

课程设计报告

**题目：基于SAT的二进制数独游戏求解程序**

|  |  |
| --- | --- |
| **课程名称：** | **程序设计综合设计课程** |
| **专业班级：** | **CS1803** |
| **学 号：** | **U201814792** |
| **姓 名：** | **宋家垚** |
| **指导教师：** | **袁凌** |
| **报告日期：** | **2020.03.21** |

**计算机科学与技术学院**

# 任 务 书

* **设计内容**

SAT问题即命题逻辑公式的可满足性问题（satisfiability problem），是计算机科学与人工智能基本问题，是一个典型的NP完全问题，可广泛应用于许多实际问题如硬件设计、安全协议验证等，具有重要理论意义与应用价值。本设计要求基于DPLL算法实现一个完备SAT求解器，对输入的CNF范式算例文件，解析并建立其内部表示；精心设计问题中变元、文字、子句、公式等有效的物理存储结构以及一定的分支变元处理策略，使求解器具有优化的执行性能；对一定规模的算例能有效求解，输出与文件保存求解结果，统计求解时间。

* **设计要求**

要求具有如下功能：

1. **输入输出功能：**包括程序执行参数的输入，SAT算例cnf文件的读取，执行结果的输出与文件保存等。(15%)
2. **公式解析与验证：**读取cnf算例文件，解析文件，基于一定的物理结构，建立公式的内部表示；并实现对解析正确性的验证功能，即遍历内部结构逐行输出与显示每个子句，与输入算例对比可人工判断解析功能的正确性。数据结构的设计可参考文献[1-3]。(15%)
3. **DPLL过程：**基于DPLL算法框架，实现SAT算例的求解。(35%)
4. **时间性能的测量：**基于相应的时间处理函数（参考time.h），记录DPLL过程执行时间（以毫秒为单位），并作为输出信息的一部分。(5%)
5. **程序优化：**对基本DPLL的实现进行存储结构、分支变元选取策略[1-3]等某一方面进行优化设计与实现，提供较明确的性能优化率结果。优化率的计算公式为：[(t-to)/t]\*100%,其中t 为未对DPLL优化时求解基准算例的执行时间，to则为优化DPLL实现时求解同一算例的执行时间。(15%)
6. **SAT应用：**将二进制数独游戏[5，6]问题转化为SAT问题[6]，并集成到上面的求解器进行问题求解，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[6-9]。(15%)

* **参考文献**

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

[2]TanbirAhmed.An Implementation of the DPLL Algorithm.Masterthesis,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]CarstenSinz.Visualizing SAT Instances and Runsof the DPLL Algorithm.JAutom Reasoning (2007) 39:219–243

[5] Binary Puzzle：http://www.binarypuzzle.com/

[6] Putranto H. Utomo and Rusydi H. Makarim. Solving a Binary Puzzle. Mathematics in Computer Science,(2017) 11:515–526

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

[8]InsLynce and JolOuaknine. Sudoku as a sat problem.In Proceedings of the 9th InternationalSymposium on Artificial Intelligence and Mathematics, AIMATH 2006, Fort Lauderdale.Springer,2006.

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

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

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

# 目 录

[课程设计报告 I](#_Toc35723486)

[任 务 书 I](#_Toc35723487)

[目 录 III](#_Toc35723488)

[1 引言 1](#_Toc35723489)

[1.1 课题背景与意义 1](#_Toc35723490)

[1.2 国内外研究情况 1](#_Toc35723491)

[1.3 课程设计的主要研究工作 2](#_Toc35723492)

[2 系统需求分析与总体设计 3](#_Toc35723493)

[2.1 系统需求分析 3](#_Toc35723494)

[2.2 系统总体设计 3](#_Toc35723495)

[3 系统详细设计 5](#_Toc35723496)

[3.1 有关数据结构的定义 5](#_Toc35723497)

[3.2 主要算法设计 6](#_Toc35723498)

[3.2.1 cnfparser 6](#_Toc35723499)

[3.2.2 solver 7](#_Toc35723500)

[3.2.3 binarypuzzle 8](#_Toc35723501)

[4 系统实现与测试 9](#_Toc35723502)

[4.1 系统实现 9](#_Toc35723503)

[4.1.1 开发环境及程序相关定义 9](#_Toc35723504)

[4.1.2 模块函数说明 10](#_Toc35723505)

[4.2 系统测试 13](#_Toc35723506)

[4.2.1 DPLL部分 13](#_Toc35723507)

[4.2.2 数独部分 19](#_Toc35723508)

[5 总结与展望 21](#_Toc35723509)

[5.1 全文总结 21](#_Toc35723510)

[5.2 工作展望 22](#_Toc35723511)

[6 体会 23](#_Toc35723512)

[参考文献 24](#_Toc35723513)

[附录A（head.h） 25](#_Toc35723514)

[附录B（main.c） 28](#_Toc35723515)

[附录C（cnfparser.c） 29](#_Toc35723516)

[附录D（solver.c） 35](#_Toc35723517)

[附录E（binarypuzzle.c） 45](#_Toc35723518)

[附录F（display.c） 56](#_Toc35723519)

[附录G（程序使用说明） 63](#_Toc35723520)

# 1 引言

## 1.1 课题背景与意义

在计算机科学中，布尔可满足性问题（有时称为命题可满足性问题，缩写为SATISFIABILITY或SAT）是确定是否存在满足给定布尔公式的解释的问题。SAT是第一个已知的NP完全问题，1971年多伦多大学的Stephen Cook和1973年国家科学院的Leonid Levin独立证明了这一问题。在此之前，NP完全问题的概念甚至不存在。SAT问题在计算机科学、复杂性理论、密码系统、人工智能等领域发挥着至关重要的作用。程控电话的自动交换、大型数据库的维护、大规模集成电路的自动布线、软件自动开发、机器人动作规划等，都可转化成SAT问题。因此致力于寻找求解SAT问题的快速而有效的算法，不仅在理论研究上而且在许多应用领域都具有极其重要的意义。

## 1.2 国内外研究情况

自1960年Davis和Putnam提出DP算法以来，SAT求解研究逐步受到关注。然而1971年Cook证明SAT问题是NPC之后，人们对SAT的重视程度减弱。后来人们对SAT问题有了新的认识。自1991年起，世界各国研究机构纷纷举办SAT竞赛，众多学者的研究热情空前高涨，SAT算法及其实现程序的求解效率大幅提高，SAT问题逐渐在许多实际应用中显现出强大的作用。SAT协会是目前推动SAT问题理论和应用进展的主要驱动力量，其Satlive网站随时更新SAT研究动态，发布了一系列有关会议、竞赛、技术报告、论文、图书等信息；每年举办一次SAT理论和应用国际学术会议，目前已召开16届；SAT国际竞赛始于2002年，每隔两年或一年举办，2016年成功举行了第10届，汇聚了大批优秀的SAT求解器，影响力很大。

目前典型的SAT求解算法包括确定性算法和随机搜索算法两大类。确定性算法采取穷举和回溯思想，从理论上保证给定命题公式的可满足性，并在实例无解的情况下给出完备证明，但不适用于求解大规模的SAT问题。随机搜索算法主要基于局部搜索思想，绝大多数随机搜索算法不能判断SAT问题的不可满足性，但由于采用了启发式策略来指导搜索，在处理可满足的大规模随机类问题时，往往能比确定性算法更快得到一个解。

## 1.3 课程设计的主要研究工作

依据Davis和Putnam在1960年提出的DPLL算法来求解合取范式，并基于原始的DPLL算法提出优化方案，并对优化前后的DPLL算法进行比较，做出一些思考。将数独游戏问题转化为SAT问题，并集成到上面的求解器进行问题求解，游戏可玩，具有一定的/简单的交互性。

# 2 系统需求分析与总体设计

## 2.1 系统需求分析

基于DPLL算法设计高效SAT求解器，对给定的中小规模SAT问题实例，理论上可判定其是否满足，满足时给出对应的一组解，并统计求解时间。要求具有输入输出功能，公式解析与验证，DPLL过程，时间性能的测量，程序优化，SAT应用（数独）等需求。

## 2.2 系统总体设计

设计程序要求模块化，程序源代码进行模块化组织。主要模块包括如下：

常量，变量，结构体定义，函数声明，头文件引用（head）

主控、交互与显示模块（main，display）

CNF解析模块（cnfparser）

核心DPLL模块（solver）

二进制数独模块,包括游戏格局读取、归约、求解（binarypuzzle）

二进制数独题库（puzzlepool）

CNF测试文件（test）

根据设计问题的功能需求，系统整体设计规划流程图如图1.1所示。

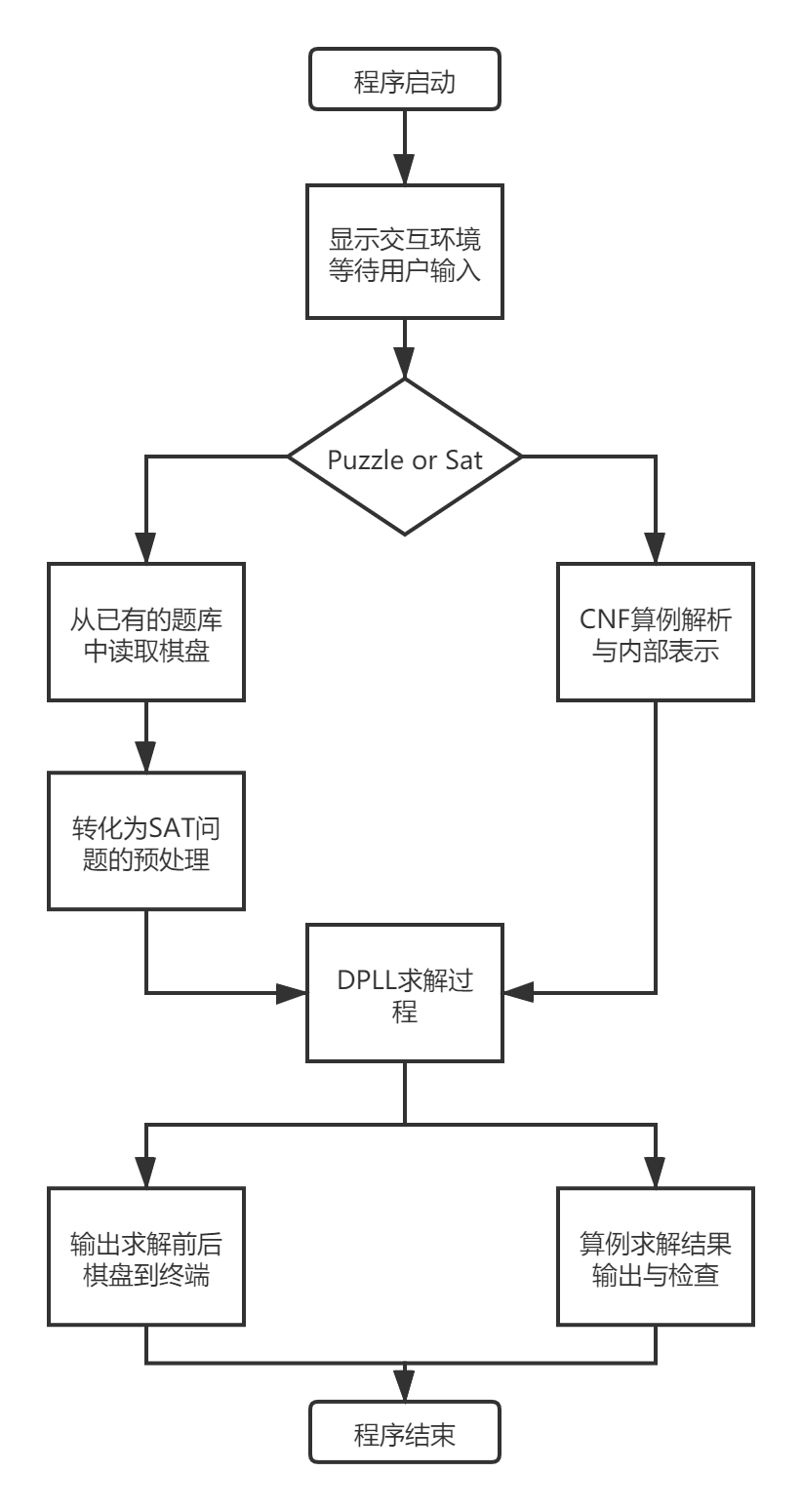


图1.1 系统总体规划流程图

# 3 系统详细设计

## 3.1 有关数据结构的定义

系统中需要处理的数据有三种：变元结点，子句结点和文字结构。

变元结点用以将子句表示为变元链表的形式，包括的数据项有：变元本身的值Var（如-13、24），指向下一变元结点的指针next\_VarNode。

子句结点包括的数据项有：指向组成子句变元结点的指针vn，指向下一个子句结点的指针next\_clauseNode。

文字结构包括的数据项有文字的值value（如1、-1、UNKNOW、NONE），决策层blevel，被赋值的次数assigned，推断标记unit\_clause，正文字相关子句数pos\_cls\_num及子句链指针pos，负文字相关子句数neg\_cls\_num及子句链指针neg。各数据间关系如图3.2所示。

