

**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： 计算机校交1803班**

**学 号： U201816136**

**姓 名： 王辰潇**

**指导教师： 祝建华**

**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**](#_Toc36491327)

[**1 引言 1**](#_Toc36491328)

[1.1 课题背景与意义 1](#_Toc36491329)

[1.1.1 课题背景 1](#_Toc36491330)

[1.1.2 课题意义 1](#_Toc36491331)

[1.2 国内外研究现状 2](#_Toc36491332)

[1.3 课程设计的主要研究工作 2](#_Toc36491333)

[**2 系统需求分析与总体设计 3**](#_Toc36491334)

[2.1 系统需求分析 3](#_Toc36491335)

[2.2 系统总体设计 3](#_Toc36491336)

[**3 系统详细设计 5**](#_Toc36491337)

[3.1 有关数据结构的定义 5](#_Toc36491338)

[3.2 主要算法设计 7](#_Toc36491339)

[3.2.1 图形界面 7](#_Toc36491340)

[3.2.2 数独模块 7](#_Toc36491341)

[3.2.3 逻辑模块 11](#_Toc36491342)

[**4 系统实现与测试 18**](#_Toc36491343)

[4.1 系统实现 18](#_Toc36491344)

[4.1.1 运行环境 18](#_Toc36491345)

[4.1.2 数据结构实现 18](#_Toc36491346)

[4.1.3 算法设计 28](#_Toc36491347)

[4.2 系统测试 32](#_Toc36491348)

[**5 总结与展望 44**](#_Toc36491349)

[5.1 全文总结 44](#_Toc36491350)

[5.2 工作展望 44](#_Toc36491351)

[**6 体会 45**](#_Toc36491352)

[**参考文献 46**](#_Toc36491353)

[**附录 48**](#_Toc36491354)

# 1 引言

## 1.1 课题背景与意义

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。SAT问题也是程序设计与竞赛的经典问题。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，具有研究性和实践性。

### 1.1.1 课题背景

对于SAT问题的研究从没有停止过，在1997年和2003年，H.Kautz与B.Selman两次列举出SAT搜索面临的挑战性问题，并于2011年和2007年，两度对当时的SAT问题研究现状进行了全面的综述。黄文奇提出的Solar算法在北京第三届SAT问题快速算法比赛中获得第一名。对SAT问题的求解主要有完备算法和不完备算法两大类。不完备算法主要是局部搜索算法，这种算法不能保证一定找到解，但是求解速度快，对于某些SAT问题的求解，局部搜索算法要比很多完备算法更有效。完备算法出现的时间更早，优点是可以正确判断SAT问题的可满足性，在算例无解的情况下可以给出完备的证明。对于求解SAT问题的优化算法主要有启发式算法、冲突子句学习算法、双文字监视法等。

### 1.1.2 课题意义

SAT问题是第一个被证明的NP完全问题，而NP完全问题由于其极大的理论价值和困难程度，破解后将会在许多领域得到广泛应用，从而在计算复杂性理论中具有非常重要的地位。由于所有的NP完全问题都能够在多项式时间内进行转换，那么如果SAT问题能够得到高效解决，所有的NP完全问题都能够在多项式时间内得到解决。对SAT问题的求解，可用于解决计算机和人工智能领域内的CSP问题（约束满足问题）、语义信息的处理和逻辑编程等问题，也可用于解决计算机辅助设计领域中的任务规划与设计、三维物体识别等问题。SAT问题的应用领域非常广泛，还能用于解决数学研究和应用领域中的旅行商问题和逻辑算数问题。许多实际问题，例如数据库检索、积木世界规划、超大规模集成电路设计、人工智能等都可以转换成SAT问题进而进行求解。可见对SAT问题求解的研究，具有重大意义。

## 1.2 国内外研究现状

1993年SAT问题被证明为是NP完全问题。

目前解决SAT问题的算法都是基于DPLL算法，并对其进行优化。DPLL算法引入单子句规则，对原本的深度优先搜索算法进行了大量剪枝。在此之后提出的chaff算法引入了WL数据结构，在2003年提出的miniSAT引入了学习子句和随机重启功能，而miniSAT也是目前常见的SAT-Solver的常用算法。目前国际上的比较新的进展是Glucose SAT-Solver。

## 1.3 课程设计的主要研究工作

本课程设计实现了对cnf文件的读入和输出，基础的DPLL算法以及对此进行的改进，包括非时间顺序回溯、启发式变元选择策略、学习子句和随机重启等优化，并调整相关参数的设置来提升效果。

此外，本课程设计也实现了二进制数独的图形界面游戏，可以随机生成数独，将数独转变为cnf文件，利用已实现的SAT-Solver解决，该数独有一定的交互性和可玩性，界面比较美观，实现了算法的应用。

# 2 系统需求分析与总体设计

## 2.1 系统需求分析

这本系统主要实现了两个问题：从cnf文件中读取SAT问题并求解、二进制数独游戏。

1. 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。有以下要求：

(1) 能够读取cnf文件，输出执行结果，并保存执行结果

(2) 能够对计算出的结果进行正确性的验证

(3) 给出求解时间

2. 二进制数独游戏。将二进制数独游戏转化为SAT问题进行求解，并具有一定的可玩性和交互性。有以下要求：

(1) 能够生成数独格局

(2) 能够将数独问题转化为SAT问题并导出cnf文件

(3) 能够将SAT的求解结果转化为数独的解

## 2.2 系统总体设计

本系统分为三个模块：图形主界面、数独模块、逻辑模块。

图形主界面：用于与用户交互，接受用户的指令并展示运算结果。用户可以输入要求解的cnf文件的地址，然后点击求解按钮，得到求解成功或者失败的提示。用户也可以选择生成数独，然后在数独对应的框内填数，并点击判断按钮判断自己填入的数字是否正确，也可以直接令程序求解，查看结果。

数独模块：用于处理二进制数独相关的问题。包括按照不同难度随机生成数独、将数独转化为cnf文件、将求解结果还原成数独等功能。

逻辑模块：利用DPLL算法解决当前的SAT问题。

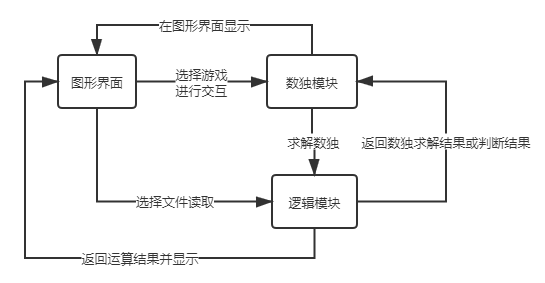


图2-1 系统总体设计

# 3 系统详细设计

## 3.1 有关数据结构的定义

两个重要的数据结构是文字(literal)和子句(clause)。

Lit结构体是文字的结构，包括一个映射后的值，将正负文字都映射到正的区间，便于下标值的操作。

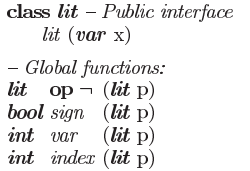


图3-1 lit结构体

Clause结构体是子句的结构，主要包含了三个部分。Learnt是一个状态，表示该子句是不是学习子句，activity是该子句的活跃值，用来处理后期的子句删除，lits代表了组成该子句的文字。

