

**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： 程序设计综合课程设计**

**专业班级：计算机科学与技术1805班**

**学 号： U201814615**

**姓 名： 于祯奇**

**指导教师： 李丹**

**报告日期： 2020年3月30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**

[任 务 书 I](#_Toc36432492)

[1 引言 5](#_Toc36432493)

[1.1 课题背景与意义 5](#_Toc36432494)

[1.1.1 背景 5](#_Toc36432495)

[1.1.2 意义 5](#_Toc36432496)

[1.2 国内外研究现状 6](#_Toc36432497)

[1.3 课程设计的主要研究工作 6](#_Toc36432498)

[2 系统需求分析与总体设计 8](#_Toc36432499)

[2.1 系统需求分析 8](#_Toc36432500)

[2.2 系统总体设计 8](#_Toc36432501)

[2.2.1 总体介绍 8](#_Toc36432502)

[2.2.2 具体介绍 9](#_Toc36432503)

[3 系统详细设计 11](#_Toc36432504)

[3.1 有关数据结构的定义 11](#_Toc36432505)

[3.1.1 待处理数据 11](#_Toc36432506)

[3.1.2 数据结构设计 11](#_Toc36432507)

[3.2 主要算法设计 14](#_Toc36432508)

[3.2.1 SAT求解模块 14](#_Toc36432509)

[3.2.2 二进制数独模块 16](#_Toc36432510)

[4 系统实现与测试 17](#_Toc36432511)

[4.1 系统实现 17](#_Toc36432512)

[4.1.1 系统实现环境 .17](#_Toc36432513)

[4.1.2 数据结构定义 17](#_Toc36432514)

[4.1.3 相关常量定义 18](#_Toc36432515)

[4.1.4 相关模块函数说明 18](#_Toc36432516)

[4.2 系统测试 24](#_Toc36432517)

[4.2.1 主控、交互与显示模块 24](#_Toc36432518)

[4.2.2 SAT求解模块 25](#_Toc36432519)

[4.2.3 二进制数独模块 31](#_Toc36432520)

[5 总结与展望 36](#_Toc36432521)

[5.1 全文总结 36](#_Toc36432522)

[5.2 工作展望 37](#_Toc36432523)

[6 体会 38](#_Toc36432524)

[参考文献 39](#_Toc36432525)

[附录2（Main） 44](#_Toc36432526)

[附录3（CreateCNF） 60](#_Toc36432527)

[附录4（Export） 63](#_Toc36432528)

[附录5（DPLL） 68](#_Toc36432529)

[附录6（BetterDpll） 80](#_Toc36432530)

[附录7（Choose） 93](#_Toc36432531)

[附录8（Tools） 95](#_Toc36432532)

[附录9（PuzzleSolve） 97](#_Toc36432533)

[附录10（系统使用说明） 122](#_Toc36432534)

# 1 引言

## 1.1 课题背景与意义

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。虽然SAT问题看似简单，但其实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，在多个领域具有非常重要的理论和实践意义。同时SAT问题也是程序设计与竞赛的经典问题。

### 1.1.1 背景

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。每年可满足性理论和应用方面的国际会议都会组织一次 SAT 竞赛以求找到一组最快的 SAT 求解器，而且会详细展示一系列的高效求解器的性能。2003 年的 SAT 竞赛中，就有 30 多种解决方案针对从成千上万的基准问题中挑选出的一些 SAT 问题实例同台竞争。国内也经常会组织一些 SAT 竞赛及研讨会，这些都促进了 SAT 算法的飞速发展 [1]。

### 1.1.2 意义

SAT问题是计算机科学领域和人工智能等领域中的重要研究对象。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。由于 SAT 问题是 NP 完全问题，它如果能够得到高效解决，那么一定可以高效地解决所有其它 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 完全问题都能够在多项式时间内转换为可满足性问题。所以，如果能找到求解SAT问题的有效算法，那么所有其它的 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]。

## 1.2 国内外研究现状

Bart Selman 和 Henry Kautz 分别于 1997 年和 2003 年在人工智能第五届国际合作会议上提出了 SAT 问题面临的十大挑战性问题，并在2001年和2007年先后对当时的可满足性问题现状进行了全面的阐述和总结。这十大挑战性问题的提出对于 SAT 基准问题的理论研究和算法改进都起到了强有力的推动作用。

最经典的求解 SAT 问题的完备算法是 DPLL 算法，它是由 Davis 和 Putnam等人在 1960 年提出，其它的完备算法大都是在 DPLL 算法的基础上衍生出来的，是对 DPLL 算法的改进。由于 SAT 问题本身的特性使得其最坏情况下的时间复杂度是指数级别，最初这使得许多的研究者望而却步。

而后，S.A.Cook 在 1971 年证明了 SAT 问题是 NP 完全问题，这更加削弱了许多学者研究 SAT 问题的兴趣，从而导致了 SAT 问题在很长的一段时间里都没有得到较好的重视，发展非常缓慢，研究成果较少。但是 1996 年以后，很多国家都相继举办了一些 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。

这些新兴的算法大都是基于 DPLL 算法的改进算法，改进的方面包括：采用新的数据结构、新的变量决策策略或者新的快速的算法实现方案。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的子句加权WSAT 算法，2000年金人超和黄文奇提出的并行 Solar 算法，2002 年张德富提出模拟退火算法[1].

## 1.3 课程设计的主要研究工作

本次课程设计的主要研究工作包括：

（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，并且初步掌握了DPLL算法思想。

（2）比较各种数据结构和设计方案的优劣，选取较为合适的数据结构和方案，并实现该方案，实现SAT问题求解模块。

（3）在初步完成后选取优化选取方案、添加子句学习、非时序回溯等合适的优化方法进行优化。

（4）将优化后的算法应用到二进制数独的求解问题。挖掘二进制数独求解问题中存在的约束条件，通过编码转换为 CNF 公式的形式后，输入到优化后的DPLL 求解器中进行求解，并最终转换为对应的二进制数独答案。

（5）设计随机生成不同阶数的二进制空盘的方案，供求解器求解。

# 2 系统需求分析与总体设计

## 2.1 系统需求分析

本次课程设计要求能通过系统正确求解输入的SAT问题，具体需要实现的功能包括：

（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。

（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，记录DPLL过程执行时间，并作为输出信息的一部分。

（5）SAT应用：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。

总的来说，本次课程设计主要是需要自主设计数据结构保存读取到的cnf算例文件，并基于DPLL算法初步实现SAT问题的求解，并有较为完善的输入和输出功能，构成求解SAT问题模块，再通过阅读文献寻找更优秀的办法来优化自己的求解模块，使其计算速度更快。

除此之外，还需要加入二进制数独问题的求解模块，将二进制数独的求解问题转化为SAT问题，并使用SAT求解模块的相关功能进行求解。并且能够通过拉斯维加斯算法随机生成终盘棋局，利用挖洞算法将生成的终局挖洞生成拥有唯一解的空盘。

## 2.2 系统总体设计

根据需求，相应设计介绍如下。

### 2.2.1 总体介绍

本系统共分为3个大模块，其中包含7个小模块，其中包括：

主控、交互与显示模块。

SAT求解模块： cnf公式输入解析模块、解析cnf公式正确性输出检查模块、DPLL求解模块、优化DPLL模块。

二进制数独模块：手动输入转化模块、电脑随机生成转化模块。

系统模块结构图见图2.1所示。

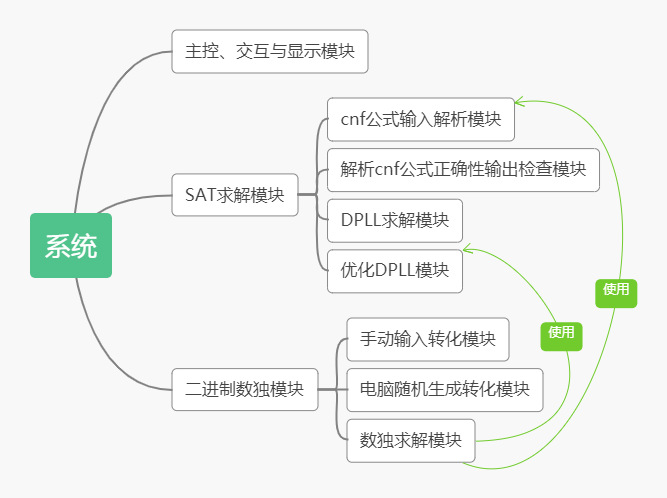


图2.1 系统模块结构图

### 2.2.2 具体介绍

（1）主控、交互与显示模块

该模块主要用于输出提示信息，完成与用户之间的交互。

（2）SAT求解模块

顾名思义就是用来求解整个SAT问题的模块，其中：

① cnf输入解析模块是用来输出菜单等提示信息，提示用户输入文件名称，打开该文件并且逐个读取解析文件中的关键信息，输入到已经设计好的数据结构中储存。

② 解析cnf公式正确性输出检查模块是遍历内部结构，逐行输出与显示每个子句，并与原文件进行对比，查验解析是否正确。

③ DPLL求解模块是基于DPLL算法思想框架，以及多个函数功能的调用，逐级回溯，不断尝试，求解SAT问题，并对于求解结果生成文件进行保存。

④ 优化DPLL模块就是对于原本DPLL的优化，通过优化选取策略、数据的存放形式 ，以及加入了非时序回溯和子句学习功能，降低了使用该模块所耗费的计算时间。

（3）二进制数独模块

该模块主要用于将实际的二进制数独问题转化为SAT问题，并求解，其中：

① 手动输入转化模块用于用户输入自己已有的二进制数独问题，该模块将其转化为SAT问题，并将生成的cnf公式保存到新的cnf文件中，文件名为用户所输入的文件名。

② 电脑随机生成转化模块即使用了拉斯维加斯算法随机生成终局，并利用挖洞算法将生成的终局变为空局。在生成空局后将其转换为SAT问题，并将生成的cnf公式保存到新的cnf文件中，文件名为用户所输入的文件名。

③ 数独求解模块使用了SAT模块中cnf输入解析模块，以及优化DPLL的相关功能，直接读取数独游戏所转化而成的cnf文件，求解相应的SAT问题，并最后将求解结果输出，并生成文件进行保存。

# 3 系统详细设计

## 3.1 有关数据结构的定义

### 3.1.1 待处理数据

在本次实验中，SAT求解模块需要处理的数据归根结底就是SAT问题所包含的cnf公式，而cnf公式中又包含子句、文字等数据元素。子句的数据项包含：1、组成该子句的文字，用整形表示，并存放在数组中；2、该子句的长度，即子句中的文字数量。而文字的数据项主要是文字的取值，有真、假、未赋值、三种形式，分别由整形1、0、-1表示。同时在之后优化DPLL算法时，为了方便非时序回溯和字句学习，我还在文字这一数据元素中添加了一些数据项，包括当前文字的决策层，导致该决策的子句等。

而对于二进制数独模块而言，所需要处理数据便是将数独问题转化而成的cnf公式，这些公式是根据其本身的三条约束加上空盘所代表的数据组成，是由整形表示的数组成的子句集构成。

### 3.1.2 数据结构设计

（1）SAT问题求解

在这个模块数据结构设计方面，我采用了邻接表作为主要数据结构，原因是这种数据结构比较熟悉，不容易出现问题，同时又能比较清楚的查找到想要的数据。根据上述待处理数据以及初步设计思想，我设计定义了如下结构体：

主体部分CNFType中包含一个结构类型指针Letter，用来分配动态分配空间变成连续的数组形式，可以用下标直接查找到文字，适用于所有的cnf公式；还包含了cnf中的子句数ClaNum和文字数LitNum，方便计算。

而LitNode是文字结构，里边包含文字当前赋值value；文字决策层数Deep；导致文字决策的子句指针CLA；链向下一个包含该文字的第一个子句结点FirstCla。

ClassNode是子句结构，其中包含子句数组clause，该数组的0号位置放置当前子句剩余文字数量，其他位置放置该子句所包含的文字；子句中原本的文字数量ClaNum；指向下一个子句的指针NextCla。

值得一提的是，在这个数据结构设计中，我希望给每个子句数组只分配了一个空间，即包含不同文字所链的子句单链表中相同的子句，是指向了同一个空间的。这导致了我只要改变其中一个空间，剩下的子句链表我就不用改变，这样做虽然导致我在可能复制与释放空间等操作时需要判断该子句是否已经操作过，但好处是简化了我修改子句的工作量。

整体数据结构的相互之间关系如图3.1所示。

（2）二进制数独模块

该模块的数据结构主要是能够有效存储由二进制数独问题转化为的cnf公式，所以可以直接定义数据长度。所以采用单链表的形式，将所有的子句都形成结点串到一起，每个子句结点需要包含当前子句数组，包含了子句的所有文字，和指向下一个结点的指针。

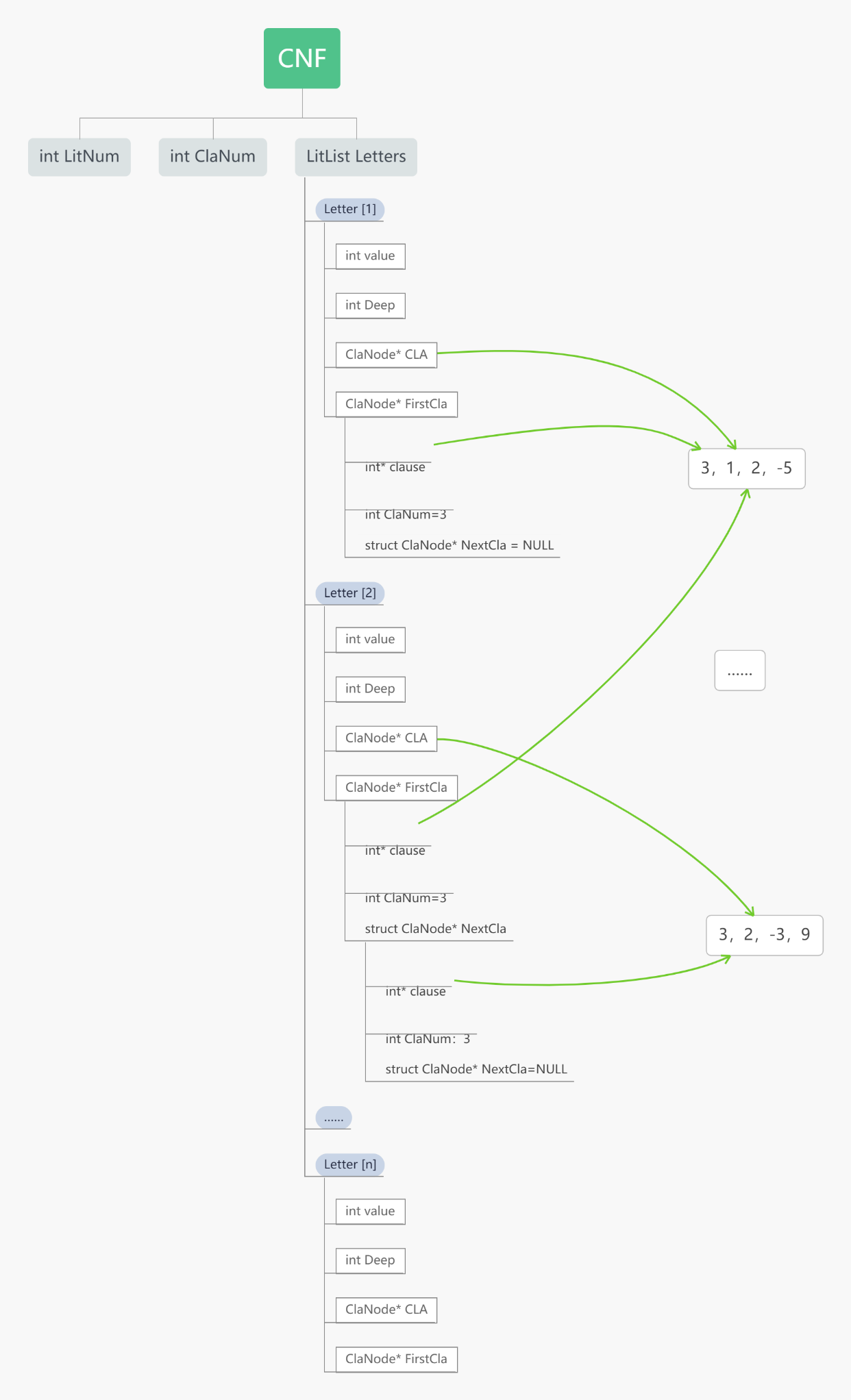


图3.1 数据结构的相互关系图

## 3.2 主要算法设计

主控、交互与显示模块在进入系统后直接显示菜单界面，并根据用户的输入显示相关的提示语，并输出相应的结果供用户选择，在此处就不再赘述。

### 3.2.1 SAT求解模块

（1）cnf公式输入解析模块

在cnf求解模块中，根据用户输入的文件名称打开目标文件，并利用fscanf函数逐个读取文件中的字符，直到遇到字符‘p’，若字符‘p’的后面是‘cnf’则证明下一个读取到的应该是文字数目，利用scanf公式读取文字数目，放入CNF结构的LitNum中并根据文字数目，动态分配空间，由于需要保证每个文字的正负，2\*LitNum+1个元素；再读取子句数目，放入ClaNum中。

