# 作业6 孔明棋

本周的作业要求编写程序来解决一个孔明棋的棋盘。

代码材料在prog6.tgz中。材料包里的README.txt解释了材料中的内容，并说明如何提交作业。

请循序渐进地、频繁地测试你的代码。你可以不回溯地解决小体量的棋盘；并且，本次作业想要拿到满分，不一定需要使用优先队列或哈希表。尽量早些捕捉概念上的错误。

学术诚信要求：不可以查看他人代码或分享代码给他人，但可以在交流平台上分享孔明棋的棋盘，也可以询求棋盘样例来发现自己的代码到底出了什么问题。

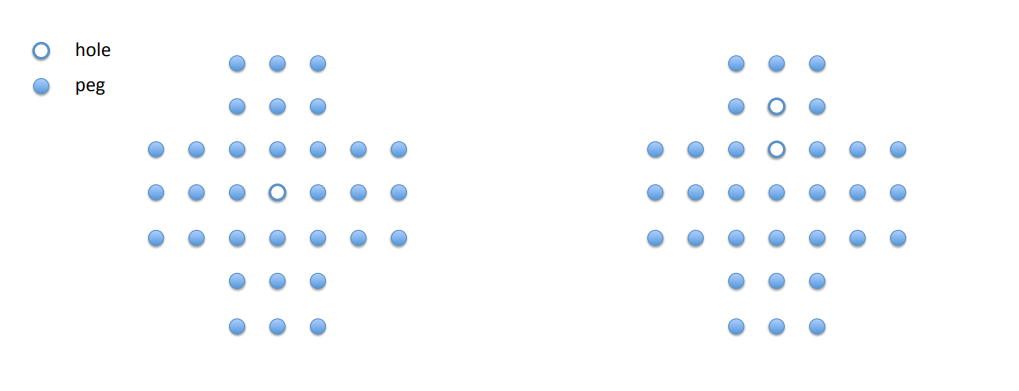
代码风格和约定：明显的代码约定问题将会导致扣分；也许会根据代码风格评分，特别是无法通过cc0的-w风格检查的代码。

这么做的原因之一是，你的代码很可能在从peg1.c0变到peg2.c0，peg2.c0再变到peg3.c0的过程中变得一团糟。为了成功完成本作业，你需要重构你的代码——它越变越复杂，你需要重构它，使其更适合本任务。根据代码风格的评分就是鼓励你做一些代码重构，因为重构也可能会帮助你编写代码和debug。

## 孔明棋

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。

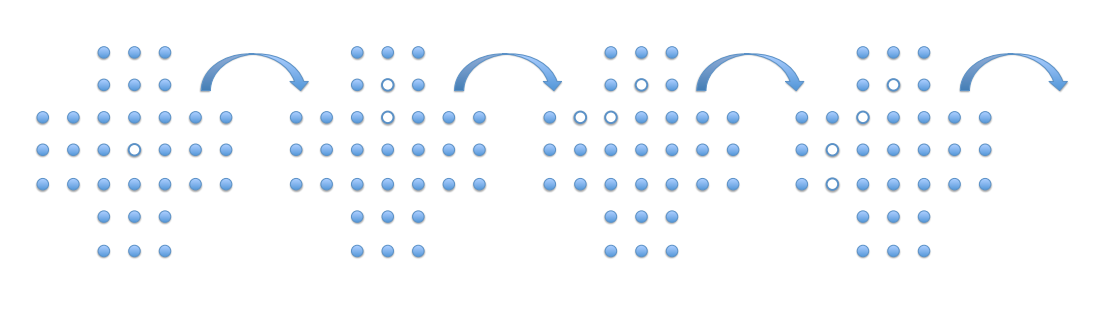


这个游戏的目标是只剩下一个棋子。在标准英式棋盘（上图左侧棋盘）中，我们从32颗棋子开始，所以任何解法都是31步的（每步移除一个棋子）。在该游戏的某些变种中，还指定了最后一个棋子的位置，但本作业不作要求，达到只剩一个棋子的状态就是唯一的目标。

