

**课程设计报告**

**题目： 基于SAT的数独游戏求解程序**

**课程名称： 综合程序设计课程设计**

**专业班级： CS1703**

**学 号： U201714607**

**姓 名： 钟子琛**

**指导教师： 袁凌**

**报告日期： 2019.3.27**

**计算机科学与技术学**

**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%)

**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

**目 录**

**任务书** I

**1 引言** 1

1.1 **课题背景与意义** 1

1.2 **国内外研究现状** 1

1.3 **课程设计的主要研究工作** 1

**2 系统需求分析与总体设计** 2

2.1 **系统需求分析** 2

2.2 **系统总体设计** 2

**3 系统详细设计** 3

3.1 **有关数据结构的定义** 3

3.2 **主要算法设计** 5

**4 系统实现与测试** 7

4.1 **系统实现** 7

4.2 **系统测试** 10

**5 总结与展望** 23

5.1 **全文总结** 23

5.2 **工作展望** 23

**参考文献** 25

**附录**  26

**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。SAT 问题看似简 单，但它却是计算机领域和人工智能领域所要研究的中心问题，被称为理论计算 机科学和数理逻辑中的第一问题，在硬件验证、人工智能、电子设计自动化、自 动化推理、组合等式检测等领域具有非常重要的理论和实践意义。

**1.2 国内外研究现状**

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。尽管命题逻辑的 可满足性问题理论研究已趋于成熟，但在 SAT 求解器被越来越多地应用到各种实 际问题领域的今天，探寻解决 SAT 问题的高效算法仍然是一个吸引人并且极具挑 战性的研究方向。

**1.3 课程设计的主要研究工作**

(1) 对 SAT 问题的研究背景、意义及现状进行了简要总结，学习了命题逻辑可 满足性问题的基本理论知识。

(2) 基于DPLL算法，研究了关于SAT问题的求解相关的知识，并自编写了一个SAT问题的求解器。

(3) 将SAT问题应用到实际问题中，本文中选择的是数独问题，并编写了运用SAT求解器来解决数独问题的程序。

**2 系统需求分析与总体设计**

**2.1 系统需求分析**

能够读取已有的SAT算例cnf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。

在此基础上，还可以使用该系统随机生成数独问题，并将数独问题转换为SAT问题，再由SAT求解器解出最后的结果。

**2.2 系统总体设计**

该系统主要功能是，读取cnf文件，输出cnf文件的解，能够求解数独，因此包含了一下几个模块。

1. 允许用户使用系统读取cnf文件。
2. 运用基于DPLL算法的SAT求解器，得到该SAT问题的解。
3. 以创建res文件的方式，输出该cnf文件的解。
4. 能够随机生成一个有解的数独。首先创建一个完整的数独，然后在该数独的 基础上挖空，从而形成一个新的数度问题。
5. 将该数独问题转化为SAT问题。
6. 运用板块1中的SAT求解器，输出创建的SAT问题的解，再将解转化为数独的解。

完整系统模块结构图如图2-1所示。

图2-1 系统模块结构图

**3 系统详细设计**

**3.1 有关数据结构的定义**

每种数据结构的具体定义如下：

Formula：

sta：标识公式是够可满足，0为不满足，1为满足，-1为待定

num\_st：表示子句数

num\_v：变元数

Statement\* root：指向第一个子句

Statement：

num\_lit：文字数

elem：指向子句中的第一个文字

next：指向下一个子句

Literal：

pos：表示文字状态 1代表正文字 0代表负文字

num：表示文字是第几号变元

next：指向下一个文字

Stack\_F：

V：储存公式中确定的变元

F：栈中储存的公式

next：下一个节点

hole：

x：行

y：列

dig\_num：可能的数字数

elem：指向第一个可能的数字

digtal：

num：数字

v\_num：数字所对应的变元编号

next：下一个数字

总结如表3-1所示：

|  |  |
| --- | --- |
| 数据结构 | 说明 |
| Formula | 公式，表示CNF文件中保存的CNF公式 |
| Statement | 子句，表示公式中的子句 |
| Literal | 表示每一个子句中的变元文字 |
| Stack\_F | 用来储存公式的栈 |
| hole | 数独中挖出来的洞 |
| digtal | 数独中每一个格子内存放的数字 |

表3-1

各个数据结构之间的关系：

Formula保存了所有的Statement，每一个Statement由若干Literal组成。Stack\_F是在关于Formula的函数中起到储存Formula的作用。hole是数独的基本单位，每一个hole中存在若干个可能填在该空格内的数字。

数据结构之间的关系如图3-1所示。



图3-1

**3.2 主要算法设计**

这部分主要描述系统中的模块实现的流程，可采用文字配合流程图的方式表示各模块的算法思想及流程。

程序主要分为两大模块：

1. SAT

该模块能够读取cnf文件生成cnf范式并求解cnf范式。

在求解cnf范式的过程中主要运用的是DPLL算法：

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：

a.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。

b.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3-2所示。



图3-2

考虑到执行效率的问题，使用循环的方法而非递归。

运用这个算法最终将cnf范式解出来。

1. Sudoku

该模块能够创建一个数独问题。

创建数独的算法：首先使用一个二维数组来储存这个数组，然后使用random函数来生成一个随机的已解数独，再通过随机挖洞的方式来创建这个数独游戏。

该模块还能够将这个数独问题转化为cnf范式，转化为cnf文件，这样我们就把数独问题转化成了SAT问题，由此我们能够使用模块一通过DPLL算法得到该SAT问题的解，最终得到这个数独的解。

大致流程图如图3-3所示：



图3-3

**4 系统实现与测试**

**4.1 系统实现**

运行环境：Windows10、16G 运行内存、Inter(R) Core(TM) i5-7300HQ

数据结构定义：

typedef struct Literal

{

int pos;//表示文字状态 1代表正文字 0代表负文字

int num;//表示文字是第几号变元

struct Literal\* next;//指向下一个文字

}Literal;

typedef struct Statement

{

int num\_lit;//文字数

struct Literal\* elem;//指向子句中的第一个文字

struct Statement\* next;//指向下一个子句

}Statement;

typedef struct Formula

{

int sta;//标识公式是够可满足，0为不满足，1为满足，-1为待定

int num\_st;//表示子句数

int num\_v;//变元数

struct Statement\* root;//指向公式中的第一个子句

}Formula;

typedef struct Stack\_F

{

int\* V;//储存已确定信息的变元

struct Formula\* S;//表示栈中存储的公式

struct Stack\_F\* next;//用于指向下一个节点

}Stack\_F;

typedef struct hole

{

int x;//标识行

int y;//标识列

int dig\_num;//标识可能的数字数

struct digtal\* elem;//指向第一个可能的数字

}hole;

typedef struct digtal

{

int num;//数字

int v\_num;//数字所对应的变元编号

struct digtal\* next;

}digtal;

所运用的函数：

**主控、交互与显示模块（display）:**

1、main()

首先创建变量time\_start和time\_end，用来记录程序运行的时间。然后让用户选择是进行cnf求解还是生成数独并求解。

若选择cnf求解，则调用SAT函数。若选择生成数独并求解，则调用Sudoku函数。

**CNF解析模块（cnfparser）以及核心DPLL模块( solver):**

2、void SAT()

该函数能够读取cnf文件生成cnf范式并求解cnf范式。

1. 让用户输入文件路径，按照路径打开对应的cnf文件，打开文件成功后调用CNF\_Reader函数，读取对应的cnf文件并由此生成cnf范式。
2. 读取文件结束后，首先对应文件中的每个变元创建一个数组Varies[i]，用来储存它们的解。之后开始计时并调用SAT\_Solver中的Solver函数，该方法能够通过传递的cnf范式将对应的解解出来并储存到Varies[i]数组中。
3. 最后，经过Check函数的检测，依据Varies[i]数组中储存的内容，生成.res文件。

3、void CNF\_Reader(FILE\* fp, Formula\* formula)

