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

①实验内容

八皇后问题：如何能够在 8×8 的国际象棋棋盘上放置八个皇后，使得任何一个皇后都无法直接吃掉其他的皇后？为了达到此目的，任两个皇后都不能处于同一条横行、纵行或斜线上。

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

②实验要求

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。

③实验背景

逻辑编程是一种编程典范，它设置答案须匹配的规则来解决问题，而非设置步骤来解决问题。过程是“事实+规则=结果”。人工智能的发展与逻辑编程的发展是一个相辅相成的过程，早期的人工智能以规则和逻辑推理作为主要研究方向，这在逻辑编程的发展中发挥了重要的影响，另外更好更快的逻辑编程也推动了人工智能的发展，例如专家系统、知识图谱和自动定理证明。

Python 是一种解释型、面向对象、动态数据类型的高级程序设计语言。在数据驱动学习时代，Python 的崛起已经是一个不争的事实，并且成为人工智能算法的第一语言。在本次实验中，我们学习将 Python 应用于逻辑编程，并尝试自主撰写逻辑规则解决八皇后问题。

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。

逻辑编程是一种基于数学逻辑的编程范式，核心思想是通过定义变量及其约束条件，将问题建模为约束满足问题（CSP, Constraint Satisfaction Problem），然后利用约束求解器（Constraint Solver）来自动寻找满足所有约束的解。与命令式编程不同，逻辑编程不直接指明如何求解问题，而是声明式地描述问题，使代码更加直观和可读。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（CLP, Constraint Logic Programming）方法，我将 N 皇后问题(N-Queens Problem)转换为一个约束满足问题(CSP, Constraint Satisfaction Problem)，其中逻辑变量用于表示皇后的位置，约束条件用于限制解空间，从而确保所有放置的皇后互不攻击。通过这种方式，问题被建模为一组变量及其约束关系，最终借助约束求解器（Constraint Solver）来寻找所有符合条件的解。

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**，在求解 N 皇后问题时，为了满足列约束（即每列只能放置一个皇后），我们采用 变量互不相等约束 来确保皇后不会出现在同一列上。具体来说，我们使用逻辑约束变量表示每个皇后在棋盘上的列位置，并利用 not_equalo 约束确保所有变量的值互不相等。

在具体实现中，我们将棋盘上的 N 个皇后位置视为 N 个逻辑变量，其中每个变量 Q_i 代表第 i 行的皇后所处的列号。由于 N 皇后问题要求每个皇后必须处于不同列，我们直接在 kanren 逻辑编程框架中添加约束，确保所有 Q_i 互不相等，即 Q_1 ≠ Q_2 ≠ ... ≠ Q_N。这种方法相当于在排列 range(N)的基础上进行筛选，确保列号不会重复，从而避免显式检查列冲突。

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。

通过约束逻辑编程，我们能够以更高层次的抽象来建模 N 皇后问题，而无需手写回溯算法。逻辑变量、约束传播、自动求解器等特性使得该方法更具直观性和可扩展性，适用于求解各种组合优化问题

⑵算法功能

本算法旨在求解 N×N 棋盘上的 N 皇后问题，即在 N×N 的棋盘上放置 N 个皇后，使得任意两个皇后都不会相互攻击。本算法的主要功能包括：计算所有可能的解，采用约束逻辑编程（CLP, Constraint Logic Programming）或回溯搜索的方法，确保找到所有满足条件的解；通过列约束、对角线约束等规则限制搜索空间，提高求解效率。

算法还支持不同规模的 N 值，允许用户指定 N 值，求解 N 皇后问题，例如 N=4、N=8、N=10 等，适应不同需求。

最终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。

⑶kanren 库的逻辑范式编程介绍

Kanren 是一个强大而灵活的工具，通过逻辑范式编程，它允许用户以声明式的方式定义问题并求解。它特别适合那些需要推理、关系建模或约束求解的任务。

①var()创建逻辑变量

作用：var()用于创建逻辑变量（Logic Variable），它是逻辑编程中的占位符，表示未知的值。逻辑变量通过推理过程可以被绑定到具体值。

用法：var()不需要参数，直接调用即可生成一个唯一的逻辑变量。

```
from kanren import var
x = var()  # 创建一个逻辑变量
print(x)   # 输出类似: ~_0（表示一个未绑定的变量）
```

