

**课程设计报告**

**题目：基于SAT的双数独游戏求解程序**

**课程名称：程序设计综合课程设计**

**专业班级： 大数据2102班**

**学 号： U202112313**

**姓 名： 陈东平**

**指导教师： 李贤芝**

**报告日期： 2022.9.10**

**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：**将数双独游戏[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%)

**目录**

**任务书** I

**1引言** 4

1.1课题背景与意义 4

1.2国内外研究现状 5

1.3课程设计的主要研究工作 5

**2系统需求分析与总体设计** 6

2.1系统需求分析 6

2.2系统总体设计 7

3**系统详细设计** 11

3.1有关数据结构的定义 11

3.2主要算法设计 15

**4系统实现与测试** 18

4.1系统实现 18

4.2主要算法设计 22

**5 总结与展望** 27

5.1全文总结 27

5.2工作展望 28

**6 体会** 29

**参考文献** 30

**附录：源代码** 31

**1．引言**

**1.1课题背景与意义**

对于计算机科学与技术、大数据、与物联网专业大二学生，在前两个学期已经学习了C语言程序设计、数据结构两门面向编程知识与技术的基础理论课，以及C语言程序设计实验、数据结构实验两门编程实践课程，不仅具有较为系统性的C语言、常用数据结构基本知识，而且具有初步的程序设计、数据抽象与建模、问题求解与算法设计的能力，奠定了进行复杂程序设计的知识基础。但两门实验课仍属于对基本编程模型与技术的验证性训练，而基于SAT的双数独游戏求解程序的综合课程设计正是使大家从简单验证到综合应用，甚至在编程中实现智慧与风格升华的重要实践环节，为后续学习与进行计算机系统编程打下坚实的基础，让综合编程技能成为大家的固有能力与通向未来专业之门的钥匙。

**1.1.1综合性训练目标**：

基于SAT的双数独游戏求解程序中涉及C语言的主要编程要素，如典型的数据类型与控制结构；覆盖多种典型的数据结构如线性结构、二叉树与树结构、图结构及查找表结构等。从先前实验课的单要素或单一结构训练向多要素，多结构综合应用训练转变；

**1.1.2培养应用问题的求解能力：**

程序设计是为问题求解服务的，提高对应用问题进行分析，数据抽象与建模，及问题定义与功能划分等综合分析与表示能力，因此基于SAT的双数独游戏求解程序可以对解决问题的能力有进一步提升；

**1.1.3程序编写向程序设计转化：**

在实验课程中，老师基本描述了相关数据结构，程序框架及主要算法（DPLL），基于此进行程序编写训练，其属于验证与复现性编程实践。综合程序设计要求同学们基于对应用问题的分析，建立求解模型，设计数据结构与主要算法，从而进行程序设计，更多地体现“设计”的内涵与份量；

**1.1.4 进一步培养编程规范性与工程化素养：**

通过“程序设计”综合课程设计实践进一步培养良好的规范性编程习惯，以及一定的程序设计与软件开发的工程化素养，按照问题定义、必要的需求分析、系统设计、编程实现、程序测试分析及编制程序设计综合课程设计报告的流程组织本实践课程的开展与进行，形成初步的工程化程序设计素养。

**1.2国内外研究现状**

DPLL算法是一种搜索算法，思想与DFS（Depth-first search，深度优先搜索）十分相似，或者说DPLL算法本身就属于DFS的范畴，其类似于上述我们设想的“暴力”算法：搜索所有可能的赋值排列。它在1962年由马丁·戴维斯、[希拉里·普特南](https://baike.baidu.com/item/%E5%B8%8C%E6%8B%89%E9%87%8C%C2%B7%E6%99%AE%E7%89%B9%E5%8D%97?fromModule=lemma_inlink" \t "_blank)、乔治·洛吉曼和多纳·洛夫兰德共同提出，作为早期戴维斯-普特南算法的一种改进。如今已经衍生出来许多相关的算法，在解决SAT问题（布尔可满足性问题）上有着重要的理论和应用意义，这也是第一个被证明NP-完全的问题。

具体地说，算法会在公式中选择一个变量（命题变号），将其赋值为TRUE，化简赋值后的公式，如果简化的公式是可满足的（递归地判断），那么原公式也是可满足的。否则就反过来将该变量赋值为FALSE，再执行一遍递归的判定，若也不能满足，那么原公式便是不可满足的。

**1.3课程设计的主要研究工作**

首先，对设计问题进行需求分析，分析问题中所涉及的数据对象，划分功能，人机交互需求与数据文件读写等，并对问题进行形式化表示；其次，基于上述需求分析，进行系统设计，明确程序的模块结构；设计相关的数据结构（逻辑结构及其物理结构），参考并设计主要子问题的求解算法；之后进行程序实现，基于系统设计，制定相应的实现方案，编写各程序模块，完成程序编写与调试任务；最后设计测试用例对程序进行功能测试，性能测量及理论分析，并且持续进行程序优化，对设计方案中的结构，算法进行一定改进，测试与分析性能改善结果。

**2．系统需求分析与总体设计**

（黑体小2加粗居中,字母、阿拉伯数字为Time New Roman小2号加粗）

**2.1系统需求分析**

本设计要求精心设计问题中变元、文字、子句、公式等有效的物理存储结构，基于DPLL过程实现一个高效SAT求解器，对于给定的中小规模算例进行求解，输出求解结果，统计求解时间。

