

**课程设计报告**

**题目：孔明棋游戏求解程序设计**

**课程名称： 命令式计算原理**

**专业班级： CS2205**

**学 号： U202215494**

**姓 名： 廖奎**

**指导教师： 李开**

**报告日期： 2024.11.16**

**计算机科学与技术学院**

**目 录**

**1　课程设计任务 1**

1.1 简介 1

1.2 设计内容 2

1.3 设计要求 3

**2　系统需求分析与总体设计 4**

2.1系统需求分析 4

2.2系统总体设计 4

**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 10**

4.1系统实现 10

4.2系统测试 11

**5　总结 20**

**6　体会 21**

**附录 22**

# 1　课程设计任务

## 1.1 简介

先简要介绍孔明棋游戏，主要是游戏规则，然后表明本课程设计需要用计算机对孔明棋游戏棋局进行求解。

孔明棋是一种单人桌游，它的目标是只留下一颗棋子在棋盘上，而其他的都移除。它的初始棋盘包括一些空洞和一些被棋子占据的洞。

一次操作是将一个棋子跳过前/后/左/右的另一个棋子（不可以对角跳），被跳过的棋子便从棋盘上移除。比如下图1-1左侧的棋盘，共有4种可能的操作，都是跳到正中央的位置。选取上方的棋子跳到正中央，将会得到右侧棋盘。右侧棋盘有3种可能的操作。

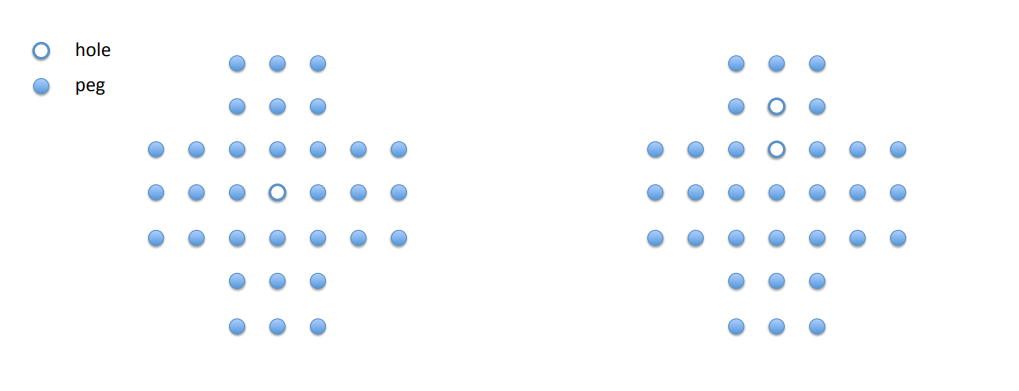


图1-1 典型的孔明棋棋盘

这个游戏的目标是只剩下一个棋子。在标准英式棋盘（上图左侧棋盘）中，我们从32颗棋子开始，所以任何解法都是31步的（每步移除一个棋子）。

本课程设计要求使用计算机程序对孔明棋游戏棋局进行求解。使用C0语言编程，通过读取初始棋盘数据，模拟移动吃棋操作，递归地调用求解函数以求获得最好的结果，即剩余棋子数最少，如果只剩一个则获胜了。

设计程序时以提高程序运行效率为重，考虑数据如何存储和表示、使用何种结构、使用哪种移动策略等问题，并不断尝试不同的棋盘数据，最终写出正确高效的程序。

## 1.2 设计内容

**任务一：解一个确定的孔明棋局**

对于一个棋盘只需要尝试一个操作（如果存在），并最终达到一个赢局（只剩一个棋子）或一个输局（还有多于一个棋子，但是没有合法操作了）。这意味着对于一个确定的棋局（每次最多只有一个操作合法），代码将在解存在时找出这个唯一的解。对于一个不确定的棋局（可以解，但是某些时候会有多于一个合法操作），代码可能找到也可能找不到解。

定义函数int peg\_solve(board B, stack S)来尝试得到一个给定棋盘的确定解。返回的整数是棋局结束时剩下的棋子个数。如果peg\_solve返回1，那么求解成功。在这个情况下，S便是解（操作组成的栈）。如果peg\_solve返回大于1的整数，那么求解失败。棋盘上还剩下2颗及以上棋子。这个情况下，S可以是任意的。

为了编写peg\_solve，需要先写一个递归的辅助函数solve，它将被peg\_solve调用。这个辅助函数的参数包括棋盘、当前的操作堆栈、棋盘上剩下的棋子个数。在有解时辅助函数solve返回1；它也应该要在堆栈S中增加操作。

这个辅助函数的策略是考虑当前棋局的所有合法操作。这可以写在又一个附加的辅助函数find\_all\_move中，这个附加的辅助函数产生所有可能的下一步操作，返回一个包括这些操作的新堆栈。

接下来，选择堆栈中的第一个操作（忽略其他可能的下一步操作），递归地尝试它，看看下一个棋盘是否会产生解。如果下一个棋盘产生了解，则说明当前棋盘也有解，确保操作按照正确的顺序压入堆栈。

**任务二：解一个不确定的孔明棋局**

在任务一中，只解决了确定的孔明棋局：对于一个处在当前状态的棋盘，生成一个所有可能的下一步组成的堆栈，选择其中一个，然后在产生的下一个棋盘上继续玩孔明棋（这是递归的）。用这个方法，如果我们在求解过程中走到死路，很不幸——我们只能认输并返回棋盘上剩下的棋子个数。

利用回溯的策略，我们可以做出改进。在回溯的过程中，我们从某种当前状态的棋盘开始，先尝试一个可能的第一步操作，让棋盘进入一个新的状态。如果这个操作不成功（产生的棋盘是无解的），我们就回到当前状态，然后尝试另一个可能的第一步操作。第一步始终产生不了解，并且还有其他可能的第一步时，我们就不断重复这个过程。如果所有可能的第一步都产生不了解，那么我们就知道当前状态是无解的。此时返回前一个状态，然后尝试任何剩下的第一步操作，以此类推。回溯也是递归的。

**任务三：记忆化**

任务2中的回溯搜索的一个基本问题是，可能重复访问同一个不可解棋盘很多很多次。在本任务中使用哈希表来将solve函数记忆化。

与把peg\_solve函数当成一个棋局和它可能达成的最小棋子数之间的一个联系类似，哈希表也是同样的一种联系。一旦发现一个棋盘B是不可解的，那它就被存在哈希表中，同时存入它剩下的棋子数量。

当我们尝试解一个棋盘式，我们先检查它是否已经被记录为无解；如果确实是这样，我们马上返回哈希表中存储的答案。注意哈希表中存储的所有棋盘都是输局，所以move组成的堆栈无需存入哈希元素。

如果我们发现一个赢局，无需把它存入哈希表，因为无需继续搜索了。

## 1.3 设计要求

任务一：递归地尝试棋盘所有可能的解，判断一个确定的棋盘是否有解。

任务二：有回溯地递归尝试棋盘所有可能的解，无解时回到前一个状态，能够判断一个不确定棋盘是否有解。

任务三：使用哈希表记录无解的棋盘状态，递归时在表中寻找，找到直接返回表中存储的答案，提高效率。

# 2　系统需求分析与总体设计

## 2.1系统需求分析

实现目标：从文本文件中读入棋盘数据，能较快判断棋盘是否有解，有解则输出移动策略，无解则返回最终能剩余的最少棋子数量。

处理事务：能处理符合表述要求的棋盘数据，包括确定棋盘和不确定棋盘。

事务处理流程图如图2-1所示。

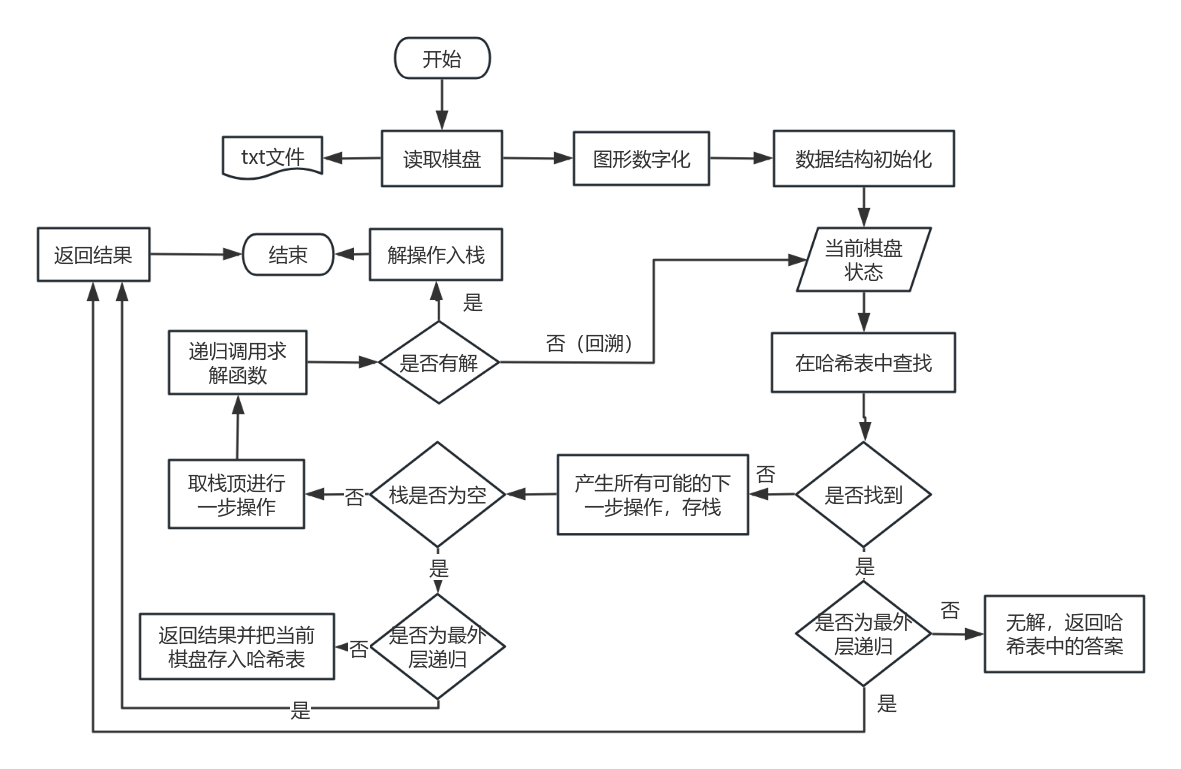


图2-1 事务处理流程图

## 2.2系统总体设计

系统具有的功能模块：

1. 主控制模块：由peg-main.c0实现，用于读取棋盘，调用求解函数，输出求解结果。
2. 数据结构模块：由peg-util.c0、ht.c0、stacks.c0组成。peg-util.c0包含对棋盘的多种操作，如格式检查、棋子数计算、初始化、打印输出、字符串棋盘转化成数字棋盘等等。ht.c0包含哈希表检验、创建、计算大小、查找、扩容等哈希表相关操作。stacks.c0包含栈的结构定义、栈判断、大小、栈为空、创建新栈、入栈出栈等操作。
3. 客户端模块：由peg-client.c0实现，包含对栈元素、哈希表元素数据类型的定义以及部分哈希相关函数。
4. 求解模块：peg1.c0、peg2.c0、peg3.c0，对孔明棋盘进行求解。
5. 数据模块：\*.txt文本文件，存储棋盘数据。