②eq(a,b)相等性目标

作用：eq(a,b)定义一个目标（Goal），表示 a 和 b 必须相等。它是 unification 的基础。

用法：eq(a,b)接受两个参数，可以是逻辑变量、常量或其他表达式。

```
from kanren import run, eq, var
x = var()
results = run(1, x, eq(x, 5))  # 查询 x，使得 x = 5
print(results)  # 输出: (5,)
```

③run(n,x,*goals)执行查询

作用:run() 是 kanren 的核心函数,用于运行逻辑查询并返回结果。它将目标(goals)应用到逻辑变量上，求解满足条件的解。

参数：

- n：返回结果的最大数量，0 表示返回所有可能结果。
- x：要查询的变量（可以是单个变量或变量列表）。
- *goals：一个或多个目标，表示需要满足的条件。

用法：run(n, x, goal1, goal2, ...)。

```
from kanren import run, eq, var
x = var()
results = run(0, x, eq(x, 1))  # 返回所有解
print(results)  # 输出: (1,)
```

④Relation()和 facts()定义关系和事实

作用：

- Relation()：创建一个关系（类似于数据库中的表），用于表示事实之间的联系。
- facts()：向关系中添加具体的事实。

用法：先用 Relation()创建关系对象。然后用 facts(relation, *tuples)添加事实。

```
from kanren import Relation, facts, run, var
parent = Relation()
facts(parent,
      ("Alice", "Bob"),    # Alice 是 Bob 的父母
      ("Bob", "Charlie")) # Bob 是 Charlie 的父母
x = var()
results = run(0, x, parent(x, "Charlie"))  # 查询 Charlie 的父母
print(results)  # 输出: ('Bob',)
```

⑤conde(*clauses)逻辑“或”

作用：conde 表示多个条件之间的“或”（OR）关系，允许定义多个可能的目标分支。

用法：conde([goal1], [goal2], ...)，每个分支是一个目标列表，表示“与”（AND）关系。

```
from kanren import run, var, eq, conde
x = var()
goal = conde([eq(x, 1)], [eq(x, 2)])  # x = 1 或 x = 2
results = run(0, x, goal)
print(results)  # 输出: (1, 2)
```

⑥lall(*goals)逻辑“与”

作用：lall 表示多个目标之间的“与”（AND）关系，所有目标必须同时满足。

用法：lall(goal1, goal2, ...)。

```
from kanren import run, var, eq, lall
x, y = var(), var()
goal = lall(eq(x, 1), eq(y, 2))  # x = 1 且 y = 2
results = run(0, [x, y], goal)
print(results)  # 输出: ([1, 2],)
```

⑦membero(x,coll)成员关系

作用：membero(x, coll) 检查变量 x 是否是集合 coll 的成员。

用法：coll 是一个可迭代对象（如列表或元组）。

```
from kanren import run, var, membero
x = var()
results = run(0, x, membero(x, (1, 2, 3)))  # x 是 (1, 2, 3) 中的成员
print(results)  # 输出: (1, 2, 3)
```

⑧add(x, y, z)等算术关系（需扩展）

作用：kanren 本身不直接提供内置算术关系，但可以通过扩展实现加法、乘法等。

用法：需要自定义关系或使用扩展库（如 kanren.arith）。

```
from kanren.arith import add  # 注意：需自行实现或使用扩展
x, y, z = var(), var(), var()
results = run(0, x, eq(z, 5), add(x, y, z), eq(y, 2))
print(results)  # 输出: (3,)
```