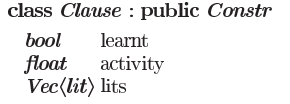
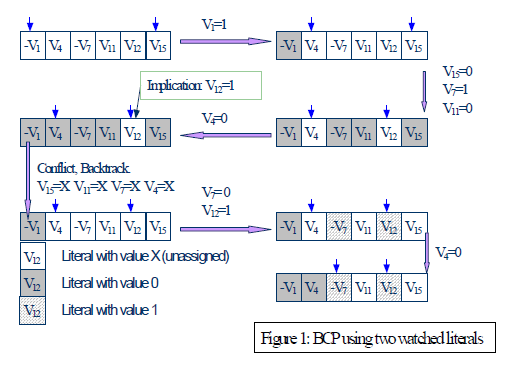


图3-2 Clause结构体

此外，指定lits中的前两个文字为观察值，相当于每个句子自带两个可移动的的指针，这是一种带观察值的数据结构（WL数据结构，2-literal watching），具体如下图所示：

 图3-3 2-literal watching结构

两个指针指向该子句中尚未被赋值的元素或者赋值为真的元素。被指向的元素被称为该子句的观察值。当两个指针分别指向不同的元素时，证明当前子句不可推断，即不是单子句。当两个指针指向同一个元素时，如果当前元素为假，则出现冲突；如果当前元素为真，则不做操作；如果当前元素未赋值，则可以推断出当前元素为真，并将该元素加入队列，用于以后的传播。

这两个指针没有先后次序，也就没有所谓的头和尾的概念，这样设置会带来很多好处，比如初始时这两个指针的位置可以是任意的，移动时也可以向前后两个方向移动，回溯时无需改动指针的位置。但这种设置也有弊端，即只有遍历完所有子句的文字后，才能识别出单元子句。相对应的，每个变量 v 也设置了两个 list 来分别存放以 watching 指针分为 v 以及非 v 的子句。当某个变量 v 赋值为 1 的话，watching 指针为 v 的子句可以忽略，watching 指针为非 v 的子句开始移动指针。

另外，我还根据代码的需要自己实现了队列（mQueue）和可变数组（mVector）。

## 3.2 主要算法设计

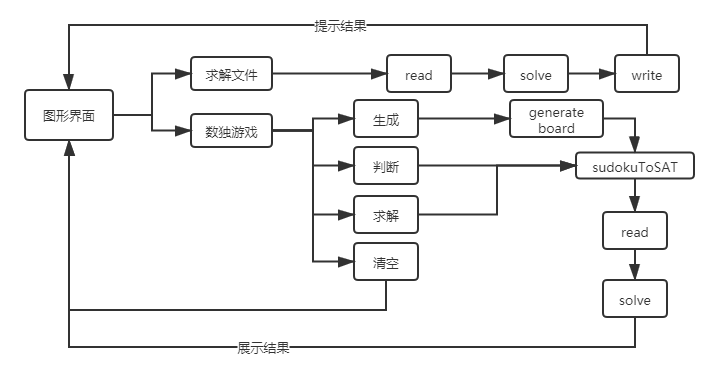
整体算法流程如下图所示：

图3-4 整体算法流程图

### 3.2.1 图形界面

图形界面用于与用户交互，接受用户输入并展示算法运行的结果。用户可以输入要求解的cnf文件的地址，然后点击求解按钮，得到求解成功或者失败的提示。用户也可以选择生成数独，然后在数独对应的框内填数，并点击判断按钮判断自己填入的数字是否正确，也可以直接令程序求解，查看结果。

### 3.2.2 数独模块

数独模块主要包括下面三个部分：

1. 随机生成数独棋盘。首先在8\*8的棋盘内随机选取指定个数个位置，依次填入0或1，每填一个数字都进行判断，确保满足约束条件，然后求解出整个棋盘。接着再根据用户选择的难度，在完整棋盘上挖空，显示出来。
2. 将数独转化为cnf文件。假设棋盘中已经给出的变元数量为count，那么整个棋盘可以转化为有1464个文字, 6520 + count个子句的cnf文件。具体转化的方法如下：

该游戏共有64个单元,每个单元对应一个布尔变元,并用该单元的行号与列号两位数进行表示｡如24表示第2行第4列单元对应的布尔变元,取真值时表示该单元填数字1;取假值时表示该单元填数字0｡图2.3左图中该单元已预填1,转换为SAT公式时产生单子句24｡同理,第1行2列单元预填0,则产生单子句¬12｡

除了这些单子句,还需将游戏规则的3条约束表示为对应的子句｡(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｡

对于约束(1),以第二行第4,5与6三个连续的单元为例,满足约束(1)必须:(24∨25∨26)∧(¬24∨¬25∨¬26)为真,即产生如下两个子句:

24∨25∨26,¬24∨¬25∨¬26

对于约束(2),以第3列为例,意味着本列中任选5个单元,则必须至少填一个1与0,不能全填1或全填0｡假如选择第3列的第1,3,4,6,7五个单元,则必须:(31∨33∨34∨36∨37)∧(¬31∨¬33∨¬34∨¬36∨¬37)为真,即产生如下两个子句:

31∨33∨34∨36∨37,¬31∨¬33∨¬34∨¬36∨¬37

对于约束(3),以第5行与第7行为例,不能有完全相同的填充,则须满足:

¬{[(51∧71)∨(¬51∧¬71)]∧[(52∧72)∨(¬52∧¬72)] ∧…∧[(58∧78)∨(¬58∧¬78)]}

但上式不符合CNF范式,我通过引入附加变元进行转换｡在这里,附加变元也可以用多位整数表示,且每位数字有相应含义,示例如下：

15711= 51∧71;15710= ¬51∧¬71;1571= 15711∨15710;

15721= 52∧72;15720= ¬52∧¬72;1572= 15721∨15720;

…

15781= 58∧78;15780= ¬58∧¬78;1578= 15781∨15780;

157= ¬[1571∧1572∧…∧1578].

其中,最高位数字1为行标志(2则表示列);次高位5及之后的一位7表示对应的第5行与第7行;第4位数字1,2,…,8分别表示行中的第1个单元,第2个单元,…,第8个单元;第5位取1或0,含义自明｡

因此,表示第5行与第7行不同的约束(3),需要引入附加变元25个;表示任意两行与两列不同的约束(3)共引入1400个附加变元;将会产生4536个子句｡举例说明如下:

15711= 51∧71转化为CNF时为

(51∨ ¬15711)∧(71∨ ¬15711)∧(¬51∨ ¬71∨15711)

即生成3个子句:51∨¬15711;71∨¬15711;¬51∨¬71∨15711｡

15720= ¬52∧¬72转化为CNF时为

(¬52∨ ¬15720)∧(¬72∨ ¬15720)∧(52∨72∨15720)｡

1578= 15781∨15780转化为CNF时为

(¬15781∨1578)∧(¬15780∨1578)∧(15781∨15780∨¬1578)｡

157= ¬[1571∧1572∧…∧1578] 转化为CNF时为

(¬157∨¬1571∨¬1572∨…∨¬1578)∧(1571∨157)∧(1572∨157)…(1578∨157),即可产生9个子句｡

将三个约束的所有具体要求分别转换成CNF子句集,连同预填提示数对应的单子句,便得到二进制数独游戏所生成的完整CNF公式｡

在实现DPLL求解算法时,布尔变元一般用连续的自然数表示,因此,上述对二进制数独游戏的布尔变元编码可进行如下转换(对第i行j列单元,M=8):

ij → (i-1)×8+j

int index(int i, int j)

{

return M \* i + j + 1;

}

这样,8阶二进制数独游戏每个单元对应的布尔变元自然数表示如下表:

表3-1 变元编码表

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
| 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
| 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 |
| 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 |
| 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 |

