華中科技大学

课 程 实 验 报 告

课程名称：

基于 SAT 的对角线数独游戏求解程序

专业班级 CS2306

学　　号 U202315594

姓　　名 贾柠泽

指导教师 袁凌

报告日期 2024 年 9 月 19 日

计算机科学与技术学院

目　　录

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。

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]。

3.DPLL 过程

基于 DPLL 算法框架，实现 SAT 算例的求解。

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. 程序优化

对基本 DPLL 的实现进行存储结构、分支变元选取策略 [1-3] 等某一方面进行优化设计与实现，提供较明确的性能优化率结果。优化率的计算公式为：[(t-to)/t]*100%, 其中 t 为未对 DPLL 优化时求解基准算例的执行时间，to 则为优化 DPLL 实现时求解同一算例的执行时间。

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]。

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.grandgame

[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$_3$485.*pd*

[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

1 引言

1.1 课题背景

数独是一项极具挑战性的智力游戏，要求玩家在 9x9 的格子中填入 1 至 9 的数字，满足每一行、每一列和每一个 3x3 的小格子内数字不重复的条件。这一游戏因其逻辑性和趣味性而广受欢迎。

数独问题在本质上是一个逻辑推理与搜索问题，适合用计算机程序来解决。计算机在处理复杂逻辑和大量数据方面具有明显优势。

SAT 是一种用于处理符合性和可满足性的算法。它能够解决一系列逻辑公式的问题，通过寻找一组变量赋值，使逻辑公式中的所有命题成立。数独问题可以转化为逻辑命题，进而利用 SAT 算法求解。

研究拓展，将数独问题与 SAT 算法结合，不仅可以解决数独游戏本身，还可以拓展到其他类似问题的求解，如逻辑推理、工艺流程优化、加密算法等。

1.2 课题意义

使用 SAT 求解数独的意义主要体现在以下几个方面：

1.2.1 解决复杂问题的能力

SAT 问题通常涉及到复杂的逻辑推理，使用 SAT 求解数独可以锻炼解决复杂问题的能力，并且能够培养逻辑思维。

1.2.2 算法与应用的结合

SAT（高斯-塞德尔交替法）是一种算法，学习如何将它应用于数独这类问题上，有助于理解算法如何解决现实世界的问题。

1.2.3 人工智能领域的研究

在人工智能领域中，数独是一个经典的搜索和约束满足问题，使用 SAT 求解可以提供对算法如何解决这类问题的洞察，对于人工智能领域的研究和发展具有重要意义。

1.3 国内外研究现状

DPLL（解析证明逻辑）的研究现状主要集中在以下几个方面：

1.3.1 算法优化

研究者们不断探索如何优化 DPLL 算法，以使其在解决特定问题（如 SAT 求解）时更加高效。这包括改进算法的时间和空间复杂度，例如通过剪枝技术减少搜索空间。

1.3.2 实践应用

DPLL 算法及其变种被广泛应用于各种实际问题中，如自动测试、软件验证和硬件设计验证。

1.3.3 与人工智能结合

DPLL 算法也被融合到更广泛的人工智能技术中，例如在游戏 AI、规划问题和数据处理中。

1.3.4 理论研究

理论研究方面，研究者们关注 DPLL 算法的理论极限以及如何从理论上证明其性能。

1.3.5 跨领域研究

DPLL 的研究还与组合优化、逻辑设计、计算复杂性等多个领域交叉。

1.4 课程设计的主要研究工作

DPLL 课程设计的主要研究工作通常包括以下几个方面：

1.4.1 算法实现

设计和实现 DPLL（Davis-Putnam-Logemann-Loveland）算法，一种用于解决布尔满意性问题（SAT）的算法。这涉及理解算法的原理，并使用编程语言（如 Python、C++ 等）将算法编码实现。

1.4.2 性能分析

通过实验测试 DPLL 算法在不同类型和规模的 SAT 问题上的性能，分析其时间和空间复杂度。

1.4.3 算法优化

针对 DPLL 算法进行优化，比如通过约束传播、启发式选择、变量.badging 等方法提高算法的效率。

1.4.4 案例研究

选择具体的 SAT 实例，应用 DPLL 算法进行求解，并分析求解过程中的关键步骤和结果。

1.4.5 算法的理论分析

研究和讨论 DPLL 算法的理论基础，包括它解决 SAT 问题的局限性和适用范围。

2 系统需求分析与总体设计

2.1 系统需求分析

基于 SAT 的对角线数独游戏求解程序系统要求有对 CNF 文件的读取操作以及解读操作，并且将解析到的子句以及文字存入到十字链表当中。十字链表的好处是便于删除操作，而利用 DPLL 算法解 CNF 文件的关键就是要不停地删除文字，最终得到一个空子句集，即得到了解集。同时，系统要求程序能够根据现有的数独棋盘来生成用解 CNF 的算法来解决数独游戏，最终解决出答案。

2.2 系统总体设计

数据结构类型为十字链表结构，SATList 型结构体保存一个子句，指向 SATNode 的头结点以及下一个 SATList 型结构体；SATNode 型结构体保存一个文字，指向下一个文字。在 DPLL 算法中，采用二叉搜索树、分支结构的思想来求解 CNF 型数据并且进行优化。在求解数独时，将数独的初始条件翻译成 CNF 句式，进一步套用 DPLL 来求解值。

3 系统详细设计

3.1 数据结构定义

```
//定义子句
typedef struct SATNode {
    int data;   //数据域
    SATNode *next;
} SATNode;

//定义文字
typedef struct SATList {
    SATNode *head;   //表头
    SATList *next;
} SATList;
```

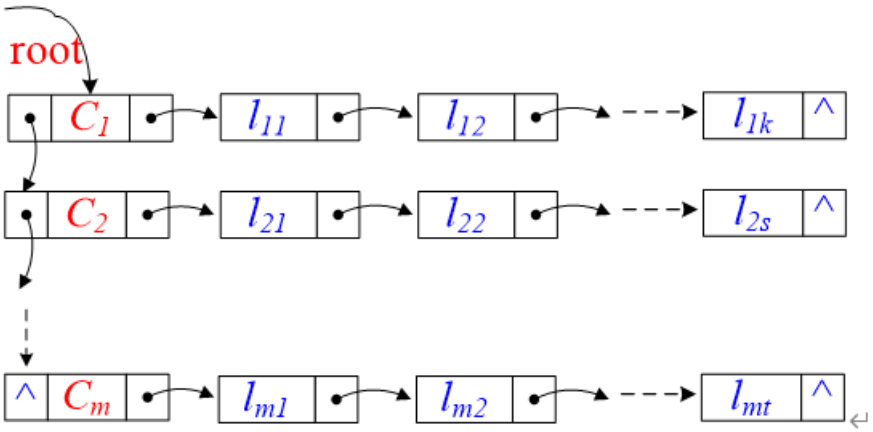

图 3-1 十字链表结构示意图

3.2 演示系统

如图3-1，演示系统包括用户操作界面和功能调用部分。

演示系统界面和所有操作和提示语言为中文。

用户操作界面输出可选的功能操作，用户输入数字选择要进行的操作。在用户选择操作后，功能调用部分会显示函数的名称，参数，返回值和作用，系统提示用户输入参数。

功能调用部分将用户输入的有关信息传递给线性数据结构的操作函数进行调用，并对函数的返回值进行处理判断输出相应的提示信息。

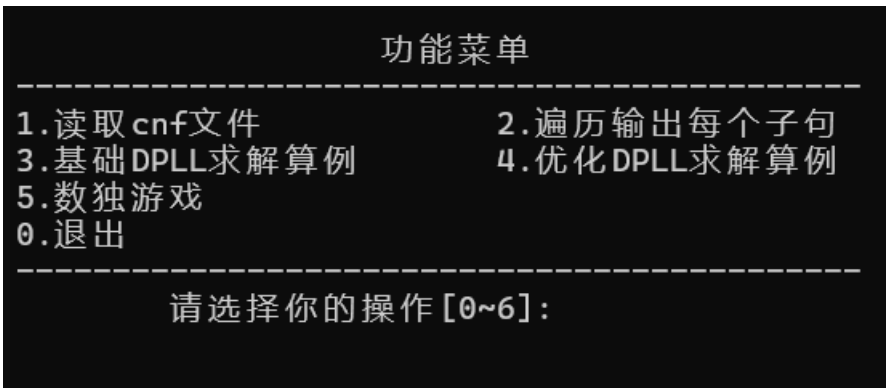

图 3-2　十字链表结构示意图

3.3　cnf 文件读取和写入操作

int ReadFile(SATList *&cnf);

函数名称: ReadFile

接受参数: SATList*&

函数功能: 用文件指针 fp 打开用户指定的文件，依次读取文件中的数据，当第一行是 c 开头时，就跳过；是 p 开头，就读入变元数 boolCount 和子句数 clauseCount；否则，就开始读入文字，并且生成新的 SATNode 结点。当读到 0 时候就结束该文字，并建立新的 SATList 结点，直到读入 EOF 为止。

返回值: int

---

bool WriteFile(int result, double time, int value[]);

函数名称: WriteFile

接受参数: int,double,int[]

函数功能: 将运行结果 result 保存至同名文件，文件拓展名为.res，如果 result=1，则接下来存入 value 中的数据，value[i]=1 则存入正值，=0 则存入负值；如果 result=0 则不存 value 中的数据，代表该 cnf 无解，最后将运行时间 time 写入文件，用到了 fprintf 函数。

返回值: bool

3.4 DPLL 算法的实现

bool addClause(SATList *cnf, SATList *&root);

