

动态规划补充

2023-2024 学年春夏学期高级数据结构与算法分析

吴一航

2024 年 4 月 22 日

动态规划的来由

1966年，Dynamic Programming 登上国际顶级期刊《Science》，理查德·贝尔曼（Richard Bellman）为唯一作者，该论文被引 28421 次。贝尔曼在他的自传中写道：“1950年秋季，我在兰德公司工作。我的第一个任务是为多阶段决策过程找到一个名称。”

“一个有趣的问题是，“动态规划”这个名字是从哪里来的？20世纪50年代不是搞数学研究的好年代。当时在华盛顿有一位非常有趣的人，他叫威尔逊（Charles E. Wilson, 1953-1957年任美国国防部长），是当时的美国国防部长，他对“研究”这个词有病态的恐惧和憎恨。提到研究，他的脸会涨得通红，如果有人在他面前用“研究”这个词，他就会变得很暴力。你可以想象他对数学这个词的感受。兰德公司受雇于空军，而空军的老板基本上是威尔逊。因此，我觉得我必须做点什么来瞒过威尔逊和空军，不让他们知道我实际上是在兰德公司做数学研究。”

动态规划的来由 (Cont'd)

“所以，给这个研究起个什么名字好呢？首先，我对 **planning**, **decision making** 和 **thinking** 比较感兴趣。但是由于种种原因，**planning** 这个词并不合适。因此，我决定使用 **programming** 这个词。我想让大家明白，这是动态的，是多级的，是时变的——我想，这个名字可以起到一石二鸟的效果。作为形容词，**dynamic** 还有一个很有趣的特性，那就是 **dynamic** 这个词不可能有贬义。我们不可能想出一些可能使它具有贬义的组合。因此，我认为 **dynamic programming** 是个好名字。这是一件连国会议员都不会反对的事情。所以我把它作为我研究活动的“保护伞。”

动态规划的范式

动态规划方法通常用来求解最优化问题，这类问题可以有太多可行解，每个解都有一个值，我们希望寻找具有最优值（最小值或最大值）的解。我们通常按如下 4 个步骤来设计一个动态规划算法：

- ① 刻画一个最优解的结构特征；
- ② 递归地定义最优解的值；
- ③ 计算最优解的值，通常采用自底向上的方法；
- ④ 利用计算出的信息构造一个最优解。

更精炼地，动态规划就是为一个具有所谓最优子结构性质（即原问题最优解可以由子问题最优解递推得到）的最优化问题寻找一个子问题到原问题的递推式，然后用记忆化方法求解，最后有时我们需要构造出这一最优解。

加权独立结合问题 (WIS)

【打家劫舍】 你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。给定一个代表每个房屋存放金额的非负整数数组，计算你不触动警报装置的情况下，一夜之内能够偷窃到的最高金额。

加权独立结合问题 (WIS)

【打家劫舍】 你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。给定一个代表每个房屋存放金额的非负整数数组，计算你不触动警报装置的情况下，一夜之内能够偷窃到的最高金额。

问题抽象：考虑一个无向图 G ，其上所有点都在一条线上（这种图我们称其为路径图），每个点都有一个非负权重。我们称 G 的独立集合是指顶点互不相邻的子集（换句话说独立集合不会同时包含一条边上的两个点），然后要求解一个具有最大顶点权重和的独立集合，这一问题我们称为一维的加权独立集合问题。



加权独立结合问题 (WIS) (Cont'd)

分为如下两种情况讨论：

- ① 如果 v_n 不在最优解 S 中，那么 S 实际上可以看成前 $n-1$ 个点构成的路径图 G_{n-1} 组成的子问题的一个可行解。事实上 S 一定是 $n-1$ 个点的路径图的最大加权独立子集（即最优解）；
- ② 如果 v_n 在最优解 S 中，那么由于是独立集合问题，那么 v_{n-1} 就不能在最优解中，因为 v_{n-1} 和 v_n 相邻。 $S - \{v_n\}$ 一定是前 $n-2$ 个点构成的路径图 G_{n-2} 的最优解，所以整个问题的最优解就是 G_{n-2} 的最优解加上 v_n 。

加权独立结合问题 (WIS) (Cont'd)