孔明棋的更多信息请百度一下，你就知道。

## 电脑的玩法

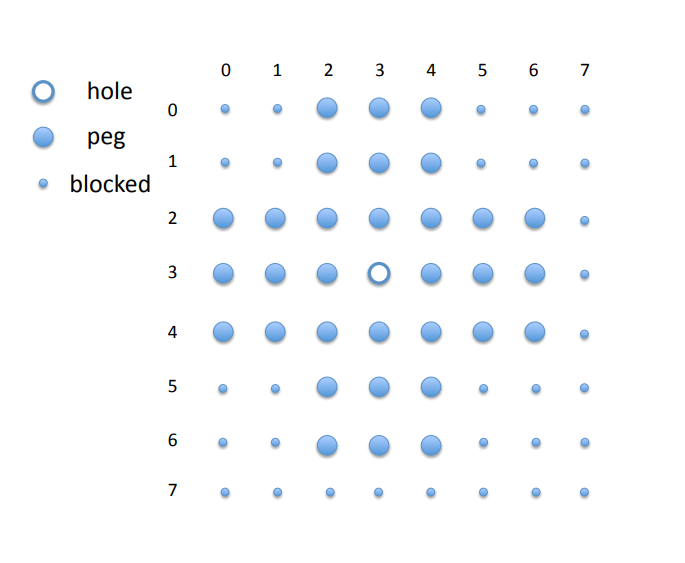
孔明棋很适合递归的解法。若我们想要解32颗棋子的棋盘，那么可以列举出这个棋盘所有的可能操作；做完其中一个操作后，我们面对的是同一个问题：解一个棋盘——只不过棋子数目是31颗。



如果棋盘上只剩一颗棋子，那么这个问题就简单了——我们赢辣！但是如果棋盘上有多于一颗的棋子，并且没有合法的操作，我们就输惹。每个操作系列都导向一个输局的棋盘是无解的。

## 棋盘的数据结构

在本次作业中，你需要自行决定合法操作、哈希键等等如何实现，但是棋盘的实现是我们规定好的。一个棋盘是一个大小为8x8=64的整数(Integer)数组，并且用-1表示该位置是墙（没有洞，不可以放棋子）；0表示该位置是没有放棋子的洞；1表示该位置是放了棋子的洞。棋盘从左上角开始编号。



我们用row : col表示一个位置，row和col都从左上角开始，且从0开始编号。比如之前的例子中，第一步就是从1：3移动到3：3。我们表示棋盘用的是一维数组，所以1：3对应的数组下标是1\*8+3=11, 而3：3对应的是3\*8+3=27。请查看lib/peg-util.c0中的定义，包括

typedef int[] board;

这个文件中也有从文件读/写一个棋盘所用的函数。

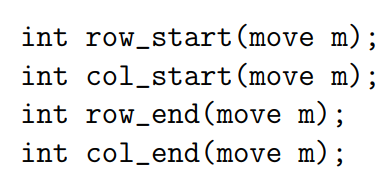
## 合法操作的数据结构

这个实现由你自行决定。

typedef \_\_\_\_\_\_ move；

关于该实现的一些建议：整数的三元组（对应一次操作改变的三个位置），整数的二元组（对应一次操作的第一个和最后一个索引），整数的四元组（对应棋子在操作前、后的行号、列号）。以上可以用结构体struct实现，或者压缩进某个类型，比如压缩入单个int。需要权衡数据的密度和找出合法操作的难度或速度。一次操作不会包括多次跳跃。

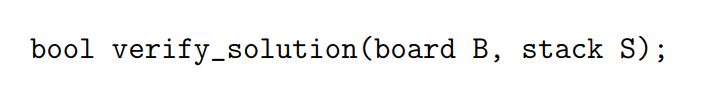
测试工具视move为抽象类型（abstract）。这意味着它只会用下列四个函数来检查一个给定的move **m**，查看它是否合法：