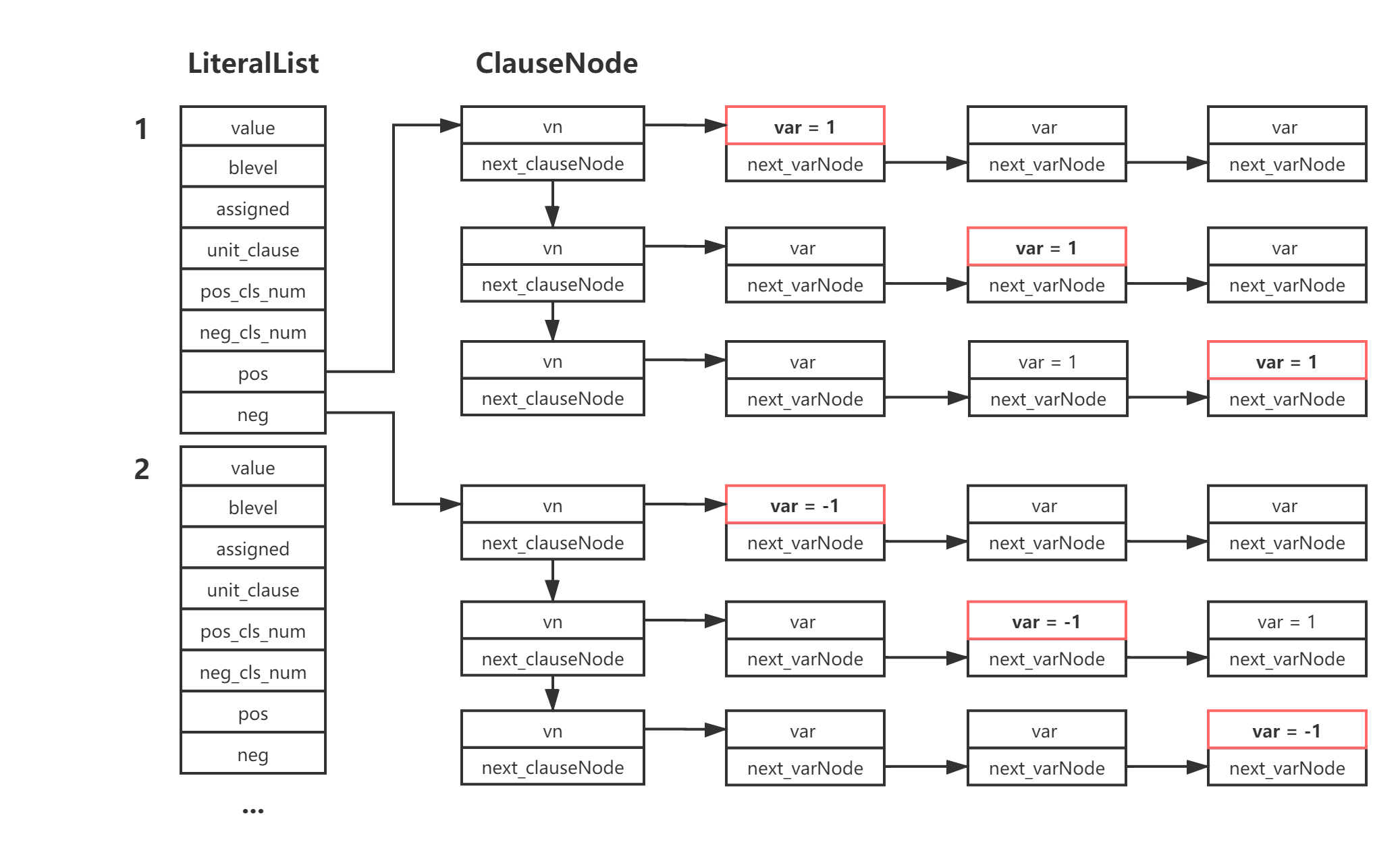


图3.1 数据结构间关系

## 3.2 主要算法设计

### 3.2.1 cnfparser

该模块主要实现将cnf算例读入内存中并按照构建好的数据结构存储。

1. 初始化内存中的文字数组。
2. 找到文件中有效信息的开始，过滤掉前面的无用信息。
3. 开始正式读取子句。每开始读取一个新的子句，创建一个子句临时结点，用于存放该子句对应的变元链，每当读取到一个变元（例如-x），就将其链接到临时结点的变元链上，并使该变元对应文字（例如x）的对应子句链指针（LiteralList[x].neg）指向该子句临时结点，对应文字相关信息更新（如文字值LiteralList[x].value由NONE变为UNKNOW，负文字相关子句数neg\_cls\_num等）。依此读取文字执行此操作直至该子句结束（读到文字0），记录该子句变元数clause\_var\_num。
4. 若该子句变元数clause\_var\_num为1，说明为单子句，则该变元对应文字值value已知，为变元值比其绝对值，即var/|var|，对应文字值（LiteralList[x].value）更新。
5. 文件未结束则重复步骤（3），结束则退出，读取结束。

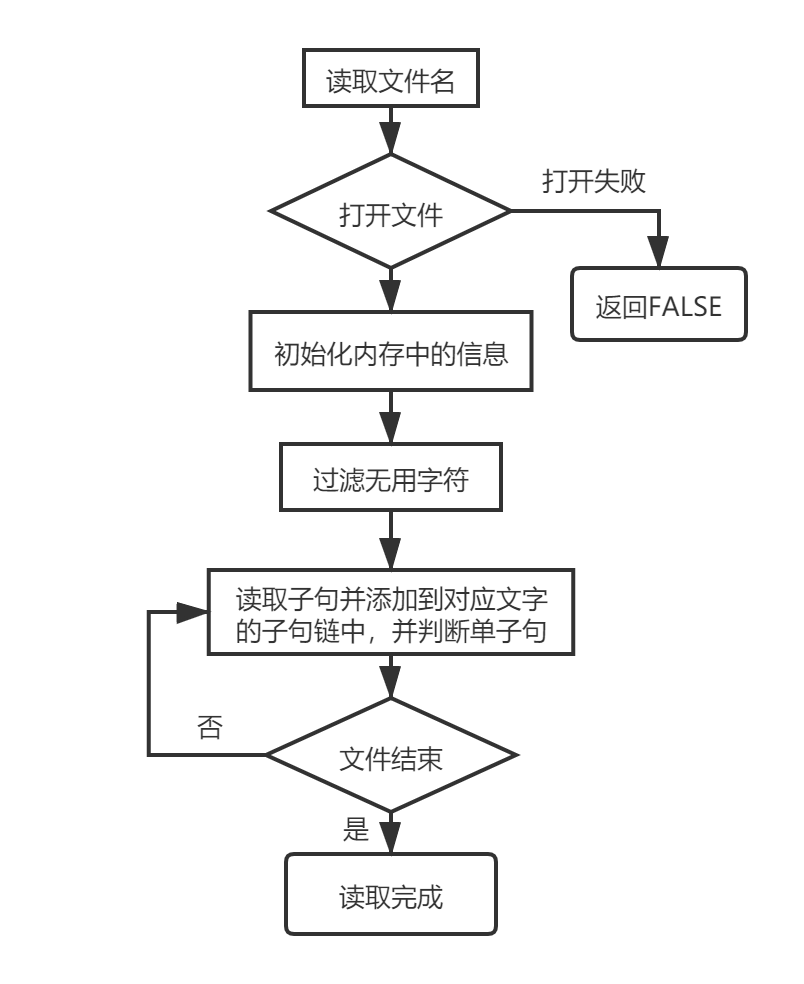


图3.2 cnfparser模块流程图

### 3.2.2 solver

该模块主要实现基于dpll过程的cnf算例求解。分支变元选择策略可选择随机选择和VSIDS策略选择。

1. 根据分支策略选择要赋值为真的变元，对应文字的信息变更（如值value，决策层blevel等）。
2. 利用单子句规则进行相应子句的推断（如选择-x为真，则推断含有变元x的子句链；选择x为真，则推断含有变元-x的子句链），推断出的变元决策级与决策变元相等，但置有标记，即LitralList[x].unit\_clause为1，用于回溯。
3. 若推断返回SATISFIABLE则求解结束；若推断返回CONFLICT则进入时序回溯，当进入回溯时决策级blevel为0则该算例无解；若推断返回OTHERS则进入下一决策级，回到步骤（1）。

