八皇后问题

——AI 课程设计作业 1

实验报告

计算机科学与技术学院

2352018 刘彦

2025 年 3 月 31 日

AI 课程设计作业 1——八皇后问题（逻辑推理实验）

2352018 刘彦

1.问题概述

⑴问题的直观描述

N 皇后问题是一个经典的问题，具体要求是：

- 在 N×N 的棋盘上放置 N 个皇后
- 任意两个皇后不能处于同一行、同一列或同一斜线上
- 要求找出所有满足条件的放置方案

对于 8 皇后问题：

- 使用 8×8 的国际象棋棋盘
- 需要放置 8 个皇后
- 皇后可以攻击同一行、列和对角线上的所有格子

⑵解决问题的思路分析

对于 N 皇后问题约束条件是：任意两个皇后不能在同一行，任意两个皇后不能在同一列，以及任意两个皇后不能在同一对角线上。基于这个约束，我有以下思路：第一，行约束简化：每行必须且只能放置一个皇后，可以按行逐个放置；第二，列约束：需要确保每列最多放置一个皇后；第三，对角线约束：需要检查主对角线和副对角线是否有冲突。

主要解决思路有两个——逻辑编程和回溯法。

对于逻辑编程，基本思路是通过定义变量和约束条件，将问题转化为约束满足问题，最后使用约束求解器找出所有解。优点是代码更加声明式，以及问题描述更直观。

对于回溯法，基本思路是从第一行开始，逐行放置皇后，遇到冲突时返回上一步，尝试新的位置，直到找到所有解决方案。优点是实现简单直观，空间复杂度低，适合小规模问题。但同时还有时间复杂度高和对于大规模问题效率较低的问题。

由于本实验要求是基本掌握逻辑编程的思想，了解逻辑编程与命令式编程的区别；能够依据给定的事实以及规则编写代码，解决逻辑约束问题（CLP）。所以接下来会重点介绍逻辑编程的算法设计与实现。

2.算法设计

⑴算法原理

为了实现约束逻辑编程方法，我将 N 皇后问题转换为约束满足问题(CSP)，使用逻辑变量表示皇后位置，通过定义约束条件来限制解空间。

对于 N 皇后问题，核心约束分别是行约束、列约束和对角线约束。

对于**行约束**，每行只能放置一个皇后，我通过数据结构隐式实现。使用一维数组存储皇后位置，数组索引表示行号，数组值表示列号。由于每个皇后自动被分配到不同行，所以无需显式检查行冲突。数据结构示意如下：

索引(行号):0 1 2 3 4 5

数组值(列):3 5 2 0 1 4

其中，索引 0-5 自动保证了皇后分布在不同行，每个位置存储的是该行皇后所在的列号。

对于**列约束**，每列只能放置一个皇后，我通过变量互不相等约束实现。使用逻辑约束变量表示每个皇后的列位置，通过 not_equalo 约束确保任意两个变量值不相等。

对于**对角线约束**，主对角线：行号+列号相等，副对角线：行号-列号相等，我通过数学关系检查实现。

a.主对角线检查

- 如果两个皇后在同一主对角线上：

$$row1 + col1 = row2 + col2$$

- 约束条件：

$$row1 + col1 \neq row2 + col2$$

b.副对角线检查

- 如果两个皇后在同一副对角线上：

$$row1 - col1 = row2 - col2$$

- 约束条件：

$$row1 - col1 \neq row2 - col2$$

通过这三重约束的配合，可以有效地找出所有合法的放置方案。

⑵算法功能

求解 N×N 棋盘上的 N 皇后问题,找出所有可能的解决方案并按需输出，可视化展示每个解决方案。

⑶算法思路

在初始化阶段，设置棋盘大小 N，创建列值域(0 到 N-1)，初始化 N 个逻辑变量表示皇后。然后分别定义行约束、列约束和对角线约束，合并所有约束并调用约束求解器，获取所

