华中科技大学

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：计算机科学与技术 2407 班

学　　号：U202414802

姓　　名：朱俊瑞

指导教师：李丹

报告日期：2025.9.11

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%)

目录

任务书 I
1 引言 1
1.1 课程背景与意义 1
1.2 国内外研究现状 3
1.3 课程设计的主要研究工作 4
2 系统需求分析与总体设计 5
2.1 系统需求分析 5
2.2 系统总体设计 5
3 系统详细设计 6
3.1 有关数据结构的定义 6
3.2 主要设计算法 6
3.3 程序模块化 7
4 系统实现与测试 7
4.1 系统实现 7
4.2 系统测试 14
5 总结与展望 20
5.1 全文总结 20
5.2 工作展望 21
6 体会 22
参考文献 23
附录 24

1 引言

1.1 课题背景与意义

1.1.1 课题背景

SAT问题即命题逻辑公式的可满足性问题（satisfiability problem），是计算机科学与人工智能基本问题，是一个典型的NP完全问题，可广泛应用于许多实际问题如硬件设计、安全协议验证等，具有重要理论意义与应用价值。SAT问题也是程序设计与竞赛的经典问题。

对于任一布尔变元x，x与其非“$\neg x$”称为文字(literal)。对于多个布尔变元，若干个文字的或运算$l_1 \vee l_2 \vee \ldots \vee l_k$称为子句(clause)。只含一个文字的子句称为单子句。不含任何文字的子句称为空子句，常用符号□表示。子句所含文字越多，越易满足，空子句不可满足。

SAT问题一般可描述为：给定布尔变元集合$\{x_1, x_2, ..., x_n\}$以及相应的子句集合$\{c_1, c_2, ..., c_m\}$，对于合取范式（CNF范式）：$F = c_1 \wedge c_2 \wedge ... \wedge c_m$，判定是否存在对每个布尔变元的一组真值赋值使$F$为真，当为真时（问题是可满足的，SAT），输出对应的变元赋值（一组解）结果。

一个CNF公式也可以表示成子句集合的形式：$S = \{c_1, c_2, ..., c_m\}$.

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

一个CNF SAT公式或算例的具体信息通常存储在一个cnf 文件中，下图1.1是算例problem1.cnf文件前若干行的截图。

```
c
c SOURCE: Kazuo Iwama, Eiji Miyano,and Yuichi Asahiro
c
c DESCRIPTION: Artifical instances from generator by source.
c
c
p cnf 200 320
46 72 115 0
-46 72 130 0
-46 72 -130 0
50 -72 115 0
-50 -72 115 0
92 -95 -115 0
```

图1.1 cnf文件格式

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

DPLL算法是经典的SAT完备型求解算法，对给定的一个SAT问题实例，理论上可判定其是否满足，满足时可给出对应的一组解。本设计要求实现基于DPLL的算法与程序框架，包括程序的改进也必须在此算法的基础上进行。

DPLL 算法是基于树/二叉树的回溯搜索算法，主要使用两种基本处理策略：

单子句规则。如果子句集 S 中有一个单子句 L, 那么 L 一定取真值，于是可以从 S 中删除所有包含 L 的子句（包括单子句本身），得到子句集 S_1，如果它是空集，则 S 可满足。否则对 S_1 中的每个子句，如果它包含文字¬L, 则从该子句中去掉这个文字，这样可得到子句集合 S_2。S 可满足当且仅当 S_2 可满足。单子句传播策略就是反复利用单子句规则化简 S 的过程。

分裂策略。按某种策略选取一个文字 L. 如果 L 取真值，则根据单子句传播策略，可将 S 化成 S_2；若 L 取假值（即¬L 成立）时，S 可化成 S_1.

交错使用上述两种策略可不断地对公式化简，并最终达到终止状态，其执行过程可表示为一棵二叉搜索树, 如下图 2. 2 所示。

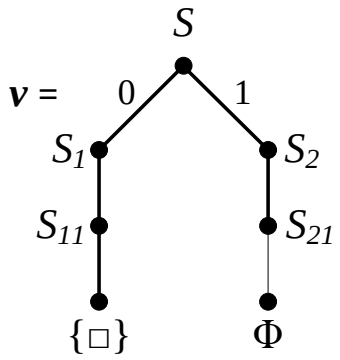

图 2.2 DPLL 算法搜索树

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

DPLL(S) :

/* S 为公式对应的子句集。若其满足，返回 TURE；否则返回 FALSE. */

```
{
  while(S 中存在单子句) {//单子句传播
      在 S 中选一个单子句 L；
      依据单子句规则，利用 L 化简 S；
      if S = Φ return(TRUE);
      else if (S 中有空子句 ) return（FALSE）；
  }//while
  基于某种策略选取变元 v；                //策略对 DPLL 性能影响很大
  if DPLL（S ∪v ）return(TURE);   //在第一分支中搜索
  return DPLL(S ∪¬v);//回溯到对 v 执行分支策略的初态进入另一分支
}
```

1.1.2 课题意义

深入理解 DPLL 算法，学会 NP 问题的求解逻辑，学习使用各种优化方式进行算法优化；养成模块化、规范化的编程习惯，通过将主控交互、CNF 解析、DPLL 求解、百分号数独处理这几个模块的代码放在不同的.h 和.c 文件里，规划好“每个模块负责什么功能”“模块之间怎么传递数据”，避免把所有代码堆在一个文件里；为后续专业学习铺垫，拓展技术应用视野，锻炼从问题分析到成果交付的完整流程能力。

1.2 国内外研究现状

在 SAT 求解算法领域，国内研究早期多集中在对 DPLL 算法核心逻辑的实现与基础优化。由于 DPLL 是 SAT 问题的经典完备型算法，国内学者最初聚焦于其关键步骤的效率提升——比如在分支变元选取策略上，不少研究提出基于“子句活跃度”“变元出现频率”的动态选择方法，相比传统的固定顺序选取，能有效减少搜索树分支数量，降低回溯次数。

在数独的 SAT 归约与求解方面，国内研究以普通 9×9 数独为基础，已形成较为成熟的约束转化方法。多数研究采用“语义编码”定义布尔变元，并将“每行、每列、3×3 盒子数字不重复”的约束系统转化为 CNF 子句集——例如，“每个单元格至少填一个数字”对应析取子句“每个单元格最多填一个数字”对应矛盾文字的合取子句，这种转化逻辑已成为国内相关课程设计和小规模研究的通用方案。对于百分号数独这类变型数独，国内研究起步稍晚，目前集中在 “额外约束的 CNF 转化”上：部分学者通过分析撇对角线、窗口的单元格分布规律，设计循环

生成算法，自动将这两类新增约束转化为规范子句，通过双重循环生成“任意两个单元格不同时填同一数字”的矛盾子句，确保约束转化的完整性。

在 SAT 求解算法领域，国外研究起步早且成果丰富，从经典 DPLL 到现代优化算法形成了完整脉络。早期，Chaff、MiniSat 等主流求解器的提出，推动了 DPLL 算法的工程化突破——例如 MiniSat 引入“冲突驱动子句学习（CDCL）”机制，在 DPLL 的回溯过程中记录冲突信息并生成新的约束子句，大幅减少重复搜索，使求解器能处理变元数上万的大规模算例。

1.3 课程设计的主要研究工作

理解 SAT 问题（布尔变元、子句、CNF 范式等）与 DPLL 算法（单子句规则、分裂策略及递归回溯流程），解析 CNF 文件格式并设计变元、子句、公式的物理存储结构（不使用 C++vector 等类库）；开发输入输出（读取 CNF 算例、输出.res 结果文件）、公式解析验证（将 CNF 数据转为内部结构并校验）、DPLL 核心求解、时间性能测量模块，还需对 DPLL 进行存储结构或分支变元选取策略优化并计算优化率；将百分号数独转化为 SAT 问题，即生成普通数独的格、行、列、3×3 盒约束子句，补充百分号数独的撇对角线与窗口约束子句，将提示数转为单子句，通过读取文件或挖洞法生成数独初始格局并集成 SAT 求解器实现求解与简单交互；准备≥18 个不同规模（小、中、大）且含可满足与不满足类型的 SAT 算例进行测试，验证功能与性能，同时遵循程序模块化、编码规范，按要求撰写设计报告并提交相关成果。

2 系统需求分析与总体设计

2.1 系统需求分析

需实现 CNF 算例文件的读取与解析，将数据转化为自定义的变元、子句、公式物理存储结构并验证解析正确性；需基于 DPLL 算法（含单子句规则、分裂策略）开发 SAT 求解模块，同步实现求解时间测量功能，还需对 DPLL 的存储结构或分支变元选取策略优化并计算优化率；需将百分号数独转化为 SAT 问题，生成普通数独（格、行、列、3×3 盒）及百分号数独（撇对角线、窗口）的约束子句，通过读取文件或挖洞法生成数独初始格局，集成求解器实现数独求解与简单交互；需输出求解结果至.res 文件，保证程序模块化、符合编码规范，按要求撰写设计报告并提交相关成果。

2.2 系统总体设计

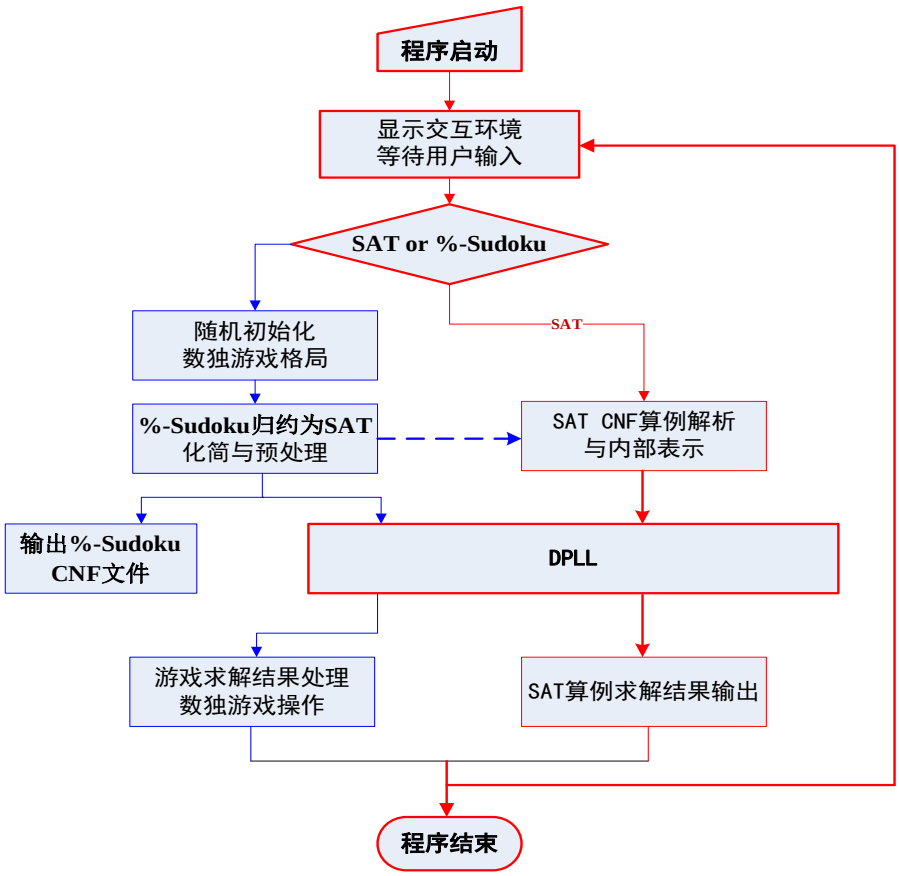

图 2-1 系统程序流程图

程序系统启动后呈现交互界面，供用户选择处理 SAT 算例或百分号数独：若为百分号数独，先随机初始化游戏格局，再将其归约为 SAT 并化简预处理（可输出 CNF 文件）；若为 SAT 算例，直接解析 CNF 文件。随后均通过 DPLL 算法求解，分别进行数独游戏结果处理或 SAT 算例求解结果输出，支持循环交互或结束程序。

3 系统详细设计

3.1 有关数据结构的定义

变元(unit)类型：链表形式表示，存储当前变元的符号，可指向下一个变元。

子句(clause或Clause)类型：链表形式表示，记录包含的变元序列，可指向下一条子句。或直接用数组存储子句中所有文字。

子句集(ClauseSet)类型：记录全部子句，同时记录子句数量。

变元-子句映射表(Litmap)类型：建立变元-子句映射表，记录存在某变元的子句在子句集中的索引，以优化DPLL过程中子句集的化简过程。

解决器(Solver)类型：解决器用于存储子句集、变元-子句映射表等信息，用于DPLL求解过程。

棋盘位置(place)类型：存储挖洞法可能挖洞的位置，包括行数和列数。

3.2 主要算法设计

3.2.1 优化前算法

采用递归实现的DPLL算法，首先在有单子句的情况下循环进行单子句传播，单子句传播结束后，选取在当前化简后所有子句中出现次数最多的变元（区分正负符号），将其取真值或假值作为单子句分别加入根据当前子句集复制的两个子句集，分别进行递归DPLL，返回可能为true（有解）的结果，否则返回false（无解）；

3.2.2 优化后算法

采用非递归实现的 DPLL 算法，首先进行一次单子句传播，随后进入循环，每次循环开始通过变元活动度大小选取活动度最高的变元，尝试将其赋值为真或假，赋值后进行单子句传播，赋值后若导致子句集不满足有解则回溯进行撤销赋

值，同时通过维护一个决策栈，记录每次选取的变元的决策，从而使得回溯以及尝试新变元可控。程序中适时进行子句集是否满足有解的判断。

3.3 程序模块化

程序源代码进行模块化组织。主要模块包括如下：

1）主控、交互与显示模块（display）；

2）CNF 解析模块（cnfparser）；

3）核心 DPLL 模块(solver)；

4）百分号数独模块，包括游戏格局生成、归约、求解(Sudoku)。

每个.h 文件对应.cpp 文件作为实现。

4 系统实现与测试

4.1 系统实现

本实验使用 VS2022，Windows11 平台下 C++语言搭建。

4.1.1 数据结构实现

变元(unit)：

```
struct unit
{
 int unitnum;
 unit* nextunit;
};
```

子句(clause 或 Clause):

```
struct clause
{
 unit* firstunit;
 clause* nextclause;
};  //优化前

 struct Clause
{
      int literals[MAXSIZE];   // 子句中的文字
```

```
    int size;                          // 文字数量
};  //优化后
```

