

**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： CS2004**

**学 号： U202015392**

**姓 名： 陈铭铭**

**指导教师： 袁凌**

**报告日期： 2021.9.16**

**计算机科学与技术学院**

**任务书**

**设计内容**

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

**设计要求**

要求具有如下功能：

1. **输入输出功能：**包括程序执行参数的输入，SAT算例cnf文件的读取，执行结果的输出与文件保存等。(15%)
2. **公式解析与验证：**读取cnf算例文件，解析文件，基于一定的物理结构，建立公式的内部表示；并实现对解析正确性的验证功能，即遍历内部结构逐行输出与显示每个子句，与输入算例对比可人工判断解析功能的正确性。数据结构的设计可参考文献[1-3]。(15%)
3. **DPLL过程：**基于DPLL算法框架，实现SAT算例的求解。(35%)
4. **时间性能的测量：**基于相应的时间处理函数（参考time.h），记录DPLL过程执行时间（以毫秒为单位），并作为输出信息的一部分。(5%)
5. **程序优化：**对基本DPLL的实现进行存储结构、分支变元选取策略[1-3]等某一方面进行优化设计与实现，提供较明确的性能优化率结果。优化率的计算公式为：[(t-to)/t]\*100%,其中t 为未对DPLL优化时求解基准算例的执行时间，to则为优化DPLL实现时求解同一算例的执行时间。(15%)
6. **SAT应用：**将数独游戏[5]问题转化为SAT问题[6-8]，并集成到上面的求解器进行数独游戏求解，游戏可玩，具有一定的/简单的交互性。应用问题归约为SAT问题的具体方法可参考文献[3]与[6-8]。(15%)

**参考文献**

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

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

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

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

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

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

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

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

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

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

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

**目录**

**任务书** I

设计内容 I

设计要求 I

参考文献 II

**1.引言** 1

1.1课题背景与意义 1

1.2国内外研究现状 2

1.3课程设计的主要研究工作 4

**2.系统需求分析与总体设计** 5

2.1系统需求分析 5

2.2系统总体设计 5

**3.系统详细设计** 6

3.1有关数据结构的定义 6

3.2主要算法设计 7

**4.系统实现与测试** 9

4.1系统的实现 9

4.2系统的测试 10

**5.总结与展望** 13

5.1全文总结 13

5.2工作展望 13

**6.体会** 14

**参考文献** 15

**附录** 16

**1引言**

**1.1课题背景与意义**

**1.1.1课题背景**

P与NP问题是计算机科学中尚未解决的巨大问题。攻破P=NP问题，NP完全问题的概念十分重要。NP完全问题是一个问题的集合，它们之间可以在多项式时间相互归约。任何一个NP问题都可以转成NP完全问题，形式上，NP完全问题是一个NP问题，它至少象其它NP问题一样具有代表性。依据库克.史提芬定理，任何一个NP问题都可以在多项式时间内，机械地转化成布尔可满足性问题（SAT）。布尔可满足性问题是一个NP完全问题。如果布尔可满足性问题有多项式时间算法，则它是P问题，必然导致有P=NP的结论。

命题逻辑公式又叫布尔逻辑表达式，它由逻辑变量、与运算符（/）、或运算符（/）、非运算符（'）和括号组成。如果能够找到一组变量值（真或假），使得一个逻辑公式的运算结果为“真”，那么称这个逻辑公式为可满足的。SAT问题是给出一个逻辑公式（合取范式）来确定其是否可满足的问题。这个判定问题在不同的计算机科学领域都非常重要，其中包括计算机科学理论、计算复杂度理论、算法、密码学和人工智能等方面。

**1.1.2研究意义**

SAT问题是第一个被证明的完全NP问题。可满足性问题是计算机科学领域和人工智能等领域中的重要研究对象"。但是其求解算法的时间开销和空间开销却异常惊人。对于一个含有n个命题逻辑变量的合取范式来说，如果使用穷举法来罗列所有真值指派进行求解，虽然在理论上是可行的，但算法的时间复杂度却是指数级规模为0(2")，计算机如果采用这种方式进行求解将负担不起如此大的开销。搜索空间如此庞大，使得计算机在可等待的时间里不能计算出结果，进而产生组合爆炸问题，所花费的计算时间是人们不能容忍的。S.A.Cook于1971年首次证明了布尔表达式的可满足性问题属于NP完全问题。NP完全问题排在七大数学难题之首，在计算复杂性理论中具有非常重要的地位，一方面因为它有着极大的理论价值并且非常难解，另一方面是一旦被破解以后，在诸多的工程领域里还可以得到广泛的应用。但是在P≠NP的假设条件下，SAT问题不可能有多项式时间的求解算法。SAT问题已经成为了一类问题的难度标准，所以称其为NP完全问题的种子。

**1.2国内外研究现状**

从SAT问世至今，国际上举办过很多次SAT竞赛，在这些比赛中获得优胜的求解器主要有:GRASP、zChaff、BerkMin和MiniSAT等，这些求解器几乎都是在DPLL算法1川(Davis-Putnam-Logemann-Loveland算法，简称DPLL算法)或在对该算法进行优化得到的算法的基础上实现的。DPLL算法是一种判定SAT问题的高效算法。早在1960年，Davis和Putnam就提出了最早的DPLL算法，当时称为DP算法(Davis-Putnam算法，简称DP算法)，该算法的提出有效地降低了SAT问题的复杂性和求解器的空间限制这些棘手的情况，奠定了DPLL算法在判定SAT问题时的地位。DP算法通过在给定的表达式中进行变量消解，达到了降低搜索空间大小的目的。但当Logemann和Loveland尝试实现DP算法时，发现该算法在消解时占用过多的内存空间,这在当时的条件下是受限的。于是在1962年，Logemann和Loveland等人对DP算法进行了改进，改变了变量消解的方式，形成了最初的DPLL算法叫。此时的DPLL算法采用的是分支回溯策略，即不断地选择变量进行分支赋值，当发生冲突时再进行回溯。这种DPLL算法求解效率较低，最多只能解决10变量的SAT问题，且求解范围受限，对随机生成的实例求解效果较好，对实际应用转化而来的实例求解效果不佳。由于DPLL算法提出时间较早，发展并不完善，所以在初始阶段应用该算法判定SAT问题的效果并不理想。又因为SAT问题是一类NP完全问题，这更加打击了许多研究者对DPLL算法的研究信心，从而导致了在相当长的一段时间里该算法都没有得到足够多的重视，使得它的发展缓慢,研究成果寥寥无几，在一定程度上阻碍了DPLL算法的进步。很多学者为了改善这一境况，针对DPLL算法相继提出了很多启发式搜索策略，如变量的启发式选择、尽早剪除搜索空间、用DAG像搜索空间替换树形搜索空间等。直到1996年GRASP算法的提出，系统的总结了DPLL算法的学习过程，才稍微缓解了这一境况，使得DPLL算法再次成为研究的热点。 1996年,Marques-Silva和Sakallah提出了GRASP算法,并于1999年加以完善,写成文献。该算法采用的是DLIS启发式策略选取判定变量，同时又提出了基于冲突的学习过程和非同步回溯，并将这些改进与原DPLL算法结合到一起。在实际求解问题时，该算法可以尽早剪除不满足搜索空间,极大地降低了搜索时间，提高了求解效率。1997年，H.Zhang提出了SATO算法，它是在DPLL算法的基础\_上采用了智能回溯策略、搜索重新启动策略和BCP数据结构等，并将这些策略与原有的策略完美的融合在一起，共同协作解决SAT问题。该算法的实现降低了原有的SAT求解器的求解时间，并解决了之前不能处理的一些SAT问题，有效的扩大了DPLL算法的实用范围。

上面阐述的GRASP算法、SATO算法等都是在DPLL算法的基础上，优化了一些启发式搜索策略提出的，称为二代DPLL算法。较最初的DPLL算法高效一些，但这些算法中没有引入学习冲突子句的策略，所以在判断某些SAT问题时，仍存在无法解决的SAT实例。针对这一难题，研究者们又提出了一系列新的优化技术，如:冲突子句的分析和学习子句的添加、回跳规则和学习子句的删除策略等。随着这些有效的优化策略的提出,相对应地又实现了一批更高效的SAT算法,如zChaff、BerkMin、Limmat和MiniSAT等。2001年，L.Zhang提出了zChaff算法，该算法是在改进的DPLL算法GRASP的基础上进行的优化，在GRASP算法的基础上加入了如下的启发式策略:

