回溯算法：从电影蝴蝶效应中学习回溯算法的核心思想

在前面的章节中，图的深度优先搜索算法利用的就是回溯算法想，这个算法思想非常简单，但是应用很广泛，但是很多软件开发场景中都使用到了，比如正则表达式算法、编译原理中的语法分析等；

除此之外还有数独，八皇后，0-1背包问题，图的着色旅行商问题，全排列问题等等；

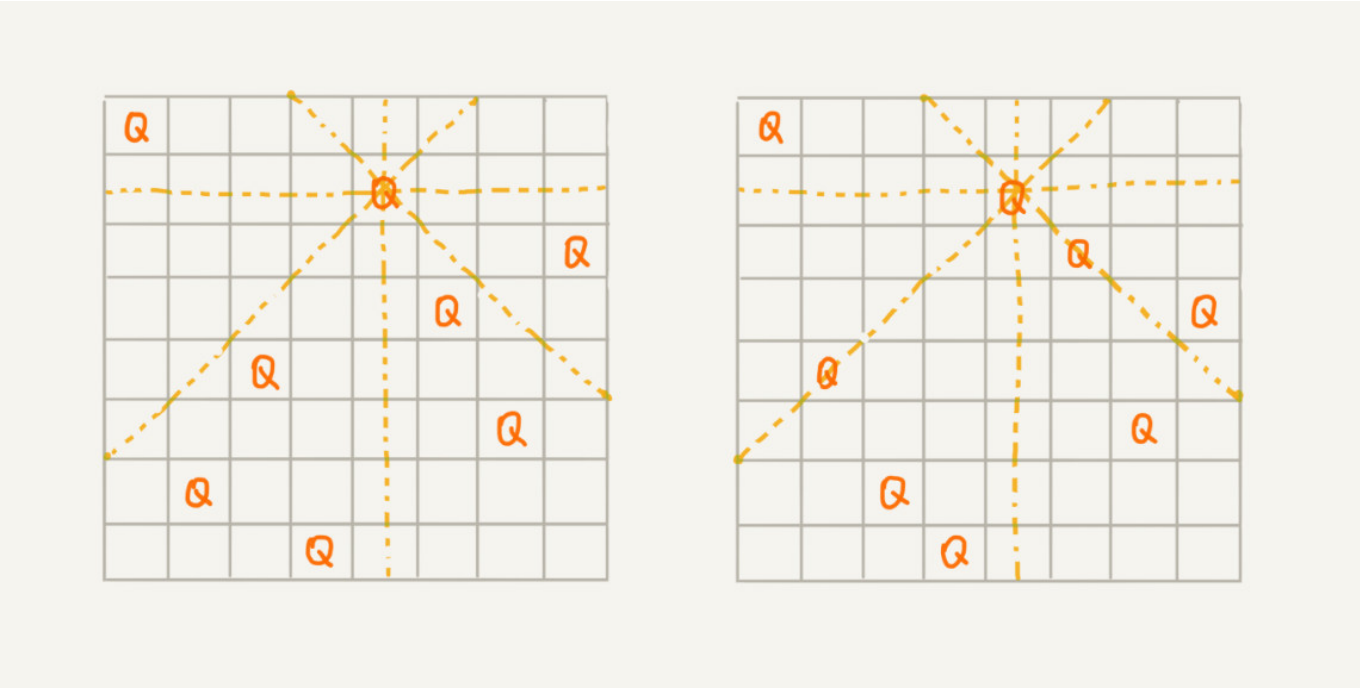
如何理解“回溯算法”

笼统的讲，回溯算法很多时候都应用在搜索这类问题上，不过这里说的搜索并不是狭义的图的搜索，而是在一组可能的解中搜索满足期望的解；

回溯的处理思想，优点类似于枚举搜索，我们枚举所有的解，找到满足期望的解，为了有规律的枚举可能的解，避免遗漏和重复，我们把问题的求解过程分为很多阶段。每个阶段我们都会面对一个岔路口，我们先任意选择一条路走，当发现这条路走不通的时候，就回退到上一个岔路口，另选一条走法继续走；

一个比较经典的问题：八皇后的问题；

我们有一个8*8的棋盘，希望往里面放8个棋子，每个棋子所在的行列对角线都不能有另一个棋子。你可以看到下面的画，第一幅画就是满足条件的一种方法，第二幅就不满足条件，八皇后问题就是期望找到所有满足这种要求的放棋子方法。



我们把问题分成8个阶段，依次将8个棋子放到第一行，第二行到第八行，在放置的过程中，我们不断的检查当前放法，是否满足要求，如果满足就跳到下一行继续放棋子；不满足就再换一种放法；

