

**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： 大数据2201**

**学 号： U202215566**

**姓 名： 刘师言**

**指导教师： 李开**

**报告日期： 2024.5.15**

**计算机科学与技术学院**

**目 录**

**1　课程设计任务 1**

1.1 简介 2

1.2 设计内容 2

1.2.1 任务1 2

1.2.2 任务2 2

1.2.3 任务3 4

1.3 设计要求 5

1.3.1 任务1 5

1.3.2 任务2 6

1.3.3 任务3 6

**2　系统需求分析与总体设计 7**

2.1系统需求分析 7

2.2系统总体设计 7

**3　系统详细设计 8**

3.1有关数据结构的定义 8

3.2 主要算法设计 9

3.2.1 回溯法 9

3.2.2 记忆化搜索 11

**4　系统实现与测试 13**

4.1系统实现 13

4.1.1运行环境 13

4.1.2系统实现 13

4.2系统测试 17

**5　总结 23**

**6　体会 24**

**附录 25**

# 1　课程设计任务

## 1.1 简介

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、左或者右的另一个棋子（不可以对角跳），被跳过的棋子便从棋盘上移除。比如图1.1的棋盘，共有4种可能的操作，都是跳到正中央的位置。

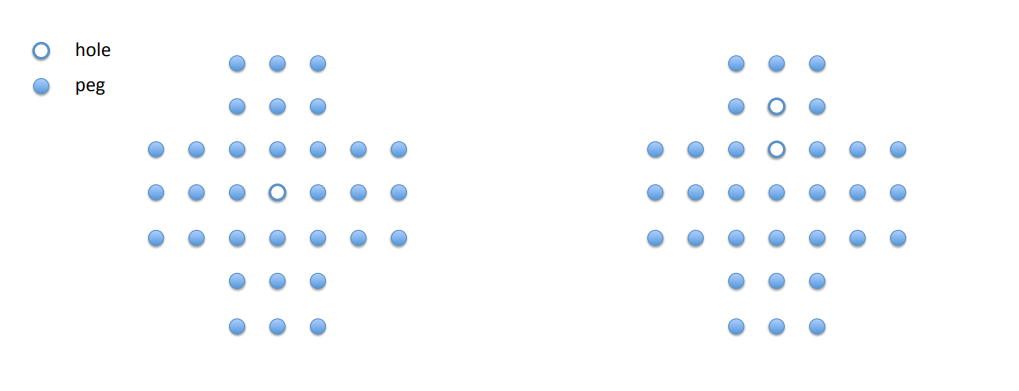


图1.1 孔明棋棋盘

选取上方的棋子跳到正中央，将会得到图1.2的棋盘。图1.2棋盘有3种可能的操作。

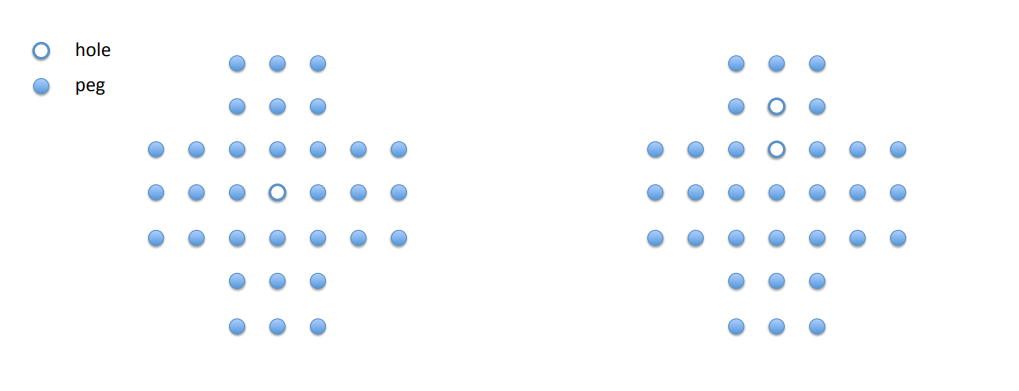


图1.2 进行一步合法操作

这个游戏的目标是只剩下一个棋子。在标准英式棋盘（如图1.1所示）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）。在该游戏的某些变种中，还指定了最后一个棋子的位置，但本课程设计不作要求，达到只剩一个棋子的状态就是唯一的目标。

孔明棋很适合递归的解法。若我们想要解32颗棋子的棋盘，那么可以列举出这个棋盘所有的可能操作；做完其中一个操作后，我们面对的是同一个问题：解一个棋盘——只不过棋子数目是31颗。

如果棋盘上只剩一颗棋子，那么我们就取得了游戏的胜利；但是如果棋盘上有多于一颗的棋子，并且没有合法的操作，那我们就失败了。每个操作系列都导向一个输局的棋盘是无解的。

因此，用计算机求解孔明棋游戏棋局是一种有效且高效的方法，本课程设计需要我们用计算机对棋局进行求解。

## 1.2 设计内容

#### 1.2.1任务1

任务1是后续任务的基础。本任务中，你的代码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。任务1的代码只会用于测试确定的棋局（可解或不可解）。

这个任务的目的是保证你正确地找出合法操作，正确地产生解，并且正确实现了指定的接口。

#### 1.2.2任务2

在先前的任务中，我们只解决了确定的孔明棋局：对于一个处在当前状态的棋盘，我们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进行求解（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。例如图1.3中第一步是1：2到1：0，第二步是2：1到2：3。

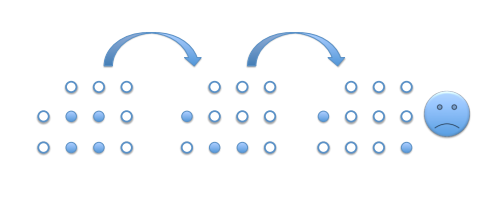


图1.3 任务1中一种失败的情形

利用回溯的策略，我们可以做出改进。现在我们知道上图种第三个棋盘（有两个棋子）是输局，即无解的。但我们不知道第二个棋盘（有三个棋子）是否无解，因为有另一个可行的操作——2：2到2：0。所以我们回溯到第二个棋盘再进行尝试，如图1.4所示。

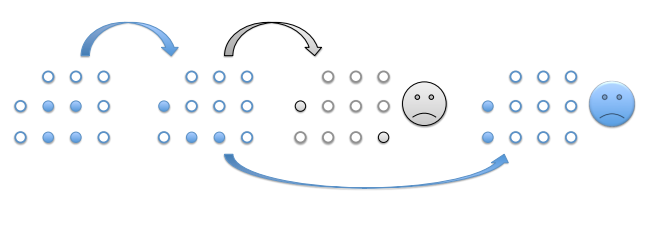


图1.4 回溯法棋盘图示

很不巧，第二种走法也导向一个输局。这意味着第二个棋盘也是无解的——所有走法都会导向输局。我们必须回溯到更早些，到有四个棋子的棋盘，然后选择不同的操作。如图1.5所示，我们在第一个棋盘选择走2：1到0：1，就会成功。

