八皇后启发式搜索

一、实验目的：

①掌握逻辑与推理的相关知识基础知识点

②了解算法推理的相关基础知识

③掌握搜索算法的主要步骤

④熟悉 python 编程

二、实验环境：

软件环境：python3.7

开发环境：Modelarts: pytorch1.4-cuda10.1-cudnn7-ubuntu18.04

三、待解决的问题：

如何能够在 8×8 的国际象棋棋盘上放置八个皇后，使得任何一个皇后都无法直接吃掉其他的皇后？为了达到此目的，任两个皇后都不能处于同一条横行、纵行或斜线上。

四、问题分析：

状态空间：用 N*N 的矩阵 Qans[N][N]表示当前棋盘皇后放置的位置，若第 i 行第 j 列放置了皇后，则 $Qans[i-1][j-1]=1,\ i,\ j\ =\ 1,2,3,4,5,6,7,8$ 否则 $Qans[i-1][j-1]=0$

操作规则：从第一行开始摆皇后，第一个皇后 Q 放在第 1 行，第 k 个皇后 Q 放在第 k 行且不能与之前所有的皇后互相攻击。

初始状态：初始时棋盘中没有放置皇后，矩阵 Qans[N][N]中所有元素的值为 0

目标状态：每一行中均有一个皇后 即 $Sum(Qans[i-1,:])=1\ ,\ i=1,2,3,4,5,6,7,8$

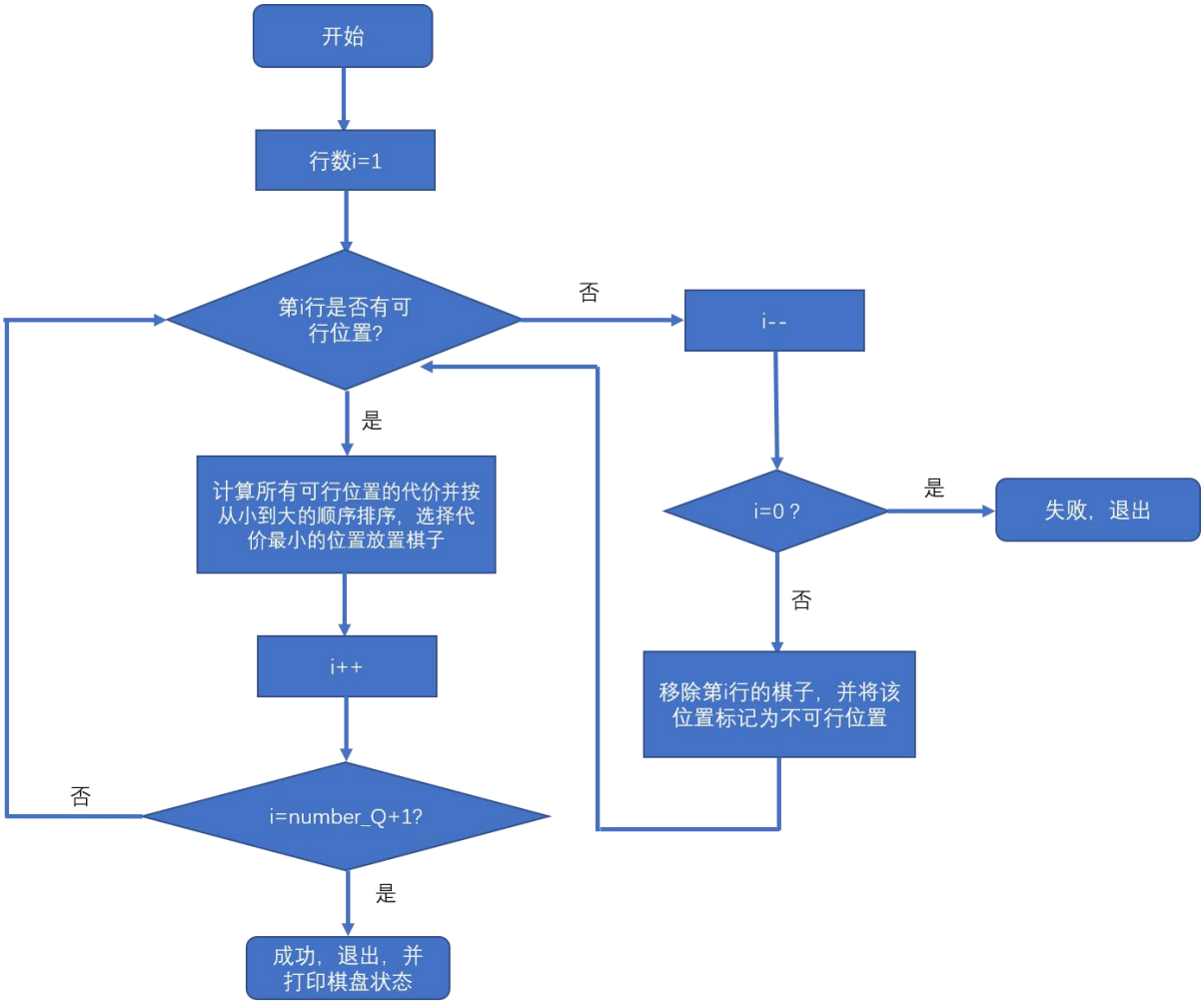
启发式函数：在第 i 行的某一个格子放置皇后之后，会导致剩下的 8-i 行中部分格子无法放置皇后，我们将此步导致的剩下 8-i 行中无法放置皇后的格子数目记作此步的代价。我们令 C(x,y)=在第棋盘第 x 行，第 y 列放置皇后后，导致剩余 8-x 行中无法放置皇后的格子数目。以 C(x,y)作为搜索时的启发式函数。