你可以在peg-main.c0中找到测试工具。该文件不可修改，但你可以查看它（还可以查看lib/peg-util.c0）并使用其中有用的东东。

## 解的数据结构

一局孔明棋的解是一个由操作组成的栈。栈顶应该是第一个操作，第二个元素应该是第二个操作，以此类推。peg-main.c0中的方程



检查S这个栈是孔明棋局B的一个合法解。

由于解是一个操作组成的栈，你的代码中需要在peg-client.c0中定义栈的元素的类型stackelem（在堆栈库被使用之前）。这个文件中也包括哈希表相关的用户定义（peg3.c0中可能用到）。

## 1 解一个确定的孔明棋局

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姐。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。这没关系，peg1.c0只会用于测试确定的棋局（可解/不可解）。

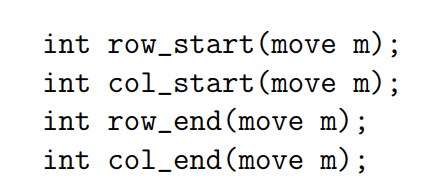
这个任务的目的是保证你正确地找出合法操作，正确地产生解，并且正确实现了指定的接口。参考peg-main.c0中的测试代码是很有用的（直接用，或者当作模板）。直接拷贝进peg1.c0并在注释中标明它的来源即可。也要记得查看lib/peg-util.c0，你可以调用其中任何函数（该文件在编译时被包括进来，所以不要复制函数，只要调用他们就好）

### 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。

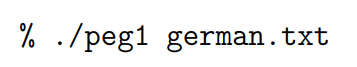
为了编写peg\_solve，你应该先写一个递归的辅助函数solve，它将被peg\_solve调用。这个辅助函数的参数应该包括棋盘、当前的操作堆栈、棋盘上剩下的棋子个数。在有解时辅助函数solve返回1；它也应该要在堆栈S中增加操作。思考：这个递归的函数的基本案例是什么？什么时候可以得知当前棋局有解？

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。

### 测试

你可以根据README.txt里的描述来编译你的peg1.c0，接着运行

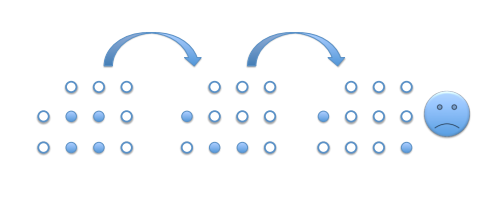


不过这是一个作用不大的测试，显示的信息很少，你需要用其他数据来测试你的代码（欢迎在交流平台上分享测试数据）。

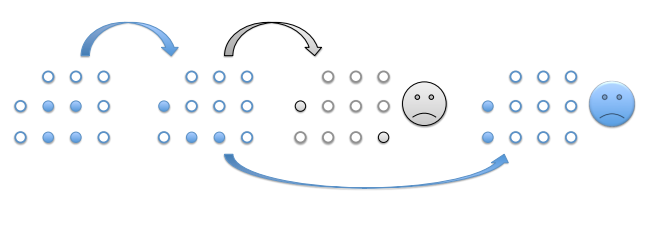
我们的测试只会用**确定的孔明棋局**（下一步永远只有1种或者0种可能）来测试你的任务1代码，但是你可能需要用更复杂的棋盘来测试你的代码。如果你这么做了，那么你的代码应该有时会过早地放弃，而找不到本应存在的解。

