n");
86.   printf("  boolCount:%d\n", cnf->boolCount);     // 打印布尔变元个数
87.   printf("  clauseCount:%d\n", cnf->clauseCount); // 打印子句个数
88.   while (p) {
89.     literalList q = p->head; // 指向子句的第一个文字
90.     printf("  ");
91.     while (q) {
92.       printf("%-5d", q->literal); // 打印文字
93.       q = q->next;
94.     }
95.     printf("0\n"); // 子句结束
96.     p = p->next;   // 指向下一个子句
97.   }
98.   return OK;
99. }
```

solver.cpp

```
1. #include "SAT.hpp"
2.
3. long long cnt = 0;
4.
5. // 判断是否为单子句
6. status IsUnitClause(literalList l) {
7.   if (l != NULL && l->next == NULL) // 只有一个文字
8.     return TRUE;
9.   else
10.    return FALSE;
11. }
12.
13. // 找到单子句并返回该文字
14. int FindUnitClause(clauseList cL) {
15.   clauseList p = cL;
16.   while (p) {
17.     if (IsUnitClause(p->head)) // 是单子句
18.       return p->head->literal; // 返回该文字
```

```
19.     p = p->next;
20.   }
21.   return 0;
22.}
23.
24.// 销毁子句
25.status DestroyClause(clauseList &cL) {
26.   literalList p = cL->head;
27.   while (p) {
28.     literalList temp = p;
29.     p = p->next; // 指向下一个文字
30.     free(temp);  // 释放文字
31.   }
32.   free(cL);  // 释放子句
33.   cL = NULL; // cL 指向 NULL
34.   return OK;
35.}
36.
37.// 根据选择的文字化简
38.void Simplify(clauseList &cL, int literal) {
39.   clauseList pre = NULL, p = cL; // pre 指向前一个子句
40.   while (p != NULL) {
41.     bool clauseDeleted = false;          // 是否删除子句
42.     literalList lpre = NULL, q = p->head; // lpre 指向前一个文
    字
43.     while (q != NULL) {
44.       if (q->literal == literal) // 删除该子句
45.       {
46.         if (pre == NULL) // 删除的是第一个子句
47.           cL = p->next;
48.         else // 删除的不是第一个子句
49.           pre->next = p->next;
50.         DestroyClause(p);                  // 销毁该子句
51.         p = (pre == NULL) ? cL : pre->next; // 指向下一个子句
52.         clauseDeleted = true;              // 子句已删除
53.         break;
54.       } else if (q->literal == -literal) // 删除该文字
55.       {
56.         if (lpre == NULL) // 删除的是第一个文字
57.           p->head = q->next;
58.         else // 删除的不是第一个文字
59.           lpre->next = q->next;
60.         free(q);                                  // 释放该文字
```

```
61.         q = (lpre == NULL) ? p->head : lpre->next; // 指向下
    一个文字
62.       } else                                        // 未删除
63.       {
64.         lpre = q;
65.         q = q->next;
66.       }
67.     }
68.     if (!clauseDeleted) // 子句未删除
69.     {
70.       pre = p;
71.       p = p->next;
72.     }
73.   }
74.}
75.
76.// 复制 cnf
77.
78.clauseList CopyCnf(clauseList cL) {
79.   // 初始化新的 CNF
80.   clauseList newCnf = (clauseList)malloc(sizeof(clauseNode));
81.   clauseList lpa, lpb;  // lpa 指向新的子句,lpb 指向旧的子句
82.   literalList tpa, tpb; // tpa 指向新的文字,tpb 指向旧的文字
83.   newCnf->head = (literalList)malloc(sizeof(literalNode));
84.   newCnf->next = NULL;
85.   newCnf->head->next = NULL;
86.   for (lpb = cL, lpa = newCnf; lpb != NULL; lpb = lpb->next
    , lpa = lpa->next) {
87.     for (tpb = lpb->head, tpa = lpa->head; tpb != NULL;
88.          tpb = tpb->next, tpa = tpa->next) {
89.       tpa->literal = tpb->literal;
90.       tpa->next = (literalList)malloc(sizeof(literalNode));
91.       tpa->next->next = NULL;
92.       if (tpb->next == NULL) // 旧的子句中的文字已经复制完
93.       {
94.         free(tpa->next);
95.         tpa->next = NULL;
96.       }
97.     }
98.     lpa->next = (clauseList)malloc(sizeof(clauseNode));
99.     lpa->next->head = (literalList)malloc(sizeof(literalNod
    e));
100.       lpa->next->next = NULL;
101.       lpa->next->head->next = NULL;
```

```
102.       if (lpb->next == NULL) // 旧的CNF中的子句已经复制完
103.       {
104.         free(lpa->next->head);
105.         free(lpa->next);
106.         lpa->next = NULL;
107.       }
108.     }
109.     return newCnf;
110.   }
111.
112.   // 选择文字(第一个文字)
113.   int ChooseLiteral_1(CNF cnf) { return cnf->root->head->l
   iteral; }
114.
115.   //(没有单子句时的策略)选择文字(出现次数最多的文字)
116.   int ChooseLiteral_2(CNF cnf) {
117.     clauseList lp = cnf->root;
118.     literalList dp;
119.     int *count, MaxWord,
120.         max; // count记录每个文字出现次数,MaxWord记录出现最
   多次数的文字
121.     count = (int *)malloc(sizeof(int) * (cnf->boolCount *
   2 + 1));
122.     for (int i = 0; i <= cnf->boolCount * 2; i++)
123.       count[i] = 0; // 初始化
124.     // 计算子句中各文字出现次数
125.     for (lp = cnf->root; lp != NULL; lp = lp->next) {
126.       for (dp = lp->head; dp != NULL; dp = dp->next) {
127.         if (dp->literal > 0) // 正文字
128.           count[dp->literal]++;
129.         else
130.           count[cnf->boolCount - dp->literal]++; // 负文字
131.       }
132.     }
133.     max = 0;
134.     // 找到出现次数最多的正文字
135.     for (int i = 1; i <= cnf->boolCount; i++) {
136.       if (max < count[i]) {
137.         max = count[i];
138.         MaxWord = i;
139.       }
140.     }
141.     if (max == 0) {
142.       // 若没有出现正文字,找到出现次数最多的负文字
```

```
143.       for (int i = cnf->boolCount + 1; i <= cnf->boolCount
    * 2; i++) {
144.         if (max < count[i]) {
145.           max = count[i];
146.           MaxWord = cnf->boolCount - i;
147.         }
148.       }
149.     }
150.     free(count);
151.     return MaxWord;
152.   }
153.
