N 皇后问题实验报告

肖俊浩 2023141461149

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、实验结果、优化思路、实验分析、更优算法求解 N≥12 时的情况。

一、算法说明

源代码核心算法采用回溯法，而回溯法本质上一种深度优先（DFS）的算法。也就是我们通过递归的方式穷举整个解空间（在不优化的情况下），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，还是选择暴力穷举的话非常耗费计算时间。

二、优化思路

我的优化思路是通过剪枝的策略减少对无效路径的遍历。通过一些辅助条件来尽早判断某些路径是否有效。主要的优化点有两个，其一是通过初始化三个集合记录冲突的行、列、对角线；其二是通过镜像的思想减少对无效路径的遍历，直接找出对称解。

这是我的优化思路伪代码

```
def backtrack(row):
    if row == n:
        # 找到一个有效解，保存结果
        solution = [''.join(row) for row in board]
        result.append(solution)
        return
```

```
    # 确定当前行的列搜索范围
    if row == 0 and n > 1:
        # 镜像优化：仅遍历前一半列
        col_range = range((n + 1) // 2) if n % 2 == 1 else range(n // 2)
    else:
        col_range = range(n)
    for col in col_range:
        # 检查当前列是否冲突
        if col in cols or (row + col) in pos_diag or (row - col) in neg_diag:
            continue
        # 放置皇后
        board[row][col] = 'Q'
        cols.add(col)
        pos_diag.add(row + col)
        neg_diag.add(row - col)
        # 递归下一行
        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)
    # 生成镜像解（仅当 row == 0 时）
    if row == 0 and n > 1 and result:
        # 镜像翻转已找到的解
        original_count = len(result)
        for i in range(original_count):
            mirror_solution = [row_str[::-1] for row_str in result[i]]
            # 避免重复添加对称解（如奇数 n 时的中心对称）
            if n % 2 == 1 and result[i][0][(n-1)//2] == 'Q':
                continue
            if mirror_solution not in result:
                result.append(mirror_solution)

```

首先由三个条件来判断是否进行剪枝：
列冲突：col in cols → 当前列已被占用。正斜线冲突：row + col in pos_diag → 同一正斜线（行+列相同）。负斜线冲突：row - col in neg_diag → 同一负斜线（行-列相同），这样就快速实现了冲突检测；然后是镜像优化，由于 N 皇后问题的对称性，对第一行而言，col 列与 N-col-1 列的解具有对称性，仅遍历第一行的前半部分列（range((n+1)//2)），通过反转每行字符串生成镜像解，这样就减少了很多冗余搜索。这样时间复杂度接近于 O(N * 2^N)。

三、实验结果

按照老师要求，提供 N=4 和 N=8 的测试用例。

```
D:\Anaconda\python.exe C:\Users\junhaoxiao\Desktop\作业文件夹\大二下学期\人智导作业\N_empress\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..
```

图表 1 N=4 实验结果

```
D:\Anaconda\python.exe C:\Users\junhaoxiao\Desktop\作业文件夹\大二下学期\人智导作业\N_empress\n_queens.py
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: 0.0010 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
```