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。

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。

**2.1.1 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。最后对于每一个cnf文件，都要输出一个对应的res文件，里面包含算例是否满足、每个变元的取值以及程序运行所需的时间。

**2.1.2 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，可人工判断或者使用verify.exe解析功能的正确性。

**2.1.3 DPLL过程：**

基于DPLL算法框架，实现SAT算例的求解。作为程序设计的核心功能，这一部分也是最有难度以及最需要优化的部分，需要给出详细的代码以及注释来注明功能。

**2.1.4 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。通过时间来判定DPLL算法的合理性以及优秀程度。

**2.1.5 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%)

**2.1.6 SAT应用到双数独游戏中：**

将双数独游戏[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。

**2.2系统总体设计**

程序具有三个重要的功能，以及两个相关的功能：

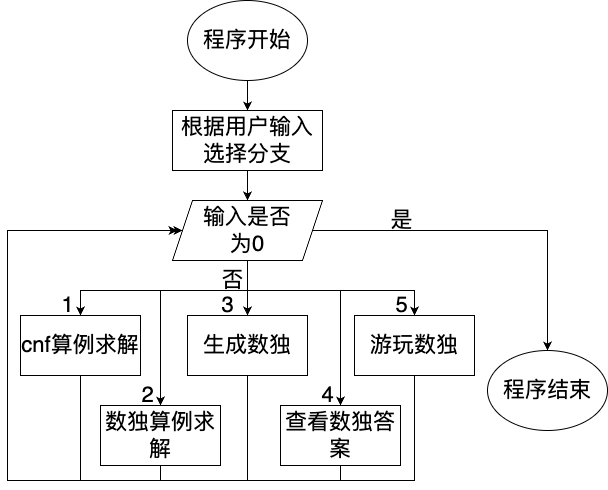


图2-1 总体流程图

**2.2.1 Cnf算例求解**

该情况是用来读取用户提供的cnf算例的，最后求出答案，如果有情况使得所有子句满足，则输出这种情况以及所需要的时间；如果没有解则输出无解；如果用时超过30s，则判定为超时，直接退出程序并输出失败。首先对算例进行单字句消除，也就是查找只含有一个变元的子句，那么这个子句当且仅当取这个变元是正确时才是正确的；如果所有子句中的某个变元有且仅有一个时，那么这个变元一定是取正确的，而且这一个子句就满足了；选择了变元的情况（取正还是取负）后，则对cnf中所有的子句进行筛查，如果有这个变元，则删去这个子句；如果有这个变元的反面，则删去这个变元。一直进行单字句消除直至不造成任何改变，之后采用分裂策略，选取一个变元删去后，重复以上的过程。如果直到所有数据都删除完成后还是有子句不满足，那么则回溯到上一个分支处重新开始，直至查找到一个解，或者所有分支都不行，也就是无解的情况。超时会自动退出所有的循环到最外层。

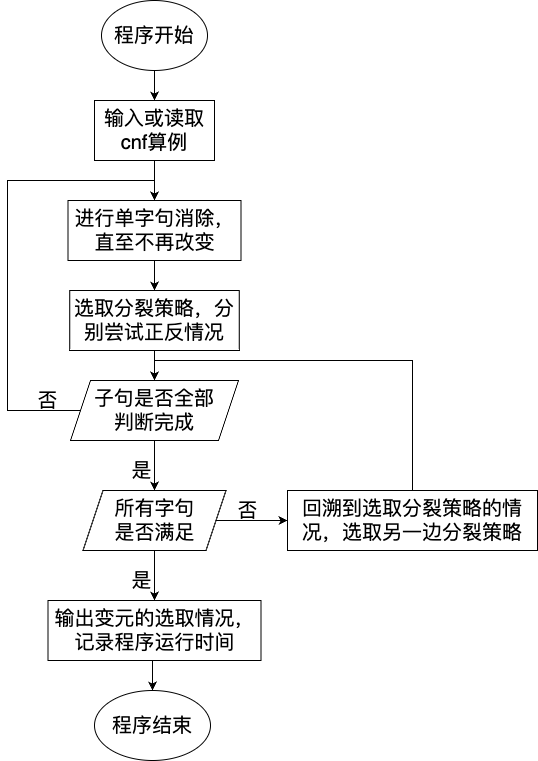


图2-2 cnf求解流程图

**2.2.2数独转化成SAT问题进行求解（其中包含生成数独、查看数独答案）**

生成数独采用类似八皇后问题的解法，也就是先把所有的1填进去，然后填入所有的2，直到所有的数字都填完；如果出现问题，回溯到填入这个位置之前的情况，换一个位置进行填入。填完第一个数独后，把第一个数独的右下角3X3位置赋给第二个数独左上角3X3位置，然后再对第二个数独进行填入数据，直至完成一个双数独。

此时采用挖洞法，随机制造出用户想要的谜题。之后将数独转化为SAT问题采用DPLL算法来求解，保证该数独有唯一解。

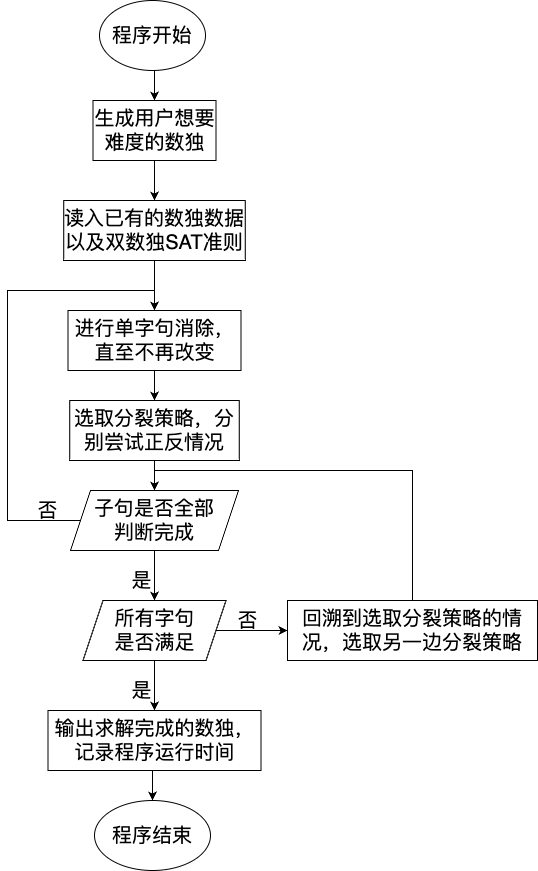


图2-3 双数独求解流程图

**2.2.3 游玩数独**

游玩数独采用用户输入的模式，传统易懂好上手。如图2-4，用户先输入是退出、查看答案还是填入数独，如果是填入数独，那么再输入填入的位置和数字，填入后系统会用数独模式验证填入是否正确，如果正确就继续游玩，并且根据用户填入的数字去修改答案（答案不唯一的情况下）；如果错误则复原到上一步，用户重新开始填入。