函数名称: addClause

接受参数: SATList*,SATList*&

函数功能: 在 root 链表中添加 cnf 子句，并且 root 指向的链表变为 cnf 所在的子句。如果 cnf 是空就返回 false，否则就返回 true。

返回值: bool

---

bool baseDPLL(SATList *&cnf, int *&value, int *&count);

函数名称: baseDPLL

接受参数: SATList*&,int *&,int *&

函数功能: 求解 SAT 句式的基础 DPLL 算法，找到 SAT 句式中出现的第一个单子句，用 e 记录该单子句中的文字，然后调用 delete 函数删除 e 所在的所有子句以及-e 文字，并继续找到下一个单子句。直到 SAT 句式中已经没有单子句，如果此时 cnf 已经为空，则直接返回 true，表明存在一组解使得 SAT 句式成立；如果 cnf 不为空，先暂时存储所有数据，然后找到出现最多的变元，删除其正值，递归调用 baseDPLL 函数，如果 cnf 为空，如上，则返回 true，一直返回到原函；如果 cnf 不为空但是某个子句为空，说明某个子句的所有文字都是假，返回 false 并回到上一层递归，此时需要将 e 的值改为-e 继续上述操作，为 true，得到解；为 false，说明不存在一组解使得 SAT 句式成立，返回 0。

返回值: bool

---

void CopyClause(SATList *&a, SATList *b);

函数名称: CopyClause

接受参数: SATList*,SATList*

函数功能: 将链表 b 的值复制到链表 a 中

返回值: void

---

void compute_literal_num(SATList *cnf, int *count);

函数名称: conpute_literal_num

接受参数: SATList *,int*

函数功能: 计算十字链表中每一个变元的数量，保存在 count 数组中

返回值: void

---

int *copy(int *a, int n);

函数名称: copy

接受参数: int*,int

函数功能: 复制 a 数组

返回值: int *

---

void destroyClause(SATList *&cnf);

函数名称: destroyClause

接受参数: SATList*&

函数功能: 销毁链表

返回值: void

---

bool evaluateClause(SATNode *cnf, int v[]);

函数名称: evaluateClause

接受参数: SATList*,int*

函数功能: 评估子句的真假状态，真返回 true，假返回 false

返回值: bool

---

bool emptyClause(SATList *cnf);

函数名称: emptyClause

接受参数: SATList*

函数功能: 判断是否含有空子句，是返回 1，不是返回 0

返回值: bool

---

void Delete(SATList *&cnf, SATList *&pre, SATList *&List, int *count);

函数名称: Delete

接受参数: SATList *&,SATList *&,SATList *&,int *

函数功能: 删除 cnf 链表中 List 指向的子句，pre 为 List 的前继指针

返回值: void

---

void Delete(SATList *&cnf, int literal, int *count);

函数名称: Delete

接受参数: SATLis *&,int,int*

函数功能: 删除 cnf 链表中含有 literal 文字的子句以及-literal 的文字

返回值: void

---

bool isUnitClause(SATNode *cnf);

函数名称: isUnitClause

接受参数: SATNode*

函数功能: 判断是否为单子句，是返回 true，不是返回 false

返回值: bool

---

void init(int *&arr, int e = 0);

函数名称: init

接受参数: int*&,int

函数功能: 初始化数组

返回值: void

---

int optimizeDPLL(SATList *&cnf, int value[]);

函数名称: optimizeDPLL

接受参数: SATList*&,int *

函数功能: 求解 SAT 问题，首先找到单子句，文字为 e，如果 e 大于 0，则 value[e]=1，否则 value[-e]=0。然后重新从头到尾开始遍历，删除含有 e 的子句以及-e 文字，并且接着再找下一个单子句，直到 SAT 句式中不存在单子句，记录每一个文字的个数，如 e 和-e 代表两个不同的数，先找到正文字中出现最多的文字 e，将单子句 e 添加到 cnf 的头子句，并且将 cnf 指向 e 作为头结点，其余步骤与 baseDPLL 相同。

返回值: int

---

void print(SATList *CNFList);

函数名称: print

接受参数: SATList*

函数功能: 输出 cnf 链表

返回值: void

---

void removeClause(SATList *&cnf, SATList *&root);

函数名称: removeClause

接受参数: SATList*&,SATList*&

函数功能: 在已有的十字链表中删除指定的子句

返回值: void

---

void removeNode(SATNode *&cnf, SATNode *&head);

函数名称: removeNote

接受参数: SATNode*&,SATNode*&

函数功能: 在指定的子句中删除指定的文字

返回值: void

3.5 数独游戏的算法实现

void copy(int a[N][N], int b[N][N]);

函数名称: copy

接受参数: int [][],int [][]

函数功能: 将数组 b 复制给数组 a

返回值: void

---

bool isSafe(int board[N][N], int row, int col, int num);

函数名称: isSafe

接受参数: int[][],int,int,int

函数功能: 判断数独的输入或求解是否满足数独条件，即行、列、宫、对角线上都没有重复的数字。

返回值: bool

---

void init(int *&arr, int e = 0);

函数名称: init

接受参数: int[][]

函数功能: 初始化数独

返回值: void

---

void mySolveSudoku(int a[N][N], int board[N][N]);

函数名称: mySolveSudoku

接受参数: int [][],int [][]

函数功能: 玩家自己求解数独游戏

返回值: void

---

void printBoard(int board[N][N]);

函数名称: printBoard

接受参数: int[][]

函数功能: 输出棋盘

返回值: void

---

void removeNumbers(int board[N][N], int count);

函数名称: init

接受参数: int[][],int

函数功能: 挖洞法将数独挖空，随机挖出 count 个洞。

返回值: void

---

int SolvePuzzle(int chess[]);

函数名称: SolvePuzzle

接受参数: int[]

函数功能: 求解输入棋盘，有解返回 1，无解返回 0

返回值: int

---

bool solveSudoku(int board[N][N]);

函数名称: solveSudoku

接受参数: int[][]

