N 皇后问题实验报告

李亚飞 2023141461130

本实验要求使用基于回溯法的算法求解 N 皇后问题，我采用优化剪枝后的回溯法解决了这个问题，并完成本报告。本实验报告分为三个部分，即算法说明、优化思路、实验结果

一、算法说明

该代码的核心算法采用回溯法，而回溯法在本质上是一种深度优先（DFS）的算法。 也就是我们通过递归的方式穷举整个解空间，逐步构建可行解。回溯法一般来说包含这几步：构造解空间（通常通过构造树这样的数据结构实现）；然后从根节点出发，沿着树的一条路径递归地向下一层搜索；如果不进行优化，回溯算法将不进行冲突检测，继续一路向下直到发现是无效解为止；发现是无效解之后，回溯算法将回到上一层，重新向下搜索；回溯算法会将所有可能路径遍历完之后结束。

这是不优化的回溯算法的伪代码

```
def backtrack(row):
    if row == N: # 递归终止条件
        if is_valid_solution(board): # 延迟检测冲突
            add_to_result(board)
        return
    for col in 0..N-1: # 尝试所有可能的列
        board[row][col] = 'Q' # 做选择
        backtrack(row + 1) # 递归
        board[row][col] = '.' # 撤销选择（回溯）
```

可以看出这样的搜索策略下， 时间复杂度是 O(M)（M 为解空间总路径数， 而 M 可视为 N！）。当 N=8 时，路劲总数增加到了 40320 条，如果此时继续选择暴力穷举的话非常耗费计算时间。

二、优化思路

优化思路概述

针对 N 皇后问题的回溯解法，我提出了两个核心优化策略：利用集合快速检测冲突位置，以及通过对称性减少搜索空间。这些优化显著减少了无效路径的遍历，将时间复杂度优化至接近 O（$N \cdot 2^N$）。

优化策略详解

1. **冲突检测优化**

使用三个集合实时记录已被占用的列和对角线：

列冲突：cols 集合存储已被占用的列索引

正对角线冲突：pos_diag 集合存储 row + col 值（正斜线上的所有位置具有相同的行加列值）

反对角线冲突：neg_diag 集合存储 row - col 值（反对角线上的所有位置具有相同的行减列值）

通过这三个集合，在 O（1）时间内即可判断当前位置是否有效，避免了传统解法中逐行逐列检查的 O（N）时间复杂度。

2. **镜像对称优化**

利用棋盘对称性，仅搜索第一行的前半部分列：

当 N 为偶数：仅搜索前 N/2 列

当 N 为奇数：搜索前 (N+1)/2 列，覆盖中间列

对于每个找到的解，通过水平镜像生成对称解：

对于第一行皇后在中间位置的解（仅当 N 为奇数时存在），其镜像解与原解相同，需排除重复

其他解通过反转每行字符串生成镜像解，并检查是否与已有解重复

优化后的回溯算法实现

```
def backtrack(row):
    if row == n:
        # 找到有效解，转换为字符串格式并保存
        solution = [''.join(row) for row in board]
        result.append(solution)
        return

    # 确定当前行的列搜索范围，仅第一行应用镜像优化
    if row == 0 and n > 1:
        col_range = range((n + 1) // 2) if n % 2 == 1 else range(n // 2)
    else:
        col_range = range(n)

    for col in col_range:
        # 快速检测冲突
        if col in cols or (row + col) in pos_diag or (row - col) in neg_diag:
```

```
                continue

            # 放置皇后并更新冲突集合
            board[row][col] = 'Q'
            cols.add(col)
            pos_diag.add(row + col)
            neg_diag.add(row - col)

            # 递归处理下一行
            backtrack(row + 1)

            # 回溯撤销选择
            board[row][col] = '.'
            cols.remove(col)
            pos_diag.remove(row + col)
            neg_diag.remove(row - col)

        # 第一行处理完毕后，生成镜像解
        if row == 0 and n > 1 and result:
            original_count = len(result)
            for i in range(original_count):

                # 生成镜像解
                mirror_solution = [row_str[::-1] for row_str in result[i]]
                # 避免添加重复解：奇数 N 时，若第一行皇后在中间列，则镜像解与原解相同
                if n % 2 == 1 and result[i][0][(n-1)//2] == 'Q':
                    continue

                # 确保镜像解不重复
                if mirror_solution not in result:
                    result.append(mirror_solution)
```

