N 皇后问题实验报告

范盛颉　软件学院　2023141479277

一、算法说明

1. 基础回溯算法

本实验采用回溯法作为基础算法求解 N 皇后问题。回溯法通过逐行放置皇后，在每一行尝试所有可能的列位置，并通过 is_valid 函数检查是否与已放置的皇后冲突（同一列、左上对角线、右上对角线）。若合法则继续递归放置下一行，若到达最后一行则记录一个解。

冲突检测函数 is_valid(row, col)：

- 列冲突：检查当前列是否已有皇后（遍历已放置行，判断列值是否重复）。
- 左上对角线冲突：检查左上方向（行减 1，列减 1）是否有皇后。
- 右上对角线冲突：检查右上方向（行减 1，列加 1）是否有皇后。
- 时间复杂度：每次调用 is_valid 需遍历当前行以上的所有皇后，时间复杂度为 O(n)。

回溯核心函数 backtrack(row)：

- 若已放置 n 个皇后（到达第 n 行），记录当前解。
- 对当前行的每一列，若合法则递归放置下一行，否则回溯。
- 时间复杂度：最坏情况下，每一行有 n 种选择，总时间复杂度为 O(n!)（近似阶乘级增长）。

```python
def is_valid(self, row, col):  3 用法
    """
    检查在(row, col)位置放置皇后是否合法
    由于我们是按行放置皇后，所以只需要检查列冲突和对角线冲突
    """
    # 检查列冲突：当前列是否已有皇后
    for i in range(row):
        if self.board[i] == col:
            return False

    # 检查左上对角线：左上方向是否已有皇后
    for i, j in zip(range(row - 1, -1, -1), range(col - 1, -1, -1)):
        if self.board[i] == j:
            return False

    # 检查右上对角线：右上方向是否已有皇后
    for i, j in zip(range(row - 1, -1, -1), range(col + 1, self.n)):
        if self.board[i] == j:
            return False

    return True  # 如果没有冲突，则该位置合法
```

```python
def backtrack(self, row):  2 用法
    """
    回溯算法核心函数
    row: 当前要放置皇后的行
    """
    # 基本情况：如果已经放置了n个皇后（到达第n行），说明找到了一个解
    if row == self.n:
        self.solutions.append(self.board.copy())  # 保存当前解的副本
        return

    # 尝试在当前行的每一列放置皇后
    for col in range(self.n):
        if self.is_valid(row, col):  # 检查当前位置是否合法
            self.board[row] = col  # 在(row, col)位置放置皇后
            self.backtrack(row + 1)  # 递归求解下一行
            # 回溯：撤销选择，尝试下一列
            self.board[row] = -1
```

2. 对称性剪枝优化

核心思想：利用棋盘的左右对称性减少搜索空间，仅搜索第一行的前半列，通过对称变换生成另一半解。

实现细节：

- 当 n 为偶数时，第一行仅需搜索前 n/2 列，非中心列的解通过镜像变换（col → n-1-col）生成。
- 当 n 为奇数时，第一行搜索前 (n+1)/2 列，中心列的解无需镜像，非中心列的解通过镜像生成。

