

**课程设计报告**

**题目：基于SAT的数独游戏求解程序**

**课程名称：程序设计综合课程设计**

**专业班级：**

**学 号：**

**姓 名：**

**指导教师： 向文**

**报告日期： 2021年9月15日**

**计算机科学与技术学院**

**任 务 书**

**设计内容**

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

**设计要求**

要求具有如下功能：

1. 输入输出功能：包括程序执行参数的输入，SAT算例cnf文件的读取，执行结果的输出与文件保存等。(15%)
2. 公式解析与验证：读取cnf算例文件，解析文件，基于一定的物理结构，建立公式的内部表示；并实现对解析正确性的验证功能，即遍历内部结构逐行输出与显示每个子句，与输入算例对比可人工判断解析功能的正确性。数据结构的设计可参考文献[1-3]。(15%)
3. DPLL过程：基于DPLL算法框架，实现SAT算例的求解。(35%)
4. 时间性能的测量：基于相应的时间处理函数（参考time.h），记录DPLL过程执行时间（以毫秒为单位），并作为输出信息的一部分。(5%)
5. 程序优化：对基本DPLL的实现进行存储结构、分支变元选取策略[1-3]等某一方面进行优化设计与实现，提供较明确的性能优化率结果。优化率的计算公式为：[(t-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：将数独游戏[5]问题转化为SAT问题[6-8]，并集成到上面的求解器进行数独游戏求解，游戏可玩，具有一定的/简单的交互性。应用问题归约为SAT问题的具体方法可参考文献[3]与[6-8]。(15%)

**参考文献**

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

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

[3] 陈稳. 基于DPLL的SAT算法的研究与应用.硕士学位论文，电子科技大学，2011

[4]Carsten Sinz. Visualizing SAT Instances and Runs of the DPLL Algorithm. J Autom Reasoning (2007) 39:219–243

[5]360百科：数独游戏<https://baike.so.com/doc/3390505-3569059.html>

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

[7] Ins Lynce and Jol Ouaknine. Sudoku as a sat problem. In Proceedings of the 9th International Symposium on Artificial Intelligence and Mathematics, AIMATH 2006, Fort Lauderdale. Springer, 2006.

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

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

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

[10] Robert Ganian and Stefan Szeider. Community Structure Inspired Algorithms for SAT and #SAT. SAT 2015, 223-237360

**目 录**

[任 务 书](#_Toc85135727) Ⅰ

[1 引言 1](#_Toc85135728)

[1.1 课题的背景与意义 1](#_Toc85135729)

[1.1.1 SAT问题简介 1](#_Toc85135730)

[1.1.2 数独游戏简介 1](#_Toc85135731)

[1.2 国内外研究现状 2](#_Toc85135732)

[1.3 课程设计的主要研究工作 3](#_Toc85135733)

[2 系统需求分析与总体设计 4](#_Toc85135734)

[2.1 系统需求分析 4](#_Toc85135735)

[2.2 系统总体设计 4](#_Toc85135736)

[3 系统详细设计 6](#_Toc85135737)

[3.1 有关数据结构的定义 6](#_Toc85135738)

[3.2 主要算法设计 7](#_Toc85135739)

[4 系统实现与测试 9](#_Toc85135740)

[4.1 系统实现 9](#_Toc85135741)

[4.2 系统测试 12](#_Toc85135742)

[4.2.1 SAT问题求解模块 13](#_Toc85135743)

[4.2.2 数独游戏模块 16](#_Toc85135744)

[4.2.3 主控模块 19](#_Toc85135745)

[5 总结与展望 21](#_Toc85135746)

[5.1 全文总结 21](#_Toc85135747)

[5.2 工作展望 21](#_Toc85135748)

[6 体会 22](#_Toc85135749)

[参考文献 23](#_Toc85135750)

[附录…………………………………………………………………………………..24](#_Toc85135751)

1. **引言**

**1.1 课题的背景与意义**

**1.1.1 SAT问题简介**

可满足性问题（Satisfiability Problem）即SAT问题，是对一个以合取范式（Conjunctive Normal Form，常简称CNF）的形式给出的命题逻辑公式进行判断，以找出是否存在一个真值指派，使得该命题逻辑公式的值为真。SAT问题看似简单，但它却是计算机领域和人工智能领域所要研究的中心问题，被称为理论计算机科学和数理逻辑中的第一问题，在硬件验证、人工智能、电子设计自动化、自动化推理、组合等式检测等领域具有非常重要的理论和实践意义。

从1960年至今，SAT问题一直备受人们的关注，世界各国的研究人员在这方面都做了大量的工作，提出了许多求解算法。尽管命题逻辑的可满足性问题理论研究已趋于成熟，但在SAT求解器被越来越多地应用到各种实际问题的今天，探寻解决SAT问题的高效算法仍然是一个吸引人并且极具挑战性的研究方向。

研究能够解决SAT问题的高效算法在理论上有着重要的意义。S.A.Cook于1971年首次证明了布尔表达式的可满足性问题属于NP完全问题。NP完全问题排在七大数学难题之首，在计算复杂性理论中具有非常重要的地位，一方面因为它有着极大的理论价值并且非常难解，另一方面是一旦被破解以后，在诸多的工程领域里还可以得到广泛的应用。

现实生产和生活中存在着大量NP完全问题，寻找高效快速的算法来解决该类问题是计算机理论研究和实际应用领域中的重要工作。由于SAT问题是NP完全问题，它如果能够得到高效解决，那么一定可以高效化解所有其他NP完全问题，这是因为所有的NP完全问题都能在多项式时间内进行相互转化，即所有的NP完全问题都能在多项式时间内转为可满足性问题。所以，如果能找到求解SAT问题的有效算法，那么所有其他NP完全问题也都可以解决了，设计和实现更高效的求解算法意义重大。

* + 1. **数独游戏简介**

数独是一种数学逻辑游戏，游戏由9×9个格子组成，[玩家](https://zh.wikipedia.org/wiki/%E7%8E%A9%E5%AE%B6" \o "玩家)需要根据格子提供的[数字](https://zh.wikipedia.org/wiki/%E6%95%B8%E5%AD%97)推理出其他格子的数字。游戏设计者会提供最少17个数字使得解答谜题只有一个[答案](https://zh.wikipedia.org/wiki/%E7%AD%94%E6%A1%88" \o "答案)。

游戏一般由9个3×3个的九宫格组成。每一列的数字均须包含1～9，不能缺少，也不能重复。每一宫(粗黑线围起来的区域，通常是3\*3的九宫格)的数字均须包含1～9，不能缺少，也不能重复。

数独游戏利用计算机可以转化为SAT问题，从而进行求解。

* 1. **国内外研究现状**

由于SAT问题在计算机理论研究和人工智能领域的重要作用，使得很多学者都在SAT问题求解领域做了大量的研究，可满足性问题进而也成为了国内外研究的热点问题，并在算法研究和技术实现上取得了较大突破，这也推动了形式实验和人工智能等领域的发展。

SAT求解器的实现是我们关心的主要问题，目前的SAT算法大致可以归结为两大类：完备算法（也称回溯搜索算法）和局部搜索算法。完备算法基于穷举法思想，它的优点是能保证找到对应SAT问题的解或证明公式不可满足，但是效率极低，它的平均时间复杂度虽是多项式级的，但是最坏情况下的时间复杂度却是指数级的。局部搜索算法相对于完备算法而言，由于采用了启发式策略来指导搜索，使得求解速度相对较快，但是在某些实例上可能得不到解，它不保证一定能够找到对应的SAT问题的解，即它不能证明SAT问题的不可满足性。