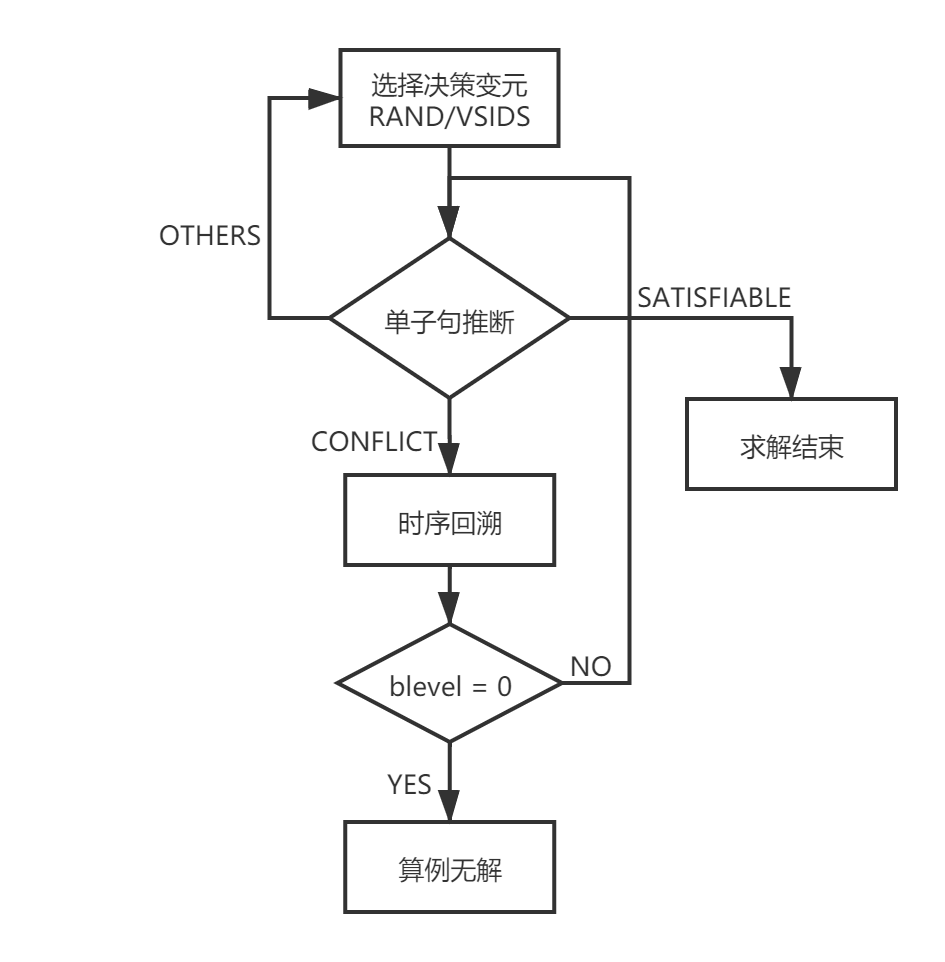


图3.3 solver模块流程图

### 3.2.3 binarypuzzle

该模块主要实现二进制数独游戏的相关操作。

1. 读取事先准备好的puzzlepool.cnf文件，用户根据操作提示选择难度和题目，分三个难度每个难度十道题。
2. 按照转换规则生成rule.cnf文件并读入内存。
3. 根据用户选择的棋盘给相应文字赋值。
4. 算例求解。

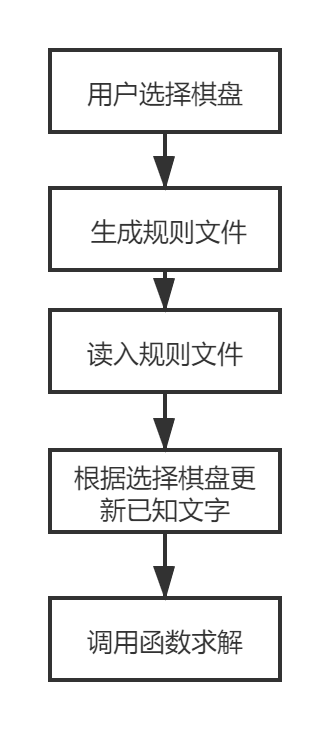


图3.4 binarypuzzle模块流程图

# 4 系统实现与测试

## 4.1 系统实现

### 4.1.1 开发环境及程序相关定义

开发环境：操作系统：Windows10家庭中文版 版本号：1909

系统类型：64位操作系统，基于x64的处理器

编译调试：Mingw-w64 GCC 8.1,GDB 8.1

编辑器：VSCode

根据3.1的设计，相关常量，变量及数据类型定义如图4.1、4.2、4.3所示。

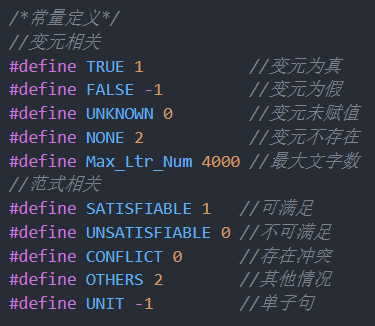


图4.1 相关常量定义

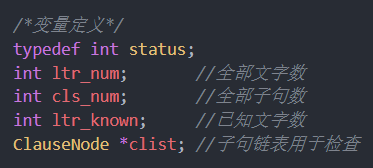


图4.2 相关变量量定义

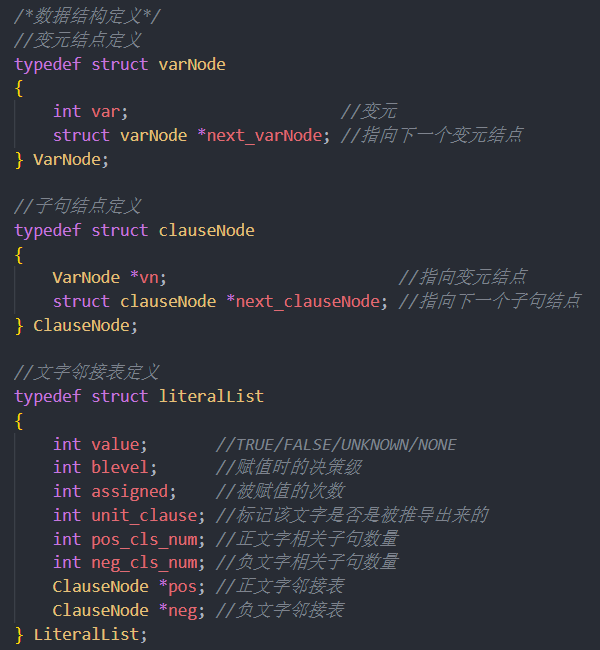


图4.3 相关数据结构定义

### 4.1.2 模块函数说明

**1. cnfparser部分函数说明**

函数声明：void add\_clause(LiteralList literals[], ClauseNode \*ctemp, int val)

函数功能：将子句添加到对应文字的邻接表。

函数说明：若val>0，则将子句ctemp添加到val对应文字的正子句链pos中，负文字相关子句数量加一；反之添加到负子句链neg中，负文字相关子句数量加一。

函数声明：void init\_cnf(LiteralList literals[])

函数功能：将子句添加到对应文字的邻接表。

函数说明：初始化全局变量已知变元数ltr\_num，扫描文字数组并逐一释放空间后初始化。

函数声明：status load\_file(LiteralList literals[], char filename[])

函数功能：读取cnf文件到内存。

函数说明：打开文件名对应文件，失败则返回FALSE；过滤无用字符后，读取文字数量ltr\_num和子句数量cls\_num，调用函数init\_cnf进行初始化；读取第一个子句的第一个变元，创建并初始化子句临时结点，当前变元不为0，进入子句创建循环，调用函数add\_clause将子句添加到对应文字的邻接表；变元为0表示当前子句结束，若该子句只有一个变元，则该变元值已知；读取下一个变元创建子句，若到文件尾，则创建完毕，返回TRUE。

**2. solver部分函数说明**

函数声明：void rand\_decide(LiteralList literals[], int \*val, int \*blevel)

函数功能：随机选择搜索分支。

函数说明：扫描文字邻接表，将第一个未赋值的变元赋值为真并返回，决策级加一。

函数声明：void VSIDS\_decide(LiteralList literals[], int \*val, int \*blevel)

函数功能：运用VSIDS策略选择决策变元。

函数说明：先进行简单的预处理：从1到ltr\_num范围内扫描文字数组，若文字不存在，即value为NONE，则已知文字数ltr\_known加一；若文字仅有相关正子句或相关负子句，则该文字值已知，赋值并且已知文字数ltr\_known加一。选择变元部分：从1到ltr\_num范围内扫描文字数组，找到相关子句数量最多的未赋值变元赋值为真，决策级blevel加一，被赋值次数assigned加一，已知文字数ltr\_known加一。

函数声明：status unit\_clause\_deduce(LiteralList literals[], ClauseNode \*\*cp, int blevel)

函数功能：检查单子句，若为单子句，通过指针返回推导出的文字影响的子句链。

函数说明：扫描该子句下的变元链，若有已知值为真的变元，停止扫描；若有未知的变元，则未知数加一并记录第一个未知的变元。扫描结束后若没有未知变元也没有值为真的变元，说明冲突，返回CONFLICT；若仅有一个未知变元且没有值为真的变元，说明为单子句，将仅有的未知变元赋值为真，推断标记unit\_clause置1，决策级blevel为当前决策级，并通过二级指针cp返回推理出的变元所影响的子句链，返回UNIT；若扫描结束后ltr\_known<ltr\_num，即仍有未知变元，则初始化二级指针cp，返回OTHERS；以上情况都不满足，说明文字全部已知，返回SATISFIABLE。

函数声明：status deduce(LiteralList literals[], ClauseNode \*root, int blevel)

函数功能；使用单子句规则进行搜索化简。

函数说明：创建一个子句结点指针栈，根结点为第一个需要推断的子句，入栈，调用函数unit\_clause\_deduce，广度搜索将所有产生单子句的所影响的子句结点入栈，直到不再产生单子句，若冲突，则返回CONFLICT，访问栈顶元素并退栈，其next\_clauseNode不为空则入栈；如此循环直到栈空。循环结束后若ltr\_known<ltr\_num，即仍有未知变元，返回OTHERS，否则返回SATISFIABLE。

函数声明：int back\_track(LiteralList literals[], int \*blevel, int val)

函数功能：时序回溯。

函数说明：首先初始化当前决策层被推理出来的文字相关信息，若当前决策变量被赋值次数为1，则返回该变量，否则回退上一决策级重复此步骤。

函数声明：status dpll(LiteralList literals[], int op)

函数功能：基于dpll算法求解

函数说明：根据用户选择的决策策略来决策变元，调用deduce函数对相关子句进行推断，推断冲突则调用back\_track函数时序回溯，若回溯后决策级为0说明无解，返回UNSATISFIABLE，否则反转回溯后决策变元的值，被赋值次数加一，重新调用deduce函数推断，若无法得出结果，则进入下一决策级，直到全部文字已知。

**3. binarypuzzle部分函数说明**

函数声明：void rule\_1(FILE \*fp)、void rule\_2(FILE \*fp)、void rule\_3(FILE \*fp)

函数功能：规则1、2、3的cnf子句生成。

函数说明：规则1，在每一行、每一列中不允许有连续的3个1或3个0出现；规则2，在每一行、每一列中1与0的个数相同；规则3，不存在重复的行与重复的列

函数声明：void rules\_generate(FILE \*fp)

函数功能：生成二进制数独规则的cnf文件。

函数说明：调用函数rule\_1、rule\_2、rule\_3生成数独规则cnf文件。

函数声明：void choose\_puzzle(LiteralList literals[])

函数功能：选择数独题目。

函数说明：用户根据操作提示选择难度和题目，在文件中找到对应题目，根据题目将已知变元赋值。

## 4.2 系统测试

### 4.2.1 DPLL部分

1. 模块功能及设计目标

基于DPLL过程实现一个高效SAT求解器，对于给定的中小规模算例进行求解，输出求解结果，统计求解时间。在此之上对基本DPLL的实现进行存储结构、分支变元选取策略等某一方面进行优化设计与实现，提供明确的性能优化率结果。

2. 测试大纲

共测试18个SAT算例，其中可满足的算例不少于15个，不满足的算例不少于3个，大中小算例各占三分之一，算例结果有解为1，无解为0。对算例规模的要求为：小型算例变元数为100个左右；中型算例变元数介于200-500个；大型算例变元数500个以上。每一种算例分别采用未优化版本和优化后版本进行求解并计时，计算优化率并记录，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。基准算例运行结果截图如下，其他算例在表1中。具体测试文件在压缩包中。