子句集(ClauseSet):

```
struct ClauseSet
{
    Clause clauses[MAXSIZE]; // 子句数组
    int count;                         // 子句数量
};
```

变元-子句映射表(Litmap):

```
struct Litmap
{
    int* pos;   // 包含 v 的子句索引列表(子句在 clauses 数组中的索引)
    int* neg;   // 包含-v 的子句索引列表
    int pos_len;   // pos 数组长度
    int neg_len;   // neg 数组长度
};
```

解决器(Solver):

```
struct Solver
{
    int* assignment; // 变量赋值 (1-based 索引)
    int* level;          // 变量赋值的决策层
    double* activity;    // 变量活动度，用于选择下一个变量
    int* tried;
    int confictcount;
    int boolnum;                       // 变量总数
    int clausenum;                     // 子句总数

    Litmap* litmap;
```

```
        ClauseSet original;              // 原始子句集
        ClauseSet working;               // 工作子句集，用于传播和化简

        // 用于非递归实现的栈和队列
        int stack[MAXSIZE][3];       // 决策栈 [变量][值]
        int stackptr;                // 栈指针

        int queue[MAXSIZE];          // 传播队列
        int queuehead;               // 队列头指针
        int queuetail;               // 队列尾指针

        int currentlevel;            // 当前决策层
    };
```

期盘位置(place):

```
    struct place
    {
        int x,y;
    };
```

4.1.2 函数声明

(1)void scancnf(string filename,int*& boolcountarr,clause*& clausearr,Solver*& solver); //输入标准 cnf 文件

filename 是要输入的 cnf 文件名称，boolcountarr 是记录每个变元的出现次数的数组，clausearr 是存储子句的链表，solver 是指定的解决器。

打开文件后先读取以 p 开头的行，忽略以 p 开头的行的注释。

随后读取 cnf 文件里记录的变元数量和子句数量这两个数据。

在 for 循环里使用 while 循环读取并存储每个子句里存在的变元，直到读取到 0，随后进行下一次 for 循环。以此完成优化前数据结构的存储记录。

根据优化前数据结构的存储记录，进行 solver 的初始化和记录子句及变元。同样 for 循环里使用 while 循环存储每个子句里存在的变元，同时更新变元-子句映射表对应的变元数据。

(2)void showclause(clause* clausearr);

根据 clausearr 链表记录的变元，使用双重 while 循环在控制台输出每一个子句里的变元。

(3)clause* findsoloclause(clause* tempclause);

使用 while 循环遍历 tempclause 链表，寻找并返回第一个出现的单子句，如果没有单子句则返回 NULL。

(4)int chooseordinary(int* boolcountarr);

在 boolcountarr 数组里按顺序遍历，选择出出现次数最多的变元并返回。

(5)void simplify(clause*& clausearr,int* boolcountarr,int unitnum);//化简子句集

首先通过双重 while 循环查找并删除含 unitnum 的子句，此处的 unitnum 即 findsoloclause 找到的单子句中的单个变元。

如果此时子句集为空，则直接返回。

如果此时子句集不为空，则双重 while 循环遍历所有子句的变元，从中删去 unitnum 的相反数。完成子句集的化简。

(6)clause* copy(clause* clausearr);

双重 while 循环遍历传进来的子句集的每个变元，复制到新子句集里，返回新子句集的第一个子句。

(7)int* copycountarr(int* boolcountarr);

遍历复制 boolcounrarr，用于更新新的变元数量表，参与 DPLL 分支的化简计算。以此在回溯 DPLL 同时回溯变元数量表。

(8)void freeclausearr(clause* clausearr);

free 掉 clausearr 子句集里的每一个子句以及子句里的变元。防止内存泄露。

(9)bool DPLLordinary(clause*& clausearr,int* boolcountarr,int* assignment); //普通未优化 DPLL:采用链表数据结构，使用递归解决问题，遍历寻找单子句和出现次数最多的变元

clausearr 是当前子句集，boolcountarr 是当前的变元出现次数数组，assignment 是当前变元赋值表。

先进行超时检测，超时则直接返回 false。

while 循环不断寻找新的单子句直到找不到单子句，在循环中调用 simplify 函数对当前子句集进行化简得到化简后的子句集。如果化简后的子句集是空集，

则说明有解，就将当前的变元赋值 assignment 数组复制到最终结果数组 finalassignment 数组里，返回 true 结果;如果化简后的子句集中有空子句(通过遍历查找判断)，则说明当前子句集及其各个变元赋值不满足有解情况，返回 false;如果化简后的子句集都不符合上述两种情况，则继续执行之后的代码。

通过 chooseordinary 函数选择下一个应该尝试赋值的变元。

通过 copy 函数将当前的子句集复制出两份复制子句集，一个复制子句集尝试赋值选择的这个变元为真，并作为单子句成为复制子句集的第一个子句，将对应的变元赋值表和变元出现次数数组作为参数，进行新的 DPLL 分支。另一个复制子句集尝试赋值选择的这个变元为假，其余同理，再进行新的 DPLL 分支。任意分支返回 true 说明递归中找到有解情况，同样向上返回 true；都返回 false 说明找不到解，于是该层 DPLL 向上返回 false。通过二叉搜索树加回溯的思路最终实现查找 cnf 文件给出的子句集的有解与否情况。

(10)void initsolver(Solver*& solver);

初始化 solver，对 solver 内部的变量进行赋值，对数组进行内存分配。

(11)void decayactivity(Solver* solver);

计算衰减活动度，衰减 solver 的每一个变元的活动度，以便于下次选择新变元赋值时可以找到更合适的变元。

(12)void conflictoccur(Solver* solver);

冲突发生时增加冲突数量，冲突数量每增加 50 次调用 decayactivity 函数进行一次衰减活动度。

(13)void addclause(Solver*& solver, int* literals, int count);

添加子句，用 memcpy 函数将 literals 数组复制一份成为 solver 内的新子句，增加 solver 的子句数变量。

(14)void copyclauseset(ClauseSet* dest, ClauseSet* src);

复制子句集，通过 for 循环将 src 子句集里的每一个子句用 memcpy 函数复制到 dest 子句集内。

(15)void enqueue(Solver* solver, int lit); //入队

将变元 lit 添加到存储在 solver 内的一个数组实现的队列的队尾。该队列作用在介绍 unit_propapation 函数时解释。

(16)int dequeue(Solver* solver); //出队

将存储在 solver 内的一个数组实现的队列的队头的变元取出，返回这个变元。该队列作用在介绍 unit_propapation 函数时解释。

(17)int popstack(Solver* solver, int* abslit, int* value,int* level);//出栈

将存储在 solver 内的一个数组实现的栈的栈顶元素取出，返回这个元素即变元。value 是该变元的当前正负赋值(1 或-1)，level 是该变元的当前层级。该栈作用在介绍 unit_propapation 函数时解释。

(18)void pushstack(Solver* solver, int abslit, int value,int level); //入栈

将变元的绝对值 abslit 添加到存储在 solver 内的一个数组实现的栈的栈顶。value 是该变元的当前正负赋值(1 或-1)，level 是该变元的当前层级。该栈作用在介绍 unit_propapation 函数时解释。

(19)void assign(Solver* solver, int abslit, int value, int level, int isdecision);//赋值变元 1 || -1

给 solver 内 assignment 数组里索引为 abslit 的元素赋值为 value，level 数组里索引为 abslit 的元素赋值为 level。如果 isdecision 为 true 即该要赋值的变元是选择出来的变元，需要放入决策栈内，则将 abslit 变元以及它的 value 和 level 作为参数调用 pushstack 函数，放入决策栈。随后无论是否是选择出来的变元都将其相反的 value 赋值下的 opplit 调用 enqueue 函数放入队列里，以进行之后的子句集化简。

(20)void unassign(Solver* solver, int abslit); //取消赋值

将 solver 内 abslit 作为索引的 assignment 数组和 level 数组的元素都设置为 0。

(21)int unit_propagation(Solver* solver); //单子句传播

while 循环条件为 solver 内的 queue 队列不为空。

循环内先调用 dequeue 函数取出队头变元 lit，获取这个变元的绝对值 abslit，通过 lit 的正负值获取对应该变元的变元-子句映射表。for 循环遍历这个变元-子句映射表里的所有子句，再 for 循环遍历每一个子句里的变元，如果查找到有变元的当前赋值为真，则说明当前子句已满足，此时 break 跳出；如果查找到有变元未赋值，则增加未赋值变元计数器。随后如果当前子句已满足直接 continue 遍历下一条子句。如果未赋值变元计数器等于 1，说明可以尝试向该变元使子句满足的方向进行传播，此时对这个变元调用 assign 函数进行赋值使子句满足。如果未赋值变元计数器等于 0，说明产生冲突，对当前子句的所有变元累加活动度并

调用 conflictoccur 函数，返回 0，代表单子句传播失败。

当 while 循环结束时，返回 1，代表单子句传播成功。

(22)int select_variable(Solver* solver);//选择新变元

在 solver 的所有变元中，遍历选择出未赋值的活动度最大的变元并返回。

(23)int is_satisfied(Solver* solver); //检查是否满足

for 循环遍历 solver 里的 working 子句集，再 for 循环遍历每个子句的变元，如果有变元赋值为真，说明该子句已满足，此时 break 跳出；如果该子句不满足，说明当前情况下整个子句集无解，返回 0，代表不满足。外层 for 循坏结束，函数返回 1，代表已满足有解。

(24)void backtrack(Solver* solver, int target_level); //回溯上一层状态

首先 for 循环将 solver 里的 level 数组里所有 level 高于 target_level 的变元调用 unassign 函数取消赋值。

while 循环里将所有 level 大于 tatget_level 的变元移出决策栈。

重置 solver 的队列为空，将 solver 的 currentlevel 设置成 target_level。

(25)bool DPLLoptimize(Solver* solver);　　//DPLL

首先将 solver 里的 original 子句集通过 copyclauseset 函数复制到 working 子句集。再通过一个 for 循环遍历将 working 子句集中的单子句的未赋值变元进行赋值操作。随后进行一次单子句传播和子句集是否满足的判断。

接下来 while 循环内，先计算当前运行时间，如果超时则直接返回 false。随后调用 select_variable 函数选出一个未赋值变元。如果能选出变元（有未赋值变元），则增加 currentlevel，用 assign 函数赋值为真，进行单子句传播和子句集是否满足的判断。如果单子句传播失败，则回溯一层，用 assign 函数赋值该变元为假，再进行一次单子句传播，如果同时子句集满足则直接返回 true，否则先回溯一层，进入出栈的 while 循环。出栈 while 循环每次 popstack 出一个变元，就回溯到这个变元的上一层，并在 tried 数组中增加一次尝试该变元的记录，如果记录数是 2 的倍数，说明该变元的真假两种赋值情况已经尝试过，continue 继续 while 循环回溯更深层级。如果不是 2 的倍数则将其赋值为原来尝试过的相反数（尝试过真则尝试假，尝试过假则尝试真），进行单子句传播和子句集是否满足的判断，如果满足则直接返回 true，反之则回溯一层，增加该变元尝试次数。

决策栈和单子句传播队列用于非递归 DPLL 的回溯实现，solver 里的 tried 数

组用于防止回溯时重复尝试单个变元的正反值陷入死循环。

(26)void shufflenumber(int* num);

随机打乱 num 数组的各个数字。

(27)bool issatified(int chessboard[11][11],int x,int y,int num);

通过多个 if 语句和 for 循环遍历，检查当前作为棋盘的 chessboard 二维数组是否满足百分号数独的规则。

(28)bool generatebase(int chessboard[11][11]);//回溯遍历生成终盘

先遍历查找到第一个空的棋盘位置，随后使用打乱的 num 数组，num 数组里存储 1-9 的数字，遍历每一个数字，如果满足百分号数独的规则就放进去，继续递归填数，若递归失败则恢复这个棋盘位置为空，继续尝试其他数字。

(29)void shuffleplace(place* places);

随机打乱 places 数组的各个元素。

(30)int findsolutions(int chessboard[11][11]);

遍历查找 chessboard 的第一个空位，对其进行从 1-9 每个数字填入 chessboard 的尝试，如果满足百分号数独的规则，则继续递归，返回当前填入的方法数。

(31)void generatesudoku(int chessboard[11][11],int leftnumber);

先调用 generatebase 函数生成终盘，随后打乱从 1-81 的棋盘位置的 places 数组，使用挖洞法在复制的棋盘上挖洞，如果挖了当前洞后棋盘填数解法超过 1 则补回原来这个洞，继续循环。直到挖洞数达到目标挖洞数目。

(32)void sudoku();

while 循环显示百分号数独界面，根据生成的百分号数独棋盘创建添加子句，进而调用 DPLLoptimize 函数得出唯一解并记录。根据输入的指令完成相应棋盘改变和显示。每次填数判断是否违反百分号棋盘的规则。