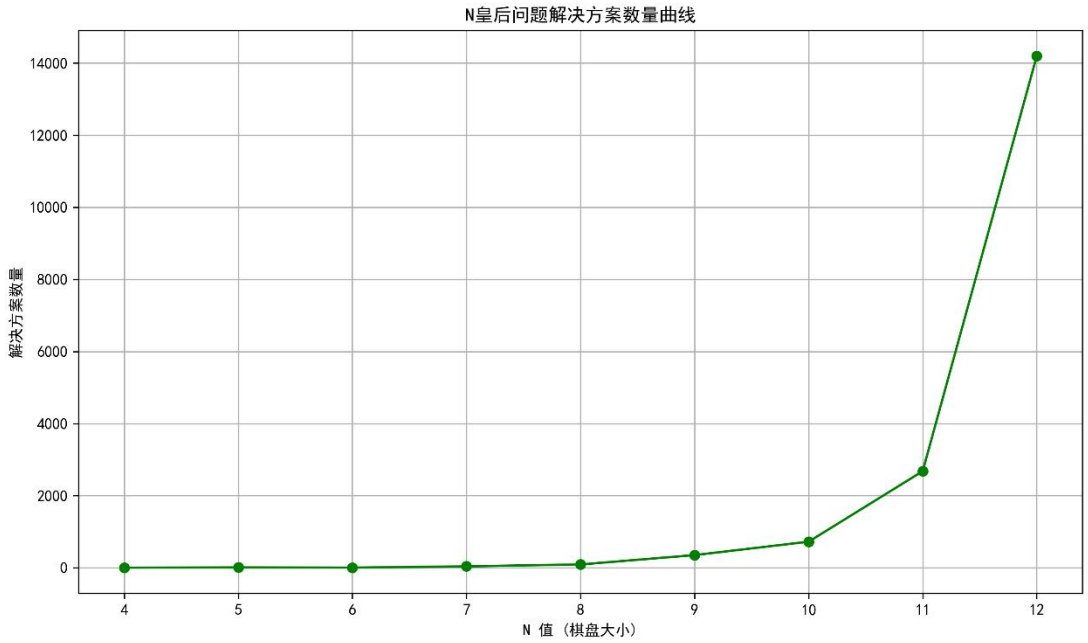
图表 2 N=8 实验结果

可以看出，算法成功求出正确值，提供了异常检测，并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

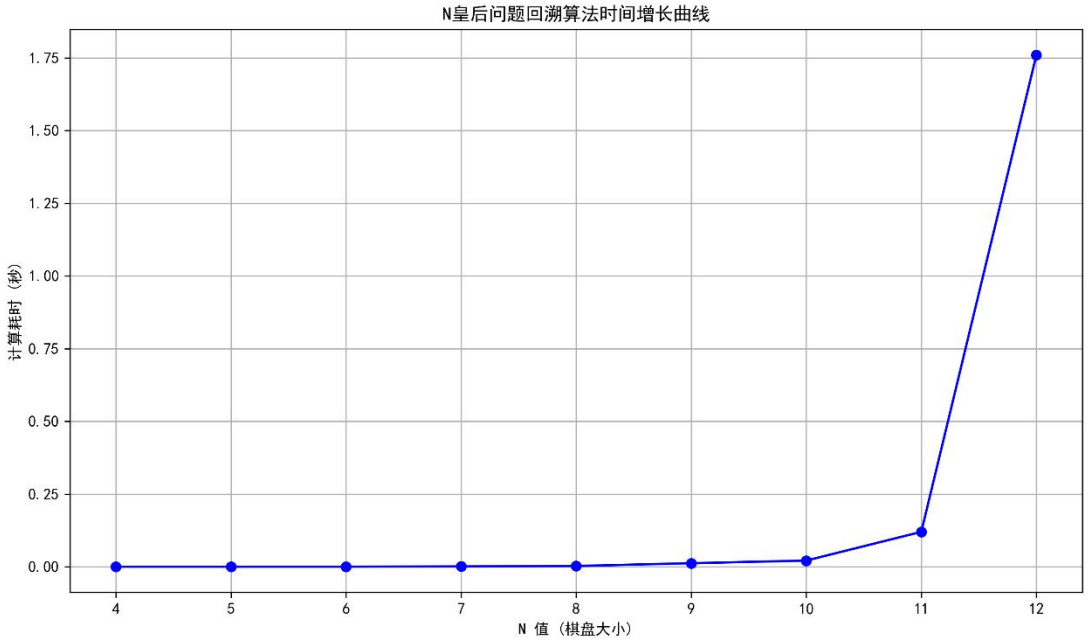
在 10^-3s。

四、实验分析

应实验要求，绘制了两个曲线图，分别是可行解数量随 N 变化增长曲线和求解时间曲线。



图表 3 解决方案数量曲线



图表 4 回溯算法求解时间曲线

根据以上两张图，可以看出 N 皇后问题解的数量在 N 增大的同时爆炸性增长，与其 NP 完全性问题的特性相符，但是与之前的时间复杂度公式相符性较差。我认为回溯剪枝算法在没有硬件优化的情况下求解 N≥12 的问题时超出了处理能力，故而运行较慢。而我的算法在 N 处于较小值（N≤10）时效率较高，N=11 时计算速度尚在可接受范围内。当 N=12 时，计算