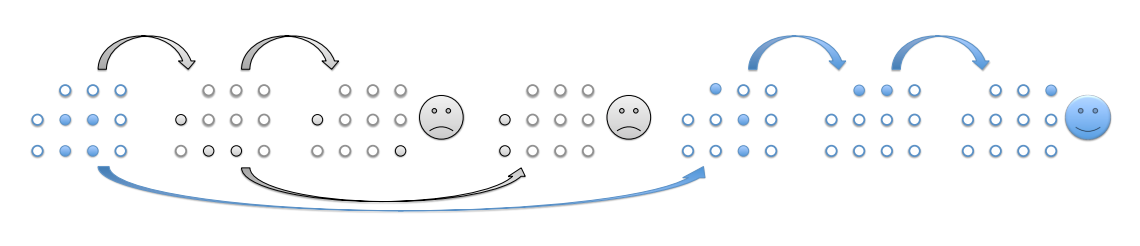


图1.5 回溯法棋盘图示

在任务2中，你需要对任务1的代码进行推广，使它能够解决需要回溯法来解的棋盘。

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。

总而言之，相比任务1，任务2中你的代码要做到肯定地回答某个特定孔明棋局是否有解。如果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。

#### 1.2.3任务3

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。在任务3中，用哈希表来将solve函数记忆化。

如果你把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，那么这个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时，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。

代码效率受三个关键因素的影响：key的选择，哈希函数的选择，以及尝试move的顺序。

在key的选择方面，我们推荐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。而key\_equal函数应该是很快的。注意你不可以不复制地用board数组本身，因为这样一来你的solve函数就会改变哈希表的内容。如果你的key是整个长度为64的整数数组，那么效率肯定太慢。拿字符串做key也是不好的，尽管在课程中我们有这么做过。

考虑效率时，记住你可以重新定义你的solve函数，改变它的参数到多少个都可以。比如，你可能发现在更新棋盘的同时更新key是有好处的，就可以把它们一起传过去。

## 1.3 设计要求

#### 1.3.1任务1

根据问题描述，决定你如何表示一次操作。在peg1.c0中定义move类型：

typedef \_\_\_\_\_\_ move;

并在peg1.c0中定义如下函数来从move中提取行、列信息。start表示棋子跳动的开始位置，end表示结束位置。

如之前所述，解是一个由操作组成的栈。在本作业中，stack是stackelem构成的。在peg-client.c0中定义stackelem：

typedef \_\_\_\_\_\_ stackelem;

使stackelem和move的类型相匹配，从而进一步实现由操作组成的栈。（不可以直接写typedef move stackelem，因为编译时move类型在后面才进行定义）

在peg1.c0中，定义如下函数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。

int peg\_solve(board B, stack S);

如果peg\_solve返回1，那么我们赢了。在这个情况下，S应该是解（操作组成的栈）。

如果peg\_solve返回大于1的整数，那么我们输了。棋盘上还剩下 \result 颗棋子。这个情况下，S可以是任意的。

注意peg\_solve如何利用返回值和堆栈S传递信息给调用者。由于堆栈S实际上是一个指针，peg\_solve和它调用的函数可以通过该指针共同操作内存中的同一片区域。

为了编写peg\_solve，你应该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。

这个辅助函数的一个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中，这个附加的辅助函数产生所有可能的下一步操作，也许返回一个包括这些操作的新堆栈。（尽管本任务只需要产生一个可能的下一步操作，但还是推荐现在就找出所有可能的下一步操作）。

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。为了正确地产生解，需要确保操作按照正确的顺序压入堆栈。最终的一组操作很容易以错误的顺序压入堆栈。

#### 1.3.2任务2

把你的代码从peg1.c0 拷贝到peg2.c0。

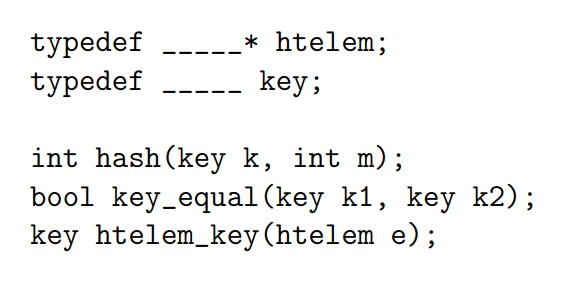
拓展peg\_solve方程，使其能够解决需要回溯法的棋盘。我们会用简单的需要回溯法的数据来测试你的代码。最复杂的一个是english.txt（标准英式棋盘，并且是初始状态）。

但是，如果你的peg\_solve方程返回比1大的数，表示整个棋局是完全无解的，那么返回的这个数字应该是你得到的所有棋盘中最少的棋子数量。而堆栈S仍然只需要在结果是1的时候包含一系列合法的操作。

#### 1.3.3任务3

把你的代码从peg2.c0 拷贝到peg3.c0。

在peg-client.c0中，提供哈希表所需的用户端类型和函数，如下：



已经为你提供了一个可行的例子。在peg3.c0中，拓展你的代码来利用哈希表减少搜索空间。

# 2　系统需求分析与总体设计

## 2.1系统需求分析

本次课设需要我们利用回溯法、记忆化搜索等方法，通过对指定接口的逐个实现，最终对一个给定的孔明棋棋局进行求解。如果该棋局有解，得到由一系列合法操作组成的解；如果该棋局无解，得到棋盘能剩下的最少棋子个数。

## 2.2系统总体设计

要对一个给定的孔明棋棋局进行求解，系统总体可设计以下几个模块：

1. 定义合法操作的数据结构及解的数据结构；
2. 定义记忆化搜索运用的棋盘压缩算法和哈希函数；
3. 利用回溯法、记忆化搜索等方法对棋局进行求解，并对每步的合法操作进行记录；
4. 根据求解结果返回由一系列合法操作组成的解或棋盘能剩下的最少棋子个数。

