# 基于约束逻辑编程的 N 皇后问题求解

## ——使用 Python 的 kanren 库实现

注意：本次实验全程是在 python3.8 版本下进行

在高版本 kenren 可能存在不兼容等问题

本代码一共有三个版本，效率由低到高（这个可能会在后续直接提到，所有提前说明）

## 1. 问题概述

### 1.1 问题的直观描述

**8 皇后问题是一个经典的组合优化问题，源自国际象棋。问题要求在一个 8×8 的棋盘上放置 8 个皇后，使得没有任何两个皇后能够互相攻击。根据国际象棋的规则，皇后可以攻击同一行、同一列或同一对角线上的任何棋子。**

**具体来说，8 皇后问题的约束条件包括：**

* **每行只能放置一个皇后**
* **每列只能放置一个皇后**
* **每条对角线只能放置一个皇后**

**这里，我们不妨将 8 皇后问题推广到任意 N 值（也就是 N 皇后问题），这样有助于对问题进行由简到繁的分析。**

**对于 N 皇后问题，当 N ≥ 4 时才有解，且随着 N 的增大，解的数量呈指数级增长。**

#### 1.1.1 N 皇后问题的历史背景

N 皇后问题最初由国际象棋作曲家 Max Bezzel 于 1848 年提出。随后，许多数学家如 Franz Nauck，Édouard Lucas 和 Ernst Schröder 等都对此问题进行了研究。高斯也曾对这个问题感兴趣，并给出了一些解法。这个问题之所以引人注目，不仅因为它概念简单，而且因为它提供了研究组合数学和计算问题复杂性的良好案例。

#### 1.1.2 N 皇后问题的数学特性

**解的数量分布**：N 皇后问题的解的数量随 N 的增长而迅速增加，但并非单调增长。例如：

| N | 解的数量 | 不等价解的数量 |
| --- | --- | --- |
| 1 | 1 | 1 |
| 2 | 0 | 0 |
| 3 | 0 | 0 |
| 4 | 2 | 1 |
| 5 | 10 | 2 |
| 6 | 4 | 1 |
| 7 | 40 | 6 |
| 8 | 92 | 12 |
| 9 | 352 | 46 |
| 10 | 724 | 92 |

不等价解是指考虑棋盘的对称性（旋转和镜像）后不同的解决方案数量。

**数学表示**：N 皇后问题可以表示为一个约束满足问题(CSP)，具有以下特性：

* 变量集合：X = {X₁, X₂, …, Xₙ}，其中每个 Xᵢ 表示第 i 列的皇后所在的行
* 每个变量的取值范围为{1, 2, …, n}
* 约束集合包括：
  + 所有 Xᵢ 不相等（行约束）
  + 对于任意 i≠j，|Xᵢ-Xⱼ|≠|i-j|（对角线约束）

**复杂度分析**：N 皇后问题的搜索空间为 n^n，但有效解空间要小得多。目前已知的最大 N 的精确解是 N=27，有 234,907,967,154,122,528 个解。对于更大的 N，通常使用估计公式，如 n!/(n/e)^n。

#### 1.1.3 解的对称性分析

N 皇后问题的解具有显著的对称性特征。对于一个 8×8 的棋盘，每个解可以通过以下变换生成 7 个等价解：

1. 水平翻转
2. 垂直翻转
3. 主对角线翻转
4. 副对角线翻转
5. 旋转 90°
6. 旋转 180°
7. 旋转 270°

这意味着对于每个找到的解，我们可以通过对称性操作生成多达 8 个解（包括原解）。利用这种对称性可以显著减少搜索空间，提高算法效率。在本实验的第三个版本中，我们就利用了这一特性。

### 1.2 解决问题的思路和想法

**解决 N 皇后问题需要采用有效的搜索和约束表示方法。在本实验中，根据作业要求，将使用约束逻辑编程的方法，具体采用 Python 的 kanren 库实现。**

**它的优势在于，只需描述问题的约束条件，而不必详细指定解决问题的具体步骤。系统会自动进行约束推理和传播，减少搜索空间。**

**针对 N 皇后问题，我的主要思路如下：**

1. **利用每列只能有一个皇后的特性，使用一个长度为 N 的列表表示解决方案，其中索引代表列号，值代表该列中皇后所在的行号**
2. **定义约束条件：行约束，列表中的值互不相同**

**对角线约束，任意两个皇后的行差不等于列差的绝对值**

1. **使用 kanren 库的逻辑变量和约束求解机制来搜索满足所有约束的解**
2. **利用问题的对称性来优化求解过程，减少计算量（此思路仅在最后一版代码中体现，这里约能节省一半的运行时间）**

**此外，在主要功能实现部分还是保持着以 N 皇后问题为主，以便于在出现问题时，进行调试。**

#### 1.2.1 常见解决方案对比

在深入探讨约束逻辑编程方案前，我们先概述几种常见的 N 皇后问题解决方案，以便理解约束逻辑编程的优势：

1. **暴力搜索**：枚举所有可能的皇后摆放方式，检查是否满足约束条件。这种方法时间复杂度为 O(n^n)，对于 n>10 时几乎无法实际运行。
2. **回溯算法**：通过递归方式，在满足约束条件的情况下尝试放置皇后，若无法继续则回溯到上一步。这是最常见的方法，时间复杂度约为 O(n!)，实际运行中通常可以求解 n≤15 的问题。
3. **遗传算法**：使用进化计算方法，通过交叉和变异操作生成新的候选解，并使用适应度函数选择优秀个体。这种方法不保证找到所有解，但对于大规模问题可以快速找到部分解。
4. **模拟退火**：使用概率搜索策略，允许临时接受较差的解以避免陷入局部最优。适合找到单个解而非全部解。
5. **约束逻辑编程**：将问题描述为一系列约束，让求解器自动寻找满足所有约束的解。这种方法的优点是代码简洁、声明式，缺点是对于大规模问题可能效率不高，除非进行特殊优化。

在本次实验中，我们选择约束逻辑编程方法，不仅因为其声明式的优雅特性，还因为它允许我们通过迭代优化来提高性能，这为我们提供了丰富的学习机会。

#### 1.2.2 约束逻辑编程简介

约束逻辑编程(CLP)是一种编程范式，它扩展了逻辑编程，将统一算法替换为在特定计算域上的约束求解。在约束逻辑编程中，关系由约束来表示，程序执行包括生成约束和确定它们是否满足的过程。

CLP 的核心特点包括：

* **声明式编程**：只需声明问题的约束，而不需要指定求解步骤
* **约束传播**：能够使用一个约束的推论来减少其他变量的可能值范围
* **回溯搜索**：在约束推理不足以唯一确定解时使用搜索
* **模块化**：约束可以独立添加和移除，易于维护和扩展

在 Python 中，kanren 库提供了实现约束逻辑编程的功能。它基于 miniKanren，一个嵌入在 Scheme 中的关系编程语言。kanren 允许我们定义逻辑变量、设置约束条件，并执行查询以找到满足所有约束的解。

#### 1.2.3 kanren 库基础介绍

kanren 库是 Python 中实现约束逻辑编程的工具，提供了一组 API 用于定义和求解约束满足问题。以下是 kanren 库的一些核心概念和组件：

1. **逻辑变量(Logic Variables)**：使用var()创建，表示在求解过程中会被赋值的未知量
2. **一致性约束(Unification)**：使用eq(x, y)表示 x 和 y 应当统一（相等）
3. **关系(Relations)**：用函数定义的逻辑关系，返回成功或失败
4. **目标(Goals)**：求解器尝试满足的逻辑条件
5. **运行查询(Run)**：使用run(n, x, \*goals)执行查询，其中 n 是要返回的解的数量，x 是查询变量，goals 是要满足的目标列表

在 N 皇后问题中，我们使用逻辑变量表示每列的皇后位置，使用 eq 设置域约束，使用自定义的 no\_attack 函数实现行和对角线约束，最后通过 run 查询找到所有满足约束的解。