154.   // 选择最短子句中出现次数最多的文字
155.   int ChooseLiteral_3(CNF cnf) {
156.     clauseList p = cnf->root;
157.     int *count = (int *)calloc(cnf->boolCount * 2 + 1, siz
   eof(int));
158.     int minSize = INT_MAX; // 初始化为大于可能的最大子句长度
159.     int literal = 0;
160.     clauseList temp = NULL;
161.     // 遍历子句，找到最小子句并统计其文字
162.     while (p != NULL) {
163.       literalList q = p->head;
164.       int clauseSize = 0;
165.       while (q != NULL) {
166.         clauseSize++;
167.         q = q->next;
168.       }
169.       if (clauseSize < minSize) {
170.         minSize = clauseSize; // 更新最小子句大小
171.         temp = p;
172.       }
173.       p = p->next;
174.     }
175.     // 遍历子句，统计最小子句中各文字出现次数
176.     literalList q = temp->head;
177.     while (q != NULL) {
178.       count[q->literal + cnf->boolCount]++;
179.       q = q->next;
180.     }
181.     // 找到最频繁的文字
182.     int maxCount = 0;
183.     for (int i = 0; i < cnf->boolCount * 2 + 1; i++) {
184.       if (count[i] > maxCount) {
```

```
185.         maxCount = count[i];
186.         literal = i - cnf->boolCount;
187.       }
188.     }
189.     free(count);
190.     return literal;
191.   }
192.
193.   // 是否满足
194.   status Satisfy(clauseList cL) {
195.     if (cL == NULL)
196.       return OK;
197.     else
198.       return ERROR;
199.   }
200.
201.   // 是否有空子句
202.   status EmptyClause(clauseList cL) {
203.     clauseList p = cL;
204.     while (p) {
205.       if (p->head == NULL) // 空子句，返回UNSAT
206.         return TRUE;
207.       p = p->next;
208.     }
209.     return FALSE;
210.   }
211.
212.   // DPLL 算法求解 SAT 问题
213.   status DPLL(CNF cnf, bool value[], int flag) {
214.     /*1.单子句规则*/
215.     int unitLiteral = FindUnitClause(cnf->root); // 找单子
    句
216.     while (unitLiteral != 0) {
217.       cnt++;
218.       if (cnt % 10000 == 0 && cnt != 0)
219.         printf("Round:%d\n", cnt);
220.       value[abs(unitLiteral)] = (unitLiteral > 0) ? TRUE :
     FALSE;
221.       Simplify(cnf->root, unitLiteral); // 删句子(true)或者
    删文字(false)
222.       // 终止条件
223.       if (Satisfy(cnf->root) == OK)
224.         return OK;
225.       if (EmptyClause(cnf->root) == TRUE)
```

```
226.         return ERROR;
227.       unitLiteral = FindUnitClause(cnf->root);
228.     }
229.     /*2.选择一个未赋值的文字*/
230.     int literal;
231.     if (flag == 1)
232.       literal = ChooseLiteral_1(cnf); // 未优化
233.     else if (flag == 2)
234.       literal = ChooseLiteral_2(cnf); // 优化
235.     else
236.       literal = ChooseLiteral_3(cnf);
237.     /*3.将该文字赋值为真，递归求解*/
238.     CNF newCnf = (CNF)malloc(sizeof(cnfNode));
239.     newCnf->root = CopyCnf(cnf->root); // 复制CNF
240.     newCnf->boolCount = cnf->boolCount;
241.     newCnf->clauseCount = cnf->clauseCount;
242.     clauseList p = (clauseList)malloc(sizeof(clauseNode));
243.     p->head = (literalList)malloc(sizeof(literalNode));
244.     p->head->literal = literal;
245.     p->head->next = NULL;
246.     p->next = newCnf->root;
247.     newCnf->root = p; // 插入到表头
248.     if (DPLL(newCnf, value, flag) == 1)
249.       return 1; // 在第一分支中搜索
250.     DestroyCnf(newCnf->root);
251.     /*4.将该文字赋值为假，递归求解*/
252.     clauseList q = (clauseList)malloc(sizeof(clauseNode));
253.     q->head = (literalList)malloc(sizeof(literalNode));
254.     q->head->literal = -literal;
255.     q->head->next = NULL;
256.     q->next = cnf->root;
257.     cnf->root = q;                    // 插入到表头
258.     status re = DPLL(cnf, value, flag); // 回溯到执行分支策
    略的初态进入另一分支
259.     // DestroyCnf(cL);
260.     return re;
261.   }
262.
263.   // 保存求解结果
264.   // status SaveResult(int result, double time, double tim
    e_, bool value[],
265.   //                   char fileName[], int boolCount) {
266.   //   FILE *fp;
267.   //   char name[100];
```

```
268.   //   for (int i = 0; fileName[i] != '\0'; i++) {
269.   //     // 修改拓展名.res
270.   //     if (fileName[i] == '.' && fileName[i + 4] == '\0') {
271.   //       name[i] = '.';
272.   //       name[i + 1] = 'r';
273.   //       name[i + 2] = 'e';
274.   //       name[i + 3] = 's';
275.   //       name[i + 4] = '\0';
276.   //       break;
277.   //     }
278.   //     name[i] = fileName[i];
279.   //   }
280.   //   if (fopen_s(&fp, name, "w")) {
281.   //     printf(" Fail!\n");
282.   //     return ERROR;
283.   //   }
284.
285.   //   // 按照要求格式输出
286.   //   fprintf(fp, "s %d\n", result); // 求解结果
287.
