数独求解及生成程序说明文档

一、分工介绍

本程序由崔轶锴、刘松铭合作完成。

前期准备：刘松铭负责查阅求解数独相关的文献，给出求解算法的初步思路，崔轶锴负责寻找生成数独的有关资料，给出生成数独的初步思路。

中期讨论与代码编写：二人共同商定进一步的算法方案和数据结构，崔轶锴编写了生成数独，DLX 算法和暴力法模块的代码，刘松铭编写了暴力法和 DLX 算法模块的代码。

后期整合与调试：崔轶锴编写了最终代码的模板，并将所有模块的代码取长补短，整合到一个 cpp 中，刘松铭搭建了说明文档的框架，二人共同完成了代码的检查与维护、程序的运行测试、算法的进一步改进和说明文档的编写。

二、总流程介绍

1. 简要介绍

本程序共包含两大功能：生成数独和求解。一方面，本程序可以输出一个有唯一解且少于 40 个格子填有数字的数独。另一方面，本程序可以求解任意一个数独题，判断解的情况（无解、唯一解或多解）并在唯一解的情况下输出数独的解在多解的情况下输出数独的两个解。

本程序用结构体 Board 存储数独，用全局变量 vector<Board> g_ans 存储数据的解，用结构体 Block 存储单个方格的信息。

在计算数独部分：

本程序同时采用搜索法和 Dancing Link X 算法。

本程序采用 Solve 函数对输入的数独矩阵进行预处理，用 Search 函数求解数独。用结构体 DancingLink 来存储 Dancing Link X 算法所需的全部信息和实现其功能的成员函数，并用 SolveDLX 函数实现用 Dancing Link X 算法解数独。

