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：U202414735

姓　　名：肖金诺

指导老师：李丹

报告日期：2025/9/13

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

1 引言

1.1 课题背景与意义

作为华中科技大学计算机科学与技术专业的大二学生，在前两个学期中，我们已经系统地学习了《C 语言程序设计》与《数据结构》两门核心专业基础理论课，并完成了对应的编程实践课程。通过上述课程的学习与训练，我们不仅掌握了 C 语言的核心语法、典型数据结构（如线性表、栈、队列、树、图）的逻辑与物理结构，更初步具备了数据抽象与建模、算法设计与实现、以及程序调试与测试的问题求解能力，为承担更复杂的程序设计任务奠定了坚实的知识基础。

然而，之前的实验课程多侧重于对特定数据结构或算法机制的验证性训练，题目规模相对较小，问题定义和算法框架也常由教师预先设定。本次《基于 SAT 的百分号数独游戏求解程序》的综合课程设计，正是引导我们从验证性编程向综合性、设计性实践跨越的关键环节。它要求我们自主完成从问题分析、模型建立、算法选择、数据结构设计、程序实现到测试优化的全流程，是对以往所学知识的一次深度整合与综合运用。

该设计课题直面一个经典的 NP 完全问题（SAT），并挑战其在一个有趣且复杂的应用场景（百分号数独）中的求解，极具理论深度与实践趣味。完成此设计，将极大地锻炼我们的系统设计能力、工程实现能力和创新思维能力，是实现编程能力升华、培养计算思维、并为后续《编译原理》、《操作系统》等计算机系统类课程的学习打下坚实基础的至关重要一步，旨在让综合编程技能成为我们固有能力与通向未来专业之门的钥匙。

1.1.1 SAT 问题的理论核心与实践价值

命题逻辑公式可满足性问题（SAT）是计算机科学理论中一座里程碑式的存在。作为首个被证明的 NP 完全问题，SAT 在计算复杂性理论中占据着核心地位，理解 SAT 问题本质上是理解 NP 类问题计算难度的钥匙。NP 完全性意味着，该问题本身可能存在内在的计算困难，但一旦获得解，验证其正确性却相对容易。这一特性使得 SAT 成为了连接理论计算机科学与实际应用的桥梁。

在实践层面，SAT 问题的求解技术已不再是纯理论的空中楼阁，而是广泛应用于芯片设计、形式化方法、人工智能、软件安全验证等工业界关键领域。许多复杂的约束满足、规划与调度问题都可以被归约（Reduction）为 SAT 问题，进

而利用高效的 SAT 求解器得到解答。因此，学习和实现一个 SAT 求解器，不仅仅是完成一个算法作业，更是直面一个在真实世界中具有巨大影响力的核心计算问题，是一次从理论到实践的深刻演练。

1.1.2 DPLL 算法的奠基作用与设计挑战

DPLL 算法是 SAT 求解领域经久不衰的经典与基石。尽管后续发展出了冲突驱动子句学习（CDCL）等更高效的现代算法，但它们无一不是建立在 DPLL 算法的框架之上。DPLL 算法清晰地展现了如何通过确定性算法（回溯搜索）来应对 NP 难问题的典型范式，其核心的“单子句传播”和“分裂策略”是理解所有基于搜索的 SAT 求解器工作原理的蓝本。

1.2 国内外研究现状

SAT 求解器的研究在国际上一直处于活跃状态。早期的研究以 DPLL 算法为基础，此后出现了革命性的冲突驱动子句学习（CDCL）框架，成为现代 SAT 求解器的绝对主流。CDCL 在 DPLL 的回溯搜索基础上，引入了冲突分析、子句学习、非时序回溯和随机重启等机制，极大地提升了求解效率，催生出了如 MiniSat、Glucose、Lingeling 等一批高效求解器，能够处理百万变量级别的工业级问题，广泛应用于硬件验证、软件安全等领域。当前的研究前沿还包括利用机器学习优化启发式策略、并行求解以及满足性模理论（SMT）等更高级的应用。

国内在该领域的研究同样成果显著，清华大学、北京大学、中国科学院等机构在算法优化和应用方面均有深入探索。国内研究注重与国际前沿接轨，并将 SAT 技术广泛应用于集成电路设计、人工智能、网络安全等国家重大需求领域。

尽管 CDCL 等现代技术已成为主流，但 DPLL 算法作为其核心基石，其教育价值依然至关重要。对于本科生而言，完整实现 DPLL 算法是理解 SAT 求解原理不可或缺的基础训练。本设计选择从 DPLL 算法入手，旨在夯实基础，深刻理解 SAT 求解的基本流程和挑战，为未来学习更复杂的 CDCL 算法打下坚实的理论和实践基础。

1.3 课程设计的主要研究工作

本课程设计围绕基于 DPLL 算法的 SAT 求解器及其在百分号数独游戏求解中

的应用展开系统性研究与实践。首先，需要深入理解 DPLL 算法的核心机理，包括单子句传播规则与分裂策略的实现细节，并设计递归与回溯框架。其次，需设计适合 CNF 公式表示的数据结构，如采用链表结构存储子句和文字，以支持高效的公式解析与算法操作。在此基础上，实现 DIMACS 标准格式的 CNF 文件解析功能与求解结果输出功能，确保程序的完整性与可用性。为进一步提升求解效率，拟引入一种启发式分支变元选取策略（如基于文字出现频率的动态启发式），并与基础 DPLL 实现进行性能对比分析，量化优化效果。同时，重点研究百分号数独的规则约束体系，设计将其转化为 SAT 问题的编码方案，实现从数独局面自动生成 CNF 公式的模块，最终通过已实现的求解器完成游戏求解。最后，通过设计多组测试用例，包括可满足与不可满足的 SAT 算例以及不同难度的百分号数独问题，对求解器的正确性、鲁棒性及性能进行综合评估，完成从理论设计到工程实现的完整闭环。

2 系统需求分析与总体设计

2.1 系统需求分析

本系统旨在实现一个基于 DPLL 算法的 SAT 求解器，并将其应用于求解百分号数独问题。根据设计任务要求，系统需满足以下需求：

功能需求：

1.输入输出功能：支持命令行参数输入，能够正确读取 CNF 文件，并将求解结果输出到指定格式的文件中

2.公式解析与验证：能够正确解析 CNF 文件内容，建立公式的内部数据结构表示，并提供验证功能以确认解析的正确性

3.DPLL 算法核心:实现完整的 DPLL 算法流程,包括单子句传播和分裂策略，能够正确判断 SAT 问题的可满足性

4.性能测量：集成时间测量功能，记录算法执行时间（毫秒级）

5.程序优化：至少实现一种优化策略（如分支变量选择策略优化）

6.SAT 应用：将百分号数独问题转化为 SAT 问题，并集成到求解器中

非功能需求：

1.正确性：对给定算例能够给出正确的可满足性判断

2.性能：能够处理变元数在 600 以内的 SAT 算例

3.健壮性：能够处理格式错误的输入文件并给出适当提示

4. 可扩展性：代码结构清晰，模块化设计，便于后续功能扩展

约束条件：

1.开发语言：C 语言

2.数据结构：不能使用 C++标准库容器，需自行实现必要的数据结构

3.模块要求：需按功能模块划分源代码文件

验收标准：

1.能够正确求解不少于 18 个测试算例（包含可满足和不可满足情况）

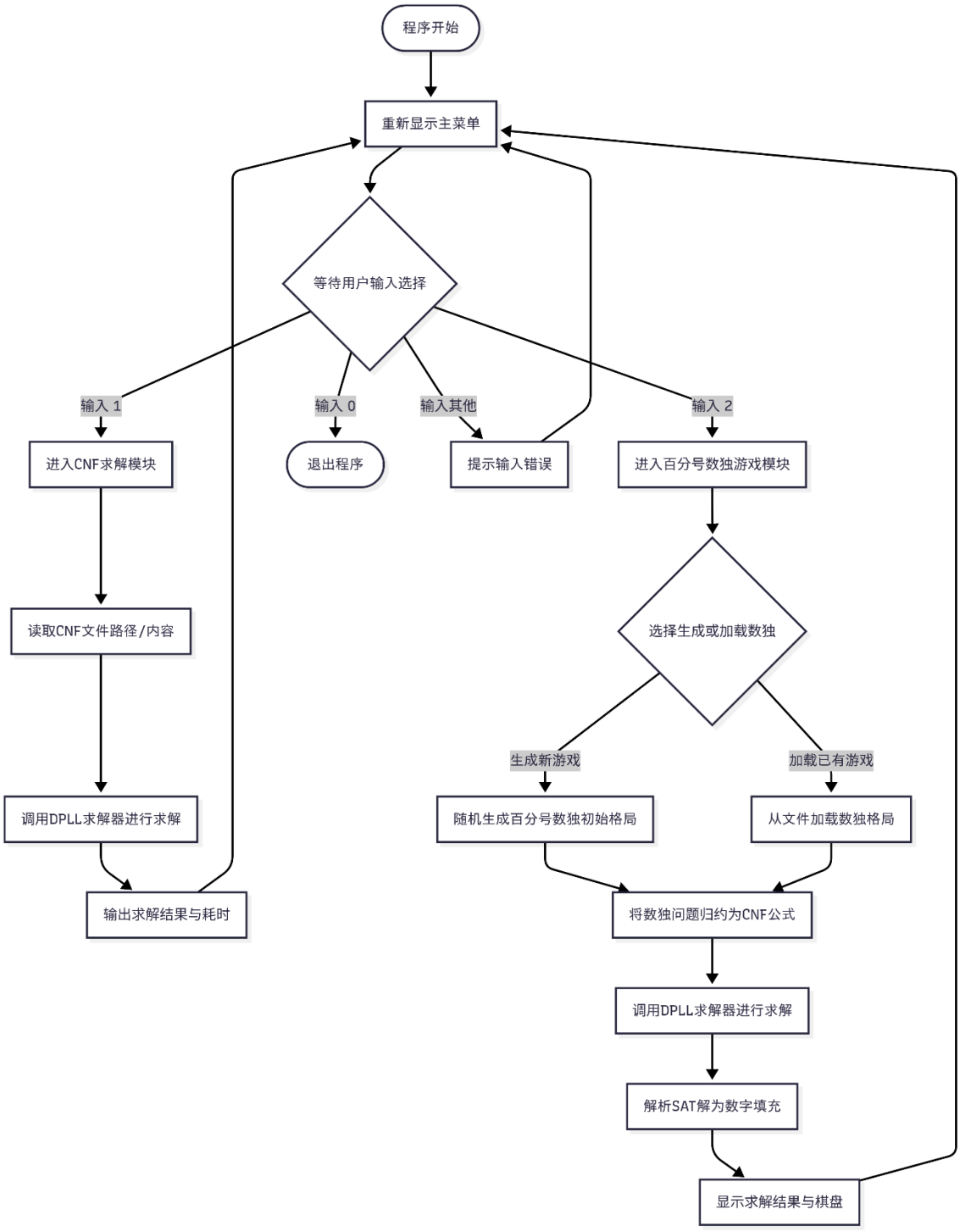
2.优化策略需提供明确的性能优化率数据

3.百分号数独应用模块能够完成问题转化和求解

4. 代码符合规范，有适当的注释和文档说明

2.2 系统总体设计

程序具有三个重要的功能，以及两个相关的功能：



图表 1 总体流程示意图

2.2.1 Cnf 算例求解

CNF 算例求解的核心任务是：给定一个合取范式（Conjunctive Normal Form, CNF）公式，判定其是否存在一组使公式为真的布尔变元赋值。如果存在，则称该公式是可满足的（Satisfiable, SAT），并输出一组解；否则，称其为不可满足的（Unsatisfiable, UNSAT）。