⑨not_equalo(x,y)不相等目标

作用：not_equalo(x,y) 定义一个目标（Goal），表示逻辑变量或值 x 和 y 必须不相等。它是逻辑编程中约束的一种，用于排除某些解。

来源：not_equalo 通常位于 kanren.goals 模块中，专门用于处理约束条件。

用法：not_equalo(x,y)接受两个参数，可以是逻辑变量、常量或它们的组合。

```
from kanren import run, var, eq, not_equalo
from kanren.goals import nequalo
x = var()
goals = (eq(x, 5), not_equalo(x, 3))  # x = 5 且 x ≠ 3
results = run(0, x, goals)
print(results)  # 输出: (5,)
```

⑩goalify(func)将函数转化为目标

作用：goalify(func)是一个工具函数，用于将普通的 Python 函数转化为 kanren 的目标（Goal）。这允许用户将自定义逻辑嵌入到逻辑编程框架中。

来源：goalify 通常位于 kanren.goals 或相关模块中，是 kanren 扩展性的体现。

用法：

- 输入：一个普通的 Python 函数（通常接受多个参数并返回布尔值）。

● 输出：一个可以作为 kanren 目标的函数，接受相同数量的参数。

```
from kanren import run, var, goalify, membero
# 定义一个普通 Python 函数
def is_even(x):
    return x % 2 == 0
# 将其转化为目标
even_goal = goalify(is_even)
x = var()
results = run(0, x, even_goal(x), (membero, x, (1, 2, 3, 4)))  # x 是偶数且属于 (1, 2, 3, 4)
print(results)  # 输出: (2, 4)
```

⑷算法思路

在初始化阶段，我首先确定棋盘大小 N，并创建列值域从 0 到 N-1 作为皇后可能的列位置，然后通过 kanren 的 var()初始化 N 个逻辑变量分别表示每行皇后的列号；接着，我定义列约束以确保所有皇后位于不同列，使用对角线约束防止皇后出现在同一对角线上，最后将这些约束合并为一个目标集合，调用 run 函数执行约束求解，获取所有满足条件的解。

