

**课程设计报告**

**题目： 基于SAT的数独游戏求解程序**

**课程名称： 程序设计综合课程设计**

**专业班级：**

**学 号：**

**姓 名：**

**指导教师：**

**报告日期： 2019年3月20日**

**计算机科学与技术学院**

# 任务书

**设计内容**

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

**设计要求**

要求具有如下功能：

1. **输入输出功能：**包括程序执行参数的输入，SAT算例cnf文件的读取，执行结果的输出与文件保存等。(15%)
2. **公式解析与验证：**读取cnf算例文件，解析文件，基于一定的物理结构，建立公式的内部表示；并实现对解析正确性的验证功能，即遍历内部结构逐行输出与显示每个子句，与输入算例对比可人工判断解析功能的正确性。数据结构的设计可参考文献[1-3]。(15%)
3. **DPLL过程：**基于DPLL算法框架，实现SAT算例的求解。(35%)
4. **时间性能的测量：**基于相应的时间处理函数（参考time.h），记录DPLL过程执行时间（以毫秒为单位），并作为输出信息的一部分。(5%)
5. **程序优化：**对基本DPLL的实现进行存储结构、分支变元选取策略[1-3]等某一方面进行优化设计与实现，提供较明确的性能优化率结果。优化率的计算公式为：[(t-to)/t]\*100%,其中t 为未对DPLL优化时求解基准算例的执行时间，to则为优化DPLL实现时求解同一算例的执行时间。(15%)
6. **SAT应用：**将数独游戏[5]问题转化为SAT问题[6-8]，并集成到上面的求解器进行问题求解，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[6-8]。(15%)

# 目录

[任务书 I](#_Toc4279136)

[目录 II](#_Toc4279137)

[1引言 3](#_Toc4279138)

[1.1课题背景与意义 3](#_Toc4279139)

[1.2国内外研究现状 4](#_Toc4279140)

[1.3课程设计的主要研究工作 5](#_Toc4279141)

[2系统需求分析与总体设计 7](#_Toc4279142)

[2.1系统需求分析 7](#_Toc4279143)

[2.2系统总体设计 8](#_Toc4279144)

[3系统详细设计 9](#_Toc4279145)

[3.1有关数据结构的定义 9](#_Toc4279146)

[3.2主要算法设计 11](#_Toc4279147)

[4系统实现与测试 18](#_Toc4279148)

[4.1系统实现 18](#_Toc4279149)

[4.2系统测试 21](#_Toc4279150)

[5总结与展望 29](#_Toc4279151)

[5.1全文总结 29](#_Toc4279152)

[5.1工作展望 29](#_Toc4279153)

[6体会 30](#_Toc4279154)

[参考文献 31](#_Toc4279155)

[附录 源代码 32](#_Toc4279156)

# 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。

**1.1.1 SAT问题描述**

SAT问题一般可描述为：给定布尔变元集合{*x*1, *x*2, ..., *x*n}以及相应的子句集合{*c*1, *c*2, ..., *c*m}，对于合取范式（CNF范式）：*F* = *c*1∧*c*2∧...∧*c*m，判定是否存在对每个布尔变元的一组真值赋值使*F*为真，当为真时（问题是可满足的，SAT），输出对应的变元赋值（一组解）结果。

**1.1.2 SAT合取范式一般存储格式.cnf文件**

一个CNF SAT公式或算例的具体信息通常存储在一个.cnf文件中，下图1.1是算例problem1.cnf文件前若干行的截图。



图1.1 cnf文件格式

在每个cnf文件的开始，由‘c’开头的是若干注释说明行；‘p’开头的行说明公式的总体信息，包括：范式为CNF；公式有200个布尔变元，由1到200的整数表示；320个子句。之后每行对应一个子句，0为结束标记。46表示第46号变元，且为正文字；-46则是对应的负文字，文字之间以空格分隔。实验目标为读取cnf文件后求解其可满足性。

## 1.2国内外研究现状

SAT问题是世界上第一个被证明的NPC问题。SAT问题在计算机科学、复杂性理论、密码系统、人工智能等领域发挥着至关重要的作用。然而，促使SAT问题成为持续研究热点的主要原因在于他在现实应用中的重要性。许多包含数以万计变量和数百万约束的组合问题 都可运用SAT求解技术处理。

**1.2.1 在不同领域的应用**

SAT求解器作为核心搜索引擎应用广泛，下面为目前不同领域的一些应用。

(1) 数学领域：数学密码、判定图和子图同构、寻找生成树、欧拉回路、布尔函数的函数依赖识別、旅行商、图着色等问题。

(2) 人工智能领域：知识编译、规划问题、机器识別、弹道轨迹、任务计划、神经网络计算、自动推理、DNA智能可编程片段生成等问题。SAT求解技术是人工智能的核心技术。

(3) 计算机科学领域：约束满足问题、难组合类问题、N皇后问题、工作流可满足性问题、护理调度问题、扩展推理、真值维护、定理证明、数据库检索与维护、数据挖掘、AES密钥调度、语义信息处理等^

(４)计算机辅助设计与制造领域：软件模型检查、硬件模型检测、计算机系统结构设计、VLSI集成电路设计与验证、错误诊断、有限状态系统的模型检测、FGPA路由、逻辑合成的技术映射、图像解释、寄存器分配、时序问题等。

**1.2.2面临的挑战**

求解SAT问题的目标是做出正确有效的判定，这可能需要指数级运行时间。在追求越来越快且有效执行的求解方法时，不可避免会遇到一些错误。一旦SAT求解技术过于复杂.且需要大量的专业知识去理解概念的正确性，以及如何有效 运行。尽管现有的SAT求解方法已经取得巨大成功但仍有一些问题没有得到高 效解决，已经解决的问题时能还存在更好求解方法。一些高效求解器忽视了算法的正确性和完备性，因此研究并实现高效率的求解方法仍是当前要解决的中心问题。未来研究主要包括编码、预处理、确定性算法、随机类算法、求解器实现、并行求解等问题。

## 1.3课程设计的主要研究工作

DPLL算法是经典的SAT完备型求解算法，对给定的一个SAT问题实例，理论上可判定其是否满足，满足时可给出对应的一组解。本设计要求实现基于DPLL的算法与程序框架，包括程序的改进也必须在此算法的基础上进行。

**1.3.1 DPLL算法介绍**

DPLL算法是一种基于树的回溯算法，主要使用两种基本处理策略：

单子句规则。如果子句集*S*中有一个单子句*L*,那么*L*一定取真值，于是可以从*S*中删除所有包含*L*的子句（包括单子句本身），得到子句集*S*1，如果它是空集，则*S*可满足。否则对*S*1中的每个子句，如果它包含文字*¬L*,则从该子句中去掉这个文字，这样可得到子句集合*S*2。*S*可满足当且仅当*S*2可满足。单子句传播策略就是反复利用单子句规则化简*S*的过程。

分裂策略。按某种策略选取一个文字*L*.如果*L*取真值，则根据单子句传播策略，可将*S*化成*S*2；若*L*取假值（即*¬L*成立）时，*S*可化成*S*1.

根据上述规则可不断对公式化简，并最终达到终止状态，其执行过程可表示为一棵二叉搜索树,如下图1.2所示。



图1.2 DPLL算法搜索树

**1.3.2算法描述**

基于单子句传播与分裂策略的DPLL算法可以描述为一个如后所示的递归过程DPLL( *S* ),为了优化执行效率，一般用非递归实现。

DPLL( *S*) :

/\* *S*为公式对应的子句集。若其满足，返回TURE；否则返回FALSE. \*/

{

while(*S*中存在单子句) {//单子句传播

在*S*中选一个单子句*L*；

依据单子句规则，利用*L*化简*S*；

if *S* = Φ return(TRUE);

else if (*S*中有空子句 ) return（FALSE）；

}//while

基于某种策略选取变元*v*；//策略对DPLL性能影响很大

if DPLL（*S* ∪*v* ）return(TURE);

return DPLL(*S* ∪¬*v*);

}

# 2系统需求分析与总体设计

## 2.1系统需求分析

**2.1.1基本需求：**

1. 用户选择读取目标cnf文件
2. 基于DPLL算法实现简单的SAT求解器，借助DPLL算法求解cnf文件的可满足性，将计算结果以res文件的格式存储。
3. 设计检查功能，协助判断计算结果的正确性。
4. 对分支变元选取策略等进行优化以改进DPLL算法的性能
5. 设计数独，生成数独cnf文件后以上述DPLL求解并返回数独格局

**2.1.2 用户需求如图2.1所示**



图2.1 用户需求

## 2.2系统总体设计

**2.2.1 系统设计**

系统分为5个模块：

交互与显示模块（display），

CNF解析模块（cnfparser），负责cnf文件的读取

核心DPLL模块( solver)，负责求解cnf文件并在res文件中写入结果

数独模块（sudoku）,包括数独格局生成、归约为cnf文件、求解cnf、返回数独格局

主控模块（main），为整个程序的入口

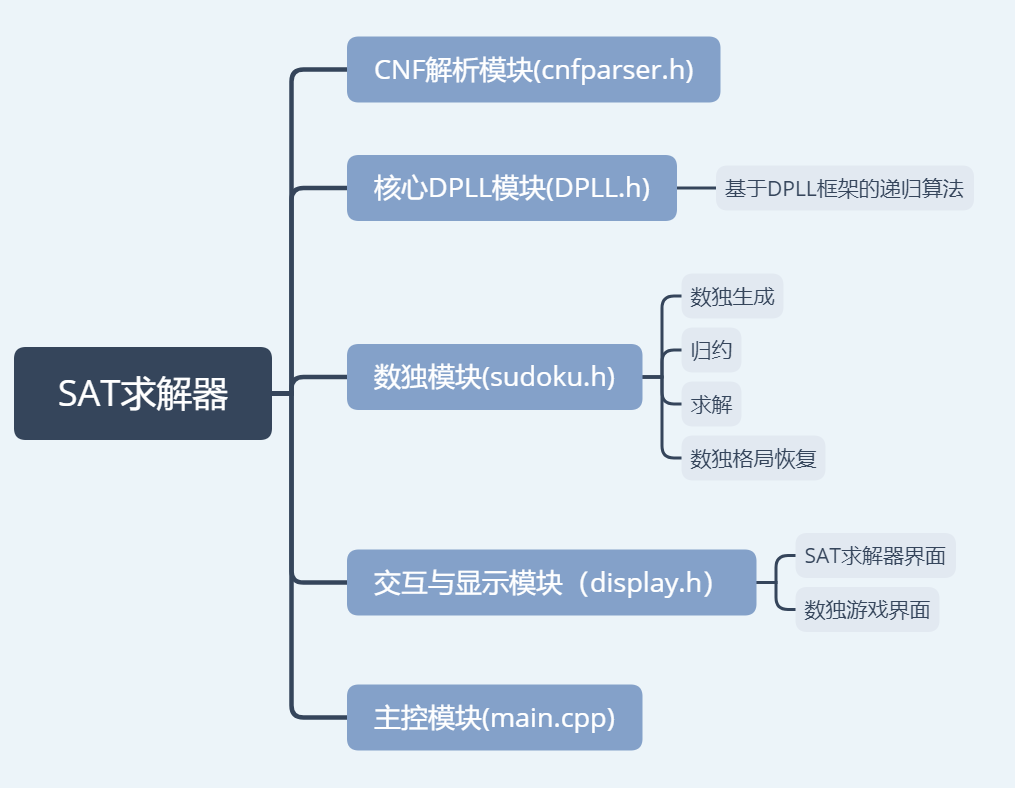
**2.2.2 系统模块结构图如图2.2所示**

图2.2 系统模块结构

# 3系统详细设计

## 3.1有关数据结构的定义

**3.1.1 数据结构类型­**

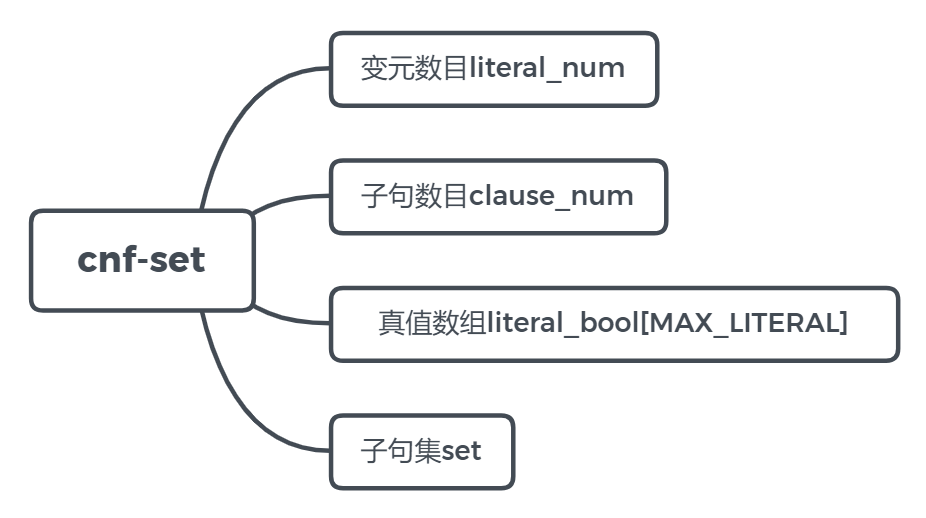
（1）综合结构体cnf-set,包括子句集set，记录变元数目（int），记录子句个数（int），记录变元真值（int数组），如图3.1

