N 皇后实验报告

> **Note**
>
> - **作者：王文杰**
> - **学号：2022141090107**

1. 问题描述

N 皇后问题是在一个 N×N 的国际象棋棋盘上放置 N 个皇后，使得任意两个皇后不在同一行、同一列，也不在同一条对角线上。

由于每个皇后必须占据不同的行和列，我们可以将问题简化为：在 N 行中，为每一行选择一个唯一的列来放置皇后，同时满足对角线约束。

- 实现求解算法：基于提供的代码，实现一个能够找到 N 皇后问题所有解或单个解的程序。
- 输出解：将找到的每一个解以直观的棋盘形式打印出来。
- 算法优化与对比：实现两种核心算法——常规回溯法和位运算优化法，并对它们的运行效率进行量化分析和对比。

2. 算法说明

代码中提供了两种解决 N 皇后问题的核心算法：经典的递归回溯法和利用位运算进行高效优化的方法。

2.1. 常规回溯算法

回溯法是一种通过探索所有可能的候选解来找出所有解的算法。如果发现候选解不可能是最终解，则回溯并尝试其他选择。

核心思想：
算法逐行放置皇后，从第 0 行开始。在每一行，它会尝试将皇后放置在所有可能的列上（从 0 到 N-1）。

1. **放置与检查**：对于当前行 `row`，遍历每一列 `col`。在尝试放置皇后于 `(row, col)` 之前，需要检查该位置是否安全。
2. **安全性检查**：`is_safe` 函数负责此项工作。由于我们是按行递增放置的，所以只需检查当前位置是否与 **之前所有行** 的皇后冲突。
    - **列冲突**：检查 `c == col`，其中 `c` 是之前某一行皇后的列位置。
    - **对角线冲突**：检查 `abs(c - col) == abs(r - row)`。如果两个皇后行号之差的绝对值等于列号之差的绝对值，则它们在同一条对角线上。

```python
def is_safe(queens, row, col):
    # 检查在该位置放置皇后是否安全
    for r in range(row):
        c = queens[r]
        # 检查列冲突 (c == col) 和对角线冲突 (abs(c-col) == abs(r-row))
        if c == col or abs(c - col) == abs(r - row):
            return False
    return True
```

3. **递归深入**：如果位置 `(row, col)` 安全，则将皇后放置在此处 (`queens[row] = col`)，然后递归地去解决下一行 `row + 1`。
4. **回溯**：如果从下一行的递归调用返回后，没有找到完整的解，程序会继续 `for` 循环，尝试当前行的下一个可用列。
5. **找到解**：当 `row == n` 时，意味着我们成功地在所有 N 行都放置了皇后，此时就找到了一个完整的解。

2.2. 位运算优化算法 (Bitwise Optimization)

常规回溯法中的 `is_safe` 函数包含一个循环，在棋盘较大时效率较低。位运算优化通过使用整数的位来表示棋盘的占用状态，将循环检查转换为几次位运算，极大地提升了速度。

核心思想：
使用三个整数作为位掩码来分别记录 **列**、**左对角线** 和 **右对角线** 的被攻击情况。

- `cols`：一个整数，如果其第 `i` 位为 `1`，表示第 `i` 列已被占用。
- `ld`：一个整数，表示所有被攻击的 **左对角线**。
- `rd`：一个整数，表示所有被攻击的 **右对角线**。

算法流程：

1. **计算可用位置**：在处理当前行 `row` 时，首先计算出所有可以放置皇后的列。
    - `cols | ld | rd`：将所有被占用的列和对角线合并，其中为 `1` 的位都不能放置皇后。
    - `~(cols | ld | rd)`：对攻击位图取反，现在为 `1` 的位代表 **安全** 的位置。
    - `& ((1 << n) - 1)`：`(1 << n) - 1` 会生成一个长度为 `n` 的全 `1` 掩码，屏蔽掉更高位的影响，确保只在 NxN 的棋盘内操作。

```
# 所有可能放置皇后的位置 (值为1的位表示可放置)
available = ~(cols | ld | rd) & ((1 << n) - 1)
```