4.2 系统测试（黑体 4 号加粗，字母、阿拉伯数字为 Time New Roman4 号加粗）

4.2.1 系统界面测试

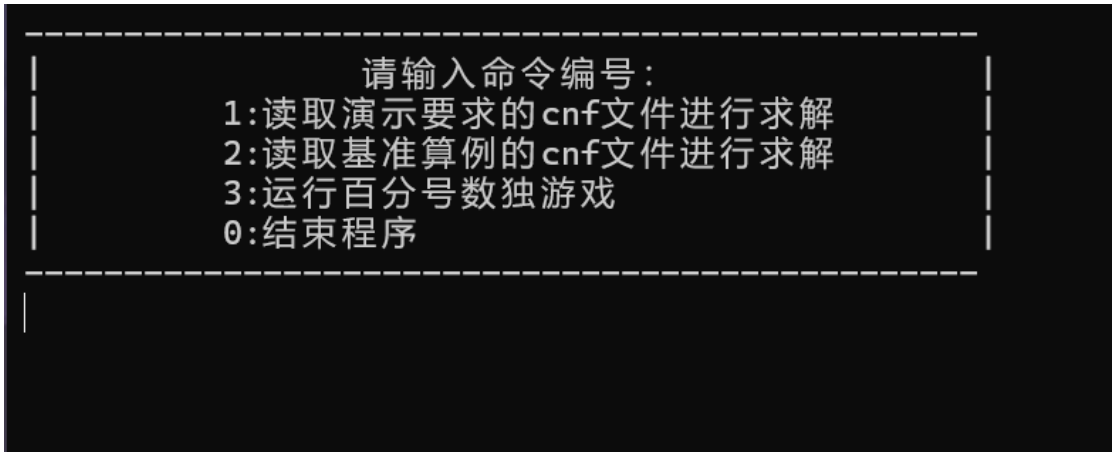

图 4.2.1.1 主界面

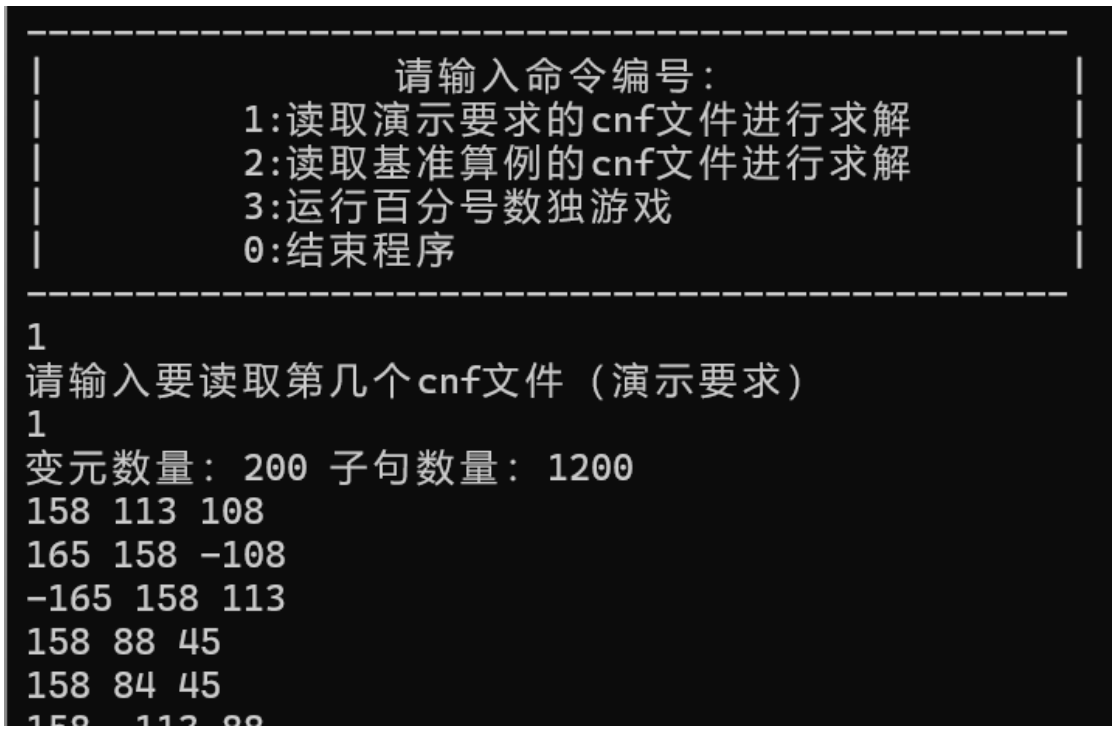

图 4.2.1.2 演示要求的 cnf 文件求解界面（例）

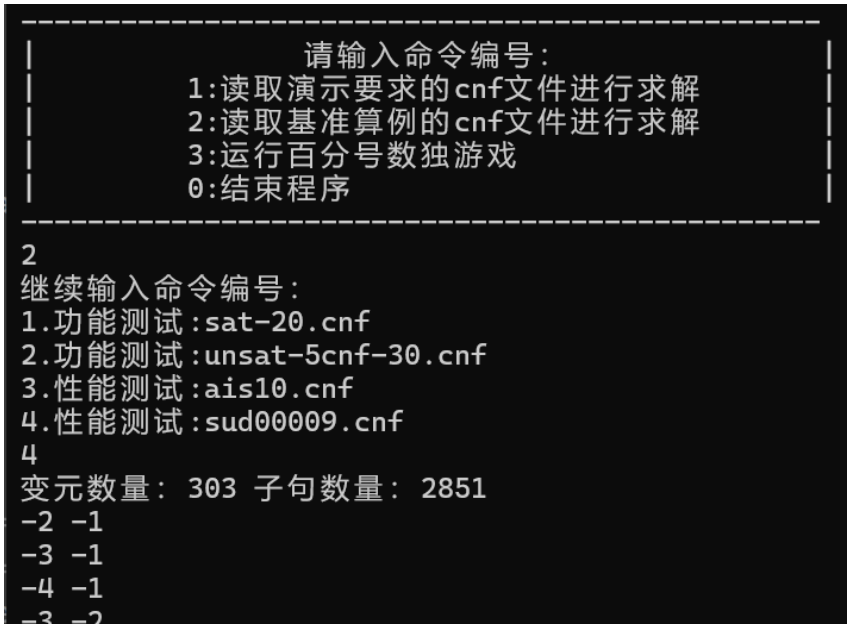

图 4.2.1.3 基准算例的 cnf 文件求解界面（例）

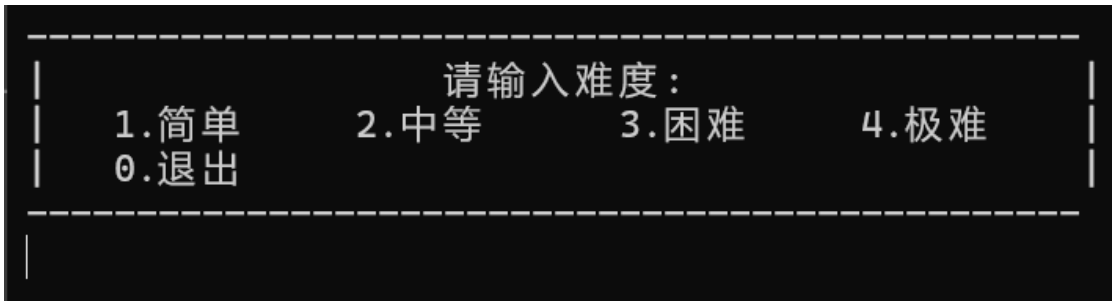

图 4.2.1.4 运行百分号数独选择难度界面

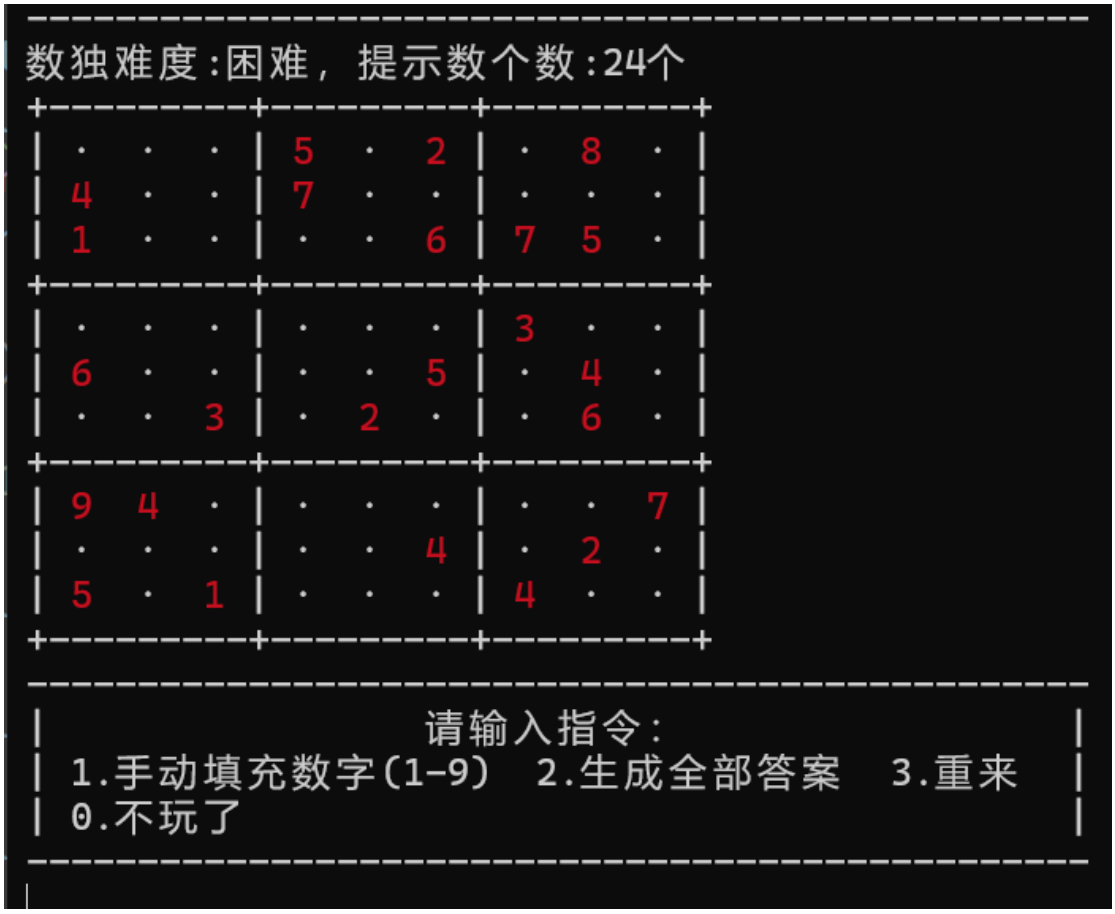

图 4.2.1.5 百分号数独游戏进行界面（例）

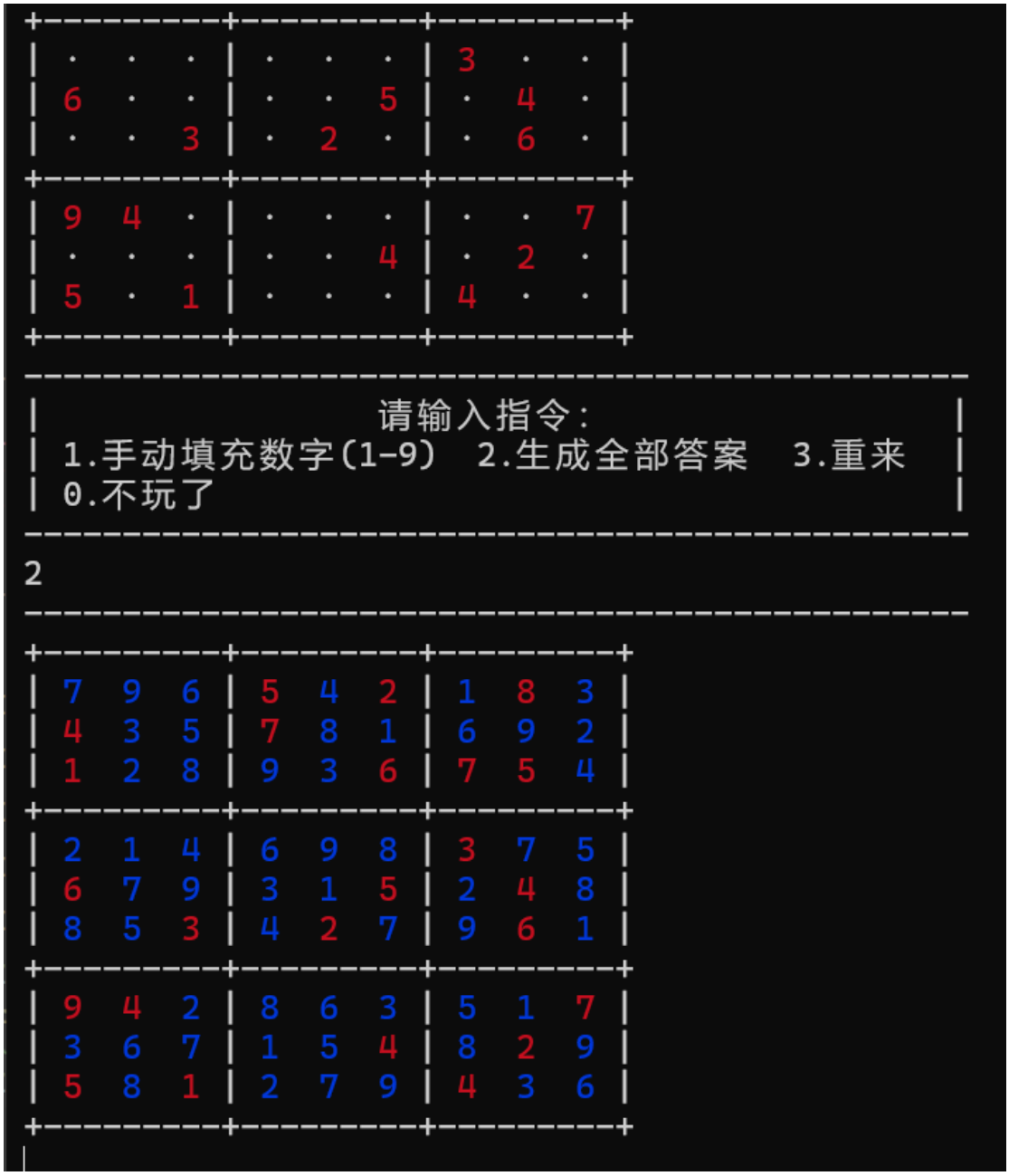

图 4.2.1.6 百分号数独游戏生成全部答案界面（例）

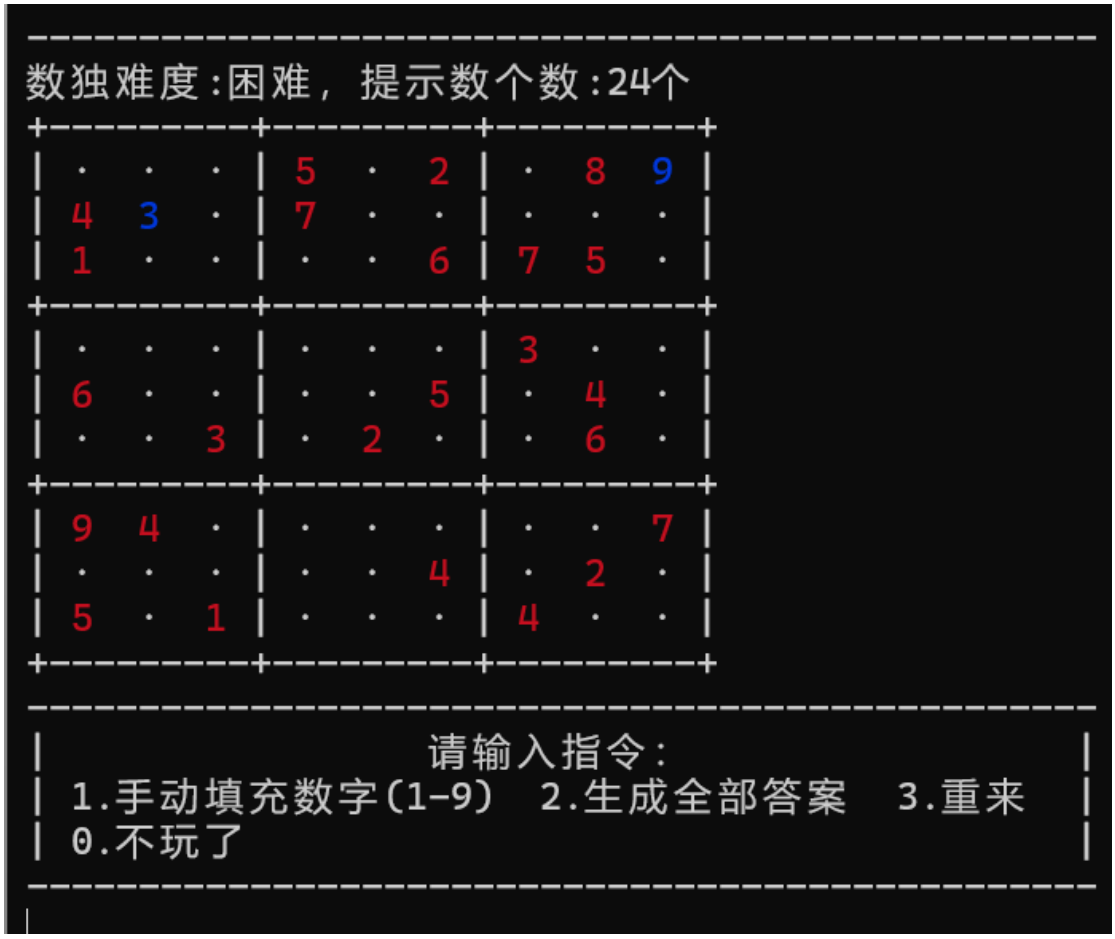

图 4.2.1.7 百分号数独游戏手动填充数字界面（例）

	优化	未优化	优化率	10min限制	-1解不出	单位:毫秒
1	5	117083	99.99573			
2	39	40803	99.90442			
3	0	42693	100			
4	42	74120	99.94334			
5	12	34	64.70588			
6	723	78078	99.074			
7	-1	-1	0			
8	-1	-1	0			
9	-1	-1	0			
10	-1	-1	0			
11	39825	354791	88.77508			
12	-1	-1	0			

图 4.2.1.8 对助教课设演示检查的 cnf 文件的程序运行结果

5 总结与展望

5.1 全文总结

(1)深入理解 SAT 问题的 CNF 范式（布尔变元、文字、子句定义）与 DPLL 算法逻辑（单子句传播、分裂策略），设计了自定义数据结构以满足存储需求 —— 包括链表形式的变元（unit）、数组与链表结合的子句（Clause）、记录子句索引的变元 - 子句映射表（Litmap），以及封装求解状态的解决器（Solver），均未依赖 C++ STL 类库。实现了 CNF 算例解析模块，支持读取含注释的.cnf 文件，提取变元与子句数量并转化为内部结构，同时通过 showclause 函数遍历输出子句，人工校验解析正确性；完成输入输出功能，可将求解结果（变元赋值、是否可满足、求解时间）保存为.res 文件。

(2)DPLL 算法优化与性能验证首先实现了递归版本的基础 DPLL（DPLLordinary），采用 “变元出现频率” 策略选取分支变元，通过单子句传播（simplify 函数）化简子句集；随后针对递归效率低、回溯可控性差的问题，优化为非递归版本（DPLLoptimize）：引入决策栈记录变元赋值层级，用传播队列加速单子句传播（unit_propagation 函数），采用 “变元活动度” 策略（decayactivity 函数衰减活动度、select_variable 函数选最高活动度变元）减少搜索分支。验证了优化后求解器在处理中大规模算例时的性能提升，同时通过 time.h 函数精确记录求解时间（毫秒级）。

(3)百分号数独到 SAT 问题的转化：首先生成普通数独的约束子句（每个单元格 “至少填一个数”“最多填一个数”、每行 / 每列 / 3×3 盒 “数字不重复”）；再补充百分号数独的额外约束 —— 撇对角线（从左上到右下的两条主对角线）、窗口（棋盘内 4 个 3×3 子窗口）的 “数字不重复” 子句，通过双重循环生成矛盾文字合取子句确保约束完整性。实现数独格局生成模块：用回溯法（generatebase 函数）生成终盘，通过 “挖洞法”（generatesudoku 函数）按难度（简单：≥40 提示数，困难：≤25 提示数）挖去部分数字，将提示数转化为单子句传入 SAT 求解器。开发交互功能（sudoku 函数），支持用户选择难度、手动填数（实时校验规则）、一键生成答案，满足 “可玩且有简单交互” 的需求。

(4)采用模块化组织代码，将主控交互（display）、CNF 解析（cnfparser）、DPLL 求解（solver）、百分号数独（Sudoku）分别封装在独立的.h/.cpp 文件中，

定义清晰的函数接口（如 scancnf 读取 CNF、addclause 添加子句），避免代码冗余；通过内存释放函数（freeclausearr、unassign）防止内存泄漏，符合编码规范。系统测试覆盖功能与界面：功能上验证 CNF 解析正确性、DPLL 求解准确性（与已知算例结果对比）、数独解的唯一性；界面上测试主菜单（SAT 算例 / 数独选择）、数独难度选择、错误填数提示等交互逻辑，确保流程通顺。

5.2 工作展望

在今后的研究中，围绕着如下几个方面开展工作。

（1）尝试引入图形化界面（GUI），使用 Qt 等架构，支持鼠标点击填数、高亮显示当前行/列/宫、错误填数标红，提升用户体验；增加难度动态调整功能，根据用户填数速度与正确率实时调整后续挖洞数量；扩展支持更多变型数独（如对角线数独、窗口数独），只需新增对应约束的子句生成逻辑，复用现有 SAT 求解核心。

（2）目前求解器未引入现代 SAT 求解的核心机制，后续可添加冲突驱动子句学习即CDCL算法：在冲突发生时（conflictoccur函数触发），通过 “1-UIP”（唯一蕴涵点）算法提取冲突子句，加入子句集以裁剪后续搜索空间；同时增加子句管理策略，如定期删除 “不活跃子句”（长期未参与传播的子句），降低内存占用并提升传播效率。此外，可尝试并行化 DPLL，将搜索树的不同分支分配给多线程处理，进一步缩短大规模算例的求解时间。

6 体会

1. 巩固了 C 语言和数据结构方面的知识。对结构体，指针，多文件编译，和一些基本的数据结构有了更深入的理解，更牢固的掌握。
2. 提升编程能力，设计数据结构（如变元-子句映射表 Litmap）、搭建系统框架，完成 DPLL 算法实现、数独约束转化等核心功能。
3. 培养时间统筹能力，规划 “算法理解→模块开发→优化测试→报告撰写” 进度，遇 unit_propagation 函数调试超时问题时，动态调整优先级，保障核心功能完成。
4. 增强自主学习能力，通过查阅文献网络文献，理解变元活动度策略、百分号数独 SAT 归约方法，解决设计中的知识盲区。
5. 锤炼抗压心态，面对 DPLL 死循环、数独校验漏约束等报错/逻辑问题，通过逐行打日志、分模块测试冷静排查，避免焦虑。

希望今后能够保持优良的码风，努力实现每一个程序功能。

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

附录

本程序使用的所有源代码。

1. cnfparser.h

```
#pragma once
#ifndef CNFPARSER
#define CNFPARSER
#include<iostream>
#include<cstring>

#define MAXSIZE 20000
using namespace std;

struct unit
{
    int unitnum;
    unit* nextunit;
};

struct clause
{
    unit* firstunit;
    clause* nextclause;
};



struct Clause
{
    int literals[MAXSIZE];  // 子句中的文字
    int size;                    // 文字数量
};

// 子句集合
struct ClauseSet
{
    Clause clauses[MAXSIZE]; // 子句数组
    int count;                   // 子句数量
};

struct Litmap
```

```
{
    int* pos;  // 包含v的子句索引列表（子句在clauses数组中的索引）
    int* neg;  // 包含-v的子句索引列表
    int pos_len;  // pos数组长度
    int neg_len;  // neg数组长度
};
struct Solver
{
    int* assignment; // 变量赋值 (1-based索引)
    int* level;      // 变量赋值的决策层
    double* activity;   // 变量活动度，用于选择下一个变量
    int* tried;
    int confictcount;
    int boolnum;                // 变量总数
    int clausenum;             // 子句总数

    Litmap* litmap;

    ClauseSet original;           // 原始子句集
    ClauseSet working;            // 工作子句集，用于传播和化简

    // 用于非递归实现的栈和队列
    int stack[MAXSIZE][3];     // 决策栈 [变量][值]
    int stackptr;                // 栈指针

    int queue[MAXSIZE];        // 传播队列
    int queuehead;               // 队列头指针
    int queuetail;               // 队列尾指针

    int currentlevel;            // 当前决策层
};
void scancnf(string filename, int*& boolcountarr, clause*& clausearr,Solver*& solver);

#endif
```

2.display.h

```
#pragma once
#ifndef DISPLAY
#define DISPLAY
#include "cnfparser.h"
void showclause(clause* clausearr);

#endif
```

3. solver.h

```
#pragma once
#ifndef SOLVER
#define SOLVER
#include "cnfparser.h"

clause* findsoloclause(clause* tempclause);
void simplify(clause*& clausearr,int* boolcountarr, int unitnum);
int chooseordinary(int* boolcountarr);
clause* copy(clause* clausearr);
int* copycountarr(int* boolcountarr);
void freeclausearr(clause* clausearr);
bool DPLLordinary(clause*& clausearr, int* boolcountarr, int* assignmengt);

void initsolver(Solver*& solver);