接下来先申请子句结点空间，逐个读取整形数据，放到临时数组中，直到遇到数字0，代表当前子句结束，统计子句中的文字数目，并根据子句数目动态分配整形数组，并放入数组首元素和ClaNode结构的ClaNum中，将临时数组中保存的文字转移到分配的数组中，并逐个将该数组的地址放在每个子句的文字结构的clause中，即包含该子句的文字结构中的clause共用一个存储空间。

当一个子句完全操作结束后，接着读取下一个子句，循环重复上述操作，共循环ClaNum次，直到读取全部子句。

（2）解析cnf公式正确性输出检查模块

遍历文字数组，在每个文字结构中逐个输出其clause中的文字值，但是要注意该子句可能已经被输出，所以要通过函数判断，该函数的功能是判断clause数组中的最小的元素与当前文字数组所在的下标是否一致，因为如果当前文字数组的下标小于clause中最小的元素，说明在以该元素为下标的文字结构中已经输出了该子句，此处就不用再重复输入。

（3）DPLL求解模块

该模块首先判断所有子句中有没有单子句存在，判断方式是遍历每个文字结构，查看其中clause[0]，也就是子句数组的标志位的值，该标志位放置的是当前子句的文字数量，若clause[0]为1，说明当前子句为单子句。

在找到单子句后，找到该单子句文字的文字结构，并将其子句链表的中的每个子句的标志位置变为-1，即删除所有包含该文字的子句；并使其文字结构中的value为1，代表该文字已被赋值为真。除此之外，还需要将该文字的反文字结构中的value置0，代表该文字为假，并遍历其子句链表，找到其子句，即clause，并且遍历该数组，找到该反文字，并使该位置置零，统计每个子句中反文字的个数，将标志位clause[0]的个数减少，使其等于当时clause中非0的个数。若依旧存在单子句并在此判断是否存在空子句以及是否整个cnf公式是否为空，判断方法为遍历整个公式，若发现有clause[0]为0，则说明有空子句，返回ERROR；若所有子句的clause[0]均为-1，则说明该cnf公式为空。接着重新检索公式集，检查公式中是否存在该子句，若存在，则重复上述过程。以上过程称为布尔约束传播，即BCP。

接下来若当前没有单子句，则利用选取规则选取出一个文字v，使其为单子句，在DPLL函数中的选取方式是选择第一个没有被选取的文字作为单子句，并递归调用DPLL模块化简公式集。但是在调用之前，为了保证有序回溯，所以需要复制当前cnf公式集F，将其副本S保留，让F本身向下传播。当遇到错误返回值FALSE时，需要将选取出的文字v翻转，将副本S和反转后的-v向下传播；若遇到正确返回值TRUE，则说明当前结果是正确的，需要将结果放入外部数组中带出。

在主函数调用的过程中，也会使用time.h中所附带的函数对于DPLL模块的求解时间进行统计，并和结果一起输出到相应的文件中。

DPLL算法大致结构如图3.2所示。

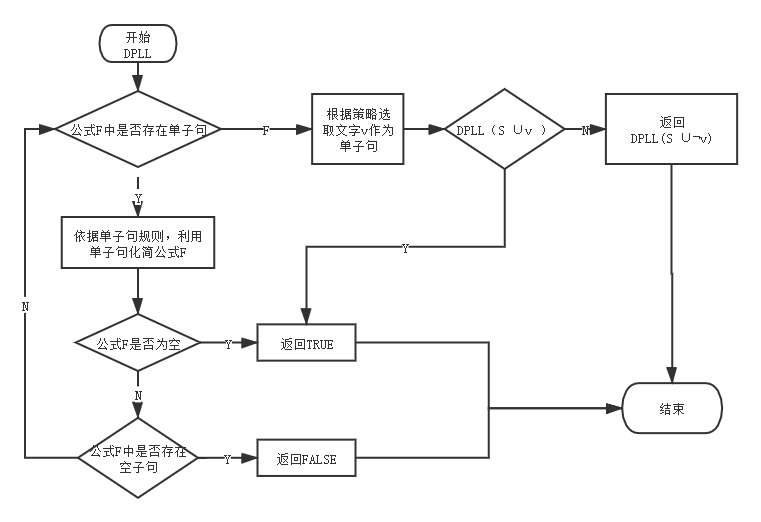


图3.2 DPLL算法结构图

（4）优化DPLL模块。

在原来DPLL模块的基础上，首先我添加了选取策略，基于我的数据结构的优势，我选取了JW选取办法，即最短子句出现频率最大优先，该算法最重要的是求解出和文字l 相对应的 J 值。最终会选取使得 J(l)取值最大的文字l 将其赋值为真，求解公式如下所示：

(3.1)

式中，l表示的是文字，ni表示的是包含文字l的子句Ci的长度（i=1,2,…,m,假设共有m个子句）。那么Ci的长度越长，相对而言，J 值便会越小[1]。采用这种选取办法是因为式中的ni就是数据结构中的标志位clause[0]，只需要直接查找即可，不需要再进行统计，增大查找效率，并且JW算法相比于其他选取方法拥有着更高的正确性，是一种高精度的计算方式，除此之外它还特别适用于长短不一致的cnf公式，对于二进制数独转化而成的问题有着更好的效果。

在加入了JW选取办法后，有些问题的求解效率得到了增强，但是有些问题却减缓了速度，于是我便加入了用户可以自己选择选取方式，使用户的求解时间达到最小。

除此之外，我还在该模块中加入了子句学习与非时序回溯的功能，能使DPLL算法的求解效率变得更高。与上面选取办法类似，在优化过程中，我还能够让用户自己选取算法，防止对于有些问题时序回溯能够更快解决。

### 3.2.2 二进制数独模块

（1）手动输入转化模块

在该模块中需要将用户输入的数独文件按照三个约束规则以及已知信息生成相应的cnf公式文件。

（2）电脑随机生成转化模块。

在该模块中需要先利用拉斯维加斯算法生成二进制数独终局，在利用挖洞算法将生成的终局挖成空局，在利用和手动输入转化模块相似的功能，将二进制棋局转化为cnf公式并保存到相应的文件中。

（3）数独求解模块

调用SAT求解模块中的功能，读取已经保存的cnf文件并进行求解，并将求解结果输出到屏幕上以便于检查，并且同样将结果保存到文件中。

# 4 系统实现与测试

## 4.1 系统实现

### 4.1.1 系统实现环境

（1）硬件环境

处理器：Inter Core i7-8750H @2.20GHz 六核

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

机带RAM：8.00GB（7.85GB可用）

（2）开发环境

开发软件：Microsoft Visual Studio Community 2019 版本 16.3.3

### 4.1.2 数据结构定义

根据3.1节的相关设计，我定义了如下数据结构

（1）SAT求解模块数据结构

1. /\*--------------------CNF公式的存储表示-----------------\*/
2. **typedef** **struct** ClaNode {
3. **int**\* clause = NULL;//包涵该文字的子句,以数组表示数据
4. **int** ClaNum = 0;
5. **struct** ClaNode\* NextCla = NULL;//指向下一个子句
6. }ClaNode;
8. **typedef** **struct** Litnode {
9. **int** value = Undefine;//该文字的真值，1为真，0为假，-1为未赋值
10. **int** Deep = 0;//文字的决策深度，用于冲突学习
11. ClaNode\* CLA = NULL;
12. ClaNode\* FirstCla = NULL;//指向包含该文字的第一条子句
13. }Litnode, \* LitList;
15. **typedef** **struct** CNFType {
16. LitList Letters = NULL;//文字数组，动态分配空间
17. **int** LitNum = 0, ClaNum = 0;//公式中的文字数和子句数
18. }CNF;
19. /\*----------------------end------------------------\*/

（2）二进制数独求解模块数据结构

1. /\*----------二进制数独转化成的SAT公式存储结构--------\*/
2. **typedef** **struct** P\_ClauseNode {
3. **int** ClauseValue[10] = { 0 };//子句数组
4. P\_ClauseNode\* NextClause = NULL;//指向下一个子句
5. }P\_ClauseNode, \* P\_ClauseList;
7. **typedef** **struct** SATFormula {
8. P\_ClauseList FirstClause = NULL;//指向第一个子句
9. **int** LitNum = 0;//子句的文字数目
10. **int** ClaNum = 0;//子句的子句数目
11. }SATFormula;
12. /\*----------------------end------------------------\*/

### 4.1.3 相关常量定义

根据系统需求，相关常量定义如下

1. #define TRUE 1
2. #define FALSE 0
3. #define Undefine -1
4. #define OK 1
5. #define ERROR 0
6. #define OVERFLOW -2
7. #define DELETE -1
8. #define MaxLearnClaNum 31

### 4.1.4 相关模块函数说明

**（1）cnf公式输入解析模块**

函数声明：status CreateAdjacencyList(char\* filename, CNF& F)

功能：读取以filename为文件名的cnf文件，并将其解析保存到结构F中

相关说明：在函数中用fopen函数以只读的形式打开文件，若没有该文件则返回ERROR。打开后按照3.2.1（1）中所描述的方法，完成功能。在其中用到了函数NotSameNum，此函数的功能为验证数组N中在第n个数之前有没有与其相等的数，调用此函数可以使若一组子句有两个相同的文字，则只添加一遍。若最后全部添加完毕，返回OK。

**（2）解析cnf公式正确性输出检查模块**

函数声明：void ExportCNF(CNF F)

功能：在屏幕上输出已经解析完成的子句。

相关说明：遍历整个邻接表，输出每个没有输出过的文字结构中的clause数组。在完成此功能调用了函数MinNum。此函数的功能是是根据传入的数组，返回数组中最小的值。若遇到负文字，则将负文字转化为其相应在文字链表中的下标，再输出整体最小的值。将MinNum的返回值与当前文字下标作比较，若与当前下标相等，则说明当前数组没有被输出，若不等则说明已经被输出过，此处就不再进行输出。

**（3）DPLL求解模块**

函数声明：void BCP(CNF& F, int i)

功能：根据传入的单子句i，实现布尔约束传播。

相关说明：根据3.2.1（3）中所描述的布尔约束规则实现相关功能。

函数声明：status Choose\_1(CNF F)

功能：第一个选取策略。

相关说明：在CNF结构中找到第一个没有赋值的文字，并返回该文字的值。

函数声明：status Choose\_2(CNF F)

功能：第二个选取策略。

相关说明：按照3.1.2（4）所设计的方法，逐个统计value值为-1，即没有赋值的文字结构中的clause[0]，并根据公式3.1计算该文字的J值，返回所有文字中J值最大的文字。

函数声明：void Copy(CNF& S, CNF F)

功能：将当前cnf结构F复制到cnf结构S中，作为副本保存。

相关说明：与cnf公式输入解析模块相似，将F中所有文字和没有被删除的子句的所有数据项按照顺序放在公式S中。但是在放入的时候要剔除子句中所有已经删除的文字，即子句中所有的0不复制。

函数声明：status ExistSingle(CNF F)

功能：检查F中是否存在单子句，若存在则返回单子句值，若不存在则返回FALSE。

相关说明：检查当前cnf公式F，按照3.2.1（3）设计的方法寻找单子句，利用循环结构逐个遍历，若存在某一个下标为n的文字结构的clause[0]为1，则返回n，若遍历整个文字数组都不存在，则返回FALSE。

函数声明：status EmptyCNF(CNF F)

功能：检查F是否为空。

相关说明：利用循环结构逐个遍历，若存在某一个下标为n的文字结构的clause[0]为-1，则说明该CNF公式不为空，返回FALSE，否则返回TRUE

函数声明：status EmptyCla(CNF F, int Key)

功能：检查F中是否存在空子句

相关说明：检查文字数组下标为Key的文字结构的clause[0]是否为0，即检验该子句是否为空，若为0，则说明F中存在空子句，返回TRUE，否则返回FALSE。

函数声明：void SATSove\_FileSave(char\* filename, int s, long t, int NUM)

功能：将SAT问题求解后的答案放入文件名为filename的文件中。

相关说明：用fopen以‘w’的形式打开filename为文件名的文件，并先放进字符‘s’，再放进s的值，再放进字符‘v’，在之后开始放入每个文字的解，若为真，则该文字为正，若为假，则文字为负值。

函数声明：status DPLL(CNF F, int i)

功能：用DPLL思想框架求解

相关说明：函数传入的便是公式F和当前寻找到的单子句i，当i不为FALSE时，则先调用BCP函数对F进行化简。再调用EmptyCNF，若EmptyCNF函数返回值为TRUE，则说明已经求解成功，将每个文字的value值放入外部变量VALUE中，并且DPLL函数返回TRUE。若返回值为FALSE则调用EmptyCla函数，若返回值为TRUE说明当前路径已经求解失败，DPLL函数返回FALSE，否则就调用ExistSingle函数查找单子句，并把返回值赋给i，继续循环。若遇到i的值为0，则说明，当前公式F中没有单子句存在，并且还没有成功找到解。则需要调用选取策略选取一个文字v作为单子句。调用Copy函数复制当前公式F，其副本为S，递归调用F和选取的文字v进入下一级DPLL。若下一级DPLL返回值为TRUE，则说明当前已经找到解，删除副本S并返回TRUE。否则说明当前路径错误，递归调用S和-v进入下一级DPLL，若返回值为TRUE则删除副本S，返回TRUE，否则删除副本S，返回FALSE。在求解之后将答案利用SATSove\_FileSave保存在文件中。

**（4）优化DPLL模块**

函数声明：void SearchConflic(CNF F, ClaNode\* P, int key)

功能：根据文字结构中的CLA回溯查找冲突发生的原因，并记录蕴含图根源的文字。

相关说明：该函数是一个递归调用自身的方法，再查找单子句的时候，会将导致单子句为真和假的子句放入该文字结构的CLA中，这样若CLA为NULL，则说明是由选取策略选取而成，是导致冲突的原因。通过递归逐个访问CLA中处理自身大的所有文字的反文字，即可找到根源，并将其放入外部数组ConflicCla中带出。

函数声明：status BetterEmptyCla(CNF F, int Key)

功能：优化后的查找有无空子句函数，是字句学习和非时序回溯的重要部分。

相关说明：再优化后的查找空子句函数中，若找到空子句则调用SearchConflic函数查找冲突，并将其添加进代表学习子句集的外部数组LearnCla中，并且通过比较确定冲突发生地点，返回冲突决策层数与现在决策层数的差，便于非时序回溯。

函数声明：void AddLearnCla(CNF& F)

功能：将学习子句集中的子句添加到公式集F中。

相关说明：逐个读取学习子句集中的子句，并创建新的空间，将子句中的文字复制到新的空间中，并将其添加到公式集中。

函数声明：void BetterBCP(CNF& F, int i, int Deep)、void BetterCopy(CNF& S, CNF F)

功能：根据优化DPLL的结构，调整了相关函数。

相关说明：函数总体的执行功能没有差异，只不过在优化DPLL中根据其不同的结构进行了相应调整。

函数声明：status BetterExistSingle(CNF F)

功能：查找单子句，并将找到的单子句的原本的子句，放入该单子句文字的CLA中。

相关说明：和上述ExistSingle函数一样，都是用来寻找公式集中的单子句的函数。不同的是在此函数在中，查找到单子句后需要将其原本的子句放入CLA中，便于BetterEmptyCla函数查找冲突。

函数声明：status BetterDPLL(CNF& F, int i, int Deep)

功能：优化后的DPLL算法求解问题，加入了非时序回溯和字句学习功能。

相关说明：在此函数中，大体结构与DPLL函数类似，只不过实在相应的函数名前加上Better，即代表优化后的函数，再次就不再赘述。不同的是在调用决策函数选择单子句之前，需要先调用AddLearnCla函数将学习子句集中的子句添加到子句集F中，再进行决策。还有若查找失败，则返回值也不是FALSE，而是负的层数差异。即若递归调用自身时的返回值小于0，则返回返回值加一，代表返回上一层，直至返回值为0，则代表到达目标返回层。若返回值为TRUE则没有变化。