时间急剧增加，应该选取其他算法对这样的解空间特大的问题求解。可以采用位运算替代集合，也可以采用启发式搜索替代回溯算法。

五、更优算法求解 N≥12 时的情况

我准备采用遗传算法实现这个问题：

```
GENETIC-ALGORITHM(N)
    初始化种群 POPULATION
    while 未找到解 or 达到最大迭代次数
        计算种群中每个个体的适应度
        选择高适应度个体作为父代
        对父代执行交叉操作生成子代
        对子代执行变异操作
        更新种群（替换低适应度个体）
        如果发现适应度为 0 的个体，则返回解
```

以上是遗传算法一般的解题思路，然后基于这样的思路可以写出 python 脚本

```python
import random
import time
from typing import List, Tuple
import numpy as np

def calculate_conflicts(individual: List[int]) -> int:
#计算个体的冲突数目
    n = len(individual)
    conflicts = 0

    # 检查行冲突（每个数字只能出现一次）
    conflicts += n - len(set(individual))

    # 检查斜线冲突
    for i in range(n):
        for j in range(i + 1, n):
            if abs(i - j) == abs(individual[i] - individual[j]):
                conflicts += 1

    return conflicts

def fitness(individual: List[int]) -> float:
#计算适应度值
    conflicts = calculate_conflicts(individual)
    return 1.0 / (1.0 + conflicts)  # 将冲突数转换为适应度值

def order_crossover(parent1: List[int], parent2: List[int]) -> List[int]:
#顺序交叉
```

```
    n = len(parent1)
    # 随机选择交叉点
    cx1, cx2 = sorted(random.sample(range(n), 2))

    # 从 parent1 复制中间段
    child = [-1] * n
    child[cx1:cx2] = parent1[cx1:cx2]

    # 从 parent2 填充剩余位置
    remaining = [x for x in parent2 if x not in child[cx1:cx2]]
    j = 0
    for i in range(n):
        if child[i] == -1:
            child[i] = remaining[j]
            j += 1

    return child

def mutate(individual: List[int], mutation_rate: float) -> List[int]:
#变异操作
    if random.random() < mutation_rate:
        n = len(individual)
        i, j = random.sample(range(n), 2)
        individual[i], individual[j] = individual[j], individual[i]
    return individual

def select_parents(population: List[List[int]], fitnesses: List[float], num_parents:
int) -> List[List[int]]:
#锦标赛
    parents = []
    for _ in range(num_parents):
        # 随机选择 3 个个体进行锦标赛
        tournament = random.sample(list(zip(population, fitnesses)), 3)
        # 选择适应度最高的个体
        winner = max(tournament, key=lambda x: x[1])[0]
        parents.append(winner)
    return parents

def genetic_algorithm(n: int, pop_size: int = 100, max_generations: int = 1000) ->
Tuple[List[int], float]:
#遗传算法
    start_time = time.time()

    # 初始化种群
```

```
    population = [random.sample(range(n), n) for _ in range(pop_size)]
    best_fitness = 0
    best_solution = None
    generation_without_improvement = 0
    mutation_rate = 0.1  # 初始变异率

    for generation in range(max_generations):
        # 计算适应度
        fitnesses = [fitness(ind) for ind in population]

        # 更新最优解
        max_fitness = max(fitnesses)
        if max_fitness > best_fitness:
            best_fitness = max_fitness
            best_solution = population[fitnesses.index(max_fitness)]
            generation_without_improvement = 0
        else:
            generation_without_improvement += 1

        # 自适应变异率
        if generation_without_improvement > 20:
            mutation_rate = min(0.5, mutation_rate * 1.1)
        else:
            mutation_rate = max(0.01, mutation_rate * 0.99)

        # 检查是否找到解
        if best_fitness == 1.0:  # 无冲突
            print(f"找到解！在第 {generation} 代")
            break

        # 选择父代
        parents = select_parents(population, fitnesses, pop_size // 2)

        # 精英保留
        elite_size = pop_size // 10
        elite = sorted(zip(population, fitnesses), key=lambda x: x[1],
reverse=True)[:elite_size]
        elite = [ind for ind, _ in elite]

        # 生成新种群
        new_population = elite.copy()
        while len(new_population) < pop_size:
            parent1, parent2 = random.sample(parents, 2)
            child = order_crossover(parent1, parent2)
```

```
            child = mutate(child, mutation_rate)
            new_population.append(child)

        population = new_population

        # 每 100 代打印一次进度
        if generation % 100 == 0:
            print(f"第 {generation} 代，当前最优适应度: {best_fitness:.4f}，变异率: {mutation_rate:.4f}")

    end_time = time.time()
    running_time = end_time - start_time

    if best_solution is None:
        print("未找到解")
        return None, running_time

    return best_solution, running_time

def print_board(solution: List[int]) -> None:
#可视化解集
    n = len(solution)
    for row in range(n):
        line = ['.'] * n
        line[solution[row]] = 'Q'
        print(' '.join(line))

def main():
    # 测试不同规模的 N 皇后问题
    n_values = [8, 12, 16]
    for n in n_values:
        print(f"\n 求解 {n} 皇后问题: ")
        solution, running_time = genetic_algorithm(n, pop_size=200, max_generations=2000)
        if solution:
            print(f"运行时间: {running_time:.2f} 秒")
            print("解: ")
            print_board(solution)
        else:
            print(f"运行时间: {running_time:.2f} 秒")
            print("未找到解")

if __name__ == "__main__":
    main()
```

