2021-2022 年度春夏学期浙江大学人工智能本科专业《人工智能基础》课程实训技术报告

姓名：秦嘉俊　学号：3210106182

学院（系）专业：竺可桢学院，计算机科学与技术（图灵班）

c++ a+=b;

1　问题概述

八皇后问题：在 8×8 格的国际象棋上摆放 8 个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种摆法。

2　算法描述

在八皇后问题的解决中，我们主要采用深度优先搜索（DFS, i.e.Depth-First Search）的算法来搜索出八皇后所有可能的放置状态。

2.1　深度优先搜索算法与回溯法

首先我们定义**搜索树**。在对图进行深度优先遍历位于点 x 时，对于某些边 (x, y)，y 是一个尚未访问过的节点，程序从 x 成功进入了更深层的对 y 的递归；对于另外的一些边 y，y 已经被访问过，从而程序继续考虑其他分支。我们称所有点（问题空间中的状态）与成功发生递归的边（访问两个状态之间的移动构成的树为一棵“搜索树”。深搜算法就算基于搜索树而完成的。

之所以称为**深度优先**，是因为它优先对可能的分支路径不停地深入，直到不能再深入为止，随后逐层往上回溯，重复寻找分支并深入，如此循环往复，直到遍历完搜索树的所有节点。

回溯法的基本思想是按照深度优先搜索的策略，从根节点开始搜索。当到某个节点时要判断是否是包含问题的解，如果包含就从该节点继续搜索下去；如果不包含，就向父节点回溯。

2.2　八皇后问题中的深搜算法

假设我们目前处于搜索树的第 k 层，对应了第 k 个皇后我们要放的位置。当达到这一层时，我们枚举所有可能的放置情况，并通过深度优先的搜索决定哪些放置方法是可行的，所有尝试均结束后回溯到上一层，到其他分支进行新的搜索。

本问题中我以行为状态，当处于第 k 行时，我枚举从第 0 列开始一直到第 7 列，若第 i 列是可以被放置的，那么我们将答案数组第 k 位设为 i，随后继续搜索 k+1 行，直到最后（即到达第八行时结束）。

2.3 八皇后问题中的边界判断

为了判断一个位置是否能放皇后，我们利用一个数组 unableMatrix 来记录目前的皇后情况。当 `unableMatrix[x][y] = True` 时，表示第 x 行第 y 列可以被放皇后，若为 False 则不能。每次我们放置皇后后，需要更新其他位置的 unableMatrix 状态。将最新放置皇后所在的这一行和这一列以及它所在的对角线的 unableMatrix 设为 False 即可。

我们从第零行开始，当到达第八行时表明我们已经搜出了一种可能的放置，于是把它加到答案数组中，回溯到上一层尝试其他解法。

3 代码实现

本次实验使用 Python 语言解决问题。

3.1 准备工作

- `board.py`
  这个文件里面提供了 `Chessboard` 这个类及其方法的实现。实验提前给出，不需要再修改。
  类中包括 `boardInit` 初始化棋盘，`trans` 棋盘绘制字符转换，`isOnChessboard` 判断落子是否在棋盘上，`setLegal` 禁止落子位置指定，`setQueen` 尝试放置皇后（若失败会输出“落子失败”），`play` 用于玩家互动体验游戏，`isWin isLose` 判断玩家互动时输赢状况，`printChessboard`绘制棋盘

- `main.py`
  这个文件里面提供了 `Game` 这个类及其方法的实现。大部分内容实验提前给出，我们要做的就是修改其中的 `run` 方法以实现搜索八皇后答案的功能。
  类中包括 `gameInit` 重置棋盘，`trans` 棋盘绘制字符转换，`run` 搜索答案，`setLegal` 禁止落子位置指定，`showResults` 展示结果，`get_results` 搜索并展示结果。
  需要注意的是 `Game` 中有两个重要的 attributes, `solves`, 以及 `Queen_setRow`.

  - `Queen_setRow` 用来存放目前正在搜索的这种放置策略
    注意到每一个答案都是一个 list, 如 `[0,6,4,7,1,3,5,2]` 就是答案的一种方式，它表示第一行的皇后放在第零列，第二行的皇后放在第六列，以此类推。当我们搜索到第八行时，说明我们已经搜索完毕且目前的放置没有冲突，则我们将答案更新。

  - `solves` 用来存放答案。
    而 `solves` 作为一个 list, 它的元素也应该是 list. 因此在更新答案的时候我们应该 `self.solves.append(self.Queen_setRow)`