当利用DPLL算法求得对应CNF公式的解后,需通过上式对应的逆变换对解的含义进行解析,获得游戏的填充方案｡我对于引入的附加变元也从65起进行连续表示,转换函数如下：

int additionalIndex(int rc, int i, int j, int k)

{

return additionalIndex(rc, i, j) + 3 \* (k - 1) + 1;

}

int additionalIndex(int rc, int i, int j, int k, int l)

{

return additionalIndex(rc, i, j, k) + l + 1;

}

int additionalIndex(int rc, int i, int j)

{

return M \* M + (3 \* M + 1) \* ((rc - 1) \* M \* (M - 1) / 2 + M \* (i - 1) - i \* (i - 1) / 2 + j - i - 1) + 1;

}

其中i,j,k,l分别代表上述表示中的第1，2，3，4位，我设计了如上所示的映射以保证数字连续。rc表示是行还是列，M=8，是棋盘的边长。

这样，保证每一行和每一列没有三个连续的1都需要96个子句，保证每一行和每一列中1与0的个数相同各需要896个子句，保证没有相同的行或列需要4592个子句，这样总共需要6576个子句，再加上已给出的count个变元对应的count个子句，生成的文件中共6576+count个子句。

### 3.2.3 逻辑模块

算法的逻辑部分是基于DPLL框架进行改进的算法，加入了非时间顺序回溯、学习子句、依靠活跃值启发式选择变元等操作。

1.非时间顺序回溯（non-chronological backtracking）

最初我采用的DPLL框架使用了递归的方法实现，没有用到非时间顺序回溯，DPLL递归的思路如下：

DPLL(formula, assignment){

necessary = deduction(formula, assignment);

new\_asgnmnt = union(necessary, assignment);

if (is\_satisfied(formula, new\_asgnmnt))

return SATISFIABLE;

else if (is\_conflicting(formula, new\_asgnmnt))

return CONFLICT;

var = choose\_free\_variable(formula, new\_asgnmnt);

asgn1 = union(new\_asgnmnt, assign(var, 1));

if (DPLL(formula, asgn1)==SATISFIABLE)

return SATISFIABLE;

else {

asgn2 = union (new\_asgnmnt, assign(var, 0));

return DPLL(formula, asgn2);

}

}

这种方法很容易理解，但是效率较低。因此，我将DPLL改为用迭代而不是递归的形式描述，基于迭代的实现相对于基于递归的实现有以下优势：

1.递归速度慢且容易发生溢出，相对于迭代就有很多自身的劣势。

2.迭代具有非时间顺序回溯（智能回溯）的优势。

3.递归需要更多的内存存储空间。

迭代的伪代码如下:

status = preprocess(); //预操作

if (status!=UNKNOWN) return status;

while(1) {

decide\_next\_branch(); //变量决策环节

while (true) {

status = deduce(); //推理环节（BCP）

if (status == CONFLICT) {

blevel = analyze\_conflict(); //冲突分析

if (blevel == 0) return UNSATISFIABLE;

else backtrack(blevel);

//智能回溯，对应

}else if (status == SATISFIABLE) return SATISFIABLE;

else break;

}

}

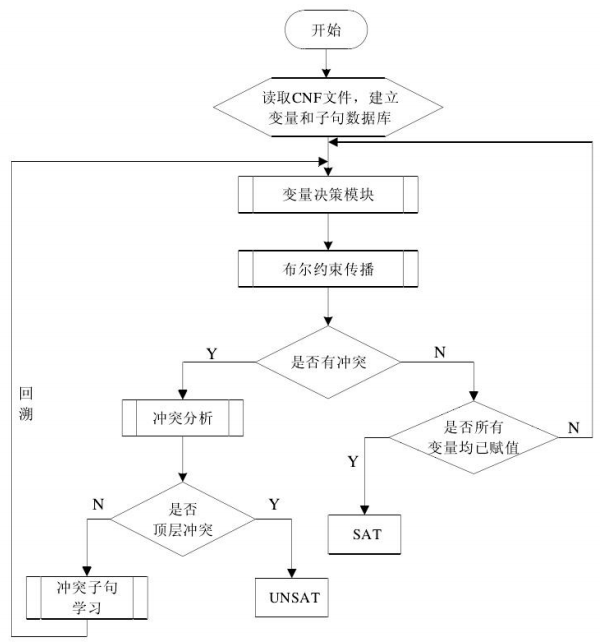
我在这种迭代框架的基础上实现了基于子句学习的非时间顺序回溯，流程图如下所示，下面具体介绍子句学习的过程。

图3-5 非时间顺序回溯流程图

2.冲突驱动子句学习（Conflict Driven Clause Learning）

这是本程序最重要的亮点。我在程序中运用了冲突驱动子句学习的算法来改进DPLL，极大的提升了求解速度。这是一种在DPLL基础上所做的优化，算法的思路如下：

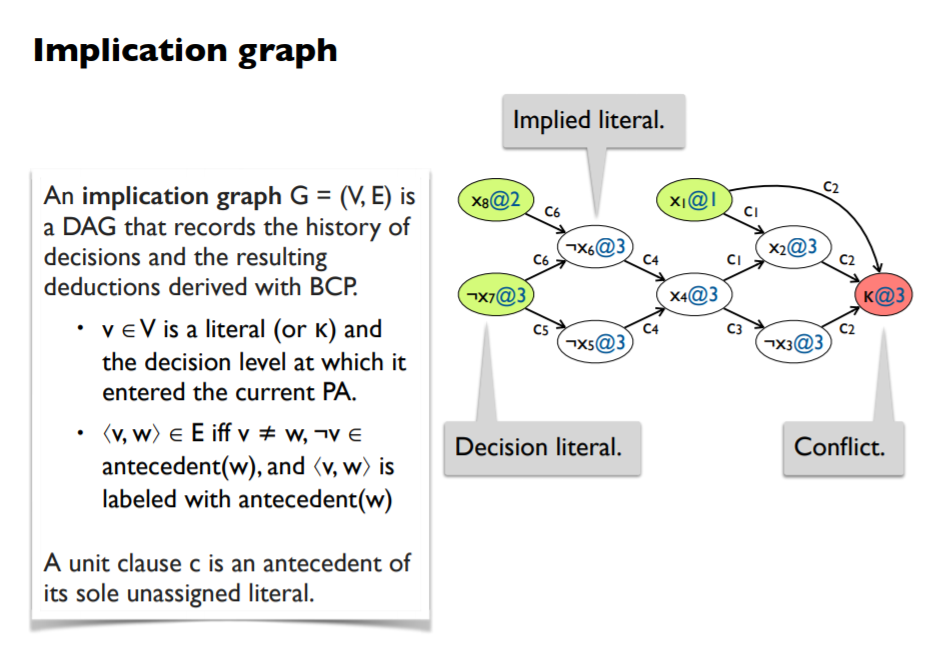
当遇到冲突时，它会查看已作出的推测，以及从 BCP 得到且最终导致冲突的那些赋值。我们把这些推测和由它们推出的结论画成一个图，称为蕴涵图（implication graph)，如下图所示：

图3-6 蕴涵图

图中可以反映出已作出的决定，BCP产生的文字赋值及其理由，以及它们如何导致冲突。通过观察此图，CDCL 能够学习到一个可能更有用的子句，而不仅仅是知道当前的部分赋值有错。这样，CDCL 就可以避免一遍又一遍地犯同样的错误，跳过 DPLL 会陷入的一大片错误的部分赋值。