系统总体设计图如图2.1所示：

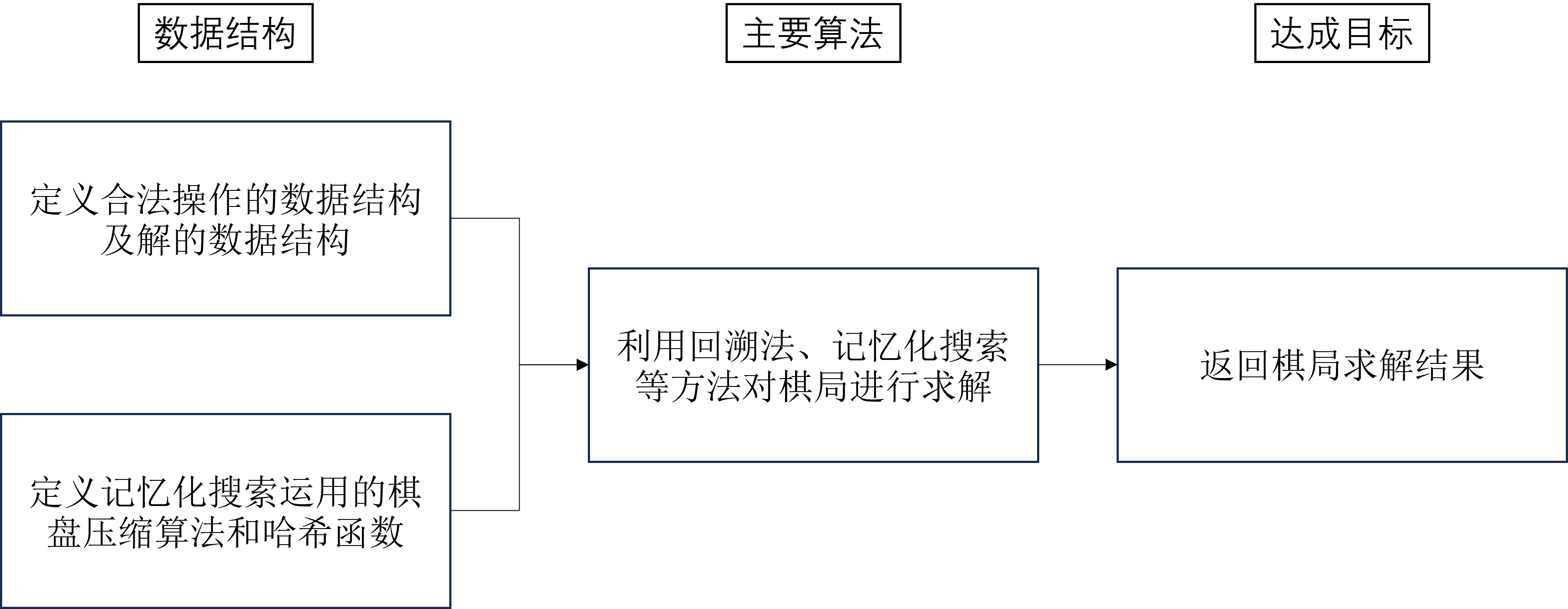


图2.1 系统总体设计图

# 3　系统详细设计

## 3.1有关数据结构的定义

首先要定义合法操作的数据结构及解的数据结构，由于解是由一系列的合法操作组成，因此二者的数据结构应一致。而当定义解的数据结构时，合法操作的数据结构还未定义，因此不能直接用合法操作作为解的数据结构类型，此时引入一个新的棋子坐标结构体：

// 棋子坐标结构体

struct Pos {

// 棋子跳动的开始位置

int start;

// 被跳过的棋子位置

int mid;

// 棋子跳动的结束位置

int end;

};

该结构体由三个不同的棋子位置构成，分别是棋子跳动的开始位置，被跳过的棋子位置和棋子跳动的结束位置。每个位置的值是由棋子所在行数乘以棋盘长度加上棋子所在列数计算而来。合法操作的数据结构及解的数据结构均是指向该结构体的指针，如下所示：

// 合法操作的数据结构

typedef struct Pos\* move;

// 解的数据结构

typedef struct Pos\* stackelem;

在利用记忆化搜索对棋局进行求解时，用到了哈希表来存储已经搜索过且无解的棋盘状态，该棋盘状态结构体如下：

// 棋盘状态结构体

struct two\_ints {

// 前4行棋盘状态

int i1;

// 后4行棋盘状态

int i2;

// 该无解棋盘能剩下的最少棋子个数

int best\_num\_pegs;

};

通过棋盘状态压缩，棋盘上下两部分分别被压缩成了两个独一无二的32位整数，并储存在棋盘状态结构体中，因此不同棋盘状态结构体能表达出不同的棋盘状态，确保了从棋盘映射到该结构体的唯一性，为哈希表的使用提供了必要条件。将复杂的二维棋盘压缩成简单的整型数据，也大大提高了搜索时算法执行的效率，有效地缩短了求解所用的时间。

哈希表的键和值则均是指向该棋盘状态结构体的指针，如下所示：

// 键的数据结构

typedef struct two\_ints\* htkey;

// 值的数据结构

typedef struct two\_ints\* htelem;

对程序用到的两个主要结构体Pos和two\_ints以表格的形式进行呈现，如表3.1所示：

表3.1 程序包含的主要结构体

|  |  |  |
| --- | --- | --- |
| 数据名 | 用途 | 数据项 |
| Pos | 记录一次合法操作中影响的棋子坐标，即棋子跳动的开始位置、被跳过的棋子位置和棋子跳动的结束位置 | int：start，end，mid； |
| two\_ints | 记录棋盘状态压缩后得到的两个独特整数，及该状态下棋盘剩下的最少棋子个数 | int：i1，i2，best\_num\_pegs; |

## 3.2 主要算法设计

#### 3.2.1 回溯法

回溯法在求解一个给定的孔明棋棋局时的基本流程是：

1. 遍历棋盘，得到当前棋局下所有合法操作的堆栈；
2. 依次从堆栈中取出一个合法操作，并对棋局进行该合法操作，继续递归执行（1）中的操作；
3. 当某个棋局下无任何合法操作时，若此时棋盘上剩下的棋子个数为1，那么我们就成功地得到了棋局的解，按顺序将合法操作压入解的栈中，并将该堆栈返回；若此时棋盘上剩下的棋子个数大于1，说明此棋局无解，撤销（2）中对棋局作出的合法操作，即回溯到执行（2）前的棋局状态，并继续执行栈中的下一个合法操作。
4. 当得到一个棋局的解或遍历完初始棋盘的所有合法操作仍没有得到一个棋局的解时，游戏结束，并根据求解结果返回解的堆栈或棋盘上能剩下的最少棋子个数。