void decayactivity(Solver* solver);
void conflictoccur(Solver* solver);
void addclause(Solver*& solver,int* literals,int count);
void copyclauseset(ClauseSet* dest, ClauseSet* src);
void enqueue(Solver* solver, int lit);
int dequeue(Solver* solver);
int popstack(Solver* solver, int* abslit, int* value,int* level);
void pushstack(Solver* solver, int abslit, int value,int level);
void assign(Solver* solver, int abslit, int value, int level, int isdecision);
void unassign(Solver* solver, int abslit);
int unit_propagation(Solver* solver);
int select_variable(Solver* solver);
int is_satisfied(Solver* solver);
void backtrack(Solver* solver, int target_level);
bool DPLLoptimize(Solver* solver);

#endif
```

4. Sudoku.h

```
#pragma once
#ifndef SUDOKU
#define SUDOKU

struct place
{
    int x,y;
```

```
};
void shufflenumber(int* num);
bool issatified(int chessboard[11][11], int x, int y, int num);
bool generatebase(int chessboard[11][11]);
void shuffleplace(place* places);
int findsolutions(int chessboard[11][11]);
void generatesudoku(int chessboard[11][11], int answer[11][11], int leftnumber);
void sudoku();

#endif
```

5.cnfparser.cpp

```
#include "cnfparser.h"
#include "solver.h"
#include<iostream>
#include <cstdio>
#include <cstring>
#include <cstdlib>
#include <ctype.h>
using namespace std;
#ifndef MAXSIZE
#define MAXSIZE 20000
#endif // !MAXSIZE

extern int boolnum, clausenum;

void scancnf(string filename,int*& boolcountarr,clause*& clausearr,Solver*& solver)  //
输入标准cnf文件
{
    FILE* fp = fopen(filename.c_str(), "r");
    if (fp == NULL)
    {
        perror("文件打开失败\n");
        return;
    }

    char firstchar;
    char buffer[300];
    do
    {
        firstchar = getc(fp);
        if (firstchar == 'p')
        {
```

```
            fgets(buffer, 6, fp);
            break;
        }
        else if (firstchar == 'c')
        {
            char* result = fgets(buffer, sizeof(buffer), fp);
            if (result == NULL)
            {
                if (feof(fp) || ferror(fp))
                {
                    perror("该cnf不含有子句");
                    break;
                }
            }
        }
        else if (firstchar == EOF)
        {
            if (feof(fp) || ferror(fp))
            {
                perror("该cnf不含有子句");
                break;
            }
        }
    } while (true);

    fscanf(fp,"%d%d", &boolnum, &clausenum);

    boolcountarr = (int*)calloc(2 * boolnum + 5,sizeof(int));

    clause* nowclause = NULL;

    for (int i = 0; i < clausenum; i++)   // 读取子句里的变元
    {
        if (clausearr == NULL)
        {
            clausearr = (clause*)malloc(sizeof(clause));
            nowclause = clausearr;
            nowclause->firstunit = NULL;
            nowclause->nextclause = NULL;
        }
        else
        {
            clause* tempclause = (clause*)malloc(sizeof(clause));
            tempclause->nextclause = NULL;
```

```
            tempclause->firstunit = NULL;
            nowclause->nextclause = tempclause;
            nowclause = tempclause;
        }

        int unitnum;
        fscanf(fp,"%d", &unitnum);
        while (unitnum)
        {
            unit* tempunit = (unit*)malloc(sizeof(unit));
            tempunit->unitnum = unitnum;
            tempunit->nextunit = nowclause->firstunit;
            nowclause->firstunit = tempunit;
            ++boolcountarr[unitnum + boolnum];
            fscanf(fp,"%d", &unitnum);
        }
    }


    //优化非递归DPLL 读取数据
    initsolver(solver);
    clause* tempclause = clausearr;
    for (int i = 0; i < clausenum; i++)
    {
        if (tempclause == NULL) {
            printf("子句数量不足（预期%d，实际%d）\n", clausenum, i);
            return;
        }
        int* literals = (int*)malloc(sizeof(int)*MAXSIZE);
        int count = 0;
        unit* tempunit = tempclause->firstunit;
        while (tempunit)
        {
            literals[count++] = tempunit->unitnum;
            int absunit = abs(tempunit->unitnum);
            if (tempunit->unitnum > 0)
            {
                if (solver->litmap[absunit].pos == NULL)
                {
                    solver->litmap[absunit].pos = (int*)malloc(sizeof(int) * (clausenum
+ 1));
                    solver->litmap[absunit].pos_len = 0;
                }
```

```
                solver->litmap[absunit].pos[solver->litmap[absunit].pos_len++] = i;
            }
            else
            {
                if (solver->litmap[absunit].neg == NULL)
                {
                    solver->litmap[absunit].neg = (int*)malloc(sizeof(int) * (clausenum
+ 1));
                    solver->litmap[absunit].neg_len = 0;
                }
                solver->litmap[absunit].neg[solver->litmap[absunit].neg_len++] = i;
            }
            tempunit = tempunit->nextunit;
        }
        addclause(solver,literals,count);
        free(literals);
        tempclause = tempclause->nextclause;
    }

    fclose(fp);
}
```