图3.1 cnf\_set结构体

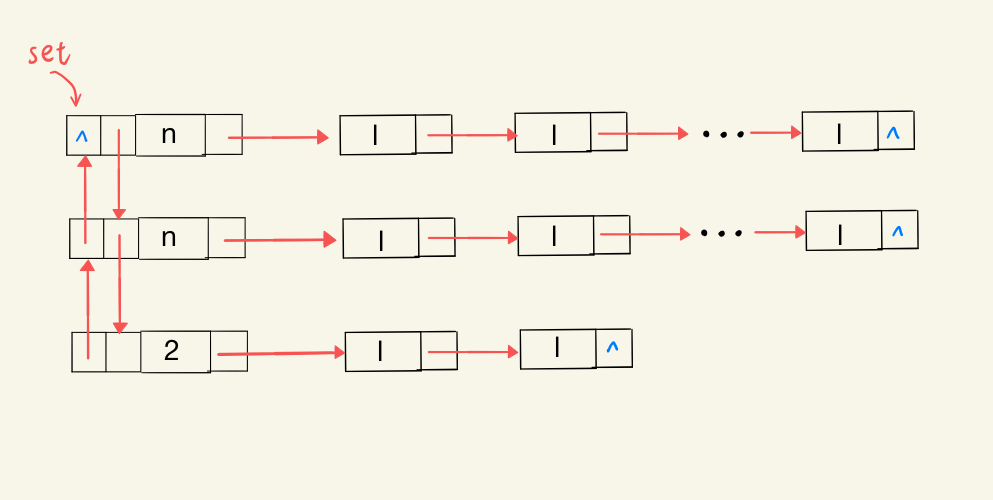
（2）主要结构体为子句集，综合到添加子句，删除子句，查找变元，判断子句变元数目等操作，选择以邻接表的物理结构来存储子句集。邻接表的结点分为头结点和变元结点，二者的结构不同。头结点包括3个指针域以及1个数据域，指针域分别指向上一个和下一个头结点以及子句的第一个变元结点，数据域存储该子句的变元数目。变元结点包括1个指针域和1个数据域，指针域指向下一个变元结点，数据域存储变元序号及真值。对应的物理结构如图3.2所示：

图3.2 set结构体

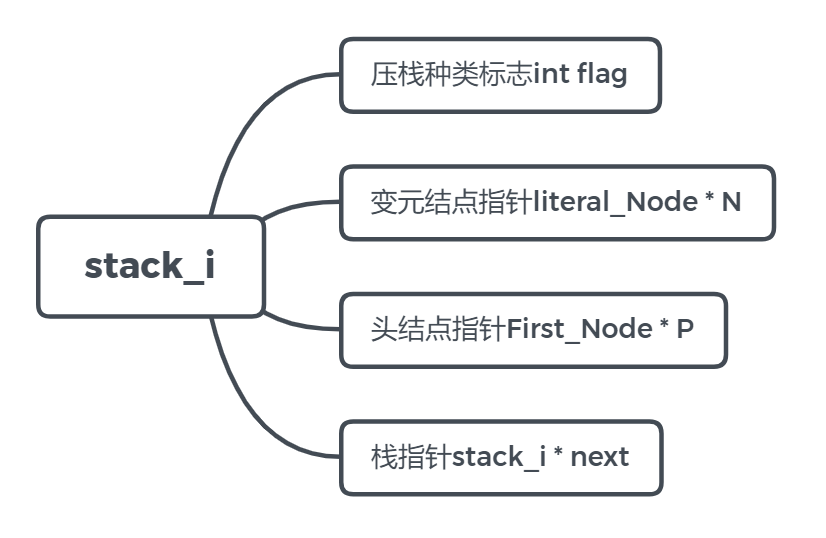
（3）堆栈结构，用来存储在化简过程中删除的变元，在回溯的过程中进行出栈操作以恢复子句集。结构stack\_i包括压栈标志flag，头结点指针，变元结点指针，指向上一层栈的指针。如图3.3

图3.3 stack\_i 结构体

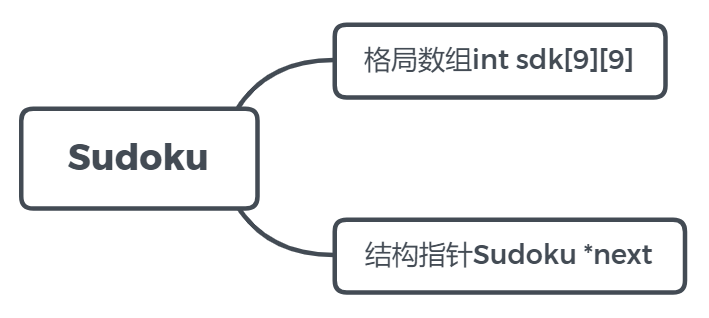
（4）数独结构。包括格局数组sdk[9][9]和结构指针，如图3.4所示；

图3.4 Sudoku 结构体

（5）系统的所有数据结构如表3.1所示：

|  |  |  |
| --- | --- | --- |
| 数据结构 | 数据项 | 数据类型 |
| Cnf-set | Literal\_num  clause\_num  literal\_bool  set | int  int  int数组  结构体 |
| First\_node | num  next\_literalnode  next\_firstnode  pre\_firstnode | int  指针  指针  指针 |
| Literal\_node | literal\_info  next\_literalnode | int  指针 |
| Stack\_i | flag  literal\_Node \* N;  First\_Node \* P;  stack\_i \* next; | Int  指针  指针  指针 |
| sudoku | sdk[9][9];  sudoku \*next; | int数组  指针 |

表3.1 系统数据结构

**3.1.2 数据结构关联**

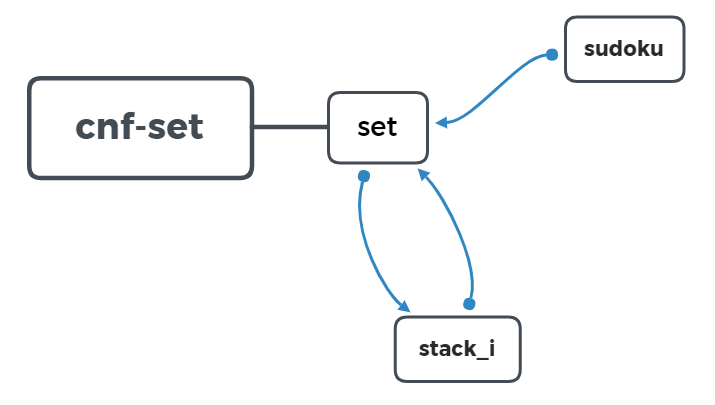
实验中最主要的数据结构为子句集set，DPLL的整个过程基于对子句集的操作，子句集为总的结构体cnf-set的数据项。栈结构stack\_i用来存储在DPLL单子句传播中删除的子句以及变元，为回溯过程做准备。数独结构sudoku存储的数独格局将转化为cnf文件后读取到子句集set中利用DPLL进行求解。其关联如图3.5所示。

图3.5 数据结构关系

## 3.2主要算法设计

（1）文件读取模块（cnf\_parser）

模块作用：读取cnf文件并构建子句集

核心函数：Create\_Set(cnf\_set \*S, int tag)

实现方式：由用户键入文件名，成功读取文件后，首先利用判断读取到的第一个字符是否为c的方式过滤掉所有的注释。跳出循环后用fscanf函数跳过’cnf’.

以字符格式读入变元数目和子句数目后，通过atoi函数将字符串转化为数字后存入结构体cnf\_set的literal\_num和clause\_num成员项中。对于后续的每一个子句，以两层循环进行读取，外层循环控制首节点的读取，结束条件为达到子句数目；内层循环控制子句中变元的读取，结束条件为指针为空值。在每次内层循环结束后，为首节点的数据域赋值计数变量i作为子句变元数目，同时进行不同子句之间的链接（第一行的情况需要独立出来）。

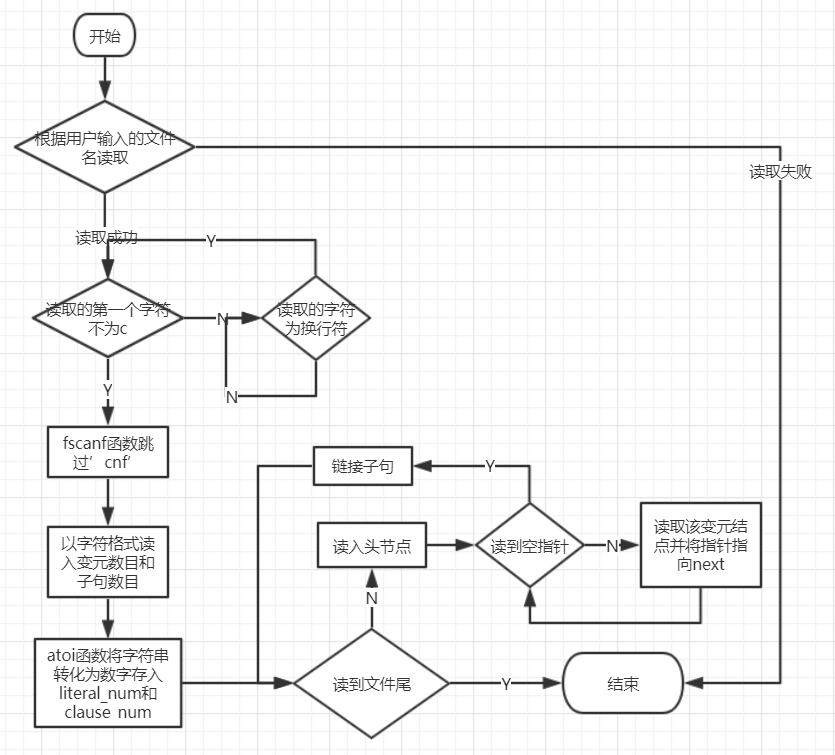
图3.6为函数流程图：

图3.6 Create\_Set流程图

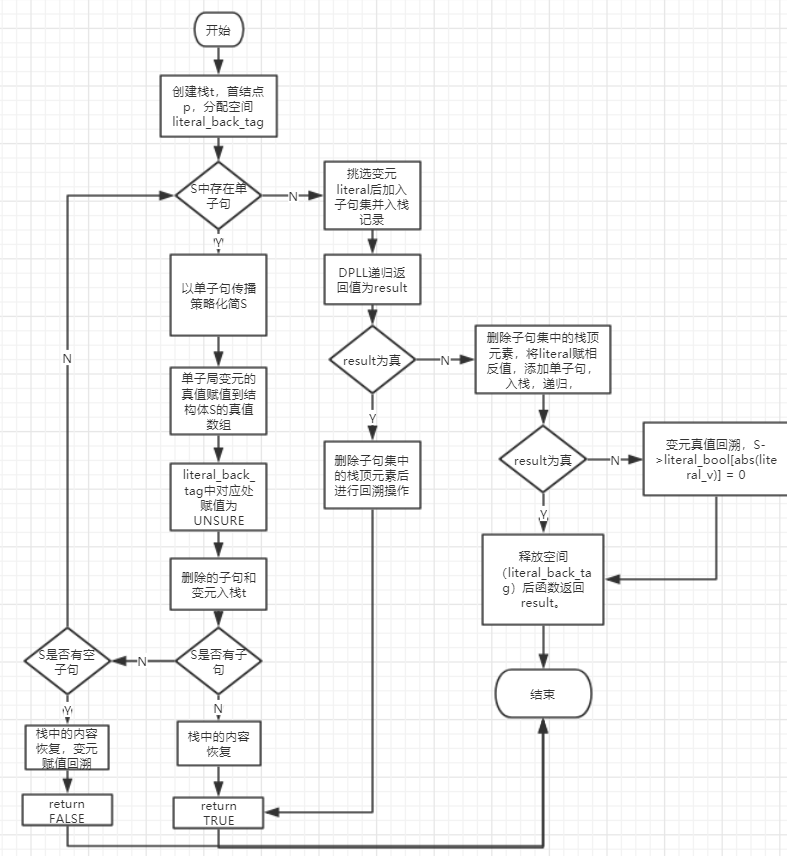
（2）DPLL求解模块（DPLL）

模块作用：根据构建的子句集求解其可满足性

实现方式：在函数的入口创建一个栈t和一个首结点p并分配一个单独的数组空间\*literal\_back\_tag，赋值均为0。当子句集S中存在单子句时，通过单子局传播策略化简单子句，将化简过程中的单子局变元的真值赋值到结构体S的真值数组的对应处，literal\_back\_tag中对应处赋值为UNSURE作为标志，方便在回溯的过程中进行变元真值的恢复。同时依据化简的种类将删除的子句和变元以不同标记的方式入栈t中。此时对S进行判断，若S无子句则将栈中的内容恢复，函数返回TRUE；若S中有空子句则将栈中的内容恢复，同时将上述化简过程中赋予真值的变元进行恢复，之后函数返回FALSE。若函数继续向下进行，说明之前未发生冲突且未求解完成，返回判断函数中是否有单子句。跳出循环后，利用设计好的变元选区策略选出变元literal，利用子句集操作将其加入到子句集中，并将这个子句入栈。进行函数递归DPLL（S），将返回值记为result。若result为真，则从子句集（此时的子句集已经回溯到递归之前的状态单多了一个认为添加的单子句，通过栈中存储的地址将其删去即可恢复原状态）中删去栈t中的栈顶元素，后将S进行回溯操作，函数返回TRUE。若函数继续向下，删除子句集中的栈顶元素，将literal赋相反值，添加单子句，入栈，递归，判断result值，若为FALSE，将变元真值回溯且S->literal\_bool[abs(literal\_v)] = 0。释放空间（literal\_back\_tag）后函数返回result。