3.2 搜索部分

run 方法的实现如下

```
def run( self , row=0):
    if row == 8:
        self . solves .append( self .Queen_setRow[:])
        return
    Saved_Chessboard = copy.deepcopy(self.chessBoard)
    for i in range(8):
        if self .chessBoard.setQueen(row, i , False ):
            self .Queen_setRow[row] = i
            self .run(row + 1)
            self .chessBoard = copy.deepcopy(Saved_Chessboard)
```

3.2.1 答案更新

row 默认为 0, 表示我们从第零行开始，当 row=8 时表明我们已经搜出了一种可能的策略，便通过准备工作里介绍的方法更新答案。需要注意的是，这里我们不能单单采用self.solves.append(self.Queen_setRow), 而是 self.solves.append(self.Queen_setRow[:]), 这是由于 Python 本身的一个特性。当我们使用前者时,程序并不会把 Queen_setRow 中的元素复制并添加到 solves 中去，而是以一种类似指针的方式，将 solves 尾部元素指向 Queen_setRow。乍一看没有什么问题，但我们全局只使用了 Queen_setRow 这唯一的一个对象来记录答案，这就意味着 Queen_setRow 在后续的运行中很可能会被更新，这就导致我们先前试图存进去的答案不见了。最后所有可能的答案都变成了同一个最后存入的答案（因为每次 append 时就会把新的位置指向 Queen_setRow）

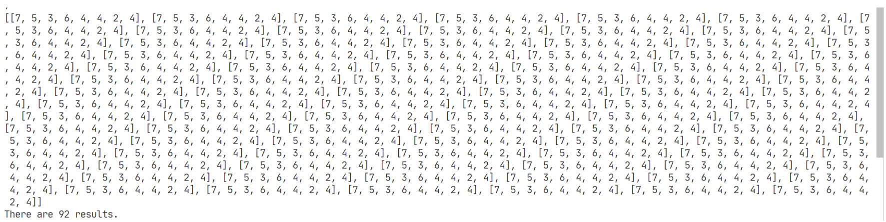

图 1: 错误地添加答案

而换用后者后，每次 Queen_setRow[:] 相当于拷贝原 list 并建立了一个新的对象，这样每次更新答案我们都是将答案指向新的对象，而新的对象不会在后续中更改，这样我们就实现了答案的更新。

3.2.2 决策当前放置的位置

代码第 6 至 10 行则是在枚举当前的皇后能放的位置，对每个位置都利用 board.py 里的 `setQueen` 来判断是否与前面的皇后冲突，如果不冲突就放下，并将目前这个位置记入当前答案，随后继续搜索下一行。

这里需要注意的是回溯问题，当一个放置位置被搜索完后，我们需要回退到这个位置被放之前的状态，而这里最重要的就是在 `setQueen` 中 `Chessboard` 的信息被改变了，我们要做的就是在每次搜索完后恢复原本的信息。同时出于类似于上文的 Python 的一些拷贝特性，这里我们需要用到 `copy` 模块中的深拷贝功能，以免出现和上文类似的问题。

4 算法实验结果与分析

```
hobbitqia@LAPTOP-EBSFK1UV:~/ai$ python3 1.py
  0 1 2 3 4 5 6 7
0 - - - - - - - -
1 - - - - - - - -
2 - - - - - - - -
3 - - - - - - - -
4 - - - - - - - -
5 - - - - - - - -
6 - - - - - - - -
7 - - - - - - - -
There are 92 results.
  0 1 2 3 4 5 6 7
0 o - - - - - - -
1 - - - - o - - -
2 - - - - - - - o
3 - - - - - o - -
4 - - o - - - - -
5 - - - - - - o -
6 - o - - - - - -
7 - - - o - - - -
```

图 2: 运行结果

可以看出，总共有 92 种可能的八皇后放置方案。(我在 Appendix 附上了所有的放置答案) 这与我们在网上搜索到的答案相同，说明算法基本正确。同时按照实验给出的框架代码，程序会输出我们找到的第一种解决方案。

5 算法进一步研究展望

经过我上网搜索发现, N 皇后问题不存在完整的数学公式，而且随着 N 的增长，答案的规模也随之呈指数级的增长。

n	fundamental	all
1	1	1
2	0	0
3	0	0
4	1	2
5	2	10
6	1	4
7	6	40
8	12	92
9	46	352
10	92	724
11	341	2,680
12	1,787	14,200
13	9,233	73,712
14	45,752	365,596
15	285,053	2,279,184
16	1,846,955	14,772,512
17	11,977,939	95,815,104
18	83,263,591	666,090,624
19	621,012,754	4,968,057,848
20	4,878,666,808	39,029,188,884
21	39,333,324,973	314,666,222,712
22	336,376,244,042	2,691,008,701,644
23	3,029,242,658,210	24,233,937,684,440
24	28,439,272,956,934	227,514,171,973,736
25	275,986,683,743,434	2,207,893,435,808,352
26	2,789,712,466,510,289	22,317,699,616,364,044
27	29,363,495,934,315,694	234,907,967,154,122,528