2. **选择并放置**：`while (available)` 循环遍历所有可用的位置。
    - `pos = available & -available`：通过效率更高的位运算，用于分离出 `available` 中最低位的 `1`。
    - `col = bin(pos).count('0') - 1`：将 `pos` 转换为二进制后，通过计算前导零的数量得到其对应的列索引。
    - 将该列存入 `queens` 数组。
3. **递归与状态更新**：递归调用 `solve_bits` 进入下一行，并更新三个状态位掩码。
    - `cols | pos`：将当前选择的列标记为占用。
    - `(ld | pos) << 1`：更新左对角线。将当前位置加入占用，并整体左移一位。
    - `(rd | pos) >> 1`：更新右对角线。将当前位置加入占用，并整体右移一位。
4. **回溯**：
    - `available &= ~pos`：在递归返回后，从 `available` 中移除刚刚尝试过的位置，以便 `while` 循环可以处理下一个可用的位置。

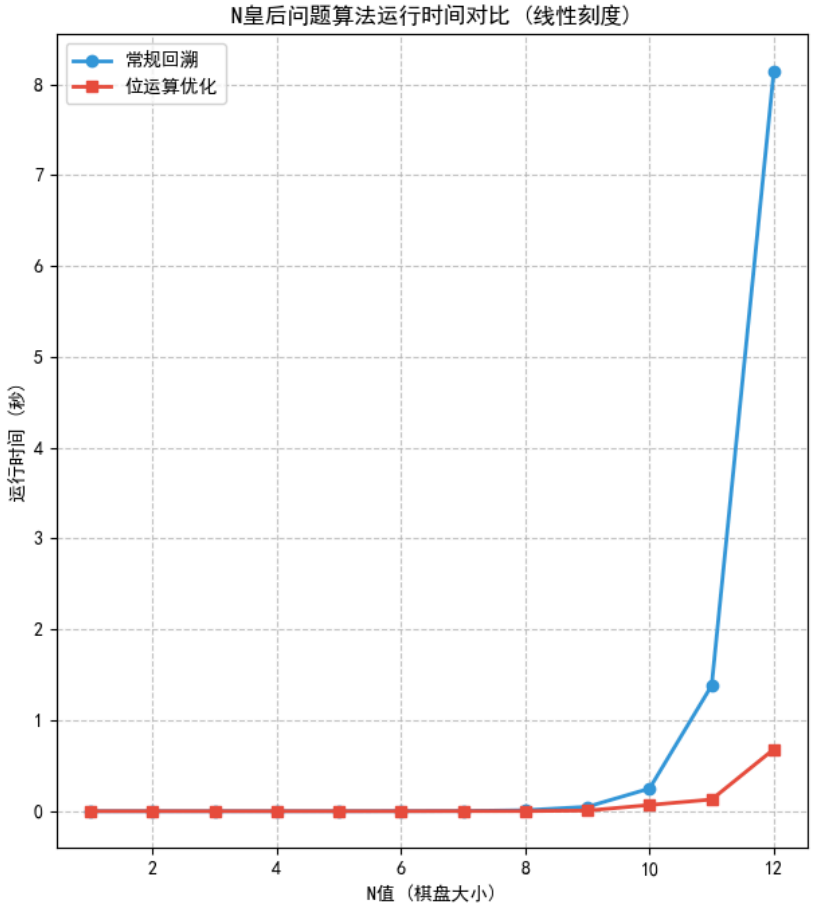
```
# 递归调用并更新状态
solve_bits(row + 1,
           cols | pos,
           (ld | pos) << 1,
           (rd | pos) >> 1)
```

3. 实验结果

3.1 实际运行情况

实验记录了常规回溯和位运算优化两种方法从 `n = 1` 到 `n = 12` 的运行时间，并绘制图像如下所示。本图展示了两种不同算法在解决 N 皇后问题时的运行时间对比。横轴为棋盘大小 $N \times N$，范围为 1 到 12；纵轴为对应运行时间（单位：秒），采用线性刻度。图中包含两条曲线，分别表示：