最经典的求解SAT问题的完备算法是DPLL算法，它是由Davis和Putman等人在1960年提出，其他完备算法大都是在DPLL算法的基础上衍生出来的，是对DPLL算法的改进。1966年以后，很多国家相继举办了一些SAT竞赛和研讨会，这使得越来越多的人开始关注并研究SAT问题，所以这段时间也涌现出了众多新的高效的SAT算法，如MINISAT、SATO、CHAFF、POSIT和GRASP等，SAT算法的研究成果显著，求解算法也越来越多地应用到了实际问题领域。国内也涌现出了许多高效的求解算法，如1998年作者梁东敏提出了改进子句加权WAST算法，2000年金人超和黄文奇提出的并行Solar算法，2002年作者张德富提出的模拟退火算法。

尽管SAT算法已经取得了举足轻重的改进，但是仍有一些问题没有得到高效地解决，已经解决的问题可能还存在更好的求解算法，因此眼界并实现高效率的求解算法仍然是当前要解决的中心问题之一。

* 1. **课程设计的主要研究工作**

此次课程设计，主要学习了命题逻辑可满足性的相关理论知识，学习了如何使用DPLL算法求解SAT问题，以及可以优化算法、加速求解过程的方法，完成了DPLL算法求解SAT问题程序，并进行优化上的实践探索，对程序进行了一定的优化。

具体来说，主要研究工作内容如下：

1. 对SAT问题的研究背景、意义及现状进行了简要总结，学习了命题逻辑可满足性问题的基本理论知识。
2. 对基于DPLL的SAT求解算法进行了深入研究，仔细分析了DPLL算法的实现，学习该算法中所使用到的数据结构和一些关键技术。
3. 学习DPLL算法的常用优化方法，优化DPLL算法过程中的程序细节，并选取了一些SAT实例进行测试。
4. 了解数独问题，学习并应用挖洞法随机生成数独格局，设计交互界面，实现游戏的可玩性，并且能够将数独格局转化为.cnf文件，利用SAT求解器求解。
5. 总结课程设计过程，梳理系统设计与结构，进行报告撰写。
6. **系统需求分析与总体设计**
   1. **系统需求分析**

系统主要实现目标为通过DPLL算法实现SAT问题求解器，并基于此实现对数独问题的求解。

系统应能够读取.cnf文件信息，将其转化为CNF公式，并且能够展示公式。能够实现通过DPLL算法求解CNF公式对应的SAT问题，给出结果与求解时间，并将输出信息写到.res文件中。同时，能够自主创建随机数独格局，能够与用户交互，具有可玩性。能够将数独格局转化为CNF公式，利用SAT求解器求解。

* 1. **系统总体设计**

提供可玩的数独游戏，并且将可直接执行的DPLL模块一并提供

执行DPLL算法，提供求解结果

提供解析.cnf文件、建立CNF公式的功能

包含于

CNF解析模块

核心DPLL模块

数独模块

主控、交互与显示模块

包含于

包含于

图2-1 系统模块结构图

根据用户需求，将系统分为4个模块：（1）CNF解析模块；（2）核心DPLL模块；（3）数独模块；（4）主控、交互与显示模块。

CNF解析模块：能够建立CNF公式数据结构，解析.cnf文件，创建CNF公式，并提供CNF公式内容展示功能。

核心DPLL模块：能够执行DPLL算法，求解SAT问题，并将结果输出到同名.res文件中。

数独模块：能够生成随机数独格局，展示数独格局，提供游戏交互，实现可玩性，同时，能够将数独格局转化为CNF公式，写入.cnf文件，送入SAT求解器求解。

主控、交互与显示模块：提供交互界面，对程序执行功能进行控制。

1. **系统详细设计**
   1. **有关数据结构的定义**

系统处理的数据主要分为CNF公式数据，和数独游戏格局数据。

CNF公式采用结构体封装，其中包括两个邻接表，分别为文字邻接表和子句邻接表。文字邻接表中包含文字邻接表表头和邻接于表头的子句结点，表头为结构体数组，其中每个元素包含一个整型的赋值记录，和一个指向邻接子句结点的指针，作为记录该文字在哪些子句中出现过的链表的表头，子句邻接结点为结构体，其中包含整型的子句序号，整型的记录该文字是否在该子句中被删去的标记，以及指向下一个结点的指针。子句邻接表中包括字句邻接表表头和邻接于表头的文字结点，表头为结构体数组，其中包含一个整型的记录该子句是否被删去的标记，和指向邻接文字结点的指针，作为记录该子句中有哪些文字的链表的表头，文字邻接结点为结构体，其中包含整型的文字内容，整型的记录该文字是否在该子句中被删除的标记，以及指向下一个结点的指针。除了两个邻接表外，CNF结构还包括两个整型参数，分别记录公式中的文字数与子句数，以及一个整型数组，用于记录优化算法所使用的变量决策顺序。

数独游戏格局数据由二维整型数组储存，用二维矩阵构成游戏格局。

表3-1 数据结构定义

|  |  |  |  |
| --- | --- | --- | --- |
| 一级结构 | 二级结构 | 三级结构 | 四级结构 |
| CNF公式结构体(struct) | 文字邻接表 | 文字邻接表表头(struct[ ]) | 赋值记录(int) |
| 子句结点指针(struct\*) |
| 子句结点(struct) | 子句序号(int) |
| 文字删除记录(int) |
| 下一子句结点指针(struct\*) |
| 子句邻接表 | 子句邻接表表头(struct[ ]) | 子句删除记录(int) |
| 文字结点指针(struct\*) |
| 文字结点(struct) | 文字内容(int) |
| 文字删除记录(int) |
| 下一文字结点指针(struct\*) |
| 文字数(int) |  |  |
| 子句数(int) |  |  |
| 变量决策顺序数组(int[ ]) |  |  |
| 数独格局(int[ ][ ]) |  |  |  |

数据间的包含或层级关系如上表所示。特别地，程序提供函数使数独格局数据可以转化为CNF公式数据。

* 1. **主要算法设计**

CNF解析模块：

.cnf文件的读取与CNF公式的建立同时进行，先读取文字数与子句数信息，开辟储存空间，然后采用for循环边读边存。

对变量决策的排序采用最短子句最大频率优先（MOM）策略，也就是常说的JW方法，该算法的核心是求解出和文字l相对应的J值，优先选取使得J(l)取值最大的文字l将其值赋为真，求解公式为J(l)=(i=1,2,…,m,假设共有m个子句)，其中是包含文字l的子句的字句长度（i=1,2,…,m）。确定变量决策数组时，先求出各个文字的J值，然后再使用冒泡排序，将得到的变量决策顺序储存于CNF公式结构中的变量决策数组中。