在生成数独部分：

本程序采用 GenerateFull 函数生成完整的数独，并用 GenerateSudoku 函数从完整的数独生成有空格的数独题。

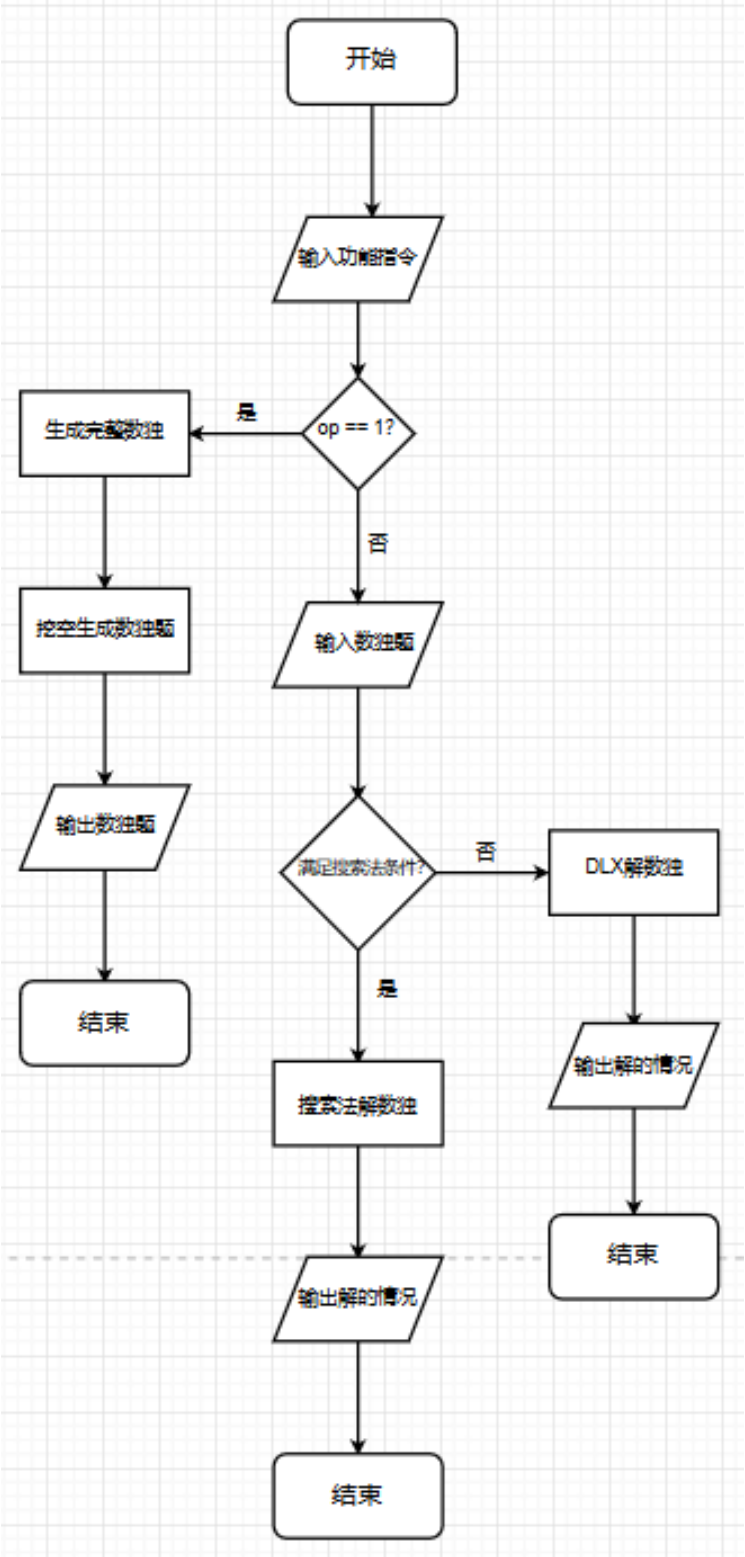
2. 总流程介绍

首先是用户输入功能指令：一个整数，1 或 2。

若为 1，本程序将会调用 GenerateFull 函数生成完整的数独并用结构体 Board 储存，命名为 full，接着调用 GenerateSudoku 函数将完整的数独 full 处理成含空的数独题并用结构体 Board 储存，并命名为 question，调用 question 的成员函数 print 将其打印到输出端。

若为 2，本程序将会创建一个 Board 变量 question，调用它的成员函数 read 从输入端读取数独。如果输入的数独满足搜索法条件，则调用 Solve 函数采用搜索法求解数独，否则采用改进的 Dancing Link X 算法解数独。即先调用 Solve 函数用搜索法填充数独直至每一个格子都有多于一个的候选解，再调用 SolveDLX 函数用 Dancing Link X 算法解数独。如果 g_ans 为空，表明无解，输出无解信息，如果 g_ans 大小为 1，则输出唯一解信息并输出一个解，如果 g_ans 的大小超过 1，输出多解信息并输出两个解。

3. 总流程图


开始
输入功能指令
op == 1?
是
生成完整数独
挖空生成数独题
输出数独题
结束
否
输入数独题
满足搜索法条件?
否
DLX解数独
输出解的情况
结束
是
搜索法解数独
输出解的情况
结束

三、主接口 1（求解数独）介绍

1. 核心算法原理介绍

a) 算法总括

本程序同时使用了两种不同的解数独的算法：搜索法和 Dancing Link X 算法。我们发现，在空格数较少的时候，搜索法的效率更好，而在空格数较多的时候，Dancing Link X 算法则更加出色。为了将二者的优势结合起来，我们以空格数为划分，在空格数较少的时候，采用搜索法求解数独，在空格数较多的时候，先采用搜索法填数独，直到所有剩余的空格都有多于两种的可填数字，然后在采用 Dancing Link X 算法求解数独。下面分别介绍两种算法。

b) 搜索法

设计思路：对于所有的空格，尝试每一种数字，向下搜索，如果可以填到最后一个格子，则说明数独有解，则当前的数独填法即为一组可行解。回溯，直到尝试完所有的可能性。然后，我们对于搜索进行了以下优化：

1）改变搜索顺序：在我们自己解数独时，常常会从剩余情况数最少的格子开始填。在该优化中，我们使用了同样的搜索策略。在每次搜索时，分别计算所有的空格子在该状况下有多少个可能的填法，从填法数最少的格子开始搜索。

2）提早截断：由于在题目要求中对于有多个解的情况只用输出两个解，故我们在搜索过程中如果发现了第二组可行解，就不用再继续搜索，可以直接输出答案从而减少时间。

3）对于特殊情况的提早判断：在该算法中，如果在输入时已经有不合法的情况发生，而该算法仍会尝试剩余格子的所有可能填法，导致运行时间变长。因此，我们对给定的数独进行了预处理，如果给的数独已经不合法，则直接输出无解，不再进行搜索，提高运行效率。