6.display.cpp

```
#include "display.h"
#include "cnfparser.h"
#include<iostream>
#include <cstdio>
using namespace std;
extern int boolnum, clausenum;

void showclause(clause* clausearr)
{
    printf("变元数量：%d 子句数量：%d\n", boolnum, clausenum);   //已debug可完整输入
    clause* tempclause = clausearr;
    while (tempclause)
    {
        unit* tempunit = tempclause->firstunit;
        while (tempunit)
        {
            printf("%d ", tempunit->unitnum);
            tempunit = tempunit->nextunit;
        }
        printf("\n");
```

```
        tempclause = tempclause->nextclause;
    }
}
```

7.solver.cpp

```
#include "solver.h"
#include "cnfparser.h"
#include<iostream>
#include <cstdio>
#include <cstring>
#include<ctime>
using namespace std;
extern clock_t begintime,begintime2;
extern double maxtime;
extern bool overtime,overtime2;
extern int boolnum, clausenum;
extern int* finalassignment;
double activity_factor = 1.0;   //增长因子
const double decay_factor = 0.5;  // 衰减因子
//未优化
clause* findsoloclause(clause* tempclause)  //遍历寻找第一个出现的单子句
{
    while (tempclause)
    {
        if (tempclause->firstunit == NULL)
        {
            tempclause = tempclause->nextclause;
            continue;
        }
        if (tempclause->firstunit->nextunit == NULL)
        {
            return tempclause;
        }
        tempclause = tempclause->nextclause;
    }
    return NULL;
}
void simplify(clause*& clausearr,int* boolcountarr,int unitnum)  //化简S
{
    clause* preclause = NULL;
    clause* tempclause = clausearr;
    while (tempclause)   //查找删除含unitnum的子句
    {
```

```
        bool shouldelete = false;
        unit* tempunit = tempclause->firstunit;
        while (tempunit)
        {
            if (tempunit->unitnum == unitnum)
            {
                shouldelete = true;
                break;
            }
            tempunit = tempunit->nextunit;
        }
        if (shouldelete)
        {
            unit* tempunit = tempclause->firstunit;
            while (tempunit)
            {
                unit* t = tempunit;
                tempunit = tempunit->nextunit;
                --boolcountarr[t->unitnum + boolnum];
                free(t);
            }
            if (preclause == NULL)
            {
                clausearr = tempclause->nextclause;
                free(tempclause);
                tempclause = clausearr;
            }
            else
            {
                preclause->nextclause = tempclause->nextclause;
                free(tempclause);
                tempclause = preclause->nextclause;
            }
        }
        else
        {
            preclause = tempclause;
            tempclause = tempclause->nextclause;
        }
    }
    if (clausearr == NULL)
    {
        return;
    }
```

```
    unitnum = -unitnum;
    tempclause = clausearr;
    while (tempclause)
    {
        unit* tempunit = tempclause->firstunit;
        unit* preunit = NULL;
        while (tempunit)
        {
            if (tempunit->unitnum == unitnum)
            {
                --boolcountarr[unitnum + boolnum];
                if (preunit == NULL)
                {
                    tempclause->firstunit = tempunit->nextunit;
                    free(tempunit);
                    tempunit = tempclause->firstunit;
                }
                else
                {
                    preunit->nextunit = tempunit->nextunit;
                    free(tempunit);
                    tempunit = preunit->nextunit;
                }
            }
            else
            {
                preunit = tempunit;
                tempunit = tempunit->nextunit;
            }
        }
        tempclause = tempclause->nextclause;
    }
}
int chooseordinary(int* boolcountarr)  //遍历寻找出现次数最多的变元 ， 此处变元区分符号
{
    int maxunitnum = 1;
    int maxnum = -1;
    for (int i = 0; i <= 2 * boolnum; i++)
    {
        if (i - boolnum == 0) continue;
        if (boolcountarr[i] > maxnum)
        {
            maxnum = boolcountarr[i];
            maxunitnum = i - boolnum;
```

```
            }
        }
        return maxunitnum;
}
clause* copy(clause* clausearr)
{
        clause* newclausearr = NULL;

        clause* tempclause = clausearr;
        clause* preclause = NULL;
        while (tempclause)
        {
            clause* nowclause = (clause*)malloc(sizeof(clause));
            nowclause->nextclause = NULL;
            nowclause->firstunit = NULL;

            unit* tempunit = tempclause->firstunit;
            unit* preunit = NULL;
            while (tempunit)
            {
                unit* nowunit = (unit*)malloc(sizeof(unit));
                nowunit->nextunit = NULL;
                nowunit->unitnum = tempunit->unitnum;
                if (preunit == NULL)
                {
                    nowclause->firstunit = nowunit;
                }
                else
                {
                    preunit->nextunit = nowunit;
                }
                preunit = nowunit;
                tempunit = tempunit->nextunit;
            }

            if (preclause == NULL)
            {
                newclausearr = nowclause;
            }
            else
            {
                preclause->nextclause = nowclause;
            }
            preclause = nowclause;
```

```
        tempclause = tempclause->nextclause;
    }
    return newclausearr;
}
int* copycountarr(int* boolcountarr)
{
    int total = 2 * boolnum + 5;
    int* newcountarr = (int*)malloc(sizeof(int) * total);
    for (int i = 0; i < total; i++)
    {
        newcountarr[i] = boolcountarr[i];
    }
    return newcountarr;
}
void freeclausearr(clause* clausearr)
{
    while (clausearr)
    {
        unit* t = clausearr->firstunit;
        while (t)
        {
            unit* next = t->nextunit;
            free(t);
            t = next;
        }
        clause* nextclausearr = clausearr->nextclause;
        free(clausearr);
        clausearr = nextclausearr;
    }
}
bool DPLLordinary(clause*& clausearr,int* boolcountarr,int* assignment)   //普通未优化
DPLL:采用链表数据结构，使用递归解决问题，遍历寻找单子句和出现次数最多的变元
{
    if (overtime)
    {
        return false;
    }
    time_t nowtime = clock();
    if ( (double)(nowtime-begintime)/CLOCKS_PER_SEC > maxtime)
    {
        overtime = true;
        return false;
    }
    int* nowcountarr = copycountarr(boolcountarr);
```

```
clause* tempclause = findsoloclause(clausearr);
while (tempclause)
{
    int unit = tempclause->firstunit->unitnum;
    int absunit = abs(unit);
    int value = (unit > 0) ? 1 : -1;
    assignment[absunit] = value;
    simplify(clausearr, nowcountarr,unit);
    if (clausearr == NULL)
    {
        free(nowcountarr);
        for (int i = 1; i <= boolnum; i++)
        {
            finalassignment[i] = assignment[i];
        }
        return true;
    }
    else    //查找是否有空子句
    {
        clause* t = clausearr;
        while (t)
        {
            if (t->firstunit == NULL)
            {
                free(nowcountarr);
                return false;
            }
            t = t->nextclause;
        }
    }
    tempclause = findsoloclause(clausearr);
}
int v = chooseordinary(nowcountarr);

  //复制两份S

clause* temp = (clause*)malloc(sizeof(clause));
temp->firstunit = (unit*)malloc(sizeof(unit));
temp->firstunit->nextunit = NULL;
temp->firstunit->unitnum = v;
temp->nextclause = copy(clausearr);

int absv = abs(v);
int oldvalue = assignment[absv];
```

```
    assignment[absv] = (v>0)?1:-1;
    ++nowcountarr[v + boolnum];
    if (DPLLordinary(temp,nowcountarr,assignment))
    {
        free(nowcountarr);
        return true;
    }

    assignment[absv] = oldvalue;
    --nowcountarr[v + boolnum];
    freeclausearr(temp);

    clause* temp2 = (clause*)malloc(sizeof(clause));
    temp2->firstunit = (unit*)malloc(sizeof(unit));
    temp2->firstunit->nextunit = NULL;
    temp2->firstunit->unitnum = -v;
    temp2->nextclause = copy(clausearr);

    assignment[absv] = (v > 0) ? -1 : 1;
    ++nowcountarr[-v + boolnum];
    bool result = DPLLordinary(temp2,nowcountarr,assignment);
    free(nowcountarr);
    if (result)
    {
        return true;
    }
    else
    {
        assignment[absv] = oldvalue;
        freeclausearr(temp2);
        return false;
    }
}


//优化后
void initsolver(Solver*& solver)//初始化solver
{
    solver = (Solver*)malloc(sizeof(Solver));
    memset(solver, 0, sizeof(Solver));
    solver->boolnum = boolnum;
    solver->clausenum = clausenum;
    solver->assignment = (int*)malloc(sizeof(int) * (solver->boolnum + 1)); // 变量从1
开始
    solver->level = (int*)malloc(sizeof(int) * (solver->boolnum + 1));
```

```
    solver->activity = (double*)malloc(sizeof(double) * (solver->boolnum + 1));
    solver->tried = (int*)malloc(sizeof(int) * (solver->boolnum + 1));
    solver->confictcount = 0;
    solver->currentlevel = 0;
    solver->stackptr = 0;
    solver->queuehead = 0;
    solver->queuetail = 0;
    solver->litmap = (Litmap*)malloc(sizeof(Litmap) * (solver->boolnum+1));
    for (int i = 0; i < solver->boolnum + 1; i++)
    {
        solver->litmap[i].pos = solver->litmap[i].neg = NULL;
        solver->litmap[i].pos_len = solver->litmap[i].neg_len = 0;
    }
    for (int i = 1; i <= solver->boolnum; i++)
    {
        solver->assignment[i] = 0;
        solver->level[i] = -1;
        solver->activity[i] = 1.0;
        solver->tried[i] = 0;
    }
}
void decayactivity(Solver* solver) //衰减活动度
{
    for (int i = 1; i <= solver->boolnum; i++)
    {
        solver->activity[i] *= decay_factor;
    }
    activity_factor *= decay_factor;
}
void conflictoccur(Solver* solver)//冲突发生时增加冲突数量
{
    solver->confictcount++;
    if (solver->confictcount % 50 == 0)
    {
        decayactivity(solver);
    }
}
void addclause(Solver*& solver, int* literals, int count)//添加子句
{
    memcpy(solver->original.clauses[solver->original.count].literals, literals,
count*sizeof(int));
    solver->original.clauses[solver->original.count].size = count;
    solver->original.count++;
}
```

```
void copyclauseset(ClauseSet* dest, ClauseSet* src) //复制子句集
{
    dest->count = src->count;
    for (int i = 0; i < src->count; i++)
    {
        dest->clauses[i].size = src->clauses[i].size;
        memcpy(dest->clauses[i].literals, src->clauses[i].literals,src->clauses[i].size
* sizeof(int));
    }
}
void enqueue(Solver* solver, int lit) //入队
{
    if ((solver->queuetail + 1) % MAXSIZE == solver->queuehead) {
        cerr << "传播队列已满，无法入队：" << lit << endl;
        return;
    }
    solver->queue[solver->queuetail] = lit;
    solver->queuetail = (solver->queuetail + 1) % MAXSIZE;
}
int dequeue(Solver* solver) //出队
{
    int lit = solver->queue[solver->queuehead];
    solver->queuehead = (solver->queuehead + 1) % MAXSIZE;
    return lit;
}
int popstack(Solver* solver, int* abslit, int* value,int* level)//出栈
{
    if (solver->stackptr <= 0) {
        return 0;
    }
    solver->stackptr--;
    *abslit = solver->stack[solver->stackptr][0];
    *value = solver->stack[solver->stackptr][1];
    *level = solver->stack[solver->stackptr][2];
    return 1;
}
void pushstack(Solver* solver, int abslit, int value,int level) //入栈
{
    solver->stack[solver->stackptr][0] = abslit;
    solver->stack[solver->stackptr][1] = value;
    solver->stack[solver->stackptr][2] = level;
    solver->stackptr++;
}
void assign(Solver* solver, int abslit, int value, int level, int isdecision)//赋值变元 1
```

```
|| -1
{
    if (solver->assignment[abslit] != 0)
    {
        return;
    }
    solver->assignment[abslit] = value;
    solver->level[abslit] = level;
    if (isdecision) {
        pushstack(solver, abslit, value, level);
    }
    int opplit = (value == 1) ? -abslit : abslit;
    enqueue(solver, opplit);
}
void unassign(Solver* solver, int abslit) //取消赋值
{
    solver->assignment[abslit] = 0;
    solver->level[abslit] = -1;
}
int unit_propagation(Solver* solver) //单子句传播
{
    while (solver->queuehead != solver->queuetail)
    {
        int lit = dequeue(solver);
        int abslit = abs(lit);
        int* targetclause;
        int targetlen;
        if (lit > 0)
        {
            targetclause = solver->litmap[abslit].pos;
            targetlen = solver->litmap[abslit].pos_len;
        }
        else
        {
            targetclause = solver->litmap[abslit].neg;
            targetlen = solver->litmap[abslit].neg_len;
        }
        for (int i = 0; i < targetlen; i++)
        {
            Clause* clause = &solver->working.clauses[targetclause[i]];
            int satisfied = 0;
            int unassigned_count = 0;
            int unassigned_lit = 0;
            for (int j = 0; j < clause->size; j++)
```

```
            {
                int l = clause->literals[j];
                int v = abs(l);
                int val = (l > 0) ? 1 :-1;
                if (solver->assignment[v] == 0)
                {
                    unassigned_count++;
                    unassigned_lit = l;
                }
                else if (solver->assignment[v] == val)
                {
                    satisfied = 1;
                    break;
                }
            }
            if (satisfied) continue;
            if (unassigned_count == 1)
            {
                int u_var = abs(unassigned_lit);
                int u_val = (unassigned_lit > 0) ? 1 : -1;
                if (solver->assignment[u_var] == -u_val)
                {
                    for (int p = 0; p < clause->size; p++)
                    {
                        int absliteral = abs(clause->literals[p]);
                        solver->activity[absliteral]+=activity_factor;
                    }
                    conflictoccur(solver);
                    return 0;
                }
                assign(solver, u_var, u_val, solver->currentlevel, 0);
            }
            else if (unassigned_count == 0)
            {
                for (int p = 0; p < clause->size; p++)
                {
                    int absliteral = abs(clause->literals[p]);
                    solver->activity[absliteral] += activity_factor;
                }
                conflictoccur(solver);
                return 0;
            }
        }
    }
```

```
    return 1;
}
int select_variable(Solver* solver)//选择新变元
{
    double max_activity = -1;
    int selected_var = -1;
    for (int i = 1; i <= solver->boolnum; i++)
    {
        if (solver->assignment[i] == 0)
        {
            if (solver->activity[i] > max_activity || (solver->activity[i] ==
max_activity && selected_var == -1))
            {
                max_activity = solver->activity[i];
                selected_var = i;
            }
        }
    }
    return selected_var;
}
int is_satisfied(Solver* solver) //检查是否满足
{
    for (int i = 0; i < solver->working.count; i++)
    {
        Clause* clause = &solver->working.clauses[i];
        int satisfied = 0;
        for (int j = 0; j < clause->size; j++)
        {
            int l = clause->literals[j];
            int v = abs(l);
            int val = (l > 0) ? 1 : -1;
            if (solver->assignment[v] == val)
            {
                satisfied = 1;
                break;
            }
        }
        if (!satisfied)
        {
            return 0;
        }
    }
    return 1;
}
```

```
void backtrack(Solver* solver, int target_level) //回溯上一层状态
{
    for (int i = 1; i <= solver->boolnum; i++)
    {
        if (solver->level[i] > target_level)
        {
            unassign(solver, i);
        }
    }
    while (solver->stackptr > 0)
    {
        int toplevel = solver->stack[solver->stackptr - 1][2];
        if (toplevel > target_level)
        {
            solver->stackptr--;
        }
        else
        {
            break;
        }
    }
    solver->queuehead = 0;
    solver->queuetail = 0;
    solver->currentlevel = target_level;
}
bool DPLLoptimize(Solver* solver)    //DPLL
{
    copyclauseset(&solver->working,&solver->original);
    for (int i = 0; i < solver->working.count; i++)
    {
        Clause* clause = &solver->working.clauses[i];
        if (clause->size == 1)
        {
            int lit = clause->literals[0];
            int abslit = abs(lit);
            if (solver->assignment[abslit] == 0)
            {
                int value = (lit > 0) ? 1 : -1;
                assign(solver, abslit, value, solver->currentlevel, 0);
            }
        }
    }
    if (!unit_propagation(solver))
    {
```

```
        return false;
    }
    if (is_satisfied(solver))
    {
        return true;
    }
    while (1)
    {
        clock_t nowtime = clock();
        if ((double)(nowtime - begintime2) / CLOCKS_PER_SEC > maxtime)
        {
            overtime2 = true;
            return false;
        }
        int var = select_variable(solver);
        if (var == -1)
        {
            if (is_satisfied(solver))
            {
                return 1;
            }
            else
            {
                while (solver->stackptr > 0)
                {
                    int back_var, back_val, back_level;
                    popstack(solver, &back_var, &back_val, &back_level);
                    backtrack(solver, back_level - 1);
                    solver->currentlevel = back_level;
                    solver->tried[back_var]++;
                    if (solver->tried[back_var] % 2 == 0)
                    {
                        continue;
                    }
                    int new_val = (back_val == 1) ? -1 : 1;
                    assign(solver, back_var, new_val, solver->currentlevel, 1);
                    if (unit_propagation(solver))
                    {
                        if (is_satisfied(solver))
                            return true;
                        break;
                    }
                    backtrack(solver, back_level - 1);
                    solver->tried[back_var]++;
```

```
            }
            if (solver->stackptr == 0)
                return false;
        }
    }
    else
    {
        solver->currentlevel++;
        assign(solver, var, 1, solver->currentlevel, 1);
        if (unit_propagation(solver))
        {
            if(is_satisfied(solver))
                return 1;
            continue;
        }
        else
        {
            backtrack(solver, solver->currentlevel - 1);
            solver->currentlevel++;
            assign(solver, var, -1, solver->currentlevel, 1);
            if (!unit_propagation(solver))
            {
                backtrack(solver, solver->currentlevel - 1);
                while (solver->stackptr > 0)
                {
                    int back_var, back_val, back_level;
                    popstack(solver, &back_var, &back_val, &back_level);
                    backtrack(solver, back_level - 1);
                    solver->currentlevel = back_level;
                    solver->tried[back_var]++;
                    if (solver->tried[back_var] %2==0)
                    {
                        continue;
                    }
                    int new_val = (back_val == 1) ? -1 : 1;
                    assign(solver, back_var, new_val, back_level, 1);
                    if (unit_propagation(solver))
                    {
                        if (is_satisfied(solver))
                            return true;
                        break;
                    }
                    backtrack(solver, back_level - 1);
                    solver->tried[back_var]++;
```

```
                    }
                    if (solver->stackptr == 0)
                    {
                        return false;
                    }
                }
                else {
                    if (is_satisfied(solver))
                    {
                        return 1;
                    }
                }
            }
        }
    }

}
```