完整的系统模块结构图如图2-2所示。

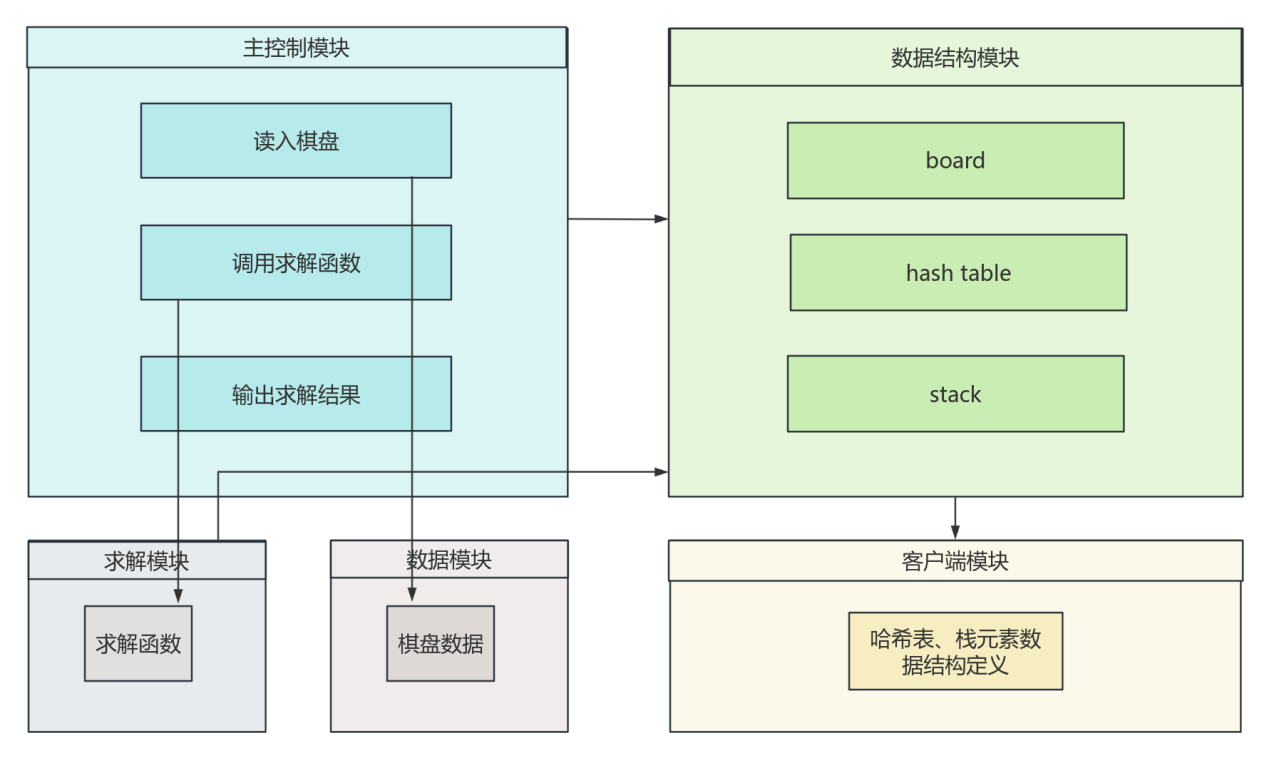


图2-2 系统模块结构图

# 3　系统详细设计

## 3.1有关数据结构的定义

系统中包含的数据：

1. 棋盘：一个大小为8\*8=64的整数数组，数组元素有3个值：-1，表示墙；0，表示该位置是没有放棋子的洞；1，表示放了棋子的洞。从棋盘左上角开始按行优先从0开始编号。
2. 移动操作：typedef int move，使用整数的四元组表示移动操作，把棋子在操作前、后的行号、列号压缩进单个int，每8位存储一个行列信息（行列号0~7，8位二进制数足够表示）。
3. 棋盘状态：棋盘由64个整数组成，为了提高效率，需要把当前棋盘状态压缩后存入哈希表中。考虑到所有棋盘状态都是由最初读入的棋盘经过操作变化得来，可知值为-1的元素不会改变，因为棋子不能移动到墙里面。那么比较两个棋盘是否相同时，值为-1的位置一定一一对应，那么我们将-1变成0对棋盘的比较结果不会有任何影响，但是这对于压缩成整型却至关重要。现在得到只有0和1的64个整数，显然可以利用位操作压缩进2个int。
4. 栈链表结点：包含栈元素、下一结点指针。栈元素类型为移动操作，下一结点指针为栈链表结点指针。
5. 栈：包含栈顶结点、栈底结点、栈大小。类型分别是栈链表结点指针、栈链表结点指针、整型。
6. 哈希链表结点：包含哈希表元素、下一结点指针。类型分别是棋盘状态加最少剩余棋子数、哈希链表结点指针。
7. 哈希表：包含大小、容量、哈希数组。类型分别是整型、整型、哈希链表结点指针数组。

系统数据、数据项及数据类型表如表3-1所示。

表3-1 数据、数据项及数据类型表

|  |  |  |
| --- | --- | --- |
| **数据** | **数据项** | **数据类型** |
| 棋盘 | 棋盘布局 | 64个元素的整数数组 |
| 移动操作 | 移动前行号 | 1字节 |
| 移动前列号 | 1字节 |
| 移动后行号 | 1字节 |
| 移动后列号 | 1字节 |
| 棋盘状态 | 棋盘布局 | 2个整型 |
| 栈链表结点 | 栈元素 | 移动操作 |
| 下一结点指针 | 栈链表结点指针 |
| 栈 | 栈顶结点 | 栈链表结点指针 |
| 栈底结点 | 栈链表结点指针 |
| 栈大小 | 整型 |
| 哈希链表结点 | 哈希表元素 | 棋盘状态加上最少剩余棋子数 |
| 下一结点指针 | 哈希链表结点指针 |
| 哈希表 | 大小 | 整型 |
| 容量 | 整型 |
| 哈希数组 | 哈希链表结点指针数组 |

哈希表的元素是棋盘状态加上最少剩余棋子数，关联图如图3-1所示。

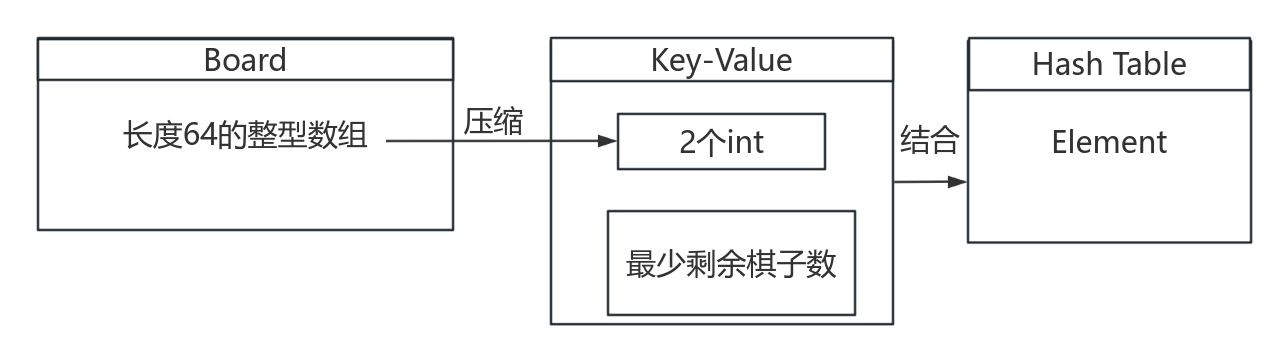


图3-1 哈希表与棋盘关联图

## 3.2 主要算法设计

在peg1.c0中，求解函数int peg\_solve(board B, stack S)，输入原始棋盘和空栈S，调用递归的辅助函数int solve(board B,stack S,int number\_pegs)。solve函数先使用一个辅助函数stack find\_all\_move(board B)，获取当前棋盘所有合法操作，存入新栈中，栈为空说明当前没有能移动的棋子了，直接返回当前的number\_pegs参数。栈不为空则读取栈顶一个元素进行移动操作，得到了一个新的棋盘，递归对新棋盘同样进行上述操作，然后把操作放入栈S中，返回得到的最小剩余棋子数。注意递归时参数number\_pegs减一，因为进行了一次移动操作，减少了一个棋子。流程图如图3-2所示。

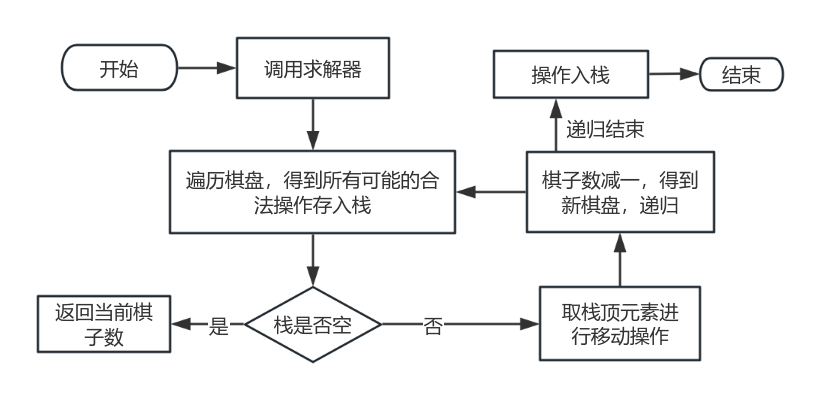


图3-2 peg1.c0流程图

peg2.c0在peg1.c0基础上加入回溯策略进行改进，不是只取栈顶一个操作，而是遍历执行当前棋盘全部的合法操作，每个操作执行后递归，得到经过该步操作后最终能得到的最少棋子数ans，和所有操作中得到的最少棋子数min\_ans进行比较，ans小则更新min\_ans，即确定当前此步移动是目前的最优移动策略。然后判断ans是否为1，为1说明经过这步后直接得到了一个解，那么将该移动存入栈S中，不难想到里层递归也会存在一个移动操作得到ans为1（因为外层的1就是里层返回值），因此里层递归会更先地把移动存栈S，所以推导出S栈底是解步骤中的最后一个移动操作，栈顶是第一个操作，从S中把元素一个一个取出得到的正好是得到解的棋子正确移动顺序。如果没返回1说明这个操作无解，这时需要进行回溯，也就是使棋盘中三个改变了的棋子回到上一个状态，然后进入下一个循环，取出栈中下一个合法的移动操作。循环退出说明初始棋盘以及后续得到的棋盘所有合法的操作都进行了，最终能得到的最少棋子数一定是min\_ans。

peg2.c0流程图如图3-3所示。

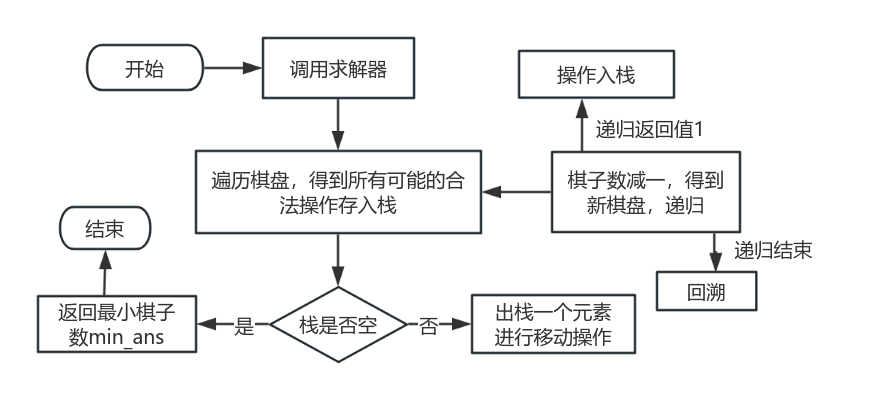


图3-3 peg2.c0流程图

peg3.c0在peg2.c0基础上加入哈希表来将solve函数记忆化，每次把无解棋局连带该棋局能得到的最少棋子数存入哈希表中，每得到一个新的棋局，在哈希表中查找，找到说明该棋盘无解，直接返回对应的最少棋子数，未找到则进行后续递归操作，从而实现了记忆化搜索。

代码中solve函数增加了五个参数，一个哈希表存储无解的棋盘状态和最小棋子值，两个整型保存压缩后的棋盘状态，用于在哈希表中查找，另外两个整型数组保存的是find\_all\_move函数中需要用到的上下左右偏移量，目的是避免数组反复创建销毁，导致效率降低。

peg3.c0流程图如图3-4所示。

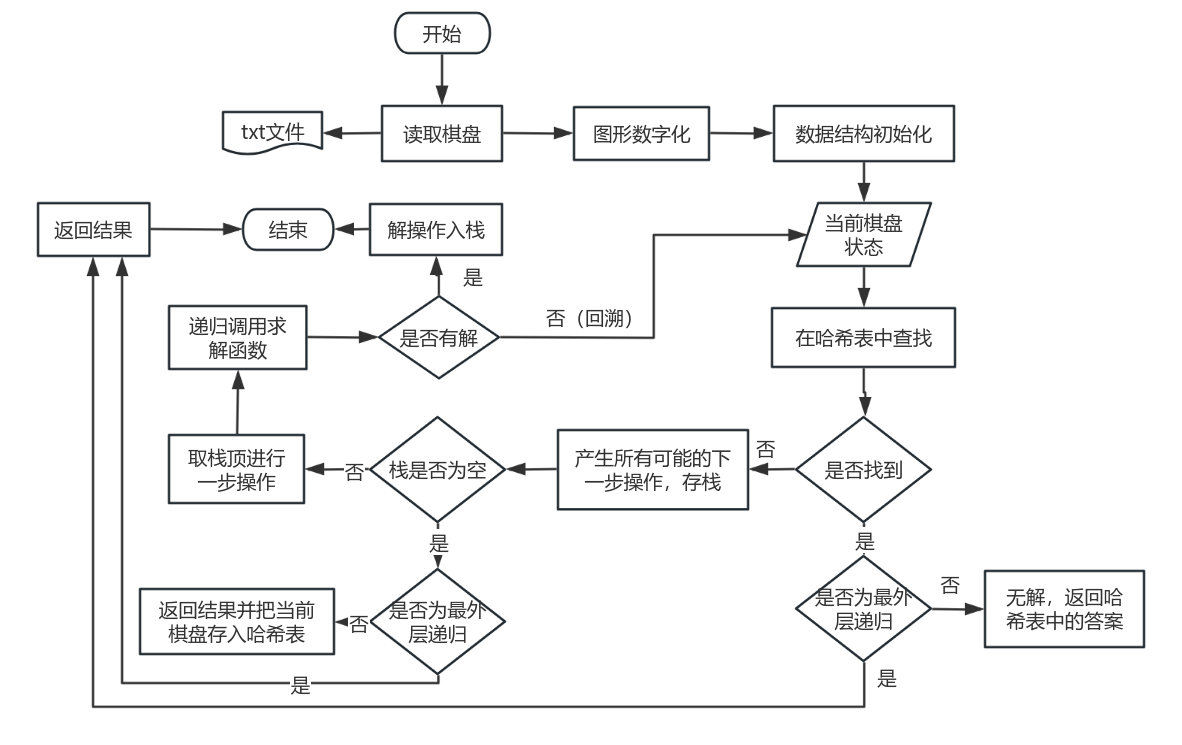


图3-4 peg3.c0流程图

# 4　系统实现与测试

## 4.1系统实现

1. 系统环境

程序在Ubuntu 22.04.5 LTS上运行（WSL2），代码在Visual Studio Code上连接Ubuntu终端编写。

1. 数据类型

棋盘：typedef int[] board;