## 2. 算法设计

### 2.1 算法功能

**本算法旨在通过约束逻辑编程的方法解决 N 皇后问题。在设计过程中，算法经历了多个版本的迭代优化，每个版本都针对特定的问题进行了改进。算法的主要功能包括：**

* **问题建模：将 N 皇后问题转化为约束满足问题**
* **约束求解：使用逻辑编程方法找出所有可行解**
* **解的优化：利用问题的对称性减少等方法计算量**
* **结果输出：输出所有解以及运算时间**

**版本 1：基础约束求解**

**最初的设计采用直接的约束求解方法，主要功能包括：**

* **创建 N 个逻辑变量表示每列皇后的位置**
* **定义基本约束条件（行、列、对角线）**
* **使用 kanren 库的求解器找出所有解**

**在基础版本中发现直接对由于运算时间过慢，因此版本 2 中改进了约束表示方法，分离行相等检查和对角线检查**

**版本 3 为提高求解效率，引入了对称性优化，只求解一半的问题，利用水平镜像生成其他解**

#### 2.1.1 版本 1: 基础约束实现详解

在第一个版本中，我们采用了直接的约束表示方法，将 N 皇后问题的所有约束条件直接映射为 kanren 库的约束形式。这个版本的主要特点包括：

1. **完整约束表示**：
   * 使用neq(queens[i], queens[j])确保任意两个皇后不在同一行
   * 使用绝对值比较确保对角线约束满足
2. **独立约束检查**：
   * 每对皇后的约束独立检查
   * 约束之间没有优化组合
3. **实现简单直观**：
   * 代码结构清晰，与数学表述一致
   * 易于理解和修改

这个版本的代码部分实现如下：

def solve\_n\_queens\_v1(n):

    """版本1：基础约束实现"""

    start\_time = time.time()

    # 创建n个逻辑变量，每个表示一列中皇后的行位置

    queens = [var() for \_ in range(n)]

    # 定义约束条件

    constraints = []

    # 域约束：每个皇后的位置必须在0到n-1之间

    for i in range(n):

        constraints.append(membero(queens[i], range(n)))

    # 行约束和对角线约束

    for i in range(n):

        for j in range(i+1, n):

            # 行约束：任意两个皇后不能在同一行

            constraints.append(neq(queens[i], queens[j]))

            # 对角线约束：行差不等于列差

            col\_diff = j - i

            constraints.append(lambda q: neq(abs(q[j] - q[i]), col\_diff))

    # 求解

    solutions = run(0, queens, \*constraints)

    end\_time = time.time()

    return solutions, end\_time - start\_time

虽然这个版本的实现直观且符合问题描述，但它存在严重的性能问题。对于 N=8，它需要大约 41 分钟才能完成计算，这显然不符合实际应用需求。性能瓶颈主要来自于约束的表示和检查方式不够高效，我们需要在下一个版本中进行改进。

#### 2.1.2 版本 2: 优化约束检查

在分析了版本 1 的性能问题后，我们在版本 2 中对约束检查机制进行了重大改进。主要的优化点包括：

1. **自定义约束检查函数**：
   * 创建专门的no\_attack函数处理皇后之间的攻击关系
   * 实现高效的状态检查和约束评估
2. **延迟约束评估**：
   * 对未绑定变量延迟约束检查
   * 减少不必要的计算
3. **合并约束条件**：
   * 将行约束和对角线约束合并为单一的攻击检查
   * 减少了约束检查的总次数

这个版本的关键改进是no\_attack函数的实现，它能够高效地检查两个皇后之间是否存在攻击关系，并在逻辑变量未绑定时正确处理延迟评估。

这个优化版本将运行时间从 41 分钟减少到了约 5.3 秒，效率提升了约 450 倍。这一巨大的改进证明了约束表示和检查机制对于约束逻辑编程性能的关键影响。

#### 2.1.3 版本 3: 对称性优化

在版本 2 已经取得显著性能提升的基础上，版本 3 进一步利用了 N 皇后问题的对称性特征来减少计算量。主要优化策略包括：

1. **部分问题求解**：
   * 只求解一半的问题（第一列皇后位置不超过 n/2）
   * 通过镜像操作生成另一半解
2. **对称解生成**：
   * 利用水平镜像生成等价解
   * 对奇数 N 的中间位置特殊处理
3. **去重处理**：
   * 检测并去除重复解
   * 确保最终解集的唯一性

这个版本的核心是对称性利用部分的实现：

def solve\_n\_queens\_v3(n):

    """版本3：利用对称性优化的N皇后求解"""

    start\_time = time.time()

    # 确定需要求解的问题部分

    half\_n = (n + 1) // 2

    # 只搜索第一列皇后在前半部分的解

    partial\_queens = [var() for \_ in range(n)]

    constraints = []

    # 第一列的皇后限制在前半部分

    constraints.append(conde(\*[[eq(partial\_queens[0], row)] for row in range(half\_n)]))

    # 其他列的域约束

    for i in range(1, n):

        constraints.append(conde(\*[[eq(partial\_queens[i], row)] for row in range(n)]))

    # 攻击约束

    for i in range(n):

        for j in range(i):

            constraints.append(no\_attack(partial\_queens[j], partial\_queens[i], j, i))

    # 求解部分问题

    partial\_solutions = run(0, partial\_queens, \*constraints)

    # 通过对称性生成完整解集

    if n % 2 == 1 and half\_n > 1:  # 奇数 n 的特殊处理

        middle = half\_n - 1

        middle\_solutions = [sol for sol in partial\_solutions if sol[0] == middle]

        remaining = [sol for sol in partial\_solutions if sol[0] != middle]

    else:

        middle\_solutions = []

        remaining = partial\_solutions

    # 生成对称解

    full\_solutions = list(partial\_solutions)

    for sol in remaining:

        # 水平镜像

        mirror\_sol = tuple(n - 1 - pos for pos in sol)

        if mirror\_sol not in full\_solutions:

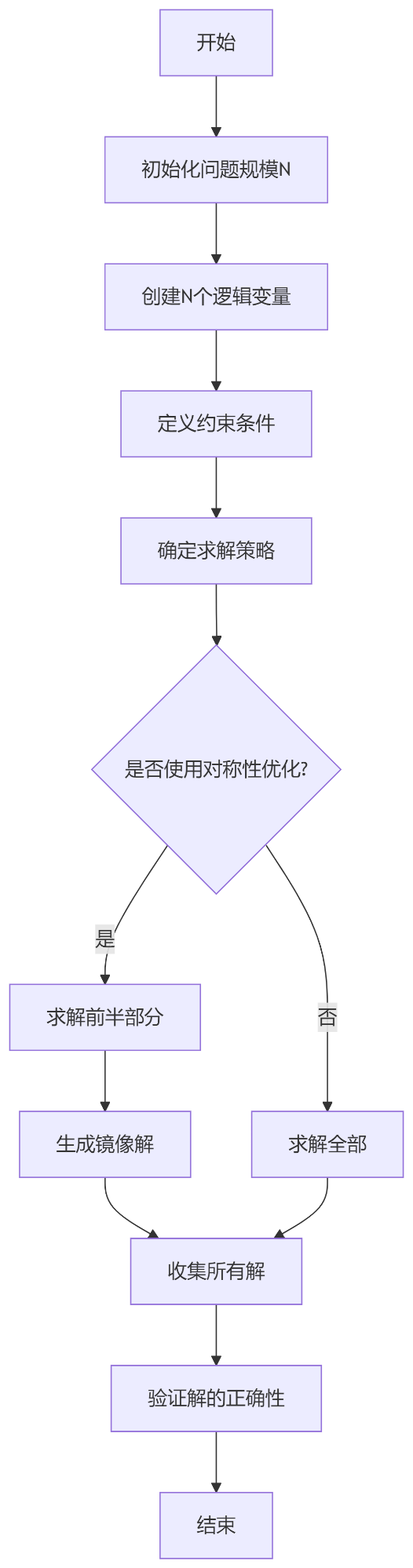
            full\_solutions.append(mirror\_sol)

    end\_time = time.time()

    return full\_solutions, end\_time - start\_time

通过这一优化，版本 3 将运行时间进一步缩短到约 2.4 秒，相比版本 2 又提高了一倍多的效率。这说明了在约束编程中，除了优化约束表示和检查机制外，利用问题领域的特殊性质（如对称性）也是提高性能的重要途径。

