

《计算复杂性理论》 第8讲 动态规划方法(1)

山东师范大学信息科学与工程学院
段会川
2014年11月

目录

- Fibonacci数的计算效率
- 动态规划方法概述
- DP入门—DAG中的最短路径
- 最长递增子序列的DP算法
- 两个字符串间编辑距离 (Levenshtein距离) 的DP算法
- 矩阵链相乘的DP算法

计算Fibonacci数的递归方法—效率问题

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-2} + F_{n-1} & n \geq 2 \end{cases} \quad \text{function fib}(n) \begin{cases} \text{if } n = 0 \text{ return } 0 \\ \text{if } n = 1 \text{ return } 1 \\ \text{return fib}(n-1) + \text{fib}(n-2) \end{cases}$$

1. fib(5)
2. fib(4) + fib(3)
3. (fib(3) + fib(2)) + (fib(2) + fib(1))
4. ((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
5. (((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))

计算第n个Fibonacci数需要计算加法fib(n-1)+fib(n-2)+1次

是一个比Fibonacci数还要大的数，指数级的!!!

$$F_n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n$$

Fibonacci数的动态规划(DP)算法

```
function FibDP(n)
  for i=1 to n
    if n=1 then f[1] = 1
    else if n=2 then
      f[2] = 1
    else
      f[i] = f[i-1]+f[i-2]
  end for
  return f[n]
end

function FibDP1(n)
  if n=1 then f=1
  else if n=2 then f=1
  else
    f1 = 1; f2 = 1
    for i=3 to n
      f = f1+f2
      f2 = f1; f1 = f
    end for
  end if
  return f
end
```

复杂度为 $\Theta(n)$ ，线性的！

目录

- Fibonacci数的计算效率
- 动态规划方法概述
- DP入门—DAG中的最短路径
- 最长递增子序列的DP算法
- 两个字符串间编辑距离 (Levenshtein距离) 的DP算法
- 矩阵链相乘的DP算法

Richard E. Bellman and DP

- Richard E. Bellman是一位美国应用数学家
- 以发明动态规划方法 (Dynamic Programming) 而闻名
- 获1976年John von Neumann Theory Prize (1976)
- 获1979年IEEE Medal of Honor



Richard E Bellman
Aug 26, 1920 – Mar 19, 1984

DP概述

- In mathematics, computer science, economics, and bioinformatics, dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable to problems exhibiting the properties of overlapping subproblems and optimal substructure. When applicable, the method takes far less time than naive methods that don't take advantage of the subproblem overlap (like depth-first search).
- 在数学、计算机科学、经济学和生物信息学中，动态规划方法 (DP) 是一种将复杂问题分解为一系列更简单的子问题的方法。
- 它适应于求解具有子问题重叠性和最优子结构的问题。
- 当DP适用时，它的时间代价将远小于不考虑子问题重叠性的朴素类方法，如深度优先方法、回溯法等。

Wikipedia

DP概述

- The idea behind dynamic programming is quite simple. In general, to solve a given problem, we need to solve different parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solution. Often when using a more naive method, many of the subproblems are generated and solved many times.
- DP背后的思想很简单。通常在解决一个问题时，我们要先解决它的不同部分，即子问题，然后合并这些子问题的解来获得整体的解。
- 而使用较朴素的方法时，解决过程中产生的许多子问题要解决许多遍。

Wikipedia

DP概述

- The dynamic programming approach seeks to solve each subproblem only once, thus reducing the number of computations: once the solution to a given subproblem has been computed, it is stored or "memoized", then the next time the same solution is needed, it is simply looked up. This approach is especially useful when the number of repeating subproblems grows exponentially as a function of the size of the input.
- 动态规划方法通过对每个子问题求解一次来降低计算量：
 - 一旦一个给定的子问题被计算过了，它就被“记忆”了，下次再遇到该问题时，DP将仅是查询它的计算结果。
- DP在重复性子问题的数量是输入规模的指数函数的情况下尤其有效。

Wikipedia

DP概述

- Dynamic programming algorithms are used for optimization (for example, finding the shortest path between two points, or the fastest way to multiply many matrices). A dynamic programming algorithm will examine the previously solved subproblems and will combine their solutions to give the best solution for the given problem.
- 动态规划算法常用于解决优化问题，如：
 - 求两点之间的最短路径
 - 计算多个矩阵连乘的快速方法
- DP算法将检查之前已经解决的子问题，并将它们合并以获得给定问题的最优解。

Wikipedia

DP概述

- The alternatives are many, such as using a greedy algorithm, which picks the locally optimal choice at each branch in the road. The locally optimal choice may be a poor choice for the overall solution. While a greedy algorithm does not guarantee an optimal solution, it is often faster to calculate. Fortunately, some greedy algorithms (such as minimum spanning trees) are proven to lead to the optimal solution.
- 用DP可求解的问题也可以用其他方法求解，如贪心算法。
 - 贪心法在求解过程的每一个分支处进行局部最优的选择。
 - 而局部最优的选择可能会对于全局最优解来说可能是一个很差的选择。
- 尽管贪心法不能保证全局最优解，但它通常可以快速地进行计算。
- 幸运的是某些贪心算法，如最小生成树，可以被证明得出全局最优解。

Wikipedia

DP概述

- Dynamic programming is both a mathematical optimization method and a computer programming method. In both contexts it refers to simplifying a complicated problem by breaking it down into simpler subproblems in a recursive manner.
- DP既是一种数学优化方法也是一种计算机程序设计方法。
- 在这两个领域里，它都指的是通过将问题递归式地分解为较为简单的问题来简化复杂问题的求解。

Wikipedia

DP概述

- While some decision problems cannot be taken apart this way, decisions that span several points in time do often break apart recursively; Bellman called this the "Principle of Optimality".
- Likewise, in computer science, a problem that can be solved optimally by breaking it into subproblems and then recursively finding the optimal solutions to the subproblems is said to have optimal substructure.