移动操作：typedef int move;

棋盘状态：int i1; int i2

栈元素：typedef int stackelem;

哈希表键值：

struct two\_ints {

int i1;

int i2;

int best\_num\_pegs;

};

typedef struct two\_ints\* htelem;

typedef struct two\_ints\* htkey;

1. 函数说明
2. peg1.c0包含函数：

int row\_start(move m)、int col\_start(move m)、int row\_end(move m)、int col\_end(move m)，用于从一个整型move得出四个整型的棋子移动位置；

stack find\_all\_move(board B) 找出当前棋盘所有合法移动；

int solve(board B,stack S,int number\_pegs) 递归的辅助函数

int peg\_solve(board B, stack S) 求解函数

1. peg2.c0包含函数与peg1.c0相同。
2. peg3.c0与peg1.c0不同的是solve函数增加了5个参数int solve(board B,stack S,int number\_pegs,ht H,int i1,int i2,int[] dx,int[] dy)。find\_all\_move也增加了2个参数，stack find\_all\_move(board B,int[] dx,int[] dy)，因为加入了哈希表，同时避免反复创建偏移数组。
3. 其它函数都是栈、哈希表中常用的函数，不在此赘述。

函数如何实现系统各模块的功能，以及函数间的调用关系已在系统总体设计中给出。

## 4.2系统测试

三个程序peg1.c0、peg2.c0、peg3.c0是逐层递进的关系，peg1.c0可以求解一个确定的孔明棋盘，peg2.c0在前者的基础上增加回溯策略还能求解简单的不确定的棋盘，但遇到复杂的棋盘效率很低。peg3.c0使用了哈希表提高递归效率，但不同的搜索顺序、策略、棋盘用例对搜索效率有很大影响，在本实验中我通过改变选择move的顺序来探究棋盘求解效率。三种选择move的策略：

peg3.c0 ：每次找到一个棋子，查看它是否能向各个方向跳跃（上下左右）

peg3-c2.c0：每次找到一个棋子，查看它能否在各个方向被跳过（上下左右）

peg3-c3.c0：每次找到一个空格，查看它能否在各个方向被跳入（上下左右）

然后主要对比三种策略对不确定棋盘english.txt、french1.txt、french2.txt、french3.txt的求解效率，评判指标为求出解时哈希表的大小，越小说明进行的无用递归少，用时更短。

还要考虑哈希函数对程序效率的影响，选取效率高的哈希函数。

1. peg1测试

对确定的孔明棋棋局german.txt进行测试，测试指令：

% cc0 -d -w -o peg1 peg-client.c0 lib/\*.c0 peg1.c0 peg-main.c0

% ./peg1 german.txt

测试结果如图4-1所示。

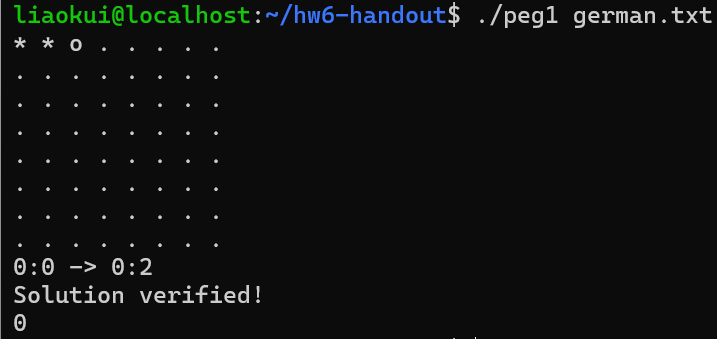


图4-1 peg1求解确定棋盘测试图

尝试对不确定棋局english.txt求解，指令如下，测试结果见图4-2。

% ./peg1 german.txt

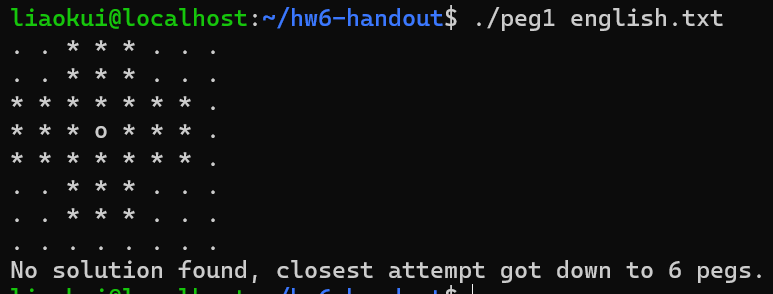


图4-2 peg1求解不确定棋盘测试图

可见只有递归而无回溯策略时最终走到6个棋子时便无法操作了。

1. peg2测试

peg2在peg1基础上增加了回溯策略，可以求解不确定棋盘english.txt。求解german.txt和english.txt指令如下，测试结果分别如图4-3、4-4所示。

% cc0 -d -w -o peg2 peg-client.c0 lib/\*.c0 peg2.c0 peg-main.c0

% ./peg2 german.txt

% cc0 -w -o peg2 -r unsafe -c-O2 peg-client.c0 lib/\*.c0 peg2.c0 peg-main.c0

% ./peg2 english.txt

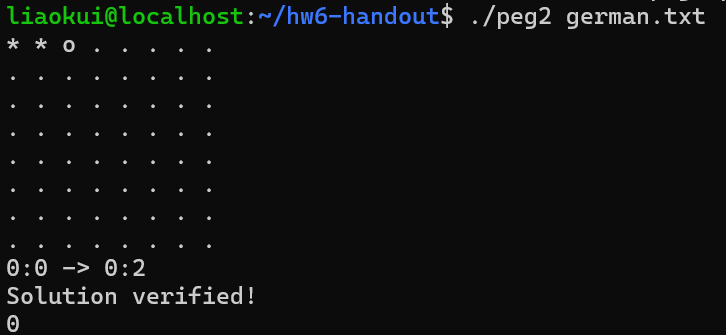


图4-3 peg2求解确定棋盘测试图

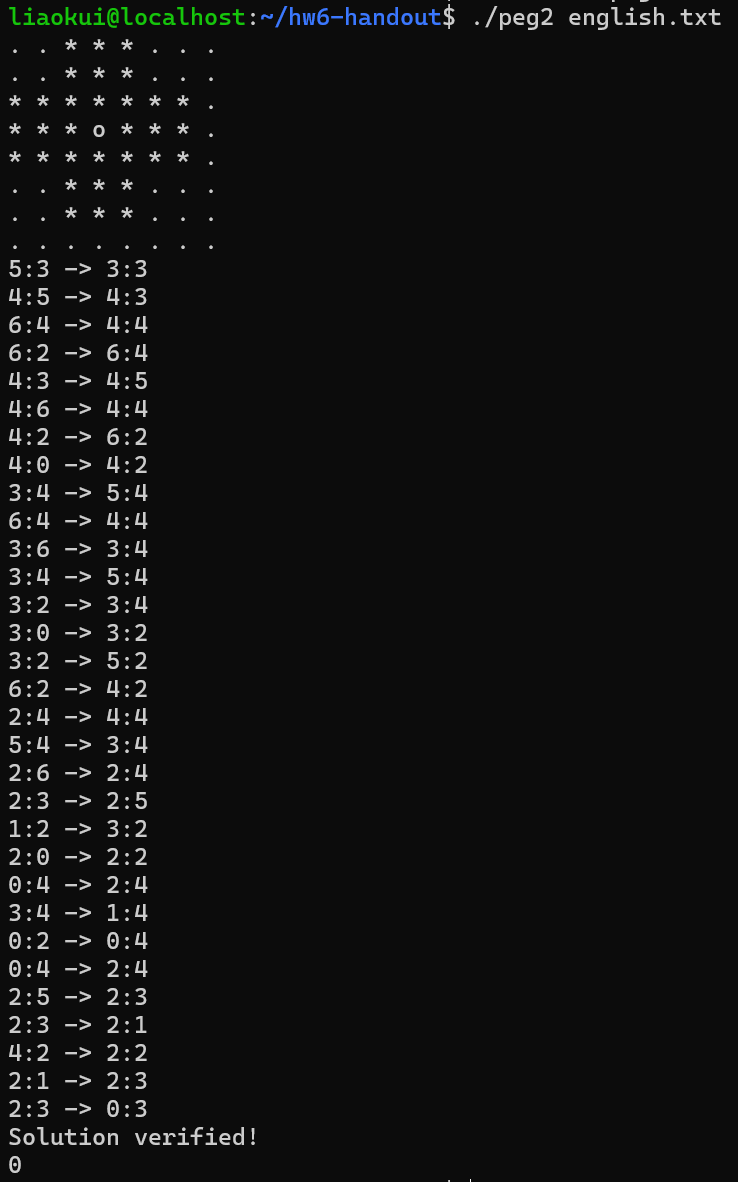


图4-4 peg2求解不确定棋盘测试图

可见peg2功能正确，能求解确定的棋盘和不确定的棋盘。现在考虑求解复杂的不确定棋盘french1.txt，测试图如图4-5所示，发现很长时间都无法得到答案，但并未报错，可见peg2程序没有出错但效率过低。

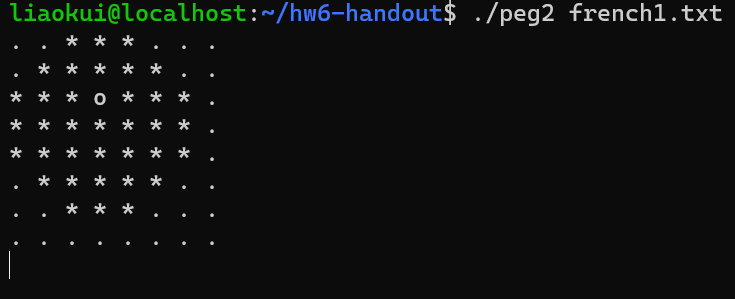


图4-5 peg2求解不确定复杂棋盘测试图

1. peg3测试

在peg2的基础上使用哈希表，提高求解效率。求解确定的棋盘测试图如图4-6所示。

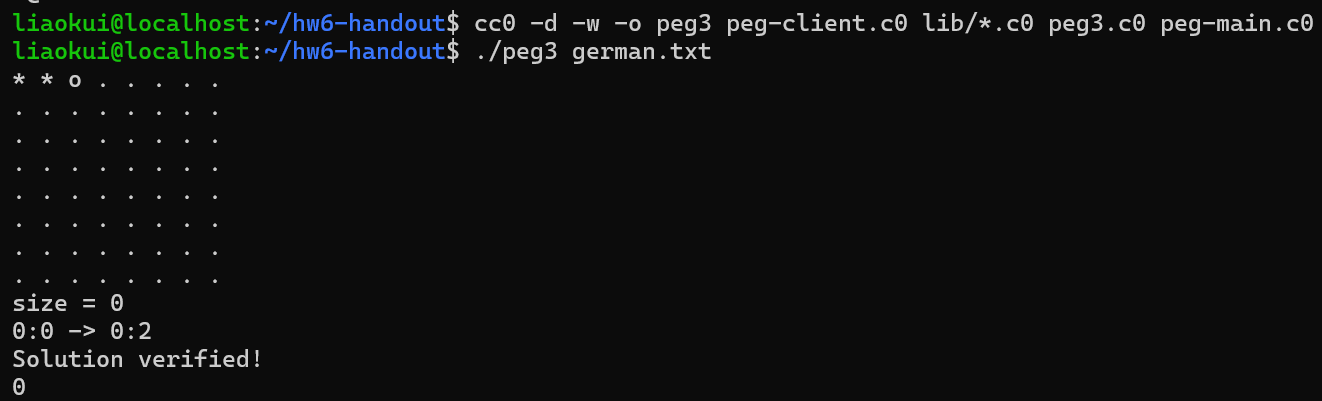


图4-6 peg3求解确定的棋盘测试图

接下来对peg3.c0 、peg3-c2.c0、peg3-c3.c0求解进行，查看三种策略求解不确定棋盘english.txt、french1.txt、french2.txt、french3.txt的效率。

