

讲堂 > 数据结构与算法之美 > 文章详情

39 | 回溯算法：从电影《蝴蝶效应》中学习回溯算法的核心思想

2018-12-24 王争



39 | 回溯算法：从电影《蝴蝶效应》中学习回溯算法的核心思想

朗读人：修阳 09'18" | 8.52M

我们在[第 31 节](#)提到，深度优先搜索算法利用的是回溯算法思想。这个算法思想非常简单，但是应用却非常广泛。它除了用来指导像深度优先搜索这种经典的算法设计之外，还可以用在很多实际的软件开发场景中，比如正则表达式匹配、编译原理中的语法分析等。

除此之外，很多经典的数学问题都可以用回溯算法解决，比如数独、八皇后、0-1 背包、图的着色、旅行商问题、全排列等等。既然应用如此广泛，我们今天就来学习一下这个算法思想，看看它是如何指导我们解决问题的。

如何理解“回溯算法”？

在我们的一生中，会遇到很多重要的岔路口。在岔路口上，每个选择都会影响我们今后的人生。有的人在每个岔路口都能做出最正确的选择，最后生活、事业都达到了一个很高的高度；而有的人一路选错，最后碌碌无为。如果人生可以量化，那如何才能在岔路口做出最正确的选择，让自己的人生“最优”呢？

我们可以借助前面学过的贪心算法，在每次面对岔路口的时候，都做出看起来最优的选择，期望这一组选择可以使得我们的人生达到“最优”。但是，我们前面也讲过，贪心算法并不一定能得到最优解。那有没有什么办法能得到最优解呢？

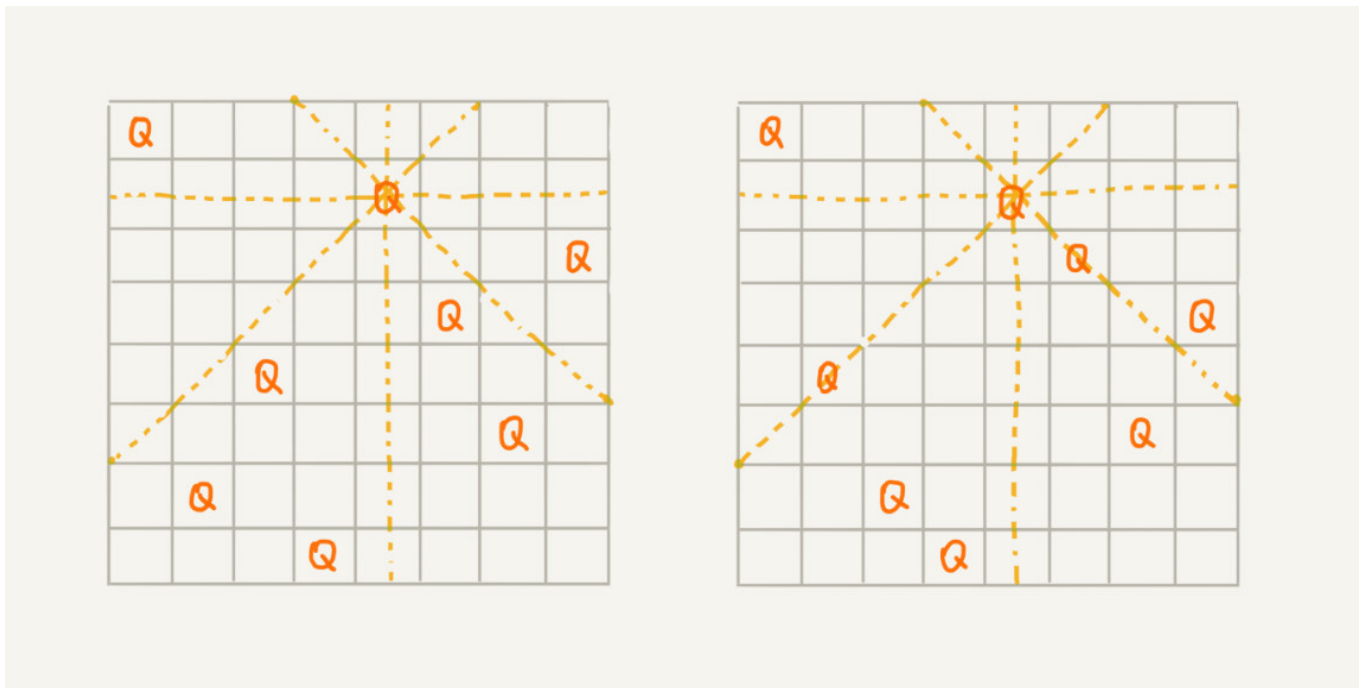
2004 年上映了一部非常著名的电影《蝴蝶效应》，讲的就是主人公为了达到自己的目标，一直通过回溯的方法，回到童年，在关键的岔路口，重新做选择。当然，这只是科幻电影，我们的人生是无法倒退的，但是这其中蕴含的思想其实就是回溯算法。