翻译成代码如下：

```
//全局变量，下标代表行，值表示queen存储在那一列
int[] result = new int[8];
```

```
/**
 *调用方式 cal8queens(0)
 * @param row
 */
public void cal8queens(int row){
    //8个棋子都放好了
    if(row == 8){
        printQueens(result);
        return ;
    }
    //每一行都有8个方法
    for(int col = 0; col < 8;col++){
        if(isOk(row,col)){ //有些放法不满足要求
            result[row] = col; //第row行的棋子放在了col行
            cal8queens(row+1); //考察下一行
        }
    }
}

/**
 * 判断row行棋子放在col列是否合适
 * @param row
 * @param col
 * @return
 */
public boolean isOk(int row,int col){
    int leftup = col - 1;
    int rightup = col + 1;
    for(int i = row - 1; i >= 0;i--){
        //考察正上方是否合适
        if(result[i] == col){
            return false;
        }
        //考察左上方是否合适
        if(leftup >= 0){
            if(result[i] == leftup){
                return false;
            }
        }
        //考察右上方是否合适
        if(rightup < 8){
            if(result[i] == rightup){
                return false;
            }
        }
        leftup--;
        rightup++;
    }
    return true;
}
```

```
/**
 * 打印
 * @param result
 */
private void printQueens(int[] result){
    for(int row = 0; row < 8; row++){
        for(int col = 0; col < 8; col++){
            if(result[row] == col){
                System.out.print("Q ");
            }else{
                System.out.print("* ");
            }
        }
        System.out.println();
    }
    System.out.println();
}
```

两个回溯算法中的经典应用

1、0-1背包

0-1背包问题有很多变体，介绍一种比较基础的，我们有一个背包，背包的总承重是Wkg，现在我们有 n 个物品，每个物品的重量不等，并且不可分割。我们现在期望选择几件物品，装载到背包中。在不超过背包所能装载重量的前提下，如何让背包中物品的总重量最大?

对于每个物品来说装进背包或者不装进背包，对于n个物品总的装法就有2^n种，去掉重量超过Wkg的，从剩下的装法中选择重量最接近Wkg的。不过我们如何才能不重复的穷举这2^n种方法呢?

回溯的方法，我们将物品一次放进排列，整个问题被分解成n阶段，每个阶段对应一个物品如何选择，先对第一个物品进行处理，选择装进去或者不装进去，然后在递归的处理下面的物品;

代码如下:

```
/**
 * f(0,0,a,10,100)
 * @param i 表示考察到那个物品了
 * @param cw 表示当前已经装进去的物品总重量
 * @param items 表示每个物品的重量
 * @param n 表示物品个数
 * @param w 背包重量
 */
public void f(int i,int cw,int[] items,int n,int w){
    //cw == w表示装满了，i== n表示考察完所以的物品了
    if(cw == w || i == n){
        if(cw > maxW) maxW = cw;
        return;
    }
    f(i+1,cw,items,n,w);
```

```
        // 已经超过可以背包承受的重量的时候，就不要再装了
        if(cw + items[i] <= w){
            f(i+1,cw+items[i],items,n,w);
        }
    }
```

2、正则表达式

正则表达式中，最重要的就是通配符，通配符结合在一起，可以表达非常丰富的语义。为了方便讲解，我假设正则表达式中只包含“*”和“?”这两种通配符，并且对这两个通配符的语义稍微做些改变，其中，“*”匹配任意多个（大于等于 0 个）任意字符，“?”匹配零个或者一个任意字符。基于以上背景假设，我们看下，如何用回溯算法，判断一个给定的文本，能否跟给定的正则表达式匹配？

我们依次考察正则表达式中的每个字符，当是非通配符的时候，就直接跟文本字符进行匹配，如果相同就继续往下处理，如果不相同就回溯；

如果遇到特殊字符的时候，我们就有多种处理方式，这就是所谓的岔路口，比如“*”有多种匹配方案，可以匹配任意个文本串中的字符，我们就随意选择一种匹配方案，然后继续考察剩下的字符，如果中途发现无法匹配下去，我们就回到这个岔路口，重新选择一种方案，然后在继续匹配下去；

代码如下：

```
public class Pattern {
    private boolean matched = false;
    private char[] pattern; //正则表达式
    private int plen; //正则表达式长度

    public Pattern(char[] pattern,int plen){
        this.pattern = pattern;
        this.plen = plen;
    }

    public boolean match(char[] text,int tlen){
        matched = false;
        rmatch(0,0,text,tlen);
        return matched;
    }

    private void rmatch(int ti,int pj,char[] text,int tlen){
        if(matched)return;
        if(pj == plen){ //正则表达式结尾了
            if(ti == tlen){ // 文本串也结尾了
                matched = true;
            }
            return;
        }
        if(pattern[pj] == '*'){ //匹配任意个字符
            for(int k = 0; k < tlen - ti; k++){
                rmatch(ti + k,pj + 1,text,tlen);
```

```
            }
        }else if(pattern[pj] == '?'){ //匹配0个或者1个字符
            rmatch(ti,pj+1,text,tlen);
            rmatch(ti+1,pj+1,text,tlen);
        }else if(ti < tlen && pattern[pj] == text[ti]){ //字符匹配才行
            rmatch(ti+1,pj+1,text,tlen);
        }
    }
}
```

课后思考

现在我们对今天讲到的 0-1 背包问题稍加改造，如果每个物品不仅重量不同，价值也不同。如何在不超过背包重量的情况下，让背包中的总价值最大?