**（5）手动输入转化模块**

函数声明：void Constraint\_1(SATFormula& Formula, int\*\* B、void Constraint\_2(SATFormula& Formula, int\*\* B)、void Constraint\_3(SATFormula& Formula, int\*\* B)

功能：根据约束1、2、3将二进制数独的基本规则转化为子句形式，并添加在结构SATFormula中。

相关说明：二进制数独的三个约束为：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；2、在每一行、每一列中1与0的个数相同；3、不存在重复的行与重复的列。根据着三个规则和任务书中的相应公式，将其转换为cnf公式，并存储在Formula中。

函数声明：int\*\* Prepare()

功能：准备一个二维数组，在调用函数约束时使用。

相关说明：在函数中根据用户所输入的数独阶数生成一个对应的二维数组，将下标所对应的空间放入下标的值，例如：B[1][1]中放入的是11，B[3][2]中放入的是32等。最后返回创建号的二维数组。

函数声明：void Convert(SATFormula& Formula)

功能：将公式中的ij形式的文字转化为标准形式，以便求解功能使用

相关说明：在进入函数后依次读取每个文字的值，将其按照数据位数划分，并按照2、3、4、5位由小到大的顺序依次排列，将Formula中的文字转换为标准模式。

函数声明：void AddClause(int\* Subset, int n, SATFormula& Formula)

功能：将Subset数组中的前n个数作为一个子句添加到Formula中

相关说明：根据n的长度动态分配空间，将Subset数组的前n个元素放入分配的空间中，并在结束之后放入结束标志0。

函数声明：status PuzzleSAT\_FileSave(SATFormula Formula, char\* filename)

功能：将Formula中的子句按照规则保存在以filename为文件名的文件中。

相关说明：以‘w’方式打开filename为文件名的文件，先放‘p’，‘cnf’等标准格式，并利用fprintf文件将Formula结构中的所有元素逐个放入，并且每个子句之后都放入标志0。

本模块数没有模块调用函数，直接在main函数的相应位置调用Prepare函数与三个约束函数，调用Printf函数逐个输入已知位置，并放入棋盘数组Puzzle中，并利用AddClause函数将每个位置作为一个单子句输入到Formula中。再利用转换函数Convert转换为标准形式，最后利用文件保存函数PuzzleSAT\_FileSave将文件保存。

**（6）电脑随机生成转化模块。**

函数声明：status Las\_vegas(SATFormula& Formula, int\*\* B, int\* Puzzle, char\* filename)

功能：利用该算法生成完整终局。

相关说明：本函数利用随机数随机填充数独的不同位置，填入数值也是由随机数生成。并利用求解模块求解该数独，若能得到解，则将解保存在Puzzle中返回OK，若不能则清空Puzzle棋盘返回ERROR。

函数声明：void DiggingHole(SATFormula& Formula, int\*\* B, int\* Puzzle,char\* filename)

功能：通过挖洞算法，根据终盘Puuzle生成空盘。

相关说明：挖洞算法是遍历棋盘的所有单元格，将当前单元格中元素值翻转，调用求解模块求解当前棋盘，若没有解，说明原来数独有唯一解，可以挖去，即将单元格内文字置为0。否则将单元格文字翻转复原。

在本模块中，与上个模块不同的是本模块调用了Las\_vegas函数和DiggingHole函数使得函数成功生成空盘，而不是用户从键盘中输入。除此之外剩下的都和手动输入转化模块一致，此处就不再赘述。

## 4.2 系统测试

### 4.2.1 主控、交互与显示模块

在此由于篇幅有限，只展示显示菜单部分，显示菜单分为整体菜单，SAT求解菜单，二进制数独菜单，以及创建数独菜单。各个菜单如图4.1~图4.4所示。

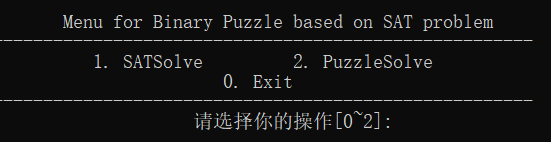


图4.1 整体菜单

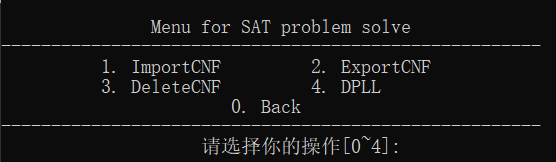


图4.2 SAT求解菜单

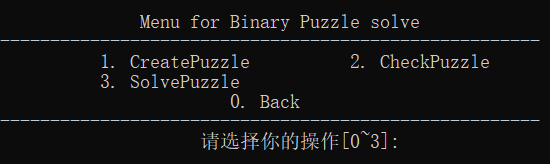


图4.3 二进制数独菜单

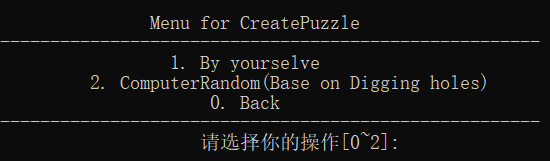


图4.4 创建数独菜单

### 4.2.2 SAT求解模块

（1）模块功能与设计目标

本设计要求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，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。

（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DPLL模块和优化DPLL模块进行求解并计时，计算优化率并记录。优化率的计算公式为：

(4.1)

式中t为未对DPLL优化时求解算例的执行时间，t0则为优化DPLL实现时求解同一算例的执行时间。

由于篇幅原因，在此只给出基准算例测试截图，剩余算例结果如表4.1所示。由于在屏幕上输出的原因，截图只包含测试结果和运行时间，不包完整解，完整解可使用压缩文件中的可执行文件运算得到，用到的测试算例在文件夹测试算例中。

基准算例测试如图4.5~图4.12所示

①功能测试中文件sat-20.cnf

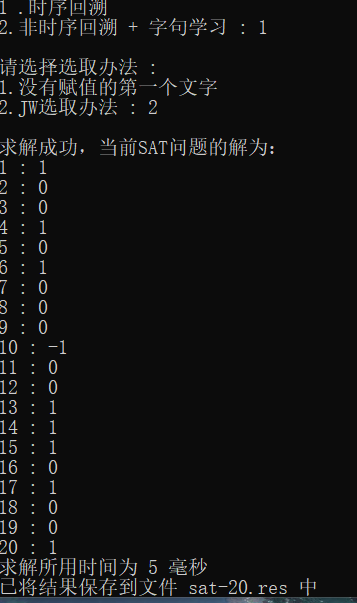


图4.5 DPLL模块求解sat-20.cnf

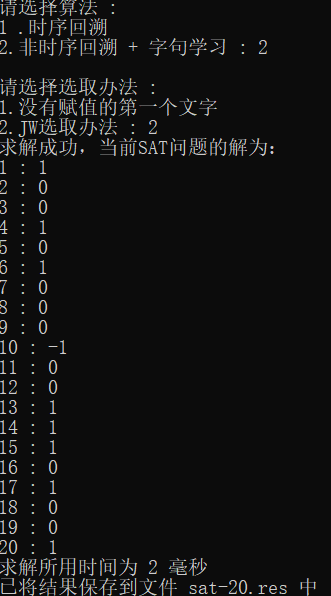


图4.6 优化DPLL模块sat-20.cnf

②功能测试中文件unsat-5cnf-30.cnf

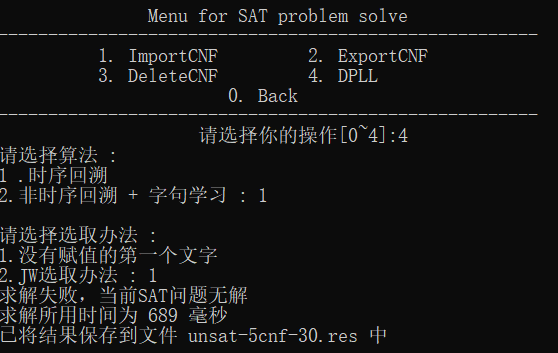


图4.7 DPLL模块求解unsat-5cnf-30.cnf

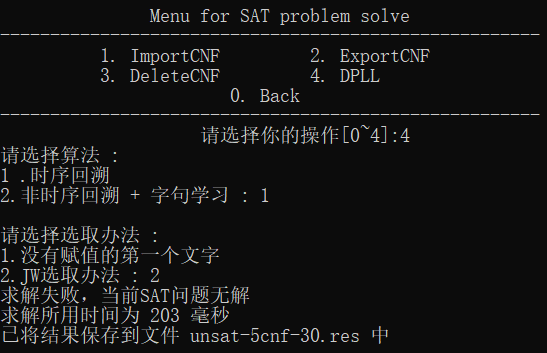


图4.8 优化DPLL模块求解unsat-5cnf-30.cnf

③性能测试中文件ais10.cnf

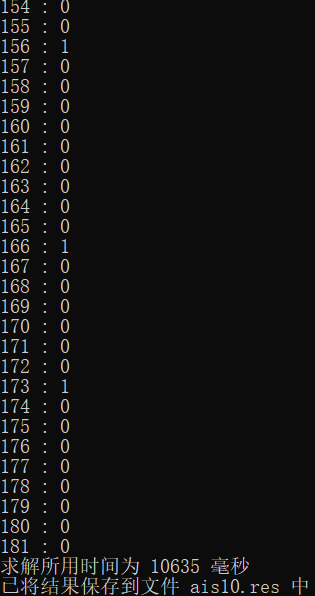


图4.9 DPLL模块求解ais10.cnf

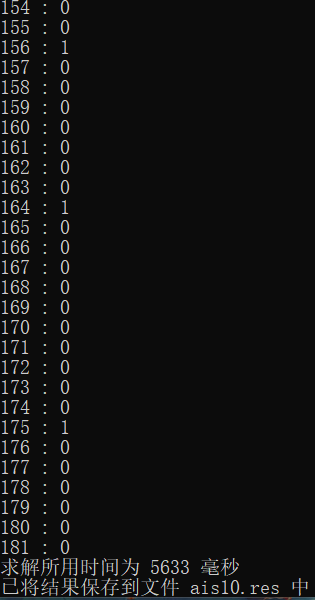


图4.10 优化DPLL模块求解ais10.cnf

④性能测试中文件sud000009.cnf

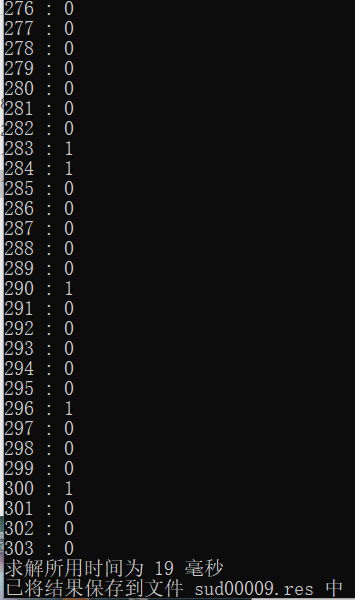


图4.11 DPLL模块求解sud000009.cnf

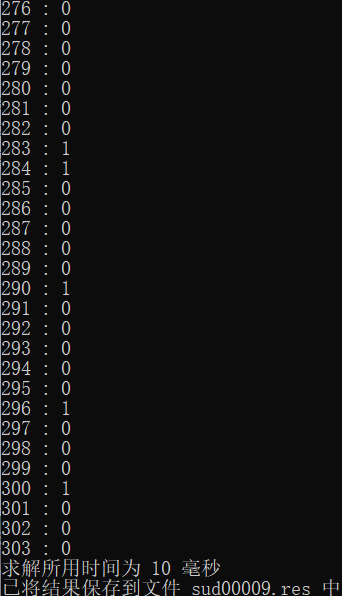


图4.12 优化DPLL模块求解sud000009.cnf

（3）总体测试结果

表4.1 测试结果

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 文件名称 | 文字数目 | 子句数目 | 文件规模 | 运算结果 | t/ms | t0/ms | 优化率 |
| Small\_1.cnf | 20 | 91 | 小 | 有解 | 3 | 1 | 66.66% |
| Small\_2.cnf | 20 | 1532 | 小 | 有解 | 270 | 45 | 83.33% |
| Small\_3.cnf | 100 | 340 | 小 | 有解 | 27393 | 172 | 99.37% |
| Small\_4.cnf | 50 | 80 | 小 | 有解 | 2036 | 2 | 99.90% |
| Small\_5.cnf | 50 | 100 | 小 | 有解 | 1268 | 6 | 99.52% |
| Middle\_1.cnf | 301 | 2708 | 中 | 有解 | 49 | 30 | 38.77% |
| Middle\_2.cnf | 303 | 2851 | 中 | 有解 | 15 | 9 | 40.00% |
| Middle\_3.cnf | 232 | 1901 | 中 | 有解 | 203 | 116 | 42.85% |
| Middle\_4.cnf | 301 | 2810 | 中 | 有解 | 52 | 40 | 23.07% |
| Middle\_5.cnf | 297 | 2721 | 中 | 有解 | 109 | 77 | 29.35% |
| Large\_1.cnf | 1075 | 3152 | 大 | 有解 | 1344 | 285 | 78.79% |
| Large\_2.cnf | 729 | 22060 | 大 | 有解 | 398 | 292 | 26.63% |
| Large\_3.cnf | 500 | 3100 | 大 | 有解 | 342 | 108 | 68.42% |
| Large\_4.cnf | 1464 | 6535 | 大 | 有解 | 137 | 103 | 24.81% |
| Large\_5.cnf | 606 | 2295 | 大 | 有解 | 4 | 4 | 00.00% |
| UNSmall.cnf | 80 | 370 | 小 | 无解 | 1 | 0 | 100% |
| UNMiddle.cnf | 172 | 560 | 中 | 无解 | 0 | 0 | 00.00% |
| UNLarge.cnf | 512 | 9685 | 大 | 无解 | 3980 | 2930 | 26.38% |

（4）测试结果分析

该求解器基本可以计算大部分中小型算例，但是依然有部分算例无法计算；再=在计算数独转化而成的算例有着较快的速度，经过非时序回溯和子句学习优化后的DPLL模块整体速度有比较明显提升，虽然个别算例会有负优化的情况，但是我认为那是正常现象相差不大，基本达到该模块设计目标。

### 4.2.3 二进制数独模块

（1）模块功能与设计目标

能够让用户自己输入任意偶数阶已有的二进制数独，也**能够随机生成6阶或8阶数独**，并且能够将二进制数独游戏问题转化为SAT问题，并使用SAT求解模块进行问题求解，生成的数独游戏可玩，**并具有唯一解**。

（2）测试大纲

先分别让用户输入一个6阶和8阶数独，并调用求解器进行求解，验证求解正确性，再让电脑随机生成一个6阶和8阶数独，验证随机生成数独正确性，以及是否具有唯一解。

（3）测试结果

①用户输入6阶数独

数独本身图像如图4.13所示，输入后数独如图4.14所示求解结果如图4.15所示。

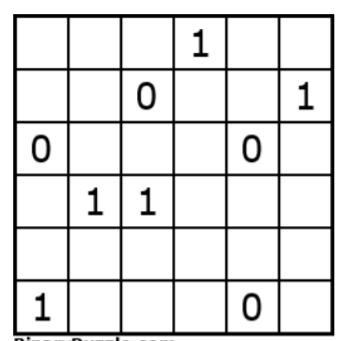


图4.13 已有6阶数独

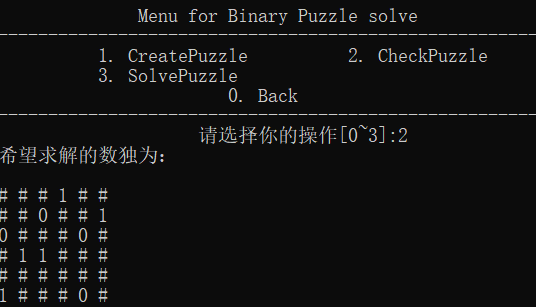
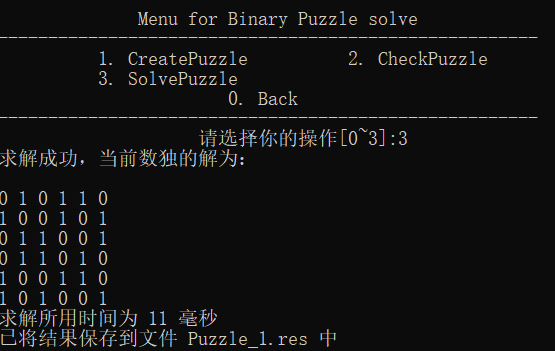