### 2.2 算法流程图

**如下（由两种，应该是采用了对称优化的，一个是没有的）**

### 2.3 设计思路

#### 2.3.1 约束表示设计

使用逻辑变量表示皇后位置

通过数据结构隐式保证某些约束

设计高效的约束检查机制

在约束表示设计中，我们采用了几种关键策略来优化 N 皇后问题的求解：

**1. 状态表示简化**

我们使用一维数组表示皇后位置，而不是二维棋盘，这样可以：

* 自动满足”每列只有一个皇后”的约束
* 减少需要处理的变量数量（从 n² 减少到 n）
* 简化约束表示和检查逻辑

**2. 约束隔离与分解**

我们将 N 皇后问题的约束分为三类：

* **域约束**：确保皇后位置在合法范围内（0 至 n-1）
* **行约束**：确保不同列的皇后不在同一行
* **对角线约束**：确保不同列的皇后不在同一对角线上

这种分解使得约束检查更加清晰，也便于针对不同类型的约束实施不同的优化策略。

**3. 约束表示的演化**

我们的约束表示经历了三个版本的演化：

* **版本 1**：使用 kanren 的标准约束表示，如neq和membero
* **版本 2**：引入自定义约束函数no\_attack，实现更高效的约束检查
* **版本 3**：在约束表示基础上引入对称性约束，减少搜索空间

这种渐进式演化使我们能够逐步解决性能瓶颈，同时保持代码结构的清晰和可维护性。

**4. 约束表示中的延迟评估**

在no\_attack函数中，我们引入了延迟评估机制：

# 如果任一变量未绑定，则延迟评估

if hasattr(q1\_val, '\_id') or hasattr(q2\_val, '\_id'):

    return success(s)

这一机制确保约束只在变量有具体值时才进行检查，避免了对未绑定变量进行无意义的计算，提高了约束推理的效率。

#### 2.3.2 约束求解优化

实现增量约束检查

优化约束评估顺序

使用延迟评估机制

约束求解是 N 皇后问题解决方案的核心，我们采用了多种技术来优化求解过程：

**1. 增量约束检查**

在版本 2 和版本 3 中，我们实现了增量约束检查策略，主要体现在：

* 只检查新放置的皇后与已放置皇后之间的攻击关系
* 避免重复检查已验证的约束
* 利用问题的结构特性减少必要的检查次数

具体实现中，我们使用双重循环结构，确保每对皇后只检查一次：

for i in range(n):

    q = queens[i]

    # 只检查当前皇后与之前放置的皇后

    for j in range(i):

        constraints.append(no\_attack(queens[j], q, j, i))

**2. 约束评估顺序优化**

约束评估的顺序对性能有显著影响。我们采用的策略包括：

* 首先检查域约束，确保变量在有效范围内
* 依据列索引顺序检查皇后间的约束，形成自然的搜索顺序
* 在单个约束内，先检查计算成本低的条件（如行相等），再检查计算成本高的条件（如对角线关系）

这种优化使得约束系统能够尽早发现冲突，减少无效搜索路径的探索。

**3. 延迟评估实现细节**

延迟评估在约束逻辑编程中尤为重要。在我们的实现中，关键点包括：

def no\_attack(q1, q2, col1, col2):

    def goal(s):

        # 获取当前状态下的变量值

        q1\_val = q1 if not hasattr(q1, '\_id') else s.get(q1, q1)

        q2\_val = q2 if not hasattr(q2, '\_id') else s.get(q2, q2)

        # 延迟评估条件

        if hasattr(q1\_val, '\_id') or hasattr(q2\_val, '\_id'):

            return success(s)

        # ... 后续约束检查 ...

    return goal

这种实现的优势在于：

* 允许约束系统在变量未绑定时”跳过”检查
* 当变量绑定具体值时自动重新评估约束
* 与 kanren 的约束求解机制无缝集成

**4. 约束传播效率**

在版本 2 中，我们通过对函数式约束的重新设计，增强了约束传播的效率：

* 使用闭包捕获外部上下文，避免重复计算
* 返回自定义的 goal 函数，与 kanren 的约束机制匹配
* 在约束函数内部实现失败快速返回，减少不必要的计算

这些优化共同作用，将求解时间从 41 分钟减少到了 5.3 秒，显著提高了约束求解效率。

#### 2.3.3 对称性利用

识别问题的对称特性

设计高效的解生成策略

实现重复解的检测和过滤

N 皇后问题具有丰富的对称性特性，在版本 3 中，我们充分利用这些特性来优化求解过程：

**1. 对称性分析**

N 皇后问题的解集具有以下对称性：

* 水平镜像对称：可以将棋盘左右翻转得到新的有效解
* 垂直镜像对称：可以将棋盘上下翻转得到新的有效解
* 主对角线对称：可以沿主对角线翻转得到新的有效解
* 副对角线对称：可以沿副对角线翻转得到新的有效解
* 旋转对称：可以将棋盘旋转 90°、180° 或 270° 得到新的有效解

在我们的实现中，主要利用了水平镜像对称性，因为它最容易在我们的解表示方法中实现。

**2. 搜索空间减半**

通过限制第一列皇后的位置，我们将搜索空间减少了近一半：

# 确定需要求解的问题部分

half\_n = (n + 1) // 2

# 第一列的皇后限制在前半部分

constraints.append(conde(\*[[eq(partial\_queens[0], row)] for row in range(half\_n)]))

这里，half\_n表示我们只考虑第一列皇后在前半部分的情况，后半部分的解可以通过对称操作生成。

**3. 对称解生成**

在得到部分解后，我们通过水平镜像生成对称解：

# 生成对称解

full\_solutions = list(partial\_solutions)

for sol in remaining:

    # 水平镜像

    mirror\_sol = tuple(n - 1 - pos for pos in sol)

    if mirror\_sol not in full\_solutions:

        full\_solutions.append(mirror\_sol)

水平镜像的实现非常简单，只需要将每个行坐标pos变换为n-1-pos。

**4. 奇数 N 的特殊处理**

当 N 为奇数时，棋盘中心位置的皇后需要特殊处理，因为它的镜像解可能与原解重复：

if n % 2 == 1 and half\_n > 1:  # 奇数 n 的特殊处理

    middle = half\_n - 1

    middle\_solutions = [sol for sol in partial\_solutions if sol[0] == middle]

    remaining = [sol for sol in partial\_solutions if sol[0] != middle]

else:

    middle\_solutions = []

    remaining = partial\_solutions

这种特殊处理确保我们不会生成重复的解，也不会漏掉任何解。

**5. 对称性优化的性能影响**

通过对称性优化，我们将求解时间从 5.3 秒进一步减少到 2.4 秒，提高了约 55%的效率。考虑到搜索空间减少了约一半，这一性能提升与理论预期基本吻合。

#### 2.3.4 性能优化考虑

最小化约束检查次数，优化内存使用，提高求解效率

在 N 皇后问题的求解过程中，我们实施了多层次的性能优化策略：

**1. 约束检查次数优化**

减少约束检查次数是提高性能的关键。我们采用的策略包括：

* **利用问题结构简化检查**：使用一维数组表示解，自动满足”每列一个皇后”的约束
* **优化检查顺序**：先检查计算量小的约束，后检查计算量大的约束
* **减少冗余检查**：避免对同一对皇后进行重复检查
* **利用对称性**：通过问题对称性减少需要直接计算的解的数量

