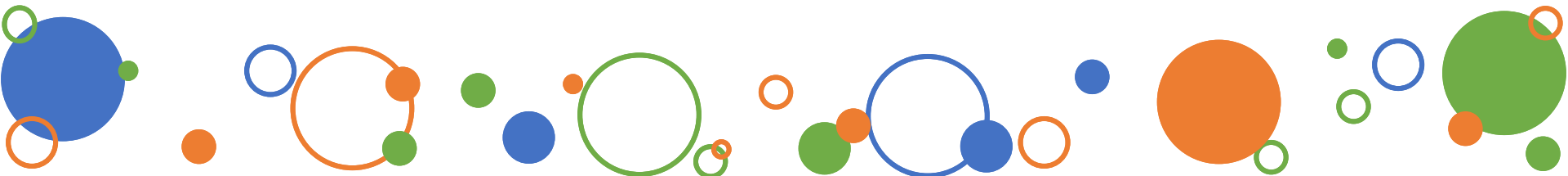
DiCA

Data Intelligence & Computing Art Laboratory

算法分析

刘权辉

2024 春



第三章：动态规划

回顾

问题可以用分治算法高效求解的特征

- 该问题的规模缩小到一定的程度就可以容易地解决；
- 该问题可以分解为若干个规模较小的相同问题，即该问题具有**最优子结构性质；**
- 利用该问题分解出的子问题的解可以合并为该问题的解；
- 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。

分治法的求解步骤，及时间效率分析

```
divide-and-conquer(P)
  {
        if ( | P | <= n0) adhoc(P);   //解决小规模的问题
        divide P into smaller subinstances P1,P2,...,Pk；//分解问题
        for (i=1,i<=k,i++)
            yi=divide-and-conquer(Pi);  //递归的解各子问题
        return merge(y1,...,yk);  //将各子问题的解合并为原问题的解
  }
```

$$T(n)=\begin{cases} O(1) & n=1 \\ kT(n/m)+f(n) & n>1 \end{cases}$$

学习要点

- 理解动态规划算法的概念。
- 掌握动态规划算法的基本要素
  - 最优子结构性质
  - 重叠子问题性质
- 掌握设计动态规划算法的步骤。
  - 找出最优解的性质，并刻画其结构特征。
  - 递归地定义最优值。
  - 以自底向上的方式计算出最优值。
  - 根据计算最优值时得到的信息，构造最优解。

学习要点

- 通过应用范例学习动态规划算法设计策略。
  - 矩阵连乘问题；
  - 凸多边形最优三角剖分；
  - 最长公共子序列；
  - 0-1背包问题;
  - TSP问题。

引言

- 将待求解问题分解为若干子问题，通过子问题的解得到原问题的解，这是问题求解的有效途径。

- 但是，如何实施分解？

- 分治策略的基本思想是将规模为n的问题分解为k个规模较小的子问题，各个子问题相互独立但与原问题求解策略相同。然而，并不是所有问题都可以这样处理。

引言

- 问题分解的另一个途径是将求解过程分解为若干阶段，依次求解每个阶段即得到原问题的解。通过分解得到的各个子阶段不要求相互独立，但希望他们具有相同类型，而且前一阶段的输出可以作为下一阶段的输入。

- 这种策略特别适合求解具有某种最优性质的问题。

- 动态规划属于这类求解策略。

引言

- 动态规划是一个分阶段判定决策过程，其问题求解策略的基础是决策过程的最优原理：
  - 假设为达到某问题的最优目标T，需要依次作出决策序列$D=<D_1, D_2, \ldots, D_k>$。如果D是最优的，则对任意$i\ (1 \le i < k)$，决策子序列$(D_{i+1}, \ldots, D_k)$也是最优的。
  - 即，当前决策的最优性取决于其后续决策序列是否最优。
  - 由此追溯至目标，再由最终目标决策向上回溯，导出决策序列$D=<D_1, D_2, \ldots, D_k>$；
  - 因此，动态规划方法可以保证问题求解是全局最优的。

算法基本思想

- 动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题。

- 但是经分解得到的子问题往往不是互相独立的。不同子问题的数目常常只有多项式量级。在用分治法求解时，有些子问题被重复计算了许多次。

- 如果能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，就可以避免大量重复计算，从而得到多项式时间算法。