一个 CNF 公式 FF 是多个子句（Clause）的合取（逻辑与 ∧），其形式化定义为：

F=c1∧c2∧...∧cmF=c1∧c2∧...∧cm

其中每个子句 cici 是多个文字（Literal）的析取（逻辑或 ∨），即：

ci=l1∨l2∨...∨lkci=l1∨l2∨...∨lk

文字 ljlj 要么是一个布尔变元（如 x），要么是该变元的否定（如¬x）。

求解过程的直观理解

求解 CNF 公式可以类比于解决一个有多重约束的决策问题：

1.公式（F） 代表整个待满足的问题。

2. 子句（c_i） 代表问题中的一个约束条件。整个问题要求所有约束条件必须同时被满足。

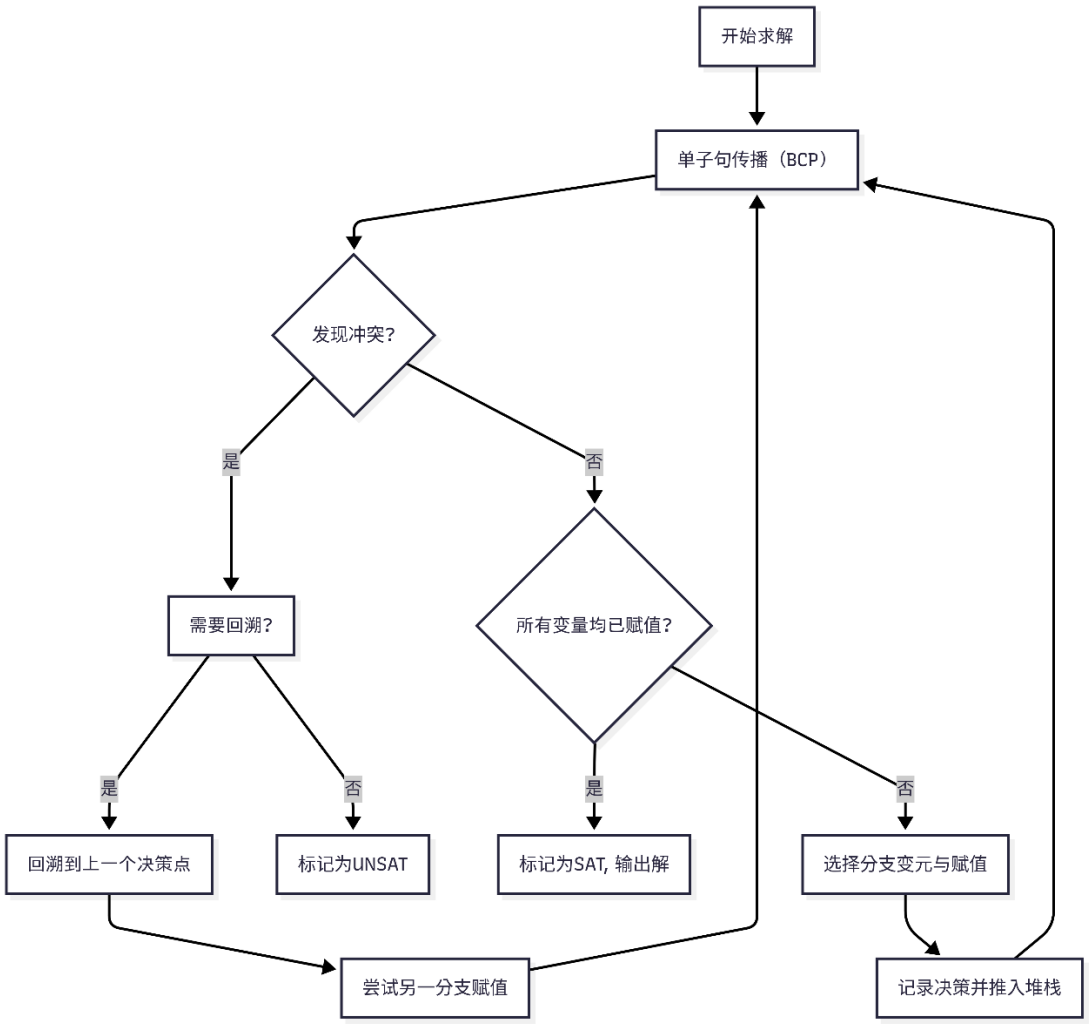
3.文字（l_j） 代表一个最基础的决策选项。

4.子句中的“或”关系（∨） 意味着在一个约束内，只需满足多个选项中的任意一个即可。

5.最终目标：找到一组对所有变元的赋值（即做出所有决策），使得每一个子句中至少有一个文字为真。

标准求解流程（以 DPLL 算法为例）

CNF 公式的求解通常遵循一个系统性的搜索、推理与回溯过程，其基本流程如下图所示，该图清晰地展示了算法从初始化到最终给出判决的核心路径与决策循环。



图表 2 cnf 求解流程示意图

2.2.2 游玩数独

数独游戏模块是本系统将核心 SAT 求解功能应用于实际趣味逻辑问题的直观体现。该模块提供了一个交互式界面，允许用户生成、加载、手动求解并最终通过 SAT 求解器自动求解百分号数独游戏。其功能组成与操作流程如下图所示，该模块围绕一个核心的“游戏状态”运行，用户可通过菜单驱动各种操作，最终通过调用 SAT 求解核心完成求解。

1. 随机生成游戏（挖空法）

系统应实现基于“挖洞法”的算法，自动生成具有唯一解的百分号数独初始格局。

生成原理：首先由一个合法且完整的百分号数独终盘（可通过算法生成或从解倒推）开始，按照一定策略（如随机顺序、对称挖洞）依次挖去（即清空）特定位置上的数字，并在每次挖洞后验证剩余格局是否仍保证唯一解。若挖洞后导

致多解，则回填该位置，尝试挖取其他位置。

难度控制：通过控制挖洞的数量和策略来间接控制游戏难度。挖洞越多，通常难度越高。本设计需至少实现一种难度级别。

评定标准：如任务书所述，实现该自动生成算法为“优”级评定标准。

2. 导入数独或 CNF 文件

系统提供从外部文件加载游戏进度的功能，分为两种方式：

导入数独格局：从文本文件（如.txt、.sdk）或自定义格式文件中读取一个百分号数独的初始格局。文件内容通常为一个 9x9 的矩阵，用数字 0 或点号.表示空白格，并需包含窗口与对角线约束的标记或默认符合百分号数独布局。系统解析文件后，将格局加载到游戏界面中。

导入 CNF 文件：允许用户直接加载一个预先由其他工具或系统本身生成的、表示某个数独问题的 CNF 文件。系统读取该 CNF 文件后，需反向解析出其对应的数独初始格局（即从单子句中提取出提示数），并加载到棋盘上。此功能展示了 SAT 问题与数独问题的可逆转换关系。

3. 填写数字

提供基本的游戏交互功能，允许用户手动参与求解过程。

操作方式：用户通过命令行输入（如行号 列号 数字）或图形界面（如光标移动后输入）在空白格中填入数字。

实时验证：系统应对用户每次输入进行实时合法性检查。检查依据包括普通数独的行、列、宫约束，以及百分号数独特有的窗口约束和对角线约束。当用户输入违反任何一条约束时，应立即给出错误提示（如高亮显示冲突位置），并可选择允许或禁止此次非法输入。

4. 系统求解数独

此功能是本模块的核心，直接调用已实现的 SAT 求解器来自动求解当前棋盘。

归约过程：系统将当前的数独格局（初始提示数 + 用户已填的正确数字）视为新的“初始格局”，根据 2.4 节所述的编码规则，动态生成对应的 CNF 公式。该公式包含了基本数独约束、百分号附加约束以及当前所有已填数字所对应的单子句约束。

调用求解器：将生成的 CNF 公式传递给系统的 DPLL 求解器核心模块进行求解。

结果显示：求解完成后，若为 SAT，则解析解向量，将得到的数字解填充到棋盘的空白格中，并以不同颜色高亮显示，以示与原始提示数的区别。若为 UNSAT，应提示用户“当前格局无解”，这可能是由于用户手动填写了错误的数字导致矛盾。

5. 导出数独为 CNF 文件

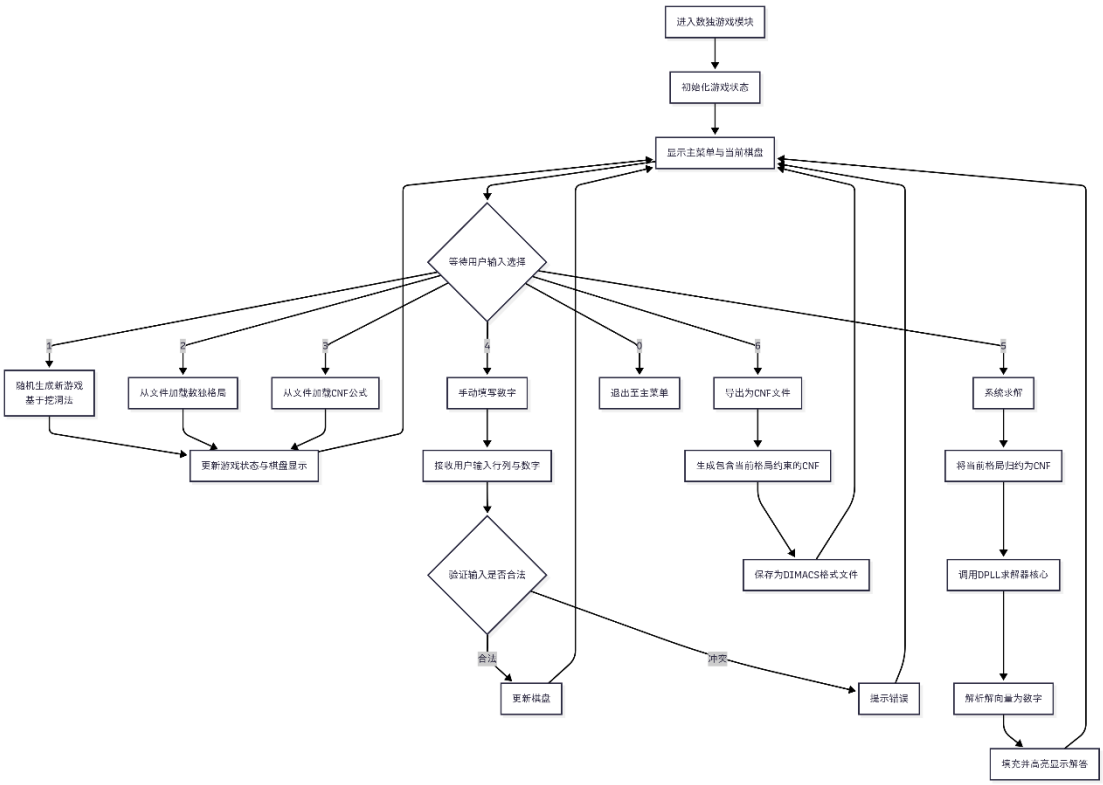
此功能将当前数独游戏格局（包括原始提示数和用户已填入的数字）输出为一个标准的 DIMACS 格式的 CNF 文件。

用途：便于存档、与其他 SAT 求解器进行交叉验证，或作为后续分析的算例。

实现：系统根据编码规则，将数独的所有规则约束（基本约束+百分号约束）以及当前所有已知数字（视为单子句）全部写入一个 CNF 文件。该文件完整定义了当前数独问题。

6. 退出数独游戏模式

结束数独游戏模块的交互过程，返回到 2.2.1 节所述的程序主菜单，用户可以继续选择进行 CNF 求解或退出程序。退出前可提示用户是否保存当前游戏进度或导出的 CNF 文件。



图表 3 数独游戏流程示意图

3 系统详细设计

3.1 有关数据结构的定义

本 SAT 求解器的实现基于 C 语言，其核心在于设计高效、灵活的数据结构来表示 CNF 公式及其组成部分（变元、文字、子句），并支持 DPLL 算法中的各种操作（如单子句传播、回溯、启发式选择等）。以下对各关键数据结构进行详细说明。

3.1.1 核心数据结构设计

1. 变元（Variable）状态数组

数据结构：signed char *val

功能说明：这是一个长度为 n+1（n 为变元总数）的动态数组，用于记录每个变元（索引 1 至 n）当前的赋值状态。

取值含义：

-1：表示该变元尚未被赋值（UNASSIGNED）。

1：表示该变元被赋值为真（TRUE）。

0：表示该变元被赋值为假（FALSE）。

设计 rationale：使用 signed char 类型在保证足够表示范围的同时，节省内存空间。该数组是算法进行赋值、回溯和评估的基础。

2. 文字（Literal）的相关操作

功能函数：

int lit_var(int lit): 提取文字的变元序号（取绝对值）。

int eval_lit(int lit, const signed char *val): 评估一个文字在当前赋值下的真值。返回 1（真）、0（假）或-1（未赋值）。

int lit_index(int lit) 与 int index_to_lit(int idx): 实现文字与其在监视链表数组中索引值的双向转换。此设计将文字 v 映射为索引 2*v，文字-v 映射为索引 2*v+1，极大方便了基于文字快速查找其相关的子句（监视链表）。

