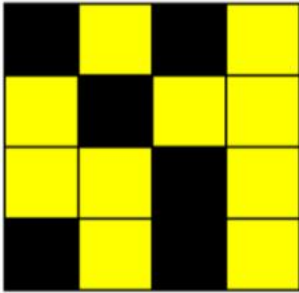
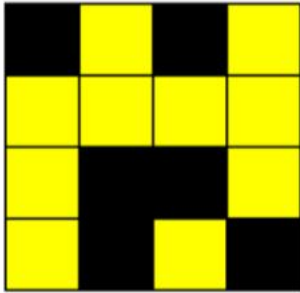


## 实验四：回溯与分支限界算法设计

学号	202018526	姓名	高树林	成绩	
友情提示	1. 算法描述及代码实现与网络或他人雷同者，均按 0 分计算； 2. 要求算法描述明确、代码清晰、格式美观； 3. 纸质版与电子版同时提交（电子版命名格式： <b>完整学号-姓名-实验 X-实验名称</b> ，其中 $X \in \{1,2,3,4\}$ ，不得省略“-”，如：2088166-乔峰-实验 1-分治与递归策略）； 4. 电子版中代码需格式化处理，方便查看（ <a href="http://www.codeinword.com/">http://www.codeinword.com/</a> ）。				
实验目的	1. 掌握回溯法解决问题的一般步骤。 2. 学会使用回溯法解决实际问题。 3. 掌握分支限界法解决问题的基本思想。 4. 学会使用分支限界法解决实际问题。				
实验内容	1. 骑士游历问题（采用回溯法）：在国际象棋的棋盘（8 行×8 列）上放置一个马，按照“马走日字”的规则，马要遍历棋盘，即到达棋盘上的每一格，并且每格只到达一次。若给定起始位置 $(x_0, y_0)$ ，编程探索出一条路径，沿着这条路径马能遍历棋盘上的所有单元格。  2. 行列变换问题(采用分支限界法):给定两个 $m \times n$ 方格阵列组成的图形A和图形B，每个方格的颜色为黑色或黄色，如下图所示。行列变换问题的每一步变换可以交换任意 2 行或 2 列方格的颜色，或者将某行或某列颠倒。上述每次变换算作一步。试设计一个算法，计算最少需要多少步，才能将图形A变换为图形B。  <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;">  <p>图形 A</p> </div> <div style="text-align: center;">  <p>图形 B</p> </div> </div>				
算法描述	1. 骑士游历问题解题思路或算法思想 在每次选择方向时，不能任意选择，而是要按照一定的策略。可以按照“先苦后甜”的策略，先估算一下当前地点的下一个位置有哪些，再估测这些路径中那些是难走的，将难走的先走，剩下简单走的就留下来了，这样一步就比一步简单。  2. 行列变换问题解题思路或算法思想 在选择结点时，不能任意选择。要先算出当前节点下，他的所有孩子结点的一个函数值，这个函数值代表应该向该孩子结点扩展的程度，以便使搜索向着解空间上有最优解的分支推进，进而更快的找到全局最优解。				
程序	1. 骑士游历问题 代码： <pre>1. def FindPath(I, J):</pre>				

及运行结果（附截图）

```
2.     chess[I][J] = 1
3.     posCount = 0
4.     posI = [1, 1, 2, 2, -1, -1, -2, -2]
5.     posJ = [2, -2, 1, -1, 2, -2, 1, -1]
6.     nowI = I
7.     nowJ = J
8.     nexI = [0] * 8
9.     nexJ = [0] * 8
10.    for queuenumber in range(2, 65):
11.        posCount = 0
12.        for k in range(8):
13.            nextI = nowI + posI[k]
14.            nextJ = nowJ + posJ[k]
15.            if nextI >= 8 or nextI < 0 or nextJ >= 8 or nextJ < 0:
16.                continue
17.            if chess[nextI][nextJ] == 0:
18.                nexI[posCount] = nextI
19.                nexJ[posCount] = nextJ
20.                posCount += 1
21.        if posCount == 0 and queuenumber < 63:
22.            return False
23.        minPosCounter = 8
24.        for posNum in range(posCount):
25.            posCountTemp = 0
26.            for k in range(8):
27.                nextnextI = nexI[posNum] + posI[k]
28.                nextnextJ = nexJ[posNum] + posJ[k]
29.                if nextnextI >= 8 or nextnextI < 0 or nextnextJ >= 8 or nextnextJ < 0:
30.                    continue
31.                if chess[nextnextI][nextnextJ] == 0:
32.                    posCountTemp += 1
33.                if minPosCounter > posCountTemp:
34.                    minPosCounter = posCountTemp
35.                nowI = nexI[posNum]
36.                nowJ = nexJ[posNum]
37.                chess[nowI][nowJ] = queuenumber
38.        return True
39.
40.
41. if __name__ == "__main__":
42.     chess = [[0] * 8 for _ in range(8)]
43.     a, b = input('请输入起始点坐标 (x y):').split(' ')
44.     a = int(a)
```

```

45.     b = int(b)
46.     if FindPath(a, b):
47.         print("路径为: ")
48.         for i in range(8):
49.             for j in range(8):
50.                 print("%4d" % chess[i][j])
51.             print(' ')
52.     else:
53.         print('未找到遍历所有结点的路径! ')