288.   //   if (result == 1) {
289.   //     fprintf(fp, "v");
290.   //     // 确保输出所有变元的取值
291.   //     for (int i = 1; i <= boolCount; i++) {
292.   //       // 检查变元是否被赋值
293.   //       if (value[i] == TRUE || value[i] == FALSE) {
294.   //         if (value[i] == TRUE)
295.   //           fprintf(fp, " %d", i);
296.   //         else
297.   //           fprintf(fp, " %d", -i);
298.   //       } else {
299.   //         // 如果变元没有被赋值，输出一个默认值（例如 TRUE）
300.   //         fprintf(fp, " %d", i);
301.   //       }
302.   //     }
303.   //     fprintf(fp, "\n");
304.   //   }
305.
306.   //   // 输出优化后的执行时间（毫秒）
307.   //   fprintf(fp, "t %.0f\n", time_ * 1000);
308.
309.   //   // 保留未优化时间和优化率信息（作为注释）
310.   //   if (time != 0) {
311.   //     fprintf(fp, "c 未优化时间: %.0fms\n", time * 1000);
```

```
312.    //     double optimization_rate = ((time - time_) / time
    ) * 100;
313.    //     fprintf(fp, "c 优化
    率: %.2lf%%\n", optimization_rate);
314.    //   }
315.
316.    //   fclose(fp);
317.    //   return OK;
318.    // }
319.    status SaveResult(int result, double time_unoptimized, d
    ouble time_optimized,
320.                      bool value[], char fileName[], int boo
    lCount) {
321.        FILE *fp;
322.        char name[100];
323.        int len = strlen(fileName);
324.
325.        // 健壮地生成.res 文件名，替换.cnf 后缀
326.        strcpy_s(name, sizeof(name), fileName);
327.        if (len > 4 && strcmp(name + len - 4, ".cnf") == 0) {
328.            name[len - 3] = 'r';
329.            name[len - 2] = 'e';
330.            name[len - 1] = 's';
331.        } else {
332.            // 如果原始文件名不以.cnf 结尾，则直接追加.res
333.            strcat_s(name, sizeof(name), ".res");
334.        }
335.
336.        // 使用fopen_s 安全地打开文件
337.        if (fopen_s(&fp, name, "w")) {
338.            printf(" Fail to open result file!\n");
339.            return ERROR;
340.        }
341.
342.        // 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]
343.        fprintf(fp, "s %d\n", result);
344.
345.        // 2. 如果可满足，写入每个变元的赋值序
    列 (v ...) [cite: 151, 153]
346.        if (result == OK) {
347.            fprintf(fp, "v");
348.            for (int i = 1; i <= boolCount; i++) {
349.                // 根据value 数组的真值，输出正整数或负整数
```

```
350.                 if (value[i] == TRUE) {
351.                     fprintf(fp, " %d", i);
352.                 } else {
353.                     fprintf(fp, " %d", -i);
354.                 }
355.             }
356.             fprintf(fp, " 0\n"); // 按照惯例，以0作为赋值行的
    结束标志
357.         }
358.
359.         // 3. 写入优化后的求解时间，以毫秒为单
    位 (t ...) [cite: 151, 154]
360.         fprintf(fp, "t %.0f\n", time_optimized * 1000);
361.
362.         // 4. 以注释形式写入性能对比的详细信息，便于分析
363.         fprintf(fp, "c === Performance Analysis ===\n");
364.         fprintf(fp, "c Unoptimized Time: %.3f ms\n", time_un
    optimized * 1000);
365.         fprintf(fp, "c Optimized Time:   %.3f ms\n", time_op
    timized * 1000);
366.
367.         // 增加一个检查，防止未优化时间为0时出现除零错误
368.         if (time_unoptimized > 1e-9) {
369.             double optimization_rate = ((time_unoptimized -
    time_optimized) / time_unoptimized) * 100;
370.             fprintf(fp, "c Optimization Rate: %.2f%%\n", opt
    imization_rate);
371.         } else {
372.             fprintf(fp, "c Optimization Rate: N/A (Unoptimiz
    ed time is zero)\n");
373.         }
374.
375.         fclose(fp);
376.         return OK;
377.     }
```

%-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, isFix
    ed, 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.
73.// 打印X数独菜单
74.void PrintMenu_X() {
75.   printf("|**************Menu for X-Sudoku(Percent Sudoku)*
   **************|\n");
76.   printf("|-----------------------------------------------
   --------------|\n");
77.   printf("|             1. Generate a X-Sudoku
             |\n");
78.   printf("|             2. Play the X-Sudoku
             |\n");
79.   printf("|             3. Reference answer
             |\n");
80.   printf("|                  0. EXIT
           |\n");
81.   printf(
```

```
82.       "|**************************************************
   **********|\n\n");
83.}
84.
85.// 生成数独
86.status Generate_Sudoku(int board[SIZE + 1][SIZE + 1],
87.                       int newBoard[SIZE + 1][SIZE + 1],
88.                       int newBoard2[SIZE + 1][SIZE + 1],
89.                       bool isFixed[SIZE + 1][SIZE + 1], in
   t num,
90.                       bool value[SIZE * SIZE * SIZE + 1]) {
91.  char name[100] = "Sudoku.cnf"; // 文件名
92.START:
93.  srand(time(NULL));
94.  // 初始化棋盘
95.  for (int i = 1; i <= SIZE; i++)
96.    for (int j = 1; j <= SIZE; j++) {
97.      board[i][j] = 0;
98.      newBoard[i][j] = 0;
99.      newBoard2[i][j] = 0;
100.         isFixed[i][j] = FALSE; // 初始化为非固定数字
101.       }
102.
103.    // 先填充两个窗口
104.    int windows[2][2] = {{2, 2}, {6, 6}};
105.    for (int w = 0; w < 2; w++) {
106.      int startRow = windows[w][0];
107.      int startCol = windows[w][1];
108.      Fill_Box(board, newBoard, newBoard2, startRow, start
   Col);
109.    }
110.
111.    WriteToFile(board, 27, name); // 将数独约束条件写入文件
112.    CNF p = (CNF)malloc(sizeof(cnfNode));
113.    p->root = NULL;
114.    if (ReadFile(p, name) != OK) {
115.      printf("Failed to read CNF file\n");
116.      return ERROR;
117.    }
118.    for (int i = 1; i <= SIZE * SIZE * SIZE; i++)
119.      value[i] = FALSE;
120.    if (DPLL(p, value, 3) == ERROR) { // 求解数独
121.      printf("DPLL failed, regenerating...\n");
122.      DestroyCnf(p->root);
```

```
123.       free(p);
124.       goto START;
125.     }
126.