将回溯法的基本流程用流程图的形式呈现，如图3.1所示：

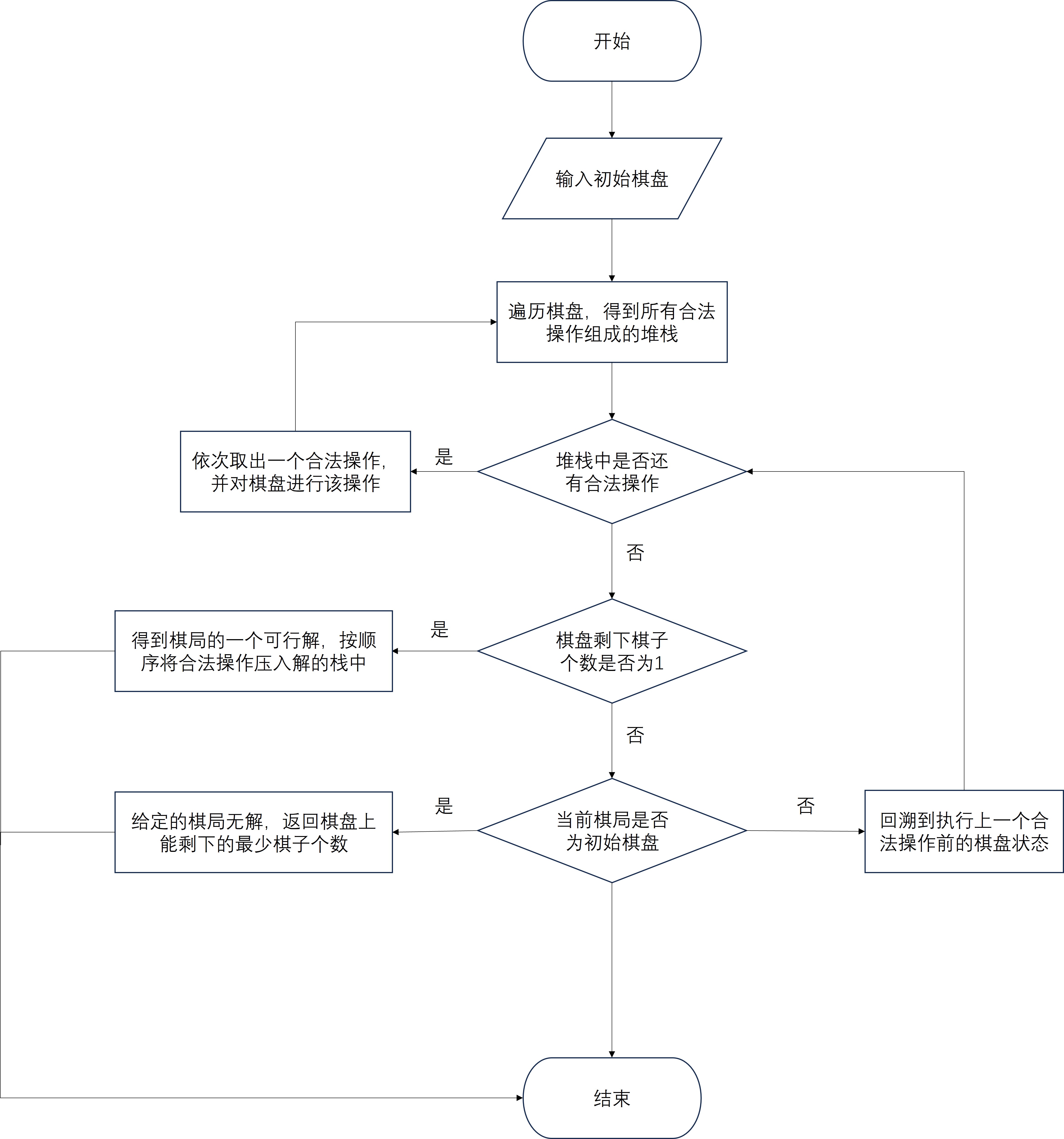


图3.1 回溯法基本流程图

#### 3.2.2 记忆化搜索

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。因此记忆化搜索采用哈希表来将solve函数记忆化，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。当尝试一种棋局时，先检查它是否已经被记录为无解，即是否已被存在哈希表中，如果确实存在，则直接对棋局进行回溯。

存入哈希表之前，需要利用棋盘状态压缩算法对当前二维棋盘数组进行压缩。具体方法是将棋盘分成上下两个部分，每个部分为4×8大小的二维矩阵，采用二进制从左至右、从上至下对棋盘进行编码，有棋子的格子记为1，其他格子记为0，这样便得到了一串由0和1组成的二进制串，且位数为32位。再对该二进制串进行十进制转换，最终得到一个32位（4个字节）整数，用int类型存储能确保不会溢出，如图3.2所示。

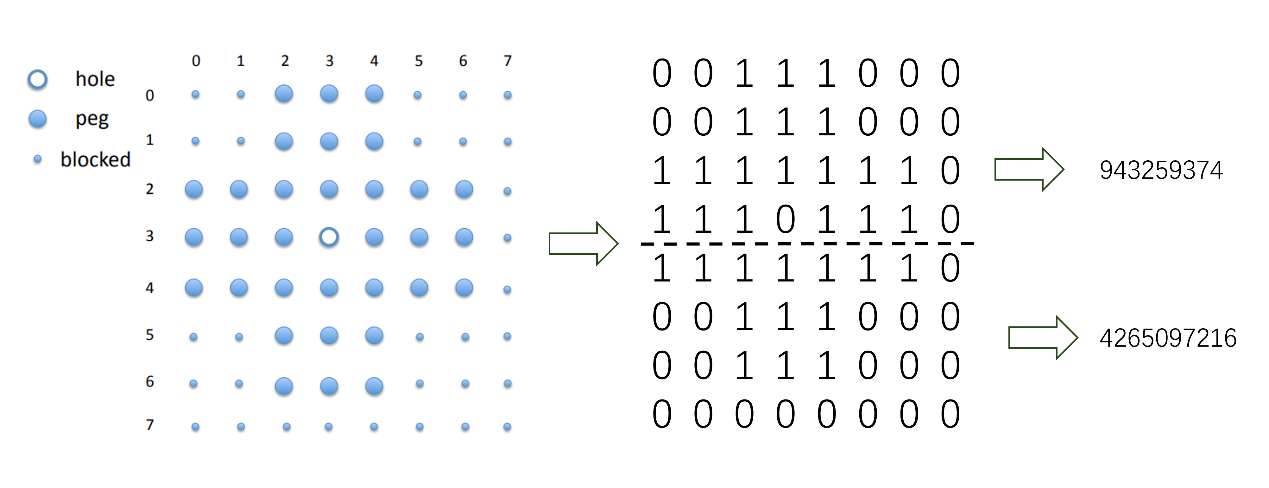


图3.2 棋盘压缩算法图示

这样一个完整的棋盘就被编码成了两个int型整数，且这两个整数的组合是独一无二的，确保了该算法的唯一性。

为了提高哈希表的运行效率，哈希表要尽可能地减少哈希碰撞的产生，因此在确保正确映射的前提下需要增大哈希函数的随机性。这里我采用了C0生成随机数和求余的方法，以棋盘状态压缩后的两个整数作为种子生成随机数，并将生成的随机数分别乘以这两个整数得到的结果作差（不相加是为避免溢出），再加上第一个整数模第二个整数加一（避免第二个整数为0时产生错误）的余数作为哈希函数的返回值，如下所示：

// 哈希函数

int hash(htkey k)

//@requires k != NULL;

{

return k->i1 \* rand(init\_rand(k->i1)) - k->i2 \* rand(init\_rand(k->i1)) + k->i1 % (k->i2 + 1);