中心函数：DPLL\_rec(cnf\_set \* S)，change\_back(cnf\_set \*S, stack\_i \* t)

图3.7为函数流程图：

图3.7 DPLL流程图

（3）数独模块（sudoku）

模块作用：构建数独格局，转化为cnf文件，恢复数独格局

核心函数：createSudoku(int difficult)，SudokuToCNF(Sudoku \*s)，\*CNFToSudoku(cnf\_set \*S)

函数实现方式：

createSudoku(int difficult)：创建数独结构Sudoku，标志flag为0，利用常规回溯算法生成随机完整的数独存储在sudoku的成员sdk中，将其随机挖掉3个空后利用DFS求解法求解数独解的个数c。如果解的个数大于1时（即解不唯一）,即c>1时，将格子[x][y]赋值为t，flag自增，当flag大于传入参数difficult时break；解的个数为1时，随机选择一个不为空的格子[x][y]赋值为0，用t记录其原来的值。重复上述过程直到break。函数返回s。（以上恢复的过程是为了在多挖空的同时保证该数独格局具有唯一解或近似唯一解）

SudokuToCNF(Sudoku \*s)：将数独需要满足的变元限制条件写入到cnf文件中，具体包括：共729个变元；大部分数已经确定；每一行一定存在1-9&&每一行不存在相同的数；每一列一定存在1-9&&每一列不存在相同的数；每一个九宫格一定存在1-9&&每一个九宫格不存在相同的数

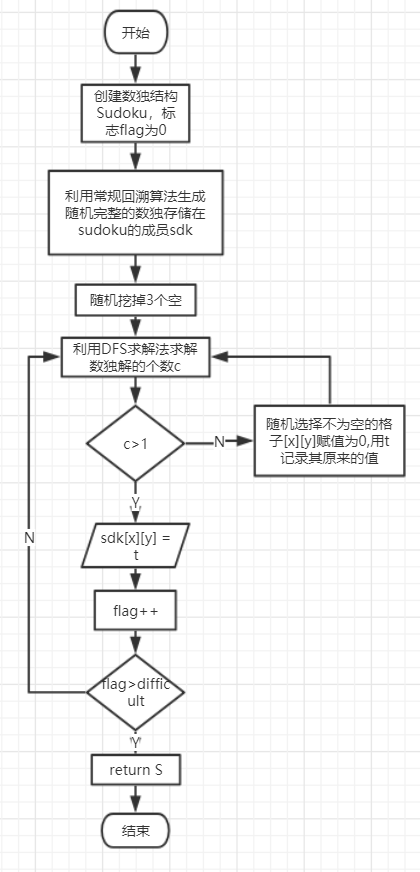
\*CNFToSudoku(cnf\_set \*S)：遍历729一个变元，若满足(S->literal\_bool[LOC(i, j, k)] == TRUE)，则将数组sdk[i][j]位置的值记录为k，在循环结束后输出其值即可

函数createSudoku流程图如图3.8所示：

（4）交互与显示模块（display）

模块作用：程序主界面的构建及交互

核心函数：CNFPage()，sudokuPage()

图3.8 createSudoku流程图

（5）DPLL优化方式

优化方式：优化变元选取方式

选取方式1：在每次重新挑选变元的过程中，统计当前未赋值过的变元再当前子句集中的出现次数。对于出现次数最多的变元，选区其出现次数较多的符号作为变元的真值。

选取优势：在每次单子句传播化简的过程中均能保证整个子句集最大的化简程度

选取方式2：对挑选变元的范围进行控制，首先对于变元长度为2以内的子句进行如同方式1的统计。如果整个子句集中未出现双变元子句，在进行方式1的选区变元。

选取优势：挑选在双子句中出现最多的变元，可以出现更多的单子句，从而在一次单子句传播的过程中进行多次化简，提升了效率。对于双子句出现较多的cnf文件该变元选取方式的效率更高。

# 4系统实现与测试

## 4.1系统实现

**4.1.1 系统概况:**

实现环境:Windows 10;

程序开发工具: Vs 2017;

编译器版本: 10.0.10011.16384

开发语言:c语言;

**4.1.2 系统使用的函数包支持:**

<stdio.h>

<time.h>

<malloc.h>

<string.h>

<stdlib.h>

<direct.h>

**4.1.3 数据类型定义**

#define MAX\_LITERAL 1000//最大变元数目

#define LINE 1024 //读取一行的最大字符数

#define ERROR -1//异常返回状态

#define OK 1//函数返回状态正常

#define TRUE 1

#define FALSE -1

#define MAX\_CLAUSE\_LENGTH 30

#define CLAUSE 1

#define LITERAL 2

#define SPLIT 3

#define UNSURE -2

#define LOC(i,j,x) ((i)\*81+(j)\*9+(x))

typedef int InfoType;

typedef int Status;//函数的返回状态

typedef int BOOL;

typedef struct First\_Node {//首结点的结构

int num; //记录该子句中变元的个数

struct literal\_Node \*next\_literalNode;// 指向子句中下一变元结点

struct First\_Node \*next\_FirstNode;//指向下一子句的首节点

struct First\_Node \*prev\_FirstNode;//指向上一子句的首节点

}First\_Node;

typedef struct literal\_Node {//变元节点的结构

InfoType literal\_info; //变元信息

struct literal\_Node \*next\_literalnode;//指向子句中下一变元结点

}literal\_Node;

typedef struct cnf\_set { //总的结构体

int literal\_num;//变元个数

int clause\_num;//子句个数

First\_Node \* set;//子句的集合

int literal\_bool[MAX\_LITERAL];//记录对应序号变元的真假,1为真，-1为假，0为未知

}cnf\_set;

typedef struct Sudoku {

int sdk[9][9];

Sudoku \*next;

}Sudoku;

typedef struct stack\_i {//栈结构

int flag;//压栈分类标志

literal\_Node \* N;

First\_Node \* P;

stack\_i \* next;

}stack\_i;