Wikipedia

- 尽管一些决策性问题不能以上述的分解方式求解，但是经历多个时间节点的决策性问题通常可以递归式地分解。
 - Bellman将此称为最优化原理(Principle of Optimality)
- 类似地，在计算机科学中，当一个问题可以通过分解为一系列子问题并递归式地求取最优解而最终获得问题的整体最优解时，被称为具有最优子结构(optimal substructure)。

第8讲 动态规划方法(1)

13

Bellman's Principle of Optimality

- An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

R Bellman, On the Theory of Dynamic Programming, Proceedings of the National Academy of Sciences, 1952

- 最优策略具有如下性质：
不管最初的状态和决策是什么，接下来的决策必须构成一个关于最初决策所产生的状态的最优策略。

第8讲 动态规划方法(1)

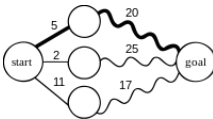
14

Optimal substructure

- In computer science, a problem is said to have optimal substructure if an optimal solution can be constructed efficiently from optimal solutions of its subproblems.
- This property is used to determine the usefulness of dynamic programming and greedy algorithms for a problem.

Wikipedia

- 在计算机科学中，说一个问题具有最优子结构，指的是该问题的最优解可以由子问题的最优解构造出来。
- 这个性质用于确定动态规划方法和贪心算法对于某个问题的有效性。



第8讲 动态规划方法(1)

15

DP概述

- If subproblems can be nested recursively inside larger problems, so that dynamic programming methods are applicable, then there is a relation between the value of the larger problem and the values of the subproblems.
- In the optimization literature this relationship is called the Bellman equation.

Wikipedia

- 如果子问题可以被递归式地嵌套在较大规模的问题中，致使DP方法适用，那么就存在关于较大问题的优化目标函数值与较小问题优化目标函数值之间的一种关系。
- 在优化文献中，这个关系被称为Bellman方程或动态规划方程。

第8讲 动态规划方法(1)

16

Dynamic programming in computer programming

- There are two key attributes that a problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping subproblems.
- If a problem can be solved by combining optimal solutions to non-overlapping subproblems, the strategy is called "divide and conquer" instead. This is why mergesort and quicksort are not classified as dynamic programming problems.

Wikipedia

- 可以施行DP的问题必须具有两个属性
 - 最优子结构性质和子问题重叠性质。
- 如果一个问题可以通过合并非重叠的子问题的最优解而求解，则相应的策略被称为分治法(divide and conquer)
 - 因此，合并排序和快速排序不能划归到DP中。

第8讲 动态规划方法(1)

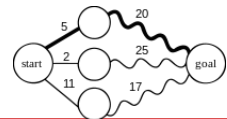
17

Dynamic programming in computer programming

- Optimal substructure means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its subproblems.
- Consequently, the first step towards devising a dynamic programming solution is to check whether the problem exhibits such optimal substructure. Such optimal substructures are usually described by means of recursion.

Wikipedia

- 最优子结构性性质指的是给定优化问题的解可以通过合并它的子问题的最优解获得。
- 因此，设计DP求解方法的第一步就是检查一个问题是否具有最优子结构。
 - 这种最优子结构通常以递归方式描述。
 - 如，图中的最短路径问题。

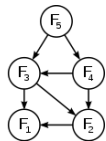


第8讲 动态规划方法(1)

18

Dynamic programming in computer programming

- Overlapping subproblems means that the space of subproblems must be small, that is, any recursive algorithm solving the problem should solve the same subproblems over and over.
 - Dynamic programming takes account of this fact and solves each subproblem only once.
- Wikipedia
- 重叠子问题指的是子问题的空间必须小，即任何求解该问题的递归算法都要一次又一次地调用相同的子问题。
 - DP考虑到了这种特征，并对每个子问题只解决一次。
 - 如Fibonacci数的计算。



DP实现—自顶向下方法(Top-down approach)

- 自顶向下方法是所有递归式表述的问题的直接方法。
- 如果一个问题的解可以用它的子问题递归式地表述，并且子问题具有重叠性，那么我们可以很容易地将子问题的解记忆或存储在一个表中。
- 每当我们遇到一个子问题时，我们首先查表看它是否已经解决了。
 - 如果已经解决了，则我们直接使用已经解得的结果。
 - 如果尚未解决，则我们就求解它，并将结果存入表中。

DP实现—自底向上方法(Bottom-up approach)

- 一旦一个问题的解被递归式地表述成了其子问题的解，我们就可以将它以自底向上的方式重新表述。
 - 先解决问题，再用子问题的解来构造并得出较大子问题的解。
 - 这通常也借助于表格方法，即迭代式地运用较小子问题的解生成规模逐渐变大的子问题的解。

设计DP算法的步骤

我们通常按如下4个步骤来设计一个动态规划算法：

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

步骤1~3是动态规划算法求解问题的基础。如果我们仅仅需要一个最优解的值，而非解本身，可以忽略步骤4。如果确实要做步骤4，有时就需要在执行步骤3的过程中维护一些额外信息，以便用来构造一个最优解。

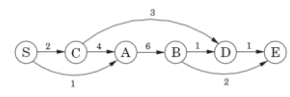
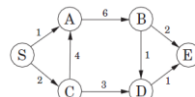
《导论》中文3版，P204