c) Dancing Link X 算法

Dancing Link X 算法是对精确覆盖问题的一种极其优秀的搜索算法。在介绍该算法之前，先介绍精确覆盖问题，以及如何将求解数独的问题转化为精确覆盖问题。

精确覆盖问题的描述如下：给定一个 0/1 矩阵，从中选出一些行构成新矩阵使得新矩阵的每一列有且仅有一个 1.。

下面介绍如何将求解数独的问题转化为精确覆盖问题。

考虑填写数独时的限制：1，每个格子内只能填写一个数；2，每行内每个数只能出现一次；3，每列内每个数只能出现一次；4，每个宫内每个数只能出现一次。可以将这些限制变成 0/1 矩阵。矩阵的前 81 列分别表示在某个格子内填数。对于之后的 81 列，每九列为一组表示一行，每组中的第 i 列表示在对应行填了 i。对于再之后的 81 列，每九列为一组表示一列，每组中的第 i 列表示在对应列填了 i。对于最后的 81 列，每九列为一组表示一宫，每组中的第 i 列表示在对应宫填了 i。对于每个格子所有可能的填法(已经填的格子只有一种填法)，按照上述建立一行。对于一组数独的可行解，我们需要选出该矩阵的若干行，并保证每一列都恰有一个 1（分别表示在某个格子内有数，在某列、行、宫内有数字 1~9)。由此，一个数独就被转化成了一个精确覆盖的问题。

精确覆盖问题的一个简单算法（X 算法）如下：对于当前的 0/1 矩阵 A，如果 A 为空，则我们已经找到一组可行解；否则，看 A 的第一列，找到值为 1 的行，选取该行，并将该行内值为 1 的对应列删除，同时删除所有（包括自己）的矛盾行（即在这些行中这些列的至少一列的值为 1）。对得到的新矩阵 A'再次重复上述算法。

而 Dancing Link X 算法是使用双向链表删除元素的简便性，以及在删除元素时只需改变指针，不需要真正删除元素的性质对上述算法进行的优化。

2. 实现流程

主程序首先调用 Board 类型变量的成员函数 read 读取数独题面，若空格数超过 36，则直接调用 SolveDLX 求解数独，否则调用 Solve 函数求解数独。Solve 函数首先会进行初始化，在这个过程中如果发现给定的数独有冲突，会输出无解信息。接着清空 g_ans，调用 Search 函数求解数独。在 Search 函数中，首先会采用暴力法求解数独，当发现每一个格子的解都大于两个的时候，再调用 SolveDLX 求解数独。

3. 子接口介绍

a) 结构体 DancingLinkX 介绍

i. 功能描述

用于存储 Dancing Link X 算法解数独所需的信息，并且提供函数执行该算法需要的功能。

ii. 变量简单描述

数组 l,r,u,d 分别每个点的左右上下的行列信息。

数组 row,col 分别对应元素所在行列的行首指针和列首指针。

变量 rows 表示已经加入的行数。

变量 ansk 用于储存数独的解。

变量 n，m 分别为矩阵的二维参数，cnt 为点的总数。

数组 s，h 分别为每一列的节点数以及每一行的头结点。

iii. 成员函数介绍

1) init 为初始化函数。该函数会生成一个只包含 324 个列首节点的链表，并将其他数据初始化为 0 或-1。
2) link 函数接受参数 R 和 C，并在 R 行 C 列建立新的结点。
3) remove 函数接受参数 c，删除 c 行 c 列上有点的行。
4) resume 函数接受参数 c，恢复 c 行 c 列上有点的行。
5) dance 是 DancingLinkX 算法的核心函数，用于执行 DancingLinkX 算法的搜索部分。传入 deep，表示当前搜索的深度；以及 allAnsFlag 表示是否需要找到所有的可行解。当 r[0] == 0 时，说明所有的列均已经被覆盖，此时选择的行编号为一组合法解，记录当前答案并返回。否则，尝试当前列的所有对应行，依次尝试，并递归继续搜索。