有可行解。

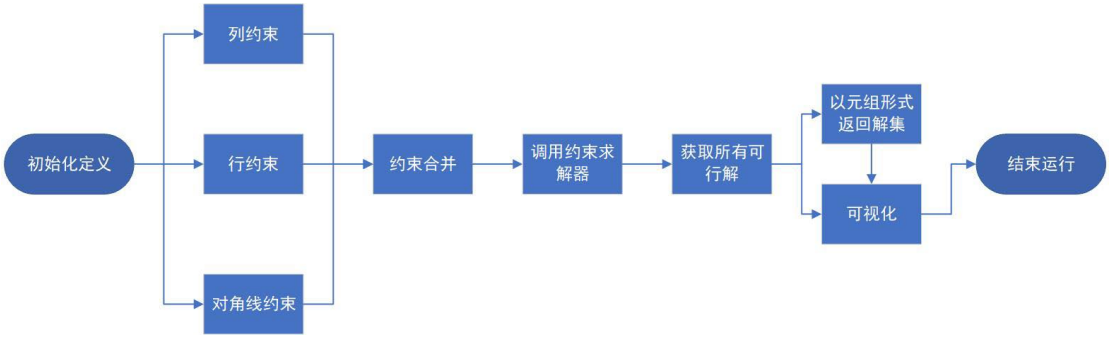


图 1 算法设计流程图

3. 算法实现

⑴初始化实现

```
    def __init__(self, n=8):
        """初始化 N 皇后问题求解器
        Args:
            n (int): 棋盘大小和皇后数量，默认为 8
        """
        self.N = n  # 设置棋盘大小
        self.cols = tuple(range(n))  # 创建列值域(0 到 n-1)
        self.queens = tuple(var() for _ in range(n))  # 创建 n 个逻辑变量表示皇后
```

⑵约束系统实现

```
    def get_constraints(self):
        """生成所有约束条件，包括列范围、列互异和对角线约束
        Returns:
            list: 包含所有约束的列表
        """
        # 约束 1：列范围约束 - 确保每个皇后的列值在合法范围内(0 到 N-1)
        column_constraints = [
            membero(queen, self.cols) for queen in self.queens
        ]

        # 约束 2：列互异约束 - 确保任意两个皇后不在同一列
        # not_equalo 用于确保两个变量的值不相等
        column_unique_constraints = [
            (not_equalo, (self.queens[i], self.queens[j]), True)
            for i in range(self.N)  # 遍历每个皇后
            for j in range(i + 1, self.N)  # 只需要检查 i 之后的皇后
        ]

        # 约束 3：对角线约束 - 确保任意两个皇后不在同一对角线上
```

```
        diagonal_constraints = []
        for i in range(self.N):
            for j in range(i + 1, self.N):
                # 创建 lambda 函数检查对角线冲突
                # i,j 是行号，col1,col2 是列号
                # abs(i-j)是行差，abs(col1-col2)是列差
                # 如果行差等于列差，说明在同一对角线上
                check_diagonal = lambda col1, col2, i=i, j=j: (
                    abs(i - j) != abs(col1 - col2) #检查主对角线和副对角线
                )
                # 使用 goalify 将 lambda 函数转换为逻辑约束
                diagonal_constraints.append(
                    (goalify(check_diagonal), (self.queens[i],
self.queens[j]), True)
                )

        # 合并所有约束条件并返回
        return column_constraints + column_unique_constraints +
diagonal_constraints
```

a. 列范围约束 column_constraints

- 输入：每个皇后变量
- 输出：membero 约束列表
- 实现：使用 membero 约束，确保每个皇后变量的值在 self.cols 中
- 功能：确保每个皇后的列位置在 0 到 N-1 之间，限制皇后只能放在合法的列位置上

b. 列互异约束 column_unique_constraints

- 输入：任意两个皇后变量
- 输出：not_equalo 约束列表
- 实现：使用 membero 约束，确保每个皇后变量的值在 self.cols 中
- 功能：确保任意两个皇后不在同一列

c. 对角线约束 diagonal_constraints

- 输入：任意两个皇后的行号和列变量
- 输出：goalify 转换的对角线约束列表
- 判定条件：$\text{abs}(row1 - row2) \neq \text{abs}(col1 - col2)$
- 实现：主对角线特征：行差 = 列差，副对角线特征：行差 = -列差，使用绝对值统一处理两种情况
- 约束转换：使用 goalify 将普通函数转换为逻辑约束
- 功能：确保任意两个皇后不在同一对角线

⑶求解函数实现

```
    def solve(self):
        """求解 N 皇后问题
        Returns:
            tuple: 所有可能的解决方案
        """
        return run(0, self.queens, *self.get_constraints())
```

run 函数参数说明：第一个参数 0：表示返回所有可能的解决方案（如果是其他数字 n，则只返回前 n 个解），第二个参数 self.queens：要求解的逻辑变量元组，*self.get_constraints()：展开所有约束条件列表。