下面通过一个例子来说明：

数据集中包含子句{1，2，3}，假设经过BCP传播之后在这个子句上发生了冲突，那么冲突的原因肯定是因为收到了（-1，-2，-3）这个条件。现在变元1的真值为假，是因为经过BCP过程得到了-1，假设推出-1的原因是（5，6 ），那么推出-1的这个子句就是{-5，-6，-1}，从这个分析中可以得出，（5，6，-2，-3）肯定也会引起冲突。为了避免这个冲突，可以把子句{-5，-6，2，3}加入到数据集中，这就是一个子句学习的例子。

子句学习的过程开始于一个句子变得不可满足（即产生冲突）的时候，我们想找到引起这一冲突的变元赋值，这对于一个矛盾的子句来说，将会是它含有的所有变元的赋值，而每一个赋值要么来源于搜索过程中的假设，要么来源于单子句传播得到的结果，再依次找到使得这些传播发生的变元赋值，从而继续这个反向传播过程，直到满足了终止条件，得到一系列导致冲突的变元赋值，然后生成一个阻止这些赋值发生的句子，并把它添加到句子数据集中。

学习子句主要有两个用途:驱动回溯过程和加速未来的冲突，因为它缓存了产生冲突的原因。每一个学习子句只能阻止有限个赋值，但是随着记录下来的子句互相再次基于已学到的子句构建，并且参与单子句传播过程，累积的学习效果会非常明显。

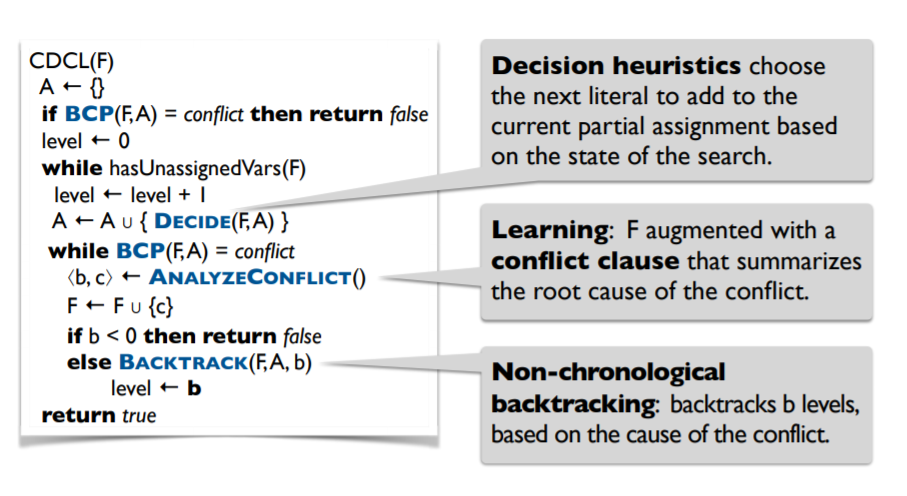
完整的子句学习的伪代码如下（基于迭代）：

图3-7 冲突驱动型子句学习伪代码

其中，BCP代表单子句传播过程，DECIDE代表选择下一个要赋值的变元，这里用了启发式决策，ANALYZECONFLICT是分析冲突并生成学习子句，决定回溯到的层数的函数，BACKTRACK可以根据冲突的分析结果直接回溯到指定的层b，实现了非时间顺序回溯。

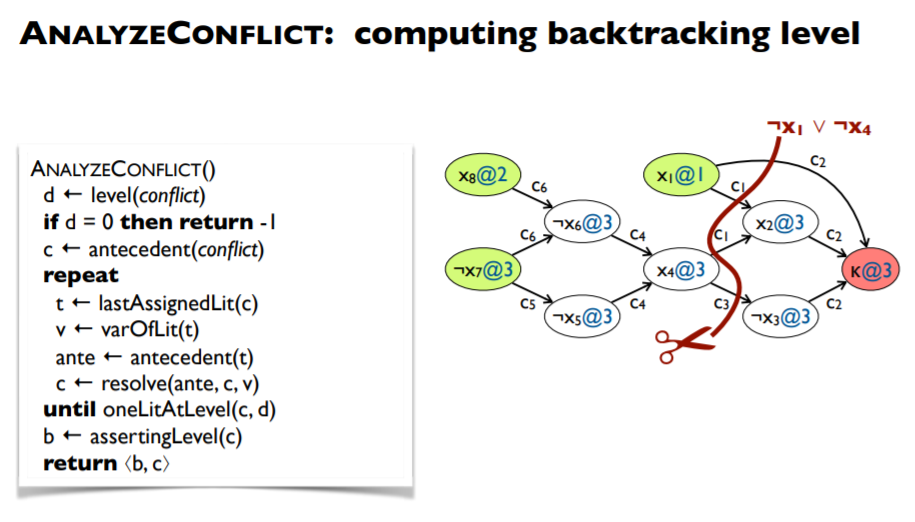
ANALYZECONFLICT是指分析冲突，这是算法最重要的一个函数，它的思路如下：

图3-8 分析冲突的思路图

这个函数中用到了First UIP(First Unique Implication Point)启发式算法的思路，即在距离冲突最近的一个UIP点处（图中的x4）剪枝，来使得产生的学习子句长度最短。

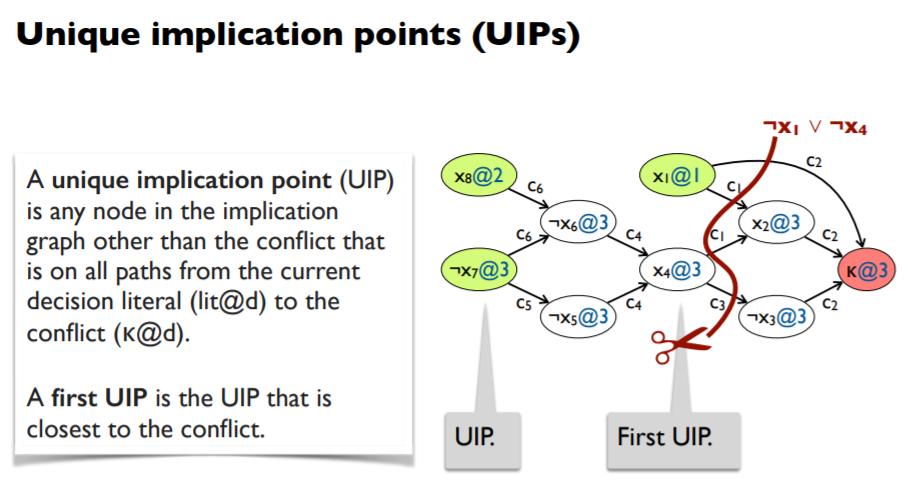
UIP点的定义如下：

图3-9 UIP点

当然，学习子句的增加会减缓传播的速度，因此学习子句不能无限制的添加，也要有适当的条件进行删除。我在本程序中使用了依靠活跃值来定期删除子句的策略，与变元的选择类似，在下面介绍。

3.基于活跃值的启发式变元选择策略（Activity Heuristics)

变元的选择策略可以非常大的影响程序运行的速度。在本程序中，我使用了CHAFF的论文中提到的，基于活跃值排序来选择下一个假设的变元的策略(VSIDS,Variable State Independent Decaying Sum) 。

每一个变量都有一个对应的活跃值（activity),每次当出现一个产生冲突的子句时，这个子句里的所有变量的活跃值都会增加（bumping）。在记录下冲突之后，系统中所有变元的活跃值都会乘以一个小于1的常数，这样就可以使变元的活跃值随时间衰减（decaying)，以保证新冲突带来的活跃值的增加比旧冲突更显著。