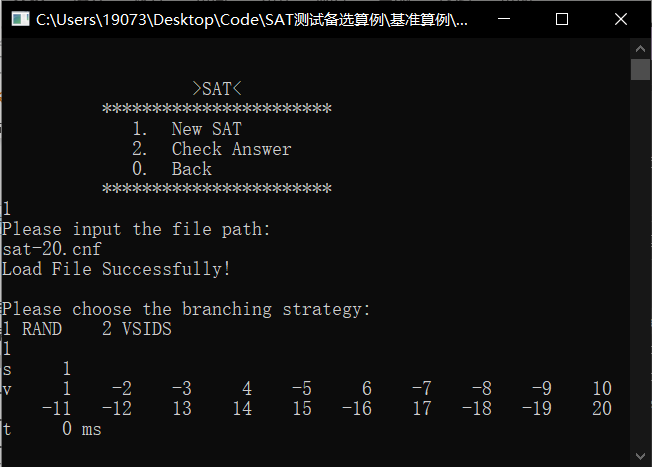


图4.4 sat-20未优化运行结果

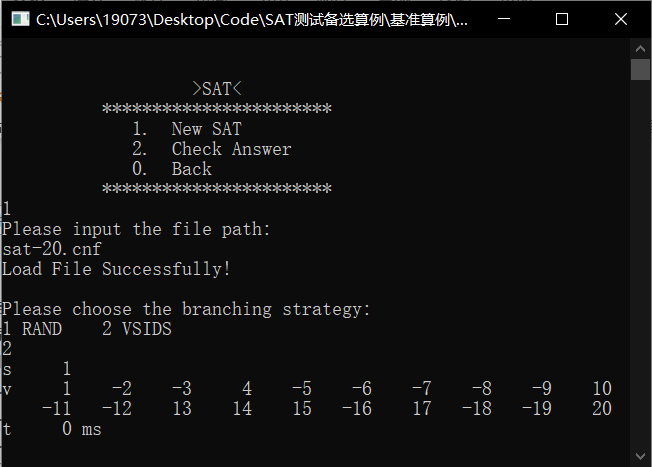


图4.5 sat-20优化后运行结果

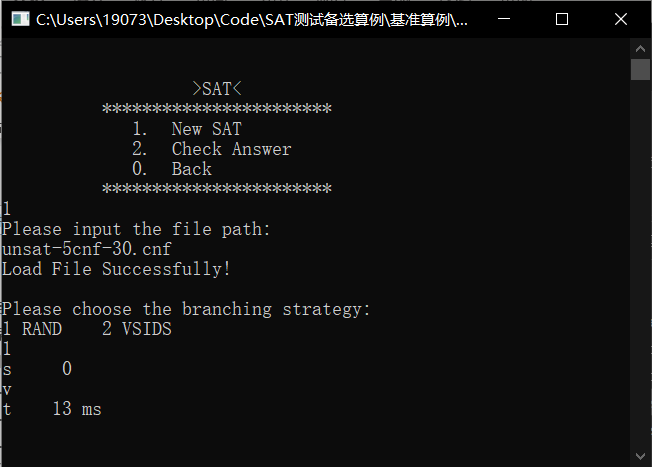


图4.6 unsat-5cnf-30未优化运行结果

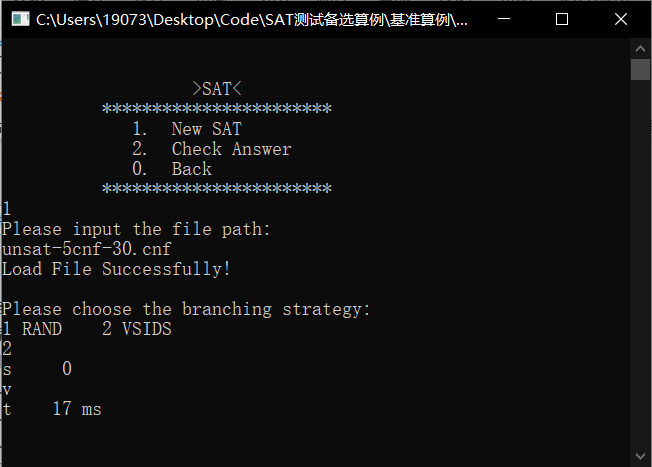


图4.7 unsat-5cnf-30优化后运行结果

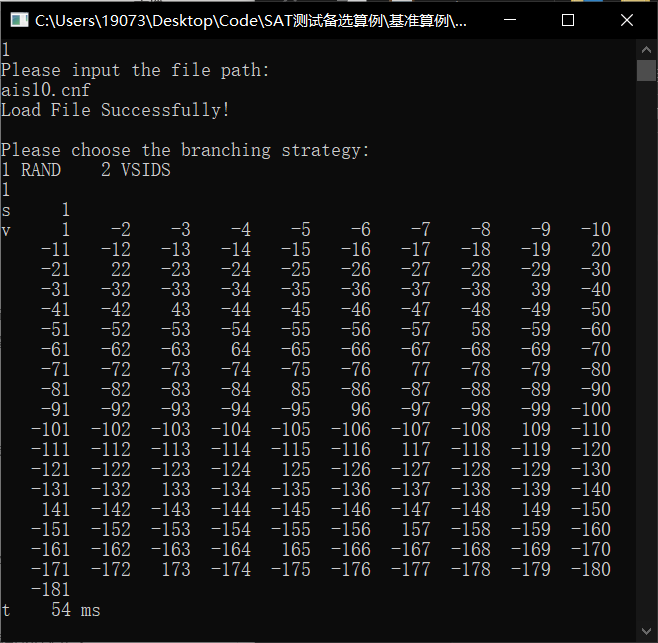


图4.8 ais10.cnf未优化运行结果

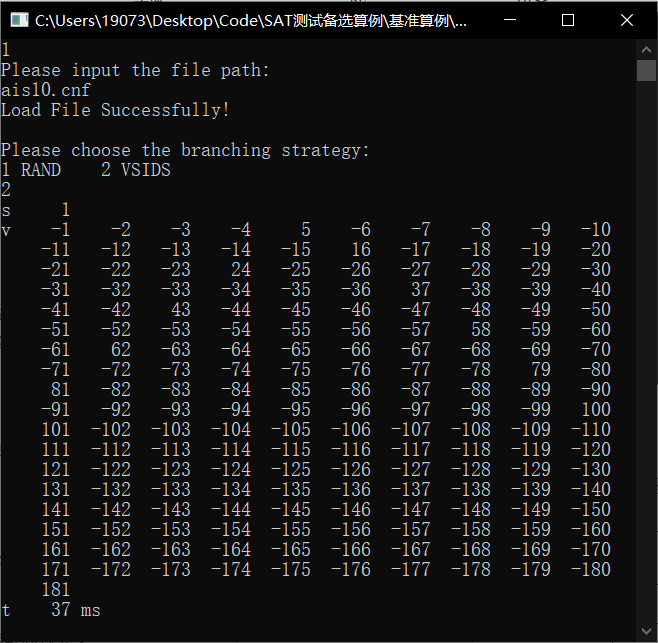


图4.9 ais10.cnf优化后运行结果

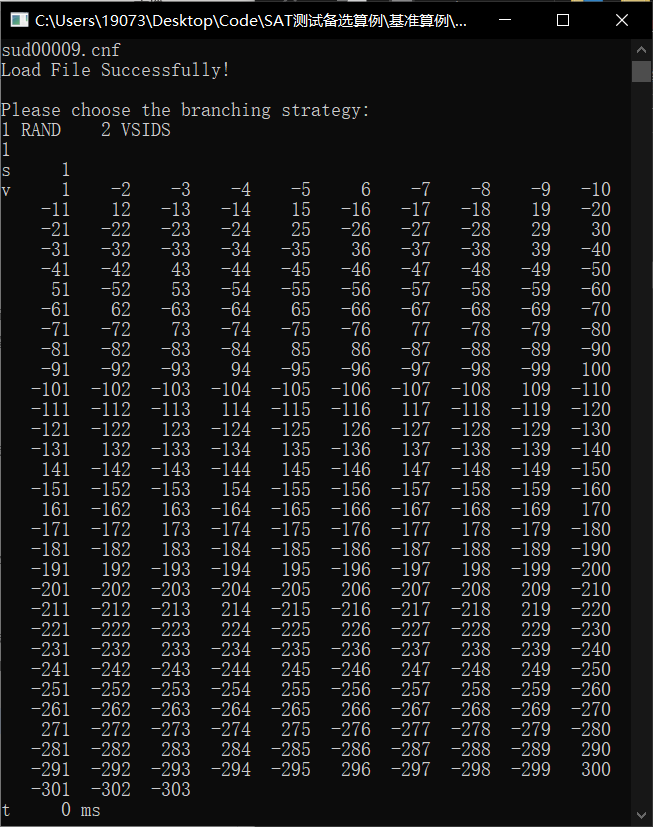


图4.10 sud00009.cnf未优化运行结果

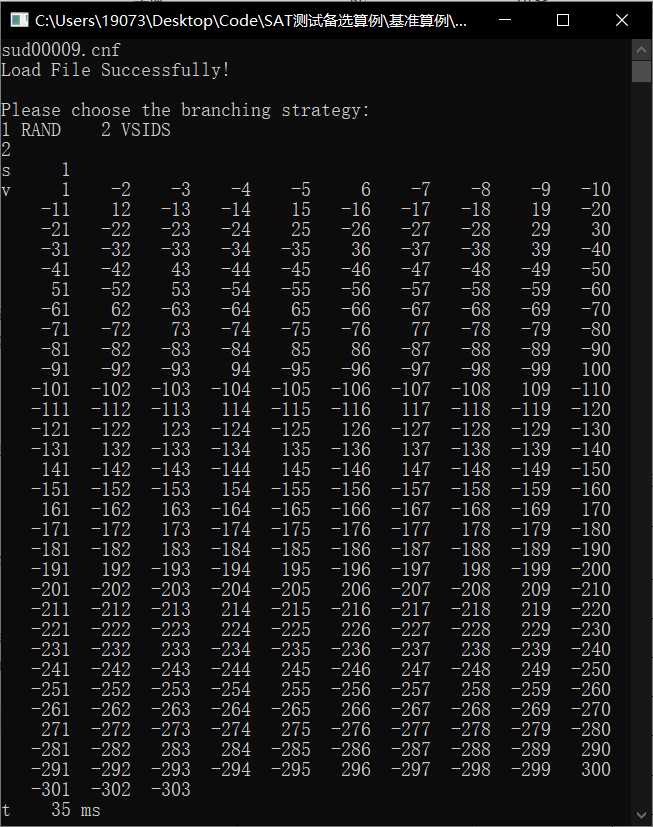


图4.11 sud00009.cnf优化后运行结果

3. 运行结果

表1 测试结果

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 算例名称 | 文字数 | 子句数 | 算例规模 | 算例结果 | t/ms | t0/ms | 优化率 |
| problem3-100 | 100 | 340 | S | 1 | 119 | 70 | 41.18% |
| problem6-50 | 50 | 100 | S | 1 | 11 | 6 | 45.45% |
| ais10 | 181 | 3151 | S | 1 | 56 | 47 | 16.07% |
| problem9-100 | 100 | 200 | S | 1 | 49242 | 3508 | 92.88% |
| problem11-100 | 100 | 600 | S | 1 | 3 | 1 | 66.67% |
| tst\_v10\_c100 | 10 | 100 | S | 0 | 0 | 0 | 00.00% |
| tst\_v200\_c210 | 200 | 210 | M | 1 | 15 | 0 | 100.00% |
| ais12 | 265 | 5666 | M | 1 | 1622 | 1087 | 32.99% |
| tst\_v200\_c220 | 200 | 220 | M | 1 | 13 | 0 | 100.00% |
| sud00082 | 224 | 1762 | M | 1 | 2 | 1 | 50.00% |
| sud00001 | 301 | 2780 | M | 1 | 19 | 9 | 52.63% |
| u-x1\_80.shuffled-238 | 238 | 634 | M | 0 | 4655 | 4511 | 03.09% |
| eh-dp04s04.shuffled-1075 | 1075 | 3152 | L | 1 | 20098 | 544 | 97.29% |
| qg7-09 | 729 | 22060 | L | 1 | 176 | 114 | 35.23% |
| bw\_large.a | 459 | 4675 | L | 1 | 11 | 2 | 81.82% |
| sw100-70 | 500 | 3100 | L | 1 | 271 | 0 | 100.00% |
| sw100-1 | 500 | 3100 | L | 1 | 2 | 0 | 100.00% |
| qg4-08 | 512 | 9685 | L | 0 | 403 | 135 | 66.50% |