CNF公式通过输出文字数与子句数，并遍历CNF公式子句邻接表输出CNF公式信息。

核心DPLL模块：



图3-1 DPLL算法搜索树

DPLL主体流程框架采用经典的DPLL算法实现，即基于树/二叉树的回溯搜索算法，主要使用两种基本处理策略：单子句规则。如果子句集S中有一个单子句L,那么L一定取真值，于是可以从S中删除所有包含L的子句（包括单子句本身），得到子句集S1，如果它是空集，则S可满足。否则对S1中的每个子句，如果它包含文字¬L,则从该子句中去掉这个文字，这样可得到子句集合S2。S可满足当且仅当S2可满足。单子句传播策略就是反复利用单子句规则化简S的过程。分裂策略。按某种策略选取一个文字L.如果L取真值，则根据单子句传播策略，可将S化成S2；若L取假值（即¬L成立）时，S可化成S1.交错使用上述两种策略可不断地对公式化简，并最终达到终止状态，其执行过程可表示为一棵二叉搜索树,如下图3-1所示。

特别地，本课程设计采用了MOM策略进行变量决策，一定程度上进行了优化。在DPLL过程中的简单细节分别封装，用独立函数实现。

通过遍历文字邻接表，将求解结果输出到指定文件中，并借助时间函数得到算法求解所花费时间。

数独模块：

数独格局的生成通过递归的方式实现，以实现自动生成随机数独格局。首先生成第一行，通过随机数生成，在确定的过程中同时考虑到数独的约束（同行不能重复），利用if进行筛选。将生成的第一行送入递归函数中，并由二行一列开始递归，先遍历当前状态的相同行、列、大格，标记可填的数，填入当前状态，然后将下一状态（右侧下一格，若进行到最右侧则进入下一行第一格）送入递归，直到最后一格（九行九列），递归结束，数独格局生成完毕。

通过遍历二维数组并输出，展示数独格局。

采用命令式的交互方式，通过读入用户的解，与答案比对，给用用户进行游戏进度提示。

输出为.cnf文件采用三位编码的方法，变元可按语义编码为1～9之间数字构成的三位整数ijk，i, j, k∈{1,2,…,9}，其中i表示单元格的行号，j表示单元格的列号，k表示单元格<i, j>填入的数字为k，正文字表示填入k，负文字表示不填入k。并将对应的行约束、列约束、盒子（大格）约束写入文件中，写入过程利用嵌套的for循环实现。

主控、交互与显示模块：

在主控程序中完成变量的定义，用for循环进行与用户进行交互，用switch语句根据用户的指令调用相应函数执行相应的功能。

1. **系统实现与测试**
   1. **系统实现**

系统主要实现目标为通过DPLL算法实现SAT问题求解器，并基于此实现对数独问题的求解。

系统能够读取.cnf文件信息，将其转化为CNF公式，并且能够展示公式。能够实现通过DPLL算法求解CNF公式对应的SAT问题，给出结果与求解时间，并将输出信息写到.res文件中。同时，能够自主创建随机数独格局，能够与用户交互，具有可玩性。能够将数独格局转化为CNF公式，利用SAT求解器求解。

系统程序通过visual studio 2019进行编写、编译，在16G内存电脑上运行，程序实际可调用内存约为4G。

数据结构按3.1描述，依C语言标准完成定义，详见3.1。

函数说明：

（1）clauseLength函数：输入子句（子句邻接表表头结构体元素），返回子句长度。通过遍历该子句邻接文字链表并计数实现。用于求子句长度。

（2）sortV函数：输入CNF公式结构体的引用，无返回值。先遍历文字邻接表表头，利用clauseLength函数求出各个文字的J值，再通过冒泡排序，得出MOM策略的变量决策顺序，将该顺序储存在CNF公式结构中。用于求解通过MOM策略优化的变量决策顺序。

（3）createCNF函数：输入读取目标文件名和CNF公式结构体的引用，返回函数状态（成功返回0，失败返回-1）。打开文件，先读取文字数目参数和子句数目参数，然后开辟相应空间，边读边存，读取完毕后利用sortV函数记录变量决策顺序。用于将.cnf文件内容转化为CNF公式储存，提供给后续程序。

（4）showCNF函数：输入CNF公式结构体，无返回值。输出文字数和子句数，遍历子句邻接表，输出公式内容。用于输出CNF公式信息，提供人工比对功能。

（5）isUnitClause函数：输入子句（子句邻接表表头结构体元素），是单子句返回1，不是返回0。遍历邻接文字链表，判断未被删除文字数目是否为1。用于判断子句是否是单子句。

（6）unitInCNF函数：输入CNF公式结构体，存在单子句返回单子句序号，不存在返回0。遍历子句邻接表，对未被删除的子句调用isUnitClause函数判断是否为单子句。用于寻找CNF公式中的单子句。

（7）isClauseEmpty函数：输入子句（子句邻接表表头结构体元素），是空子句返回1，不是返回0。遍历文字邻接表，判断未被删除的文字数是否未0。用于判断当前子句是否是空子句。

（8）isEmptyClauseExist函数：输入CNF公式结构体，存在空子句返回1，不存在返回0。遍历子句邻接表，对子句调用isClauseEmpty函数判断是否为空子句。用于检查CNF公式中是否存在空子句。

（9）isCNFEmpty函数：输入CNF公式结构体，若为空集返回1，不是返回0。遍历子句邻接表，检查是否有子句未被删除。用于判断CNF公式是否为空集。

（10）vChoose函数：输入CNF公式结构体，返回选择的变元序号。遍历变量决策数组，选择下一个未赋值的变元。用于给出变元决策。

（11）firstBranch函数：输入CNF公式结构体的引用和选择的变元，无返回值。此函数将变元赋值为真，并删除包含该变元正文字的子句，并将该变元的负文字删除。用于进入DPLL算法搜索树的第一分支。

（12）secondBranch函数：输入CNF公式结构体的引用和选择的变元，无返回值。此函数将变元赋值为假，并删除包含该变元负文字的子句，并将该变元的正文字删除。用于进入DPLL算法搜索树的第二分支。

（13）createCopy函数：输入CNF公式结构体副本的引用和原CNF公式结构体，无返回值。通过开辟新的空间，进行CNF公式结构体的深拷贝。用于制作CNF公式的副本，以便于回溯。

（14）DPLL函数：输入CNF公式结构体的引用，返回函数状态（成功返回1，失败返回0）。DPLL算法的主体执行函数，利用上述功能函数，实现DPLL算法。用于实现DPLL算法。

（15）showResult函数：输入CNF公式结构体和输出目标文件名，无返回值。调用clock函数记录DPLL算法执行时间，将求解结果和求解时间输出到目标文件中。用于输出结果到目标文件。

（16）createFirstRow函数：输入一维数组（数独格局第一行），无返回值。通过时间函数随机生成种子，从而达到真随机，通过for循环将随机产生的数1至9填入第一行。用于随机生成数独格局第一行，作为后续递归随机生成全部数独格局的基础。

（17）recursion函数：输入二维数组（数独格局）和行、列数，返回函数状态（成功返回1，失败返回0）。首先根据当前状态所处的行、列和大格情况进行约束，然后填入符合要求的数，将下一状态（右侧下一格，若进行到最右侧则进入下一行第一格）送入递归。用于依据随机生成的数独格局第一行，继续递归随机生成完整数独格局。

（18）createSudoku函数：输入二维数组（数独格局），无返回值。先调用createFirstRow函数，随机生成数独格局第一行，然后将其送入递归生成函数继续随机生成完整数独格局。用于控制和归纳执行完整的数独格局生成流程。