3. 子句（Clause）结构体

```
typedef struct ClauseLit {
    int lit;
```

```
    struct ClauseLit *next;
} ClauseLit;

typedef struct Clause {
    int lit_nums;       // 子句中文字的数量
    ClauseLit *head;    // 子句文字链表的头指针
    ClauseLit *tail;    // 子句文字链表的尾指针
    struct Clause *next; // 指向公式中下一个子句的指针
    int watch1;         // 第一个监视文字
    int watch2;         // 第二个监视文字
} Clause;
```

功能说明：

ClauseLit 节点构成了子句内部的文字链表。

Clause 结构体使用带头尾指针的单链表来管理文字，使得在子句尾部添加文字的操作时间复杂度为 O(1)。

watch1 和 watch2 是实现双文字监视（Two-Literal Watching） 这一关键优化技术的核心字段。它们记录了子句中当前被“监视”的两个文字，用于高效地实现单子句传播(Unit Propagation)，避免在每次赋值后遍历整个子句链表。

设计 rationale：链表结构可以灵活地处理不同长度的子句。双监视文字机制是现代 SAT 求解器的标准高效配置，它能最小化因变元赋值而需要检查的子句数量。

4. 公式（Formula）结构体

```
typedef struct Formula {
    int var_nums;    // 公式中变元的总数
    int clause_nums; // 公式中子句的总数
    Clause *head;    // 公式子句链表的头指针
    Clause *tail;    // 公式子句链表的尾指针
} Formula;
```

功能说明：该结构体代表整个 CNF 公式。它使用一个带头尾指针的单链表来管理所有的子句，同样支持 O(1)时间复杂度的子句追加操作。

设计 rationale：完整地表征一个 CNF 算例，是 DPLL 算法处理的核心对象。

5. 监视器（Watcher）结构体与监视链表

```
typedef struct Watcher {
    Clause *cl;            // 指向被监视子句的指针
    struct Watcher *next; // 下一个监视器节点
} Watcher;
```

功能说明：Watcher 结构体是构成监视链表的基本节点。一个监视链表将所有监视同一个文字的句子句链接起来。

全局变量：Watcher **watches; 这是一个指针数组，大小为 2*(n+1)。数组的每个元素（如 watches[lit_index(lit)]）是一个链表头，指向所有监视文字 lit 的 Watcher 节点。

设计 rationale：此数据结构是双文字监视策略的基石。当一个文字被赋值为假时，算法只需遍历其对应的监视链表，检查这些子句是否需要更新监视位置或触发单元传播，从而极大地提高了 BCP 过程的效率。

3.1.2 辅助数据结构

1. 踪迹栈（Trail Stack）

数据结构：int *trail; int trail_top;

功能说明：

trail 数组按顺序记录历次赋值决策（文字），既包括单元传播产生的赋值，也包括分支决策产生的赋值。

trail_top 指向栈顶。

设计 rationale：该栈是实现回溯（Backtracking） 的关键。当发生冲突时，算法可以沿着踪迹栈回溯到之前的某个决策点，撤销其后的所有赋值。trail 中通常还会与另一个栈（决策层栈）配合使用，以标记每次分支决策的起始位置（在您提供的代码片段中未直接体现，但 dpll 函数的参数暗示了其存在）。

2. 传播队列（Propagation Queue）

数据结构：int *queue; int qhead; int qtail;

功能说明：这是一个 FIFO（先进先出）队列，用于在单子句传播过程中暂存新产生的被赋值为真的文字（单元文字），以便继续传播其影响。

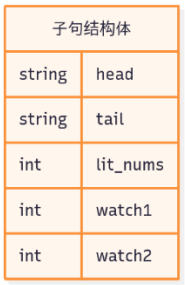
设计 rationale：使用队列可以确保单元传播以广度优先的方式进行，高效且完备。

核心数据结构功能表如下：

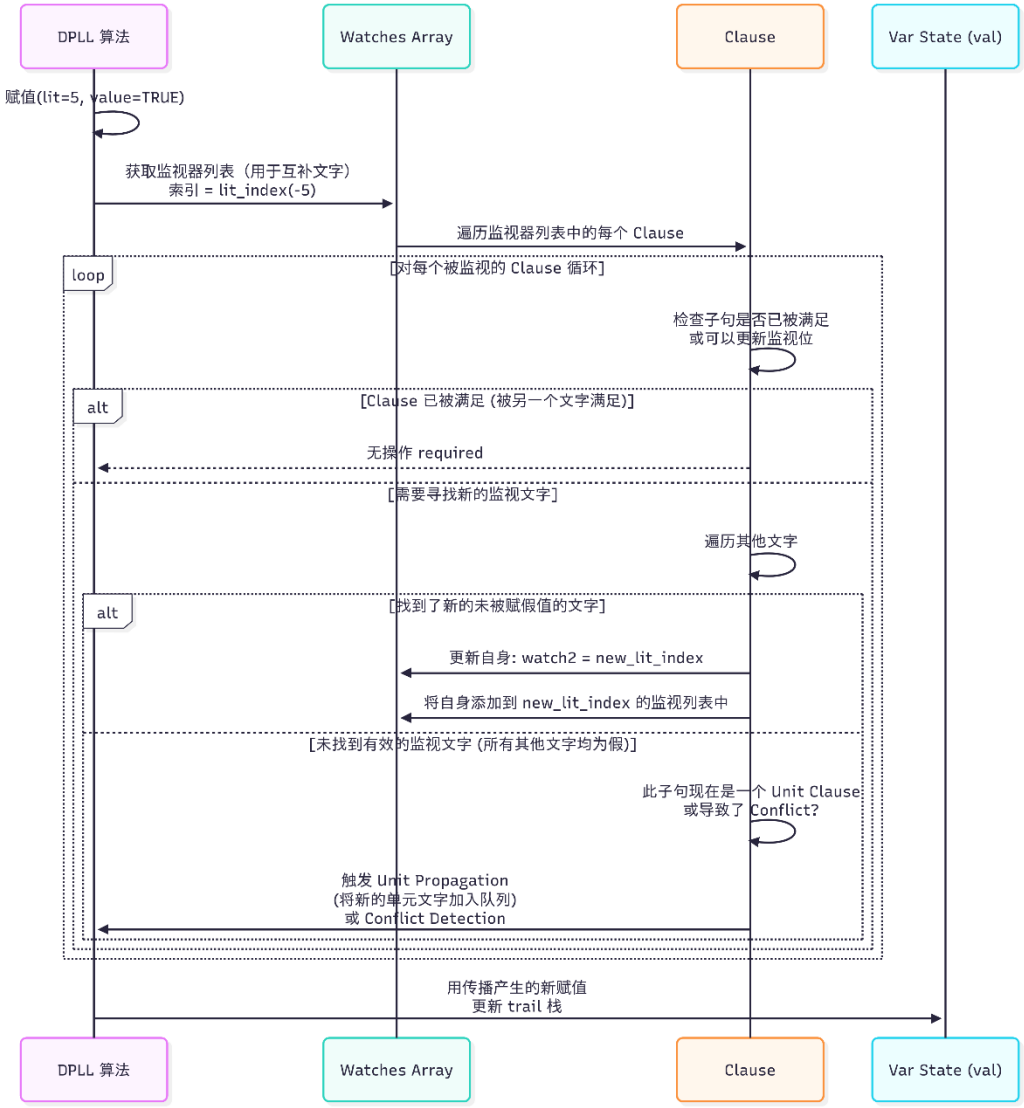
数据结构名	类型/定义	功能描述	关键成员/元素
变元状态数组	signed char *val	记录每个变元当前的赋值状态（真、假、未赋值）。索引从 1 到 n。	val[i] = -1（未赋值），0（假），1（真）
文字节点	struct ClauseLit	构成子句的基本单元，表示一个文字。	int lit（文字值），next（指向下一个文字节点的指针）
子句结构体	struct Clause	表示一个 CNF 子句，是由文字节点构成的链表。实现了双文字监视策略。	head，tail（文字链表头尾指针），lit_nums（文字数量），watch1，watch2（两个监视文字的索引），next（指向下一个子句的指针）
公式结构体	struct Formula	表示整个 CNF 公式，是由子句结构体构成的链表。	head，tail（子句链表头尾指针），var_nums（变元总数），clause_nums（子句总数）

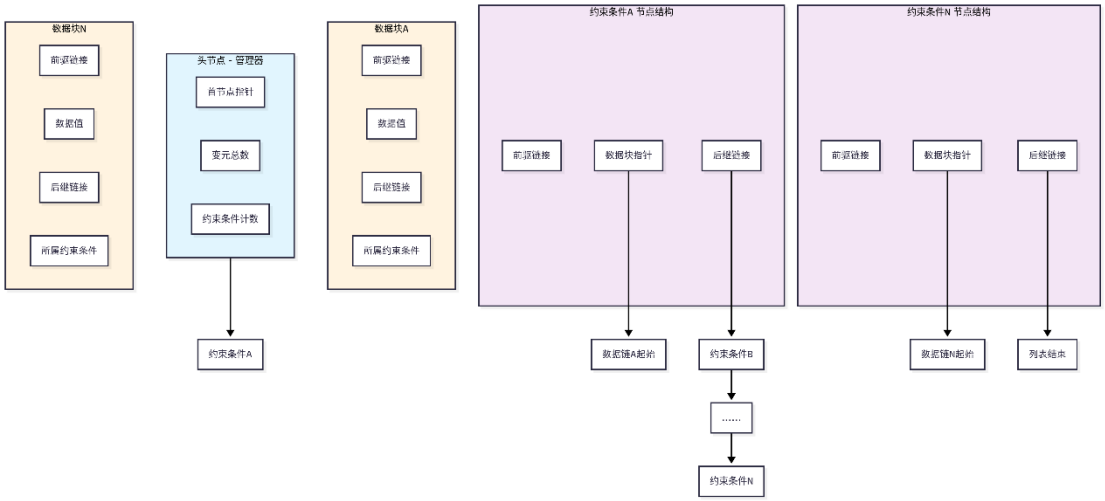
监视器节点	struct Watcher	监视链表的节点，将子句与它所监视的文字关联起来。	Clause *cl（指向被监视子句的指针），next（指向下一个监视器节点的指针）
监视链表数组	Watcher **watches	一个指针数组，索引是文字的索引（lit_index(lit)）。每个元素是一个链表，链接了所有监视对应文字的子句。	watches[i] 是一个 Watcher*，指向监视文字 index_to_lit(i) 的子句链表。
踪迹栈	int *trail, int trail_top	一个栈（数组+栈顶指针），按顺序记录所有赋值决策（文字），用于回溯。	trail 数组存储文字值，trail_top 是栈顶索引。
传播队列	int *queue, int qhead, int qtail	一个 FIFO 队列（数组+头尾指针），在单元传播时暂存新产生的单元文字。	queue 数组存储文字值，qhead 是队列头索引，qtail 是队列尾索引。

数据结构如下：



数据结构相互作用示意图如下图：





3.2 主要算法设计

本系统的算法设计分为两大核心部分：一是基于 DPLL 框架的 CNF 公式求解

引擎，二是围绕百分号数独游戏的生成、转化与求解的应用模块。

3.2.1 CNF 公式求解算法设计

CNF 公式求解的核心是基于 DPLL 算法框架,并集成了双文字监视(2-Watched Literals）和启发式分支策略等优化技术。

1. DPLL 算法主流程

算法 dpll()是递归的回溯搜索过程，其伪代码如下：

算法: dpll
输入: 公式 F, 变元赋值数组 val, 踪迹栈 trail
输出: SAT (1) 或 UNSAT (0)

1. 单子句传播（BCP）:
   - 调用 propagate_watched 进行基于监视的单元传播
   - 若发现冲突，则返回 UNSAT (0)

2. 若所有变元均已被赋值:
   - 调用 sat_satisfied 验证解是否完全满足公式 F
   - 若满足，保存解并返回 SAT (1); 否则返回 UNSAT (0)

3. 分支决策:
   - 调用 jw_heuristic_choose 选择分支变元 v 及其试探极性 polarity
   - 将决策点（v, polarity）和当前 trail 深度压入决策栈

4. 尝试第一分支:
   - 赋值: val[v] = polarity
   - 将文字 v * polarity 压入 trail 栈
   - 递归调用 dpll
   - 若返回 SAT (1), 则向上返回 SAT (1)