对于学习到的子句也是一样，在分析的过程中产生冲突的学习子句会增加活跃值，不活跃的学习子句隔一段时间会被删除。

# 4 系统实现与测试

## 4.1 系统实现

### 4.1.1 运行环境

处理器：intel(R) Core(TM) i5-8250U CPU @1.60GHz 1.80GHz

RAM: 8.00GB

Windows版本：Windows10家庭中文版

Qt Kit: Desktop Qt 5.9.9 MinGW 32bit

编译器：MinGW 5.3.0 32bit for C++

### 4.1.2 数据结构实现

由于系统定义的数据结构较多，这里简要介绍五个最主要的结构。

1.自定义的可变数组mVector，封装了插入元素、弹出元素、元素个数、清空元素、获取指定位置的元素等基本操作，为后面的代码编写提供了便利。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 自定义的mVector，用于存储可变数组

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template <class T>

class mVector{

public:

//默认构造函数

mVector(){

this->length=0;

this->maxlength=10;

this->data=new T[this->maxlength]();

};

mVector(const mVector& v){

this->length=v.length;

this->maxlength=v.maxlength;

this->data=new T[this->maxlength]();

for (unsigned i=0;i<this->maxlength;i++)

{

this->data[i]=v.data[i];

}

};

~mVector(){

this->length=0;

this->maxlength=0;

SafeDeleteArray(this->data);

};

//插入一个元素到最后

void push\_back(T element){

if (this->length>=this->maxlength)

{

unsigned i;

T\* dataTemp=new T[this->maxlength\*2]();

for (i=0;i<this->maxlength;i++)

{

dataTemp[i]=this->data[i];

}

this->maxlength=this->maxlength\*2;

SafeDeleteArray(this->data);

this->data=dataTemp;

}

this->data[this->length]=element;

this->length++;

};

//清空vector

void clear(){

SafeDeleteArray(this->data);

this->length=0;

this->maxlength=10;

this->data=new T[this->maxlength]();

};

void pop\_back(){

this->length--;

}

//获得vector元素个数

unsigned size(){

return this->length;

};

//重载[]操作符

T& operator[](unsigned i){

return this->data[i];

};

//重载=操作符

void resize(int i){

this->length=i;

}

mVector& operator=(const mVector & v){

this->length=v.length;

this->maxlength=v.maxlength;

SafeDeleteArray(this->data);

this->data=new T[this->maxlength]();

for (unsigned i=0;i<this->maxlength;i++)

{

this->data[i]=v.data[i];

}

return \*this;

};

private:

T\* data; //存储数据的数组

unsigned length; //数组元素

unsigned maxlength;

};

2.表示每个子句中对应的文字的结构体为lit。lit文字区分正负。每一个文字包括取非、取符号、取对应的变量、取值等操作。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* lit为变元，区分正负

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class lit

{

public:

int x;

lit()；

lit(int x)；

bool sign()；//return if the literal is signed

int var()；//return the underlying variable of the literal

int index()；//convert the literal to a "small" integer suitable for array indexing

int no()；//return the index of literal with opposite value

};

3.表示子句的结构体为Clause，其中的lits数组为包含的变元，数组中的前两个是观察指针。learnt表示该子句是否为学习子句，activty表示它的活跃值，calcReason函数在该子句产生冲突时记录产生冲突的原因，Clause\_new函数用于产生新的学习子句，locked函数用于判断该学习子句可不可以被化简掉（单子句不化简），simplify函数用于化简子句（去掉已经赋值的变元，仅限于迭代的第一层），propagate函数表示传播过程，用于判断和选择下一个赋值的变元。

class Clause

{

public:

bool learnt = 0;

double activity = 0;

mVector<lit> lits;

void calcReason(Solver& S, lit p, mVector<lit>& out\_reason);

bool Clause\_new(Solver& S, mVector<lit> ps, bool learnt, Clause\*& out\_clause);

bool locked(Solver S);

bool simplify(Solver& S);

bool propagate(Solver& S, lit p);

};

4. 自定义队列数据结构mQueue，基于链表实现了队列基本的插入、判空、元素个数、清空、弹出首尾元素等功能，为了便于后面的排序，也加入了转换成数组的功能。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 自定义数据结构mQueue，用作队列

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template <class T>

class mQueue

{

public:

mQueue() : Front(NULL), rear(NULL), count(0)

{

}

~mQueue()

{

clear();

}

void push\_back(const T& node)

{

if (Front == NULL)

Front = rear = new QueueNode(node);

else

{

QueueNode\* newqueuenode = new QueueNode(node);

rear->next = newqueuenode;

rear = newqueuenode;

}

count++;

}

bool empty() const

{

return Front == NULL;

}

int size() const

{

return count;

}

void clear()

{

while (Front)

{

QueueNode\* FrontofQueue = Front;

Front = Front->next;

delete FrontofQueue;

}

count = 0;

}

void pop()

{

QueueNode\* FrontofQueue = Front;

Front = Front->next;

delete FrontofQueue;

count--;

}

void pop\_back()

{

if(count==1){

delete Front;

count--;

}else{

QueueNode\* t=Front;

while(t->next!=rear){

t=t->next;

}

QueueNode\* rearofQueue=rear;

t->next=NULL;

rear=t;

delete rearofQueue;

count--;

}

}

T& front()

{

return Front->data;

}

T front() const

{

return Front->data;

}

T Rear() const

{

return rear->data;

}

T\* tolist()

{

T\* array=new T[count];

QueueNode\* p=Front;

for(int i = 0;i<count;i++){

array[i]=p->data;

p=p->next;

}

return array;

}

//private: //也可以直接用来链表list直接构造

struct QueueNode

{

T data;

QueueNode\* next;

QueueNode(const T& Newdata, QueueNode\* nextnode = NULL) : data(Newdata), next(nextnode)

{

}

// QueueNode() = default;

};

QueueNode\* Front; //队头指针

QueueNode\* rear; // 队尾指针

int count;

};

5.DPLLSolver是主求解器结构体，包含了求解所需要的各个成员变量，也封装了求解过程中用到的成员函数。

class DPLLSolver

{

friend Clause;

public:

/\*\*\*\*\*constraint database\*\*\*\*\*\*/

Clause \*\*constrs; //list of problem constraints

int constrs\_size = 0;

mQueue<Clause \*> learnts; //list of learnt clauses

double cla\_inc = 1; //clause activivty increment-amount to bump with

double cla\_decay = 1; //decay factor for clause activity

int literalNum;

int clauseNum;

/\*\*\*\*\*variable order\*\*\*\*\*\*/

double \*activity; //heuristic measurement of the activity of a variable

double var\_inc = 1; //variable activivty increment-amount to bump with

double var\_decay = 1; //decay factor for variable activity

/\*\*\*\*\*propagation\*\*\*\*\*\*/

mQueue<Clause \*> \*watches; //list of constraints watching each literal

mQueue<lit> propQ; //propagation queue

/\*\*\*\*\*assignments\*\*\*\*\*\*/

char \*assigns; //current assignments indexed on variables

mVector<lit> trail; //list of assignments in chronological order

mVector<int> trail\_lim; //separator indices for different decision levels in 'trail'

Clause \*\*reason; //constraint to imply each lit's value

int \*level; //decision level each variable is assigned

int root\_level = 0;

/\*\*\*\*\*result\*\*\*\*\*\*/

int \*model;

int model\_size = 0;