（19）createGame函数：输入两个二维数组（游戏格局和答案格局）和挖洞数，无返回值。首先用时间函数随机生成种子，然后依据挖洞数，产生随机数随即挖洞，并且利用标记数组避免重复挖洞。用于根据挖洞数使用挖洞法随机产生数独游戏格局。

（20）showSudoku函数：输入二维数组（数独格局），无返回值。遍历数独格局二维数组，输出数独格局。用于展示数独格局。

（21）playSudoku函数：输入两个二维数组（游戏格局和答案格局）和挖洞数，无返回值。调用showSudoku函数，向用户展示数独游戏格局，接受用户的指令参数，并根据答案格局向用户给出行动提示，并向用户提供查看答案的功能，在洞被填满或用户终止游戏后，退出游戏。用于完成游戏交互，实现游戏可玩性。

（22）transferToCNF函数：输入二维数组（数独格局）和挖洞数，无返回值。输出为.cnf文件采用三位编码的方法，变元可按语义编码为1～9之间数字构成的三位整数ijk，i, j, k∈{1,2,…,9}，其中i表示单元格的行号，j表示单元格的列号，k表示单元格<i, j>填入的数字为k，正文字表示填入k，负文字表示不填入k。并将对应的行约束、列约束、盒子（大格）约束写入文件中，写入过程利用嵌套的for循环实现。用于将数独格局写入.cnf文件中，提供给SAT求解器求解。

（23）main函数：初始化、主控函数。定义所需变量，通过while循环结合switch语句，设计一个一级菜单（主功能菜单）和一个二级菜单（数独游戏菜单），通过接受用户的指令，实现调用所有模块的功能。

showSudoku

recursion

createFirstRow

unitInCNF

vChoose

isUnitClause

createCopy

isClauseEmpty

isEmptyClause

Exist

secondBranch

firstBranch

isCNFEmpty

main函数

DPLL

sortV

playSudoku

transferToCNF

createGame

createSudoku

showResult

showCNF

createCNF

图4-1 函数间的调用关系

程序详见附录。

* 1. **系统测试**

[软件测试](https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E6%B5%8B%E8%AF%95/327953)是使用人工或自动的手段来运行或测定某个[软件系统](https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E7%B3%BB%E7%BB%9F/224122)的过程，其目的在于检验它是否满足规定的需求或弄清[预期结果](https://baike.baidu.com/item/%E9%A2%84%E6%9C%9F%E7%BB%93%E6%9E%9C/9582971)与实际结果之间的差别。从是否关心软件内部结构和具体实现的角度划分，测试方法主要有白盒测试和黑盒测试。白盒测试方法主要有代码检査法、静态结构分析法、静态质量度量法、[逻辑覆盖法](https://baike.baidu.com/item/%E9%80%BB%E8%BE%91%E8%A6%86%E7%9B%96%E6%B3%95/5719341)、基本路径测试法、[域测试](https://baike.baidu.com/item/%E5%9F%9F%E6%B5%8B%E8%AF%95/5021990)、符号测试、路径覆盖和程序变异。黑盒测试方法主要包括等价类划分法、[边界值分析法](https://baike.baidu.com/item/%E8%BE%B9%E7%95%8C%E5%80%BC%E5%88%86%E6%9E%90%E6%B3%95/4137943)、[错误推测法](https://baike.baidu.com/item/%E9%94%99%E8%AF%AF%E6%8E%A8%E6%B5%8B%E6%B3%95/788810)、因果图法、[判定表驱动法](https://baike.baidu.com/item/%E5%88%A4%E5%AE%9A%E8%A1%A8%E9%A9%B1%E5%8A%A8%E6%B3%95/8643441)、正交试验设计法、功能图法、[场景法](https://baike.baidu.com/item/%E5%9C%BA%E6%99%AF%E6%B3%95/4909843)等

一般情况下，系统测试采用黑盒法来进行测试的，以此来检查该系统是否符合软件需求。本阶段的主要测试内容包括[健壮性测试](https://baike.baidu.com/item/%E5%81%A5%E5%A3%AE%E6%80%A7%E6%B5%8B%E8%AF%95/4435189)、性能测试、功能测试、安装或反安装测试、用户界面测试、压力测试、可靠性及安全性测试等。为了有效保证这一阶段测试的客观性，必须由独立的测试小组来进行相关的系统测试。另外，系统测试过程较为复杂，由于在系统测试阶段不断变更需求造成功能的删除或增加，从而使程序不断出现相应的更改，而程序在更改后可能会出现新的问题，或者原本没有问题的功能由于更改导致出现问题。所以，测试人员必须进行回归测试。

**4.2.1 SAT问题求解模块**

此模块主要功能为求解.cnf文件中的SAT问题，在基础DPLL算法的基础上对程序进行一定的优化，并将求解结果与求解时间信息输出到同名的.res文件中，依此可以得出程序的优化率。

测试选取18个算例，其中可满足算例15个，不满足算例3个。考虑到实现的SAT求解器为初级求解器，算法较为简单，且程序执行环境资源有限，在能够解决的算例中，将算例按大小分为三类：小型算例变元数为100个左右，选取了8个，中型算例变元数介于200-500个，选取了5个， 大型算例变元数600个以上，选取了5个。

表4-1 小型算例测试

|  |  |  |  |
| --- | --- | --- | --- |
| 算例信息 | 优化前 | 优化后 | 优化率[(t-to)/t]\*100% |
| 算例名：7cnf20\_90000\_9000\_7.  shuffled-20；变元数：20；子句数与变元数比值：76.6；可满足 |  |  | -23.8% |
| 算例名：problem2-50；变元数：50；子句数与变元数比值：1.6；可满足 |  |  | 77.0% |
| 算例名：problem6-50；变元数：50；子句数与变元数比值：2；可满足 |  |  | -4.3% |
| 算例名：problem8-50；变元数：50；子句数与变元数比值：6；可满足 |  |  | -30.3% |
| 算例名：problem11-100；变元数：100；子句数与变元数比值：6；可满足 |  |  | -15.8% |
| 算例名：tst\_v10\_c100；变元数：10；子句数与变元数比值：10；不可满足 |  |  |  |
| 算例名：unsat-5cnf-30；变元数：30；子句数与变元数比值：14；不可满足 |  |  |  |
| 算例名：u-problem7-50；变元数：50；子句数与变元数比值：2；不可满足 |  |  |  |

表4-2 中型算例测试

|  |  |  |  |
| --- | --- | --- | --- |
| 算例信息 | 优化前 | 优化后 | 优化率[(t-to)/t]\*100% |
| 算例名：3；变元数：301；子句数与变元数比值：9.2；可满足 |  |  | 18.5% |
| 算例名：sud00009；变元数：303；子句数与变元数比值：9.4；可满足 |  |  | -281.1% |
| 算例名：sud00012；变元数：232；子句数与变元数比值：8.2；可满足 |  |  | 5.2% |
| 算例名：tst\_v200\_c210；变元数：200；子句数与变元数比值：1.1；可满足 |  |  | 12.9% |
| 算例名：tst\_v200\_c220；变元数：200；子句数与变元数比值：1.1；可满足 |  |  | 12.5% |

