Task2-Q1

2019年10月19日 10:30

- **回溯法的回顾**

回溯法是本科期间算法分析设计中重要的一课，研究生期间的算法课程中仍对他进行了讲授。现在我们来回顾一下回溯算法

回溯法的思路的简单描述是：把问题的解空间转化成了图或者树的结构，然后用深度优先搜索进行遍历，遍历过程中进行记录和寻找所有可行的最优解。其基本思想类似于深度优先搜索：

- **图的深度优先搜索**
- **二叉树的后序遍历**
  - 分支限界法：广度优先遍历
  - 思想类同于：图的广度优先遍历 / 二叉树的层序遍历

- **详细描述**

  回溯法按深度优先策略搜索问题的解空间树。首先从根节点出发搜索解空间树，当算法搜索至解空间树的某一节点时，先利用**剪枝函数**判断该节点是否可行（即能得到问题的解）。如果不可行，则跳过对该节点为根的子树的搜索，逐层向其祖先节点回溯；否则，进入该子树，继续按深度优先策略搜索。

  回溯法的基本行为是搜索，搜索过程使用剪枝函数来为了避免无效的搜索。

  剪枝函数包括两类：

  1. 使用约束函数，剪去不满足约束条件的路径；

  2.使用限界函数，剪去不能得到最优解的路径。

  - **回溯法的应用**

    当问题是需要满足某种性质（约束条件）的所有解或者最优解时，往往使用回溯法

- **回溯法的实现-递归与回溯**

  回溯法的实现方法有两种：递归和递推（也称迭代）。一般来说，一个问题两种方法都可以实现，只是在算法效率和设计复杂度上有区别

  - **递归**

    思路简单，设计容易，但是效率低，核心代码如下：

```
//针对N叉树的递归回溯方法
void backtrack (int t)
{
    if (t>n) output(x); //叶子节点，输出结果，x是可行解
    else
       for i = 1 to k//当前节点的所有子节点
        {
            x[t]=value(i); //每个子节点的值赋值给x
            //满足约束条件和限界条件
          if (constraint(t)&&bound(t))
                backtrack(t+1);  //递归下一层
        }
}
```

  - **回溯法**

    算法设计相对复杂，但是效率高

```
//针对N叉树的迭代回溯方法
void iterativeBacktrack ()
{
    int t=1;
    while (t>0) {
        if(ExistSubNode(t)) //当前节点的存在子节点
        {
            for i = 1 to k  //遍历当前节点的所有子节点
            {
                x[t]=value(i);//每个子节点的值赋值给x
                if (constraint(t)&&bound(t))//满足约束条件和限界条件
                {
                    //solution表示在节点t处得到了一个解
                    if (solution(t)) output(x);//得到问题的一个可行解，输出
                    else t++;//没有得到解，继续向下搜索
                }
            }
        }
        else //不存在子节点，返回上一层
        {
            t--;
        }
}
```

  - 回溯法与子集树

    所给的问题是从n个元素的集合S中找出满足某种性质的子集时，相应的解空间成为子集树。

    如0-1背包问题，从所给重量、价值不同的物品中挑选几个物品放入背包，使

得在满足背包不超重的情况下，背包内物品价值最大。它的解空间就是一个典型的子集树。

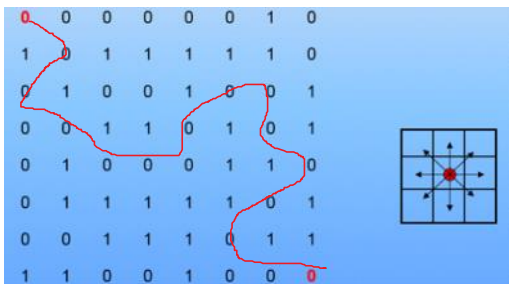
```
void backtrack (int t)
{
  if (t>n) output(x);
    else
      for (int i=0;i<=1;i++) {
        x[t]=i;
        if (constraint(t)&&bound(t)) backtrack(t+1);
      }
}
```

- **经典问题**
  - **八皇后问题**
    在一行放入棋子，然后判断是否符合规则，符合的情况下再去放下一行，下一行如果所有位置都不符合，退回到上一行，上一行的棋子再放置一个新的位置，然后再进去下一行判断有没有符合规则的棋子的位置。这种方法叫做递归回溯，每一行就相当于是一个回溯点。

```
# -*- coding: utf-8 -*-
# @Time : 2019/10/12 17:00
# @Author : BaoBao
# @Mail : baobaotql@163.com
# @File : queens.py
# @Software: PyCharm
def is_rule(queen_tup, new_queen):
    """
    :param queen_tup: 棋子队列，用于保存已经放置好的棋子，数值代表相应棋子列号
    :param new_queen: 被检测棋子，数值代表列号
    :return: True表示符合规则，False表示不符合规则
    """
    num = len(queen_tup)
    for index, queen in enumerate(queen_tup):

        if new_queen == queen:  # 判断列号是否相等
            return False
        if abs(new_queen - queen) == num - index:  # 判断列号之差绝对值是否与行号之差相等
            return False

    return True


def arrange_queen(num, queen_tup=list()):
    """
    :param num:棋盘的的行数，当然数值也等于棋盘的列数
    :param queen_tup: 设置一个空队列，用于保存符合规则的棋子的信息
    """

    for new_queen in range(num):  # 遍历一行棋子的每一列

        if is_rule(queen_tup, new_queen):  # 判断是否冲突

            if len(queen_tup) == num - 1:  # 判断是否是最后一行
                yield [new_queen]  # yield关键字

            else:
                # 若果不是最后一行，递归函数接着放置棋子
                for result in arrange_queen(num, queen_tup + [new_queen]):
                    yield [new_queen] + result


for i in arrange_queen(8):
    print(i)
```

