山东师范大学 SHANDONG NORMAL UNIVERSITY

《算法设计与分析》
第5.01讲 动态规划方法(1)

殷合川
山东师范大学信息科学与工程学院

2014年10月

目录

动态规划(dynamic programming)—概述

- 动态规划是运筹学的一个分支，是求解决策过程(decision process)最优化的数学方法
  - 20世纪50年代初美国数学家Richard E. Bellman等人在研究多阶段决策过程(multistep decision process)的优化问题时，提出了著名的最优化原理(principle of optimality)，把多阶段过程转化为一系列单阶段问题，利用各阶段之间的关系，逐个求解，创立了解决这类过程优化问题的新方法—动态规划。
  - Bellman 1957年出版的名著《Dynamic Programming》是该领域的第一部权威著作。

动态规划程序设计—概述

- 动态规划程序设计是求解最优化问题的一种途径、一种方法，而不是一种特殊算法
  - 它不象搜索算法或数值计算那样，具有一个标准的数学表达式和明确清晰的解题方法。
  - 动态规划程序设计往往是根据具体优化问题的性质和最优解的条件，以动态规划的思想提出有针对性的解题方法。
  - 不存在一种万能的解决各类最优化问题的动态规划算法。

动态规划程序设计—概述

- 运用动态规划方法解决问题，需要具体问题具体分析
  - 首先要深入地理解和掌握DP的基本概念和方法
  - 其次要对具体问题进行具体的分析处理，以丰富的想象力去建立模型，用创造性的技巧去构造具体的求解算法
  - 对有代表性问题的动态规划求解算法的学习、分析和推敲，是掌握这一方法的重要途径

动态规划—基本思想

- DP通常用于求解具有某种最优性质的问题。
  - 这类问题可能会有许多可行解。每一个解都对应一个值，我们希望找到值最优的解。
  - DP有与分治法相似的方面，即将待求解问题分解成若干个子问题，先求解子问题，然后将子问题的解合并得到原问题的解。
  - 但它与分治法有很大的不同，适合于用DP求解的问题，其子问题往往不是互相独立的。
  - 若用分治法来解这类问题，则分解得到的子问题数目太多，而且子问题可能会被重复计算很多次。

动态规划—基本思想

- DP通常用于求解具有某种最优性质的问题。
  - DP通过保存已解决的子问题的答案，在需要时直接取出答案的值，来避免大量的重复计算，从而节省时间。
  - 动态规划法的最基本思路就是用一个表来保存已求解子问题的结果。
    可以根据后续子问题对已解决子问题的依赖关系设计空间效率合理的结果保存策略。
  - 具体的动态规划算法多种多样，但它们具有保存已求解子问题答案的特征。

动态规划—基本问题结构

- 在多阶段决策问题中，各个阶段采取的决策一般来说是与时间有关的，决策依赖于当前状态，又随即引起状态的转移。
- 一个决策序列就是在变化的状态中产生出来的，故有“动态”的含义，称这种解决多阶段决策最优化问题的方法为动态规划方法。

动态规划—基本模型

1. 确定问题的决策对象。
2. 对决策过程划分阶段。
3. 对各阶段确定状态变量。
4. 根据状态变量确定费用函数和目标函数。
5. 建立各阶段状态变量的转移过程，确定状态转移方程

动态规划—适用条件

1. 最优化原理（最优子结构性质）：
   - 一个最优化策略应该具有这样的性质，不论过去状态和决策如何，对前面的决策所形成的状态而言，余下的诸决策必须构成最优策略。
   - 简而言之，一个最优化策略的子策略总是最优的。
   - 一个问题满足最优化原理又称其具有最优子结构性质。

动态规划—适用条件

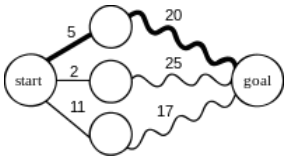
2. 无后效性
   - 将各阶段按照一定的次序排列好之后，对于某个给定的阶段状态，它以前各阶段的状态无法直接影响它未来的决策，而只能通过当前的这个状态。
   - 换句话说，每个状态都是过去历史的一个完整总结。
   - 这就是无后向性，又称为无后效性。