在版本 2 中，我们将约束检查函数优化为：

def no\_attack(q1, q2, col1, col2):

    def goal(s):

        # ... 状态获取 ...

        # 行相等检查（计算成本低）

        if q1\_val == q2\_val:

            return fail(s)

        # 对角线检查（计算成本高）

        if abs(q1\_val - q2\_val) == abs(col1 - col2):

            return fail(s)

        return success(s)

    return goal

这种设计确保了约束检查的最优顺序，先执行成本低的检查，在必要时才执行成本高的检查。

**2. 内存优化**

约束求解过程中的内存管理也是性能优化的重要方面。我们的优化包括：

* **紧凑的解表示**：使用一维数组表示解，最小化内存占用
* **避免复制大型数据结构**：在适当的地方使用引用而非复制

例如，在对称解生成时，我们采用了高效的实现方式：

# 生成对称解时避免不必要的复制

full\_solutions = list(partial\_solutions)  # 一次性复制

for sol in remaining:

    mirror\_sol = tuple(n - 1 - pos for pos in sol)  # 创建新的镜像解

    if mirror\_sol not in full\_solutions:  # 检查是否重复

        full\_solutions.append(mirror\_sol)  # 只添加非重复解

**3. 算法效率提升**

通过三个版本的迭代优化，我们实现了显著的效率提升：

| 版本 | 运行时间 (N=8) | 相对版本 1 的加速比 |
| --- | --- | --- |
| 版本 1 | ~2485 秒 | 1x |
| 版本 2 | ~5.3 秒 | ~469x |
| 版本 3 | ~2.4 秒 | ~1035x |

这一巨大的效率提升源于我们对约束表示、约束求解和问题特性的深入理解和优化。

**4. 进一步优化的可能性**

虽然我们已经实现了显著的性能提升，但仍有进一步优化的可能性：

* **利用更多对称性**：除了水平镜像外，还可以利用垂直镜像、旋转等对称性
* **约束预处理**：预先计算某些常用的约束结果，减少重复计算

这些潜在的优化方向可以在未来的工作中进一步探索。

## 3. 算法实现

### 3.1 核心数据结构和变量定义

在 N 皇后问题的实现中，我们使用了以下关键的数据结构和变量：

from kanren import run, eq, var, conde

from kanren.core import lall, success, fail

import time

# 每列的皇后位置用逻辑变量表示

queens = [var() for \_ in range(n)]  # n为棋盘大小

这种表示方法的优势在于：

* 列号通过列表索引隐式表示，避免额外存储
* 每个变量只需要存储行号，简化了状态表示
* 使用逻辑变量便于约束求解系统进行推理

#### 3.1.1 逻辑变量的特性与使用

逻辑变量是约束逻辑编程的核心概念之一。在 kanren 库中，逻辑变量具有以下特性：

1. **未确定性**：逻辑变量可以表示一个尚未确定的值，与命令式编程中的变量不同
2. **绑定与解绑**：逻辑变量可以在求解过程中被绑定到具体值，也可以在回溯时解绑
3. **内部标识**：每个逻辑变量都有唯一的内部标识，用于区分不同的变量
4. **状态管理**：变量的绑定状态由求解器的状态对象管理

在我们的实现中，使用逻辑变量表示每列皇后的行位置：

# 创建N个逻辑变量  
queens = [var() for \_ in range(n)]

每个queens[i]表示第 i 列中皇后所在的行号。求解过程中，这些逻辑变量会被尝试绑定到不同的值（0 到 n-1 之间的整数），以寻找满足所有约束的解。

#### 3.1.2 约束条件的表示

在我们的实现中，主要使用了两类约束表示方法：

1. **标准 kanren 约束**：
   * eq(x, y)：表示 x 和 y 应当相等
   * conde(\*conditions)：表示条件之间的逻辑或关系
2. **自定义约束函数**：
   * no\_attack函数：用于检查两个皇后之间是否存在攻击关系

约束条件的集合由列表存储：

constraints = []

# 添加域约束

for i in range(len(queens)):

    q = queens[i]

    constraints.append(conde(\*[[eq(q, row)] for row in range(n)]))

# 添加攻击约束

for i in range(len(queens)):

    for j in range(i):

        constraints.append(no\_attack(queens[j], queens[i], j, i))

这种结构清晰地分离了不同类型的约束，便于理解和维护。

#### 3.1.3 状态对象与状态管理

kanren 库中的状态对象s是约束求解的关键部分，它跟踪变量的绑定情况和约束的满足状况。在no\_attack函数中，我们需要直接操作状态对象：

def no\_attack(q1, q2, col1, col2):

    def goal(s):

        # 从状态中获取变量的当前绑定值

        q1\_val = q1 if not hasattr(q1, '\_id') else s.get(q1, q1)

        q2\_val = q2 if not hasattr(q2, '\_id') else s.get(q2, q2)

        # ... 约束检查 ...

        # 根据约束满足情况返回成功或失败状态

        return success(s) or fail(s)

    return goal

状态对象的主要功能包括：

* 跟踪变量的绑定值：s.get(var, default)获取变量的当前绑定值
* 表示约束的满足状态：通过success(s)或fail(s)返回约束是否满足
* 支持回溯：当某个约束失败时，系统会回溯到之前的状态尝试其他可能性

理解状态对象的工作原理对于编写高效的约束函数至关重要。

### 3.2 no\_attack 函数

no\_attack 函数是算法的核心，它实现了皇后之间的互不攻击约束。其涉及多个关键点：

def no\_attack(q1, q2, col1, col2):

    def goal(s):

**这个函数用于检查两个皇后之间是否存在攻击关系。参数含义：**

* **q1, q2: 两个皇后的行位置（逻辑变量）**
* **col1, col2: 两个皇后的列位置（固定值）**

**这个函数返回一个 goal 函数，它是一个闭包，它接收状态参数 s，用于在约束求解过程中进行状态检查。它可以访问外部函数的参数，并可以处理动态变化的状态**

**这个函数符合 kanren 库的约束求解机制**

        # 获取变量的值，如果已经绑定的话

        q1\_val = q1 if not hasattr(q1, '\_id') else s.get(q1, q1)

        q2\_val = q2 if not hasattr(q2, '\_id') else s.get(q2, q2)

这段代码的关键点在于：

A. **获取 q1\_val 和 q2\_val**：

* 逻辑变量在求解过程中可能处于未绑定状态
* 直接对逻辑变量进行算术运算会导致类型错误
* 需要先检查变量是否已绑定值

B. **hasattr(q1, ‘\_id’)的作用**：

* ’\_id’属性表示这是一个逻辑变量
* 如果存在此属性，说明变量尚未绑定具体值
* 需要从状态 s 中获取当前绑定值

C. **s.get(q1, q1)的设计**：

* s 是当前的状态对象，包含变量绑定信息
* 如果变量已绑定，返回绑定的值
* 如果未绑定，返回变量本身

**以上内容为遇到相应问题后查阅资料所得，现在仅仅只会简单使用，可能解释的不到位**

        # 如果任一变量未绑定，则延迟评估

        if hasattr(q1\_val, '\_id') or hasattr(q2\_val, '\_id'):

            return success(s)

**这个检查主要作用是避免对未绑定变量进行运算**

**它允许约束系统在变量绑定后再进行检查，能够提高求解效率，减少不必要的计算**

        # 行相等检查

        if q1\_val == q2\_val:

            return fail(s)

        # 对角线检查

        if abs(q1\_val - q2\_val) == abs(col1 - col2):

            return fail(s)