127.     // 将DPLL 的结果填入数独
128.     for (int i = 1; i <= SIZE * SIZE * SIZE; i++) {
129.       if (value[i] == TRUE) {
130.         int row = (i - 1) / (SIZE * SIZE) + 1;
131.         int col = (i - 1) / SIZE % SIZE + 1;
132.         int v = (i - 1) % SIZE + 1;
133.         board[row][col] = v;
134.         newBoard[row][col] = v;
135.         newBoard2[row][col] = v;
136.       }
137.     }
138.
139.     // 验证数独是否满足所有约束
140.     if (!Validate_Sudoku(board)) {
141.       printf(
142.           "Generated Sudoku doesn't satisfy all constraint
   s, regenerating...\n");
143.       DestroyCnf(p->root);
144.       free(p);
145.       goto START;
146.     }
147.
148.     // 标记所有格子为固定（提示数）
149.     for (int i = 1; i <= SIZE; i++) {
150.       for (int j = 1; j <= SIZE; j++) {
151.         isFixed[i][j] = TRUE;
152.       }
153.     }
154.
155.     // 挖洞,剩下num 个提示数
156.     int remove = 81 - num;
157.     int attempts = 0;
158.     int maxAttempts = 1000; // 防止无限循环
159.
160.     while (remove > 0 && attempts < maxAttempts) {
161.       int row = rand() % SIZE + 1;
162.       int col = rand() % SIZE + 1;
163.
164.       // 检查是否在窗口中
165.       bool inWindow = false;
```

```
166.       for (int w = 0; w < 2; w++) {
167.         int startRow = windows[w][0];
168.         int startCol = windows[w][1];
169.         if (row >= startRow && row < startRow + 3 && col >
   = startCol &&
170.             col < startCol + 3) {
171.           inWindow = true;
172.           break;
173.         }
174.       }
175.
176.       // 确保每个窗口至少保留 3 个数字
177.       if (inWindow) {
178.         int windowCount = 0;
179.         for (int w = 0; w < 2; w++) {
180.           int startRow = windows[w][0];
181.           int startCol = windows[w][1];
182.           if (row >= startRow && row < startRow + 3 && col
    >= startCol &&
183.               col < startCol + 3) {
184.             // 计算当前窗口中的提示数数量
185.             windowCount = 0;
186.             for (int i = startRow; i < startRow + 3; i++) {
187.               for (int j = startCol; j < startCol + 3; j++) {
188.                 if (isFixed[i][j])
189.                   windowCount++;
190.               }
191.             }
192.             break;
193.           }
194.         }
195.
196.         // 如果窗口中的提示数已经少于 3 个，不再挖洞
197.         if (windowCount <= 3) {
198.           attempts++;
199.           continue;
200.         }
201.       }
202.
203.       // 挖洞
204.       if (isFixed[row][col]) {
205.         int temp = board[row][col];
206.         board[row][col] = 0;
207.         newBoard[row][col] = 0;
```

```
208.         isFixed[row][col] = FALSE;
209.
210.         // 检查挖洞后是否仍有唯一解
211.         if (!HasUniqueSolution(board, isFixed, newBoard2))
    {
212.           // 如果没有唯一解，恢复这个洞
213.           board[row][col] = temp;
214.           newBoard[row][col] = temp;
215.           isFixed[row][col] = TRUE;
216.         } else {
217.           remove--;
218.         }
219.       }
220.
221.       attempts++;
222.     }
223.
224.     // 如果还有洞需要挖，但已达到最大尝试次数
225.     if (remove > 0) {
226.       printf(" Warning: Could not remove all requested hol
   es while maintaining "
227.              "window constraints.\n");
228.     }
229.
230.     DestroyCnf(p->root);
231.     free(p);
232.     return OK;
233.   }
234.
235.   // 验证数独是否满足所有约束
236.   status Validate_Sudoku(int board[SIZE + 1][SIZE + 1]) {
237.     // 检查行
238.     for (int i = 1; i <= SIZE; i++) {
239.       bool used[SIZE + 1] = {false};
240.       for (int j = 1; j <= SIZE; j++) {
241.         if (board[i][j] != 0) {
242.           if (used[board[i][j]])
243.             return FALSE;
244.           used[board[i][j]] = true;
245.         }
246.       }
247.     }
248.
249.     // 检查列
```

```
250.     for (int j = 1; j <= SIZE; j++) {
251.       bool used[SIZE + 1] = {false};
252.       for (int i = 1; i <= SIZE; i++) {
253.         if (board[i][j] != 0) {
254.           if (used[board[i][j]])
255.             return FALSE;
256.           used[board[i][j]] = true;
257.         }
258.       }
259.     }
260.
261.     // 检查3x3 宫格
262.     for (int boxRow = 1; boxRow <= SIZE; boxRow += 3) {
263.       for (int boxCol = 1; boxCol <= SIZE; boxCol += 3) {
264.         bool used[SIZE + 1] = {false};
265.         for (int i = boxRow; i < boxRow + 3; i++) {
266.           for (int j = boxCol; j < boxCol + 3; j++) {
267.             if (board[i][j] != 0) {
268.               if (used[board[i][j]])
269.                 return FALSE;
270.               used[board[i][j]] = true;
271.             }
272.           }
273.         }
274.       }
275.     }
276.
277.     // 检查副对角线
278.     bool usedDiag[SIZE + 1] = {false};
279.     for (int i = 1; i <= SIZE; i++) {
280.       int j = SIZE + 1 - i;
281.       if (board[i][j] != 0) {
282.         if (usedDiag[board[i][j]])
283.           return FALSE;
284.         usedDiag[board[i][j]] = true;
285.       }
286.     }
287.
288.     // 检查两个窗口
289.     int windows[2][2] = {{2, 2}, {6, 6}};
290.     for (int w = 0; w < 2; w++) {
291.       int startRow = windows[w][0];
292.       int startCol = windows[w][1];
293.       bool used[SIZE + 1] = {false};
```

```
294.