动态规划—适用条件

3. 子问题的重叠性
   - 动态规划将原来具有指数级时间复杂度的搜索算法改进成了具有多项式时间复杂度的算法。
   - 其中的关键在于解决冗余，这是动态规划算法的根本目的。
   - 动态规划实质上是一种以空间换时间的技术。它在实现过程中，需要存储运算过程中的各种状态，所以其空间复杂度一般都会大于其它的算法。

最优子结构(Optimal substructure)

- 在计算机科学中，说一个问题具有最优子结构指的是其最优解可以由其子问题的最优解来构造。
  - 如图的最短路径问题



DP效率示例—Fibonacci数的递归解

```
function fib(n)
    if n = 0 return 0
    if n = 1 return 1
    return fib(n − 1) + fib(n − 2)
```

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

复杂度为指数级的!!!

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))

DP效率示例—Fibonacci数的递归解

- 计算第n个Fibonacci数需要调用fib()函数
  fib(n-1)+ fib(n-2)+1次
- 是一个比Fibonacci数还要大的数

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

- 复杂度至少是指数级的!!!

DP效率示例—Fibonacci数的DP解

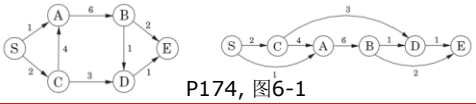
```
function FibDP(n)
  for i=1 to n
    if n=1 then f[1] = 1
    else if n=2 then f[2] = 1
    else
      f[i] = f[i-1]+f[i-2]
  end for
  return f[n]
end
```

复杂度为Θ(n)，线性的！

```
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
```


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，使得所有的边有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
  - dist(D) = min{dist(B)+1, dist(C)+3}

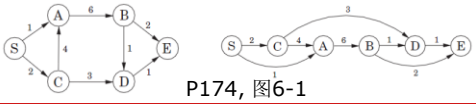


P174, 图6-1

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

initialize all $dist(\cdot)$ values to ∞
$dist(s) = 0$
for each $v \in V\backslash\{s\}$, in linearized order:
  $dist(v) = \min_{(u,v)\in E}\{dist(u) + l(u,v)\}$



P174, 图6-1

DP入门—DAG最短路径算法的DP解释

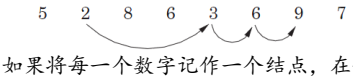
1. 最优子结构
   - 一个结点X到T的距离仅由其各前趋结点Pi到T的距离和Pi到X的边的权决定
2. 无后效性
   - 对DAG线性化后，按各结点的线性序计算距离可以保证结点X的距离仅依赖与其各前趋结点Pi的距离值，与其他更靠前的结点无关
3. 子问题的重叠性
   - 某个结点Y的距离可以用于计算其所有后继结点的距离，因而有计算一次使用多次的性质

最长递增子序列—问题描述

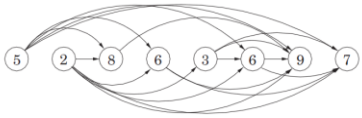
- 子序列
  - 给定数字序列$a_1, \ldots, a_n$，满足$1 \le i_1 \le i_2 \le \cdots \le i_k \le n$的子集$a_{i_1}, a_{i_2}, \ldots, a_{i_k}$称为它的一个子序列
- 递增子序列
  - 如果子序列中的各个数字都是严格单调递增的，则称其为一个递增子序列
- 问题
  - 给定一个数字序列，如何找出其中最长的递增子序列？

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



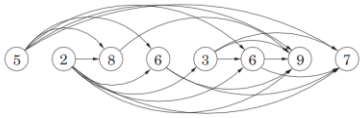
- 如果将每一个数字记作一个结点，在每一对有递增关系的数字之间画一条边可得如下的有向图



P175

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！



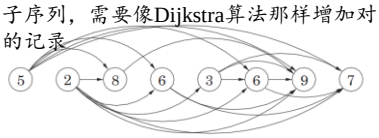
P175

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

```
for j = 1, 2, ..., n:
    L(j) = 1 + max{L(i) : (i, j) ∈ E}
return max_j L(j)
```