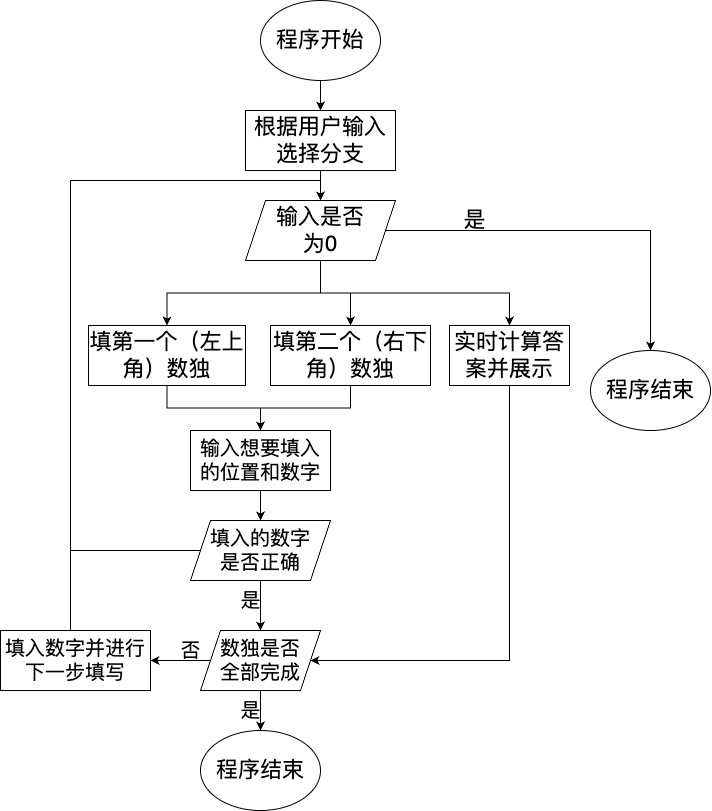


图2-4 游玩数独流程图

**3．系统详细设计**

**3.1有关数据结构的定义**

**3.1.1 系统需要处理的数据：**

系统需要处理的数据包含若干个cnf算例，形如xxx.cnf。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。

一个CNF公式也可以表示成子句集合的形式：*S*={*c*1,*c*2,...,*c*m}.

例如，由三个布尔变元a,b,c所形成的一个CNF公式（¬a∨b）∧（¬b∨c）,可用集合表示为{¬a∨b,¬b∨c}，该公式是满足的，a=0, b=0,c=1是其一组解。

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3.1是算例1.cnf文件前若干行的截图。



图3.1 cnf文件格式

**3.1.2 程序设计中使用的数据结构：**

在我的程序设计中，我采用Clausenode来存储一个子句的所有内容。该数据类型有四个指针，分别指向它的第一个数据节点、指向它的上一个子句节点、指向它的下一个子句节点、指向栈中的下一个堆栈（在没有进入堆栈前是指向空的）；

接下来，我采用datanode来存储每一个数据，也就是变元，包括正负。该数据类型具有六个指针，分别指向：包含它的子句（是唯一的）、它的下一个数据、上一个数据、在堆栈中的下一个、变元列表中的下一个、变元列表中的上一个。同时，它还包含这个变元。采用单独的节点和链表的形式来存储变元更方便删除和增添数据（回溯时需要恢复数据），拥有单独指向父亲节点的指针可以更加方便的在回溯数据时直接找到对应的子句，拥有单独变元列表的指针可以更加方便去删除数据，不需要对所有的子句中的数据一一访问，在变元子句多时可以大幅提高效率。

除此之外，我还设置了一个Flagnode作为标志节点，如果这个变元的正面或反面在这个阶段被删除了，那么将会被记录在回溯堆栈里，形式为flagnode，包含一个指向下一个flagnode的指针，同时也储存这个变元的数据。

节点说完了，接下来就该说说储存所有数据的cnfList数据类型了。该数据类型包含这个cnf算例的变元和子句数量，并且具有一个LinkList类型的数据，也就是用来储存所有子句的数据类型，实际上是clausenode的指针，指向一个已经搭建好的链表。

还有用来恢复数据的堆栈类型，clause\_stack、data\_stack、flag\_stack，都是异曲同工的，指向被删除（在该层中被单字句消去或者因分裂策略被消去的数据）数据，采用堆栈也是为了后续更快速的回溯和精确（采用唯一的指针，不易丢失）。

最后，也就是我的改进之处，采用了一个变元指针数组number\_pointers，该数组的每一个元素指向现在存在在cnfList中的该元素的一个变元（在初始化时是第一个），这个变元在哪里或者在哪个子句并不重要，因为通过这个变元指向的下一个，下一个的下一个，可以访问到所有的该变元。例如说通过1的节点，可以访问到所有的1和-1数据节点，这使得在删除一个变元的节点时变得格外的高效，不再需要访问所有的节点，大大提升了效率。同时，这个数据类型还包含这个变元分别取正面以及反面的数量（在所有子句中的），以及结果是取正还是负，可以说这个数据类型在我的程序设计中是点睛之笔。

|  |  |  |
| --- | --- | --- |
| 数据类型 | 该类型包含的数据 | 简述作用 |
| datanode | int data;  struct datanode \*nxt, \*prv, \*stack\_next, \*number\_next, \*number\_prv;  struct Clausenode \*father; | 存储一个数据 |
| Clausenode | struct Clausenode \*next\_clause, \*prv\_clause, \*stack\_next;  datanode \*next\_data; | 存储一个子句 |
| Flagnode | struct Flagnode \*next;  int data; | 储存修改的变元 |
| clause\_stack | int num;  Clausenode \*next; | 储存删除子句的栈 |
| data\_stack | int num;  datanode \*next; | 储存删除数据的栈 |
| flag\_stack | int num;  Flagnode \*next; | 储存修改元素的栈 |
| cnfList | LinkList L;  int var, num; | 储存所有数据 |
| number\_pointers | datanode \*next;  int data\_cnt\_plus, data\_cnt\_minus;  int flag; | 指向所有同类变元 |