笼统地讲，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，并不是狭义的指我们前面讲过的图的搜索算法，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，有点类似枚举搜索。我们枚举所有的解，找到满足期望的解。为了有规律地枚举所有可能的解，避免遗漏和重复，我们把问题求解的过程分为多个阶段。每个阶段，我们都会面对一个岔路口，我们先随意选一条路走，当发现这条路走不通的时候（不符合期望的解），就回退到上一个岔路口，另选一种走法继续走。

理论的东西还是过于抽象，老规矩，我还是举例说明一下。我举一个经典的回溯例子，我想你可能已经猜到了，那就是八皇后问题。

我们有一个 8x8 的棋盘，希望往里放 8 个棋子（皇后），每个棋子所在的行、列、对角线都不能有另一个棋子。你可以看我画的图，第一幅图是满足条件的一种方法，第二幅图是不满足条件的。八皇后问题就是期望找到所有满足这种要求的放棋子方式。



我们把这个问题划分成 8 个阶段，依次将 8 个棋子放到第一行、第二行、第三行……第八行。在放置的过程中，我们不停地检查当前的方法，是否满足要求。如果满足，则跳到下一行继续放置棋子；如果不满足，那就再换一种方法，继续尝试。

回溯算法非常适合用递归代码实现，所以，我把八皇后的算法翻译成代码。我在代码里添加了详细的注释，你可以对比着看下。如果你之前没有接触过八皇后问题，建议你自己用熟悉的编程语言实现一遍，这对你理解回溯思想非常有帮助。

复制代码

```
1 int[] result = new int[8]; // 全局或成员变量，下标表示行，值表示 queen 存储在哪一列
2 public void cal8queens(int row) { // 调用方式: cal8queens(0);
3     if (row == 8) { // 8 个棋子都放置好了，打印结果
4         printQueens(result);
5         return; // 8 行棋子都放好了，已经没法再往下递归了，所以就 return
6     }
7     for (int column = 0; column < 8; ++column) { // 每一行都有 8 中放法
8         if (isOk(row, column)) { // 有些放法不满足要求
9             result[row] = column; // 第 row 行的棋子放到了 column 列
10            cal8queens(row+1); // 考察下一行
11        }
12    }
13 }
14
15 private boolean isOk(int row, int column) { // 判断 row 行 column 列放置是否合适
16     int leftup = column - 1, rightup = column + 1;
17     for (int i = row-1; i >= 0; --i) { // 逐行往上考察每一行
18         if (result[i] == column) return false; // 第 i 行的 column 列有棋子吗?
19         if (leftup >= 0) { // 考察左上对角线: 第 i 行 leftup 列有棋子吗?
20             if (result[i] == leftup) return false;
21         }
22         if (rightup < 8) { // 考察右上对角线: 第 i 行 rightup 列有棋子吗?
23             if (result[i] == rightup) return false;
24         }
25         --leftup; ++rightup;
26     }
27     return true;
28 }
29
30 private void printQueens(int[] result) { // 打印出一个二维矩阵
31     for (int row = 0; row < 8; ++row) {
32         for (int column = 0; column < 8; ++column) {
33             if (result[row] == column) System.out.print("Q ");
34             else System.out.print("* ");
35         }
36         System.out.println();
37     }
38     System.out.println();
39 }
```

两个回溯算法的经典应用

回溯算法的理论知识很容易弄懂。不过，对于新手来说，比较难的是用递归来实现。所以，我们再通过两个例子，来练习一下回溯算法的应用和实现。

1.0-1 背包

0-1 背包是非常经典的算法问题，很多场景都可以抽象成这个问题模型。这个问题的经典解法是动态规划，不过还有一种简单但没有那么高效的解法，那就是今天讲的回溯算法。动态规划的解法我下一节再讲，我们先来看下，如何用回溯法解决这个问题。

0-1 背包问题有很多变体，我这里介绍一种比较基础的。我们有一个背包，背包总的承载重量是 $W\text{kg}$ 。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？