工作流程：调用 get_constraints()获取所有约束条件，使用星号操作符（*）展开约束条件列表，run 函数执行约束求解，返回满足所有约束的解决方案元组，每个解决方案：包含 N 个整数的元组，表示每行皇后的列位置。

⑷可视化函数实现

```
    def visualize_solution(self, solution):
        """将解决方案可视化为棋盘样式
        Args:
            solution (tuple): 一个解决方案，表示每行皇后的列位置
        Returns:
            str: 可视化的棋盘字符串
        """
        board = []
        for i in range(self.N):
            row = ['□'] * self.N  # 创建空棋盘行
            row[solution[i]] = '♕'  # 在对应位置放置皇后
            board.append(' '.join(row))
        return '\n'.join(board)
```

这个方法将 N 皇后问题的一个解决方案在命令行中可视化为棋盘样式的字符串表示。

字符选择：□：表示空位；♕：表示皇后位置。格式化：使用空格分隔每列。使用换行符分隔每行，形成清晰的视觉效果。

输出示例：对于 6×6 的棋盘，一个可能的输出：

□ □ ♕ □ □ □

□ □ □ □ ♕ □

♕ □ □ □ □ □

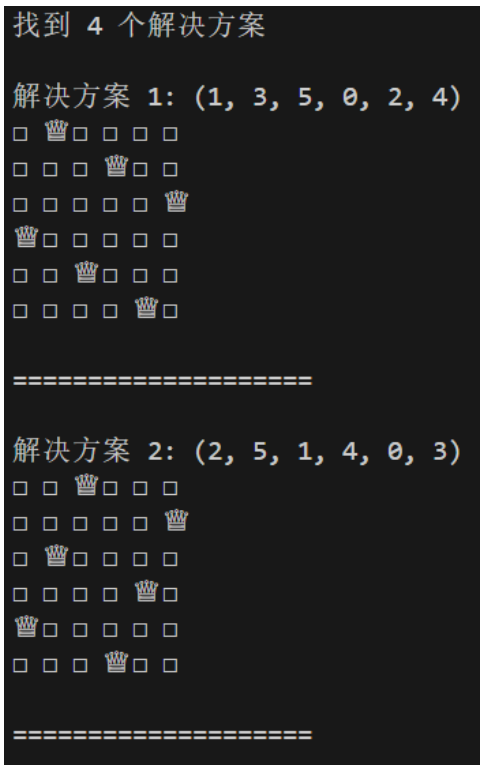
□ □ □ ♕ □ □

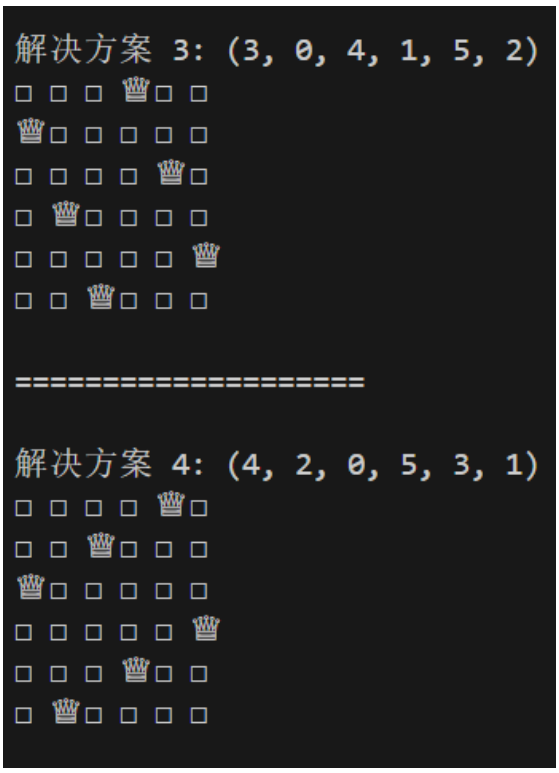
□ ♕ □ □ □ □

□ □ □ □ □ ♕

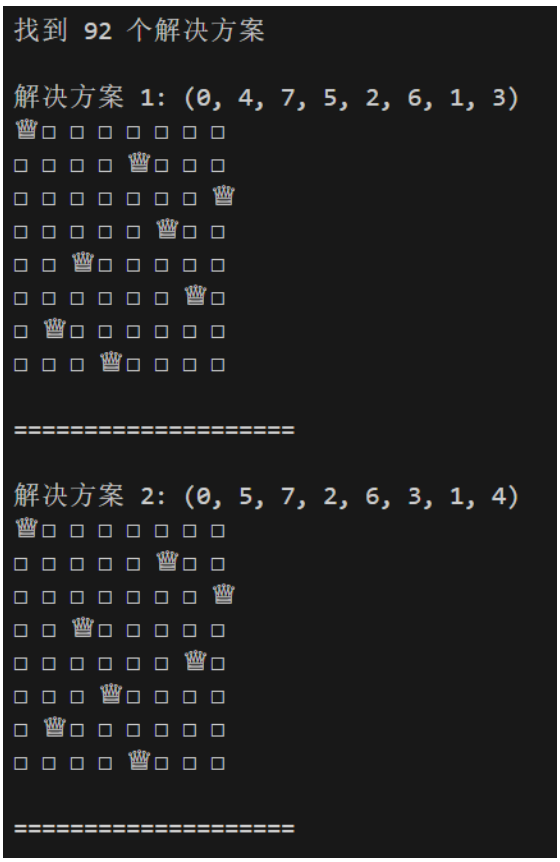
4. 实验结果

当 N=6 时，一共有 4 个不同结果：





当 N=8 时，一共有 92 个不同结果,这里可视化以前两个为例：



当 N=8 时，一共有 92 个不同结果,这里展示所有解决方案：

```
找到 92 个解决方案
解决方案 1: (0, 4, 7, 5, 2, 6, 1, 3)
解决方案 2: (0, 5, 7, 2, 6, 3, 1, 4)
解决方案 3: (0, 6, 3, 5, 7, 1, 4, 2)
解决方案 4: (0, 6, 4, 7, 1, 3, 5, 2)
解决方案 5: (1, 3, 5, 7, 2, 0, 6, 4)
解决方案 6: (1, 4, 6, 0, 2, 7, 5, 3)
解决方案 7: (1, 4, 6, 3, 0, 7, 5, 2)
解决方案 8: (1, 5, 0, 6, 3, 7, 2, 4)
解决方案 9: (1, 5, 7, 2, 0, 3, 6, 4)
解决方案 10: (1, 6, 2, 5, 7, 4, 0, 3)
解决方案 11: (1, 6, 4, 7, 0, 3, 5, 2)
解决方案 12: (1, 7, 5, 0, 2, 4, 6, 3)