4. 运行结果分析

该求解器基本可以计算全部中小型算例，仅有个别算例无法计算；优化后整体速度有明显提升，个别算例会有负优化的情况，但相差不大，基本达到该模块设计目标

### 4.2.2 数独部分

1. 模块功能及设计目标

从题库文件中读取六阶游戏棋盘，将二进制数独游戏问题转化为SAT问题，并集成到上面的求解器进行问题求解，游戏可玩，具有一定的/简单的交互性。

2. 测试大纲

题库共分三个难度，每个难度各十道题，共三十道题，每个难度取一道题截图验证结果。

3. 运行结果

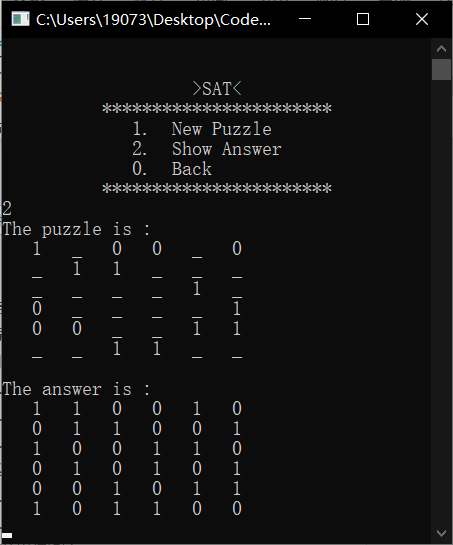
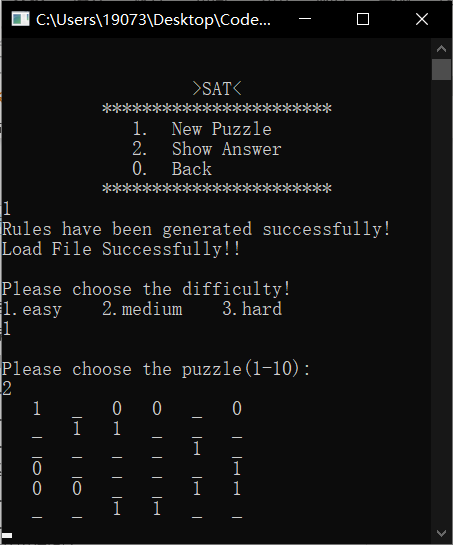


图4.12 简单难度题目运行结果

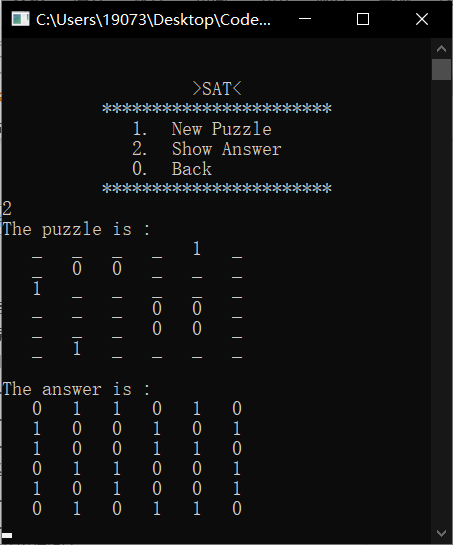
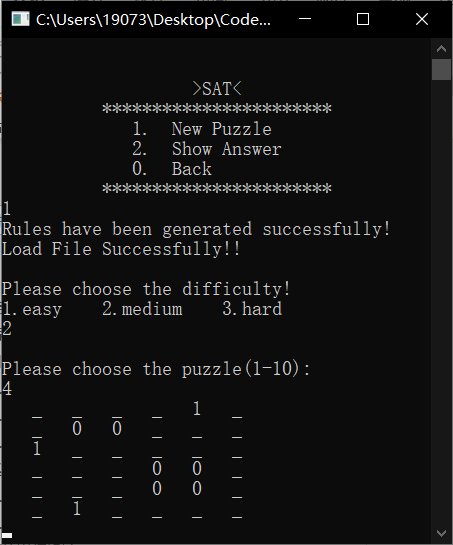


图4.13 中等难度题目运行结果

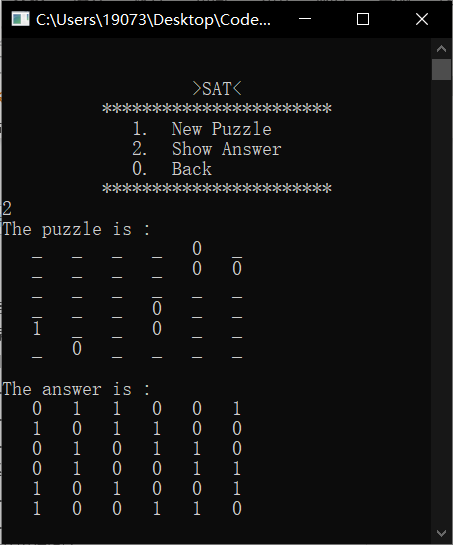
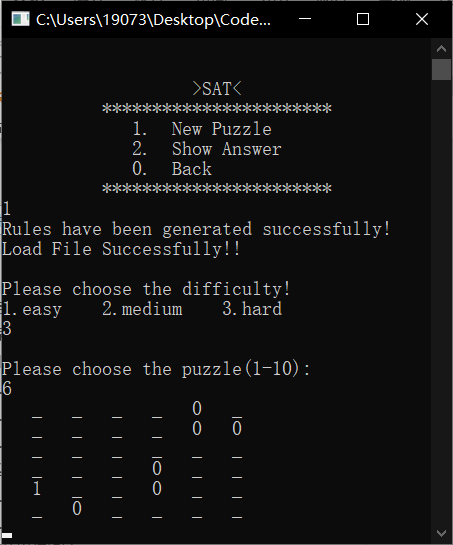


图4.14 困难难度题目运行结果

4. 运行结果分析

数独题目能够正确读取，并且能够很快的求解出答案并正确的输出，基本达到该模块设计目标。

# 5 总结与展望

## 5.1 全文总结

对自己的工作做个总结，主要工作如下：

（1）在前期准备阶段耗费时间较长，多篇论文没有集中高强度的研读下来，导致在正式写的时候依然会有很多地方不清不楚，无法形成稳定的程序框架与设计思想，降低了课设完成的效率与质量。

（2）在数据结构的选择上，试图搞懂懒惰（胀缩）数据结构的原理与实现方法，最终因不知如何具体实现与时间压力而最终放弃，选择了自行优化过的文字链表结构，发现了懂得原理与具体代码实现之间自己还有很长的一段路要走，要尽量做到懂得原理就能用代码实现。

（3）在DPLL的算法实现上选择了效率更高的迭代算法而没有选择空间复杂度和时间复杂度更大的递归算法，同样的，程序中涉及到递归思想实现的函数全部转化为了用栈去实现，在算法效率上也有了一定的提升，这一点是我认为本次任务中设计优化方面比较成功的。

（4）在选择策略的优化上最开始选择了MOM（JW）选择策略进行优化，但总体上效果一般，重新改写为VSIDS选择策略后速度有了一定程度的提升，最终决定采用VSIDS的优化策略。

（5）在回溯策略和冲突分析方面，本想采用非时序回溯与学习子句结合来对程序进一步优化，伪代码也基本完成，但碍于时间的原因放弃了这一思路和想法，是本次任务中十分可惜的一点，一旦完成理论上能够计算的算例的规模与速度都会有一定程度的提升。

（6）在数独游戏方面，采用了将规则先生成到外部文件，在调用已有读取计算的方法，省去了更为麻烦的直接写入内存；在题目生成方面，挖洞算法的思想不难理解，但因为工程量并不小与时间原因不得已选择放弃，十分遗憾。

总体上讲，本次程序综合设计完成度属于中等，完成了基本要求，但时间分配的不合理，设计调试的低效率导致了本次课设还有很大的提升空间，对我自己而言并不是很满意在这次课设上的表现，发现了很多自己不足的地方，在以后的学习道路上一定会加以改进！

## 5.2 工作展望

在今后的研究中，围绕着如下几个方面开展工作：

（1）技术相关文献的阅读，尤其是英文文献的阅读，学会概括总结，提炼精华。

（2）算法思想与实际具体实现的转化，最好可以做到思想理解了就能够用代码具体实现出来。

（3）代码编写的规范性，如命名规则的一致性，命名的合理性，关键语句的清晰注释等等，有助于代码的可读与理解

（4）工程设计的模块化思想，模块化的好处是程序层次清晰易懂，修改时也很少存在牵一发而动全身的状况出现，只需修改对应模块的功能就可以。

# 6 体会

对于计算机科学与技术专业的我们，在前三个学期学习了C语言程序设计，数据结构两门面向编程知识与技术的基础理论课，以及C语言程序设计实验、数据结构实验两门编程时间课程，不仅具有较为系统性的C语言、常用数据结构基本知识，而且具有初步的程序设计、数据抽象与建模、问题求解与算法设计的能力，奠定了进行复杂程序设计的知识基础。但两门实验课仍属于对基本编程模型与技术的验证性训练，而“程序设计”综合课程设计正是使我们从简单验证到综合应用，甚至在编程中实现智慧与风格升华的重要实践环节。

通过本次的综合课程设计，我意识到，语言就好比砖块，你学的再精通，也需要会建筑的人来把它垒起来，而这个会建筑的人，就是数据结构，数据结构的好坏就像是一栋大楼的地基，地基越稳，楼才可以建得越高，他决定了楼高的上限，而楼究竟能不能建到那么高，仅仅取决于建筑工程中的每一个细节，也就是算法，算法的好坏决定了你是否能最大的发挥当前数据结构的优势。一个好的数据结构加上一整套适合这个结构的算法，最终才能构成一个优秀的程序。

自己了解的越多，就会越觉得自己的无知与渺小，前进的空间一望无际，希望以后的自己一天比一天优秀。

# 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] 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

[3] 赵伟楠 对可满足性(SAT)问题求全解的算法研究及实现 硕士学位论文，北京交通大学，2009

[4] 刘歆 SAT数据结构与组合测试生成 华中科技大学 微电子与计算机2003年第5期

[5] 熊伟 可满足性问题DPLL算法研究 硕士学位论文，复旦大学，2007

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

[7] 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.