图4.14 输入6阶数独



4.15 输入6阶数独求解结果

②用户输入8阶数独

数独本身图像如图4.16所示，输入后数独如图4.17所示求解结果如图4.18所示。

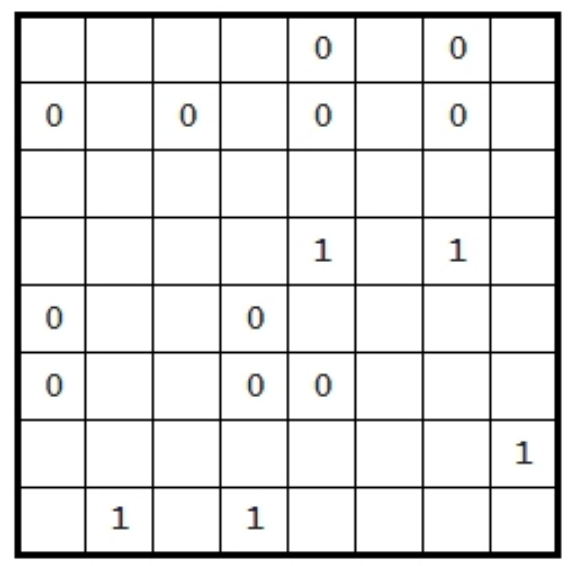


图4.16 已有8阶数独

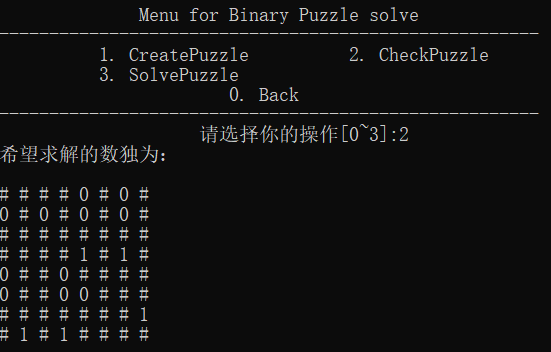
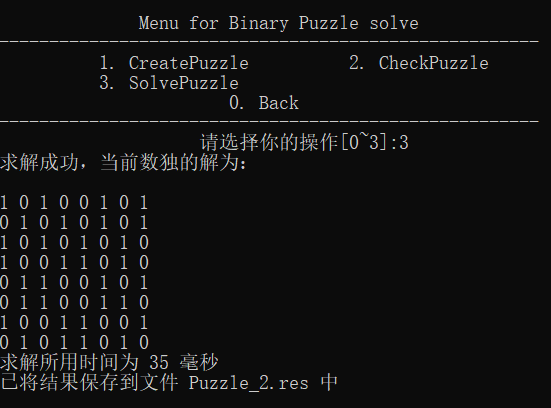


图4.17 输入8阶数独



4.18 输入8阶数独求解结果

③电脑生成6阶数独

电脑生成的6阶数独如图4.19所示，求解结果如图4.20所示。

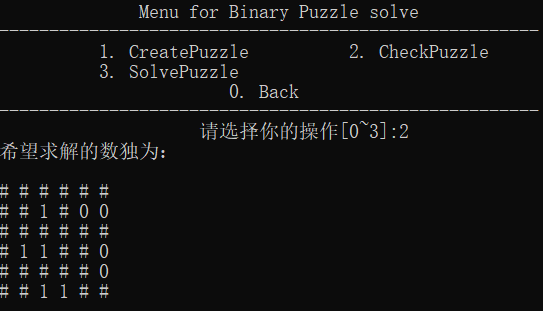
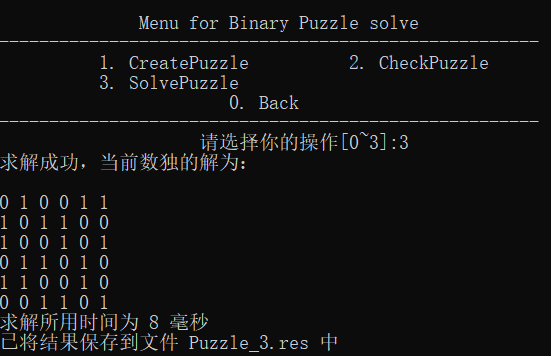


图4.19 生成6阶数独



4.20 生成6阶数独求解结果

③电脑生成8阶数独

电脑生成的8阶数独如图4.21所示，求解结果如图4.22所示。

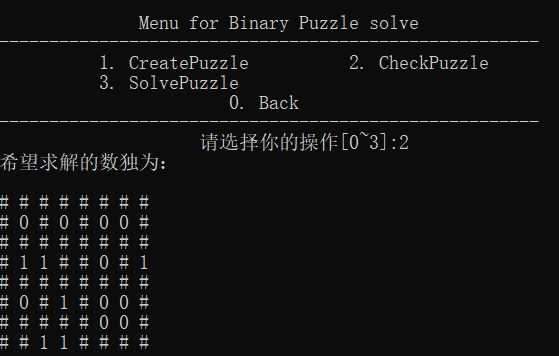


图4.21 生成8阶数独

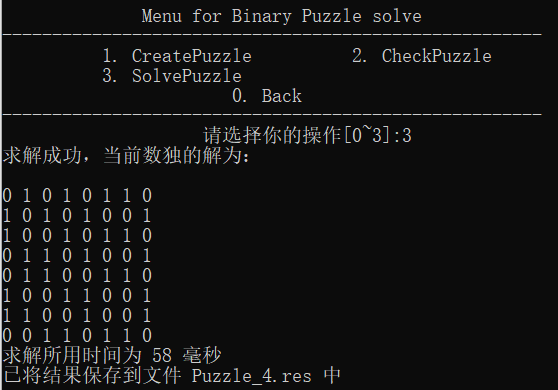


图4.22 生成8阶数独求解结果

（4）测试结果分析

能够成功输入用户输入的数独棋盘，并能正确转化为SAT问题，并保存到文件中，求解器能够正确的根据转化的cnf公式求解数独，并且能够很快的求解出答案并正确的输出，**同时也能够随机生成具有唯一解的数独**，并且调用求解器求解，比较好的达到了该模块设计目标。

# 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）在本次课程设计中，我在前期便比较快速的完成了数据结构的选取、相关算法的大致流程等前期工作，进入到准备设计与实践过程。虽然这样做给了我后期优化比较充足的时间，但是由于没有仔细阅读多篇论文，而导致选取的回溯方式和数据结构在后期发现并不是最有效的方式。

（2）但是由于前期的进展比较顺利，我很快就完成了文件的解析以及输出工作，并根据DPLL算法框架写出了初步的DPLL模块，写出来的代码也基本能够实现初步功能。这是令我比较骄傲和自豪的，

（3）在完成基本DPLL结构之后，我的课程设计的大部分时间都用来完成DPLL算法的优化，由于刚开始写出的DPLL模块对于一些比较困难的算例计算时间非常的长，导致我对于自己程序的信心不是很强。但是再后来了解到大部分同学的进程之后，我便重拾了信心继续优化自己的程序。

（4）在优化方面，一开始其实是抱着一些偷懒的想法，只想优化选取策略，而不改变DPLL算法本身，可是当我尝试过目前常见的所有选取办法后，发现其优化效率与自己一开始实现的JW选取方法类似，并没有什么很显著的效果，而且所提升的时间也不是很多，甚至有些算例还出现了负优化。在了解到其他同学的优化方案设计后，发现好多同学都在设计优化方案中写了非时序回溯和字句学习功能，于是我也下定决心加入这两个功能。

（5）这两个功能的加入可以说使我整个课程设计中耗费时间最长的地方，我首先阅读了相应的文献，认真了解了什么是冲突分析，什么是子句学习与非时序回溯。在初步了解后，我自己根据自己特殊的数据结构设计相关思想，并最终实现。由于该模块优化实在原本的程序结构的基础上的，所以有很多细小的问题并没有注意到，在最后调试的过程中造成了很大的麻烦，也曾经因为忘记修改循环判断条件而浪费了很长时间。让我明白了很微小的细节也能决定程序是否通畅完整。

（6）在花费了较长时间优化DPLL模块后，我成功的实现了子句学习和非时序回溯，这令我十分欣喜，虽然最后的结果有点低于我的预期，但是那种完成时的喜悦是难以言表的。

（7）在数独游戏方面，有了前面的实现基础，这部分只需要理解任务书上所给的转化方式，即可实现。我通过用户自己输入求解阶数，可以求解6阶和8阶的数独。同时我也完成了随机生成空盘供用户和系统解决，这也是我比较骄傲和自豪的一点。总体我认为数独模块实现的还是比较成功。

（8）美中不足的是，由于低估了报告的工作量，导致了时间不够用的问题，本次报告写的不是十分完美，在有些可以用流程图直观表述的地方，我却由于画图过于复杂的缘故，选用了不那么直观的文字描述，这在以后的学习中要加以改进。

## 5.2 工作展望

（1）在整体程序快要完成之际，我与同学交流时发现他的数据结构和回溯方法比我优秀许多，导致我优化后的求解速度比他没有优化之前还要快速。当我询问他的想法时，他说他在前期阅读了很多相关的文献，结合多篇文献的优点设计而成。这告诉我在开始实践之前一定要做好充足的准备工作，多阅读文献，再做出决定。

（2）因为本次课程设计的时间较长，需要分模块写程序，所以若不写好备注，并设计好变量的名称，很有可能在一段时间后就忘记了自己当时想好的思想。这让我明确了做好备注的意义。

（3）我会继续学习相关方向的内容，多了解当今领域的前沿技术，并持续学习，保持进步。

# 6 体会

在本次课程设计中，我运用到了前三个学期所学的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知识，也使用了C语言程序设计实验、数据结构实验两门编程所带来的实践经验。这些课堂上所学到的东西第一次融会贯通，完成一次比较大型的任务设计。

从前期的阅读文献到结构设计，再到中期的整体实现，最后到后期的调整，总结经验，撰写报告。这一系列过程使我了解到了一个程序设计的整体流程。在这个过程中，每一步都非常的重要，若要是拿建筑做比较，则前期的准备工作就好像是打地基，前期准备工作做得越充足，那么就相当于楼房的地基打的越深，建起的建筑就越稳固。

而在中期实现的过程中一定要认真仔细，在本次课程设计中，优化程序由于是直接基于原本的DPLL模块上面修改，所以一些代码我就不加思考，只想着功能差不多就拿来使用，然而出现问题后，debug时也不会重点检查和这些代码，而是检查自己新写的代码，最后发现问题就在这些代码和自己写的代码不匹配，浪费了很多时间。这告诉了我在检查时不仅要很细致，而且更重要的一点就是，不要轻易的使用别的代码来完成自己正在实现的功能，除非可以完全确定自己的使用是正确的，有的时候重写一遍会使自己对这个程序有着更深的了解。

在整个课程设计的实现过程中，我对C语言以及数据结构的知识有了更深的理解，也深刻明白了一个好的数据结构的重要性，同一个算法对于不同的数据结构而言执行效率也是截然不同的。同样我也遇到了很多问题，发现了自己知识的不足，在解决这些问题的时候也极大的提升了我的能力。

在此我也要感谢给过我支持的老师和同学们，他们的鼓励以及解答使我解决了很多问题，也看到了更加优秀的数据结构与算法，让我受益良多。除此之外我还要感谢所有将自己研究成果发表的前辈们，通过学习他们的知识和研究成果，我才能较为顺利的完成本次课程设计。

总体来说，本次课程设计完成的还是比较顺利，也基本达到了设计目标，能够求解SAT问题以及二进制数独问题。在以后的学习生活中我会更加努力的学习知识，并把它们应用到实际之中，让自己变得更加优秀。

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

**[3]** 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.

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

**附录1(head)**

#pragma once

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include<stdbool.h>

#include<time.h>

#include<math.h>

#include<string.h>

#define TRUE 1

#define FALSE 0

#define Undefine -1

#define OK 1

#define ERROR 0

#define OVERFLOW -2

#define DELETE -1

#define MaxLearnClaNum 31

typedef int status;

/\*--------------------CNF公式的存储表示-----------------\*/

typedef struct ClaNode {

int\* clause = NULL;//包涵该文字的子句,以数组表示数据

int ClaNum = 0;

struct ClaNode\* NextCla = NULL;//指向下一个子句

}ClaNode;

typedef struct Litnode {

int value = Undefine;//该文字的真值，1为真，0为假，-1为未赋值

int Deep = 0;//文字的决策深度，用于冲突学习

ClaNode\* CLA = NULL;

ClaNode\* FirstCla = NULL;//指向包含该文字的第一条子句

}Litnode, \* LitList;

typedef struct CNFType {

LitList Letters = NULL;//文字数组，动态分配空间

int LitNum = 0, ClaNum = 0;//公式中的文字数和子句数

}CNF;

/\*----------------------end------------------------\*/

/\*----------二进制数独转化成的SAT公式存储结构--------\*/

typedef struct P\_ClauseNode {

int ClauseValue[10] = { 0 };//子句数组

P\_ClauseNode\* NextClause = NULL;//指向下一个子句

}P\_ClauseNode, \* P\_ClauseList;

typedef struct SATFormula {

P\_ClauseList FirstClause = NULL;//指向第一个子句

int LitNum = 0;//子句的文字数目

int ClaNum = 0;//子句的子句数目

}SATFormula;

/\*----------------------end------------------------\*/

/\*---------------------函数声明--------------------\*/

status CreateAdjacencyList(char\* filename, CNF& F);//创建CNF公式

void ExportCNF(CNF F);//输出内部结构，比对正确性

status DPLL(CNF F, int i);//DPLL算法

void BCP(CNF& F, int i);//布尔约束传播

status Choose\_1(CNF F);//第一个选择策略

status Choose\_2(CNF F);//第二个选择策略，JW选择策略

void DeleteCNF(CNF& F);//删除公式

void SATSove\_FileSave(char\* filename, int s, long t, int NUM);//保存为.res文件

/\*--------非时序回溯 子句学习----------\*/

status BetterDPLL(CNF& F, int i, int Deep);//优化DPLL算法

void BetterBCP(CNF& F, int i, int Deep);//优化布尔约束传播

void Reversal(CNF& F);//翻转

status BetterExistSingle(CNF F);//验证F中有没有单子句存在，有则返回其值，没有返回FALSE

status BetterEmptyCla(CNF F, int Key);//验证是否有空子句

void BetterCopy(CNF& S, CNF F);//优化函数使用的复制函数

void SearchConflic(CNF F, ClaNode\* P, int key);

void AddLearnCla(CNF& F);//将学习子句添加到F中

/\*-------工具函数-------\*/

int MinNum(int\* N, int m);//找到数组N包含的 “ 最小值 ”

int BetterMinNum(CNF F, int ClaNum, int\* N);//优化函数使用的查找数组最小值

void Copy(CNF& S, CNF F);//将公式F的内容复制给S保存当前状态

status ExistSingle(CNF F);//验证F中有没有单子句存在，有则返回其值，没有返回FALSE

status EmptyCNF(CNF F);//验证公式是否为空

status EmptyCla(CNF F, int Key);//验证是否有空子句

int NotSameNum(int\* N, int n);//验证数组N中在第n个数之前有没有与其相等的数

status NotSameCla(ClaNode\* P, ClaNode\* N);//验证是否有两个子句相等

status Opposite\_SAT(int LitNum, int Key);//将key值取相反

/\*-------------------数独求解函数----------------\*/

int\*\* Prepare();//准备，填充一个目标二维数组

void Constraint\_1(SATFormula& Formula, int\*\* B);//约束1

void Constraint\_2(SATFormula& Formula, int\*\* B);//约束2

void Constraint\_3(SATFormula& Formula, int\*\* B);//约束3

void Convert(SATFormula& Formula);//转换函数，将ij形式转换为数字形式

status Las\_vegas(SATFormula& Formula, int\*\* B, int\* Puzzle, char\* filename);//拉斯维加斯算法，用于随机生成完整棋局

void DiggingHole(SATFormula& Formula, int\*\* B, int\* Puzzle,char\* filename);//挖洞生成数独

void ExportSAT(SATFormula Formula);//输出保存的公式

status PuzzleSAT\_FileSave(SATFormula Formula, char\* filename);//输出公式到文件中

void FormulaDelete(SATFormula& Formula);//删除保存的公式

/\*-------工具函数-------\*/

void Combine(int s[], int n, int k, SATFormula& Formula, void (\*Print)(int\* Subset, SATFormula& Formula));//约束2中进行组合函数

void Print\_AddCla(int\* Subset, SATFormula& Formula);//将选取出来的组合函数添加到公式集里

void AddClause(int\* Subset, int n, SATFormula& Formula);//将选取出来的子句添加到公式集里

/\*------------------end-----------------------\*/

# 附录2（Main）

#include"head.h"