目录

- Fibonacci数的计算效率
- 动态规划方法概述
- DP入门—DAG中的最短路径
- 最长递增子序列的DP算法
- 两个字符串间编辑距离 (Levenshtein距离) 的DP算法
- 矩阵链相乘的DP算法

DP入门—DAG中的最短路径

- DAG的性质—结点可以被线性化
 - 即结点可以排列到一条直线上，使得所有的边保持从左到右的方向。
- 要求下图中S到D的最短路径，由于到达D的途径只能通过其前趋，即B或C，因此有
 - $\text{dist}(D) = \min\{\text{dist}(B)+1, \text{dist}(C)+3\}$



P174, 图6-1

DP入门—DAG中的最短路径

- 对于每个结点，都可以写出类似的关系式
 - 如果按照线性序计算各个结点的dist值，则总能保证在计算结点v的dist时，其所有前趋结点的dist值均已得到，因此可以只用一遍循环计算S到所有结点的dist值。
- 1. Initialize all $\text{dist}[]$ values to ∞
- 2. $\text{dist}[s] = 0$
- 3. for each $v \in V \setminus \{s\}$, in linearized order
- 4. $\text{dist}[v] = \min_{\{u,v\} \in E} \{\text{dist}[u] + l(u,v)\}$
- 复杂度: $O(|E|)$

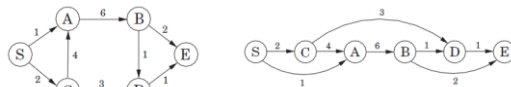


第8讲 动态规划方法(1)

25

DP入门—DAG最短路径算法的DP解释

1. 最优子结构
 - 一个结点X到T的距离仅由其各前趋结点Pi到T的距离和Pi到X的边的权决定。
2. 子问题的重叠性
 - 某个结点Y的距离可以用于计算其所有后继结点的距离，因有重复使用的性质。



第8讲 动态规划方法(1)

26

目录

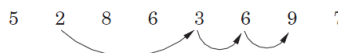
- Fibonacci数的计算效率
- 动态规划方法概述
- DP入门—DAG中的最短路径
- 最长递增子序列的DP算法
- 两个字符串间编辑距离 (Levenshtein距离) 的DP算法
- 矩阵链相乘的DP算法

第8讲 动态规划方法(1)

27

最长递增子序列—问题描述

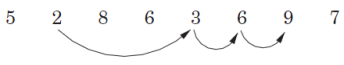
- 子序列
 - 给定数字序列 a_1, \dots, a_n ，满足 $1 \leq i_1 \leq i_2 \leq \dots \leq i_k \leq n$ 的子集 $a_{i_1}, a_{i_2}, \dots, a_{i_k}$ 称为它的一个子序列
- 递增子序列
 - 如果子序列中的各个数字都是严格单调递增的，则称其为一个递增子序列
- 问题
 - 给定一个数字序列，如何找出其中最长的递增子序列？

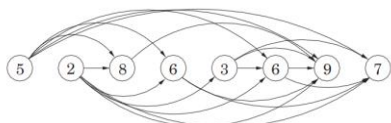


第8讲 动态规划方法(1)

28

最长递增子序列—问题分析

- 观察可知，下列数字序列的最长递增子序列为2, 3, 6, 9
- 
- 如果将每一个数字记作一个结点，在每一对有递增关系的数字之间画一条边可得如下的有向图



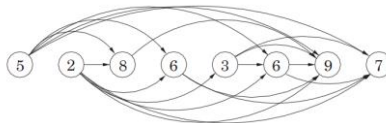
P175

第8讲 动态规划方法(1)

29

最长递增子序列—问题分析

- 显然，该图有如下性质
 1. 对于每条边 (i, j) 都有 $i < j$ ，因而该图为DAG
 2. 递增子序列和DAG的路径间存在一一对应的关系
- 因此
 - 寻找最长递增子序列问题就转化为求解图中的最长路径问题！



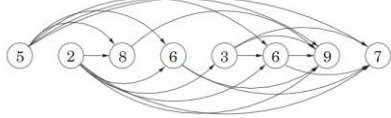
P175

第8讲 动态规划方法(1)

30

最长递增子序列—DP解法

- 因而，将最短路径DP算法稍作修改就可以得到求解最长递增子序列的算法
 1. for $j=1$ to n
 2. $L[j] = 1 + \max\{L[i] : (i, j) \in E\}$
 3. return $\max_j\{L[j] : j \in \{1, \dots, n\}\}$
- 复杂度: $O(n^2)$
- 该算法求出的是最长递增子序列的长度，要获得子序列，需要像Dijkstra算法那样增加对前趋结点的记录



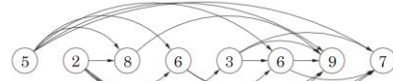
P175

第8讲 动态规划方法(1)

31

最长递增子序列—DP解法练习

- 记 $\{i : (i, j) \in E\} = \emptyset$ 时 $\max\{L(i) : (i, j) \in E\} = 0$
- $j=1: L(1) = 0+1 = 1; j=2: L(2) = 0+1 = 1$



P175

第8讲 动态规划方法(1)

32

最长递增子序列—DP性质分析

- 最长递增子序列的上述求解方法实质上是一种DP方法:
 - 为了解决所提出的问题，定义了一组子问题 $\{L(j) : 1 \leq j \leq n\}$ ，它具有下述特征 (P176)
 - 存在子问题间的一种排列以及关联关系： $L(j) = 1 + \max\{L(i) : (i, j) \in E\}$
对于任意一个子问题，这种关联关系说明了如何在解决了排列中靠前的所有子问题的情况下求出该子问题的解。
- 该特征说明该问题满足DP所要求的两个基本条件
 - 最优子结构性质、子问题的重叠性