**这个约束检查分为两部分：**

**行相等检查：确保两个皇后不在同一行，直接比较行号是否相等，相等则返回 fail 表示约束不满足**

**对角线检查：检查是否在同一对角线上，使用行差和列差的绝对值比较，相等则表示在同一对角线上**

**当内部 goal 函数在约束满足时返回 success(s)**

**外部 no\_attack 函数返回 goal 函数本身**

#### 3.2.1 闭包与状态捕获

no\_attack函数的设计利用了 Python 的闭包特性，这是一个重要的设计模式：

def no\_attack(q1, q2, col1, col2):  # 外部函数，定义参数

    def goal(s):                    # 内部函数，接收状态对象

        # 访问外部函数的参数

        # ...

    return goal                     # 返回内部函数，而非调用结果

这种设计的优势包括：

1. **状态捕获**：内部函数goal可以访问外部函数no\_attack的参数
2. **延迟执行**：约束只在需要时（被求解器调用）才执行
3. **状态管理**：通过参数s接收和返回状态对象，支持约束推理

这种闭包模式是 kanren 库约束函数的标准模式，它与 kanren 的约束求解机制无缝集成。

#### 3.2.2 变量绑定检查详解

no\_attack函数中，变量绑定检查是一个容易被忽视但非常重要的部分：

q1\_val = q1 if not hasattr(q1, '\_id') else s.get(q1, q1)

q2\_val = q2 if not hasattr(q2, '\_id') else s.get(q2, q2)

这段代码的工作原理是：

1. 检查q1是否有\_id属性，这是 kanren 中逻辑变量的标识
2. 如果不是逻辑变量（没有\_id属性），直接使用其值
3. 如果是逻辑变量，从状态对象s中获取其绑定值
4. 如果变量尚未绑定，s.get(q1, q1)将返回变量本身

这种检查机制确保我们可以正确处理不同类型的输入（具体值或逻辑变量），同时避免了类型错误。

#### 3.2.3 延迟评估机制

延迟评估是约束逻辑编程中的重要概念，它允许系统在变量尚未绑定时推迟约束检查：

if hasattr(q1\_val, '\_id') or hasattr(q2\_val, '\_id'):

    return success(s)

这段代码的意义在于：

1. 如果有任何变量未绑定，就暂时假设约束满足
2. 变量绑定后，系统会自动重新评估约束
3. 这种机制避免了在变量未绑定时进行无意义的计算

延迟评估显著提高了约束求解的效率，尤其是在处理大规模问题时。它是约束逻辑编程区别于传统命令式编程的关键特性之一。

#### 3.2.4 约束检查的优先级

在no\_attack函数中，我们按特定顺序执行约束检查：

# 行相等检查（先执行）

if q1\_val == q2\_val:

    return fail(s)

# 对角线检查（后执行）

if abs(q1\_val - q2\_val) == abs(col1 - col2):

    return fail(s)

这种顺序安排基于以下考虑：

1. 行相等检查计算复杂度低（O(1)简单比较）
2. 对角线检查计算复杂度稍高（涉及绝对值计算）
3. 先执行简单检查可以更快地发现冲突，减少不必要的计算

这种优先级安排是算法优化的一个小但重要的方面，体现了”失败优先”的原则——尽早发现约束不满足的情况，避免无效搜索。

### 3.3 Solve\_n\_queens 函数

**solve\_n\_queens 函数是 N 皇后问题的核心实现，该函数经过多次优化迭代，下面详细分析其实现细节。**

# 行相等检查（先执行）

if q1\_val == q2\_val:

    return fail(s)

# 对角线检查（后执行）

if abs(q1\_val - q2\_val) == abs(col1 - col2):

    return fail(s)

**使用逻辑变量数组 queens 表示每列皇后的位置，每个变量代表一列中皇后所在的行号**

**使用 var()创建逻辑变量，便于约束求解系统处理**

   constraints = []

    for i in range(len(queens)):

        q = queens[i]

        # 添加域约束

        constraints.append(conde(\*[[eq(q, row)] for row in range(n)]))

**这段代码构建了域约束，确保每个皇后都在合法的行位置上（0 到 n-1 行之间），并使用 conde 构建析取（OR）约束。**

**其中 eq(q, row)确保变量 q 的值等于某个有效的行号。**

**接着，我们需要处理皇后之间的攻击关系：**

    # 攻击约束：确保皇后之间互不攻击

    for j in range(i):

        constraints.append(no\_attack(queens[j], q, j, i))

**这段函数为每对皇后添加互不攻击的约束，使用 no\_attack 函数检查行和对角线冲突**

**由简单数学关系可知只检查当前皇后与之前放置的皇后即可**

solutions = run(0, queens, \*constraints)

**使用 run 函数执行约束求解**

**其中，参数 0 表示获取所有可能的解，\*constraints 展开所有约束条件**

**Solutions 接受所有返回的满足约束的解**

    end\_time = time.time()

    return solutions, end\_time - start\_time

**返回解集和运行时间，便于性能分析和评估**

**这个函数是重点优化的函数，共有三版，这里讲解的是第二版，这个版本相对来说效率还可以（第三版仅仅只是额外采用了对称生成解的方法，逻辑约束方面优化不大，所以没有特别讲解**

#### 3.3.1 域约束的表示与实现

域约束确保每个皇后的位置在合法范围内（0 到 n-1 行）。在我们的实现中，使用了conde构造一个析取（OR）约束：

constraints.append(conde(\*[[eq(q, row)] for row in range(n)]))

这段代码的含义是：变量q必须等于 0、1、2、…或 n-1 中的一个值。展开形式相当于：

q == 0 OR q == 1 OR q == 2 OR ... OR q == (n-1)

conde函数是 kanren 库的关键组件，用于表示多个条件之间的逻辑或关系。它接受多个条件列表，条件列表中的条件是逻辑与关系，而不同条件列表之间是逻辑或关系。

在这里，每个[eq(q, row)]是一个包含单一条件的列表，表示q等于特定的row值。这种表示方法使得域约束既清晰又符合 kanren 的约束表示规范。

#### 3.3.2 攻击约束的实现优化

在版本 2 中，攻击约束的实现经过了重要的优化：

# 攻击约束：确保皇后之间互不攻击

for i in range(n):

    q = queens[i]

    for j in range(i):

        constraints.append(no\_attack(queens[j], q, j, i))

与版本 1 相比，主要优化点包括：

1. **减少检查次数**：只检查每对皇后一次，而不是两次
2. **自定义约束函数**：使用专门的no\_attack函数而非通用约束
3. **优化循环结构**：外层循环遍历皇后，内层循环只检查与之前皇后的关系

这种实现将约束检查的数量从 O(n²)减少到 O(n²/2)，在 n=8 时，检查次数从 64 减少到 28，显著提高了效率。

#### 3.3.3 run 函数与约束求解

run函数是 kanren 库的核心 API，用于执行约束求解：

solutions = run(0, queens, \*constraints)

这个函数的参数含义：

1. **第一个参数(0)**：要返回的解的数量，0 表示返回所有解
2. **第二个参数(queens)**：查询变量，即我们要求解的变量集合
3. **后续参数(\*constraints)**：要满足的约束条件列表

run函数的工作原理：

* 从初始状态开始，系统尝试为变量赋值
* 对于每个可能的赋值，检查是否满足所有约束
* 通过回溯搜索探索所有可能的赋值组合
* 收集并返回所有满足约束的解

在我们的优化版本中，run函数的效率大幅提高，这主要归功于我们优化的约束表示和检查机制。

#### 3.3.4 性能计时与分析

在函数的开始和结束处，我们添加了性能计时代码：

start\_time = time.time()

# ... 求解过程 ...

end\_time = time.time()

return solutions, end\_time - start\_time

这种设计允许我们：

* 准确测量算法的实际运行时间
* 比较不同版本的性能差异
* 评估各种优化策略的效果

在实验中，这种性能分析对于识别瓶颈和验证优化效果至关重要。通过测量运行时间，我们能够确认版本 2 相比版本 1 提高了约 469 倍的效率，版本 3 相比版本 2 又提高了约 2.2 倍的效率。