表4-3 大型算例测试

|  |  |  |  |
| --- | --- | --- | --- |
| 算例信息 | 优化前 | 优化后 | 优化率[(t-to)/t]\*100% |
| 算例名：sudoku；变元数：999；子句数与变元数比值：8.9；可满足 |  |  | 39.2% |
| 算例名：sudoku1；变元数：999；子句数与变元数比值：8.9；可满足 |  |  | -1.1% |
| 算例名：sudoku2；变元数：999；子句数与变元数比值：8.9；可满足 |  |  | -1.7% |
| 算例名：sudoku3；变元数：999；子句数与变元数比值：8.9；可满足 |  |  | -0.5% |
| 算例名：sudoku4；变元数：999；子句数与变元数比值：8.9；可满足 |  |  | 65.6% |

以上运行结果截图均为输出的结果文件.res文件内容，限于篇幅，无法将求解所得的所有变元赋值展示，但经过验证程序验证，均求解正确。

从优化率看，由于采用的优化方法为采用MOM策略选取变元，通过更优的变元选取策略，从而加速程序的执行，然而确定变元选取的顺序本身也是占用内存、消耗程序执行时间的，所以可能会导致某些算例执行时间不减反增。在上面的测试算例中，确有不少负优化率的情况，考虑到此次实现的是初级求解器，算法结构简单，这是在意料之内的，但我们也很欣慰的发现，还是有不少算例成功地优化，部分算例优化程度明显。

综上，SAT求解模块能够求解SAT问题，能够正确地输出求解结果，程序优化基本实现。

**4.2.2 数独游戏模块**

此模块主要功能为利用挖洞法随机生成数独格局，提供交互人机界面实现可玩性，游戏过程中能够应对玩家的不同行为，如错误指令、终止游戏、查看答案等。并且能够将数独格局输出为.cnf文件，送入SAT求解器求解。

此模块的测试方向主要分为两个，一是数独格局产生的随机性，程序应能够利用挖洞法随机地产生数独格局，每次产生的数独格局不应相同或有关联，测试限于篇幅，选取产生的5个数独格局。二是用户交互的流畅性与安全性，测试将模拟用户可能的交互行为。

表4-4 数独格局的随机性测试

|  |  |
| --- | --- |
| 序号 | 数独格局 |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 |  |

表4-5 用户行为测试

|  |  |  |
| --- | --- | --- |
| 用户行为 | 程序应有反馈 | 程序实际反馈 |
| 选择进行游戏 | 展示数独格局，提示用户进行命令交互 |  |
| 填入错误答案 | 告知填入错误 |  |
| 查看答案 | 展示答案 |  |
| 填入正确答案 | 告知填入正确，展示填入正确后的格局 |  |
| 填入位置错误 | 告知位置填入错误 |  |
| 终止游戏 | 告知游戏结束，返回菜单1 |  |
| 菜单2（数独菜单）输入无效指令 | 告知指令无效 |  |
| 生成.cnf文件 | 生成.cnf文件，告知成功 |  |
| 返回菜单1 | 返回菜单1 |  |

从表4-4可以看出，程序成功地随机生成了数独格局。从表4-5可以看出，程序能够应对用户的各种行为。综上，数独游戏模块功能基本实现。

**4.2.3 主控模块**

此模块主要功能为提供和用户交互的界面，使程序能够根据用户的指令调用各模块，执行各功能。

测试将模拟用户可能的行为，程序应能够给予正确反馈。

表4-6 主控模块测试

|  |  |  |
| --- | --- | --- |
| 用户行为 | 程序应有反馈 | 程序实际反馈 |
| 进入程序 | 展示一级菜单 |  |
| 选择功能：SAT问题；输入待解决的文件名 | 进入模块，接受文件名，读取文件创建CNF公式 |  |
| 展示CNF公式 | 展示CNF公式 | （限于篇幅，未完全展示） |
| 不展示CNF公式 | 不展示CNF公式 |  |
| 完成SAT问题预备信息填写 | 解决SAT问题，提示用户 |  |
| 选择是否展示公式时输入无效命令 | 告知命令无效 |  |
| 选择功能：数独问题 | 提示用户数度格局已生成，进入菜单2（数独菜单） |  |
| 输入无效功能指令 | 提示用户指令无效 |  |
| 退出 | 感谢用户使用 |  |

由上表可以看出，程序能够应对用户的各种行为，模块功能基本实现。

1. **总结与展望**
   1. **全文总结**

本次课程设计，主要工作如下：

1. 阅读文献与资料，学习并了解DPLL算法的内容与背景，整体把握此次课程设计内容。
2. 深入研究基于DPLL的SAT求解算法，仔细分析DPLL算法的实现，设计实现算法的数据结构。
3. 学习DPLL算法的常用优化方法，优化DPLL算法过程中的程序细节，并选取了一些SAT实例进行测试。
4. 了解数独问题，学习并应用挖洞法随机生成数独格局，设计交互界面，实现游戏的可玩性，并且能够将数独格局转化为.cnf文件，利用SAT求解器求解。
5. 总结课程设计过程，梳理系统设计与结构，进行报告撰写。
   1. **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. 继续深度优化DPLL算法，提出更好的变元优化策略，并且从其他方面如回溯机制、数据结构等添加更多优化方法，努力使优化效率更高，适用场景更广。
2. 优化用户交互界面，提供更加美观、便捷的用户交互形式，如利用GUI等，提高数独游戏可玩性，提高程序的美观与整洁度。
3. **体会**

通过此次课程设计，加强了大工程量程序设计的能力，学会了不同功能模块的协调统一，并最后统一到主控一体化执行。完成过程中文件的包含、变量地址的传递，都得到了深刻理解。

此次课程设计的课题也让我接触到了DPLL算法，以及SAT问题，拓宽了我的眼界，让我对计算机的应用场景有了更多的了解。

此次课程设计的优化过程让我切实认识到了程序对内存的占用是如何影响程序运行的，加强了日后的程序优化算法、节省内存开销的意识。

此次课程设计的用户交互模块，丰富了我设计交互功能的手段，也让我意识到了除了终端，日后更要学习GUI等先进的人机交互功能。

**参考文献**

[1] 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

[2]数独.维基百科，<https://zh.wikipedia.org/wiki/%E6%95%B8%E7%8D%A8>

[3]软件测试方法.百度百科，<https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E6%B5%8B%E8%AF%95%E6%96%B9%E6%B3%95/1850037?fr=aladdin>

**附录**

cnfparser.h文件:

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<math.h>

typedef struct ClauseNode

{

int clause; //子句序号

int flag; //记录该子句中是否包含对应文字，包含为1，文字被删去为0

struct ClauseNode\* next;

}ClauseNode; //邻接于文字表的结点

typedef struct Literal

{

int value; //取值，1表示T，0表示F，-1表示未赋值

ClauseNode\* NextCla; //记录该文字在哪些子句中出现过

}Literal, \* LiteralArray; //文字邻接表表头

typedef struct LiteralNode

{

int literal; //文字内容，其中正负表示是否取“非”

int flag; //记录该文字是否被从该子句中去掉，0为去掉，1为存在

struct LiteralNode\* next;

}LiteralNode; //邻接于子句表的结点

typedef struct Clause

{

int flag; //记录该子句是否被去掉，0为去掉，1为存在

LiteralNode\* NextLit;

}Clause, \* ClauseArray; //子句邻接表表头

typedef struct CNF

{

int LitNum;

int ClaNum;