b) Search 函数介绍

i. 功能描述

用于求出数独的解。如果求所有解标志为真，会求出所有数独的解，否则最多求出两个解。

ii. 接口描述

输入为一个存储数独信息的 Board 结构以及求所有解和暴力法求解的 bool 标志。输出为数独的解，存储在 g_ans 数组中。

iii. 实现原理

第一部分是求出当前仍然为空的格子中，可能填法最少的格子，将坐标记录到 mnx,mny.中。

第二部分是尝试 mnx，mny 这个格子的所有可能填法并递归搜索。

当 forceFlag 为假，并且所有格子的候选解大于 1 时，采用 DLX 求解数独。

c) Solve 函数介绍

i. 功能描述

用于对搜索的初始化，并调用 Search 函数进行求解。

ii. 接口描述

输入为一个存储数独信息的 Board 结构以及求所有解和暴力法的 bool 标志。若求所有解的标志为真，则返回解的数量，否则返回解的情况（唯一解、无解或多解）。

iii. 实现原理

如果给定的数独中已经出现冲突则返回无解。

否则会清空 g_ans，并调用 Search 函数进行求解，并根据求所有解的标志以及获得的解的数量返回对应的值。

d) SolveDLX 函数介绍

i. 功能描述

用于对 DancingLinkX 算法的初始化，并通过 DancingLinkX 结构体的

成员函数实现 DancingLinkX 算法求解数独。

ii. 接口描述

输入为一个存储数独信息的 Board 结构以及求所有解的 bool 标志。若求所有解的标志为真，则返回解的数量，否则返回解的情况（唯一解、无解或多解）。

iii. 实现原理

首先将 g_ans 清空，利用全局变量 dlx 解数独。将 dlx 设为全局变量的主要原因是防止堆栈溢出。

首先调用 dlx 的 init 函数进行初始化操作。

枚举所有的格子，如果这个格子中已经填好了数，则向 dlx 中添加一行（一个限制），否则枚举这个格子能否填入 1 至 9，如果可以填入，则向 dlx 中添加对应的限制。

最后调用 dlx 的 dance 函数来解数独。

四、主接口 2（生成数独）介绍

1. 核心算法原理介绍

一种简单的算法是：随机在当前（一开始为空）的数独中选取一个格子填入一个随机数字，并通过解数独的算法检查新的数独的解的个数，如果无解，则重新选取格子和数字；如有唯一解，则这个数独就是一个合法的数独题；如有多解，则对新的数独重复以上算法，直到只有唯一解。

但是这种算法有一个弊端：在解很少的情况下，可能很难随机到合法的

格子和数。为了解决这个问题，我们使用了相反的方向来生成数独。

首先，我们会生成一个完整的数独棋盘（即每个格子均有数的数独）。方法如下：每次填充一行，随机一个 1~9 的全排列，分别尝试该排列的每一个轮换与已经填充的行是否矛盾，如果该全排列的所有轮换均不可行，则从第一行重新开始生成。

然后，我们从这个已经生成的完整数独中随机选取一个格子删除数，并判断新的数独是否只有唯一解，如果有多解，则尝试删除其他格子中的数字。如果尝试完所有的格子均无法继续删除，则停止删除。这样我们就获得了一个有较少数字的数独（经测试，生成的数独一般只有 23~25 个格子中有数，符合题目要求）。

2. 实现流程

主程序接口会先调用 GenerateFull 生成一个完整的数独。然后再调用 GenerateSudoku 对刚才获得的数据进行删数，从而获得一个填有数字的格子数量小于 40 的数独。

3. 子接口介绍

a) Rand 函数介绍

i. 功能描述

用于随机产生一个随机的自然数。

由于在 Windows 系统中产生的随机数最大为 32767，很容易出现循环。为了避免出现循环导致对数据的随机性产生影响，重新定义了 Rand 函数。

ii. 接口描述

没有输入。输出是一个 int 类型的随机数。

iii. 实现原理

RAND_MAX 是编译器中已经给出的一个常量，表示 rand 函数产生的随机数的最大值。如果 RAND_MAX 为 32767，我们则将两个随机数拼至一起产生一个较大的随机数。否则 RAND_MAX 一定为 2^{31}-1，直接返回 rand 产生的随机数即可。

b) GenerateFull 函数介绍

i. 功能描述

用于产生一个合法的填满数的数独。

ii. 接口描述

没有输入。输出是一个存储填满的数独的 Board 变量。

iii. 实现原理

依次尝试填入每一行。先使用 random_shuffle 生成一个随机的 1 至 9 的排列。然后利用队列来尝试部分排列，如果可以找到一种排列使得将该排列填入数独中没有冲突则尝试下一行，否则重新开始生成数独。

c) GenerateSudoku 函数介绍

i. 功能描述

用于产生一个少于 40 个数字的数独，且该数独的解的数量超过一个给定的值。

ii. 接口描述

输入是一个存储填满的数独的 Board 变量以及一个 int 整型变量

solCnt。若solCnt的值为-1，输出一个存储有唯一解的数独题的Board变量，若 solCnt 的值不为-1，则输出一个存储有解的数量大于等于solCnt 的数独题的 Board 变量。

iii. 实现原理

随机找到一个没有被填充过的格子，并删除这个格子中对应的数，检查新的数独的解的个数是否满足要求，如果不满足，将该格子标记并尝试删除其他格子中的数。如果所有的格子均已经被标记，则当前的数独是一个局部最优解。返回这个局部最优解。

