华中科技大学

课程设计报告

题目：基于 SAT 的蜂窝数独游戏求解程序

课程名称：程序设计综合课程设计

专业班级：----------

学　　号：

姓　　名：

指导教师：

报告日期：2023 年 9 月 1 日

计算机科学与技术学院

任务书

□ 设计内容

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

□ 设计要求

要求具有如下功能：

（1）输入输出功能：包括程序执行参数的输入，SAT 算例 cnf 文件的读取，执行结果的输出与文件保存等。(15%)

（2）公式解析与验证：读取 cnf 算例文件，解析文件，基于一定的物理结构，建立公式的内部表示；并实现对解析正确性的验证功能，即遍历内部结构逐行输出与显示每个子句，与输入算例对比可人工判断解析功能的正确性。数据结构的设计可参考文献[1-3]。(15%)

（3）DPLL 过程：基于 DPLL 算法框架，实现 SAT 算例的求解。(35%)

（4）时间性能的测量：基于相应的时间处理函数（参考 time.h），记录 DPLL 过程执行时间（以毫秒为单位），并作为输出信息的一部分。(5%)

（5）程序优化：对基本 DPLL 的实现进行存储结构、分支变元选取策略[1-3]等某一方面进行优化设计与实现，提供较明确的性能优化率结果。优化率的计算公式为：$[(t-t_o)/t]*100\%$,其中 t 为未对 DPLL 优化时求解基准算例的执行时间，t_o 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%)

口 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

目录

1 引言

1.1 课题背景与意义

通过对 C 语言和数据结构的学习，我们已经对运用高级语言设计和编写程序有了深刻的认识，并且已经具备建立结构解决复杂问题的能力。该课题“基于 SAT 的数独游戏求解程序”是一个大型复杂的问题，要求我们自己从建立数据结构开始，通过自己的结构实现 DPLL 算法，从而求解 SAT 问题，同时也要求我们在解决 SAT 问题的基础上，将其转化为数独游戏求解。该课题综合性十分强，考察了我们对于数据结构的认识与运用，对于程序和算法的实现，然后进一步培养了我们对于问题转化实现实际问题的能力。这个课程设计是对我们学习 C 语言和数据结构课程的一次很好的实践与检验，也培养我们的计算机编程思维。

1.2 国内外研究现状

近十多年来，可满足性问题研究逐渐升温，已成为了国际国内的研究热点，取得了一批相当重要的理论和实践成果，应该说当前的 SAT 问题研究比十多年前已取得了很大的突破，并直接或间接地推动了其他相关领域（比如形式验证，人工智能等领域）的发展。

国际上已提出了各种不同的局部搜索算法和回溯搜索算法，使得 SAT 解决器解决不同领域中的 SAT 问题的能力不断增强，能解决的问题的规模不断增大。其中局部搜索算法显示出对于随机的 SAT 问题特别有用，而回溯搜索算法则 被用来解决大规模实际应用领域中的 SAT 问题。事实上，国际上已提出了一大批 采用回溯搜索算法的高效的 SAT 问题解决器，其中绝大多数提出来的回溯搜索算法是对原始的 DPLL 回溯搜索算法的改进算法。这些改进措施包括:新的 变量决策策略，新的搜索空间剪除技术，新的推理和回溯技术以及新的更快的算 法实现方案和数据结构等。当前水平的 SAT 问题求解器已能够轻松解决以前传统 SAT 问题解决器完全无法解决的可满足性问题。

尽管当前的 SAT 问题解决器已取得了相当重要的进步，但是研究的脚步不会停止，我们还可以提出一些值得研究的问题。比如，是否存在新的更高效的 SAT

问题处理技术可以集成到 DPLL 算法框架内；是否可以找到除局部搜索，回溯搜索之外的其他 SAT 算法来更有效地解决 SAT 问题;是否能提出更好的 SAT 改进算法和实现方案。

1.3 课程设计的主要研究工作

本次实验中，实验者选择了 “基于 SAT 的数独游戏求解程序”作为实验课题，实现 SAT 求解器和数独游戏两个功能。

SAT 求解器基于 DPLL 的完备算法，对 CNF 范式算例文件进行求解，输出答案，并可选择遍历验证答案或将答案存入文件；数独游戏可转化为 SAT 问题，用本系统实现的 SAT 求解器可以快捷地对数独问题转化的 CNF 文件进行求解，再以变元真值数据转化的数独盘格式输出求解答案。本系统具有一定的交互功能，用户可以利用本系统进行数独游戏，系统将自动判断解的正确性，并输出正确答案。

2 系统需求分析与总体设计

2.1 系统需求分析

本系统主要实现了解析 cnf 文件以及求解其中的 SAT 问题，在此基础上设计并求解了数独游戏，实现了一定的交互性和界面美化。首先需要定义适用于 SAT 问题的存储结构，包括变元、文字、子句和公式的有效表示。我们基于 DPLL 算法制造求解器，解决中小型规模的 SAT 问题。对于蜂窝数独功能，需要以合适的方式将蜂窝数独求解问题规约成 SAT 问题，生成 cnf 文件，再使用 SAT 求解器解决，解决后解析成蜂窝数独的结果。

SAT 问题是一个关于布尔逻辑的问题，通常表示为一组布尔变量和一个布尔表达式，问题的目标是找到一组布尔变量的赋值，使得表达式成为真（可满足）。如果存在这样的赋值，那么问题的答案是"可满足"，否则是"不可满足"。