- **迷宫问题**

设有8*8的方格迷宫，入口和出口分别在左上角和右上角，迷宫格子中分别有0和1,1代表不能走，迷宫走的规则如图。当迷宫给出后，找到一条从入口到出口的通路。



- 思路：
  首先为了解决深度优先遍历出界的情况，将已有的迷宫图周围加上一圈1。
  建立抽象栈，首先将入点（初始点）入栈，然后将其置为-1表示已经走过。
  找下一个为0的点入栈；若最后无法出去，那就弹栈回退找其他为0 的点，

若找到则入栈，若找不到则继续弹栈找其他为0 的点。
最后栈内保存的就是入口到出口的路径坐标，从栈顶循环输出即可。

```
 -*- coding: utf-8 -*-
# @Time : 2019/10/19 10:26
# @Author : BaoBao
# @Mail : baobaotql@163.com
# @File : mzae.py
# @Software: PyCharm
import numpy as np

maze =np.array ([
    [0, 0, 0, 0, 0, 0, 1, 0],
    [1, 0, 1, 1, 1, 1, 1, 0],
    [0, 1, 0, 0, 1, 0, 0, 1],
    [0, 0, 1, 1, 0, 1, 0, 1],
    [0, 1, 0, 0, 0, 1, 1, 0],
    [0, 1, 1, 1, 1, 1, 0, 1],
    [0, 0, 1, 1, 1, 0, 1, 1],
    [1, 1, 0, 0, 1, 0, 0, 0],
])

p = np.array([1, 1, 1, 1, 1, 1, 1, 1])
q = np.array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])

a = np.insert(maze, 0, values=p, axis=1)
b = np.insert(a, 9,values=p, axis=1)

c = np.insert(b, 0, values=q, axis=0)
d = np.insert(c, 9, values=q, axis=0)

print("初始化迷宫: ")
print(d)#为解决越界问题 初始化迷宫

dirs = [lambda x, y: (x + 1, y),
        lambda x, y: (x - 1, y),
        lambda x, y: (x, y - 1),
        lambda x, y: (x, y + 1),
        lambda x, y: (x + 1, y - 1),
        lambda x, y: (x + 1, y + 1),
        lambda x, y: (x - 1, y - 1),
        lambda x, y: (x - 1, y + 1)
        ]

def mpath(x1, y1, x2, y2):
    stack = []      #建立抽象栈
    stack.append((x1, y1))      #加入初始点
    d[x1][y1] = -1       #表示已经走过
    while len(stack) > 0:
        curNode = stack[-1]
        if curNode == (x2, y2):
            print(stack)
            return True

        for dir in dirs:
            nextNode = dir(curNode[0], curNode[1])
            if d[nextNode[0]][nextNode[1]] == 0 : #找到了下一个
                stack.append(nextNode)
                d[nextNode[0]][nextNode[1]] = -1  # 标记为已经走过，防止死循环
                break

        else:   #八个方向都没找到
            d[curNode[0]][curNode[1]] = -1  # 死路一条,标记下次不走
            stack.pop()  #回溯

    print("没有路")
    return False

mpath(1,1,8,8)
```

运行截图：

```
C:\Users\79453\Anaconda3\python.exe "D:/华师工程中心/研一/课程  算法设计/coding tests/maze.py"
初始化迷宫:
[[1 1 1 1 1 1 1 1 1 1]
 [1 0 0 0 0 0 0 1 0 1]
 [1 1 0 1 1 1 1 1 0 1]
 [1 0 1 0 0 1 0 0 1 1]
 [1 0 0 1 1 0 1 0 1 1]
 [1 0 1 0 0 0 1 1 0 1]
 [1 0 1 1 1 1 1 0 1 1]
 [1 0 0 1 1 1 0 1 1 1]
 [1 1 1 0 0 1 0 0 0 1]
 [1 1 1 1 1 1 1 1 1 1]]
[(1, 1), (1, 2), (2, 2), (3, 1), (4, 1), (5, 1), (4, 2), (5, 3), (5, 4), (5, 5), (4, 5), (3, 6), (3, 7), (4, 7), (5, 8), (6, 7), (7, 6), (8, 6), (8, 7), (8, 8)]

Process finished with exit code 0
```