}

记忆化搜索可以在回溯法的过程中对递归产生的结点树进行剪枝处理，减少了众多不必要的重复搜索，大大提高了算法搜索的效率，有效地缩短了算法的运行时间。

将记忆化搜索整合到上一步的回溯法中，得到的程序算法完整流程如图3.3所示：

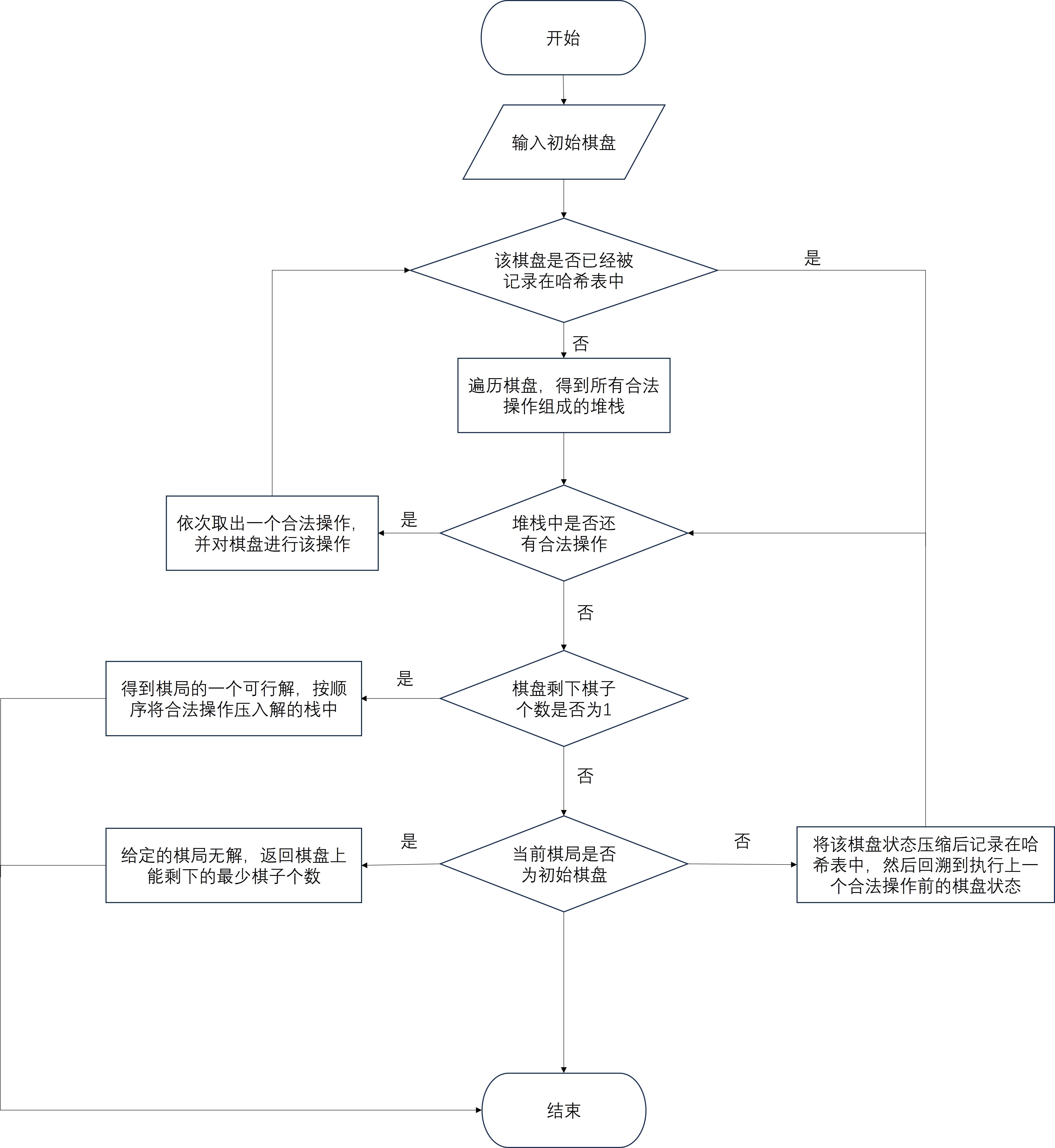


图3.3 回溯法结合记忆化搜索的完整流程图

# 4　系统实现与测试

## 4.1系统实现

#### 4.1.1 运行环境

a) 主机：

系统：Windows11 12th Gen Intel(R) Core(TM) i7-12700H 2.30 GHz

内存：16G

b) 虚拟机：

软件：Orcale VM VirtualBox

系统：Ubuntu-22.04.3

内存：4G

#### 4.1.2 系统实现

客户端和孔明棋求解主要算法的函数实现分别如下所示，每个函数都有丰富的前置条件与后置条件等约束，符合C0代码开发规范。

1. peg-client.c0客户端：

（1）

/\*

哈希函数

参数：htkey k 要哈希的键

返回值：哈希结果

/\*

int hash(htkey k)

//@requires k != NULL;

{

return k->i1 \* rand(init\_rand(k->i1)) - k->i2 \* rand(init\_rand(k->i1)) + k->i1 % (k->i2 + 1);

}

2. peg3.c0（由于peg3.c0是peg1.c0和peg2.c0的结合改良版本，因此省去对前两版本中函数的介绍）：

（1）

/\*

生成表示上下左右四个方向的二维数组

参数：int[][] d 要存放四个方向的二维数组

\*/

void alloc\_directions(int[][] d)

（2）

/\*

通过合法操作结构体获取棋子跳动开始位置的行数

参数：move m 一次合法操作

返回值：棋子跳动开始位置的行数

\*/

int row\_start(move m)

（3）

/\*

通过合法操作结构体获取棋子跳动开始位置的列数

参数：move m 一次合法操作

返回值：棋子跳动开始位置的列数

\*/

int col\_start(move m)

（4）

/\*

通过合法操作结构体获取棋子跳动结束位置的行数

返回值：棋子跳动结束位置的行数

参数：move m 一次合法操作

\*/

int row\_end(move m)

（5）

/\*

通过合法操作结构体获取棋子跳动结束位置的列数

返回值：棋子跳动结束位置的列数

参数：move m 一次合法操作

\*/

int col\_end(move m)

（6）

/\*

得到2的k次方

参数：int k 2的幂

返回值：2的k次方

\*/

int pow(int k)

//@requires k>=0;

（7）

/\*

将棋盘状态压缩得到棋盘状态结构体

参数：board B 当前棋盘

返回值：htkey 棋盘状态压缩后的结构体

\*/

htkey board\_to\_key(board B)

//@requires is\_board(B);

（8）

/\*

获取当前棋局所有合法操作的堆栈

参数：board B 当前棋盘

返回值：当前棋局所有合法操作的堆栈

\*/

stack all\_solutions(board B)

//@requires is\_board(B);

//@ensures is\_board(B);

（9）

/\*

获取棋局的一个解，并将解的合法操作堆栈保存在参数S中

参数：board B 当前棋盘

stack S 解的合法操作堆栈

ht H 哈希表

htkey k 当前棋盘的哈希值

num\_pegs 当前棋盘上剩余的棋子个数

返回值：若初始棋局有解返回1，无解返回最少剩余棋子个数

\*/

int solve(board B, stack S, ht H, htkey k, int num\_pegs)

//@requires is\_board(B);

//@requires num\_pegs >= 1;

//@ensures is\_board(B);

//@ensures \result >= 1;

（10）

/\*

棋局求解接口，由客户端调用该函数，该函数内再调用solve函数对棋局进行求解

参数：board B 初始棋盘

stack S 解的合法操作堆栈

返回值：若初始棋局有解返回1，无解返回最少剩余棋子个数

\*/

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;

## 4.2系统测试

按照任务1、2、3的顺序依次用C0编译器对peg1.c0，peg2.c0，peg3.c0进行编译，得到相应的可执行文件。

peg1.c0仅用于测试确定的棋局（可解或不可解），保证能正确地找出合法操作，正确地产生解。给定最基础的german.txt棋盘，运行peg1的结果如图4.1所示：

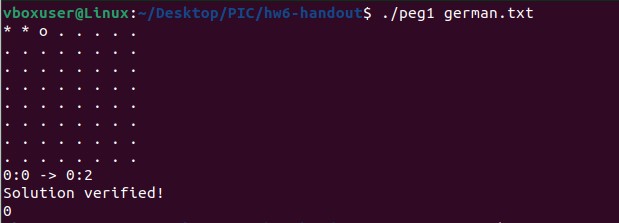


图4.1 peg1在german.txt棋盘上的运行结果

可以看出终端中输出了“Solution verified”的语句，说明求解成功，peg1.c0编写正确。

peg2.c0在peg1.c0的基础上加入了回溯法，并能在初始棋盘有解的情况下返回解的合法操作堆栈，在初始棋盘无解的情况下返回棋盘上能剩下的最少棋子个数。给定稍有难度的english.txt完整孔明棋棋盘，并在命令前加入time指令以记录程序的运行时间，运行peg2的结果如图4.2所示：