[8] Binary Puzzle：http://www.binarypuzzle.com/

# 附录A（head.h）

/\*\*

\* 文件名称：head.h

\* 文件描述：SAT求解器相关常量，数据结构定义，相关操作函数声明

\*/

#ifndef \_HEAD\_H

#define \_HEAD\_H

/\*头文件引用\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#include <math.h>

/\*常量定义\*/

//变元相关

#define TRUE 1 //变元为真

#define FALSE -1 //变元为假

#define UNKNOWN 0 //变元未赋值

#define NONE 2 //变元不存在

#define Max\_Ltr\_Num 4000 //最大文字数

//范式相关

#define SATISFIABLE 1 //可满足

#define UNSATISFIABLE 0 //不可满足

#define CONFLICT 0 //存在冲突

#define OTHERS 2 //其他情况

#define UNIT -1 //单子句

/\*数据结构定义\*/

//变元结点定义

typedef struct varNode

{

int var; //变元

struct varNode \*next\_varNode; //指向下一个变元结点

} VarNode;

//子句结点定义

typedef struct clauseNode

{

VarNode \*vn; //指向变元结点

struct clauseNode \*next\_clauseNode; //指向下一个子句结点

} ClauseNode;

//文字邻接表定义

typedef struct literalList

{

int value; //TRUE/FALSE/UNKNOWN/NONE

int blevel; //赋值时的决策级

int assigned; //被赋值的次数

int unit\_clause; //标记该文字是否是被推导出来的

int pos\_cls\_num; //正文字相关子句数量

int neg\_cls\_num; //负文字相关子句数量

ClauseNode \*pos; //正文字邻接表

ClauseNode \*neg; //负文字邻接表

} LiteralList;

/\*变量定义\*/

typedef int status;

int ltr\_num; //全部文字数

int cls\_num; //全部子句数

int ltr\_known; //已知文字数

ClauseNode \*clist; //子句链表用于检查

/\*函数声明\*/

//cnfparser.c中函数声明

void add\_clause(LiteralList literals[], ClauseNode \*ctemp, int val);

void init\_cnf(LiteralList literals[]);

status load\_file(LiteralList literals[], char filename[]);

//solver.c中函数声明

void rand\_decide(LiteralList literals[], int \*val, int \*blevel);

void VSIDS\_decide(LiteralList literals[], int \*val, int \*blevel);

status deduce(LiteralList literals[], ClauseNode \*root, int blevel);

status unit\_clause\_deduce(LiteralList literals[], ClauseNode \*\*cp, int blevel);

int back\_track(LiteralList literals[], int \*blevel, int val);

status dpll(LiteralList literals[], int op);

//diplay.c中函数声明

void show\_answer(LiteralList literals[], clock\_t cost, int result, char filename[]);

void check\_answer(LiteralList literals[]);

void sat();

void puzzle();

//binary\_puzzle.c中函数声明

void rules\_generate(FILE \*fp);

void rule\_1(FILE \*fp);

void rule\_2(FILE \*fp);

void rule\_3(FILE \*fp);

void show\_puzzle(LiteralList literalList[]);

void choose\_puzzle(LiteralList literalList[]);

#endif

# 附录B（main.c）

/\*\*

\* 文件名称：main.c

\* 文件描述：主界面的显示，函数调用

\*/

#include "head.h"

int main()

{

int op = 1;

while (op)

{

system("cls");

printf("\n\n");

printf("%20cSAT<\n", '>');

printf("%10c\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n", ' ');

printf("%16c1. SAT\n", ' ');

printf("%16c2. Puzzle\n", ' ');

printf("%16c0. Exit\n", ' ');

printf("%10c\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n", ' ');

scanf("%d", &op);

switch (op)

{

case 1:

sat();

break;

case 2:

puzzle();

break;

case 0:

break;

default:

break;

}

}

return 0;

}

# 附录C（cnfparser.c）

/\*\*

\* 文件名称：cnf.c

\* 文件描述：CNF相关操作函数定义

\*/

#include "head.h"

/\*\*

\* 函数名称：load\_file

\* 函数功能：读取cnf文件到内存

\* 返回值：TRUE/FALSE

\*/

status load\_file(LiteralList literals[], char filename[])

{

VarNode \*vtemp = NULL, \*vp = NULL; //变元结点操作指针

ClauseNode \*ctemp = NULL, \*cp = NULL; //子句操作指针

FILE \*fp = NULL;

int val = 0; //变元值

int clause\_var\_num = 0; //每个字句的变元数

char string[20];

//打开文件失败则返回

if ((fp = fopen(filename, "r")) == NULL)

{

return FALSE;

}

//过滤无用字符

while (1)

{

fscanf(fp, "%s", string);

if (strcmp(string, "cnf") == 0) //读到cnf退出循环

break;

}

fscanf(fp, "%d", &ltr\_num); //读取文字数量

fscanf(fp, "%d", &cls\_num); //读取子句数量

init\_cnf(literals); //初始化

fscanf(fp, "%d", &val); //读取第一个变元

while (1)

{

//初始化子句临时结点

ctemp = (ClauseNode \*)malloc(sizeof(ClauseNode));

ctemp->vn = NULL;

ctemp->next\_clauseNode = NULL;

//创建子句链表用于检查

if (clist == NULL)

{

clist = ctemp;

}

else

{

cp = clist;

while (cp->next\_clauseNode != NULL)

{

cp = cp->next\_clauseNode;

}

cp->next\_clauseNode = ctemp;

}

clause\_var\_num = 0; //初始化当前子句变元数

//创建子句

while (val)

{

clause\_var\_num++; //子句变元数加一

//文字值状态更改

if (literals[abs(val)].value == NONE)

{

literals[abs(val)].value = UNKNOWN;

}

//初始化变元临时结点

vtemp = (VarNode \*)malloc(sizeof(VarNode));

vtemp->var = val; //结点赋值

vtemp->next\_varNode = NULL;

if (clause\_var\_num == 1)

{

//存储子句中首个文字

vp = vtemp;

ctemp->vn = vp;

}

else

{

//存储子句中非首个文字

vp->next\_varNode = vtemp;

vp = vp->next\_varNode;

}

add\_clause(literals, ctemp, val); //将子句添加到对应文字的邻接表

fscanf(fp, "%d", &val);

}

//该子句为单子句，则该变元为真，该文字值已知

if (clause\_var\_num == 1)

{

literals[abs(vp->var)].value = vp->var / abs(vp->var);

ltr\_known++;

}

fscanf(fp, "%d", &val); //读取下一个变元

//读取结束

if (feof(fp))

{

break;

}

}

return TRUE;

}

/\*\*

\* 函数名称：InitCnf

\* 函数功能：初始化cnf相关结构

\* 返回值：void

\*/

void init\_cnf(LiteralList literals[])

{

ltr\_known = 0; //初始化已知变元数

//子句链空间释放鸡初始化

ClauseNode \*cp = NULL;

while (clist != NULL)

{

cp = clist->next\_clauseNode;

free(clist);

clist = cp;

}

//文字数组初始化

for (int i = 1; i <= ltr\_num; i++)

{

//文字相关信息初始化

literals[i].value = NONE;

literals[i].assigned = 0;

literals[i].blevel = 0;

literals[i].unit\_clause = 0;

literals[i].pos\_cls\_num = 0;

literals[i].neg\_cls\_num = 0;

//释放空间

if (literals[i].pos != NULL)

{

free(literals[i].pos);

literals[i].pos = NULL;

}

if (literals[i].neg != NULL)

{

free(literals[i].neg);

literals[i].neg = NULL;

}

//正文字头结点初始化

literals[i].pos = (ClauseNode \*)malloc(sizeof(ClauseNode));

literals[i].pos->vn = NULL;

literals[i].pos->next\_clauseNode = NULL;

//负文字头结点初始化

literals[i].neg = (ClauseNode \*)malloc(sizeof(ClauseNode));

literals[i].neg->vn = NULL;

literals[i].neg->next\_clauseNode = NULL;

}

}

/\*\*

\* 函数名称：add\_clause

\* 函数功能：将子句添加到对应文字的邻接表

\* 返回值：void

\*/

void add\_clause(LiteralList literals[], ClauseNode \*ctemp, int val)

{

ClauseNode \*cp = NULL;

//按正负文字分类该文字对应子句

if (val > 0)

{

cp = literals[val].pos;

literals[val].pos\_cls\_num++;

}

else

{

cp = literals[-val].neg;

literals[-val].neg\_cls\_num++;

}

//找到最后一个子句结点

while (cp->next\_clauseNode)

{

cp = cp->next\_clauseNode;

}

//创建新子句结点

cp->next\_clauseNode = (ClauseNode \*)malloc(sizeof(ClauseNode));

cp = cp->next\_clauseNode;

//新结点赋值

cp->vn = ctemp->vn;

cp->next\_clauseNode = NULL;

}

# 附录D（solver.c）

/\*\*

\* 文件名称：solver.c

\* 文件描述：dpll求解过程相关函数定义

\*/

#include "head.h"

/\*\*

\* 函数名称：dpll

\* 函数功能：dpll算法求解

\* 返回值：SATISFIABLE/UNSATISFIABLE

\*/

status dpll(LiteralList literals[], int op)

{

int status = 0, val = 0, blevel = 0;

ClauseNode \*cp = NULL;

//DPLL求解过程

while (1)

{

//分支决策

switch (op)

{

case 1:

//随机分支选择，返回选择为真的变元

rand\_decide(literals, &val, &blevel);

break;

case 2:

//VSIDS分支选择，返回选择为真的变元

VSIDS\_decide(literals, &val, &blevel);

break;

default:

break;

}

while (TRUE)

{

//指向受影响的子句链

if (val > 0)

{

//正文字为真，指向负文字子句链

cp = literals[val].neg->next\_clauseNode;

}

else

{

//负文字为真，指向正文字子句链

cp = literals[-val].pos->next\_clauseNode;

}

status = deduce(literals, cp, blevel); //BCP传播

if (status == SATISFIABLE)

{

return SATISFIABLE;

}

else if (status == CONFLICT)

{

//冲突回溯,返回需要更改决策的变元

val = back\_track(literals, &blevel, val);

if (blevel == 0)

{

//回溯至决策级0，则问题无解

return UNSATISFIABLE;

}

else

{

//反转决策变元的值，被赋值次数加一

literals[val].value \*= -1;

literals[val].assigned++;

if (literals[val].value < 0)

{

val \*= -1;

}

}

}

else if (status == OTHERS)

{

//无法得出结论，继续选择决策变元

break;

}

}

}

}

/\*\*

\* 函数名称：rand\_decide

\* 函数功能：随机选择搜索分支

\* 返回值：void

\*/

void rand\_decide(LiteralList literals[], int \*val, int \*blevel)

{

//扫描文字邻接表，将第一个未赋值的变元赋值为真并返回

for (int i = 1; i <= ltr\_num; i++)

{

if (literals[i].value == UNKNOWN)

{

//决策级，被赋值次数，已知文字数加一

\*val = i;

(\*blevel)++;

literals[\*val].value = 1;

literals[\*val].blevel = \*blevel;

literals[\*val].assigned++;

ltr\_known++;

return;

}

}

}

/\*\*

\* 函数名称：VSIDS\_decide

\* 函数功能：选择搜索分支

\* 返回值：void

\*/

void VSIDS\_decide(LiteralList literals[], int \*val, int \*blevel)

{

int temp = 0;

int var = 0;

//找到子句最多的变元

for (int i = 1; i <= ltr\_num; i++)

{

//简单的预处理

if (\*blevel == 0)

{

if (literals[i].value == NONE)

{

//范围内变元不存在，已知变元数加一，防止求解死循环

ltr\_known++;

continue;

}

else if (literals[i].pos\_cls\_num == 0 || literals[i].neg\_cls\_num == 0)

{

//将只有正子句或负子句存在的文字赋值

if (literals[i].neg\_cls\_num == 0)