peg3.c0运行测试图如图4-7、4-8所示。

cc0 -w -o peg3 -r unsafe -c-O2 peg-client.c0 lib/\*.c0 peg3.c0 peg-main.c0

./peg3 english.txt

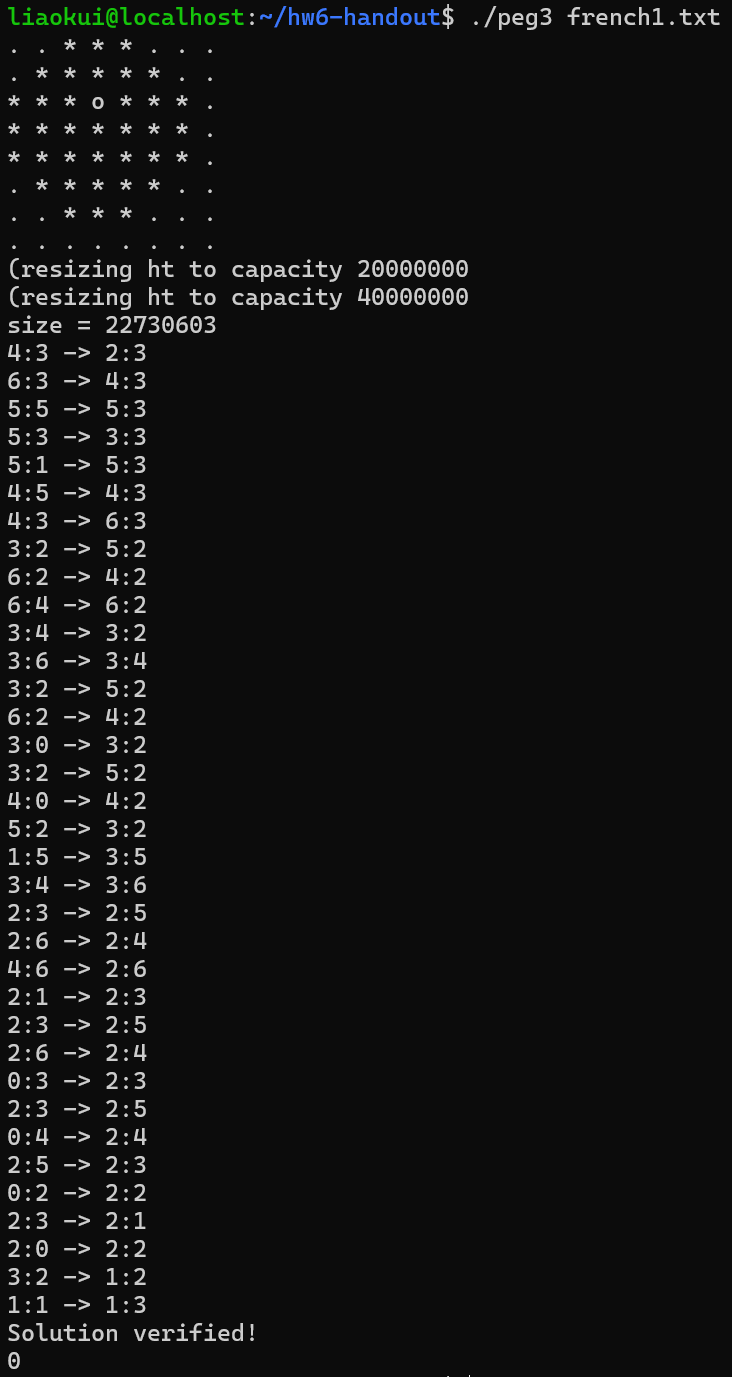
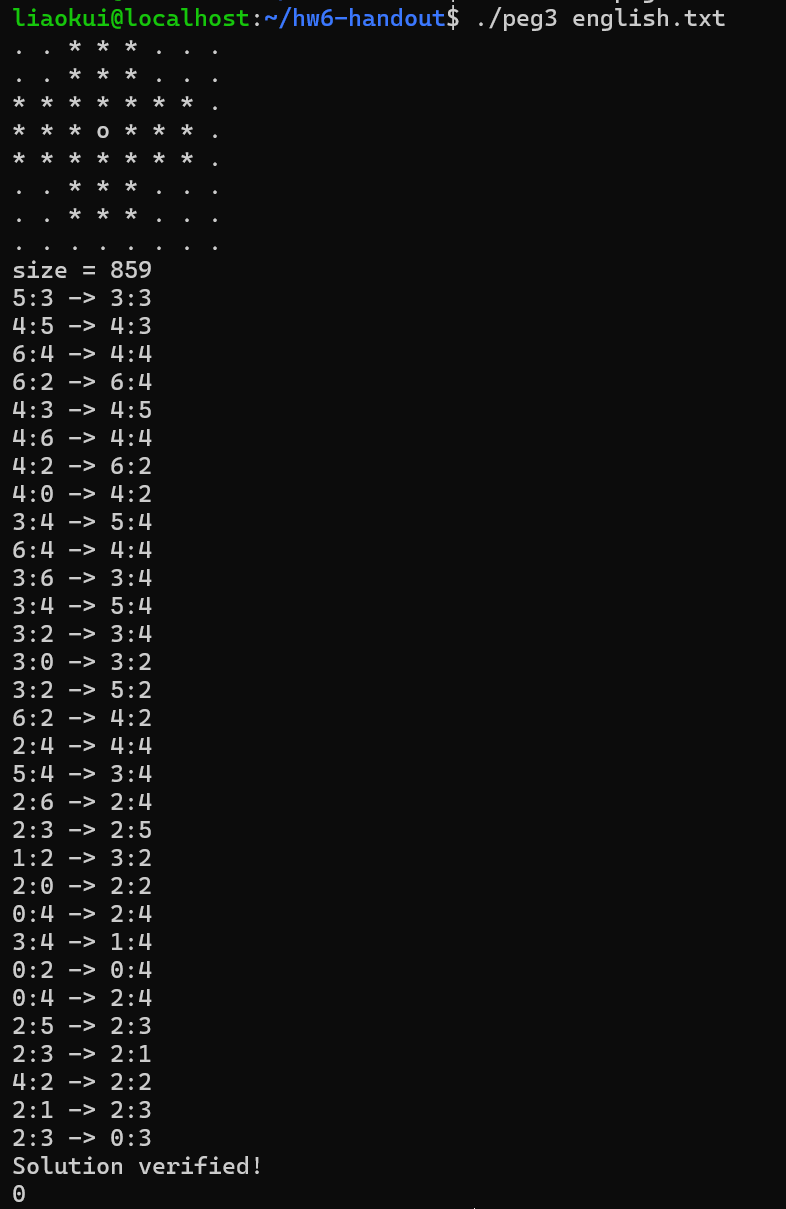


图4-7 peg3求解测试图1

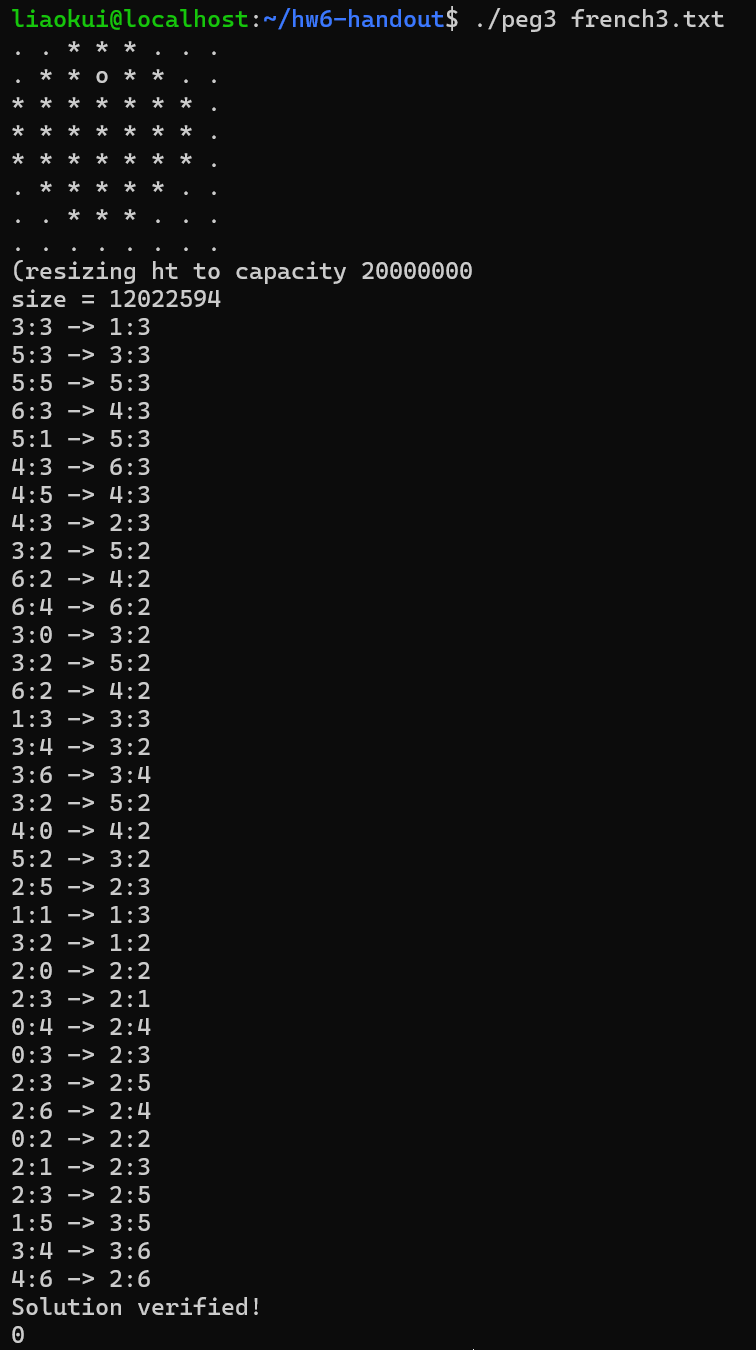
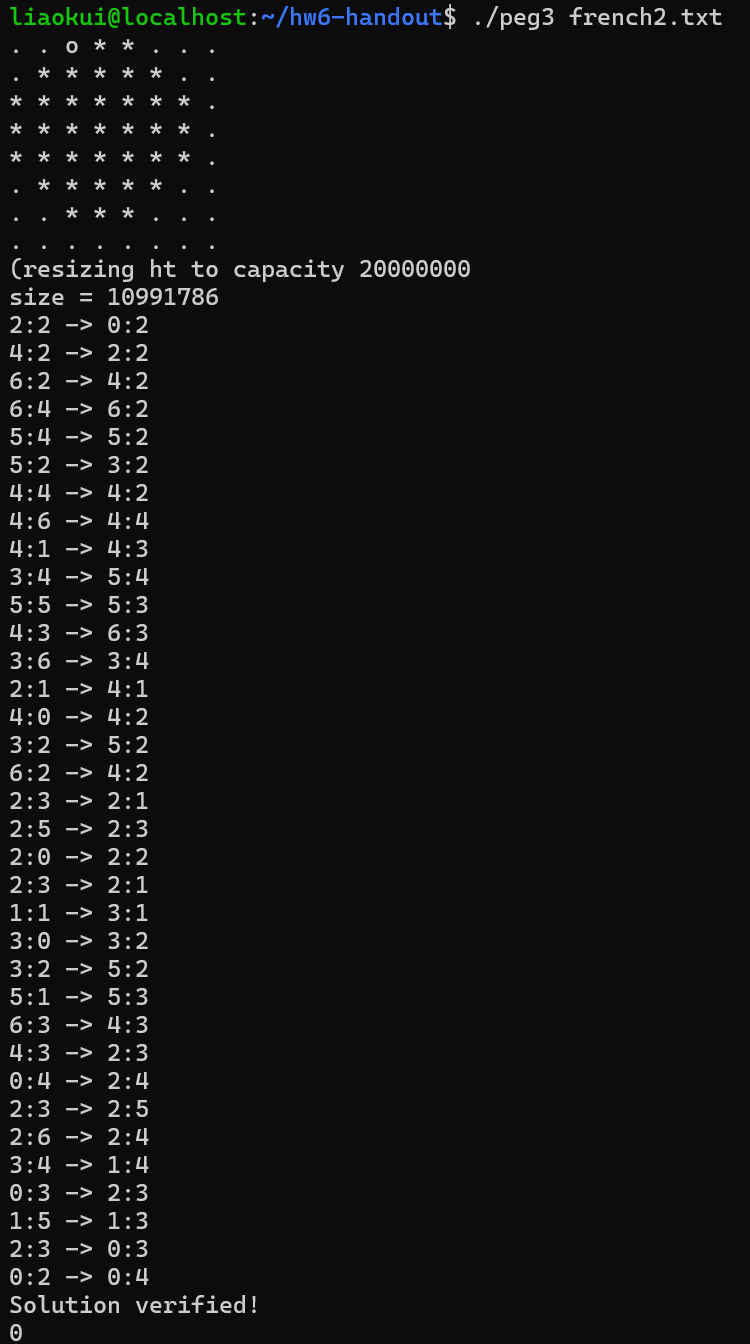