通过该函数能够读取cnf文件来生成cnf范式。

1. 若每一行开头为c，则判断该行为注释，忽略该行。
2. 若某一行开头为cnf，则说明之后为cnf公式，将之后的变量数和子句数保存到formula->num\_v以及formula->num\_st中。
3. 以Statement数据结构创建储存子句的首节点，赋值到传入参数的formula的root中，之后在子句中每读取完一个子句，就将该子句储存到Statement链表中。
4. 每读取到一个子句，创建一个Literal首节点，赋值给Statement->elem，用来储存子句中的每一个字。
5. 当读取到最后一个字后，该cnf文件读取完毕，返回到SAT函数中。

4、status Solver(Formula\* formula, int\* Varies)

该函数能够通过传入的cnf范式得到该范式的解。主要是运用DPLL算法，但是为了优化执行效率将递归换成了循环。采取的策略为：选出长度最短的子句，在子句中选择出现次数最多的变元。

5、bool Check(int\* Varies\_res, Formula\* formula)

该函数能够检测得到的cnf算例的解是否正确。在传输的参数Varies\_res中存储着该cnf算例的解，从formula中开始以此检测每一个子句，如果每一个子句中都有一个文字满足，那么说明结果正确，否则，结果错误，重新读取cnf文件，重新求解。

**数独模块,包括数独生成、归约、求解(Sudoku)：**

6、void Sudoku()

该函数主要用来处理数独游戏的问题。

1. 首先调用CreateFinalSud函数生成完全数独格局，然后再调用CreateSudoku来对完全数独格局挖空，从而生成一个数独游戏。
2. 生成数独游戏之后将它输出到控制台上。
3. 调用TransfSud函数，将这个数独游戏化为cnf范式。
4. 将生成的cnf范式输出为cnf文件，通过之前SAT函数进行求解，得到这个数独游戏的解。
5. 将数独游戏的解化为可视化的解输出到控制台上。

7、void CreateFinalSud(int(\*sud)[9])

该函数能够创建一个完整的数独格局储存到传入的数组中。

8、int CreateSudoku(int(\*sud)[9])

该函数对之前创建的完全数独格局进行挖空，从而生成一个完整的数独游戏。

9、int\* TransfSud(int(\*sud)[9], Formula\*S, hole\* holes)

该函数将生成的数独游戏问题转化为cnf范式。算法：根据每个空格内有且仅有一个数字、1~9中的数字在每行、每列和每个宫内出现且仅出现一次创建子句，通过邻接表记录每个空格中可能出现的数字。具体步骤：

1. 生成空格及其可能的数字的邻接表。
2. 空格内必须有一个数字，由此创建子句。
3. 1~9每个数字在每行最多出现一次，最少出现一次，由此创建子句。
4. 1~9每个数字在每列最多出现一次，最少出现一次，由此创建子句。
5. 1~9每个数字在每个小九宫格内最多出现一次，最少出现一次，由此创建子句。
6. 将创建的所有子句组合成cnf范式。

10、void ShowSudResult(int\* Varies, hole\* holes, int(\*sud)[9])

在由SAT\_Solver将这个数独生成的cnf范式解出来之后，得到的解储存在Varies中，该函数能够将Varies中的解反映到数独中，并将解出来的数独输出到控制台。

**--------------------------------------------------------------------**

*程序能够实现的所有功能到此结束。程序具体代码见附录*。

**4.2 系统测试**

常用软件测试方法：

1. 静态测试：指测试不运行的部分，例如测试产品说明书，对此进行检查和审阅.。[静态方法](https://baike.baidu.com/item/%E9%9D%99%E6%80%81%E6%96%B9%E6%B3%95)是指不运行被测程序本身，仅通过分析或检查源程序的文法、结构、过程、接口等来检查程序的正确性。
2. 动态测试：是指通过运行软件来检验软件的动态行为和运行结果的正确性。
3. [单元测试](https://baike.baidu.com/item/%E5%8D%95%E5%85%83%E6%B5%8B%E8%AF%95)：是最微小规模的测试;以测试某个功能或代码块。
4. [集成测试](https://baike.baidu.com/item/%E9%9B%86%E6%88%90%E6%B5%8B%E8%AF%95)：是指一个应用系统的各个部件的联合测试，以决定它们能否在一起共同工作并没有冲突。
5. [系统测试](https://baike.baidu.com/item/%E7%B3%BB%E7%BB%9F%E6%B5%8B%E8%AF%95)：是针对整个产品系统进行的测试，目的是验证系统是否满足了需求规格的定义，找出与需求规格不相符合或与之矛盾的地方。
6. [性能测试](https://baike.baidu.com/item/%E6%80%A7%E8%83%BD%E6%B5%8B%E8%AF%95)：是在交替进行负荷和[强迫测试](https://baike.baidu.com/item/%E5%BC%BA%E8%BF%AB%E6%B5%8B%E8%AF%95)时常用的术语。理想的“性能测试”(和其他类型的测试)应在需求文档或质量保证、[测试计划](https://baike.baidu.com/item/%E6%B5%8B%E8%AF%95%E8%AE%A1%E5%88%92)中定义。性能测试一般包括[负载测试](https://baike.baidu.com/item/%E8%B4%9F%E8%BD%BD%E6%B5%8B%E8%AF%95)和压力测试。
7. [随机测试](https://baike.baidu.com/item/%E9%9A%8F%E6%9C%BA%E6%B5%8B%E8%AF%95)：没有书面[测试用例](https://baike.baidu.com/item/%E6%B5%8B%E8%AF%95%E7%94%A8%E4%BE%8B)、记录期望结果、检查列表、脚本或指令的测试。主要是根据测试者的经验对软件进行功能和性能抽查。随机测试是根据测试说明书执行用例测试的重要补充手段，是保证[测试覆盖](https://baike.baidu.com/item/%E6%B5%8B%E8%AF%95%E8%A6%86%E7%9B%96)完整性的有效方式和过程。
8. ……

对于本程序，主要使用动态测试、集成测试、性能测试和随机测试。

**SAT测试方法：**

1. 让程序读取读取基准算例中的功能测试，检测程序的基础功能是否正确。
2. 基准算例中的性能测试计算运行时间，除此之外，再使用未采用优化策略时的程序来读取这些算例，通过对比，得到程序的优化率。
3. 随机抽取20个满足算例中的S规模以及M规模算例，5个不满足算例，进行测试。

**Sudoku测试方法：**

运行Sudoku模块，随机生成5组有20空格的数独，检查结果的正确性，记录运行时间。

**--------------------------------------------------------------------**

**SAT测试结果：**

1. 未优化程序采取策略：统计所有子句中每个变元的数量，选取其中数量最多的变元。测试结果如下表4-1所示：

|  |  |  |
| --- | --- | --- |
|  | 算例运行截图 | 时间/ms |
| 性能测试 | ais10.cnf | 5700 |
|  | sud00009.cnf | 351 |
| 功能测试 | sat-20.cnf | 52 |
|  | unsat-5cnf-30.cnf | 384 |
|  | 7cnf20\_90000\_90000\_7.shuffled-20.cnf | 508 |
|  | problem1-20.cnf | 1 |
|  | problem2-50.cnf | 2 |
|  | problem3-100.cnf | 182 |
| S满足算例 | problem6-50.cnf | 2 |
|  | problem8-50.cnf | 22 |
|  | problem9-100.cnf | 3 |
|  | problem11-100.cnf | 35 |
|  | tst\_v25\_c100.cnf | 1 |
|  | problem12-200.cnf | 284 |
|  | sud00001.cnf | 156 |
| M满足算例 | sud00009.cnf | 359 |
|  | sud00012.cnf | 81 |
|  | sud00079.cnf | 251 |
|  | sud00861.cnf | 107 |
|  | u-problem7-50.cnf | 198 |
|  | u-problem10-100.cnf | 296 |
| 不满足算例 | u-5cnf\_3500\_3500\_30f1.shuffled-30.cnf | 379 |
|  | tst\_v10\_c100.cnf | 1 |
|  | php-010-008.shuffled-as.sat05-1171 | 24564 |