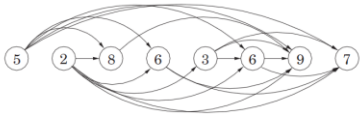
- 该算法求出的是最长递增子序列的长度，要获得子序列，需要像Dijkstra算法那样增加对前趋结点的记录



P175

最长递增子序列—DP解法练习

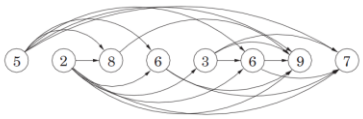
- 记$\{i: (i, j) \in E\} = \emptyset$时$\max\{L(i): (i, j) \in E\} = 0$
- j=1: L(1) = 1; j=2: L(2) = 1



P175

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) = 1; j=2: L(2) = 1



P175

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$\{L(j):1\le j\le n\}$，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所要求的三个基本条件
  - 最优子结构、无后效性、子问题的重叠性

最长递增子序列—DP算法复杂度

for $j=1,2,\ldots,n$:
  $L(j)=1+\max\{L(i):(i,j)\in E\}$
return $\max_j L(j)$

- 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$O(|V||E|)=O(n|E|)=O(n^2)$。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$O(n)$

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

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$\mathrm{lev}_{a,b}(|a|,|b|)$给出，它定义为

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

- 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解—子问题构造

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

$$\begin{matrix} x[i] \\ - \end{matrix} \quad \text{or} \quad \begin{matrix} - \\ y[j] \end{matrix} \quad \text{or} \quad \begin{matrix} x[i] \\ y[j] \end{matrix}$$

编辑距离的DP解—子问题分析

$$\begin{matrix} x[i] \\ - \end{matrix} \quad \text{or} \quad \begin{matrix} - \\ y[j] \end{matrix} \quad \text{or} \quad \begin{matrix} x[i] \\ y[j] \end{matrix}$$

- 情况1：该列产生代价1，余下的问题是最佳对齐$x[1\ldots i-1]$和$y[1\ldots j]$，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$x[1\ldots i]$和$y[1\ldots j-1]$，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$x[1\ldots i-1]$和$y[1\ldots j-1]$，而它对应子问题$E(i-1,j-1)$

编辑距离的DP解—子问题分析

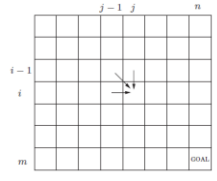
- 显然$E(i,j)$应该是上述三个情况中的最优者，即
- $E(i,j)=\min\begin{cases}1+E(i-1,j),\\ 1+E(i,j-1),\\ diff(i,j)+E(i-1,j-1)\end{cases}$
- 其中$diff(i,j)=\begin{cases}0 & \text{当}x[i]=y[j]\\ 1 & \text{当}x[i]\neq y[j]\end{cases}$
- 尽管得到的是一个递归关系式，但是如果用递归方法解决显然复杂度是不可接受的

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

(a)



(b)

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

E X P O N E N – T I A L
– – P O L Y N O M I A L

编辑距离的DP解—练习

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

编辑距离的DP解—练习

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

编辑距离的DP解—算法伪代码与复杂度

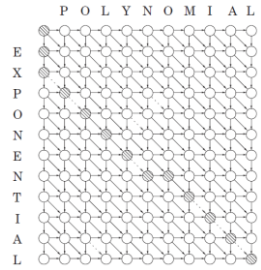
```
for i = 0, 1, 2, ..., m:
    E(i, 0) = i
for j = 1, 2, ..., n:
    E(0, j) = j
for i = 1, 2, ..., m:
    for j = 1, 2, ..., n:
        E(i, j) = min{E(i − 1, j) + 1, E(i, j − 1) + 1, E(i − 1, j − 1) + diff(i, j)}
return E(m, n)
```

