动态规划思想及经典算法

原创 尚科彤 LOA算法学习笔记 2021-02-09 13:31

01 DP基本思想

对于给定一个问题，可以将其分解成不同子问题，之后对对其不同子问题 进行求解，再合并子问题解以得出原问题的解。
通常情况下，子问题具有一定的相似性，因此，动态规划试图仅仅解决每个子问题一次，从而减少计算量：一旦某个给定的子问题的解已经算出，则将其存储，以便下次需要同一个子问题时直接查表使用。
分治与动态规划
共同点：二者都要求原问题具有最优子结构，都是将原问题进行分解，分而治之，分解成若干个子问题，然后将子问题的解进行合并，形成原问题的解。
不同点：

- 分治法分解后的子问题是相互独立的，最后通过递归进行合并
- 动态规划分解后的子问题相互之间有联系，最后通过迭代来做

02 求解过程

1.找出最优解的性质，刻画其结构特征和最优子结构特征，将原问题分解成若干个子问题；
把原问题分解为若干个子问题，子问题和原问题形式相同或类似，只不过规模变小了。子问题都解决，原问题即解决，子问题的解一旦求出就会被保存，所以每个子问题只需求解一次。
2.递归地定义最优值，刻画原问题解与子问题解间的关系，确定状态；
在用动态规划解题时，我们往往将和子问题相关的各个变量的一组取值，称之为一个“状态”。一个“状态”对应于一个或多个子问题， 所谓某个“状态”下的“值”，就是这个“状态”所对应的子问题的解。所有“状态”的集合，构成问题的“状态空间”。“状态空间”的大小，与用动态规划解决问题的时间复杂度直接相关。
3.以自底向上的方式计算出各个子问题、原问题的最优值，并避免子问题的重复计算；
定义出什么是“状态”，以及在该“状态”下的“值”后，就要找出不同的状态之间如何迁移--即如何从一个或多个“值”已知的“状态”，求出另一个“状态”的“值”(递推型)。
4.根据计算最优值时得到的信息，构造最优解，确定转移方程;
状态的迁移可以用递推公式表示，此递推公式也可被称作“状态转移方程”。

03 具体问题

3.1 01背包

1) 问题描述

有 N 件物品和一个容量为 V 的背包。第i件物品的费用是 c[i]，价值是 w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

2) 求解过程

f[i][v] 表示前 i 件物品恰放入一个容量为 v 的背包可以获得的最大价值。则其状态转移方程便是：

$$f[i][v] = max\{f[i-1][v], f[i-1][v-c[i]] + w[i]\}$$

将"前 i 件物品放入容量为 v 的背包中"这个子问题，若只考虑第 i 件物品的策略（放或不放），那么就可以转化为一个只牵扯前 i-1 件物品的问题。如果不放第 i 件物品，那么问题就转化为"前 i-1 件物品放入容量为 v 的背包中"；如果放第 i 件物品，那么问题就转化为"前 i-1 件物品放入剩下的容量为 v-c[i] 的背包中"，此时能获得的最大价值就是 f [i-1][v-c[i]] 再加上通过放入第 i 件物品获得的价值 w[i] 。

3) 伪代码

```
void package()
{
  for(int i = 1; i <= N; i++)    //i对应第i个物品，i的循环放外面，这个是关键点，一行一行的写入数据。因为V[i
  {
    for(int j = 1; j <= M; j++)  //j对应当前背包能承受重量为j
    {
      if(j >= weight[i])
      {
        V[i][j] = max(V[i - 1][j], V[i - 1][j - weight[i]] + value[i]);
      }
      else
      {
        V[i][j] = V[i - 1][j];
      }
    }
  }
}
```

4) 空间优化

二位数组的空间复杂度显而易见为 O(nm) ， n 可能不大，但是实际应用的 w 可能会很大，这样造成了比较大的空间占用。而仔细观察发现，矩阵中的值只与当前值的左上角的矩阵里的值有关。并且是按行进行更新的，所以可以用一个一维数据就能进行状态的更替，不过要注意更新的方向。

所以依赖关系为：下面依赖上面，右边依赖左边；

如果正常地从左到右边，那么右边面等待更新的值需要的依赖（左边）就会被覆盖掉，所以应该每行从右边开始更新。

代码如下：

```
void packageOpt()
{
  for(int i = 1; i <= N; i++)
  {
    for (int j = M; j >= 1; j--)
    {
      if (j >= weight[i])
      {
        V[j] = max(V[j], V[j - weight[i]] + value[i]);
      }
    }
  }
}
```

3.2 字符匹配

1）问题描述

给定两个字符word1和word2，找到将word1和word2匹配的最大字符，所需操作的最小步数。（每个操作计为1步）。
对单词允许以下3种操作：
a）插入字符
b）删除字符
c）替换字符