#include<stdio.h>

int\* VELUE = NULL;

int CHOOSE = 2;

int ConflicCla[100] = { 0 };

int ConflicNum = 1;

int\* LearnCla[100] = { 0 };

int LearnNum = 0;

int Time = 0;

int Order = 0;

int main()

{

int op\_Menu = 1, op\_SAT = 1, op\_Puzzle = 1, op\_CreatePuzzle = 1, i = 0, n = 0, tem[10] = { 0 }, \* Puzzle = NULL, \*\* B = NULL; CNF F, S; clock\_t Start = 0, Finish = 0, Duration = 0; SATFormula Formula; P\_ClauseList P = NULL; char filename[20] = { 0 }, temporary[10] = { 0 };

while (op\_Menu)//显示菜单界面

{

system("cls"); printf("\n\n");

printf(" Menu for Binary Puzzle based on SAT problem \n");

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

printf(" 1. SATSolve 2. PuzzleSolve\n");

printf(" 0. Exit\n");

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

printf(" 请选择你的操作[0~2]:");

scanf("%d", &op\_Menu);

switch (op\_Menu)

{

case 1://SAT问题求解

op\_SAT = 1;

while (op\_SAT)//SAT问题

{

system("cls"); printf("\n\n");//SAT问题菜单

printf(" Menu for SAT problem solve \n");

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

printf(" 1. ImportCNF 2. ExportCNF\n");

printf(" 3. DeleteCNF 4. DPLL\n");

printf(" 0. Back\n");

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

printf(" 请选择你的操作[0~4]:");

scanf("%d", &op\_SAT);

switch (op\_SAT)

{

case 1://输入CNF文件名称，导入CNF

printf("请输入想验证的文件名：");

scanf("%s", filename);

if (CreateAdjacencyList(filename, F))

{

printf("创建邻接表成功");

}

else

{

printf("创建邻接表失败");

}

getchar(); getchar();

break;//end of case 1

case 2://输出CNF公式

Copy(S, F);

ExportCNF(S);

DeleteCNF(S);

getchar(); getchar();

break;//end of case 2

case 3://删除CNF公式

DeleteCNF(F);

printf("删除CNF公式成功");

getchar(); getchar();

break;//end of case 3

case 4://求解SAT问题

printf("请选择算法 :\n1 .时序回溯\n2.非时序回溯 + 字句学习 : ");

scanf("%d", &n);

printf("\n请选择选取办法 :\n1.没有赋值的第一个文字\n2.JW选取办法 : ");

scanf("%d", &CHOOSE);

switch (n)

{

case 1://时序回溯求解

Copy(S, F);//复制F，创建副本S

i = ExistSingle(S);//查找单子句

Start = clock();//计时开始

if (DPLL(S, i))//调用DPLL进行求解

{

Finish = clock();//计时结束

printf("\n求解成功，当前SAT问题的解为：\n");

for (i = 1; i <= S.LitNum; i++)//在屏幕上输出

{

printf("%d : %d\n", i, VELUE[i]);

}

Duration = Finish - Start;//计算所需时间

printf("求解所用时间为 %ld 毫秒\n", Duration);

SATSove\_FileSave(filename, 1, Duration, S.LitNum);//问题答案文件保存

DeleteCNF(S);//删除副本

free(VELUE);

VELUE = NULL;

}

else

{

Finish = clock();//计时结束

printf("求解失败，当前SAT问题无解\n");

Duration = Finish - Start;//计算求解时间

printf("求解所用时间为 %ld 毫秒\n", Duration);

SATSove\_FileSave(filename, 0, Duration, S.LitNum);//文件保存

DeleteCNF(S);

}

break;

case 2://非时序回溯+子句学习求解

Copy(S, F);//复制F，创建副本S

Reversal(S);//翻转副本S

for (i = 0; i < 100; i++)//准备工作，清空学习子句数组

{

if (LearnCla[i] != NULL)

{

free(LearnCla[i]);

LearnCla[i] = NULL;

}

}

for (i = 0; i < 100; i++)

{

ConflicCla[i] = 0;

}

i = BetterExistSingle(S);//查找单子句

LearnNum = 0;

ConflicNum = 1;

Time = 0;

Start = clock();//计时开始

if (BetterDPLL(S, i, 1) > 0)//调用优化方法求解

{

Finish = clock();//计时结束

printf("求解成功，当前SAT问题的解为：\n");

for (i = 1; i <= S.LitNum; i++)//在屏幕上输出

{

printf("%d : %d\n", i, VELUE[i]);

}

Duration = Finish - Start;//计算求解时间

printf("求解所用时间为 %ld 毫秒\n", Duration);

SATSove\_FileSave(filename, 1, Duration, S.LitNum);//文件保存

DeleteCNF(S);//删除副本S

free(VELUE);

VELUE = NULL;

}

else

{

Finish = clock();//计时结束

printf("求解失败，当前SAT问题无解\n");

Duration = Finish - Start;//计算求解时间

printf("求解所用时间为 %ld 毫秒\n", Duration);

SATSove\_FileSave(filename, 0, Duration, S.LitNum);//文件保存

DeleteCNF(S);//删除副本S

}

break;

default:

printf("输入不合法");

break;

}

getchar(); getchar();

break;//end of case 4

case 0:

break;

default:

printf("请输入 0 ~ 4 之间的操作数");

getchar(); getchar();

break;

}

}

break;

case 2://数独问题求解

op\_Puzzle = 1;

while (op\_Puzzle)//数独问题

{

system("cls"); printf("\n\n");//数独问题菜单

printf(" Menu for Binary Puzzle solve \n");

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

printf(" 1. CreatePuzzle 2. CheckPuzzle\n");

printf(" 3. SolvePuzzle\n");

printf(" 0. Back\n");

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

printf(" 请选择你的操作[0~3]:");

scanf("%d", &op\_Puzzle);

switch (op\_Puzzle)

{

case 1://创建新数独

op\_CreatePuzzle = 1;

while (op\_CreatePuzzle)//创建数独

{

system("cls"); printf("\n\n");//创建数独菜单

printf(" Menu for CreatePuzzle \n");

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

printf(" 1. By yourselve\n");

printf(" 2. ComputerRandom(Base on Digging holes)\n");

printf(" 0. Back\n");

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

printf(" 请选择你的操作[0~2]:");

scanf("%d", &op\_CreatePuzzle);

switch (op\_CreatePuzzle)

{

case 1://用户自己输入数独

//询问数独阶数

printf("输入您想求解的数独阶数：");

scanf("%d", &Order);

if (Order % 2 != 0)

{

printf("输入不合法，请输入偶数阶数独");

getchar(); getchar();

break;

}

//准备

B = Prepare();

if (Puzzle != NULL)

{

free(Puzzle);

Puzzle = 0;

}

if (!(Puzzle = (int\*)calloc(Order \* Order, sizeof(int))))

{

exit(OVERFLOW);

}

for (i = 0; i < Order \* Order; i++)

{

Puzzle[i] = 0;

}

//输入用户想解决的数独

printf("请输入原始数独，以 '0' 结尾：");

scanf("%d", &tem[0]);//读取

while (tem[0] != 0)//放入子句集和棋盘数Puzzle

{

AddClause(tem, 1, Formula);//添加进公式集

if (tem[0] > 0)//若当前位置为1

{

//计算棋盘位置

i = tem[0] % 10;

n = (tem[0] - i) / 10;

n = (n - 1) \* Order + i - 1;

Puzzle[n] = tem[0];//放入

}

else//若为0

{

//计算棋盘位置

i = (-tem[0]) % 10;

n = ((-tem[0]) - i) / 10;

n = (n - 1) \* Order + i - 1;

Puzzle[n] = tem[0];//放入

}

scanf("%d", &tem[0]);

}

Constraint\_1(Formula, B);

Constraint\_2(Formula, B);

Constraint\_3(Formula, B);

Convert(Formula);

//询问保存文件名称

printf("请输入想保存的文件编号：");

scanf("%s", temporary);

strcpy(filename, "Puzzle\_");

strcat(filename, temporary);

strcat(filename, ".cnf");

if (PuzzleSAT\_FileSave(Formula, filename))//调用函数保存

{

printf("已经成功创建数独CNF，并将其保存在文件 %s 中", filename);

}

//结束

FormulaDelete(Formula);//删除子句集

for (i = 0; i <= Order; i++)//释放准备数组B

{

free(B[i]);

}

free(B);

B = NULL;

op\_CreatePuzzle = 0;//返回上级菜单

getchar(); getchar();

break;//end of case 1

case 2://挖洞生成数独

//询问数独阶数

printf("输入您想求解的数独阶数（当前只支持6阶和8阶）：");

scanf("%d", &Order);

if (Order % 2 != 0)

{

printf("输入不合法，请输入偶数阶数独");

getchar(); getchar();

break;

}

//询问保存文件名称

printf("请输入想保存的文件编号：");

scanf("%s", temporary);

strcpy(filename, "Puzzle\_");

strcat(filename, temporary);

strcat(filename, ".cnf");

//准备

B = Prepare();

if (Puzzle != NULL)

{

free(Puzzle);

Puzzle = 0;

}

if (!(Puzzle = (int\*)calloc(Order \* Order, sizeof(int))))

{

exit(OVERFLOW);

}

for (i = 0; i < Order \* Order; i++)

{

Puzzle[i] = -1;

}

while (!Las\_vegas(Formula, B, Puzzle, filename));//利用拉斯维加斯算法不断尝试生成完整终局，并放入棋盘数组Puzzle带出

DiggingHole(Formula, B, Puzzle, filename);//利用挖洞算法对终局进行挖洞，形成空局

op\_CreatePuzzle = 0;//返回上层菜单

getchar(); getchar();

break;//end of case 2

case 0:

break;

default:

printf("请输入 0 ~ 2之间的操作数");

getchar(); getchar();

break;

}

}

break;

case 2://查看当前数独

printf("希望求解的数独为：\n");

for (i = 0; i < Order \* Order; i++)//逐个输出当前的数独棋盘

{

if (i % Order == 0)

{

printf("\n");

}

if (Puzzle[i] > 0)//若棋盘内容大于0

{

printf("1 ");//输出1

}

else if (Puzzle[i] < 0)//若小于0

{

printf("0 ");//输出0

}

else//若没有

{

printf("# ");//输出#

}

}

getchar(); getchar();

break;//end of case 2

case 3://求解数独

CHOOSE = 2;//默认选取办法为JW选取办法

//解决问题

CreateAdjacencyList(filename, F);

Copy(S, F);

Reversal(S);

for (i = 0; i < 100; i++)

{

if (LearnCla[i] != NULL)

{

free(LearnCla[i]);

LearnCla[i] = NULL;

}

}

for (i = 0; i < 100; i++)

{

ConflicCla[i] = 0;

}

i = BetterExistSingle(S);

LearnNum = 0;

ConflicNum = 1;

Time = 0;

Start = clock();

if (BetterDPLL(S, i, 1) > 0)

{

Finish = clock();

printf("求解成功，当前数独的解为：\n");

for (i = 1; i <= Order \* Order; i++)

{

if (i % Order == 1)

{

printf("\n");

}

printf("%d ", VELUE[i]);

}

Duration = Finish - Start;

printf("\n求解所用时间为 %ld 毫秒\n", Duration);

SATSove\_FileSave(filename, 1, Duration, S.LitNum);//保存文件

DeleteCNF(S);

free(VELUE);

VELUE = NULL;

}

else

{

Finish = clock();

printf("求解失败，当前数独无解\n");

Duration = Finish - Start;

printf("求解所用时间为 %ld 毫秒\n", Duration);

SATSove\_FileSave(filename, 0, Duration, S.LitNum);

DeleteCNF(S);

}

getchar(); getchar();

break;//end of case 3

case 0:

break;

default:

printf("请输入 0 ~ 3 之间的操作数");

getchar(); getchar();

break;

}

}

break;

case 0:

break;

default:

printf("请输入 0 ~ 4 之间的操作数");

getchar(); getchar();

break;

}

}

}

# 附录3（CreateCNF）

#include"head.h"

status CreateAdjacencyList(char\* filename, CNF& F)

{

char s[4]; int i = 0, j = 0, k = 0;/\*计数器\*/ int a = 0, n[50] = { 0 }, \* Cla = NULL; FILE\* fp = NULL;

if ((fp = fopen(filename, "r")) == NULL)//如果filename文件打开失败

{

return ERROR;

}

for (; s[0] != 'p'; fscanf(fp, "%c", &s[0]));//逐个读取文件中的元素，直到遇到以p为开始的正文

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

if (s[0] == 'c' && s[1] == 'n' && s[2] == 'f')//若是cnf文件

{

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

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

if (!(F.Letters = (LitList)calloc(2 \* F.LitNum + 1, sizeof(Litnode))))//动态分配数组

{

exit(OVERFLOW);

}

for (i = 0; i < F.ClaNum; i++)//循环输入第i个子句

{

j = 0;//初始化计数器

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

while (a != 0)//逐个录入每个子句的文字

{

n[j] = a;

j++;//统计每个子句的文字个数

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

}

if (!(Cla = (int\*)calloc(j + 1, sizeof(int))))//动态分配数组

{

exit(OVERFLOW);

}

Cla[0] = j;//将标志位设置为子句长度，大于0表明则该子句存在

for (k = 0; k < j; k++)//将保存好的数放在动态数组中

{

Cla[k + 1] = n[k];

}

for (k = 1; k < j + 1; k++)//构建每个文字的邻接表

{

ClaNode\* L;//创建新的结点，方便插入

if (!(L = (ClaNode\*)malloc(sizeof(ClaNode))))

{

exit(OVERFLOW);

}

L->clause = Cla;

L->ClaNum = Cla[0];

if (Cla[k] > 0 && NotSameNum(Cla, k))//若该文字为正文字

{

L->NextCla = F.Letters[Cla[k]].FirstCla;

F.Letters[Cla[k]].FirstCla = L;

}

else if (Cla[k] < 0 && NotSameNum(Cla, k))//若为负文字

{

L->NextCla = F.Letters[F.LitNum - Cla[k]].FirstCla;

F.Letters[F.LitNum - Cla[k]].FirstCla = L;

}

}

}

for (i = 1; i <= 2 \* F.LitNum; i++)

{

F.Letters[i].value = Undefine;

}

return OK;

}

else

{

return ERROR;//不是cnf文件，创建失败

}

}

# 附录4（Export）

#include"head.h"

extern int\* VELUE;

void ExportCNF(CNF F)

{

int i = 0, j = 0; ClaNode\* P = NULL;

for (i = 1; i <= 2 \* F.LitNum; i++)

{

P = F.Letters[i].FirstCla;

while (P != NULL)

{

if (P->clause[0] > 0 && i == MinNum(P->clause, F.LitNum))//当公式F的第i个文字的子句没有被删除且不为空,并且没有被输出

{

for (j = 1; j <= P->clause[0]; j++)//逐个输出子句内元素

{

printf("%d ", P->clause[j]);

}

printf("\n");

}

P = P->NextCla;

}

}

printf("输出结束，已经输出所有CNF公式");

}

void ExportSAT(SATFormula Formula)

{

P\_ClauseList P = NULL; int i = 0;

P = Formula.FirstClause;

while (P != NULL)

{

for (i = 0;; i++)

{

if (P->ClauseValue[i] != 0)

{

printf("%d ", P->ClauseValue[i]);

}

else

{

break;

}

}

printf("\n");

P = P->NextClause;

}

printf("ClaNum = %d\n", Formula.ClaNum);

printf("LitNum = %d", Formula.LitNum);

}

status PuzzleSAT\_FileSave(SATFormula Formula, char\* filename)

{

P\_ClauseList P = NULL; int i = 0; FILE\* fp = NULL;

if ((fp = fopen(filename, "w")) == NULL)//如果filename文件打开失败

{

return ERROR;