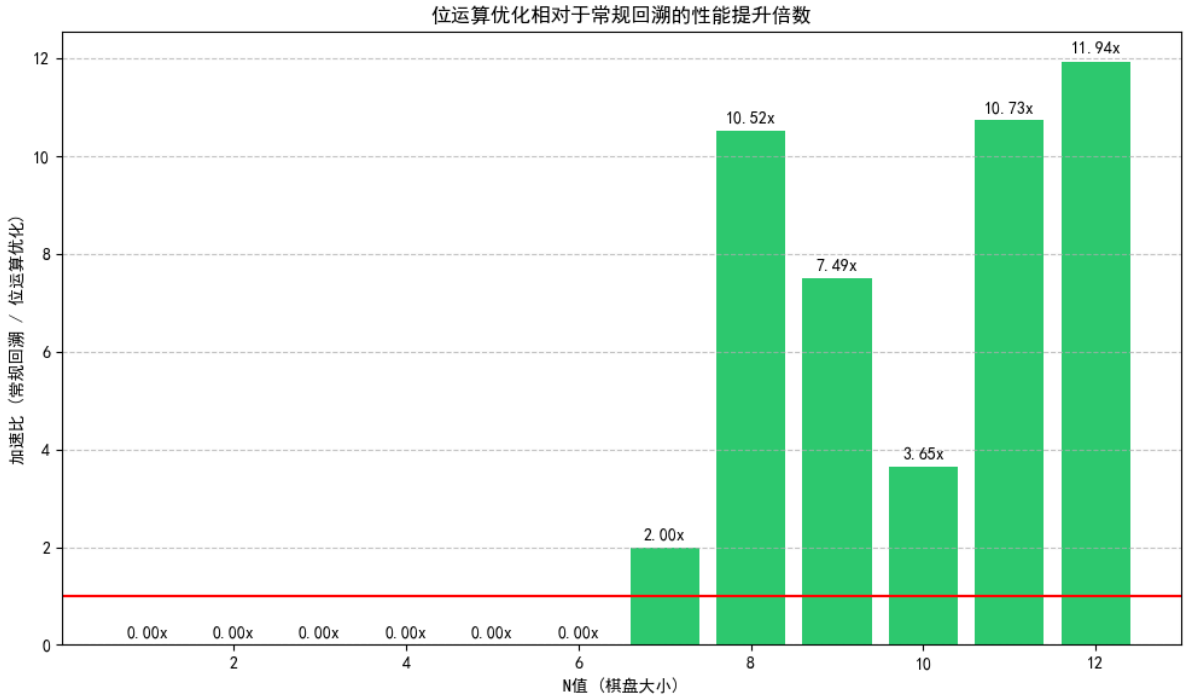
- **蓝色曲线**：常规回溯法（未优化）
- **红色曲线**：使用位运算的优化算法



两种算法详细分析对比如下所示：

比较维度	常规回溯法	位运算优化法
算法复杂度	指数级 $O(N!)$	接近线性或多项式
性能趋势	快速恶化	稳定可控
扩展能力	差，超过 $N = 10$ 后瓶颈明显	强，可支持更大规模
实际耗时表现	$N = 12$ 耗时 > 8 秒	$N = 12$ 耗时 < 1 秒

除此以外还会之了位运算优化相对于常规回溯的性能提升倍数对比图：



这张图展示了两种算法在解决 N 皇后问题时随棋盘大小 N 变化的运行时间差异，直观反映了优化对性能的显著影响。横轴为 N 值（即棋盘大小或皇后数量），纵轴为运行时间（单位为秒），采用线性刻度，便于对比算法在不同规模下的绝对运行耗时。

图中蓝线代表传统的回溯算法，其运行时间在 $N \leq 8$ 时基本可以接受，但从 $N = 9$ 起呈现快速增长趋势，尤其在 $N = 12$ 时达到约 8 秒，几乎呈指数级上升。这种增长趋势符合传统回溯在解空间为 $N!$ 时的复杂度特征。相比之下，红线代表经过位运算优化后的算法，在整个测试区间内运行时间始终维持在极低水平，即使在 $N = 12$ 时也仅为不到 1 秒，且增长趋势相对平缓，显示出更优的时间复杂度和更强的可扩展性。

两条曲线的走势差异清晰地说明：位运算优化在剪枝效率、状态表示与递归深度控制等方面极大地提升了性能。这种优化不仅减少了不必要的分支搜索，还有效压缩了状态空间，适合用于更大规模的 N 皇后问题求解。整体来看，该图不仅验证了算法复杂度的理论差异，也提供了明确的实践依据，强调了在处理指数型搜索问题时使用高效状态编码的重要性。

3.2 理论与现实复杂度情况对比

3.2.1 理论时间复杂度分析

1. 标准回溯算法

- **复杂度上界**: 算法的搜索空间可以被视为一个树状结构。在第 `r` 行，算法会尝试 `N` 个不同的列。由于总共有 `N` 行，一个非常宽松的复杂度上界是 $O(N^N)$。一个更精确、更常被引用的上界是 $(O(N!)$。

2. 位运算优化算法 (`solve_bits`)

- **渐进复杂度**: 该算法的递归结构和它所探索的搜索空间与标准回溯法完全相同。因此，其渐进时间复杂度依然是 $O(N!)$。
- **安全检查的成本**: 这正是优化的关键所在。位运算版本不再需要一个 $O(\text{row})$ 的循环，而是通过几次**常量时间**的位运算（`|`, `&`, `<<`, `>>`）来完成安全检查。这使得在每个节点上完成核心工作的成本降至 $O(1)$。