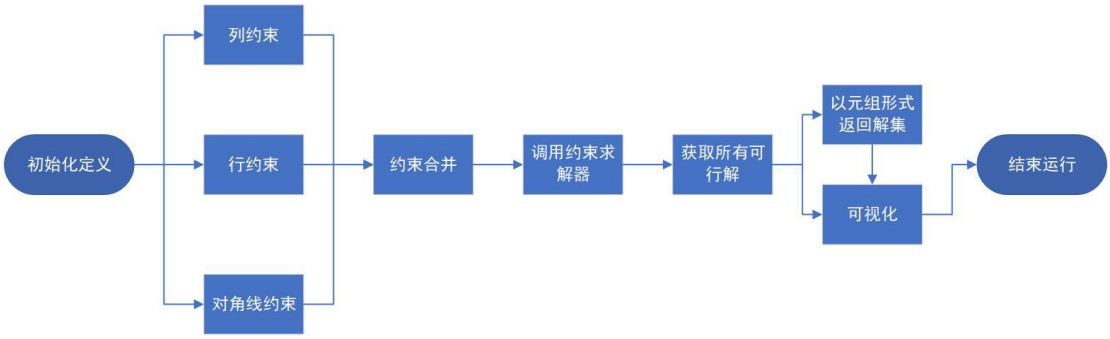


图 1 算法设计流程图

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)
```

这个方法旨在将 N 皇后问题的一个解决方案以直观的方式呈现出来，通过在命令行中生成棋盘样式的字符串表示，让用户能够轻松理解和验证结果。在实现过程中，我精心设计了字符选择和格式化规则，以确保输出的可视化效果既清晰又美观。具体来说，我选用了两种字符来表示棋盘上的不同状态：□（方框）代表空位，表示该位置没有放置皇后；♕（皇后符号）则标记皇后的具体位置，这种符号选择不仅直观，还带有一定的象形意义，便于快速识别。为了让棋盘在命令行中易于阅读，我在格式化时加入了空格来分隔每一列的字符，同时使用换行符分隔每一行，从而形成一个规整的网格结构。这种排版方式有效地模拟了真实棋盘的视觉效果，使得用户能够一目了然地看到每个皇后的分布情况。

在实现这一可视化方法时，我首先从 N 皇后问题的求解结果中获取一个具体的解决方案，通常是一个长度为 N 的列表，其中每个元素表示对应行的皇后所在列号。接着，我根据棋盘大小 N 构建一个 N×N 的字符串表示：对于每一行，我遍历所有列位置，若当前列号与该行皇后的列号相符，则放置 ♕，否则放置 □；然后用空格连接该行的所有字符，并通过换行符将每一行拼接成完整的棋盘字符串。这种方法不仅简单高效，还能灵活适应不同大小的棋盘。

以下是一个具体的输出示例，以 8×8 棋盘为例，假设求解结果为 [0, 4, 7, 5, 2, 6, 1, 3]（表示第 0 行的皇后在第 0 列，第 1 行的皇后在第 4 列，依次类推），可视化结果如下：

♕ □ □ □ □ □ □ □

□ □ □ □ ♕ □ □ □

□ □ □ □ □ □ □ ♕

□ □ □ □ □ ♕ □ □

□ □ ♕ □ □ □ □ □

□ □ □ □ □ □ ♕ □

□ ♕ □ □ □ □ □ □

□ □ □ ♕ □ □ □ □

⑸主函数实现

主函数 main()是整个 N 皇后问题求解程序的入口，它负责协调问题的实例化、求解以及结果的可视化展示。通过清晰的结构设计和模块化的调用方式，主函数将逻辑编程的核心

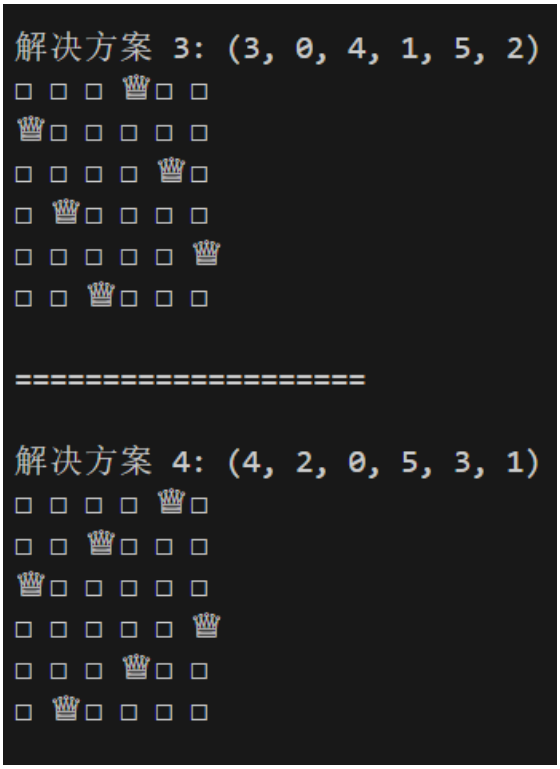
功能与用户友好的输出展示结合起来，为用户提供了一个直观的交互体验。以下是对其实现过程的详细说明和扩写：

```
def main():
    """程序入口函数"""
    # 创建 N 皇后问题实例
    n_queens = NQueens(6)
    solutions = n_queens.solve()

    # 打印结果
    print(f"找到 {len(solutions)} 个解决方案")

    # 遍历并展示每个解决方案
    for idx, solution in enumerate(solutions, 1):
        print(f"\n 解决方案 {idx}: {solution}")
        print(n_queens.visualize_solution(solution))
        print("\n" + "="*20)