分为如下两种情况讨论：

- ① 如果 v_n 不在最优解 S 中，那么 S 实际上可以看成前 $n-1$ 个点构成的路径图 G_{n-1} 组成的子问题的一个可行解。事实上 S 一定是 $n-1$ 个点的路径图的最大加权独立子集（即最优解）；
- ② 如果 v_n 在最优解 S 中，那么由于是独立集合问题，那么 v_{n-1} 就不能在最优解中，因为 v_{n-1} 和 v_n 相邻。 $S - \{v_n\}$ 一定是前 $n-2$ 个点构成的路径图 G_{n-2} 的最优解，所以整个问题的最优解就是 G_{n-2} 的最优解加上 v_n 。

问题：

- ① 为什么一定要子问题的最优解？
- ② 子问题最优一定能得到全问题最优吗？

加权独立结合问题 (WIS) (Cont'd)

由此，整个问题的最优解依赖于前 $n-1$ 和 $n-2$ 个点构成的子图的最优解，或者说，这两个子问题的最优是构成原问题最优的基础，这就是这一问题对应的最优子结构性质。

设前 i 个点的最优加权独立集合的权重之和为 W_i ，我们可以写出递推关系：

$$W_n = \max\{W_{n-1}, W_{n-2} + w_n\},$$

这里是最后一步的递推关系，事实上我们可以不断利用这一递推关系自顶向下向前构建出整个问题的解，即我们有更一般的表达

$$W_i = \max\{W_{i-1}, W_{i-2} + w_i\},$$

其中 $i = 2, 3, \dots, n$, $W_0 = 0$ 。

加权独立结合问题 (WIS) (Cont'd)

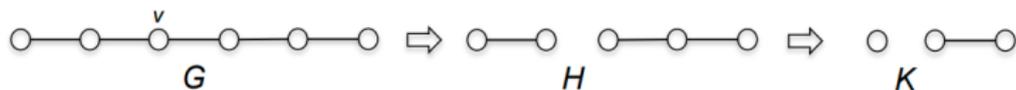
```
A := length-(n + 1) array // subproblem solutions
A[0] := 0 // base case #1
A[1] := w1 // base case #2
for i = 2 to n do
    // use recurrence from Corollary 16.2
    A[i] := max{ $\underbrace{A[i - 1]}_{\text{Case 1}}, \underbrace{A[i - 2] + w_i}_{\text{Case 2}}$ }
return A[n] // solution to largest subproblem
```

加权独立结合问题 (WIS) (Cont'd)

```
 $S := \emptyset$  // vertices in an MWIS
 $i := n$ 
while  $i \geq 2$  do
    if  $A[i - 1] \geq A[i - 2] + w_i$  then // Case 1 wins
         $i := i - 1$  // exclude  $v_i$ 
    else // Case 2 wins
         $S := S \cup \{v_i\}$  // include  $v_i$ 
         $i := i - 2$  // exclude  $v_{i-1}$ 
if  $i = 1$  then // base case #2
     $S := S \cup \{v_1\}$ 
return  $S$ 
```

加权独立结合问题 (WIS): 另一种解法

考虑一个任意的无向图 $G = (V, S)$, 所有顶点的权重均不为负, 并且任意一个顶点 $v \in V$ 具有权重 w_v . 通过从 H 中删除 v 的相邻项点以及相关边得到 K , 如下图所示:

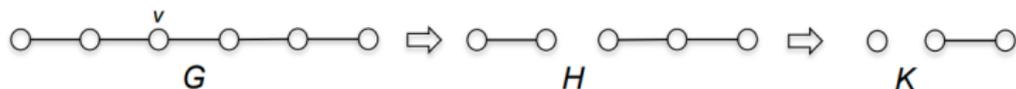


设 W_G 、 W_H 和 W_K 分别表示 G 、 H 和 K 的最大加权独立集合的总权重, 考虑下面这个公式:

$$W_G = \max\{W_H, W_K + w_v\},$$

加权独立结合问题 (WIS): 另一种解法

考虑一个任意的无向图 $G = (V, S)$, 所有顶点的权重均不为负, 并且任意一个顶点 $v \in V$ 具有权重 w_v . 通过从 H 中删除 v 的相邻项点以及相关联的边得到 K , 如下图所示:



设 W_G 、 W_H 和 W_K 分别表示 G 、 H 和 K 的最大加权独立集合的总权重, 考虑下面这个公式:

$$W_G = \max\{W_H, W_K + w_v\},$$

回忆最优二叉搜索树 (矩阵乘法顺序、钢条切割都类似):

$$c_{ij} = \sum_{k=i}^j p_k + \min_{i \leq k \leq j} \{c_{i,k-1} + c_{k+1,j}\}, \quad i \leq j.$$

最优二叉搜索树代码

```
// subproblems (i indexed from 1, j from 0)
A := (n + 1) × (n + 1) two-dimensional array
// base cases (i = j + 1)
for i = 1 to n + 1 do
    A[i][i - 1] := 0
// systematically solve all subproblems (i ≤ j)
for s = 0 to n - 1 do // s=subproblem size-1
    for i = 1 to n - s do // i + s plays role of j
        // use recurrence from Corollary 17.5
        A[i][i + s] :=
            
$$\sum_{k=i}^{i+s} p_k + \min_{r=i}^{i+s} \underbrace{\{A[i][r-1] + A[r+1][i+s]\}}_{\text{Case } r}$$

    return A[1][n] // solution to largest subproblem
```

背包问题

0-1 背包问题：我们有 n 个物品，每个物品的重量为 s_i ，价值为 v_i ，我们有一个容量为 C 的背包，我们希望找到一个最优的装载方案，使得背包中的物品总价值最大。这一问题的特点是每个物品你要么不放进包里，要么完整的 1 个放进去，因此称为 0-1 背包问题。

背包问题

0-1 背包问题：我们有 n 个物品，每个物品的重量为 s_i ，价值为 v_i ，我们有一个容量为 C 的背包，我们希望找到一个最优的装载方案，使得背包中的物品总价值最大。这一问题的特点是每个物品你要么不放进包里，要么完整的 1 个放进去，因此称为 0-1 背包问题。

我们设 $V_{i,c}$ 表示总重量不超过 c 的前 i 件物品组成的子集的最大总价值，那么我们可以得到如下递推关系：

$$V_{i,c} = \begin{cases} V_{i-1,c}, & s_i > c, \\ \max\{V_{i-1,c}, V_{i-1,c-s_i} + v_i\}, & s_i \leq c. \end{cases}$$

背包问题 (Cont'd)

```
// subproblem solutions (indexed from 0)
A := (n + 1) × (C + 1) two-dimensional array
// base case (i = 0)
for c = 0 to C do
    A[0][c] = 0
// systematically solve all subproblems
for i = 1 to n do
    for c = 0 to C do
        // use recurrence from Corollary 16.5
        if  $s_i > c$  then
            A[i][c] := A[i - 1][c]
        else
            A[i][c] :=
                max{  $\underbrace{A[i - 1][c]}_{\text{Case 1}}, \underbrace{A[i - 1][c - s_i] + v_i}_{\text{Case 2}} \}$ 
return A[n][C] // solution to largest subproblem
```

背包问题 (Cont'd)

```
// subproblem solutions (indexed from 0)
A := (n + 1) × (C + 1) two-dimensional array
// base case (i = 0)
for c = 0 to C do
    A[0][c] = 0
// systematically solve all subproblems
for i = 1 to n do
    for c = 0 to C do
        // use recurrence from Corollary 16.5
        if  $s_i > c$  then
            A[i][c] := A[i - 1][c]
        else
            A[i][c] :=
                max{  $\underbrace{A[i - 1][c]}_{\text{Case 1}}, \underbrace{A[i - 1][c - s_i] + v_i}_{\text{Case 2}}$  }
return A[n][C] // solution to largest subproblem
```

能否降成只用一维数组解决?