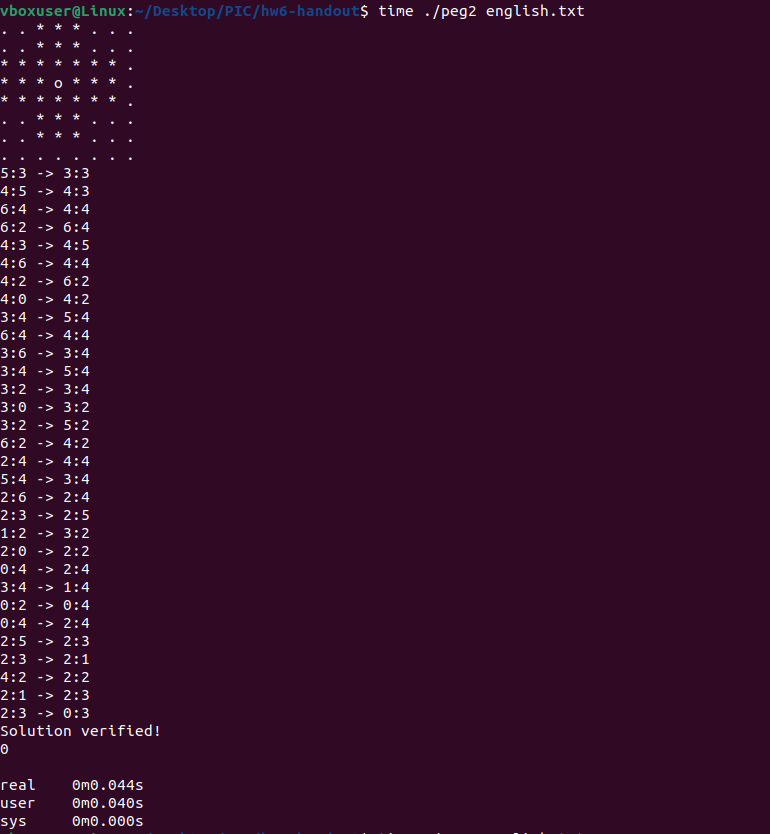


图4.2 peg2在english.txt棋盘上的运行结果

可以看出终端中输出了“Solution verified”语句，且依据解的合法操作堆栈依次打印出了求解过程每一步的操作，说明求解成功，peg2.c0编写正确。程序的运行时间为0.044s，可见回溯法能在较短时间内对孔明棋完成求解，进一步证明了计算机在求解实际应用问题中的高效性与可靠性。

然而当给定难度更大的french1.txt棋盘时，peg2经过非常长的时间仍无法对其完成求解，说明仅用回溯法进行搜索的效率不高，仍有较大提升空间，因此接下来在peg3.c0的程序中引入了记忆化搜索。

peg3.c0在peg2.c0的基础上加入了记忆化搜索，即采用哈希表对无解棋盘的棋盘状态进行存储。为确保哈希表运行的效率，先对哈希函数的性能进行测试。使用命令cc0 peg-client.c0 lib/\*.c0 peg3.c0 pegmark.c0对peg3.c0进行编译，使pegmark程序调用我们在peg3.c0中编写的接口。使用命令time ./a.out执行编译生成的可执行文件，同时记录程序运行的时间，结果如图4.3所示：

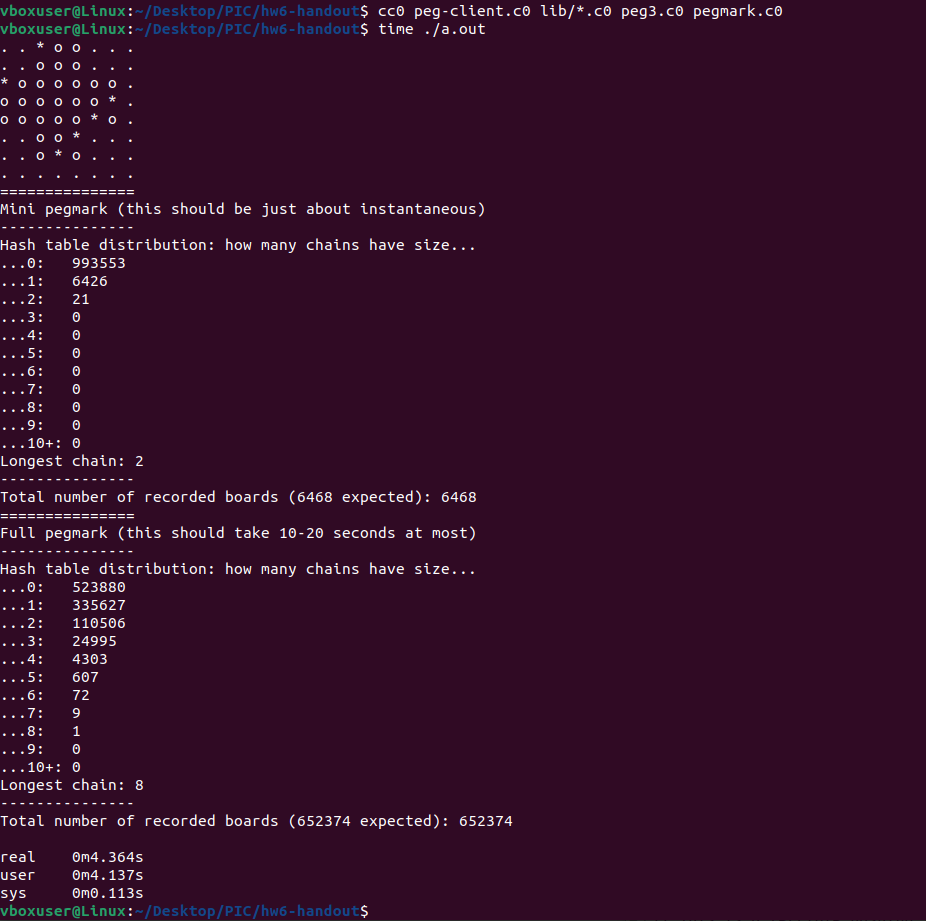


图4.3 pegmark性能测试结果

从测试结果可以看出，哈希表中绝大多数非空链的长度为1或2，且所有链中最长链的长度为8，同时测试程序的总运行时间在4秒左右，速度较快，说明哈希表的性能极佳，哈希函数的编写十分成功。