剪枝效果：将搜索空间减少约 1/2，时间复杂度降为 O(n!/2)。

关键函数 solve_optimized(n)：

处理对称解时，需区分奇偶棋盘，避免重复计算中心列解（奇数情况）。

```
def backtrack_optimized(self, row):  3 用法
    """
    使用剪枝策略的回溯算法核心函数
    利用棋盘的对称性进行优化，只考虑第一行的前半部分列
    """
    # 基本情况：找到一个解
    if row == self.n:
        self.solutions.append(self.board.copy())
        return

    # 优化：第一行只考虑前半部分列，利用对称性减少计算量
    if row == 0:
        end_col = self.n // 2 if self.n % 2 == 0 else self.n // 2 + 1
        for col in range(end_col):
            if self.is_valid(row, col):
                self.board[row] = col
                self.backtrack_optimized(row + 1)
                self.board[row] = -1
    else:
        # 非第一行，考虑所有列
        for col in range(self.n):
            if self.is_valid(row, col):
                self.board[row] = col
                self.backtrack_optimized(row + 1)
                self.board[row] = -1
```

```
def solve_optimized(self, n, find_all=True):  1 个用法
    """
    求解N皇后问题（使用剪枝策略）
    n: 棋盘大小
    find_all: 是否找出所有解，默认为True
    """
    self.n = n
    self.board = [-1] * n
    self.solutions = []
    self.start_time = time.time()

    self.backtrack_optimized(0)  # 使用优化的回溯算法

    # 处理对称解：根据第一行皇后的位置，生成对应的对称解
    if self.n % 2 == 0:
        # 偶数棋盘：对每个解生成其镜像解
        for solution in self.solutions.copy():
            symmetric_solution = [self.n - 1 - col for col in solution]
            self.solutions.append(symmetric_solution)
    else:
        # 奇数棋盘：需要特殊处理中心列的解
        center_col_solutions = [solution for solution in self.solutions if solution[0] == self.n // 2]
        non_center_col_solutions = [solution for solution in self.solutions if solution[0] != self.n // 2]

        # 非中心列的解需要生成镜像解
        for solution in non_center_col_solutions:
            symmetric_solution = [self.n - 1 - col for col in solution]
            self.solutions.append(symmetric_solution)

    elapsed_time = time.time() - self.start_time
    return elapsed_time
```

3. 启发式算法

核心思想：贪心策略减少搜索空间，优先选择冲突最少的列放置皇后。

- 初始化：随机在每一行放置皇后（可能冲突）。
- 迭代优化：对每一行皇后，计算各列冲突数（get_conflict_count），选择冲突最少的列移动，直至无冲突或达到最大迭代次数。

优缺点：

- 优点：在寻找单个解时可能快速收敛（尤其当初始解接近合法解时）。
- 缺点：计算冲突数需 O(n2) 时间，且可能陷入局部最优，导致总时间在 n 较大时高于回溯法。

```
def backtrack_with_heuristic(self, row):  2 用法
    """带启发式的回溯算法：优先选择冲突少的列"""
    if row == self.n:
        self.solutions.append(self.board.copy())
        return
    # 计算每列的冲突数，并按冲突数从小到大排序
    cols_with_conflict = [(col, self.get_conflict_count(row, col)) for col in range(self.n)]
    sorted_cols = sorted(cols_with_conflict, key=lambda x: x[1])
    for col, _ in sorted_cols:
        if self.is_valid(row, col):
            self.board[row] = col
            self.backtrack_with_heuristic(row + 1)
            self.board[row] = -1
```

```python
def get_conflict_count(self, row, col):  1个用法
    """计算当前列的冲突数（启发式评分）"""
    conflict = 0
    for i in range(row):
        if self.board[i] == col or abs(self.board[i] - col) == abs(i - row):
            conflict += 1
    return conflict
```

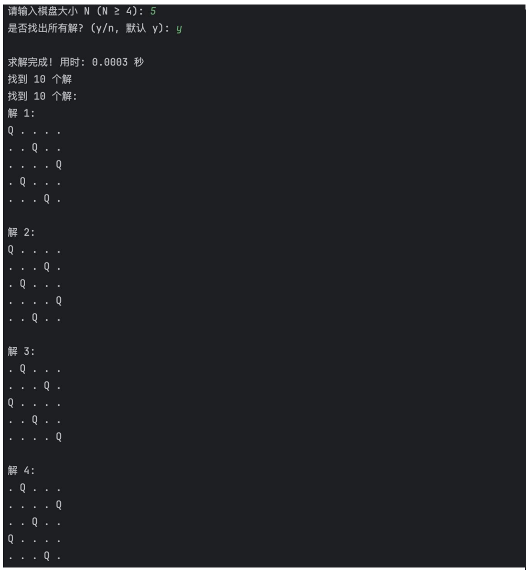
4. 算法复杂度总结

算法类型	时间复杂度（理论）	时间复杂度（实际）	空间复杂度
基础回溯算法	O(n!)	O(n!)（优化后降低）	O(n)（递归栈）
对称性剪枝	O(n!/2)	约 O(n!/2)	O(n)
启发式算法	依赖初始解和迭代次数	平均 O(n^2)（冲突计算主导）	O(n)