表4-1

1. 优化程序采取策略：首先选出长度最短的子句，在子句中选择出现次数最多的变元。测试结果如下表4-2所示：

|  |  |  |
| --- | --- | --- |
|  | 算例运行截图 | 时间/ms |
| 性能测试 | ais10.cnf | 5227 |
|  | sud00009.cnf | 319 |
| 功能测试 | sat-20.cnf | 1 |
|  | unsat-5cnf-30.cnf | 342 |
|  | 7cnf20\_90000\_90000\_7.shuffled-20.cnf | 443 |
|  | problem1-20.cnf | 1 |
|  | problem2-50.cnf | 3 |
|  | problem3-100.cnf | 164 |
| S满足算例 | problem6-50.cnf | 3 |
|  | problem8-50.cnf | 19 |
|  | problem9-100.cnf | 4 |
|  | problem11-100.cnf | 31 |
|  | tst\_v25\_c100.cnf | 2 |
|  | problem12-200.cnf | 251 |
|  | sud00001.cnf | 140 |
| M满足算例 | sud00009.cnf | 314 |
|  | sud00012.cnf | 69 |
|  | sud00079.cnf | 215 |
|  | sud00861.cnf | 98 |
|  | u-problem7-50.cnf | 183 |
|  | u-problem10-100.cnf | 265 |
| 不满足算例 | u-5cnf\_3500\_3500\_30f1.shuffled-30.cnf | 345 |
|  | tst\_v10\_c100.cnf | 1 |
|  | php-010-008.shuffled-as.sat05-1171 | 22039 |

对比可见，优化率大致为5%左右

**Sudoku测试结果：**

|  |  |  |
| --- | --- | --- |
|  | 生成的数独 | 时间/ms |
|  |  | 8515 |
| 简 |  | 18072 |
| 单 |  | 10890 |
| 数 |  | 6354 |
| 独 |  | 13594 |

**--------------------------------------------------------------------**

**程序检测结束，基本满足要求。**

**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）准备工作：通过自己区看论文和查资料，了解了SAT问题的解法、学习了CNF文件的阅览方式、掌握了DPLL算法的使用以及使用SAT问题的技巧来解决数独问题。

（2）开始编写程序：首先一边整理程序需要的模块，一边构思需要实现的函数以及数据结构，然后将所有模块之间的关系确定下来。运用模块化编程，将自己构思的程序思路一块一块地实现，比如首先实现cnf文件地读取，转化为cnf范式，然后将cnf范式传入DPLL模块中进行求解，最后输出结果并生成文件。将所有模块基本编写完毕后，开始构建模块与模块之间的关系，编写程序与用户的交互界面，最终完成这项课程设计。

（3）调试阶段：在基本完成课程设计之后，程序依然存在许多问题。运用Visual Studio 2017，通过单步调试、变量监控，一步一步修复bug，最终形成了一个能基本达成课设要求，成功运行的程序。

（4）遇到的问题：在整个过程中，遇到了可许多问题，首先是在构思程序时，因为是第一次接触如此之大的C语言工程，如何管理项目文件是我必须解决的第一个问题。因此，我学习了C语言的分文件编程，对于以后编辑大规模程序有很大的帮助吧。在思考数据结构的时候也是在一边编程时一边完善。在思考算法时，因为要查询大量的英文资料和论文，阅读和理解这些文字也花费了很多时间。最后就是在调试阶段，一开始有大量的bug，一时间有很多都弄不清楚，只有在自己慢慢总结尝试才调试完成。

**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）尽量多的去尝试参加这样的大型项目。一是为了锻炼自己的编程能力、增加自己的工作经验；二是为了习惯以后与其他人一起合作的工作环境，虽然本次实验是由自己完成，但是参加工作以后要和其他人一起完成一个项目，这样每个人编程习惯不同，对于代码的可扩展性以及兼容性都要多多考虑。

（2）需要多多了解国外的前沿技术。通过这次课程设计，让我知道了国内的学习资源确实受到了很大的限制。因此，能够多多参考学校外的资料对我的学习来说十分重要。就像这次了解DPLL算法以及了解数独游戏的规约都是参考了国外的论文了解到的。

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

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

[4] 百度百科：软件测试方法

<https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E6%B5%8B%E8%AF%95%E6%96%B9%E6%B3%95/1850037?fr=aladdin#1>

**附录**

**head.h**

#pragma once

#ifndef HEAD\_H\_INCLUDED

#define HEAD\_H\_INCLUDED

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <time.h>

//一些定量

#define TURE 1

#define FALSE 0

#define CONFUSE -1

//status用于做状态变量TRUE、FALSE等的类型

typedef int status;

/\*

int pos; 表示文字状态 1代表正文字 0代表负文字

int num; 表示文字是第几号变元

struct Literal\* next; 指向下一个文字

\*/

typedef struct Literal

{

int pos;//表示文字状态 1代表正文字 0代表负文字

int num;//表示文字是第几号变元

struct Literal\* next;//指向下一个文字

}Literal;

/\*

int num\_lit; 文字数

struct Literal\* elem; 指向子句中的第一个文字

struct Statement\* next; 指向下一个子句

\*/

typedef struct Statement

{

int num\_lit;//文字数

struct Literal\* elem;//指向子句中的第一个文字

struct Statement\* next;//指向下一个子句

}Statement;

/\*

int sta; 标识公式是够可满足，0为不满足，1为满足，-1为待定

int num\_st; 表示子句数

int num\_v; 变元数

struct Statement\* root 指向公式中的第一个子句

\*/

typedef struct Formula

{

int sta;//标识公式是够可满足，0为不满足，1为满足，-1为待定

int num\_st;//表示子句数

int num\_v;//变元数

struct Statement\* root;//指向公式中的第一个子句

}Formula;

/\*

int\* V; 储存已确定信息的变元

struct Formula\* S; 表示栈中存储的公式

struct Stack\_F\* next; 用于指向下一个节点

\*/

typedef struct Stack\_F

{

int\* V;//储存已确定信息的变元

struct Formula\* S;//表示栈中存储的公式

struct Stack\_F\* next;//用于指向下一个节点

}Stack\_F;

/\*---------------用于数独格局转cnf范式的数据结构-------------\*/

//通过邻接表储存数独格局中的每个空格，空格由可能的数字组成，每个可能的数字对应于一个变元号

//hole用于储存每一个空格

typedef struct hole

{

int x;//标识行

int y;//标识列

int dig\_num;//标识可能的数字数

struct digtal\* elem;//指向第一个可能的数字

}hole;

//digtal用于储存每个空格中可能出现的数字

typedef struct digtal

{

int num;//数字

int v\_num;//数字所对应的变元编号

struct digtal\* next;

}digtal;

//原函数

void SAT(long, long);

void CNF\_Reader(FILE\*, Formula\*);

status Solver(Formula\*, int\*);

bool Check(int\* Varies\_res, Formula\* formula);

Statement\* SearchStatement(int k, Formula\* formula);

Literal\* SearchLiteral(int k, Statement\* statement);

status HaveUnitClause(Formula\*);

void RemoveClause(Literal, Formula\*);

void ClearClause(Literal, Formula\*);

status HaveEmptyClause(Formula\*);

Formula\* AddClause(Literal, Formula\*);

void DestoryFormula(Formula\*);

Literal Choose\_L(Formula\*);

Formula\* Copy\_F(Formula\*);

int\* Copy\_V(int\*, int);

void Sudoku(long time\_start, long time\_end);

void CreateFinalSud(int(\*)[9]);

int CreateSudoku(int(\*)[9]);

int\* TransfSud(int(\*)[9], Formula\*, hole\*);

void ShowSudResult(int \*, hole\*, int(\*)[9]);

void CnfToFile(Formula\*);

#endif // HEAD\_H\_INCLUDED

**FinishedProject.cpp**

#include "head.h"

#include "conio.h"

void main(void) {

int choice = 1;

while (choice != 0) {

fflush(stdin);

long time\_start = 0;//记录程序开始时间

long time\_end = 0; //记录程序结束时间

printf("1、通过文件生成cnf范式并求解\n\n");

printf("2、随机生成一个数独并求解\n\n");

printf("0、退出\n\n");