}

fprintf(fp, "%s", "p cnf ");//存放开始标志

fprintf(fp, "%d ", Formula.LitNum);//存放文字数

fprintf(fp, "%d\n", Formula.ClaNum);//存放子句数

P = Formula.FirstClause;//指向第一个子句

while (P != NULL)//遍历所有子句

{

for (i = 0;; i++)//逐个输出当前子句所有文字

{

if (P->ClauseValue[i] != 0)//若当前文字不为0

{

fprintf(fp, "%d ", P->ClauseValue[i]);//输出

}

else//若为0

{

break;//当前子句结束

}

}

fprintf(fp, "%d\n", 0);//放入回车

P = P->NextClause;//下一个子句

}

fclose(fp);//关闭文件

return OK;

}

void SATSove\_FileSave(char\* filename, int s, long t, int NUM)

{

char savefile[20] = { 0 }; int i = 0, S\_len = 0; FILE\* fp = NULL;

S\_len = strlen(filename);//求出当前文件名长度

for (i = 0; i < S\_len - 3; i++)//将文件名复制到想要保存的文件名中，后三个字符即“CNF”不要

{

savefile[i] = filename[i];

}

savefile[i] = '\0';//放入结束标志

strcat(savefile, "res");//放入res

fp = fopen(savefile, "w");//打开想要写入的文件

fprintf(fp, "%c", 's');//放入字符's'

fprintf(fp, "%7d\n", s);//放入s的值，即问题有没有解

if (s == 0)//若无解

{

fprintf(fp, "%c\n", 'v');//放入'v'字符

fprintf(fp, "%c", 't');//放入't'字符

fprintf(fp, "%7ld", t);//放入t的值

}

else if (s == 1)//若有解

{

fprintf(fp, "%c", 'v');//放入'v'字符

if (VELUE[1] == 0)//为了回车数量，现先放入第一个字符，若第一个文字为假

{

fprintf(fp, "%7d", -1);//放入-1

}

else//若为真或不确定

{

fprintf(fp, "%7d", 1);//放入1

}

for (i = 2; i <= NUM; i++)//循环放入剩下文字的解

{

if (i % 10 == 1)//每隔10个文字换行

{

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

}

if (VELUE[i] == 0)

{

fprintf(fp, "%8d", -i);//为了对其，保存8个长度

}

else

{

fprintf(fp, "%8d", i);

}

}

fprintf(fp, "\n%c", 't');//放入't'字符

fprintf(fp, "%7ld", t);//放入t的值

}

fclose(fp);//关闭文件

printf("已将结果保存到文件 %s 中", savefile);//输出提示

}

# 附录5（DPLL）

#include"head.h"

extern int\* VELUE;

extern int CHOOSE;

status DPLL(CNF F, int i)

{

int j = 0, n = 0; ClaNode\* P = NULL;

while (i)//单子句传播

{

BCP(F, i);

if (EmptyCNF(F))

{

VELUE = (int\*)calloc(2 \* F.LitNum + 1, sizeof(int));

for (j = 1; j <= 2 \* F.LitNum; j++)

{

VELUE[j] = F.Letters[j].value;

}

return TRUE;

}

else if (EmptyCla(F, i))

{

return FALSE;

}

i = ExistSingle(F);//重新查找单子句

}//while

int v = 0; CNF S;

switch (CHOOSE)

{

case 1:

v = Choose\_1(F);//第一个选取策略

break;

case 2:

v = Choose\_2(F);//第二个选取策略

break;

default:

break;

}

Copy(S, F);//保存当前公式副本

if (DPLL(F, v))//在第一分支中搜索

{

DeleteCNF(S);;//删除副本公式集S

return TRUE;

}

else

{

v = Opposite\_SAT(F.LitNum, v);//取v的相反

if (DPLL(S, v))//回溯到对v执行分支策略的初态进入另一分支

{

DeleteCNF(S);//删除副本公式集S

return TRUE;

}

else

{

DeleteCNF(S);//删除副本公式集S

return FALSE;

}

}

}

void BCP(CNF& F, int i)

{

int j = 0, n = 0, OppNum = 0; ClaNode\* P = NULL;

if (i < 0)

{

i = F.LitNum - i;

}

P = F.Letters[i].FirstCla;//P指向单子句文字的第一个子句结点

F.Letters[i].value = TRUE;//单子句文字值为真

if (i <= F.LitNum)//若为正文字

{

F.Letters[i + F.LitNum].value = FALSE;//单子句的负文字值为假

}

else//若为负文字

{

F.Letters[i - F.LitNum].value = FALSE;//单子句的正文字值为假

}

while (P != NULL)//当子句不为空时

{

P->clause[0] = DELETE;//删除包含该文字的子句

P = P->NextCla;

}

if (i <= F.LitNum)//若该文字为正文字

{

P = F.Letters[i + F.LitNum].FirstCla;//令P为该文字的负文字的第一个子句

while (P != NULL)

{

for (j = 1, n = 0; j <= P->clause[0]; j++)//逐个遍历该子句

{

if (P->clause[j] == -i) //找到该负文字

{

P->clause[j] = 0;//删除

n += 1;//记录要删除的文字数目

}

else if (P->clause[j] == 0)//如果已经被删除

{

P->clause[0] += 1;//则总数加一

n += 1;//待删除数加一

}

}

P->clause[0] -= n;//将子句中的文字数目进行更改

P = P->NextCla;

}

}

else//若该文字为负文字

{

P = F.Letters[i - F.LitNum].FirstCla;//令P为该文字的正文字的第一个子句

while (P != NULL) {

for (j = 1, n = 0; j <= P->clause[0]; j++)//逐个遍历该子句

{

if (P->clause[j] == i - F.LitNum)//找到该正文字

{

P->clause[j] = 0;//删除

n += 1; //记录要删除的文字数目

}

else if (P->clause[j] == 0)//如果已经被删除

{

P->clause[0] += 1;//则总数加一

n += 1;//待删除数加一

}

}

P->clause[0] -= n;//将子句中的文字数目进行更改

P = P->NextCla;

}

}

}

status ExistSingle(CNF F)

{

int i = 0, j = 0; ClaNode\* P = NULL;

for (i = 1; i <= 2 \* F.LitNum; i++)//逐个遍历每个文字的邻接链表

{

if (F.Letters[i].value == Undefine)

{

P = F.Letters[i].FirstCla;//P指向当前文字的第一个子句结点

while (P != NULL)//当该子句存在时

{

if (P->clause[0] == 1)//当找到单子句时

{

return i;//返回该单子句

}

P = P->NextCla;//查询下一个子句

}

}

}

return FALSE;//没有单子句则返回FALSE

}

status EmptyCNF(CNF F)

{

int i = 0; ClaNode\* P = NULL;

for (i = 1; i <= 2 \* F.LitNum; i++)//逐个遍历每个文字的邻接链表

{

if (F.Letters[i].value != TRUE)

{

P = F.Letters[i].FirstCla;//P指向当前文字的第一个子句结点

while (P != NULL)//若当前子句不为空

{

if (P->clause[0] >= 0)//且没有被删除或为空子句

{

return FALSE;//则当前公式不为空，返回FAlSE

}

P = P->NextCla;

}

}

}

return TRUE;

}

status EmptyCla(CNF F, int Key)

{

int i = 0, j = 0, n = 0; ClaNode\* P = NULL;

Key = Opposite\_SAT(F.LitNum, Key);

P = F.Letters[Key].FirstCla;//P指向当前文字的第一个子句结点

while (P != NULL)//若当前子句不为空

{

if (P->clause[0] == 0)//若是空子句

{

return TRUE;//则找到空子句，返回TRUE

}

P = P->NextCla;

}

return FALSE;

}

void DeleteCNF(CNF& F)

{

int i = 0, j = 0, n = 0, \*\* T = NULL; ClaNode\* P = NULL, \* Q = NULL, \* R = NULL, \* S = NULL;

if (!(T = (int\*\*)calloc(F.ClaNum, sizeof(int\*))))//动态分配数组

{

exit(OVERFLOW);

}

for (i = 1; i <= 2 \* F.LitNum; i++)//逐个收集每个子句空间的地址

{

P = F.Letters[i].FirstCla;//指向文字链表的第一个结点

while (P != NULL)//当结点存在时

{

P->clause[0] = 1;

P = P->NextCla;

}

}

for (i = 1, n = 0; i <= 2 \* F.LitNum; i++)//逐个收集每个子句空间的地址

{

P = F.Letters[i].FirstCla;//指向文字链表的第一个结点

while (P != NULL)//当结点存在时

{

if (P->clause[0] != DELETE)

{

T[n] = P->clause;//将存储空间的指针放入数组中保存

P->clause[0] = DELETE;//利用标志位确定是否被统计过

n++;

}

P = P->NextCla;

}

}

for (n = 0; n < F.ClaNum; n++)//循环释放所有子句的存储空间

{

free(T[n]);

}

free(T);//释放指针数组

for (i = 1; i <= 2 \* F.LitNum; i++)//删除全部文字链表

{

P = F.Letters[i].FirstCla;//指向文字链表的第一个结点

while (P != NULL)//当结点存在时

{

Q = P->NextCla;

free(P);

P = Q;

}

}

free(F.Letters);

F.Letters = NULL;

F.ClaNum = 0;

F.LitNum = 0;

}

void Copy(CNF& S, CNF F)/\*感觉这里有问题\*/

{

int i = 0, j = 0, k = 0, num = 0, Num = 0, \* Cla = NULL; ClaNode\* P = NULL;

S.LitNum = F.LitNum;//复制文字数目

if (!(S.Letters = (LitList)calloc(2 \* S.LitNum + 1, sizeof(Litnode))))//动态分配数组

{

exit(OVERFLOW);

}

for (i = 1; i <= 2 \* S.LitNum; i++)//将F的每个文字的真值复制

{

S.Letters[i].value = F.Letters[i].value;

S.Letters[i].FirstCla = NULL;

}

for (i = 1; i <= 2 \* S.LitNum; i++)

{

if (F.Letters[i].value == Undefine)

{

Num = 0;

P = F.Letters[i].FirstCla;

while (P != NULL)

{

if (P->clause[0] > 0)//当公式F的第i个文字的子句没有被删除且不为空,并且没有被输出

{

Num += 1;

if (i == MinNum(P->clause, F.LitNum))

{

if (!(Cla = (int\*)calloc(P->clause[0] + 1, sizeof(int))))//动态分配数组

{

exit(OVERFLOW);

}

Cla[0] = P->clause[0];

for (j = 1, k = 1; k <= P->clause[0]; j++)//逐个输出子句内元素

{

if (P->clause[j] != 0)

{

Cla[k] = P->clause[j];

k++;

}

}

for (k = 1; k <= Cla[0]; k++)//构建每个文字的邻接表

{

ClaNode\* L;//创建新的结点，方便插入

if (!(L = (ClaNode\*)malloc(sizeof(ClaNode))))

{

exit(OVERFLOW);

}

L->clause = Cla;

L->ClaNum = P->ClaNum;//复制子句中文字的数目

if (Cla[k] > 0 && NotSameNum(Cla, k))//若该文字为正文字

{

L->NextCla = S.Letters[Cla[k]].FirstCla;

S.Letters[Cla[k]].FirstCla = L;

}

else if (Cla[k] < 0 && NotSameNum(Cla, k))//若为负文字

{

L->NextCla = S.Letters[S.LitNum - Cla[k]].FirstCla;

S.Letters[S.LitNum - Cla[k]].FirstCla = L;

}

}

num += 1;//统计子句数目

}

}

P = P->NextCla;

}

}

S.ClaNum = num;//子句数目

}

}

int MinNum(int\* N, int m)

{

int i = 0, j = 0, \* Num = NULL;

if (!(Num = (int\*)calloc(N[0], sizeof(int))))//动态分配数组

{

exit(OVERFLOW);

}

for (i = 1, j = 0; j < N[0]; i++)

{

if (N[i] < 0)

{

Num[j] = m - N[i];

j++;

}

else if (N[i] > 0)

{

Num[j] = N[i];

j++;

}

}

for (i = 1, j = Num[0]; i < N[0]; i++)

{

if (j > Num[i])

{

j = Num[i];

}

}

free(Num);

return j;

}

# 附录6（BetterDpll）

#include"head.h"

extern int\* VELUE;

extern int CHOOSE;

extern int ConflicCla[100];

extern int ConflicNum;

extern int\* LearnCla[100];

extern int LearnNum;

status BetterDPLL(CNF& F, int i, int Deep)

{

int j = 0, DifferenceDeep = 0; ClaNode\* P = NULL;

while (i)//单子句传播

{

BetterBCP(F, i, Deep);//进行BCP过程

if (EmptyCNF(F))

{

if (!(VELUE = (int\*)calloc(2 \* F.LitNum + 1, sizeof(int))))

{

exit(OVERFLOW);

}

for (j = 1; j <= 2 \* F.LitNum; j++)

{

VELUE[j] = F.Letters[j].value;

}

return TRUE;

}

else if ((DifferenceDeep = BetterEmptyCla(F, i)) > 0)

{

return 1 - DifferenceDeep;

}

i = BetterExistSingle(F);//重新查找单子句

}//while

int v = 0, Difference = 0; CNF S;

AddLearnCla(F);//将学习子句添加到子句集中

switch (CHOOSE)

{

case 1:

v = Choose\_1(F);//第一个选取策略

break;

case 2:

v = Choose\_2(F);//第二个选取策略

if (v == 0)

{

exit(-5);

}

break;

default:

break;

}

BetterCopy(S, F);//保存当前公式副本

if ((Difference = BetterDPLL(F, v, Deep + 1)) > 0)//在第一分支中搜索

{

DeleteCNF(S);//删除副本公式集S

return TRUE;

}

else

{

if (Difference != 0)

{

return Difference + 1;

}

else

{

v = Opposite\_SAT(F.LitNum, v);//取v的相反

AddLearnCla(S);

if ((Difference = BetterDPLL(S, v, Deep + 1)) > 0)//回溯到对v执行分支策略的初态进入另一分支

{

DeleteCNF(S);//删除副本公式集S

return TRUE;

}

else

{

DeleteCNF(S);//删除副本公式集S

if (Difference == 0)

{

return Difference;

}

else

{

return Difference + 1;

}

}

}

}

}

void BetterBCP(CNF& F, int i, int Deep)

{

int j = 0, n = 0, OppNum = 0; ClaNode\* P = NULL;

P = F.Letters[i].FirstCla;//P指向单子句文字的第一个子句结点

F.Letters[i].value = TRUE;//单子句文字值为真

F.Letters[i].Deep = Deep;//当前决策深度

while (P != NULL)//当子句不为空时

{

P->clause[0] = DELETE;//删除包含该文字的子句

P = P->NextCla;

}

OppNum = Opposite\_SAT(F.LitNum, i);//取i的相反数

F.Letters[OppNum].value = FALSE;//单子句的相反文字值为假

F.Letters[OppNum].Deep = Deep;//当前决策深度

P = F.Letters[OppNum].FirstCla;//令P为该文字的相反文字的第一个子句

while (P != NULL)

{

if (P->clause[0] > 0)

{

for (j = 1, n = 0; j <= P->ClaNum; j++)//逐个遍历该子句

{

if (F.Letters[P->clause[j]].value == FALSE) //找到该相反文字

{

n += 1;//记录要删除的文字数目

}

}

P->clause[0] = P->ClaNum - n;//将子句中的文字数目进行更改

}

P = P->NextCla;

}

}

status BetterExistSingle(CNF F)

{

int i = 0, j = 0; ClaNode\* P = NULL;

for (i = 1; i <= 2 \* F.LitNum; i++)//逐个遍历每个文字的邻接链表

{

if (F.Letters[i].value == Undefine)

{

P = F.Letters[i].FirstCla;//P指向当前文字的第一个子句结点

while (P != NULL)//当该子句存在时

{

if (P->clause[0] == 1)//当找到单子句时

{

if (P->ClaNum != 1)

{

F.Letters[i].CLA = P;

F.Letters[Opposite\_SAT(F.LitNum, i)].CLA = P;

}

return i;

}

P = P->NextCla;//查询下一个子句

}

}

}

return FALSE;//没有单子句则返回FALSE

}

status BetterEmptyCla(CNF F, int Key)

{

ClaNode\* P = NULL; int Deep = 0, MaxDeep = 0, i = 0, j = 0, N[MaxLearnClaNum] = { 0 }, \* Cla = NULL;

Key = Opposite\_SAT(F.LitNum, Key);

P = F.Letters[Key].FirstCla;//P指向当前文字的第一个子句结点

while (P != NULL)//若当前子句不为空

{

if (P->clause[0] == 0)//若是空子句

{

for (i = 0; i < 100; i++)

{

ConflicCla[i] = 0;