# 程序入口
if __name__ == "__main__":
    main()
```

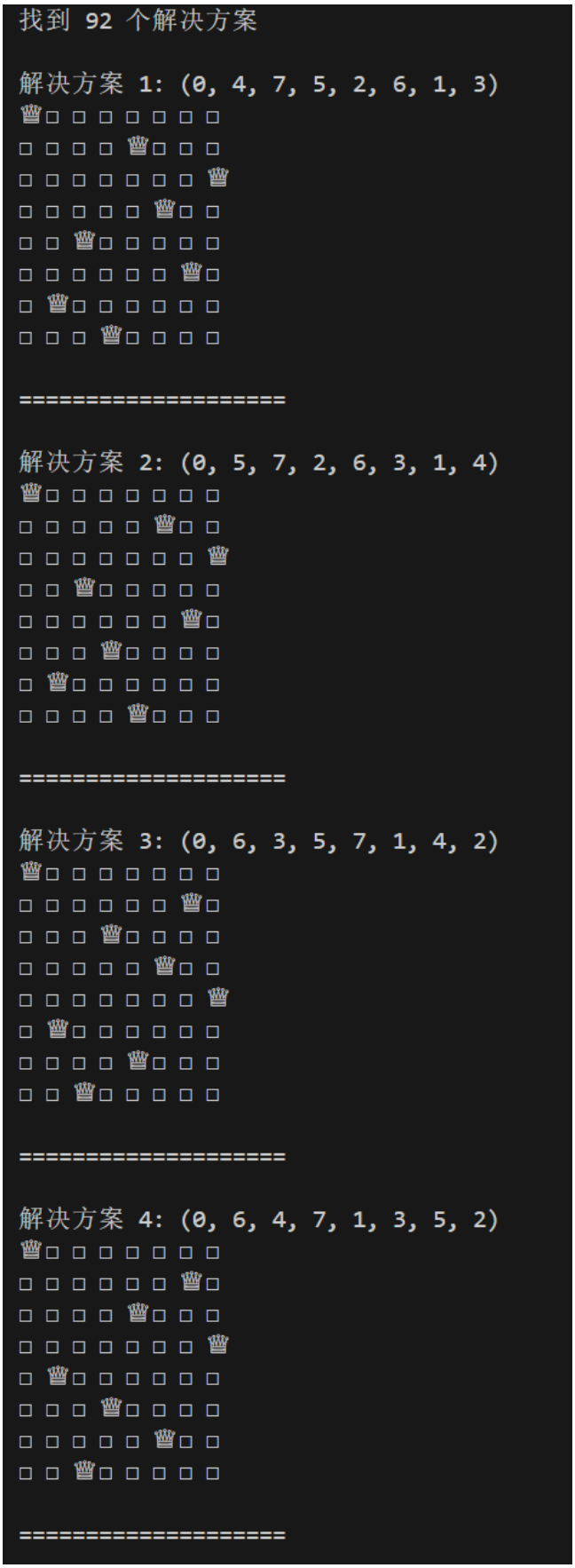
4. 实验结果

当 N=6 时，一共有 4 个不同结果：

```
找到 4 个解决方案

解决方案 1: (1, 3, 5, 0, 2, 4)
□ ♕ □ □ □ □
□ □ □ ♕ □ □
□ □ □ □ □ ♕
♕ □ □ □ □ □
□ □ ♕ □ □ □
□ □ □ □ ♕ □

====================

解决方案 2: (2, 5, 1, 4, 0, 3)
□ □ ♕ □ □ □
□ □ □ □ □ ♕
□ ♕ □ □ □ □
□ □ □ □ ♕ □
♕ □ □ □ □ □
□ □ □ ♕ □ □

====================
```


解决方案 3: (3, 0, 4, 1, 5, 2)
□ □ □ ♛ □ □
♛ □ □ □ □ □
□ □ □ □ ♛ □
□ ♛ □ □ □ □
□ □ □ □ □ ♛
□ □ ♛ □ □ □
==========
解决方案 4: (4, 2, 0, 5, 3, 1)
□ □ □ □ ♛ □
□ □ ♛ □ □ □
♛ □ □ □ □ □
□ □ □ □ □ ♛
□ □ □ ♛ □ □
□ ♛ □ □ □ □

当 N=8 时，一共有 92 个不同结果,这里可视化以前 4 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

①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。

(2)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'
```