动态规划基本步骤

- 找出最优解的性质，并刻画其结构特征；
- 递归地定义最优值；
- 以自底向上的方式计算出最优值；
- 根据计算最优值时得到的信息，构造最优解。

动态规划的基本要素

□ 最优子结构

✓ 问题的最优解包含着其子问题的最优解。这种

注意：

➢ 同一个问题可以有多种方式刻画它的最优子结构，有些表示方法的求解速度更快（空间占用小，问题的维度低）

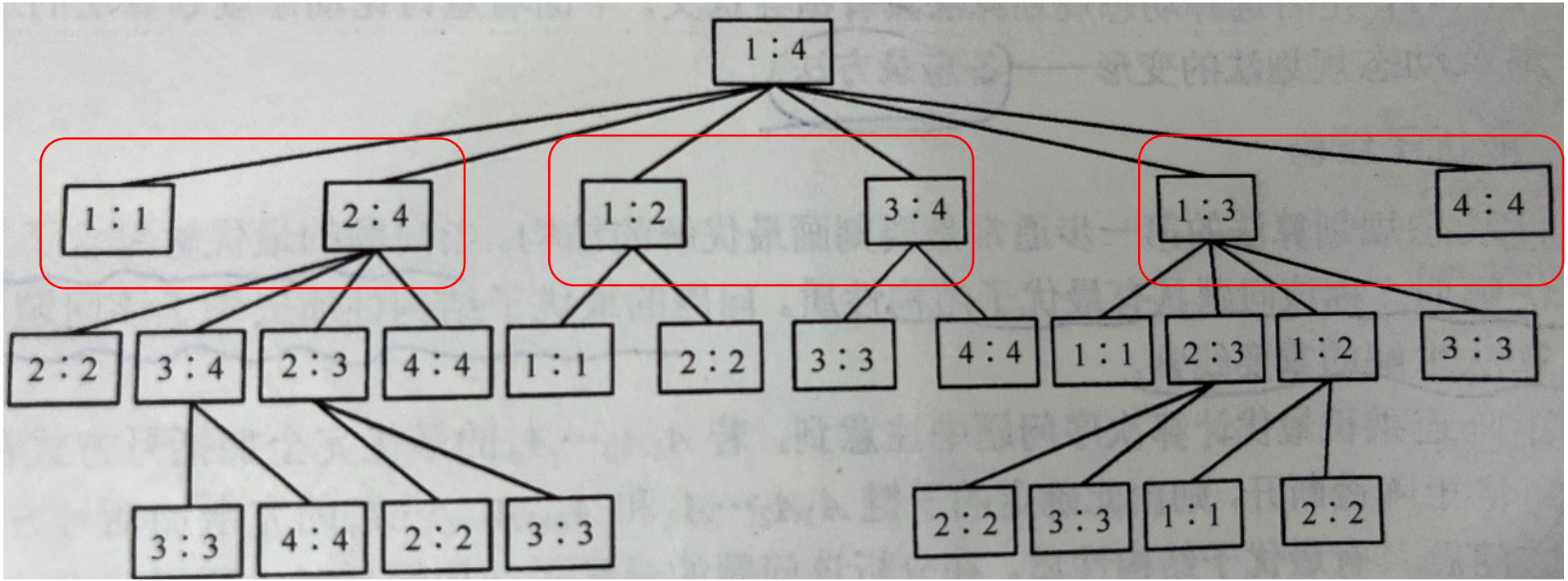
从而导致矛盾。

✓ 利用问题的最优子结构性质，以**自底向上**的方式递归地从子问题的最优解逐步构造出整个问题的最优解。最优子结构是问题能用动态规划算法求解的**前提**。

动态规划的基本要素

重叠子问题

- 递归算法求解问题时，每次产生的子问题并**不总是新问题**，有些子问题被反复计算多次。这种性质称为**子问题的重叠性质。**在递归算法中，子问题被重复计算导致时间复杂度随问题规模n呈指数增长。
- 计算四个矩阵$A_1 * A_2 * A_3 * A_4$, 记为A[1:4]



动态规划的基本要素

动态规划算法，以自底向上的方式对每一个子问题只解一次，而后将其解保存在一个**表格**中，当再次需要解此子问题时，只是简单地用常数时间查看一下结果。通常不同的子问题个数随问题的大小呈**多项式增长**。因此用动态规划算法只需要**多项式时间**，从而获得较高的解题效率。

备忘录方法

- 动态规划算法的变形；
- 与动态规划算法类似，通过表格存储已解决子问题的答案，在下次需要解决子问题时，只要简单地查看子问题的解答，而不必重新计算；
- 不同：递归方式是自顶向下的。