**4.1.4 各模块函数**

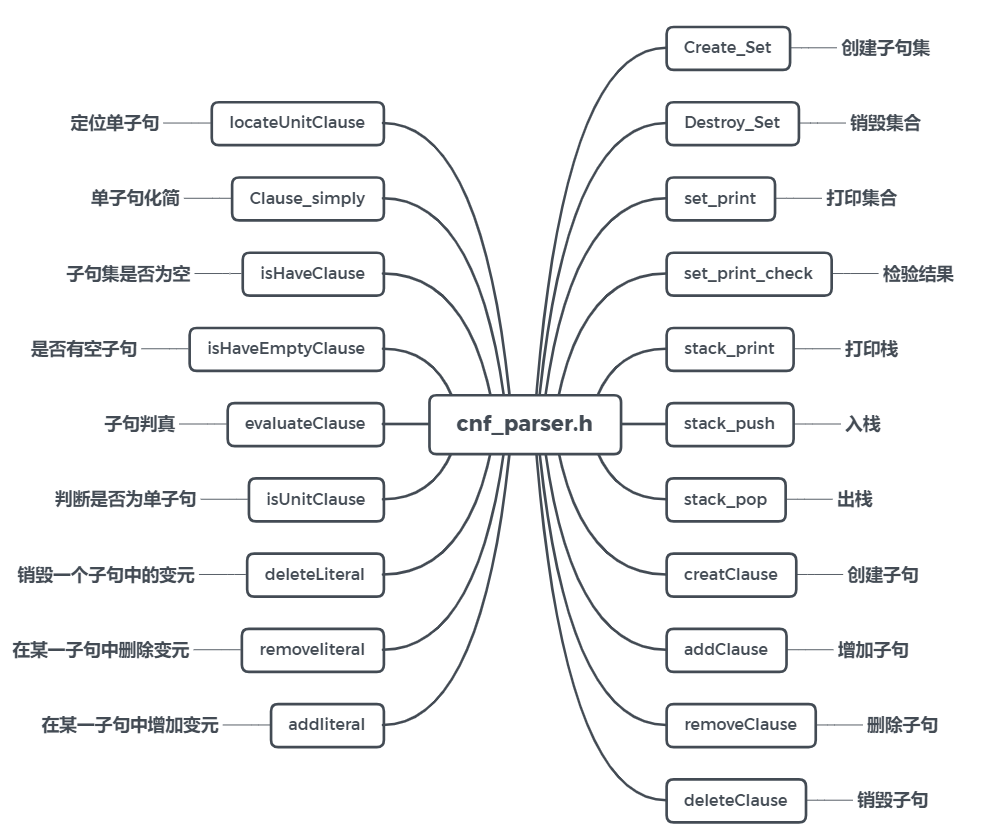
A．cnf\_parser模块包括的函数如图4.1

图4.1 cnf\_parser模块函数

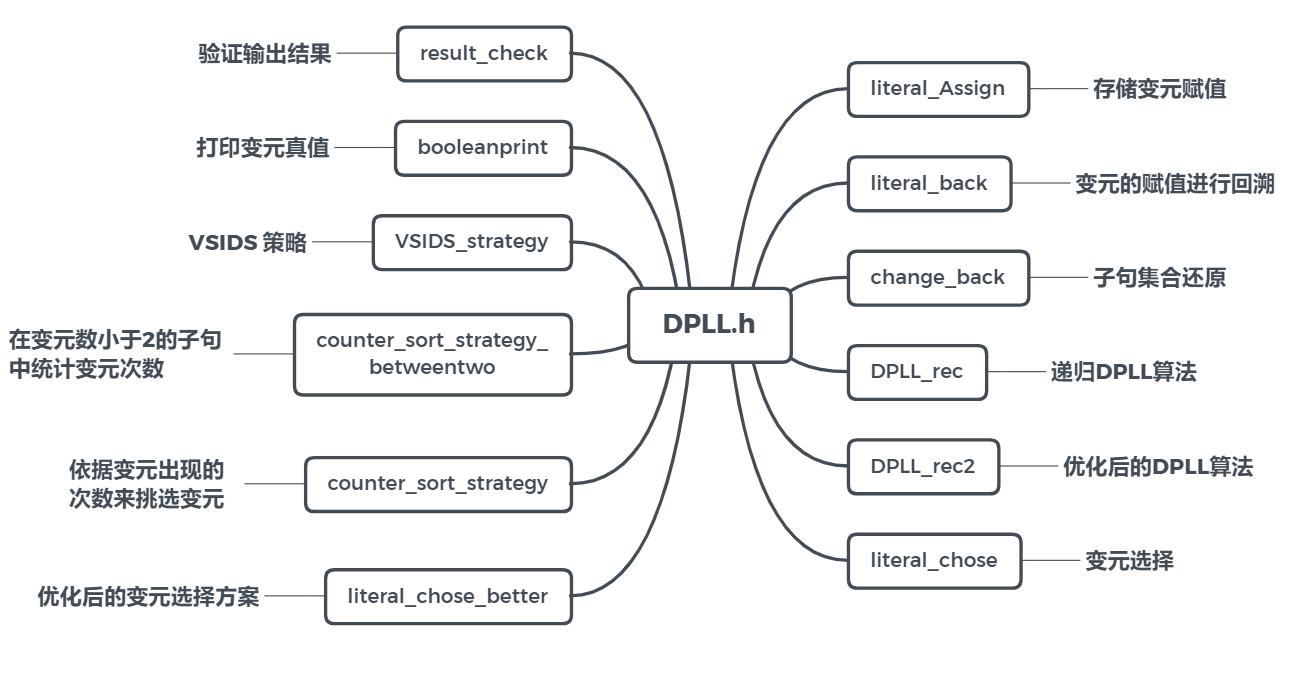
B．DPLL模块包括的函数如图4.2

图4.2 DPLL模块函数

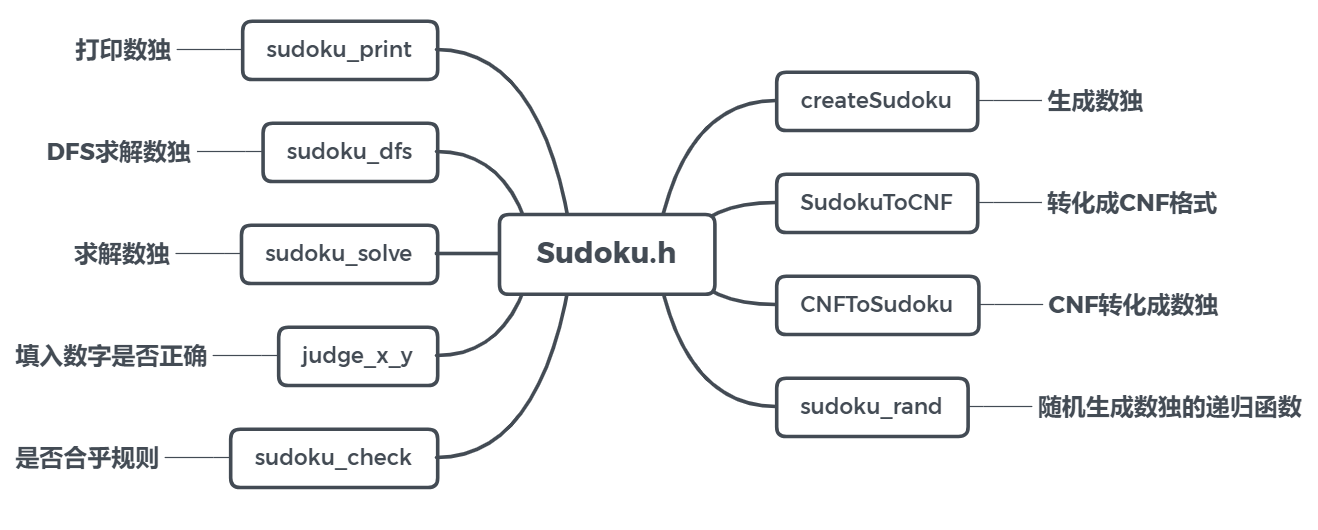
C：sudoku模块包括的函数如图4.3

图4.3 sudoku模块函数

## 4.2系统测试

**4.2.1 软件的常用测试方法：**

（1）功能测试：对软件需求规格说明书中的功能需求逐项进行的测试，以验证功能是否满足要求。

（2）性能测试：对软件需求规格说明书的功能需求逐项进行的测试，以验证功能是否满足要求。

（3）接口测试：对软件需求规格说明中的接口需求逐项进行的测试。

（4）交互界面测试：对所有人机交互界面提供的操作和显示界面进行的测试，以检验是否满足用户的需求。

（5）强度测试：强制软件运行在异常乃至发生故障的情况下（设计的极限状态到超出极限），验证程序可以运行到何种程度的测试。

（6）可靠性测试：在真实的或仿真的环境中，为做出可靠性估计而对程序进行的功能进行覆盖性测试

（7）恢复性测试：对有恢复或重置功能的软件的每一类导致恢复或重置的情况，逐一进行的测试。

（8）边界测试：对软件处在边界或端点情况下运行状态的测试。

（9）数据处理测试：对完成专门数据处理功能所进行的测试。

**4.2.2 模块测试**

A.文件读取模块（cnf\_parser）

模块功能：读取cnf文件

测试对象：函数SetCreat

测试过程：读取文件分为开头有注释和开头无注释两种类型,读取cnf文件后打印子句集的内容，与原cnf文件进行比对处理结果如表4.1

|  |  |
| --- | --- |
| Cnf文件内容 | 子句集输出内容 |
| （a有注释情况）  c  c  c  p cnf 50 15  7 19 24 0  7 -19 26 0  7 24 -26 0  7 22 -24 0  -7 22 46 0  -7 22 -46 0  8 -22 -27 0  -8 -22 -27 0  -22 27 43 0  9 27 -50 0  -9 -43 -50 0  38 -43 50 0  34 -38 50 0  -34 -38 -41 0  -34 41 -46 0 |  | |
| （b无注释情况）  p cnf 20 30  4 -18 19 0  3 18 -5 0  -5 -8 -15 0  -20 7 -16 0  10 -13 -7 0  -12 -9 17 0  17 19 5 0  -16 9 15 0  11 -5 -14 0  18 -10 13 0  -3 11 12 0  -6 -17 -8 0  -18 14 1 0  -19 -15 10 0  12 18 -19 0  -8 4 7 0  -8 -9 4 0  7 17 -15 0  12 -7 -14 0  -10 -11 8 0  2 -15 -11 0  9 6 1 0  -11 20 -17 0  9 -15 13 0  12 -7 -17 0  -18 -2 20 0  20 12 4 0  19 11 14 0  -16 18 -4 0  -1 -17 -19 0 |  |

表4.1 文件读取模块测试

如表4.1所示，经过左右对比，对于开头有注释和无注释的cnf文件均可以正确读取到子句集中，测试通过。

B．DPLL模块（DPLL）

模块功能：对cnf文件进行满足性求解

测试对象：现有的cnf文件

测试过程： 测试DPLL求解结果的正确性（利用可执行程序verify.exe验证res文件与cnf文件的满足性），测试过程如表4.2所示

可满足算例：

|  |  |  |
| --- | --- | --- |
| Cnf文件 | 求解结果 | Vertify结果 |
| sat-20.cnf |  |  |
| ais10.cnf |  |  |
| sud00009.cnf |  |  |
| 1.cnf |  |  |
| 2.cnf |  |  |
| 3.cnf |  |  |
| 4.cnf |  |  |
| 5.cnf |  |  |
| 6.cnf |  |  |
| problem1-20.cnf |  |  |
| problem2-50.cnf |  |  |
| problem3-100.cnf |  |  |
| problem6-50.cnf |  |  |
| problem8-50.cnf |  |  |
| problem11-100.cnf |  |  |
| tst\_v25\_c100.cnf |  |  |

表4.2 满足算例运行结果

不满足算例如表4.3所示：

|  |  |  |
| --- | --- | --- |
| Cnf文件 | 求解结果 | Vertify结果 |
| unsat-5cnf-30.cnf |  | Not Satisfied！ |
| tst\_v10\_c100.cnf |  | Not Satisfied！ |
| u-5cnf\_3500\_3500\_30f1.shuffled-30.cnf |  | Not Satisfied！ |
| u-problem7-50.cnf |  | Not Satisfied！ |

表4.3 不满足算例运行结果

两类算例的处理结果均符合预期结果。

C：数独模块（Sudoku）

模块功能：生成数独，求解数独

测试对象：a.数独生成

b.生成数独求解

测试过程：a.验证生成数独是否

b.验证该数独求解的正确性

测试截图如表4.4：

|  |  |  |
| --- | --- | --- |
| 生成完整数独 | 挖空后数独 | 求解数独 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

表4.4 数独生成与求解

经过DPLL过程求解后的数独于原数独具有相同的格局，验证了DPLL递归算法的正确性。

**4.2.3 DPLL优化测试：**

对于四个基准算例进行测试，结果如表4.5：

|  |  |
| --- | --- |
| Cnf算例 | 运行结果 |
| sat-20.cnf |  |
| unsat-5cnf-30.cnf |  |
| ais10.cnf |  |
| sud00009.cnf |  |

表4.5 DPLL优化结果

对于不同的算例，优化后的变元选取策略均有一定的优化效果，但效果并不明显，由于本次设计的DPLL算法采用的是递归形式，只是优化变元选取策略对整个DPLL过程的事件影响较小。

# 5总结与展望

## 5.1全文总结

本次课程设计的主要工作

（1）物理存储结构的设计，包括对于基本数据结构操作函数的编写

（2）对于DPLL算法的优化

（3）数独与cnf文件之间的相互转化

## 5.1工作展望

在今后的研究中，将围绕着如下几个方面开展工作：

（1）优化算法，对于递归DPLL来说，仅仅是改变变元选取策略对于DPLL过程的优化效果并不理想，其他优化方式如冲突处理等，可以尝试使用学习子句的方式来提升算法的性能

（2）尝试改变物理存储结构，采取更加优良的物理结构，如胀缩数据结构，进一步提升解决问题的规模

# 6体会

第一次在没有模板指导的情况下进行整个项目设计，与数据结构实验课上的几次实验相比，这次的课程设计更加锻炼个人能力。在课程的初期，是有些许摸不着头脑，关于物理结构的设计，算法选择等方面都有些束手束脚，担心方向的偏离会带来太多的无用功，经过两天的考虑以及实现的完备性并且与之前的实验内容相对接，最终选择以邻接表作为物理存储结构。在着手准备DPLL算法的过程中，常常涉及到关于子句的操作，于是在算法的半途停下，回过头整理子句的所有操作并将其组织成模块，程序的结构显得更加清晰。让我印象非常深刻的是，在改bug的过程中，有一处经过接近一天的排查，最终的原因存在于cnf文件的读取中。最初完成读取模块的时候，对于此处不够重视，验证次数不够，未排除掉所有异常，反而在后期其他模块的编写中出现了意料之外的问题，这一点节省下来的时间得不偿失，这是个值得注意的地方，必要的工作项不能缺漏。除此之外，本次设计极大得锻炼了我调试程序的能力，尤其是对于本次选择的递归DPLL算法，对我个人来说调试的难度还是比较大的，我对于＃ifdef等调试代码的运用更加得心应手了。

总的来说，第一次进行较大规模的实验且具有一定的实用性，收获颇多宝贵经验，希望在以后的编程中能更加顺利。

# 参考文献

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

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

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

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

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

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

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

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

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

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

[10] Robert Ganian and Stefan Szeider. Community Structure Inspired Algorithms for SAT and #SAT. International Conference on Theory and Applications of Satisfiability Testing(SAT 2015),223-237360

# 附录 源代码

#include <stdio.h>

#include <time.h>

#include<malloc.h>

#include<string.h>

#include<stdlib.h>

#include <direct.h>

//#define DEBUG

#define STRATEGY1

char filename[256];

#define MAX\_LITERAL 1000//最大变元数目

#define LINE 1024 //读取一行的最大字符数

#define ERROR -1//异常返回状态

#define OK 1//函数返回状态正常

#define TRUE 1

#define FALSE -1

#define MAX\_CLAUSE\_LENGTH 30

#define CLAUSE 1

#define LITERAL 2

#define SPLIT 3

#define UNSURE -2

#define LOC(i,j,x) ((i)\*81+(j)\*9+(x))

typedef int InfoType;

typedef int Status;//函数的返回状态

typedef int BOOL;

typedef struct First\_Node {//首结点的结构

int num; //记录该子句中变元的个数

struct literal\_Node \*next\_literalNode;// 指向子句中下一变元结点

struct First\_Node \*next\_FirstNode;//指向下一子句的首节点

struct First\_Node \*prev\_FirstNode;//指向上一子句的首节点

}First\_Node;

typedef struct literal\_Node {//变元节点的结构

InfoType literal\_info; //变元信息

struct literal\_Node \*next\_literalnode;//指向子句中下一变元结点

}literal\_Node;

typedef struct cnf\_set { //总的结构体

int literal\_num;//变元个数

//int literal\_num\_init;//最初变元个数

int clause\_num;//子句个数

First\_Node \* set;//子句的集合

int literal\_bool[MAX\_LITERAL];//记录对应序号变元的真假,1为真，-1为假，0为未知

}cnf\_set;

//typedef struct Clause {//子句结构

// //int isexist=0;//判断子句是否存在

// int literalnum;//变元数目

// int newclause[MAX\_CLAUSE\_LENGTH];//记录变元序号

// int clauseboolean[MAX\_CLAUSE\_LENGTH];//记录变元正负,1为正，-1为负

//}Clause;

typedef struct Sudoku {

int sdk[9][9];

Sudoku \*next;

}Sudoku;

typedef struct stack\_i {//栈结构

int flag;//压栈分类标志

literal\_Node \* N;

First\_Node \* P;

stack\_i \* next;

}stack\_i;