DPLL 算法是一种用于解决 SAT 问题的递归回溯算法，它的名字来源于其发明者 Davis, Putnam, Logemann 和 Loveland。DPLL 算法的核心思想是不断地选择一个未赋值的变量，尝试将其设置为真和假，并根据结果来缩小搜索空间。算法具体步骤如下：

① 如果所有变量都被赋值了，检查表达式是否为真。如果是真的，返回可满足；否则，返回不可满足。

② 如果存在一个单子句（只包含一个变量的子句），则强制该变量为真，然后删除所有包含该变量的子句。

③ 如果存在一个纯文字（只在表达式中出现一次，无论是正还是反），则将其设置为真，并删除所有包含该文字的子句。

④ 选择一个未赋值的变量，尝试将其设置为真或假，然后递归调用 DPLL 算法。

⑤ 如果在递归调用中找到了可满足的解决方案，则返回可满足；如果递归调用中找不到解决方案，则返回不可满足。

DPLL 算法反复应用这些步骤，直到找到可满足的解决方案或确定问题是不可满足的。它是一种非确定性指数时间算法，通常在实践中表现良好，特别是对于具有大量变量和子句的 SAT 问题。

以下简要介绍下本系统的功能。

2.1.1 SAT 求解器

SAT 求解器可对 CNF 范式算例文件进行求解，输出求解答案及求解所用时间，求解结果存入同名文件（文件拓展名为.res），并可选择逐行输出 CNF 范式中子句及其真值从而验证求解结果的正确性。

2.1.2 数独游戏

数独部分包括生成可玩的数独游戏及求解数独 CNF 范式算例文件两部分。数独游戏板块输出未填充部分数字的数独游戏盘，并给予操作提示，用户可输入自己求解的答案，系统将检验其答案的正确性，答案错误时可选择直接查看正确答案或继续解题。数独文件求解后系统将以数独终盘的形式输出求解结果。

系统通过菜单显示操作类别，用户可自行选择要进行的操作。

2.2 系统总体设计

SAT 求解器求解 CNF 范式算例文件，实现该功能包括 4 个部分：

①CNF 范式算例文件解析功能模块，建立其对应的数据存储结构；

②基于 DPLL 算法的 SAT 求解模块，对文件算例进行求解；

该部分模块的功能实现需进行对 CNF 范式的删除子句，删除文字，判断单子句等操作；

③得到解后可选择进入解的验证功能模块，遍历整个 CNF 范式，通过验证每个子句的真值情况，验证求解结果正确性；

④进入文件保存模块，将求得的结果保存入 CNF 算例同名文件（拓展名为.res）中。

2.1.1 cnf 算例求解

求解 cnf 算例的算法的关键是通过不断选择未赋值的变量，并在每一步中分

支出两种可能的赋值情况，递归地搜索解空间。如果找到可满足性解决方案，算法返回该解决方案。如果没有可满足性解决方案，算法回溯到之前的状态并尝试其他赋值，直到所有可能的情况都被探索。如果仍然找不到可满足性解决方案，则返回不可满足。

① 初始化：

将 CNF 表达式转化为一个包含多个子句的集合。每个子句是一个包含多个布尔变量的合取子句。

初始化一个变量赋值集合，最初为空。

② 检查终止条件：

检查是否存在一个子句为空的情况。如果存在这样的子句，说明当前的变量赋值集合无法满足所有子句，因此返回不可满足。

③ 检查完成条件：

检查是否所有子句都已满足。如果是，返回当前的变量赋值集合，表示找到了一个可满足的解决方案。

④ 选择未赋值的变量：

从剩余的未赋值变量中选择一个变量。这可以通过不同的策略来实现，例如随机选择或者启发式方法。

⑤ 尝试赋值：

尝试将选定的变量分别设置为真和假两种情况，并复制当前的变量赋值集合以进行尝试。

⑥ 递归调用：

对于两种赋值情况，递归调用 DPLL 算法。这将导致两个分支，其中一个分支假设选定变量为真，另一个分支假设为假。

。

⑦ 检查递归调用结果：

对于每个递归调用，检查返回的结果。

如果有一个分支返回可满足性，那么整个算法返回可满足性并传递可满足的变量赋值集合。

如果两个分支都返回不可满足性，那么回溯到上一层并尝试其他变量赋值。

⑧ 重复：

重复步骤2到步骤7，直到找到可满足的解决方案或确定问题是不可满足的。

cnf.h文件内定义了CNF算例文件的基本操作，具体函数定义如下：

（1）创建CNF范式邻接链表及变元表：

```
status CreateParadigm(FILE **fp);
```

（2）创建子句链表及文字链表：

```
int CreateClause(FILE **fp,Clause **sentence,Paradigm *ClausHead,int first);
```

（3）销毁所有链表及线性表结构

```
status DestroyParadigm(Root *r);
```

（4）判断CNF范式中是否还含有单子句：

```
int HasUnitClause(Root *r);
```

（5）判断CNF范式中是否还含有单子句（优化前版本）：

```
Clause * HasUnitClause_Before(Root *r);
```

（6）判断指针c指向的子句链表是否为单子句链表：

```
status isUnitClause(Paradigm *c);//
```

（7）在整个CNF公式中取一个文字：

```
int FindLiteral(Root *r);//取每次DPLL处理后公式中出现频率最大的文字
Clause * FindLiteral_Before(Root *r);//优化前版本
```

（8）删除出现了文字l的所有单子句：

```
status DeleteClause(Root *r,int l);
```

（9）在CNF范式邻接链表表头添加只含有文字l的单子句链表

```
status AddClause(Root *r,int l);
```

（10)删除 CNF 范式邻接链表中从表头开始第一个只含有文字 l 的单子句链表：

```
status RemoveHeadClaus(Root *r,int l);
```