算法概要：启发式+dfs+回溯算法

在第 i 行放置皇后时，先判断该行是否有格子能放置皇后。若有，计算出所有可行格子的 代价 C(x,y), 按从小到大的顺序排序，优先选择在 C(x,y)最小的格子放置皇后，之后在第 i+1 行放置皇后。若第 i 行没有格子可以放置皇后，则回溯到上一行（即第 i-1 行），将第 i-1 行的皇后放置在该行 C(x,y)次小的位置，直到搜索出目标状态。

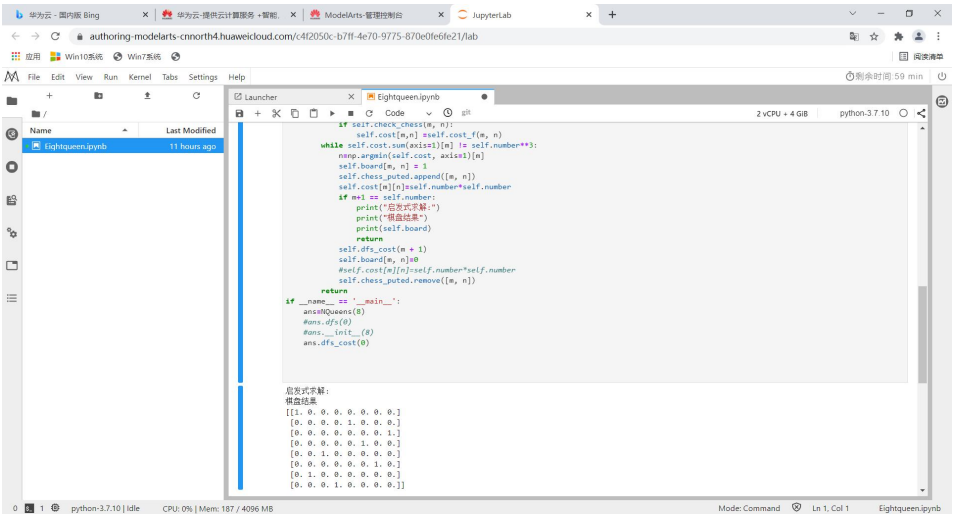
人工智能概论 19170100004 付凯文

算法流程：



程序代码：见附录

运行结果：利用启发式搜索成功搜索出了八皇后问题的一个解，其中 1 代表该位置放置了皇后，0 代表没有放置皇后。



五、实验收获

通过本次实验，我了解并掌握了回溯算法，对 DFS 算法有了更加熟练的应用。同时，对于启发式搜索，我有了更深层次的理解

六、附录

```
1.  import numpy as np
2.  class NQueens:
3.
4.      def __init__(self,number):
5.          #number 为皇后的数量
6.          self.number=number
7.          self.board=np.zeros((number,number))
8.          self.chess_puted = []
9.          #将棋子下在(m,n)的代价
10.         self.cost=(number*number)*np.ones((number,number))
11.
12.     #检查(m,n)是否可以放置皇后
13.     def check_chess(self, m, n):
14.         if len(self.chess_puted)==0:
15.             #棋盘为空则(m,n)处一定可以放置皇后
16.             return True
17.         else:
18.             #棋盘非空，判定是否会相互攻击
19.             coordinate = np.array([m, n])
20.             for i in range(len(self.chess_puted)):
21.                 # 坐标之差
22.                 diff = coordinate-self.chess_puted[i]
23.                 # 在(m,n)放置皇后之后，是否会与已放置的皇后是否相互攻击
24.                 if diff[0] != diff[1] and diff[0] != -diff[1] and diff[0] !=
    0 and diff[1] != 0:
25.                     continue
26.                 else:
27.                 #相互攻击则该位置不能放置
28.                     return False
29.         return True
30.
31.     #计算代价
32.     def cost_f(self,m,n):
33.         cost=0
34.         row,col=m,n
35.         while row<self.number and col>-1:
36.             row+=1
37.             col-=1
```

```
            cost+=1
        row,col=m,n
        while row<self.number and col<self.number:
            row+=1
            col-=1
            cost+=1
        return cost

    #启发式搜索
    def dfs_cost(self, m):
        for n in range(self.number):
            if self.check_chess(m, n):
                self.cost[m,n] =self.cost_f(m, n)
        while self.cost.sum(axis=1)[m] != self.number**3:
            n=np.argmin(self.cost, axis=1)[m]
            self.board[m, n] = 1
            self.chess_puted.append([m, n])
            self.cost[m][n]=self.number*self.number
            if m+1 == self.number:
                print("启发式求解:")
                print("棋盘结果")
                print(self.board)
                return
            self.dfs_cost(m + 1)
            self.board[m, n]=0
            #self.cost[m][n]=self.number*self.number
            self.chess_puted.remove([m, n])
        return
if __name__ == '__main__':
    ans=NQueens(8)
    #ans.dfs(0)
    #ans.__init__(8)
    ans.dfs_cost(0)
```