}

其中constrs是原cnf文件中包含的初始子句，learnts用来存储学习子句，cla\_inc和cla\_decay分别为子句活跃值的增加量和衰减度，literalNum和clauseNum分别是变元数和子句数。activity数组存储所有变元的活跃值，var\_inc和var\_decay分别为文字活跃值的增加量和衰减度。watches数组存储观察每个变元的子句，propQ是传播队列，里面存储了下一个传播的变元。assigns数组表示目前对于变元的赋值，trail记录了按时间顺序已经赋的值，trail\_lim记录每一决策层的赋值数，这两个数组用于回溯。reason数组记录了推断出每个变元真值的原因子句，level表示现在的决策层数，model记数组录最终结果，用于输出。此外还封装了成员函数，由于过于复杂此处不再列出，在下面的算法实现部分具体展开。

### 4.1.3 算法设计

系统的主要函数调用图如下：



图4-1 函数总体调用图

1．输入输出部分

（1）文件读入

函数名称：int read(char filename[])

函数说明：参数filename代表读入文件的名称，由于使用了图形界面读入文件，格式从QString转化为char\*，函数作用为解析cnf文件中的子句并将结果存放到对应的数据结构中，同时根据变元数量为各个变量分配存储空间。函数返回是否读取成功。

（2）文件输出

函数名称：void write(long long timer, int literalnum, char filename[])

函数说明：timer代表求解的时间，literalnum为变元个数，filename为文件的名称，作用为输出求解结果。

（3）棋盘读入

函数名称：void MainWindow::readboard()

函数说明：首先将棋盘赋初值为-1（表示未赋值），然后依次读入界面上各个输入框的值，空表示未赋值，0表示赋值为0，1表示赋值为1。

（4）棋盘输出

函数名称：void MainWindow::showboard()

函数说明：根据棋盘的值更新界面上各个输入框的值，1更新内容为1，0更新内容为0，-1更新内容为空。

（5）棋盘清空

函数名称：void MainWindow::on\_pushButton\_clear\_clicked()

函数说明：将所有输入框的内容改为空，对应棋盘的赋值改为-1。

2. 数独部分

（1）判断是否满足

函数名称：int isSatisfied(int x, int y, int board[M][M], int a)

函数说明：判断赋值board[x][y]=a是否满足数独的三个约束条件。

（2）创建棋盘

函数名称：void generateBoard(int board[M][M], int num)

函数说明：board是棋盘存放的位置，初始值均为-1，说明未赋值，num为随机赋初值的位置的数目。函数将在棋盘中随机赋值num个变元，每赋一个值调用一次isSatisfied函数判断一次是否满足数独条件，直至赋满m个初值。

（3）随机生成棋盘

函数名称：MainWindow::on\_pushButton\_generate\_clicked()

函数说明：调用generateBoard生成棋盘，然后求解出来，确保有解，无解的话再次生成直到有解，然后在解出来的完整棋盘上根据界面上用户选择的难度挖洞，将挖好洞的棋盘通过showboard函数展示出来。

（4）数独转化为cnf文件

函数名称：void SudokuToSAT(int board[M][M], FILE \*out, int count)

函数说明：根据已知的数独，按照规则将数独转化为cnf文件的格式并输出。

（5）给附加变元编号

函数名称：int additionalIndex(int rc, int i, int j, int k, int l)

函数说明：通过计算为所有的附加变元编号，将其映射为连续值。函数有三个重载，每个参数即代表原本附加变元的每一位。函数返回映射后的值。

3.逻辑部分

（1）求解

函数名称：int solve()

函数说明：不断调用sarch()函数执行搜索过程直到产生结果，每次都提高冲突子句和学习子句的上限（当冲突子句达到上限时，重新开始(restart)搜索过程，当学习子句达到上限时，调用reduceDB()函数减少一部分不活跃的学习子句），每次搜索都提高这两个上限，目的是为了随着搜索过程的深入，减少重启发生的概率。最后返回搜索完成的结果。

（2）搜索

函数名称：int search(int nof\_conflicts, int nof\_learnt, SearchParams params)

函数参数说明：nof\_conflicts是随机重启的条件，当冲突子句达到nof\_conflicts时，重新开始(restart)搜索过程，当学习子句达到nof\_learnt时，调用reduceDB()函数减少一部分不活跃的学习子句，params包含活跃值衰减参数。

函数思路：这是进行DPLL的主函数，循环调用propagate()函数进行单子句传播，如果出现冲突，先判断是否冲突到了最低层，如果是则直接返回False，如果不是就调用analyze()函数分析冲突，并通过cancelUntil()函数回退到相应层数，然后调用record()函数记录冲突子句，再衰减活跃值。如果没有出现冲突，就判断是否需要删除学习子句，再判断是否已经完成赋值，如果完成了就记录最终结果并return True，如果没有完成，判断是否需要随即重启，再选取活跃值最大的文字进行赋值。函数返回值为搜索状态（是否完成）。

（3）分析冲突

函数名称：void analyze(Clause\* confl, mVector<lit>& out\_learnt, int& out\_btlevel)

函数参数说明：confl是出现冲突的子句，out\_learnt是学习到的子句存放的位置，out\_btlevel是要返回的递归层数存放的位置。

函数思路：用广度优先的方式持续扩展分析当前决策层的文字，直到仅留下一个文字。对于每个文字，找出推断出这个文字赋值的子句，然后对这个子句中引起冲突的每个文字，判断它们所处的决策层，如果不是当前决策层，就将其反文字加入到学习子句中，将out\_btlevel更新为out\_btlevel和该文字所处的决策层中的较大值。过程中为了避免重复访问文字，使用标记数组进行访问标记。最后生成的学习子句存放在out\_learnt里面。

## 4.2 系统测试

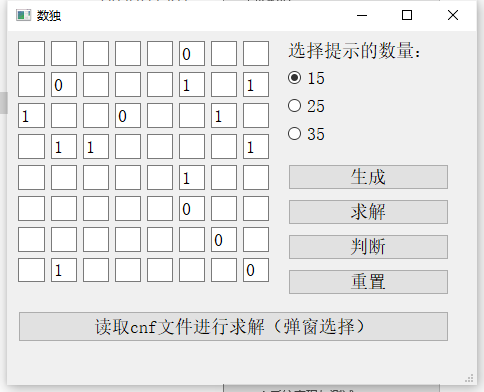
分别测试数独模块和cnf文件求解模块。数独模块主要测试在用户选择的难度下进行的生成求解判断重置操作，cnf文件求解模块主要测试不同的文件的求解情况，为了测试给出的解是否正确，我专门编写了res解的验证程序进行验证。