函数功能: 回溯法求解数独

返回值: bool

4 系统实现与测试

4.1 系统演示

程序主控流程：

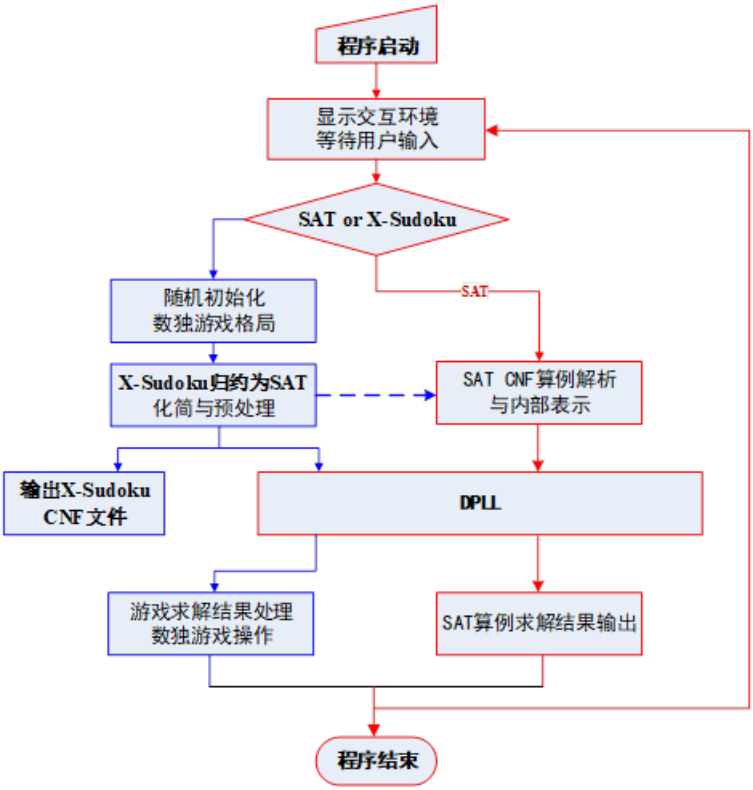

图 4-1 主控程序示意图

界面展示：

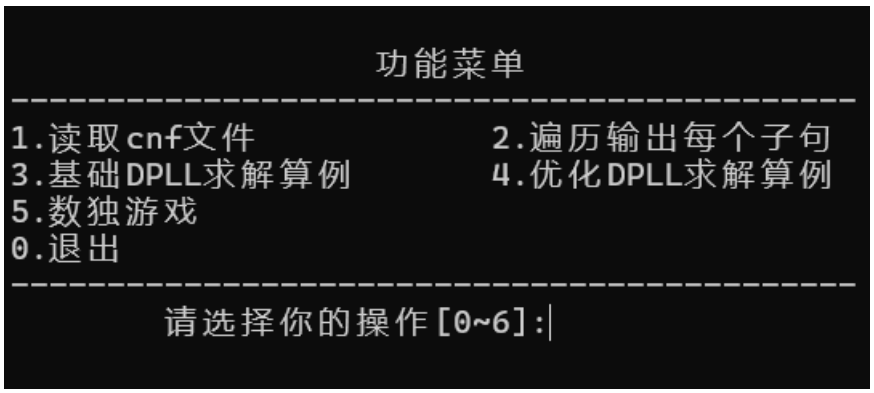

图 4-2 界面展示

4.2 系统测试

4.2.1 cnf 文件读取

读取文件：sat-20.cnf

读取结果：

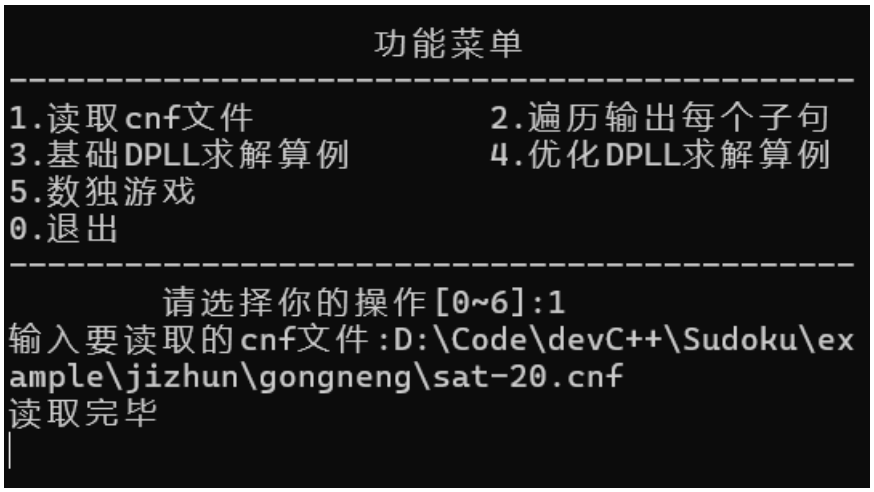

图 4-3 cnf 文件读取

输出遍历 cnf 中的子句：

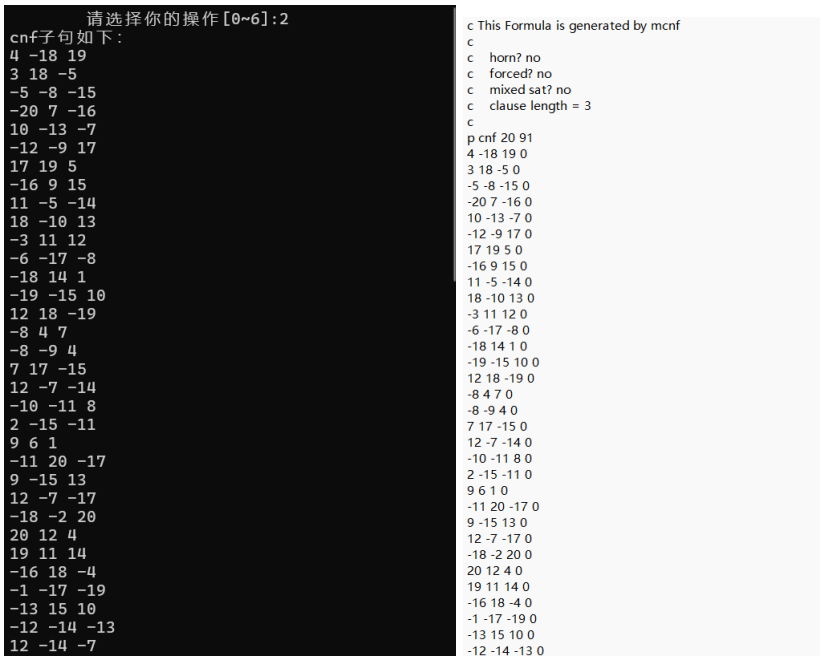

图 4-4 读取文件部分内容对比

4.2.2 DPLL 求解算例

选取了满足算例、不满足算例、基准算例和其他算例中的一些例子，求解信息如表4-1，4-2所示。

表 4-1 求解算例

算例名	变元数	子句与变元数比值	满足与否	DPLL 基础求解时间 t/ms	DPLL 优化求解时间 t_0/ms	优化率/%
problem1-20.cnf	20	4.55	1	0	0	0
problem2-50.cnf	50	1.6	1	15	0	100
problem3-100.cnf	100	3.4	1	751	706	6.0
problem6-50.cnf	50	2	1	79	31	60.8
problem8-50.cnf	50	6	1	15	0	100
problem9-100.cnf	100	2	1	78897	59371	24.7
problem11-100.cnf	100	6	1	126	50	60.3
tst_v25_c100.cnf	25	4	1	0	0	0
7cnf20_90000_90000_7.shuffled-20.cnf	20	76.7	1	119	31	73.9
bart17.shuffled-231.cnf	231	5.05	1	120	18	85.0
tst_v200_c210.cnf	200	1.05	1	7	6	14.2
problem12-200.cnf	200	6	1	101785	101313	0.5

表 4-2　求解算例

算例名	变元数	子句与变元数比值	满足与否	DPLL 基础求解时间 t/ms	DPLL 优化求解时间 t_0/ms	优化率/%
u-problem7-50.cnf	50	2	0	267	143	46.4
sat-20.cnf	20	4.55	1	0	0	0
unsat-5cnf-30.cnf	30	14	0	155	125	19.4
ais10.cnf	30	14	1	60	32	46.7
sud00009.cnf	303	8.51	1	9485	148	98.4
ais6.cnf	61	9.52	1	0	0	0
ais8.cnf	113	13.45	1	49	12	75.5
ais10.cnf	181	17.41	1	65	16	75.4
ais12.cnf	265	21.38	1	3594	268	92.5

以 problem1-20.cnf 文件为例，在求解 DPLL 算例之后，将结果保存在 res 文件中如图4-5。文件的第一行 s 保存 SAT 句式求解的值，1 代表有一组解使得 SAT 句式成立，0 代表不存在；如果 s=1，第二行则保存每一个变元取得的值使得 SAT 句式成立，如果 s=0，则不保存 v；下一行，存储用 DPLL 求解 SAT 问题的执行时间，单位是毫秒。

```
s 1
v -1 2 3 4 -5 -6 -7 8 9 10 11 -12 -13 14 15 -16 17 18 19 20
t 0.000000ms
```

图 4-5　res 文件示例

4.2.3 数独游戏

首先使用 srand 函数，以时间为自变量来生成随机棋盘。然后玩家可进行选择，如图4-6。

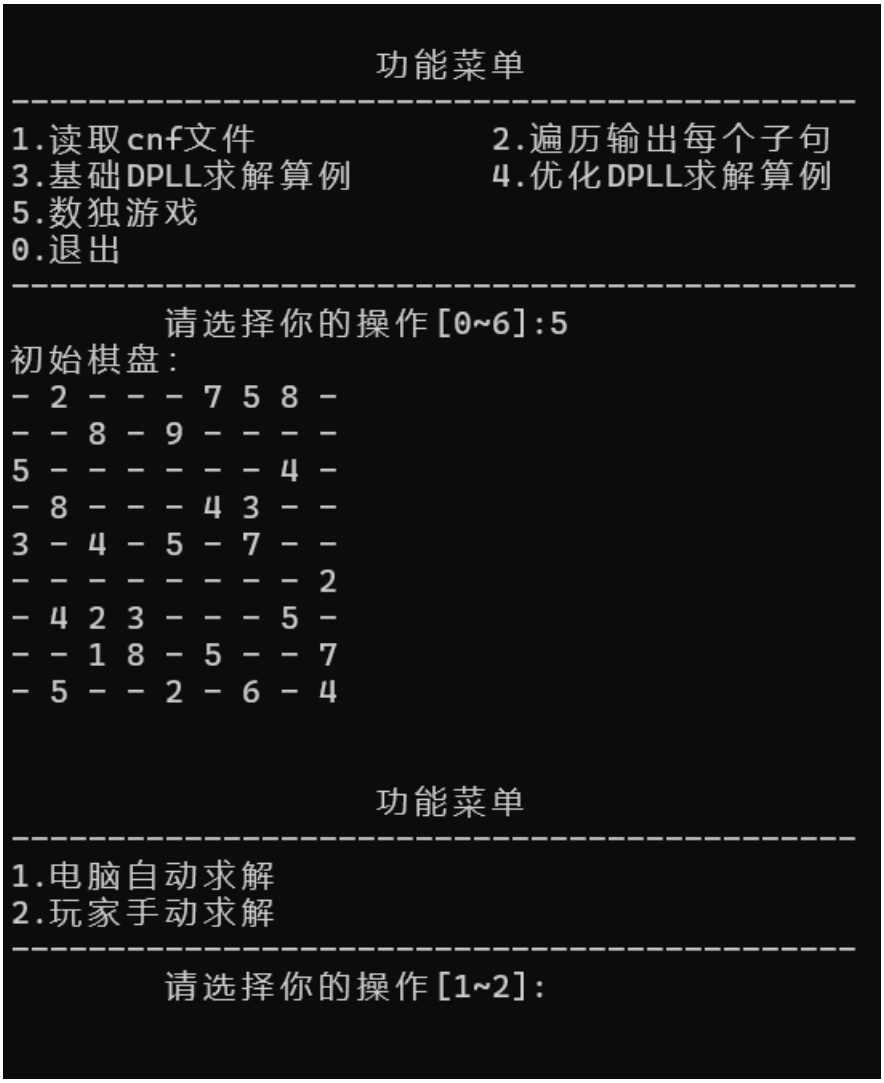

图 4-6 数独游戏菜单

若玩家选择 1，则电脑调用回溯算法直接求解数独得到答案，并输出最终棋盘，如图4-7。

若玩家选择 2，则玩家可以手动输入想要填入输入的数字的行、列、值，如图4-8，此时玩家需要输入三个值，若输入正确，则会在数独中填入该数字，如图4-9，如果输入错误，则会直接显示错误，不会对数独棋盘做任何改动，如图4-10。