//cnf\_parser.h

Status Create\_Set(cnf\_set \*, int);//创建子句集

Status set\_print(cnf\_set \*);//打印集合

Status set\_print\_check(cnf\_set \*);//检验结果

Status stack\_push(stack\_i \*, int, First\_Node \*, literal\_Node \*);//入栈

First\_Node \* stack\_pop(stack\_i \*);//出栈

//Status stack\_free(stack\_i \*);//栈释放

First\_Node \* creatClause(int \*, int);//创建子句/1

Status addClause(First\_Node \*, cnf\_set \*);//增加子句/1

First\_Node \* removeClause(cnf\_set \*, First\_Node \*);//删除子句，返回其地址/1

Status deleteClause(cnf\_set \*, First\_Node \*);//销毁子句/1

Status addliteral(First\_Node \*, int);//在某一子句中增加变元/1

literal\_Node \* removeliteral(First\_Node \*, int);//在某一子句中删除变元

Status deleteLiteral(First\_Node \*, int);//销毁一个子句中的变元

BOOL isUnitClause(First\_Node \*);//判断是否为单子句/1

Status evaluateClause(First\_Node \*, int);//子句判真/1

BOOL isHaveEmptyClause(cnf\_set \*);//是否有空子句/1

BOOL isHaveClause(cnf\_set \*);//子句集是否为空/1

Status Clause\_simply(cnf\_set \*, int, stack\_i \*);//单子句化简

First\_Node \* locateUnitClause(cnf\_set \*);//定位单子句

Status stack\_print(stack\_i \*);//打印栈

Status Destroy\_Set(cnf\_set \* S);//销毁集合

//DPLL.h

Status literal\_Assign(cnf\_set \*, int);//存储变元赋值

Status literal\_back(cnf\_set \*, int \*);//变元的赋值进行回溯；

Status change\_back(cnf\_set \*, stack\_i \*); //子句集合还原

int literal\_chose(cnf\_set \*);

int literal\_chose\_better(cnf\_set \*);//优化后的变元选择方案

Status counter\_sort\_strategy(cnf\_set \*);//依据变元出现的次数来挑选变元

Status counter\_sort\_strategy\_betweentwo(cnf\_set \*);//在变元数小于2的子句中统计变元次数

int VSIDS\_strategy(cnf\_set \*);//VSIDS 策略

Status DPLL\_rec(cnf\_set \*);

Status DPLL\_rec2(cnf\_set \*);//优化后的DPLL算法

Status booleanprint(cnf\_set \*);

Status result\_check(cnf\_set \*);//验证输出结果

//Sudoku

Sudoku\* createSudoku(int);//生成数独

Status SudokuToCNF(Sudoku \*);//转化成CNF格式

Sudoku \*CNFToSudoku(cnf\_set \*);//CNF转化成数独

BOOL sudoku\_check(Sudoku \*);//是否合乎规则

BOOL judge\_x\_y(Sudoku \*, int, int);//填入数字是否正确

BOOL sudoku\_rand(Sudoku \*, int, int);//随机生成数独的递归函数

BOOL sudoku\_solve(Sudoku \*, int \*);//求解数独

BOOL sudoku\_dfs(Sudoku \*, int, int, int \*);//DFS求解数独

Status sudoku\_print(Sudoku \*);//打印数独

//diaplay

void CNFPage();

void sudokuPage();

Status path\_generate(char \*savepath, char \*cnfpath);//找到当前的操作文件

Status CnfToRes\_path(char \*, char \*);//生成res文件存储地址

Status RES\_save(char \*respath, BOOL DPLL, cnf\_set \*s, int time);//保存结果

Status CNF\_save(cnf\_set\* s, char \*cnf\_path);//保存数据

//cnf\_parser

Status literal\_print(cnf\_set \*);//变元打印

Status Create\_Set(cnf\_set \*S, int tag) {//创建集合的结构体

//if (tag == 1) {//tag == 1 表示读取cnf文件进行求解

FILE \*fp;//目标文件指针

//int is\_eof;//判断是否读取到文件尾

int is\_first\_line = 1;//判断是否读取的是第一行

int m = 1;

if (tag == 1) {

printf("输入文件名称：");

scanf("%s", filename);

}

else if (tag == 2) {

strcpy(filename, "sudoku.cnf");

}

else if (tag == 3);

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

{

printf(" 文件打开失败!\n ");

return ERROR;

}

printf("文件打开成功！\n");

char buff[10];

char ch;

//char c[10];

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

//while (strcmp(c, "c") == 0)//过滤掉所有以c开头的注释内容

//{

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

// if (strcmp(c, "p") == 0)

// break;//针对空注释的情况

// else

// {

// fscanf(fp, "%\*[^\n]%\*c");//跳行，将文件指针置于下一行的开头

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

// }

//}

while ((ch = fgetc(fp)) == 'c') {

while ((ch = fgetc(fp)) != '\n');

}

//变元个数以及子句个数读取

fscanf(fp, "%s", buff);//滤掉cnf

fscanf(fp, "%s", buff);//以字符串格式读取变元个数

S->literal\_num = atoi(buff);//字符->数字

//S->literal\_num\_init = S->literal\_num;

fscanf(fp, "%s", buff);//以字符串格式读取子句个数

S->clause\_num = atoi(buff);//字符->数字

#ifdef DEBUG

printf("变元个数为%d 子句个数为%d\n", S->literal\_num, S->clause\_num);//......DEBUG......

#endif

//子句读取及构建

int i = 0;// 记录子句中变元的个数

First\_Node \*u, \*v;//临时变量

literal\_Node \*p, \*q;//临时变量

//第一个子句

S->set = (First\_Node \*)malloc(sizeof(First\_Node));

S->set->next\_FirstNode = NULL;

S->set->prev\_FirstNode = NULL;

p = (literal\_Node \*)malloc(sizeof(literal\_Node));//读取子句的第一个变元

p->next\_literalnode = NULL;

S->set->next\_literalNode = p;

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

p->literal\_info = atoi(buff);//录入变元信息

i = 1;

#ifdef DEBUG

printf("%d ", p->literal\_info);//....DEBUG....

#endif

while (TRUE) {//同一子句的后续变元

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

if (strcmp(buff, "0") == 0)

break;//判断当前子句是否读取完成

q = (literal\_Node \*)malloc(sizeof(literal\_Node));

q->next\_literalnode = NULL;

q->literal\_info = atoi(buff);//录入变元信息

i++;

//链接

p->next\_literalnode = q;

p = q;

#ifdef DEBUG

printf("%d ", p->literal\_info);//......DEBUG......

#endif

}

S->set->num = i;//记录子句中的变元个数

//printf("子句变元个数为%d\n", S->set->num);//....DEBUG....

//后续子句

while (m < S->clause\_num/\*&&(is\_eof = fscanf(fp, "%s", buff)) != -1\*/) {//判断是否读到文件尾

v = (First\_Node \*)malloc(sizeof(First\_Node));

v->next\_FirstNode = NULL;

p = (literal\_Node \*)malloc(sizeof(literal\_Node));//读取子句的第一个变元

p->next\_literalnode = NULL;

v->next\_literalNode = p;

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

p->literal\_info = atoi(buff);//录入变元信息

i = 1;

#ifdef DEBUG

printf("%d ", p->literal\_info);//......DEBUG......

#endif

while (TRUE) {//同一子句的后续变元

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

if (strcmp(buff, "0") == 0)

break;//判断当前子句是否读取完成

q = (literal\_Node \*)malloc(sizeof(literal\_Node));

q->next\_literalnode = NULL;

q->literal\_info = atoi(buff);//录入变元信息

i++;

//变元结点链接

p->next\_literalnode = q;

p = q;

#ifdef DEBUG

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

#endif

}

v->num = i;

//首节点链接

if (is\_first\_line)//第一行与第二行

{

S->set->next\_FirstNode = v;

v->prev\_FirstNode = S->set;

u = v;

is\_first\_line = 0;

#ifdef DEBUG

printf("子句变元个数为%d\n", S->set->num);//....DEBUG....

#endif

}

else {//后续行

#ifdef DEBUG

printf("子句变元个数为%d\n", u->num);//....DEBUG....

#endif

u->next\_FirstNode = v;

v->prev\_FirstNode = u;

u = v;

}

m++;

}

//}

//else if (tag == 2) {//tag ==2 表示创建空集合后以数独结构进行转换。

// S->clause\_num = 0;

// S->literal\_num = 9 \* 9 \* 9;

// S->literal\_num\_init = 0;

// S->set = NULL;

//}

return TRUE;

}

Status set\_print(cnf\_set \* S) {

First\_Node \* p = S->set;

for (; p != NULL; p = p->next\_FirstNode)

{

printf("变元个数：%d\t", p->num);

literal\_Node \* q = p->next\_literalNode;

for (; q != NULL; q = q->next\_literalnode)

printf("%d ", q->literal\_info);

printf("\n");

}

printf("\n");

return OK;

}

Status set\_print\_check(cnf\_set \*S) {

First\_Node \* p = S->set;

for (; p != NULL; p = p->next\_FirstNode)

{

printf("变元个数：%d\t", p->num);

literal\_Node \* q = p->next\_literalNode;

for (; q != NULL; q = q->next\_literalnode)

printf("%d ", S->literal\_bool[abs(q->literal\_info)]\*q->literal\_info);

printf("\n");

}

printf("\n");

return OK;

}

Status literal\_print(cnf\_set \*S) {

int i;

for (i = 1; i <= S->literal\_num; i++) {

if ((i - 1) % 10 == 0)

printf("\n");

if (S->literal\_bool[i] > 0)

printf("%d:TRUE ", i);

else if (S->literal\_bool[i] < 0)

printf("%d:FALSE ", i);

else

printf("%d:UNSURE ", i);

}

return OK;

}

Status stack\_push(stack\_i \*t, int flag, First\_Node \*P, literal\_Node \*N) {

//t是不含数据的头节点指针

if (t == NULL) {

return FALSE;

}

stack\_i \*new\_room = (stack\_i\*)malloc(sizeof(stack\_i));

if (!new\_room)

{

printf("栈溢出！");

exit(-1);

}

new\_room->flag = flag;

switch (flag)

{

case CLAUSE://整个子句

new\_room->P = P;//保存子句的地址

new\_room->N = NULL;

break;

case LITERAL://某个变元

new\_room->P = P;//保存所在子句地址

new\_room->N = N;//保存该变元

break;

case SPLIT://添加的单子句变元

new\_room->P = P;//保存所在子句地址

new\_room->N = NULL;//保存该变元

break;

default:

break;

}

new\_room->next = t->next;

t->next = new\_room;

return OK;

}

First\_Node \* stack\_pop(stack\_i \*t) {

stack\_i \* p = t->next;

t->next = p->next;

First\_Node \* q = p->P;

free(p);

return q;

}

//Status stack\_free(stack\_i \* t) {//栈释放

// stack\_i \*p, \*q;

// p = t;

// q = t->next;

// while (p != NULL)

// {

// free(p);

// p = q;

// if (q != NULL)

// q = q->next;

// }

// return OK;

//}

First\_Node \* creatClause(int \*x, int num) {//创建子句

int i = 0;

First\_Node \* p = (First\_Node \*)malloc(sizeof(First\_Node));

literal\_Node \* u, \*v;

p->num = num;

p->next\_FirstNode = NULL;

p->prev\_FirstNode = NULL;

u = (literal\_Node \*)malloc(sizeof(literal\_Node));

u->literal\_info = x[i];

u->next\_literalnode = NULL;

p->next\_literalNode = u;

for (i = 1; i < num; i++) {

v = (literal\_Node \*)malloc(sizeof(literal\_Node));

v->literal\_info = x[i];

v->next\_literalnode = NULL;

u->next\_literalnode = v;

u = v;

}

return p;

}

Status addClause(First\_Node \* P, cnf\_set \* S) {//增加子句

if (S->set) {

S->set->prev\_FirstNode = P;

}

P->next\_FirstNode = S->set;

P->prev\_FirstNode = NULL;

S->set = P;

S->clause\_num++;

return OK;

}

Status addliteral(First\_Node \*P, int literal) {//在某一子句头部增加变元lilteral

literal\_Node \*p = (literal\_Node\*)malloc(sizeof(literal\_Node));

p->literal\_info = literal;

p->next\_literalnode = P->next\_literalNode;

P->next\_literalNode = p;

P->num++;

return OK;

}

First\_Node \* removeClause(cnf\_set \* S, First\_Node \* N) {//删除子句,返回其地址

First\_Node \*p = N->prev\_FirstNode;

if (N->next\_FirstNode)

N->next\_FirstNode->prev\_FirstNode = p;

if (p)

p->next\_FirstNode = N->next\_FirstNode;

else//N为第一个子句

S->set = N->next\_FirstNode;

S->clause\_num--;

return N;

}

Status deleteClause(cnf\_set \* S, First\_Node \* P) {//销毁子句/

{

literal\_Node \*N, \*M;

First\_Node \*p1 = P->prev\_FirstNode;

if (P->next\_FirstNode) {

P->next\_FirstNode->prev\_FirstNode = p1;

}

if (p1) {

p1->next\_FirstNode = P->next\_FirstNode;

}

else {

S->set = P->next\_FirstNode;

}

for (N = P->next\_literalNode; N; N = M) {

M = N->next\_literalnode;

free(N);

}

}

free(P);

S->clause\_num--;

return OK;

}

literal\_Node \* removeliteral(First\_Node \* P, int literal) {//在某一子句中删除变元

literal\_Node \*p, \*q = NULL;

for (p = P->next\_literalNode; p; p = p->next\_literalnode) {

if (p->literal\_info == literal) {

P->num--;

if (!q) {//针对第一个节点匹配成功的情况

P->next\_literalNode = p->next\_literalnode;

}

else {

q->next\_literalnode = p->next\_literalnode;

}

return p;

}

q = p;

}

return NULL;