实际上，背包问题我们在贪心算法那一节，已经讲过一个了，不过那里讲的物品是可以分割的，我可以装某个物品的一部分到背包里面。今天讲的这个背包问题，物品是不可分割的，要么装要么不装，所以叫 0-1 背包问题。显然，这个问题已经无法通过贪心算法来解决。我们现在来看看，用回溯算法如何来解决。

对于每个物品来说，都有两种选择，装进背包或者不装进背包。对于 n 个物品来说，总的装法就有 2^n 种，去掉总重量超过 $W\text{kg}$ 的，从剩下的装法中选择总重量最接近 $W\text{kg}$ 的。不过，我们如何才能不重复地穷举出这 2^n 种装法呢？

这里就可以用回溯的方法。我们可以把物品依次排列，整个问题就分解为了 n 个阶段，每个阶段对应一个物品怎么选择。先对第一个物品进行处理，选择装进去或者不装进去，然后再递归地处理剩下的物品。描述起来很费劲，我们直接看代码，反而会更加清晰一些。

这里还稍微用到了一点搜索剪枝的技巧，就是当发现已经选择的物品的重量超过 $W\text{kg}$ 之后，我们就停止继续探测剩下的物品。你可以看我写的具体的代码。

```
1 public int maxW = Integer.MIN_VALUE; // 存储背包中物品总重量的最大值
2 // cw 表示当前已经装进去的物品的重量和；i 表示考察到哪个物品了；
3 // w 背包重量；items 表示每个物品的重量；n 表示物品个数
4 // 假设背包可承受重量 100，物品个数 10，物品重量存储在数组 a 中，那可以这样调用函数：
5 // f(0, 0, a, 10, 100)
6 public void f(int i, int cw, int[] items, int n, int w) {
7     if (cw == w || i == n) { // cw==w 表示装满了；i==n 表示已经考察完所有的物品
8         if (cw > maxW) maxW = cw;
9         return;
10    }
11    f(i+1, cw, items, n, w);
12    if (cw + items[i] <= w) { // 已经超过可以背包承受的重量时候，就不要再装了
13        f(i+1, cw + items[i], items, n, w);
14    }
15 }
```

[复制代码](#)

2. 正则表达式

看懂了 0-1 背包问题，我们再来看另外一个例子，正则表达式匹配。

对于一个开发工程师来说，正则表达式你应该不陌生吧？在平时的开发中，或多或少都应该用过。实际上，正则表达式里最重要的一种算法思想就是回溯。

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，当是非通配符时，我们就直接跟文本的字符进行匹配，如果相同，则继续往下处理；如果不同，则回溯。

如果遇到特殊字符的时候，我们就有多种处理方式了，也就是所谓的岔路口，比如“*”有多种匹配方案，可以匹配任意个文本串中的字符，我们就先随意的选择一种匹配方案，然后继续考察剩下的字符。如果中途发现无法继续匹配下去了，我们就回到这个岔路口，重新选择一种匹配方案，然后再继续匹配剩下的字符。

有了前面的基础，是不是这个问题就好懂多了呢？我把这个过程翻译成了代码，你可以结合着一块看下，应该有助于你理解。

[复制代码](#)

```
1 public class Pattern {
2     private boolean matched = false;
3     private char[] pattern; // 正则表达式
4     private int plen; // 正则表达式长度
5
6     public Pattern(char[] pattern, int plen) {
7         this.pattern = pattern;
8         this.plen = plen;
9     }
10
11     public boolean match(char[] text, int tlen) { // 文本串及长度
12         matched = false;
13         rmatch(0, 0, text, tlen);
14         return matched;
15     }
16
17     private void rmatch(int ti, int pj, char[] text, int tlen) {
18         if (matched) return; // 如果已经匹配了，就不要继续递归了
19         if (pj == plen) { // 正则表达式到结尾了
20             if (ti == tlen) matched = true; // 文本串也到结尾了
21             return;
22         }
23         if (pattern[pj] == '*') { // * 匹配任意个字符
24             for (int k = 0; k <= tlen-ti; ++k) {
25                 rmatch(ti+k, pj+1, text, tlen);
26             }
27         }
28     }
29 }
```

```
27     } else if (pattern[pj] == '?') { // ? 匹配 0 个或者 1 个字符
28         rmatch(ti, pj+1, text, tlen);
29         rmatch(ti+1, pj+1, text, tlen);
30     } else if (ti < tlen && pattern[pj] == text[ti]) { // 纯字符匹配才行
31         rmatch(ti+1, pj+1, text, tlen);
32     }
33 }
34 }
```