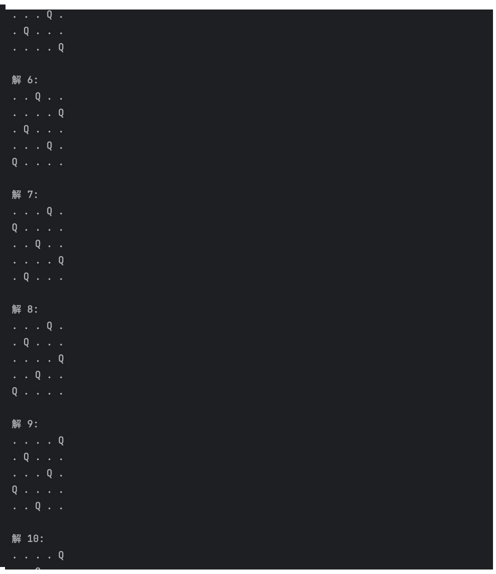
二、实验结果及分析

实验结果

输入 N = 5，是否输出所有解 y

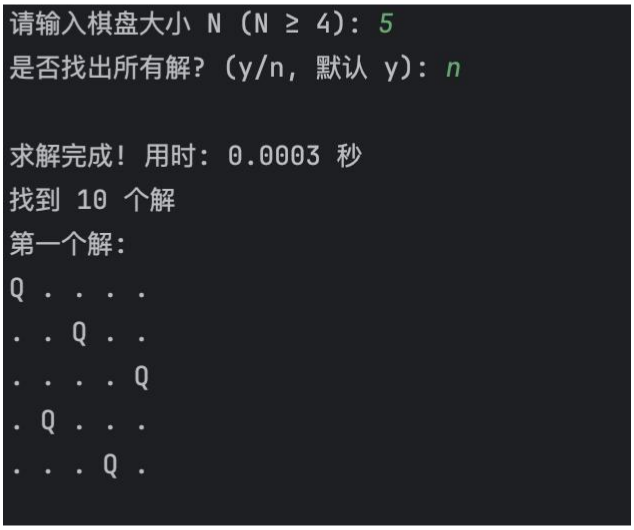


```
请输入棋盘大小 N (N ≥ 4): 5
是否找出所有解? (y/n, 默认 y): y

求解完成! 用时: 0.0003 秒
找到 10 个解
找到 10 个解:
解 1:
Q . . . .
. . Q . .
. . . . Q
. Q . . .
. . . Q .

解 2:
Q . . . .
. . . Q .
. Q . . .
. . . . Q
. . Q . .

解 3:
. Q . . .
. . . Q .
Q . . . .
. . Q . .
. . . . Q

解 4:
. Q . . .
. . . . Q
. . Q . .
Q . . . .
. . . Q .
```



```
. . . Q .
. Q . . .
. . . . Q

解 6:
. . Q . .
. . . . Q
. Q . . .
. . . Q .
Q . . . .

解 7:
. . . Q .
Q . . . .
. . Q . .
. . . . Q
. Q . . .

解 8:
. . . Q .
. Q . . .
. . . . Q
. . Q . .
Q . . . .

解 9:
. . . . Q
. Q . . .
. . . Q .
Q . . . .
. . Q . .

解 10:
. . . . Q
```

