递归回溯解决 n 皇后问题实现报告

王资 18214668 计算机技术

1.基本思想

1.1 对已有的一维向量的递归回溯算法进行修改，使得其可以对二维矩阵也能够做类似的递归回溯，最终得到解。具体细节如下，图为一维向量的递归回溯算法：

```
void backtrack (int t)
{
      if (t>n) output(x);
      else
        for (int i=f(n,t);i<=g(n,t);i++) {
          x[t]=h(i);
          if (constraint(t)&&bound(t))
backtrack(t+1);
          }
}
```

将输入改为 2 个参数 x_, y_ ，因为是二维矩阵；递归结束条件为 x_ == n-1 && y_ == n-1 && 该解包含了 n 个皇后；循环只有二值，即 1 和 0（true 和 false）；若 x_, y_的解符合条件，则需要根据 x_, y_ 的状态决定是 y_ + 1 还是 x_+1 且 y_=0. 可以看出该思想与递归回溯的基本框架一致，只是在某些地方针对具体问题进行了修改。

1.2 对于全排序，使用排列树的递归回溯算法，每次递归均输出，且无条件进入下一状态即是全排序的结果。

2.代码中的数据结构

使用变量 n 代表 n 皇后问题，根据规则，只有 n>=4 时才有解；x[n][n]代表棋盘，是 bool 矩阵，因为取值是二值的。

全排序问题中则只需要用一个数组储存需要全排序的数列的最初状态即可。

3.程序流程

在程序中设定 n 的大小后运行程序，程序就从 x(0,0)即棋盘左上角的位置开始进行递归回溯：分别置该位置值为 0 或 1，符合规则则进入下一个位置（下一个递归调用），下一个位置的说明如基本思想中所述，否则继续尝试下一个可能的值，当尝试过所有的可能取值后，结束该递归栈，也即返回到上一次递归栈中，直到达到递归结束的条件。

全排序问题的流程与 n-queens 问题基本一致，只是进入下一个递归调用的条件为永真，且每次进入调用都要做一次输出（也可以在结束递归栈时进行输出）。

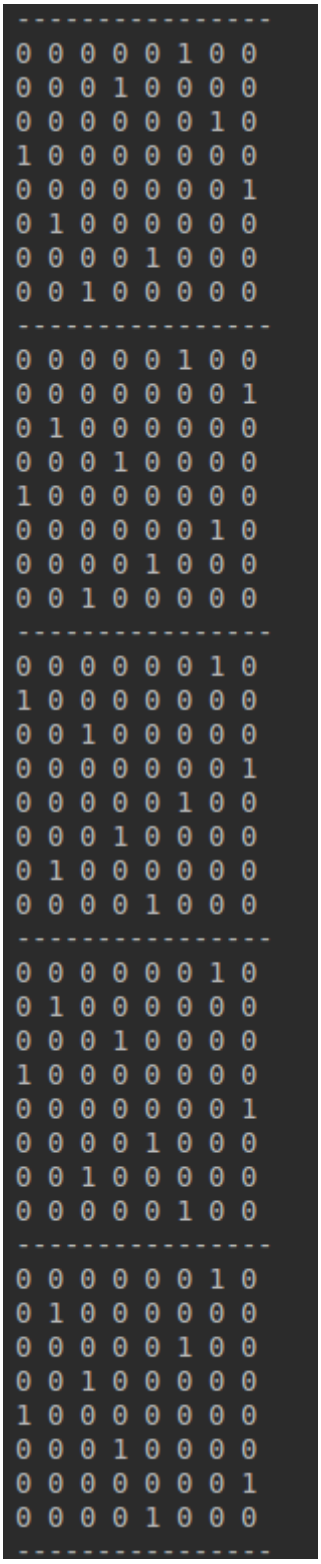
4.调试过程出现的问题和解决方法

1.根据原先的递归回溯框架，有可能会出现棋盘中的皇后总数不足 n 的情况，即某行（或列，在棋盘中进行转置或对称是同一个图，下面省略该说明）会是全 0 的情况。这个问题的产生是使用二维矩阵替代一维数组后，判断当前状态是否符合规则会更加的复杂，需要判断当前状态的行是否不含皇后，同时，因此在遍历到某行时，只遍历到了若干列，因此该行是可以不含皇后的，所以只需要判断该行之前的每一行是否都有且只有一个皇后。

2.使用上述的解决方法仍可能导致最后一行是不含皇后的，因为没有第 n 行的元素去判断第 n-1 行及之前的行是否含皇后，所以需要在递归终止条件上再加上一个棋盘中有且只有 n 个皇后的约束，该约束可以简化成每行有且只有一个皇后，因为前者是后者的必要条件。

5.运行结果

n=8 的部分结果



当 n=20 时，程序运行了很长时间（～4 小时）仍未能输出所有的解，也即程序还未结束，且还是(0,0)=1 的子树中，因此可以认为 n>=20 时，使用矩阵结构的递归回溯算法需要的时间不可接受。

