2025-01 N 皇后问题简短报告

算法说明

1. 回溯法（Backtracking）

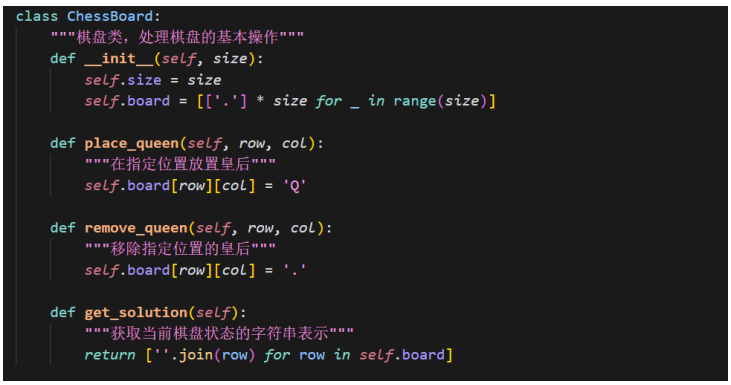
本项目采用回溯法求解 N 皇后问题。回溯法是一种通过递归方式尝试所有可能解的算法思路。具体来说，我们通过逐行放置皇后的方式，在每一列尝试放置一个皇后，并判断是否符合约束条件。

具体实现中，`solve_n_queens(n, find_all=True)` 函数是主入口。

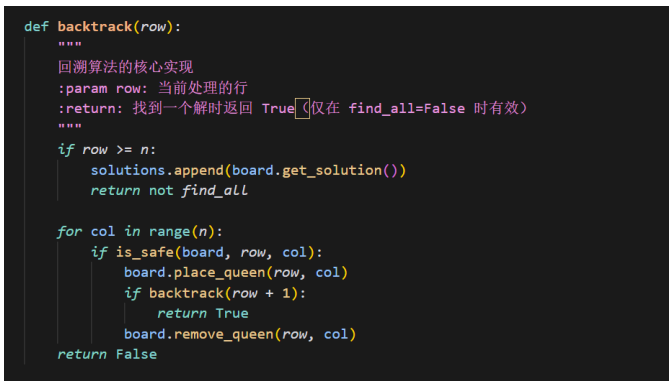
函数内部创建了一个 `ChessBoard` 对象，它封装了棋盘状态的维护（放置皇后、移除皇后、获取当前棋盘）。

核心的递归函数 `backtrack(row)` 按行尝试放置皇后：

- 若当前行数等于 n，表示所有皇后已成功放置，当前棋盘即为一个合法解，加入解集合；
- 否则，依次在每一列尝试放置皇后，若 `is_safe()` 判断该列安全，则递归继续下一行；若递归返回失败，则撤销当前步骤（回溯）。



```python
class ChessBoard:
    """棋盘类，处理棋盘的基本操作"""
    def __init__(self, size):
        self.size = size
        self.board = [['.'] * size for _ in range(size)]

    def place_queen(self, row, col):
        """在指定位置放置皇后"""
        self.board[row][col] = 'Q'

    def remove_queen(self, row, col):
        """移除指定位置的皇后"""
        self.board[row][col] = '.'

    def get_solution(self):
        """获取当前棋盘状态的字符串表示"""
        return [''.join(row) for row in self.board]
```



```python
def backtrack(row):
    """
    回溯算法的核心实现
    :param row: 当前处理的行
    :return: 找到一个解时返回 True（仅在 find_all=False 时有效）
    """
    if row >= n:
        solutions.append(board.get_solution())
        return not find_all

    for col in range(n):
        if is_safe(board, row, col):
            board.place_queen(row, col)
            if backtrack(row + 1):
                return True
            board.remove_queen(row, col)
    return False
```