程序主界面如下：

图4-2 程序主界面

1.数独模块的验证：

①生成功能

依次选择15、25、35作为提示的数量，点击‘生成’按钮，生成棋盘。

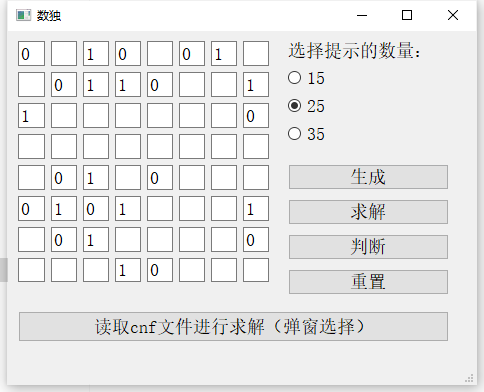
图4-3 生成带15个提示的棋盘

图4-4 生成带25个提示的棋盘

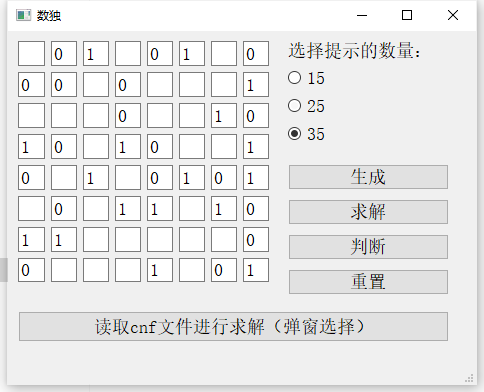


图4-5 生成带35个提示的棋盘

可见，生成棋盘的功能符合预期结果。

②求解功能

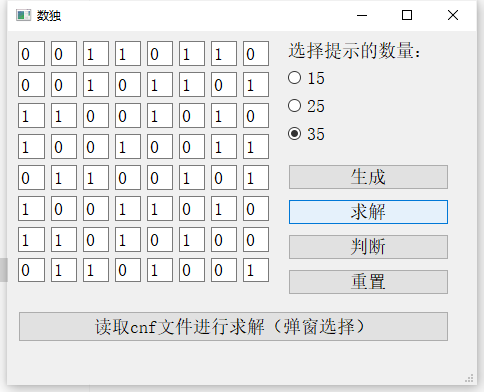
在生成的棋盘上点击求解按钮，结果如下图所示：

图4-6 点击求解按钮

求解出的棋盘符合二进制数独游戏的三个约束条件，求解功能正确。

③判断功能

判断功能可以判断当前数独是否有解，并给出提示。下面举例说明：

在使用‘生成’按钮随机生成的有解的棋盘上点击判断按钮，结果如图所示：



图4-7 判断数独有解

当前数独有解，判断正确。

下面更改数独为不符合约束条件的样子，再来测试此时的判断结果。

假如出现连续的三个0，结果如图所示：

图4-8 验证能否出现三个连续的0

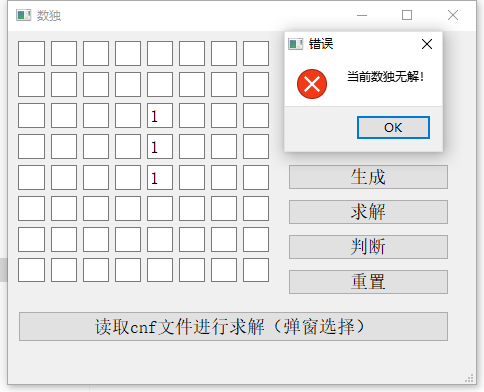
假如出现连续的三个1，结果如图所示：

图4-9 验证能否出现三个连续的1

假如出现两个完全相同的行，结果如图所示：

图4-10 验证能否出现两个完全相同的行

假如出现两个完全相同的列，结果如图所示：

图4-11 验证能否出现两个完全相同的列

假如一列中0和1的个数不等，结果如图所示：

图4-12 验证能否出现一列中0和1的个数不等

假如一行中0和1的个数不等，结果如图所示：

图4-13 验证能否出现一行中0和1的个数不等

由此可见，判断是否有解的功能正确。

④重置功能

按下重置按钮，棋盘清空，如图所示：

图4-14 验证重置功能

可见，重置功能正确。

2.cnf模块的验证

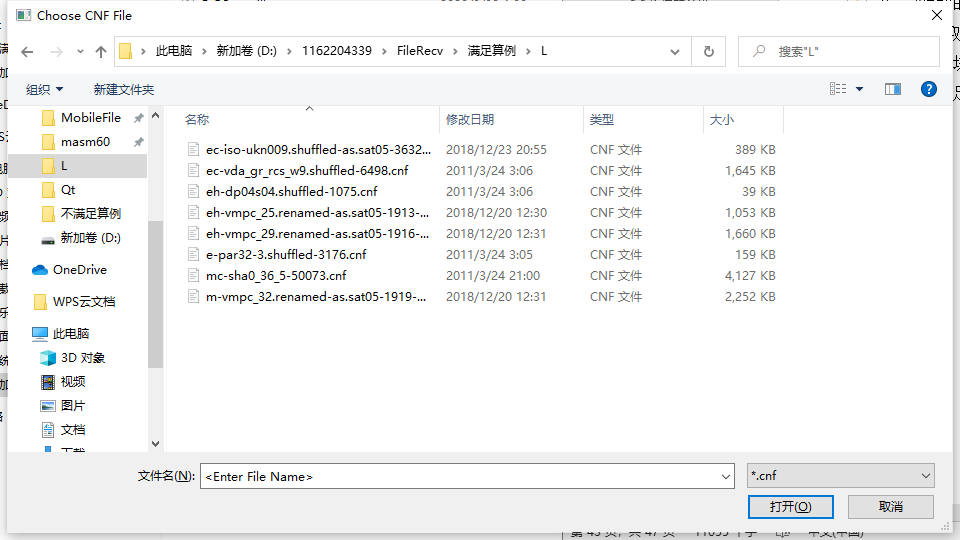
点击“读取cnf文件进行求解”的按钮，会弹出文件选择的界面：

图4-15 文件选择的界面

测试cnf模块主要分为可满足和不可满足两种算例进行测试。

1. 不可满足算例

在界面中选择tst\_v10\_c100.cnf

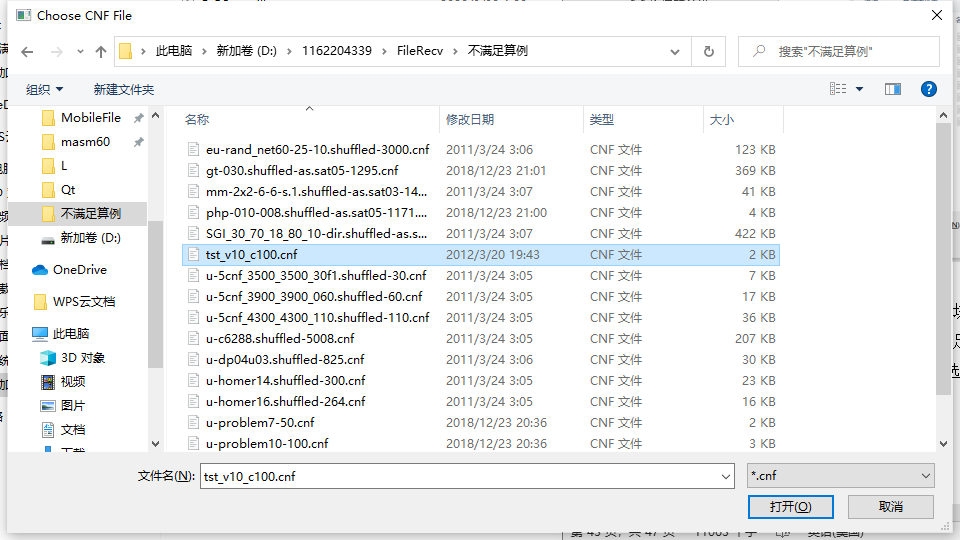


图4-16 选择tst\_v10\_c100.cnf

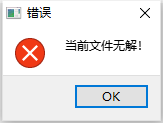
弹出提示：

图4-17 无解的提示

与此同时在同目录下输出了结果：tst\_v10\_c100.res

图4-18 tst\_v10\_c100.res

无解。这说明求解功能正确。

1. 可满足算例

选取L大小的算例eh-dp04s04.shuffled-1075.cnf

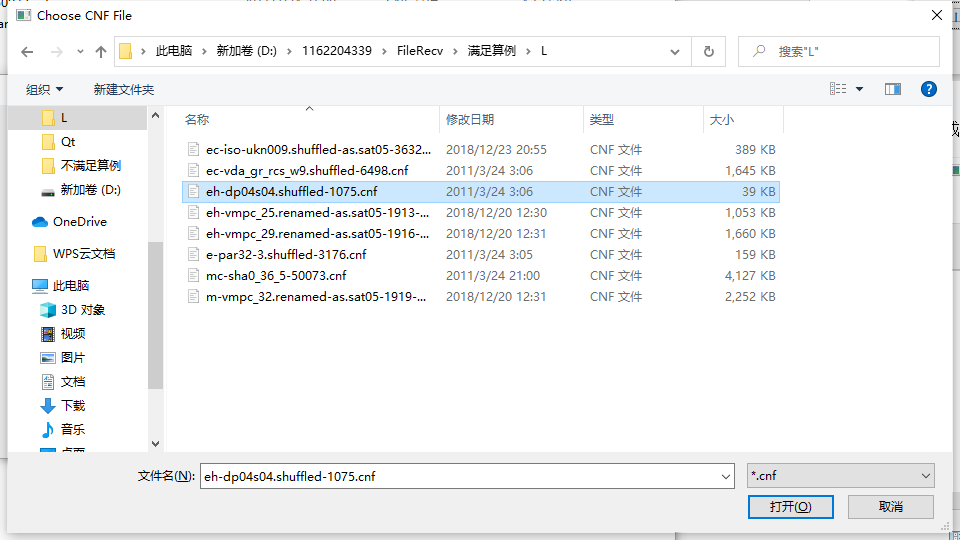


图4-19 选择算例

求解成功后弹出提示：

图4-20 求解成功的提示

也成功输出了文件：

图4-21 输出的文件

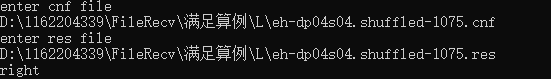
然后我用自己编写的验证程序对结果进行验证：

图4-22 验证程序的输出结果

结果正确。这说明cnf求解模块的功能正确。

下面验证cnf求解模块的性能：

表中的优化是与我开始写的没有加入子句学习的递归DPLL算法进行比较，可以看到优化之后效果非常明显。（表中的\表示没有解出来）

表4-1 测试算例

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 序号 | 文件名称 | 算例大小 | 变元数目 | 子句数目 | 是否满足 | 优化前ms | 优化后ms | 优化率 |
| 1 | tst\_v10\_c100 | S | 10 | 100 | 否 | 1 | 0 | 100.00% |
| 2 | sat-20 | S | 20 | 91 | 是 | 0 | 0 | 0.00% |
| 3 | 7cnf20\_90000\_90000\_7.shuffled-20 | S | 20 | 1532 | 是 | 1240 | 287 | 76.85% |
| 4 | unsat-5cnf-30 | S | 30 | 420 | 否 | 1125 | 74 | 93.42% |
| 5 | problem6-50 | S | 50 | 100 | 是 | 13 | 0 | 100.00% |
| 6 | problem9-100 | S | 100 | 200 | 是 | \ | 2 | 100.00% |
| 7 | u-problem10-100. | S | 100 | 200 | 否 | \ | 2 | 100.00% |
| 8 | ais10 | S | 181 | 3151 | 是 | 31826 | 368 | 98.84% |
| 9 | tst\_v200\_c210 | M | 200 | 210 | 是 | 4 | 2 | 50.00% |
| 10 | sud00082 | M | 224 | 1762 | 是 | 148 | 8 | 94.59% |