五、其他模块介绍

1. 结构体 Board 介绍

a) 功能描述

用于存储一个数独的信息，方便传递参数。

b) 变量描述

board 表示当前数独的状态，对应的数字是 0 表示该格子为空，否则该格子填入了相应的数字。

filled 表示当前数独已经有多少个格子填好了数。

c) 成员函数介绍

1) Board 为类的构造函数，用于初始化一个空的数独。

2) read 用于读入数独，并计算该数独有多少个格子已经填好了数字。

3) print 用于输出一个数独，格子之间用空格隔开。空格子用'-

‘（不含引号）表示。

4） 重载了[]运算符，这样可以直接使用[]来调用格子的信息，而不需要写*.board[][]来调用，减小代码量。

5） check 为检查函数，用于检查对应的格子是否可以填入某个数字。需要传入三个整数，分别表示要检查的格子的两个坐标，以及要检查的数字。

6） putNumber 和 eraseNumber 分别用于向数独中填入或删除一个数字，并同时更改 filled 的值，防止出现因忘记修改 filled 而导致的 bug 出现。

2. 结构体 Block 介绍

a) 功能描述

用于存储单个数独方格的信息。

b) 变量描述

含有三个 int 整型变量 x，y，value，分别用于存储方格的行坐标，列坐标和方格内的数值。初始化三者皆为 0。

c) 成员函数介绍

含有一个构造函数，用于为 Block 型变量赋值。

3. 宏定义介绍

a) 用于求解数独的宏

无解标志为 NO_SOLUTION，定义其值为 0，唯一解标志为 ONE_SOLUTION，定义其值为 1，多解标志为 MANY_SOLUTION，定义其值为-1。使用标志而非数值可以增强程序的可读性。

b) 用于 Dancing Link X 算法解数独的宏

将 HEAD 定义为 0，表明链表头，将 COLUMNS 定义为 324，表明列的数量。将 MAX 定义为 101101，表明 DLX 算法解数独中数组的大小。

4. 输入模式介绍

a) Mode 1

只从控制台输入一个整数 1。程序会输出一个只有唯一解的数独，并且该数独所含有的数字个数少于 40。

b) Mode 2

从控制台输入一个整数 2，并输入一个数独。

如果该数独无解，输出"No_Solution"（不含引号）。

如果有唯一解，输出"OK"（不含引号），并输出该唯一解。

如果有多解，输出"Multiple_Solutions"，并输出两组合法解。

c) Mode 3

在调用该程序时，传入一个参数表示输入文件的文件名。

程序会从该文件中读取信息并将输出内容输出到文件"output.txt"（不含引号）中。

d) Mode 4

在调用该程序时，传入了两个参数，第一个参数为文件名，第二个参数为开关"-MoreModes"。程序会从该文件中读取信息并将输出内容输出到文件"output.txt"（不含引号）中。在这种模式下，程序会求解所有的合法解，生成的数独的解会超过给定的数量。

六、实验数据

我们利用爬虫将数独网站的上的数独数据收集到本地，我们共收集了500 道数独题，其中有 100 道入门级题目，100 道初级题目，100 道中级题目，100 道高级题目，100 道骨灰级题目。我们分别使用纯暴力搜索法，DLX 法，DLX（暴力预处理至所有的格子备选数多于一个时，换用 DLX 算法求解数独）以及最终的算法（DLX 预处理，以空格数分界结合纯暴力法）进行测试。得到的平均耗时如下。

题目类型/时间（ms）	纯暴力	DLX	DLX 预处理	最终算法
1	2.0036	3.7327	2.8206	2.0068
2	2.4423	3.9236	2.9972	2.4516
3	4.2089	5.3465	3.5686	3.2434
4	8.7742	7.8592	3.7554	3.5919
5	9.6212	8.7791	3.9498	3.6228

表格中的难度从 1-5 逐渐上升。从表格中可以看出，在难度较低的时候，纯暴力表现得要比 DLX 好，而难度较高的时候 DLX 更好一些。经过预处理过后的 DLX 比没有预处理的 DLX 表现要好，最终的算法综合了前者的优势。

```
8 - - - - - - - -
- - 3 6 - - - - -
- 7 - - 9 - 2 - -
- 5 - - - 7 - - -
- - - - 4 5 7 - -
- - - 1 - - - 3 -
- - 1 - - - - 6 8
- - 8 5 - - - 1 -
- 9 - - - - 4 - -
```