内容小结

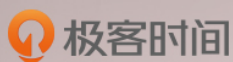
回溯算法的思想非常简单，大部分情况下，都是用来解决广义的搜索问题，也就是，从一组可能的解中，选择一个满足要求的解。回溯算法非常适合用递归来实现，在实现的过程中，剪枝操作是提高回溯效率的一种技巧。利用剪枝，我们并不需要穷举搜索所有的情况，从而提高搜索效率。

尽管回溯算法的原理非常简单，但是却可以解决很多问题，比如我们开头提到的深度优先搜索、八皇后、0-1 背包问题、图的着色、旅行商问题、数独、全排列、正则表达式匹配等等。如果感兴趣的话，你可以自己搜索研究一下，最好还能用代码实现一下。如果这几个问题都能实现的话，你基本就掌握了回溯算法。

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？

欢迎留言和我分享，也欢迎点击“[请朋友读](#)”，把今天的内容分享给你的好友，和他一起讨论、学习。



数据结构与算法之美

为工程师量身打造的数据结构与算法私教课

王争

前 Google 工程师



上一篇 不定期福利第三期 | 测一测你的算法阶段学习成果

写留言

精选留言



slvher

👍 2

0-1 背包问题的回溯实现技巧：

第 11 行的递归调用表示不选择当前物品，直接考虑下一个（第 $i+1$ 个），故 cw 不更新

第 13 行的递归调用表示选择了当前物品，故考虑下一个时， cw 通过入参更新为 $cw + items[i]$

函数入口处的 if 分支表明递归结束条件，并保证 $maxW$ 跟踪所有选择中的最大值

2018-12-24



你有资格吗?

👍 1

打个卡，终于跟上更新进度了

2018-12-24



纯洁的憎恶

👍 1

回溯算法本质上就是枚举，优点在于其类似于摸着石头过河的查找策略，且可以通过剪枝少走冤枉路。它可能适合应用于缺乏规律，或我们还不了解其规律的搜索场景中。

2018-12-24

作者回复



2018-12-24



纯洁的憎恶

👍 0

0-1背包的递归代码里第11行非常巧妙，它借助回溯过程，实现了以每一个可能的物品，作为第一个装入背包的，以尝试所有物品组合。但如果仅按从前向后执行的顺序看，是不太容易发现这一点的。

2018-12-24



你有资格吗?

👍 0

背包问题代码没给全啊

2018-12-24

作者回复

比较全了吧

2018-12-24



Kudo

0

这节看得有点糊涂。关于八皇后问题，文中给出的代码有用到回溯的思想吗？恕我愚钝，咋感觉就是一路执行到底了，没看到往回返的过程啊？

2018-12-24

作者回复

一路到底 咋找出所有解的啊

2018-12-24



blacknhole

0

有个疑问：

示例代码中的以下部分在功能上是没问题的，但在语意上不够严谨：

```
else if (pattern[pj] == '?') {  
    rmatch(ti, pj+1, text, tlen);  
    rmatch(ti+1, pj+1, text, tlen);  
}
```

语句 `rmatch(ti+1, pj+1, text, tlen);` 在 `ti` 等于 `tlen` 时，`ti + 1` 大于 `tlen`。并有可能在下次递归调用中 `ti + 1` 变得更大。所以改成这样可能更严谨：

```
else if (pattern[pj] == '?') {  
    rmatch(ti, pj+1, text, tlen);  
    if (ti < tlen) {  
        rmatch(ti+1, pj+1, text, tlen);  
    }  
}
```

2018-12-24

作者回复

嗯嗯 我知道的 我写的时候就意识到你说的优化了

2018-12-24



siegfried

0

回溯就是暴力枚举的解法吧？遍历所有情况，当满足情况就停止遍历（剪枝）。

2018-12-24

作者回复

是的

2018-12-24



林

0

正则表达式第20行，`ti <= tlen`： `match = True`。如果是等于那必须每次都匹配到最后吗？

2018-12-24





NeverMore

👍 0

老师思考题应该就是下章的动态规划了吧。

2018-12-24

作者回复

回溯也能解决的

2018-12-24



陈子昭czz

👍 0

背包问题：if (cw > maxW) maxW = cw;这样不是超重了嘛？还有代码里cw一直没有赋值操作啊

2018-12-24



铁丑-王立丰

👍 0

0-1背包问题的例子中，第11行不需要吧

2018-12-24