}

Status deleteLiteral(First\_Node \* P, int literal) {//删除一个子句中的变元

literal\_Node \*p, \*q = NULL;

for (p = P->next\_literalNode; p; p = p->next\_literalnode) {

if (p->literal\_info == literal) {

P->num--;

if (!q) {//针对第一个节点匹配成功的情况

P->next\_literalNode = p->next\_literalnode;

}

else {

q->next\_literalnode = p->next\_literalnode;

}

free(p);

return OK;

}

q = p;

}

return ERROR;

}

BOOL isUnitClause(First\_Node \* p) {//判断是否为单子句

return (p->num == 1) ? TRUE : FALSE;

}

Status evaluateClause(First\_Node \* P, int literal)//子句判真

{

literal\_Node \*p = P->next\_literalNode;

//int i = 0;

for (; p; p = p->next\_literalnode) {

if (literal == p->literal\_info) {

return TRUE;//子句为真

}

else if (-literal == p->literal\_info) {

return UNSURE;//不确定

}

}

return UNSURE;

}

BOOL isHaveEmptyClause(cnf\_set \*S) {//是否有空子句

First\_Node \*p;

for (p = S->set; p; p = p->next\_FirstNode) {

if (!(p->num)) {

return TRUE;

}

}

return FALSE;

}

BOOL isHaveClause(cnf\_set \* S) {//子句集是否为空

return (S->set) ? TRUE : FALSE;

}

Status Clause\_simply(cnf\_set \* S, int literal, stack\_i \*head) {//单子句规则化简

//set\_print(S);

literal\_Node \* N;

First\_Node \* P = S->set;

First\_Node \* q;

while (P) {

switch (evaluateClause(P, literal))

{

case TRUE:

removeClause(S, P);

stack\_push(head, CLAUSE, P, NULL);//存子句

P = P->next\_FirstNode;

break;

case UNSURE://移除子句中选定变元的非

N = removeliteral(P, -literal);

if (N != NULL)

stack\_push(head, LITERAL, P, N);//存变元

P = P->next\_FirstNode;

break;

default:

P = P->next\_FirstNode;

break;

}

}

return OK;

}

First\_Node \* locateUnitClause(cnf\_set \* S) {//定位单子句

First\_Node \* p;

for (p = S->set; p; p = p->next\_FirstNode) {

if (isUnitClause(p) == TRUE) {

return p;

}

}

return NULL;

}

Status stack\_print(stack\_i \* t) {

stack\_i \* p;

p = t->next;

printf("栈为：\n");

while (p) {

switch (p->flag)

{

case CLAUSE:

printf("CLAUSE:%d\n", p->P->next\_literalNode->literal\_info);

break;

case LITERAL:

printf("LITERAL:%d.....%d\n", p->P->next\_literalNode->literal\_info, p->N->literal\_info);

break;

case SPLIT:

printf("SPLILT:%p\n", p->P);

break;

default:

break;

}

p = p->next;

}

return OK;

}

Status Destroy\_Set(cnf\_set \* S)

{

if (!S) {

return ERROR;

}

while (S->clause\_num != 0)

deleteClause(S, S->set);

return OK;

}

//DPLL

Status literal\_Assign(cnf\_set \* S, int literal)//存储变元赋值

{

S->literal\_bool[abs(literal)] = (literal > 0 ? 1 : -1);//记录到数组

return OK;

}

Status literal\_back(cnf\_set \*S, int \* tag) {//变元的赋值回溯

int i;

for (i = 1; i <= S->literal\_num; i++)

if (tag[i] == UNSURE)//回溯

S->literal\_bool[i] = 0;

return OK;

}

Status change\_back(cnf\_set \*S, stack\_i \* t) {//子句还原

stack\_i \* p = t->next;

while (p) {

switch (p->flag) {

case CLAUSE:

addClause(p->P, S);

break;

case LITERAL:

addliteral(p->P, p->N->literal\_info);

break;

case SPLIT:

break;

default:

break;

p = t->next;

}

t->next = t->next->next;

free(p);

p = t->next;

}

return OK;

}

Status DPLL\_rec(cnf\_set \* S) {

First\_Node \* p;

stack\_i t;

int literal\_v;

//static int literal\_back\_tag[MAX\_LITERAL];

int \*literal\_back\_tag = (int\*)malloc(sizeof(int)\*(S->literal\_num + 1));//存储子句回溯信息

if (literal\_back\_tag == NULL)exit(-1);

int i;

for (i = 0; i <= S->literal\_num; i++)

literal\_back\_tag[i] = 0;

t.next = NULL;

#ifdef DEBUG1

set\_print(S);

}

#endif // DEBUG

while ((p = locateUnitClause(S)) != NULL) {

literal\_Assign(S, p->next\_literalNode->literal\_info);//保存变元,这里的变元得想办法赋值回去

literal\_back\_tag[abs(p->next\_literalNode->literal\_info)] = UNSURE;

//printf("\n子句集中单子句为:%d", p->next\_literalNode->literal\_info);

Clause\_simply(S, p->next\_literalNode->literal\_info, &t);//化简子句

//printf("\n化简后为：\n");

//set\_print(S);

#ifdef DEBUG1

//stack\_print(&t);

//booleanprint(S);

#endif // DEBUG

if (isHaveClause(S) == FALSE) {//没有子句

//set\_print(S);

change\_back(S, &t);

//set\_print(S);

return TRUE;

}

else if (isHaveEmptyClause(S) == TRUE) {//有空子句

//stack\_print(&t);

//printf("回溯前:\n");

//set\_print(S);

change\_back(S, &t);

//printf("回溯后的子句集合为：\n");

//set\_print(S);

literal\_back(S, literal\_back\_tag);//变元回溯

return FALSE;

}

}//While

//printf("\n");

//booleanprint(S);

literal\_v = literal\_chose(S);//选择变元

#ifdef DEBUG1

#endif // DEBUG1

//printf("选取变元：%d\n", literal\_v);

literal\_Assign(S, literal\_v);//保存变元

p = creatClause(&literal\_v, 1);//创建子句

addClause(p, S);//添加子句

stack\_push(&t, SPLIT, S->set, NULL);//把添加的子句压入栈，在回溯时需要销毁掉

//set\_print(S);

//stack\_print(&t);

#ifdef DEBUG1

//printf("选择变元为：%d",literal\_v);

//printf("\n添加变元后为：\n");

//set\_print(S);

#endif // DEBUG

BOOL result = DPLL\_rec(S);//递归

if (result == TRUE) {//如果是真

deleteClause(S, stack\_pop(&t));

change\_back(S, &t);

//set\_print(S);

//printf("恢复后：\n");

//set\_print(S);

return TRUE;

}

S->literal\_bool[abs(literal\_v)] = 0;

//booleanprint(S);

#ifdef DEBUG1

//printf("\n栈：\n");

//stack\_print(&t);

#endif // DEBUG

literal\_v = -literal\_v;

//printf("\t选取新变元（反）：%d\n", literal\_v);

literal\_Assign(S, literal\_v);//保存变元

p = creatClause(&literal\_v, 1);

deleteClause(S, stack\_pop(&t));

addClause(p, S);

//printf("旧的变元销毁后:\n");

//set\_print(S);

stack\_push(&t, SPLIT, S->set, NULL);

First\_Node \* temp = S->set;

//set\_print(S);

result = DPLL\_rec(S);//递归

change\_back(S, &t);

//set\_print(S);

deleteClause(S, temp);

//set\_print(S);

//free(temp);

/\*printf("回溯后的子句集：\n");

set\_print(S);\*/

if (result == FALSE) {

literal\_back(S, literal\_back\_tag);

S->literal\_bool[abs(literal\_v)] = 0;

}

free(literal\_back\_tag);

return result;

}

Status DPLL\_rec2(cnf\_set \* S) {//优化后的DPLL算法

{

First\_Node \* p;

stack\_i t;

int literal\_v;

int \*literal\_back\_tag = (int\*)malloc(sizeof(int)\*(S->literal\_num + 1));//存储子句回溯信息

if (literal\_back\_tag == NULL)exit(-1);

int i;

for (i = 0; i <= S->literal\_num; i++)

literal\_back\_tag[i] = 0;

t.next = NULL;

while ((p = locateUnitClause(S)) != NULL) {

literal\_Assign(S, p->next\_literalNode->literal\_info);//保存变元,这里的变元得想办法赋值回去

literal\_back\_tag[abs(p->next\_literalNode->literal\_info)] = UNSURE;

//printf("\n子句集中单子句为:%d", p->next\_literalNode->literal\_info);

Clause\_simply(S, p->next\_literalNode->literal\_info, &t);//化简子句

if (isHaveClause(S) == FALSE) {//没有子句

change\_back(S, &t);

return TRUE;

}

else if (isHaveEmptyClause(S) == TRUE) {//有空子句

change\_back(S, &t);

literal\_back(S, literal\_back\_tag);//变元回溯

return FALSE;

}

}//While

literal\_v = literal\_chose\_better(S);//选择变元

literal\_Assign(S, literal\_v);//保存变元

p = creatClause(&literal\_v, 1);//创建子句

addClause(p, S);//添加子句

stack\_push(&t, SPLIT, S->set, NULL);//把添加的子句压入栈，在回溯时需要销毁掉

BOOL result = DPLL\_rec(S);//递归

if (result == TRUE) {//如果是真

deleteClause(S, stack\_pop(&t));

change\_back(S, &t);

return TRUE;

}

S->literal\_bool[abs(literal\_v)] = 0;

literal\_v = -literal\_v;

literal\_Assign(S, literal\_v);//保存变元

p = creatClause(&literal\_v, 1);

deleteClause(S, stack\_pop(&t));

addClause(p, S);

stack\_push(&t, SPLIT, S->set, NULL);

First\_Node \* temp = S->set;

result = DPLL\_rec(S);//递归

change\_back(S, &t);

deleteClause(S, temp);

if (result == FALSE) {

literal\_back(S, literal\_back\_tag);

S->literal\_bool[abs(literal\_v)] = 0;

}

free(literal\_back\_tag);

return result;

}

}

Status booleanprint(cnf\_set \* S) {

First\_Node \*p = S->set;

int i;

for (i = 1; i <= S->literal\_num; i++)

printf("%d:%d ", i, S->literal\_bool[i]);

return OK;

}

Status counter\_sort\_strategy(cnf\_set \* S) {

{

int i;

int max = 0;

int index = 0;

First\_Node \* Cp;

literal\_Node \*Lp;

int \*visitedplus = (int\*)malloc((S->literal\_num + 1) \* sizeof(int));

if (visitedplus == NULL)exit(-1);

int \*visitedminus = (int\*)malloc((S->literal\_num + 1) \* sizeof(int));

if (visitedminus == NULL)exit(-1);

for (i = 1; i <= S->literal\_num; i++) {

visitedplus[i] = visitedminus[i] = 0;

}

for (Cp = S->set; Cp; Cp = Cp->next\_FirstNode) {

for (Lp = Cp->next\_literalNode; Lp; Lp = Lp->next\_literalnode) {

if (Lp->literal\_info > 0) {

visitedplus[Lp->literal\_info]++;

}

else {

visitedminus[0 - Lp->literal\_info]++;

}

}

}

for (i = 1; i <= S->literal\_num; i++)

if (max < visitedplus[i] + visitedminus[i] && S->literal\_bool[i] == 0) {

max = visitedplus[i] + visitedminus[i];

index = i;

}

index = (visitedplus[index] < visitedminus[index]) ? -index : index;

free(visitedplus);

free(visitedminus);

return index;

}

}

Status counter\_sort\_strategy\_betweentwo(cnf\_set \* S) {

int i;

int flag = 0;//记录是否存在双变元子句，若不存在则返回0

int max = 0;

int index = 0;

First\_Node \* Cp;

literal\_Node \*Lp;

for (Cp = S->set; Cp; Cp = Cp->next\_FirstNode)

{

if (Cp->num == 2)

{

flag = 1;

break;

}

}

if (flag == 0)

return 0;

int \*visitedplus = (int\*)malloc((S->literal\_num + 1) \* sizeof(int));

if (visitedplus == NULL)exit(-1);

int \*visitedminus = (int\*)malloc((S->literal\_num + 1) \* sizeof(int));

if (visitedminus == NULL)exit(-1);

for (i = 1; i <= S->literal\_num; i++) {

visitedplus[i] = visitedminus[i] = 0;

}

for (Cp = S->set; Cp&&Cp->num <= 2; Cp = Cp->next\_FirstNode) {//改动地方

for (Lp = Cp->next\_literalNode; Lp; Lp = Lp->next\_literalnode) {

if (Lp->literal\_info > 0) {

visitedplus[Lp->literal\_info]++;

}

else {

visitedminus[0 - Lp->literal\_info]++;

}

}

}

for (i = 1; i <= S->literal\_num; i++)

if (max < visitedplus[i] + visitedminus[i] && S->literal\_bool[i] == 0) {

max = visitedplus[i] + visitedminus[i];

index = i;