**3.1.3 图解数据结构及之间的关系：**

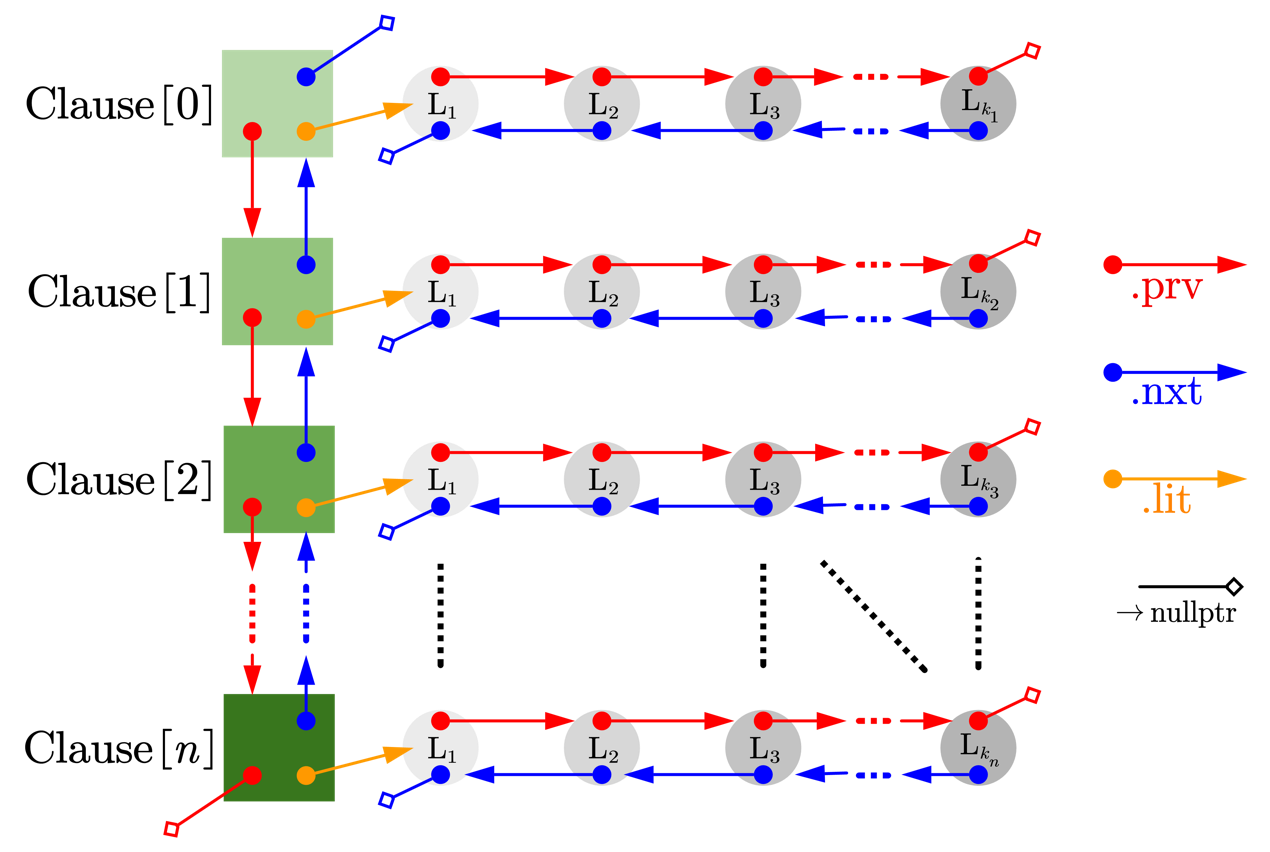


图3-2：用二维链表来存储合取范式

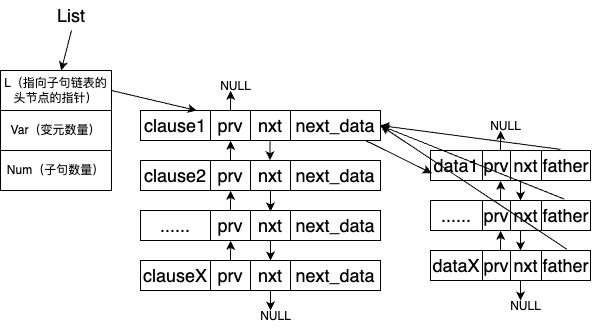


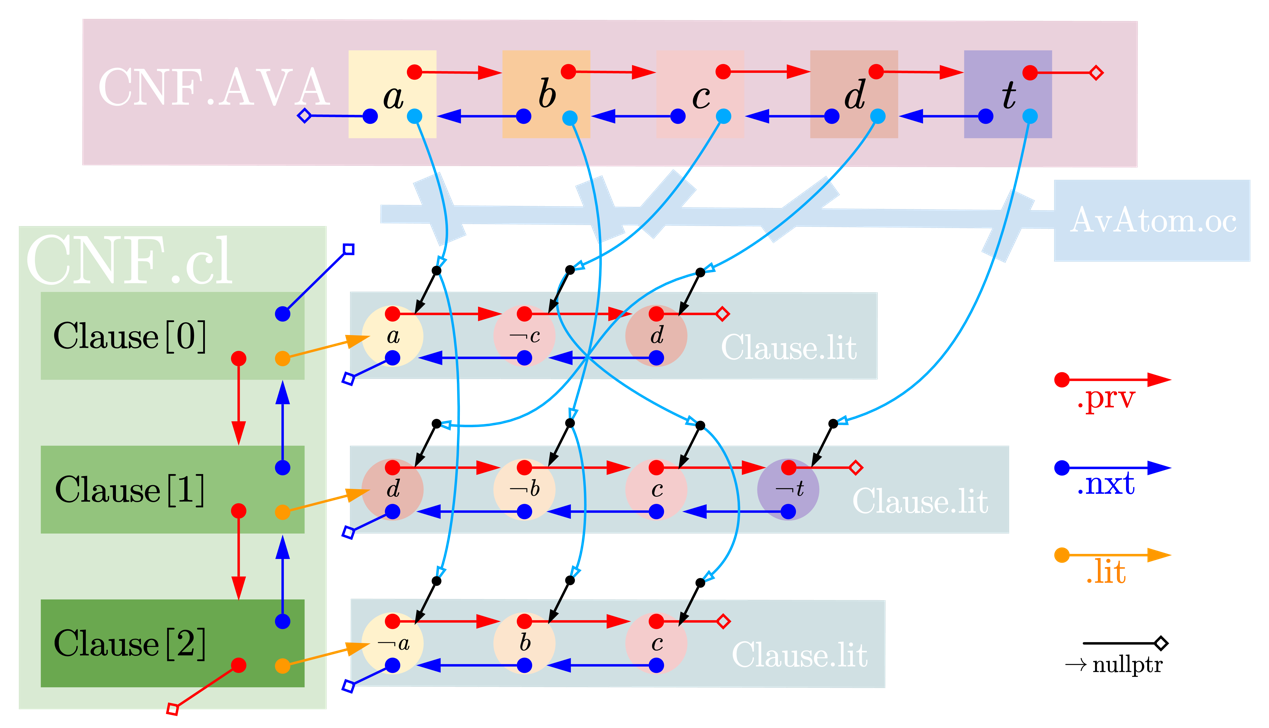
图3-3：核心储存结构

图3-4：在二维链表的基础上添加“跳线”以实现更高效的遍历

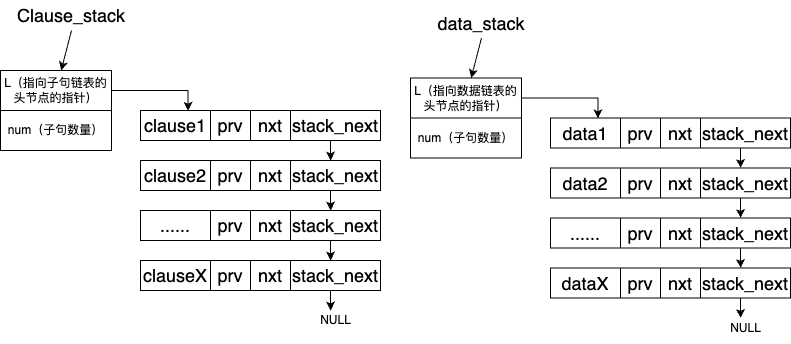


图3-5：回溯采用的堆栈的数据结构形式

**3.2 主要算法设计**

**3.2.1 cnf算例求解：**

该情况是用来解决用户提供的cnf算例的，最后求出答案有解还是无解。如果有解使得所有子句满足，则输出这组解以及所需要的时间；如果没有解则输出无解；如果用时超过30s，则判定为超时，直接退出程序并输出失败。

首先先从cnf文件中读取每一个子句中的每个变元，将子句和变元用二维链表（如图3-2）进行连接，并将number\_pointers指向这个元素，这里所有的插入都是头插法，这样子后删除的先恢复，不会破坏原有顺序。读取完所有数据后，所有子句都通过指针串起来了，子句中的变元也通过指针串起来了，同时“跳线”也“安插”在相同的变元中了。

之后，函数一直持续进行如下循环，直到找到一组解或者所有情况都不行：

1.首先对算例进行单字句消除，也就是查找只含有一个变元的子句。如果找到了，那么这个子句当且仅当取这个变元是正确时才是正确的，所以这个变元一定取这个形式，算例才能有解。之后对所有子句中的这个变元进行处理（使用DPLL\_X函数进行处理，将含有这个变元的子句添加到子句栈clause\_stack中，将这个变元反面的数据添加到数据栈data\_stack中，由于一些子句中的非该变元的数据被删去了，因此number\_pointers也要随之发生变化；最后这个变元存入标志栈Flag\_stack中），处理完之后继续进行单字句消除，直至不存在单字句，也就是单字句消除对cnfList不产生影响。

2.查找有且仅有一个的某个变元，那么这个变元一定是取正确的，而且这一个子句就满足了；因此包含这个变元的子句就可以消除掉了，之后再一直进行这个过程直至不再造成任何改变。

3.前两步完成后，就要采取分裂策略了。我的程序选取的是数量最多的变元，因为这样子可以解决最多的子句，如果出现空子句也可以更早暴露出这种情况不行。（方法来源于*Chaff: Engineering an Efficient SAT Solver*）如果选择这种情况下不存在解，那么就选择这个这个变元的反面，再进行如上的循环。

结束循环的条件：

a.找到一个解，也许不是唯一解，但是这证明这个算例是有解的，那么就跳出到最外层，输出变元的解的情况和时间。

b.如果所有分支都不行，也就是无解的情况，则输出无解以及使用的时间。c.超时会自动退出循环到最外层，这种情况不会恢复子句和数据。（但是可以制造一个副本来储存）

**3.2.2 双数独求解及游玩：**

生成数独采用类似八皇后问题的解法，也就是先把所有的1填进去，此时数独中会有九个1，然后填入所有的2，直到所有的数字都填完；填的顺序是用时间作种子的乱序，这样不仅可以避免产生的数独都一样，而且比按顺序尝试更加高效。如果出现矛盾（一行、一列、九宫格中已经存在另一个），回溯到填入这个位置之前的情况，换一个位置进行填入。填完第一个数独后，把第一个数独的右下角3X3位置赋给第二个数独左上角3X3位置，然后再对第二个数独进行填入数据，直至完成一个双数独。