若玩家选择 0，则退出数独游戏。

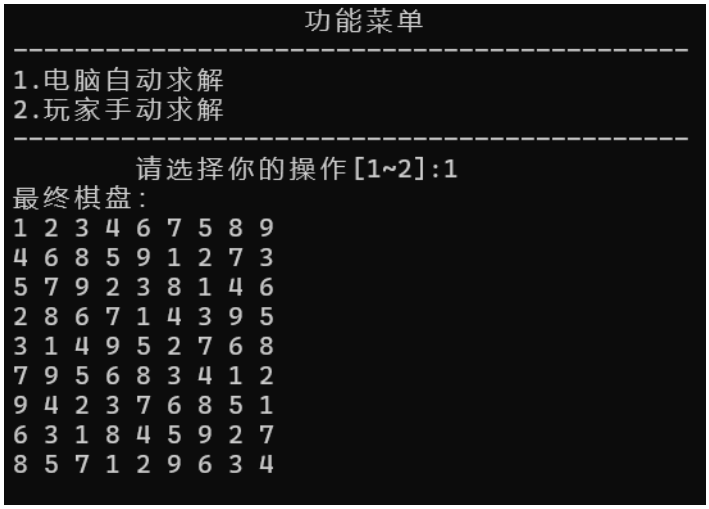

图 4-7　电脑求解

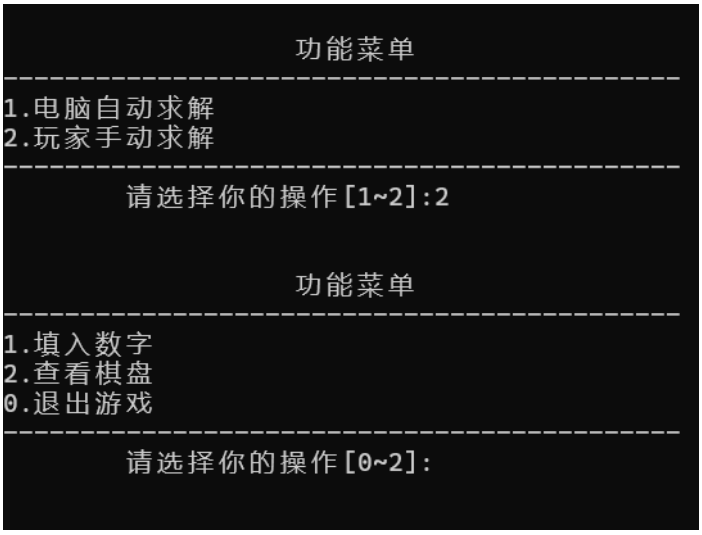

图 4-8　玩家手动求解

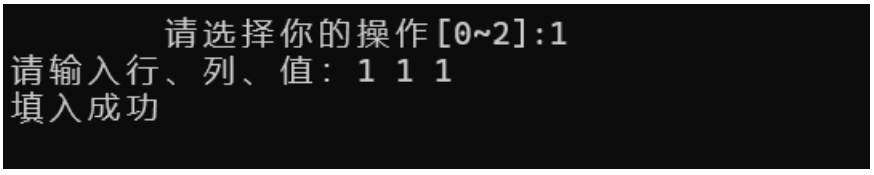

图 4-9　填入数字成功

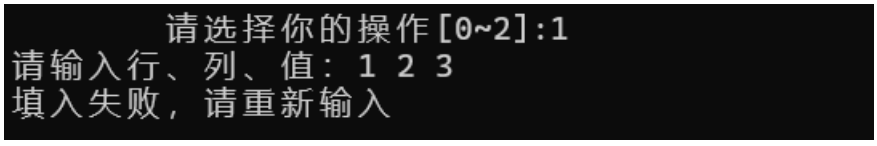

图 4-10　填入数字失败

5 总结与展望

5.1 总结

5.1.1 基本原理

数独是一种 9x9 的网格，其中每个横行、竖列及 3x3 的小方格需填入 1 到 9 的数字，且数字不能重复。通过将数独问题转化为逻辑公式，可以利用 SAT 求解器寻找解。

5.1.2 问题建模

将数独的约束条件用布尔变量表示。例如，可以用 xijk 来表示在第 i 行第 j 列填入数字 k。接着，通过一系列的布尔公式表达数独的约束条件。

5.1.3 约束条件

单元格约束：每个单元格只能包含一个数字。

行、列、块约束：每个数字在每一行、每一列和每个 3x3 块中只能出现一次。

5.1.4 求解过程

将构建的逻辑公式输入 SAT 求解器，求解器通过特定算法（如 DPLL 算法或 CDCL 算法）进行处理，以判定其可满足性并返回解。

5.1.5 效率

SAT 求解器在处理大规模问题时表现出色，尤其在数独这样结构明确且约束条件较强的问题上，解决效率高。

5.2 展望

5.2.1 算法改进

未来的研究可以集中在更高效的数独建模和 SAT 求解算法上，特别是为了处理更大规模和更复杂的变种数独。

5.2.2 多样化应用

除了传统的数独，SAT 技术还可以扩展到其他逻辑谜题的求解，例如变种逻辑游戏、组合优化问题等。

5.2.3 结合深度学习

探索将深度学习与 SAT 求解相结合的新方法，以提高求解效率，尤其是在生成和解析数独模版方面。

5.2.4 用户友好的工具与平台

开发更为用户友好的数独求解器和在线平台，允许广泛的用户体验 SAT 求解的强大能力，同时提供可视化工具以帮助理解求解过程。

通过不断地研究与创新，SAT 在数独求解方面的应用前景广阔，未来可望实现更高效、灵活的解决方案。

6 体会

对于这道利用 DPLL 算法求解 SAT 句式，并应用于数独的问题，我一开始做的时候有些手足无措，在理解 SAT 的优点这一方面花费了很长时间，逐步理清楚了 DPLL 的神奇之处，接着，在编写 DPLL 代码时，逻辑不清楚成了我最大的编写困难。我最开始写的程序能够算出较小的 cnf 文件，但遇到大文件容易出现内存爆满的情况，后来检查时发现是在 DPLL 函数递归调用过程中有些申请的空间没有释放掉，导致占用内存十分庞大。后来经过改进，虽然还是会出现内存崩掉的情况，但大部分文件还是可以处理。变元选取策略我最开始选择的是直接选择出现次数最多的文字，后来根据一些资料查阅，改进为启发式策略，但通过实践发现效率仍然不高，最后在摸索和试错当中改进为当前模式，先找出现次数最多的正文字，然后再找负文字，这种策略下大部分文件都能快速得到解，但对一些特殊的文件和较大的文件，解起来十分慢也基本上解不出来，因为会出现内存不够的情况。

在数独游戏中，采用的是最简单最直接的随机生成和随机挖洞法，并且实现了玩家自主填入数据的功能，对于求解数独的回溯算法，花费了比较多的时间去理解消化。总之，在这次程序设计实验的过程中，学到了不少实质性的东西，对我以后编写代码也有很大的帮助。

7 参考文献

[1] 严蔚敏等. 数据结构（C 语言版）. 清华大学出版社

[2] Larry Nyhoff. ADTs, Data Structures, and Problem Solving with C++. Second Edition,Calvin College,2005

[3] 殷立峰. Qt C++ 跨平台图形界面程序设计基础. 清华大学出版社,2014:192 ~ 197

[4] 严蔚敏等. 数据结构题集（C 语言版）. 清华大学出版社

附录

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 9 //数独的边长
#define MIN_NUM 17 //数独要想保证有唯一解，最少的初始数字个数就是17
int boolCount;  //布尔变元数量
int clauseCount;  //子句数量
char fileName[100]; //文件名

//十字链表结构体
typedef struct SATNode {
    int data;  //数据域
    SATNode *next;
} SATNode;

typedef struct SATList {
    SATNode *head;  //表头
    SATList *next;
} SATList;

//函数声明
bool addClause(SATList *cnf, SATList *&root);
bool baseDPLL(SATList *&cnf, int *&value, int *&count);
void CopyClause(SATList *&a, SATList *b);
void compute_literal_num(SATList *cnf, int *count);
int *copy(int *a, int n);
void copy(int a[N][N], int b[N][N]);
void destroyClause(SATList *&cnf);
```

```
bool evaluateClause(SATNode *cnf, int v[]);
bool emptyClause(SATList *cnf);
void diySudoku(int board[N][N]);
void Delete(SATList *&cnf, SATList *&pre, SATList *&List, int *count);
void Delete(SATList *&cnf, int literal, int *count);
void fillBoard(int board[N][N]);
bool isUnitClause(SATNode *cnf);
bool isSafe(int board[N][N], int row, int col, int num);
void init(int board[N][N]);
void init(int *&arr, int e = 0);
void mySolveSudoku(int a[N][N], int board[N][N]);
int optimizeDPLL(SATList *&cnf, int value[]);
void print(SATList *CNFList);
void printBoard(int board[N][N]);
int ReadFile(SATList *&cnf);
void removeClause(SATList *&cnf, SATList *&root);
void removeNode(SATNode *&cnf, SATNode *&head);
void removeNumbers(int board[N][N], int count);
int SolvePuzzle(int chess[]);
bool solveSudoku(int board[N][N]);
bool WriteFile(int result, double time, int value[]);

//函数定义
/*
函数名称: main
接受参数: void
函数功能: 主函数
返回值: int
 */
int main(void) {
    srand(time_t(NULL));
    SATList* CNFList = NULL;
```

```
int board[N][N];
clock_t start, finish;  //设置时间变量
double time;
int op = 1, i, result;
int* value, *count; //保存结果，每个变元出现的数量
while (op) {
    system("cls");
    printf("\n\n                    功能菜单 \n");
    printf("-----------------------------------------\n");
    printf("1.读取cnf文件              2.遍历输出每个子句\n");
    printf("3.基础DPLL求解算例         4.优化DPLL求解算例\n");
    printf("5.数独游戏\n");
    printf("0.退出\n");
    printf("-----------------------------------------\n");
    printf("          请选择你的操作[0~6]:");
    scanf("%d", &op);
    switch (op) {
        case 1:
            printf("输入要读取的cnf文件:");
            scanf("%s", fileName);
            ReadFile(CNFList);
            getchar();
            getchar();
            break;
        case 2:
            if (CNFList == NULL)
                printf("未导入文件\n");
            else {
                print(CNFList);
            }
            getchar();
            getchar();
```