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

printf("输入相应序号:\n");

scanf("%d", &choice);

getchar();

switch (choice)

{

case 1:SAT(time\_start, time\_end); break;

case 2:Sudoku(time\_start, time\_end); break;

case 0:break;

default:printf("输入非法\n");

}

// setbuf(stdin, NULL);

system("pause");

system("cls");

}

}

**SAT.cpp**

#include "head.h"

/\*SAT问题求解：通过读取cnf文件，求出其生成的cnf范式的解\*/

void SAT(long time\_start, long time\_end) {

system("cls");

FILE\* fp;

char filename[100];

Formula formula; //保存生成的cnf范式

int SAT; //cnf可满足性标记，0为不满足，1为满足

int\* Varies = NULL; //保存所有变元信息，0为假，1为真，-1为不确定

//formula初始化

formula.sta = -1;

formula.num\_st = 0;

formula.root = NULL;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*CNF文件读取\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

printf("输入文件名：\n");

scanf("%s", filename);

getchar();

fp = fopen(filename, "r");

if (!fp) {

printf("文件打开失败\n");

return;

}

printf("文件打开成功\n");

CNF\_Reader(fp, &formula);

fclose(fp);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*CNF文件读取完毕\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*运用SAT\_Solver验证cnf范式的可满足性\*\*\*\*\*/

Varies = (int\*)malloc((formula.num\_v + 1) \* sizeof(int));

int i;

//初始化Varies

for (i = 1; i <= formula.num\_v; i++) {

Varies[i] = -1;

}

//开始计时

time\_start = clock();

SAT = Solver(&formula, Varies);

time\_end = clock();//记录程序结束时间

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

if (SAT == 1) {

if (Check(Varies, &formula) == true) {

printf("经验证，结果正确\n");

}

else {

printf("经验证，结果错误\n");

}

}

/\*输出验证结果\*/

//将文件名改为以.res结尾

int str\_length;

str\_length = strlen(filename);

char str[] = ".res";

strcpy(filename + str\_length - 4, str);

fp = fopen(filename, "w");//打开一个与算例同名的res文件

if (!fp)

{

printf("文件打开失败！\n");

}

//输出可满足性

printf("s %d\n", SAT);

//向文件中写入可满足性

char c = 's';

fprintf(fp, "%c\t", c);

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

//文件可满足时，输出一组真值，并向文件中写入一组真值

if (SAT == 1) {

c = 'v';

printf("%c ", c);

fprintf(fp, "%c\t", c);

for (i = 1; i <= formula.num\_v; i++) {

if (Varies[i]) {

printf("%d ", i);

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

}

else {

printf("-%d ", i);

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

}

}

printf("\n");

fprintf(fp, "\n");

}

//输出程序运行时间

long k;

k = (time\_end - time\_start);

c = 't';

printf("%c %ldms\n", c, k);

fprintf(fp, "%c\t", c);

fprintf(fp, "%ld\n", k);

fclose(fp);

}

bool Check(int\* Varies\_res, Formula\* formula) {

//找每一个子句

for (int i = 0; i < formula->num\_st; i++) {

//找到子句的每一个文字

for (int j = 0; j < SearchStatement(i, formula)->num\_lit; j++) {

if (SearchLiteral(j, SearchStatement(i, formula))->pos == Varies\_res[SearchLiteral(j, SearchStatement(i, formula))->num]) {

//一个文字是真的，直接下一条语句

break;

}

else {

//已经检索到最后一个文字

if (j == SearchStatement(i, formula)->num\_lit - 1) {

return false;

}

}

}

}

return true;

}

Statement\* SearchStatement(int k, Formula\* formula) {

Statement\* res = formula->root;

for (int i = 0; i <= formula->num\_st; i++) {

if (i == k) {

return res;

}

else {

res = res->next;

}

}

printf("ERROR!Beyond the range!");

return NULL;

}

Literal\* SearchLiteral(int k, Statement\* statement) {

Literal\* res = statement->elem;

for (int i = 0; i <= statement->num\_lit; i++) {

if (i == k) {

return res;

}

else {

res = res->next;

}

}

printf("ERROR!Beyond the range!");

return NULL;

}

**CNF\_Reader.cpp**

#include "head.h"

/\*通过cnf文件生成cnf范式\*/

void CNF\_Reader(FILE\* fp, Formula\* formula) {

char buff[1000];//存储注释

//将注释存到buff数组中

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

fgets(buff, 1000, fp);

}

//通过fscanf读取变元数和子句数

int i;

//读取“ cnf ”字符串

for (i = 1; i <= 5; i++) {

fgetc(fp);

}

//将cnf文件中给出的变元数和子句数写入公式中

fscanf(fp, "%d %d", &(formula->num\_v), &(formula->num\_st));

//读取换行符

fgetc(fp);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*根据文件创建公式\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Statement\* statement;//创建子句

formula->root = (Statement\*)malloc(sizeof(Statement));

statement = formula->root;

statement->next = NULL;

statement->num\_lit = 0;

for (i = 1; i <= formula->num\_st; i++) {

if (formula->root->num\_lit != 0) {

statement->next = (Statement\*)malloc(sizeof(Statement));

statement = statement->next;

statement->next = NULL;

statement->num\_lit = 0;

}

//创建第一个文字

Literal\* literal;

statement->elem = (Literal\*)malloc(sizeof(Literal));

literal = statement->elem;

literal->next = NULL;

int k;

//将文件中目前读取的文字写入k

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

//完成后续文字的读取

while (k != 0) {

if (statement->num\_lit != 0) {

literal->next = (Literal\*)malloc(sizeof(Literal));

literal = literal->next;

literal->next = NULL;

}

if (k > 0) {

literal->pos = 1;

literal->num = k;

}

else {

literal->pos = 0;

literal->num = (-k);

}

(statement->num\_lit)++;//每读取一个文字，子句的文字数加一

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

}

fgetc(fp);//读取每行子句尾的换行符

}

}

SAT\_Solver.cpp

#include "head.h"

/\*

在提供cnf范式的前提下判断公式的可满足性。

\*/

status Solver(Formula\* formula, int\* Varies)

{

Formula\* formula\_temp; //用于临时储存生成的公式

Formula\* formula\_current = formula; //用于表示当前公式

Literal literal; //用于记录分裂策略选出的文字

int\* current\_Varies = 0; //用于表示当前变元真值状况

bool token = true; //用于标记是否为第一次循环

int i;

Stack\_F\* top;

Stack\_F\* p;

top = (Stack\_F\*)malloc(sizeof(Stack\_F));

top->S = Copy\_F(formula);

top->V = Copy\_V(Varies, formula->num\_v);

top->next = NULL;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*DPLL求解\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

while ((formula\_current->sta != 1) && (top != NULL)) {

//不是第一次进行DPLL，则进行删除公式和真值关系操作

if (token == true) {

token = false;

}

else {

DestoryFormula(formula\_current);

free(current\_Varies);

}

//栈头公式出栈并作为当前公式，栈头变元真值关系出栈作为当前变元真值关系

formula\_current = top->S;

current\_Varies = top->V;

p = top;

top = top->next;

free(p);

//当存在单子句时，用单子句传播规则处理公式

while (HaveUnitClause(formula\_current)) {

//选择一个单子句，并记录文字

Literal L;

Statement\* p\_s; //用于遍历公式

p\_s = formula\_current->root;

// 找到单子句

while (p\_s->num\_lit != 1) {

p\_s = p\_s->next;

}

L = \*(p\_s->elem);

//记录此单子句所含文字的变元使单子句为真

if (L.pos == 1)

current\_Varies[L.num] = 1;

else if (L.pos == 0)

current\_Varies[L.num] = 0;

//删除所有包含L的子句

RemoveClause(L, formula\_current);

//对S1中的每个子句，如果它包含文字 非L,则从该子句中去掉这个文字

if (L.pos == 1)