此时采用挖洞法，随机制造出用户想要的谜题。这里挖洞也是采用时间作种子的乱序（随机选取两个次序进行交换，交换若干次后选择前n个，n是要挖的洞数量）。之后将数独转化为SAT问题采用DPLL算法来求解，保证该数独有唯一解，如果不存在唯一解也没有关系，因为后续答案会根据用户的填入情况进行修正（例如把标答转换到另一种情况）

游玩数独采用用户输入的模式，没有图形化的界面。用户先输入是退出、查看答案还是填入数独。如果是填入数独，那么再输入填入的位置和数字，填入后系统会用数独模式验证填入是否正确，如果正确就继续游玩，并且根据用户填入的数字去修改答案（答案不唯一的情况下）；如果错误则复原到上一步，用户重新开始填入。

**4．系统实现与测试**

**4.1系统实现**

环境：Visual Studio for Mac, 编码：UTF-8

**4.1.1：头文件的定义：**

#include <time.h>

#include <string.h>

#include <stdio.h>

#include <math.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define NUMBER 10000

typedef struct datanode

{

int data;

struct datanode \*nxt, \*prv, \*stack\_next, \*number\_next, \*number\_prv;

struct Clausenode \*father;

} datanode;

typedef struct Clausenode

{

struct Clausenode \*next\_clause, \*prv\_clause, \*stack\_next;

datanode \*next\_data;

} Clausenode, \*LinkList;

typedef struct Flagnode

{

struct Flagnode \*next;

int data;

} Flagnode;

typedef struct clause\_stack

{

int num;

Clausenode \*next;

} clause\_stack;

typedef struct data\_stack

{

int num;

datanode \*next;

} data\_stack;

typedef struct flag\_stack

{

int num;

Flagnode \*next;

} flag\_stack;

typedef struct

{

LinkList L;

int var, num;

} cnfList;

typedef struct

{

datanode \*next;

int data\_cnt\_plus, data\_cnt\_minus;

int flag;

} number\_pointers;

**4.1.2 函数及函数功能的说明：**

**在cnf\_writter.c中的函数：**

1.void cnfList\_Mode()：不接收输入，但是会读取用户输入的名称的文件，之后使用read.c读取内容，用cnfparser.c中的函数来进行cnf算例求解。函数不返回值，但是会输出是否有解，解的情况以及总共的时间。

2.int sudoku\_Mode(int sudo[9][9], int sudo2[9][9], int hole[9][9], int hole2[9][9])：