（11）删除所有文字为-l 的子句链表结点

```
status DeleteLiteral(Root *r,int l);
```

（12）恢复认为文字 l 为真时对 CNF 范式邻接链表所作的操作

```
status RecoverCNF(Root *r,int l);
```

（13）遍历 CNF 范式邻接链表

```
void ParadigmTrabverse(Root *r);
```

(14) 保存 CNF 范式的解及求解时间信息：

```
status SaveValue(ArgueValue *ValueList,int solut,int time);
```

（15）处理读取的文件中变元出现次数信息，决策 DPLL 过程中分裂策略的变元选取策略：

```
int OccurTimeCount(void);
```

2.1.2 DPLL 文件

DPLL.h 文件内定义了 SAT 求解器功能模块的主函数，输出 SAT 求解器功能模块的副菜单；并定义了核心求解算法 DPLL 函数，函数内调用了 cnf.h 文件中定义的 CNF 算例文件基本操作函数。具体函数定义如下：

（1） SAT求解器功能模块主函数：

```
status SAT(void);
```

（2） 采用优化后的变元选取策略的递归算法DPLL函数：

```
status DPLL(int num,int op,int times);
```

（3） 优化前版本DPLL：

```
status DPLL_Before(Root *r,int op);
```

（4） 完善SAT求解结果：

```
void AnswerComplete(void);
```

（5） 检查SAT求解结果正确性：

```
status AnswerCheck(int solut);
```

2.1.3 hanidoku 文件

使用挖洞法生成蜂窝数独初盘的原理：

① 生成完整数独终盘：

首先，生成一个已知的完整数独终盘。这是一个填满了数字的数独，满足蜂窝数独规则。

② 挖洞：

选择要挖洞的格子。通常，从终盘中随机选择一些格子来挖洞。挖洞的数量和位置可以根据难度级别和生成数独初盘的要求来调整。

将选定的格子中的数字从终盘中去掉，使这些格子变为空白。

③ 检验唯一解性：

每次挖洞后，都要确保生成的数独初盘仍然具有唯一解。这是为了确保数独初盘的解决方案是唯一的，符合数独的要求。

为了检验唯一解性，通常采用一种求解数独的算法，例如回溯法。尝试填充被挖空的格子，看是否有唯一的解。如果有多个解或无解，那么重新挖洞或调整挖洞的位置。

④ 重复挖洞和检验：

重复步骤 2 和步骤 3，直到达到所需的挖洞数量或达到了所需的数独初盘难度级别。

⑤ 生成数独初盘：

当挖洞和检验完成后，你将得到一个具有唯一解的数独初盘，其中一些格子为空白，等待玩家填入数字。

这样，通过不断挖洞并确保数独初盘仍然有唯一解，你就可以生成一个合法的数独初盘，可以作为数独游戏的起始状态。这个过程可以根据挖洞的数量和位置来控制数独初盘的难度，从容易到困难不等。

hanidoku 文件内定义了数独功能模块的主函数以及其全部功能函数，函数内调用了 DPLL.h 文件内的 SAT 求解器相关函数以及 cnf.h 文件内的 CNF 算例文件处理函数。具体函数声明如下：

(1) 蜂窝数独功能模块主函数

```
void hanidoku();
```

(2) 蜂窝数独自然编码转语义编码

```
int toSemanticEncoding(int i, int j, int n, int* L);
```

(3) 蜂窝数独语义编码转自然编码

```
int TransLiteral(int x);
```

(4) 蜂窝数独语义编码转自然编码

```
void toDirectEncoding(int n, int* row, int* col, int* number);
```

(5) 创建蜂窝数独完整盘

```
void createHanidoku(int a[], char* filename);
```

(6) 挖洞法创建蜂窝数独初始盘

```
void createStartinggrid(int a[], int b[], int holes);
```

打印蜂窝数独

(7) `void print(const int a[]);`

翻转蜂窝数独用于编写对角线约束

(8) `void turnAnAngle(numNode a[][10], numNode b[][10]);`

蜂窝数独二维数组转一维数组

(9) `void createNode(numNode b[][10]);`

(10) 行约束

```
void lineDis(FILE* in, numNode b[][10]);
```

(11) 规约成 cnf 文件

```
void ToCnf(int a[][9], int holes, char* cnf_filename);
```

(12) 一维数组转二维数组

```
void changeGrid(int hani[][9], int b[]);
```

(13) 二维数组转一维数组

```
void to_1d(int hani[9][9], int b[]);
```

(14) 每格只填一个数字的约束

```
void Singularity(FILE* in);
```

(15) 用户解决蜂窝数独主函数

```
void usersolve(int origin[9][9], int correctarr[9][9], int
```

correct1d[61]);

(16) 用户输入答案

```
void inputans(int a[9][9]);
```

(17) 根据 cnf 文件解决蜂窝数独问题

```
status solvehanidoku(char* cnf_filename, int hani[9][9]);
```

数独游戏模块包括 2 个功能部分：

①生成可玩的数独游戏：

利用 SAT 求解器生成随机的数独终盘，再基于挖洞法生成有唯一解的数独游戏盘，输出至用户界面，读取用户输入的答案，与已生成的终盘解对比，判断用户解答是否正确，如果错误，可选择继续解答或直接查看答案；