L.pos = 0;

else if (L.pos == 0)

L.pos = 1;

//清除公式中L记录文字的负文字

ClearClause(L, formula\_current);

//如果它是空集，当前CNF范式满足

if (formula\_current->root == NULL) {

//当公式可满足时，复制当前变元真值关系

formula\_current->sta = 1;

for (i = 1; i <= formula->num\_v; i++) { //复制此时的真值状态

Varies[i] = current\_Varies[i];

}

}

//如果不是空集，当前CNF范式不满足

else if (HaveEmptyClause(formula\_current)) {

formula\_current->sta = 0;

}

}

//当不存在单子句时，且不知道范式是否满足，则使用分裂策略将S分为S1，S2

//策略：选出长度最短的子句，在子句中选择出现次数最多的变元

if (formula\_current->sta == -1) {

literal = Choose\_L(formula\_current);

//先向当前范式的子句集合中加入由选出的文字，组成S2

formula\_temp = AddClause(literal, formula\_current);

p = (Stack\_F\*)malloc(sizeof(Stack\_F));

p->S = formula\_temp;

p->V = Copy\_V(current\_Varies, formula->num\_v);

p->next = top;

top = p;

//再向子句集中加入选出文字的负文字，组成S1

literal.pos = 0;

formula\_temp = AddClause(literal, formula\_current);

p = (Stack\_F\*)malloc(sizeof(Stack\_F));

p->S = formula\_temp;

p->V = Copy\_V(current\_Varies, formula->num\_v);

p->next = top;

top = p;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*DPLL结束\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

if (formula\_current->sta == 1) {

return TURE;

}

else if (formula\_current->sta == 0) {

return FALSE;

}

else {

return CONFUSE;

}

}

**Solver\_Functions.cpp**

#include "head.h"

/\*------------solver中需要的函数--------------\*/

/\*拷贝当前的公式\*/

Formula\* Copy\_F(Formula\* formula) {

Formula\* S1;

S1 = (Formula\*)malloc(sizeof(Formula));

//复制当前公式中的所有信息

S1->sta = formula->sta;

S1->num\_st = formula->num\_st;

S1->num\_v = formula->num\_v;

S1->root = (Statement\*)malloc(sizeof(Statement));

S1->root->next = NULL;

//复制公式链表

Statement\* p\_s1;

Statement\* p\_s;

p\_s1 = S1->root;

p\_s = formula->root;

while (p\_s != NULL) {

p\_s1->num\_lit = p\_s->num\_lit;

p\_s1->elem = (Literal\*)malloc(sizeof(Literal));

p\_s1->elem->next = NULL;

Literal\* p\_l1;

Literal\* p\_l;

p\_l1 = p\_s1->elem;

p\_l = p\_s->elem;

while (p\_l != NULL) {

p\_l1->num = p\_l->num;

p\_l1->pos = p\_l->pos;

if (p\_l->next != NULL) {

p\_l1->next = (Literal\*)malloc(sizeof(Literal));

p\_l1 = p\_l1->next;

}

p\_l1->next = NULL;

p\_l = p\_l->next;

}

if (p\_s->next != NULL) {

p\_s1->next = (Statement\*)malloc(sizeof(Statement));

p\_s1 = p\_s1->next;

}

p\_s1->next = NULL;

p\_s = p\_s->next;

}

return S1;

}

/\*拷贝当前变元真值关系\*/

int\* Copy\_V(int\* V, int num\_v) {

int\* V1;//拷贝者

int i;

V1 = (int\*)malloc((num\_v + 1) \* sizeof(int));

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

V1[i] = V[i];

}

return V1;

}

/\*判断当前公式是否含有单子句\*/

status HaveUnitClause(Formula\* formula) {

int token = 0;//用于标识公式是否含有单子句，如有则标志为1，如没有则标志为0

Statement\* p\_s;

p\_s = formula->root;

//遍历公式检验公式中是否含有单子句

while ((token != 1) && (p\_s != NULL)) {

if (p\_s->num\_lit == 1)

token = 1;

p\_s = p\_s->next;

}

//当标志为1的时候返回有单子句，不为1时返回无单子句

if (token == 1) {

return TURE;

}

else {

return FALSE;

}

}

/\*去除含有指定文字的子句\*/

void RemoveClause(Literal lit, Formula\* formula) {

int token = 0; //用于标记当前子句是否含有特定文字,1代表含有

int token\_root = 0; //用于标记根节点是否被检验,1表示已检验

Statement\* p\_s = NULL; //用于遍历公式

Statement\* p\_sf = NULL; //用于删除子句

Literal\* p\_l = NULL; //用于遍历子句

Literal\* p\_lf = NULL; //用于删除文字

//遍历所有子句，碰到含有特定文字的子句则删除之

//先判断根子句是否含有特定文字

while ((token\_root != 1) && formula->root != NULL)

{

p\_l = formula->root->elem;

while ((token != 1) && (p\_l != NULL)) {

if ((p\_l->pos == lit.pos) && (p\_l->num == lit.num))

token = 1;

p\_l = p\_l->next;

}

if (token == 1) {

p\_sf = formula->root;

formula->root = formula->root->next;

p\_l = p\_sf->elem;

free(p\_sf);

p\_sf = NULL;

while (p\_l != NULL) {

p\_lf = p\_l->next;

free(p\_l);

p\_l = p\_lf;

}

(formula->num\_st)--;//子句数减一

}

else {

token\_root = 1;

}

token = 0;

}

if (formula->root) {

//遍历剩余公式

p\_sf = formula->root;

p\_s = p\_sf->next;

while (p\_s != NULL) {

p\_l = p\_s->elem;

while ((token != 1) && (p\_l != NULL)) {

if ((p\_l->pos == lit.pos) && (p\_l->num == lit.num))

token = 1;

p\_l = p\_l->next;

}

//删除含有特定文字的子句

if (token == 1) {

p\_s = p\_s->next;

p\_l = p\_sf->next->elem;

free(p\_sf->next);

while (p\_l != NULL) {

p\_lf = p\_l->next;

free(p\_l);

p\_l = p\_lf;

}

p\_sf->next = p\_s;

(formula->num\_st)--;//子句数减一

}

else {

p\_s = p\_s->next;

p\_sf = p\_sf->next;

}

token = 0;//重置标记

}

}

}

/\*清除子句中的特定文字\*/

void ClearClause(Literal lit, Formula\* formula) {

Statement\* p\_s = NULL; //用于遍历公式

Literal\* p\_l = NULL; //用于遍历子句

Literal\* p\_lf = NULL; //用于删除文字

int token = 0; //用于标记根文字是否被检验,1表示已检验

p\_s = formula->root;

while (p\_s != NULL) {

while ((token != 1) && (p\_s->elem != NULL)) { //先检验根文字

p\_l = p\_s->elem;

if ((p\_l->pos == lit.pos) && (p\_l->num == lit.num)) { //根文字为特定文字

p\_s->elem = p\_s->elem->next;

free(p\_l);//删除根文字

p\_l = NULL;

(p\_s->num\_lit)--;//子句文字数减一

}

else {

token = 1;//根文字已检验

}

}

if (p\_s->elem) {

p\_lf = p\_s->elem;

p\_l = p\_lf->next;

while (p\_l != NULL) { //检验其他文字

if ((p\_l->pos == lit.pos) && (p\_l->num == lit.num)) { //此文字为特定文字

p\_l = p\_l->next;

free(p\_lf->next);//删除根文字

(p\_s->num\_lit)--;//子句文字数减一

p\_lf->next = p\_l;

}

else {

p\_l = p\_l->next;

p\_lf = p\_lf->next;

}

}

}

p\_s = p\_s->next;

token = 0;//下一句根文字未检验

}

}

/\*判断当前公式是否含有空子句，有则返回TRUE\*/

status HaveEmptyClause(Formula\* formula) {

int token;

token = 0;

Statement\* p;

p = formula->root;

while ((token != 1) && (p != NULL)) {

if (p->num\_lit == 0) {

token = 1;