第8讲 动态规划方法(1)

33

最长递增子序列—DP算法复杂度

- 最长递增子序列—DP算法伪代码
 1. for $j=1$ to n
 2. $L[j] = 1 + \max\{L[i] : (i, j) \in E\}$
 3. return $\max_j\{L[j] : j \in \{1, \dots, n\}\}$
- 算法外层循环对所有的结点进行，循环体内对所有的边检查一遍，因而时间复杂度为 $O(|E|) = O(n^2)$ 。
- 算法要存储以每一个结点为终点的最长递增子序列的长度，因而空间复杂度为 $O(n)$ 。

第8讲 动态规划方法(1)

34

目录

- Fibonacci数的计算效率
- 动态规划方法概述
- DP入门—DAG中的最短路径
- 最长递增子序列的DP算法
- 两个字符串间编辑距离 (Levenshtein距离) 的DP算法
- 矩阵链相乘的DP算法

第8讲 动态规划方法(1)

35

编辑距离—Levenshtein距离

- 判断两个单词 (或两个字符串) 间的相似性可以使用编辑距离来定义
 - 它可以用来解决如下的实际问题：当一个单词拼写错误时，如何从字典中找到最好的拼写建议？
- 最常用的编辑距离是，是由Vladimir Levenshtein于1965年提出的：
 - 两个字符串间的Levenshtein距离是将一个字符串变换为另一个字符串的最小编辑操作数，这里的编辑操作指的是对单个字符的插入、删除和替换3种操作。

第8讲 动态规划方法(1)

36

编辑距离—Levenshtein距离

Levenshtein距离的形式化表示

- 两个字符串 a, b 间的Levenshtein距离由 $lev_{a,b}(|a|, |b|)$ 给出, 它定义为

$$lev_{a,b}(i, j) = \begin{cases} \max(i, j) & \min(i, j) = 0 \\ \min \begin{cases} lev_{a,b}(i-1, j) + 1 \\ lev_{a,b}(i, j-1) + 1 \\ lev_{a,b}(i-1, j-1) + [a_i \neq b_j] \end{cases} & \text{else} \end{cases}$$

- Levenshtein因在纠错码理论和包括Levenshtein距离的信息论方面的贡献获得了2006年的IEEE理查德·海明奖章

编辑距离—直观解释

- 两个字符串间的距离可以看作是它们在多大程度上相互对齐, 或它们之间的匹配程度如何。

S - N O W Y - S N O W - Y
S U N N - Y S U N - - N Y
Cost: 3 Cost: 5

- 对于SNOWY和SUNNY, 不存在代价比3更小的对齐了。
- 对于两个字符串, 由于存在非常多的对齐方式, 用简单的遍历搜索其中的最优(代价最小)的对齐方式, 即最短编辑距离, 是非常可怕的任务。

编辑距离的DP求解法—子问题构造

- 目标: 寻找字符串 $x[1 \dots m]$ 和 $y[1 \dots n]$ 间的编辑距离
- 考虑两个字符串的前缀(prefix) $x[1 \dots i]$ 和 $y[1 \dots j]$ 间的距离, 记其为 $E(i, j)$, 则最终目标变为计算 $E(m, n)$
- 要对齐 $x[1 \dots i]$ 和 $y[1 \dots j]$, 最右侧的列只可能是如下三种情况之一:

x[i] 或 - 或 x[i]
- y[j] y[j]

编辑距离的DP解—子问题分析

- 要对齐 $x[1 \dots i]$ 和 $y[1 \dots j]$, 最右侧的列只可能是如下三种情况之一:

x[i] 或 - 或 x[i]
- y[j] y[j]

- 情况1: 该列产生代价1, 余下的问题是最佳对齐 $x[1 \dots i-1]$ 和 $y[1 \dots j]$, 而它对应子问题 $E(i-1, j)$
- 情况2: 该列产生代价1, 余下的问题是最佳对齐 $x[1 \dots i]$ 和 $y[1 \dots j-1]$, 而它对应子问题 $E(i, j-1)$
- 情况3: 该列在 $x[i] = y[j]$ 时产生代价0, 在 $x[i] \neq y[j]$ 时产生代价1, 余下的问题是最佳对齐 $x[1 \dots i-1]$ 和 $y[1 \dots j-1]$, 而它对应子问题 $E(i-1, j-1)$

编辑距离的DP解—子问题分析

- 显然 $E(i, j)$ 应该是上述三个情况中的最优者, 即

$$E(i, j) = \min \begin{cases} 1 + E(i-1, j), \\ 1 + E(i, j-1), \\ (1 - \delta(x[i], y[j])) + E(i-1, j-1) \end{cases}$$

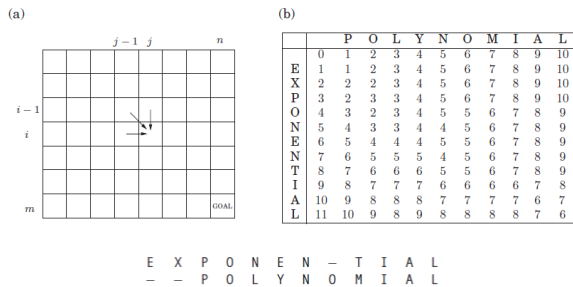
- 其中 $\delta(i, j) = \begin{cases} 0 & \text{当 } i \neq j \\ 1 & \text{当 } i = j \end{cases}$ 为Kronecker delta