在实验过程中，我发现代码中出现了类型不匹配的问题，具体表现为 col1 和 col2 是 Var 类型（逻辑变量），而 diff 是 int 类型。如果直接尝试将 col2 与 diff 相加，例如 col2+diff，会导致类型错误。这是因为在 kanren 的逻辑编程框架中，Var 类型代表未绑定的逻辑变量，无法直接与 Python 的内置整数类型进行数学运算。这种类型限制反映了逻辑编程与命令式编程的根本差异：在逻辑编程中，变量的绑定和计算需要通过约束和推理间接实现，而非直接操作。为此，我对代码进行了调整，通过合理的方式规避了对 Var 类型逻辑变量的直接操作，确保程序能够在逻辑框架下正确运行。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，具体修改如下。

```
# 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
    # membero 确保 queen 变量的值属于 self.cols 集合，无需手动算术检查
]

# 约束 2：列互异约束 - 确保任意两个皇后不在同一列
# not_equalo 用于比较两个变量，保证值不相等
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)  # 只需检查 i 之后的皇后，避免重复
```

```
    # not_equalo 处理逻辑变量的比较，无需直接操作数值
]

# 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):
        # 定义 lambda 函数检查对角线冲突
        # i,j 为行号（固定值），col1,col2 为列号（逻辑变量）
        # 使用比较操作检查行差与列差是否相等，避免直接算术运算
        check_diagonal = lambda col1, col2, i=i, j=j: (
            abs(i - j) != abs(col1 - col2)  # 比较主对角线和副对角线条件
        )
        # 使用 goalify 将 lambda 函数转为逻辑约束
        # goalify 自动适配逻辑变量（如 Var 类型），无需手动类型转换
        diagonal_constraints.append(
            (goalify(check_diagonal), (self.queens[i], self.queens[j]),
True)
        )
```

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
```

经针对这一问题，我查阅了相关资料并进行了深入分析，发现 EarlyGoalError 通常由以下原因引发：一是约束条件的组合方式存在缺陷，导致逻辑推理引擎在处理目标（Goals）时无法正确展开或求解；二是使用 lall 函数时，约束条件的定义或展开顺序出现问题，例如约束之间的依赖关系未妥善处理，或者条件顺序不当导致推理提前终止。

这一错误揭示了逻辑编程对目标定义和组合的高度敏感性，促使我更加注重约束逻辑结构的清晰性。为了解决这一问题，我对代码进行了系统性修改，逐步优化实现方式，主要包括以下几个方面：首先，我移除了 lall 函数的使用，转而直接以约束列表形式（如(eq(x, 1),membero(y, [1, 2]))）传递给 run 函数，避免了 lall 在底层处理中可能导致的展开顺序异常，同时简化了代码结构；其次，我调整了约束条件的组合方式，通过拆分复合约束并显式指定依赖关系，确保每个条件在逻辑上更加独立，防止变量过早绑定或冲突；最后，我将约束条件分开构建，例如分别定义 goal1 = eq(x, 1) 和 goal2 = membero(y, [1, 2])，然后在 run(0, [x, y], goal1, goal2) 中合并，这种分步方式不仅降低了出错概率，还便于调试时逐一验证约束的有效性。通过这些修改，我成功消除了 EarlyGoalError 错误，使代码在逻辑编程框架下运行更加稳定，同时提升了可读性和可维护性，这一过程也让我对 kanren 的目标处理机制有了更深刻的理解。具体修改见“**3.算法实现**”部分，伪代码如下。