## 2 解一个不确定的孔明棋局

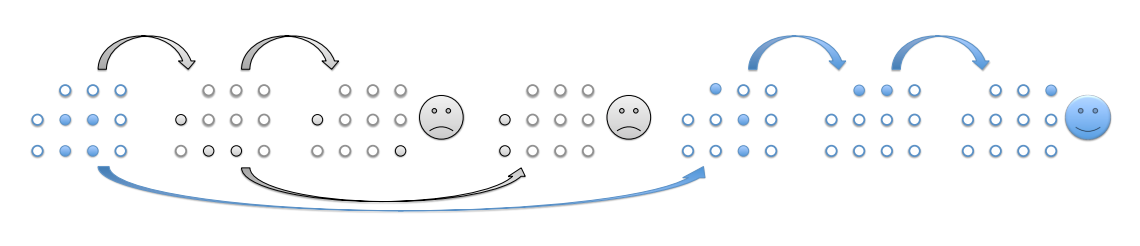
在先前的任务中，我们只解决了确定的孔明棋局：对于一个处在当前状态的棋盘，我们生成一个所有可能的下一步组成的堆栈，选择其中一个，然后在产生的下一个棋盘上继续玩孔明棋（这是递归的）。用这个方法，如果我们在求解过程中走到死路，很不幸——我们只能认输并返回棋盘上剩下的棋子个数。（下图中第一步是1：2到1：0，第二步是2：1到2：3）



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。



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。 如下图，我们在第一个棋盘选择走2：1到0：1，就会成功。



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。（你知道为什么吗？）

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。

### 任务2

把你的代码从peg1.c0 拷贝到peg2.c0。

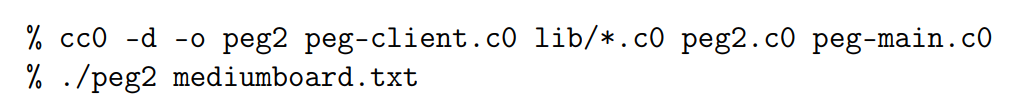
拓展peg\_solve方程，使其能够解决需要回溯法的棋盘。我们会用简单的需要回溯法的数据来测试你的代码。最复杂的一个是english.txt（标准英式棋盘，并且是初始状态）。

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。

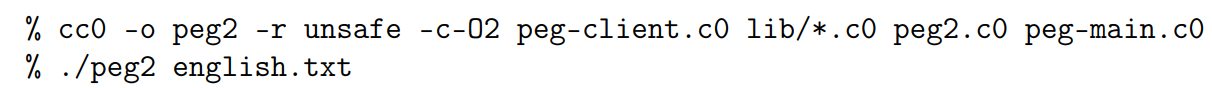
注意：在本任务中，*不要使用哈希表*。在你需要使用哈希表这种数据结构来缩减不断扩张的搜索空间之前，你可以探索使用此技术可以使问题变得多么复杂。

### 测试

你应该先用小的单元测试和容易解的棋盘（像README.txt中描述的那样加上-d选项）来测试peg2.c0。



接着，通过编译README.txt中描述的“速度快但不安全”的选项，你可以尝试解决更有难度的棋盘。



我们会用几个递增难度的棋盘来测试你的代码，并以english.txt（标准英式棋盘，并且是初始状态）作结。依赖你选择操作的顺序，你的代码可能可以解决english.txt，就算不使用哈希表。

你需要注意代码的效率，否则运行时间会很长。

### 一些建议

任务2代码应该会使用很多任务1的代码。你要解决的问题主要是棋盘在一次操作后的变化。当你需要撤销这一步操作时，棋盘需要变回先前的状态。这可以在solve函数中用一个不变量表示：在返回之前，棋盘必须返回它被调用时的状态。

如果你不这么做，而是拷贝下整个棋盘，你的代码可能会很慢。

这时效率绝对是一个重要因素。比如，你很可能要思考如何高效地查看当前棋局是否是赢局，因为这是需要频繁做的操作。我们推荐一种被称为暴力搜索的算法，而不是根据赢的可能性给所有可能的下一步操作排序。

peg1.c0中，几乎所有move的类型都足够拿到满分。但是，依赖你其他代码部分的设计，如果你用单个整数（integer）来表示move，你会发现效率提升不少（回忆ARGB像素是如何把4个值压缩进一个整数的）。是否这么实现完全是你的选择。如果你选择这种做法，别忘了修改peg2.c0和peg-client.c0！