```
            break;
        case 3:
            if (CNFList == NULL)
                printf("未导入文件\n");
            else {
                init(value, 1);
                init(count);
                compute_literal_num(CNFList, count);
                start = clock();  //计时开始;
                result = baseDPLL(CNFList, value, count);
                finish = clock();     //结束
                printf("求解结果: %d\n", result);
                if (result == 1) {
                    for (i = 1; i <= boolCount; i++) {
                        if (value[i] == 1)
                            printf("%d ", i);
                        else
                            printf("%d ", -i);
                    }
                    printf("\n");
                }
                time = (double)(finish - start) /
                    CLOCKS_PER_SEC * 1000; //计算运行时间
                printf("start=%ld,finish=%ld,运行时间=%lf ms\n",
                    start, finish, 1.0 * time); //输出运行时间
                WriteFile(result, time, value);
                destroyClause(CNFList);
                printf("文件保存成功\n");
            }
            getchar();
            getchar();
            break;
```

```
case 4:
    if (CNFList == NULL)
        printf("未导入文件\n");
    else {
        value = (int *)malloc(sizeof(int)*(boolCount+1));
        for (i = 1; i <= boolCount; i++)
            value[i] = 1;  //初始化，均赋为1
        start = clock();  //计时开始;
        result = optimizeDPLL(CNFList, value);
        finish = clock();     //结束
        printf("求解结果: %d\n", result);
        if (result == 1) {
            for (i = 1; i <= boolCount; i++) {
                if (value[i] == 1)
                    printf("%d ", i);
                else
                    printf("%d ", -i);
            }
            printf("\n");
        }
        time = (double)(finish - start) /
            CLOCKS_PER_SEC;//计算运行时间
        printf("运行时间=%lfms\n",
            time * 1000);//输出运行时间
        if (WriteFile(result, time, value) == 1)
            printf("结果已保存至同名文件.res\n");
        else
            printf("结果保存失败\n");
    }
    getchar();
    getchar();
    break;
```

```
        case 5:
            init(board);
            fillBoard(board);
            int a[N][N];
            copy(a, board);
            int C = rand() % (N *N - MIN_NUM);
            removeNumbers(board, C); // Remove 40 numbers
            printf("初始棋盘：\n");
            printBoard(board);
            printf("\n\n                    功能菜单 \n");
            printf("----------------------------------------\n");
            printf("1.电脑自动求解\n");
            printf("2.玩家手动求解\n");
            printf("----------------------------------------\n");
            printf("          请选择你的操作[1~2]:");
            scanf("%d", &i);
            if (i == 1) {
                if (solveSudoku(board)) {
                    printf("最终棋盘：\n");
                    printBoard(board);
                } else {
                    printf("No solution exists\n");
                }
            } else if (i == 2) {
                mySolveSudoku(a, board);
            } else {
                printf("输入错误");
            }
            getchar();
            getchar();
            break;
    }
```

```
    }
    printf("欢迎下次使用！");
    return 0;
}

/*
函数名称：ReadFile
接受参数：SATList*&
函数功能：读取文件内容保存到给定参数中
返回值：int
*/
int ReadFile(SATList *&cnf) {
    FILE* fp = fopen(fileName, "r"); //打开文件
    char ch;
    int number;
    SATList* lp;
    SATNode* tp;
    if (!fp) {
        printf("文件打开失败!\n");
        return 0;
    }
    while ((ch = getc(fp)) == 'c') {
        while ((ch = getc(fp)) != '\n')
            continue;              //弃去一整行
    }   //运行到此，已经读取了字符p
    getc(fp);
    getc(fp);
    getc(fp);
    getc(fp); //弃去cnf三个字母
    fscanf(fp, "%d", &boolCount);      //p后的第1个数值是布尔变元数量
    fscanf(fp, "%d", &clauseCount);    //p后的第2个数值是子句数量
    cnf = (SATList *)malloc(sizeof(SATList));
```

```
    cnf->next = NULL;
    cnf->head = (SATNode *)malloc(sizeof(SATNode));
    cnf->head->next = NULL;
    lp = cnf;
    tp = cnf->head;

    //创建数量为clauseCount的子句，i为计数器
    for (int i=0; i<clauseCount;i++,lp=lp->next,tp=lp->head) {
        fscanf(fp, "%d", &number);
        for (; number != 0; tp = tp->next) {
            tp->data = number;  //数据域赋值
            tp->next = (SATNode *)malloc(sizeof(SATNode)); //开辟新结点
            fscanf(fp, "%d", &number);
            if (number == 0)
                tp->next = NULL;
        }
        lp->next = (SATList *)malloc(sizeof(SATList)); //开辟新表
        lp->next->head = (SATNode *)malloc(sizeof(SATNode));
        if (i == clauseCount - 1) {
            lp->next = NULL;
            break;
        }
    }
    printf("读取完毕\n");
    fclose(fp);
    return 1;
}

/*
函数名称: print
接受参数: SATList*
函数功能: 输出cnf链表
```

```
返回值: void
*/
void print(SATList *CNFList) {
    printf("cnf子句如下: \n");
    SATList* lp;
    SATNode* tp;
    for (lp = CNFList; lp != NULL; lp = lp->next) {
        for (tp = lp->head; tp != NULL; tp = tp->next) {
            printf("%d ", tp->data);
        }
        printf("\n");
    }
}

/*
函数名称: baseDPLL
接受参数: SATList*&,int *&,int *&
函数功能: 求解SAT句式的基础DPLL算法
返回值: bool
*/
bool baseDPLL(SATList *&cnf, int *&value, int *&count) {
    if (cnf == NULL) { //所有的子句都变成了空句, 表明存在解
        return true;
    }
FIND:
    SATList *List = cnf;
    while (List && !isUnitClause(List->head)) { //不是单子句
        if (List->head == NULL) {
            //所有文字都删除了, 子句变成了空句, 证明每一个文字都是假
            return false;
        }
        List = List->next;
```

```
}
if (List) { //存在一个子句有单子句
    //找到了单句
    int e = List->head->data; //e是单子句中的文字，可以直接得出其值
    if (e < 0) { //修改最终结果
        value[-e] = 0;
    }
    Delete(cnf, e, count); //删除所有含有e的子句以及删除-e文字
    goto FIND;
}
//所有子句都不是单句
SATList *new_List;
CopyClause(new_List, cnf); //暂存数据
int *new_value = copy(value, boolCount);
int *new_every_bool_num = copy(count, boolCount);
int e = 0, max = 0;
for (int i = 1; i <= boolCount; i++) { //找到出现最多的变元
    if (max < count[i]) {
        max = count[i];
        e = i;
    }
}
if (e == 0) {
    return true;
}
Delete(cnf, e, count);
int result = baseDPLL(cnf, value, count);
if (result) { //e的正项分支正确
    destroyClause(new_List); //释放内存
    free(new_value);
    free(new_every_bool_num);
    return true;
```

```
    } else { //该分支错误，采用e的负项分支
        destroyClause(cnf); //释放现有内存
        free(value);
        free(count);
        CopyClause(cnf, new_List); //重新赋值暂存数据
        value = copy(new_value, boolCount);
        count = copy(new_every_bool_num, boolCount);
        value[e] = 0;
        Delete(cnf, -e, count);
        result = baseDPLL(cnf, value, count);
        destroyClause(new_List);
        free(new_value);
        free(new_every_bool_num);
        if (result) { //另一分支存在解
            return true;
        } else { //另一分支也不对，继续回溯到上一分支
            return false;
        }
    }
    if (cnf == NULL) //将所有子句都删除了，即得到了最终解
        return true;
    else //无解
        return false;
}

/*
函数名称: destroyClause
接受参数: SATList*&
函数功能: 销毁链表
返回值: void
*/
void destroyClause(SATList *&cnf) {
```

```
    SATList* lp1, *lp2;
    SATNode* tp1, *tp2;
    for (lp1 = cnf; lp1 != NULL; lp1 = lp2) {
        lp2 = lp1->next;
        for (tp1 = lp1->head; tp1 != NULL; tp1 = tp2) {
            tp2 = tp1->next;
            free(tp1);
        }
        free(lp1);
    }
    cnf = NULL;
}

/*
函数名称: isUnitClause
接受参数: SATNode*
函数功能: 判断是否为单子句, 是返回true, 不是返回false
返回值: bool
*/
bool isUnitClause(SATNode *cnf) {
    if (cnf != NULL && cnf->next == NULL)
        return true;
    else
        return false;
}

/*
函数名称: evaluateClause
接受参数: SATList*,int*
函数功能: 评估子句的真假状态, 真返回true, 假返回false
返回值: bool
*/
```

```
bool evaluateClause(SATNode *cnf, int v[]) {
    SATNode* tp = cnf;
    while (tp != NULL) {
        if ((tp->data > 0 && v[tp->data] == 1) ||
                (tp->data < 0 && v[-tp->data] == 0))
            return true;
    }
    return false;
}