图 3: N 皇后问题对应的答案数量（图源：维基百科）

因此对应的搜索树规模一定是相当巨大的，简单的搜索算法不足以满足 N 逐渐增长的需求。如果想要尽可能快地计算出结果，我们需要优化我们的搜索方法。

5.1 可行性剪枝

在搜索过程中，及时对当前状态进行检查，如果发现分支已经无法达到递归边界，就执行回溯。这就好比我们在道路上行走时，远远看到前方是一个死胡同，就应该立即折返绕路，而不是走到路的尽头再返回。

朴素的 N 皇后算法（即枚举每一行皇后可以放的位置，共 N^N 种可能随后再判断这种放置是否满足要求）的时间复杂度为 $O(N^N)$，我们采用的算法实际上已经使用了可行性剪枝的方法，即每次选择皇后放置位置时判断这个位置是否已经和前面的皇后冲突，如果冲突就不会在这个位置放置皇后，减少无效搜索。这样优化之后，从算法上来讲，N 皇后问题本质相当于全排列问题，因此我们算法时间复杂度 $O(N!)$, 空间复杂度 $O(N)$.

5.2 排除等效冗余

在搜索过程中，如果我们能判定从搜索树的当前节点上沿着某几条不同分支到达的子树是等效的，那么只需要对其中的一条分支执行搜索。

我们可以对这个解法再次进行优化，我们可以把第一个皇后的位子限制到前半个区间（即第一行的前一半列），这样对于偶数个皇后来说，我们只需要搜索一半的空间就可以得到最终的结果，对于奇数个皇后，我们只需要求解前半个区间和中间值，最后结果为前半个区间解数 * 2 + 中间值的解数。这样可以缩小搜索空间，提升效率。

值得一提的是，N 皇后本身是一个 NP 完全问题，目前不存在多项式级别的算法解决。以上优化也只是有一定的效果，但算法整体的复杂度仍然高居不下。

6 心得体会

逻辑编程是一种编程典范，它设置答案须匹配的规则来解决问题，而非设置步骤来解决问题。过程是“事实 + 规则 = 结果”。本次实验中，我切实体会到何为逻辑编程。当我们输入八皇后的规则和搜索策略后，程序就能给我们答案。

同时，本次实验还帮助我重拾了久别的 Python 语法，在踩了几个经典的坑点（list 的 append 方式, 深拷贝等）后方才恍然大悟。这也促使我要在国庆假期温习一遍 Python 的语法和应用，这样在后续的实验中才能更好地将目光集中在问题的解决上。

此外，我也在自己解决八皇后问题中回忆起高中竞赛中所学的搜索算法及其剪枝。在研究自身算法并想办法优化时，我着实感受到算法的魅力与奇妙。虽然最后没能做出太多的优化，但我在上网搜索学习，思考算法中也回顾了几种常见的剪枝方法（优化搜索顺序，排除等效冗余，可行性剪枝，最优化剪枝，记忆化）而搜索也是 AI 中的一个相当重要的内容，相信这次的探索不是做无用功，而会大有裨益。