```
Function GenerateConstraints()
    constraints = empty list
    // 1. 列范围约束
    For each queen in queens:
        Add constraint: queen must be in cols range
    // 2. 列互异约束
    For i = 0 to N-1:
        For j = i+1 to N-1:
            Add constraint: queens[i] ≠ queens[j]
    // 3. 对角线约束
    For i = 0 to N-1:
        For j = i+1 to N-1:
            DiagonalCheck(queen_i, queen_j, i, j):
                Return |i - j| ≠ |queen_i - queen_j|
            Add constraint: DiagonalCheck(queens[i], queens[j])
    Return constraints
```

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。

```python
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

在求解 N 皇后问题时，我们结合 itertools.permutations 和 kanren 逻辑编程进行优化。首先，利用 itertools.permutations(range(N))生成所有可能的列排列，每个排列代表一种皇后放置方案，避免了显式搜索棋盘状态。由于 N 个皇后不能在同一列，这一步已自动满足列约束。随后，我们使用 kanren 逻辑约束筛选出满足对角线约束的解，即确保任意两个皇后不在同一主对角线或副对角线上。这种方法在 N 较小时计算效率较高，比纯逻辑推理方法更快，因 kanren 在处理 N 个独立变量的层层约束时计算开销较大。然而，当 N 较大时，N!级的排列增长会导致计算复杂度过高，因此更适合中小规模问题。

```
# 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在 solve_recursive 函数中得到了充分体现，其设计基于一种系统化的试探与撤销策略，通过逐行放置皇后的方式逐步构建出问题的解决方案。具体而言，该函数利用递归的机制，在每一行中尝试放置一个皇后，并通过调用 is_safe 函数验证当前放置位置的合法性。is_safe 函数负责检查新放置的皇后是否与已有皇后在同一列或对角线上发生冲突，从而确保每个位置都符合 N 皇后问题的约束条件。当找到一个合法的放置位置时，该位置会被记录下来，并将当前解添加到解决方案列表中；随后，算法通过递归调用继续探索下一行的所有可能位置。如果当前路径无法产生有效解，则回溯到上一行，撤销之前的选择并尝试其他位置。这种“尝试-验证-前进或回退”的过程正是回溯算法的精髓所在，它通过逐步构建和调整解空间，最终找到所有满足条件的解。

然而，这种实现方式属于典型的命令式编程风格，与本次实验要求的逻辑编程范式存在本质差异。在命令式编程中，程序员需要明确指定每一步的操作流程，包括如何遍历行、如何检查约束以及如何管理解的集合。这种方式虽然直观且易于理解，但与逻辑编程的声明式特性——即仅描述问题的约束条件而非求解过程——形成了鲜明对比。由于本次实验的重点在于探索 kanren 库的逻辑编程能力，我不会在此对命令式实现的细节展开过多描述，而是将其作为一种对比，突出逻辑编程的独特优势和局限性。

值得注意的是，回溯算法在 N 皇后问题中的实现通过其逐步试探与回溯的机制，避免了对所有可能解空间的暴力枚举。与直接生成所有$N \times N$棋盘配置并逐一验证的方式相比，回溯算法显著减少了计算复杂度，尤其是在问题规模较大时，其效率优势更为明显。相比之下，使用 kanren 进行逻辑编程的实现可能在效率上有所不及。这种效率差异可能源于 kanren 库在处理 N 个独立变量时需要分层定义约束的特性：它通过逻辑变量和目标的统一（unification）逐步推理出解，而这一过程在底层实现上可能涉及更多的符号计算和约束传播开销，尤其当变量数量增加时，性能瓶颈会更加显著。尽管如此，kanren 的声明式方法在表达复杂关系和探索解空间的灵活性上具有独特优势，这也是逻辑编程的价值所在。

为了更清晰地展示回溯算法的核心逻辑，我将在下方给出 solve_recursive 函数的关键部分伪代码。这种伪代码以简洁的方式概述了算法的流程，便于理解其命令式实现的结构，同时也为与逻辑编程的对比提供了参考。需要强调的是，虽然伪代码体现了回溯算法的高效性但它并不符合本次实验对逻辑编程的要求，因此仅作为一种补充说明。

IsSafe(board, row, col, N)安全性检查：该函数用于检查在(row, col)位置放置皇后是否安全。