/*
函数名称: removeClause
接受参数: SATList*&,SATList*&
函数功能: 在已有的十字链表中删除指定的子句
返回值: void
*/
void removeClause(SATList *&cnf, SATList *&root) {
    SATList* lp = root;
    if (lp == cnf)
        root = root->next;  //删除为头
    else {
        while (lp != NULL && lp->next != cnf)
            lp = lp->next;
        lp->next = lp->next->next;
    }
    free(cnf);
    cnf = NULL;
}

/*
函数名称: removeNote
接受参数: SATNode*&,SATNode*&
```

```
函数功能: 在指定的子句中删除指定的文字
返回值: void
*/
void removeNode(SATNode *&cnf, SATNode *&head) {
    SATNode* lp = head;
    if (lp == cnf)
        head = head->next;  //删除为头
    else {
        while (lp != NULL && lp->next != cnf)
            lp = lp->next;
        lp->next = lp->next->next;
    }
    free(cnf);
    cnf = NULL;
}

/*
函数名称: addClause
接受参数: SATList*,SATList*&
函数功能: 在root链表中添加cnf子句, 添加成功返回1, 失败返回0
返回值: bool
*/
bool addClause(SATList *cnf, SATList *&root) {
    //直接插入在表头
    if (cnf != NULL) {
        cnf->next = root;
        root = cnf;
        return true;
    }
    return false;
}
```

```
/*
函数名称: emptyClause
接受参数: SATList*
函数功能: 判断是否含有空子句, 是返回1, 不是返回0
返回值: bool
*/
bool emptyClause(SATList *cnf) {
    SATList* lp = cnf;
    while (lp != NULL) {
        if (lp->head == NULL) //有空子句
            return true;
        lp = lp->next;
    }
    return false;
}

/*
函数名称: CopyClause
接受参数: SATList*,SATList*
函数功能: 将链表b的值复制到链表a中
返回值: void
*/
void CopyClause(SATList *&a, SATList *b) {
    SATList* lpa, *lpb;
    SATNode* tpa, *tpb;
    a = (SATList *)malloc(sizeof(SATList));
    a->head = (SATNode *)malloc(sizeof(SATNode));
    a->next = NULL;
    a->head->next = NULL;
    for (lpb=b,lpa=a;lpb!=NULL;lpb=lpb->next,lpa=lpa->next) {
        for (tpb = lpb->head, tpa = lpa->head; tpb != NULL;
            tpb = tpb->next, tpa = tpa->next) {
```

```
                tpa->data = tpb->data;
                tpa->next = (SATNode *)malloc(sizeof(SATNode));
                tpa->next->next = NULL;
                if (tpb->next == NULL) {
                    free(tpa->next);
                    tpa->next = NULL;
                }
            }
            lpa->next = (SATList *)malloc(sizeof(SATList));
            lpa->next->head = (SATNode *)malloc(sizeof(SATNode));
            lpa->next->next = NULL;
            lpa->next->head->next = NULL;
            if (lpb->next == NULL) {
                free(lpa->next->head);
                free(lpa->next);
                lpa->next = NULL;
            }
        }
    }

/*
函数名称: optimizeDPLL
接受参数: SATList*&,int *
函数功能: 求解SAT问题, 若有解则返回1, 无解返回0
返回值: int
*/
int optimizeDPLL(SATList *&cnf, int value[]) {
    SATList* tp = cnf, *lp = cnf, *sp;
    SATNode* dp;
    int* count, i, MaxWord, max, re;
    //count记录每个文字出现次数,MaxWord记录出现最多次数的文字
    count = (int *)malloc(sizeof(int) * (boolCount * 2 + 1));
```

```
FIND:
    while (tp != NULL && isUnitClause(tp->head) == 0)
        tp = tp->next;  //找到表中的单子句
    if (tp != NULL) {
        //单子句规则简化
        if (tp->head->data > 0)
            value[tp->head->data] = 1;
        else
            value[-tp->head->data] = 0;
        re = tp->head->data;
        for (lp=cnf;lp!=NULL;lp=sp) {
            sp = lp->next;

            //查找含有核心文字的句子
            for (dp = lp->head; dp != NULL; dp = dp->next) {
                if (dp->data == re) {
                    removeClause(lp, cnf);  //删除子句, 简化式子
                    break;
                }
                if (dp->data == -re) {
                    removeNode(dp, lp->head);  //删除文字, 简化子句
                    break;
                }
            }
        }
        //极简化规则简化后
        if (cnf == NULL) {
            free(count);
            return 1;
        } else if (emptyClause(cnf)) {
            free(count);
            destroyClause(cnf);
```

```
            return 0;
        }
        tp = cnf;
        goto FIND;   //继续简化
    }
    for (i = 0; i <= boolCount * 2; i++)
        count[i] = 0;   //初始化

    for (lp = cnf; lp != NULL; lp = lp->next) {
        for (dp = lp->head; dp != NULL; dp = dp->next) {
            if (dp->data > 0)
                count[dp->data]++;
            else
                count[boolCount - dp->data]++;
        }
    }
    max = 0;

    //找到出现次数最多的正文字
    for (i = 2; i <= boolCount; i++) {
        if (max < count[i]) {
            max = count[i];
            MaxWord = i;
        }
    }

    if (max == 0) {
        //若没有出现正文字,找到出现次数最多的负文字
        for (i = boolCount + 1; i <= boolCount * 2; i++) {
            if (max < count[i]) {
                max = count[i];
                MaxWord = -i;
```

```
            }
        }
    }
    free(count);
    lp = (SATList *)malloc(sizeof(SATList));
    lp->head = (SATNode *)malloc(sizeof(SATNode));
    lp->head->data = MaxWord;
    lp->head->next = NULL;
    lp->next = NULL;
    CopyClause(tp, cnf);
    addClause(lp, tp);
    if (optimizeDPLL(tp, value) == 1)
        return 1;  //在第一分支中搜索
    destroyClause(tp);
    lp = (SATList *)malloc(sizeof(SATList));
    lp->head = (SATNode *)malloc(sizeof(SATNode));
    lp->head->data = -MaxWord;
    lp->head->next = NULL;
    lp->next = NULL;
    addClause(lp, cnf);
    re = optimizeDPLL(cnf, value);
    //回溯到执行分支策略的初态进入另一分支
    destroyClause(cnf);
    return re;
}

/*
函数名称: WriteFile
接受参数: int,double,int[]
函数功能: 将运行结果保存至同名文件, 文件拓展名为.res,
    保存成功返回1, 失败返回0
返回值: bool
```

```
*/
bool WriteFile(int result, double time, int value[]) {
    FILE* fp;
    int i;
    for (i = 0; fileName[i] != '\0'; i++) {
        //修改拓展名
        if (fileName[i] == '.' && fileName[i + 4] == '\0') {
            fileName[i + 1] = 'r';
            fileName[i + 2] = 'e';
            fileName[i + 3] = 's';
            break;
        }
    }
    if (fopen_s(&fp, fileName, "w")) {
        printf("文件打开失败!\n");
        return false;
    }
    fprintf(fp, "s %d\nv ", result);  //求解结果
    if (result == 1) {
        //保存解值
        for (i = 1; i <= boolCount; i++) {
            if (value[i] == 1)
                fprintf(fp, "%d ", i);
            else
                fprintf(fp, "%d ", -i);
        }
    }
    fprintf(fp, "\nt %lfms", time * 1000);  //运行时间/毫秒
    fclose(fp);
    return true;
}
```

```
/*
函数名称: SolvePuzzle
接受参数: int[]
函数功能: 求解输入棋盘，有解返回1，无解返回0
返回值: int
*/
int SolvePuzzle(int chess[]) {
    SATList* cnf = NULL, *lp;
    SATNode* dp;
    int* remember, i, j, k, rol;
    boolCount = N * N;

    //添加单子句
    for (i = 1; i <= N *N; i++) {
        if (chess[i] == 0) {
            lp = (SATList *)malloc(sizeof(SATList));
            lp->head = (SATNode *)malloc(sizeof(SATNode));
            lp->head->next = NULL;
            lp->next = NULL;
            lp->head->data = -i;
            addClause(lp, cnf);
        } else if (chess[i] == 1) {
            lp = (SATList *)malloc(sizeof(SATList));
            lp->head = (SATNode *)malloc(sizeof(SATNode));
            lp->head->next = NULL;
            lp->next = NULL;
            lp->head->data = i;
            addClause(lp, cnf);
        }
    }