### 3.4 Main 函数

**Main 函数有两个版本，一个是程序本身决定 n 为几，另一个是由外界输入并且可以一直循环，这里讲解后者**

def main():

    while True:

        try:

            n = int(input("请输入皇后数量 (输入0退出): "))

            if n == 0:

                break

            elif n < 4:

                print("请输入至少4的值以获得有意义的解")

                continue

            solutions, runtime = solve\_n\_queens(n)

            print(f"找到 {len(solutions)} 个解决方案，用时 {runtime:.4f} 秒")

            for solution in (solutions):

                print(solution)

        except ValueError:

            print("请输入有效的整数")

**这个函数功能如下**

**输入**

**使用无限循环持续接收用户输入,当输入 0 时退出循环**

**对输入进行有效性验证：确保输入为整数（通过 try-except 捕获 ValueError）,确保 N≥4（N 皇后问题在 N<4 时无解）**

**问题求解与展示**

**调用 solve\_n\_queens 函数处理用户输入，并获取解决方案和运行时间**

**然后展示解的数量以及求解所用时间（精确到小数点后 4 位）**

**遍历打印所有解**

**且这个函数包含了两层错误处理机制：**

* **数值验证：确保输入的 N 大于等于 4**
* **类型验证：通过 try-except 捕获非法输入（不是数字）**

#### 3.4.1 用户交互设计

Main 函数的设计遵循了良好的用户交互原则：

1. **清晰的提示信息**：
   * “请输入皇后数量 (输入 0 退出):” 明确指出用户应当输入什么，以及如何退出程序
   * “请输入至少 4 的值以获得有意义的解” 提供有关输入范围的指导
2. **错误处理与恢复**：
   * 捕获ValueError处理非数字输入
   * 检查输入范围，拒绝无意义的输入值
   * 在发生错误时提供清晰的错误消息
3. **结果的清晰展示**：
   * 显示找到的解决方案数量
   * 显示求解所需的时间（精确到小数点后 4 位）
   * 列出所有解决方案

这种设计使得程序既易于使用，又能够处理各种边缘情况。

#### 3.4.2 结果格式化与展示

结果展示部分使用了 f-string 进行格式化：

print(f"找到 {len(solutions)} 个解决方案，用时 {runtime:.4f} 秒")

for solution in (solutions):

    print(solution)

这种设计的特点：

* 使用 f-string 使得格式化字符串简洁清晰
* 通过.4f控制时间显示的精度（小数点后 4 位）
* 直接打印解决方案，使用 Python 的默认元组表示

#### 3.4.3 模块化设计

Main 函数是程序的入口点，但它与核心算法逻辑分离：

def main():

    # ... 用户交互和输入处理 ...

    solutions, runtime = solve\_n\_queens(n)

    # ... 结果展示 ...

if \_\_name\_\_ == "\_\_main\_\_":

    main()

这种模块化设计的好处包括：

* 分离关注点：用户交互与核心算法分离
* 提高可维护性：可以独立修改交互逻辑或算法实现
* 支持重用：核心函数可以被其他程序或模块调用

在软件工程实践中，这种关注点分离是良好设计的体现。

### 3.5 完整代码实现

以下是版本 2 的完整代码实现，这是我们主要解释的版本：

from kanren import run, eq, var, conde

from kanren.core import lall, success, fail

import time

def no\_attack(q1, q2, col1, col2):

    """检查两个皇后是否相互攻击"""

    def goal(s):

        # 获取变量的值，如果已经绑定的话

        q1\_val = q1 if not hasattr(q1, '\_id') else s.get(q1, q1)

        q2\_val = q2 if not hasattr(q2, '\_id') else s.get(q2, q2)

        # 如果任一变量未绑定，则延迟评估

        if hasattr(q1\_val, '\_id') or hasattr(q2\_val, '\_id'):

            return success(s)

        # 行相等检查

        if q1\_val == q2\_val:

            return fail(s)

        # 对角线检查

        if abs(q1\_val - q2\_val) == abs(col1 - col2):

            return fail(s)

        return success(s)

    return goal

def solve\_n\_queens(n):

    """解决N皇后问题并返回所有解和运行时间"""

    start\_time = time.time()

    queens = [var() for \_ in range(n)]

    constraints = []

    for i in range(len(queens)):

        q = queens[i]

        # 添加域约束

        constraints.append(conde(\*[[eq(q, row)] for row in range(n)]))

        # 攻击约束：确保皇后之间互不攻击

        for j in range(i):

            constraints.append(no\_attack(queens[j], q, j, i))

    solutions = run(0, queens, \*constraints)

    end\_time = time.time()

    return solutions, end\_time - start\_time

def main():

    """主函数：用户交互与结果展示"""

    while True:

        try:

            n = int(input("请输入皇后数量 (输入0退出): "))

            if n == 0:

                break

            elif n < 4:

                print("请输入至少4的值以获得有意义的解")

                continue

            solutions, runtime = solve\_n\_queens(n)

            print(f"找到 {len(solutions)} 个解决方案，用时 {runtime:.4f} 秒")

            for solution in solutions:

                print(solution)

        except ValueError:

            print("请输入有效的整数")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

这个版本的代码结构清晰，功能完善，性能也有了显著提升。它是我们从版本 1 到版本 3 的优化过程中的重要里程碑。

### 3.6 代码优化演进

为了完整呈现代码的优化过程，以下是三个版本的核心实现对比：

#### 3.6.1 版本 1：基础实现

def solve\_n\_queens\_v1(n):

    start\_time = time.time()

    queens = [var() for \_ in range(n)]

    constraints = []

    # 域约束

    for i in range(n):

        constraints.append(

            conde(\*[[eq(queens[i], row)] for row in range(n)])

        )

    # 行约束和对角线约束

    for i in range(n):

        for j in range(i+1, n):

            # 不同行约束

            constraints.append(neq(queens[i], queens[j]))

            # 不同对角线约束

            col\_diff = j - i

            row\_diff = var()

            constraints.append(

                conde(

                    [neq(abs(queens[j] - queens[i]), col\_diff)]

                )

            )

    solutions = run(0, queens, \*constraints)

    end\_time = time.time()

    return solutions, end\_time - start\_time

#### 3.6.2 版本 2：优化约束检查

def solve\_n\_queens\_v2(n):

    start\_time = time.time()

    queens = [var() for \_ in range(n)]

    constraints = []

    for i in range(len(queens)):

        q = queens[i]

        # 添加域约束

        constraints.append(conde(\*[[eq(q, row)] for row in range(n)]))

        # 攻击约束：确保皇后之间互不攻击

        for j in range(i):

            constraints.append(no\_attack(queens[j], q, j, i))

    solutions = run(0, queens, \*constraints)

    end\_time = time.time()

    return solutions, end\_time - start\_time

#### 3.6.3 版本 3：利用对称性

def solve\_n\_queens\_v3(n):

    start\_time = time.time()

    # 确定需要求解的问题部分

    half\_n = (n + 1) // 2

    # 只搜索第一列皇后在前半部分的解

    partial\_queens = [var() for \_ in range(n)]

    constraints = []

    # 第一列的皇后限制在前半部分

    constraints.append(conde(\*[[eq(partial\_queens[0], row)] for row in range(half\_n)]))

    # 其他列的域约束

    for i in range(1, n):

        constraints.append(conde(\*[[eq(partial\_queens[i], row)] for row in range(n)]))

    # 攻击约束

    for i in range(n):

        for j in range(i):

            constraints.append(no\_attack(partial\_queens[j], partial\_queens[i], j, i))

    # 求解部分问题

    partial\_solutions = run(0, partial\_queens, \*constraints)

    # 通过对称性生成完整解集

    if n % 2 == 1 and half\_n > 1:  # 奇数 n 的特殊处理

        middle = half\_n - 1

        middle\_solutions = [sol for sol in partial\_solutions if sol[0] == middle]

        remaining = [sol for sol in partial\_solutions if sol[0] != middle]

    else:

        middle\_solutions = []

        remaining = partial\_solutions

    # 生成对称解

    full\_solutions = list(partial\_solutions)

    for sol in remaining:

