人工智能实验

约束满足问题——回溯法和前向搜索解决 N 皇后问题

16 级计科教务 2 班

16337327

郑映雪

实验题目

约束满足问题——回溯法和前向搜索解决 N 皇后问题

实验内容

算法原理

1、回溯法

回溯法采用的是“试错”的思想，是暴力搜索法的一种。在解决问题的过程中，回溯法进行一步步的试探，当答案不能满足约束时，则返回上一步正确的步骤，尝试其他的步骤。回溯法可以找到存在的正确答案或尝试所有可能的步骤后发现没有正确答案。

假设解决一个问题的步骤可以化为一棵树，回溯法则是对这棵树进行深度优先搜索。对搜索到的每一点判断是否满足解的约束，如果满足则继续向下搜索，如果不满足则逐层回溯。回溯法在某个叶子结点会搜索到解，如果一直回溯到了根节点且所有子树都被搜索完毕，则代表这个问题无解。

但是，回溯法需要将一个解的各种情况完全试探后才检查满足条件，时间复杂度较高。如果问题是无解的，回溯法的时间复杂度可以达到指数级。解决这一问题的办法是在传播中加入约束。

2、前向搜索

前向搜索是回溯法的变种，它改善了回溯法每一步都要遍历所有解从而导致时间复杂度较高的问题。当搜索到当前问题时，根据当前节点的情况和约束条件，修改未赋值的节点的值域。这样能大大减少搜索的节点数。

3、用回溯法和前向搜索解决 N 皇后问题

N 皇后问题，则是在 N*N 棋盘的每一行都放置一个皇后，所有皇后互相不能攻击。根据国际象棋中皇后的性质，可以理解为每一个皇后同行、同列以及两条对角线上没有别的皇后。我们可以用数学公式描述这个约束：

$$abs(Q_i - Q_j) \neq abs(i - j)$$

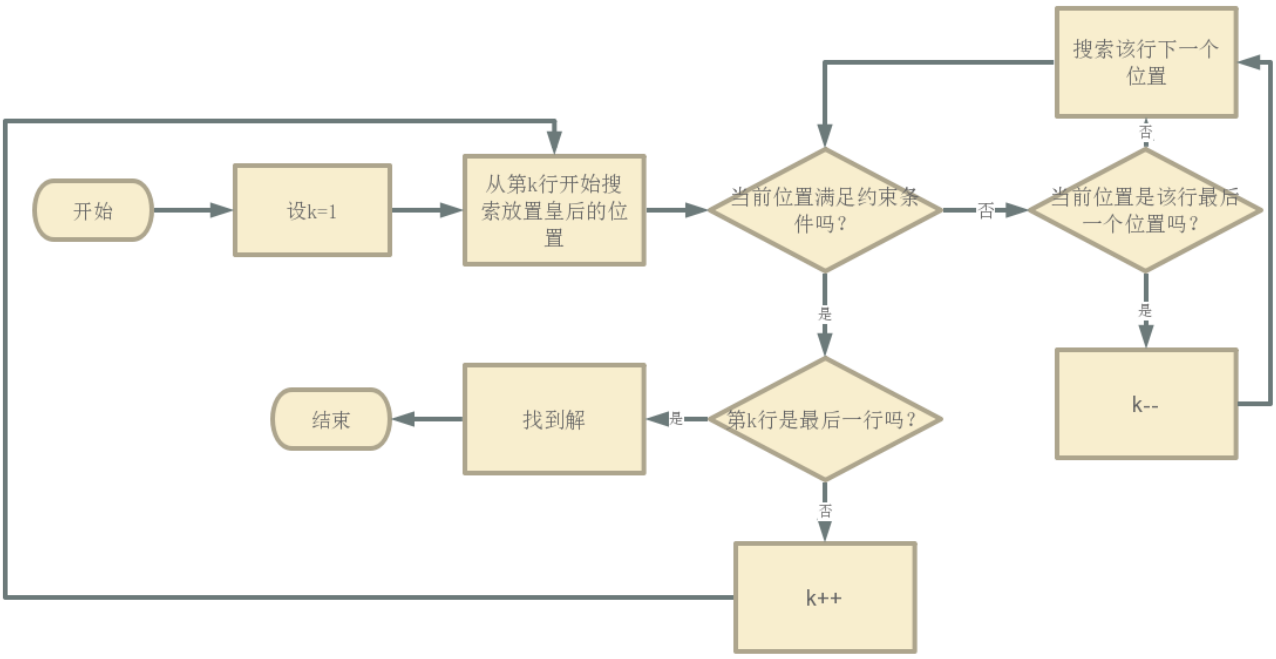
（其中，i、j 分别为任意两个皇后的行数，Qi、Qj 分别为在该行的列数）

使用回溯法解决 N 皇后问题，则是在每一个结点，都判断该列放置皇后的位置与前面已经放置皇后的行的位置是否满足这个约束，如果不满足，则换一种放置方法。如果所有放置方法都尝试过仍不满足约束，则回溯到上一次正确的放置方法，再换一种方法继续往下尝试。