图4-8 peg3求解测试图2

peg3-c2.c0运行测试图如图4-9、4-10、4-11所示。

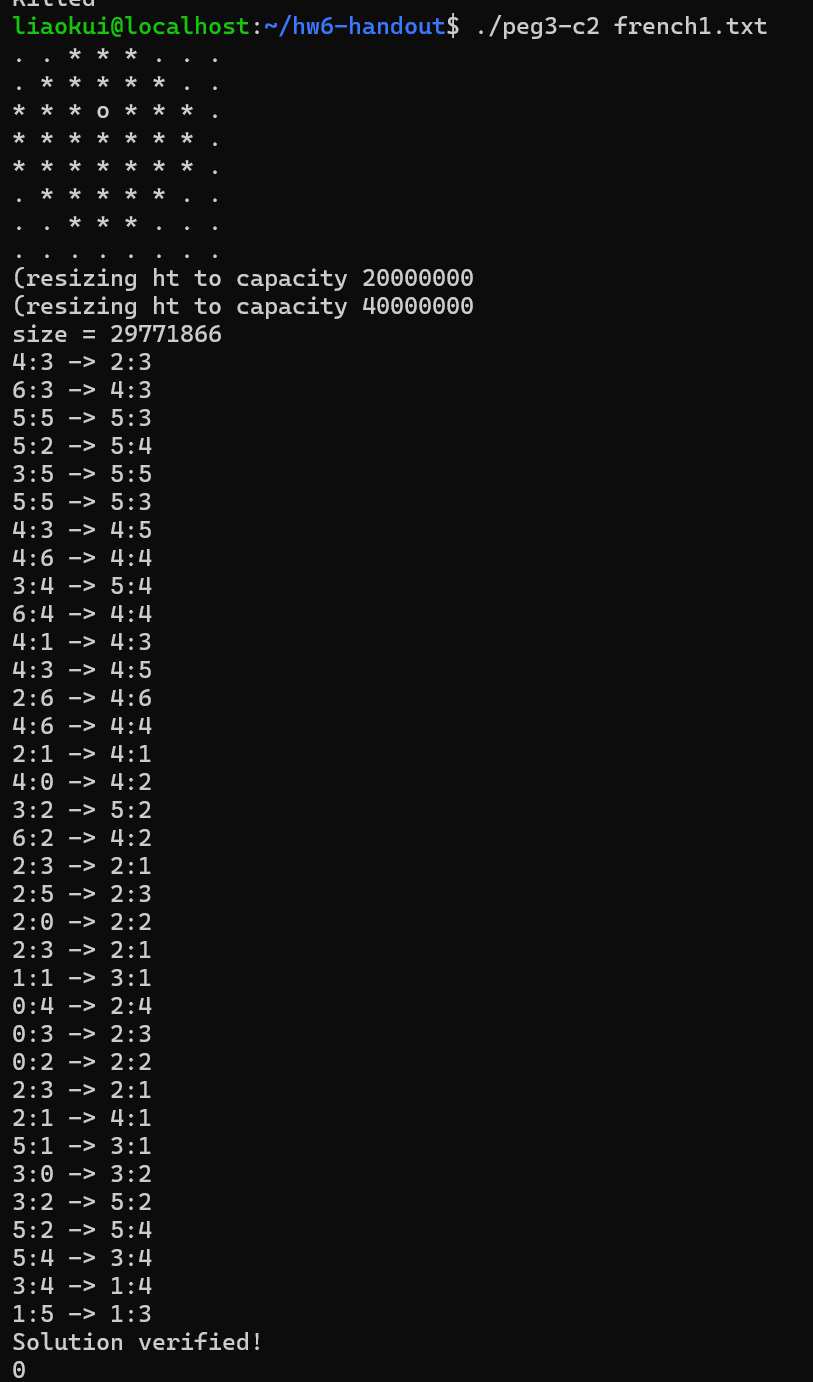
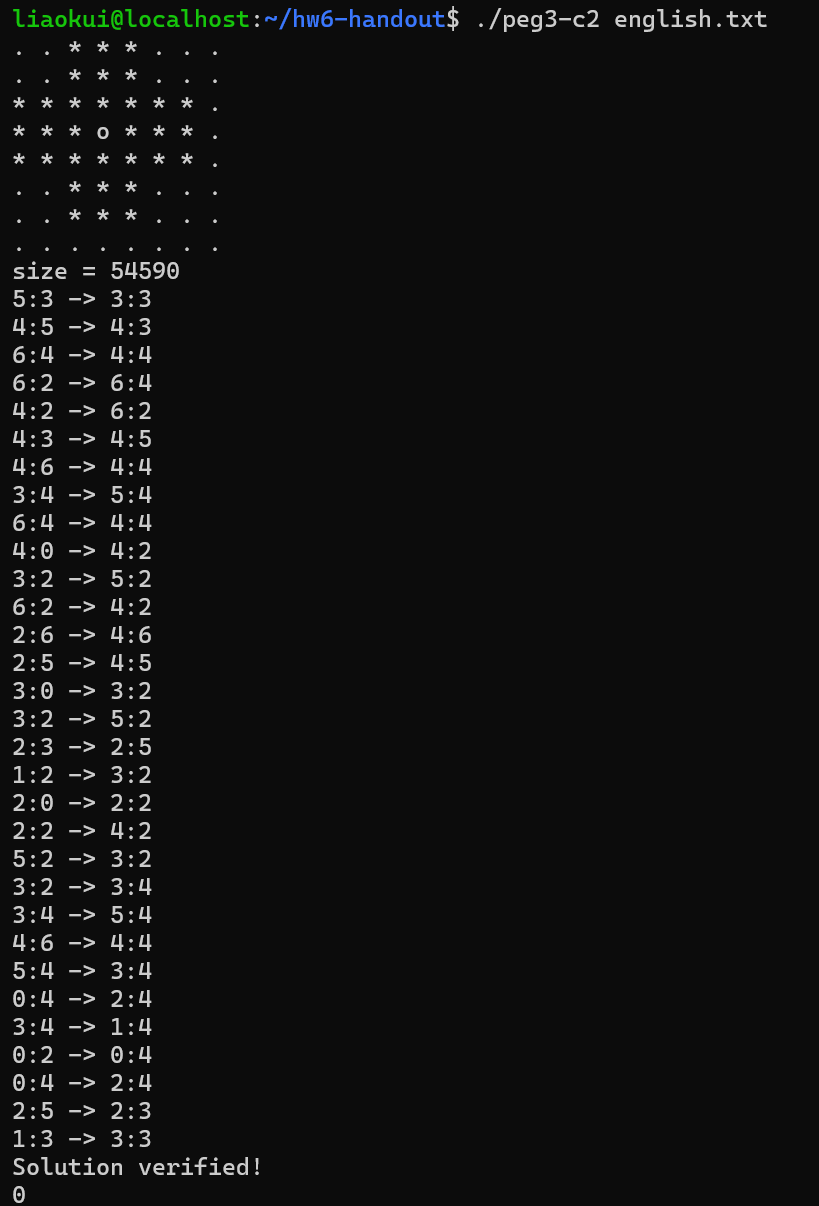