解决方案 13: (2, 0, 6, 4, 7, 1, 3, 5)
解决方案 14: (2, 4, 1, 7, 0, 6, 3, 5)
解决方案 15: (2, 4, 1, 7, 5, 3, 6, 0)
解决方案 16: (2, 4, 6, 0, 3, 1, 7, 5)
解决方案 17: (2, 4, 7, 3, 0, 6, 1, 5)
解决方案 18: (2, 5, 1, 4, 7, 0, 6, 3)
解决方案 19: (2, 5, 1, 6, 0, 3, 7, 4)
解决方案 20: (2, 5, 1, 6, 4, 0, 7, 3)
解决方案 21: (2, 5, 3, 0, 7, 4, 6, 1)
解决方案 22: (2, 5, 3, 1, 7, 4, 6, 0)
解决方案 23: (2, 5, 7, 0, 3, 6, 4, 1)
解决方案 24: (2, 5, 7, 0, 4, 6, 1, 3)
解决方案 25: (2, 5, 7, 1, 3, 0, 6, 4)
解决方案 26: (2, 6, 1, 7, 4, 0, 3, 5)
解决方案 27: (2, 6, 1, 7, 5, 3, 0, 4)
解决方案 28: (2, 7, 3, 6, 0, 5, 1, 4)
解决方案 29: (3, 0, 4, 7, 1, 6, 2, 5)
解决方案 30: (3, 0, 4, 7, 5, 2, 6, 1)
解决方案 31: (3, 1, 4, 7, 5, 0, 2, 6)
解决方案 32: (3, 1, 6, 2, 5, 7, 0, 4)
解决方案 33: (3, 1, 6, 2, 5, 7, 4, 0)
解决方案 34: (3, 1, 6, 4, 0, 7, 5, 2)
解决方案 35: (3, 1, 7, 4, 6, 0, 2, 5)
解决方案 36: (3, 1, 7, 5, 0, 2, 4, 6)
解决方案 37: (3, 5, 0, 4, 1, 7, 2, 6)
解决方案 38: (3, 5, 7, 1, 6, 0, 2, 4)
解决方案 39: (3, 5, 7, 2, 0, 6, 4, 1)
解决方案 40: (3, 6, 0, 7, 4, 1, 5, 2)
解决方案 41: (3, 6, 2, 7, 1, 4, 0, 5)
解决方案 42: (3, 6, 4, 1, 5, 0, 2, 7)
解决方案 43: (3, 6, 4, 2, 0, 5, 7, 1)
解决方案 44: (3, 7, 0, 2, 5, 1, 6, 4)
解决方案 45: (3, 7, 0, 4, 6, 1, 5, 2)
解决方案 46: (3, 7, 4, 2, 0, 6, 1, 5)
解决方案 47: (4, 0, 3, 5, 7, 1, 6, 2)
解决方案 48: (4, 0, 7, 3, 1, 6, 2, 5)
解决方案 49: (4, 0, 7, 5, 2, 6, 1, 3)
解决方案 50: (4, 1, 3, 5, 7, 2, 0, 6)
解决方案 51: (4, 1, 3, 6, 2, 7, 5, 0)
解决方案 52: (4, 1, 5, 0, 6, 3, 7, 2)