1)独立下降式变量和(Variable State Independent Decaying Sum，简称VSIDS)选择策略;

2)两观察文字法;

3）冲突分析和学习子句;

4)学习子句的删除策略和定时重启策略。

优化后的zChaff算法有效地提高了BCP推导的效率，找到了较为高效的学习方式，使得应用该算法解决SAT问题的效率实现了质的飞跃,奠定了其在SAT算法发展中的重要作用。使得在这之后的很多SAT算法都以它为蓝本，如BerkMin和Limmat。2005年，Een和Srensson提出了SAT算法MiniSAT,这是迄今为止各方面性能都较好的SAT求解器。该求解器是在DPLL算法的基础上，采用了冲突导向的学习方式和变量动态排序的判定策略，并结合了高效的预处理技术，如Niver预处理器。在2006年的SAT国际竞赛中，该算法夺冠。通过竞赛结果可知，在规定的资源内(如:求解时间和运行空间的限制)，MiniSAT可以解决158个SAT实例，而zChaff仅能解决147个。正是由于MiniSAT突破了以往的局限，不仅优化了算法本身，而且还结合了各种高效的预处理器，这使得它在SAT国际竞赛上-一直名列前茅，甚至是蝉联冠军。随着研究的深入，优化技术的不断改进，DPLL算法亦趋于完善。但无论DPLL算法结合何种优化技术，在判定SAT问题时，总会有约80%的时间耗费在BCP过程中，而判定变量的选取结果直接影响该过程的运行时间，所以判定变量选取策略在DPLL算法的优化技术研究中占有重要的地位，这也就成为了历年来DPLL算法优化领域的热门课题之一。针对这一现象，国内外研究者们在DPLL算法的基础上提出了许多有意义的策略，如最短子句出现频率最大法，在搜索过程中可尽早满足不易满足的子句;VSIDS算法，可使可满足子句数达到最多;基于子句长度动态约束的变量消除算法,实行变量分解策略降低了SAT问题的规模等。这些策略都是通过对CNF子句进行处理，进而选取判定变量。虽然可以有效的降低原始问题的规模，降低计算代价，但在判定变量的选取上仍具有一定的盲目性，求解效率的提升效果不显著。

为了更有针对性的解决随机选取判定变量的问题，研究者们不断探索，使得这一领域的研究成果不断涌现，如骨干点搜索算法，根据骨干点选取判定变量减少搜索空间;启发式极性决策算法，根据Karmaugh图覆盖分布计算出变量极性选取判定变量:基于子句权重决策策略，根据子句信息进行加权选取判定变量;子句前移策略，分析冲突子句，选取判定变量等。这些优化判定策略的提出，使得DPLL算法的整体性能得到了提高，实际运行速度有了质的飞跃，显著地提高了求解效率。

**1.3课程设计的主要研究工作**

本课程设计主要研究工作为根据DPLL算法对特定的cnf文件进行解析，而后求解出cnf文件转换而来的SAT问题，并对DPLL过程进行优化，比对优化前后的文字选择策略对于程序运行时间的改变。

**2系统需求分析与总体设计**

**2.1系统需求分析**

该系统要实现根据读取的cnf文件进行求解其可满足性，若满足的情况下输出其一个可满足解，还能随机生成一个具有唯一解的9\*9数独棋盘进行数独游戏。其中进行求解的流程是先读取文件，进行解析，把cnf文件中的条件解析输入程序中，然后利用DPLL算法进行可行性分析和答案求解，并且能够把求解出的答案进行检查分析。数独游戏则是根据数独特点先生成规则文件，然后根据规则文件进行DPLL推导，求解出唯一解棋盘。

**2.2系统总体设计**

系统总体由四个部分组成，其中四个头文件分别用于定义全局变量以及声明函数原型，对cnf文件进行解析并把数据导入程序中，用DPLL算法将cnf文件数据转换为SAT问题进行答案求解，以及最后一个用于随机生成数独棋盘并利用挖洞法创建一个具有唯一解的数独题目进行可交互性的数独游戏。Display.cpp则用于创建菜单，根据不同的功能序号调用不同的函数完成目标。（如图2-1所示）

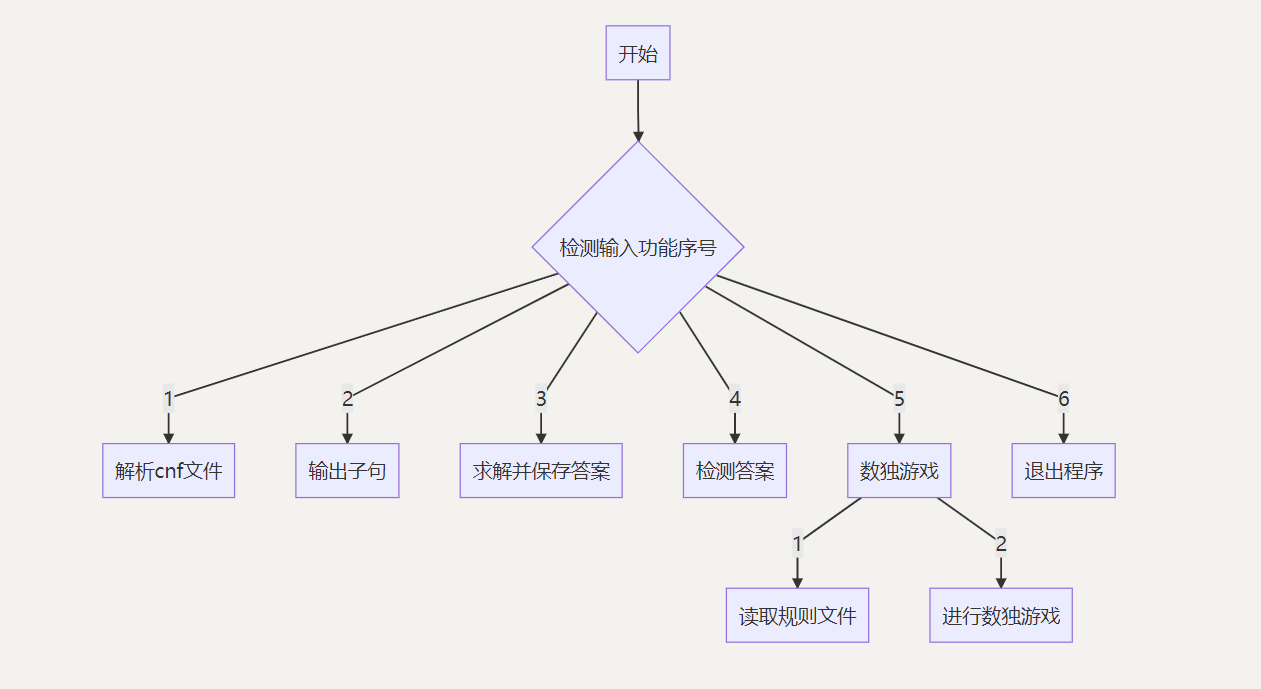


图2-1系统总体设计图

**3系统详细设计**

**3.1有关数据结构的定义**

系统要处理的数据主要是两个链表：子句链表以及文字链表，其中文字链表中储存读取到的文字，以及指向下一个文字链表的指针。子句链表中则储存着当前子句的第一个文字以及指向下一个子句链表的指针。还要定义储存子句和变量个数的int型变量，此外还要定义三个整型二维数据，一个用于储存棋盘，一个用于挖洞，一个用于玩家进行输入，同时还要定义一个字符串用来读取cnf的文件路径。（如图3-1所示）

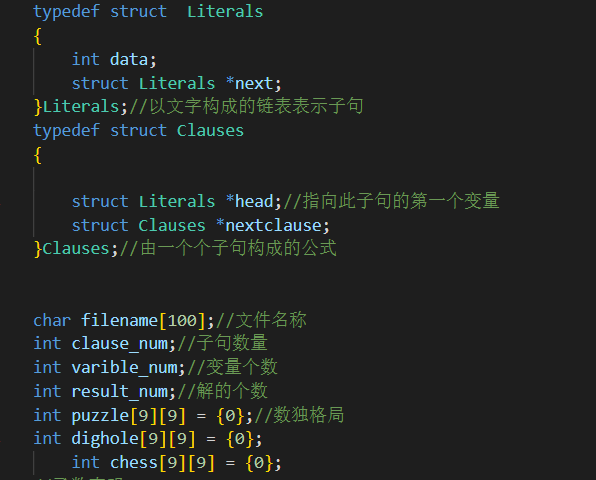


图3-1 数据结构定义

关于这些数据结构之间的联系，可以用图3-2表示：



图3-2 数据项之间的联系

**3.2主要算法设计**

主控，交互与显示模块(display.cpp):