3.1矩阵连乘

完全加括号的矩阵连乘

- 完全加括号的矩阵连乘积可递归地定义为：

（1）单个矩阵是完全加括号的；

（2）矩阵连乘积A是完全加括号的，则A可表示为2个完全加括号的矩阵连乘积B和C的乘积并加括号，即A=(BC)。

完全加括号的矩阵连乘

- 例：设有四个矩阵A、B、C、D，它们的维数分别是：A: 50*10, B: 10*40, C: 40*30, D: 30*5，则总共有五种完全加括号的方式：(A((BC)D)), (A(B(CD))), ((AB)(CD)), ((AB)C)D), ((A(BC))D)；

  矩阵连乘的条件：$A_{p\times q}$与$A_{\mathrm{q}\times \mathrm{r}}$

  矩阵连乘的乘法计算次数：p*q*r

- 如按照上述五种方式进行矩阵乘运算，所需乘法次数为：16000, 10500, 36000, 87500, 34500。

矩阵连乘

- 问题描述：给定n个矩阵 $\{A_1, A_2, \ldots, A_n\}$，其中 A_i 与 A_{i+1} 是可乘的，$i = 1,2,\ldots,n-1$ 。考察这n个矩阵的连乘积 $A_1 A_2 \ldots A_n$

分析：

- 由于矩阵乘法满足结合律，所以计算矩阵的连乘可以有许多不同的计算次序，这种计算次序可以用加括号的方式来确定；
- 若一个矩阵连乘积的计算次序完全确定，也就是说该连乘积已完全加括号，则可以依此次序反复调用2个矩阵相乘的标准算法计算出矩阵连乘积；
- 如何确定矩阵计算的次序，使得连乘次数最少？

矩阵连乘

- **穷举法**：列举出所有可能的计算次序，并计算出每一种计算次序相应乘法的次数，从中找出一种数乘次数最少的计算次序。

复杂度分析：

- 假设n个矩阵的连乘积不同计算次序的种类为P(n)；
- 由于每种加括号方式都可以分解为两个子矩阵的加括号问题：$(A_1 \ldots A_k)(A_{k+1} \ldots A_n)$可以得到关于P(n)的递推式如下：

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & n > 1 \end{cases} \Rightarrow P(n) = \Omega(4^n / n^{3/2})$$

口动态规划：

如何求解？

矩阵连乘

动态规划：

- 将矩阵连乘积 $A_iA_{i+1}\ldots A_j$ 简记为： **A[i:j]**，其中 i≤j
- 考察计算A[i:j]的最优计算次序。设这个计算次序在矩阵A_k和A_{k+1}之间将矩阵链断开， i≤k<j，则其相应完全加括号（(AB)）方式为：

$$(A_iA_{i+1}\ldots A_k)(A_{k+1}A_{k+2}\ldots A_j)$$

- A[i:j]的计算量：A[i:k]的计算量加上A[k+1:j]的计算量，再加上A[i:k]和A[k+1:j]相乘的计算量，具体为：

A_i的维数为： $P_{i-1}\times P_i$

A[i:k]和A[k+1:j]的矩阵维度分别为：$P_{i-1}\times P_k$和 $P_k\times P_j$
则上述两矩阵相乘计算量为：$P_{i-1}P_kP_j$

（1）最优子结构分析：

- 特征：计算A[i:j]的最优次序所包含的计算矩阵子链 A[i:k]和A[k+1:j]的次序也是最优的(反证法证明);
- 最优子结构性质：矩阵连乘计算次序问题的最优解包含着其子问题的最优解;
- 问题的最优子结构性质是该问题可用动态规划算法求解的显著特征。

矩阵连乘

（2）建立递归关系：

- 设计算A[i:j]，1≤i≤j≤n，所需要的最少数乘次数为m[i,j]，则原问题的最优值为m[1,n]。
- 当i=j时，A[i:j]=A_i，因此，m[i,i]=0，i=1,2,...,n
- 当i<j时，$m[i,j]=\min\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\}$