{

//负文字对应子句数为0，则该文字值必为正

literals[i].value = 1;

}

else

{

//正文字对应子句数为0，则该文字值必为负

literals[i].value = -1;

}

ltr\_known++;

continue;

}

}

if (literals[i].value != UNKNOWN)

{

//该文字值已知，跳过本次循环

continue;

}

//暂存子句最多的文字

if (literals[i].pos\_cls\_num >= temp)

{

temp = literals[i].pos\_cls\_num;

var = i;

}

if (literals[i].neg\_cls\_num >= temp)

{

temp = literals[i].neg\_cls\_num;

var = -i;

}

}

//决策变元赋值，决策级，赋值次数，已知文字数加一

\*val = var;

(\*blevel)++;

literals[abs(\*val)].value = abs(\*val) / (\*val);

literals[abs(\*val)].blevel = \*blevel;

literals[abs(\*val)].assigned++;

ltr\_known++;

}

/\*\*

\* 函数名称：deduce

\* 函数功能；使用单子句规则进行搜索化简

\* 返回值；SATISIABLE/CONFLICT/OTHERS

\*/

status deduce(LiteralList literals[], ClauseNode \*root, int blevel)

{

ClauseNode \*stack[Max\_Ltr\_Num]; //子句链指针栈

ClauseNode \*cp = NULL;

int top = 0; //栈顶指针

int status;

//根结点进栈

stack[top] = root;

top++;

//栈空退出循环

while (top)

{

cp = stack[top - 1]; //访问栈顶元素

status = UNIT;

//广度搜索进栈

while (status == UNIT && cp != NULL)

{

status = unit\_clause\_deduce(literals, &cp, blevel);

stack[top] = cp;

top++;

}

top--; //空指针退栈

if (status == CONFLICT)

return CONFLICT;

//栈不为空，深度搜索进栈

if (top)

{

//访问栈顶元素并退栈

top--;

cp = stack[top];

//深度搜索，不为空则进栈

if (cp->next\_clauseNode)

{

stack[top] = cp->next\_clauseNode;

top++;

}

}

}

if (ltr\_known < ltr\_num)

{

return OTHERS;

}

else

{

return SATISFIABLE;

}

}

/\*\*

\* 函数名称：unit\_clause\_deduce

\* 函数功能：检查单子句，若为单子句，返回推导出的文字影响的子句链

\* 返回值：SATISFIABLE/CONFLICT/UNIT/OTHERS

\*/

status unit\_clause\_deduce(LiteralList literals[], ClauseNode \*\*cp, int blevel)

{

VarNode \*vp = (\*cp)->vn;

int unknown\_num = 0, first\_unknown = 0, satisfiable = 0;

//扫描该子句下的变元

while (vp)

{

//该子句有值为真的变元

if (literals[abs(vp->var)].value \* vp->var > 0)

{

satisfiable = 1;

break;

}

//该子句有未赋值的变元

if (literals[abs(vp->var)].value == UNKNOWN)

{

//未知文字数加一

unknown\_num++;

//记录第一个未知的变元

if (first\_unknown == 0)

{

first\_unknown = vp->var;

}

}

vp = vp->next\_varNode;

}

//该子句无未知变元且子句不满足

if (unknown\_num == 0 && satisfiable == 0)

{

return CONFLICT;

}

//该子句仅有一个未知变元且子句不满足（单子句）

if (unknown\_num == 1 && satisfiable == 0)

{

//推理出的变元赋值

literals[abs(first\_unknown)].unit\_clause = 1;

literals[abs(first\_unknown)].value = first\_unknown / abs(first\_unknown);

literals[abs(first\_unknown)].blevel = blevel;

ltr\_known++;

//返回推理出的变元所影响的子句链

if (first\_unknown > 0)

{

\*cp = literals[first\_unknown].neg->next\_clauseNode;

}

else

{

\*cp = literals[-first\_unknown].pos->next\_clauseNode;

}

return UNIT;

}

//推理结束且仍有未知变元

if (ltr\_known < ltr\_num)

{

\*cp = NULL; //返回空，选择下一分支

return OTHERS;

}

return SATISFIABLE;

}

/\*\*

\* 函数名称：back\_track

\* 函数功能：回溯

\* 返回值：需要反转的决策变量

\*/

int back\_track(LiteralList literals[], int \*blevel, int val)

{

int parent = abs(val);

//初始化当前决策级被推理出的文字的相关参数

for (int i = 1; i <= ltr\_num; i++)

{

if (i != parent && literals[i].blevel == \*blevel)

{

literals[i].value = UNKNOWN;

literals[i].blevel = 0;

literals[i].assigned = 0;

literals[i].unit\_clause = 0;

ltr\_known--;

}

}

//决策变元被赋值次数为2，初始化该变元，回退上一决策级

while (\*blevel != 1 && literals[parent].assigned == 2)

{

(\*blevel)--;

literals[parent].value = UNKNOWN;

literals[parent].blevel = 0;

literals[parent].assigned = 0;

ltr\_known--;

//找到上一级的决策变元,初始化上一决策级推理出的文字信息

for (int j = 1; j <= ltr\_num; j++)

{

//找到上一级的决策变元

if (literals[j].blevel == \*blevel && literals[j].unit\_clause == 0)

{

parent = j;

}

//初始化上一决策级被推理出的文字的相关参数

if (j != parent && literals[j].blevel == \*blevel)

{

literals[j].value = UNKNOWN;

literals[j].blevel = 0;

literals[j].assigned = 0;

literals[j].unit\_clause = 0;

ltr\_known--;

}

}

}

//回退到的决策级是1且被赋值次数为2

if (\*blevel == 1 && literals[parent].assigned == 2)

{

//决策级减一变为0，无解

(\*blevel)--;

}

//返回要回退到的决策级对应的决策变量

return parent;

}

# 附录E（binarypuzzle.c）

/\*\*

\* 文件名称：binary\_puzzle.c

\* 文件描述：二进制数独模块,包括游戏格局生成、归约、求解

\*/

#include "head.h"

/\*\*

\* 函数名称：rules\_generate

\* 函数功能：生成二进制数独规则的cnf文件

\* 返回值：void

\*/

void rules\_generate(FILE \*fp)

{

char filename[] = "puzzle.cnf";

fp = fopen(filename, "w");

fprintf(fp, "p cnf 30000 30000\n");

rule\_1(fp);

rule\_2(fp);

rule\_3(fp);

fclose(fp);

}

/\*\*

\* 函数名称：rule\_1

\* 函数功能：规则1子句生成（在每一行、每一列中不允许有连续的3个1或3个0出现）

\* 返回值：void

\*/

void rule\_1(FILE \*fp)

{

int var = 0;

//行生成

for (int i = 1; i <= 6; i++)

{

for (int j = 1; j <= 4; j++)

{

var = 10 \* i + j;

fprintf(fp, "%d %d %d 0\n", var, var + 1, var + 2);

fprintf(fp, "%d %d %d 0\n", -var, -(var + 1), -(var + 2));

}

}

//列生成

for (int j = 1; j <= 6; j++)

{

for (int i = 1; i <= 4; i++)

{

var = 10 \* i + j;

fprintf(fp, "%d %d %d 0\n", var, var + 10, var + 20);

fprintf(fp, "%d %d %d 0\n", -var, -(var + 10), -(var + 20));

}

}

}

/\*\*

\* 函数名称：rule\_2

\* 函数功能：规则2子句生成（在每一行、每一列中1与0的个数相同）

\* 返回值：void

\*/

void rule\_2(FILE \*fp)

{

int j = 0;

int var = 0;

int exhaustion[] = {1, 2, 3, 4, 0, 1, 2, 3, 5, 0, 1, 2, 3, 6, 0, 1, 2, 4, 5, 0, 1, 2, 4, 6, 0, 1, 2, 6, 5, 0, 1, 3, 4, 5, 0, 1, 3, 4, 6, 0, 1, 3, 5, 6, 0, 1, 4, 5, 6, 0, 2, 3, 4, 5, 0, 2, 3, 4, 6, 0, 2, 3, 5, 6, 0, 2, 4, 5, 6, 0, 3, 4, 5, 6, -1};

//行生成

for (int i = 1; i <= 6; i++)

{

//任选4个至少有一个1

while (exhaustion[j] != -1)

{

var = 10 \* i + exhaustion[j];

if (exhaustion[j] == 0)

{

//数组到零换行

fprintf(fp, "0\n");

}

else

{

//写入文件

fprintf(fp, "%d ", var);

}

j++;

}

fprintf(fp, "0\n");

j = 0;

//任选4个至少有一个0

while (exhaustion[j] != -1)

{

var = 10 \* i + exhaustion[j];

if (exhaustion[j] == 0)

{

//数组到零换行

fprintf(fp, "0\n");

}

else

{

//写入文件

fprintf(fp, "%d ", -var);

}

j++;

}

fprintf(fp, "0\n");

j = 0;

}

//列生成

for (int i = 1; i <= 6; i++)

{

//任选4个至少有一个1

while (exhaustion[j] != -1)

{

var = i + 10 \* exhaustion[j];

if (exhaustion[j] == 0)

{

//数组到零换行

fprintf(fp, "0\n");

}

else

{

//写入文件

fprintf(fp, "%d ", var);

}

j++;

}

fprintf(fp, "0\n");

j = 0;

//任选4个至少有一个0

while (exhaustion[j] != -1)

{

var = i + 10 \* exhaustion[j];

if (exhaustion[j] == 0)

{

//数组到零换行

fprintf(fp, "0\n");

}

else

{

//写入文件

fprintf(fp, "%d ", -var);

}

j++;

}

fprintf(fp, "0\n");

j = 0;

}

}

/\*\*

\* 函数名称：rule\_3

\* 函数功能：规则3子句生成（不存在重复的行与重复的列）

\* 返回值：void

\*/

void rule\_3(FILE \*fp)

{

int var1 = 0, var2 = 0; //31 51

//附加变元存储数组

int pos\_addition[7] = {'\0'}; //13511...13561

int neg\_addition[7] = {'\0'}; //13510...13560

int addition[7] = {'\0'}; //1351...1356

//行生成

for (int std\_line = 1; std\_line <= 6; std\_line++) //基准行

{

for (int cmp\_line = std\_line + 1; cmp\_line <= 6; cmp\_line++) //比较行

{

//两行的每一列比较

for (int column = 1; column <= 6; column++)

{

pos\_addition[column] = 10000 + std\_line \* 1000 + cmp\_line \* 100 + column \* 10 + 1; //13511

neg\_addition[column] = 10000 + std\_line \* 1000 + cmp\_line \* 100 + column \* 10; //13510

addition[column] = 1000 + std\_line \* 100 + cmp\_line \* 10 + column; //1351

var1 = std\_line \* 10 + column; //31

var2 = cmp\_line \* 10 + column; //51

//13511= 31∧51转化为CNF生成3个子句：31∨¬13511；51∨¬13511；¬31∨¬51∨13511

fprintf(fp, "%d %d 0\n", var1, -pos\_addition[column]);

fprintf(fp, "%d %d 0\n", var2, -pos\_addition[column]);

fprintf(fp, "%d %d %d 0\n", -var1, -var2, pos\_addition[column]);

//13510= ¬31∧¬51转化为CNF生成3个子句：¬31∨¬13510；¬51∨¬13510；31∨51∨13510

fprintf(fp, "%d %d 0\n", -var1, -neg\_addition[column]);

fprintf(fp, "%d %d 0\n", -var2, -neg\_addition[column]);

fprintf(fp, "%d %d %d 0\n", var1, var2, neg\_addition[column]);