```
Function IsSafe(board, row, col, N)
    // Check if a queen can be placed on board[row][col]
    // Check previous rows in same column
    For i from 0 to row-1:
        If board[i] equals col:
            Return False

        // Check diagonals
        If absolute(board[i] - col) equals absolute(i - row):
            Return False
    Return True
```

SolveNQueens(N)求解 N 皇后问题：该函数用于寻找所有可能的解。

```
Function SolveNQueens(N)
    Initialize solutions as empty list
    Initialize board as array of size N with -1 values

    Function Backtrack(row)
        If row equals N:
            Add current board configuration to solutions
            Return

        For col from 0 to N-1:
            If IsSafe(board, row, col, N):
                Place queen at board[row] = col
                Backtrack(row + 1)
                // No need to remove queen as next iteration will overwrite
    Call Backtrack(0)
    Return solutions
```

主程序：该部分用于执行算法并输出结果。

```
Algorithm: N-Queens Solver
Main Program
    Input: N (board size)
    solutions = SolveNQueens(N)
    Print number of solutions found
    For each solution:
        Display board configuration
        Display visual representation
```

(4)实验收获

通过本次实验，我对逻辑编程的特点有了更加深入的理解，认识到它与传统的命令式编程截然不同的思维方式。逻辑编程通过声明式的描述，将问题的求解过程转化为约束条件的定义与推理，而非逐行指定计算步骤。这种范式的核心在于“描述问题是什么”，而非“如何解决”，这让我对编程的多样性有了新的认识。在实验过程中，我学会了如何将具体问题抽象为逻辑约束条件，并通过合理的建模来实现问题的求解。这种能力不仅提升了我的编程思维，也让我在面对复杂问题时多了一种有力的工具。

在学习和使用 kanren 库的过程中，我掌握了其约束求解机制的核心功能及其应用方式。首先，通过 var()创建逻辑变量，我能够灵活地表示未知量，为后续的推理奠定了基础。其

次，利用 membero 定义变量的域约束，我可以将问题限制在特定的取值范围内，从而有效缩小求解空间，提高推理效率。此外，通过 goalify 将普通的 Python 函数转化为逻辑约束，我成功地将自定义逻辑融入到逻辑编程框架中，进一步增强了 kanren 的表达能力。这些功能的结合让我能够以声明式的方式处理多样化的逻辑问题，例如集合成员关系、数值约束以及自定义规则的推理。

通过本次实验，我不仅加深了对约束逻辑编程的理解，还显著提升了自己的问题建模能力。在实践中，我逐渐熟悉了如何将现实问题分解为可推理的逻辑关系，并通过 kanren 的工具将这些关系转化为可执行的代码。这种从理论到实践的转化过程让我对逻辑编程范式的优势有了切身体会，例如其在知识表示、约束求解和探索性问题上的强大潜力。同时，这次实验也为我积累了宝贵的实践经验，让我在编写代码时更加注重问题的本质，而非仅仅关注实现细节。

在本次实验的启发下，我对逻辑编程产生了浓厚的兴趣，并计划在接下来的学习中进一步探索这一领域。我准备尝试其他更先进的逻辑编程库，例如基于 Prolog 的实现（如 pyDatalog 或 PySWIP），以对比不同工具的特点和适用场景。通过与 Prolog 等经典逻辑编程语言的结合，我希望更全面地掌握逻辑编程的理论基础和应用技巧，进一步提升自己的逻辑推理能力和编程水平。我相信，这种持续的学习和实践将为我在人工智能、知识工程或复杂系统建模等领域打下坚实的基础，同时也让我在编程思维上更加开阔和灵活。未来，我期待能将逻辑编程的技能应用到更具挑战性的实际项目中，不断深化对这一范式的理解与运用。