图4-9 peg3-c2求解测试图1

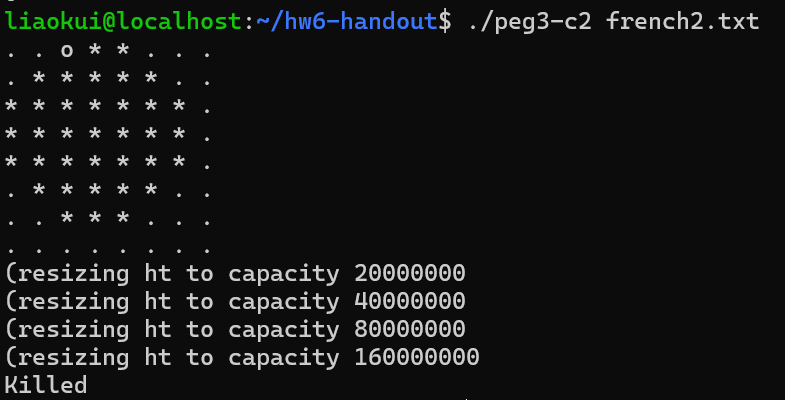


图4-10 peg3-c2求解测试图2

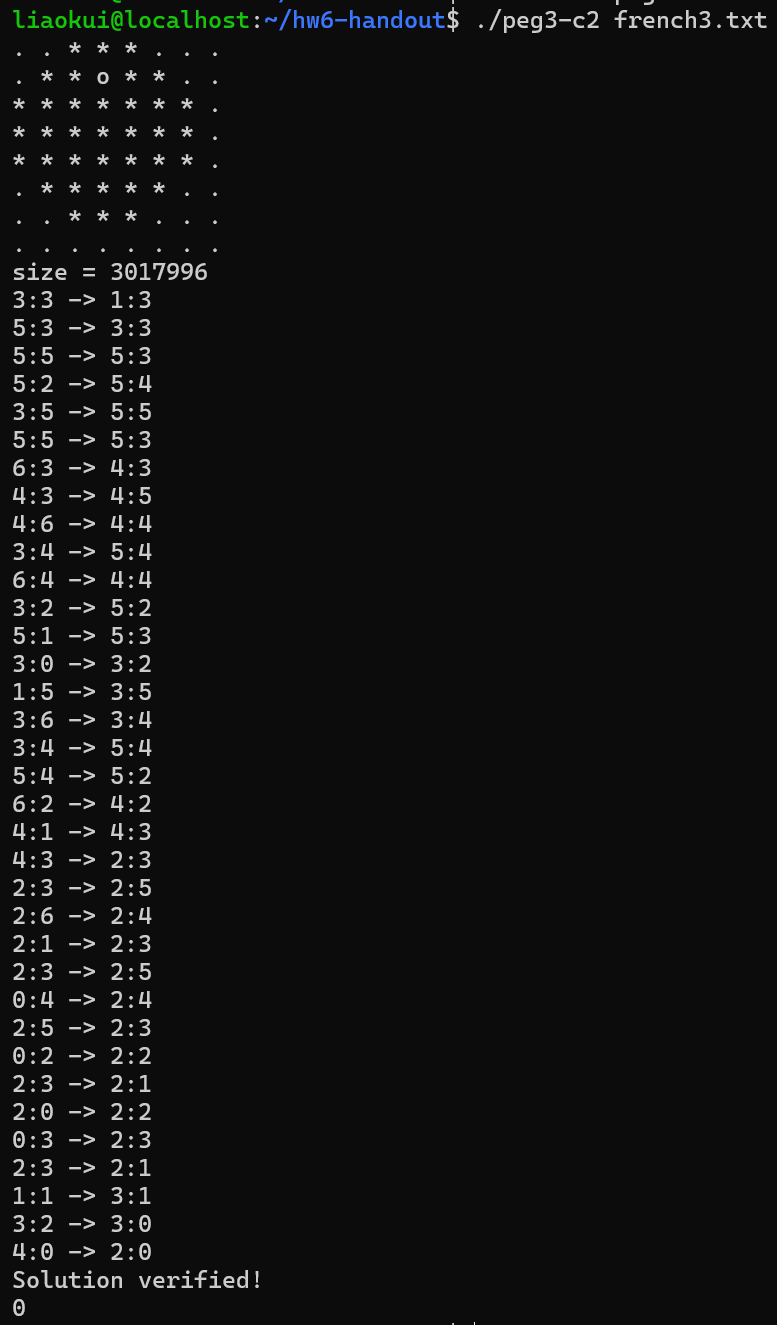


图4-11 peg3-c2求解测试图3

peg3-c3.c0运行测试图如图4-12、4-13所示。

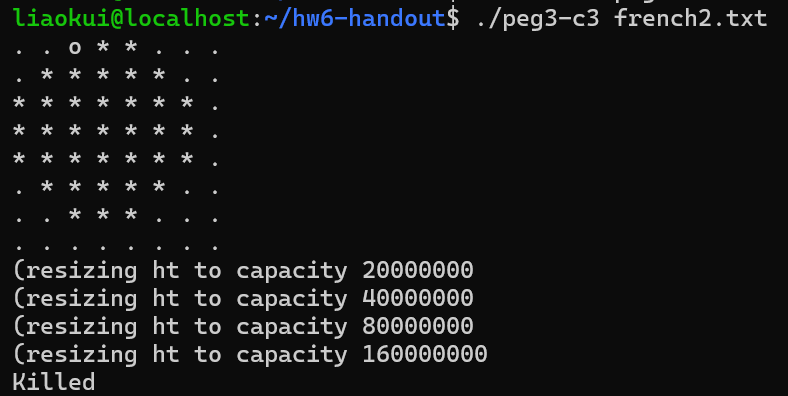


图4-12 peg3-c3求解测试图1

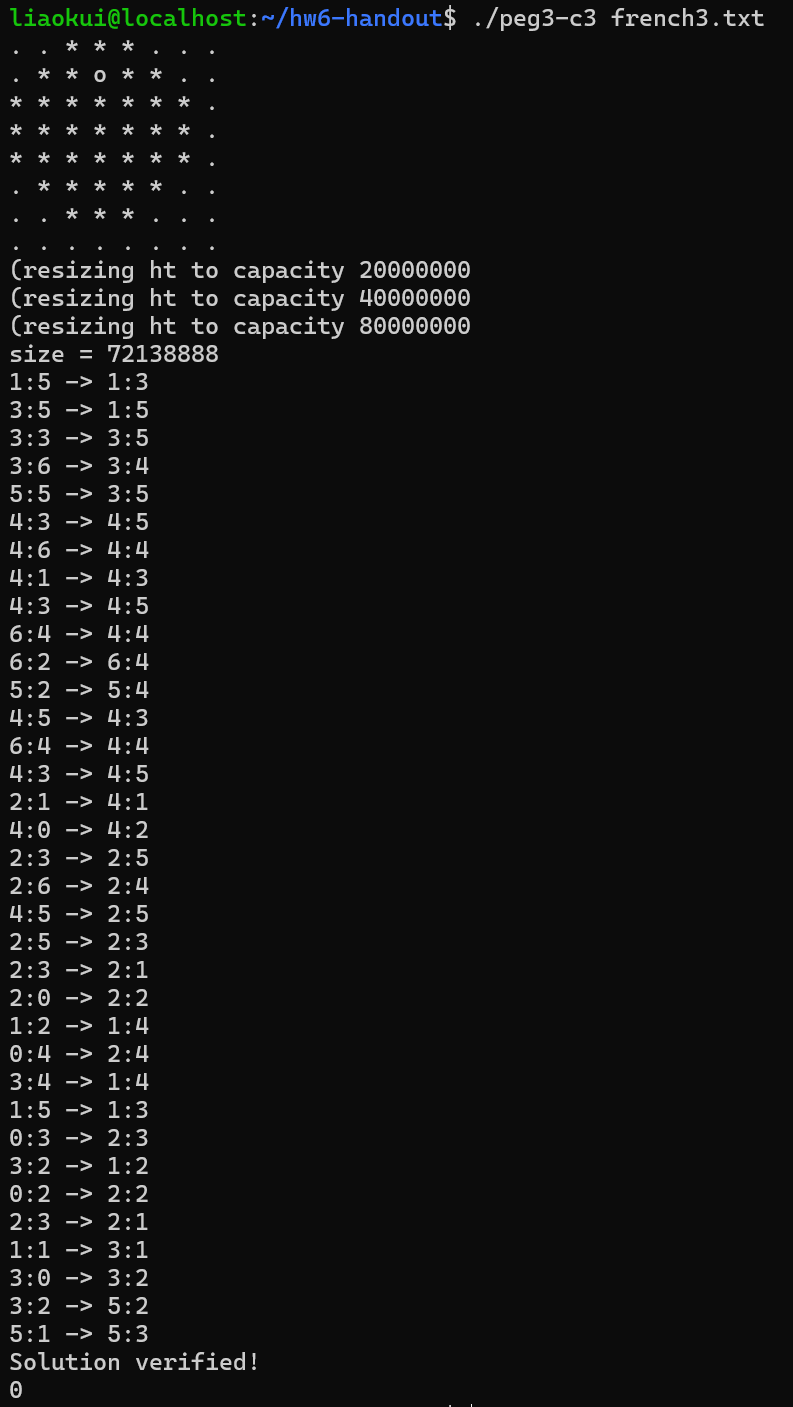
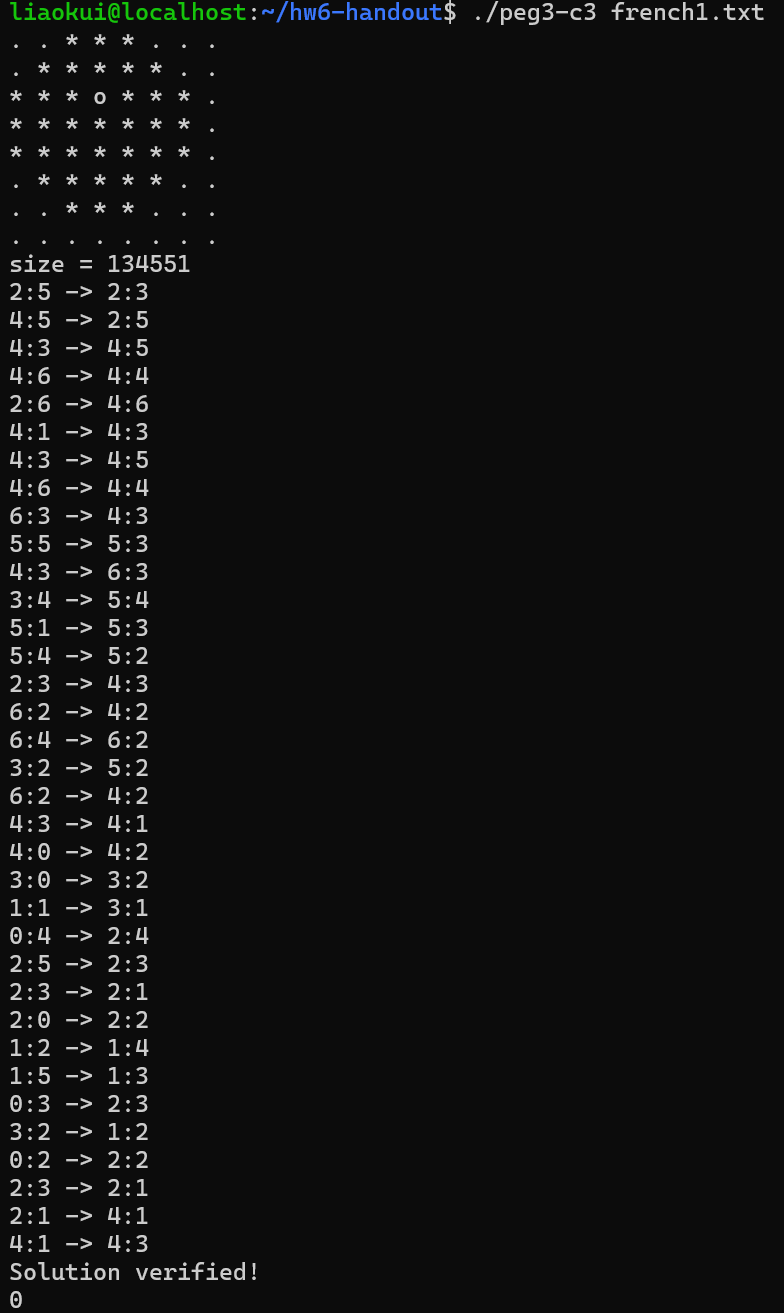
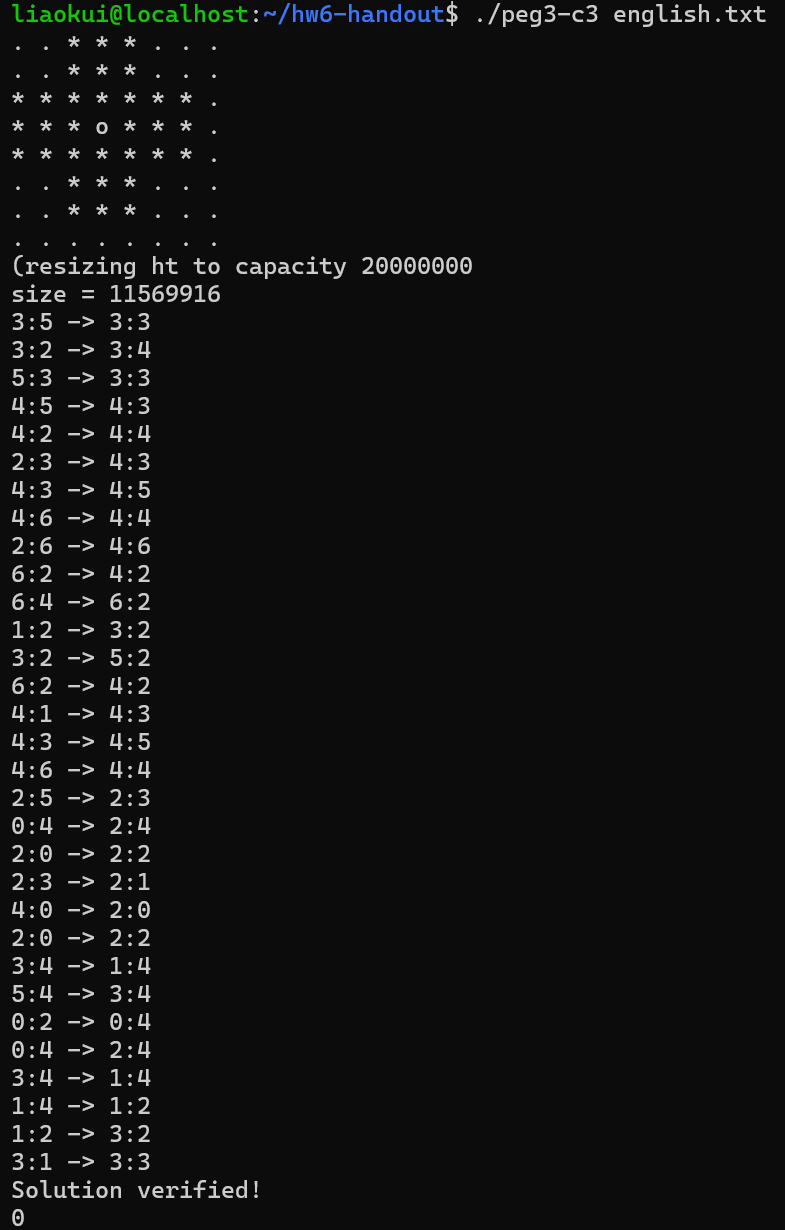


图4-13 peg3-c3求解测试图2

将结果汇总成表格，见表4-1。

表4-1 不同move顺序选择策略对比表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 求得解时哈希表大小 | | | |
| 程序 | english.txt | french1.txt | french2.txt | french3.txt |
| peg3 | **859** | 22730603 | **10991786** | 12022594 |
| peg3-c2 | 54590 | 29771866 | >160000000 | **3017996** |
| peg3-c3 | 11569916 | **134551** | >160000000 | 72138888 |

由上表可知，策略1：每次找到一个棋子，查看它是否能向各个方向跳跃适用于棋盘english.txt、french2.txt；策略2：每次找到一个棋子，查看它能否在各个方向被跳过适用于棋盘french3.txt；策略3：每次找到一个空格，查看它能否在各个方向被跳入适用于棋盘french1.txt；

1. 哈希测试

考虑到哈希函数不同会造成不同程度的冲突，使得哈希表链长不同，对效率也有重大影响，修改peg-client.c0中的哈希函数int hash(htkey k)，尝试不同的哈希策略。这里使用4种不同的哈希函数，用peg3.c0对english.txt求解，查看哈希表链长分布。

哈希函数1：简单相加 return k->i1 + k->i2;

哈希函数2：线性组合 return 31\*k->i1 + 17\*k->i2;

哈希函数3：异或 return (k->i1)^(k->i2);

哈希函数4：复合 int a=31\*k->i1 + 17\*k->i2;

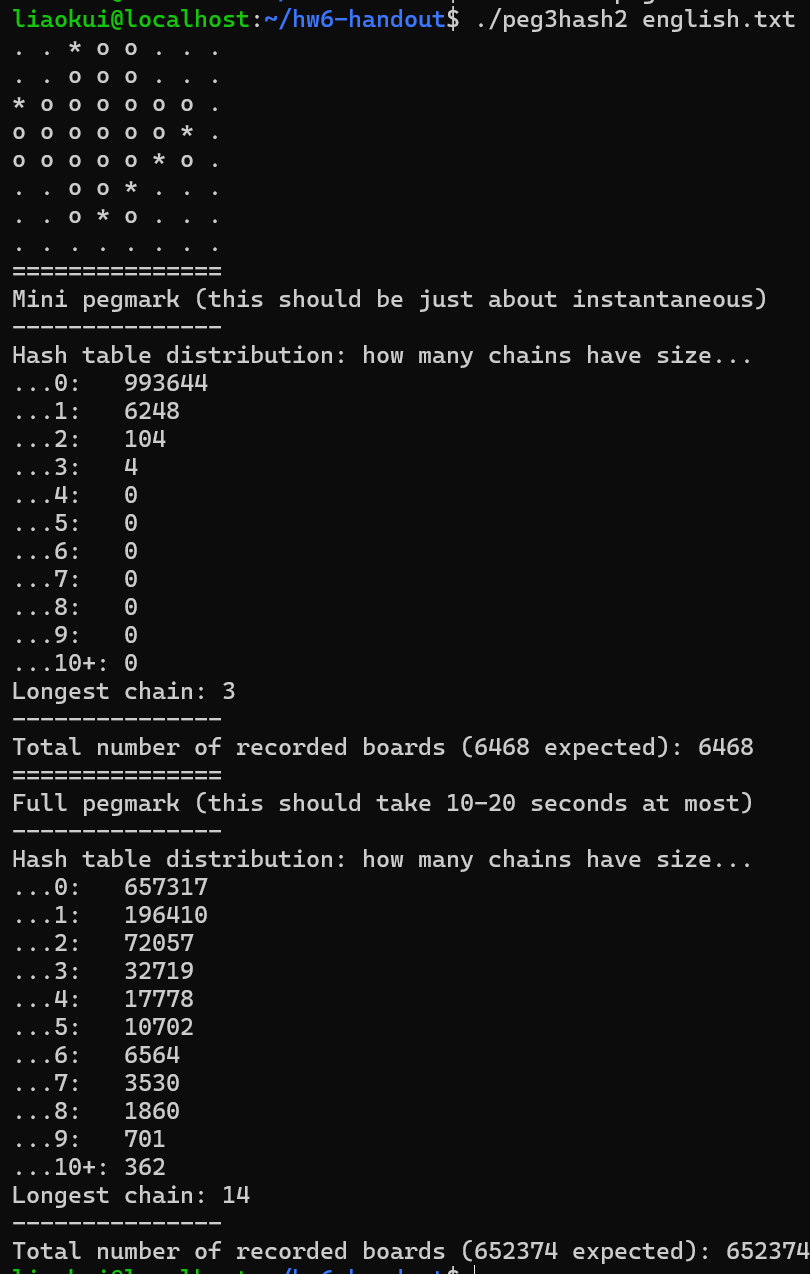
int b=(k->i1)^(k->i2);

return (a+b);