显然该算法的时间和空间复杂度均为$O(mn)$

编辑距离问题—隐含的DAG

- 每个DP都隐含着一个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：$(i-1,j)\to(i,j),(i,j-1)\to(i,j),(i-1,j-1)\to(i,j)$。
- 如果将$x[i]=x[j]$时边$(i-1,j-1)\to(i,j)$的权定为0，其它所有边的权均记为1，则编辑距离即是从$(0,0)$到(m,n)的距离，即最短路径。

编辑距离问题—隐含的DAG示例

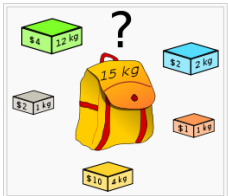


DP中子问题的通用构造方法

1. 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
2. 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
3. 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
4. 输入为一棵树，则其每个子树都可以作为一个子问题

背包问题(knapsack problem)—概述

- 背包问题(Knapsack problem)是一种组合优化的NP完全问题。问题可以描述为：
  - 给定一组物品，每种物品都有自己的重量和价格，在限定的总重量内，我们如何选择，才能使得物品的总价格最高。
  - 问题的名称来源于如何选择最合适的物品放置于给定背包中。



背包问题的一个例子：应该选择哪些盒子，才能使价格尽可能地大，而保持重量小于或等于15 kg?

背包问题—概述

- 给定重量W和n种物品，其重量分别为$w_1, w_2, \ldots, w_n$，价值分别为$v_1, v_2, \ldots, v_n$，怎样的组合才能得到最高的价值？

Item	Weight	Value
1	6	$30
2	3	$14
3	4	$16
4	2	$9

W=10

- 当每种物品数量都是无限(多副本背包问题)时，右侧示例的答案是选1号物品1件、4号物品2件，得总重量10，总价值48
- 当每种物品数量都只有1件(单副本背包问题)时，上面示例的答案是选1号物品和3号物品，得总重量10，总价值46

多副本(无界)背包问题的DP解

- 子问题定义
  - $K(w)$为容量w所能容纳的最高价值，则$K(W)$就是问题的解
- 最优子结构
  - 如果所有可能的$K(w-w_i)$已经获得，$K(w)$就应该是各$K\big((w-w_i)+w_i\big)=K(w-w_i)+v_i$中的最高者，即$K(w)=\max\limits_{i:w_i\le W} K(w-w_i)+v_i$

多副本(无界)背包问题的DP解

- 伪代码

```
K(0) = 0
for w = 1 to W:
    K(w) = max{K(w − w_i) + v_i : w_i ≤ w}
return K(W)
```

- 复杂度
  - 循环共执行W次，每次对所有的物品进行尝试，需要$O(n)$次尝试，总复杂度为$O(nW)$
  - 这是一个伪多项式时间算法，因为当W应该用二进制位数m表示，因而时间复杂度应为$O(n2^m)$，是指数时间的
  - 空间复杂度显然为$O(W)=O(2^m)$

多副本(无界)背包问题的DP解—练习

	v	w	1	2	3	4	5	6	7	8	9	10
1	30	6	0	0								
2	14	3	0	0								
3	16	4	0	0								
4	9	2	0	9								
K(w)			0	9								

多副本(无界)背包问题的DP解—练习

	v	w	1	2	3	4	5	6	7	8	9	10
1	30	6	0	0								
2	14	3	0	0								
3	16	4	0	0								
4	9	2	0	9								
K(w)			0	9								

单副本(0-1)背包问题的DP解

- 子问题定义
  - $K(w,j)$为基于容量w和物品$1,\ldots,j$所能得到的最高价值，则$K(W,n)$就是问题的解
- 最优子结构
  - $K(w,j)$只可能以两种方式获得
    - 一是$K(w-w_j,j-1)$，此时有$K(w,j)=K(w-w_j,j-$

单副本(0-1)背包问题的DP解

- 伪代码

```
Initialize all K(0, j) = 0 and all K(w, 0) = 0
for j = 1 to n:
    for w = 1 to W:
        if w_j > w:  K(w, j) = K(w, j − 1)
        else:  K(w, j) = max{K(w, j − 1), K(w − w_j, j − 1) + v_j}
return K(W, n)
```