这里 A_i的维数为：$p_{i-1}\times p_i$

- 可以递归地定义m[i,j]为：

k的位置只有j-i种可能

$$m[i,j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & i<j\end{cases}$$

矩阵连乘

（3）计算最优值：

$$m[i,j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & i<j\end{cases}$$

口 分析：

- 许多子问题被重复计算，这也是该问题可用动态规划算法求解的又一显著特征;
- 用动态规划算法解此问题，可依据其递归式以自底向上的方式进行计算。在计算过程中，保存已解决的子问题答案。每个子问题只计算一次，而在后面需要时只要简单查一下，从而避免大量的重复计算，最终得到多项式时间的算法；

（3）计算最优值：

- 基于前面递归式，求m[i,j]时，只需计算：

  m[i,k]和m[k+1,j]（i≤k＜j）

- 因此，先计算m[i,k]和m[k+1,j]（i≤k＜j)就可得到m[i,j]（即按照m[i,j]左边和下方的元素即可得到m[i,j]）；
- 采用**自底向上**方法。

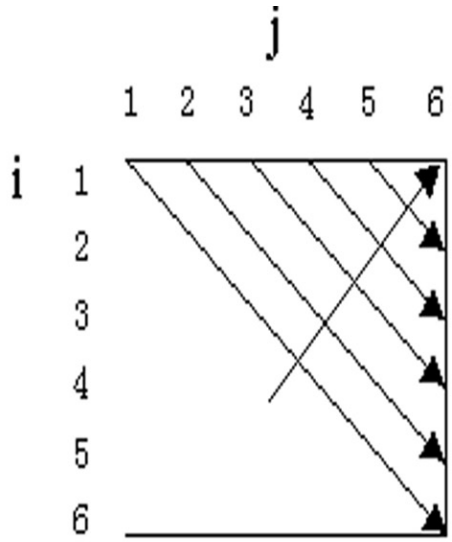
$$m[i,j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & i<j\end{cases}$$

矩阵连乘

- 例如： 对于矩阵连乘：$A_1A_2A_3A_4A_5A_6$

A1	A2	A3	A4	A5	A6
30×35	35×15	15×5	5×10	10×20	20×25

- 计算顺序：



j

i	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

$P_{i-1} \times P_k \times P_r$

$$m[2][5]=\min\begin{cases} m[2][2]+m[3][5]+p_1p_2p_5=0+2500+35\times15\times20=13000 \\ m[2][3]+m[4][5]+p_1p_3p_5=2625+1000+35\times5\times20=7125 \\ m[2][4]+m[5][5]+p_1p_4p_5=4375+0+35\times10\times20=11375 \end{cases}$$

矩阵连乘

矩阵A_i的维度为 p[i-1]*p[i]

```
void MatrixChain(int *p,  int n,  int **m,  int **s){
for (int i = 1; i <= n; i++) m[i][i] = 0;
for (int r = 2; r <= n; r++)
    for (int i = 1; i <= n - r+1; i++) {
        int j=i+r-1;
        m[i][j] =0+ m[i+1][j]+ p[i-1]*p[i]*p[j];
        s[i][j] = i;
        for (int k = i+1; k < j; k++) {
        int t = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
        if (t < m[i][j])
       {    m[i][j] = t; s[i][j] = k; }  }
         }}
```

r：连乘的矩阵个数；该层循环分别用来计算2个、3个、....n个矩阵连乘需要的乘法次数

矩阵连乘

```
void MatrixChain(int *p,  int n,  int **m,  int **s){
for (int i = 1; i <= n; i++) m[i][i] = 0;
for (int r = 2; r <= n; r++)
    for (int i = 1; i <= n - r+1; i++) {
        int j=i+r-1;
        m[i][j] = 0+m[i+1][j]+ p[i-1]*p[i]*p[j];
        s[i][j] = i;
        for (int k = i+1; k < j; k++) {
        int t = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
        if (t < m[i][j])
       {    m[i][j] = t; s[i][j] = k; }  }
         }}
```

i: 控制数组m的行，给定r (r=2), 1个循环得到两个矩阵的连乘所有可能：m[1][2]; m[2][3]; m[3][4]; m[4][5];等。

矩阵连乘

```
void MatrixChain(int *p,  int n,  int **m,  int **s){
for (int i = 1; i <= n; i++) m[i][i] = 0;
for (int r = 2; r <= n; r++)
    for (int i = 1; i <= n - r+1; i++) {
        int j=i+r-1;
        m[i][j] = 0+m[i+1][j]+ p[i-1]*p[i]*p[j];
        s[i][j] = i;
        for (int k = i+1; k < j; k++) {
        int t = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
        if (t < m[i][j])
       {    m[i][j] = t; s[i][j] = k; }  }
         }}
```

j: 控制数组m的列，逐列计算m的值（随着r的增大）

r=2:m[1][2],m[2][3]

r=3: m[1][3],m[2][4]