2）求解过程

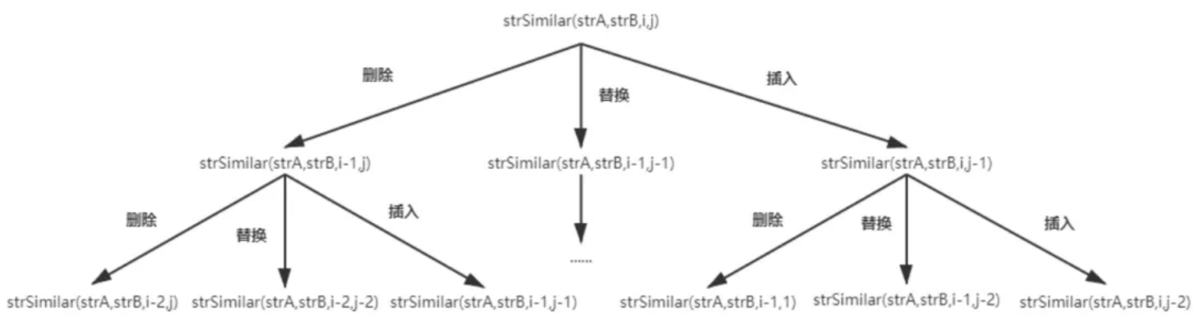
定义两个字符串之间的编辑距离：从字符串**str1**到**str2**的最少的操作次数。首先，编辑距离是不会大于**str1.length + str2.length**的（可以通过删除操作把两个字符串都转化为空串）。假设求字符**A**、**B**的编辑距离，考虑下面几种情况：
如果A[i] = B[j]，那么这时候还需要操作吗？
这个时候的删除和替换操作只会让情况变得更坏，而且插入操作不会使情况变得更好，只要计算 $A[2...lenA]$ 和 $B[2...lenB]$ 的距离就可以了，所以此时**F(i, j) = F(i-1, j-1)**。
如果A[i] != B[j]，怎么办呢？需要进行如下操作：
1. 删除A串的第一个字符，然后计算 $A[2...lenA]$ 和 $B[1...lenB]$ 的距离；
2. 删除B串的第一个字符，然后计算 $A[1...lenA]$ 和 $B[2...lenB]$ 的距离；
3. 修改A串的第一个字符为B串的第一个字符，然后计算 $A[2...lenA]$ 和 $B[2...lenB]$ 的距离；
4. 修改B串的第一个字符为A串的第一个字符，然后计算 $A[2...lenA]$ 和 $B[2...lenB]$ 的距离；
5. 增加B串的第一个字符到A串的第一个字符之前，然后计算 $A[1...lenA]$ 和 $B[2...lenB]$ 的距离；
6. 增加A串的第一个字符到B串的第一个字符之前，然后计算 $A[2...lenA]$ 和 $B[1...lenB]$ 的距离；

合并之后可以简化为以下三步：
1. 从 $F(i-1,j-1)$ 变过来，这时候只需要把 $A[i]$ 替换为 $B[j]$ 即可；
2. 从 $F(i-1,j)$ 变过来，这时候只需要将 $A[i]$ 删除即可；
3. 从 $F(i,j-1)$ 变过来，这时候只需要在 $A[i]$ 后插入字符 $B[j]$ 即可；

那么此时

$$F(i,j) = min\{F(i-1，\ j-1)，\ F(i-1，\ j)，\ F(i，\ j-1)\}+1$$

注：其中 $F(i,j)$ 表示 $A[0..i]$ 和 $B[0..j]$ 之间的编辑距离。
多步决策可如下表示：



3）伪代码

```
int strSimilar(char *strA, char *strB,int n,int m)
{
  int **A = new int *[n+1];
       for(int i = 0; i< n + 1; i++)
```

```
  {
    A[i] = new int[m+1];
  }
  A[0][0] = 0;
  A[1][0] = 1;
  A[0][1] = 1;
  for (int i = 1; i<=m; i++)
  {
    A[0][i] = i;
  }
  for (int i = 1; i<=n; i++)
  {
    A[i][0] = i;
  }
  for(int i = 0; i<n; i++)
  {
    for(int j= 0; j<m; j++)
    {
      if (strA[i] == strB[j])
      {
        A[i+1][j+1] = A[i][j];
      }
      else
      {
        A[i+1][j+1] = min(A[i+1][j], A[i][j+1], A[i][j]) + 1;
      }
    }
  }
  return A[n][m];
}
```

4) 空间优化

二维数组复杂度为 $O(nm)$，分析矩阵法的执行过程，会发现每次计算的时候只需要本行和上一行的结果，而和再之前的行无关，如果第 i 行已经完成计算，则第 i-1 行就不再需要了，所以可以只采用两行的矩阵来实现，当字符串长度大的时候可以有效的节省存储空间。