优化效果分析

这两个优化策略协同工作：

1. **冲突检测集合**将每次放置皇后的冲突检测时间从 O(N) 降至 O(1)
2. **镜像对称优化**将搜索空间减少近一半，尤其在 N 较大时效果显著

综合这两个优化，算法的时间复杂度接近 $O(N \cdot 2^N)$，显著优于朴素回溯算法的 O(N!)。该优化在保持代码简洁性的同时，大幅提升了求解效率，特别是对于中等规模的 N（如 N=8 至 N=16）表现尤为出色。

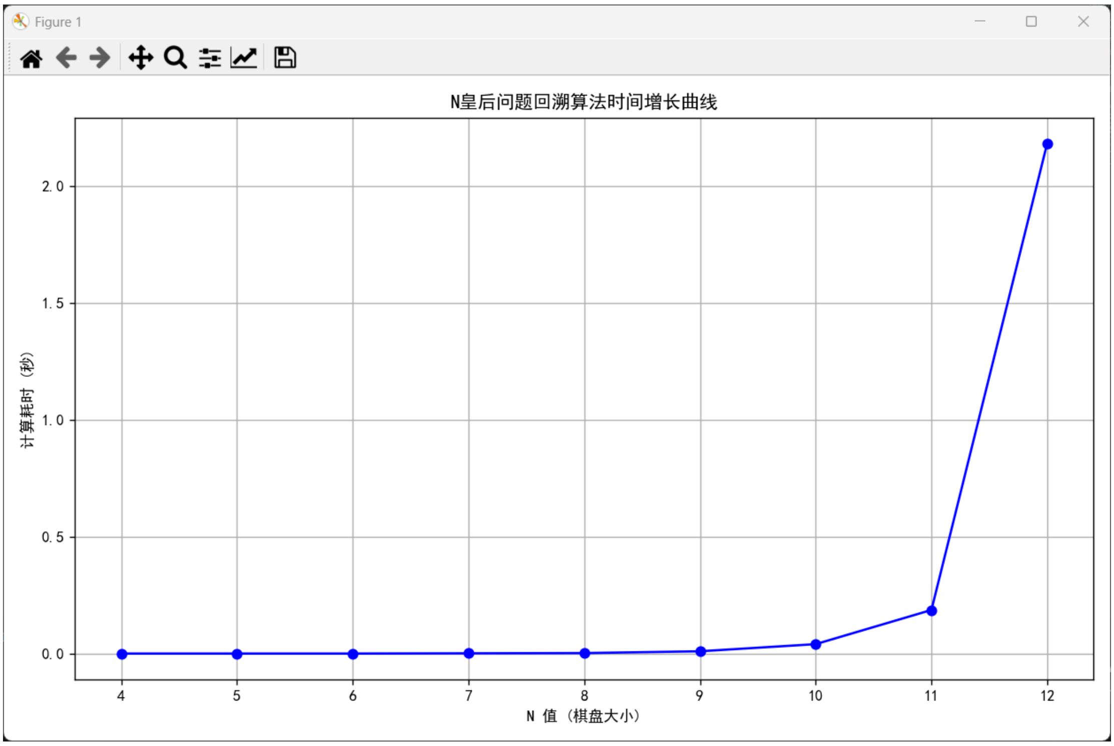
三、实验结果

N=4

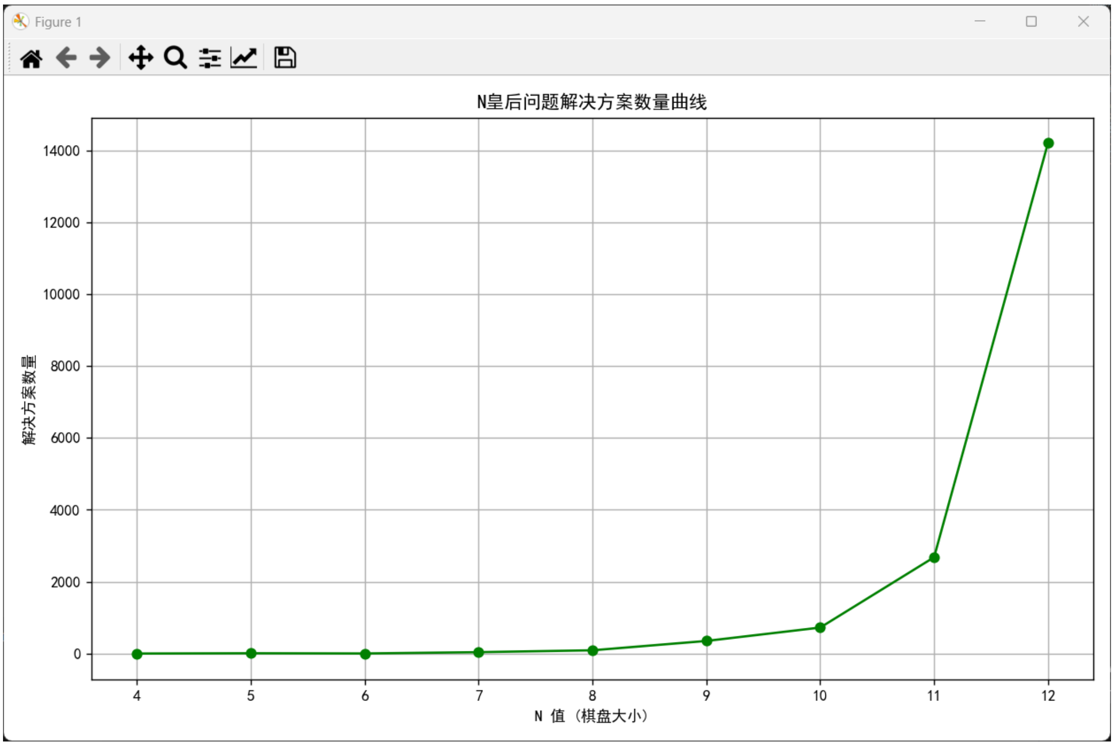
```
(base) PS C:\Users\11911> & E:/AIclass/python.exe "c:/Users/11911/Desktop/homework/01_2023141461130_Li Yafei/n_queens.py"
请输入N的值（棋盘大小和皇后数量，N>=4）: 4
N = 4 时共有 2 种解决方案。
计算耗时: 0.0000 秒

请选择输出方式:
1. 只显示一个解决方案
2. 显示所有解决方案
3. 显示指定数量的解决方案
4. 显示指定索引的解决方案
请输入选择 (1-4): 2

所有解决方案:
解决方案 1:
.Q..
...Q
Q...
..Q.

解决方案 2:
..Q.
Q...
...Q
.Q..
```

N=8

```
请输入N的值（棋盘大小和皇后数量，N>=4）: 8
N = 8 时共有 92 种解决方案。
计算耗时: 0.0020 秒

请选择输出方式:
1. 只显示一个解决方案
2. 显示所有解决方案
3. 显示指定数量的解决方案
4. 显示指定索引的解决方案
请输入选择 (1-4): 2

所有解决方案:
解决方案 1:
Q.......
....Q...
.......Q
.....Q..
..Q.....
......Q.
.Q......
...Q....

解决方案 2:
Q.......
.....Q..
.......Q
..Q.....
......Q.
...Q....
.Q......
....Q...

解决方案 3:
```



回溯算法时间增长曲线



解决方案数量曲线

可以看出，算法成功求出正确值，并且提供了异常检测，通过终端实现了一定的用户交互机制，用户可以自主选择显示解决方案的方式：显示全部、显示指

定数量、显示全部、显示指定索引等等。同样看出，算法的计算效率也处在一个较优的水平，N=4 时耗时在 10^-4s 以下，N=8 时耗时在 10^-3s 范围。