}

index = (visitedplus[index] < visitedminus[index]) ? -index : index;

free(visitedplus);

free(visitedminus);

return index;

}

int VSIDS\_strategy(cnf\_set \*S)//VIDS 策略

{

int i;

static int count = 0;

int max = 0;

int index = 0;

First\_Node \*Cp;

literal\_Node \*Lp;

int \*visited\_num = (int\*)malloc((S->literal\_num + 1) \* sizeof(int));

for (Cp = S->set; Cp; Cp = Cp->next\_FirstNode) {

for (Lp = Cp->next\_literalNode; Lp; Lp = Lp->next\_literalnode) {

if (Lp->literal\_info > 0) {

visited\_num[Lp->literal\_info]++;

}

else {

visited\_num[-Lp->literal\_info]++;

}

}

}

for (i = 1; i <= S->literal\_num; i++) {

if (max < visited\_num[i] && S->literal\_bool[i] == 0) {

max = visited\_num[i];

index = i;

}

}

count++;

if (count > 10) {

count = 0;

for (i = 1; i <= S->literal\_num; i++) {

visited\_num[i] /= 3;

}

}

index = (count % 2) ? index : -index;

return index;

}

int shortest\_strategy(cnf\_set \*S) {//在最短的变元随便选一个

if (!S) return 0;

int index\_num = 2;

First\_Node \*p;

while(TRUE)

for (p = S->set; p; p->next\_FirstNode)

if (index\_num = p->num)

{

return p->next\_literalNode->literal\_info;

}

else

index\_num++;

}

int literal\_chose(cnf\_set \* S) {

return counter\_sort\_strategy(S);

}

int literal\_chose\_better(cnf\_set \*S) {

#ifdef STRATEGY1

int result = counter\_sort\_strategy\_betweentwo(S);

if (!result)

return counter\_sort\_strategy(S);

return result;

#endif // STRATEGY1

#ifdef STRATEGY2

return VSIDS\_strategy(S);

#endif // STRATEGY2

#ifdef STRATEGY3

return shortest\_strategy(S);

#endif // STRATEGY3

}

Status result\_check(cnf\_set \*S) {//验证输出结果

//set\_print(S);

int i, b;

int flag;

First\_Node \* Cp;

literal\_Node \* Lp;

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

printf("正在检查: \n");

for (Cp = S->set, i = 1; Cp; Cp = Cp->next\_FirstNode, i++) {

printf("第%d个子句: ", i);

for (Lp = Cp->next\_literalNode, flag = 0; Lp; Lp = Lp->next\_literalnode) {

b = S->literal\_bool[abs(Lp->literal\_info)];

if ((Lp->literal\_info > 0 ? 1 : -1) == b) {

printf("√");

flag = 1;

break;

}

else if (b == 0) {

printf("%dNOTsure ", abs(Lp->literal\_info));

}

else if ((Lp->literal\_info > 0 ? 1 : -1) == -b) {

continue;

}

}

if (!flag) {

printf("x");

}

printf("\n");

}

printf("\n");

return OK;

}

//sudoku

Sudoku\* createSudoku(int difficult)

{

int i, j, t;

int c, flag = 0;

int x, y;

srand(time(0));

Sudoku \*s = (Sudoku\*)malloc(sizeof(Sudoku));

Sudoku \*ss = NULL;

s->next = NULL;

if (!s) {

return NULL;

}

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

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

s->sdk[i][j] = 0;

}

}

if (sudoku\_rand(s, 0, 0) == TRUE) {//随机生成完整数独

sudoku\_print(s);

for (c = 0; c < 3; c++) {//挖掉3个空

x = rand() % 9;

y = rand() % 9;

s->sdk[x][y] = 0;

}

while (1) {

sudoku\_solve(s, &c);//求解该数独，得到解的个数c

if (c > 1) {

s->sdk[x][y] = t;

flag++;

if (flag >= difficult) {

break;

}

for (ss = s->next; ss; ss = s->next) {

s->next = ss->next;

free(ss);

}

}

else {

do {

x = rand() % 9;

y = rand() % 9;

} while (s->sdk[x][y] == 0);

t = s->sdk[x][y];

s->sdk[x][y] = 0;

}

}

return s;

}

else {

return NULL;

}

}

BOOL sudoku\_rand(Sudoku \*s, int x, int y)

{

int c = 0;

s->sdk[x][y] = rand() % 9 + 1;

do {

do {

c++;

if (c > 9) {

s->sdk[x][y] = 0;

return FALSE;

}

s->sdk[x][y] = s->sdk[x][y] % 9 + 1;

} while (judge\_x\_y(s, x, y) != TRUE);

if (x == 8 && y == 8) {

return TRUE;

}

} while (((y == 8) ? sudoku\_rand(s, x + 1, 0) : sudoku\_rand(s, x, y + 1)) != TRUE);

return TRUE;

}

Status SudokuToCNF(Sudoku \*s)

{

int i, j, k, t;

FILE \*fp;

if ((fp = fopen("sudoku.cnf", "w")) == NULL)

{

printf("File open error\n ");

}

for (t = 0, i = 0; i < 9; i++) {

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

if (s->sdk[i][j]) t++;

}

}

//将数独变为cnf公式保存在sudoku.cnf文件中

fprintf(fp, "p cnf 729 %d\n", t + 10530);

for (i = 0; i < 9; i++) {//保存已经存在的数

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

if (k = s->sdk[i][j]) {

fprintf(fp, "%d 0\n", 81 \* i + 9 \* j + k);

}

}

}

for (i = 1; i < 10; i++) {

for (j = 1; j < 10; j++) {

for (k = 1; k < 10; k++) {

fprintf(fp, "%d ", 81 \* (i - 1) + 9 \* (j - 1) + k);

}

fprintf(fp, "0\n");

}

}

for (i = 1; i < 10; i++) {

for (j = 1; j < 10; j++) {

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

for (t = k + 1; t < 10; t++) {

fprintf(fp, "-%d ", 81 \* (i - 1) + 9 \* (j - 1) + k);

fprintf(fp, "-%d 0\n", 81 \* (i - 1) + 9 \* (j - 1) + t);

}

}

}

}

for (i = 1; i < 10; i++) {//对行处理，每一行一定存在1-9&&每一行不存在相同的数

for (k = 1; k < 10; k++) {

for (j = 1; j < 10; j++) {

fprintf(fp, "%d ", 81 \* (i - 1) + 9 \* (j - 1) + k);

}

fprintf(fp, "0\n");

}

for (k = 1; k < 10; k++) {

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

for (t = j + 1; t < 10; t++) {

fprintf(fp, "-%d ", 81 \* (i - 1) + 9 \* (j - 1) + k);

fprintf(fp, "-%d 0\n", 81 \* (i - 1) + 9 \* (t - 1) + k);

}

}

}

}

for (j = 1; j < 10; j++) {//对列处理 方式同行

for (k = 1; k < 10; k++) {

for (i = 1; i < 10; i++) {

fprintf(fp, "%d ", 81 \* (i - 1) + 9 \* (j - 1) + k);

}

fprintf(fp, "0\n");

}

for (k = 1; k < 10; k++) {

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

for (t = i + 1; t < 10; t++) {

fprintf(fp, "-%d ", 81 \* (i - 1) + 9 \* (j - 1) + k);

fprintf(fp, "-%d 0\n", 81 \* (t - 1) + 9 \* (j - 1) + k);

}

}

}

}

int m, n;

for (m = 0; m < 3; m++) {

for (n = 0; n < 3; n++) {

for (k = 1; k < 10; k++) {

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

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

fprintf(fp, "%d ", 81 \* ((i - 1) + 3 \* m) + 9 \* ((j - 1) + 3 \* n) + k);

}

}

fprintf(fp, "0\n");

}

for (k = 1; k < 10; k++) {

i = 1;

j = 1;

fprintf(fp, "-%d ", 81 \* (3 \* m + i - 1) + 9 \* (3 \* n + j - 1) + k);

fprintf(fp, "-%d 0\n", 81 \* (3 \* m + i) + 9 \* (3 \* n + j) + k);

fprintf(fp, "-%d ", 81 \* (3 \* m + i - 1) + 9 \* (3 \* n + j - 1) + k);

fprintf(fp, "-%d 0\n", 81 \* (3 \* m + i) + 9 \* (3 \* n + j + 1) + k);

fprintf(fp, "-%d ", 81 \* (3 \* m + i - 1) + 9 \* (3 \* n + j - 1) + k);

fprintf(fp, "-%d 0\n", 81 \* (3 \* m + i + 1) + 9 \* (3 \* n + j) + k);

fprintf(fp, "-%d ", 81 \* (3 \* m + i - 1) + 9 \* (3 \* n + j - 1) + k);

fprintf(fp, "-%d 0\n", 81 \* (3 \* m + i + 1) + 9 \* (3 \* n + j + 1) + k);

j = 2;

fprintf(fp, "-%d ", 81 \* (3 \* m + i - 1) + 9 \* (3 \* n + j - 1) + k);

fprintf(fp, "-%d 0\n", 81 \* (3 \* m + i) + 9 \* (3 \* n + j - 2) + k);

fprintf(fp, "-%d ", 81 \* (3 \* m + i - 1) + 9 \* (3 \* n + j - 1) + k);

fprintf(fp, "-%d 0\n", 81 \* (3 \* m + i) + 9 \* (3 \* n + j) + k);

fprintf(fp, "-%d ", 81 \* (3 \* m + i - 1) + 9 \* (3 \* n + j - 1) + k);

fprintf(fp, "-%d 0\n", 81 \* (3 \* m + i + 1) + 9 \* (3 \* n + j - 2) + k);

fprintf(fp, "-%d ", 81 \* (3 \* m + i - 1) + 9 \* (3 \* n + j - 1) + k);

fprintf(fp, "-%d 0\n", 81 \* (3 \* m + i + 1) + 9 \* (3 \* n + j) + k);

j = 3;

fprintf(fp, "-%d ", 81 \* (3 \* m + i - 1) + 9 \* (3 \* n + j - 1) + k);

fprintf(fp, "-%d 0\n", 81 \* (3 \* m + i) + 9 \* (3 \* n + j - 1 - 2) + k);

fprintf(fp, "-%d ", 81 \* (3 \* m + i - 1) + 9 \* (3 \* n + j - 1) + k);

fprintf(fp, "-%d 0\n", 81 \* (3 \* m + i) + 9 \* (3 \* n + j - 1 - 1) + k);

fprintf(fp, "-%d ", 81 \* (3 \* m + i - 1) + 9 \* (3 \* n + j - 1) + k);

fprintf(fp, "-%d 0\n", 81 \* (3 \* m + i + 1) + 9 \* (3 \* n + j - 3) + k);

fprintf(fp, "-%d ", 81 \* (3 \* m + i - 1) + 9 \* (3 \* n + j - 1) + k);

fprintf(fp, "-%d 0\n", 81 \* (3 \* m + i + 1) + 9 \* (3 \* n + j - 2) + k);

i = 2;

j = 1;

fprintf(fp, "-%d ", 81 \* (3 \* m + i - 1) + 9 \* (3 \* n + j - 1) + k);

fprintf(fp, "-%d 0\n", 81 \* (3 \* m + i) + 9 \* (3 \* n + j) + k);

fprintf(fp, "-%d ", 81 \* (3 \* m + i - 1) + 9 \* (3 \* n + j - 1) + k);

fprintf(fp, "-%d 0\n", 81 \* (3 \* m + i) + 9 \* (3 \* n + j + 1) + k);

j = 2;

fprintf(fp, "-%d ", 81 \* (3 \* m + i - 1) + 9 \* (3 \* n + j - 1) + k);

fprintf(fp, "-%d 0\n", 81 \* (3 \* m + i) + 9 \* (3 \* n + j - 2) + k);

fprintf(fp, "-%d ", 81 \* (3 \* m + i - 1) + 9 \* (3 \* n + j - 1) + k);

fprintf(fp, "-%d 0\n", 81 \* (3 \* m + i) + 9 \* (3 \* n + j) + k);

j = 3;

fprintf(fp, "-%d ", 81 \* (3 \* m + i - 1) + 9 \* (3 \* n + j - 1) + k);

fprintf(fp, "-%d 0\n", 81 \* (3 \* m + i) + 9 \* (3 \* n + j - 3) + k);

fprintf(fp, "-%d ", 81 \* (3 \* m + i - 1) + 9 \* (3 \* n + j - 1) + k);

fprintf(fp, "-%d 0\n", 81 \* (3 \* m + i) + 9 \* (3 \* n + j - 2) + k);

}

}

}

fclose(fp);

return OK;

}

Sudoku \*CNFToSudoku(cnf\_set \*S) {

Sudoku \*s = (Sudoku\*)malloc(sizeof(Sudoku));

s->next = NULL;

int i, j, k;

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

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

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

if (S->literal\_bool[LOC(i, j, k)] == TRUE) {

s->sdk[i][j] = k;

}

}

}

}

return s;

}

BOOL sudoku\_check(Sudoku \*s)