②求解数独 CNF 范式算例文件：

读取文件中备注部分的数独残局信息，建立变元与数独格取值的对应关系，再利用 SAT 求解器求解数独文件，将解得的真值信息转换为二维数组并以数独盘的格式输出。

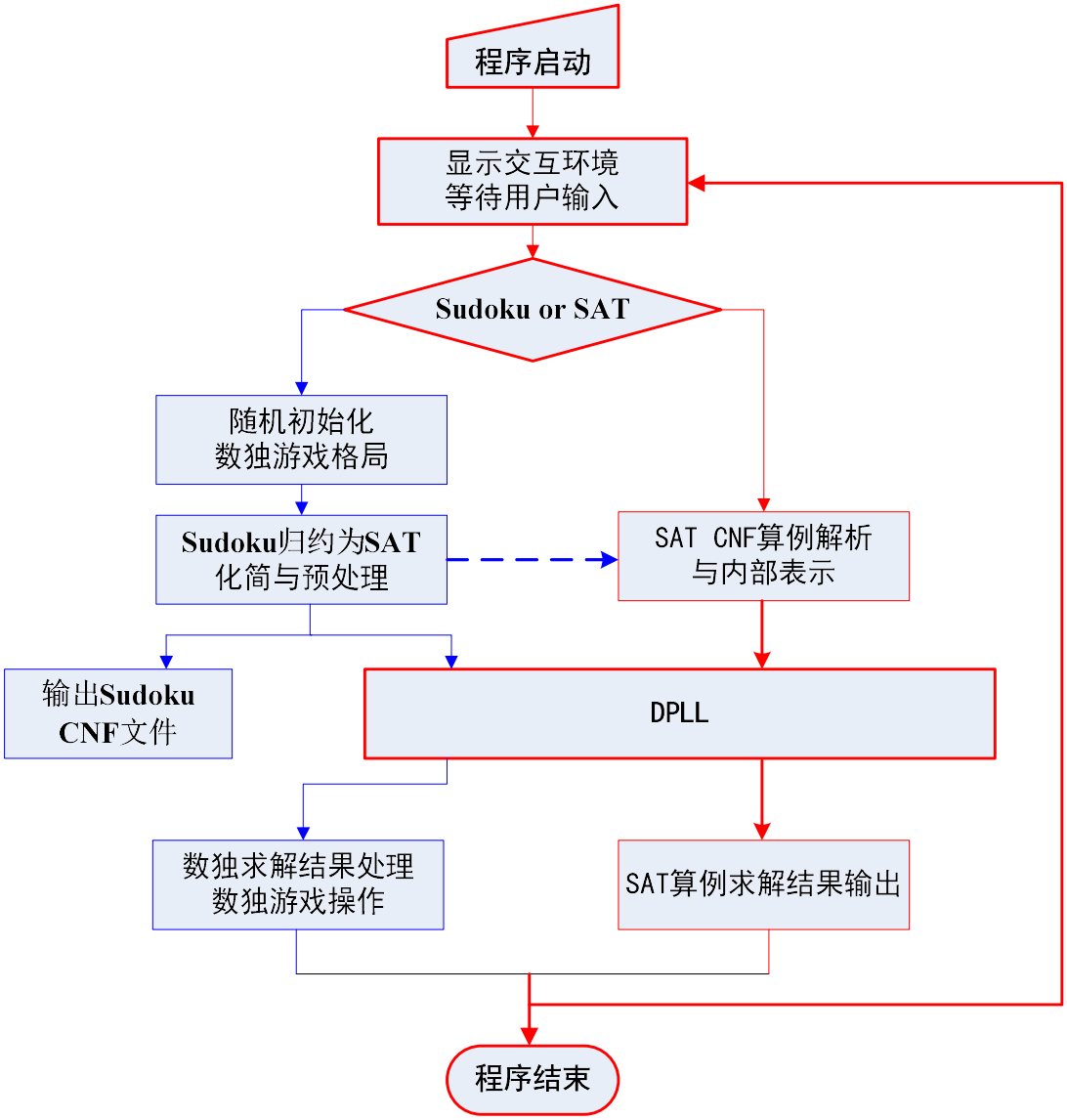
程序启动
显示交互环境
等待用户输入
Sudoku or SAT
随机初始化
数独游戏格局
Sudoku归约为SAT
化简与预处理
SAT CNF算例解析
与内部表示
输出Sudoku
CNF文件
DPLL
数独求解结果处理
数独游戏操作
SAT算例求解结果输出
程序结束

图 2-1 程序设计示意图

3 系统详细设计

3.1 有关数据结构的定义

我定义了子句、文字结点，每一个子句有一个子句头，右指针指向该子句第一个文字，而下指针指向下一个子句，子句头和文字会设置 flag 标记这个子句或文字是否被删除，或者是否被赋值。这个这样可以比较好的串联起整个 cnf 求解的结构。

对于文字，还设计了数据结构记录文字是否被赋值、以及文字出现的频率。便于更方便的进行变元选择。

3.1.1 文字和子句的定义

使用单链表对文字和子句进行定义

```
/*定义子句链表结点结构类型*/
typedef struct Clause {
    int literal;//记录子句中的文字
    int flag;//标记该文字是否已被删除，未删除时值为 0，否则值为使之删除的变元序号
    struct Clause* nextl;//指向该子句中下一文字的指针
    struct Clause* litline;//指向整个 CNF 公式邻接链表中下一个文字相同的子句结点
}Clause;

/*定义 CNF 范式链表结点（即子句链表头结点）结构类型*/
typedef struct Paradigm {
    int number;//子句中显示的文字数
    int flag;//标记该子句是否已被删除，未删除时值为 0，否则值为使之删除的变元序号
    struct Paradigm* nextc;//指向下一子句的头结点
```

```
    struct Clause* sentence;//子句头指针
}Paradigm;

/*定义 CNF 范式链表头结点类型，存储 CNF 范式信息*/
typedef struct Root {
    int litsize;//存储文字数量
    int parasize;//存储子句数量
    Paradigm* first;//指向第一个子句
}Root;

/*定义指向子句链表头结点的指针链表结点结构类型*/
typedef struct Paraline {
    Paradigm* claline;//指向子句链表头结点 Paradigm
    struct Paraline* next;//指向下一链表结点
} Paraline;

/*定义文字相关信息链表结构类型*/
typedef struct LitTrabverse {
    Paraline* Tra_cla;//指向含有该正文字或负文字的子句头结点链表的头结点
    Clause* Tra_lit;//指向该正文字或负文字的文字结点
}LitTrabverse;
```