解决方案 53: (4, 1, 7, 0, 3, 6, 2, 5)
解决方案 54: (4, 2, 0, 5, 7, 1, 3, 6)
解决方案 55: (4, 2, 0, 6, 1, 7, 5, 3)
解决方案 56: (4, 2, 7, 3, 6, 0, 5, 1)
解决方案 57: (4, 6, 0, 2, 7, 5, 3, 1)
解决方案 58: (4, 6, 0, 3, 1, 7, 5, 2)
解决方案 59: (4, 6, 1, 3, 7, 0, 2, 5)
解决方案 60: (4, 6, 1, 5, 2, 0, 3, 7)
解决方案 61: (4, 6, 1, 5, 2, 0, 7, 3)
解决方案 62: (4, 6, 3, 0, 2, 7, 5, 1)
解决方案 63: (4, 7, 3, 0, 2, 5, 1, 6)
解决方案 64: (4, 7, 3, 0, 6, 1, 5, 2)
解决方案 65: (5, 0, 4, 1, 7, 2, 6, 3)
解决方案 66: (5, 1, 6, 0, 2, 4, 7, 3)
解决方案 67: (5, 1, 6, 0, 3, 7, 4, 2)
解决方案 68: (5, 2, 0, 6, 4, 7, 1, 3)
解决方案 69: (5, 2, 0, 7, 3, 1, 6, 4)
解决方案 70: (5, 2, 0, 7, 4, 1, 3, 6)
解决方案 71: (5, 2, 4, 6, 0, 3, 1, 7)
解决方案 72: (5, 2, 4, 7, 0, 3, 1, 6)
解决方案 73: (5, 2, 6, 1, 3, 7, 0, 4)
解决方案 74: (5, 2, 6, 1, 7, 4, 0, 3)
解决方案 75: (5, 2, 6, 3, 0, 7, 1, 4)
解决方案 76: (5, 3, 0, 4, 7, 1, 6, 2)
解决方案 77: (5, 3, 1, 7, 4, 6, 0, 2)
解决方案 78: (5, 3, 6, 0, 2, 4, 1, 7)
解决方案 79: (5, 3, 6, 0, 7, 1, 4, 2)
解决方案 80: (5, 7, 1, 3, 0, 6, 4, 2)
解决方案 81: (6, 0, 2, 7, 5, 3, 1, 4)
解决方案 82: (6, 1, 3, 0, 7, 4, 2, 5)
解决方案 83: (6, 1, 5, 2, 0, 3, 7, 4)
解决方案 84: (6, 2, 0, 5, 7, 4, 1, 3)
解决方案 85: (6, 2, 7, 1, 4, 0, 5, 3)
解决方案 86: (6, 3, 1, 4, 7, 0, 2, 5)
解决方案 87: (6, 3, 1, 7, 5, 0, 2, 4)
解决方案 88: (6, 4, 2, 0, 5, 7, 1, 3)
解决方案 89: (7, 1, 3, 0, 6, 4, 2, 5)
解决方案 90: (7, 1, 4, 2, 0, 6, 3, 5)
解决方案 91: (7, 2, 0, 5, 1, 4, 6, 3)
解决方案 92: (7, 3, 0, 2, 5, 1, 6, 4)
```