//1351= 13511∨13510转化为CNF生成3个子句：1351∨¬13511；1351∨¬13510；13511∨13510∨¬1351

fprintf(fp, "%d %d 0\n", addition[column], -pos\_addition[column]);

fprintf(fp, "%d %d 0\n", addition[column], -neg\_addition[column]);

fprintf(fp, "%d %d %d 0\n", pos\_addition[column], neg\_addition[column], -addition[column]);

}

//135= ¬[1351∧1352∧…∧1356]转化为CNF生成7个子句：（¬135∨¬1351∨¬1352∨…∨¬1356）∧（1351∨135）∧（1352∨135）…（1356∨135）

int var3 = 100 + std\_line \* 10 + cmp\_line; //135

//（¬135∨¬1351∨¬1352∨…∨¬1356）

fprintf(fp, "%d %d %d %d %d %d %d 0\n", -var3, -addition[1], -addition[2], -addition[3], -addition[4], -addition[5], -addition[6]);

//（1351∨135）∧（1352∨135）…（1356∨135）

for (int i = 1; i <= 6; i++)

{

fprintf(fp, "%d %d 0\n", var3, addition[i]);

}

}

}

//列生成

for (int std\_col = 1; std\_col <= 6; std\_col++) //基准列

{

for (int cmp\_col = std\_col + 1; cmp\_col <= 6; cmp\_col++) //比较列

{

//两行的每一列比较

for (int line = 1; line <= 6; line++)

{

pos\_addition[line] = 20000 + std\_col \* 1000 + cmp\_col \* 100 + line \* 10 + 1; //23511

neg\_addition[line] = 20000 + std\_col \* 1000 + cmp\_col \* 100 + line \* 10; //23510

addition[line] = 2000 + std\_col \* 100 + cmp\_col \* 10 + line; //2351

var1 = std\_col + line \* 10; //13

var2 = cmp\_col + line \* 10; //15

//23511= 13∧15转化为CNF生成3个子句：13∨¬23511；15∨¬13511；¬13∨¬15∨23511

fprintf(fp, "%d %d 0\n", var1, -pos\_addition[line]);

fprintf(fp, "%d %d 0\n", var2, -pos\_addition[line]);

fprintf(fp, "%d %d %d 0\n", -var1, -var2, pos\_addition[line]);

//23510= ¬13∧¬15转化为CNF生成3个子句：¬13∨¬23510；¬15∨¬23510；13∨15∨23510

fprintf(fp, "%d %d 0\n", -var1, -neg\_addition[line]);

fprintf(fp, "%d %d 0\n", -var2, -neg\_addition[line]);

fprintf(fp, "%d %d %d 0\n", var1, var2, neg\_addition[line]);

//2351= 23511∨23510转化为CNF生成3个子句：2351∨¬23511；2351∨¬23510；23511∨23510∨¬2351

fprintf(fp, "%d %d 0\n", addition[line], -pos\_addition[line]);

fprintf(fp, "%d %d 0\n", addition[line], -neg\_addition[line]);

fprintf(fp, "%d %d %d 0\n", pos\_addition[line], neg\_addition[line], -addition[line]);

}

//235= ¬[2351∧2352∧…∧2356]转化为CNF生成7个子句：（¬235∨¬2351∨¬2352∨…∨¬2356）∧（2351∨235）∧（2352∨235）…（2356∨235）

int var3 = 200 + std\_col \* 10 + cmp\_col; //235

// fprintf(fp, "%d 0\n", var3); //235不能为假

//（¬235∨¬2351∨¬2352∨…∨¬2356）

fprintf(fp, "%d %d %d %d %d %d %d 0\n", -var3, -addition[1], -addition[2], -addition[3], -addition[4], -addition[5], -addition[6]);

//（2351∨235）∧（2352∨235）…（2356∨235）

for (int i = 1; i <= 6; i++)

{

fprintf(fp, "%d %d 0\n", var3, addition[i]);

}

}

}

}

/\*\*

\* 函数名称：show\_puzzle

\* 函数功能：打印数独

\* 返回值：void

\*/

void show\_puzzle(LiteralList literals[])

{

int x = 0;

//行打印

for (int i = 1; i <= 6; i++)

{

//列打印

for (int j = 1; j <= 6; j++)

{

x = 10 \* i + j;

if (literals[x].value == UNKNOWN)

{

//文字值未知，打印' \_ '

printf("%4c", '\_');

}

else if (literals[x].value > 0)

{

//文字值为正，打印' 1 '

printf("%4d", literals[x].value);

}

else

{

//文字值为负，打印' 0 '

printf("%4d", 0);

}

}

printf("\n");

}

}

/\*\*

\* 函数名称：choose\_puzzle

\* 函数功能：选择数独题目

\* 返回值：void

\*/

void choose\_puzzle(LiteralList literals[])

{

char \*dif = NULL;

char string[10];

int choise;

int op;

//选择难度

printf("\nPlease choose the difficulty!\n");

printf("1.easy 2.medium 3.hard\n");

scanf("%d", &op);

switch (op)

{

case 1:

dif = "easy";

break;

case 2:

dif = "medium";

break;

case 3:

dif = "hard";

break;

}

//选择题号

printf("\nPlease choose the puzzle(1-10):\n");

scanf("%d", &choise);

FILE \*fp;

fp = fopen("puzzle pool.txt", "r");

//找到题目

while (1)

{

fscanf(fp, "%s", string);

if (strcmp(string, dif) == 0) //读到对应难度退出循环

break;

}

while (1)

{

fscanf(fp, "%d", &op);

if (op == choise) //读到对应题号退出循环

break;

}

//题目写入内存

int var = 0;

while (TRUE)

{

fscanf(fp, "%d", &var);

if (var == 0)

{

break;

}

literals[abs(var)].value = var / abs(var);

ltr\_known++;

}

fclose(fp);

}

# 附录F（display.c）

/\*\*

\* 文件名称：display.c

\* 文件描述：界面的显示，函数调用

\*/

#include "head.h"

/\*\*

\* 函数名称：show\_answer

\* 函数功能：输出结果到终端并写入同名文件

\* 返回值：void

\*/

void show\_answer(LiteralList literals[], clock\_t cost, int result, char filename[])

{

FILE \*fp = NULL;

char \*suffix = strrchr(filename, '.'); //找到'.'最后一次出现的位置

//文件名后缀更改

suffix[1] = 'r';

suffix[2] = 'e';

suffix[3] = 's';

//写入,输出求解结果

fp = fopen(filename, "w");

fprintf(fp, "s%6d\n", result);

printf("s%6d\n", result);

//写入,输出具体解

fprintf(fp, "v");

printf("v");

if (result == TRUE)

{

for (int i = 1; i <= ltr\_num; i++)

{

fprintf(fp, "%6d", literals[i].value \* i);

printf("%6d", literals[i].value \* i);

if (i % 10 == 0 && i != ltr\_num)

{

//每10个换行

fprintf(fp, "\n ");

printf("\n ");

}

}

}

//写入,输出求解时间

fprintf(fp, "\nt%6ld", cost);

printf("\nt%6ld ms\n", cost);

fclose(fp);

}

/\*\*

\* 函数名称：check\_answer

\* 函数功能：检查解是否正确

\* 返回值：void

\*/

void check\_answer(LiteralList literals[])

{

//判断是否有答案以供检查

if (literals[1].value == UNKNOWN)

{

printf("There's no Answer to check!\n");

return;

}

FILE \*fp = NULL;

char filename[] = "check report.txt";

ClauseNode \*cp = clist;

VarNode \*vp = NULL;

int status = 0; //子句状态

//检查结果写入文件

fp = fopen(filename, "w");

while (cp != NULL)

{

status = 0;

vp = cp->vn;

//子句文字扫描

while (vp != NULL)

{

fprintf(fp, "%4d", vp->var);

if (literals[abs(vp->var)].value \* vp->var > 0)

{

//有变元为真，则该子句为真

status = TRUE;

}

vp = vp->next\_varNode;

}

//输出该子句真假值

if (status == TRUE)

{

fprintf(fp, "\tT\n");

}

else

{

fprintf(fp, "\tF\n");

}

cp = cp->next\_clauseNode; //扫描下一子句

}

fprintf(fp, "\n");

fclose(fp);

printf("The check report has been generated!\n");

}

/\*\*

\* 函数名称：sat

\* 函数功能：sat操作界面

\* 返回值：void

\*/

void sat()

{

LiteralList literals[Max\_Ltr\_Num];

clock\_t begin = 0, end = 0, cost = 0; //计时相关变量

char filename[100];

int result = 0;

int op = 1;

while (op)

{

system("cls");

printf("\n\n");

printf("%20cSAT<\n", '>');

printf("%10c\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n", ' ');

printf("%13c1. New SAT\n", ' ');

printf("%13c2. Check Answer\n", ' ');

printf("%13c0. Back\n", ' ');

printf("%10c\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n", ' ');

scanf("%d", &op);

switch (op)

{

case 1:

//文件读取

printf("Please input the file path:\n");

scanf("%s", filename);

if (load\_file(literals, filename) == FALSE)

{

printf("Failed to Load File !\n");

getchar();

getchar();

break;

}

else

{

printf("Load File Successfully!\n");

}

//dpll求解并计时

int i = 0;

printf("\nPlease choose the branching strategy:\n");

printf("1 RAND 2 VSIDS\n");

scanf("%d", &i);

begin = clock();

result = dpll(literals, i);

end = clock();

cost = (end - begin);

//输出答案到终端和文件

show\_answer(literals, cost, result, filename);

getchar();

getchar();

break;

case 2:

//检查正误

check\_answer(literals);

getchar();

getchar();

break;

case 0:

return;

default:

break;

}

}

}

/\*\*

\* 函数名称：puzzle

\* 函数功能：数独操作界面

\* 返回值：void

\*/

void puzzle()

{

LiteralList \*literals = (LiteralList \*)calloc(30001, sizeof(LiteralList));

FILE \*fp = NULL;

int op = 1;

while (op)

{

system("cls");

printf("\n\n");

printf("%20cSAT<\n", '>');

printf("%10c\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n", ' ');

printf("%13c1. New Puzzle\n", ' ');

printf("%13c2. Show Answer\n", ' ');

printf("%13c0. Back\n", ' ');

printf("%10c\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n", ' ');

scanf("%d", &op);

switch (op)

{

case 1:

//规则文件生成

rules\_generate(fp);

printf("Rules have been generated successfully!\n");

//规则文件读入内存

load\_file(literals, "puzzle.cnf");

printf("Load File Successfully!!\n");

choose\_puzzle(literals); //选择题目

show\_puzzle(literals); //打印未解答棋盘

getchar();

getchar();

break;

case 2:

//打印未解答棋盘

printf("The puzzle is :\n");

show\_puzzle(literals);

printf("\n");

//当前棋盘未求解

if (ltr\_known != ltr\_num)

{

dpll(literals, 2); //求解

//打印解答棋盘

printf("The answer is :\n");

show\_puzzle(literals);

}

getchar();

getchar();

break;

case 0:

return;

default:

break;

}

}

}

# 附录G（程序使用说明）

主菜单选项

1：SAT

1.New Sat：从文件中读取新的cnf算例进行求解并输出到终端和文件

2.Check：检查上一次求解是否正确，生成检查报告文件

3.Back：回到上一级

2：Puzzle

1.New Puzzle：根据操作提示选取难度和题目

2.Show Puzzle：求解已选择的题目

3.Back：回到上一级

3：Exit