2. 优化剪枝策略

在每次放置前调用 `is_safe(board, row, col)` 进行安全性判断。其检查逻辑如下：

- 检查当前列是否已有皇后：遍历当前行之前的每一行，查看对应列是否已有 'Q'；
- 检查左上至右下对角线是否冲突；
- 检查右上至左下对角线是否冲突。

```python
def is_safe(board, row, col):
    """
    检查在指定位置放置皇后是否安全
    :param board: 棋盘对象
    :param row: 当前行
    :param col: 当前列
    :return: 布尔值，表示是否可以放置
    """
    # 检查同列
    for i in range(row):
        if board.board[i][col] == 'Q':
            return False

    # 检查左上到右下对角线
    for i, j in zip(range(row-1, -1, -1), range(col-1, -1, -1)):
        if board.board[i][j] == 'Q':
            return False

    # 检查右上到左下对角线
    for i, j in zip(range(row-1, -1, -1), range(col+1, board.size)):
        if board.board[i][j] == 'Q':
            return False

    return True
```

这些判断在运行时大大缩小了搜索空间，有效避免明显冲突的递归路径。在 `find_all=False` 模式下，一旦找到一个合法解即返回，进一步加快执行速度。回溯函数还利用 `place_queen` 和 `remove_queen` 方法动态维护棋盘状态，确保状态干净，避免副作用。

3. 优化冲突检测——位运算优化

在进一步优化中，引入了 **位运算** 来表示列、主对角线、副对角线的占用状态，从而将原本 O(n) 复杂度的冲突判断优化为 O(1)，极大地提高了程序运行效率。实现中使用三个整数变量 `cols`、`diag1`、`diag2` 表示这三类约束：

代码块

```
self.cols |= (1 << col)
self.diag1 |= (1 << (row - col + self.size - 1))
self.diag2 |= (1 << (row + col))
```

判断某个位置是否安全可简化为：

代码块

```
return not (
    self.cols & (1 << col) or
    self.diag1 & (1 << (row - col + self.size - 1)) or
    self.diag2 & (1 << (row + col))
)
```

这种方法不仅执行速度更快，而且避免了大量重复判断和数组访问，是本实验中性能提升的关键。

位运算方法的优势/代码性能会有显著提升的原因：从O(n)的遍历操作优化为O(1)的位运算操作

a. 空间效率：一个整数就能表示一整行/列/对角线的占用情况

b. 时间效率：位运算操作（&, |, ~）是CPU的基本操作，执行速度极快

c. 代码简洁：不需要多重循环来检查冲突

d. 易于维护：状态的更新和检查都是原子操作

实验结果

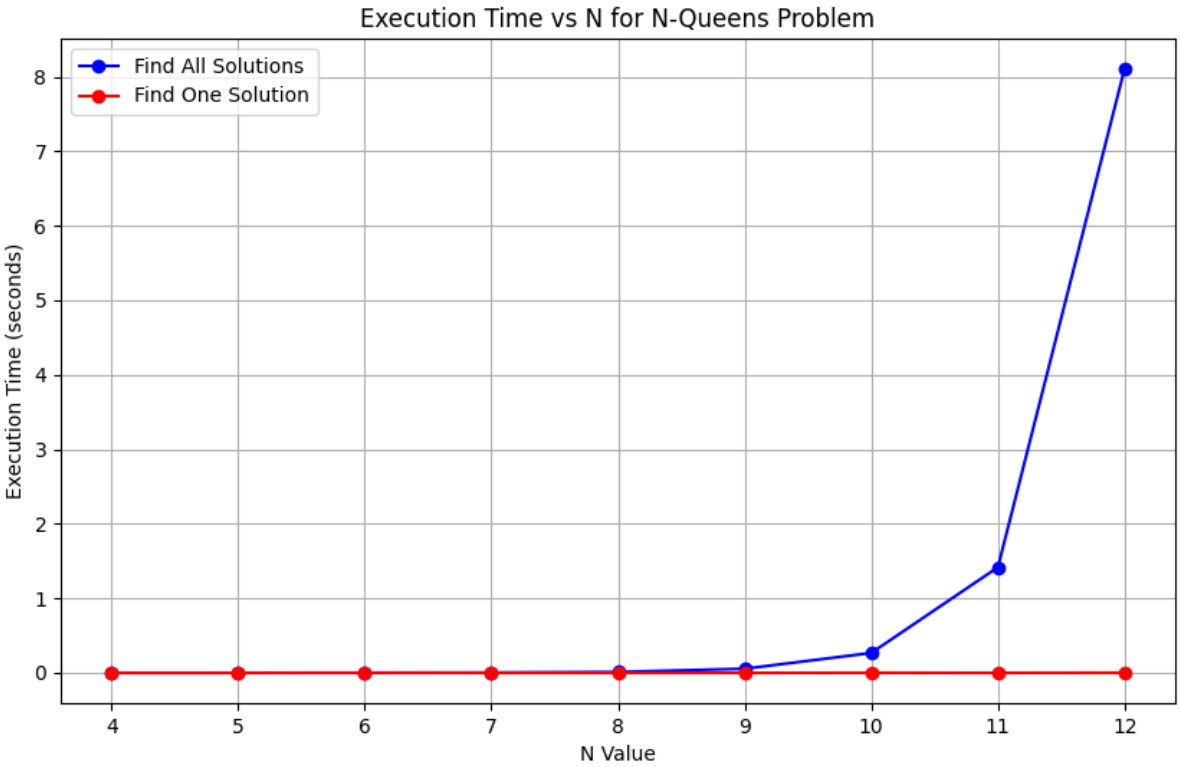
实验设置与数据 实验使用 `n_queens_experiment.py` 脚本，通过 `measure_time()` 函数测量从 N=4 到 N=12 的运行时间。分别对查找所有解（`find_all=True`）与查找一个解（`find_all=False`）两种模式进行测试，并使用 `matplotlib` 绘制时间随 N 增长的趋势图。