```

截图:



```

请输入起始点坐标 (x y):0 0
路径为:
  1   4  57  20  47   6  49  22
 34  19   2   5  58  21  46   7
   3  56  35  60  37  48  23  50
 18  33  38  55  52  59   8  45
 39  14  53  36  61  44  51  24
 32  17  40  43  54  27  62   9
 13  42  15  30  11  64  25  28
 16  31  12  41  26  29  10  63

```

# 1. 行列变换问题

代码:

```

1. def solve():
2.     queue = [sour]
3.     while len(queue):
4.         status_of_1 = queue[0]
5.         del queue[0]
6.         for i in range(4):
7.             for j in range(3):
8.                 c1 = 1 << (i * 4 + j)
9.                 c2 = 1 << (i * 4 + j + 1)
10.                if status_of_1 & c1 != status_of_1 & c2:
11.                    status_of_2 = status_of_1
12.                    status_of_2 ^= c1
13.                    status_of_2 ^= c2
14.                    if a[status_of_2] == -1:
15.                        a[status_of_2] = a[status_of_1] + 1
16.                        b[status_of_2] = (i * 4 + j) + 1
17.                    if status_of_2 == dest:

```

```

18.         return True
19.         queue.append(status_of_2)
20.     for i in range(3):
21.         for j in range(4):
22.             c1 = 1 << (i * 4 + j)
23.             c2 = 1 << (i * 4 + j + 4)
24.             if status_of_1 & c1 != status_of_1 & c2:
25.                 status_of_2 = status_of_1
26.                 status_of_2 ^= c1
27.                 status_of_2 ^= c2
28.                 if a[status_of_2] == -1:
29.                     a[status_of_2] = a[status_of_1] + 1
30.                     b[status_of_2] = - (i * 4 + j) - 4
31.                     if status_of_2 == dest:
32.                         return True
33.                     queue.append(status_of_2)
34.     return False
35.
36.
37. def output(status, moves):
38.     if status != sour:
39.         c1 = c2 = tem_state = 0
40.         if b[status] > 0:
41.             c1 = 1 << b[status] - 1
42.             c2 = 1 << b[status]
43.             status_of_temp = status
44.             status_of_temp ^= c1
45.             status_of_temp ^= c2
46.             output(status_of_temp, moves - 1)
47.             c1 = (b[status] - 1) // 4
48.             c2 = (b[status] - 1) % 4
49.             print("第%d 步" % moves)
50.             map1[c1][c2], map1[c1][c2 + 1] = map1[c1][c2 + 1], map1[c1][c2
51. ]
52.         for i in range(4):
53.             for j in range(4):
54.                 print(map1[i][j], end=' ')
55.                 print('')
56.         else:
57.             b[status] = -b[status]
58.             c1 = 1 << (b[status] - 4)
59.             c2 = 1 << b[status]
60.             status_of_temp = status
61.             status_of_temp ^= c1

```

```

61.         status_of_temp ^= c2
62.         output(status_of_temp, moves - 1)
63.         c1 = (b[status] - 4) // 4
64.         c2 = (b[status] - 4) % 4
65.         print("第%d 步" % moves)
66.         map1[c1][c2], map1[c1][c2 + 1] = map1[c1][c2 + 1], map1[c1][c2
    ]
67.         for i in range(4):
68.             for j in range(4):
69.                 print(map1[i][j], end='')
70.                 print('')
71.         b[status] = -b[status]
72.
73.
74. if __name__ == '__main__':
75.     map1 = []
76.     Capacity = 1 << 16
77.     a = [-1] * Capacity
78.     b = [0] * Capacity
79.     print("请输入转换前的图形（0 表示黄色方块，1 表示黑色方块）：")
80.     end = []
81.     s = ''
82.     sour = dest = 0
83.     for i in range(4):
84.         s += input(' ')
85.     map1 = list(s)
86.     print(map1)
87.     for i in range(16):
88.         sour |= int(ord(map1[i]) - ord('0')) << i
89.     print(sour)
90.     print("请输入转换后的图形（0 表示黄色方块，1 表示黑色方块）：")
91.     s = ''
92.     for i in range(4):
93.         s += input(' ')
94.     map2 = list(s)
95.     for i in range(16):
96.         sour |= int(ord(map2[i]) - ord('0')) << i
97.     Capacity = 2 ** 16
98.     a = [-1] * Capacity
99.     b = [0] * Capacity
100.    solve()
101.    if a[dest] != -1:
102.        print("至少需要%d 步" % a[dest])

```

截图：

	<p>请输入转换前的图形（0表示黄色方块，1表示黑色方块）：</p> <pre> 1010 0100 0010 1010 </pre> <p>请输入转换后的图形（0表示黄色方块，1表示黑色方块）：</p> <pre> 0110 0001 0010 1010 </pre> <p>最少需要2步！</p>
总结	<p>本次实验是四个实验报告中最难的，也是花时间最多的，大概花费了两周时间，通过两周的调代码，看算法，我深刻认识到了回溯法和分支界限法的异同点。相同点在于回溯法和分支界限法都是在梳状的接空间上求解，回溯法找出满足条件的所有解，通过回溯法找出满足条件的所有解，通过约束函数和限界函数可以找到问题的最优解，同时分支界限法主要应用于找满足条件的一个解或者最优解。但是两者在相同中又有很大的不同之处，他们最大的区别在于，回溯法采用的是深度优先遍历搜索的策略，而分支界限法采用的是广度优先遍历的搜索策略，回溯法中一般用栈数据结构实现结点的存储，分支界限法一般多用队列实现结点的存储。回溯法只有在所有子结点被遍历之后才出栈，而分支界限法中的每个结点只能被访问一次。</p>