5. 回溯并尝试第二分支:

- 回溯到上一个决策点（撤销该决策点之后的所有赋值）
- 赋值: val[v] = 1 - polarity
- 将文字 v * (1-polarity) 压入 trail 栈
- 递归调用 dpll 并返回其结果

2. 单子句传播（BCP）与双文字监视机制

propagate_watched 是实现高效 BCP 的核心，其流程如下：

从传播队列中取出一个文字 l（其值为真）。

取其互补文字 l_comp = -l，计算其索引 idx = lit_index(l_comp)。

遍历 watches[idx]指向的监视链表中的所有子句 c。

对于每个子句 c：

a. 若其另一个监视文字为真，则该子句已满足，跳过。

b. 否则，遍历子句中的其他文字，寻找一个既非假也未被监视的文字 l_new。

c. 若找到，则将子句 c 从当前监视链表 watches[idx]中移除，并添加到 watches[lit_index(l_new)]的链表中，同时更新 c 的监视文字。

d. 若未找到（意味着除被监视的假文字外，其他文字均为假），则剩下的那个未赋值文字即为单元文字。将其加入传播队列。

e. 若连一个未赋值的文字都找不到（所有文字均为假），则触发冲突。

3. 启发式分支变元选择策略

本设计实现了 JW（Jerosolim-Wah）启发式策略的变种，函数 jw_heuristic_choose 的工作流程如下：

遍历所有未赋值的变元。

对每个变元 v，计算其正文字 v 和负文字-v 在所有未满足子句中出现的次数（权重）。

文字的权重计算采用近似策略：weight = 1 << (JW_SHIFT - k)，其中 k 是该文字所在子句的长度。此公式赋予短子句中的文字更高的权重。

变元 v 的最终得分是其正负文字权重之和。选择得分最高的变元进行分支。

极性选择：选择其正负文字中权重较高的一方作为首次试探赋值方向。

3.2.2 数独游戏模块算法设计

1. 百分号数独生成算法（挖洞法）

算法：generate_sudoku

输入：无

输出：一个具有唯一解的百分号数独初始格局

1. 生成一个完全随机的、满足所有约束（行、列、宫、窗口、对角线）的百分号数独终盘。
2. 复制终盘作为当前游戏格局。
3. 随机选择一个单元格，尝试将其清空（挖洞）。
4. 调用 SAT 求解器验证当前格局是否仍具有唯一解。

- 此处需采用更严格的验证：归约为 CNF 后，求解并确认是否存在不同于原终盘的另一解。

5. 若唯一解性质保持不变，则确认此次挖洞，否则回填该单元格。
6. 重复步骤 3-5，直到达到预定的挖洞数量或难度级别。
7. 返回最终的初始格局。

2. 数独问题归约算法（数独 → CNF）

算法：sudoku_to_cnf

输入：数独棋盘 board（9x9 二维数组，0 表示空白）

输出：表示该数独问题的 CNF 公式 F

1. 初始化一个空的公式 F，变元总数设为 729（9 行 x9 列 x9 种可能）。
2. 生成规则约束子句:

a. 单元格约束：对 81 个单元格，每个生成“至少填一个数”和“至多填一个数”的子句集。

b. 行约束 & 列约束：对 9 行 9 列的每一行每一列，生成“每个数字必须出现一次”的子句集。

c. 宫约束：对 9 个 3x3 宫，生成“每个数字必须出现一次”的子句集。

d. 百分号附加约束：对 2 个窗口和 2 条对角线，生成“每个数字必须出现一次”的子句集。

- 所有这些约束子句被添加到公式 F 中。

3. 生成已知数约束（单子句）：

- 遍历棋盘，对于每个非零数字 n（位于<i, j>），生成一个单子句：`(x_ijk)`（其中 k=n），并将其加入公式 F。

4. 返回公式 F。

3. 数独求解与答案解析

将当前棋盘状态（初始提示数 + 用户正确填入的数字）归约为 CNF 公式。

调用 solve()函数求解此 CNF 公式。

若结果为 SAT，则解向量是一个长度为 729 的赋值数组。解析该数组：

对于每个变元索引 idx，将其转换为(i, j, k)三元组。

若该变元被赋值为真，则在数独棋盘的<i, j>位置填入数字 k。

将解析出的数字填充到图形界面或输出终端，并以不同颜色区分解答和原题。

4 系统实现与测试

4.1 系统实现

4.1.1 开发与运行环境

本系统的开发与测试在以下环境中进行：

硬件环境：x86-64 架构的通用计算机，内存≥8GB。满足处理大型 CNF 算例（变元数>600）的内存需求。

操作系统： Windows 11。

开发工具：代码使用 C 语言 编写，主要编译器为 MinGW-w64 (Windows)，集成开发环境为 Visual Studio Code。

依赖项：实现为标准 C99。

4.1.2 数据结构的实现

根据 3.1 节的设计，系统中定义了以下核心数据类型（其具体声明见附录源代码）：

ClauseLit 结构体：表示子句中的一个文字，是构成子句链表的基本节点。

Clause 结构体：表示一个 CNF 子句。其核心成员包括指向文字链表的头尾指针(head, tail)、子句长度(lit_nums)、两个监视文字索引(watch1, watch2)以及指向下一个子句的指针(next)。

Formula 结构体：表示整个 CNF 公式。其核心成员包括指向子句链表的头尾指针(head, tail)、变元总数(var_nums)和子句总数(clause_nums)。

Watcher 结构体：监视链表的节点，用于关联子句与被监视的文字。

全局数组：

signed char *val：变元状态数组，记录每个变元的赋值（真、假、未赋值）。

Watcher **watches：监视链表数组，是实现高效单子句传播的关键。

int *trail：踪迹栈，与栈顶指针 trail_top 配合，记录赋值顺序以实现回溯。

int *queue, qhead, qtail：传播队列，用于在单子句传播时管理新产生的单元文字。

4.1.3 系统函数说明与模块化实现

系统的实现严格遵循模块化原则，将不同的功能封装成独立的函数，并通过清晰的调用关系组织起来。所有功能均基于 3.1 节设计的数据结构进行构建。以下对各模块的主要函数及其功能进行详细说明。

1. CNF 解析模块

该模块负责读取并解析标准的 DIMACS 格式 CNF 文件，将其转换为系统内部的公式表示（Formula 结构体）。

parseCNF(FILE *fp, Formula *F)：此函数是文件解析的核心入口。它逐行读取输入文件，首先调用 skipCom 函数跳过所有以字符'c'开头的注释行。当遇到以'p'开头的行时，解析出变元总数和子句总数并初始化公式 F。随后，为每一个子句行，调用 clause_create 创建一个新的子句结构体，并循环调用 readNextInt 读取子句中的文字,使用 clause_add_lit 将文字添加到该子句中。最后，使用 formula_append 将构建完成的子句添加到公式 F 的尾部，从而建立起完整的子句链表。

readNextInt(FILE *fp, int *out)：一个辅助工具函数，用于从文件流中安全地读取下一个整数，处理可能的 IO 错误，是 parseCNF 的基础。

skipCom(FILE *fp)：另一个辅助函数，用于识别并跳过输入流中的注释行，确保解析逻辑正确聚焦于有效数据。

2. 核心数据结构操作模块

此模块提供对核心数据结构（Clause, Formula, 监视链表）的创建、修改和销毁等基本操作。

clause_create(void)：负责在堆内存中分配并初始化一个空的 Clause 对象，设置其文字链表头尾指针为 NULL，并初始化监视文字等字段。

clause_add_lit(Clause *c, int lit)：用于向一个已存在的子句 c 的文字链表尾部追加一个新的文字。此函数维护子句文字链表的完整性，是构建子句内容的关键。

formula_append(Formula *f, Clause *c)：将一个新构建好的 Clause 对象链接到 Formula 结构的子句链表末尾，并更新公式的子句计数。

watcher_build(const Formula *F, Watcher ***watches_out, int *cap_out)：在公式 F 解析完成后，此函数被调用以构建双文字监视机制所需的

watches 数组。它遍历公式中的每一个子句，为每个子句的初始监视文字创建 Watcher 节点，并将其插入到对应文字的监视链表中。

3. DPLL 算法核心模块

这是本系统的计算中枢，实现了 SAT 求解的完整回溯搜索逻辑。

dpll(const Formula *F, signed char *var, int *trail, int *trailtop)：DPLL 算法的递归入口和主控制器。它首先调用 propagate_watched 进行单子句传播和冲突检测。若未发现冲突且存在未赋值变元，则调用 jw_heuristic_choose 选择分支变元和极性，进行赋值和递归调用。若递归调用返回冲突，则调用 backtrack 进行回溯并尝试另一分支。

propagate_watched(const Formula *F, Watcher **watches, signed char *val, int *trail, int *trail_top, int *queue, int *qhead, int *qtail)：这是系统中最关键的性能敏感函数。它基于构建好的 watches 数组，高效地处理因赋值而触发的子句状态更新和单元文字推导，是实现快速 BCP 的核心。

jw_heuristic_choose(const Formula *F, const signed char *var, int *polarity)：实现 JW 启发式分支策略。它遍历未赋值的变元，计算其正负文字在所有未满足短子句中的权重，选择得分最高的变元及其初始试探极性。

backtrack(signed char *var, int *trail, int *trailtop, int old_top)：当搜索路径发生冲突时，此函数负责回溯到指定的决策层数（old_top），通过重置 val 数组中相应变元的状态和调整 trail_top 指针来撤销之后的所有赋值。

4. 工具、验证与 I/O 模块

该模块提供通用的工具函数、解验证以及结果输出功能。

lit_var(int lit), eval_lit(int lit, const signed char *val)：基础的文字操作函数，用于获取文字对应的变元号以及评估文字在当前赋值下的真值。

verify(const Formula *F, const signed char *var)：在求解器声称找到解后，此函数被调用来全面验证当前赋值是否满足公式 F 中的所有子句，确保解的正确性。

solve(const Formula *F, const char *cnfName)：求解流程的高级封装函数。它协调了解析后的主要流程：初始化运行环境（分配 val, trail 等数组）、调用 dpll 执行求解、计时、验证结果，并最终调用输出函数将结果写入.res 文件。

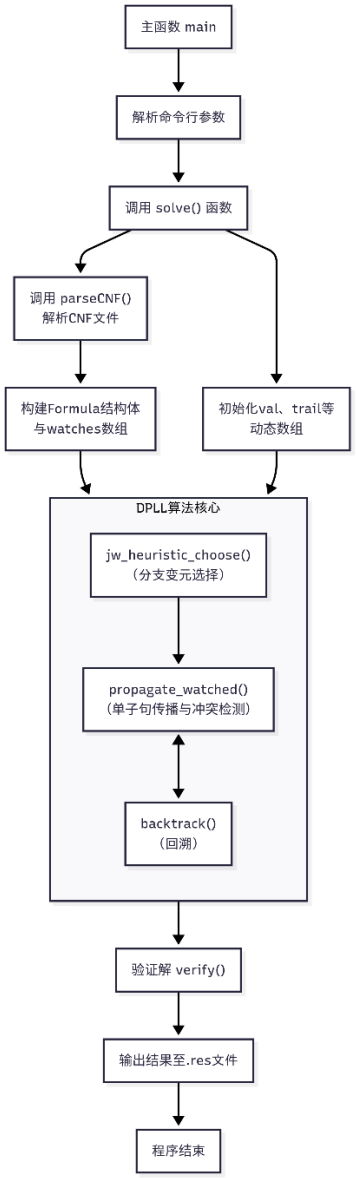
res_name(const char *cnfName, char *resName)：根据输入的 CNF 文件名生成对应结果文件的路径。

main(int argc, char **argv)：程序的最终入口点。处理命令行参数，调用 parseCNF 和 solve，驱动整个求解流程。

各函数通过清晰的调用层次结构进行协作，共同构成了一个完整、高效且模块化的 SAT 求解系统。数独应用模块将通过调用 CNF 生成函数和最终的 solve 函数，复用本系统的核心求解能力。

4.1.4 函数调用关系与系统工作流程

系统中主要函数的调用关系，清晰地反映了程序从启动到完成求解的控制流与数据流，其核心流程如下图所示：



其工作流程简述如下：

初始化：main 函数是程序的入口，它接收命令行参数（如 CNF 文件名），并调用 solve 函数进入求解流程。

解析与构建：solve 函数首先调用 parseCNF 函数。该函数读取 CNF 文件，并调用 clause_create、clause_add_lit、formula_append 等一系列函数，逐步构建出表示整个公式的 Formula 结构体和子句链表。随后，调用 watcher_build