1. 记录 N=4 至 N=12 时的运行时间，绘制时间增长曲线

实验截图：

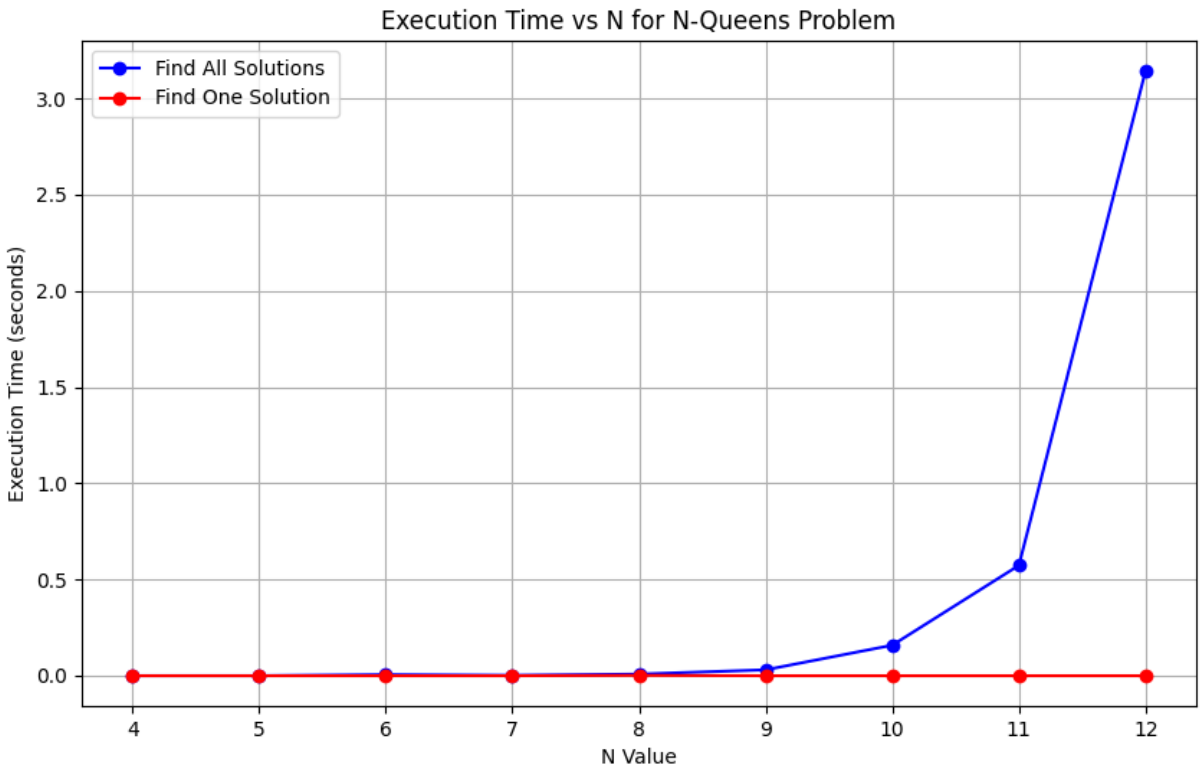
```
PS D:\nqueens> python n_queens_experiment.py
N 皇后问题性能分析实验
测量 N=4 至 N=12 的运行时间

开始实验:
================================================
  N   |    查找所有解(秒)     |    查找一个解(秒)
------------------------------------------------
  4   |    0.000000     |    0.000000
  5   |    0.000000     |    0.000000
  6   |    0.001269     |    0.000000
  7   |    0.003156     |    0.000000
  8   |    0.011397     |    0.000000
  9   |    0.055244     |    0.000000
 10   |    0.266981     |    0.001107
 11   |    1.417057     |    0.000000
 12   |    8.107342     |    0.002220
================================================

实验结果图表已保存为 'n_queens_time_analysis.png'
PS D:\nqueens>
```