打印出一个功能菜单列表，利用switch case对不同的输入调用不同的函数实现功能，如图3-3所示。

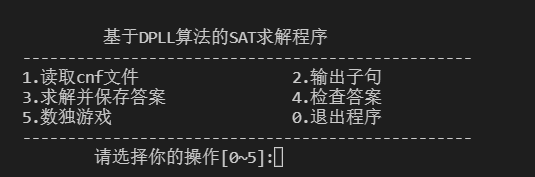


图3-3 功能菜单

cnf文件解析模块(cnfparser.h)：

利用fgetc()函数不断读取，抛弃cnf文件中注释型内容，直至读取到p cnf xxx格式开始正式读取数据，根据子句个数确定循环次数，然后以0作为单个子句中文字读取的结束符号，读取到非最后一个文字，就为下一个文字链表分配内存，并利用next指针切换到下一个链表读取下一个文字，直至该子句读取完毕，读取完一个子句并且不是最后一个子句，就为下一个子句链表及其第一个文字分配内存，重复上述过程。

DPLL求解模块(solver.h)：

循环判断子句链表中否存在单子句，若是存在单子句则利用规则遍历删除含有该文字的子句，删除所有子句中该文字的负的存在。没有单子句了则利用先找最多正文字的策略选取某文字作为单子句加入字句链表，重复上述操作，直至判断出来该算例无法实现或则求解出算例的一个可行解为止。（如图3-4所示）

数独模块(sudoku.h)：

先将数独游戏的约束转换成cnf文件的形式写入数独规则文件，再读取该文件，随机生成四个位置的数字，作为单子句添加到字句链表中，利用DPLL模块算出该条件下的棋盘完整格局，再利用挖洞法不断增加提示数，每次增加都把提示数作为单子句加入字句链表，判断是否能形成唯一解棋盘，不能就继续添加提示数，直至确定可以形成唯一解数独为止，保存当前的提示数格局和字句链表，计算在这种情况下利用DPLL模块求解棋盘的时间，打印数独题目，根据玩家输入的坐标和数字更改显示的棋盘格局，直至完成游戏，输出答案。（如图3-5所示）

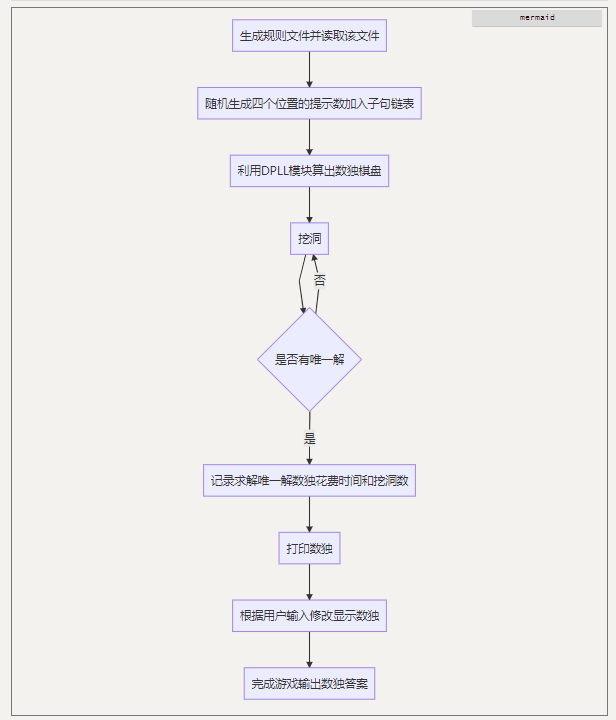
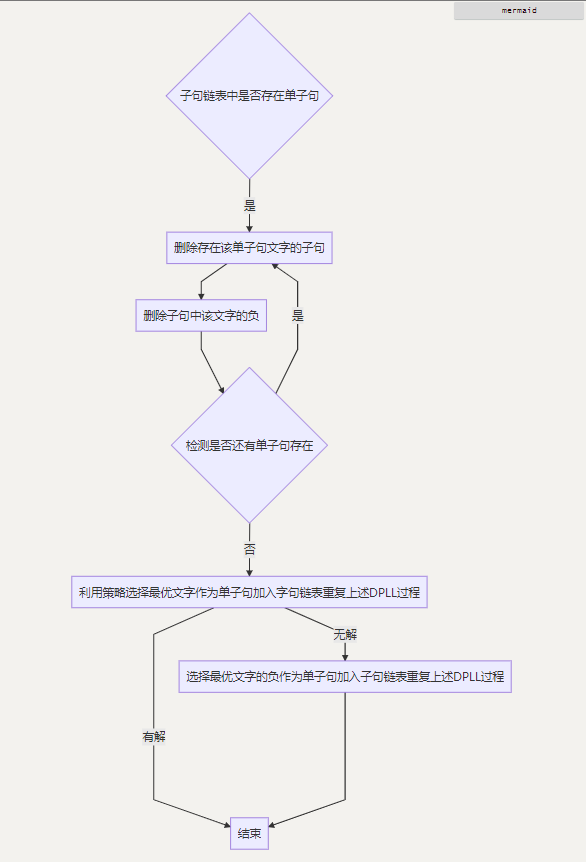


图3-4 DPLL模块流程 图3-5 数独模块流程

**4系统实现与测试**

**4.1系统实现**

该系统所要实现的是对于中小规模cnf算例利用DPLL算法进行求解，并且记录求解不同算例所需要的时间。完成DPLL求解SAT问题后，进一步应用于数独游戏，将数独棋盘转换成cnf文件的形式进行求解，寻找唯一解棋盘作为数独题目，并实现一定的交互性，即玩家可以根据数独当前的格局输入坐标和数字，一步步填充数独格局，每一步程序都能判断玩家是否合理填充数独以及填充数字是否有误，最终完成游戏，显示正确答案。

系统在Windows10的电脑中运行，开发环境是Visual Studio Code。

各函数功能和模块实现参见表4-1。

|  |  |  |
| --- | --- | --- |
| 函数名称 | 函数功能 | 实现模块 |
| loadcnf | 读入cnf文件中的数据到子句链表中 | 首先调用loadcnf读取文件，进入DPLL函数，根据isunitclause判断是否存在单子句，存在则用eraseclause函数和  eraseliteral函数删除相关子句和文字，否则赋值子句链表，选择最有文字用addclause加入复制的字句链表，递归调用DPLL函数。 |
| cleanclause | 清除所有子句 |
| eraseclause | 清除满足条件的某个子句 |
| eraseliteral | 清除指定子句中的特定文字 |
| isunitclause | 判断是否为单子句 |
| isemptyclause | 判断是否有子句被删空了 |
| DPLL | 利用dpll算法对子句进行化简运算 |
| copyclause | 拷贝子句 |
| addclause | 添加子句 |
| saveanswer | 保存答案到同名文件 |
| checkanswer | 查SAT问题求解答案的正确与否 |
| createpuzzle | 创建数独格局 | 利用createrule函数创建规则文件，读取文件，调用createpuzzle函数，不断调用dig\_hole和dfs找唯一解数独格局。 |
| createrule | 创建数独规则文件 |
| dfs | 求解答案个数 |
| dig\_hole | 对生成的数独格局进行挖洞 |

表4-1 各函数功能

具体程序详见附录。

**4.2系统测试**

4.2.1常用软件测试方法

按模块运行不同的代码，测试输入不同类型和大小的算例的功能是否正常。

4.2.2各模块功能及设计目标

cnf解析模块功能：能够解析cnf文件，并将文件中数据输入到文字链表和字句链表中。

DPLL求解模块：利用DPLL算法，实现对中小型规模cnf算例的求解和时间记录。并将答案输出到同名res文件中。

数独模块：实现唯一解数独棋盘的随机生成和挖洞，具有一定交互性和可玩性，可以判断玩家每一步的正确与否，并能输出完整数独棋盘的答案和求解该棋盘的时间。

4.2.3测试大纲

先选取老师提供的功能性测试算例，检查是否能够正确求解可满足和不可满足算例，而后输入性能测试算例，查看算例的运行时间，判断优化情况，最终去可满足和不可满足算例中从小到大依次选取算例进行求解，估计程序能够求解的最大算例规模。