{

int i, j, k, l;

int flag = 0;

int b[9];//记录出现次数

/\*横向检查\*/

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

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

b[j] = 0;

}

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

if (!s->sdk[i][j]) {

flag = 1;

}

else if (b[s->sdk[i][j] - 1] == 0) {

b[s->sdk[i][j] - 1] = 1;//数字sdk[i][j]已出现

}

else {

return FALSE;

}

}

}

/\*纵向检查\*/

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

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

b[j] = 0;

}

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

if (!s->sdk[j][i]) {

flag = 1;

}

else if (b[s->sdk[j][i] - 1] == 0) {

b[s->sdk[j][i] - 1] = 1;

}

else {

return FALSE;

}

}

}

/\*九格检查\*/

for (i = 0; i < 9; i += 3) {

for (j = 0; j < 9; j += 3) {

for (k = 0; k < 9; k++) {

b[k] = 0;

}

for (k = i; k < i + 3; k++) {

for (l = j; l < j + 3; l++) {

if (s->sdk[k][l] == 0) {

flag = 1;

}

else if (b[s->sdk[k][l] - 1] == 0) {

b[s->sdk[k][l] - 1] = 1;

}

else {

return FALSE;

}

}

}

}

}

return (flag) ? UNSURE : TRUE;

}

BOOL judge\_x\_y(Sudoku \*s, int x, int y)

{

int i, j;

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

if (s->sdk[i][y] == s->sdk[x][y] && i != x) {

return FALSE;

}

}

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

if (s->sdk[x][i] == s->sdk[x][y] && i != y) {

return FALSE;

}

}

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

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

if (s->sdk[i][j] == s->sdk[x][y] && (i != x || j != y)) {

return FALSE;

}

}

}

return TRUE;

}//1

BOOL sudoku\_solve(Sudoku \*s, int \*solvecount)

{

\*solvecount = 0;

sudoku\_dfs(s, 0, 0, solvecount);

return (solvecount > 0) ? TRUE : FALSE;

}

BOOL sudoku\_dfs(Sudoku \*s, int x, int y, int \*solvecount)

{

int i, j, k;

Sudoku \*ss;

while (s->sdk[x][y] != 0 && x <= 8) {

y++;

if (y == 9) {

x += 1;

y = 0;

}

}

if (x > 8) {

ss = (Sudoku\*)malloc(sizeof(Sudoku));

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

for (k = 0; k < 9; k++) {

ss->sdk[j][k] = s->sdk[j][k];

}

}

ss->next = s->next;

s->next = ss;

(\*solvecount)++;

return TRUE;

}

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

s->sdk[x][y] = i;

switch (judge\_x\_y(s, x, y))

{

case FALSE:

break;

case TRUE:

if (x == 8 && y == 8) {

ss = (Sudoku\*)malloc(sizeof(Sudoku));

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

for (k = 0; k < 9; k++) {

ss->sdk[j][k] = s->sdk[j][k];

}

}

ss->next = s->next;

s->next = ss;

s->sdk[x][y] = 0;

(\*solvecount)++;

return TRUE;

}

else {

(y == 8) ? sudoku\_dfs(s, x + 1, 0, solvecount) : sudoku\_dfs(s, x, y + 1, solvecount);

}

break;

default:

break;

}

}

s->sdk[x][y] = 0;

return FALSE;

}

Status sudoku\_print(Sudoku \*s)

{

int i, j;

printf("数独为:\n");

for (j = 0; j < 19; j++) {

printf("-");

}

printf("\n");

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

printf("|");

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

printf("%d%c", s->sdk[i][j], ((j + 1) % 3 == 0) ? '|' : ' ');

}

printf("\n");

if ((i + 1) % 3 == 0) {

printf("|");

for (j = 0; j < 17; j++) {

printf("-");

}

printf("|\n");

}

}

return OK;

}

//display

void CNFPage()

{

char cnf\_path[256];

char res\_path[256];

int is\_load = 0;//记录是否已载入cnf

int is\_solved = 0;//记录是否求解

int is\_repeated = 0;//记录是否进行过优化率计算

cnf\_set \*S = NULL;//总的结构体

BOOL result = FALSE;

BOOL result2 = FALSE;

int i,t1 ,t2;

double rate = 0;

int op = 1;

while (op) {

system("cls");

printf("\n\n");

printf("---------------基于SAT的CNF程序菜单-------------\n");

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

if (is\_load)

printf("当前加载文件: %s\n", cnf\_path);

printf("1. 加载CNF文件\n");

printf("2. 求解CNF(dpll\_rec\_1)\n");

printf("3. 求解CNF(dpll\_rec\_2)\n");

printf("4. 打印CNF\n");

printf("5. 打印答案\n");

printf("6. 检查答案\n");

printf("0. 返回\n");

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

printf("请选择你的操作[0~4]: ");

scanf("%d", &op);

switch (op)

{

case 1:

if (S != NULL) {

Destroy\_Set(S);

S = NULL;

}

S = (cnf\_set\*)malloc(sizeof(cnf\_set));

for (i = 0; i <= MAX\_LITERAL; i++)

S->literal\_bool[i] = 0;

//printf("debug!");

if (Create\_Set(S, 1) == TRUE)

{

printf("CNF Load Sucessfully!\n");

\_getcwd(cnf\_path, 256);//获取工作路径

path\_generate(cnf\_path, filename);

is\_load = 1;

is\_solved = 0;

}

else

{

S = NULL;

is\_load = 0;

}

is\_repeated = 0;

getchar();

getchar();

break;

case 2:

if (S == NULL)

{

printf("CNF未加载!\n");

getchar();

getchar();

break;

}

if (is\_solved)

{

printf("已求解，请勿重复操作!\n");

getchar();

getchar();

break;

}

t1 = clock();

result = DPLL\_rec(S);

t1 = clock() - t1;

is\_solved = 1;

//duration = (double)(finish - start) / CLOCKS\_PER\_SEC;

if (result == TRUE)

{

printf("Satisfied！");

//literal\_print(S);

//printf("\n\nDPLL\_rec takes %f seconds\n", duration);

printf("\nDPLL\_rec1 takes %d ms\n", t1);

CnfToRes\_path(cnf\_path, res\_path);//生成cnf文件存储地址

if (RES\_save(res\_path, result, S, t1)) {

printf("RES保存成功!\n");

}

else {

printf("RES保存失败!\n");

}

}

else if (result == FALSE)

{

printf("Not Satisfied!\n");

printf("DPLL\_rec takes %d ms\n", t1);

CnfToRes\_path(cnf\_path, res\_path);//生成cnf文件存储地址

if (RES\_save(res\_path, result, S, t1)) {

printf("RES保存成功!\n");

}

else {

printf("RES保存失败!\n");

}

}

else

printf("求解失败!\n");

getchar();

getchar();

break;

case 3:

if (is\_repeated)

{

printf("已完成计算，请勿重复操作！\n");

getchar();

getchar();

break;

}

if (!is\_solved)

{

printf("请优先使用dpll\_rec\_1完成计算！\n");

getchar();

getchar();

break;

}

else

{

Destroy\_Set(S);

S = NULL;

}

S = (cnf\_set\*)malloc(sizeof(cnf\_set));

for (i = 0; i <= MAX\_LITERAL; i++)

S->literal\_bool[i] = 0;

if (Create\_Set(S, 3) == TRUE);

is\_repeated = 1;

t2 = clock();

result2 = DPLL\_rec2(S);

t2 = clock() - t2;

if (result2 == TRUE)

{

printf("Satisfied！\n");

printf("DPLL\_rec1 takes %d ms\n", t1);

printf("DPLL\_rec2 takes %d ms\n", t2);

rate = (float)(t1 - t2)/ t1;

printf("优化率为:%%%f\n", rate\*100);

}

else if (result2 == FALSE)

{

printf("Not Satisfied!\n");

printf("DPLL\_rec1 takes %d ms\n", t1);

printf("DPLL\_rec2 takes %d ms\n", t2);

rate = float(t1 - t2) / t1;

printf("优化率为:%%%f\n", rate\*100);

}

else

printf("求解失败!\n");

getchar();

getchar();

break;

case 4:

if (S == NULL)

{

printf("CNF文件未加载!\n");

getchar();

getchar();

break;

}

set\_print(S);

printf("打印完成!\n");

getchar();

getchar();

break;

case 5:

if (S == NULL)

{

printf("CNF文件未加载!\n");

getchar();

getchar();

break;

}

if (!is\_solved)

{

printf("未求解！\n");

getchar();

getchar();

break;

}

if (result == TRUE)

{

printf("Satisfied！\n");

literal\_print(S);

}

else if (result == FALSE)

printf("Not Satisfied!\n");

else

printf("求解失败!\n");

getchar();

getchar();

break;

case 6:

if (S == NULL)

{

printf("CNF文件未加载!\n");

getchar();

getchar();

break;

}

if (!is\_solved)

{

printf("未求解！\n");

getchar();

getchar();

break;

}

//result\_check(S);

set\_print\_check(S);

getchar();

getchar();

break;

case 0:

break;

}

}

if (S) {

Destroy\_Set(S);

S = NULL;

is\_load = 0;

}

}

void sudokuPage()

{

int t, i, k;

int is\_exist = 0;

int dpll;

int op = 1;

char path[256];

cnf\_set \* S = NULL;

S = (cnf\_set\*)malloc(sizeof(cnf\_set));

for (i = 0; i <= MAX\_LITERAL; i++)

S->literal\_bool[i] = 0;

Sudoku \*s = (Sudoku \*)malloc(sizeof(Sudoku));

s->next = NULL;

while (op) {

system("cls");

printf("\n\n");

printf("-------------基于SAT的CNF程序菜单---------------\n\n");

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

printf("1. 生成数独(重复操作可生成新数独)\n");

printf("2. 生成数独求解\n");

printf("0. 回退\n");

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

printf("请选择你的操作[0~2]: ");

scanf("%d", &op);

switch (op)

{

case 1:

if (!(s = createSudoku(20))) {

printf("数独生成失败！\n");

}

else {

printf("数独生成成功!\n");

is\_exist = 1;

sudoku\_print(s);

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

}

getchar();

getchar();

break;

case 2: {

printf("数独为:\n");

sudoku\_print(s);

printf("\n正在求解...\n");

t = clock();

Status result = SudokuToCNF(s);

if (result) {

Create\_Set(S, 2);

dpll = DPLL\_rec(S);

if (dpll == TRUE) {

t = clock() - t;

printf("该CNF成功解出!\n");

Sudoku \*ss = NULL;

ss = CNFToSudoku(S);

sudoku\_print(ss);

free(ss);

printf("The process takes %dms\n", t);

}

Destroy\_Set(S);

S = NULL;

}

getchar();

getchar();

break;

}

case 0:

break;

}

}

}

void main()

{

int op = 1;

while (op) {

system("cls");

printf("\n\n");

printf("---------------基于SAT的CNF程序菜单-------------\n");

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

printf("1. CNF文件\n");

printf("2. 数独游戏\n");

printf("0. 退出\n");

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

printf("请选择你的操作[0~2]: ");

scanf("%d", &op);

switch (op)

{

case 1:

CNFPage();

getchar();

getchar();

break;

case 2:

sudokuPage();

getchar();

getchar();

break;

case 0:

break;

}

}

printf("欢迎下次使用本系统！\n");

getchar();

getchar();

return;

}

Status CnfToRes\_path(char \*cnf\_path, char \*res\_path) {//生成res文件存储地址

int i = 0;

while (cnf\_path[i] != '\0')

{

res\_path[i] = cnf\_path[i];

i++;

}

res\_path[i] = '\0';

i -= 3;

res\_path[i++] = 'r';

res\_path[i++] = 'e';

res\_path[i] = 's';

return OK;

}

Status path\_generate(char \*res\_path, char \*filename) {

char temp[2];

temp[0] = '\\';

temp[1] = '\0';

strcat(res\_path, temp);

strcat(res\_path, filename);

return OK;

}

Status RES\_save(char \*res\_path, BOOL DPLL, cnf\_set \*s, int time) {

int i;

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

if (!fp) {

return ERROR;

}

if (DPLL == TRUE)

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

if (DPLL == FALSE)

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

fprintf(fp, "v");

for (i = 1; i <= s->literal\_num; i++) {

fprintf(fp, " %d", s->literal\_bool[i] \* i);

}

fprintf(fp, "\nt %d", time);

fclose(fp);

return OK;

}

Status CNF\_save(cnf\_set\* s, char \*cnf\_Path) {

FILE \*fP = fopen(cnf\_Path, "w");

First\_Node \* P;

literal\_Node \*N;

if (!fP) {

return ERROR;

}

fprintf(fP, "p cnf %d %d\n", s->literal\_num, s->clause\_num);

for (P = s->set; P; P = P->next\_FirstNode) {

for (N = P->next\_literalNode; N; N = N->next\_literalnode) {

fprintf(fP, "%d ", N->literal\_info);

}

fprintf(fP, "0\n");

}

fclose(fP);

return OK;

}