295.       for (int i = startRow; i < startRow + 3; i++) {
296.         for (int j = startCol; j < startCol + 3; j++) {
297.           if (board[i][j] != 0) {
298.             if (used[board[i][j]])
299.               return FALSE;
300.             used[board[i][j]] = true;
301.           }
302.         }
303.       }
304.     }
305.
306.     return TRUE;
307.   }
308.
309.   // // 检查数独是否有唯一解
310.   // status HasUniqueSolution(int board[SIZE + 1][SIZE + 1],
311.   //                          bool isFixed[SIZE + 1][SIZE + 1],
312.   //                          int solution[SIZE + 1][SIZE +
   1]) {
313.   //   // 创建临时数独副本
314.   //   int tempBoard[SIZE + 1][SIZE + 1];
315.   //   for (int i = 1; i <= SIZE; i++) {
316.   //     for (int j = 1; j <= SIZE; j++) {
317.   //       tempBoard[i][j] = board[i][j];
318.   //     }
319.   //   }
320.
321.   //   // 使用DPLL 求解数独
322.   //   char name[100] = "temp_Sudoku.cnf";
323.   //   WriteToFile(tempBoard, 0, name);
324.
325.   //   CNF p = (CNF)malloc(sizeof(cnfNode));
326.   //   p->root = NULL;
327.   //   if (ReadFile(p, name) != OK) {
328.   //     printf("Failed to read temporary CNF file\n");
329.   //     return FALSE;
330.   //   }
331.
332.   //   bool value[SIZE * SIZE * SIZE + 1];
333.   //   for (int i = 1; i <= SIZE * SIZE * SIZE; i++)
334.   //     value[i] = FALSE;
335.
336.   //   status result = DPLL(p, value, 3);
```

```
337.
338.   //   if (result == OK) {
339.   //     // 将解填入 solution 数组
340.   //     for (int i = 1; i <= SIZE * SIZE * SIZE; i++) {
341.   //       if (value[i] == TRUE) {
342.   //         int row = (i - 1) / (SIZE * SIZE) + 1;
343.   //         int col = (i - 1) / SIZE % SIZE + 1;
344.   //         int v = (i - 1) % SIZE + 1;
345.   //         solution[row][col] = v;
346.   //       }
347.   //     }
348.   //   }
349.
350.   //   DestroyCnf(p->root);
351.   //   free(p);
352.   //   remove(name); // 删除临时文件
353.
354.   //   return result == OK;
355.   // }
356.   // [%-Sudoku.cpp]
357.   // 请用这个新版本替换原有的 HasUniqueSolution 函数
358.
359.   status HasUniqueSolution(int board[SIZE + 1][SIZE + 1],
360.                            bool isFixed[SIZE + 1][SIZE + 1],
361.                            int solution[SIZE + 1][SIZE + 1
   ]) {
362.       char name[100] = "temp_Sudoku_uniqueness_check.cnf";
363.       bool firstSolution[SIZE * SIZE * SIZE + 1];
364.       status finalResult = FALSE; // 最终返回结果，默认为非唯
   一解
365.
366.       // ==================== 阶段一：第一次求解，寻找第一个
   解 ====================
367.       CNF cnf1 = (CNF)malloc(sizeof(cnfNode));
368.       cnf1->root = NULL;
369.       bool value1[SIZE * SIZE * SIZE + 1];
370.       for (int i = 1; i <= SIZE * SIZE * SIZE; i++) value1
   [i] = FALSE;
371.
372.       WriteToFile(board, 0, name); // 将当前棋盘转化为CNF
373.       if (ReadFile(cnf1, name) != OK) {
374.           printf("Error: Failed to read temp CNF for first
    solve.\n");
375.           free(cnf1);
```

```
376.             remove(name);
377.             return FALSE; // 文件读取失败，无法验证
378.         }
379.
380.         status result1 = DPLL(cnf1, value1, 3); // 调用DPLL
    求解
381.
382.         if (result1 == ERROR) {
383.             // 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
384.             DestroyCnf(cnf1->root);
385.             free(cnf1);
386.             remove(name);
387.             return FALSE;
388.         }
389.
390.         // 保存第一个解
391.         for (int i = 1; i <= SIZE * SIZE * SIZE; i++) {
392.             firstSolution[i] = value1[i];
393.         }
394.
395.         // 如果找到了解，将其解码并存入 solution 输出参数
396.         for (int i = 1; i <= SIZE * SIZE * SIZE; i++) {
397.             if (firstSolution[i] == TRUE) {
398.                 int row = (i - 1) / (SIZE * SIZE) + 1;
399.                 int col = (i - 1) / SIZE % SIZE + 1;
400.                 int v = (i - 1) % SIZE + 1;
401.                 solution[row][col] = v;
402.             }
403.         }
404.
405.         DestroyCnf(cnf1->root); // 释放第一次求解的 CNF 结构
406.         free(cnf1);
407.
408.
409.         // ==================== 阶段二：增加“阻塞子句”，尝试寻
    找第二个解 ====================
410.         // 构建一个“阻塞子句”，这个子句是第一个解的逻辑否定
411.         clauseList blockingClause = (clauseList)malloc(sizeo
    f(clauseNode));
412.         blockingClause->head = NULL;
413.         blockingClause->next = NULL;
414.         literalList lastLiteral = NULL;
415.
```

```
416.        for (int i = 1; i <= SIZE * SIZE * SIZE; i++) {
417.            literalList newLiteral = (literalList)malloc(siz
    eof(literalNode));
418.            // 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
419.            newLiteral->literal = (firstSolution[i] == TRUE)
     ? -i : i;
420.            newLiteral->next = NULL;
421.            if (blockingClause->head == NULL) {
422.                blockingClause->head = newLiteral;
423.            } else {
424.                lastLiteral->next = newLiteral;
425.            }
426.            lastLiteral = newLiteral;
427.        }
428.
429.        // 重新读取原始 CNF 问题，并加入阻塞子句
430.        CNF cnf2 = (CNF)malloc(sizeof(cnfNode));
431.        cnf2->root = NULL;
432.        bool value2[SIZE * SIZE * SIZE + 1]; // 第二次求解的结
    果数组（虽然内容不重要）
433.