函数构建监视链表数组 watches。

求解核心：solve 函数初始化赋值数组 val 和踪迹栈 trail 后，调用 dpll 函数开始递归求解。

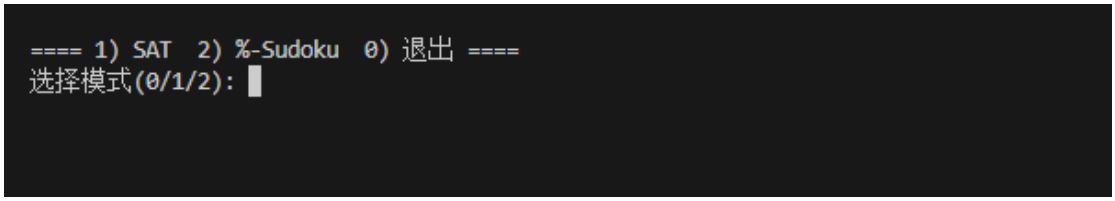
DPLL 过程：在 dpll 函数内部，循环调用 propagate_watched 进行单子句传播和冲突检测。若无冲突且未求解完毕，则调用 jw_heuristic_choose 选择分支变元，进行赋值和递归调用。若遇到冲突，则调用 backtrack 进行回溯。

结果处理：dpll 返回后，solve 函数通过 verify 验证解的正确性，最后将结果和统计信息（如求解时间，通过 now_ns 计算）写入.res 文件。

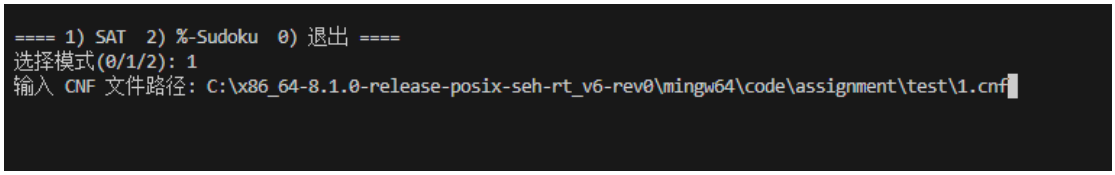
数独游戏模块的函数（如数独生成、归约、交互等）虽未在上述代码中体现，但其实现将遵循类似的模块化原则，并通过调用 solve 函数这个核心求解器来完成其功能。

4.2 系统测试

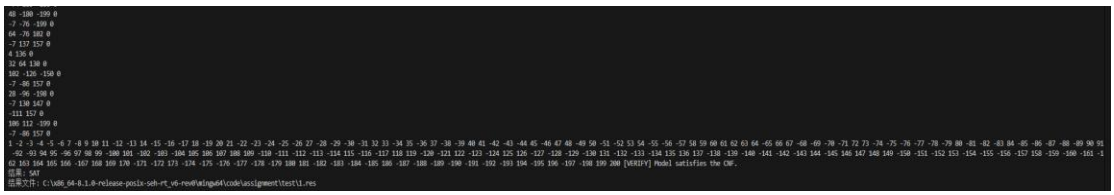
主页面：



Sat 模式：

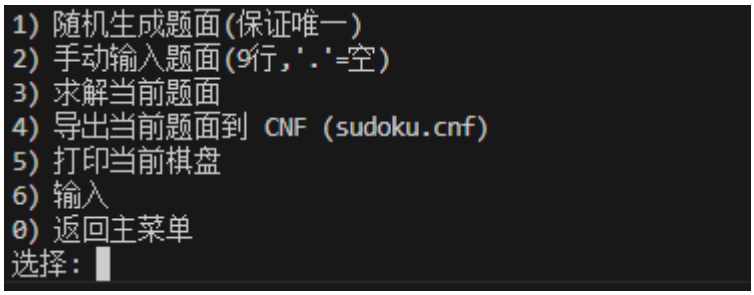


解析 cnf 文件并输出结果：

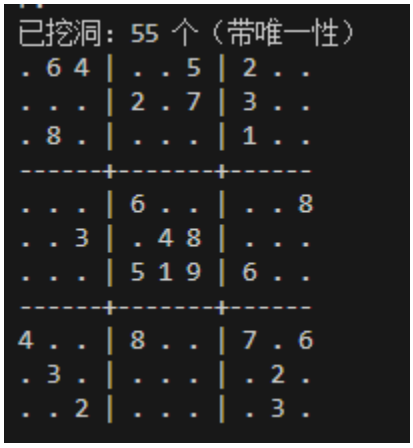


选择数独模式(2)：

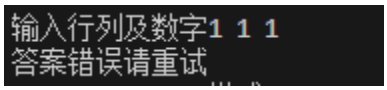
游戏页面：

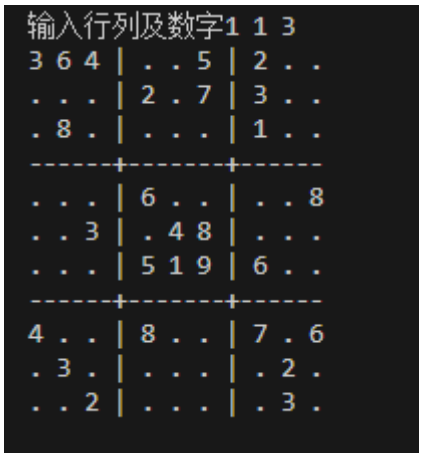


随机生成数独：



输入答案(错误与正确两种情况)：





输出答案：

```
解:
3 6 4 | 1 9 5 | 2 8 7
1 9 5 | 2 8 7 | 3 6 4
2 8 7 | 3 6 4 | 1 9 5
------+-------+------
9 4 1 | 6 3 2 | 5 7 8
6 5 3 | 7 4 8 | 9 1 2
7 2 8 | 5 1 9 | 6 4 3
------+-------+------
4 1 9 | 8 2 3 | 7 5 6
5 3 6 | 4 7 1 | 8 2 9
8 7 2 | 9 5 6 | 4 3 1
```

退出程序：

```
 1) 随机生成题面(保证唯一)
 2) 手动输入题面(9行,'.'=空)
 3) 求解当前题面
 4) 导出当前题面到 CNF (sudoku.cnf)
 5) 打印当前棋盘
 6) 输入
 0) 返回主菜单
 选择: 0

 ==== 1) SAT  2) %-Sudoku  0) 退出 ====
 选择模式(0/1/2): 0
 程序结束。
 PS C:\x86_64-8.1.0-release-posix-seh-rt_v6-rev
```

一些报错：

```
==== 1) SAT  2) %-Sudoku  0) 退出 ====
选择模式(0/1/2): 4
无效选择。
```

```
==== 1) SAT  2) %-Sudoku  0) 退出 ====
选择模式(0/1/2): 1
输入 CNF 文件路径: 22
无法打开文件: 22
```

得到的 res 文件：

```
1  s 1
2  v -1 2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 -16 -17 -18 -19 -20 -21 22 -23 -24 -25 -26 27 -28 -
3  t 80.528128
4
```

```
1  s 0
2  t 2382.354000
3
```

5 总结与展望

5.1 全文总结

敲下最后一行代码，完成这份报告，也意味着这段充满挑战与乐趣的程序设计之旅暂时告一段落。回看这次课程设计，它远不止是一次简单的编程作业，更像是一次将理论知识付诸实践的完整探险。

从最初拿到题目，看到“SAT 求解器”、“DPLL”、“百分号数独”这些概念时的“一头雾水”，到后来一点点翻阅资料、理解算法、画流程图、设计数据结构，再到最后在调试中看着程序成功解出第一个 SAT 算例和数独难题，这个过程充满了“山重水复疑无路，柳暗花明又一村”的成就感。

这次设计让我真切体会到，一个好的程序不仅仅是能让编译器通过，更重要的是其背后清晰的设计思路和高效的数据结构。为了实现高效的单元传播，我亲手实现了双文字监视（2-Watched Literals） 这一巧妙的机制；为了让求解器“ smarter”，我钻研并实现了 JW 启发式分支策略。当看到这些课堂上学习的算法和数据结构（链表、栈、队列）被完美地应用起来，并实实在在地提升了程序性能时，那种兴奋感是无可替代的。

更重要的是，这次项目让我对“复杂系统”的开发有了更深的敬畏之心。如何将数独的规则严谨地编码成 CNF 公式，如何管理内存避免泄漏，如何设计函数模块保证代码既清晰又可扩展……每一个细节都考验着编程功底和工程设计能力。

当然，过程中也没少和“Bug”作斗争。有时候一个微小的逻辑错误就可能导致整个求解器陷入死循环或者给出错误答案，排查的过程虽然痛苦，但每一次成功的 Debug 都让我对程序的理解更深一层。

总而言之，这次课程设计让我过足了“编程瘾”。它不仅仅让我巩固了 C 语言编程、数据结构与算法这些基础知识，更让我体验了一个从无到有、从一个想法到一个完整可运行程序的创造过程。这其中的乐趣、挫折与最终的收获，无疑是我专业学习道路上非常宝贵的一笔财富。

5.1 工作展望

虽然眼前的这个 SAT 求解器已经能够顺利地运行，但在我心里，这更像是一

个充满潜力的“初版”，还有很多让我跃跃欲试的想法和可以优化的方向。编程的魅力不就在于此吗？永远有下一座山峰可以攀登。

首先，我最想动手改造的就是用冲突驱动子句学习（CDCL） 来彻底升级现在的 DPLL 框架。现在的回溯还显得有些“笨拙”，每次遇到冲突只能一步步撤销。而 CDCL 能够从冲突中学习，生成新的子句来避免未来走入同样的死胡同，这肯定会让求解器的智商和效率飙升一个档次，想想就让人觉得兴奋。

其次，在启发式策略上，我也只是尝鲜了一下 JW 策略。未来我特别想试试更主流的、像 VSIDS 这样的动态启发式策略，让它能在求解过程中不断学习，自适应地调整分支顺序，应该会非常酷。

当然，还有性能这个永恒的话题。现在的数据结构已经不错，但肯定还有压榨的空间。比如，能不能用更紧凑的内存布局来缓存子句？或者甚至引入并行计算，让多个求解线程同时探索不同的分支？虽然挑战很大，但光是想想能让它在大型算例上跑得更快，就有点手痒。

对于数独游戏部分，现在的界面还只是个“命令行玩具”。我真心希望能给它做一个漂亮的图形界面（GUI），让用户可以点点鼠标就能填数字、提示、检查错误，那样才像一个真正完整的产品。同时，数独生成算法也可以做得更智能，不仅能控制难度，还能生成不同对称模式的、更“优雅”的谜题。

最后，这个项目也让我意识到工程化和测试的重要性。未来如果继续迭代，我一定要引入更完善的单元测试框架和性能分析工具，稳扎稳打地提升代码质量和可靠性。

总之，这个项目与其说是一个结束，不如说为我打开了一扇新的大门，让我看到了理论和实践之间那道有趣的鸿沟，以及填补它的无限可能。我已经开始期待下一个版本了！

6 体会

回望整个项目的开发过程，与其说是完成了一项任务，不如说是亲手搭建了一个复杂的生态系统。其中最大的体会就是：数据结构的设计是程序的骨架，直接决定了算法的效率和实现的优雅程度。

最初，我选择用最直观的“子句链表”和“文字链表”来构建公式。但当实现双文字监视机制时，我才真正体会到这种设计的威力。在 propagate_watched 函数中，看着程序通过 watches 数组精准地定位受影响的子句，而不是傻傻地遍历整个公式，那种性能提升带来的快感是实实在在的。这让我明白，在系统编程中，“选择比努力更重要”，一个聪明的数据结构顶得上一百行优化代码。

另一个深刻的体会是关于调试的艺术。SAT 求解器的 Bug 极其狡猾。记得有一次，求解器对一个简单算例给出了错误解。我花了整整一个下午，逐行跟踪 dpll 的递归和 backtrack 的回溯过程，最后发现是在回溯恢复赋值状态时，漏掉了一个 val 数组的更新。还有一次，内存泄漏导致大规模算例运行时崩溃，使用 Valgrind 一点点分析才定位到是一个析构函数中的 ClauseLit 节点没有完全释放。这个过程虽然痛苦，但却极大地锻炼了我系统性排查和解决问题的能力。它教会我，清晰的模块边界和良好的日志输出，就是程序员最好的调试器。

此外，我也更加理解了理论到实践的鸿沟。课本上的 DPLL 算法伪代码可能只有十几行，但将其实现为一个健壮、高效的工程系统，需要考虑无数的细节：内存管理、文件解析、错误处理、算法优化等等。将数独规则转化为 CNF 公式的编码过程，更是对逻辑思维和严谨性的终极考验。任何一个约束遗漏或编码错误，都会导致生成的 SAT 问题与原问题不等价。

最后，这个项目让我感受到了创造的乐趣。从零开始，亲手搭建起一个能够解决复杂逻辑问题的系统，并看着它成功运行，这种成就感是无与伦比的。它让我对编程的热情不再是停留在表面，而是真正体会到了用代码构建复杂系统的深度乐趣。这无疑是我大学生活中最充实的一段技术旅程之一。

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

附录

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#include <time.h>

#define MAXVAR 729

#define VAR(i,j,k)    ( ( (i)-1 )*81 + ( (j)-1 )*9 + (k) )

#define UNIT_QUEUE_MAX    (MAXVAR*2+1)

int decision_level = 0;


#define DEBUG_LEVEL 0


#if DEBUG_LEVEL > 0

#define DEBUG_PRINT(level, ...) if (DEBUG_LEVEL >= level)
{ printf(__VA_ARGS__); fflush(stdout); }

#else

#define DEBUG_PRINT(level, ...)