时间增长曲线（引入位运算前）：

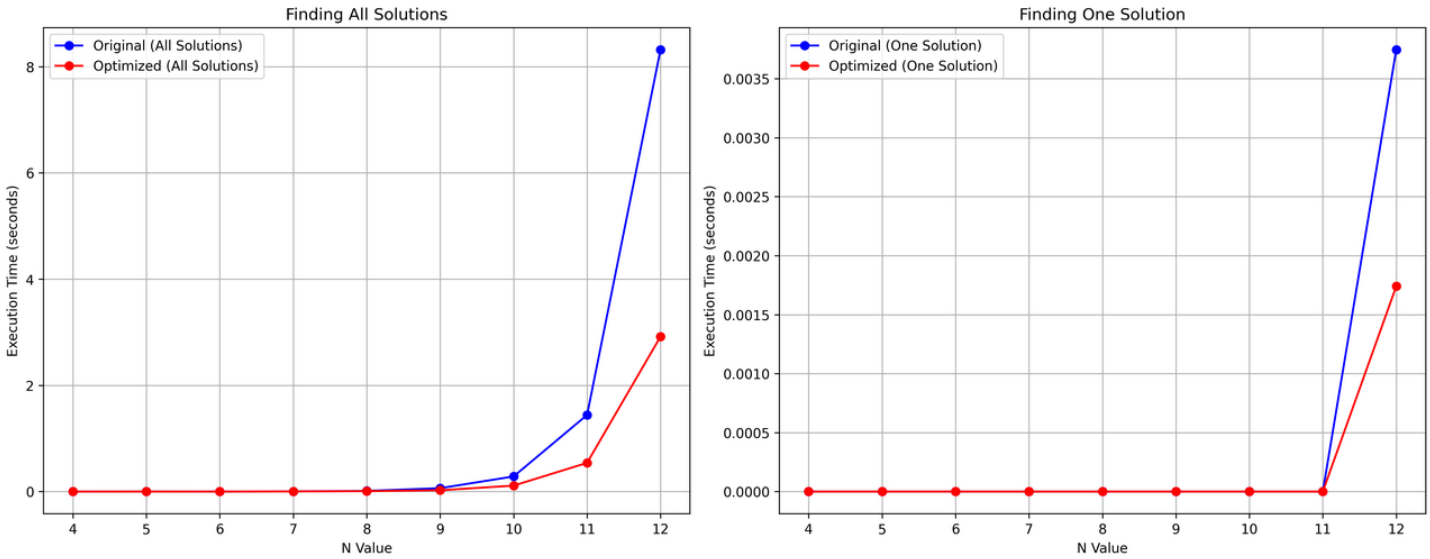


从结果可见，查找所有解的时间随 N 指数增长，而查找一个解几乎保持恒定，说明剪枝策略效果显著。

时间增长曲线（引入位运算后）：



对比实验：



从运行结果可以看到，优化后的代码运行时间明显减少。对比初始版本的实验数据，可明显看出执行时间有数量级的下降，尤其在 N=12 时由原来的 8.1 秒下降到约 3.0 秒，提升非常显著。

