《人工智能导论》作业 2025-01

N 皇后问题

专　　业 软件工程

姓　　名 郭　政

学　　号 2023141461076

指导老师 毋攀良

成绩分数

二零二五 年 五 月 二十 日

N 皇后问题实验报告

一、算法设计

1. 基本思路

采用回溯法逐行尝试在棋盘上放置皇后，每次放置前判断该位置是否被当前路径中的皇后攻击。若可放置，则递归尝试下一行；否则回溯并尝试其他列。

2. 剪枝策略

为了提高效率，引入如下优化：

（1）使用三个辅助结构判断攻击冲突：

[1] cols[col]：当前列是否已有皇后；

[2] hill_diagonals[row - col]：主对角线冲突；

[3] dale_diagonals[row + col]：副对角线冲突；

（2）这三者可以用布尔值快速判断，从而有效减少无效搜索路径。

二、程序功能实现

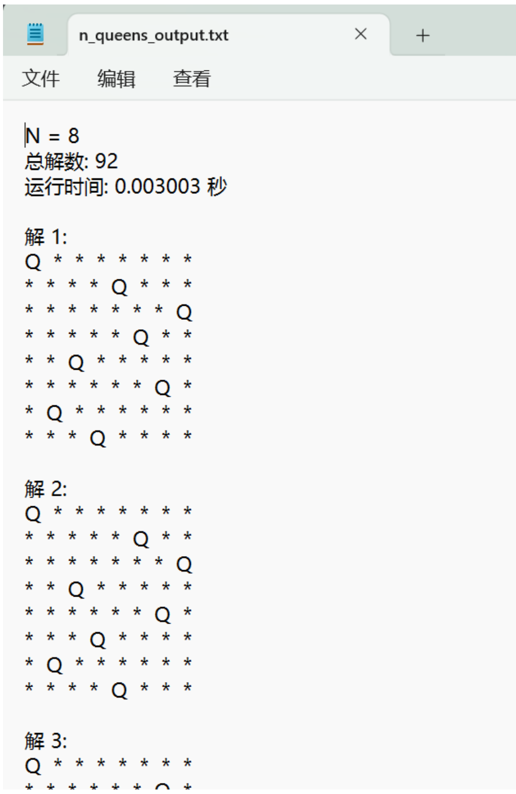
（1）用户输入合法整数 N（N ≥ 4），否则提示重新输入

（2）通过参数控制是否输出所有解或只输出一个解

（3）所有合法解以棋盘形式展示，写入 n_queens_output.txt 文件

（4）记录解总数与运行时间

（5）代码采用模块化结构，具备良好的可读性和扩展性

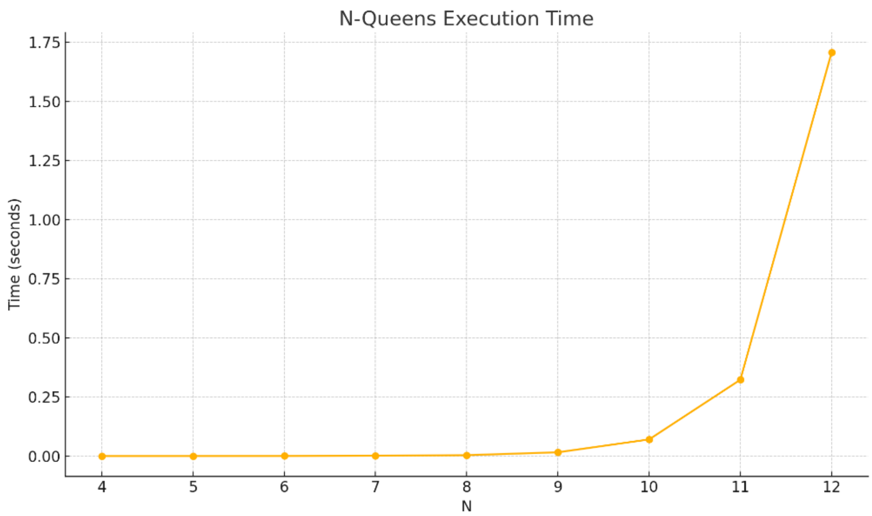


```
请输入一个大于等于4的整数N: 8
是否在控制台输出所有解? 输入 Y 输出所有解, 输入 N 仅输出一个解: n

其中的一个可行解:
Q  *  *  *  *  *  *  *
*  *  *  *  Q  *  *  *
*  *  *  *  *  *  *  Q
*  *  *  *  *  Q  *  *
*  *  Q  *  *  *  *  *
*  *  *  *  *  *  Q  *
*  Q  *  *  *  *  *  *
*  *  *  Q  *  *  *  *

总解数: 92
运行时间: 0.003003 秒
N = 8 的所有解已经写入文件 'n_queens_output.txt'
```

```
请输入一个大于等于4的整数N: 4
是否在控制台输出所有解? 输入 Y 输出所有解, 输入 N 仅输出一个解: y

解 1:
*  Q  *  *
*  *  *  Q
Q  *  *  *
*  *  Q  *

解 2:
*  *  Q  *
Q  *  *  *
*  *  *  Q
*  Q  *  *

总解数: 2
运行时间: 0.000000 秒
N = 4 的所有解已经写入文件 'n_queens_output.txt'
```