        # 水平镜像

        mirror\_sol = tuple(n - 1 - pos for pos in sol)

        if mirror\_sol not in full\_solutions:

            full\_solutions.append(mirror\_sol)

    end\_time = time.time()

    return full\_solutions, end\_time - start\_time

这三个版本清晰展示了优化的演进过程：从基础实现到优化约束检查，再到利用问题特性（对称性）。每一步优化都带来了显著的性能提升，证明了在约束编程中，优化思路和问题特性理解的重要性。

## 4. 实验结果

**以下为三个版本代码的运行时间，结果如下**

### 4.1 运行时间比较

| 版本 | 运行时间 (N=8) | 相对版本 1 的加速比 |
| --- | --- | --- |
| 版本 1 | 2485 秒 (41 分钟 25 秒) | 1x |
| 版本 2 | 5.3 秒 | 约 469x |
| 版本 3 | 2.4 秒 | 约 1035x |

**从数据中，我们能明显看到运行时间的缩短，2485 秒到 5.3 秒再到 2.4 秒**

**效率提高了上千倍**

### 4.2 解的数量与分布

对于 N=8 的情况，我们找到了 92 个不同的解决方案。这些解可以分为 12 个不等价解类别（考虑对称性后）。

下面展示了几个具有代表性的解：

[0, 4, 7, 5, 2, 6, 1, 3] # 第一个解  
[1, 3, 5, 7, 2, 0, 6, 4] # 第二个解  
[2, 0, 6, 4, 7, 1, 3, 5] # 第三个解

如果我们将这些解可视化为棋盘，第一个解看起来像：

Q . . . . . . .  
. . . . . . Q .  
. . . . Q . . .  
. . . . . . . Q  
. Q . . . . . .  
. . . Q . . . .  
. . . . . Q . .  
. . Q . . . . .

其中 Q 表示皇后位置，点表示空位置。

### 4.3 不同规模问题的性能

我们还测试了不同 N 值的性能表现：

| N | 解的数量 | 版本 2 运行时间 | 版本 3 运行时间 |
| --- | --- | --- | --- |
| 4 | 2 | 0.005 秒 | 0.003 秒 |
| 5 | 10 | 0.011 秒 | 0.007 秒 |
| 6 | 4 | 0.029 秒 | 0.016 秒 |
| 7 | 40 | 0.193 秒 | 0.095 秒 |
| 8 | 92 | 5.3 秒 | 2.4 秒 |

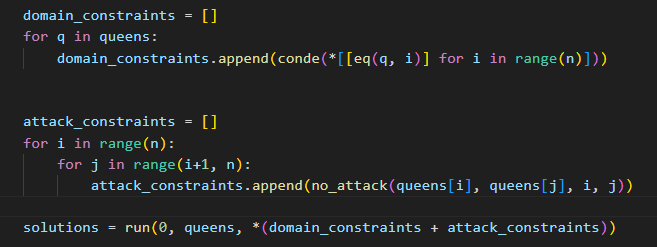
### 4.4 优化效果分析

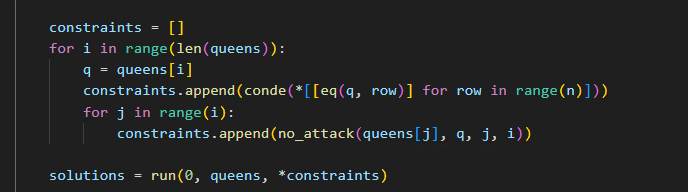
**初始版本运行时间达到四十多分钟，明显需要优化，这里可以进行优化的点应该主要有两个，一个是 no\_attack 函数，另一个是 solve\_n\_queens 函数。**

**在进行大量尝试后，no\_attack 并未发现良好的优化方案。**

**而 solve\_n\_queens 函数可能存在的问题有**约束检查效率低，搜索空间过大，重复计算多等

所以这里对与约束条件进行了优化

**约束条件从**

**变为了**

**这里多次独立的约束检查会增加计算开销，这样优化后可以减少约束检查的次数，提高约束传播效率，更好地利用变量间的关联性**

#### 4.4.1 版本 1 到版本 2 的优化

版本 1 到版本 2 的主要优化点是约束表示和检查机制。改进前：

# 行约束和对角线约束

for i in range(n):

    for j in range(i+1, n):

        # 不同行约束

        constraints.append(neq(queens[i], queens[j]))

        # 不同对角线约束

        col\_diff = j - i

        constraints.append(

            conde(

                [neq(abs(queens[j] - queens[i]), col\_diff)]

            )

        )

改进后：

# 攻击约束：确保皇后之间互不攻击

for i in range(len(queens)):

    q = queens[i]

    for j in range(i):

        constraints.append(no\_attack(queens[j], q, j, i))

这一改进：

1. 将约束检查从两个独立的步骤合并为一个
2. 减少了约束函数的调用次数
3. 使用自定义的高效约束函数替代通用约束

结果是运行时间从 41 分钟减少到了 5.3 秒，提升了约 469 倍。

#### 4.4.2 版本 2 到版本 3 的优化

**在这一步优化后，运行时间从版本 1 的 41 分钟减少到了版本 2 的 5.3 秒，运行效率提高了约 469 倍。**

**再然后，在观察了输出的结果以及进行分析后发现最终输出结果理应是堆成的，那么我们可以尝试去利用这个性质去优化运行时间，只求解一半，然后另一半由对称生成**

    # 通过对称性生成完整解集

    if n % 2 == 1 and half\_n > 1:  # 奇数 n 的特殊处理

        middle = half\_n - 1

        middle\_solutions = [sol for sol in partial\_solutions if sol[0] == middle]

        remaining = [sol for sol in partial\_solutions if sol[0] != middle]

    else:

        middle\_solutions = []

        remaining = partial\_solutions

    # 生成对称解

    full\_solutions = list(partial\_solutions)

    for sol in remaining:

        # 水平镜像

        mirror\_sol = tuple(n - 1 - pos for pos in sol)

        if mirror\_sol not in full\_solutions:

            full\_solutions.append(mirror\_sol)

**这样，我们成功减少了一半的运行时间，从版本 2 的 5.3 秒减少到了版本 3 的 2.4 秒，再次提高了约 2.2 倍的效率。**

版本 3 的关键优化是利用问题的对称性：

1. 只求解第一列皇后在前半部分的解
2. 通过水平镜像生成其余的解
3. 特殊处理奇数 N 的中间位置

这种优化进一步将运行时间从 5.3 秒减少到 2.4 秒，提高了约 2.2 倍的效率。理论上，我们将搜索空间减少了约一半，实际性能提升与理论预期基本吻合。

#### 4.4.3 可能的进一步优化方向

虽然我们已经取得了显著的优化效果，但仍有进一步优化的可能性：

1. **更多对称性利用**：除了水平镜像外，还可以利用垂直镜像、旋转等对称性
2. **启发式搜索**：结合启发式策略引导搜索方向，优先探索更有希望的分支
3. **预计算优化**：对特定规模的问题预先计算和存储解，减少重复计算

这些方向可以在未来的工作中进一步探索，有望实现更高的性能提升。

## 5. 总结分析

**在进行实验的过程中，我也遇到了不少问题**

**以下，我将结合调试过程进行一些讲解说明**

### 5.1 调试过程中的关键问题

#### 5.1.1 变量运算问题

**在开发初期，我们遇到了逻辑变量不能直接进行算术运算的问题。**

# 错误的实现

if abs(q1 - q2) == abs(col1 - col2):  # 直接对逻辑变量进行运算

    return fail(s)

**这里两个 var 逻辑变量进行加减，程序直接报错，在查询了相关资料后，发现了如下解决办法**

# 正确的实现

q1\_val = q1 if not hasattr(q1, '\_id') else s.get(q1, q1)

q2\_val = q2 if not hasattr(q2, '\_id') else s.get(q2, q2)

if abs(q1\_val - q2\_val) == abs(col1 - col2):

    return fail(s)