- 尽管得到的是一个递归关系式, 但是如果用递归方法解决显然复杂度是不可接受的

编辑距离的DP解—子问题解的关系分析

- 当 $i = 0$ 时, x 为一个空串, 从一个空串变换到一个长度为 $j \geq 0$ 的串显然需要的最少操作是 j 次的插入操作, 即 $E(0, j) = j$, 同样 $E(i, 0) = i$ 。
- 显然所有子问题 $E(i, j), 0 \leq i \leq m, 0 \leq j \leq n$ 的解可以用一张大小为 $(m+1) \times (n+1)$ 的表来存放, 而表中的第0行可由 $E(0, j) = j$ 初始化, 第0列可由 $E(i, 0) = i$ 初始化。
- 表中的第 $(i > 0, j > 0)$ 项的值根据上页的分析由其上面的元素 $E(i-1, j)$ 、左面的元素 $E(i, j-1)$ 和左上的元素 $E(i-1, j-1)$ 与 $x[i]$ 、 $y[j]$ 决定。

编辑距离的DP解—示例



第8讲 动态规划方法(1)

43

编辑距离的DP解—练习

		d	i	s	t	a	n	c	e
d									
y									
n									
a									
m									
i									
c									

第8讲 动态规划方法(1)

44

编辑距离的DP解—算法伪代码与复杂度

- for $i=0$ to m
 - $E[i, 0] = i$; 显然该算法的时间和空间复杂度均为 $O(mn)$
 - for $j=0$ to n
 - $E[0, j] = j$;
 - for $i=1$ to m
 - for $j=1$ to n
 - $E[i, j] = \min\{E[i-1, j]+1, E[i, j-1]+1, E[i-1, j-1]+1-\delta(x[i], y[j])\}$
 -
 - return $E[m, n]$
- 显然，该算法的时间和空间复杂度均为 $O(mn)$

第8讲 动态规划方法(1)

45

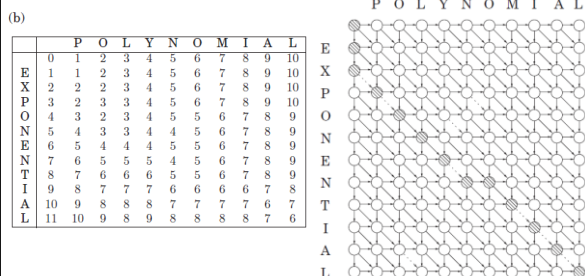
编辑距离问题—隐含的DAG

- 每个DP都隐含有一个DAG结构
 - 将每个子问题表示为一个结点，解决每个子问题 v 所依赖的各个子问题 u_1, \dots, u_k 生成一条从 u_i 到 v 的一条有向边，显然这将得到一个DAG
- 编辑距离问题中的一般结点有三条以其为终点的边，它们分别形如：
 - $(i-1, j) \rightarrow (i, j)$
 - $(i, j-1) \rightarrow (i, j)$
 - $(i-1, j-1) \rightarrow (i, j)$ 。
- 如果将 $x[i] = x[j]$ 时的边 $(i-1, j-1) \rightarrow (i, j)$ 的权定为0，其它三种情况下边的权均记为1，则编辑距离即是从 $(0,0)$ 到 (m,n) 的最短路径长度。

第8讲 动态规划方法(1)

46

编辑距离问题—隐含的DAG示例



第8讲 动态规划方法(1)

47

DP中子问题的通用构造方法

- 输入为 x_1, x_2, \dots, x_n ，子问题取 x_1, x_2, \dots, x_i ，则子问题共有 $n-1$ 个
- 输入为 x_1, x_2, \dots, x_n ，子问题取 x_i, x_{i+1}, \dots, x_j ，则子问题数量有 $O(n^2)$ 个
- 输入为 $x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_n$ ，子问题取 $x_1, x_2, \dots, x_i, y_1, y_2, \dots, y_j$ ，则子问题共有 $O(mn)$ 个
- 输入为一棵树，则其每个子树都可以作为一个子问题，因而子问题个数为树种的结点总数



第8讲 动态规划方法(1)

48

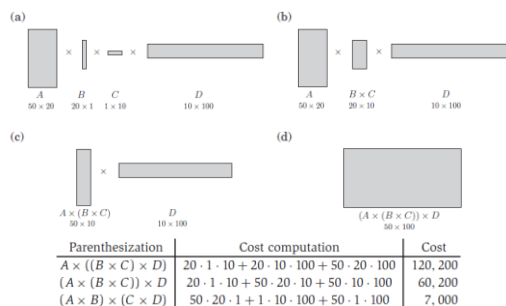
目录

- Fibonacci数的计算效率
- 动态规划方法概述
- DP入门—DAG中的最短路径
- 最长递增子序列的DP算法
- 两个字符串编辑距离 (Levenshtein距离) 的DP算法
- 矩阵链相乘的DP算法

矩阵链相乘

- 假设 A, B, C, D 分别是维数为 50×20 、 20×1 、 1×10 和 10×100 的二维矩阵，则以什么次序计算 $A \times B \times C \times D$ 可以得到最少的元素乘法次数？
 - 矩阵乘法不遵守交换律，即通常 $A \times B \neq B \times A$
 - 但遵守结合律，即 $A \times B \times C = (A \times B) \times C = A \times (B \times C)$

矩阵链相乘—示例

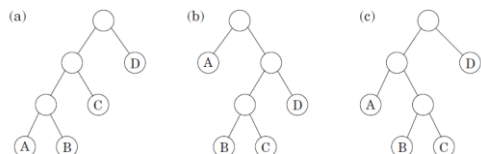


矩阵链相乘—问题分析

- 假设需要计算 $A_1 \times A_2 \times \dots \times A_n$ ，其中各矩阵的维数分别为 $m_0 \times m_1, m_1 \times m_2, \dots, m_{n-1} \times m_n$
 - 则每种特定的乘法集合次序都可表示为一棵满二叉树，每个矩阵都是二叉树的叶结点
 - 每棵树都有 n 个叶结点，可能的树的总数为 n 的指数，因而不能通过遍历的方法尝试所有的树

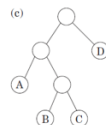
矩阵链相乘—问题分析

Figure 6.7 (a) $((A \times B) \times C) \times D$; (b) $A \times ((B \times C) \times D)$; (c) $(A \times (B \times C)) \times D$.