#endif

typedef struct Clause {

      int *lits;

      int size;

      bool sat;

      struct Clause *next;

      int remaining;
```

```
	int id;
} Clause;
typedef struct OccurrenceNode {
	Clause *clause;
	struct OccurrenceNode *next;
} OccurrenceNode;

typedef struct Var {
	char value;
	bool decided;
	OccurrenceNode *pos;
	OccurrenceNode *neg;
} Var;

extern Clause *clauses;
extern Var	vars[730];
int	var_cnt;
int	assignment[730];
int decision[MAXVAR + 1];
int level = 0;
int	trail_top = 0;
int	trail[MAXVAR];
int clause_cnt = 0;
int clause_id_counter = 0;

typedef struct {
```

```
    int   v;
    int   branch;
    int   level;
} Frame;

Frame stack[MAXVAR];
int    sp = 0;
Clause *new_clause(const int *lits, int size);
Clause *insert_clause_head(Clause *head, Clause *newclause);
void add_cell_constraints(Clause **head);
void add_row_constraints(Clause **head);
void add_col_constraints(Clause **head);
void add_box_constraints(Clause **head);
void add_diagonal_constraints(Clause **head);
void add_window_constraints(Clause **head);
Clause *read_sdk_to_clauses(const char *filename);
Clause *read_cnf(const char *fname, int *var_cnt, int *clause_cnt);
void build_occurrence();
void print_conflict_clause(Clause *c);
bool unit_propagate(void);
double get_ms(void);
int pick_var_moms(void);
int pick_var(void);
bool dpll(void);
void save_assign(int v, int val);
void backtrack(int level);
```

```
void write_cnf(const char *fname, int var_cnt, Clause *clauses);
void fill_box(char grid[9][9], int row, int col);
void solve_sudoku(char grid[9][9]);
bool is_complete(char grid[9][9]);
bool is_valid(char grid[9][9]);
void play_sudoku();
void print_sudoku(char grid[9][9]);
Clause* create_sudoku_cnf(char grid[9][9]);


void save_assign(int v, int val)
{
	assignment[v]    = val;
	vars[v].decided = true;
	vars[v].value = val;
	trail[trail_top++] = v;
	DEBUG_PRINT(2, "[DEBUG] Assignment: variable %d = %d\n", v, val);
}


void backtrack(int level)
{
	DEBUG_PRINT(2, "[DEBUG] Backtracking to level %d (current
level: %d)\n", level, decision_level);


	for (Clause *c = clauses; c; c = c->next) {
		c->sat = false;
		c->remaining = 0;
```

```
            for (int *p = c->lits; *p; ++p) {
                if (!vars[abs(*p)].decided) {
                    c->remaining++;
                }
            }
        }

        while (trail_top > level)
        {
            --trail_top;
            int v = trail[trail_top];
            assignment[v]    = 0;
            vars[v].decided = false;
            vars[v].value    = 0;
            DEBUG_PRINT(2, "[DEBUG] Undo assignment: variable %d\n", v);
        }
}

Var vars[730];
Clause *clauses;

void write_cnf(const char *fname, int var_cnt, Clause *clauses)
{
        FILE *fp = fopen(fname, "w");
        if (!fp) { perror("open"); exit(EXIT_FAILURE); }
```

```
    int clause_cnt = 0;
    for (Clause *p = clauses; p; p = p->next) ++clause_cnt;
    fprintf(fp, "p cnf %d %d\n", var_cnt, clause_cnt);

    for (Clause *p = clauses; p; p = p->next) {
        for (int i = 0; i < p->size; ++i)
            fprintf(fp, "%d ", p->lits[i]);
        fprintf(fp, "0\n");
    }

    fclose(fp);
}

Clause *new_clause(const int *lits, int size)
{
    Clause *c = (Clause*)malloc(sizeof(Clause));
    c->lits = (int*)malloc((size + 1) * sizeof(int));
    for (int i = 0; i < size; ++i) c->lits[i] = lits[i];
    c->lits[size] = 0;    // End with 0
    c->size = size;
    c->next = NULL;
    c->sat = 0;
    c->id = clause_id_counter++; // Assign unique ID
    DEBUG_PRINT(3, "[DEBUG] Created new clause ID=%d: ", c->id);
    for (int i = 0; i < size; ++i) DEBUG_PRINT(3, "%d ", lits[i]);
    DEBUG_PRINT(3, "0\n");
```

```
	return c;
}

//O(1)
Clause *insert_clause_head(Clause *head, Clause *newclause)
{
	DEBUG_PRINT(3, "[DEBUG] Inserting clause ID=%d at head\n",
newclause->id);
	newclause->next = head;
	return newclause;
}

void add_cell_constraints(Clause **head)
{
	DEBUG_PRINT(1, "[DEBUG] Adding cell constraints\n");
	for (int i = 1; i <= 9; ++i)
		for (int j = 1; j <= 9; ++j) {
			int at_least[9];
			for (int k = 1; k <= 9; ++k) at_least[k-1]
=   VAR(i,j,k); // Each cell must have a number
			*head = insert_clause_head(*head,
new_clause(at_least, 9));

			for (int k1 = 1; k1 <= 9; ++k1)
				for (int k2 = k1+1; k2 <= 9; ++k2) {
					int lits[2] = { -VAR(i,j,k1),
-VAR(i,j,k2) }; // Each cell can only have one number
```

```
                                *head = insert_clause_head(*head,
new_clause(lits, 2));
                        }
            }
}

void add_row_constraints(Clause **head)
{
      DEBUG_PRINT(1, "[DEBUG] Adding row constraints\n");
      for (int r = 1; r <= 9; ++r)
            for (int k = 1; k <= 9; ++k) {
                  int exist[9];
                  for (int c = 1; c <= 9; ++c) exist[c-1] = VAR(r,c,k);
// Each row must contain each number
                  *head = insert_clause_head(*head, new_clause(exist,
9));

                  for (int c1 = 1; c1 <= 9; ++c1)
                        for (int c2 = c1+1; c2 <= 9; ++c2) {
                              int lits[2] = { -VAR(r,c1,k),
-VAR(r,c2,k) };
                              *head = insert_clause_head(*head,
new_clause(lits, 2)); // Each number can appear only once per row
                        }
            }
}
```

```
void add_col_constraints(Clause **head)
{
	DEBUG_PRINT(1, "[DEBUG] Adding column constraints\n");
	for (int c = 1; c <= 9; ++c)
		for (int k = 1; k <= 9; ++k) {
			int exist[9];
			for (int r = 1; r <= 9; ++r) exist[r-1] = VAR(r,c,k);
// Each column must contain each number
			*head = insert_clause_head(*head, new_clause(exist,
9));

			for (int r1 = 1; r1 <= 9; ++r1)
				for (int r2 = r1+1; r2 <= 9; ++r2) {
					int lits[2] = { -VAR(r1,c,k),
-VAR(r2,c,k) };
					*head = insert_clause_head(*head,
new_clause(lits, 2)); // Each number can appear only once per column
				}
		}
}

void add_box_constraints(Clause **head)
{
	DEBUG_PRINT(1, "[DEBUG] Adding box constraints\n");
	for (int br = 0; br < 3; ++br)
		for (int bc = 0; bc < 3; ++bc)
			for (int k = 1; k <= 9; ++k) {
```

```
                        int exist[9], idx = 0;
                        for (int i = 1; i <= 3; ++i)
                              for (int j = 1; j <= 3; ++j)
                                    exist[idx++] = VAR(br*3+i, bc*3+j,
k);
                        *head = insert_clause_head(*head,
new_clause(exist, 9)); // Each box must contain each number


                        for (int i1 = 1; i1 <= 3; ++i1)
                              for (int j1 = 1; j1 <= 3; ++j1)
                                    for (int i2 = i1; i2 <= 3; ++i2)
                                          for (int j2 = (i1==i2?j1+1:1);
j2 <= 3; ++j2) {
                                                int lits[2] = {
                                                      -VAR(br*3+i1,
bc*3+j1, k),
                                                      -VAR(br*3+i2,
bc*3+j2, k)
                                                };
                                                *head =
insert_clause_head(*head, new_clause(lits, 2)); // Each number can
appear only once per box
                                          }
                  }
}

void add_diagonal_constraints(Clause **head)
{
```

```
	DEBUG_PRINT(1, "[DEBUG] Adding diagonal constraints\n");
	for (int k = 1; k <= 9; ++k) {
		int exist[9];
		for (int r = 1; r <= 9; ++r)
			exist[r-1] = VAR(r, 10 - r, k);
		*head = insert_clause_head(*head, new_clause(exist, 9)); //
Diagonal must contain each number

		for (int r1 = 1; r1 <= 9; ++r1)
			for (int r2 = r1 + 1; r2 <= 9; ++r2) {
				int lits[2] = {
					-VAR(r1, 10 - r1, k),
					-VAR(r2, 10 - r2, k)
				};
				*head = insert_clause_head(*head,
new_clause(lits, 2)); // Each number can appear only once on diagonal
			}
	}
}

void add_window_constraints(Clause **head)
{
	DEBUG_PRINT(1, "[DEBUG] Adding window constraints\n");
	struct { int r0, c0; } win[2] = { {2,2}, {6,6} };

	for (int w = 0; w < 2; ++w) {
```

```
            int r0 = win[w].r0, c0 = win[w].c0;


            for (int k = 1; k <= 9; ++k) {
                  int exist[9], idx = 0;
                  for (int dr = 0; dr < 3; ++dr)
                        for (int dc = 0; dc < 3; ++dc)
                              exist[idx++] = VAR(r0 + dr, c0 + dc, k);
                  *head = insert_clause_head(*head, new_clause(exist,
9)); // Each window must contain each number


                  for (int dr1 = 0; dr1 < 3; ++dr1)
                        for (int dc1 = 0; dc1 < 3; ++dc1)
                              for (int dr2 = dr1; dr2 < 3; ++dr2)
                                    for (int dc2 = (dr1==dr2?dc1+1:0);
dc2 < 3; ++dc2) {
                                          int lits[2] = {
                                                -VAR(r0 + dr1, c0 + dc1,
k),
                                                -VAR(r0 + dr2, c0 + dc2,
k)
                                          };
                                          *head =
insert_clause_head(*head, new_clause(lits, 2)); // Each number can
appear only once per window
                                    }
            }
      }
```

```
}


Clause *read_sdk_to_clauses(const char *filename)
{
	DEBUG_PRINT(1, "[DEBUG] Reading Sudoku from file: %s\n",
filename);
	FILE *fp = fopen(filename, "r");
	if (!fp) { perror("open sdk"); exit(EXIT_FAILURE); }

	Clause *head = NULL;
	char line[82] = { 0 };
	size_t len = fread(line, 1, 81, fp);
	if (len != 81) {
		fprintf(stderr, "sdk format error: not 81 chars\n");
		exit(EXIT_FAILURE);
	}

	for (int r = 1; r <= 9; ++r) {
	for (int c = 1; c <= 9; ++c) {
		int pos = (r - 1) * 9 + (c - 1);
		char ch = line[pos];
		if (ch >= '1' && ch <= '9') {
			int val = ch - '0';
			int lit = VAR(r, c, val);
			head = insert_clause_head(head, new_clause(&lit, 1));
			}
```