在 N 皇后问题中，解决方案表示在棋盘上皇后的放置位置。这串数字的含义如下：索引(行号)：表示棋盘的行，从 0 开始；值(列号)：表示该行皇后所在的列，从 0 开始。

对于八皇后问题的第一个解决方案(0, 4, 7, 5, 2, 6, 1, 3)，索引(行) -> 值(列)的对应关系：

0 行 -> 0 列

1 行 -> 4 列

2 行 -> 7 列

3 行 -> 5 列

4 行 -> 2 列

5 行 -> 6 列

6 行 -> 1 列

7 行 -> 3 列

其余同理。这种表示方法非常紧凑和高效，只需要一个长度为 N 的数组就能表示完整的棋盘状态，行号通过索引隐含表示，不需要额外存储。

5. 总结与分析

(1)对算法的理解和评价

算法核心特点是：声明式编程范式、约束自动传播和基于逻辑变量的求解。

算法优缺点评价

a. 优点

- 代码简洁，问题描述直观，易于理解和修改
- 能找到所有解，不会遗漏可行解，解的正确性有保证
- 易于添加新约束，支持问题变体，逻辑清晰

b. 缺点

- 性能限制，N 较大时计算慢，内存消耗较大，约束检查开销大
- 依赖 kanren 库，而 kanren 库本身十分老旧并且效率非常低，导致逻辑编程本身推理速度十分缓慢，对于八皇后问题所需的推理时间非常长。
- 错误信息模糊，约束冲突难查，性能优化困难

适用场景分析

- 适合场景:在教学演示，算法原理展示，逻辑编程教学等对于效率和具体结果要求不

高的场景；以及面对小规模问题，例如 N 值较小（N ≤ 6）时。

- 不适合场景：存在大规模计算，例如 N 值很大，且对效率要求高，资源受限时。

⑵调试中遇到的问题

①Var 类型操作问题带来 TypeError 错误

在一开始的代码中，我试图对逻辑变量直接进行算术运算。曾经的代码实现方式如下：

```
# 定义对角线约束
def not_same_diagonal(i, j, col1, col2):
    diff = abs(i - j)  # 计算行索引之差
    return conde(
        (eq, col1, col2), fail,  # 排除同列情况
        (eq, col1, col2 + diff), fail,  # 右下对角线
        (eq, col1, col2 - diff), fail   # 左下对角线
    )
```

会出现报错：

```
Traceback (most recent call last):
  File "c:\Users\PC\Desktop\ai\test.py", line 22, in <module>
    solutions = run(0, q,  # 0 表示返回所有解
  File "c:\Users\PC\Desktop\ai\test.py", line 25, in <genexpr>
    *(not_same_diagonal(i, j, q[i], q[j]) for i in range(N) for j in range(i + 1, N))  # 不能在对角线上
  File "c:\Users\PC\Desktop\ai\test.py", line 17, in not_same_diagonal
    (eq, col1, col2 + diff), fail,  # 右下对角线
TypeError: unsupported operand type(s) for +: 'Var' and 'int'
```

因为 col1 和 col2 是 Var 类型（逻辑变量），diff 是 int 类型。直接进行 col2 和 diff 相加的操作会导致类型错误。为了解决这个问题，在代码中，我通过以下方式避免了直接对 Var 类型进行操作，具体修改见“**3.算法实现**”部分。

a.对角线约束的处理

- 不直接对 col1 和 col2 进行算术运算
- 使用比较操作而不是加减运算
- 将行号 i,j 作为固定值传入

b. 使用 goalify 函数转换

- goalify 自动处理 Var 类型的操作
- 避免手动处理类型转换
- 确保约束条件正确执行

c.使用内置约束函数，避免算术运算

- membero：处理成员关系
- not_equalo：处理不等关系
- 这些函数内部已处理 Var 类型

②使用 kanren.lall 函数时出现 EarlyGoalError 错误

lall 是 kanren 库中的一个重要函数，用于组合多个约束条件。基本功能是将多个约束条件组合成一个单一的约束，确保所有约束条件同时满足（逻辑 AND），类似于将所有约束放在一个列表中。