矩阵链相乘—问题分析

- 最优子结构
 - 如果一棵树对应的子链是最优的，则其两棵子树对应的子链也必然是最优的
- 子问题表达
 - 由于矩阵乘法不遵循交换律，一棵子树对应的就是形如 $A_i \times A_{i+1} \times \dots \times A_j$ 的矩阵子链积
 - 对于 $1 \leq i \leq j \leq n$ 定义 $C(i, j)$ 为计算 $A_i \times A_{i+1} \times \dots \times A_j$ 的最小代价，该子问题的规模为 $|j - i|$



矩阵链相乘—问题分析

- $i = j$ 对应最小的子问题，此时没有矩阵相乘，因而 $C(i, j) = 0$
- 当 $i < j$ 时，子问题对应子树的两个分枝分别对应形如 $A_i \times \cdots \times A_k$ 和 $A_{k+1} \times \cdots \times A_j$ 两个矩阵子链的乘积
 - 整个子树的代价就等于两个子链各自的代价 $C(i, k)$ 和 $C(k+1, j)$ 与两个子链的结果乘起来的代价
 - 两个子链的积分别是维数为 $m_{i-1} \times m_k$ 和 $m_k \times m_j$ 的矩阵，因而它们相乘的代价为 $m_{i-1} \times m_k \times m_j$
 - 最优的 $C(i, j)$ 应是各种可能 k 值中的最好结果，即：
$$C(i, j) = \min_{i \leq k < j} \{C(i, k) + C(k+1, j) + m_{i-1} \times m_k \times m_j\}$$

第8讲 动态规划方法(1)

55

矩阵链相乘的DP算法—求解步骤

□ 步骤一：最优括号化方案的结构特征

下面我们给出本问题的最优子结构。假设 $A_{i+1} \cdots A_j$ 的最优括号化方案的分割点在 A_k 和 A_{k+1} 之间。那么，继续对“前缀”子链 $A_i A_{i+1} \cdots A_k$ 进行括号化时，我们应该直接采用独立求解它时所得的最优方案。这样做的原因是什么呢？如果不采用独立求解 $A_i A_{i+1} \cdots A_k$ 所得的最优方案来对它进行括号化，那么可以将此最优解代入 $A_i A_{i+1} \cdots A_j$ 的最优解中，代替原来对于子链 $A_i A_{i+1} \cdots A_k$ 进行括号化的方案（比 $A_i A_{i+1} \cdots A_k$ 最优解的代价更高），显然，这样得到的解比 $A_i A_{i+1} \cdots A_j$ 原来的“最优解”代价更低，产生矛盾。对于子链 $A_{k+1} A_{k+2} \cdots A_j$ ，我们有相似的结论：在原问题 $A_i A_{i+1} \cdots A_j$ 的最优括号化方案中，对于子链 $A_{k+1} A_{k+2} \cdots A_j$ 进行括号化的方法，就是它自身的最优括号化方案。

《导论》中文3版，P212

第8讲 动态规划方法(1)

56

矩阵链相乘的DP算法—求解步骤

□ 步骤一：最优括号化方案的结构特征

现在我们展示如何利用最优子结构性质从子问题的最优解构造原问题的最优解。我们已经看到，一个非平凡的矩阵链乘法问题实例的任何解都需要划分链，而任何最优解都是由子问题实例的最优解构成的。因此，为了构造一个矩阵链乘法问题实例的最优解，我们可以将问题划分为两个子问题（ $A_i A_{i+1} \cdots A_k$ 和 $A_{k+1} A_{k+2} \cdots A_j$ 的最优括号化问题），求出子问题实例的最优解，然后将子问题的最优解组合起来。我们必须保证在确定分割点时，已经考察了所有可能的划分点，这样就可以保证不会遗漏最优解。

《导论》中文3版，P212

第8讲 动态规划方法(1)

57

矩阵链相乘的DP算法—求解步骤

□ 步骤二：一个递归求解方案

下面用子问题的最优解来递归地定义原问题最优解的代价。对矩阵链乘法问题，我们可以将对所有 $1 \leq i \leq j \leq n$ 确定 $A_i A_{i+1} \cdots A_j$ 的最小代价括号化方案作为子问题。令 $m[i, j]$ 表示计算矩阵 $A_{i,j}$ 所需标量乘法次数的最小值，那么，原问题的最优解——计算 $A_{1,n}$ 所需的最低代价就是 $m[1, n]$ 。