}

p = p->next;

}

if (token == 1)

return TURE;

else

return FALSE;

}

/\*

在公式中选择一个特定的正文字并返回

遍历公式选出长度最短的子句，在子句中选择出现次数最多的变元，返回其正文字

\*/

Literal Choose\_L(Formula\* formula) {

Literal lit; //用于返回选择变元的正文字

Statement\* shortest;//用于指向最短的子句

Literal\* p\_l; //用于遍历最短子句

Statement\* p; //用于遍历公式

int v[2]; //用于保存出现次数最多的变元及出现次数

int i;

//遍历公式找出最短的子句

p = formula->root;

shortest = p;

while (p != NULL) {

if ((p->num\_lit) < (shortest->num\_lit))

shortest = p;

p = p->next;

}

//在最短子句中找出出现次数最多的变元

int\*\* num\_v;//保存变元及其出现数目的二维数组

num\_v = (int\*\*)malloc((shortest->num\_lit) \* sizeof(int\*));

for (i = 0; i < shortest->num\_lit; i++) {

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

num\_v[i][0] = 0;

num\_v[i][1] = 0;

}

p\_l = shortest->elem;

i = 0;

while (p\_l != NULL) {

for (i = 0; (num\_v[i][0] != p\_l->num) && (num\_v[i][0] != 0) && (i < shortest->num\_lit); i++);

if (num\_v[i][0] == p\_l->num)

num\_v[i][1]++;

else if (num\_v[i][0] == 0) {

num\_v[i][0] = p\_l->num;

num\_v[i][1] = 1;

}

p\_l = p\_l->next;

}

v[0] = num\_v[0][0];

v[1] = num\_v[0][1];

for (i = 0; i < shortest->num\_lit; i++) {

if (num\_v[i][1] > v[1])

{

v[0] = num\_v[i][0];

v[1] = num\_v[i][1];

}

}

lit.pos = 1;

lit.num = v[0];

return lit;

}

/\*

将一个给定的文字加入一个公式中

输入一个文字和一个公式

返回一个新的公式(该公式的另外开辟储存空间)

\*/

Formula\* AddClause(Literal lit, Formula\* formula) {

Formula\* S1; //用于返回生成的公式

Statement\* p\_s; //用于生成一个单子句

Statement\* p; //用于遍历公式

S1 = Copy\_F(formula); //拷贝公式

//根据文字生成一个单子句

p\_s = (Statement\*)malloc(sizeof(Statement));

p\_s->elem = (Literal\*)malloc(sizeof(Literal));

p\_s->elem->num = lit.num;

p\_s->elem->pos = lit.pos;

p\_s->elem->next = NULL;

p\_s->num\_lit = 1;

p\_s->next = NULL;

//将单子句纳入公式中

p = S1->root;

while (p->next != NULL)

p = p->next;

p->next = p\_s;

S1->num\_st++;//子句数加一

return S1;

}

/\*

销毁输入指针所指向的公式所占用的内存区域

\*/

void DestoryFormula(Formula\* formula) {

Statement\* p\_s; //用于遍历公式

Statement\* p\_sf;//用于删除子句

Literal\* p\_l; //用于遍历子句

Literal\* p\_lf; //用于删除文字

//删除公式，记录根子句位置

p\_s = formula->root;

free(formula);

//外层循环遍历删除子句，内层循环遍历删除文字

while (p\_s != NULL) {

p\_sf = p\_s;

p\_s = p\_s->next;

p\_l = p\_sf->elem;//记录根文字位置，删除子句

free(p\_sf);

while (p\_l != NULL) {

p\_lf = p\_l;

p\_l = p\_l->next;

free(p\_lf);

}

}

}

**Sudoku.cpp**

#include "head.h"

/\*

生成一个数独格局，并用SAT相关知识解决问题

\*/

void Sudoku(long time\_start, long time\_end) {

int sud[9][9];

int i;

int j;

Formula formula;

int \*Varies;

Varies = NULL;

hole\* holes;

int hole\_num;

hole\_num = 0;

system("cls");

printf("\n随机生成一个数独格局并转化为SAT问题解决\n\n");

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

//创建数独数组9×9格局

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

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

sud[i][j] = 0;//初始格局为空

}

}

CreateFinalSud(sud);//生成数独最终格局

//生成完成了一个数独

hole\_num = CreateSudoku(sud);

//展示生成的数独问题格局

system("cls");

printf("\n生成的数独：\n\n");

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

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

{

printf("\t|");

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

{

if (sud[i][j])

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

else

printf(" |");

}

printf("\n");

}

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

//数独问题转化为SAT问题

holes = (hole\*)malloc(hole\_num \* sizeof(hole));

formula.num\_st = 0;

formula.num\_v = 0;

formula.root = NULL;

formula.sta = -1;

//开始计时

time\_start = clock();

Varies = TransfSud(sud, &formula, holes);

//输出cnf文件

CnfToFile(&formula);

//利用dpll算法解决数独产生的SAT问题

Solver(&formula, Varies);

//输出dpll算法求出的解

ShowSudResult(Varies, holes, sud);

//结束计时

time\_end = clock();//记录程序结束时间

long k;

k = (time\_end - time\_start);

char c = 't';

printf("%c %ldms\n", c, k);

}

**Sudoku\_Functions.cpp**

#include "head.h"

void CreateFinalSud(int(\*sud)[9]) {

int i;

int j;

int k;

int a;

//在数独格局中给定最初的位于数独格局中央的3×3格局

k = 1;

while (k <= 9) { //由1到9依次赋值

for (i = 3; i < 6; i++) {

for (j = 3; j < 6; j++) {

sud[i][j] = k++;

}

}

}

//得到初始3×3格局左右3×3位置中的数字

srand((unsigned)time(NULL));

k = rand() % 2;//通过随机数决定完全行变换产生的3×3格局的左右位置

if (k == 1) {

//初始化左边

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

sud[3][j] = sud[5][j + 3];

}

for (i = 4; i < 6; i++) {

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

sud[i][j] = sud[i - 1][j + 3];

}

}

//初始化右边

for (j = 6; j < 9; j++) {

sud[5][j] = sud[3][j - 3];

}

for (i = 3; i < 5; i++) {

for (j = 6; j < 9; j++) {

sud[i][j] = sud[i + 1][j - 3];

}

}

}

else {

//初始化左边

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

sud[5][j] = sud[3][j + 3];

}

for (i = 3; i < 5; i++) {

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

sud[i][j] = sud[i + 1][j + 3];

}

}

//初始化右边

for (j = 6; j < 9; j++) {

sud[3][j] = sud[5][j - 3];

}

for (i = 4; i < 6; i++) {

for (j = 6; j < 9; j++) {

sud[i][j] = sud[i - 1][j - 3];

}

}

}

//通过列变换得到其他3×3格局

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

k = rand() % 2;//每个列变换具有两种可能

if (k == 1) {

//初始化上边

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

sud[i][a] = sud[i + 3][a + 2];

}

for (j = a + 1; j < a + 3; j++) {

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

sud[i][j] = sud[i + 3][j - 1];

}

}

//初始化下边

for (i = 6; i < 9; i++) {

sud[i][a + 2] = sud[i - 3][a];

}

for (j = a; j < a + 2; j++) {

for (i = 6; i < 9; i++) {

sud[i][j] = sud[i - 3][j + 1];

}

}

}

else {

//初始化上边

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

sud[i][a + 2] = sud[i + 3][a];

}

for (j = a; j < a + 2; j++) {

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

sud[i][j] = sud[i + 3][j + 1];

}

}

//初始化下边

for (i = 6; i < 9; i++) {

sud[i][a] = sud[i - 3][a + 2];

}

for (j = a + 1; j < a + 3; j++) {

for (i = 6; i < 9; i++) {

sud[i][j] = sud[i - 3][j - 1];

}

}

}

}

}

/\*

creatSudProblem用于产生数独问题

\*/

int CreateSudoku(int(\*sud)[9]) {

int level;

int hole\_num;

int i;

