由打家劫舍扩展问题看树形DP

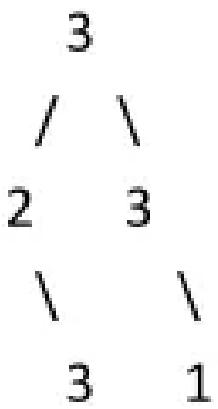
原创 张忠诚 LOA算法学习笔记 2021-02-28 18:20

作业的打家劫舍问题想必大家已经非常熟悉了，问题是说一个地区有很多房屋，每个房屋有一定的现金，一个小偷计划在该地区抢劫，由于相邻的房屋有报警系统所以他不能抢劫相邻的屋子，问小偷在该地区能抢到的最大金额数。显然该问题有个变数，就是房屋的排列形式，作业出现了两种，一种是线性排列一种是环形排列。此外还有一种排列方式，就是以二叉树的形式进行排列，这时仍可以通过dp来求解，这种在树结构的dp我们通常称之为树形dp，以下是leetcode中打家劫舍3的题目描述。

01 题目描述

在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为“根”。除了“根”之外，每栋房子有且只有一个“父”房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

示例：如果输入为如下所示的一颗二叉树，那么小偷能抢劫的最大金额为7，即在第一层和第三层的房子抢劫3+3+1=7。



02 问题分析与实现

让我们简单回忆一下线性排列的的解题思路，在线性排列中我们定义了一个一维的数组dp[i]表示前i间屋子所能抢到的最大值，之后遍历所有的房子，每个房子都有抢或者不抢两种选择，如果抢了第i间房子就不能抢第i-1间，此时的dp[i]就是dp[i-2]加上第i间房子的价值，如果不抢第i房子，此时的dp[i]就等于dp[i-1], 而最终的dp[i]就是上述两种选择取最大值。那么当房屋排列形式转化为二叉树时，我们又应该如何解决该问题呢?

借鉴上面线性排列的思路我们很容易想出如下所示的思路，定义dp[i]表示以第i间屋子为根的子树能抢到的最大值，当抢该间屋子的时候，位于它儿子节点的屋子就不能抢，此时抢的价值为

$$M[i] + sum_{j \in grandson}(dp[j])$$

即当前屋子的价值加上其所有孙子节点为根的节点抢到的最大值，如果不抢该间屋子，那么就是他两个儿子节点能抢到的最大值之和，即为

$$sum_{j \in son}(dp[j])$$

这个思路暂时看起来是没有问题的，那么让我们给出它相应的代码看看吧。

```
int rob(TreeNode* root) {
        if (root == NULL) return 0;
        int robRes = root->val +
        (root->left == NULL? 0 : rob(root->left->left) + rob(root->left->right)) +
        (root->right == NULL? 0 : rob(root->right->left) + rob(root->right->right));
        int notRobRes = rob(root->left) + rob(root->right);
        return max(robRes, notRobRes);
    }
```

阅读上面的代码就会发现出了问题，代码交到leetcode上显然是对的，但是效率却不高，因为出现了很多的重复计算。原来我们前面的思路在考虑某一节点的时候，不仅考虑到了子节点还考虑到了孙子节点，这就导致到子节点成为父节点的时候，原孙子节点就成了子节点，从导致其计算了两遍。可是为什么呢？线性排列的时候我们不也是考虑了i-1和i-2吗？这是因为线性排列的时候我们是自底向上进行计算的，当我们考虑i的时候i-1和i-2都已经计算完了，直接从数组里取就可以了，但是由于树的遍历需要自顶向下递归进行，如果我们同时考虑三层的话，就会导致子问题的重复计算。我个人认为这是树状dp与线性dp思考问题时是最大的不同。

那么我们应该怎么避免子问题的重复计算呢？一种很简单的思路就是用map保存计算结果，但是这种方法的实际性能并不高。所以我们只能尝试换一种思路，既然同时考虑子节点和孙子节点会导致重复计算，那我们能不能不考虑孙子节点？我们前面之所以要考虑孙子节点，是因为我们不知道当前节点偷或者不偷，那么我们就可以再增加一维用来记录当前节点偷或者不偷这两个状态。即定义dp[i][0]为当前节点不偷时得到的最大钱数，dp[i][1]为当前节点偷时得到的最大钱数，最终的结果就是dp[root][0]和dp[root][1]取一个最大值。接下来仍然是分偷或不偷两种情况讨论。

1. 当前节点选择不偷时，当前节点能得到的最大钱数=左孩子能得到的最大钱数+右孩子能得到的最大钱数。
2. 当前节点选择偷时，当前节点能得到的最大钱数=左孩子不偷时得到的最大钱数+右孩子不偷时得到的最大钱数+当前节点的钱数。

根据上面的分析，我们可以写出相应的状态转移方程，如下所示：

$$\begin{cases} dp[i][0] = max(dp[i.left][0], dp[i.left][1]) + max(dp[i.right][0], dp[i.right][1]) \\ dp[i][1] = dp[i.left][0] + dp[i.right][0] + M[i] \end{cases}$$

可以看到通过增加一维的方式，我们避免了考虑孙子节点，当前节点只跟他的子节点有关，从而避免了重复计算。以下是该思路对应的代码实现：

```
pair<int, int> dp(TreeNode *root) {
        if (root == NULL) return make_pair(0, 0);
        pair<int, int> leftRes = dp(root->left);
        pair<int, int> rightRes = dp(root->right);
        int notRobRes = max(leftRes.first, leftRes.second) +
max(rightRes.first, rightRes.second);
        int robRes = leftRes.first + rightRes.first + root->val;
        return make_pair(notRobRes, robRes);
    }
    int rob(TreeNode* root) {
        pair<int, int> res = dp(root);
        return max(res.first, res.second);
    }
```

03 时间复杂度分析

通过上面的分析可知，我们求解该问题时相当于对二叉树进行了一次遍历，故时间复杂度为O(n)，n为二叉树的节点数目。

04 总结

树由于其无环的特性，很适合使用dp进行求解，但是由于树遍历方式的约束，有些dp问题不再像线性dp那么直观，需要我们考虑一下怎么定义dp数组的含义以避免子问题的重复计算。此外，除了本题目二叉树的树形dp还有多叉树的树形dp，我们考试第五题的第二问就是一个多叉树的树形dp问题，大家可以回忆一下。