434.        if (ReadFile(cnf2, name) != OK) {
435.            printf("Error: Failed to read temp CNF for secon
    d solve.\n");
436.            DestroyClause(blockingClause); // 清理创建的阻塞子
    句
437.            free(cnf2);
438.            remove(name);
439.            return FALSE; // 文件读取失败
440.        }
441.
442.        // 将阻塞子句添加到 CNF 公式的头部
443.        blockingClause->next = cnf2->root;
444.        cnf2->root = blockingClause;
445.        cnf2->clauseCount++;
446.
447.        status result2 = DPLL(cnf2, value2, 3); // 进行第二次
    求解
448.
449.        if (result2 == ERROR) {
450.            // 如果在排除了第一个解之后，问题变为“不可满足”，
451.            // 这恰好证明了第一个解是唯一的。
452.            finalResult = TRUE;
```

```
453.        } else {
454.            // 如果还能找到另一个解，说明解不唯一。
455.            finalResult = FALSE;
456.        }
457.
458.        DestroyCnf(cnf2->root); // 释放第二次求解的CNF 结构（这
    会同时释放阻塞子句）
459.        free(cnf2);
460.        remove(name); // 删除临时文件
461.
462.        return finalResult;
463.    }
464.    // 判断board[row][col]是否可以填入 v
465.
466.    status Is_Valid(int board[SIZE + 1][SIZE + 1], int row,
    int col, int v) {
467.      // 检查行
468.      for (int j = 1; j <= SIZE; j++) {
469.        if (j != col && board[row][j] == v) {
470.          return FALSE;
471.        }
472.      }
473.
474.      // 检查列
475.      for (int i = 1; i <= SIZE; i++) {
476.        if (i != row && board[i][col] == v) {
477.          return FALSE;
478.        }
479.      }
480.
481.      // 检查 3x3 宫格
482.      int startRow = ((row - 1) / 3) * 3 + 1;
483.      int startCol = ((col - 1) / 3) * 3 + 1;
484.      for (int i = startRow; i < startRow + 3; i++) {
485.        for (int j = startCol; j < startCol + 3; j++) {
486.          if ((i != row || j != col) && board[i][j] == v) {
487.            return FALSE;
488.          }
489.        }
490.      }
491.
492.      // 检查副对角线
493.      if (row + col == SIZE + 1) {
494.        for (int i = 1; i <= SIZE; i++) {
```

```
495.         int j = SIZE + 1 - i;
496.         if (i != row && board[i][j] == v) {
497.           return FALSE;
498.         }
499.       }
500.     }
501.
502.     // 检查两个窗口
503.     int windows[2][2] = {{2, 2}, {6, 6}};
504.     for (int w = 0; w < 2; w++) {
505.       int startRow = windows[w][0];
506.       int startCol = windows[w][1];
507.
508.       // 检查当前单元格是否在这个窗口中
509.       if (row >= startRow && row < startRow + 3 && col >=
   startCol &&
510.           col < startCol + 3) {
511.         // 检查窗口内是否有重复数字
512.         for (int i = startRow; i < startRow + 3; i++) {
513.           for (int j = startCol; j < startCol + 3; j++) {
514.             if ((i != row || j != col) && board[i][j] == v) {
515.               return FALSE;
516.             }
517.           }
518.         }
519.         break; // 找到所属窗口后跳出循环
520.       }
521.     }
522.
523.     return TRUE;
524.   }
525.
526.   // 打印数独
527.
528.   void Print_Sudoku(int board[SIZE + 1][SIZE + 1]) {
529.     printf("   ");
530.     for (int j = 1; j <= SIZE; j++) {
531.       printf("%2d ", j);
532.       if (j % 3 == 0 && j != SIZE)
533.         printf("| ");
534.     }
535.     printf("\n");
536.     printf("   ");
537.     for (int j = 1; j <= SIZE; j++) {
```

```
538.       printf("---");
539.       if (j % 3 == 0 && j != SIZE)
540.         printf("+");
541.     }
542.     printf("\n");
543.
544.     for (int i = 1; i <= SIZE; i++) {
545.       printf("%2d|", i);
546.       for (int j = 1; j <= SIZE; j++) {
547.         if (board[i][j] == 0) // 未填入
548.           printf(" . ");
549.         else // 已填入
550.           printf("%2d ", board[i][j]);
551.         if (j % 3 == 0 && j != SIZE) // 每3列打印一个竖线
552.           printf("| ");
553.       }
554.       printf("\n");
555.       if (i % 3 == 0 && i != SIZE) { // 每3行打印一个横线
556.         printf("   ");
557.         for (int j = 1; j <= SIZE; j++) {
558.           printf("---");
559.           if (j % 3 == 0 && j != SIZE)
560.             printf("+");
561.         }
562.         printf("\n");
563.       }
564.     }
565.   }
566.
567.   // 玩数独的交互界面
568.
569.   void Play_Sudoku(int board[SIZE + 1][SIZE + 1],
570.                    bool isFixed[SIZE + 1][SIZE + 1]) {
571.     system("cls");       // 清屏
572.     Print_Sudoku(board); // 打印初始数独
573.     printf("\n");
574.     while (1) {
575.       int row, col, v;
576.       printf(" Please enter the row, col and value(0 to EX
    IT): ");
577.       scanf("%d", &row);
578.       if (row == 0) // 退出
579.       {
580.         system("cls");
```

```
581.           return;
582.         }
583.         scanf("%d%d", &col, &v);
584.         if (row < 1 || row > SIZE || col < 1 || col > SIZE |
     | v < 1 ||
585.             v > SIZE) // 输入不合法
586.         {
587.           printf(" Invalid input.\n");
588.           continue;
589.         }
590.         if (isFixed[row][col]) // 是提示数
591.         {
592.           printf(" This is a fixed number.\n");
593.           continue;
594.         }
595.         if (!Is_Valid(board, row, col, v)) // 不符合数独规则
596.         {
597.           printf(" Wrong answer.\n");
598.           continue;
599.         } else // 符合数独规则
600.         {
601.           board[row][col] = v;
602.           system("cls");
603.           Print_Sudoku(board); // 打印新数独
604.           printf("\n");
605.