    //约束1: 不出现连续三个重复数字
```

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N - 2; j++) {
        //横向，正文字
        lp = (SATList *)malloc(sizeof(SATList));
        lp->head = (SATNode *)malloc(sizeof(SATNode));
        lp->head->next = (SATNode *)malloc(sizeof(SATNode));
        lp->head->next->next = (SATNode *)malloc(sizeof(SATNode));
        lp->head->next->next->next = NULL;
        lp->next = NULL;
        lp->head->data = i * N + j + 1;
        lp->head->next->data = i * N + j + 2;
        lp->head->next->next->data = i * N + j + 3;
        addClause(lp, cnf);
        //横向，负文字
        lp = (SATList *)malloc(sizeof(SATList));
        lp->head = (SATNode *)malloc(sizeof(SATNode));
        lp->head->next = (SATNode *)malloc(sizeof(SATNode));
        lp->head->next->next = (SATNode *)malloc(sizeof(SATNode));
        lp->head->next->next->next = NULL;
        lp->next = NULL;
        lp->head->data = -(i *N + j + 1);
        lp->head->next->data = -(i *N + j + 2);
        lp->head->next->next->data = -(i *N + j + 3);
        addClause(lp, cnf);
        //纵向，正文字
        lp = (SATList *)malloc(sizeof(SATList));
        lp->head = (SATNode *)malloc(sizeof(SATNode));
        lp->head->next = (SATNode *)malloc(sizeof(SATNode));
        lp->head->next->next = (SATNode *)malloc(sizeof(SATNode));
        lp->head->next->next->next = NULL;
        lp->next = NULL;
        lp->head->data = i + j * N + 1;
```

```
            lp->head->next->data = i + (j + 1) * N + 1;
            lp->head->next->next->data = i + (j + 2) * N + 1;
            addClause(lp, cnf);
            //纵向，负文字
            lp = (SATList *)malloc(sizeof(SATList));
            lp->head = (SATNode *)malloc(sizeof(SATNode));
            lp->head->next = (SATNode *)malloc(sizeof(SATNode));
            lp->head->next->next = (SATNode *)malloc(sizeof(SATNode));
            lp->head->next->next->next = NULL;
            lp->next = NULL;
            lp->head->data = -(i + j *N + 1);
            lp->head->next->data = -(i + (j + 1) * N + 1);
            lp->head->next->next->data = -(i + (j + 2) * N + 1);
            addClause(lp, cnf);
        }
    }

    //约束2: 在每一行、每一列中1与0的个数相同
    remember = (int *)malloc(sizeof(int) * (N / 2 + 1));
    //每一行
    for (rol = 0; rol < N; rol++) {
        for (i = 0; i < N / 2 + 1; i++)
            remember[i] = i + 1;  //初始化
COMBINATION1:
        for (i = N / 2; remember[i] <= N; remember[i]++) {
            lp = (SATList *)malloc(sizeof(SATList));
            lp->head = (SATNode *)malloc(sizeof(SATNode));
            lp->head->next = NULL;
            lp->next = NULL;
            for(j=0,dp=lp->head;j<N/2+1;j++,dp=dp->next) {
                dp->data = remember[j] + rol * N;
                if (j == N / 2)
```

```
                    break;
                dp->next = (SATNode *)malloc(sizeof(SATNode));
                dp->next->next = NULL;
            }
            addClause(lp, cnf);
            lp = (SATList *)malloc(sizeof(SATList));
            lp->head = (SATNode *)malloc(sizeof(SATNode));
            lp->head->next = NULL;
            lp->next = NULL;
            for(j=0,dp=lp->head;j<N/2+1;j++,dp=dp->next) {
                dp->data = -(remember[j] + rol *N);
                if (j == N / 2)
                    break;
                dp->next = (SATNode *)malloc(sizeof(SATNode));
                dp->next->next = NULL;
            }
            addClause(lp, cnf);
        }
        for (i = N / 2; i >= 0 && remember[i] >= N / 2 + i; i--);
//找到达到饱和的最高位
        if (i <= 0)
            continue;  //该行组合序列全部排完,进入下一行
        remember[i]++;
        for (j = i + 1; j < N / 2 + 1; j++)
            remember[j] = remember[j - 1] + 1; //序列后移
        goto COMBINATION1;
    }
    //每一列
    for (rol = 1; rol <= N; rol++) {
        for (i = 0; i < N / 2 + 1; i++)
            remember[i] = i;  //初始化
COMBINATION2:
```

```
        for (i = N / 2; remember[i] < N; remember[i]++) {
            lp = (SATList *)malloc(sizeof(SATList));
            lp->head = (SATNode *)malloc(sizeof(SATNode));
            lp->head->next = NULL;
            lp->next = NULL;
            for (j=0,dp=lp->head;j<N/2+1; j++, dp = dp->next) {
                dp->data = remember[j] * N + rol;
                if (j == N / 2)
                    break;
                dp->next = (SATNode *)malloc(sizeof(SATNode));
                dp->next->next = NULL;
            }
            addClause(lp, cnf);
            lp = (SATList *)malloc(sizeof(SATList));
            lp->head = (SATNode *)malloc(sizeof(SATNode));
            lp->head->next = NULL;
            lp->next = NULL;
            for(j=0,dp=lp->head;j<N/2+ 1; j++, dp = dp->next) {
                dp->data = -(remember[j] * N + rol);
                if (j == N / 2)
                    break;
                dp->next = (SATNode *)malloc(sizeof(SATNode));
                dp->next->next = NULL;
            }
            addClause(lp, cnf);
        }
        for (i = N / 2; i >= 0 && remember[i] > N / 2 + i; i--);
//找到达到饱和的最高位
        if (i <= 0)
            continue;  //该列组合序列全部排完,进入下一列
        remember[i]++;
        for (j = i + 1; j < N / 2 + 1; j++)
```

```
            remember[j] = remember[j - 1] + 1; //序列后移
        goto COMBINATION2;
    }

    //约束3: 不存在重复的行与重复的列
    //不重复行:
    for (i = 0; i < N - 1; i++) {
        for (j = i + 1; j < N; j++) {
            rol = boolCount; //记录添加变元之前变元数量
            for (k = 1; k <= N; k++) {
                //引入前一个新变元:
                boolCount++;
                //语句1:
                lp = (SATList *)malloc(sizeof(SATList));
                lp->head = (SATNode *)malloc(sizeof(SATNode));
                lp->head->next = (SATNode *)malloc(sizeof(SATNode));
                lp->head->next->next = NULL;
                lp->next = NULL;
                lp->head->data = i * N + k;
                lp->head->next->data = -boolCount;
                addClause(lp, cnf);
                //语句2:
                lp = (SATList *)malloc(sizeof(SATList));
                lp->head = (SATNode *)malloc(sizeof(SATNode));
                lp->head->next = (SATNode *)malloc(sizeof(SATNode));
                lp->head->next->next = NULL;
                lp->next = NULL;
                lp->head->data = -(j *N + k);
                lp->head->next->data = -boolCount;
                addClause(lp, cnf);
                //语句3:
                lp = (SATList *)malloc(sizeof(SATList));
```

```
lp->head = (SATNode *)malloc(sizeof(SATNode));
lp->head->next = (SATNode *)malloc(sizeof(SATNode));
lp->head->next->next = (SATNode *)malloc(sizeof(SATNode))
lp->head->next->next->next = NULL;
lp->next = NULL;
lp->head->data = -(i *N + k);
lp->head->next->data = j * N + k;
lp->head->next->next->data = boolCount;
addClause(lp, cnf);
//引入后一个新变元:
boolCount++;
//语句1:
lp = (SATList *)malloc(sizeof(SATList));
lp->head = (SATNode *)malloc(sizeof(SATNode));
lp->head->next = (SATNode *)malloc(sizeof(SATNode));
lp->head->next->next = NULL;
lp->next = NULL;
lp->head->data = -(i *N + k);
lp->head->next->data = -boolCount;
addClause(lp, cnf);
//语句2:
lp = (SATList *)malloc(sizeof(SATList));
lp->head = (SATNode *)malloc(sizeof(SATNode));
lp->head->next = (SATNode *)malloc(sizeof(SATNode));
lp->head->next->next = NULL;
lp->next = NULL;
lp->head->data = j * N + k;
lp->head->next->data = -boolCount;
addClause(lp, cnf);
//语句3:
lp = (SATList *)malloc(sizeof(SATList));
lp->head = (SATNode *)malloc(sizeof(SATNode));
```

```
                lp->head->next = (SATNode *)malloc(sizeof(SATNode));
                lp->head->next->next=(SATNode *)malloc(sizeof(SATNode));
                lp->head->next->next->next = NULL;
                lp->next = NULL;
                lp->head->data = i * N + k;
                lp->head->next->data = -(j *N + k);
                lp->head->next->next->data = boolCount;
                addClause(lp, cnf);
            }
            //添加长句：不重复行满足的关系
            lp = (SATList *)malloc(sizeof(SATList));
            lp->head = (SATNode *)malloc(sizeof(SATNode));
            lp->head->next = NULL;
            lp->next = NULL;
            for(k=rol+1,dp=lp->head;k<= boolCount;k++,dp=dp->next){
                dp->data = k;
                if (k == boolCount)
                    break;
                dp->next = (SATNode *)malloc(sizeof(SATNode));
                dp->next->next = NULL;
            }
            addClause(lp, cnf);
        }
    }
    //不重复列
    for (i = 1; i <= N - 1; i++) {
        for (j = i + 1; j <= N; j++) {
            rol = boolCount; //记录添加变元之前变元数量
            for (k = 0; k < N; k++) {
                //引入前一个新变元：
                boolCount++;
                //语句1：
```

```
lp = (SATList *)malloc(sizeof(SATList));
lp->head = (SATNode *)malloc(sizeof(SATNode));
lp->head->next = (SATNode *)malloc(sizeof(SATNode));
lp->head->next->next = NULL;
lp->next = NULL;
lp->head->data = i + k * N;
lp->head->next->data = -boolCount;
addClause(lp, cnf);
//语句2:
lp = (SATList *)malloc(sizeof(SATList));
lp->head = (SATNode *)malloc(sizeof(SATNode));
lp->head->next = (SATNode *)malloc(sizeof(SATNode));
lp->head->next->next = NULL;
lp->next = NULL;
lp->head->data = -(j + k *N);
lp->head->next->data = -boolCount;
addClause(lp, cnf);
//语句3:
lp = (SATList *)malloc(sizeof(SATList));
lp->head = (SATNode *)malloc(sizeof(SATNode));