}

ConflicNum = 1;

for (i = 1; i <= P->ClaNum; i++)

{

SearchConflic(F, P, P->clause[i]);//寻找蕴含图中的没有前驱的结点

}

for (i = 1, j = 0; i < ConflicNum && j < MaxLearnClaNum; i++)//只保存长度不大于10的学习子句

{

if (NotSameNum(ConflicCla, i) == TRUE)//排除重复的结点

{

if (F.Letters[ConflicCla[i]].value == TRUE)

{

ConflicCla[i] = Opposite\_SAT(F.LitNum, ConflicCla[i]);

}

N[j] = ConflicCla[i];//保存在临时数组中

j++;//统计学习子句的个数

}

}

if (i == 100 || j == 0 || j == MaxLearnClaNum)

{

return TRUE;

}

if (!(Cla = (int\*)calloc(j + 1, sizeof(int))))//申请空间

{

exit(OVERFLOW);

}

Cla[0] = j;

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

{

Cla[i] = N[i - 1];//将学习到的子句保存在数组中

}

for (i = 1, MaxDeep = 0; i <= Cla[0]; i++)//统计学习子句中最大的层数

{

if (F.Letters[Cla[i]].Deep > MaxDeep)

{

MaxDeep = F.Letters[Cla[i]].Deep;

}

}

Deep = F.Letters[Key].Deep - MaxDeep;//计算与当前的层数差

LearnCla[LearnNum % 100] = Cla;//将学习到的子句保存在整个学习子句库中

LearnNum++;

return Deep + 1;//返回深度差

}

P = P->NextCla;

}

return FALSE;

}

void BetterCopy(CNF& S, CNF F)

{

int i = 0, j = 0, k = 0, num = 0, Num = 0, \* Cla = NULL; ClaNode\* P = NULL;

S.LitNum = F.LitNum;//复制文字数目

if (!(S.Letters = (LitList)calloc(2 \* S.LitNum + 1, sizeof(Litnode))))//动态分配数组

{

exit(OVERFLOW);

}

for (i = 1; i <= 2 \* S.LitNum; i++)//复制每个文字的信息

{

S.Letters[i].value = F.Letters[i].value;

S.Letters[i].Deep = F.Letters[i].Deep;

S.Letters[i].CLA = F.Letters[i].CLA;

S.Letters[i].FirstCla = NULL;

}

for (i = 1; i <= 2 \* S.LitNum; i++)

{

if (F.Letters[i].value == Undefine)

{

Num = 0;

P = F.Letters[i].FirstCla;

while (P != NULL)

{

if (P->clause[0] > 0)

{

Num += 1;

if (i == BetterMinNum(F, P->ClaNum, P->clause))//当公式F的第i个文字的子句没有被删除且不为空,并且没有被输出

{

if (!(Cla = (int\*)calloc(P->ClaNum + 1, sizeof(int))))//动态分配数组

{

exit(OVERFLOW);

}

for (j = 0; j <= P->ClaNum; j++)//逐个输出子句内元素

{

Cla[j] = P->clause[j];

}

for (k = 1; k <= P->ClaNum; k++)//构建每个文字的邻接表

{

if (F.Letters[Cla[k]].value != FALSE)

{

ClaNode\* L;//创建新的结点，方便插入

if (!(L = (ClaNode\*)malloc(sizeof(ClaNode))))

{

exit(OVERFLOW);

}

L->clause = Cla;

L->ClaNum = P->ClaNum;//复制子句中文字的数目

if (NotSameNum(Cla, k))//若该文字为正文字

{

L->NextCla = S.Letters[Cla[k]].FirstCla;

S.Letters[Cla[k]].FirstCla = L;

}

}

}

num += 1;//统计子句数目

}

}

P = P->NextCla;

}

}

}

S.ClaNum = num;//子句数目

}

void SearchConflic(CNF F, ClaNode\* P, int key)

{

int i = 0;

for (i = 1; i <= P->ClaNum; i++)

{

if (P->clause[i] != key && ConflicNum < 100 && P->ClaNum != 1)

{

if (F.Letters[P->clause[i]].CLA == NULL)

{

ConflicCla[ConflicNum] = P->clause[i];

ConflicNum++;

}

else

{

SearchConflic(F, F.Letters[P->clause[i]].CLA, Opposite\_SAT(F.LitNum, P->clause[i]));

}

}

}

}

void AddLearnCla(CNF& F)

{

int i = 0, j = 0, n = 0, \* Cla = NULL, \* P = NULL; ClaNode\* N = NULL;

for (i = 0; i < 100; i++)

{

if (LearnCla[i] != NULL)//当前学习子句存在

{

P = LearnCla[i];

n++;//子句数加一

if (!(Cla = (int\*)calloc(P[0] + 1, sizeof(int))))//分配空间

{

exit(OVERFLOW);

}

for (j = 0; j <= P[0]; j++)//将学习字句复制

{

Cla[j] = P[j];

}

for (j = 1; j <= P[0]; j++)

{

if (F.Letters[Cla[j]].value == TRUE)//如果在子句中有已经判断为真

{

Cla[0] = DELETE;//将子句删除

break;

}

else if (F.Letters[Cla[j]].value == FALSE)//若为假

{

Cla[0] -= 1;//子句中文字数减一

}

}

for (j = 1; j <= P[0]; j++)

{

if (!(N = (ClaNode\*)malloc(sizeof(ClaNode))))

{

exit(OVERFLOW);

}

N->ClaNum = P[0];

N->clause = Cla;

N->NextCla = F.Letters[Cla[j]].FirstCla;

F.Letters[Cla[j]].FirstCla = N;

}

}

F.ClaNum += n;//将子句数添加

}

}

status NotSameCla(ClaNode\* P, ClaNode\* N)

{

int i = 0;

while (P != NULL)

{

if (P->ClaNum != N->ClaNum)

{

P = P->NextCla;

continue;

}

else

{

for (i = 1; i <= P->ClaNum; i++)

{

if (P->clause[i] != N->clause[i])

{

break;

}

}

if (i > P->ClaNum)

{

return FALSE;

}

}

P = P->NextCla;

}

return TRUE;

}

int BetterMinNum(CNF F, int ClaNum, int\* N)

{

int i = 0, j = 0;

for (i = 1, j = 1000000; i <= ClaNum; i++)

{

if (j > N[i] && F.Letters[N[i]].value != FALSE)

{

j = N[i];

}

}

return j;

}

# 附录7（Choose）

#include"head.h"

extern int Time;

status Choose\_1(CNF F)

{

int i = 0;

for (i = 1; i <= 2 \* F.LitNum; i++)//遍历每个文字

{

if (F.Letters[i].value == Undefine)//如果该文字没有被赋值

{

return i;//返回该文字的序号

}

}

}

status Choose\_2(CNF F)

{

int i = 0, Need = 0; double n = 0, J = 0, Jmax = 0; ClaNode\* P = NULL;

for (i = 1; i <= 2 \* F.LitNum; i++)//遍历每个文字

{

if (F.Letters[i].value == Undefine)//如果该文字没有被赋值

{

J = 0;

P = F.Letters[i].FirstCla;//P指向文字的第一个子句

while (P != NULL)//当存在子句时

{

if (P->clause[0] > 0)//当子句没有被删除或不为空

{

n = pow(2, -P->clause[0]);//n为2的负当前子句数目次方

J = J + n;//J的值为所有子句n值的和

}

P = P->NextCla;

}

if (J > Jmax)

{

Jmax = J;

Need = i;

}

}

}

return Need;

}

# 附录8（Tools）

#include"head.h"

void Reversal(CNF& F)

{

int i = 0, j = 0; ClaNode\* P = NULL;

for (i = 1; i <= 2 \* F.LitNum; i++)

{

P = F.Letters[i].FirstCla;

while (P != NULL)

{

for (j = 1; j <= P->ClaNum; j++)

{

if (P->clause[j] < 0)

{

P->clause[j] = F.LitNum - P->clause[j];

}

}

P = P->NextCla;

}

}

}

status Opposite\_SAT(int LitNum, int Key)

{

if (Key <= LitNum)

{

Key = Key + LitNum;

}

else

{

Key = Key - LitNum;

}

return Key;

}

int NotSameNum(int\* N, int n)

{

int i = 0;

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

{

if (N[i] == N[n])

{

return FALSE;

}

}

return TRUE;

}

# 附录9（PuzzleSolve）

#include"head.h"

CNF F, S;

extern int CHOOSE;

extern int ConflicCla[100];

extern int ConflicNum;

extern int\* LearnCla[100];

extern int LearnNum;

extern int\* VELUE;

extern int Order;

int\* Subset = NULL;

int ClaNumber = 0;

int\*\* Prepare()//准备，填充一个目标二维数组

{

int\*\* B = NULL, i = 0, j = 0;

ClaNumber = 0;

if (!(B = (int\*\*)calloc(Order + 1, sizeof(int\*))))//申请空间

{

exit(OVERFLOW);

}

for (i = 1; i <= Order; i++)//逐个申请空间

{

if (!(B[i] = (int\*)calloc(Order + 1, sizeof(int))))

{

exit(OVERFLOW);

}

for (j = 1; j <= Order; j++)//填充数据

{

B[i][j] = i \* 10 + j;

}

}

return B;

}

void Constraint\_1(SATFormula& Formula, int\*\* B)//约束1

{

int i = 0, j = 0, k = 0; P\_ClauseList P = NULL;

//行输入

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

{

for (j = 1; j <= Order - 2; j++)

{

//正输入

if (!(P = (P\_ClauseList)malloc(sizeof(P\_ClauseNode))))

{

exit(OVERFLOW);

}

for (k = 0; k < 3; k++)

{

P->ClauseValue[k] = B[i][j + k];

}

P->ClauseValue[k] = 0;

P->NextClause = Formula.FirstClause;

Formula.FirstClause = P;

ClaNumber++;

//负输入

if (!(P = (P\_ClauseList)malloc(sizeof(P\_ClauseNode))))

{

exit(OVERFLOW);

}

for (k = 0; k < 3; k++)

{

P->ClauseValue[k] = -B[i][j + k];

}

P->ClauseValue[k] = 0;

P->NextClause = Formula.FirstClause;

Formula.FirstClause = P;

ClaNumber++;

}

}

//列输入

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

{

for (i = 1; i <= Order - 2; i++)

{

//正输入

if (!(P = (P\_ClauseList)malloc(sizeof(P\_ClauseNode))))

{

exit(OVERFLOW);

}

for (k = 0; k < 3; k++)

{

P->ClauseValue[k] = B[i + k][j];

}

P->ClauseValue[k] = 0;

P->NextClause = Formula.FirstClause;

Formula.FirstClause = P;

ClaNumber++;

//负输入

if (!(P = (P\_ClauseList)malloc(sizeof(P\_ClauseNode))))

{

exit(OVERFLOW);

}

for (k = 0; k < 3; k++)

{

P->ClauseValue[k] = -B[i + k][j];

}

P->ClauseValue[k] = 0;

P->NextClause = Formula.FirstClause;

Formula.FirstClause = P;

ClaNumber++;

}

}

Formula.ClaNum = ClaNumber;

}

void Constraint\_2(SATFormula& Formula, int\*\* B)//约束2

{

int i = 0, j = 0; P\_ClauseList P = NULL; void (\*Print)(int\* Subset, SATFormula & Formula);

if (Order == 4)

{

//行输入

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

{

//正输入

if (!(P = (P\_ClauseList)malloc(sizeof(P\_ClauseNode))))

{

exit(OVERFLOW);

}

P->ClauseValue[0] = B[i][1];

P->ClauseValue[1] = B[i][2];

P->ClauseValue[2] = B[i][4];

P->ClauseValue[3] = 0;

P->NextClause = Formula.FirstClause;

Formula.FirstClause = P;

ClaNumber++;

if (!(P = (P\_ClauseList)malloc(sizeof(P\_ClauseNode))))

{

exit(OVERFLOW);

}

P->ClauseValue[0] = B[i][1];

P->ClauseValue[1] = B[i][3];

P->ClauseValue[2] = B[i][4];

P->ClauseValue[3] = 0;

P->NextClause = Formula.FirstClause;

Formula.FirstClause = P;

ClaNumber++;

//负输入

if (!(P = (P\_ClauseList)malloc(sizeof(P\_ClauseNode))))

{

exit(OVERFLOW);

}

P->ClauseValue[0] = -B[i][1];

P->ClauseValue[1] = -B[i][2];

P->ClauseValue[2] = -B[i][4];

P->ClauseValue[3] = 0;

P->NextClause = Formula.FirstClause;

Formula.FirstClause = P;

ClaNumber++;

if (!(P = (P\_ClauseList)malloc(sizeof(P\_ClauseNode))))

{

exit(OVERFLOW);

}

P->ClauseValue[0] = -B[i][1];

P->ClauseValue[1] = -B[i][3];

P->ClauseValue[2] = -B[i][4];

P->ClauseValue[3] = 0;

P->NextClause = Formula.FirstClause;

Formula.FirstClause = P;

ClaNumber++;

}

//列输入

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

{

//正输入

if (!(P = (P\_ClauseList)malloc(sizeof(P\_ClauseNode))))

{

exit(OVERFLOW);

}

P->ClauseValue[0] = B[1][i];

P->ClauseValue[1] = B[2][i];

P->ClauseValue[2] = B[4][i];

P->ClauseValue[3] = 0;

P->NextClause = Formula.FirstClause;

Formula.FirstClause = P;

ClaNumber++;

if (!(P = (P\_ClauseList)malloc(sizeof(P\_ClauseNode))))

{

exit(OVERFLOW);

}

P->ClauseValue[0] = B[1][i];

P->ClauseValue[1] = B[3][i];

P->ClauseValue[2] = B[4][i];

P->ClauseValue[3] = 0;

P->NextClause = Formula.FirstClause;

Formula.FirstClause = P;

ClaNumber++;

//负输入

if (!(P = (P\_ClauseList)malloc(sizeof(P\_ClauseNode))))

{

exit(OVERFLOW);

}

P->ClauseValue[0] = -B[1][i];

P->ClauseValue[1] = -B[2][i];

P->ClauseValue[2] = -B[4][i];

P->ClauseValue[3] = 0;

P->NextClause = Formula.FirstClause;

Formula.FirstClause = P;

ClaNumber++;

if (!(P = (P\_ClauseList)malloc(sizeof(P\_ClauseNode))))

{

exit(OVERFLOW);

}

P->ClauseValue[0] = -B[1][i];

P->ClauseValue[1] = -B[3][i];

P->ClauseValue[2] = -B[4][i];

P->ClauseValue[3] = 0;

P->NextClause = Formula.FirstClause;

Formula.FirstClause = P;

ClaNumber++;

}

Formula.ClaNum = ClaNumber;

}

else

{

Print = Print\_AddCla;//函数指针赋值

if (!(Subset = (int\*)calloc(Order / 2 + 1, sizeof(int))))

{

exit(OVERFLOW);

}

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

{

Combine(B[i], Order, Order / 2 + 1, Formula, Print);//进行组合函数选取，用鸽巢原理选取

}

Formula.ClaNum = ClaNumber;//输入当前子句值

free(Subset);//释放存储空间

Subset = NULL;

}

}

void Constraint\_3(SATFormula& Formula, int\*\* B)//约束3

{

int i = 0, j = 0, k = 0, n = 0, t = 0, o = 0, num = 0;

if (!(Subset = (int\*)calloc(Order + 1, sizeof(int))))

{

exit(OVERFLOW);

}

for (i = 1; i < 3; i++)//列或行

{

for (j = 1; j < Order; j++)//第一个数值

{

for (k = j + 1; k <= Order; k++)//第二个数值

{

n = i \* 100 + j \* 10 + k;

//转换157= -[1571,1572,…,1578]

Subset[0] = -n;//选取约束三中（-157,-1571,-1572,…,-1578）子句

for (num = 1; num <= Order; num++)

{

Subset[num] = -(n \* 10 + num);

}

AddClause(Subset, num, Formula);//添加该子句

Subset[0] = n;//选取约束三中（1571,157）,（1572,157）…（1578,157）

for (num = 1; num <= Order; num++)

{

Subset[1] = (n \* 10 + num);

AddClause(Subset, 2, Formula);//添加该子句

}

//转换15711= 51,71和15710= -51,-71和1571= 15711,15710

for (num = 1; num <= Order; num++)//从1到8

{

//转化1571= 15711,15710

Subset[0] = -(n \* 10 + num);//-1571

Subset[1] = (n \* 100 + num \* 10 + 1);//15711

Subset[2] = (n \* 100 + num \* 10 + 0);//15710

AddClause(Subset, 3, Formula);//添加1571，15711，15710

for (t = 0; t < 3; t++)