我们可以递归定义 $m[i, j]$ 如下。对于 $i=j$ 时的平凡问题，矩阵链只包含唯一的矩阵 $A_{i,i} = A_i$ ，因此不需要做任何标量乘法运算。所以，对所有 $i=1, 2, \dots, n$ ， $m[i, i]=0$ 。若 $i < j$ ，我们利用步骤1中得到的最优子结构来计算 $m[i, j]$ 。我们假设 $A_i A_{i+1} \cdots A_j$ 的最优括号化方案的分割点在矩阵 A_k 和 A_{k+1} 之间，其中 $i \leq k < j$ 。那么， $m[i, j]$ 就等于计算 $A_{i,k}$ 和 $A_{k+1,j}$ 的代价加上两者相乘的代价的最小值。由于矩阵 A_k 的大小为 $p_{i-1} \times p_i$ ，易知 $A_{i,k}$ 与 $A_{k+1,j}$ 相乘的代价为 $p_{i-1} p_k p_j$ 次标量乘法运算。因此，我们得到

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$$

《导论》中文3版，P212-3

第8讲 动态规划方法(1)

58

矩阵链相乘的DP算法—求解步骤

□ 步骤二：一个递归求解方案

$p_k p_j$ 次标量乘法运算。因此，我们得到

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$$

此递归公式假定最优分割点 k 是已知的，但实际上我们是不知道的。不过， k 只有 $j-i$ 种可能的取值，即 $k=i, i+1, \dots, j-1$ 。由于最优分割点必在其中，我们只需检查所有可能情况，找到最优者即可。因此， $A_i A_{i+1} \cdots A_j$ 最小代价括号化方案的递归求解公式变为：

$$m[i, j] = \begin{cases} 0 & \text{如果 } i = j \\ \min_{i \leq k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{如果 } i < j \end{cases} \quad (15.7)$$

$m[i, j]$ 的值给出了子问题最优解的代价，但它并未提供足够的信息来构造最优解。为此，我们用 $[i, j]$ 保存 $A_i A_{i+1} \cdots A_j$ 最优括号化方案的分割点位置 k ，即使得 $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$ 成立的 k 值。

《导论》中文3版，P213

第8讲 动态规划方法(1)

59

矩阵链相乘的DP算法—求解步骤

□ 步骤三：计算最优代价

现在，我们可以很容易地基于递归公式(15.7)写出一个递归算法，来计算 $A_i A_{i+1} \cdots A_n$ 相乘的最小代价 $m[1, n]$ 。像我们在钢条切割问题一节中所看到的，以及即将在15.3节中看到的那样，此递归算法是指数时间的，并不比检查所有括号化方案的暴力搜索方法更好。

注意到，我们需要求解的不同的子问题的数目是相对较少的：每对满足 $1 \leq i \leq j \leq n$ 的 i 和 j 对应一个唯一的子问题，共有 $\binom{n}{2} + n = \Theta(n^2)$ 个。递归算法会在递归调用树的不同分支中多次遇到同一个子问题。这种子问题重叠的性质是应用动态规划的另一个标识（第一个标识是最优子结构）。

《导论》中文3版，P213

第8讲 动态规划方法(1)

60

矩阵链相乘的DP算法—求解步骤

□ 步骤三：计算最优代价

我们采用自底向上表格法代替基于公式(15.7)的递归算法来计算最优代价(我们将在15.3节中给出对应的带备忘的自顶向下方法)。下面给出的过程 MATRIX-CHAIN-ORDER 实现了自底向上表格法。此过程假定矩阵 A_i 的规模为 $p_{i-1} \times p_i$ ($i=1, 2, \dots, n$)。它的输入是一个序列 $p=\langle p_0, p_1, \dots, p_n \rangle$ ，其长度为 $p.length=n+1$ 。过程用一个辅助表 $m[1..n, 1..n]$ 来保存代价 $m[i, j]$ ，用另一个辅助表 $s[1..n-1, 2..n]$ 记录最优值 $m[i, j]$ 对应的分割点 k 。我们就可以利用表 s 构造最优解。

为了实现自底向上方法，我们必须确定计算 $m[i, j]$ 时需要访问哪些其他表项。公式(15.7)显示， $j-i+1$ 个矩阵链相乘的最优计算代价 $m[i, j]$ 只依赖于那些少于 $j-i+1$ 个矩阵链相乘的最优计算代价。也就是说，对 $k=i, i+1, \dots, j-1$ ，矩阵 $A_{k+1} \dots A_j$ 是 $k-i+1 < j-i+1$ 个矩阵的积，矩阵 $A_{i+1} \dots A_k$ 是 $j-k < j-i+1$ 个矩阵的积。因此，算法应该按长度递增的顺序求解矩阵链括号化问题，并按对应的顺序填写表 m 。对矩阵链 $A_1 A_2 \dots A_n$ 最优括号化的子问题，我们认为其规模为链的长度 $j-i+1$ 。

《导论》中文3版, P213

第8讲 动态规划方法(1)

61

矩阵链相乘的DP算法—求解步骤

□ 步骤三：计算最优代价

MATRIX-CHAIN-ORDER(p)

```
1   $n = p.length - 1$ 
2  let  $m[1..n, 1..n]$  and  $s[1..n-1, 2..n]$  be new tables
3  for  $i = 1$  to  $n$ 
4       $m[i, i] = 0$ 
5  for  $l = 2$  to  $n$  //  $l$  is the chain length
6      for  $i = 1$  to  $n-l+1$ 
7           $j = i+l-1$ 
8           $m[i, j] = \infty$ 
9          for  $k = i$  to  $j-1$ 
10              $q = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$ 
11             if  $q < m[i, j]$ 
12                  $m[i, j] = q$ 
13                  $s[i, j] = k$ 
14  return  $m$  and  $s$ 
```

算法时间复杂度: $O(n^3)$
算法空间复杂度: $\Theta(n^2)$

《导论》中文3版, P213-4

第8讲 动态规划方法(1)