ClauseArray AllClause;

LiteralArray AllLiteral;

int\* vSequence; //记录变量决策顺序

}CNF;

//优化前

//typedef struct CNF

//{

// int LitNum;

// int ClaNum;

// ClauseArray AllClause;

// LiteralArray AllLiteral;

//}CNF;

int clauseLength(Clause clause)

{ //求子句长度，返回子句长度

LiteralNode\* p = NULL;

int l = 0;

for (p = clause.NextLit; p != NULL; p = p->next)

l++;

return l;

}

void sortV(CNF &cnf)

{ //求边缘决策顺序

struct J

{

int lit; //文字序号

double j; //J值

};

struct J\* litJ = (struct J\*)malloc(sizeof(struct J) \* (cnf.LitNum + 3));

struct J t;

int i, j, n;

ClauseNode\* p = NULL;

for (i = 1; i <= cnf.LitNum; i++)

{

litJ[i].lit = i;

litJ[i].j = 0;

for (p = cnf.AllLiteral[i].NextCla; p != NULL; p = p->next)

{

n = clauseLength(cnf.AllClause[p->clause]);

litJ[i].j += pow(2, -n);

}

}

//冒泡排序

for (i = 1; i < cnf.LitNum; i++)

{

for (j = i; j < cnf.LitNum; j++)

{

if (litJ[j].j < litJ[j + 1].j)

{

t = litJ[j];

litJ[j] = litJ[j + 1];

litJ[j + 1] = t;

}

}

}

for (i = 1; i <= cnf.LitNum; i++)

cnf.vSequence[i - 1] = litJ[i].lit;

}

int createCNF(char fileName[], CNF &cnf)

{ //创建cnf公式，成功返回0，失败返回-1

strcat(fileName, ".cnf");

FILE\* fp = fopen(fileName, "r");

if (fp == NULL)

{

printf("文件打开失败\n");

return -1;

}

char s[50];

int i;

while (fscanf(fp, "%s", s) != EOF)

{

if (strcmp(s, "p") == 0)

break;

}

while (fscanf(fp, "%s", s) != EOF)

{

if (strcmp(s, "cnf") == 0)

{

fscanf(fp, "%d", &cnf.LitNum);

fscanf(fp, "%d", &cnf.ClaNum);

break;

}

}

//初始化

cnf.AllLiteral = (LiteralArray)malloc(sizeof(Literal) \* (cnf.LitNum + 5));

cnf.AllClause = (ClauseArray)malloc(sizeof(Clause) \* (cnf.ClaNum + 5));

int l;

for (i = 1; i <= cnf.LitNum; i++)

{

cnf.AllLiteral[i].NextCla = NULL;

cnf.AllLiteral[i].value = -1;

}

for (i = 1; i <= cnf.ClaNum; i++)

{

cnf.AllClause[i].NextLit = NULL;

cnf.AllClause[i].flag = 1;

}

//读取文件，建立CNF

ClauseNode\* pCNode = NULL;

LiteralNode\* pLNode = NULL;

for (i = 1; i <= cnf.ClaNum; i++)

{

do {

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

if (l != 0)

{

//对文字邻接表的处理

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

pCNode->clause = i;

pCNode->flag = 1;

if (l > 0)

{

pCNode->next = cnf.AllLiteral[l].NextCla;

cnf.AllLiteral[l].NextCla = pCNode;

}

else

{

pCNode->next = cnf.AllLiteral[-l].NextCla;

cnf.AllLiteral[-l].NextCla = pCNode;

}

//对子句邻接表的处理

pLNode = (LiteralNode\*)malloc(sizeof(LiteralNode));

pLNode->literal = l;

pLNode->flag = 1;

pLNode->next = cnf.AllClause[i].NextLit;

cnf.AllClause[i].NextLit = pLNode;

}

} while (l != 0);

}

printf("CNF公式已创建\n");

fclose(fp);

cnf.vSequence = (int\*)malloc(sizeof(int) \* (cnf.LitNum + 3));

sortV(cnf);

return 0;

}

void showCNF(CNF cnf)

{ //展示cnf公式内容

printf("此公式 文字数：%d 子句数：%d\n", cnf.LitNum, cnf.ClaNum);

printf("公式内容：\n");

int i;

LiteralNode\* p = NULL;

for (i = 1; i <= cnf.ClaNum; i++)

{

for (p = cnf.AllClause[i].NextLit; p != NULL; p = p->next)

printf("%d ", p->literal);

printf("\n");

}

}

solver.h文件:

#include"cnfparser.h"

#include<time.h>

int isUnitClause(Clause clause)

{ //判断当前子句是否为单子句，是返回1，否返回0

LiteralNode\* p = NULL;

int num = 0; //记录子句中的文字数

if (clause.flag == 0)

return 0;

for (p = clause.NextLit; p != NULL; p = p->next)

{

if (p->flag == 1)

num++;

}

if (num == 1)

return 1;

else return 0;

}

int unitInCNF(CNF cnf)

{ //判断CNF公式中是否存在单子句，存在则返回单子句序号，不存在返回0

int i;

for (i = 1; i <= cnf.ClaNum; i++)

{

if (cnf.AllClause[i].flag == 0)

continue;

else

{

if (isUnitClause(cnf.AllClause[i]))

return i;

}

}

return 0; //未找到单子句

}

int isClauseEmpty(Clause clause)

{ //判断该子句是否为空子句，是返回1，不是返回0

LiteralNode\* p = NULL;

for (p = clause.NextLit; p != NULL; p = p->next)

{

if (p->flag == 1)

return 0;

}

return 1;

}

int isEmptyClauseExist(CNF cnf)

{ //判断cnf中是否存在空子句，存在返回1，不存在返回0

int i;

for (i = 1; i <= cnf.ClaNum; i++)

{

if (isClauseEmpty(cnf.AllClause[i]) == 1)

return 1;

}

return 0;

}

int isCNFEmpty(CNF cnf)

{ //判断cnf是否为空集，是返回1，不是返回0

int i;

for (i = 1; i <= cnf.ClaNum; i++)

{

if (cnf.AllClause[i].flag == 1)

return 0;

}

return 1;

}

int vChoose(CNF cnf)

{ //选择一个变元，返回变元序号

int i;

for (i = 0; i < cnf.LitNum; i++)

{

if (cnf.AllLiteral[cnf.vSequence[i]].value == -1)

return cnf.vSequence[i];

}

return 0; //所有变元已赋值完毕

}

//优化前

//int vChoose(CNF cnf)

//{ //选择一个变元，返回变元序号

// int i;

// for (i = 0; i < cnf.LitNum; i++)

// {

// if (cnf.AllLiteral[i].value == -1)

// return i;

// }

// return 0; //所有变元已赋值完毕

//}

void firstBranch(CNF& cnf, int v)

{ //选择变元后对cnf进行处理

//处理文字邻接表

cnf.AllLiteral[v].value = 1;

//处理子句邻接表

int i;

LiteralNode\* p = NULL;

for (i = 1; i <= cnf.ClaNum; i++)

{

if (cnf.AllClause[i].flag == 0)

continue;

for (p = cnf.AllClause[i].NextLit; p != NULL; p = p->next)

{

if (p->flag == 0)

continue;

else if (p->literal == v)

cnf.AllClause[i].flag = 0;

else if (p->literal + v == 0)

p->flag = 0;