最后，期待下次实验带给我新的惊喜。

Appendix

[0, 4, 7, 5, 2, 6, 1, 3]	[0, 5, 7, 2, 6, 3, 1, 4]	[0, 6, 3, 5, 7, 1, 4, 2]
[0, 6, 4, 7, 1, 3, 5, 2]	[1, 3, 5, 7, 2, 0, 6, 4]	[1, 4, 6, 0, 2, 7, 5, 3]
[1, 4, 6, 3, 0, 7, 5, 2]	[1, 5, 0, 6, 3, 7, 2, 4]	[1, 5, 7, 2, 0, 3, 6, 4]
[1, 6, 2, 5, 7, 4, 0, 3]	[1, 6, 4, 7, 0, 3, 5, 2]	[1, 7, 5, 0, 2, 4, 6, 3]
[2, 0, 6, 4, 7, 1, 3, 5]	[2, 4, 1, 7, 0, 6, 3, 5]	[2, 4, 1, 7, 5, 3, 6, 0]
[2, 4, 6, 0, 3, 1, 7, 5]	[2, 4, 7, 3, 0, 6, 1, 5]	[2, 5, 1, 4, 7, 0, 6, 3]
[2, 5, 1, 6, 0, 3, 7, 4]	[2, 5, 1, 6, 4, 0, 7, 3]	[2, 5, 3, 0, 7, 4, 6, 1]
[2, 5, 3, 1, 7, 4, 6, 0]	[2, 5, 7, 0, 3, 6, 4, 1]	[2, 5, 7, 0, 4, 6, 1, 3]
[2, 5, 7, 1, 3, 0, 6, 4]	[2, 6, 1, 7, 4, 0, 3, 5]	[2, 6, 1, 7, 5, 3, 0, 4]
[2, 7, 3, 6, 0, 5, 1, 4]	[3, 0, 4, 7, 1, 6, 2, 5]	[3, 0, 4, 7, 5, 2, 6, 1]
[3, 1, 4, 7, 5, 0, 2, 6]	[3, 1, 6, 2, 5, 7, 0, 4]	[3, 1, 6, 2, 5, 7, 4, 0]
[3, 1, 6, 4, 0, 7, 5, 2]	[3, 1, 7, 4, 6, 0, 2, 5]	[3, 1, 7, 5, 0, 2, 4, 6]
[3, 5, 0, 4, 1, 7, 2, 6]	[3, 5, 7, 1, 6, 0, 2, 4]	[3, 5, 7, 2, 0, 6, 4, 1]
[3, 6, 0, 7, 4, 1, 5, 2]	[3, 6, 2, 7, 1, 4, 0, 5]	[3, 6, 4, 1, 5, 0, 2, 7]
[3, 6, 4, 2, 0, 5, 7, 1]	[3, 7, 0, 2, 5, 1, 6, 4]	[3, 7, 0, 4, 6, 1, 5, 2]
[3, 7, 4, 2, 0, 6, 1, 5]	[4, 0, 3, 5, 7, 1, 6, 2]	[4, 0, 7, 3, 1, 6, 2, 5]
[4, 0, 7, 5, 2, 6, 1, 3]	[4, 1, 3, 5, 7, 2, 0, 6]	[4, 1, 3, 6, 2, 7, 5, 0]
[4, 1, 5, 0, 6, 3, 7, 2]	[4, 1, 7, 0, 3, 6, 2, 5]	[4, 2, 0, 5, 7, 1, 3, 6]
[4, 2, 0, 6, 1, 7, 5, 3]	[4, 2, 7, 3, 6, 0, 5, 1]	[4, 6, 0, 2, 7, 5, 3, 1]
[4, 6, 0, 3, 1, 7, 5, 2]	[4, 6, 1, 3, 7, 0, 2, 5]	[4, 6, 1, 5, 2, 0, 3, 7]
[4, 6, 1, 5, 2, 0, 7, 3]	[4, 6, 3, 0, 2, 7, 5, 1]	[4, 7, 3, 0, 2, 5, 1, 6]
[4, 7, 3, 0, 6, 1, 5, 2]	[5, 0, 4, 1, 7, 2, 6, 3]	[5, 1, 6, 0, 2, 4, 7, 3]
[5, 1, 6, 0, 3, 7, 4, 2]	[5, 2, 0, 6, 4, 7, 1, 3]	[5, 2, 0, 7, 3, 1, 6, 4]
[5, 2, 0, 7, 4, 1, 3, 6]	[5, 2, 4, 6, 0, 3, 1, 7]	[5, 2, 4, 7, 0, 3, 1, 6]
[5, 2, 6, 1, 3, 7, 0, 4]	[5, 2, 6, 1, 7, 4, 0, 3]	[5, 2, 6, 3, 0, 7, 1, 4]
[5, 3, 0, 4, 7, 1, 6, 2]	[5, 3, 1, 7, 4, 6, 0, 2]	[5, 3, 6, 0, 2, 4, 1, 7]
[5, 3, 6, 0, 7, 1, 4, 2]	[5, 7, 1, 3, 0, 6, 4, 2]	[6, 0, 2, 7, 5, 3, 1, 4]
[6, 1, 3, 0, 7, 4, 2, 5]	[6, 1, 5, 2, 0, 3, 7, 4]	[6, 2, 0, 5, 7, 4, 1, 3]
[6, 2, 7, 1, 4, 0, 5, 3]	[6, 3, 1, 4, 7, 0, 2, 5]	[6, 3, 1, 7, 5, 0, 2, 4]
[6, 4, 2, 0, 5, 7, 1, 3]	[7, 1, 3, 0, 6, 4, 2, 5]	[7, 1, 4, 2, 0, 6, 3, 5]
[7, 2, 0, 5, 1, 4, 6, 3]	[6, 1, 3, 0, 7, 4, 2, 5]	