156.

运行上述代码得到结果：

```
(base) PS C:\Users\junhaoxiao\Desktop\作业文件夹\大二下学期\人智导作业\N_empress> & D:/Anaconda/python.exe c:/Users/junhaoxiao/Desktop/作业文件夹/大二下学期/人智导作业/N_empress/N_queens_evo.py

求解 8 皇后问题:
第 0 代，当前最优适应度: 0.5000，变异率: 0.0990
找到解！在第 1 代
运行时间: 0.00 秒
解:
. . . Q . . . .
. Q . . . . . .
. . . . . . . Q
. . . . Q . . .
. . . . . Q . .
Q . . . . . . .
. . Q . . . . .
. . . . . Q . .

求解 12 皇后问题:
第 0 代，当前最优适应度: 0.3333，变异率: 0.0990
找到解！在第 52 代
运行时间: 0.13 秒
解:
. . . . . Q . . . . . .
. . . . . . . . . Q . .
. . . . . Q . . . . . .
Q . . . . . . . . . . .
. . Q . . . . . . . . .
. . . . Q . . . . . . .
. Q . . . . . . . . . .
. . . . . . . . . . Q .
. . . . . . . . Q . . .
. . . . . . . . . . . Q
. . . Q . . . . . . . .
. . . . . . . Q . . . .
```

```
求解 16 皇后问题:
第 0 代，当前最优适应度: 0.2500，变异率: 0.0990
找到解！在第 46 代
运行时间: 0.16 秒
解:
. . . . . . . . . . Q . . . . .
. . . . . . . . . . . . Q . . .
. . . . . . . . . . Q . . . . . .
. . . . . . Q . . . . . . . . . .
. . . . Q . . . . . . . . . . . .
. Q . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . Q .
Q . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . Q . .
. . . . . . . . . Q . . . . . . .
. . Q . . . . . . . . . . . . . .
. . . . . Q . . . . . . . . . . .
. . . . . . . Q . . . . . . . . .
. . . . . . . . . . . . . . . . Q
. . . . . . . . . . . Q . . . . .
. . . Q . . . . . . . . . . . . .
```

可以看出，遗传算法相比于回溯剪枝算法速度上有了一定的提升，在 N=12 时运行时间从 1.75s 缩减到 0.13s，在 N=16 时运行时间也仅仅是 0.16s。但是遗传算法的缺点是无法找到所有解，倾向于寻找一个局部最优解。这也是这里选择输出一个解的原因。这个问题是遗传算法的局限性。