}

}

}

void secondBranch(CNF& cnf, int v)

{ //进入另一分支

//处理文字邻接表

cnf.AllLiteral[v].value = 0;

//处理子句邻接表

int i;

LiteralNode\* p = NULL;

for (i = 1; i <= cnf.ClaNum; i++)

{

if (cnf.AllClause[i].flag == 0)

continue;

for (p = cnf.AllClause[i].NextLit; p != NULL; p = p->next)

{

if (p->flag == 0)

continue;

else if (p->literal == v)

p->flag = 0;

else if (p->literal + v == 0)

cnf.AllClause[i].flag = 0;

}

}

}

void createCopy(CNF& copy, CNF cnf)

{ //制作副本

//初始化

copy.ClaNum = cnf.ClaNum;

copy.LitNum = cnf.LitNum;

copy.AllClause = (ClauseArray)malloc(sizeof(Clause) \* (cnf.ClaNum + 5));

copy.AllLiteral = (LiteralArray)malloc(sizeof(Literal) \* (cnf.LitNum + 5));

ClauseNode\* pCNode = NULL, \*p = NULL;

LiteralNode\* pLNode = NULL, \*q = NULL;

int i;

//处理子句邻接表

for (i = 1; i <= cnf.ClaNum; i++)

{

copy.AllClause[i].flag = cnf.AllClause[i].flag;

copy.AllClause[i].NextLit = NULL;

for (q = cnf.AllClause[i].NextLit; q != NULL; q = q->next)

{

pLNode = (LiteralNode\*)malloc(sizeof(LiteralNode));

pLNode->flag = q->flag;

pLNode->literal = q->literal;

pLNode->next = copy.AllClause[i].NextLit;

copy.AllClause[i].NextLit = pLNode;

}

}

//处理文字邻接表

for (i = 1; i <= cnf.LitNum; i++)

{

copy.AllLiteral[i].value = cnf.AllLiteral[i].value;

copy.AllLiteral[i].NextCla = NULL;

for (p = cnf.AllLiteral[i].NextCla; p != NULL; p = p->next)

{

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

pCNode->clause = p->clause;

pCNode->flag = p->flag;

pCNode->next = copy.AllLiteral[i].NextCla;

copy.AllLiteral[i].NextCla = pCNode;

}

}

}

int DPLL(CNF& cnf)

{ //DPLL算法, 成功解决返回1，失败返回0

while (unitInCNF(cnf))

{

int l = unitInCNF(cnf); //单子句序号

//找到单子句中文字内容

LiteralNode\* q = NULL;

int lit;

for (q = cnf.AllClause[l].NextLit; q != NULL; q = q->next)

{

if (q->flag != 0)

{

lit = q->literal;

break;

}

}

//依据单子句规则化简

//处理文字邻接表

ClauseNode\* p1 = NULL, \*p2 = NULL;

if (lit > 0)

{

cnf.AllLiteral[lit].value = 1;

p1 = cnf.AllLiteral[lit].NextCla;

}

else

{

cnf.AllLiteral[-lit].value = 0;

p1 = cnf.AllLiteral[-lit].NextCla;

}

//处理子句邻接表

for (p2 = p1; p2 != NULL; p2 = p2->next)

{

for (q = cnf.AllClause[p2->clause].NextLit; q != NULL; q = q->next)

{

if (q->literal == lit)

cnf.AllClause[p2->clause].flag = 0; //将子句去掉

}

}

if (isCNFEmpty(cnf) == 1)

return 1;

else

{

for (p2 = p1; p2 != NULL; p2 = p2->next)

{

for (q = cnf.AllClause[p2->clause].NextLit; q != NULL; q = q->next)

{

if (q->literal == -lit)

q->flag = 0;

}

}

}

if (isCNFEmpty(cnf) == 1)

return 1;

else if (isEmptyClauseExist(cnf) == 1)

return 0;

}

int v = vChoose(cnf);

CNF cnf1; //cnf副本

createCopy(cnf1, cnf);

firstBranch(cnf, v);

if (DPLL(cnf) == 1)

return 1;

//回溯到对v执行分支策略的初态

createCopy(cnf, cnf1);

secondBranch(cnf, v);

if (DPLL(cnf) == 1)

return 1;

else

{

createCopy(cnf, cnf1);

return 0;

}

}

void showResult(CNF cnf, char fileName[])

{ //输出cnf结果

strcat(fileName, ".res");

FILE \*fp = fopen(fileName, "w");

clock\_t start\_t, end\_t;

int state;

start\_t = clock();

state = DPLL(cnf);

end\_t = clock();

if (state == 0)

{

fprintf(fp, "s 0\n");

return;

}

else fprintf(fp, "s 1\n");

double t = (double)(end\_t - start\_t) / CLOCKS\_PER\_SEC;

fprintf(fp, "v ");

int i;

for (i = 1; i <= cnf.LitNum; i++)

{

if (cnf.AllLiteral[i].value == 1 || cnf.AllLiteral[i].value == -1)

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

else if (cnf.AllLiteral[i].value == 0)

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

}

fprintf(fp, "\nt %f", t \* 1000);

fclose(fp);

}

sudoku.h文件:

#include"solver.h"

void createFirstRow(int a[])

{ //生成数独第一行

int i, j;

srand((unsigned)time(NULL));

for (i = 0; i < 9; i++)

{

a[i] = rand() % 9 + 1;

j = 0;

while (j < i)

{

if (a[i] == a[j])

{

a[i] = rand() % 9 + 1;

j = 0;

}

else j++;

}

}

}

int recursion(int \*\*graph, int row, int col)

{ //递归生成余下的数独格局，成功返回1，失败返回0

//对填入数字进行限定

if (row < 9 && col < 9)

{

int i, j;

int mark[10];

for (i = 0; i < 10; i++)

mark[i] = 0; //0表示当前数字可填，1表示不可填

//处理相同列

for (i = 0; i < row; i++)

mark[graph[i][col]] = 1;

//处理相同行

for (i = 0; i < col; i++)

mark[graph[row][i]] = 1;

//处理相同大格

for (i = row / 3 \* 3; i <= row; i++)

{

if (i == row)

{

for (j = col / 3 \* 3; j < col; j++)

mark[graph[i][j]] = 1;

}

else

{

for (j = col / 3 \* 3; j < col / 3 \* 3 + 3; j++)

mark[graph[i][j]] = 1;

}

}

//进行递归填入

int flag = 0;

for (i = 1; i <= 9 && flag == 0; i++)

{

if (mark[i] == 0)

{

flag = 1;

graph[row][col] = i;

if (col == 8 && row != 8)

{

if (recursion(graph, row + 1, 0) == 1)

return 1;

else flag = 0;

}

else if (col != 8)

{

if (recursion(graph, row, col + 1) == 1)

return 1;

else flag = 0;

}

}

}

if (flag == 0)

{

graph[row][col] = 0;

return 0;

}

}

return 1;

}

void createSudoku(int\*\* graph)

{ //生成数独

createFirstRow(graph[0]);

recursion(graph, 1, 0);

}

void createGame(int\*\* gameGraph, int\*\* graph, int hole)

{ //生成数独游戏（挖洞）

srand((unsigned)time(NULL));