接下来同样给定难度更大的french1.txt棋盘，并在命令前加入time指令以记录程序的运行时间，运行peg3的结果如图4.4所示：

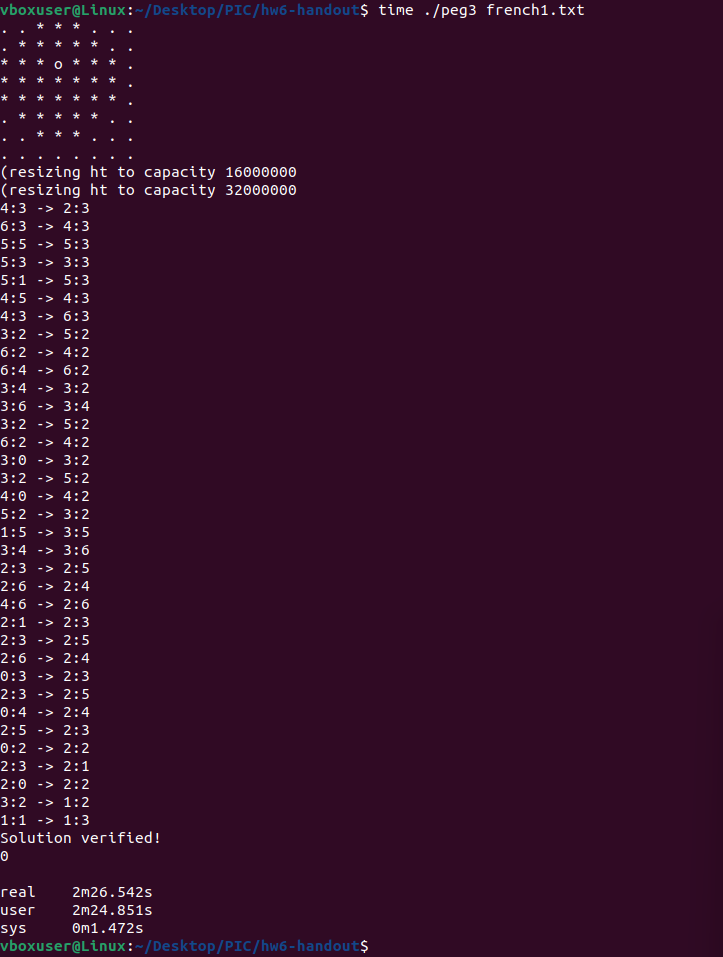


图4.4 peg3在french1.txt棋盘上的运行结果

从运行结果可以看出，对于难度更大的french1.txt棋盘，利用回溯法并结合了记忆化搜索的peg3程序仍可以在较短时间内对其完成求解，并将正确的解返回，再次说明了记忆化搜索对程序运行效率的提升之大，有效地缩短了程序的运行时间。

另外，为确保程序的稳定性，给定难度与french1.txt相当的french2.txt和french3.txt棋盘分别对peg3程序进行测试，运行结果如图4.5，图4.6所示：

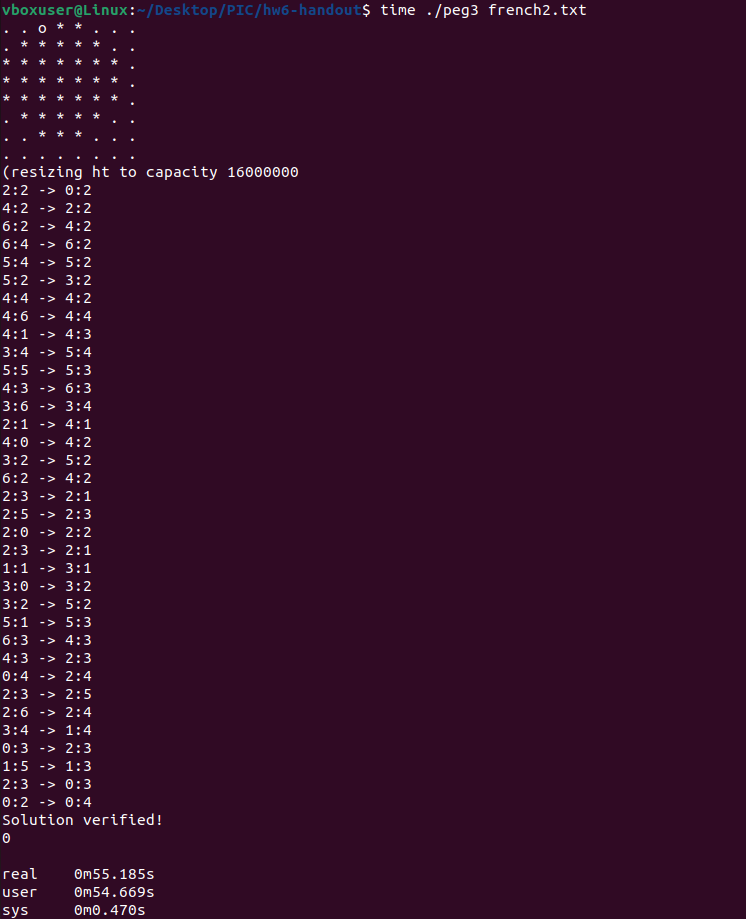


图4.5 peg3在french2.txt棋盘上的运行结果

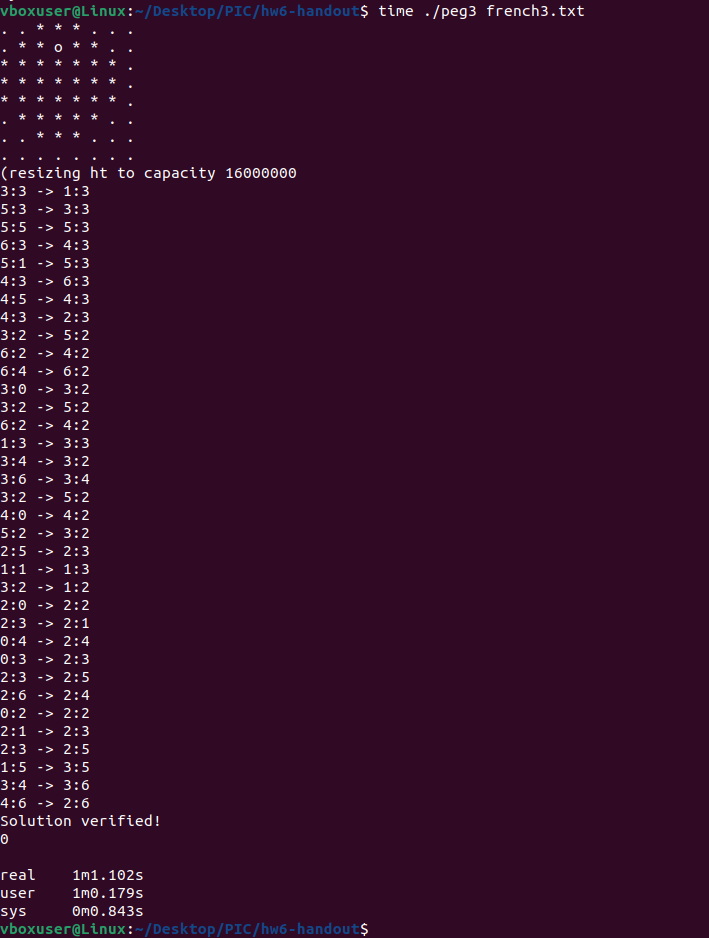


图4.6 peg3在french3.txt棋盘上的运行结果

可以看到，peg3在这两张难度较大的棋盘上依然能以较短时间完成求解，说明程序效率较高，同时稳定性良好。至此，所有的测试环节已经完成，通过一步步的迭代改进与层层优化，最终编写的peg3.c0程序具有较高的求解效率，同时具有良好的稳定性，能对复杂棋盘进行高效求解。

# 5　总结

本次课设主要工作如下：

（1）对于确定的孔明棋棋盘，正确地找出合法操作，正确地产生解；

（2）利用回溯法对完整孔明棋棋盘进行求解，并得到由一系列合法操作组成的解，能对大部分简单的棋盘进行求解；

（3）结合记忆化搜索进一步提高了求解的效率，大大缩短了程序运行的时间，能对较复杂的棋盘进行高效求解。

（4）一步步对简单的求解程序进行迭代改进与层层优化，使程序的运行效率得到不断提升，最终圆满达到预期目标。

# 6　体会

本次课设的主要任务是采用回溯法并结合记忆化搜索对孔明棋棋局进行求解。由于在前几个学期已经接触过不少算法方面的知识，因此在设计算法和编写代码的过程中，通过查阅相关资料和询问老师同学，能较为顺利地完成整个课设任务的实现。

然而，在从peg1.c0编写简单求解程序开始，到peg2.c0利用回溯法和peg3.c0结合记忆化搜索的一步步迭代改进的过程中，我仍经历了许多大大小小的挑战，同时也给我留下了十分深刻的印象。例如，C0语言虽然是C语言的子集，但由于其检查严格的特性和一些独有函数的定义，在初步接触时要学习掌握的知识仍不少，且一不小心就容易出现编译错误，这在之后一点点的学习积累中有了较好的改善，现在已能流畅地运用其内置的部分函数，并能顺利地编写出一个个风格优美的C0程序。