//记录挖洞位置

int hole\_x;

int hole\_y;

//生成挖洞数

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

printf("输入数字选择难度：【1~3】\n");

printf("1、easy\n2、normal\n3、evil\n\n");

scanf("%d", &level);

while (level > 3 || level < 1)

printf("输入非法,重新输入：\n");

hole\_num = level \* 20;

//挖洞，利用随机数生成挖洞位置

srand((unsigned)time(NULL));

for (i = 1; i <= hole\_num; i++) {

hole\_x = rand() % 9;

hole\_y = rand() % 9;

while (sud[hole\_x][hole\_y] == 0) {

hole\_x = rand() % 9;

hole\_y = rand() % 9;

}

sud[hole\_x][hole\_y] = 0;

}

return hole\_num;

}

/\*

将给定的cnf范式输入到文件中

输入为给定的cnf范式，输出一个cnf公式

\*/

void CnfToFile(Formula\* formula) {

FILE\* fp;

char filename[100];

Statement\* statement;

Literal\* literal;

printf("请输入保存文件名：\n");

scanf("%s", filename);

fp = fopen(filename, "w");

char str[] = ".cnf";

strcpy(filename + strlen(filename), str);//为文件名加上后缀

if (!fp)

printf("文件打开失败!\n");

else {

time\_t t;//用于输出当前时间

struct tm\* lt;

time(&t);

lt = localtime(&t);

fprintf(fp, "c in %d/%d/%d %d:%d:%d\nc\n", lt->tm\_year + 1900, lt->tm\_mon, lt->tm\_mday, lt->tm\_hour, lt->tm\_min, lt->tm\_sec);

printf("c in %d/%d/%d %d:%d:%d\nc\n", lt->tm\_year + 1900, lt->tm\_mon, lt->tm\_mday, lt->tm\_hour, lt->tm\_min, lt->tm\_sec);

//输入变元数和子句数等信息

fprintf(fp, "p cnf %d %d\n", formula->num\_v, formula->num\_st);

printf("p cnf %d %d\n", formula->num\_v, formula->num\_st);

//输入子句部分

statement = formula->root;

while (statement) {

literal = statement->elem;

while (literal) {

if (literal->pos) {

fprintf(fp, "%d ", literal->num);

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

}

else {

fprintf(fp, "%d ", -literal->num);

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

}

literal = literal->next;

}

fprintf(fp, "0\n");

statement = statement->next;

}

printf("文件打开成功！\n");

}

fclose(fp);

}

/\*

将数独问题转化为SAT问题

算法：根据每个空格内有且仅有一个数字、1~9中的数字在每行、每列和每个宫内出现且仅出现一次创建子句，通过邻接表记录每个空格中可能出现的数字

优化思路：用9×10的数组保存每个数字可能出现的位置

\*/

int\* TransfSud(int(\*sud)[9], Formula\*S, hole\* holes) {

int\* Varies; //用于表示变元真值关系的数组

int hole\_num = 0;//用于记录空格数

digtal\* p\_d; //用于生成空格中可能出现的数字,在生成子句中用于遍历

digtal\* p\_df; //用于指向空格可能的数字的尾位置，在生成子句中用于遍历

Statement\* p\_s; //用于创建子句

Statement\* p\_sf = NULL;//用于遍历公式

Literal\* p\_l; //用于创建文字

Literal\* p\_lf = NULL; //用于遍历子句

//用作循环变量

int i;

int j;

int k;

//用于保存行列

int x;

int y;

//生成空格及其可能的数字邻接表

for (i = 0; i < 9; i++) { //遍历格局找到每一个空格

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

if (sud[i][j] == 0) {

(holes + hole\_num)->x = i;

(holes + hole\_num)->y = j;

hole\_num++;

}

}

}

for (i = 0; i < hole\_num; i++) { //找到每个空格对应的数字

holes[i].elem = NULL;

p\_df = NULL;

holes[i].dig\_num = 0;

for (j = 1; j <= 9; j++) { //遍历检查1~9中哪些数可能出现于此空格

bool found = false;//标志此数是否已出现

for (k = 0; k < 9 && (found == false); k++) { //在行中查找

if (sud[holes[i].x][k] == j)

found = true;

}

for (k = 0; k < 9 && (found == false); k++) { //在列中查找

if (sud[k][holes[i].y] == j)

found = true;

}

x = (holes[i].x) / 3;//用于记录空格所在的宫

y = (holes[i].y) / 3;

for (int a = 3 \* x; (a >= 3 \* x) && (a < 3 \* (x + 1)) && (found == false); a++) { //在宫中查找

for (int b = 3 \* y; (b >= 3 \* y) && (b < 3 \* (y + 1)) && (found == false); b++) {

if (sud[a][b] == j)

found = true;

}

}

//为空格内的数字生成变元编号

if (found == 0) {

p\_d = (digtal\*)malloc(sizeof(digtal));

p\_d->num = j; //记录该数字

p\_d->v\_num = ++(S->num\_v); //给数字一个变元编号

p\_d->next = NULL;

holes[i].dig\_num++; //空格可能出现的数字数目加一

if (holes[i].elem == NULL) {

holes[i].elem = p\_d;

p\_df = p\_d;

}

else {

p\_df->next = p\_d;

p\_df = p\_d;

}

}

}

}

//由空格内最多一个数字，最少一个数字 创建子句

for (i = 0; i < hole\_num; i++) {

p\_d = holes[i].elem;

p\_df = p\_d;

p\_s = (Statement\*)malloc(sizeof(Statement));

p\_s->elem = NULL;

p\_s->next = NULL;

S->num\_st++;

if (i == 0 && (S->root == NULL)) { //创建子句,当为第一个子句时，优先考虑公式的根子句指针

S->root = p\_s;

}

else {

p\_sf->next = p\_s;

}

p\_sf = p\_s;

while (p\_d) { //根据每个空格最少一个数字创建子句

p\_l = (Literal\*)malloc(sizeof(Literal));

p\_l->next = NULL;

p\_sf->num\_lit++;//子句文字数加一

if (p\_s->elem == NULL) {

p\_s->elem = p\_l;

}

else {

p\_lf->next = p\_l;

}

p\_lf = p\_l;

p\_lf->pos = 1;//文字设为正，以保证析取之后至少有一个为1

p\_lf->num = p\_d->v\_num;

p\_d = p\_d->next;

}

while (p\_df) { //根据每个空格最多一个数字创建子句

p\_d = p\_df->next;

while (p\_d) {

p\_s = (Statement\*)malloc(sizeof(Statement));//生成一个新子句

p\_s->next = NULL;

S->num\_st++;

p\_sf->next = p\_s;

p\_sf = p\_s;

p\_sf->num\_lit = 2;//每个子句只有两个文字以代表只有一个文字为真

p\_sf->elem = (Literal\*)malloc(sizeof(Literal));

p\_sf->elem->num = p\_df->v\_num;

p\_sf->elem->pos = 0;//两个文字只有一个为真，故都是负文字

p\_sf->elem->next = (Literal\*)malloc(sizeof(Literal));

p\_sf->elem->next->num = p\_d->v\_num;

p\_sf->elem->next->pos = 0;

p\_sf->elem->next->next = NULL;

p\_d = p\_d->next;

}

p\_df = p\_df->next;

}

}

//由每个数字在每行最多出现一次，最少出现一次 创建子句

j = 0;

for (i = 0; i < 9; i++) { //遍历每一行

x = j;//用于记录每一行的开始空格位置

for (k = 1; k <= 9; k++) { //遍历每个数字

int v[10];//用于储存在行的空格上每个数字可能出现位置的变元号

v[0] = 0;//首行用于存储数字在此行出现次数

int a;//用于遍历v数字

int b;

a = 1;

j = x;

while ((holes[j].x == i) && j < hole\_num) { //找到每个数字对应的空格出现位置保持相应变元号

p\_df = holes[j].elem;

while (p\_df) {

if (p\_df->num == k) {

v[a] = p\_df->v\_num;

v[0]++;

a++;

break;

}

p\_df = p\_df->next;