lp->head->next = (SATNode *)malloc(sizeof(SATNode));
lp->head->next->next=(SATNode *)malloc(sizeof(SATNode));
lp->head->next->next->next = NULL;
lp->next = NULL;
lp->head->data = -(i + k *N);
lp->head->next->data = j + k * N;
lp->head->next->next->data = boolCount;
addClause(lp, cnf);
//引入后一个新变元:
boolCount++;
//语句1:
lp = (SATList *)malloc(sizeof(SATList));
```

```
            lp->head = (SATNode *)malloc(sizeof(SATNode));
            lp->head->next = (SATNode *)malloc(sizeof(SATNode));
            lp->head->next->next = NULL;
            lp->next = NULL;
            lp->head->data = -(i + k *N);
            lp->head->next->data = -boolCount;
            addClause(lp, cnf);
            //语句2:
            lp = (SATList *)malloc(sizeof(SATList));
            lp->head = (SATNode *)malloc(sizeof(SATNode));
            lp->head->next = (SATNode *)malloc(sizeof(SATNode));
            lp->head->next->next = NULL;
            lp->next = NULL;
            lp->head->data = j + k * N;
            lp->head->next->data = -boolCount;
            addClause(lp, cnf);
            //语句3:
            lp = (SATList *)malloc(sizeof(SATList));
            lp->head = (SATNode *)malloc(sizeof(SATNode));
            lp->head->next = (SATNode *)malloc(sizeof(SATNode));
            lp->head->next->next=(SATNode *)malloc(sizeof(SATNode));
            lp->head->next->next->next = NULL;
            lp->next = NULL;
            lp->head->data = i + k * N;
            lp->head->next->data = -(j + k *N);
            lp->head->next->next->data = boolCount;
            addClause(lp, cnf);
        }
        //添加长句: 不重复行满足的关系
        lp = (SATList *)malloc(sizeof(SATList));
        lp->head = (SATNode *)malloc(sizeof(SATNode));
        lp->head->next = NULL;
```

```
                lp->next = NULL;
                for(k=rol+1,dp=lp->head;k<= boolCount;k++,dp=dp->next){
                    dp->data = k;
                    if (k == boolCount)
                        break;
                    dp->next = (SATNode *)malloc(sizeof(SATNode));
                    dp->next->next = NULL;
                }
                addClause(lp, cnf);
            }
        }
        free(remember);
        remember = (int *)malloc(sizeof(int) * (boolCount + 1));
        for (i = 1; i <= boolCount; i++)
            remember[i] = 1;  //初始化
        if (optimizeDPLL(cnf, remember) == 1) {
            for (i = 1; i <= N *N; i++)
                chess[i] = remember[i];
            free(remember);
            destroyClause(cnf);
            return 1;
        } else {
            free(remember);
            destroyClause(cnf);
            return 0;
        }
}

/*
函数名称: printBoard
接受参数: int[][]
函数功能: 输出棋盘
```

```
返回值: void
*/
void printBoard(int board[N][N]) {
    for (int r = 0; r < N; r++) {
        for (int d = 0; d < N; d++) {
            if (board[r][d])
                printf("%d ", board[r][d]);
            else
                printf("- ");
        }
        printf("\n");
    }
}

/*
函数名称: isSafe
接受参数: int[][],int,int,int
函数功能: 判断数独的输入或求解是否满足数独条件
返回值: bool
*/
bool isSafe(int board[N][N], int row, int col, int num) {
    for (int x = 0; x < N; x++) {
        if (board[row][x] == num || board[x][col] == num) {
            return false;
        }
    }
    int startRow = row / 3 * 3, startCol = col / 3 * 3;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (board[i + startRow][j + startCol] == num) {
                return false;
            }
```

```
        }
    }
    if (row == col) {
        for (int x = 0; x < N; x++) {
            if (board[x][x] == num)
                return false;
        }
    }
    if (row + col == N - 1) {
        for (int x = 0; x < N; x++) {
            if (board[x][N - 1 - x] == num)
                return false;
        }
    }
    return true;
}

/*
函数名称: solveSudoku
接受参数: int[][]
函数功能: 回溯法求解数独
返回值: bool
*/
bool solveSudoku(int board[N][N]) {
    for (int row = 0; row < N; row++) {
        for (int col = 0; col < N; col++) {
            if (board[row][col] == 0) {
                for (int num = 1; num <= 9; num++) {
                    if (isSafe(board, row, col, num)) {
                        board[row][col] = num;
                        if (solveSudoku(board)) {
                            return true;
```

```
                        }
                        board[row][col] = 0;
                    }
                }
                return false;
            }
        }
    }
    return true;
}

/*
函数名称: init
接受参数: int[][]
函数功能: 初始化数独
返回值: void
*/
void init(int board[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            board[i][j] = 0;
        }
    }
}

/*
函数名称: fillBoard
接受参数: int[][]
函数功能: 填充数独数字
返回值: void
*/
void fillBoard(int board[N][N]) {
```

```
    for (int num = 1; num <= 9; num++) {
        int row = rand() % N;
        int col = rand() % N;
        while (!isSafe(board, row, col, num)) {
            row = rand() % N;
            col = rand() % N;
        }
        board[row][col] = num;
    }
    solveSudoku(board);
}

/*
函数名称: init
接受参数: int[][],int
函数功能: 挖洞法将数独挖空
返回值: void
*/
void removeNumbers(int board[N][N], int count) {
    while (count != 0) {
        int i = rand() % N;
        int j = rand() % N;
        if (board[i][j] != 0) {
            board[i][j] = 0;
            count--;
        }
    }
}

/*
函数名称: conpute_literal_num
接受参数: SATList *,int*
```

```
函数功能：计算十字链表中每一个变元的数量，保存在count数组中
返回值：void
*/
void compute_literal_num(SATList *cnf, int *count) {
    for (SATList *lp = cnf; lp != NULL; lp = lp->next) {
        for (SATNode *dp = lp->head; dp != NULL; dp = dp->next) {
            count[abs(dp->data)]++;
        }
    }
}

/*
函数名称：Delete
接受参数：SATList *&,SATList *&,SATList *&,int *
函数功能：删除cnf链表中List指向的子句，pre为List的前继指针
返回值：void
*/
void Delete(SATList *&cnf, SATList *&pre, SATList *&List, int *count) {
    SATNode *p = List->head, *next = p->next;
    while (next != NULL) {
        count[abs(p->data)]--;
        free(p);
        p = next;
        next = p->next;
    }
    count[abs(p->data)]--;
    free(p); //释放每个文字的指针空间
    p = next = NULL;
    if (List == cnf) {
        cnf = List->next;
        pre = cnf;
    } else {
```

```
            pre->next = List->next;
        }
        free(List); //释放子句的指针空间
        List = NULL;
}

/*
函数名称: Delete
接受参数: SATLis *&,int,int*
函数功能: 删除cnf链表中含有literal文字的子句以及-literal的文字
返回值: void
*/
void Delete(SATList *&cnf, int literal, int *count) {
    int flag = 0;
    SATList *List = cnf, *pre = cnf;
    while (List) {
        SATNode *Node = List->head, *pre_Node = Node;
        while (Node) {
            if (Node->data == literal) {
            //在非单子句中找到了变元e, 直接删除整个子句
                Delete(cnf, pre, List, count); //删除了整个子句
                Node = NULL;
                List = pre;
                flag = 1;
                break;
            } else if (Node->data == -literal) {
            //-e为假, 只删除该文字, 由子句中其他文字判断真假
                if (Node == pre_Node) { //被删除的文字在链头
                    pre_Node = Node->next;
                    count[abs(literal)]--;
                    free(Node);
                    Node = pre_Node;
```

```
                    List->head = Node; //改变头结点
                } else { //被删除的文字不在链头
                    pre_Node->next = Node->next;
                    count[abs(literal)]--;
                    free(Node);
                    Node = pre_Node->next;
                }
            } else { //不是e或-e
                pre_Node = Node;
                Node = Node->next;
            }
        }
        pre = List;
        if (flag == 0)
            List = pre->next;
        flag = 0;
    }
}

/*
函数名称: copy
接受参数: int*,int
函数功能: 复制a数组
返回值: int *
*/
int *copy(int *a, int n) {
    int *arr = (int *)malloc(sizeof(int) * (n + 1));
    for (int i = 0; i <= n; i++) {
        arr[i] = a[i];
    }
    return arr;
}
```

```
/*
函数名称: init
接受参数: int*&,int
函数功能: 初始化数组
返回值: void
*/
void init(int *&arr, int e) {
    arr = (int *)malloc(sizeof(int) * (boolCount + 1));
    for (int i = 1; i <= boolCount; i++) {
        arr[i] = e;
    }
}

/*
函数名称: copy
接受参数: int [][],int [][]
函数功能: 将数组b复制给数组a
返回值: void
*/
void copy(int a[N][N], int b[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            a[i][j] = b[i][j];
        }
    }
}

/*
函数名称: mySolveSudoku
接受参数: int [][],int [][]
函数功能: 玩家自己求解数独游戏
```

```
返回值: void
*/
void mySolveSudoku(int a[N][N], int board[N][N]) {
    int op = 1;
    while (op) {
        printf("\n\n                    功能菜单 \n");
        printf("----------------------------------------\n");
        printf("1.填入数字\n");
        printf("2.查看棋盘\n");
        printf("0.退出游戏\n");
        printf("----------------------------------------\n");
        printf("          请选择你的操作[0~2]:");
        scanf("%d", &op);
        if (op == 1) {
            int line, con, num;
            printf("请输入行、列、值: ");
            scanf("%d %d %d", &line, &con, &num);
            if (a[line - 1][con - 1] == num) {
                printf("填入成功\n");
                board[line - 1][con - 1] = num;
            } else {
                printf("填入失败, 请重新输入\n");
            }
        } else if (op == 2) {
            printBoard(board);
        } else if (op == 0) {
            printf("欢迎下次游戏! ");
        } else {
            printf("输入错误, 请重新输入\n");
        }
        getchar();
        getchar();
```

```
    }
}
```