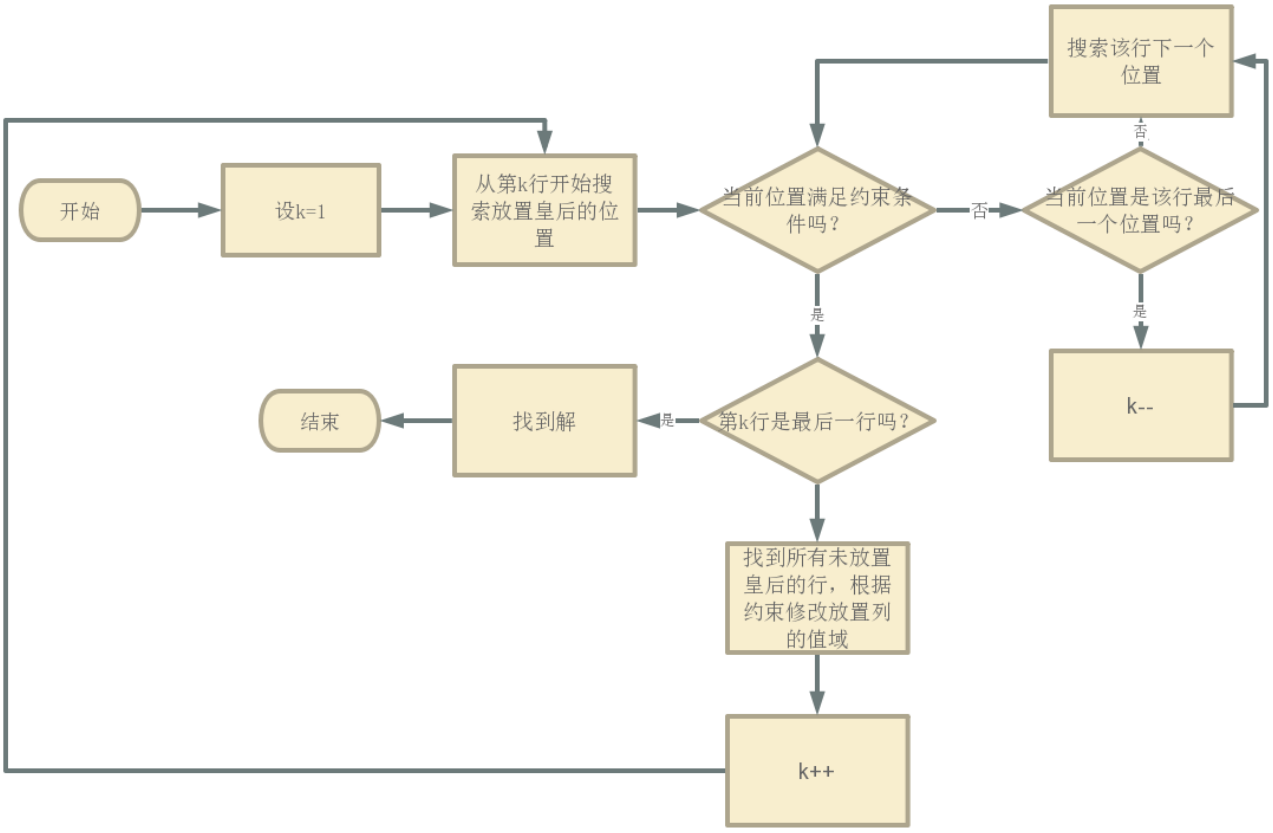
使用前向搜索解决 N 皇后问题，则是在每一行确定放置之后，就修改还没有放置皇后的行中可放位置的值域，这样可以大大减少搜索结点的个数，节省时间。

流程图

1、回溯法解决 N 皇后问题



2、向前搜索解决 N 皇后问题



关键代码

1、回溯法关键代码

回溯法的递归函数是算法的主体。首先判断是否是最后一行，如果是，则输出解，如果不是，对于当前行的每一列找到一个满足于前面判断过的行不冲突的列，将当前解作为根节点继续往下递归搜索子树。

```
1. def nqueen(k, n, location):
2.     global num
3.     if k > n:#上一行已经是最后一行
4.         for i in range(1, n + 1):
5.             print(location[i], end=" ")
6.         print("\t")#输出解
7.         num += 1#解的个数+1
```

```
8.          return#继续寻找下一个解
9.      for i in range(1,n+1):#每一列
10.         flag=1
11.         for j in range(1,k):
12.             if i==location[j] or k-j==abs(i-location[j]):#跟前面判断过的行有冲突，则该位置不可行，继续外循环
13.                 flag=0
14.                 break
15.         if (flag):#如果该位置可行，则该行位置为当前判断的列标志
16.             location[k]=i
17.             nqueen(k+1,n,location)#递归搜索当前情况的子树
```