曾经的代码实现方式：

```
# 组合所有约束条件
constraints = lall(
    # 约束 1：每个皇后必须在一个有效的列中
    *[membero(queens[i], cols) for i in range(N)],
    # 约束 2：任意两个皇后不能在同一列
    *[not_equalo(queens[i], queens[j]) for i in range(N) for j in
range(i + 1, N)],
    # 约束 3：任意两个皇后不能在同一对角线上
    *[goalify(not_same_diagonal)(queens[i], queens[j], i, j) for i in
range(N) for j in range(i + 1, N)],
)
```

会出现报错：

```
Traceback (most recent call last):
  File "c:\Users\PC\Desktop\ai\test.py", line 23, in <module>
    *[not_equalo(queens[i], queens[j]) for i in range(N) for j in range(i + 1, N)],
  File "c:\Users\PC\Desktop\ai\test.py", line 23, in <listcomp>
    *[not_equalo(queens[i], queens[j]) for i in range(N) for j in range(i + 1, N)],
  File "D:\Anaconda\envs\env_py38\lib\site-packages\kanren\goals.py", line 157, in funco
    raise EarlyGoalError()
kanren.core.EarlyGoalError
```

经查询资料，EarlyGoalError 错误通常发生在以下情况：约束条件的组合方式不正确，和使 lall 函数时约束条件的展开方式有误。后来对代码进行修改，主要修改点是：移除了 lall 的使用，直接使用约束列表，修改了约束条件的组合方式，以及将约束条件分开构建，然后在 run 函数中合并，具体修改见**“3.算法实现”**部分。

这个修改避免了 kanren.lall 函数带来的问题，并且使约束条件更容易组织，约束冲突更容易调试，异常信息更明确，最终达到了目的，程序没有报错。

⑶算法改进与其他方法的尝试

①算法改进总结

约束条件模块化，优点是约束条件分类明确，维护和调试方便，代码结构清晰，便于扩展和修改。

```
def get_constraints(self):
    # 将约束分为三类，便于管理和维护
    column_constraints = [...]      # 列范围约束
    column_unique_constraints = [...]  # 列互异约束
    diagonal_constraints = [...]    # 对角线约束
    return column_constraints + column_unique_constraints +
diagonal_constraints
```

对代码结构进行改进，对函数实现类封装，可以同时创建不同规模的问题，集中管理问题状态，实例变量共享数据，保证数据一致性。

```
class NQueens:
    def get_constraints(self): ...  # 生成约束
    def solve(self): ...            # 求解问题
    def visualize_solution(self): ...  # 可视化结果
```

②对其他方法的尝试

a. 利用 itertools.permutations 生成所有可能的皇后位置排列

在代码中使用 itertools.permutations 生成所有可能的皇后排列，使用 kanren 库的逻辑约束编程功能进行解的筛选。利用了逻辑约束编程的灵活性，实现相对简单。在 N 较小时，这个方法直接使用排列筛选，比只使用 kanren 库的运算速度更快（这可能与 kanren 库对 N 个独立变量分层定义约束效率较低有关）。

```
# 生成所有可能的皇后排列
all_permutations = tuple(permutations(range(N)))
# 使用 permutations 直接生成候选解
membero(queens, all_permutations)  # 约束 queens 必须是合法排列
```

b. 利用回溯算法

回溯算法的核心思想体现在 solve_recursive 函数中：通过逐行放置皇后的方式构建解决方案，使用 is_safe 函数验证每个放置位置的合法性，当找到一个合法解时，将其添加到解决方案列表中，通过递归方式继续探索下一行的可能位置。但是这个实现方式是命令式编程，并不是这次实验要求的逻辑编程，因此不详细描述。

该算法过回溯算法避免了暴力枚举所有可能性，比使用 kanren 进行逻辑编程效率更高（这可能与 kanren 库对 N 个独立变量分层定义约束效率较低有关）。下面给出关键部分伪代码：

```
function solve_recursive(board, row):
    if row equals N:
        add board state to solutions
        return
    for col from 0 to N-1:
        if is_safe(board, row, col):
            board[row] = col
            solve_recursive(board, row + 1)
```

⑷实验收获

通过本次实验，我深入理解了逻辑编程的特点，学会将具体问题转化为约束条件。并且学习了 kanren 库的约束求解机制：使用 var()创建逻辑变量，使用 membero 定义域约束，使用 goalify 转换普通函数为逻辑约束。

在本次实验后，我深入理解了约束逻辑编程，提升了问题建模能力，掌握了新的编程范式，积累了实践经验。接下来我也会尝试其他更先进的 python 逻辑编程库（如 prolog），进一步提高逻辑编程能力。