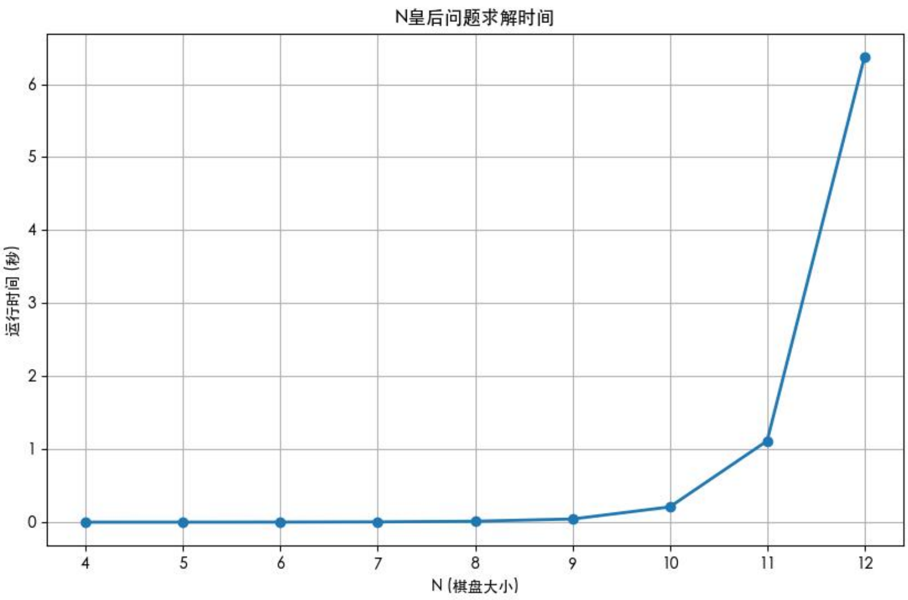
输出可视化的结果以及结果个数
是否输出所有解 n

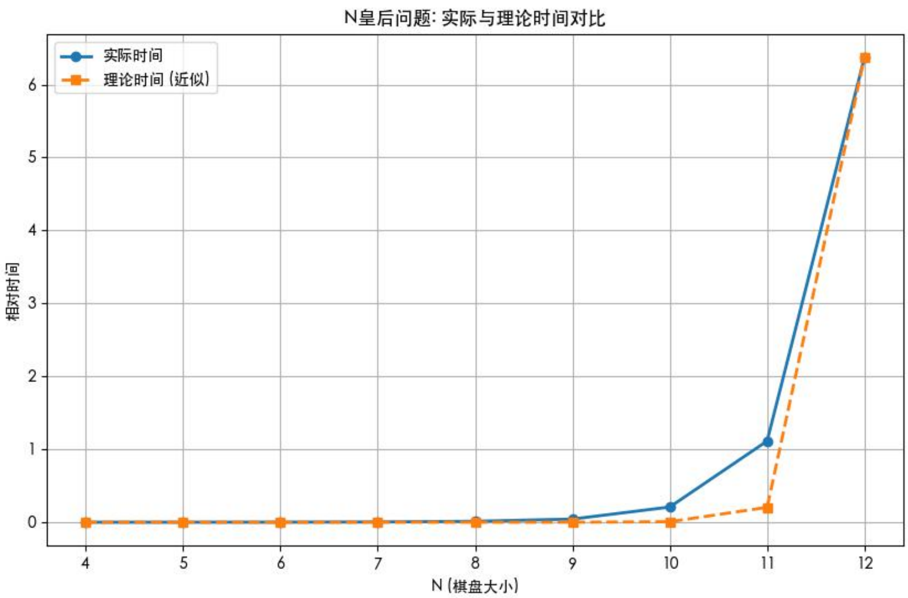


```
请输入棋盘大小 N (N ≥ 4): 5
是否找出所有解? (y/n, 默认 y): n

求解完成! 用时: 0.0003 秒
找到 10 个解
第一个解:
Q . . . .
. . Q . .
. . . . Q
. Q . . .
. . . Q .
```

