同濟大學

同濟大學

TONGJI UNIVERSITY

人工智能课程设计一

第一次实验报告

班级：国豪工科嘉定3班

姓名：倪雨舒

学号： 2353105

完成日期： 2025年3月13日

1. 问题概述

1.1问题背景

利用逻辑编程的范式，借助python语言中的kanren库，解决8皇后问题。8皇后问题简述为：在一个8*8的棋盘上，放置8个皇后棋子，要求任意两个皇后棋子不能处于同一条横行、竖行、对角线上。

1.2问题思路

首先应该确保所有的皇后都处于不同的行与列，然后再进行对对角线条件的判断，并且对角线的判断函数应该被单独定义出来便于筛选答案。首先利用回溯法以及约束条件生成所有可能的解，用于生成给序列的全部排列；导入kanren库，运用其中的run，var等内容，实现对答案列表的筛选。

2. 算法思路

首先生成约束条件：约束条件为遍历棋盘，如果当前的棋盘布局中的某一个棋子与其余某一个棋子处于同一条对角线的话，那么就返回false，否则为真。约束条件如下：

```
# 约束函数：确保皇后不在同一对角线
def queens_constraints(qs):
    """
    检查当前摆放的皇后是否符合八皇后规则：
    - 每个皇后都处于不同的行（递归构造保证）
    - 不能在同一条对角线上
    """
    for i in range(len(qs)): # 遍历已放置的皇后
        for j in range(i + 1, len(qs)): # 只比较后续皇后，避免重复
            if abs(qs[i] - qs[j]) == abs(i - j): # 判断是否在同一对角线上
                return False
    return True
```

然后利用回溯法通过递归的方法生成所有的八皇后问题的解的序列，如果把序列当初一个数组的话，数组的下标相当于列，数组的值相当于行。递归函数如下：

```
# 递归回溯函数来生成所有可能的皇后排列
def generate_permutations(N, current_permutation, possible_solutions):
    """
    递归构造所有合法的皇后排列，避免 itertools.permutations
    :param N: 棋盘大小（通常为 8）
    :param current_permutation: 当前放置的皇后排列（存储每列皇后的行索引）
    :param possible_solutions: 存储所有合法解的列表
    """
    # 终止条件：当已经放置 N 个皇后时，找到一个解
    if len(current_permutation) == N:
        possible_solutions.append(tuple(current_permutation)) # 记录找到的解
        return
    # 遍历所有可能的行位置（0 到 N-1）
    for row in range(N):
        if row not in current_permutation: # 确保不重复使用同一行
            current_permutation.append(row) # 选择当前行
            if queens_constraints(current_permutation): # 剪枝：检查是否符合对角线规则
                generate_permutations(N, current_permutation, possible_solutions) # 递归搜索下一个列
            current_permutation.pop() # 回溯：撤销当前选择，尝试下一个可能的行
```

然后定义逻辑变量q代表问题的可能解，membero定义可行解一定属于预处理之后的合法解集合之中，然后lall组合约束条件，形成约束规则，最后执行查询run函数，给出结果solution。

上述方法的代码如下：

```
# 生成所有符合条件的皇后排列
N = 8 # 棋盘大小
possible_solutions = [] # 存储所有可能的解
generate_permutations(N, [], possible_solutions) # 递归搜索
# 定义逻辑变量 q 并创建查询规则
q = var()
rules = lall(membero(q, possible_solutions)) # lall 作用是逻辑合取，membero 用于查找解
# 获取全部解,通过修改n的值
solutions = run(0, q, rules)
```

这种方法类似于穷举法，但是这种方法在更大的棋盘上就会出现运行时间过慢的情况，因此可以在查找答案的过程中应用判断规则找到答案并且不断剪枝，代码如下：

```
N = 8  # 8皇后
# 约束逻辑：确保皇后不在同列 & 不在同一对角线
def queens_constraints(qs):  1 个用法
    return lall(
        # 剪枝1: 不在同一列
        *[conde([qs[i] != qs[j], abs(qs[i] - qs[j]) != abs(i - j)])
          for i in range(N) for j in range(i + 1, N)]
    )
# 定义8个逻辑变量（代表每行皇后所在的列）
q_vars = [var() for _ in range(N)]

# 逻辑规则
rules = lall(
    # 剪枝2: 每个皇后只能放在 0~7 的列
    *[conde([q_vars[i] == c]) for i in range(N) for c in range(N)],
    # 应用剪枝1
     *goals: queens_constraints(q_vars)
)
# 运行逻辑推理，求解八皇后问题
solutions = run( n: 5, q_vars,  *goals: rules)  # 获取前5个解
```

但是在实际运行的过程中，我发现var逻辑变量无法进行加减运算，eq运算也无法取反，这应该是kanren库的局限性导致的，因此这种方法无法实际运行出来结果，只能作为一种思维上的思考过程。

3. 实验结果

所用到的库如下：

```
from kanren import run, var, lall, membero
import matplotlib.pyplot as plt
```

其中kanren库用于逻辑推理，matplotlib库用于绘可视化结果。

问题的约束函数如下：主要约束皇后之间不处于同一条对角线，其中qs[i]中储存皇后所处的行数，i代表所在列的编号。

```python
# 约束函数：确保皇后不在同一对角线
def queens_constraints(qs):  1 个用法
    """
    检查当前摆放的皇后是否符合八皇后规则：
    - 每个皇后都处于不同的行（递归构造保证）
    - 不能在同一条对角线上
    """
    for i in range(len(qs)):  # 遍历已放置的皇后
        for j in range(i + 1, len(qs)):  # 只比较后续皇后，避免重复
            if abs(qs[i] - qs[j]) == abs(i - j):  # 判断是否在同一对角线上
                return False
    return True
# 递归回溯函数来生成所有可能的皇后排列
```