8. Sudoku. cpp

```
#include "Sudoku.h"
#include "solver.h"
#include "cnfparser.h"
#include<iostream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include<ctime>
using namespace std;
extern int boolnum, clausenum;
extern bool overtime, overtime2;
extern clock_t begintime, begintime2;

void shufflenumber(int* num)
{
    for (int i = 9; i >0; i--)
    {
        int j = rand() % i +1;
        swap(num[i], num[j]);
    }
}
bool issatified(int chessboard[11][11],int x,int y,int num)
{
    if (num == 0)
```

```
    return true;
for (int i = 1; i < 10; i++)
{
    if (chessboard[x][i] == num) return false;
}
for (int i = 1; i < 10; i++)
{
    if (chessboard[i][y] == num) return false;
}

int start_row = ((x - 1) / 3) * 3 + 1;  // 宫格起始行：1、4、7
int start_col = ((y - 1) / 3) * 3 + 1;  // 宫格起始列：1、4、7

for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < 3; j++)
    {
        if (chessboard[start_row + i][start_col + j] == num) return false;
    }
}

if (x + y == 10)
{
    for (int i = 1; i < 10; i++)
    {
        if (chessboard[i][10 - i] == num) return false;
    }
}
if ((x > 1 && x < 5) && (y > 1 && y < 5))
{
    for (int i = 2; i < 5; i++)
    {
        for (int j = 2; j < 5; j++)
        {
            if (chessboard[i][j] == num) return false;
        }
    }
}
if ((x > 5 && x < 9) && (y > 5 && y < 9))
{
    for (int i = 6; i < 9; i++)
    {
        for (int j = 6; j < 9; j++)
        {
```

```
                if (chessboard[i][j] == num) return false;
            }
        }
    }
    return true;
}
bool generatebase(int chessboard[11][11])//回溯遍历生成终盘
{
    int x = 0, y=0;
    for (int i = 1; i < 10; i++)
    {
        for (int j = 1; j < 10; j++)
        {
            if (chessboard[i][j] == 0)
            {
                x = i;
                y = j;
                break;
            }
        }
    }
    if (x != 0 && y != 0)
    {
        int num[10] = { 0,1,2,3,4,5,6,7,8,9 };
        shufflenumber(num);
        for (int i = 1; i < 10; i++)
        {
            int number = num[i];
            if (issatified(chessboard,x, y, number))
            {
                chessboard[x][y] = number;
                if (generatebase(chessboard))
                    return true;
                chessboard[x][y] = 0;
            }
        }
        return false;
    }
    else
    {
        return true;
    }
}
void shuffleplace(place* places)
```

```
{
    for (int i = 81; i > 0; i--)
    {
        int j = rand() % i + 1;
        swap(places[i], places[j]);
    }
}

int findsolutions(int chessboard[11][11])
{
    int x = 0, y = 0;
    for (int i = 1; i < 10; i++)
    {
        for (int j = 1; j < 10; j++)
        {
            if (chessboard[i][j] == 0)
            {
                x = i;
                y = j;
                break;
            }
        }
        if (x != 0) break;
    }
    if (x != 0 && y != 0)
    {
        int count = 0;
        for (int i = 1; i < 10; i++)
        {
            if (issatified(chessboard, x, y, i))
            {
                chessboard[x][y] = i;
                count += findsolutions(chessboard);
                chessboard[x][y] = 0;
                if (count >= 2) return count;
            }
        }
        return count;
    }
    else
    {
        return 1;
    }
}
```

```
void generatesudoku(int chessboard[11][11],int leftnumber)//挖洞法
{
    generatebase(chessboard);

    place places[85];
    int len = 0;
    for (int i = 1; i < 10; i++)
    {
        for (int j = 1; j < 10; j++)
        {
            places[++len] = { i,j };
        }
    }
    shuffleplace(places);
    int nowholesnumber = 0;
    int targetholesnumber = 81 - leftnumber;
    for (int i = 1; i < 82; i++)
    {
        if (nowholesnumber >= targetholesnumber)
        {
            break;
        }
        int x = places[i].x;
        int y = places[i].y;
        int temp = chessboard[x][y];
        chessboard[x][y] = 0;
        int copyboard[11][11];
        for (int k = 1; k < 10; k++)
        {
            for (int j = 1; j < 10; j++)
            {
                copyboard[k][j] = chessboard[k][j];
            }
        }
        int solution = findsolutions(copyboard);
        if (solution == 1)
        {
            nowholesnumber++;
        }
        else
        {
            chessboard[x][y] = temp;
        }
    }
```

```
}

void sudoku()
{
    while (1)
    {
        system("cls");
        printf("-----------------------------------------------\n");
        printf("|                  请输入难度:                  |\n");
        printf("|   1.简单     2.中等      3.困难     4.极难    |\n");
        printf("|   0.退出                                      |\n");
        printf("-----------------------------------------------\n");
        srand(time(0));
        int nandu = 0,leftnum;
        scanf("%d", &nandu);
        if (nandu == 0) break;
        if (nandu > 0 && nandu < 5)
        {
            srand(time(0));
            leftnum = 30;
            if (nandu == 1)
                leftnum = rand() % (35 - 30 + 1) + 30;
            else if (nandu == 2)
                leftnum = rand() % (30 - 25 + 1) + 25;
            else if (nandu == 3)
                leftnum = rand() % (25 - 20 + 1) + 20;
            else if (nandu == 4)
                leftnum = rand() % (20 - 17 + 1) + 17;
            int chessboard[11][11],answer[11][11];
            memset(chessboard, 0, sizeof(chessboard));
            generatesudoku(chessboard,leftnum);

            int renewchessboard[11][11];
            for (int i = 1; i < 10; i++)
            {
                for (int j = 1; j < 10; j++)
                {
                    renewchessboard[i][j] = chessboard[i][j];
                }
            }

            Solver* solver = NULL;
            boolnum = 729;
            clausenum = MAXSIZE;
```

```
            initsolver(solver);
            int clausetotal = 0;

            //根据约束添加子句
            //格约束
            int* literals = (int*)malloc(sizeof(int) * 10);
            for (int i = 1; i < 10; i++)//能填
            {
                for (int j = 1; j < 10; j++)
                {
                    for (int k = 1; k < 10; k++)
                    {
                        literals[k-1] = (i-1)*81 + (j-1) * 9 + k;
                        int lit = literals[k-1];
                        if (solver->litmap[lit].pos == NULL)//添加变元-子句映射表
                        {
                            solver->litmap[lit].pos = (int*)malloc(sizeof(int) *
clausenum);
                            solver->litmap[lit].pos_len = 0;
                        }
                        solver->litmap[lit].pos[solver->litmap[lit].pos_len++] =
solver->original.count;
                    }
                    addclause(solver, literals, 9);
                }
            }
            for (int i = 1; i < 10; i++)//不能同时填
            {
                for (int j = 1; j < 10; j++)
                {
                    for (int k = 1; k < 9; k++)
                    {
                        literals[0] = -((i - 1) * 81 + (j - 1) * 9 + k);
                        int abslit = abs(literals[0]);
                        if (solver->litmap[abslit].neg == NULL)//添加变元-子句映射表
                        {
                            solver->litmap[abslit].neg = (int*)malloc(sizeof(int) *
clausenum);
                            solver->litmap[abslit].neg_len = 0;
                        }
                        solver->litmap[abslit].neg[solver->litmap[abslit].neg_len++]
= solver->original.count;
                        for (int p = k + 1; p < 10; p++)
                        {
```

```
                                literals[1] = -((i - 1) * 81 + (j - 1) * 9 + p);
                                int abslit2 = abs(literals[1]);
                                if (solver->litmap[abslit2].neg == NULL)//添加变元-子句映
射表
                                {
                                    solver->litmap[abslit2].neg = (int*)malloc(sizeof(int)
* clausenum);
                                     solver->litmap[abslit2].neg_len = 0;
                                }

    solver->litmap[abslit2].neg[solver->litmap[abslit2].neg_len++] =
solver->original.count;
                                addclause(solver, literals, 2);
                            }
                        }
                    }
                }
                //行约束
                for (int i = 1; i < 10; i++)
                {
                    for (int j = 1; j < 10; j++)
                    {
                        for (int k = 1; k < 10; k++)
                        {
                            literals[k-1] = (i - 1) * 81 + (k - 1) * 9 + j;
                            int lit = literals[k-1];
                            if (solver->litmap[lit].pos == NULL)//添加变元-子句映射表
                            {
                                solver->litmap[lit].pos = (int*)malloc(sizeof(int) *
clausenum);
                                solver->litmap[lit].pos_len = 0;
                            }
                            solver->litmap[lit].pos[solver->litmap[lit].pos_len++] =
solver->original.count;
                        }
                        addclause(solver, literals, 9);
                    }
                }
                for (int i = 1; i < 10; i++)
                {
                    for (int j = 1; j < 10; j++)
                    {
                        for (int k = 1; k < 9; k++)
                        {
```

```
                        literals[0] = -((i - 1) * 81 + (k - 1) * 9 + j);
                        int abslit = abs(literals[0]);
                        if (solver->litmap[abslit].neg == NULL)//添加变元-子句映射表
                        {
                            solver->litmap[abslit].neg = (int*)malloc(sizeof(int) *
clausenum);
                            solver->litmap[abslit].neg_len = 0;
                        }
                        solver->litmap[abslit].neg[solver->litmap[abslit].neg_len++]
= solver->original.count;
                        for (int p = k + 1; p < 10; p++)
                        {
                            literals[1] = -((i - 1) * 81 + (p - 1) * 9 + j);
                            int abslit2 = abs(literals[1]);
                            if (solver->litmap[abslit2].neg == NULL)//添加变元-子句映
射表
                            {
                                solver->litmap[abslit2].neg = (int*)malloc(sizeof(int)
* clausenum);
                                 solver->litmap[abslit2].neg_len = 0;
                            }

    solver->litmap[abslit2].neg[solver->litmap[abslit2].neg_len++] =
solver->original.count;
                            addclause(solver, literals, 2);
                        }
                    }
                }
            }
            //列约束
            for (int i = 1; i < 10; i++)
            {
                for (int j = 1; j < 10; j++)
                {
                    for (int k = 1; k < 10; k++)
                    {
                        literals[k - 1] = (k - 1) * 81 + (i - 1) * 9 + j;
                        int lit = literals[k - 1];
                        if (solver->litmap[lit].pos == NULL)//添加变元-子句映射表
                        {
                            solver->litmap[lit].pos = (int*)malloc(sizeof(int) *
clausenum);
                            solver->litmap[lit].pos_len = 0;
                        }
```

```
                        solver->litmap[lit].pos[solver->litmap[lit].pos_len++] =
solver->original.count;
                    }
                    addclause(solver, literals, 9);
                }
            }
            for (int i = 1; i < 10; i++)
            {
                for (int j = 1; j < 10; j++)
                {
                    for (int k = 1; k < 9; k++)
                    {
                        literals[0] = -((k - 1) * 81 + (i - 1) * 9 + j);
                        int abslit = abs(literals[0]);
                        if (solver->litmap[abslit].neg == NULL)//添加变元-子句映射表
                        {
                            solver->litmap[abslit].neg = (int*)malloc(sizeof(int) *
clausenum);
                            solver->litmap[abslit].neg_len = 0;
                        }
                        solver->litmap[abslit].neg[solver->litmap[abslit].neg_len++]
= solver->original.count;
                        for (int p = k + 1; p < 10; p++)
                        {
                            literals[1] = -((p - 1) * 81 + (i - 1) * 9 + j);
                            int abslit2 = abs(literals[1]);
                            if (solver->litmap[abslit2].neg == NULL)//添加变元-子句映
射表
                            {
                                solver->litmap[abslit2].neg = (int*)malloc(sizeof(int)
* clausenum);
                                 solver->litmap[abslit2].neg_len = 0;
                            }

    solver->litmap[abslit2].neg[solver->litmap[abslit2].neg_len++] =
solver->original.count;
                            addclause(solver, literals, 2);
                        }
                    }
                }
            }
            //宫约束
            for (int hangzengliang = 0; hangzengliang < 9; hangzengliang += 3)
            {
```

```
                for (int liezengliang = 0; liezengliang < 9; liezengliang += 3)
                {
                    for (int num = 1; num < 10; num++)
                    {
                        int count = 0;
                        for (int i = 1; i < 4; i++)//能填
                        {
                            for (int j = 1; j < 4; j++)
                            {
                                literals[count] = (hangzengliang+i - 1) * 81 +
(liezengliang+j - 1) * 9 + num;
                                int lit = literals[count];
                                if (solver->litmap[lit].pos == NULL)//添加变元-子句映
射表
                                {
                                    solver->litmap[lit].pos =
(int*)malloc(sizeof(int) * clausenum);
                                    solver->litmap[lit].pos_len = 0;
                                }

    solver->litmap[lit].pos[solver->litmap[lit].pos_len++] = solver->original.count;
                                count++;
                            }
                        }
                        addclause(solver, literals, count);
                        for (int i = 1; i < 4; i++)//不能填
                        {
                            for (int j = 1; j < 4; j++)
                            {
                                if (i == 3 && j == 3)
                                    break;
                                literals[0] = -((hangzengliang + i - 1) * 81 +
(liezengliang + j - 1) * 9 + num);