函数接受两个数独以及它们挖洞的情况，求解出数独的解。如果数独有解，则返回1，无解则返回0，超时返回-1。

在read.c中的函数：

1.int Init(cnfList \*List)：创建一个带有头节点的L、其余变量一致置空或者为0的cnfList。

2.void ReadinSAT(LinkList \*L, int \*nums, number\_pointers \*\*np)：输入已经创建好的Linklist，需要存到L中的数据，以及指向数据的np。功能为从cnf算例中读取子句和数据，需要调用Read\_data函数，mode为0。

3.void ReadinSudoku(LinkList \*L, int \*nums, number\_pointers \*\*np)：输入已经创建好的Linklist，需要存到L中的数据，以及指向数据的np。从双数独的标准cnf算例中读取子句和数据，需要调用Read\_data函数和Read\_Sudoku函数，mode为1。

4.void Read\_Sudoku(cnfList \*List, number\_pointers \*\*np, int sudo[9][9], int sudo2[9][9], int hole[9][9], int hole2[9][9])：输入为需要加入数据的cnfList和np以及数独和挖洞的情况。功能为把数独中已经填好的数字转化为SAT问题，用来后续求解

5.void Read\_data(cnfList \*L, char \*filename, number\_pointers \*\*np, int mode)：

该函数用来读取名称为 filename 中的cnf算例，将数据存储到L和np中。Mode的取值是关于cnf和数独的，0为cnf模式，1为数独模式

6.void PrintOut(LinkList L)：该函数用来输出所有储存到L中的子句中的数据，逐个子句进行输出，用来一览cnf算例。

在cnfparser.c中的函数：

1.int DPLL(cnfList \*L, int x, number\_pointers \*\*np, clock\_t start)：该函数是程序设计中最重要的函数之一，输入为一个已经读入了数据的cnfList，需要进行处理的x，指针数组np，以及开始的时间start。输出为True或False以及INFEASIBLE，分别表示有解和无解的情况。进入到该函数后，首先进行单字句消除，之后选择一个出现次数最多的数据进行分裂策略处理，也就是到下一层递归。如果超时了，返回INFEASIBLE，退出所有的循环递归。

2.int num\_detect(cnfList \*List, number\_pointers \*\*np)：该函数用来检查是否有数字出现了一次，也就是检测有没有单字句出现的情况。输入为cnfList和np，输出为检查到为单字句的变元，如果没有则输出0。

3.int final\_detect(cnfList \*List, number\_pointers \*\*np)：该函数用来检查是否所有变元都检查完成。输入为cnfList和np，如果所有变元都检查完成就输出1，否则输出0；

4.int List\_detect(cnfList \*List)：该函数用来检查是否有存在单字句的情况。输入为cnfList和np。如果有，则输出1，表示存在单字句，否则输出0，表示单字句消除已经完成，可以进行下一步了。

5.int choose\_number(cnfList \*List, number\_pointers \*\*np)：该函数作用为选取一个接下来进行DPLL算法的变元，输入为cnfList和np，输出为当前剩余的变元里数量最多的。这里是区分正反的，也就是如果正面情况多，就会返回变元的，先进行一次变元正面的消除，如果不成立再去消除反面，反之同理。

6.void DPLL\_X(cnfList \*L, int detect\_num, clause\_stack \*CS, data\_stack \*DS, flag\_stack \*FS, number\_pointers \*\*np)：另一个最重要的函数之一，作用是消除所有含有detect\_num的子句，以及detect\_num反面的数据节点，同时将这个数字存入flag\_stack，把所有删除的子句存入clause\_stack，删除的数据存入data\_stack，把删去的节点从cnfList和np中除去，也就是说之后在这个基础上的递归不会再收到删除数据的影响。

7.void recover\_datas(cnfList \*L, data\_stack \*DS, number\_pointers \*\*np)：用来恢复删除的数据节点，通过数据节点的前后以及父亲节点，可以把删去的节点重新插入回去。同时，恢复的数据也会重新出现在np中，方便后续的处理，也保证了数据完全恢复到了删除前的样子。恢复的原因是在这个情况下的所有分支都走不通，或者已经出现了空子句（不满足的子句），因此要回到上一层，所以把删除的所有节点和子句以及改变的标志全部加回来。输入为一个需要恢复的cnfList，一个储存了在这一层被删除了的数据节点的data\_stack，以及需要恢复的np。

8.void recover\_clauses(cnfList \*L, clause\_stack \*CS, number\_pointers \*\*np)：同上，用来恢复所有被删去的子句。输入为一个需要恢复的cnfList，一个储存了在这一层被删除了的子句节点的clause\_stack，以及需要恢复的np。

9.void recover\_flags(flag\_stack \*FS, number\_pointers \*\*np)：同上，用来恢复所有在这一层标志了的变元，恢复为没有标志的样子（从1或-1变成0）。输入为一个需要恢复的cnfList，一个储存了在这一层被改变了的标志节点的flag\_stack，以及需要恢复的np。

10.void eliminate\_datas(cnfList \*L, clause\_stack \*CS, data\_stack \*DS, flag\_stack \*FS, number\_pointers \*\*np)：该函数用来消除所有总数量为1的所有变元，搭配num\_detect进行使用。输入为待检查的cnfList，存储被删去子句的clause\_stack，存储被删去数据的data\_stack，存储被修改的flag\_stack，以及待更改的np。该函数会不断循环，直到找不到一个总数量为1的变元或者说cnfList不发生改变为止。

11.void eliminate\_clauses(cnfList \*L, clause\_stack \*CS, data\_stack \*DS, flag\_stack \*FS, number\_pointers \*\*np)：该函数用来消除所有变元量为1的所有子句，并且把这个变元标志为真，使用DPLL\_X进行消除搭配list\_detect进行使用。输入为待检查的cnfList，存储被删去子句的clause\_stack，存储被删去数据data\_stack，存储被修改的flag\_stack，以及待更改的np。

在sudoku.c中的函数：

1.void draw(cnfList \*L, number\_pointers \*\*np):该函数接收一个cnfList和一个np，作用是把双数独“画出来”，也就是把数据可视化，而且这个画的是用sudoku\_Mode计算出来的，不需要知道原始生成的数独。

2.bool set(int sudo[9][9], int x, int y, int val)：该函数接收一个9X9的数独，x、y坐标（x为行，y为列），以及要填入的数值val。如果这个地方填入val会有矛盾，那么函数返回0，表示无法填入；否则返回1，表示填入成功。