606.           // 检查是否完成
607.           int complete = 1;
608.           for (int i = 1; i <= SIZE; i++) {
609.             for (int j = 1; j <= SIZE; j++) {
610.               if (board[i][j] == 0) {
611.                 complete = 0;
612.                 break;
613.               }
614.             }
615.             if (!complete)
616.               break;
617.           }
618.
619.           if (complete) {
620.             if (Validate_Sudoku(board)) {
621.               printf(" Congratulations! You solved the Sudok
     u!\n");
622.             } else {
```

```
623.             printf(" Sorry, your solution is incorrect.\n");
624.           }
625.           return;
626.         }
627.       }
628.     }
629.   }
630.
631.   // 将数独约束条件写入文件
632.   status WriteToFile(int board[SIZE + 1][SIZE + 1], int nu
   m, char name[]) {
633.     FILE *fp;
634.     if (fopen_s(&fp, name, "w")) {
635.       printf(" Fail!\n");
636.       return ERROR;
637.     }
638.
639.     // 计算总子句数
640.     int clauseCount = num; // 提示数子句
641.
642.     // 每个格子必须填入一个数字: 81 个子句
643.     clauseCount += 81;
644.     // 每个格子不能填入两个数字: C(9,2)*81 = 36*81 = 2916 个子
   句
645.     clauseCount += 2916;
646.     // 行约束: 每行必须包含1-9: 9*9 = 81 个子句
647.     clauseCount += 81;
648.     // 行约束: 每行不能有重复数字: C(9,2)*9*9 = 36*81 = 2916
   个子句
649.     clauseCount += 2916;
650.     // 列约束: 每列必须包含1-9: 9*9 = 81 个子句
651.     clauseCount += 81;
652.     // 列约束: 每列不能有重复数字: C(9,2)*9*9 = 36*81 = 2916
   个子句
653.     clauseCount += 2916;
654.     // 3x3 宫格约束: 每个宫格必须包含1-9: 9*9 = 81 个子句
655.     clauseCount += 81;
656.     // 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
657.     clauseCount += 2916;
658.     // 副对角线约束: 必须包含1-9: 9 个子句
659.     clauseCount += 9;
660.     // 副对角线约束: 不能有重复数字: C(9,2) = 36 个子句
661.     clauseCount += 36;
```

```
662.     // 窗口约束: 两个窗口各必须包含1-9: 2*9 = 18 个子句
663.     clauseCount += 18;
664.     // 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
665.     clauseCount += 72;
666.
667.     fprintf(fp, "c %s\n", name);
668.     fprintf(fp, "p cnf 729 %d\n", clauseCount);
669.
670.     /*提示数约束(写在前面,便于单子句规则进行)*/
671.     for (int i = 1; i <= SIZE; i++) {
672.       for (int j = 1; j <= SIZE; j++) {
673.         if (board[i][j] != 0) {
674.           fprintf(fp, "%d 0\n",
675.                   (i - 1) * SIZE * SIZE + (j - 1) * SIZE +
    board[i][j]);
676.         }
677.       }
678.     }
679.
680.     /*每个格子的约束*/
681.     // 每个格子必须填入一个数字
682.     for (int i = 1; i <= SIZE; i++) {
683.       for (int j = 1; j <= SIZE; j++) {
684.         for (int k = 1; k <= SIZE; k++) {
685.           fprintf(fp, "%d ", (i - 1) * SIZE * SIZE + (j -
   1) * SIZE + k);
686.         }
687.         fprintf(fp, "0\n");
688.       }
689.     }
690.     // 每个格子不能填入两个数字
691.     for (int i = 1; i <= SIZE; i++) {
692.       for (int j = 1; j <= SIZE; j++) {
693.         for (int k = 1; k <= SIZE; k++) {
694.           for (int l = k + 1; l <= SIZE; l++) {
695.             fprintf(fp, "%d %d 0\n",
696.                     -((i - 1) * SIZE * SIZE + (j - 1) * SI
   ZE + k),
697.                     -((i - 1) * SIZE * SIZE + (j - 1) * SI
   ZE + l));
698.           }
699.         }
700.       }
```

```
701.     }
702.
703.     /*行约束*/
704.     // 每一行必须填入1-9
705.     for (int i = 1; i <= SIZE; i++) {
706.       for (int k = 1; k <= SIZE; k++) {
707.         for (int j = 1; j <= SIZE; j++) {
708.           fprintf(fp, "%d ", (i - 1) * SIZE * SIZE + (j -
   1) * SIZE + k);
709.         }
710.         fprintf(fp, "0\n");
711.       }
712.     }
713.     // 每一行不能填入两个相同的数字
714.     for (int i = 1; i <= SIZE; i++) {
715.       for (int k = 1; k <= SIZE; k++) {
716.         for (int j = 1; j <= SIZE; j++) {
717.           for (int l = j + 1; l <= SIZE; l++) {
718.             fprintf(fp, "%d %d 0\n",
719.                     -((i - 1) * SIZE * SIZE + (j - 1) * SI
   ZE + k),
720.                     -((i - 1) * SIZE * SIZE + (l - 1) * SI
   ZE + k));
721.           }
722.         }
723.       }
724.     }
725.
726.     /*列约束*/
727.     // 每一列必须填入1-9
728.     for (int j = 1; j <= SIZE; j++) {
729.       for (int k = 1; k <= SIZE; k++) {
730.         for (int i = 1; i <= SIZE; i++) {
731.           fprintf(fp, "%d ", (i - 1) * SIZE * SIZE + (j -
   1) * SIZE + k);
732.         }
733.         fprintf(fp, "0\n");
734.       }
735.     }
736.     // 每一列不能填入两个相同的数字
737.     for (int j = 1; j <= SIZE; j++) {
738.       for (int k = 1; k <= SIZE; k++) {
739.         for (int i = 1; i <= SIZE; i++) {
740.           for (int l = i + 1; l <= SIZE; l++) {
```

```
741.             fprintf(fp, "%d %d 0\n",
742.                     -((i - 1) * SIZE * SIZE + (j - 1) * SI
   ZE + k),
743.                     -((l - 1) * SIZE * SIZE + (j - 1) * SI
   ZE + k));
744.           }
745.         }
746.       }
747.     }
748.