4种哈希函数的链长结果如图4-14所示。

cc0 -w -o peg3hash1 -r unsafe -c-O2 peg-client.c0 lib/\*.c0 peg3.c0 pegmark.c0

./peg3hash1 english.txt



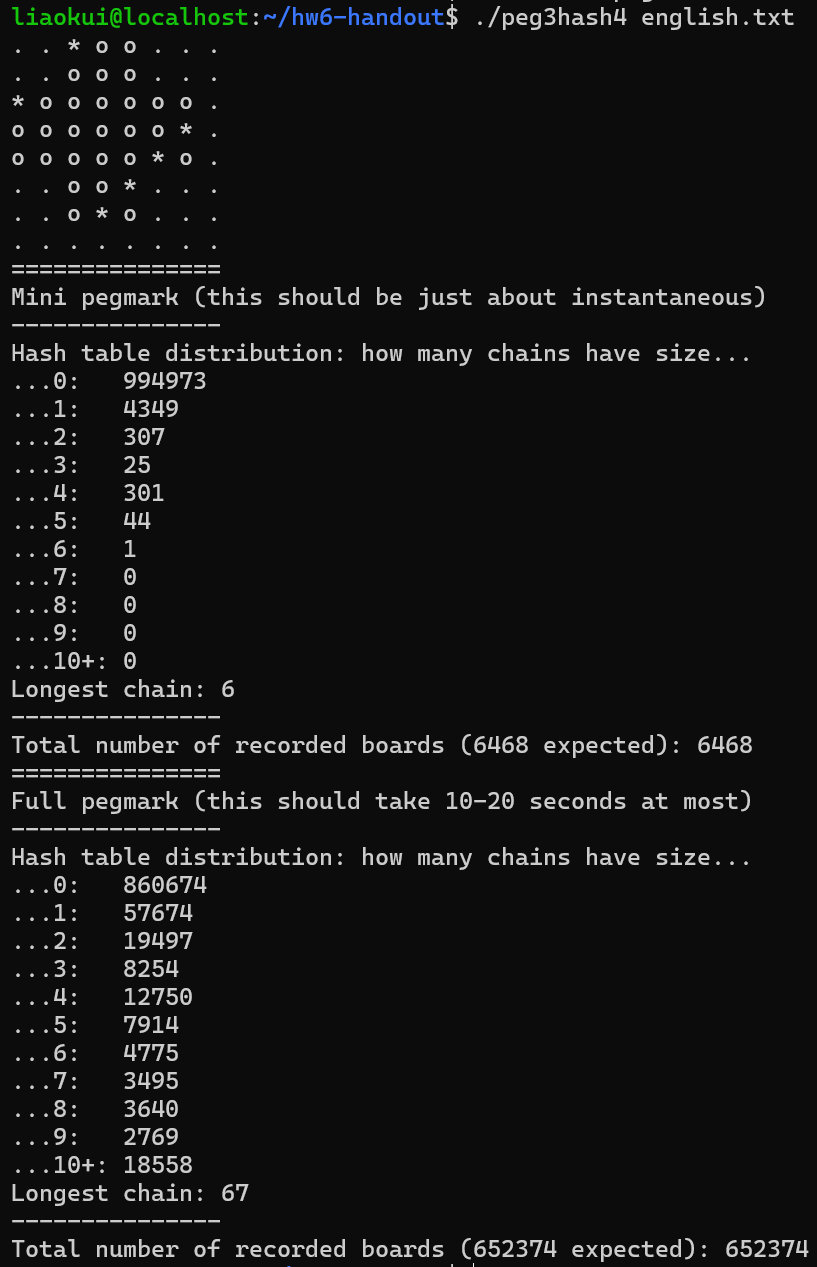
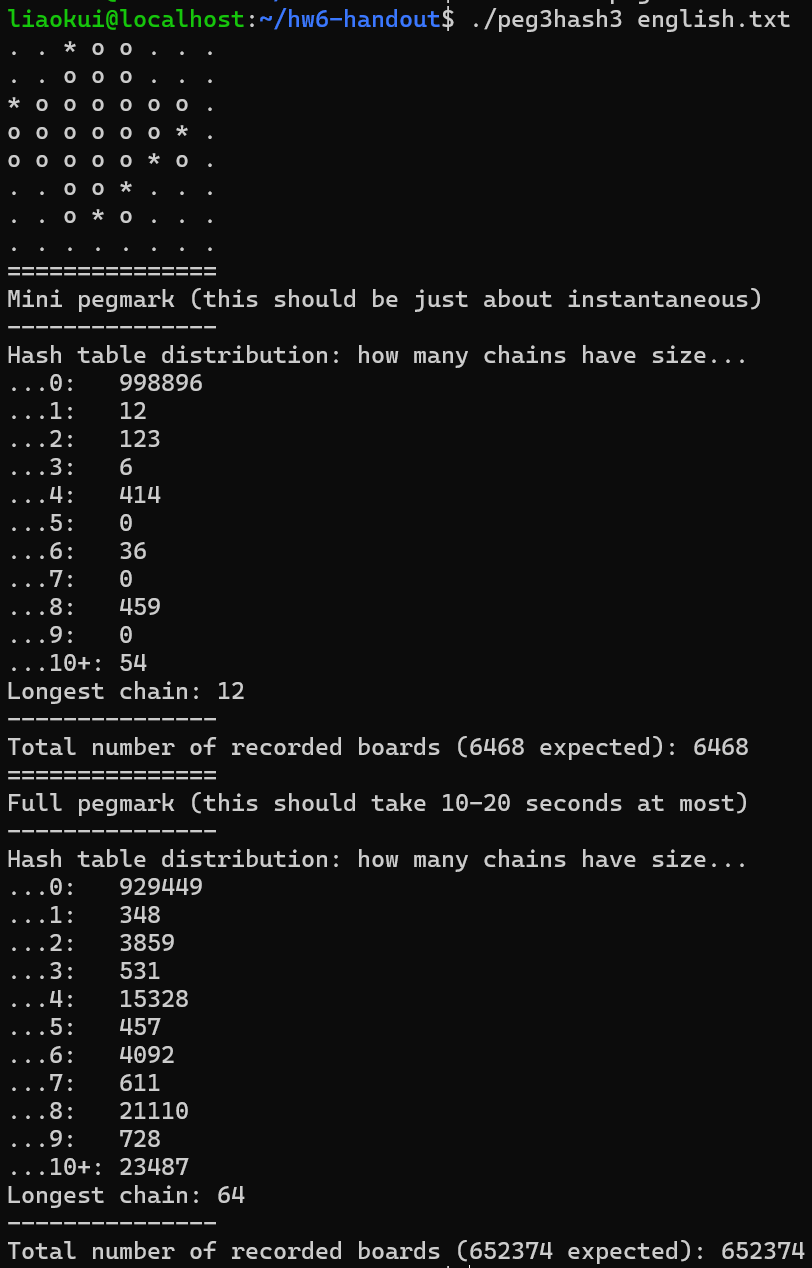


图4-14 4种哈希函数测试图

由上图可知采用哈希函数2最佳，最长链最短为14，短链所占比例更多。复合哈希函数情况却很差，分析得出是异或操作在本课程设计中表现地很差。

1. 优化

由于递归次数很多，即使看似较小的开销，在代码运行时也能导致运行时间大大增加，因此尽量优化代码十分重要。

避免反复申请内存和销毁数组空间造成较大开销。例如dx、dy数组，在peg\_solve函数中定义，调用solve函数时进行引用，而不用在solve函数中再创建，solve(B,S,num\_pegs(B),H,i1,i2,dx,dy)。

乘除法使用位运算实现，例如乘8改为左移3位，除2改为右移1位，从而在一个时钟周期就可以完成运算。

# 5　总结

主要工作如下：

（1）补充了peg-client.c0中待定的数据结构，完成了peg1.c0、peg2.c0、peg3.c0的基本功能要求，即peg1.c0能求解确定棋盘german.txt，peg2.c0能求解不确定棋盘english.txt，peg3.c0在前者基础上还能在不超时的情况下求解出法式孔明棋。

（2）探究了搜索棋盘时move的选择策略对不确定棋盘求解效率的影响，发现3种搜索策略在求解不同的英式棋盘、法式棋盘时各有所长。

（3）探究不同的哈希函数对哈希表链长的影响，得出结论：使用线性组合哈希函数最佳。

（4）不足的是解法完全是暴力的，对于可能的下一步move没有任何启发式的排序。正如作业中提示的，可以加入启发式的排序，选择先走最有希望的move。在这里我想到了一种策略，不过并未实现，即优先考虑那些具有较少可行移动的棋子，把所有合法的移动操作排序后存栈，取栈顶操作优先将具有较少可行移动的棋子移除，从而减少搜索空间。因为如果一个棋子有很多可行移动，那么它可能在后续步骤中有更多的选择，这会导致搜索树更加庞大。因此，优先处理这些棋子可能会导致更多的分支和更长的搜索时间。如果一个棋子只有很少的可行移动，这意味着它很快就会被移除或陷入无法移动的状态。通过优先处理这些棋子，可以更快地减少棋盘上的棋子数量，从而缩小搜索空间。此策略还有助于避免死局，因为如果优先处理了有很多可行移动的棋子，可能有较少可行移动的棋子的可行移动数量会被影响而减少，那就更有可能变成死棋。

# 6　体会

此次课程中认真看了课程要求，想要实现基本功能并不难，任务书中将需求说得很详细，把步骤进行了划分，逐渐递进，并给出可以用的数据结构示例，例如哈希表元素的数据结构，并适时地补充了可能出现的错误和解决办法，例如往哈希表里更新任何增加的数据，一定要保证传递的是一个ht\_insert的副本，而不是更新数据本身。还给出了很多优化的建议，像压缩棋盘、表示move、move的选择策略等等。

不过，想要写出一个高效的、能解决所有棋盘样例的求解程序并不容易，这需要添加启发式的排序，如果只是暴力地求解，只能对某些特定棋盘有较快的速度。

在实际编程中我遇到了一些挑战，比如我没有注意到数组的频繁创建和删除问题，导致运行时间很长而超时，还有往哈希表里更新数据，我没有把哈希表元素作为函数参数，使得更新的不是副本而出错。然后是c0语言的使用，c0语言去掉了一些复杂的特性，编译更简单，支持一些约定条件。但是功能相对有限，缺乏丰富的标准库支持，在实验中就对我造成了一些困扰，例如break语句我无法使用，数组创建格式不规范等。

总的来说，通过本次课程设计，我对算法数据结构的使用和规范性有了更深刻的理解，学会了根据实际问题选择合适的数据结构，进行多方面的优化，获益良多。

# 附录

**peg-client.c0**

/\* Client interface for stacks. \*/

/\* See lib/stacks.c0 \*/

/\* You can't say 'typedef move stackelem', because given the order in

\* which things are defined, we haven't defined what a move is yet \*/

typedef int stackelem;

/\* Client side implementation for hashtables. (Optional) \*/

/\* See lib/ht.c0 \*/

struct two\_ints {

int i1;

int i2;

int best\_num\_pegs;

};

typedef struct two\_ints\* htelem;

typedef struct two\_ints\* htkey;

// This is a terrible hashing implementation

int hash(htkey k)

//@requires k != NULL;

{//线性组合法

return 31\*k->i1 + 17\*k->i2;

}

// This is a reasonable key equality function

bool htkey\_equal(htkey k1, htkey k2)

//@requires k1 != NULL;

//@requires k2 != NULL;

{

return k1->i1 == k2->i1 && k1->i2 == k2->i2;

}

// This is a reasonable key extraction function (they key is the whole elem!)

htkey htelem\_key(htelem e)

//@requires e != NULL;

{

return e;

}

**peg1.c0**

#use <args>

#use <conio>

#use <util>

typedef int move;

/\*这里使用整数的四元组（对应棋子在操作前、

后的行号、列号）实现

压缩进单个int，每8位存储一个行列信息\*/

int row\_start(move m)

{

return m&0xff;

}

int col\_start(move m)

{

return (m>>8)&0xff;

}

int row\_end(move m)

{

return (m>>16)&0xff;

}

int col\_end(move m)

{

return (m>>24)&0xff;

}

/\*考虑当前棋局的所有合法操作，产生所有可能的

下一步操作，返回一个包括这些操作的新堆栈。\*/

stack find\_all\_move(board B)

{

stack new=stack\_new();

//实现寻找当前棋子上下左右偏移的数组

int[] dx=alloc\_array(int, 4);

int[] dy=alloc\_array(int, 4);

dx[0]=-2;dx[1]=2;dx[2]=0;dx[3]=0;

dy[0]=0;dy[1]=0;dy[2]=-2;dy[3]=2;

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

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

if(B[(row<<3)+col]==1)//寻找棋子的上下左右

{

for(int s=0;s<4;s++){

int row2=row+dx[s];

int col2=col+dy[s];

if(0<=row2&&row2<8&&0<=col2&&col2<8){

int i = (row<<3)+col;

int k = (row2<<3)+col2;

int j = (i+k)>>1;

if(B[j]==1&&B[k]==0){

move m=row|(col<<8)|(row2<<16)|(col2<<24);

push(new,m);

}

}

}

}

return new;

}

/\*递归的辅助函数solve，它将被peg\_solve调用。这个辅助

函数的参数应该包括棋盘、当前的操作堆栈、棋盘上剩下

的棋子个数。在有解时辅助函数solve返回1；

它也应该要在堆栈S中增加操作。\*/

int solve(board B,stack S,int number\_pegs)

//@requires is\_board(B);

//@ensures is\_board(B);

//@ensures \result >= 1;

{

stack s\_move=find\_all\_move(B);

//当前棋盘所有可能的移动操作

if (stack\_empty(s\_move)) {

//栈空说明当前棋盘无法移动棋子，直接返回剩余棋子数

return number\_pegs;