矩阵连乘

```
void MatrixChain(int *p,  int n,  int **m,  int **s){
for (int i = 1; i <= n; i++) m[i][i] = 0;
for (int r = 2; r <= n; r++)
    for (int i = 1; i <= n - r+1; i++) {
        int j=i+r-1;
        m[i][j] = m[i][i]+m[i+1][j]+ p[i-1]*p[i]*p[j];
        s[i][j] = i; //存储断开位置
        for (int k = i+1; k < j; k++) {
        int t = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
        if (t < m[i][j])
       {    m[i][j] = t; s[i][j] = k; }  }
        }}
```

第一个位置切开：
(A1)(A2A3A4…An)

依次循环计算从第2个矩阵后面切开时的计算量

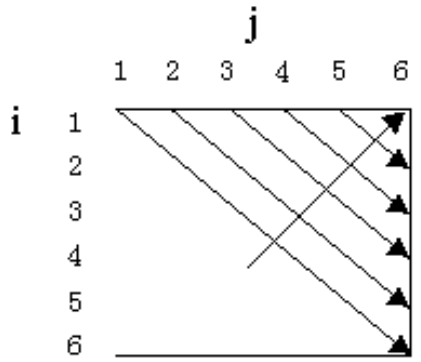
更新、保存最小值及断开位置

矩阵连乘

（4）构造最优解：

- 通过上述方法，即可获得最小的乘积次数为m[1,6]
- 还需得到最优划分方案

$$m[2][5]=\min\begin{cases} m[2][2]+m[3][5]+p_1p_2p_5=0+2500+35\times15\times20=13000 \\ m[2][3]+m[4][5]+p_1p_3p_5=2625+1000+35\times5\times20=7125 \\ m[2][4]+m[5][5]+p_1p_4p_5=4375+0+35\times10\times20=11375 \end{cases}$$



(a) 计算次序

j

i	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

(b) m[i][j]

j

i	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

(c) s[i][j]

（4）构造最优解:

- 由s[1,n]中的k可知计算A[1,n]的最优加括号方式为：(A[1:k])(A[k+1:n])；而A[1:k]中最优的加括号方式为： (A[1:s[1][k]]) (A[s[1][k]+1:s[1][k]])。
- 同理可知, A[k+1,n]中最优的断开位置为:s[k+1][n]
- 递归下去即可得到A[1:n]的加括号方式。

矩阵连乘

（4）构造最优解:

- 由s[1,n]中的k可知计算A[1,n]的最优加括号方式为：(A[1:k])(A[k+1:n])；而A[1:k]中最优的加括号方式为：(A[1:s[1][k]]) (A[s[1][k]+1: s[1][k]])。
- 前例中的最有划分为：$(A_1(A_2A_3))((A_4A_5)A_6)$

m[i][j]

i \ j	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

s[i][j]