应用到kanren库的代码如下：

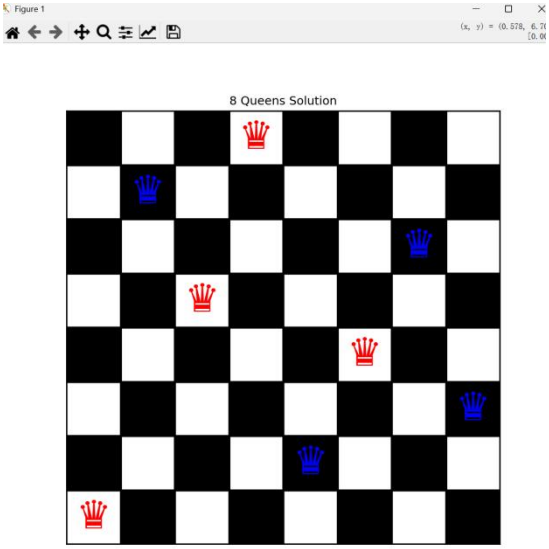
```python
# 生成所有符合条件的皇后排列
N = 8  # 棋盘大小
possible_solutions = []  # 存储所有可能的解
generate_permutations(N,  current_permutation: [], possible_solutions)  # 递归搜索

# 定义逻辑变量 q 并创建查询规则
q = var()
rules = lall(membero(q, possible_solutions))  # lall 作用是逻辑合取，membero 用于查找解

# 获取全部解,通过修改n的值
solutions = run( n: 0, q,  *goals: rules)
```

var() 定义逻辑变量 q，代表 待求解的皇后排列。membero(q, possible_solutions) 表示 q 必须是 possible_solutions 中的一个，lall() 用于 合取（AND 关系），即所有规则同时成立，run(5, q, rules) 获取前 5 个解（run(0, q, rules) 可获取所有 92 个解）。

由于python在绘制图像时比较方便，给出了一个最终的绘制出的8皇后的棋盘图，通过调整代码中n的个数，可以控制最终输出的解的个数。




```
D:\Anoconda\python.exe D:\PythonProject\test.py
Solution 1: (0, 4, 7, 5, 2, 6, 1, 3)
Solution 2: (0, 5, 7, 2, 6, 3, 1, 4)
Solution 3: (0, 6, 3, 5, 7, 1, 4, 2)
Solution 4: (0, 6, 4, 7, 1, 3, 5, 2)
Solution 5: (1, 3, 5, 7, 2, 0, 6, 4)
Solution 6: (1, 4, 6, 0, 2, 7, 5, 3)
Solution 7: (1, 4, 6, 3, 0, 7, 5, 2)
Solution 8: (1, 5, 0, 6, 3, 7, 2, 4)
Solution 9: (1, 5, 7, 2, 0, 3, 6, 4)
Solution 10: (1, 6, 2, 5, 7, 4, 0, 3)
Solution 11: (1, 6, 4, 7, 0, 3, 5, 2)
Solution 12: (1, 7, 5, 0, 2, 4, 6, 3)
Solution 13: (2, 0, 6, 4, 7, 1, 3, 5)
Solution 14: (2, 4, 1, 7, 0, 6, 3, 5)
Solution 15: (2, 4, 1, 7, 5, 3, 6, 0)
```

4.总结分析：

比较遗憾的是使用了一种比较暴力的方法进行求解，实际上最理想的模式是利用kanren中的逻辑表达式进行剪枝，最终在搜索过程中找出可行解，但是在多次修改未果后，放弃了这个想法，因为一个是对python的语法实在是不熟悉，而且对kanren库的建构也不是很熟悉，因此采取了这种较暴力的解决方法，没有充分发挥出kanren库的实力，主要问题在于操作符的不匹配，以及kanren库本身的一些限制。除此之外，在应对其他一些逻辑推理问题的时候，kanren库的优势还是很明显的，只需要对求解规则进行编写即可。

附可视化代码：

```
# 可视化函数
def plot_queens(solution):
    """
    绘制八皇后棋盘并显示皇后的位置
    :param solution: 由 8 个数（皇后所在的行索引）组成的元组
    """
    plt.figure(figsize=(8, 8))

    # 创建棋盘背景（黑白相间）
    board = [[(i + j) % 2 for j in range(8)] for i in range(8)]
    plt.imshow(board, cmap='binary', origin='lower', extent=(-0.5, 7.5, -0.5, 7.5))

    # 绘制网格线
    for x in range(8):
        for y in range(8):
            plt.gca().add_patch(plt.Rectangle((x - 0.5, y - 0.5), 1, 1, fill=False, edgecolor='black', linewidth=2))

    # 在棋盘上绘制皇后
    for col, row in enumerate(solution):  # 遍历列索引及对应的行索引
        plt.text(col, row, '♛', fontsize=40, ha='center', va='center',
                 color='red' if (col + row) % 2 == 0 else 'blue')  # 确保皇后颜色适应背景

    plt.xticks([])  # 隐藏 x 轴刻度
    plt.yticks([])  # 隐藏 y 轴刻度
    plt.title('8 Queens Solution')  # 添加标题
    plt.show()

# 打印并可视化每个解
count = 0
for sol in solutions:
    count += 1
    print(f'Solution {count}: {sol}')
    plot_queens(sol)

print(f'Solutions found: {count}')
```