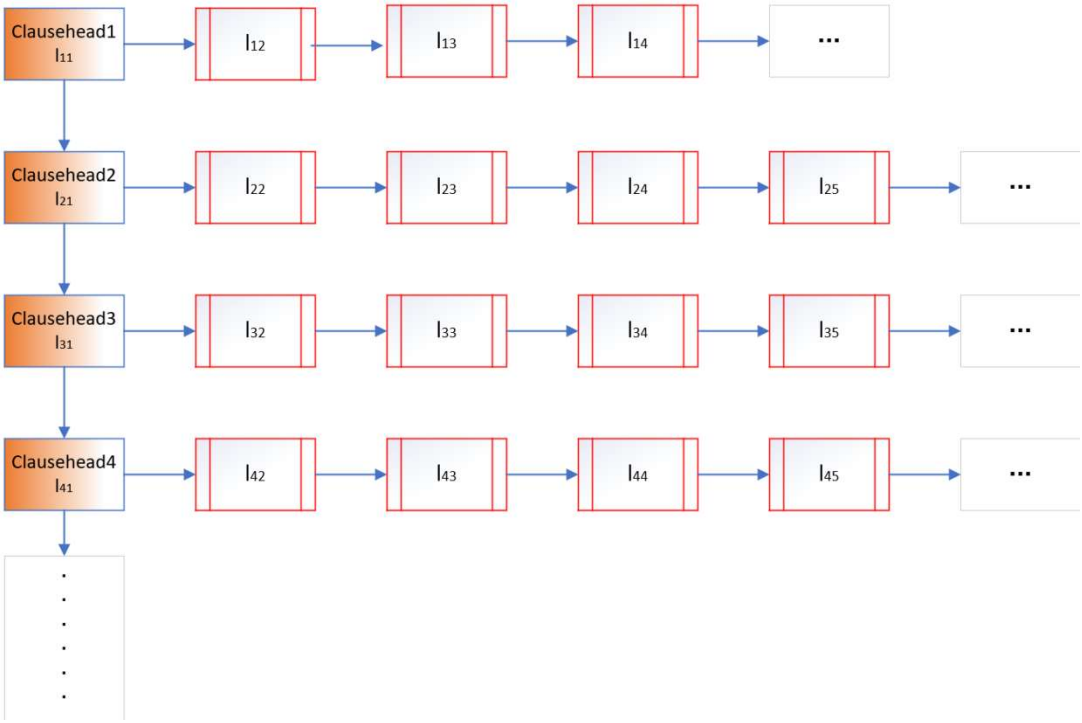

图 3-1 文字和子句存储结构

3.1.2 用于统计文字出现频率的数据结构

```
/*定义存储变元信息的变元线性表结点结构类型*/
typedef struct ArgueValue {
    int Value;//变元的真值
    int IsInit;//变元是否已赋值
    int Occur_times;//变元在所有子句中出现的总次数
    LitTrabverse Pos;//变元所有正文字的相关信息结构
    LitTrabverse Neg;//变元所有负文字的相关信息结构
    sudokusolver xyz;//求解数独文件时所需的变元信息
} ArgueValue;


typedef struct {
    int number;//变元在 CNF 范式中出现次数
```

```
    int amount;//出现次数为number的不同变元总数
}Frequent;
typedef struct {
    int* amount;//对不同的次数，出现次数相同的变元数量相同的数目
    int totalvariety;//*amount不同的总数
} SameFre_AmountList;

typedef struct {
    int argue;//记录真值设为1的文字
    int flag;//记录是否已经测试过argue的反文字的真值
} mainstack;//非递归式DPLL中记录每次设为1的文字的栈结构
```

3.2 主要算法设计

3.2.1 DPLL 算法

DPLL 算法是 SAT 求解器的核心算法，利用递归的思想，和单子句策略、分裂策略两个策略将 CNF 公式逐步化简。单子句策略是指挑出 CNF 范式中的所有单子句，易得要使 CNF 范式有解其文字真值必为 1，进而可以得出公式中所有含有该文字的子句真值均为 1，可以删去，同时公式中所有该文字的反文字真值皆为 0，可以删去子句中的该文字的反文字；分裂策略是指按某种策略选出一个未赋真值的变元的文字，设其真值为 1，对 CNF 范式进行探测，再循环执行单子句策略和分裂策略，如若过程中某个子句中只剩下设真值为 1 的文字的反文字，说明此次探测失败，则需返回前一次分裂策略，得该选择文字的真值必为 0，再重新探测 CNF 范式。

3.2.2 CNF 文件处理

(1) 创建 CNF 范式邻接链表及变元表

status CreateParadigm(FILE** open)

功能：从 cnf 文件中读取文字数和子句数，创建相应 cnf 范式邻接列表，并

且统计每个变元出现的次数

(2) 销毁所有链表和线性表结构

status DestroyParadigm(Root* r)