}

/\*选择堆栈中的第一个操作（忽略其他可能的下一步操作）

，递归地尝试它，看看下一个棋盘是否会产生解。如果下

一个棋盘产生了解，则说明当前棋盘也有解\*/

move m = pop(s\_move);

int row\_start = row\_start(m);

int col\_start = col\_start(m);

int row\_end = row\_end(m);

int col\_end = col\_end(m);

int i = (row\_start<<3)+col\_start;

int k = (row\_end<<3)+col\_end;

int j = (i+k)>>1;

B[i] = 0;

B[j] = 0;

B[k] = 1;

int ans = solve(B, S, number\_pegs - 1);

//递归的move都入栈后最外层才入栈，最外层正是第一个操作

push(S, m);

return ans;

}

int peg\_solve(board B, stack S)

//@requires is\_board(B);

//@requires num\_pegs(B) >= 1;

//@requires stack\_empty(S);

//@ensures is\_board(B);

//@ensures \result >= 1;

{

return solve(B,S,num\_pegs(B));

}

**peg2.c0**

#use <args>

#use <conio>

#use <util>

typedef int move;

/\*这里使用整数的四元组（对应棋子在操作前、

后的行号、列号）实现

压缩进单个int，每8位存储一个行列信息\*/

int row\_start(move m)

{

return m&0xff;

}

int col\_start(move m)

{

return (m>>8)&0xff;

}

int row\_end(move m)

{

return (m>>16)&0xff;

}

int col\_end(move m)

{

return (m>>24)&0xff;

}

/\*考虑当前棋局的所有合法操作，产生所有可能的

下一步操作，返回一个包括这些操作的新堆栈。\*/

stack find\_all\_move(board B)

{

stack new=stack\_new();

//实现寻找当前棋子上下左右偏移的数组

int[] dx=alloc\_array(int, 4);

int[] dy=alloc\_array(int, 4);

dx[0]=-2;dx[1]=2;dx[2]=0;dx[3]=0;

dy[0]=0;dy[1]=0;dy[2]=-2;dy[3]=2;

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

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

if(B[(row<<3)+col]==1)//寻找棋子的上下左右

{

for(int s=0;s<4;s++){

int row2=row+dx[s];

int col2=col+dy[s];

if(0<=row2&&row2<8&&0<=col2&&col2<8){

int i = (row<<3)+col;

int k = (row2<<3)+col2;

int j = (i+k)>>1;

if(B[j]==1&&B[k]==0){

move m=row|(col<<8)|(row2<<16)|(col2<<24);

push(new,m);

}

}

}

}

return new;

}

/\*递归的辅助函数solve，它将被peg\_solve调用。这个辅助

函数的参数应该包括棋盘、当前的操作堆栈、棋盘上剩下

的棋子个数。在有解时辅助函数solve返回1；

它也应该要在堆栈S中增加操作。\*/

int solve(board B,stack S,int number\_pegs)

//@requires is\_board(B);

//@ensures is\_board(B);

//@ensures \result >= 1;

{

stack s\_move=find\_all\_move(B);

//当前棋盘所有可能的移动操作

if (stack\_empty(s\_move)) {

//栈空说明当前棋盘无法移动棋子，直接返回剩余棋子数

return number\_pegs;

}

//在返回之前，棋盘必须返回它被调用时的状态

//每次比较剩余棋子数，记录下最小的即为最终答案

int min\_ans=number\_pegs;

/\*如果走到一个死胡同，没有任何下一步操作了，那么

代码应该撤销最后一步操作，然后尝试另一个操作。如

果那个操作也失败了，那就撤销那一步并继续这个过程。\*/

while(!stack\_empty(s\_move))

{

move m = pop(s\_move);

int row\_start = row\_start(m);

int col\_start = col\_start(m);

int row\_end = row\_end(m);

int col\_end = col\_end(m);

int i = (row\_start<<3)+col\_start;

int k = (row\_end<<3)+col\_end;

int j = (i+k)>>1;

B[i] = 0;

B[j] = 0;

B[k] = 1;

int ans = solve(B, S, number\_pegs - 1);

if(ans<min\_ans) min\_ans=ans;

if(ans==1){//只有解出棋盘的那些操作才能入栈

push(S,m);

return 1;

}

//回溯，前面没返回1说明这个操作无解，回到上一

//个状态，然后while又取出下一个可能得操作测试

B[i] = 1;

B[j] = 1;

B[k] = 0;

}

return min\_ans;//未返回1时则返回的是最少棋子数

}

int peg\_solve(board B, stack S)

//@requires is\_board(B);

//@requires num\_pegs(B) >= 1;

//@requires stack\_empty(S);

//@ensures is\_board(B);

//@ensures \result >= 1;

{

return solve(B,S,num\_pegs(B));

}

**peg3.c0**

#use <args>

#use <conio>

#use <util>

typedef int move;

/\*这里使用整数的四元组（对应棋子在操作前、

后的行号、列号）实现

压缩进单个int，每8位存储一个行列信息\*/

int row\_start(move m)

{

return m&0xff;

}

int col\_start(move m)

{

return (m>>8)&0xff;

}

int row\_end(move m)

{

return (m>>16)&0xff;

}

int col\_end(move m)

{

return (m>>24)&0xff;

}

/\*考虑当前棋局的所有合法操作，产生所有可能的

下一步操作，返回一个包括这些操作的新堆栈。\*/

stack find\_all\_move(board B,int[] dx,int[] dy)

{

stack new=stack\_new();

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

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

if(B[(row<<3)+col]==1)//寻找棋子的上下左右

for(int s=0;s<4;s++){

int row2=row+dx[s];

int col2=col+dy[s];

if(0<=row2&&row2<8&&0<=col2&&col2<8){

int i = (row<<3)+col;

int k = (row2<<3)+col2;

int j = (i+k)>>1;

if(B[j]==1&&B[k]==0){

move m=row|(col<<8)|(row2<<16)|(col2<<24);

push(new,m);

}

}

}

return new;

}

/\*递归的辅助函数solve，它将被peg\_solve调用。这个辅助

函数的参数应该包括棋盘、当前的操作堆栈、棋盘上剩下

的棋子个数。在有解时辅助函数solve返回1；

它也应该要在堆栈S中增加操作。\*/

int solve(board B,stack S,int number\_pegs,ht H,int i1,int i2,int[] dx,int[] dy)

//@requires is\_board(B);

//@ensures is\_board(B);

//@ensures \result >= 1;

{

/\*当我们尝试解一个棋盘式，我们先检查它是否

已经被记录为无解；如果确实是这样，我们马上

返回哈希表中存储的答案。\*/

struct two\_ints\* now\_elem=alloc(struct two\_ints);

now\_elem->i1=i1;

now\_elem->i2=i2;

now\_elem->best\_num\_pegs=number\_pegs;

struct two\_ints\* lookup=ht\_lookup(H,now\_elem);

if(lookup!=NULL){//已记录为无解

return lookup->best\_num\_pegs;

}

//有解，递归

stack s\_move=find\_all\_move(B,dx,dy);

//当前棋盘所有可能的移动操作

if (stack\_empty(s\_move)) {

//栈空说明当前棋盘无法移动棋子，直接返回剩余棋子数

if (number\_pegs == 1) {

printf("size = %d\n", ht\_size(H));

}

return number\_pegs;

}

//在返回之前，棋盘必须返回它被调用时的状态

//每次比较剩余棋子数，记录下最小的即为最终答案

/\*如果走到一个死胡同，没有任何下一步操作了，那么

代码应该撤销最后一步操作，然后尝试另一个操作。如

果那个操作也失败了，那就撤销那一步并继续这个过程\*/

while(!stack\_empty(s\_move))

{

move m = pop(s\_move);

int row\_start = row\_start(m);

int col\_start = col\_start(m);

int row\_end = row\_end(m);

int col\_end = col\_end(m);

//位运算提速

int i = (row\_start<<3)+col\_start;

int k = (row\_end<<3)+col\_end;

int j = (i+k)>>1;

B[i] = 0;

B[j] = 0;

B[k] = 1;

//根据索引更新i1、i2

int ii1=i1;

int ii2=i2;

if(i<32) ii1^=(1<<i);//对应位异或1

else ii2^=(1<<(i-32));

if(j<32) ii1^=(1<<j);

else ii2^=(1<<(j-32));

if(k<32) ii1^=(1<<k);

else ii2^=(1<<(k-32));

//更新i1、i2是副本，不是更新数据本身。

int ans = solve(B, S, number\_pegs - 1,H,ii1,ii2,dx,dy);

if(ans<now\_elem->best\_num\_pegs) now\_elem->best\_num\_pegs=ans;

if(ans==1){//只有解出棋盘的那些操作才能入栈

push(S,m);

return 1;

}

//回溯，前面没返回1说明这个操作无解，回到上一

//个状态，然后while又取出下一个可能得操作测试

B[i] = 1;

B[j] = 1;

B[k] = 0;

}

//未返回1时则无解，记录进哈希表

ht\_insert(H,now\_elem);

return now\_elem->best\_num\_pegs;//返回记录的最少棋子数

}

int peg\_solve(board B, stack S)

//@requires is\_board(B);

//@requires num\_pegs(B) >= 1;

//@requires stack\_empty(S);

//@ensures is\_board(B);

//@ensures \result >= 1;

{

ht H = ht\_new(10000000);

/\*把棋盘压缩成一个紧密的表达，同时包含足够的信息，

使得对于一个特定问题，两个棋盘只有在形态完全相同

时才对应同一个key。这里64个点-1 0 1我们把-1变成

0，然后按位存入2个int\*/

int i1 = 0;//哈希表元素副本，作为参数传递

int i2 = 0;

/\*棋盘上每个棋子为1，-1和0都看成0，因为-1永远不能放

棋子，对一个棋盘求解时永远不变，也就是两个状态中对

应的-1位置都一样，那用0替代没问题。不同的棋盘对应

棋子和空洞不同，也即1不同，而我们哈希表只需要判断

两个棋盘是否一样，一样的棋盘把-1看做0无疑仍一样，

不同的棋盘显然是1不一样，那么改-1毫无影响\*/

for (int i = 0; i < 32; i++)

i1 |= (max(B[i],0) << i);

for (int i = 32; i < 64; i++)

i2 |= (max(B[i],0) << (i-32));

//实现寻找当前棋子上下左右偏移的数组

//不放在find\_all\_move函数中是为了避免多次创建数组占用空间

int[] dx=alloc\_array(int, 4);

int[] dy=alloc\_array(int, 4);

dx[0]=-2;dx[1]=2;dx[2]=0;dx[3]=0;

dy[0]=0;dy[1]=0;dy[2]=-2;dy[3]=2;

return solve(B,S,num\_pegs(B),H,i1,i2,dx,dy);

}

**peg3-c2.c0仅有find\_all\_move函数以及dx、dy数组与peg3.c0不同，仅列出不同部分。**

stack find\_all\_move(board B,int[] dx,int[] dy)

{

stack new=stack\_new();

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

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

if(B[8\*row+col]==1)//寻找棋子的上下左右

for(int s=0;s<4;s++){

int row2=row;

int col2=col;

int row1=row+dx[s];

int col1=col+dy[s];

int row3=row-dx[s];

int col3=col-dy[s];

if(0<=row1&&row1<8&&0<=col1&&col1<8&&0<=row3

&&row3<8&&0<=col3&&col3<8){

int i = (row1<<3)+col1;

int k = (row3<<3)+col3;

if(B[i]==1&&B[k]==0){

move m=row1|(col1<<8)|(row3<<16)|(col3<<24);

push(new,m);

}

if(B[i]==0&&B[k]==1){

move m=row3|(col3<<8)|(row1<<16)|(col1<<24);

push(new,m);

}

}

}

return new;

}

int peg\_solve(board B, stack S)

//@requires is\_board(B);

//@requires num\_pegs(B) >= 1;

//@requires stack\_empty(S);

//@ensures is\_board(B);

//@ensures \result >= 1;

{

ht H = ht\_new(10000000);

int i1 = 0;

int i2 = 0;

for (int i = 0; i < 32; i++)

i1 |= (max(B[i],0) << i);

for (int i = 32; i < 64; i++)

i2 |= (max(B[i],0) << (i-32));

//实现寻找当前棋子上下左右偏移的数组

//不放在find\_all\_move函数中是为了避免多次创建数组占用空间

int[] dx=alloc\_array(int, 4);

int[] dy=alloc\_array(int, 4);

dx[0]=-1;dx[1]=1;dx[2]=0;dx[3]=0;

dy[0]=0;dy[1]=0;dy[2]=-1;dy[3]=1;

return solve(B,S,num\_pegs(B),H,i1,i2,dx,dy);

}

**peg3-c3.c0仅有find\_all\_move函数peg3.c0不同，仅列出不同部分。**

stack find\_all\_move(board B,int[] dx,int[] dy)

{

stack new=stack\_new();

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

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

if(B[8\*row+col]==0)//寻找空格的上下左右

for(int s=0;s<4;s++){

int row2=row+dx[s];

int col2=col+dy[s];

if(0<=row2&&row2<8&&0<=col2&&col2<8){

int j = 8\*row+col;

int i = 8\*row2+col2;

int k = (i+j)>>1;

if(B[i]==1&&B[k]==1){

move m=row2|(col2<<8)|(row<<16)|(col<<24);

push(new,m);

}

}

}

return new;

}