此外我们还使用号称世界上最难的数独题—芬兰题的其中一个版本，对四种算法进行了测试，结果如下。

题目类型/时间（ms）	纯暴力	DLX	DLX 预处理	最终算法
芬兰题	144.63	6.6908	6.1123	5.1012

在极难的题目下，我们发现 DLX 算法对比暴力法有巨大的优势。因此我

们的策略仍以 DLX 算法为求解核心。

注意到，在比较简单的题目中，纯暴力法可以优化得非常快，可达0.331ms。可是一旦消除全局变量，程序进行模块化后，运行速度会下降很大（因为模块化后会涉及很多传参和对象构造，在简单的情形下会严重拖慢程序运行的效率，而在复杂的情形下带来的影响则可以忽略不计）。但是为了程序的清晰简洁起见，我们最终还是选择牺牲在简单数独题上的效率，而采用 oop 的编程方法，换取程序的高鲁棒性，强可读性和高可维护性。

以下是最快的暴力法（涉及了众多全局变量且没有太多的模块）的源代码：

```
//*****************************************************************************************
**************************************
//更新说明 3.0版本在预处理部分加入了列优化，但是这次速度增加并不明显，在一些简单题上甚至比之前的版本慢，但是有些难题会快一倍
//*****************************************************************************************
**************************************

#include<iostream>
#include<algorithm>
#include<cstdlib>
#include<cstring>

using namespace std;

//定义一些即将要用到的变量
int cnt = 0; //待填方格数
int flag = 0; //解数标志
int sudo[10][10];  //存储数独信息
int sudo_ans[10][10]; //暂存数独的一个解
int vis[10][10] = { {0} }; //是否应该被排序的标志，已经被排过或者填有数的不参与排序
int col[10][10] = { {0} }; //存储每列上已填数字的情况吗，0表示可用，1表示不可用
int row[10][10] = { {0} }; //存储每行情况
```

```
int pal[10][10] = { {0} }; //存储每宫情况
int col_n[10] = { 0 }; //每行已填数字数
int row_n[10] = { 0 };
int pal_n[10] = { 0 };
int po_ans[10][10][10]; //每个位置上的候选解
int po_ans_n[10][10] = { {0} }; //每个位置上的候选解数
int po_row_n[10][10] = { {0} }; //每行每个数的候选解数
int po_row[10][10][10]; //每行每个数的候选解，存储的是列号
int po_col_n[10][10] = { {0} }; //相应的列情况 **更新**
int po_col[10][10][10]; //存储的是行号

pair<int, int> s[100];

bool Check(int i, int j, int pal_num, int num) //判断这个位置上能否填num
{
    if (col[j][num] == 1 || row[i][num] == 1 || pal[pal_num][num] == 1)
        return false;
    else
        return true;
}

inline void Fill_block(int i, int j, int pal_num, int num) //有优化余地 或许搜索的时候只需要占位 不用统计
{
    sudo[i][j] = num;
    col[j][num] = 1;
    col_n[j]++;
    row[i][num] = 1;
    row_n[i]++;
    pal[pal_num][num] = 1;
    pal_n[pal_num]++;
    vis[i][j] = 1;
}

void Search(int x, int y, int now) //按照不确定度大小进行搜索
{
    if (now == cnt + 1)
    {
        if (flag == 0)
        {
            flag++;
            memcpy(sudo_ans, sudo, sizeof(sudo));
        }
        else if (flag == 1)
```

```
            {
                cout << "Multiple solutions" << endl;
                for (int i = 1; i <= 9; i++)
                {
                    for (int j = 1; j <= 8; j++)
                    {
                        cout << sudo_ans[i][j] << " ";
                    }
                    cout << sudo_ans[i][9] << endl;
                }
                cout << endl;
                for (int i = 1; i <= 9; i++)
                {
                    for (int j = 1; j <= 8; j++)
                    {
                        cout << sudo[i][j] << " ";
                    }
                    cout << sudo[i][9] << endl;
                }
                exit(0);
            }
        }
        int pal_num = (x - 1) / 3 * 3 + (y - 1) / 3 + 1;
        for (int g = 1; g <= po_ans_n[x][y]; g++)  //***更新***不再遍历所有的数，而是从可能的选项中选择
        {
            int i = po_ans[x][y][g];
            if (!row[x][i] && !col[y][i] && !pal[pal_num][i])
            {
                Fill_block(x, y, pal_num, i); //填上格子
                Search(s[now + 1].first, s[now + 1].second, now + 1);
                row[x][i] = col[y][i] = pal[pal_num][i] = sudo[x][y] = 0;//回溯
            }
        }
    }

    void Preprocess() //**更新**加入了列优化
    {
        //格子候选法：先填满那些只有一个选择的格子
        int filled = 0;
        for (int i = 1; i <= 9; i++)
            for (int j = 1; j <= 9; j++)
            {
                if (sudo[i][j] == 0)
```

```
            {
                int pal_num = (i - 1) / 3 * 3 + (j - 1) / 3 + 1;
                for (int k = 1; k <= 9; k++)
                {
                    if (Check(i, j, pal_num, k))
                    {
                        po_ans_n[i][j]++;
                        po_ans[i][j][po_ans_n[i][j]] = k;