4.2.4运行结果

cnf解析功能测试：

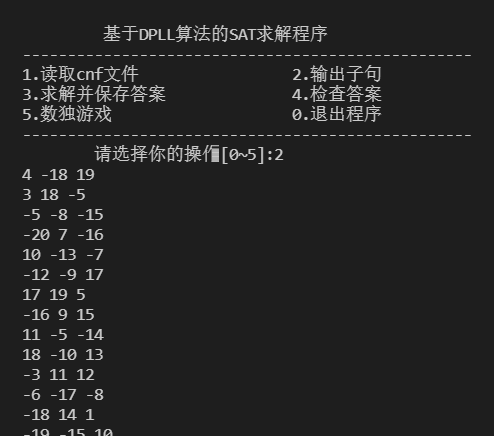


图4-1 cnf解析功能

DPLL功能性测试：

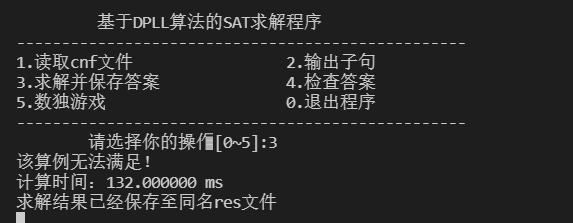
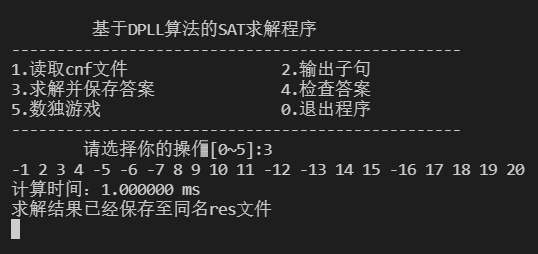


图4-2 可满足算例 图4-3 不可满足算例

DPLL性能测试：

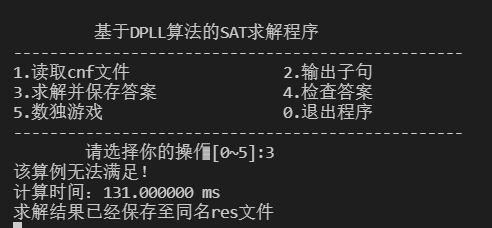


图4-4不可满足算例（u-5cnf\_3500\_3500\_30f1.shuffled-30）

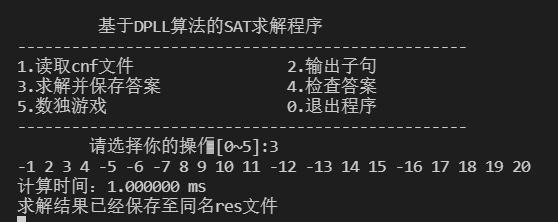


图4-5 可满足S型算例（problem1-20）

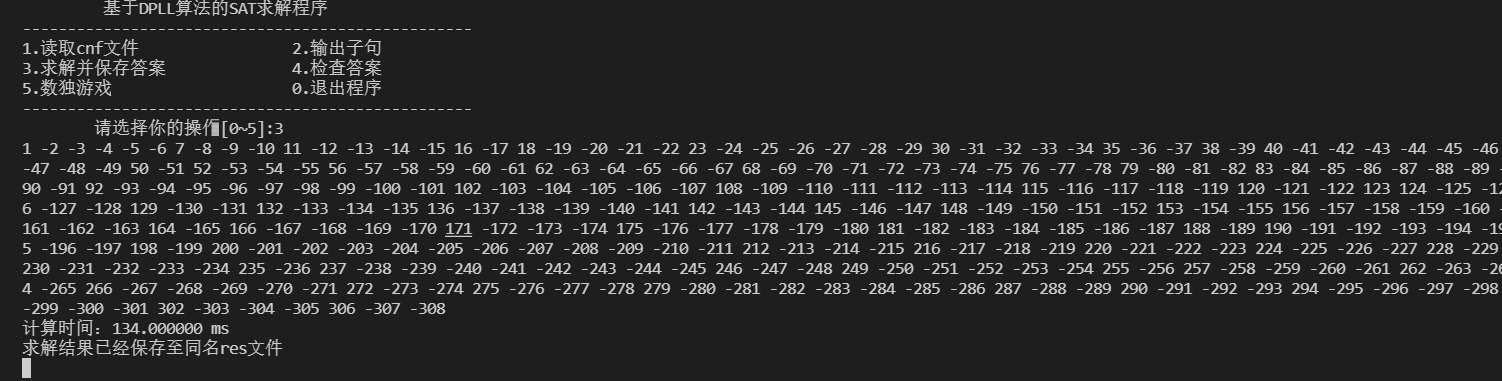


图4-6 可满足M型算例（sud00021）

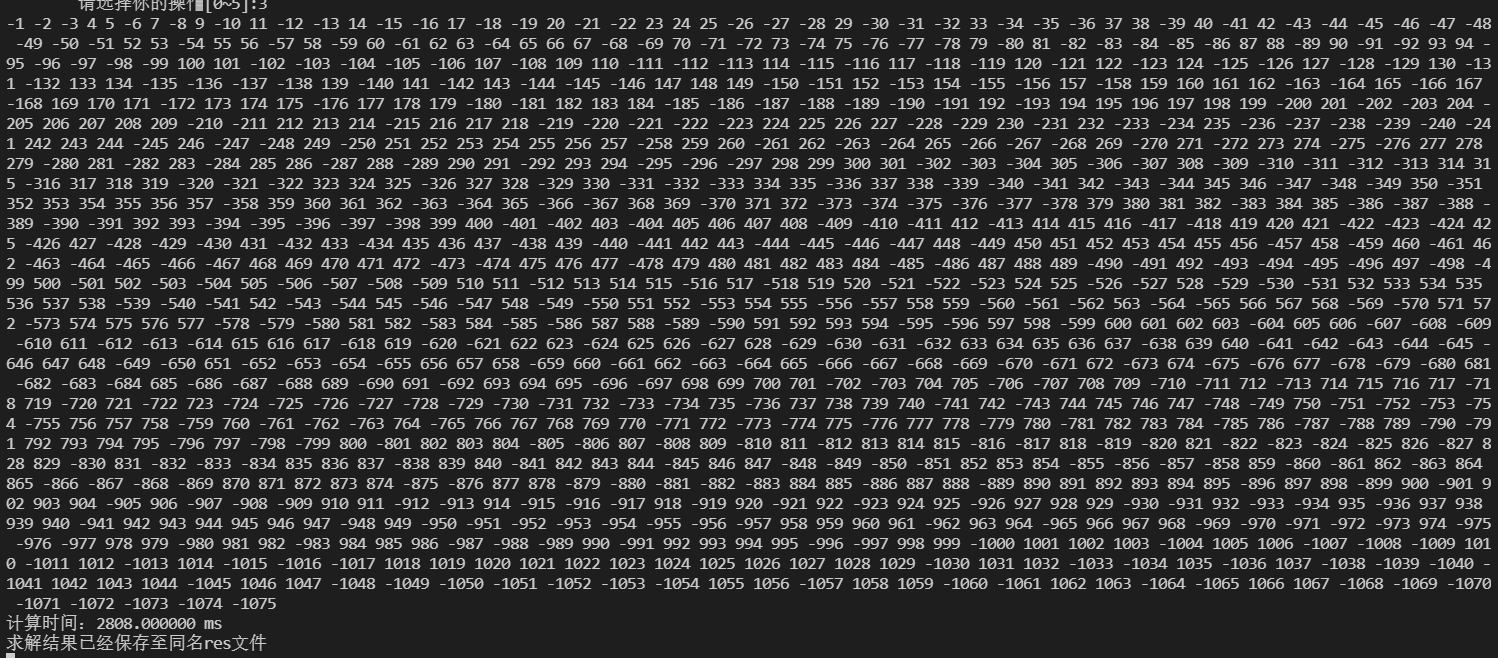


图4-7 可满足L型算例（eh-dp04s04.shuffled-1075）

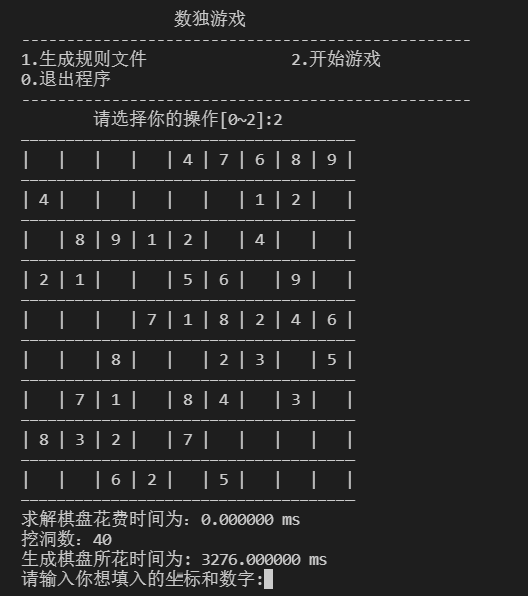


图4-8 数独棋盘生成 图4-9数独答案

4.2.5结果分析

根据测试结果，该系统具有良好的可满足和不可满足算例的判断和求解功能，能够很好的通过功能测试和性能测试，可完成大部分中小型规模的算例的答案求解。数独部分能够完成唯一性数独格局的生成，挖洞，可以根据用户的输入改变当前棋盘格局，可判断用户输入正误，可提前中断游戏，查看答案。综上所述，该程序满足了各模块的设计目标，较好地完成了既定任务。