三、实验结果

使用 time 模块记录从 N=4 到 N=12 的运行时间与解的数量：



从 N=4 到 N=12 的运行时间

1. 理论最大时间复杂度：O(N!)

N 皇后问题的回溯算法，本质是一个深度为 N 的决策树：

（1）每一层代表一行（从第 0 行到第 N−1 行）

（2）每一层最多有 N 个分支（列）

在没有剪枝的情况下，递归尝试每一行所有列，理论时间复杂度为：

$$T(N) = N \times (N-1) \times (N-2) \times \cdots\cdots \times 1 = O(N!)$$

即：最多尝试 N! 种放置方式。

2. 实际复杂度：远优于 O(N!)，依赖剪枝有效性

使用了三种剪枝判断，显著减少无效递归分支：

列剪枝：用 cols[col] 数组排除同列冲突。

```
# 剪枝：判断当前位置(row, col)是否安全
def is_not_under_attack(row, col):
    return not (cols[col] or hill_diagonals[row - col] or dale_diagonals[row + col])
```

主对角线剪枝：用 hill_diagonals[row - col] 排除左上到右下冲突。

副对角线剪枝：用 dale_diagonals[row + col] 排除右上到左下冲突。

```
# 放置皇后，并标记列和对角线
def place_queen(row, col):
    queens[row] = col
    cols[col] = True
    hill_diagonals[row - col] = True
    dale_diagonals[row + col] = True
```

因此，实际尝试的状态远少于 N!，实验中 N=12 时也可在秒级时间内完成。

```
# 回溯主逻辑
def backtrack(row=0):
    for col in range(n):
        # 剪枝：如果当前位置安全才尝试放置皇后
        if is_not_under_attack(row, col):
            place_queen(row, col)
            if row + 1 == n:
                add_solution()
                if not output_all:
                    return True  # 只需找到一个解时提前返回
            else:
                found = backtrack(row + 1)
                if found and not output_all:
                    return True
            # 回溯：撤销当前选择，尝试下一个位置
            remove_queen(row, col)
    return False
```

3. 空间复杂度分析：O(N)

每次递归只记录当前棋盘状态和状态标记，使用了以下辅助空间：

queens[n]: 存储每行皇后列号；

cols[n]: 列标记；

hill_diagonals[2n-1], dale_diagonals[2n-1]: 对角线标记。

因此总空间复杂度为：

$$O(N)+O(N)+O(2N-1)+O(2N-1)=O(N)$$

4. 解的数量（搜索空间大小）

已知 N 皇后问题的解数量增长如下：

N	**解的数量**
4	2
5	10
6	4
7	40
8	92
9	352
10	724
11	2680
12	14200

实际分支大大少于 N!，剪枝和状态记录极大地缩小了搜索空间。

四、优化思路

1. 剪枝策略

为了避免无效状态的递归扩展，使用了以下三个布尔结构进行剪枝判断：

列冲突检查（cols[col]）：用于判断当前列是否已有皇后。

主对角线冲突检查（hill_diagonals[row - col]）：用于检查从左上到右下方向是否冲突。

副对角线冲突检查（dale_diagonals[row + col]）：用于判断从右上到左下方向是否有皇后。

2. 状态表示优化

使用 queens[i]=col 表示第 i 行皇后放置在第 col 列，这种一维数组简化了棋盘的记录，节省了空间。

棋盘输出仅在记录解时生成，避免在递归过程中频繁构建字符串，提升了运行效率。

3. 提前终止策略（可选一个解）

若用户选择仅需一个解，程序在找到第一个合法解后立即返回，避免继续搜索所有可能状态，显著减少计算时间。

4. 使用 defaultdict(bool) 防止 KeyError

对对角线标记使用 Python 的 collections.defaultdict(bool)，避免了频繁初始化，提高代码稳定性与简洁性。

```
solutions = []
cols = [False] * n
hill_diagonals = defaultdict(bool)
dale_diagonals = defaultdict(bool)
queens = [-1] * n
```

附：提交结构

包含以下文件：

n_queens.py （源代码）

n_queens_output.txt （输出结果）

report.pdf（报告 PDF 格式）

demo（文件夹，存储了 N=8 和 N=12 的输出结果）