3.void initXOrd(int \*xOrd)：该函数接收一个9个元素的顺序数组，从1-9，修改成一个顺序打乱的数组，作用是使顺序数组被打乱。

4.bool fillForm(int sudo[9][9], int x, int val)：递归函数，接收一个9X9的数独，x坐标（行标），需要填入的值，返回True或False，表示填入成功和失败。如果成功填入，则继续填下一行的；如果已经填到第九行，那么就从第一行重新开始填val+1的数，一直到全部填入完成或者所有的填入方案都失败为止。类似八皇后问题的递归解法，但是要填入9X9个数。

5.void digHole(int sudo[9][9], int sudo2[9][9], int hole[9][9], int hole2[9][9], int holeCnt)：函数作用是随机选取holeCnt个数的洞，如果挖了洞，相应的hole的值也要发生改变，0为已填入，1为待填入。随机选取采用的是交换次序的策略，最后选择前holeCnt个进行挖洞。

6.void printTwodoku(int sudo[9][9], int sudo2[9][9])：函数功能为输出双数独，接收两个已经填入完成的数独。

7.void printTwodoku\_holes(int sudo[9][9], int sudo2[9][9], int hole[9][9], int hole2[9][9])：函数功能是输出双数独的谜题，接收两个已经填入的数独，以及两个数独挖洞的情况。

**4.2系统测试**

操作系统界面：如图4-1

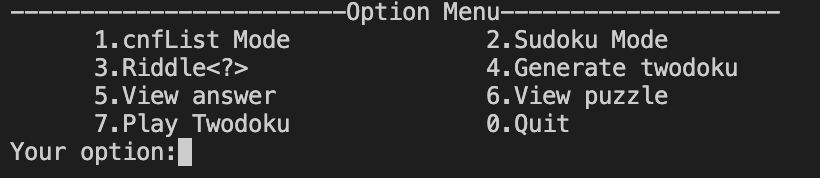


图4-1 操作系统界面

**4.2.1:cnfList Mode：**

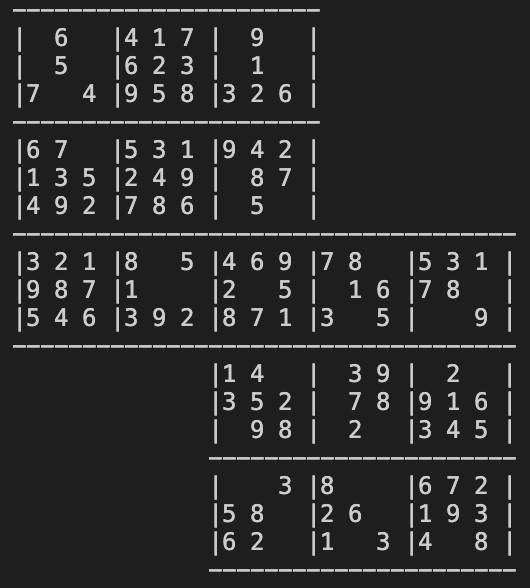
功能：输入cnf算例的文件名称（.cnf），输出一个.res文件，其中包含是否有解，解的具体情况，所花费的时间。

测试数据：这里选取了两个算例，都是当时验收时使用的算例，一个有解一个无解。经过验证，程序解得的答案均正确。

|  |  |  |
| --- | --- | --- |
| 算例名称 | 算例基础情况截图 | 算例运行结果截图 |
| Ais10.cnf |  |  |
| Example.Cnf |  |  |

**4.2.2:Sudoku Mode：**

功能：输入一个填了一部分的数独，将其转化为SAT问题用DPLL求解，之后把解出来的数独打印出来，同时输出一个.res文件，其中包含是否有解，解的具体情况，所花费的时间。

测试数据：由系统随机生成的一个具有40个洞的双数独谜题，经过验证，程序解得的答案均正确。

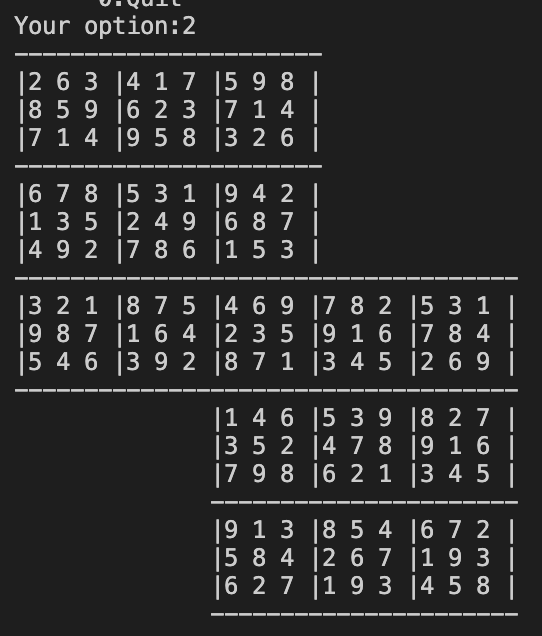
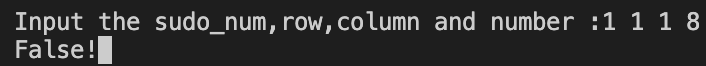
图4-3 双数独谜题

图4-4 双数独谜题答案

**4.2.3:游玩双数独：**

功能：游玩数独采用用户输入的模式，没有图形化的界面。用户先输入是退出、查看答案还是填入数独。如果是填入数独，那么再输入填入的位置和数字，填入后系统会用数独模式验证填入是否正确，如果正确就继续游玩，并且根据用户填入的数字去修改答案（答案不唯一的情况下）；如果错误则复原到上一步，用户重新开始填入。