实验分析

记录 N=4 至 N=12 时的运行时间，绘制时间增长曲线。
4–12 求解时间表，可以注意到N=10，11 时有明显的增长

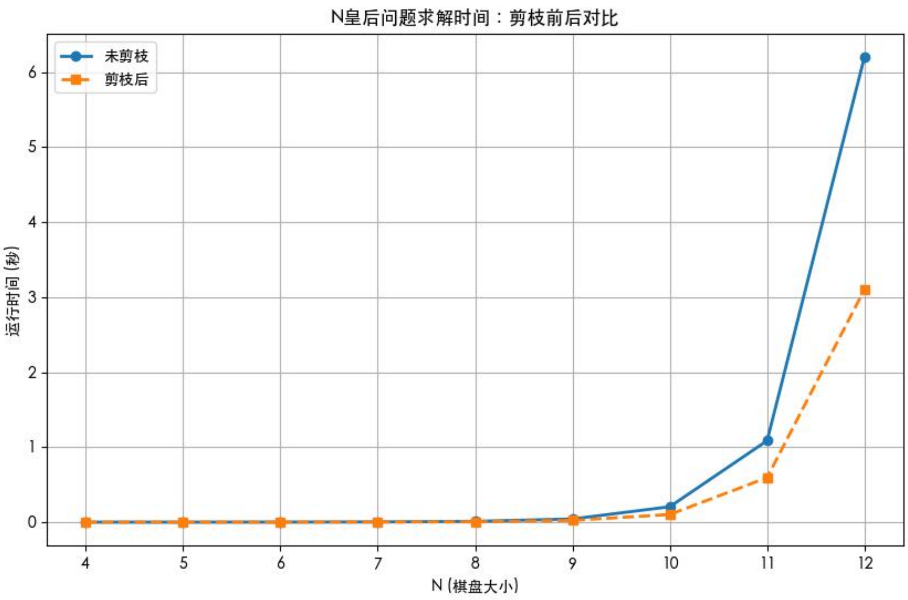




可以看到在实际运行程序时，n<=9 时，实际与理论时间几乎相同，而 n=10，11 时，实际时间会略高于理论时间

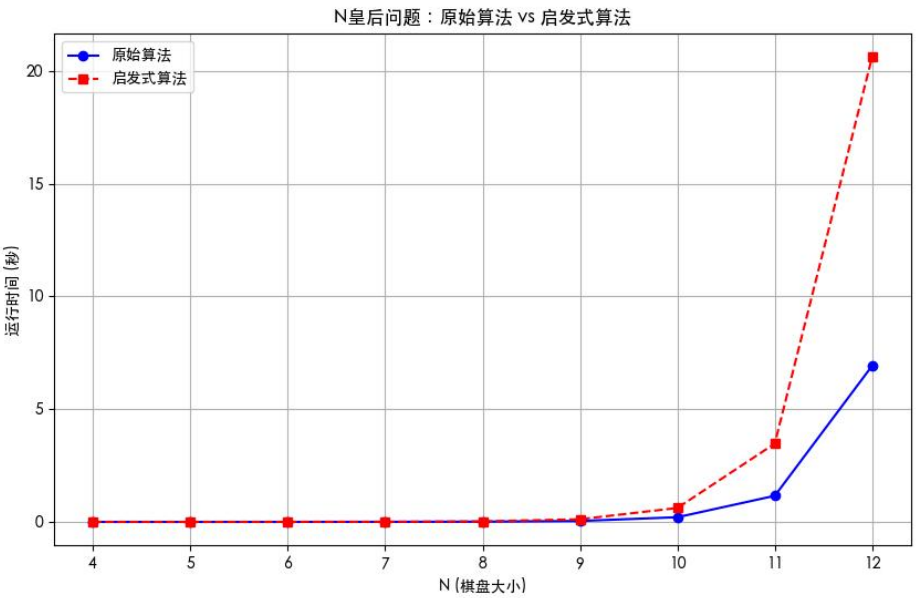
三、优化思路

回溯算法——剪枝优化



剪枝后时间显著减少，尤其当 n>9 时，时间差距随 n 增大呈指数级扩大，验证了对称性剪枝对阶乘级复杂度的优化效果。

启发式搜索的可行性分析



当 n≤9 时，启发式算法因减少无效搜索而略快；当 n>9 时，冲突计算的 O(n^2) 开销主导时间，导致启发式算法慢于剪枝回溯法。

其他算法优化可能性

1. 位运算优化:
使用位掩码（Bitmask）表示列、对角线冲突，将冲突检测从 O(n) 降至 O(1)（通过位运算快速判断）。

2. 并行计算:
将不同列选择分支分配到多核处理器并行搜索，适用于求解所有解的场景。

3. 动态数据结构优化:
使用哈希表或集合实时记录冲突列和对角线，提高查询效率。
例如，用集合存储已占用的列、左对角线（row−col）、右对角线（row+col），冲突检测时间降为 O(1)。

5. 遗传算法：通过种群进化寻找解，但需设计合适的交叉/变异算子，可能收敛速度较慢。

6. 启发式算法改进建议
动态调整步长：在迭代中根据冲突数动态调整移动策略。
混合策略：结合回溯法的确定性搜索与启发式的随机优化，例如在回溯中引入冲突检测提前剪枝。