| 11 | sud00012 | M | 232 | 1901 | 是 | 521 | 21 | 95.97% |
| 12 | sud00861 | M | 297 | 2721 | 是 | 1777 | 15 | 99.16% |
| 13 | sud00001 | M | 301 | 2780 | 是 | 267 | 11 | 95.88% |
| 14 | sud00009 | M | 303 | 2851 | 是 | 2864 | 39 | 98.64% |
| 15 | sud00021 | M | 308 | 2911 | 是 | 51145 | 41 | 99.92% |
| 16 | eh-dp04s04.shuffled-1075 | L | 1075 | 3152 | 是 | 2909 | 112 | 92.40% |
| 17 | ec-iso-ukn009.shuffled-as.sat05-3632-1584 | L | 1584 | 3632 | 是 | \ | 207 | 100.00% |
| 18 | ec-vda\_gr\_rcs\_w9.shuffled-6498 | L | 6498 | 130997 | 是 | \ | 3759 | 100.00% |

# 5 总结与展望

## 5.1 全文总结

我在这次程序设计综合课程设计中做的主要工作如下：

（1）实现了基本的DPLL算法框架。

（2）查找资料，通过论文和PPT学习到了学习子句的算法。

（3）再根据论文实现了优化后的SAT求解器。

（4）实现了数独求解的功能，将SAT求解器应用于实际问题。

（5）学习使用Qt，实现了图形界面。

（6）体会到了完成一个完整程序的流程。

。

## 5.2 工作展望

在今后的研究中，围绕着如下几个方面开展工作：

（1）学习速度较快的唯一解二进制数独的生成方法。

（2）了解多线程的知识，使耗时操作在线程中进行，防止程序无响应。

（3）增加程序的提示信息和交互性，使用户体验更好。

# 6 体会

在本次程序设计综合课程设计中，我学到了很多东西。

比如这是我第一次尝试根据英文论文复现模型，我发现了论文的严谨性和可读性是可以兼备的，一篇论文既可以严谨的介绍出模型和实现方法，也可以让人清楚的理解各个部分的功能和实现思路，好的论文可以带给我醍醐灌顶的感觉。我在看文献的过程中也提高了英文的阅读能力。

此外，我通过编写各个模块然后组装，完整的实现了一个系统。这也让我对程序的实现流程有了更多的认识。我认识到，模块化便于修改和测试，而且代码之间耦合越少修改越方便。

另外，我在不断追求更快速度的过程中，优化代码结构，尽可能不做赘余的工作，这个不断分析和改进再debug的过程也让我对数据结构有了更加清楚和深刻的理解，我真正明白了不同的数据结构是如何与代码逻辑紧密联系，共同发挥作用的。

总的来说，我认为这是一份非常好的课程设计作业，我在完成的过程中学到了很多。

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]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 for Programming, Artificial Intelligence and Reasoning, LPAR 2005, pages 11–15, 2005.

[8] 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.

[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>

[11] Baochen SUN, Xiwei SUN, Yue WU,etc. A New Algorithm for Generating Unique-solution Sudoku. IEEE Fourth International Conference on Natural Computation,2008

[12] Niklas Eén, Niklas Sörensson An Extensible SAT-solver

https://www.comp.nus.edu.sg/~cs3234/notes/MiniSat.pdf

[13] Add Gritman, Anthony Ha, Tony Quach, Derek Wenger Conflict Driven Clause Learning

[14] Lecture 2 Slides for CSE 507 A Modern SAT Solver

https://courses.cs.washington.edu/courses/cse507/17wi/lectures/L02.pdf

[15] Matthew W. Moskewicz. Chaff: Engineering an Efficient SAT Solver

[16] Joao Marques-Silva, Ines Lynce, and Sharad Malik. Chapter 4:

Conflict-Driven Clause Learning SAT Solvers. Handbook of Satisfiability. 2008.

# 附录

源程序放在了单独的文档中，所以不在此处展示。