- 复杂度
  - 算法含有双重循环，时间复杂度显然为$O(nW) = O(n2^m)$，与多副本背包问题相同
  - 对每一个重量上都要保存与每件物品对应的最佳价值，因而空间复杂度与时间复杂度相同

单副本(0-1)背包问题的DP解—练习

	v	w	0	1	2	3	4	5	6	7	8	9	10
			0	0	0	0	0	0	0	0	0	0	0
1	30	6	0										
2	14	3	0										
3	16	4	0										
4	9	2	0										

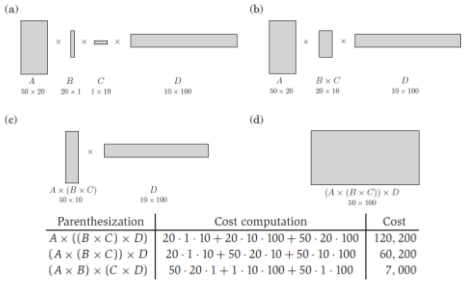
单副本(0-1)背包问题的DP解—练习

	v	w	0	1	2	3	4	5	6	7	8	9	10
			0	0	0	0	0	0	0	0	0	0	0
1	30	6	0										
2	14	3	0										
3	16	4	0										
4	9	2	0										

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



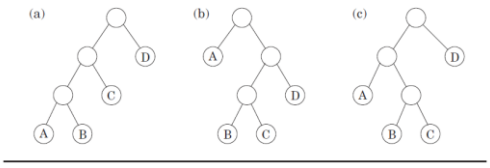
Parenthesization	Cost computation	Cost
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	120, 200
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	60, 200
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	7, 000

矩阵链相乘—问题分析

- 假设需要计算$A_1 \times A_2 \times \cdots \times A_n$，其中各矩阵的维数分别为$m_0 \times m_1, m_1 \times m_2, \ldots, m_{n-1} \times m_n$
  - 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$A_i \times A_{i+1} \times \cdots \times A_j$的矩阵子链积
  - 对于$1 \le i \le j \le n$定义$C(i,j)$为计算$A_i \times A_{i+1} \times \cdots \times A_j$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
  - $C(i,j) = \min\limits_{i \le k < j}\{C(i,k) + C(k+1,j) + m_{i-1} \times m_k \times m_j\}$

矩阵链相乘—DP伪代码

```
for i = 1 to n:  C(i, i) = 0
for s = 1 to n − 1:
  for i = 1 to n − s:
    j = i + s
    C(i, j) = min{C(i, k) + C(k + 1, j) + m_{i−1} · m_k · m_j : i ≤ k < j}
return C(1, n)
```

- 所有的子问题构成了一个$(n+1) \times (n+1)$的二维表格，因而空间复杂度为$O(n^2)$
- 每个单元格最多有$O(n)$次的矩阵乘法运算，总的时间复杂度为$O(n^3)$次矩阵乘法运算

图中所有顶点对间的最短路径

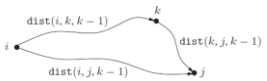
- 如果考虑图中可能有负权边的一般情况，求图中所有顶点对间的最短路径，可以通过运行Bellman-Ford算法$|V|$次来实现，这种方法的复杂度为$O(|V|^2|E|)$
- 使用DP可以获得较有效的$O(|V|^3)$的算法，这就是Floyd-Warshall算法

Floyd-Warshall算法—基本思想

- 首先，在不允许有任何中间结点的情况下，任意一对结点(u,v)之间如果存在一条边，则其距离就是该边的权，如果不存在边，则其距离就是∞
- 接下来，我们逐渐加入中间结点，并不断更新所有的最短路径，则当全部顶点均加入完毕后，就能得到所有各对顶点间的最短路径

Floyd-Warshall算法—最优子结构

- 记$dist(i,j,k)$为允许$\{1,2,\ldots,k\}$作为中间结点时i到j的最短路径长度
  - 当存在i到j的边时，$dist(i,j,0)$为该边的长度，否则取$dist(i,j,0)=\infty$
- 假定已经得到了所有的$dist(i,j,k-1)$，则加入第k个结点后，应该有如下的更新规则
  - $dist(i,j,k)=\min\big(dist(i,k,k-1)+dist(k,j,k-$