//制作副本，用于生成游戏

int i, j;

for (i = 0; i < 9; i++)

{

for (j = 0; j < 9; j++)

gameGraph[i][j] = graph[i][j];

}

//创建二维数组，用于记录是否挖到相同的洞

int\*\* mark = (int\*\*)malloc(sizeof(int\*) \* hole);

for (i = 0; i < hole; i++)

{

mark[i] = (int\*)malloc(sizeof(int) \* 2);

mark[i][0] = mark[i][1] = 0;

}

int flag; //记录是否成功挖洞

int k, t;

for (i = 0; i < hole; i++)

{

j = rand() % 9;

k = rand() % 9;

flag = 0;

for (t = 0; t < i; t++)

{

if (j == mark[t][0] && k == mark[t][1])

flag = 1;

}

if (flag == 0)

{

gameGraph[j][k] = 0;

mark[i][0] = j;

mark[i][1] = k;

}

else i--;

}

}

void showSudoku(int\*\* graph)

{ //展示数独

int i, j;

printf("\n");

for (i = 0; i < 9; i++)

{

if (i == 3 || i == 6)

printf("-----------------------------\n");

for (j = 0; j < 9; j++)

{

if (j == 3 || j == 6)

printf("|");

if (graph[i][j] != 0)

printf(" %d ", graph[i][j]);

else printf(" / ");

}

printf("\n");

}

printf("\n");

}

void playSudoku(int\*\* gameGraph, int\*\* answer, int hole)

{ //数独游戏交互

int row, col, value;

while (hole != 0)

{

showSudoku(gameGraph);

printf("请输入您想进行的填空操作：（行、列、值，用空格隔开）\n");

printf("（退出请输入-1，查看答案请输入0）\n");

scanf("%d", &row);

if (row == -1)

{

printf("游戏结束\n");

return;

}

else if (row == 0)

{

showSudoku(answer);

continue;

}

scanf("%d %d", &col, &value);

row--;

col--;

if (row < 0 || row > 8 || col < 0 || col > 8)

printf("无效位置\n");

else

{

if (gameGraph[row][col] == '/')

printf("不能修改初始格局\n");

else

{

if (answer[row][col] == value)

{

gameGraph[row][col] = value;

hole--;

printf("填入正确\n");

}

else

{

printf("填入错误\n");

}

}

}

}

printf("成功！\n");

}

void transferToCNF(int\*\* graph, int hole)

{

FILE\* fp = fopen("sudoku.cnf", "w");

int i, j, k, l;

fprintf(fp, "p cnf %d %d \n", 999, 8829 + 81 - hole);

//已填入数的单子句

for (i = 0; i < 9; i++)

{

for (j = 0; j < 9; j++)

{

if (graph[i][j] != 0)

fprintf(fp, "%d %d\n", (i + 1) \* 100 + (j + 1) \* 10 + graph[i][j], 0);

}

}

//每一个格子中均有数

for (i = 1; i <= 9; i++)

{

for (j = 1; j <= 9; j++)

{

for (k = 1; k <= 9; k++)

fprintf(fp, "%d ", i \* 100 + j \* 10 + k);

fprintf(fp, "%d\n", 0);

}

}

//列约束

for (i = 1; i <= 9; i++)

{

for (j = 1; j <= 9; j++)

{

for (k = 1; k <= 8; k++)

{

for (l = k + 1; l <= 9; l++)

fprintf(fp, "%d %d %d\n", 0 - (k \* 100 + i \* 10 + j), 0 - (l \* 100 + i \* 10 + j), 0);

}

}

}

//行约束

for (i = 1; i <= 9; i++)

{

for (j = 1; j <= 9; j++)

{

for (k = 1; k <= 8; k++)

{

for (l = k + 1; l <= 9; l++)

fprintf(fp, "%d %d %d\n", 0 - (i \* 100 + k \* 10 + j), 0 - (i \* 100 + l \* 10 + j), 0);

}

}

}

//盒子约束

int x, y, z;

for (z = 1; z <= 9; z++)

{

for (i = 0; i <= 2; i++)

{

for (j = 0; j <= 2; j++)

{

for (x = 1; x <= 3; x++)

{

for (y = 1; y <= 3; y++)

{

for (k = y + 1; k <= 3; k++)

fprintf(fp, "%d %d %d\n", 0 - ((3 \* i + x) \* 100 + (3 \* j + y) \* 10 + z), 0 - ((3 \* i + x) \* 100 + (3 \* j + k) \* 10 + z), 0);

}

}

}

}

}

for (z = 1; z <= 9; z++)

{

for (i = 0; i <= 2; i++)

{

for (j = 0; j <= 2; j++)

{

for (x = 1; x <= 3; x++)

{

for (y = 1; y <= 3; y++)

{

for (k = x + 1; k <= 3; k++)

{

for (l = 1; l <= 3; l++)

fprintf(fp, "%d %d %d\n", 0 - ((3 \* i + x) \* 100 + (3 \* j + y) \* 10 + z), 0 - ((3 \* i + k) \* 100 + (3 \* j + l) \* 10 + z), 0);

}

}

}

}

}

}

fclose(fp);

}

main.cpp文件:

#include"sudoku.h"

int main()

{

int option, i, hole, state;

char fileName[50], s[50];

CNF cnf;

int\*\* graph = (int\*\*)malloc(sizeof(int\*) \* 10);

for (i = 0; i < 10; i++)

graph[i] = (int\*)malloc(sizeof(int) \* 10);

int\*\* gameGraph = (int\*\*)malloc(sizeof(int\*) \* 10);

for (i = 0; i < 10; i++)

gameGraph[i] = (int\*)malloc(sizeof(int) \* 10);

while (1)

{ //一级菜单

iny:

printf("--------------------Menu1-----------------------\n");

printf(" 1.SAT问题 2.数独问题\n");

printf(" 0.退出\n");

printf("------------------------------------------------\n");

scanf("%d", &option);

switch (option)

{

case 1:

printf("请输入文件名：");

scanf("%s", fileName);

if (createCNF(fileName, cnf) == 0)

printf("CNF公式已创建，是否展示公式内容？（请输入yes或no）");

else

{

printf("未找到文件\n");

break;

}

while(1)

{

scanf("%s", s);

if (strcmp(s, "yes") == 0)

{

showCNF(cnf);

break;

}

else if (strcmp(s, "no") == 0)

break;

else printf("无效命令\n");

}

printf("SAT问题已解决\n");

showResult(cnf, fileName);

break;

case 2:

createSudoku(graph);

srand((unsigned)time(NULL));

do {

hole = rand() % 65;

} while (hole < 20);

createGame(gameGraph, graph, hole);

printf("已生成数独格局\n");

goto inx;

case 0:

printf("感谢使用！\n");

return 0;

default:

printf("无效指令\n");

}

}

while (1)

{ //数独的二级菜单

inx:

printf("--------------------Menu2-----------------------\n");

printf("1.输出为cnf文件 2.进行游戏\n");

printf("0.返回上一级\n");

printf("------------------------------------------------\n");

scanf("%d", &option);

switch (option)

{

case 1:

transferToCNF(gameGraph, hole);

printf("已生成cnf文件\n");

break;

case 0:

goto iny;

case 2:

playSudoku(gameGraph, graph, hole);

goto iny;

break;

default:

printf("无效指令\n");

}

}

return 0;

}