749.     /*3x3 宫格约束*/
750.     // 每个3x3 宫格必须填入1-9
751.     for (int boxRow = 1; boxRow <= SIZE; boxRow += 3) {
752.       for (int boxCol = 1; boxCol <= SIZE; boxCol += 3) {
753.         for (int k = 1; k <= SIZE; k++) {
754.           for (int i = boxRow; i < boxRow + 3; i++) {
755.             for (int j = boxCol; j < boxCol + 3; j++) {
756.               fprintf(fp, "%d ", (i - 1) * SIZE * SIZE + (
   j - 1) * SIZE + k);
757.             }
758.           }
759.           fprintf(fp, "0\n");
760.         }
761.       }
762.     }
763.     // 每个3x3 宫格不能填入两个相同的数字
764.     for (int boxRow = 1; boxRow <= SIZE; boxRow += 3) {
765.       for (int boxCol = 1; boxCol <= SIZE; boxCol += 3) {
766.         for (int k = 1; k <= SIZE; k++) {
767.           for (int i = boxRow; i < boxRow + 3; i++) {
768.             for (int j = boxCol; j < boxCol + 3; j++) {
769.               for (int p = i; p < boxRow + 3; p++) {
770.                 for (int q = (p == i ? j + 1 : boxCol); q
   < boxCol + 3; q++) {
771.                   fprintf(fp, "%d %d 0\n",
772.                           -((i - 1) * SIZE * SIZE + (j - 1
   ) * SIZE + k),
773.                           -((p - 1) * SIZE * SIZE + (q - 1
   ) * SIZE + k));
774.                 }
775.               }
776.             }
777.           }
778.         }
```

```
779.       }
780.     }
781.
782.     /*副对角线约束*/
783.     // 副对角线必须包含1-9
784.     for (int k = 1; k <= SIZE; k++) {
785.       for (int i = 1; i <= SIZE; i++) {
786.         int j = SIZE + 1 - i;
787.         fprintf(fp, "%d ", (i - 1) * SIZE * SIZE + (j - 1)
    * SIZE + k);
788.       }
789.       fprintf(fp, "0\n");
790.     }
791.     // 副对角线不能有重复数字
792.     for (int k = 1; k <= SIZE; k++) {
793.       for (int i = 1; i <= SIZE; i++) {
794.         int j = SIZE + 1 - i;
795.         for (int p = i + 1; p <= SIZE; p++) {
796.           int q = SIZE + 1 - p;
797.           fprintf(fp, "%d %d 0\n", -((i - 1) * SIZE * SIZE
    + (j - 1) * SIZE + k),
798.                   -((p - 1) * SIZE * SIZE + (q - 1) * SIZE + k));
799.         }
800.       }
801.     }
802.
803.     /*两个窗口约束*/
804.     int windows[2][2] = {{2, 2}, {6, 6}};
805.     for (int w = 0; w < 2; w++) {
806.       int startRow = windows[w][0];
807.       int startCol = windows[w][1];
808.
809.       // 窗口必须包含1-9
810.       for (int k = 1; k <= SIZE; k++) {
811.         for (int i = startRow; i < startRow + 3; i++) {
812.           for (int j = startCol; j < startCol + 3; j++) {
813.             fprintf(fp, "%d ", (i - 1) * SIZE * SIZE + (j
   - 1) * SIZE + k);
814.           }
815.         }
816.         fprintf(fp, "0\n");
817.       }
818.
819.       // 窗口内不能有重复数字
```

```
820.       for (int k = 1; k <= SIZE; k++) {
821.         for (int i = startRow; i < startRow + 3; i++) {
822.           for (int j = startCol; j < startCol + 3; j++) {
823.             for (int p = i; p < startRow + 3; p++) {
824.               for (int q = (p == i ? j + 1 : startCol); q
   < startCol + 3; q++) {
825.                 fprintf(fp, "%d %d 0\n",
826.                         -((i - 1) * SIZE * SIZE + (j - 1)
   * SIZE + k),
827.                         -((p - 1) * SIZE * SIZE + (q - 1)
   * SIZE + k));
828.               }
829.             }
830.           }
831.         }
832.       }
833.     }
834.
835.     fclose(fp);
836.     return OK;
837.   }
838.
839.   // DPLL 求解数独
840.
841.   status Slove(int board[SIZE + 1][SIZE + 1],
842.                bool value[SIZE * SIZE * SIZE + 1]) {
843.     for (int i = 1; i <= SIZE * SIZE * SIZE; i++) {
844.       if (value[i] == TRUE) {
845.         int row = (i - 1) / (SIZE * SIZE) + 1;
846.         int col = (i - 1) / SIZE % SIZE + 1;
847.         int v = (i - 1) % SIZE + 1;
848.         board[row][col] = v;
849.       }
850.     }
851.     return OK;
852.   }
853.
854.   // 填充 3x3 的宫格
855.   status Fill_Box(int board[SIZE + 1][SIZE + 1], int newBo
   ard[SIZE + 1][SIZE + 1],
856.                   int newBoard2[SIZE + 1][SIZE + 1], int r
   owStart, int colStart) {
857.     int numbers[SIZE];
858.     for (int i = 0; i < SIZE; i++) {
```

```
859.       numbers[i] = i + 1;
860.     }
861.     Shuffle(numbers, SIZE);
862.
863.     int index = 0;
864.     for (int i = rowStart; i < rowStart + 3; i++) {
865.       for (int j = colStart; j < colStart + 3; j++) {
866.         if (board[i][j] == 0) {
867.           board[i][j] = numbers[index];
868.           newBoard[i][j] = numbers[index];
869.           newBoard2[i][j] = numbers[index];
870.           index++;
871.         }
872.       }
873.     }
874.     return OK;
875.   }
876.
877.   // 打乱数组顺序
878.   void Shuffle(int arr[], int n) {
879.     srand(time(NULL)); // 用时间做种子
880.     // 每次从后面的数中随机选一个数与前面的数交换
881.     for (int i = n - 1; i > 0; i--) {
882.       int j = rand() % (i + 1);
883.       int temp = arr[i];
884.       arr[i] = arr[j];
885.       arr[j] = temp;
886.     }
```