功能：销毁 CNF 范式链表、销毁单个子句链表、释放存储文字的单个子句链表结点空间、释放子句链表头结点存储空间、释放变元正负文字信息链表存储空间

(3) 判断 CNF 范式中是否还有单子句

int HasUnitClause(Root* r)

功能：判断 CNF 范式中是否存在单子句

返回类型：int

返回值：存在单子句：该单子句所含的唯一文字

不存在单子句：0

3.2.3 程序优化

变元选择策略在 DPLL 算法的效率上影响很大。我采用了两种策略，未优化的第一种策略是直接选取第一个文字变元，第二种是统计所有文字变元的出现频率，选取出现频率最高的变元。

但经过对比，发现这两个选择策略各有优劣，针对不同的 CNF 范式算例文件，求解最快的文字选择策略也不同，如若系统单单选用一个选择策略，则难以做到对尽量多的算例文件进行快速求解。由于这个文字选取函数的主要区别在于是否利用文字的出现频率的高低进行选择，因此我挑选了若干个较有代表性的 CNF 算例文件，依次测试 2 个算法的求解时间。

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。

4 系统实现与测试

4.1 系统实现

1. 硬件环境：

处理器：Intel(R) Core(TM) i5-11300H CPU @ 3.10GHz

机带 RAM：16.00GB

系统类型：64 位操作系统，基于 x64 的处理器

2. 软件环境：

Windows11 下 Visual Studio

4.2 系统测试

4.2.1 交互系统展示

(1) 主菜单：提供 SAT 求解和蜂窝数独两个入口

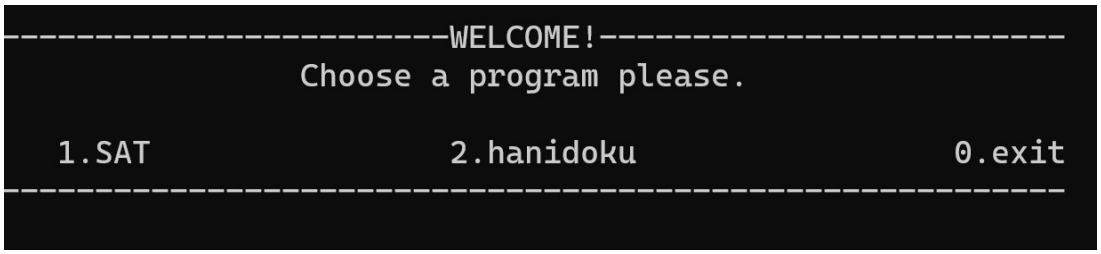

图 4-1 主菜单展示

(2) SAT 板块：

输入 cnf 文件名，系统读取该文件，读取失败打印“打开文件失败”，否则进入二级菜单，提醒用户输入是否选择优化算法，算法执行后会输出算法计算时间，当用户用两种算法计算过，程序会输出优化率。用户可以选择将答案输出到相同路径下的同名.res 文件中。

```
-----------------------WELCOME!-------------------------
                Choose a program please.

   1.SAT                2.hanidoku                  0.exit
--------------------------------------------------------
1
请输入需求解的cnf文件名: ./case/sud00009.cnf
                Choose an option please.
--------------------------------------------------------
  1.求解该算例    2.验证CNF公式解析结果    3.保存求解结果
  0.退出
--------------------------------------------------------
1
 使用: 1.优化算法                          2.优化前算法
1
计算时间为: 8.000000ms

                Choose an option please.
--------------------------------------------------------
  1.求解该算例    2.验证CNF公式解析结果    3.保存求解结果
  0.退出
--------------------------------------------------------
1
 使用: 1.优化算法                          2.优化前算法
2
计算时间为: 127.000000ms
优化率93.700787%
```

图 4-2 DPLL 算法模块展示

(3) Hanidoku 板块

进入蜂窝数独板块，输入挖洞数，系统根据挖洞数同各国挖洞法生成初始格局，并将蜂窝数独规约成 cnf 问题输出到 cnf 文件。

```
****************hanidoku*****************
1.开始蜂窝数独             0. 退出蜂窝数独
1
请输入挖洞数:
20
*****************initialize*****************
生成初始数独格局:
         / \ / \ / \ / \ / \
        | 2 | 0 | 3 | 4 | 5 |
       / \ / \ / \ / \ / \ / \
      | 5 | 0 | 4 | 2 | 6 | 3 |
     / \ / \ / \ / \ / \ / \ / \
    | 4 | 3 | 6 | 0 | 0 | 2 | 7 |
   / \ / \ / \ / \ / \ / \ / \ / \
  | 0 | 5 | 8 | 1 | 2 | 0 | 3 | 0 |
 / \ / \ / \ / \ / \ / \ / \ / \ / \
| 0 | 0 | 2 | 7 | 8 | 9 | 5 | 1 | 6 |
 \ / \ / \ / \ / \ / \ / \ / \ / \ /
  | 2 | 7 | 0 | 0 | 4 | 8 | 0 | 5 |
   \ / \ / \ / \ / \ / \ / \ / \ /
    | 0 | 8 | 7 | 6 | 0 | 3 | 4 |
     \ / \ / \ / \ / \ / \ / \ /
      | 0 | 3 | 0 | 0 | 5 | 7 |
       \ / \ / \ / \ / \ / \ /
        | 4 | 0 | 0 | 2 | 0 |
         \ / \ / \ / \ / \ /
已规约生成cnf文件./hanidoku.cnf
```