}

j++;

}

if (v[0] == 0)

continue;

p\_s = (Statement\*)malloc(sizeof(Statement));//根据一个数字在一行内最少出现一次生成公式

S->num\_st++;//子句数加一

p\_s->num\_lit = v[0];

p\_s->next = NULL;

p\_sf->next = p\_s;

p\_sf = p\_s;

for (a = 1; a <= v[0]; a++) {

p\_l = (Literal\*)malloc(sizeof(Literal));

p\_l->next = NULL;

if (a == 1) {

p\_sf->elem = p\_l;

}

else {

p\_lf->next = p\_l;

}

p\_lf = p\_l;

p\_lf->pos = 1;//保证只有数字最少出现一次，故此子句的文字全为正

p\_lf->num = v[a];

}

for (a = 1; a <= v[0]; a++) { //根据一个数字在一行内最多出现一次生成公式

for (b = a + 1; b <= v[0]; b++) {

p\_s = (Statement\*)malloc(sizeof(Statement));

S->num\_st++;//子句数加一

p\_s->num\_lit = 2;

p\_s->next = NULL;

p\_sf->next = p\_s;

p\_sf = p\_s;

p\_sf->elem = (Literal\*)malloc(sizeof(Literal));

p\_sf->elem->num = v[a];

p\_sf->elem->pos = 0;//两个文字只有一个为真，故都是负文字

p\_sf->elem->next = (Literal\*)malloc(sizeof(Literal));

p\_sf->elem->next->num = v[b];

p\_sf->elem->next->pos = 0;

p\_sf->elem->next->next = NULL;

}

}

}

}

//由每个数字在每列最多出现一次，最少出现一次 创建子句

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

for (k = 1; k <= 9; k++) { //遍历每个数字

int v[10];//用于储存在列的空格上每个数字可能出现位置的变元号

v[0] = 0;//首行用于存储数字在此列出现次数

int a;//用于遍历v数字

a = 1;

int b;

for (i = 0; i < hole\_num; i++) { //找到每个数字对应的空格出现位置保持相应变元号

if (holes[i].y == j) {

p\_df = holes[i].elem;

while (p\_df) {

if (p\_df->num == k) {

v[a] = p\_df->v\_num;

v[0]++;

a++;

break;

}

p\_df = p\_df->next;

}

}

}

if (v[0] == 0)

continue;

p\_s = (Statement\*)malloc(sizeof(Statement));//根据一个数字在一行内最少出现一次生成公式

S->num\_st++;//子句数加一

p\_s->num\_lit = v[0];

p\_s->next = NULL;

p\_sf->next = p\_s;

p\_sf = p\_s;

for (a = 1; a <= v[0]; a++) {

p\_l = (Literal\*)malloc(sizeof(Literal));

p\_l->next = NULL;

if (a == 1) {

p\_sf->elem = p\_l;

}

else {

p\_lf->next = p\_l;

}

p\_lf = p\_l;

p\_lf->pos = 1;//保证只有数字最少出现一次，故此子句的文字全为正

p\_lf->num = v[a];

}

for (a = 1; a <= v[0]; a++) { //根据一个数字在一行内最多出现一次生成公式

for (b = a + 1; b <= v[0]; b++) {

p\_s = (Statement\*)malloc(sizeof(Statement));

S->num\_st++;//子句数加一

p\_s->num\_lit = 2;

p\_s->next = NULL;

p\_sf->next = p\_s;

p\_sf = p\_s;

p\_sf->elem = (Literal\*)malloc(sizeof(Literal));

p\_sf->elem->num = v[a];

p\_sf->elem->pos = 0;//两个文字只有一个为真，故都是负文字

p\_sf->elem->next = (Literal\*)malloc(sizeof(Literal));

p\_sf->elem->next->num = v[b];

p\_sf->elem->next->pos = 0;

p\_sf->elem->next->next = NULL;

}

}

}

}

//由每个数字在每个子宫内最多出现一次，最少出现一次 创建子句

int place;//用于遍历宫

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

int ub\_x;//表示宫的上界

int ub\_y;

int lb\_x;//表示宫的下界

int lb\_y;

lb\_x = (place / 3) \* 3;

ub\_x = (place / 3 + 1) \* 3;

switch (place) { //确定当前宫对应的上下界

case 0:

case 3:

case 6:

{

lb\_y = 0;

ub\_y = 3;

break;

}

case 1:

case 4:

case 7:

{

lb\_y = 3;

ub\_y = 6;

break;

}

case 2:

case 5:

case 8:

{

lb\_y = 6;

ub\_y = 9;

break;

}

}

for (k = 1; k <= 9; k++) //遍历每个数字

{

int v[10];//用于储存在列的空格上每个数字可能出现位置的变元号

v[0] = 0;//首行用于存储数字在此列出现次数

int a;//用于辅助遍历

int b;

a = 0;

b = 1;

for (a = 0; a < hole\_num; a++) { //找到每个数字对应的空格出现位置保持相应变元号

if ((holes[a].x >= lb\_x) && (holes[a].x < ub\_x) && (holes[a].y >= lb\_y) && (holes[a].y < ub\_y)) { //当空格在当前宫内时

p\_df = holes[a].elem;

while (p\_df) { //遍历空格中可能出现的数字

if (p\_df->num == k) { //如果恰为当前数字则计入数组

v[b] = p\_df->v\_num;

v[0]++;

b++;

break;

}

p\_df = p\_df->next;

}

}

}

if (v[0] == 0)

continue;

p\_s = (Statement\*)malloc(sizeof(Statement));//根据一个数字在一行内最少出现一次生成公式

S->num\_st++;//子句数加一

p\_s->num\_lit = v[0];

p\_s->next = NULL;

p\_sf->next = p\_s;

p\_sf = p\_s;

for (a = 1; a <= v[0]; a++) {

p\_l = (Literal\*)malloc(sizeof(Literal));

p\_l->next = NULL;

if (a == 1) {

p\_sf->elem = p\_l;

}

else {

p\_lf->next = p\_l;

}

p\_lf = p\_l;

p\_lf->pos = 1;//保证只有数字最少出现一次，故此子句的文字全为正

p\_lf->num = v[a];

}

for (a = 1; a <= v[0]; a++) { //根据一个数字在一行内最多出现一次生成公式

for (b = a + 1; b <= v[0]; b++) {

p\_s = (Statement\*)malloc(sizeof(Statement));

S->num\_st++;//子句数加一

p\_s->num\_lit = 2;

p\_s->next = NULL;

p\_sf->next = p\_s;

p\_sf = p\_s;

p\_sf->elem = (Literal\*)malloc(sizeof(Literal));

p\_sf->elem->num = v[a];

p\_sf->elem->pos = 0;//两个文字只有一个为真，故都是负文字

p\_sf->elem->next = (Literal\*)malloc(sizeof(Literal));

p\_sf->elem->next->num = v[b];

p\_sf->elem->next->pos = 0;

p\_sf->elem->next->next = NULL;

}

}

}

}

//由变元数生成变元真值关系数组

Varies = (int\*)malloc((S->num\_v + 1) \* sizeof(int));

for (i = 1; i < (S->num\_v + 1); i++)

Varies[i] = -1;

return Varies;

}

/\*

将cnf变元真值关系转化为数独的解并输出

输入为一组cnf变元真值关系

输出为数独的解

算法思想：输出数独元素，当此元素为0时，找到空格数组中对应的空格，遍历查看空格中那个数字对应的变元为真则输出此数字

\*/

void ShowSudResult(int\* Varies, hole\* holes, int(\*sud)[9]) {

int i;//用于遍历数独格局

int j;

int k = 0;//用于遍历空格数组

digtal\* p;

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

printf("数独的解为：\n\n");

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

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

printf("\t|");

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

if (sud[i][j] == 0) { //当此处为空时

p = holes[k++].elem;

while (p) {

if (Varies[p->v\_num]) {

sud[i][j] = p->num;

break;

}

else

p = p->next;

}

}

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

}

printf("\n");

}

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

}