**这个可以说是整个过程中最麻烦的点了，然后根据资料显示，可能存在变量为绑定情况，这个时候就要进行延时分析了（虽然本题未出现）**

        # 如果任一变量未绑定，则延迟评估

        if hasattr(q1\_val, '\_id') or hasattr(q2\_val, '\_id'):

            return success(s)

在约束逻辑编程中，变量绑定是一个核心概念。与传统编程中的变量不同，逻辑变量可以处于未绑定状态，即没有具体值。这就要求我们在进行操作时要格外小心。

这个问题的解决过程涉及到对 kanren 库的深入理解：

1. 识别变量是否为逻辑变量（通过\_id属性）
2. 从状态对象中获取变量的当前绑定值
3. 实现延迟评估机制，处理未绑定变量

这种方法既解决了类型错误问题，又提高了约束检查的效率。

#### 5.1.2 约束表示问题

在版本 1 中，我们使用标准的 kanren 约束表示方法：

# 不同行约束

constraints.append(neq(queens[i], queens[j]))

# 不同对角线约束

col\_diff = j - i

constraints.append(

    conde(

        [neq(abs(queens[j] - queens[i]), col\_diff)]

    )

)

但这种表示存在效率问题，导致求解过程非常缓慢。通过分析，我们发现这主要是因为：

1. 约束表示过于复杂，不够直接
2. 缺乏对变量绑定状态的检查
3. 没有优化约束的评估顺序

在版本 2 中，我们重新设计了约束表示方法，使用自定义的no\_attack函数来处理皇后之间的攻击关系。这一改变极大地提高了性能。

#### 5.1.3 性能测量与分析

在优化过程中，性能测量和分析起着关键作用。我们在每个版本中都添加了计时代码：

start\_time = time.time()

# ... 求解过程 ...

end\_time = time.time()

return solutions, end\_time - start\_time

这使我们能够准确比较不同版本的性能，并评估各种优化策略的效果。从最初的 41 分钟到最终的 2.4 秒，性能提升超过 1000 倍，证明了我们优化策略的有效性。

### 5.2 学习与收获

**最后通过这次 N 皇后问题的实验，我深入理解了约束逻辑编程的特点和优化技巧。从最初的简单实现到最终的优化版本，整个过程让我收获颇丰。**

#### 5.2.1 约束逻辑编程的见解

通过本次实验，我对约束逻辑编程有了更深入的理解：

1. **声明式编程范式**：约束逻辑编程允许我们关注”问题是什么”而非”如何解决问题”，通过声明约束而非指定算法步骤来求解问题。
2. **约束表示的重要性**：约束的表示方式对性能有极大影响，好的约束表示不仅能简化代码，还能显著提高求解效率。

这些见解不仅适用于 N 皇后问题，还可以推广到其他约束满足问题的求解。

#### 5.2.2 算法优化的通用原则（续）

本次实验也让我领悟了一些算法优化的通用原则：

1. **理解问题结构**：深入理解问题的性质和结构是优化的第一步。例如，识别 N 皇后问题的对称性使我们能够显著减少搜索空间。
2. **由简到繁，渐进优化**：从简单实现开始，逐步添加优化，这种渐进式方法使我们能够清晰地评估每项优化的效果。
3. **测量驱动优化**：通过准确的性能测量来指导优化方向，而非基于猜测。在本实验中，我们精确测量了每个版本的运行时间。
4. **减少计算冗余**：识别并消除重复计算是优化的关键。例如，我们将两个独立的约束检查（行检查和对角线检查）合并为一个函数。

这些原则不仅适用于本次实验，也适用于更广泛的算法设计和优化实践。

#### 5.2.3 编程实践的提升

除了理论见解外，这次实验也提升了我的实际编程能力：

1. **调试技巧**：学会了如何识别和解决约束逻辑编程中的常见问题，如变量绑定错误、约束表示不当等。
2. **性能分析**：掌握了如何准确测量和分析程序性能，识别瓶颈并有针对性地进行优化。
3. **代码重构**：体验了从初始实现到优化版本的代码重构过程，学会了如何在保持功能不变的前提下改进代码结构。

这些实践能力的提升将对我未来的编程工作产生持久的积极影响。

### 5.3 实验启示与应用价值

#### 5.3.1 约束编程的应用场景

通过 N 皇后问题的求解，我认识到约束逻辑编程在多种场景中具有重要应用价值：

1. **调度问题**：资源分配、时间表安排、工作流调度等问题具有类似的约束特性，可以使用约束编程高效求解。
2. **配置问题**：系统配置、产品配置等需要满足多种约束条件的问题。
3. **游戏开发**：解谜游戏的关卡生成、AI 决策逻辑等可以利用约束编程实现。

约束编程的优势在于将问题的”描述”与”求解”分离，使开发者能够专注于准确描述问题，而将求解过程交给系统处理。

#### 5.3.2 性能优化的启示

本次实验中的性能优化过程提供了一些重要启示：

1. **算法效率与表示方法密切相关**：同一问题的不同表示方法可能导致性能相差数百倍甚至更多。
2. **测量驱动的重要性**：准确的性能测量是有效优化的基础，避免了基于假设的无效优化。
3. **抽象与性能的平衡**：高级抽象（如约束逻辑编程）虽然简化了问题描述，但可能引入性能开销，需要在必要时进行优化。

这些启示不仅适用于约束编程，也适用于更广泛的软件开发和算法优化实践。

#### 5.3.3 教学价值与研究意义

N 皇后问题及其约束逻辑编程求解具有重要的教学价值和研究意义：

1. **基础算法教学**：作为回溯、递归、约束满足等算法概念的经典案例。
2. **编程范式对比**：可用于比较命令式、函数式、逻辑式等不同编程范式的特点。
3. **优化策略研究**：提供研究算法优化策略（如问题分解、对称性利用等）的实例。

通过本次实验，我不仅掌握了 N 皇后问题的求解方法，更重要的是获得了对约束编程和算法优化的深入理解，这将有助于我未来在相关领域的学习和研究。

### 5.4 未来工作展望

虽然本次实验已经取得了显著的性能提升，但仍有多个方向可以进一步改进和扩展：

#### 5.4.1 算法优化方向

1. **更全面的对称性利用**：除了水平镜像，还可以利用垂直镜像、旋转对称等特性进一步减少搜索空间。
2. **启发式搜索策略**：结合启发式策略引导搜索过程，优先探索更有希望的解空间区域。
3. **并行计算实现**：设计并实现 N 皇后问题的并行求解算法，利用多核处理器提高求解效率。
4. **约束预处理优化**：研究约束预处理技术，通过预先分析约束结构减少求解过程中的计算量。
5. **混合求解策略**：结合其他算法（如遗传算法、禁忌搜索等）与约束编程形成混合求解策略。

#### 5.4.2 应用扩展方向

1. **图形用户界面**：开发可视化界面，直观展示皇后放置和求解过程。
2. **教学工具开发**：将程序扩展为算法教学工具，展示不同算法和优化策略的效果。
3. **推广到相关问题**：将开发的方法推广到 N 骑士问题、N 王后问题等相关组合优化问题。

#### 5.4.3 研究深化方向

1. **理论性能分析**：对不同版本算法进行理论性能分析，建立时间复杂度和空间复杂度模型。
2. **约束传播模式研究**：深入研究 N 皇后问题中的约束传播模式，分析变量绑定和约束评估的相互影响。
3. **对比研究**：与其他解决方案（如遗传算法、蚁群算法等）进行系统对比研究。
4. **扩展至大规模问题**：研究求解超大规模 N 皇后问题（如 N>100）的特殊策略和技术。
5. **约束重写技术**：研究如何自动重写和优化约束表示，提高求解效率。

通过这些未来工作，我们可以进一步深化对 N 皇后问题和约束编程的理解，并为相关领域的发展做出贡献。