2、前向搜索关键代码

前向搜索在每次判断之前，不是与之前所有已经判断完毕的行作比较，而是修改所有未赋值行的值域，这样当前行赋值的时候，值域里的值一定是可行的，这样就减少了递归树的结点。

```
1. def nqueen(k, n, location, domain):
2.     global num
3.     if k > n:#上一行已经是最后一行
4.         for i in range(1, n + 1):
5.             print(location[i], end=" ")
6.         print("\t")#输出解
7.         num += 1#解的个数+1
8.         return#继续寻找下一个解
9.     if len(domain[k]) == 0:#如果当前行可选值域为空集，则回溯
10.        return
11.    for i in domain[k]:#对当前行的值域的每一个可选项
12.        location[k] = i#因为可选，所以可以直接赋值
13.        newdomain = {0:[]}
14.        for i1 in range(1, n + 1):#深复制 domain 字典
15.            tmp = [datas for datas in domain[i1]]
16.            newdomain[i1]=tmp
17.        for j in range(k + 1, n + 1):#对接下来的行，在 domain 里删除不满足约束条件的值
18.            if i in newdomain[j]:
19.                newdomain[j].remove(i)
20.            if i+(j-k) in newdomain[j]:
```

```
                newdomain[j].remove(i + (j - k))
            if i - (j - k) >= 1 and i-(j-k) in newdomain[j]:
                newdomain[j].remove(i - (j - k))
        nqueen(k + 1, n, location, newdomain)#对新的 domain 字典递归搜索当前的子节点
```

实验结果及分析

实验结果展示

结果说明：在这个实验里，我找到 N*N 情况下的所有解。每行输出一个解，每一个解由 N 个数组成，这 N 个数是从第一行到最后一行，每一行皇后所放置的列数。

8 皇后的部分结果如下，共有 92 个解：

```
6 4 7 1 8 2 5 3
6 8 2 4 1 7 5 3
7 1 3 8 6 4 2 5
7 2 4 1 8 5 3 6
7 2 6 3 1 4 8 5
7 3 1 6 8 5 2 4
7 3 8 2 5 1 6 4
7 4 2 5 8 1 3 6
7 4 2 8 6 1 3 5
7 5 3 1 6 8 2 4
8 2 4 1 7 5 3 6
8 2 5 3 1 7 4 6
8 3 1 6 2 5 7 4
8 4 1 3 6 2 7 5
解的个数为： 92
```

10 皇后的部分结果如下，共有 724 个解：

```
10 7 2 4 1 8 5 9 6 3
10 7 2 4 9 1 8 5 3 6
10 7 2 6 3 1 8 5 9 4
10 7 4 1 3 6 9 2 8 5
10 7 4 1 3 8 6 2 9 5
10 7 4 1 3 9 6 8 5 2
10 7 4 1 5 2 9 6 8 3
10 7 4 1 8 2 9 6 3 5
10 7 4 1 9 2 6 8 3 5
10 7 5 2 8 1 3 9 6 4
10 8 2 4 1 7 9 6 3 5
10 8 5 2 4 1 7 9 3 6
10 8 5 2 4 1 7 9 6 3
10 8 5 3 1 6 2 9 7 4
解的个数为： 724
```

其余 N 的取值的解可以运行附上的代码。

评测指标展示

在本实验中，我采取的时间检测是找到所有的解的时间而不是找到一个解就退出的时间。（由于使用语言是 python，所以绝对时间上来说不如 C++运行得快，但是两种算法我都是用 python 写的，所以可以进行横纵向的相对比较）

经过对不同的 N、两种不同算法的横纵向比较，得出的时间比较如下表所示：

n	解的个数	回溯法花费时间	前向搜索花费时间
8	92	0.021201502379307s	0.0182959011395844s
10	724	0.476907723480628s	0.4073350004629336s
11	2680	2.855825431818851s	1.7405289559590211s
12	14200	16.790437457253315s	10.542716391558994s
15	2279184	4235.661361775515s	2386.2090108625116s

可以看出前向搜索相比于回溯法节省了很多时间，且 N 越大，前向搜索的优越性就越能显现。所以在解决回溯问题时，当 N 很大时，可以考虑使用前向搜索的改进从而节省时间。