```
            }
        }
        fclose(fp);
        return head;
}

Clause *read_cnf(const char *fname, int *var_cnt, int *clause_cnt)
{
        DEBUG_PRINT(1, "[DEBUG] Reading CNF file: %s\n", fname);
        FILE *fp = fopen(fname, "r");
        if (!fp) { perror("read_cnf"); exit(EXIT_FAILURE); }

        Clause *head = NULL;
        char line[256];

        while (fgets(line, sizeof(line), fp)) {
            if (line[0] == 'c' || line[0] == '\n') continue;

            if (line[0] == 'p') {
                sscanf(line, "p cnf %d %d", var_cnt, clause_cnt);
                DEBUG_PRINT(1, "[DEBUG] CNF file info: variables=%d,
clauses=%d\n", *var_cnt, *clause_cnt);
                continue;
            }

            int buf[1024], idx = 0;
```

```
            char *tok = strtok(line, " \t\n");
            while (tok) {
                int lit = atoi(tok);
                if (lit == 0) break;
                buf[idx++] = lit;
                tok = strtok(NULL, " \t\n");
            }
            if (idx) head = insert_clause_head(head, new_clause(buf,
idx));
        }
        fclose(fp);
        return head;
}

void build_occurrence()
{
        DEBUG_PRINT(1, "[DEBUG] Building occurrence lists\n");
        for (int i = 1; i <= MAXVAR; ++i) {
            vars[i].pos = NULL;
            vars[i].neg = NULL;
        }

        int clause_count = 0;
        int literal_count = 0;

        Clause *c = clauses;
```

```
	while (c != NULL) {
		clause_count++;
		DEBUG_PRINT(3, "[DEBUG] Processing clause ID=%d\n", c->id);

		int *p = c->lits;
		while (*p != 0) {
			literal_count++;
			int v = abs(*p);
			if (v < 1 || v > MAXVAR) {
				DEBUG_PRINT(1, "[WARNING] Invalid variable
number: %d\n", v);
				p++;
				continue;
			}

			OccurrenceNode *new_node =
(OccurrenceNode*)malloc(sizeof(OccurrenceNode));
			new_node->clause = c;
			new_node->next = NULL;

			if (*p > 0) {
				new_node->next = vars[v].pos;
				vars[v].pos = new_node;
				DEBUG_PRINT(3, "[DEBUG] Variable %d appears in
positive literal clause ID=%d\n", v, c->id);
			} else {
```

```
                        new_node->next = vars[v].neg;
                        vars[v].neg = new_node;
                        DEBUG_PRINT(3, "[DEBUG] Variable %d appears in
negative literal clause ID=%d\n", v, c->id);
                    }

                    p++;
                }
                c = c->next;
            }

            DEBUG_PRINT(1, "[DEBUG] Occurrence list building complete,
processed %d clauses, %d literals\n", clause_count, literal_count);
}

void print_conflict_clause(Clause *c) {
            DEBUG_PRINT(1, "[CONFLICT] Conflict clause ID=%d: ", c->id);
            for (int *p = c->lits; *p; ++p) {
                int lit = *p;
                int var = abs(lit);
                DEBUG_PRINT(1, "%d(", lit);
                if (vars[var].decided) {
                    DEBUG_PRINT(1, "%d", vars[var].value);
                } else {
                    DEBUG_PRINT(1, "undecided");
                }
```

```
            DEBUG_PRINT(1, ") ");
        }
        DEBUG_PRINT(1, "0\n");
}

bool unit_propagate(void) {
        static char inQ[UNIT_QUEUE_MAX];
        static int queue[UNIT_QUEUE_MAX];
        memset(inQ, 0, sizeof(inQ));
        int ql = 0;
        DEBUG_PRINT(2, "[DEBUG] Starting unit propagation, recalculating
clause states\n");

        for (Clause *c = clauses; c; c = c->next) {
            if (c->sat) continue;

            // Check if clause is already satisfied
            bool satisfied = false;
            for (int *p = c->lits; *p; ++p) {
                int u = *p;
                int u_var = abs(u);
                if (vars[u_var].decided && vars[u_var].value == (u >
0 ? 1 : -1)) {
                    satisfied = true;
                    break;
                }
```

```
            }
            if (satisfied) {
                  c->sat = true;
                  DEBUG_PRINT(3, "[DEBUG] Clause ID=%d is satisfied\n",
c->id);
                  continue;
            }

            // Count remaining undecided literals
            c->remaining = 0;
            for (int *p = c->lits; *p; ++p) {
                  int u = *p;
                  if (!vars[abs(u)].decided) {
                        c->remaining++;
                  }
            }

            DEBUG_PRINT(3, "[DEBUG] Clause ID=%d remaining undecided
literals: %d\n", c->id, c->remaining);

            if (c->remaining == 0) {
                  DEBUG_PRINT(1, "[CONFLICT] Clause ID=%d has no
remaining undecided literals and is unsatisfied\n", c->id);
                  print_conflict_clause(c);
                  return false; // Conflict: clause is unsatisfied
            }
```

```
            if (c->remaining == 1) {
                  // Find undecided literal and add to queue
                  for (int *p = c->lits; *p; ++p) {
                        int u = *p;
                        if (!vars[abs(u)].decided) {
                              if (!inQ[u + MAXVAR]) {
                                    inQ[u + MAXVAR] = 1;
                                    queue[ql++] = u;
                                    DEBUG_PRINT(2, "[DEBUG] Added unit
literal to queue: %d (from clause ID=%d)\n", u, c->id);
                              }
                              break;
                        }
                  }
            }
      }

      // Process queue of unit literals
      while (ql > 0) {
            int lit = queue[--ql];
            DEBUG_PRINT(2, "[DEBUG] Processing unit literal: %d\n",
lit);
            inQ[lit + MAXVAR] = 0;

            int v = abs(lit);
```

```
            int val = lit > 0 ? 1 : -1;


            if (vars[v].decided) {
                  if (vars[v].value != val) {
                        DEBUG_PRINT(1, "[CONFLICT] Unit literal %d
conflicts with existing assignment\n", lit);
                        return false;
                  }
                  continue;
            }


            // Assign value
            vars[v].value = val;
            vars[v].decided = true;
            assignment[v] = val;
            trail[trail_top++] = v;
            DEBUG_PRINT(2, "[DEBUG] Unit propagation assignment:
variable %d = %d\n", v, val);


            // Process positive occurrence list
            OccurrenceNode *list_pos = vars[v].pos;
            for (OccurrenceNode *node = list_pos; node; node =
node->next){
                  Clause *c = node->clause;
                  if (c->sat) continue;
                  if (val == 1) {
```

```
                        c->sat = true;
                        DEBUG_PRINT(3, "[DEBUG] Clause ID=%d satisfied
because variable %d=1\n", c->id, v);
                        continue;
                    }
                    c->remaining--;
                    DEBUG_PRINT(3, "[DEBUG] Clause ID=%d remaining
undecided literals reduced to: %d\n", c->id, c->remaining);

                    if (c->remaining == 0) {
                        // Check if clause is really unsatisfied
                        bool satisfied = false;
                        for (int *p = c->lits; *p; ++p) {
                            int u = *p;
                            int u_var = abs(u);
                            if (vars[u_var].decided &&
vars[u_var].value == (u > 0 ? 1 : -1)) {
                                satisfied = true;
                                break;
                            }
                        }
                        if (!satisfied) {
                            DEBUG_PRINT(1, "[CONFLICT] Clause ID=%d
has no remaining undecided literals and is unsatisfied\n", c->id);
                            print_conflict_clause(c);
                            return false; // Conflict
                        } else {
```

```
                        c->sat = true; // Mark as satisfied
                        DEBUG_PRINT(3, "[DEBUG] Clause ID=%d is
satisfied\n", c->id);
                    }
                }

                if (c->remaining == 1) {
                    // Find undecided literal and add to queue
                    for (int *p = c->lits; *p; ++p) {
                        int u = *p;
                        if (!vars[abs(u)].decided) {
                            if (!inQ[u + MAXVAR]) {
                                inQ[u + MAXVAR] = 1;
                                queue[ql++] = u;
                                DEBUG_PRINT(2, "[DEBUG]
Added new unit literal to queue: %d (from clause ID=%d)\n", u, c->id);
                            }
                            break;
                        }
                    }
                }
            }

            // Process negative occurrence list
            OccurrenceNode *list_neg = vars[v].neg;
            for (OccurrenceNode *node = list_neg; node; node = node->next)
```

```
{
                    Clause *c = node->clause;
                    if (c->sat) continue;
                    if (val == -1) {
                          // Clause is satisfied because v is false
                          c->sat = true;
                          DEBUG_PRINT(3, "[DEBUG] Clause ID=%d satisfied
because variable %d=-1\n", c->id, v);
                          continue;
                    }
                    // v is true, reduce remaining count
                    c->remaining--;
                    DEBUG_PRINT(3, "[DEBUG] Clause ID=%d remaining
undecided literals reduced to: %d\n", c->id, c->remaining);

                    if (c->remaining == 0) {
                          // Check if clause is really unsatisfied
                          bool satisfied = false;
                          for (int *p = c->lits; *p; ++p) {
                                int u = *p;
                                int u_var = abs(u);
                                if (vars[u_var].decided &&
vars[u_var].value == (u > 0 ? 1 : -1)) {
                                      satisfied = true;
                                      break;
                                }
```

```
                        }
                        if (!satisfied) {
                              DEBUG_PRINT(1, "[CONFLICT] Clause ID=%d
has no remaining undecided literals and is unsatisfied\n", c->id);
                              print_conflict_clause(c);
                              return false; // Conflict
                        } else {
                              c->sat = true; // Mark as satisfied
                              DEBUG_PRINT(3, "[DEBUG] Clause ID=%d is
satisfied\n", c->id);
                        }
                  }

                  if (c->remaining == 1) {
                        // Find undecided literal and add to queue
                        for (int *p = c->lits; *p; ++p) {
                              int u = *p;
                              if (!vars[abs(u)].decided) {
                                    if (!inQ[u + MAXVAR]) {
                                          inQ[u + MAXVAR] = 1;
                                          queue[ql++] = u;
                                          DEBUG_PRINT(2, "[DEBUG]
Added new unit literal to queue: %d (from clause ID=%d)\n", u, c->id);
                                    }
                                    break;
                              }
```

```
                        }
                    }
                }
            }

    DEBUG_PRINT(2, "[DEBUG] Unit propagation completed\n");
    return true;
}

double get_ms(void)
{
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec * 1000.0 + ts.tv_nsec / 1e6;
}

int pick_var_moms(void) {
    DEBUG_PRINT(2, "[DEBUG] Using MOMS heuristic to select
variable\n");
    int maxcnt = -1, bestv = 0;
    for (int v = 1; v <= var_cnt; ++v) {
        if (assignment[v]) continue; // Skip assigned variables
        int cnt = 0;
        // 遍历正文字出现列表
        for (OccurrenceNode *node = vars[v].pos; node; node =
node->next) {
```

```
                Clause *c = node->clause;
                if (c->remaining == 2) cnt++;
            }
            // 遍历负文字出现列表
            for (OccurrenceNode *node = vars[v].neg; node; node =
node->next) {
                Clause *c = node->clause;
                if (c->remaining == 2) cnt++;
            }
            if (cnt > maxcnt) {
                maxcnt = cnt;
                bestv = v;
            }
        }
        if (bestv != 0) {
            DEBUG_PRINT(2, "[DEBUG] MOMS selected variable: %d
(count: %d\n", bestv, maxcnt);
            return bestv;
        }
        // Fallback: select first unassigned variable
        DEBUG_PRINT(2, "[DEBUG] MOMS found no variable, using first
unassigned variable\n");
        for (int v = 1; v <= var_cnt; ++v) {
            if (assignment[v] == 0) return v;
        }
        return 0; // All variables assigned
```

```
}


int pick_var(void)
{
	DEBUG_PRINT(2, "[DEBUG] Selecting first unassigned variable\n");
	for (int v = 1; v <= var_cnt; ++v)
		if (assignment[v] == 0) return v;
	return 0;
}


bool dpll(void)
{
	static Frame stack[MAXVAR];
	int sp = 0;
	int decision_level = 0;