**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）对cnf文件进行解析，用特定的数据结构进行储存；

（2）用DPLL算法对读取的cnf文件进行可满足分析和答案求解；

（3）优化DPLL算法，通过改变递归文字选择策略缩短较大算例运算时间；

（4）将数独游戏转换为可满足性问题，利用之前的DPLL算法改造后进行唯一解的推导。

**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）进一步优化DPLL算法中的文字选择策略，使之能够运行大型算例，并且缩短其运行时间。

（2）增强数独游戏的可玩性，例如控制挖洞数目从而实现难度选择等功能。

# 6体会

这是大学的第一次课设，是C语言和数据结构的综合课设，难度不小，刚看到时无从下手，后来根据过程性编程的思想，对繁琐复杂的题目抽丝剥茧，一点点提取任务目标，用不同的函数来完成，最终利用各个函数的组合调用，实现了庞大的系统任务。这其中遇到了不少难题，受到很多挫折，很多次都感到自己以及无以为继，写不下去了，终究还是在他人博客和论文中寻找问题的答案，请教学长和同学，最终才成功完成了这个艰巨的任务。过程中的苦很多，但完成任务收获的进步和成就感让我明白了课设的重要性和价值

**参考文献**

[1]王静康,张凤宝,夏淑倩等.论化工本科专业国际认证与国内认证的“实质性”.高等工程教育研究,2014,5:1-4

[2]Stone J A, Howard L P. A simple technique for observing periodic nonlinearities in Michelson interferometers. Precision Engineering,1998,22(4):220-232

[3]朱印红,袁衍明.Dreamweaver完美网页设计——技术入门篇.(第一版).北京:中国电力出版社,2006:19～20

[4]Lewis S L. Physics and chemistry of the solar system.北京:北京大学出版社,2014.1～2

[5]陈剑.上博简《民之父母》“而得既塞於四海矣”句解释[EB/OL］.简帛研究网站，http://www.bamboosilk.org/Wssf/2003/chenjian03.htm．2003-01-18

**附录**

def.h

#ifndef \_def\_h\_

#define \_def\_h\_

#include<stdio.h>

#include<string.h>

#include<time.h>

#include<stdlib.h>

#include<string.h>

#include<math.h>

#define OK 1

#define ERROR -1

#define TRUE 1

#define FALSE 0

#define satisfied 1

#define unsatisfied 0

#define UNFEASIBLE 2

typedef struct Literals

{

int data;

struct Literals \*next;

}Literals;//以文字构成的链表表示子句

typedef struct Clauses

{

struct Literals \*head;//指向此子句的第一个变量

struct Clauses \*nextclause;

}Clauses;//由一个个子句构成的公式

char filename[100];//文件名称

int clause\_num;//子句数量

int varible\_num;//变量个数

int result\_num;//解的个数

int puzzle[9][9] = {0};//数独格局

int dighole[9][9] = {0};

int chess[9][9] = {0};

//函数声明

int loadcnf(Clauses \*&cls); //读入cnf文件中的数据到子句链表中

void cleanclause(Clauses \*&cls); //清除所有子句

int eraseclause(Clauses \*&cls , Clauses \*&fcls); //清除满足条件的某个子句

int eraseliteral(Literals \*&lit,Literals \*&flit); //清除指定子句中的特定文字

int isunitclause(Literals \*lit); //判断是否为单子句

//int evaluateclause(Literals \*lit,int v[]); //评估子句的真假

int isemptyclause(Clauses \*cls); //判断是否有子句被删空了

int DPLL(Clauses \*&cls,int value[]); //利用dpll算法对子句进行化简运算

void copyclause(Clauses \*&cls1 , Clauses \*cls2); //把cls2拷贝到cls1

int addclause(Clauses \*&cls,Clauses \*&cls\_add); //添加子句

int saveanswer(int result , double duration , int value[]);

//保存答案到同名文件

int checkanswer(int value[] , Clauses \*&cls , int result);

//检查SAT问题求解答案的正确与否

void createpuzzle(Clauses \*&cls , int value[]); //创建数独格局

int createrule(Clauses \*&cls);

int dfs(Clauses \*&cls , int value[]); //求解答案个数

int dig\_hole(Clauses \*&cls , int value[]);//对生成的数独格局进行挖洞

#endif

Cnfparser.h

#include"def.h"

//读入cnf文件

int loadcnf(Clauses \*&cls)

{

FILE \*fp;

char ch;

char chs[30];

Clauses \*cp;

Literals \*lp;

int num;

fp = fopen(filename,"r");

if(fp == NULL)

{

printf("文件打开失败！\n");

return 0;

}

while ((ch = getc(fp)) == 'c')//当读入行为注释行的时候

{

while ((ch = getc(fp)) != '\n')//重复读入，抛弃改行内容直至换行

{

continue;

}

}

fscanf(fp,"%[ cnf ]",chs); //读取cnf这个字符串

fscanf(fp,"%d",&varible\_num); //读取变量个数

fscanf(fp,"%d",&clause\_num); //读取字句个数

cls = (Clauses \*)malloc(sizeof(Clauses));

cls ->nextclause = NULL;

cls->head = (Literals \*)malloc(sizeof(Literals));

cls->head->next = NULL;

cp = cls; //指向第一个子句

lp = cls->head;//指向第一个子句的第一个变量

for(int i = 1;i <= clause\_num;i++)

{

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

while (num != 0)

{

lp->data = num;

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

if(num == 0)

lp -> next = NULL;

else

{

lp -> next = (Literals \*)malloc(sizeof(Literals));

lp = lp -> next;

lp -> next = NULL;

}

}

if(i == clause\_num)//当读取到最后一个子句的时候

{

cp -> nextclause = NULL;

break;

}

//当不是最后一个子句的时候

cp->nextclause = (Clauses \*)malloc(sizeof(Clauses));//为下一个子句分配内存

cp->nextclause->head = (Literals \*)malloc(sizeof(Literals));//为下一个子句的第一个变量分配内存

cp = cp->nextclause; //子句指针指向下一个子句

lp = cp->head; //文字指针指向新子句的开头

}

printf("cnf文件数据读取完毕！\n");

fclose(fp);

return 1;

}

Solver.h

//#include"def.h"

#include"cnfparser.h"

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

\* 函数名称：cleanclause

\* 参数：Clauses\*&(子句链表的指针的引用)

\* 功能：清楚当前储存的所有子句链表

\* 返回值：无

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

void cleanclause(Clauses \*&cls)//清除所有子句

{

Clauses \*cp1 , \*cp2;

Literals \*lp1 , \*lp2;

for(cp1 = cls;cp1 != NULL;cp1 = cp2)

{

cp2 = cp1->nextclause;

for(lp1 = cp1->head;lp1 != NULL;lp1 = lp2)

{

lp2 = lp1->next;

free(lp1);

}

free(cp1);

}

cls = NULL;

}

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

\* 函数名称：isunitclause

\* 参数：Literal\*(文章链表的指针)

\* 功能：判断是否为单子句

\* 返回值：成功返回1，否返回0

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

int isunitclause(Literals \*lit)

{

if(lit != NULL && lit->next == NULL)

return 1;

else

return 0;

}

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

\* 函数名称：eraseclause

\* 参数：Clauses\*& Clauses\*&(子句链表的指针的引用) 第一个子句指针指向要删除子句 第二个指针指向子句链表的头部

\* 功能：清除满足条件的某个子句

\* 返回值：成功返回1，失败返回0

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

int eraseclause(Clauses \*&cls, Clauses \*&fcls)