Floyd-Warshall算法—伪代码与复杂度

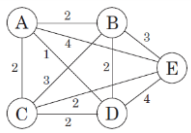
- 伪代码

```
for i = 1 to n:
    for j = 1 to n:
        dist(i, j, 0) = ∞
for all (i, j) ∈ E:
    dist(i, j, 0) = ℓ(i, j)
for k = 1 to n:
    for i = 1 to n:
        for j = 1 to n:
            dist(i, j, k) = min{dist(i, k, k − 1) + dist(k, j, k − 1), dist(i, j, k − 1)}
```

- 复杂度
  - 显然Floyd-Warshall算法的复杂度是$O(|V|^3)$

旅行商问题(TSP, Travelling Salesman Problem)

- TSP问题定义
  - 给定以$1,2,\ldots,n$标记的n座城市和城市间距离矩阵$D=\{d_{ij}\}$，求家乡为城市1的商人从城市1出发遍历其他城市各一次最后回到家乡的最短路径



P191，图6-9
最短路径：
A-D-B-E-C-A
长度：10

TSP—分析

- 使用穷尽式搜索，需要耗费$O(n!)$的时间
- 使用DP可以获得比$O(n!)$好一些的结果
- 子问题定义
  - 对于包含城市1的子集$S\subseteq\{1,2,\ldots,n\}$以及$j\in S$,令$C(S,j)$为从1出发终点为$j$的经过$S$中所有结点恰好一次的最短路径长度
  - 当$|S|>1$时，令$C(S,1)=\infty$，因为路径不能同时从1出发和结束

TSP—DP算法

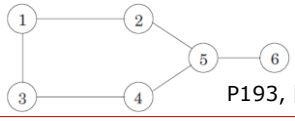
- 最优子结构：$C(S,j)$应该由某个$C(S-\{j\},i)$与d_{ij}得来，即：$C(S,j)=\min_{i\in S-\{j\}} C(S-\{j\},i)+d_{ij}$
- 伪代码

```
C({1}, 1) = 0
for s = 2 to n:
    for all subsets S ⊆ {1, 2, ..., n} of size s and containing 1:
        C(S, 1) = ∞
        for all j ∈ S, j ≠ 1:
            C(S, j) = min{C(S − {j}, i) + d_ij : i ∈ S, i ≠ j}
return min_j C({1, ..., n}, j) + d_j1
```

- 复杂度
  - 子集共有2^n个，每个子集涉及$O(n)$个子问题，每个子问题需要对$O(n)$个元素进行min运算，因而总复杂度为$O(n^2 2^n)$

树中的独立集

- 结点集$S\subset V$称为图$G(V,E)$的一个独立集，是指对于其中任意两个结点，E中都不存在将其相连的边
  - 下图中的结点集$\{1,5\}$构成了一个独立集，而$\{1,4,5\}$却不是。
  - $\{1,4,6\}$和$\{2,3,6\}$是该图的两个最大独立集



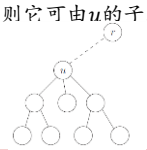
P193, 图6-10

树中的独立集—DP解

- 与背包和TSP问题类似，寻找图的最大独立集是非常困难的。
  - 但是如果图成为了一棵树，则可以利用DP方法在线性时间内求解。
- 子问题构造
  - 树中的任意一棵子树均称为该问题的一个子问题，定义根结点为u的子树的最大独立集为$I(u)$，则根结点为r的树的最大独立集可以由$I(r)$表示和求出。

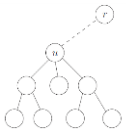
树中的独立集—DP解

- 最优子结构
  - 根结点为u的子树的最大独立集S或者包括u，或者不包括u
  - 如S包括u，则它一定不会包括u的子结点，而要包括u的各个孙结点
  - 如S不包括u，则它可由u的子结点的最大独立集合并得到



树中的独立集—DP解

- 由最优子结构可得下面的DP式
  - $I(u) = \max\{1 +$



P194, 图6-11

目录

The End
Thanks