                    }
                }
                if (po_ans_n[i][j] == 1)
                {
                    Fill_block(i, j, pal_num, po_ans[i][j][1]);
                    filled++;
                    cnt--;
                }
                else if (po_ans_n[i][j] == 0)
                {
                    cout << "No solution" << endl;
                    exit(0);
                }
            }
        }

    //行候选法：每个行必定含有1-9的数字，对于还没有填入的数字，搜索每一个格，如果该数字只有一个格子可以填，填之 _可优化余地：或许可以只搜索所有的未填数
    for (int i = 1; i <= 9; i++)
        for (int k = 1; k <= 9; k++)
        {
            if (row[i][k] == 0)
            {
                for (int j = 1; j <= 9; j++)
                    if (sudo[i][j] == 0 && Check(i, j, (i - 1) / 3 * 3 + (j - 1) / 3 + 1, k))
                    {
                        po_row_n[i][k]++;
                        po_row[i][k][po_row_n[i][k]] = j;
                    }
                if (po_row_n[i][k] == 1)
                {
                    Fill_block(i, po_row[i][k][1], (i - 1) / 3 * 3 + (po_row[i][k][1] - 1) / 3 + 1, k);
                    filled++;
                    cnt--;
```

```
                }
                else if (po_row_n[i][k] == 0)
                {
                    cout << "No solution" << endl;
                    exit(0);
                }
            }
        }

    //列候选法：每列必有1-9，原理与行候选相同
    for (int j = 1; j <= 9; j++)
        for (int k = 1; k <= 9; k++)
        {
            if (col[j][k] == 0)
            {
                for (int i = 1; i <= 9; i++)
                    if (sudo[i][j] == 0 && Check(i, j, (i - 1) / 3 * 3 + (j - 1) / 3
+ 1, k))
                    {
                        po_col_n[j][k]++;
                        po_col[j][k][po_col_n[j][k]] = i;
                    }
                if (po_col_n[j][k] == 1)
                {
                    Fill_block(po_col[j][k][1], j, (po_col[j][k][1] - 1) / 3 * 3 + (j
- 1) / 3 + 1, k);
                    filled++;
                    cnt--;
                }
                else if (po_col_n[j][k] == 0)
                {
                    cout << "No solution" << endl;
                    exit(0);
                }
            }
        }


    //宫候选法还没写 后续再说
    if (filled == 0) //无数可填，没有优化余地
        return;

    while (true) //迭代填数，填到填不下去为止
    {
```

```
        int filled_2 = 0;
        for (int i = 1; i <= 9; i++)
            for (int j = 1; j <= 9; j++)
            {
                if (sudo[i][j] == 0)
                {
                    int pal_num = (i - 1) / 3 * 3 + (j - 1) / 3 + 1;
                    for (int k = 1; k <= po_ans_n[i][j]; k++)
                    {
                        if (!Check(i, j, pal_num, po_ans[i][j][k]))
                        {
                            swap(po_ans[i][j][k], po_ans[i][j][po_ans_n[i][j]]);
                            po_ans_n[i][j]--;
                        }
                    }
                    if (po_ans_n[i][j] == 1)
                    {
                        Fill_block(i, j, pal_num, po_ans[i][j][1]);
                        filled_2++;
                        cnt--;
                    }
                    else if (po_ans_n[i][j] == 0)
                    {
                        cout << "No solution" << endl;
                        exit(0);
                    }
                }
            }

        for (int i = 1; i <= 9; i++)
            for (int k = 1; k <= 9; k++)
            {
                if (row[i][k] == 0)
                {
                    for (int b = 1; b <= po_row_n[i][k]; b++)
                    {
                        int j = po_row[i][k][b];
                        if (!(sudo[i][j] == 0 && Check(i, j, (i - 1) / 3 * 3 + (j -
1) / 3 + 1, k)))
                        {
                            swap(po_row[i][k][b], po_row[i][k][po_row_n[i][k]]);
                            po_row_n[i][k]--;
                        }
                    }