背包问题 (Cont'd)

$$\begin{aligned} \max \quad & \sum_{i=1}^n v_i x_i, \\ \text{s.t.} \quad & \sum_{i=1}^n s_i x_i \leq C, \\ & x_i \in \{0, 1\}. \end{aligned}$$

可以有很多的变体，例如 x_i 可以是任意整数，可以是 $[0, 1]$ 之间的任意实数，优化目标可以是最小化，约束条件可以是大于等于的不等式等等。

最长公共子序列

给定两个字符串 `text1` 和 `text2`，返回这两个字符串的最长 **公共子序列** 的长度。如果不存在 **公共子序列**，返回 `0`。

一个字符串的 **子序列** 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。

- 例如，`"ace"` 是 `"abcde"` 的子序列，但 `"aec"` 不是 `"abcde"` 的子序列。

两个字符串的 **公共子序列** 是这两个字符串所共同拥有的子序列。

示例 1:

输入: `text1 = "abcde"`, `text2 = "ace"`

输出: `3`

解释: 最长公共子序列是 `"ace"`，它的长度为 `3`。

<https://leetcode.cn/problems/longest-common-subsequence/>

最长公共子序列 (Cont'd)

与矩阵链乘法问题相似，设计 LCS 问题的递归算法首先要建立最优解的递归式。我们定义 $c[i, j]$ 表示 X_i 和 Y_j 的 LCS 的长度。如果 $i=0$ 或 $j=0$ ，即一个序列长度为 0，那么 LCS 的长度为 0。根据 LCS 问题的最优子结构性质，可得如下公式：

$$c[i, j] = \begin{cases} 0 & \text{若 } i = 0 \text{ 或 } j = 0 \\ c[i-1, j-1] + 1 & \text{若 } i, j > 0 \text{ 且 } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{若 } i, j > 0 \text{ 且 } x_i \neq y_j \end{cases} \quad (15.9)$$

序列对齐问题

DNA 序列相似性: AGGGCT 和 AGGCA 需要对齐, 可以插入空格, 其中空格和非空格扣 1 分, 两个字母对不上扣 2 分, 求最小扣分。

序列对齐问题

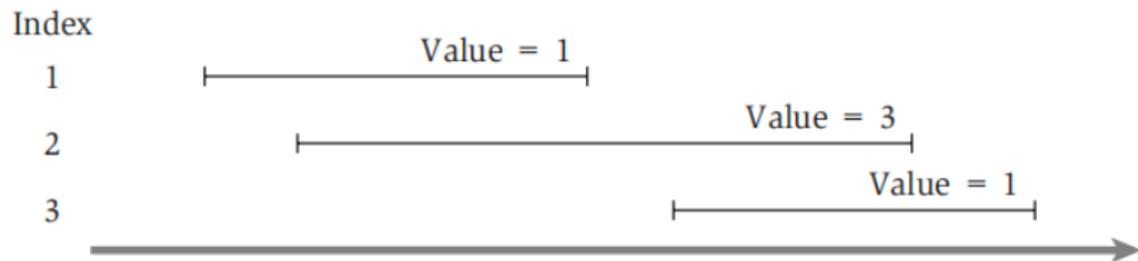
DNA 序列相似性：AGGGCT 和 AGGCA 需要对齐，可以插入空格，其中空格和非空格扣 1 分，两个字母对不上扣 2 分，求最小扣分。

$$P_{i,j} = \min\{\underbrace{P_{i-1,j-1} + \alpha_{x_i y_j}}_{\text{Case 1}}, \underbrace{P_{i-1,j} + \alpha_{gap}}_{\text{Case 2}}, \underbrace{P_{i,j-1} + \alpha_{gap}}_{\text{Case 3}}\}.$$

编辑距离（类似题目）：<https://leetcode.cn/problems/edit-distance/>

加权活动选择问题

给定一个活动集合 $S = \{a_1, a_2, \dots, a_n\}$ ，其中每个活动 a_i 都有一个开始时间 s_i 和结束时间 f_i ，且 $0 \leq s_i < f_i < \infty$ 。每个活动 a_i 也有一个权重值 w_i 。如果活动 a_i 和 a_j 满足 $f_i \leq s_j$ 或者 $f_j \leq s_i$ ，则称活动 a_i 和 a_j 是兼容的（即二者时间不会重合）。活动选择问题就是要找到一个权重最大的兼容活动子集。



其它问题

最大子序列和

最长递增子序列:

<https://leetcode.cn/problems/longest-increasing-subsequence/>

最长回文子串: <https://leetcode.cn/problems/longest-palindromic-substring/>

分割回文串: <https://leetcode.cn/problems/palindrome-partitioning-ii/>

图上相关的问题: Bellman-Ford 算法、Floyd-Warshall 算法

红黑树 project