	DEBUG_PRINT(1, "[DEBUG] Starting DPLL algorithm\n");

	while(1)
	{
	continue_loop:
		DEBUG_PRINT(2, "[DEBUG] Calling unit propagation at
level %d\n", decision_level);
		if (!unit_propagate()) {
			DEBUG_PRINT(1, "[CONFLICT] Conflict detected at
level %d\n", decision_level);
```

```
                goto backtrack;
            }

            int v = pick_var_moms();
            if (v == 0) {
                --decision_level;
                DEBUG_PRINT(1, "[DEBUG] All variables assigned,
returning true\n");
                return true;
            }

            DEBUG_PRINT(1, "[DECISION] Choosing variable %d at level %d,
assigning true first\n", v, decision_level);
            stack[sp].v         = v;
            stack[sp].branch = 1;
            stack[sp].level    = decision_level;
            ++sp;
            ++decision_level;
            save_assign(v, 1);
            continue;

      backtrack:
            DEBUG_PRINT(1, "[BACKTRACK] Backtracking from level %d\n",
decision_level);
            while (sp > 0)
            {
```

```
                --sp;
                int v = stack[sp].v, branch = stack[sp].branch;
                if (branch == 1)
                {
                        stack[sp].branch = -1;
                        int lvl = stack[sp].level;
                        backtrack(lvl);
                        decision_level = lvl;
                        DEBUG_PRINT(1, "[DECISION] Trying variable %d
to false at level %d\n", v, decision_level);
                        save_assign(v, -1);
                        ++decision_level;
                        if (unit_propagate()) {
                                DEBUG_PRINT(1, "[DEBUG] Unit propagation
after backtrack successful, continuing\n");
                                goto continue_loop;
                        }
                        else {
                                DEBUG_PRINT(1, "[DEBUG] Unit propagation
after backtrack failed, continuing backtrack\n");
                        }
                }
                else
                {
                        DEBUG_PRINT(1, "[BACKTRACK] Both branches tried
for variable %d,backtracking further\n", v);
                        backtrack(stack[sp].level);
```

```
                        decision_level = stack[sp].level;
                    }
                }
                if (sp == 0) {
                    decision_level = 0;
                    DEBUG_PRINT(1, "[DEBUG] Stack empty, returning
false\n");
                    return false;
                }
            }
        }
        void fill_box(char grid[9][9], int row, int col) {
            int nums[9] = {1, 2, 3, 4, 5, 6, 7, 8, 9};

            // 随机打乱数字
            for (int i = 0; i < 9; i++) {
                int j = rand() % 9;
                int temp = nums[i];
                nums[i] = nums[j];
                nums[j] = temp;
            }

            // 填充宫格
            int index = 0;
            for (int i = 0; i < 3; i++) {
                for (int j = 0; j < 3; j++) {
```

```
                grid[row + i][col + j] = nums[index++];
            }
        }
    }

// 生成百分号数独游戏
void generate_percent_sudoku(char grid[9][9], int difficulty) {
    // 初始化网格为全 0
    memset(grid, 0, sizeof(char) * 81);

    // 首先创建一个完整的合法数独
    // 这是一个简化版本，实际应用中可能需要更复杂的算法

    // 填充对角线上的 3x3 宫格
    for (int i = 0; i < 9; i += 3) {
        fill_box(grid, i, i);
    }

    // 尝试填充剩余部分
    solve_sudoku(grid); // 使用 SAT 求解器填充完整

    // 根据难度挖空
    int holes = difficulty; // 难度决定挖空数量
    srand(time(NULL));

    while (holes > 0) {
```

```
            int r = rand() % 9;
            int c = rand() % 9;

            if (grid[r][c] != 0) {
                grid[r][c] = 0;
                holes--;
            }
        }
}
Clause* create_sudoku_cnf(char grid[9][9]) {
        Clause* head = NULL;

        // 添加基本约束
        add_cell_constraints(&head);
        add_row_constraints(&head);
        add_col_constraints(&head);
        add_box_constraints(&head);

        // 添加百分号数独的特殊约束
        add_diagonal_constraints(&head);
        add_window_constraints(&head);

        // 添加已知数字的约束
        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                if (grid[i][j] != 0) {
```

```
                        int k = grid[i][j];
                        int lit = VAR(i+1, j+1, k);
                        head = insert_clause_head(head,
new_clause(&lit, 1));
                    }
                }
            }

            return head;
}
// 使用 SAT 求解器解决数独
void solve_sudoku(char grid[9][9]) {
            // 创建数独的 CNF 公式
            Clause* sudoku_clauses = create_sudoku_cnf(grid);

            // 保存原始子句列表
            Clause* original_clauses = clauses;

            // 设置新的子句列表
            clauses = sudoku_clauses;

            // 重置变量状态
            trail_top = 0;
            sp            = 0;
            for (int i = 0; i <= MAXVAR; ++i){
                  vars[i].pos = vars[i].neg = NULL;
```

```
        vars[i].decided = false;
        vars[i].value    = 0;
    }
    build_occurrence();
    for (Clause *c = clauses; c; c = c->next) {
        c->sat = false;
        c->remaining = 0;
        for (int *p = c->lits; *p; ++p) {
            if (!vars[abs(*p)].decided) {
                c->remaining++;
            }
        }
    }
    bool solved = dpll();


    if (solved) {
        // 从赋值中提取数独解
        for (int i = 1; i <= 9; i++) {
            for (int j = 1; j <= 9; j++) {
                for (int k = 1; k <= 9; k++) {
                    int var = VAR(i, j, k);
                    if (vars[var].value > 0) {
                        grid[i-1][j-1] = k;
                        break;
                    }
```

```
                        }
                    }
                }
        } else {
                printf("Failed to solve the Sudoku!\n");
        }

        // 恢复原始子句列表
        clauses = original_clauses;

        // 释放数独子句内存
        Clause* current = sudoku_clauses;
        while (current) {
                Clause* next = current->next;
                free(current->lits);
                free(current);
                current = next;
        }
}

// 检查数独是否已完成
bool is_complete(char grid[9][9]) {
        for (int i = 0; i < 9; i++) {
                for (int j = 0; j < 9; j++) {
                        if (grid[i][j] == 0) {
                                return false;
```

```
                }
            }
        }
        return true;
}

// 检查数独是否有效
bool is_valid(char grid[9][9]) {
        // 检查行
        for (int i = 0; i < 9; i++) {
            bool seen[10] = {false};
            for (int j = 0; j < 9; j++) {
                int num = grid[i][j];
                if (num != 0 && seen[num]) {
                    return false;
                }
                seen[num] = true;
            }
        }

        // 检查列
        for (int j = 0; j < 9; j++) {
            bool seen[10] = {false};
            for (int i = 0; i < 9; i++) {
                int num = grid[i][j];
                if (num != 0 && seen[num]) {
```

```
                return false;
            }
            seen[num] = true;
        }
    }

    // 检查宫格
    for (int box = 0; box < 9; box++) {
        bool seen[10] = {false};
        int startRow = (box / 3) * 3;
        int startCol = (box % 3) * 3;

        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                int num = grid[startRow + i][startCol + j];
                if (num != 0 && seen[num]) {
                    return false;
                }
                seen[num] = true;
            }
        }
    }

    // 检查对角线
    bool seen1[10] = {false};
    bool seen2[10] = {false};
```

```
        for (int i = 0; i < 9; i++) {
            int num1 = grid[i][i];
            int num2 = grid[i][8 - i];

            if (num1 != 0 && seen1[num1]) {
                return false;
            }
            if (num2 != 0 && seen2[num2]) {
                return false;
            }

            seen1[num1] = true;
            seen2[num2] = true;
        }

        // 检查窗口
        int windows[2][2] = {{1, 1}, {5, 5}}; // 窗口的起始位置

        for (int w = 0; w < 2; w++) {
            bool seen[10] = {false};
            for (int i = 0; i < 3; i++) {
                for (int j = 0; j < 3; j++) {
                    int num = grid[windows[w][0] + i][windows[w][1]
+ j];
                    if (num != 0 && seen[num]) {
                        return false;
```

```
                        }
                        seen[num] = true;
                    }
                }
            }

            return true;
}

// 打印数独网格
void print_sudoku(char grid[9][9]) {
            printf("+-------+-------+-------+\n");
            for (int i = 0; i < 9; i++) {
                printf("| ");
                for (int j = 0; j < 9; j++) {
                    if (grid[i][j] == 0) {
                        printf(". ");
                    } else {
                        printf("%d ", grid[i][j]);
                    }

                    if ((j + 1) % 3 == 0) {
                        printf("| ");
                    }
                }
                printf("\n");
```

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

// 玩数独游戏
void play_sudoku() {
    char grid[9][9];

    // 生成数独
    generate_percent_sudoku(grid, 30); // 30 个空格

    printf("Welcome to Percent Sudoku Game!\n");
    printf("Rules:\n");
    printf("1. Each row, column, and 3x3 box must contain numbers 1-9\n");
    printf("2. The two diagonals must also contain numbers 1-9\n");
    printf("3. The two window areas (top-left and bottom-right) must also contain numbers 1-9\n");
    printf("Input format: row column value (e.g., 1 2 3 means row 1, column 2, value 3)\n");
    printf("Enter 0 0 0 to solve the Sudoku\n\n");

    while (1) {
```

```
            print_sudoku(grid);

            int r, c, v;
            printf("Please enter row, column and value: ");
            scanf("%d %d %d", &r, &c, &v);

            if (r == 0 && c == 0 && v == 0) {
                  printf("Solving...\n");
                  solve_sudoku(grid);
                  print_sudoku(grid);
                  printf("Game over!\n");
                  break;
            }

            if (r < 1 || r > 9 || c < 1 || c > 9 || v < 1 || v > 9) {
                  printf("Invalid input! Row and column must be between
1-9, value must be between 1-9.\n");
                  continue;
            }

            grid[r-1][c-1] = v;

            // 检查是否完成
            if (is_complete(grid)) {
                  if (is_valid(grid)) {
                        print_sudoku(grid);
```

```
                        printf("Congratulations! You solved the
Sudoku!\n");
                        break;
                } else {
                        printf("Sorry, the solution is incorrect.
Please try again.\n");
                }
            }
        }
}

int main(int argc, char *argv[])
{
        if (argc == 2 && strcmp(argv[1], "-play") == 0) {
            play_sudoku();
            return 0;
        }
        DEBUG_PRINT(0, "=== Program started ===\n");
        trail_top = 0;
        sp            = 0;
        DEBUG_PRINT(0, "\n=== Starting tiny test ===\n");
        for (int i = 0; i <= MAXVAR; ++i){
            vars[i].pos = vars[i].neg = NULL;
            vars[i].decided = false;
            vars[i].value    = 0;
        }
```

```
clauses = read_cnf(argv[1], &var_cnt, &clause_cnt);
build_occurrence();
for (Clause *c = clauses; c; c = c->next) {
      c->sat = false;
      c->remaining = 0;
      for (int *p = c->lits; *p; ++p) {
            if (!vars[abs(*p)].decided) {
                  c->remaining++;
            }
      }
}
double start_time = get_ms();
bool ok = dpll();
double end_time = get_ms();
long long elapsed_time = (long long)(end_time - start_time);
char output_file[256];
strcpy(output_file, argv[2]);
FILE *fp1 = fopen(argv[2], "w");
if (ok) {
      fprintf(fp1, "s 1\n");
      // 输出变元赋值
      fprintf(fp1, "v ");
      for (int i = 1; i <= var_cnt; i++) {
            fprintf(fp1, "%s%d ", vars[i].value > 0 ? "" : "-", i);
      }
      fprintf(fp1, "\n");
```

```
} else {
        fprintf(fp1, "s 0\n");
}

// 输出执行时间
fprintf(fp1, "t %lldms\n", elapsed_time);

fclose(fp1);
DEBUG_PRINT(0, "=== Program ended ===\n");
Clause *current = clauses;
while (current) {
        Clause *next = current->next;
        free(current->lits);
        free(current);
        current = next;
}

// 释放出现列表
for (int i = 1; i <= MAXVAR; i++) {
        OccurrenceNode *node = vars[i].pos;
        while (node) {
                OccurrenceNode *next = node->next;
                free(node);
                node = next;
        }
```

```
            node = vars[i].neg;
            while (node) {
                OccurrenceNode *next = node->next;
                free(node);
                node = next;
            }
        }
        return 0;
}
```