                                int abslit = abs(literals[0]);
                                if (solver->litmap[abslit].neg == NULL)//添加变元-子
句映射表
                                {
                                    solver->litmap[abslit].neg =
(int*)malloc(sizeof(int) * clausenum);
                                    solver->litmap[abslit].neg_len = 0;
                                }

    solver->litmap[abslit].neg[solver->litmap[abslit].neg_len++] =
solver->original.count;
```

```
                                    for (int k = 1; k < 4; k++)
                                    {
                                        for (int p = 1; p < 4; p++)
                                        {
                                            if (k < i)
                                            {
                                                break;
                                            }
                                            else
                                            {
                                                if (p <= j)
                                                    continue;
                                                literals[1] = -((hangzengliang + k - 1)
* 81 + (liezengliang + p - 1) * 9 + num);
                                                int abslit2 = abs(literals[0]);
                                                if (solver->litmap[abslit2].neg ==
NULL)//添加变元-子句映射表
                                                {
                                                    solver->litmap[abslit2].neg =
(int*)malloc(sizeof(int) * clausenum);
                                                    solver->litmap[abslit2].neg_len =
0;
                                                }

    solver->litmap[abslit2].neg[solver->litmap[abslit2].neg_len++] =
solver->original.count;
                                                addclause(solver, literals, 2);
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
//百分号约束----------
//撒对角线约束
for (int k = 1; k < 10; k++)
{
    for (int i = 1; i < 10; i++)//能填
    {
        int j = 10 - i;
        literals[i - 1] = (i - 1) * 81 + (j - 1) * 9 + k;
        int lit = literals[i - 1];
```

```
        if (solver->litmap[lit].pos == NULL)//添加变元-子句映射表
        {
            solver->litmap[lit].pos = (int*)malloc(sizeof(int) * clausenum);
            solver->litmap[lit].pos_len = 0;
        }
        solver->litmap[lit].pos[solver->litmap[lit].pos_len++] =
solver->original.count;
    }
    addclause(solver, literals, 9);
    for (int i = 1; i < 9; i++)//不能填
    {
        literals[0] = -((i - 1) * 81 + (10-i - 1) * 9 + k);
        int abslit = abs(literals[0]);
        if (solver->litmap[abslit].neg == NULL)//添加变元-子句映射表
        {
            solver->litmap[abslit].neg = (int*)malloc(sizeof(int) * clausenum);
            solver->litmap[abslit].neg_len = 0;
        }
        solver->litmap[abslit].neg[solver->litmap[abslit].neg_len++] =
solver->original.count;
        for (int j = 10 - i-1; j > 0; j--)
        {
            literals[1] = -((i - 1) * 81 + (j - 1) * 9 + k);
            int abslit = abs(literals[1]);
            if (solver->litmap[abslit].neg == NULL)//添加变元-子句映射表
            {
                solver->litmap[abslit].neg = (int*)malloc(sizeof(int) * clausenum);
                solver->litmap[abslit].neg_len = 0;
            }
            solver->litmap[abslit].neg[solver->litmap[abslit].neg_len++] =
solver->original.count;
            addclause(solver, literals, 2);
        }

    }
}
//窗口约束1
for (int num = 1; num < 10; num++)
{
    int count = 0;
    for (int i = 1; i < 4; i++)//能填
    {
        for (int j = 1; j < 4; j++)
        {
```

```
            literals[count] = (1 + i - 1) * 81 + (1 + j - 1) * 9 + num;
            int lit = literals[count];
            if (solver->litmap[lit].pos == NULL)//添加变元-子句映射表
            {
                solver->litmap[lit].pos = (int*)malloc(sizeof(int) * clausenum);
                solver->litmap[lit].pos_len = 0;
            }
            solver->litmap[lit].pos[solver->litmap[lit].pos_len++] =
solver->original.count;
            count++;
        }
    }
    addclause(solver, literals, count);
    for (int i = 1; i < 4; i++)//不能填
    {
        for (int j = 1; j < 4; j++)
        {
            if (i == 3 && j == 3)
                break;
            literals[0] = -((1 + i - 1) * 81 + (1 + j - 1) * 9 + num);
            int abslit = abs(literals[0]);
            if (solver->litmap[abslit].neg == NULL)//添加变元-子句映射表
            {
                solver->litmap[abslit].neg = (int*)malloc(sizeof(int) * clausenum);
                solver->litmap[abslit].neg_len = 0;
            }
            solver->litmap[abslit].neg[solver->litmap[abslit].neg_len++] =
solver->original.count;
            for (int k = 1; k < 4; k++)
            {
                for (int p = 1; p < 4; p++)
                {
                    if (k < i)
                    {
                        break;
                    }
                    else
                    {
                        if (p <= j)
                            continue;
                        literals[1] = -((1 + k - 1) * 81 + (1 + p - 1) * 9 + num);
                        int abslit2 = abs(literals[0]);
                        if (solver->litmap[abslit2].neg == NULL)//添加变元-子句映射表
                        {
```

```
                                solver->litmap[abslit2].neg = (int*)malloc(sizeof(int) *
clausenum);
                                solver->litmap[abslit2].neg_len = 0;
                            }

    solver->litmap[abslit2].neg[solver->litmap[abslit2].neg_len++] =
solver->original.count;
                            addclause(solver, literals, 2);
                        }
                    }
                }
            }
        }
}
//窗口约束2
for (int num = 1; num < 10; num++)
{
    int count = 0;
    for (int i = 1; i < 4; i++)//能填
    {
        for (int j = 1; j < 4; j++)
        {
            literals[count] = (5 + i - 1) * 81 + (5 + j - 1) * 9 + num;
            int lit = literals[count];
            if (solver->litmap[lit].pos == NULL)//添加变元-子句映射表
            {
                solver->litmap[lit].pos = (int*)malloc(sizeof(int) * clausenum);
                solver->litmap[lit].pos_len = 0;
            }
            solver->litmap[lit].pos[solver->litmap[lit].pos_len++] =
solver->original.count;
            count++;
        }
    }
    addclause(solver, literals, count);
    for (int i = 1; i < 4; i++)//不能填
    {
        for (int j = 1; j < 4; j++)
        {
            if (i == 3 && j == 3)
                break;
            literals[0] = -((5 + i - 1) * 81 + (5 + j - 1) * 9 + num);
            int abslit = abs(literals[0]);
            if (solver->litmap[abslit].neg == NULL)//添加变元-子句映射表
```

```
            {
                solver->litmap[abslit].neg = (int*)malloc(sizeof(int) * clausenum);
                solver->litmap[abslit].neg_len = 0;
            }
            solver->litmap[abslit].neg[solver->litmap[abslit].neg_len++] =
solver->original.count;
            for (int k = 1; k < 4; k++)
            {
                for (int p = 1; p < 4; p++)
                {
                    if (k < i)
                    {
                        break;
                    }
                    else
                    {
                        if (p <= j)
                            continue;
                        literals[1] = -((5 + k - 1) * 81 + (5 + p - 1) * 9 + num);
                        int abslit2 = abs(literals[0]);
                        if (solver->litmap[abslit2].neg == NULL)//添加变元-子句映射表
                        {
                            solver->litmap[abslit2].neg = (int*)malloc(sizeof(int) *
clausenum);
                            solver->litmap[abslit2].neg_len = 0;
                        }

    solver->litmap[abslit2].neg[solver->litmap[abslit2].neg_len++] =
solver->original.count;
                        addclause(solver, literals, 2);
                    }
                }
            }
        }
    }
}
//填数约束
for (int i = 1; i < 10; i++)
{
    for (int j = 1; j < 10; j++)
    {
        if (chessboard[i][j] != 0)
        {
            literals[0] = (i - 1) * 81 + (j - 1) * 9 + chessboard[i][j];
```

```
            int lit = literals[0];
            if (solver->litmap[lit].pos == NULL)//添加变元-子句映射表
            {
                solver->litmap[lit].pos = (int*)malloc(sizeof(int) * clausenum);
                solver->litmap[lit].pos_len = 0;
            }
            solver->litmap[lit].pos[solver->litmap[lit].pos_len++] =
solver->original.count;
            addclause(solver, literals, 1);
        }
    }
}



            if (!DPLLoptimize(solver))
            {
                printf("此次生成数独失败！\n");
                getchar(); getchar();
                continue;
            }

            //输出到文件
            string newfilename = "C:\\Users\\rui\\Desktop\\暂时使用\\2025秋-程序设计综
合课程设计任务及指导学生包\\助教课设演示检查要求\\sudokuresult.cnf";
            FILE* fp = fopen(newfilename.c_str(), "w");

            fprintf(fp, "p cnf %d %d\n", solver->boolnum, solver->original.count);

            for (int i = 0; i < solver->original.count; i++)
            {
                for (int j = 0; j < solver->original.clauses[i].size; j++)
                {
                    fprintf(fp, "%d ", solver->original.clauses[i].literals[j]);
                }
                fprintf(fp,"0\n");
            }

            fclose(fp);

            while (1)
            {
                bool isend = true;
                for (int i = 1; i < 10; i++)
                {
                    for (int j = 1; j < 10; j++)
```

```
        {
            if (chessboard[i][j] == 0)
            {
                isend = false;
                break;
            }
        }
        if (isend == false)
            break;
    }
    if (isend)
    {
        printf("恭喜你成功完成该百分号数独！！！\n");
        getchar(); getchar();
        break;
    }
    system("cls");
    printf("-----------------------------------------------\n");
    string nandustr;
    switch (nandu)
    {
    case 1:
        nandustr = "简单";
        break;
    case 2:
        nandustr = "中等";
        break;
    case 3:
        nandustr = "困难";
        break;
    case 4:
        nandustr = "极难";
        break;
    default:
        nandustr = "简单";
        break;
    }
    printf("数独难度:%s，提示数个数:%d个\n", nandustr.c_str(), leftnum);
    printf("+---------+---------+---------+\n");
    for (int i = 1; i < 10; i++)
    {
        printf("|");
        for (int j = 1; j < 10; j++)
        {
```

```
                if (chessboard[i][j] != 0)
                {
                    if (renewchessboard[i][j] != 0)
                        printf("\033[31m %d \033[0m", chessboard[i][j]);
                    else
                        printf("\033[34m %d \033[0m", chessboard[i][j]);
                }
                else
                {
                    printf("\033[37m • \033[0m");
                }
                if (j %3 == 0)
                {
                    printf("|");
                }
                if (j == 9)
                    printf("\n");
            }
            if (i % 3 == 0)
            {
                printf("+---------+---------+---------+\n");
            }
        }
        printf("-------------------------------------------------\n");
        printf("|                  请输入指令：                  |\n");
        printf("| 1.手动填充数字(1-9)  2.生成全部答案  3.重来  |\n");
        printf("| 0.不玩了                                      |\n");
        printf("-------------------------------------------------\n");
        int zhiling = 0;
        scanf("%d", &zhiling);
        if (zhiling == 1)
        {
            printf("请依次输入行号，列号和填入数字：\n");
            int x, y, num;
            if (scanf("%d%d%d", &x, &y, &num) == 3 &&
x>0&&x<10&&y>0&&y<10&&num>=0&&num<10)
            {
                if (renewchessboard[x][y] != 0)
                {
                    printf("不能填预设好的位置!\n");
                }
                else
                {
                    if (chessboard[x][y] != num)
```

```
                {
                    if (!issatified(chessboard, x, y, num))
                    {
                        printf("不符合百分号数独约束！\n");
                    }
                    else
                    {
                        chessboard[x][y] = num;
                    }
                }
            }
        }
        else
        {
            printf("输入格式错误，请确保行列号合法，填入数字1-9\n");
        }

    }
    else if (zhiling == 2)
    {
        for (int i = 1; i <= solver->boolnum; i++)
        {
            if (solver->assignment[i] == 1)
            {
                int x = (i - 1) / 81 + 1;
                int y = (i - (x - 1) * 81 - 1) / 9 + 1;
                int number = i - (x - 1) * 81 - (y - 1) * 9;
                answer[x][y] = number;
            }
        }
        //system("cls");
        printf("-----------------------------------------------\n");
        printf("+---------+---------+---------+\n");
        for (int i = 1; i < 10; i++)
        {
            printf("|");
            for (int j = 1; j < 10; j++)
            {
                if (renewchessboard[i][j] != 0)
                {
                    printf("\033[31m %d \033[0m", renewchessboard[i][j]);
                }
                else
                {
```

```
                                printf("\033[34m %d \033[0m", answer[i][j]);
                            }
                            if (j % 3 == 0)
                            {
                                printf("|");
                            }
                            if (j == 9)
                                printf("\n");
                        }
                        if (i % 3 == 0)
                        {
                            printf("+---------+---------+---------+\n");
                        }
                    }
                }
                else if (zhiling == 3)
                {
                    for (int i = 1; i < 10; i++)
                    {
                        for (int j = 1; j < 10; j++)
                        {
                            chessboard[i][j] = renewchessboard[i][j];
                        }
                    }
                }
                else if (zhiling == 0)
                {
                    break;
                }
                getchar();
                getchar();
            }
            free(solver);
        }
        else
        {
            printf("该难度不存在\n");
            getchar();
            getchar();
        }
    }
}
```