{

Clauses \*cp = fcls;

if(cp == cls)//删除子句为头部子句

fcls = fcls->nextclause;

else

{

while(cp != NULL && cp->nextclause != cls)

cp = cp->nextclause;

cp->nextclause = cp->nextclause->nextclause;

}

free(cls);

cls = NULL;

return 1;

}

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

\* 函数名称：eraseliteral

\* 参数：Literal\*& Literal\*&(文字链表的指针的引用) 第一个子句指针指向要删除文字 第二个指针指向文字链表的头部

\* 功能：清除指定子句中的特定文字

\* 返回值：成功返回1，失败返回0

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

int eraseliteral(Literals \*&lit,Literals \*&flit)

{

Literals \*lp = flit;

if(lp == lit)//删除的文字在链表头部

flit = flit->next;

else

{

while (lp != NULL && lp->next != lit)

{

lp = lp->next;

}

lp->next = lp->next->next;

}

free(lit);

lit = NULL;

return OK;

}

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

\* 函数名称：isemptyclause

\* 参数：Clause\*

\* 功能: 判断是否有子句是空的

\* 返回值：是返回1，否返回0

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

int isemptyclause(Clauses \*cls)

{

Clauses \*cp = cls;

while (cp != NULL)

{

if(cp->head == NULL)

return 1;

cp = cp->nextclause;

}

return 0;

}

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

\* 函数名称：copyclause

\* 参数：Clause\*& Clause\*

\* 功能: 拷贝一份子句链表到前者链表中

\* 返回值：无

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

void copyclause(Clauses \*&cls1,Clauses \*cls2)

{

Clauses \*cp1 , \*cp2;

Literals \*lp1 , \*lp2;

cls1 = (Clauses \*)malloc(sizeof(Clauses));

cls1->head = (Literals \*)malloc(sizeof(Literals));

cls1->nextclause = NULL;

cls1->head->next = NULL;

for(cp2 = cls2 , cp1 = cls1;cp2 != NULL;cp1 = cp1->nextclause,cp2 = cp2->nextclause)

{

for(lp1 = cp1->head , lp2 = cp2->head; lp2 != NULL;lp2 = lp2 ->next,lp1 = lp1->next)

{

lp1->data = lp2->data;

lp1->next = (Literals\*)malloc(sizeof(Literals));

lp1->next->next = NULL;

if(lp2->next == NULL)

{

free(lp1->next);

lp1->next = NULL;

}

}

//cp1->head = cp2->head;

cp1->nextclause = (Clauses\*)malloc(sizeof(Clauses));

cp1->nextclause->head = (Literals\*)malloc(sizeof(Literals));

cp1->nextclause->nextclause = NULL;

cp1->nextclause->head->next = NULL;

if(cp2->nextclause == NULL)

{

free(cp1->nextclause->head);

free(cp1->nextclause);

cp1->nextclause = NULL;

}

}

}

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

\* 函数名称：addclause

\* 参数：Clause\*& clause\*&

\* 功能: 向前者子句链表中添加后者

\* 返回值：成功返回1，否则返回0

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

int addclause(Clauses \*&cls,Clauses \*&cls\_add)

{

cls\_add->nextclause = cls;

cls = cls\_add;

return 1;

}

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

\* 函数名称：DPLL

\* 参数：Clause\*& int[]

\* 功能: 利用dpll算法求解sat问题

\* 返回值：有解返回1，无解返回0

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

int DPLL(Clauses \*&cls,int value[])

{

Clauses \*clsp1 = cls , \*clsp2 = cls , \*clsp3;

Literals \*lp;

int \*count;

count = (int \*)malloc(sizeof(int) \* (2 \* varible\_num + 1));

search: while(clsp1 != NULL && isunitclause(clsp1->head) == 0)

clsp1 = clsp1->nextclause;

if(clsp1 != NULL)//有单子句存在

{

//先给单子句中的变量赋值

if(clsp1->head->data > 0)

value[clsp1->head->data] = 1;

else

value[-clsp1->head->data] = 0;

int knownnum = clsp1->head->data;//记录一下已赋值变量

for(clsp1 = cls;clsp1 != NULL;clsp1 = clsp2)

{

clsp2 = clsp1->nextclause;

for(lp = clsp1->head;lp != NULL;lp = lp->next)

{

if(lp->data == knownnum)//子句中含有该文字，删除子句

{

eraseclause(clsp1,cls);

break;

}

if(lp->data == -knownnum)//子句中含有该文字的反文字，删除文字

{

eraseliteral(lp,clsp1->head);

break;

}

}

}

if(cls == NULL)//子句链表清空，该cnf可满足

{

free(count);

return satisfied;

}

if(isemptyclause(cls))//若存在空子句，说明该子句无法满足,则cnf无法满足

{

free(count);

cleanclause(cls);

return unsatisfied;

}

clsp1 = cls;

goto search;//返回继续利用规则化简cnf

}

for(int i = 0;i <= 2 \* varible\_num;i++)

count[i] = 0;

//计算各文字出现个数，作为选取标准

for(clsp2 = cls;clsp2 != NULL;clsp2 = clsp2->nextclause)

{

for(lp = clsp2->head;lp != NULL;lp = lp->next)

{

if(lp->data > 0)

count[lp->data]++;

if(lp->data < 0)

count[varible\_num - lp->data]++;

}

}

int maxcount = 0;

int word;//记录出现最多的文字

//此处先找出现次数最多的正文字，遍历完正文字再找负文字，提升了效率，比单纯找最多文字还快

for(int i = 1;i <= varible\_num;i++)

{

if(maxcount < count[i])

{

maxcount = count[i];

word = i;

}

}

if(maxcount == 0)//cnf中不存在正文字了

{

for(int i = varible\_num + 1;i <= varible\_num \* 2;i++)

{

if(maxcount < count[i])

{

maxcount = count[i];

word = varible\_num - i;

}

}

}

/\*for(int i = 1;i <= varible\_num \* 2;i++)

{

if(maxcount < count[i])

{

maxcount = count[i];

if(i > varible\_num)

word = varible\_num - i;

else

word = i;

}

}\*/

free(count);

//建立一个该文字的单子句，添加到子句链表中进行化简

clsp2 = (Clauses \*)malloc(sizeof(Clauses));

clsp2->head = (Literals \*)malloc(sizeof(Literals));

clsp2->head->next = NULL;

clsp2->head->data = word;

clsp2->nextclause = NULL;

copyclause(clsp3,cls);

addclause(clsp3,clsp2);

if(DPLL(clsp3,value) == satisfied)//进入第一分支搜索

return satisfied;

cleanclause(clsp3);//释放内存

//第一分支没有答案，返回到第二分支进行搜索

clsp2 = (Clauses \*)malloc(sizeof(Clauses));

clsp2->head = (Literals \*)malloc(sizeof(Literals));

clsp2->head->next = NULL;

clsp2->head->data = -word;

clsp2->nextclause = NULL;

copyclause(clsp3,cls);

addclause(clsp3,clsp2);

int result = DPLL(clsp3,value);//记录第二分支状态

cleanclause(clsp3);

//cleanclause(cls);

return result;

}

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

\* 函数名称：saveanswer

\* 参数：int double int\*

\* 功能: 将答案保存到同名文件中

\* 返回值：成功返回1，失败返回0

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

int saveanswer(int result , double duration , int value[])

{

FILE \*fp;

for(int i = 0;filename[i] != '\0';i++)

{

if(filename[i] == '.' && filename[i+1] == 't' &&filename[i+2] == 'x' && filename[i + 3] == 't')

{

//更换文件名后缀

filename[i + 1] = 'r';

filename[i + 2] = 'e';

filename[i + 3] = 's';

break;

}

}

fp = fopen(filename,"w");

if(fp == NULL)

{

printf("文件打开失败！\n");

return 0;

}

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

fprintf(fp,"v ");

if(result == 1)

{

for(int i = 1;i <= varible\_num;i++)

{

if(value[i] == 1)

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

else

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

}

}

fprintf(fp,"\n");

fprintf(fp,"t %lf", duration \* 1000);

fclose(fp);

return 1;

}

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

\* 函数名称：checkanswer

\* 参数：int\* clause\* int

\* 功能: 检查求解答案的正确性

\* 返回值：正确返回1，错误返回0

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

int checkanswer(int value[] , Clauses \*&cls , int result)

{

Clauses \*clsp;