图 4-3 hanidoku 模块展示

4.2.2 测试大纲

(1) 测试流程：

对于 DPLL 算法模块测试，依次读入任务文件夹下测试样例，使用优化算法和优化前算法进行求解，并输出计算时间和优化率。

对于蜂窝数独游戏模块，模拟用户进行蜂窝数独游戏，首先输入挖洞数，然后进行用户填充交互，最后输出正确数独终盘。

(2) 测试数据：

对于 DPLL 算法模块，测试基准算例，和部分小型算例，对于中型算例，尽量测试所有的算例，以测试系统的算法效率。

对于蜂窝数独模块，随机输入不同的挖洞数，并测试输入答案正确和输入答案错误两种情况。

(3) 测试目标；

测试 DPLL 算法的效率，用户交互的完整性，蜂窝数独的可玩性。

4.2.3 DPLL 算法模块测试

表 4-1 DPLL 算法时间测试

算例名	优化前计算时间 (ms)	优化后计算时间 (ms)	优化率
Problem1-20.cnf	1	0.0001	99.9%
Sud00009.cnf	132	9	93.181818%
Sud00001.cnf	33	31	6.060606%
Sud00012.cnf	26	24	7.692308%
Sud00021.cnf	687	191	72.197962%
bart17.shuffled-231.cnf	7	6	14.285714%
Sud00861.cnf	21	19	9.523810%