3.2.2 实证性能对比

从上述图像来看，随着 `N` 的增长，两种算法的运行时间都呈指数级增长，这与 $O(N!)$ 的理论复杂度相符。然而，性能提升倍数（标准回溯时间 / 位运算时间）将非常可观，并可能随 `N` 增大而继续增长。当 `N` 较大时，标准回溯算法会变得不切实际地缓慢，而位运算版本则可以在此基础上多算几阶。

4. 优化思路

4.1. 剪枝

剪枝决策发生在 `for` 循环内部，通过调用 `is_safe` 函数实现。

```
# 常规回溯中的剪枝点
for col in range(n):
    if is_safe(queens, row, col):  # <--- 在此进行剪枝决策
        # 如果安全，则深入搜索；否则，跳过，即剪掉该分支
        solve_backtrack(row + 1)
```

这里的 `is_safe` 函数必须通过一个循环来 **逐一检查** 当前位置是否与之前所有行已放置的皇后冲突。这是一种 **串行剪枝**，为了判断一个分支是否要被剪掉，需要进行 O(N) 复杂度的计算。当 N 增大时，这种高昂的剪枝成本使其效率低下。

4.2. 位运算

位运算利用 CPU 对二进制位操作的极高效率，将复杂的逻辑判断转变为几次简单的指令。

- **核心位运算及其作用**：
    1. **按位或 `|`**：用于 **合并** 攻击范围。`cols | ld | rd` 将所有被占用的列和对角线信息整合到一个“攻击位图”中。
    2. **按位非 `~`**：用于 **反转** 状态。`~(...)` 将“攻击位图”变为“安全位图”，值为 1 的位代表可以放置皇后的安全列。
    3. **按位与 `&`**：用于 **筛选和隔离**。
        - `& ((1 << n) - 1)`：用一个全1的掩码来屏蔽高位干扰，确保操作始终在 N×N 的棋盘内。
        - `available & -available`：这是一个精妙的技巧，用于分离出 `available` 中最低位的 1（即最右边的那个安全位置），从而逐一尝试所有可行的选择。
    4. **位移 `<<` 和 `>>`**：用于 **状态转移**。在递归到下一行时，上一行皇后的对角线影响会发生平移。这恰好对应位运算中的左移和右移，以 O(1) 的代价完成了对角线攻击状态的更新。

```
# 高效的状态更新
solve_bits(row + 1,
           cols | pos,         # 更新列
           (ld | pos) << 1,  # 更新左对角线
           (rd | pos) >> 1)  # 更新右对角线
```

通过这些操作，原本需要 O(N) 循环的 `is_safe` 检查，被 O(1) 的位运算所取代，实现了并行化的判断。

4.3. 状态压缩

- **常规方法**：使用一个数组 `queens`，`queens[r] = c` 表示第 `r` 行的皇后在第 `c` 列。状态分散，难以进行整体运算。
- **优化方法**：使用三个整数 `cols`, `ld`, `rd` 作为 **位掩码 (Bitmask)**。
    - `cols`：其二进制的第 `k` 位为 1，代表棋盘的第 `k` 列已被占用。
    - `ld`：其第 `k` 位为 1，代表第 `k` 条左对角线 (`/`) 被占用。
    - `rd`：其第 `k` 位为 1，代表第 `k` 条右对角线 (`\`) 被占用。

4.5. 启发式排序

> 但是完成后运行发现效率更低，所以没有在代码中体现

启发式排序是一种更高级的剪枝策略，它关注于 **先走哪一步”** 的问题。其核心思想是 **最快失败原则**，即优先选择那些最可能导致后续无解的分支进行探索，以便能更早地剪掉更大的搜索子树。

可行的启发式策略：

1. **优先选择中间列**：一个常见的启发是，棋盘中间的列通常比边缘的列受到更多的约束。因此，可以修改 while 循环的逻辑，不再总是取最低位的1，而是优先尝试 available 位图中靠近中间 n/2 的位置。这可能会更快地发现冲突，从而加速剪枝。
2. **最少剩余值**：虽然在本问题中每行的选择是独立的，但这个思想可以引申为：在所有可用的 available 位置中，选择那个能让 **下一行** 剩余选择最少的位置。这需要更复杂的预计算，但体现了启发式搜索的核心。