62

矩阵链相乘的DP算法—求解步骤

□ 步骤三：计算最优代价

算法首先在第3~4行对所有 $i=1, 2, \dots, n$ 计算 $m[i, i]=0$ (长度为1的链的最小计算代价)。接着在第5~13行 for 循环的第一个循环步中，利用递归公式(15.7)对所有 $i=1, 2, \dots, n-1$ 计算 $m[i, i+1]$ (长度 $l=2$ 的链的最小计算代价)。在第二个循环步中，算法对所有 $i=1, 2, \dots, n-2$ 计算 $m[i, i+2]$ (长度 $l=3$ 的链的最小计算代价)，依此类推。在每个循环步中，第10~13行计算代价 $m[i, j]$ 时仅依赖于已经计算出的表项 $m[i, k]$ 和 $m[k+1, j]$ 。

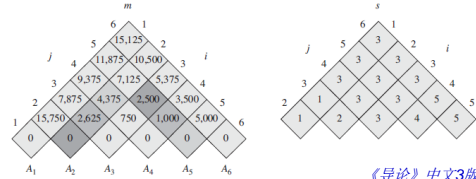
图15-5展示了对一个长度为6的矩阵链执行此算法的过程。由于我们定义 $m[i, j]$ 仅在 $i \leq j$ 时有意义，因此表 m 只使用主对角线之上的部分。图中的表是经过旋转的，主对角线已经旋转到了水平方向。矩阵链的规模列在了图的下方。在这种布局中，我们可以看到子矩阵链 $A_i A_{i+1} \dots A_j$ 相乘的代价 $m[i, j]$ 恰好位于始于 A_i 的东北至西南方向的直线与始于 A_j 的西北至东南方向的直线的交点上。表中同一行中的表项都对应长度相同的矩阵链。MATRIX-CHAIN-ORDER 按自下而上、自左至右的顺序计算所有行。当计算表项 $m[i, j]$ 时，会用到乘积 $p_{i-1} p_k p_j$ ($k=i, i+1, \dots, j-1$)，以及 $m[i, j]$ 西南方向和东南方向上的所有表项。

《导论》中文3版, P213-4

第8讲 动态规划方法(1)

63

矩阵链相乘的DP算法—求解步骤



《导论》中文3版, P213-4

Figure 15.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

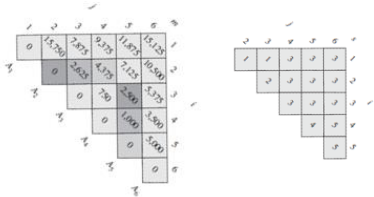
matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

$$m[2, 5] = \min \begin{cases} m[2, 2] + m[3, 5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13,000 \\ m[2, 3] + m[4, 5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 \\ m[2, 4] + m[5, 5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11,375 \end{cases} = 7125$$

第8讲 动态规划方法(1)

64

矩阵链相乘的DP算法—求解步骤



《导论》中文3版, P213-4

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

$$m[2, 5] = \min \begin{cases} m[2, 2] + m[3, 5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13,000 \\ m[2, 3] + m[4, 5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 \\ m[2, 4] + m[5, 5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11,375 \end{cases} = 7125$$

第8讲 动态规划方法(1)

65

矩阵链相乘的DP算法—求解步骤

□ 步骤四：构造最优解

虽然 MATRIX-CHAIN-ORDER 求出了计算矩阵链乘积所需的最少标量乘法运算次数，但它并未直接指出如何进行这种最优代价的矩阵链乘法计算。表 $s[1..n-1, 2..n]$ 记录了构造最优解所需的信息。每个表项 $s[i, j]$ 记录了一个 k 值，指出 $A_i A_{i+1} \dots A_j$ 的最优括号化方案的分割点应在 A_k 和 A_{k+1} 之间。因此，我们知道 $A_{1..n}$ 的最优计算方案中最后一次矩阵乘法运算应该是 $A_{1..k} A_{k+1..n}$ 。我们可以用相同的方法递归地求出更早的矩阵乘法的具体计算过程，因为 $s[1, s[1, n]]$ 指出了计算 $A_{1..k} A_{k+1..n}$ 时应进行的最后一次矩阵乘法运算； $s[s[1, n]+1, n]$ 指出了计算 $A_{k+1..n}$ 时应进行的最后一次矩阵乘法运算。下面给出的递归过程可以输出 $(A_1, A_{k+1}, \dots, A_k)$ 的最优括号化方案，其输入为 MATRIX-CHAIN-ORDER 得到的表 s 及下标 i 和 j 。调用 PRINT-OPTIMAL-PARENS($s, 1, n$) 即可输出 (A_1, A_2, \dots, A_n) 的最优括号化方案。

《导论》中文3版, P215

第8讲 动态规划方法(1)

66

矩阵链相乘的DP算法—求解步骤

步骤四：构造最优解

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i==j
2      print "A";
3  else print "("
4      PRINT-OPTIMAL-PARENS(s, i, s[i,j])
5      PRINT-OPTIMAL-PARENS(s, s[i,j]+1, j)
6      print ")"
```

对图 15-5 中的例子，调用 PRINT-OPTIMAL-PARENS(s, 1, 6)输出括号化方案
 $((A_1(A_2A_3))((A_4A_5)A_6))$

《导论》中文3版, P215

目录

- Fibonacci数的计算效率
- 动态规划方法概述
- DP入门—DAG中的最短路径
- 最长递增子序列的DP算法
- 两个字符串间编辑距离 (Levenshtein距离) 的DP算法
- 矩阵链相乘的DP算法

The End
Thanks!