此外，《命令式计算原理》这门课让我重温了许许多多数据结构的知识，并在一些方面有了全新的认知。在编写记忆化搜索模块的过程中，之前对哈希函数的理解仅浅浅停留在原理，并未深入到动手实现一个性能较好的哈希函数。而此次课设我经过一次次地试验与查阅资料，一次次地编译与性能测试，终于从一个糟糕的哈希函数改进得到一个性能极佳的哈希函数，使哈希表上链长的集中分布由10左右减少到了1和2，最长链的长度也由原来的几十甚至上百一步步缩减至现在的个位数，这不仅给我带来了极大的成就感，也使得我对哈希表的实质有了更为深刻的理解，受益匪浅。

最后，本次课设不仅是对命令式一门课的总结与考查，更是对以往所有相关知识的一次重温与再认识。从C语言到C0，从数据结构到算法，课设涵盖知识点的范围十分全面。本次课设是对自己这门课程的掌握和运用的一次绝佳检验，希望在今后的学习工作中能继续秉持钻研精神，学以致用，一步步提高自己的认知水准与代码能力。路漫漫其修远兮，吾将上下而求索。

# 附录

peg-client.c0：

/\*

\* Peg solitaire

\* Client-side implementations

\*

\* 15-122 Principles of Imperative Computation \*/

/\* 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 \*/

#use <rand>

#use <args>

#use <conio>

struct Pos {

// 棋子跳动的开始位置

int start;

// 被跳过的棋子位置

int mid;

// 棋子跳动的结束位置

int end;

};

typedef struct Pos\* stackelem;

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

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

/\* It is possible to correctly do peg3.c0 with both elements and keys

\* defined to be a "struct two\_ints\*". It's maybe not too obvious that

\* this is the case. It might be easier to think of how boards could

\* be represented as a struct containing \*four\* integers first. \*/

struct two\_ints {

int i1;

int i2;

int best\_num\_pegs;

};

typedef struct two\_ints\* htelem;

typedef struct two\_ints\* htkey;

int hash(htkey k)

//@requires k != NULL;

{

return k->i1 \* rand(init\_rand(k->i1)) - k->i2 \* rand(init\_rand(k->i1)) + k->i1 % (k->i2 + 1);

}

// 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;

}

peg3.c0：

/\*

\* Peg solitaire solver

\*

\* 15-122 Principles of Imperative Computation \*/

#use <args>

#use <conio>

#use <util>

/\*

\* Copy here your code from peg2.c0 to start

\*/

#use "lib/peg-util.c0"

#use "peg-client.c0"

#use "lib/stacks.c0"

#use "lib/ht.c0"

typedef struct Pos\* move;

void alloc\_directions(int[][] d) {

// 上下左右四个方向

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

d[i]=alloc\_array(int,2);

}

d[0][0]=-1;d[0][1]=0;

d[1][0]=1;d[1][1]=0;

d[2][0]=0;d[2][1]=-1;

d[3][0]=0;d[3][1]=1;

}

int row\_start(move m) {

return m->start / 8;

}

int col\_start(move m) {

return m->start % 8;

}

int row\_end(move m) {

return m->end / 8;

}

int col\_end(move m) {

return m->end % 8;

}

int pow(int k)

//@requires k>=0;

//@ensures \result>=1;

{

return 1<<k;

}

htkey board\_to\_key(board B)

//@requires is\_board(B);

{

htkey k=alloc(struct two\_ints);

k->i1=0;

k->i2=0;

k->best\_num\_pegs=num\_pegs(B);

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

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

if(B[i\*8+j]==1) {

if(i<4) {

k->i1+=pow(i\*8+j);

}else {

k->i2+=pow((i-4)\*8+j);

}

}

}

}

return k;

}

/\*

获取当前棋局所有合法操作的堆栈

\*/

stack all\_solutions(board B)

//@requires is\_board(B);

//@ensures is\_board(B);

{

stack sols=stack\_new();

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

alloc\_directions(d);

// 遍历棋盘，寻找合法操作

for(int start\_x=0;start\_x<8;start\_x++) {

for(int start\_y=0;start\_y<8;start\_y++) {

if(B[start\_x\*8+start\_y]==1) {

// 如果该坐标上是棋子

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

int mid\_x=start\_x+d[k][0];

int mid\_y=start\_y+d[k][1];

int end\_x=mid\_x+d[k][0];

int end\_y=mid\_y+d[k][1];

int start=start\_x\*8+start\_y;

int mid=mid\_x\*8+mid\_y;

int end=end\_x\*8+end\_y;

if(0<=mid\_x && mid\_x<8 && 0<=mid\_y && mid\_y<8 && B[mid]==1 && 0<=end\_x && end\_x<8 && 0<=end\_y && end\_y<8 && B[end]==0) {

// 如果该棋子在某方向上相邻两格依次为棋子和洞，说明可进行一次合法操作

stackelem m=alloc(struct Pos);

m->start=start;

m->mid=mid;

m->end=end;

push(sols,m);

}

}

}

}

}

return sols;

}

/\*

获取棋局的一个解，并将操作堆栈保存在参数S中

\*/

int solve(board B, stack S, ht H, htkey k, int num\_pegs)

//@requires is\_board(B);

//@requires num\_pegs >= 1;

//@ensures is\_board(B);

//@ensures \result >= 1;

{

int best\_num\_pegs = num\_pegs;

if(ht\_lookup(H,k)!=NULL) {

return 0;

}

stack sols=all\_solutions(B);

if(stack\_empty(sols)) {

// 当当前操作堆栈为空时，无更多的合法操作

// 若此时棋盘上剩下的棋子个数为1，说明棋盘有解，反之无解

if(num\_pegs==1) {

return 1;

}else {

k->best\_num\_pegs=num\_pegs;

ht\_insert(H,k);

return num\_pegs;

}

}

while(!stack\_empty(sols)) {

move m=pop(sols);

// 对棋盘进行一次合法操作

B[m->start]=0;

B[m->mid]=0;

B[m->end]=1;

num\_pegs--;

htkey new\_k=alloc(struct two\_ints);

new\_k->i1=k->i1;

new\_k->i2=k->i2;

new\_k->best\_num\_pegs=num\_pegs;

if(m->start<32) {

new\_k->i1-=pow(m->start);

}else {

new\_k->i2-=pow(m->start-32);

}

if(m->mid<32) {

new\_k->i1-=pow(m->mid);

}else {

new\_k->i2-=pow(m->mid-32);

}

if(m->end<32) {

new\_k->i1+=pow(m->end);

}else {

new\_k->i2+=pow(m->end-32);

}

int left\_num\_pegs = solve(B,S,H,new\_k,num\_pegs);

if(left\_num\_pegs==1) {

push(S,m);

return 1;

}else if(best\_num\_pegs > left\_num\_pegs) {

best\_num\_pegs = left\_num\_pegs;

}

B[m->start]=1;

B[m->mid]=1;

B[m->end]=0;

num\_pegs++;

}

ht\_insert(H,k);

return best\_num\_pegs;

}

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(8000000);

return solve(B,S,H,board\_to\_key(B),num\_pegs(B));

}