```

```
                    if (po_row_n[i][k] == 1)
                    {
                        Fill_block(i, po_row[i][k][1], (i - 1) / 3 * 3 +
(po_row[i][k][1] - 1) / 3 + 1, k);
                        filled_2++;
                        cnt--;
                    }
                    else if (po_row_n[i][k] == 0)
                    {
                        cout << "No solution" << endl;
                        exit(0);
                    }
                }
            }

        for (int j = 1; j <= 9; j++)
            for (int k = 1; k <= 9; k++)
            {
                if (col[j][k] == 0)
                {
                    for (int b = 1; b <= po_col_n[j][k]; b++)
                    {
                        int i = po_col[j][k][b];
                        if (!(sudo[i][j] == 0 && Check(i, j, (i - 1) / 3 * 3 + (j -
1) / 3 + 1, k)))
                        {
                            swap(po_col[j][k][b], po_col[j][k][po_col_n[j][k]]);
                            po_col_n[j][k]--;
                        }
                    }
                    if (po_col_n[j][k] == 1)
                    {
                        Fill_block(po_col[j][k][1], j, (po_col[j][k][1] - 1) / 3 * 3
+ (j - 1) / 3 + 1, k);
                        filled_2++;
                        cnt--;
                    }
                    else if (po_col_n[j][k] == 0)
                    {
                        cout << "No solution" << endl;
                        exit(0);
                    }
                }
            }
```

```
        if (filled_2 == 0)
            return;
    }
}

void Read()
{
    bool legal = true;
    char a;
    for (int i = 1; i <= 9; i++)
        for (int j = 1; j <= 9; j++)
        {
            cin >> a;
            if (a == '-')
            {
                sudo[i][j] = 0;
                cnt++;
            }
            else
            {
                sudo[i][j] = a - '0';
                int pal_num = (i - 1) / 3 * 3 + (j - 1) / 3 + 1; //计算宫号
                //判断合法性
                if (!Check(i, j, pal_num, sudo[i][j]))
                    legal = false;
                //占位
                col[j][sudo[i][j]] = 1;
                col_n[j]++;
                row[i][sudo[i][j]] = 1;
                row_n[i]++;
                pal[pal_num][sudo[i][j]] = 1;
                pal_n[pal_num]++;
                vis[i][j] = 1;
            }
        }

    //判断合法性
    if (!legal)
    {
        cout << "No solution" << endl;
        exit(0);
    }
}
```

```
void Makeorder()
{
    for (int k = 1; k <= cnt; k++)
    {
        int Max = -1, px, py;
        for (int i = 1; i <= 9; i++)
            for (int j = 1; j <= 9; j++)
            {
                int pal_num = (i - 1) / 3 * 3 + (j - 1) / 3 + 1;
                if (row_n[i] + col_n[j] + pal_n[pal_num] > Max && !vis[i][j])
                {
                    Max = row_n[i] + col_n[j] + pal_n[pal_num], px = i, py = j;
                }
            }
        int pal_num = (px - 1) / 3 * 3 + (py - 1) / 3 + 1;
        s[k] = make_pair(px, py);
        //找到可能性最少的点，并假设他已经填好，继续找下一个
        row_n[px]++, col_n[py]++, pal_n[pal_num]++;
        vis[px][py] = 1;
    }
}

int main()
{
    //读取数据
    Read();

    //对数独进行预处理
    Preprocess();

    //对待填的格子进行排序，先填那些限制条件多的格子
    Makeorder();

    //进行搜索
    Search(s[1].first, s[1].second, 1);

    if (flag == 0)
    {
        cout << "No solution" << endl;
    }
    else if (flag == 1)
    {
        cout << "OK" << endl;
        for (int i = 1; i <= 9; i++)
```

```
        {
            for (int j = 1; j <= 8; j++)
            {
                cout << sudo_ans[i][j] << " ";
            }
            cout << sudo_ans[i][9] << endl;
        }
    }
    return 0;
}
```

从上述代码中可以看到，在省略了传参和模块化后虽然运行速度会有明显的提升，但是程序的可读性和可维护性都比较差。我们认为解数独的程序不是越快越好，像这样的代码虽然可以很快地解数独，但是不满足工程的要求。我们希望在工程性和效率之间找一个平衡点，既能让程序相对高效地运行，同时又让程序的层次逻辑相对清晰，易维护，满足工程性的要求，因此我们选择了最终的 oop 版本。