4.2.4 蜂窝数独游戏测试

(1) 输入挖洞数输出数独初盘

根据挖洞数从完整正确的蜂窝数独格局中删除一定数目的格子，呈现数独初盘。

图 4-4 蜂窝数独生成初盘测试

(2) 输入答案检验答案正确性

比对用户输入的答案和正确的答案，如果有错，展示错误数据的行列数、正确答案和用户输入的答案。

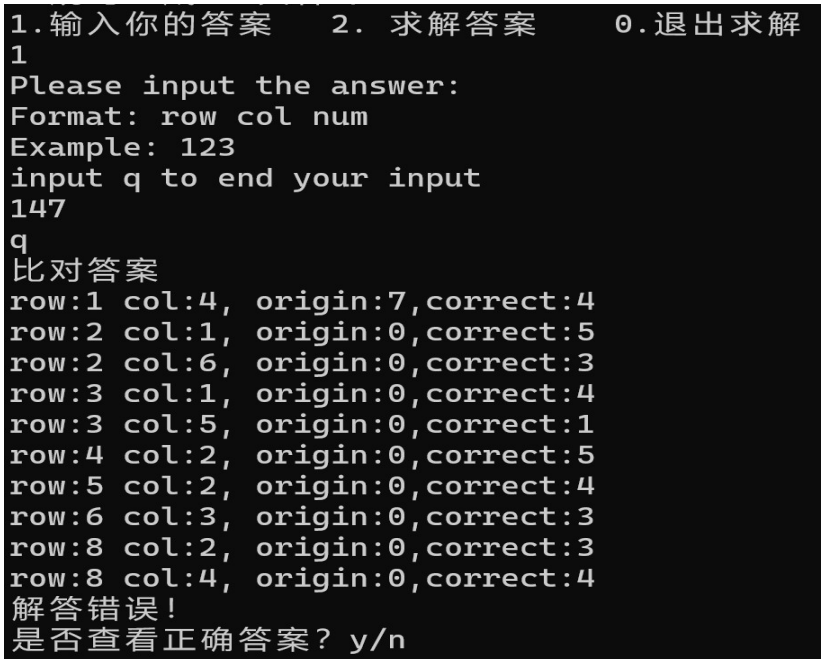

图 4-5 蜂窝数独比对答案测试

(3) 输入答案错误时查看正确答案

用户输入答案错误时，可以选择查看正确答案。

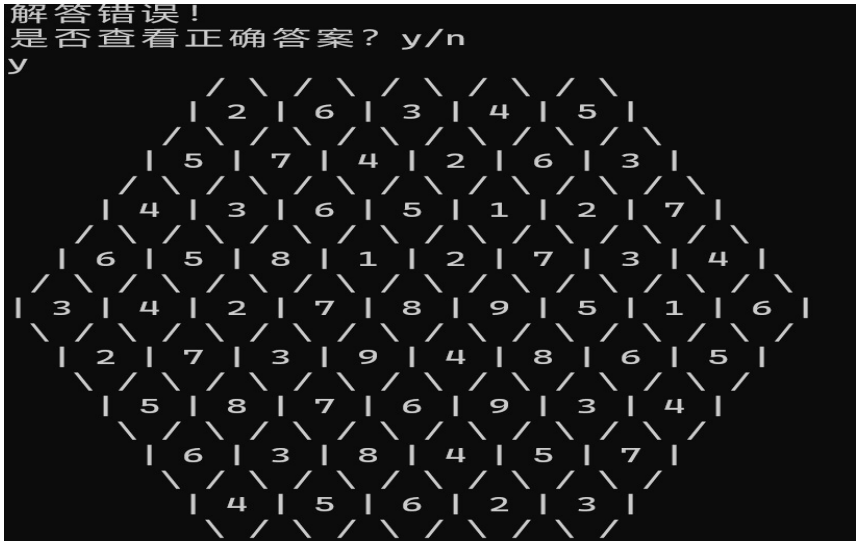

图 4-6 蜂窝数独查看答案测试

(4) 输入正确时提示答案正确

当用户输入的答案正确时，提示用户输入的答案正确。

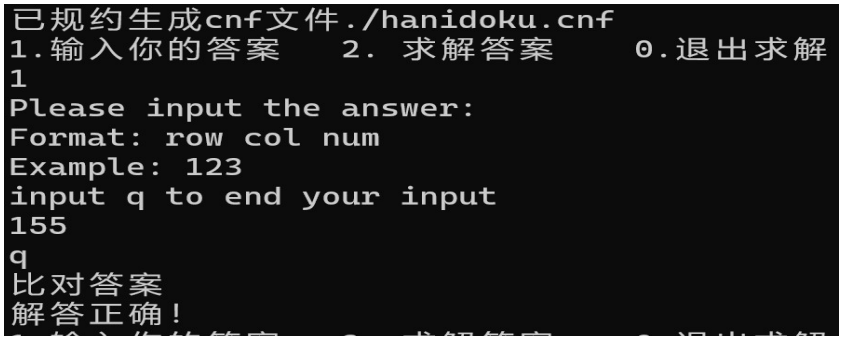

图 4-7 蜂窝数独比对答案正确测试

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）实现了一个基于 DPLL 算法的 SAT 求解器；

（2）在选词策略和判断是否满足的方式上对求解器做了一定优化；

（3）完成了数独游戏的初始化，cnf 文件转化，并实现了游戏的可交互性；

（4）设计了测试方案并完成了多个算例的测试；

（5）完善了各种部分的衔接，构建一个简易系统。

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：

因为设计的 SAT 求解器较为低级，用了较多的数据结构体去构建。测试过程中，面对较多变元时，可以求解但时间效率往往有点低下。起初仅仅是用链表储存文字和子句，后续优化了才加入变元真值表和正文字负文字信息链表（便于更快的查找到要删除的文字节点），即便如此，时间效率还是有一些低下。在今后的工作中，希望可以得到更多的编程和项目经验；

在 DPLL 算法实现中，最主要的是变元选择方案。我选择了递归回溯方案，因为设置了较复杂的数据结构体，所以回溯时都要考虑到。变元选择方案也是需要根据变元数和子句数之间的关系。暂未发现如何用一个函数能归纳总结选择如何选择变元方案。在今后的工作中，希望可以得到更多的关于 DPLL 算法变元选择编程知识；

在数独游戏模块，没有优化图形界面，使用的用户输入数独模块，为输入行列值、数字，输入后没有很直观的反馈。希望可以通过更多高级语言的学习以及更深入的 c 语言理解学习到更好的数独游戏交互实现。

6 体会

这次课设是对我们C语言和数据结构能力的一次很好的实践，我们通过课程设计对于链表等结构有了更深的了解与实现能力，使得我们的数据结构的知识不是再浮于理论层面。同时这也是我们第一次做一个大工程，难免会有一些困难。从最开始的设计整个系统开始，便是一件很头疼的事情。但是比较麻烦的是当你编译通过之后仍然有许多你找不出来的bug，这个时候需要你单步调试，找出错误在哪里。通过这一次训练，我们对于大工程的理解，寻找bug的能力都有所提高，对于C语言和数据结构的理解也有所提高。

万事开头难。在最开始的时候阅读理解很多次任务书才能较为清晰的感受到大致的方向，比如数独棋盘构建CNF文件的三个约束，将CNF文件中语义编码转换为自然顺序编码的方式。当上手慢慢摸索的时候，经历了多次失败后才慢慢开始懂得如何去构建SAT求解器，DPLL求解和Hanidoku系统；

在先开始，最大的问题就是设计整个系统，在理清的问题的基本思路之后，开始对于完成问题的基本数据结构进行设计。在任务书的提示下，我建立了以子句链表链接文字链表的数据结构，从效率上看，这样做的效率很高。但是到后来进行DPLL的时候我发现，链表中删除过的节点便无法再恢复，而重新存一个链表空间开销很大，而且无法准确定位，需要一个个遍历来恢复。并且，因为不知道递归循环的次数，所以并不知道要复制多少个这样的链表，这样程序很可能会崩溃。在空间上，会重复构建相同的链表，运行大型算例时会出现内存爆满的情况。开始我想的是可能是因为递归过深的问题，但是后来发现可能是因为对于链表删除的处理可以再优化，比如可以设置一个表示状态的变量，删除结点时并不是真的删除，而是把它的结点状态设置为flase，这样在恢复结点时就不需要再重复创建。

但是最后呈现出来的算法效率还是不够高，不能解决大型算例，算法还有值得优化的地方。

而后遇到的主要问题就是一些隐性的bug。程序编译通过之后能跑过一些样例而有一些样例跑不过。面对这样一个工程，单步调试还是十分困难的，经过大约一周不断的尝试，终于发现了有几个会使得DPLL崩溃的小错误，这个调试的经理是很痛苦也是很宝贵的。

最后便是数独界面的设置，我使用的是命令行交互，观感上不是很美观，如果时间允许的话，应该使用 qt 或者 easyx 做一个图形界面，完成绘图、定位鼠标等功能。

综上，通过这次课设，我们对于 C 语言及其工程有了更好的认识，锻炼了我们的结构设计能力以及调试代码的能力。这次课设也让我们学习了一些新的东西，比如如何设计界面等等，也让我们接触到了更多 C 语言的库，对于我们是十分有好处的。

参考文献

[1] 严蔚敏，吴伟民.数据结构（C 语言版）.清华大学出版社

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

[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] 熊伟. 可满足性 DPLL 算法研究. 硕士学位论文，复旦大学：20070522

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

[6] Sudoku Puzzles Generating: from Easy to Evil.
http://zhangroup.aporc.org/images/files/Paper_3485.pdf

[7]薛源海，蒋彪彬，李永卓. 基于“挖洞”思想的数独游戏生成算法 北京理工大学

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