全排序问题：

左图为输入为 4 的全部结果，右图为输入为 9 的部分结果

```
4
A B C D
A B D C
A C B D
A C D B
A D C B
A D B C
B A C D
B A D C
B C A D
B C D A
B D C A
B D A C
C B A D
C B D A
C A B D
C A D B
C D A B
C D B A
D B C A
D B A C
D C B A
D C A B
D A C B
D A B C

Process finished with exit code 0
```

```
H I B C D F E A G
H I B C D F E G A
H I B C D G F A E
H I B C D G F E A
H I B C D G A F E
H I B C D G A E F
H I B C D G E A F
H I B C D G E F A
H I B C D A G F E
H I B C D A G E F
H I B C D A F G E
H I B C D A F E G
H I B C D A E F G
H I B C D A E G F
H I B C D E G A F
H I B C D E G F A
H I B C D E A G F
H I B C D E A F G
H I B C D E F A G
H I B C D E F G A
I B C D E F G H A
I B C D E F G A H
I B C D E F H G A
I B C D E F H A G
I B C D E F A H G
I B C D E F A G H
I B C D E G F H A
I B C D E G F A H
I B C D E G H F A
I B C D E G H A F
I B C D E G A H F
I B C D E G A F H
I B C D E H G F A
I B C D E H G A F
I B C D E H F G A
```

6.总结

本实验更换了构造结构去使用递归回溯的算法解决 n-queens 问题，这种解决方法的时间复杂度更高，且需要处理的判断条件也会更多，因而实现所需的代码量也比使用数组的构造方法多。对于运行情况，使用矩阵的构造结构，搜索树的最大深度为 n^2 因此空间使用率不会太高，也就是 n^2 的量级，但由于需要遍历 2^(n^2)个节点，且是使用递归的方法遍历，因此时间效率非常差，再 n 大于 20 时，需要消耗的时间已经是不可接受的了，但由于空间使用率不高，就算得到所有的解需要花费极长时间，也不会耗尽内存资源导致宕机。

全排序则使用了排序树的递归回溯方案，若使用该方案解决 n-queens 问题，复杂度会比用子集树方法更优，前者是 O(n!)，后者是 O(n^n).

7.存在问题和改进设想

在实现全排序程序的时候，回想到了原先的递归回溯框架应当是属于子集树的递归回溯框架，因此即使是使用数组的储存结构，其时间复杂度也是 O(n^n)，回想到课上说使用数组结构的应当是使用排序树结构的递归回溯结构，对比全排序的时间复杂度，就结合使用数组结构和排序树结构的递归回溯框架，该算法的时间复杂度就能继续下降到 O(n!).

迭代回溯算法修正

相关伪代码应修正如下

```
void iterativeBacktrack () {
  std::queue queue[t];
  int t=1;
  queue[t].push(1:n);
  while (t>0) {
    if (!queue[t].empty()) {
      int h = queue[t].pop();
      x[t]=h;
      if (constraint(t)) {
        if (solution(t))  output(x);
        else {
          t++;
          queue[t].push(1:n);
        }
      }
    }
    else t--;
  }
}
```

对 n=4 的程序执行流程如下

迭代 t	队列变化	t 的变化	文字备注
1	1 √	t++	对 t=1 尝试时，队列首的元素 1 可以满足棋盘约束，因此进入到下一个迭代 t++
2	1 2 3 X X √	t++	对 t=2，尝试队列中的元素 1，2 都发现与约束条件相悖，因此不会在 1 或 2 的位置进入下轮迭代，而 3 满足约束条件，因此在 3 的时候 t++
3	1 2 3 4 X X X X	t--	对 t=3，尝试了队列中全部 4 个元素都发现不能满足约束条件，此时队列为空，t--
2	4 √	t++	对 t=2 的队列，前面 3 个元素已经移出队列，因此从第 4 个元素开始
3	1 2 X √	t++	
4	1 2 3 4 X X X X	t--	
3	3 4 X X	t--	
2		t--	对 t=2，此时队列已空，因此直接 t--
1	2 √	t++	
2	1 2 3 4 X X X √	t++	
3	1 √	t++	
4	1 2 3 X X √		此时得到一个解，因此要 output，结果是[2, 4, 1, 3]
4	4 X	t--	
3	2 3 4 X X X	t--	
2		t--	
1	3 √	t++	
2	1 √	t++	
3	1 2 3 4 X X X √	t++	
4	1 2 X √		此时得到一个解，因此要 output，结果是[3, 1, 4, 2]
4	3 4	t--	

	X X		
3		t--	
2	2 3 4 X X X	t--	
1	4 √	t++	
2	1 √	t++	
3	1 2 3 X X √	t++	
4	1 2 3 4 X X X X	t--	
3	4 X	t--	
2	2 √	t++	
3	1 2 3 4 X X X X	t--	
2	3 4 X X	t--	
1		t--	
0			达到迭代终止条件，迭代结束，最终得到了两个解。