 测试结果：如图4-5、图4-6和图4-7

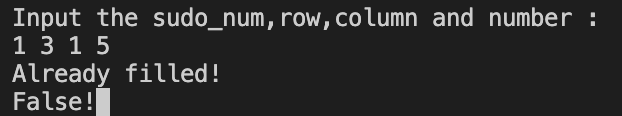
图4-5：输入错误的答案

图4-6：填入已存在数字的位置

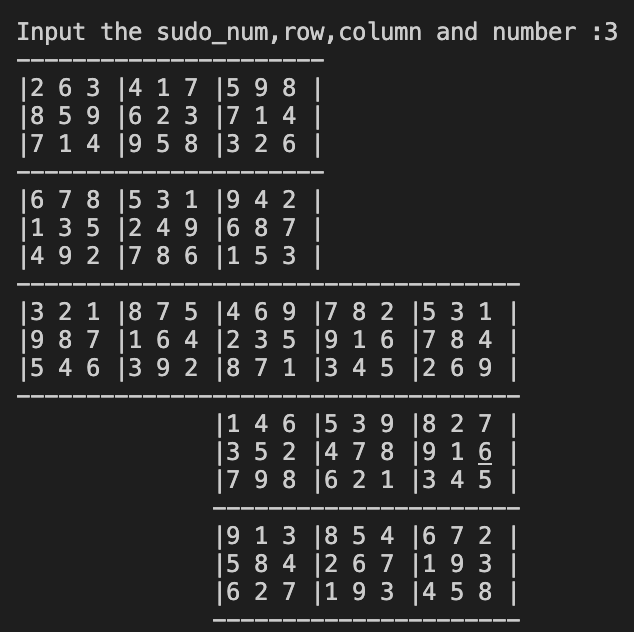
图4-7：填入正确的答案

图4-8：输入3以查看正确答案

**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）独立完成了课程设计所有的代码编写，在有限的课时中展现了高效的代码技能，迅速完成了课程设计作业。实际上，现在提交的版本已经是第四代代码了：第一代由于采用数组，删除和增加较为麻烦，所以后面重新改写了数据结构（基本上等于重做）；第二代和现在代码已经比较接近了，采用的是二维链表来存储数据，比一代已经快了许多了，但是由于没有构建合适的回溯系统，所以基本上就是个半成品，每次递归都要先创建一个副本，非常麻烦且占空间；第三代采用了堆栈的形式来存储被删除的子句和数据，可以说大大提高了回溯的效率，使得轻松回溯到修改前，且需要的空间大幅减少；第四代加入了“跳针”，也就是指向某个变元的一串链表，这使得在删除某个特定数据时的速度有了质的飞跃，不用再去检索所有的子句中的数据，只需要跟着跳针一路向下即可，当然，这给回溯也增加了额外的工作量，但是跳针实际上的意义就是把数据串起来，所以使用头插法即可。

（2）独立完成了数独项目的所有代码编写。说实话，很多人基本上都是使用网上的数据进行数独，这样虽然省了很多功夫，但是没有了随机性也就没有了乐趣。开始编写数独项目时已经是第二周第二节课了，本来以为会很难，但是一节课下来也做了个林林总总，回去再改了一下就完全可以用了，可以说确实是有点赶的。现在数独项目可以随机生成单数独，双数独，而且可以游玩，为此还是挺欣慰的。

（3）可以说，这次的程序设计既然选择了c语言（或者c++），就注定了这会是一场指针的盛宴。在我编写的程序中，指针基本上无处不在，以及一些重要的数据结构例如栈，树，队列这些也有所体现，甚至还使用到了指针数组。所有的指针都是动态分配的，因此我也学到了许多关于动态分配的操作，还是很有用的，我相信在未来可能c语言会用的少了，但是指针的特性会一直铭记在我脑海，真的太深刻了。关于递归的使用也学到了许多，收获颇丰。

**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）继续关注SAT问题和DPLL算法相关领域的知识，继续学习充实自己，确实是比较有意思的，而且在生活中用处也不小，可以继续深入学习。会关注相关领域的学术成果，像之前看的那篇文章已经是几年前的了，现在选择变元的策略都已经基本上很完善了，只要能写出来就可以实现。这次课程设计也算是把论文中的伪代码第一次翻译成了c语言代码（我看有用python就两三百行，我写了快一千多行了），算是比之前数据结构更大的一次尝试吧，学到了许多多文件管理相关的知识，未来也会不断精进自己。

（2）会继续优化代码，给出更多详细的注释，提高算法的效率，降低操作的门槛，努努力把图形化游玩数独界面做出来，但是据说是要用到c++的代码，如果有空倒是可以再用c++或者python把这个代码重构一遍，删去一些冗余的代码，优化一下数据结构。程序设计这种东西永远没有最优解，每次当改进完一个人版本测试之后，又会有新的可以改进的地方，可谓是学海无涯。

# 6 体会

两周的程序设计课就这么愉快的过去了，总共在代码上耕耘了37个小时，这还不算上网查阅资料、看论文和测试数独的时间，加起来估计都40小时往上了。再加上这份报告，花费在课程设计这门课的时间估计要有50小时了。这两周可以说是前所未有的充实体验，除了周末是我个人习惯尽量不会碰学习方面的事情，工作日基本上是每天拉满满，起床就是代码代码，想办法怎么优化，和同学一起讨论，怎么做可以更完善更好，写完代码就开始debug。说实话，写代码的时间可能都没有debug时间长，特别是后面数据结构越来越错综复杂，找到问题就越来越难，一个程序上的缺陷可能要花费几十分钟才能找出来，找出来也许长舒一口气，也或许兴奋的起来打猴拳。这种经历，以后还会有，但是这么高强度的写代码可能不会再有了。

在项目开始阶段，还是有些闲庭信步的，觉得两周时间怎么可能做不完一个这么小的项目，但是随着项目的进行，功能和分支越来越多，时间就像是偷偷溜走了一样抓也抓不住。作为一个颇具个人主义的人，我是有点心气的，既然决定了一个人做完，就一定不会摸鱼或者套用别人的代码。虽然有些人套着别人的代码早早就验收完成了，也谈不上羡慕，就是挺不齿的吧。

有些感悟倒是想着重谈谈，就是做什么事情都应该先宏观的考虑一遍。体现在这次程序设计上就是这个数据结构的修改，来来去去改了几遍数据结构，如果一开始就考虑到了时间复杂度或者空间复杂度，在开始写的时候就会有注意这个问题，会采用别的数据结构。错误的数据结构使我浪费了最开始的两天时间，忙活着和我的“数组”们斗智斗勇。后面更新“跳针”又花费了我将近两天（不是两节课，那两天基本上都在改bug(◐‿◑)﻿），改到最后还剩两天的时间，数独功能还一点都没做，同学都已经验收完成了，我还在搞基础功能的代码，当时是有点着急的，结果一天下来就把全部搞定了，只能说当时状态确实不错，一心扑在程序设计上，一下子就搞定了。

最后感谢一下程序设计的老师，她也帮助了我挺多的，最后验收的时候也非常有耐心，平时相处下来也能感受到她是个很有意思的人，感谢！

**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>

Twodoku： https://en.grandgames.net/multisudoku/twodoku

[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] 薛源海，蒋彪彬，李永卓. 基于“挖洞”思想的数独游戏生成算法. 数学的实践与认识,2009,39(21):1-7

[11] 黄祖贤. 数独游戏的问题生成及求解算法优化. 安徽工业大学学报(自然科学版), 2015,32(2):187-191

**附录**

# 源码在电子版课程设计实验报告中。