Literals \*lp;

if(result == 1)//有解情况

{

//遍历所有子句

for(clsp = cls;clsp != NULL;clsp = clsp->nextclause)

{

for(lp = cls->head;lp != NULL;lp = lp->next)

{

if(lp->data > 0 && value[lp->data] == 1)

break;

else if(lp->data < 0 && value[-(lp->data)] == 0)

break;

else if(lp -> next == NULL)//如果遍历到一个子句的末尾还没找到满足子句的文字

return FALSE;

}

}

return TRUE;

}

else if(result == 0)//无解情况

{

for(clsp = cls;clsp != NULL;clsp = clsp->nextclause)

{

for(lp = cls->head;lp != NULL;lp = lp->next)

{

if(value[lp->data] == 1 && lp->data > 0)

return FALSE;

else if(value[lp->data] == 0 && lp->data < 0)

return FALSE;

}

}

return TRUE;

}

return UNFEASIBLE;

}

Sudoku.h

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

\* 函数名称：createrule

\* 参数：Clause\*& int[]

\* 功能: 利用dpll算法求解sat问题,计算答案个数

\* 返回值：唯一解返回1，否则返回0

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

int createrule(Clauses \*&cls)

{

int x , y , z , i , j , k , l;

FILE \*fp;

fp = fopen("D:\\sudoku\_rule.txt", "w");

fprintf(fp, "p cnf 729 10287\n");

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

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

for (z = 1; z <= 8; ++z) {

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

fprintf(fp, "%d %d 0\n", -(81 \* x + 9 \* y + z), -(81 \* x + 9 \* y + i));//每个位置，数字1~9至多出现一次

}

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

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

for (y = 0; y < 8; ++y) {

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

fprintf(fp, "%d %d 0\n", -(81 \* x + 9 \* y + z), -(81 \* x + 9 \* i + z));//每一行，数字1~9至多出现一次

}

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

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

for (x = 0; x < 8; ++x) {

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

fprintf(fp, "%d %d 0\n", -(81 \* x + 9 \* y + z), -(81 \* i + 9 \* y + z));//每一列，数字1~9至多出现一次

}

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

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

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

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

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

fprintf(fp, "%d ", 81 \* (3 \* i + x) + 9 \* (3 \* j + y) + z);//数字1~9在每个3×3数独中至少出现一次

fprintf(fp, "0\n");

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

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

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

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

if (l != y)

fprintf(fp, "%d %d 0\n", -(81 \* (3 \* i + x) + 9 \* (3 \* j + y) + z),

-(81 \* (3 \* i + k) + 9 \* (3 \* j + l) + z));//数字1~9在每个3×3数独中至多出现一次

}

}

}

fclose(fp);

strcpy(filename,"D:\\sudoku\_rule.txt");

FILE \*fp1;

char ch;

char chs[30];

Clauses \*cp;

Literals \*lp;

int num;

fp1 = fopen(filename,"r");

if(fp1 == NULL)

{

printf("文件打开失败！\n");

return 0;

}

while ((ch = getc(fp1)) == 'c')//当读入行为注释行的时候

{

while ((ch = getc(fp1)) != '\n')//重复读入，抛弃改行内容直至换行

{

continue;

}

}

fscanf(fp1,"%[ cnf ]",chs); //读取cnf这个字符串

fscanf(fp1,"%d",&varible\_num); //读取变量个数

fscanf(fp1,"%d",&clause\_num); //读取字句个数

cls = (Clauses \*)malloc(sizeof(Clauses));

cls ->nextclause = NULL;

cls->head = (Literals \*)malloc(sizeof(Literals));

cls->head->next = NULL;

cp = cls; //指向第一个子句

lp = cls->head;//指向第一个子句的第一个变量

for(int i = 1;i <= clause\_num;i++)

{

fscanf(fp1,"%d",&num);

while (num != 0)

{

lp->data = num;

fscanf(fp1,"%d",&num);

if(num == 0)

lp -> next = NULL;

else

{

lp -> next = (Literals \*)malloc(sizeof(Literals));

lp = lp -> next;

lp -> next = NULL;

}

}

if(i == clause\_num)//当读取到最后一个子句的时候

{

cp -> nextclause = NULL;

break;

}

//当不是最后一个子句的时候

cp->nextclause = (Clauses \*)malloc(sizeof(Clauses));//为下一个子句分配内存

cp->nextclause->head = (Literals \*)malloc(sizeof(Literals));//为下一个子句的第一个变量分配内存

cp = cp->nextclause; //子句指针指向下一个子句

lp = cp->head; //文字指针指向新子句的开头

}

printf("cnf文件数据读取完毕！\n");

fclose(fp1);

return 1;

}

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

\* 函数名称：dfs

\* 参数：Clause\*& int[]

\* 功能: 利用dpll算法求解sat问题,计算答案个数

\* 返回值：唯一解返回1，否则返回0

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

int dfs(Clauses \*&cls,int value[])

{

Clauses \*clsp1 = cls , \*clsp2 = cls , \*clsp3;

Literals \*lp;

int \*count;

int re;

count = (int \*)malloc(sizeof(int) \* (2 \* varible\_num + 1));

find: while(clsp1 != NULL && isunitclause(clsp1->head) == 0)

clsp1 = clsp1->nextclause;

if(clsp1 != NULL)//有单子句存在

{

//先给单子句中的变量赋值

if(clsp1->head->data > 0)

value[clsp1->head->data] = 1;

else

value[-clsp1->head->data] = 0;

int knownnum = clsp1->head->data;//记录一下已赋值变量

for(clsp1 = cls;clsp1 != NULL;clsp1 = clsp2)

{

clsp2 = clsp1->nextclause;

for(lp = clsp1->head;lp != NULL;lp = lp->next)

{

if(lp->data == knownnum)//子句中含有该文字，删除子句

{

eraseclause(clsp1,cls);

break;

}

if(lp->data == -knownnum)//子句中含有该文字的反文字，删除文字

{

eraseliteral(lp,clsp1->head);

break;

}

}

}

if(cls == NULL)//子句链表清空，该cnf可满足

{

free(count);

result\_num++;

return satisfied;

}

if(isemptyclause(cls))//若存在空子句，说明该子句无法满足,则cnf无法满足

{

free(count);

cleanclause(cls);

return unsatisfied;

}

clsp1 = cls;

goto find;//返回继续利用规则化简cnf

}

for(int i = 0;i <= 2 \* varible\_num;i++)

count[i] = 0;

//计算各文字出现个数，作为选取标准

for(clsp2 = cls;clsp2 != NULL;clsp2 = clsp2->nextclause)

{

for(lp = clsp2->head;lp != NULL;lp = lp->next)

{

if(lp->data > 0)

count[lp->data]++;

if(lp->data < 0)

count[varible\_num - lp->data]++;

}

}

int maxcount = 0;

int word;//记录出现最多的文字

//此处先找出现次数最多的正文字，遍历完正文字再找负文字，提升了效率，比单纯找最多文字还快

for(int i = 1;i <= varible\_num;i++)

{

if(maxcount < count[i])

{

maxcount = count[i];

word = i;

}

}

if(maxcount == 0)//cnf中不存在正文字了

{

for(int i = varible\_num + 1;i <= varible\_num \* 2;i++)

{

if(maxcount < count[i])

{

maxcount = count[i];

word = varible\_num - i;

}

}

}

free(count);

//建立一个该文字的单子句，添加到子句链表中进行化简

clsp2 = (Clauses \*)malloc(sizeof(Clauses));

clsp2->head = (Literals \*)malloc(sizeof(Literals));

clsp2->head->next = NULL;

clsp2->head->data = word;

clsp2->nextclause = NULL;

copyclause(clsp3,cls);

addclause(clsp3,clsp2);

if(result\_num > 1)

{

return 0;

}

if(dfs(clsp3,value) == 0)//进入第一分支搜索

return 0;

cleanclause(clsp3);//释放内存

if(result\_num > 1)

{

return 0;

}

//第一分支搜索完，返回到第二分支进行搜索

clsp2 = (Clauses \*)malloc(sizeof(Clauses));

clsp2->head = (Literals \*)malloc(sizeof(Literals));

clsp2->head->next = NULL;

clsp2->head->data = -word;

clsp2->nextclause = NULL;

copyclause(clsp3,cls);

addclause(clsp3,clsp2);

if(dfs(clsp3,value) == 0)//进入第二分支

return 0;

cleanclause(clsp3);

//cleanclause(cls);

if(result\_num > 1)

{

return 0;

}

return 1;