## 3 附加题：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。在这个附加任务中，你会用哈希表来将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。

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。

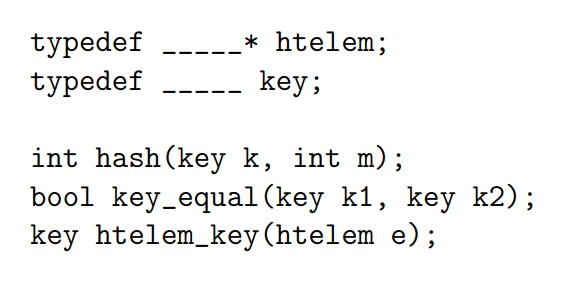
在key的选择方面，我们推荐把棋盘压缩成一个紧密的表达，同时包含足够的信息，使得对于一个特定问题，两个棋盘只有在形态完全相同时才对应同一个key。而key\_equal函数应该是很快的。注意你不可以不复制地用board数组本身，因为这样一来你的solve函数就会改变哈希表的内容。如果你的key是整个长度为64的整数数组，那么效率肯定太慢。拿字符串做key也是不好的，尽管在课程中我们有这么做过。

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。

### 任务3（附加）

把你的代码从peg2.c0 拷贝到peg3.c0。

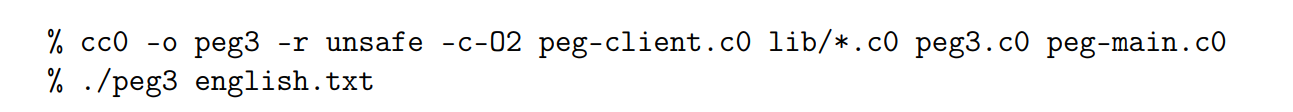
在peg-client.c0中，提供哈希表所需的用户端类型和函数，如下：



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。

### 测试

我们将测试你的代码如下



使用几个递增难度的棋盘，并以english.txt和french\*.txt（一个法式孔明棋）作结。

和任务2中一样，你的代码需要有合理的效率，且不能使用无限的系统资源。

### 一些建议

这时效率成为了一个主要因素。move的选择仍是一个主要问题。你的哈希函数的效率、key值的紧密程度、哈希表的大小，以及你的哈希函数防止冲突的能力，也都是重要的因素。这些问题在performance-debugging.txt中有部分地讨论。

注意，为了让哈希正常工作，key和元素不可以在放入哈希表后再更新。如果你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。

即使有了记忆化和写的很好的代码，一部分棋盘（包括法式孔明棋）也可能让你的程序跑很久很久。发生这种情况的时候不要认为你的代码出错了。记住法式孔明棋不影响你的最终得分，而是一个附加题。

### 更多

我们的解法完全是暴力的，对于可能的下一步move没有任何启发式的排序。做完本作业以后，你可以考虑加入这样启发式的排序，选择先走最有希望的move。题目条件不变。如果你完成了这部分，请在HEURISTIC.txt中描述你的做法。

## 附录A Move的选择

选择move的顺序不会对无解的棋盘的搜索效率产生很大影响，但对有解的棋盘的搜索效率很重要。如果通过搜索棋盘来选择move，那么有三个明显的选择：

* 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）；
* 每次找到一个棋子，查看它能否在各个方向被跳过。
* 每次找到一个空格，查看它能否在各个方向被跳入。

我们推荐你使用第一种策略，然后尝试方向的不同排序（上下左右、上右下左，等等）来找到对英式棋盘最有效的一种。方向的正确排序策略，可以让你在任务3的english.txt中非常快地找到答案（解决完问题后哈希表里有少于1100个棋盘）。

还有另外一个你可以尝试的选择：第一种做法是在每个棋子/空格处尝试各个方向，然后找下一个；第二种做法是在每个棋子/空格处只尝试一个方向，走到头以后再尝试另外的方向。这也会造成很大的不同，取决于各种因素。