在检查位置安全性时，不再需要遍历，而是通过简单的位运算就能立即得到结果。引入位运算不仅提高了性能，也使得代码更加优雅和高效。

2. 分析算法的时间复杂度，并与理论值对比

时间复杂度分析 N 皇后问题的理论解空间规模约为 O(N!)，属于指数级复杂度问题。在传统实现中，每次判断位置是否安全都需要 O(N) 时间，导致总体回溯过程代价较高。

通过引入位运算优化，冲突检测从 O(N) 降为 O(1)，虽然整体问题仍是指数级（回溯搜索的路径不变），但每一步的计算开销显著降低。因此，优化后的时间复杂度可以描述为 O(N!) 的基础上乘以一个较小的常数因子。

实验数据也证实了这一点：

- 在 N=12 时，查找所有解时间从 8 秒下降到 3 秒，约提升 2.5 倍；
- 查找一个解由于回溯路径极短，优化前后均为 0 秒（精度范围内）。

可见位运算是一种有效的剪枝与加速策略，适用于需要频繁判断位置合法性的组合优化问题。

优化思路

1. 模块化设计

本项目采用模块化设计思路，使代码结构清晰、易于维护和拓展。

1.1 分离输入处理

`get_valid_n()` 和 `get_solution_mode()` 函数负责用户输入的合法性检查，确保输入为有效整数且 N>=4，提高了系统鲁棒性。 `print_solutions()` 负责打印解决方案。

输入处理：

```python
def get_valid_n():
    """
    获取有效的棋盘大小输入
    :return: 有效的棋盘大小 N
    """
    while True:
        try:
            n = int(input("请输入棋盘大小 N（N >= 4）: "))
            if n < 4:
                print("错误: N 必须大于或等于 4, 因为 N < 4 时问题无解")
                continue
            return n
        except ValueError:
            print("错误: 请输入一个有效的整数")
```

```python
def get_solution_mode():
    """
    获取用户选择的求解模式
    :return: 是否需要所有解
    """
    while True:
        choice = input("\n请选择求解模式: \n1. 只需要一个解\n2. 需要所有解\n请输入选项（1 或 2）: ").strip()
        if choice == "1":
            return False
        elif choice == "2":
            return True
        else:
            print("无效的选择，请输入 1 或 2")
```

输出处理：

```python
def print_solutions(solutions, n, find_all):
    """
    打印解决方案
    :param solutions: 解决方案列表
    :param n: 棋盘大小
    :param find_all: 是否是查找所有解模式
    """
    if not solutions:
        print(f"未找到 {n} 皇后问题的解决方案")
    else:
        if find_all:
            print(f"\n找到 {len(solutions)} 个解决方案: ")
            for i, solution in enumerate(solutions, 1):
                print(f"\n解决方案 #{i}:")
                print('\n'.join(solution))
                print("-" * (n * 2))
        else:
            print("\n找到一个解决方案: ")
            print('\n'.join(solutions[0]))
```

1.2 冲突检测

代码中的冲突检测是通过 `is_safe()` 函数实现。

原先无位运算时，冲突检测的三个关键部分：

1. 同列检测：检查当前列上方是否已有皇后；只需检查 [0, row-1] 范围，因为下方还没有放置皇后；不需要检查同行，因为每行只会放置一个皇后。
2. 左上对角线检测：使用 zip 同时递减行和列坐标；检查左上方的所有位置；对角线上行列坐标差值相等
3. 右上对角线检测：行坐标递减，列坐标递增；检查右上方的所有位置；对角线上行列坐标和相等。