}

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

\* 函数名称：createpuzzle

\* 参数：Clauses \*& int[]

\* 功能: 创建数独棋盘

\* 返回值：无

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

void createpuzzle(Clauses \*&cls , int value[])

{

int x, y, z, i, j, k, l;

int dig\_order[82], index;

Clauses \*clsp1 = NULL, \*clsp2 = NULL;

int cnt = 0;//记录随机变量个数

copyclause(clsp1,cls);//先拷贝一份数独规则

srand((unsigned int)time(0));

while(cnt != 4)//四个随机数是一定多解并且没有无解情况的最大值

{

x = rand() % 9;

y = rand() % 9;

z = rand() % 8 + 1;

if(puzzle[x][y] == 0)

{

puzzle[x][y] = z;

cnt++;

}

clsp2 = (Clauses \*)malloc(sizeof(Clauses));

clsp2->head = (Literals \*)malloc(sizeof(Literals));

clsp2->head->next = NULL;

clsp2->head->data = x \* 81 + y \* 9 + z;

clsp2->nextclause = NULL;

addclause(clsp1,clsp2);

}

cnt = 0;

DPLL(clsp1,value);

cleanclause(clsp1);//及时释放内存

for(x = 0;x <= 8;x++)//把数度格局存入棋盘

{

for(y = 0;y <= 8;y++)

{

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

if(value[x \* 81 + y \* 9 + z] == 1)

puzzle[x][y] = z;

}

}

}

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

\* 函数名称：dighole

\* 参数：Clauses \*& int[]

\* 功能: 对生成的数独进行挖洞

\* 返回值：无

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

int dig\_hole(Clauses \*&cls , int value[])

{

clock\_t begin , finish;

double t;

int x , y , z;

int cnt = 0;

srand((unsigned int)time(0));

Clauses \*clsp1 = NULL, \*clsp2 = NULL;//clsp1用来copy cls，clsp2用来添加子句;

copyclause(clsp1,cls);

do{

result\_num = 0;

x = rand() % 9;

y = rand() % 9;

if(dighole[x][y] == 0)

{

dighole[x][y] = 1;

clsp2 = (Clauses \*)malloc(sizeof(Clauses));

clsp2->head = (Literals \*)malloc(sizeof(Literals));

clsp2->head->next = NULL;

clsp2->head->data = x \* 81 + y \* 9 + puzzle[x][y];

clsp2->nextclause = NULL;

addclause(clsp1,clsp2);

}

}while(dfs(clsp1,value) != 1);

begin = clock();

DPLL(clsp1,value);

finish = clock();

t = (double)(finish - begin)/CLOCKS\_PER\_SEC;

cleanclause(cls);

cleanclause(clsp1);

printf("—————————————————————————————————————\n");

for(int i = 0;i <= 8;i++)

{

for(int j = 0;j <= 8;j++)

{

if(j == 0)

printf("| ");

if(dighole[i][j] == 1)

{

chess[i][j] = puzzle[i][j];

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

cnt++;

}

else

printf(" ");

if(j != 8)

printf("| ");

else

printf("|");

}

printf("\n");

printf("—————————————————————————————————————\n");

}

printf("求解棋盘花费时间为：%lf ms\n",t\*1000);

printf("挖洞数：%d\n",cnt);

return cnt;

}

Display.cpp

#include"solver.h"

#include "sudoku.h"

int main()

{

Clauses \*cls = NULL;

Clauses \*clsp;

Literals \*lit = NULL;

clock\_t start , finish;

clock\_t a , b;

double duration;

int op = 1,result = 2, re = 0 , option = 1 , covernum = 0;

int x , y , z;

int \*value;

while (op)

{

system("cls");

printf("\n\n");

printf(" 基于DPLL算法的SAT求解程序\n");

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

printf("1.读取cnf文件 2.输出子句\n");

printf("3.求解并保存答案 4.检查答案\n");

printf("5.数独游戏 0.退出程序\n");

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

printf(" 请选择你的操作[0~5]:");

scanf("%d", &op);

switch (op)

{

case 1:

printf("请输入要读取的文件路径：\n");

scanf("%s",filename);

loadcnf(cls);

getchar();getchar();

break;

case 2:

if(cls == NULL)

printf("还未读取cnf文件！\n");

else

{

for(clsp = cls;clsp != NULL;clsp = clsp->nextclause)

{

for(lit = clsp->head;lit != NULL;lit = lit->next)

{

printf("%d ",lit->data);

}

printf("\n");

}

}

getchar();getchar();

break;

case 3:

if(cls == NULL)

printf("文件未导入！\n");

else

{

value = (int \*)malloc(sizeof(int) \* (varible\_num +1));

for(int i = 0;i <= varible\_num;i++)

value[i] = -1;//只是初始化一下，没有赋值含义！

start = clock();

result = DPLL(cls,value);

finish = clock();

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

if(result == satisfied)

{

for(int i = 1;i <= varible\_num;i++)

{

if(value[i] == 1)

printf("%d ",i);

else

printf("%d ",-i);

}

printf("\n");

}

else

{

printf("该算例无法满足！\n");

}

printf("计算时间：%lf ms\n",duration \* 1000);

if(saveanswer(result,duration,value) == 1)

printf("求解结果已经保存至同名doc文件\n");

else

printf("结果保存失败！\n");

}

getchar();getchar();

break;

case 4:

re = checkanswer(value,cls,result);

if(re == TRUE)

printf("答案正确！\n");

else if(re == FALSE)

printf("答案错误！\n");

else

printf("没用可供验证的答案！\n");

getchar();getchar();

break;

case 5:

option = 1;

while (option)

{

system("cls");

printf("\n\n");

printf(" 数独游戏\n");

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

printf("1.生成规则文件 2.开始游戏\n");

printf("0.退出程序\n");

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

printf(" 请选择你的操作[0~2]:");

scanf("%d",&option);

switch (option)

{

case 1:

createrule(cls);

getchar();getchar();

break;

case 2:

value = (int \*)malloc(sizeof(int) \* (varible\_num +1));

for(int i = 0;i <= varible\_num;i++)

value[i] = -1;//只是初始化一下，没有赋值含义！

a = clock();

createpuzzle(cls,value);

covernum = dig\_hole(cls,value);

b = clock();

duration = (double)(b - a)/ CLOCKS\_PER\_SEC;

printf("生成棋盘所花时间为: %lf ms\n",duration\*1000);

while (1)

{

printf("请输入你想填入的坐标和数字:");

scanf("%d %d %d",&x , &y , &z);

if(x == 0 && y == 0 && z == 0)//没做出来又不想玩了

break;

if(x <= 0 || x > 9 || y <= 0 || y > 9 || z > 9 || z <= 0)

{

printf("输入格式错误！\n");

continue;

}

if(dighole[x - 1][y - 1] == 1)//修改到提示数位置去了

{

printf("该点为题目提示数，不可以修改！\n");

continue;

}

else if(chess[x - 1][y - 1] == 0)//此处还没填写数字

{

chess[x - 1][y - 1] = z;

covernum++;

}

else//修改自己填的数字

{

chess[x - 1][y - 1] = z;

}

//输出当前情况下的棋盘

printf("—————————————————————————————————————\n");

for(int i = 0;i <= 8;i++)

{

for(int j = 0;j <= 8;j++)

{

if(j == 0)

printf("| ");

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

{

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

}

else

printf(" ");

if(j != 8)

printf("| ");

else

printf("|");

}

printf("\n");

printf("—————————————————————————————————————\n");

}

//比对数字正误

if(covernum < 81 && chess[x - 1][y - 1] == puzzle[x - 1][y - 1])

printf("恭喜你，这个位置填对啦！\n");

else if(chess[x - 1][y - 1] != puzzle[x - 1][y - 1])

{

printf("很遗憾，这个数字填错了！\n");

}

else if(covernum == 81 && chess[x - 1][y - 1] == puzzle[x - 1][y - 1])

{

printf("恭喜你！你已经成功做出了这个数独！\n");

break;

}

}

printf("按下enter键输出答案");

getchar();getchar();

printf("—————————————————————————————————————\n");

for(int i = 0;i <= 8;i++)

{

for(int j = 0;j <= 8;j++)

{

if(j == 0)

printf("| ");

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

if(j != 8)

printf("| ");

else

printf("|");

}

printf("\n");

printf("—————————————————————————————————————\n");

}

getchar();getchar();

break;

case 0:

break;

}

}

getchar();

break;

case 0:

break;

}

}

return 0;

}