9. SATSolver.cpp

```
#include <iostream>
#include <cstdio>
#include <sstream>
#include<ctime>
#include "cnfparser.h"
#include "display.h"
#include "solver.h"
#include "Sudoku.h"
using namespace std;

int boolnum, clausenum;
int* finalassignment;
clock_t begintime,begintime2;
double maxtime = 600; //限制最多运行秒数
bool overtime = false;
bool overtime2 = false;
void init()
{
    boolnum = clausenum = 0;
    overtime = overtime2 = false;
}
int main()
{
    while (1)
    {
        system("cls");
        printf("-----------------------------------------------\n");
        printf("|                  请输入命令编号：             |\n");
        printf("|          1:读取演示要求的cnf文件进行求解      |\n");
        printf("|          2:读取基准算例的cnf文件进行求解      |\n");
        printf("|          3:运行百分号数独游戏                 |\n");
        printf("|          0:结束程序                           |\n");
        printf("-----------------------------------------------\n");
        int command,cnfnumber;
        stringstream ss;
        string fileName;
        scanf("%d", &command);
        if(command == 1)
        {
            printf("请输入要读取第几个cnf文件（演示要求）\n");
            scanf("%d", &cnfnumber);
            ss << cnfnumber;
            fileName = "..\\助教课设演示检查要求\\";
            if ((cnfnumber >= 1 && cnfnumber <= 3) || (cnfnumber >= 5 && cnfnumber <= 6)
```

```
|| cnfnumber == 10 || cnfnumber == 12)
            {
                fileName = fileName + ss.str() + ".cnf";
            }
            else if (cnfnumber == 4 || (cnfnumber >= 7 && cnfnumber <= 9) || cnfnumber ==
11)
            {
                fileName = fileName + ss.str() + "（unsatisfied）" + ".cnf";
            }
            else
            {
                printf("没有这个文件请重新开始\n");
                break;
            }
            init();
            clause* clausearr = NULL;
            int* boolcountarr = NULL;
            Solver* solver = NULL;

            string filename = fileName;
            scancnf(filename, boolcountarr, clausearr, solver);

            showclause(clausearr);        //输出 输入的数据供人工检验

            begintime2 = clock();
            clock_t time1 = clock();
            bool result = DPLLoptimize(solver);
            clock_t time2 = clock();
            printf("优化后：\n");
            if (overtime2)
            {
                printf("s -1\n");
            }
            else
            {
                printf("s %d\n", result ? 1 : 0);
                if (result)
                {
                    printf("v");
                    for (int i = 1; i <= boolnum; i++)
                    {
                        printf(" %d", solver->assignment[i] * i);
                    }
                    printf("\n");
```

```
    }
}
printf("t %.0lf\n", (double)(time2 - time1) / CLOCKS_PER_SEC * 1000);
//输出到文件
string newfilename = filename.erase(filename.length() - 4, 4);
newfilename += ".res";
FILE* fp = fopen(newfilename.c_str(), "w");
if (overtime2)
{
    fprintf(fp,"s -1\n");
}
else
{
    fprintf(fp,"s %d\n", result ? 1 : 0);
    if (result)
    {
        fprintf(fp,"v");
        for (int i = 1; i <= boolnum; i++)
        {
            fprintf(fp," %d", solver->assignment[i] * i);
        }
        fprintf(fp,"\n");
    }
}
fprintf(fp,"t %.0lf\n", (double)(time2 - time1) / CLOCKS_PER_SEC * 1000);

fclose(fp);

free(solver);

printf("////////////////////////////////\n");

int* assignment = (int*)calloc(boolnum + 1, sizeof(int));

finalassignment = (int*)calloc(boolnum + 1, sizeof(int));

begintime = clock();
clock_t starttime = clock();
bool issolved = DPLLordinary(clausearr, boolcountarr, assignment);
clock_t endtime = clock();
printf("优化前：\n");
if (overtime)
{
    printf("s -1\n");
```

```
            }
            else
            {
                printf("s %d\n", issolved ? 1 : 0);
                if (issolved)
                {
                    printf("v");
                    for (int i = 1; i <= boolnum; i++)
                    {
                        printf(" %d", finalassignment[i] * i);
                    }
                    printf("\n");
                }
            }

            printf("t %.0lf\n", (double)(endtime - starttime) / CLOCKS_PER_SEC * 1000);

            free(assignment);
            free(finalassignment);
            if (boolcountarr != NULL)
                free(boolcountarr);

            if (result && issolved)
            {
                printf("优化率：%.1lf\n", (double)(((double)(endtime - starttime) /
CLOCKS_PER_SEC - (double)(time2 - time1) / CLOCKS_PER_SEC) / ((double)(endtime - starttime)
/ CLOCKS_PER_SEC)) * 100);
            }


        }
        else if (command == 2)
        {
            printf("继续输入命令编号：\n");
            printf("1.功能测试:sat-20.cnf\n");
            printf("2.功能测试:unsat-5cnf-30.cnf\n");
            printf("3.性能测试:ais10.cnf\n");
            printf("4.性能测试:sud00009.cnf\n");
            scanf("%d", &cnfnumber);
            fileName = "..\\SAT测试备选算例\\基准算例\\";
            if (cnfnumber == 1)
            {
                fileName = fileName + "功能测试\\" + "sat-20.cnf";
            }
```

```
else if (cnfnumber == 2)
{
    fileName = fileName + "功能测试\\" + "unsat-5cnf-30.cnf";
}
else if (cnfnumber == 3)
{
    fileName = fileName + "性能测试\\" + "ais10.cnf";
}
else if (cnfnumber == 4)
{
    fileName = fileName + "性能测试\\" + "sud00009.cnf";
}
else
{
    printf("没有这个文件请重新开始\n");
}
init();
clause* clausearr = NULL;
int* boolcountarr = NULL;
Solver* solver = NULL;

string filename = fileName;
scancnf(filename, boolcountarr, clausearr, solver);

showclause(clausearr);        //输出 输入的数据供人工检验

begintime2 = clock();
clock_t time1 = clock();
bool result = DPLLoptimize(solver);
clock_t time2 = clock();
if (overtime2)
{
    printf("s -1\n");
}
else
{
    printf("s %d\n", result ? 1 : 0);
    if (result)
    {
        printf("v");
        for (int i = 1; i <= boolnum; i++)
        {
            printf(" %d", solver->assignment[i] * i);
        }
```

```
            printf("\n");
        }
    }
    printf("t %.0lf\n", (double)(time2 - time1) / CLOCKS_PER_SEC * 1000);

    //输出到文件
    string newfilename = filename.erase(filename.length() - 4, 4);
    newfilename += ".res";
    FILE* fp = fopen(newfilename.c_str(), "w");
    if (overtime2)
    {
        fprintf(fp, "s -1\n");
    }
    else
    {
        fprintf(fp, "s %d\n", result ? 1 : 0);
        if (result)
        {
            fprintf(fp, "v");
            for (int i = 1; i <= boolnum; i++)
            {
                fprintf(fp, " %d", solver->assignment[i] * i);
            }
            fprintf(fp, "\n");
        }
    }
    fprintf(fp, "t %.0lf\n", (double)(time2 - time1) / CLOCKS_PER_SEC * 1000);

    fclose(fp);

    free(solver);

    printf("////////////////////////////////\n");

    int* assignment = (int*)calloc(boolnum + 1, sizeof(int));

    finalassignment = (int*)calloc(boolnum + 1, sizeof(int));

    begintime = clock();
    clock_t starttime = clock();
    bool issolved = DPLLordinary(clausearr, boolcountarr, assignment);
    clock_t endtime = clock();
    if (overtime)
    {
```

```
                printf("s -1\n");
            }
            else
            {
                printf("s %d\n", issolved ? 1 : 0);
                if (issolved)
                {
                    printf("v");
                    for (int i = 1; i <= boolnum; i++)
                    {
                        printf(" %d", finalassignment[i] * i);
                    }
                    printf("\n");
                }
            }

            printf("t %.0lf\n", (double)(endtime - starttime) / CLOCKS_PER_SEC * 1000);

            free(assignment);
            free(finalassignment);
            if (boolcountarr != NULL)
                free(boolcountarr);

            if (result && issolved)
            {
                printf("优化率：%.11f\n", (double)(((double)(endtime - starttime) /
CLOCKS_PER_SEC - (double)(time2 - time1) / CLOCKS_PER_SEC) / ((double)(endtime - starttime)
/ CLOCKS_PER_SEC)) * 100);
            }
        }
        else if (command == 0)
        {
            break;
        }
        else if (command == 3)
        {
            init();
            sudoku();
        }
        /*//else if (command == 4)//个人测试使用
        //{
        //    fileName = "C:\\Users\\rui\\Desktop\\暂时使用\\2025秋-程序设计综合课程设计
任务及指导学生包\\程序设计综合课程设计任务及指导学生包\\SAT测试备选算例\\不满足算例
\\u-problem7-50.cnf";
```

```
//    init();
//    clause* clausearr = NULL;
//    int* boolcountarr = NULL;
//    Solver* solver = NULL;

//    string filename = fileName;
//    scancnf(filename, boolcountarr, clausearr, solver);

//    //showclause(clausearr);       //输出 输入的数据供人工检验

//    begintime2 = clock();
//    clock_t time1 = clock();
//    bool result = DPLLoptimize(solver);
//    clock_t time2 = clock();
//    if (overtime2)
//    {
//        printf("s -1\n");
//    }
//    else
//    {
//        printf("s %d\n", result ? 1 : 0);
//        if (result)
//        {
//            printf("v");
//            for (int i = 1; i <= boolnum; i++)
//            {
//                printf(" %d", solver->assignment[i] * i);
//            }
//            printf("\n");
//        }
//    }
//    printf("t %.0lf\n", (double)(time2 - time1) / CLOCKS_PER_SEC * 1000);

//    free(solver);

//    printf("////////////////////////////////\n");

//    int* assignment = (int*)calloc(boolnum + 1, sizeof(int));

//    finalassignment = (int*)calloc(boolnum + 1, sizeof(int));

//    begintime = clock();
//    clock_t starttime = clock();
//    bool issolved = DPLLordinary(clausearr, boolcountarr, assignment);
```

```
        //    clock_t endtime = clock();
        //    if (overtime)
        //    {
        //        printf("s -1\n");
        //    }
        //    else
        //    {
        //        printf("s %d\n", issolved ? 1 : 0);
        //        if (issolved)
        //        {
        //            printf("v");
        //            for (int i = 1; i <= boolnum; i++)
        //            {
        //                printf(" %d", finalassignment[i] * i);
        //            }
        //            printf("\n");
        //        }
        //    }

        //    printf("t %.0lf\n", (double)(endtime - starttime) / CLOCKS_PER_SEC * 1000);

        //    free(assignment);
        //    free(finalassignment);
        //    if (boolcountarr != NULL)
        //        free(boolcountarr);

        //    if (result && issolved)
        //    {
        //        printf("优化率：%.11f\n", (double)(((double)(endtime - starttime) /
CLOCKS_PER_SEC - (double)(time2 - time1) / CLOCKS_PER_SEC) / ((double)(endtime - starttime)
/ CLOCKS_PER_SEC)) * 100);
        //    }
        //}*/
        else
        {
            printf("没有这个命令请重新开始\n");
        }
        getchar();
        getchar();
    }
    return 0;
}
```