{

Subset[t] = -Subset[t];//使子句数组中元素值值翻转

}

AddClause(Subset, 2, Formula);//添加该子句1571，-15711

Subset[1] = Subset[2];//将-71放在第二个

AddClause(Subset, 2, Formula);//添加该子句1571，-15710

//转化15711= 51,71和15710= -51,-71

for (o = 0; o < 2; o++)//o为最末位时0还是1

{

if (i == 1)//若为两行相比较

{

if (o == 0)//若为0

{

Subset[0] = (n \* 100 + num \* 10 + o);//15710

Subset[1] = (j \* 10 + num);//51

Subset[2] = (k \* 10 + num);//71

AddClause(Subset, 3, Formula);//添加15710，51，71

}

else//若为1

{

Subset[0] = (n \* 100 + num \* 10 + o);//15711

Subset[1] = -(j \* 10 + num);//-51

Subset[2] = -(k \* 10 + num);//-71

AddClause(Subset, 3, Formula);//添加该子句15711，-51，-71

}

}

else//若为两列相比较

{

if (o == 0)//若为0

{

Subset[0] = (n \* 100 + num \* 10 + o);//25710

Subset[1] = (num \* 10 + j);//15

Subset[2] = (num \* 10 + k);//17

AddClause(Subset, 3, Formula);//添加25710，15，17

}

else//若为1

{

Subset[0] = (n \* 100 + num \* 10 + o);//15711

Subset[1] = -(num \* 10 + j);//-15

Subset[2] = -(num \* 10 + k);//-17

AddClause(Subset, 3, Formula);//添加该子句25711，-15，-17

}

}

for (t = 0; t < 3; t++)

{

Subset[t] = -Subset[t];//使子句数组中元素值值翻转

}

AddClause(Subset, 2, Formula);//添加该子句-15710，-51

Subset[1] = Subset[2];//将-71放在第二个

AddClause(Subset, 2, Formula);//添加该子句-15710，-71

}

}

}

}

}

Formula.ClaNum = ClaNumber;//放入子句总数

free(Subset);//释放存储空间

Subset = NULL;

}

void Convert(SATFormula& Formula)

{

int i = 0, j = 0, k = 0, n = 0, Basic = 0, SIGN = 0, RESULT = 6; P\_ClauseList P = NULL;

if (!(Subset = (int\*)calloc(7, sizeof(int))))

{

exit(OVERFLOW);

}

P = Formula.FirstClause;

while (P != NULL)

{

for (i = 0;; i++)

{

n = P->ClauseValue[i];

if (n == 0)//遇到了结束符号

{

break;

}

//判断正负

if (n > 0)//想要转变的数大于0

{

Subset[SIGN] = 1;//符号位为正数

}

else//想要转变的数小于0

{

Subset[SIGN] = 0;//符号位为正数

n = -n;//转为正数

}

//开始转换

if (n < 100)//两位数

{

Subset[2] = n % 10;//个位数

Subset[1] = (n - Subset[2]) / 10;//十位数

Subset[RESULT] = (Subset[1] - 1) \* Order + Subset[2];

}

else if (n < 1000)//三位数

{

Subset[3] = n % 10;//个位数

Subset[2] = (n % 100 - Subset[3]) / 10;//十位数

Subset[1] = (n - Subset[2] \* 10 - Subset[3]) / 100;//百位数

switch (Subset[1])

{

case 1:

Basic = Order \* Order;

Subset[RESULT] = (Basic + Order \* (Subset[2] - 1) - (Subset[2] - 1) \* Subset[2] / 2 + Subset[3] - Subset[2]);

break;

case 2:

Basic = Order \* Order + Order \* (Order - 1) / 2;

Subset[RESULT] = (Basic + Order \* (Subset[2] - 1) - (Subset[2] - 1) \* Subset[2] / 2 + Subset[3] - Subset[2]);

break;

default:

break;

}

}

else if (n < 10000)//四位数

{

Subset[4] = n % 10;//个位数

Subset[3] = (n % 100 - Subset[4]) / 10;//十位数

Subset[2] = (n % 1000 - Subset[3] \* 10 - Subset[4]) / 100;//百位数

Subset[1] = (n - Subset[2] \* 100 - Subset[3] \* 10 - Subset[4]) / 1000;//千位数

switch (Subset[1])

{

case 1:

Basic = Order \* Order + Order \* (Order - 1);

Subset[RESULT] = (Basic + (Order \* (Subset[2] - 1) - (Subset[2] - 1) \* Subset[2] / 2 + Subset[3] - Subset[2] - 1) \* Order + Subset[4]);

break;

case 2:

Basic = Order \* Order + Order \* (Order - 1) + Order \* (Order - 1) / 2 \* Order;

Subset[RESULT] = (Basic + (Order \* (Subset[2] - 1) - (Subset[2] - 1) \* Subset[2] / 2 + Subset[3] - Subset[2] - 1) \* Order + Subset[4]);

break;

default:

break;

}

}

else//五位数

{

Subset[5] = n % 10;//个位数

Subset[4] = (n % 100 - Subset[5]) / 10;//十位数

Subset[3] = (n % 1000 - Subset[4] \* 10 - Subset[5]) / 100;//百位数

Subset[2] = (n % 10000 - Subset[3] \* 100 - Subset[4] \* 10 - Subset[5]) / 1000;//千位数

Subset[1] = (n - Subset[2] \* 1000 - Subset[3] \* 100 - Subset[4] \* 10 - Subset[5]) / 10000;//万位数

switch (Subset[1])

{

case 1:

Basic = Order \* Order + Order \* (Order - 1) + Order \* (Order - 1) \* Order;

Subset[RESULT] = (Basic + ((Order \* (Subset[2] - 1) - (Subset[2] - 1) \* Subset[2] / 2 + Subset[3] - Subset[2] - 1) \* Order + Subset[4] - 1) \* 2 + Subset[5] + 1);

break;

case 2:

Basic = Order \* Order + Order \* (Order - 1) + Order \* (Order - 1) \* Order \* 2;

Subset[RESULT] = (Basic + ((Order \* (Subset[2] - 1) - (Subset[2] - 1) \* Subset[2] / 2 + Subset[3] - Subset[2] - 1) \* Order + Subset[4] - 1) \* 2 + Subset[5] + 1);

break;

default:

break;

}

}

//完成转换

if (Subset[SIGN] == 1)

{

P->ClauseValue[i] = Subset[RESULT];

}

else

{

P->ClauseValue[i] = -Subset[RESULT];

}

}

P = P->NextClause;

}

Formula.LitNum = Order \* Order + Order \* (Order - 1) \* (Order \* 3 + 1);

}

void DiggingHole(SATFormula& Formula, int\*\* B, int\* Puzzle, char\* filename)

{

int i = 0, j = 0;

for (i = 0; i < Order \* Order; i++)//循环挖洞，根据剪叶规则，每个单元格只挖一次

{

//准备工作

FormulaDelete(Formula);//删除上一步的公式

Constraint\_1(Formula, B);//进行三个约束

Constraint\_2(Formula, B);

Constraint\_3(Formula, B);

Puzzle[i] = -Puzzle[i];//将当前位置翻转

for (j = 0; j < Order \* Order; j++)//添加到公式集

{

AddClause(Puzzle + j, 1, Formula);

}

Convert(Formula);//转换

PuzzleSAT\_FileSave(Formula, filename);//保存在文件中

//求解数独

CreateAdjacencyList(filename, F);

Copy(S, F);

Reversal(S);

for (j = 0; j < 100; j++)

{

if (LearnCla[j] != NULL)

{

free(LearnCla[j]);

LearnCla[j] = NULL;

}

}

for (j = 0; j < 100; j++)

{

ConflicCla[j] = 0;

}

j = BetterExistSingle(S);

LearnNum = 0;

ConflicNum = 1;

CHOOSE = 2;

if (BetterDPLL(S, j, 1) > 0)

{

Puzzle[i] = -Puzzle[i];//若有解则不能挖洞，将原来的位置补回

}

else

{

Puzzle[i] = 0;//若无解则说明有唯一解，可以去掉

}

}

//结束工作，将挖好的数独的CNF公式保存

FormulaDelete(Formula);

Constraint\_1(Formula, B);

Constraint\_2(Formula, B);

Constraint\_3(Formula, B);

for (j = 0; j < Order \* Order; j++)

{

AddClause(Puzzle + j, 1, Formula);

}

Convert(Formula);

PuzzleSAT\_FileSave(Formula, filename);

printf("已随机生成数独，并保存到文件 %s 中", filename);

}

status Las\_vegas(SATFormula& Formula, int\*\* B, int\* Puzzle, char\* filename)//拉斯维加斯算法，用来随机生成终盘

{

int i = 0, j = 0, n = 0, ran = 0;

if (Order == 6)//若想生成6阶棋盘，随机9个

{

n = 9;

}

else if (Order == 8)//生成8阶棋盘，随机19个

{

n = 19;

}

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

{

ran = rand() % (Order \* Order);//随机确定棋盘位置

Puzzle[ran] = rand() % 2;//随机确定放入值

}

for (i = 1; i <= Order \* Order; i++)//逐个将放入的数字转化为ij形式，便于计算

{

if (i % Order != 0)//若转化的不是最后一个数

{

if (Puzzle[i - 1] == 0)//若放入的数为0

{

Puzzle[i - 1] = -(i % Order + ((i - i % Order) / Order + 1) \* 10);

}

else if (Puzzle[i - 1] == 1)//若放入的数为1

{

Puzzle[i - 1] = (i % Order + ((i - i % Order) / Order + 1) \* 10);

}

else//若没有放入数

{

Puzzle[i - 1] = 0;

}

}

else//若转化的是最后一个数

{

if (Puzzle[i - 1] == 0)//若放入的数为0

{

Puzzle[i - 1] = -(Order + (i / Order) \* 10);

}

else if (Puzzle[i - 1] == 1)//若放入的数为1

{

Puzzle[i - 1] = (Order + (i / Order) \* 10);

}

else//若没有放入数

{

Puzzle[i - 1] = 0;

}

}

}

//转化为CNF公式并保存

FormulaDelete(Formula);

Constraint\_1(Formula, B);

Constraint\_2(Formula, B);

Constraint\_3(Formula, B);

for (j = 0; j < Order \* Order; j++)

{

AddClause(Puzzle + j, 1, Formula);

}

Convert(Formula);

PuzzleSAT\_FileSave(Formula, filename);

//进行计算

CreateAdjacencyList(filename, F);

Copy(S, F);

Reversal(S);

for (j = 0; j < 100; j++)

{

if (LearnCla[j] != NULL)

{

free(LearnCla[j]);

LearnCla[j] = NULL;

}

}

for (j = 0; j < 100; j++)

{

ConflicCla[j] = 0;

}

j = BetterExistSingle(S);

LearnNum = 0;

ConflicNum = 1;

CHOOSE = 2;

if (BetterDPLL(S, j, 1) > 0)//若计算成功，即成功生成一个终局

{

for (i = 1; i <= Order \* Order; i++)//逐个将解转化为ij形式，方便计算

{

if (i % Order != 0)//放置非行尾的元素

{

if (VELUE[i] == 0)//当前解为0

{

Puzzle[i - 1] = -(i % Order + ((i - i % Order) / Order + 1) \* 10);

}

else//当前解不是0

{

Puzzle[i - 1] = (i % Order + ((i - i % Order) / Order + 1) \* 10);

}

}

else//行尾元素

{

if (VELUE[i] == 0)

{

Puzzle[i - 1] = -(Order + (i / Order) \* 10);

}

else

{

Puzzle[i - 1] = (Order + (i / Order) \* 10);

}

}

}

return OK;//返回成功

}

else//若没有成功

{

for (i = 0; i < Order \* Order; i++)

{

Puzzle[i] = -1;//清空棋盘

}

return ERROR;//返回失败

}

}

void FormulaDelete(SATFormula& Formula)

{

P\_ClauseList P = NULL, S = NULL;

P = Formula.FirstClause;

while (P != NULL)//循环释放公式集

{

S = P->NextClause;

free(P);

P = S;

}

Formula.FirstClause = NULL;

Formula.ClaNum = 0;//子句数清零

Formula.LitNum = 0;//符号数清零

ClaNumber = 0;

}

void Combine(int s[], int n, int k, SATFormula& Formula, void (\*Print)(int\* Subset, SATFormula& Formula))

{

if (k == 0)//如果要想选取的数目为零

{

Print(Subset, Formula);//打印选取数组

return;

}

for (int i = n; i >= k; i--)//从第n个开始往后选取

{

Subset[k - 1] = s[i];//将选取的数放入选取数组之中

if (k > 1)

{

Combine(s, i - 1, k - 1, Formula, Print);//选取之后的一个数

}

else//当选取数为零的时候

{

Print(Subset, Formula);//打印选取数组

}

}

}

void Print\_AddCla(int\* Subset, SATFormula& Formula)//将选取数组放入到公式集中

{

int k = 0; P\_ClauseList P = NULL;

//行输入

//正输入

if (!(P = (P\_ClauseList)malloc(sizeof(P\_ClauseNode))))

{

exit(OVERFLOW);

}

for (k = 0; k < Order / 2 + 1; k++)

{

P->ClauseValue[k] = Subset[k];

}

P->ClauseValue[k] = 0;

P->NextClause = Formula.FirstClause;

Formula.FirstClause = P;

ClaNumber++;

//负输入

if (!(P = (P\_ClauseList)malloc(sizeof(P\_ClauseNode))))

{

exit(OVERFLOW);

}

for (k = 0; k < Order / 2 + 1; k++)

{

P->ClauseValue[k] = -Subset[k];

}

P->ClauseValue[k] = 0;

P->NextClause = Formula.FirstClause;

Formula.FirstClause = P;

ClaNumber++;

//列输入

//正输入

if (!(P = (P\_ClauseList)malloc(sizeof(P\_ClauseNode))))

{

exit(OVERFLOW);

}

for (k = 0; k < Order / 2 + 1; k++)

{

P->ClauseValue[k] = (Subset[k] - Subset[k] % 10) / 10 + ((Subset[k] % 10) \* 10);

}

P->ClauseValue[k] = 0;

P->NextClause = Formula.FirstClause;

Formula.FirstClause = P;

ClaNumber++;

//负输入

if (!(P = (P\_ClauseList)malloc(sizeof(P\_ClauseNode))))

{

exit(OVERFLOW);

}

for (k = 0; k < Order / 2 + 1; k++)

{

P->ClauseValue[k] = -((Subset[k] - Subset[k] % 10) / 10 + ((Subset[k] % 10) \* 10));

}

P->ClauseValue[k] = 0;

P->NextClause = Formula.FirstClause;

Formula.FirstClause = P;

ClaNumber++;

}

void AddClause(int\* Subset, int n, SATFormula& Formula)

{

int k = 0; P\_ClauseList P = NULL;

if (!(P = (P\_ClauseList)malloc(sizeof(P\_ClauseNode))))//申请空间

{

exit(OVERFLOW);

}

if (Subset[0] != 0)//当公式子句不为空时

{

for (k = 0; k < n; k++)//逐个将公式子句中的内容添加

{

P->ClauseValue[k] = Subset[k];

}

P->ClauseValue[k] = 0;//放置结尾标志

P->NextClause = Formula.FirstClause;//链接到子句集中

Formula.FirstClause = P;

ClaNumber++;//子句数加一

}

Formula.ClaNum = ClaNumber;//放入子句数

}

# 附录10（系统使用说明）

根据提示语直接输入数字选项使用该功能。

主菜单：

1. SATSolve：（SAT求解）

1. ImportCNF（输入cnf文件文件名，并读取里面的公式）

2. ExportCNF（将数据结构中解析完成的公式输出）

3. DeleteCNF（将当前公式删除）

4. DPLL（使用DPLL求解，可以选择求解算法和选取方式）

0. Back（返回上级菜单）

2. PuzzleSolve：（二进制数独求解）

1. CreatePuzzle：（创建数独）

1. By yourselves（用户自己输入，输入以行列为准，如第一行第一个为1则输入11，若第三行第二个为0则输入-32，以0结尾）

2. ComputerRandom（计算机自行生成）

0. Back（返回上级菜单）

2. CheckPuzzle(查看当前数独)

3. SolvePuzzle（求解当前数独）

0. Back（返回上级菜单）

0. Exit（退出程序）