该方法使用 Python 的 range 和 zip 使代码更简洁，分三个部分清晰地处理三个方向的检查，这种实现方式虽然不是最优的（时间复杂度为 O(n)），但是代码清晰易懂。

```python
def is_safe(board, row, col):
    """
    检查在指定位置放置皇后是否安全
    :param board: 棋盘对象
    :param row: 当前行
    :param col: 当前列
    :return: 布尔值，表示是否可以放置
    """
    # 检查同列
    for i in range(row):
        if board.board[i][col] == 'Q':
            return False

    # 检查左上到右下对角线
    for i, j in zip(range(row-1, -1, -1), range(col-1, -1, -1)):
        if board.board[i][j] == 'Q':
            return False

    # 检查右上到左下对角线
    for i, j in zip(range(row-1, -1, -1), range(col+1, board.size)):
        if board.board[i][j] == 'Q':
            return False

    return True
```

于是项目改进了原有的冲突检测方法，引入了位运算优化方法。

位运算方法的优势/代码性能会有显著提升的原因：从O(n)的遍历操作优化为O(1)的位运算操作

a. 状态表示

```python
class ChessBoard:
    """棋盘类，处理棋盘的基本操作"""
    def __init__(self, size):
        self.size = size
        self.board = [['.'] * size for _ in range(size)]
        # 使用位运算记录占用情况
        self.cols = 0  # 列占用情况
        self.diag1 = 0  # 主对角线占用情况（左上到右下）
        self.diag2 = 0  # 副对角线占用情况（右上到左下）
```

b. 放置和移除皇后的位运算

```python
def place_queen(self, row, col):
    """在指定位置放置皇后"""
    self.board[row][col] = 'Q'
    self.cols |= (1 << col)
    self.diag1 |= (1 << (row - col + self.size - 1))
    self.diag2 |= (1 << (row + col))

def remove_queen(self, row, col):
    """移除指定位置的皇后"""
    self.board[row][col] = '.'
    self.cols &= ~(1 << col)
    self.diag1 &= ~(1 << (row - col + self.size - 1))
    self.diag2 &= ~(1 << (row + col))
```

c. 冲突检测

```python
def is_safe(self, row, col):
    """检查在指定位置放置皇后是否安全"""
    return not (
        self.cols & (1 << col) or
        self.diag1 & (1 << (row - col + self.size - 1)) or
        self.diag2 & (1 << (row + col))
    )
```

2. 处理非法输入与异常处理

在 `main()` 函数中加入 try-except 结构，对用户中断、类型错误、值错误等常见异常进行处理。`validate_input()` 函数对输入参数进行类型和值验证，防止程序运行时崩溃。

```python
def validate_input(n):
    """
    验证输入参数的有效性
    :param n: 棋盘大小
    :raises: TypeError 如果n不是整数
    :raises: ValueError 如果n小于4
    """
    if not isinstance(n, int):
        raise TypeError("棋盘大小必须是整数")
    if n < 4:
        raise ValueError("N 皇后问题在 N < 4 时无解")
```

```python
def main():
    try:
        n = get_valid_n()
        find_all = get_solution_mode()

        print(f"\n求解 {n} 皇后问题..." + ("（查找所有解）" if find_all else "（查找第一个解）"))
        solutions = solve_n_queens(n, find_all)
        print_solutions(solutions, n, find_all)

    except (ValueError, TypeError) as e:
        print(f"错误：{str(e)}")
    except KeyboardInterrupt:
        print("\n程序已被用户中断")
    except Exception as e:
        print(f"发生未预期的错误：{str(e)}")
```

3. 用户简单交互

```
请输入棋盘大小 N（N >= 4）：4

请选择求解模式：
1. 只需要一个解
2. 需要所有解
请输入选项（1 或 2）：2
```

部分测试用例截图

```
PS D:\nqueens> python n_queens.py
请输入棋盘大小 N（N >= 4）：4

请选择求解模式：
1. 只需要一个解
2. 需要所有解
请输入选项（1 或 2）：2

求解 4 皇后问题...（查找所有解）

找到 2 个解决方案：

解决方案 #1:
.Q..
...Q
Q...
..Q.
--------

解决方案 #2:
..Q.
Q...
...Q
.Q..
--------
```