i \ j	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

动态规划基本概念

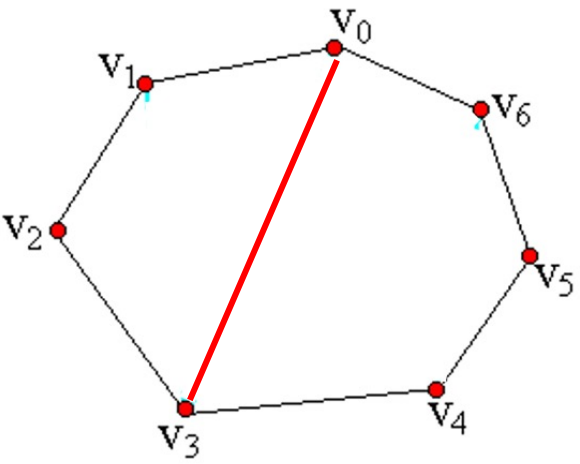
3.2 凸多边形最优三角剖分

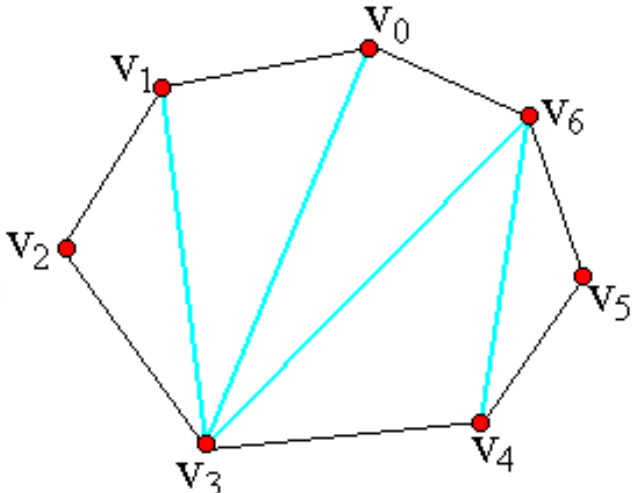
凸多边形最优三角剖分：问题描述

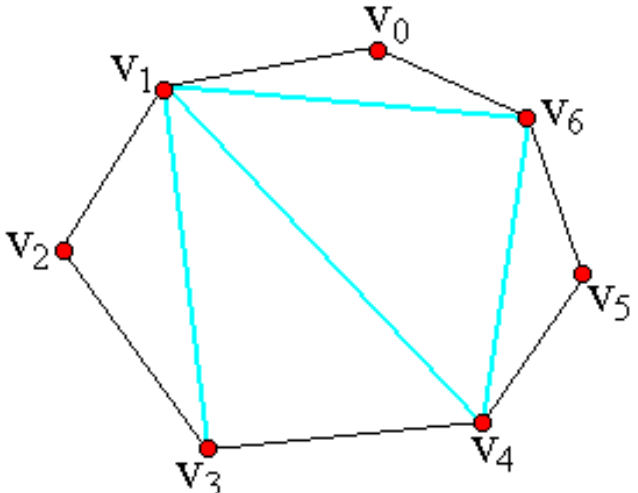
1) 用多边形顶点的逆时针序列表示凸多边形，即P={v_0, v_1,…, v_{n-1}}表示具有n条边的凸多边形;

2) 若v_i与v_j是多边形上**不相邻**的2个顶点，则线段v_iv_j称为多边形的一条弦, 弦将多边形分割成**2个多边形**$\{v_i, v_{i+1},\ldots, v_j\}$**和**$\{v_j, v_{j+1},\ldots, v_i\}$;

3) **多边形P的三角剖分**是将多边形分割成**互不相交三角形的弦的集合T**。在有n个顶点的凸多边形的三角剖分中，恰有**n-3条弦**和**n-2个三角形**；





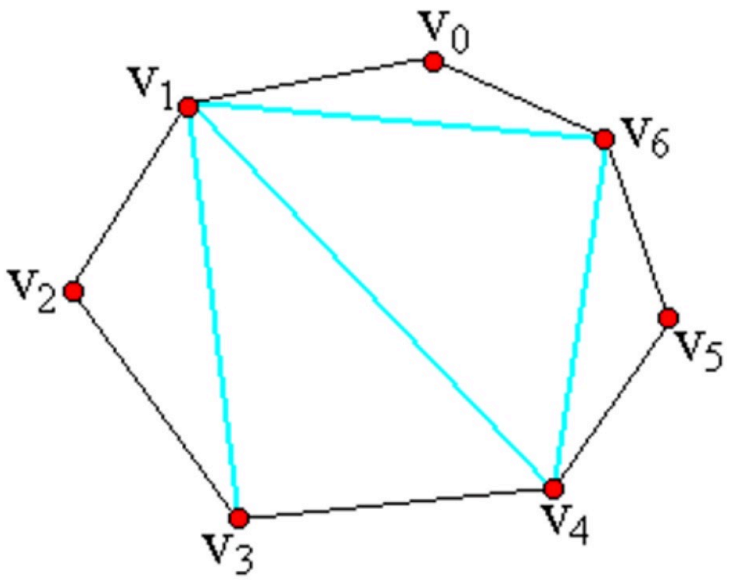


凸多边形最优三角剖分：问题描述

4) 给定凸多边形P，以及定义在由多边形的边和弦组成三角形上的**权函数W**, 要求确定该凸多边形的三角剖分，使得三角剖分得到**诸三角形上权之和为最小，即凸多边形最优三角剖分问题**;

例如：W可定义为三角形三条边的长度之和，

$$W(v_i v_j v_k)=w_{ij}+w_{jk}+w_{ki}$$



$$T=\{v_1v_3\ ,\ v_1v_4\ ,\ v_1v_6\ ,\ v_4v_6\}$$

$$W(v_0v_1v_6)+W(v_1v_2v_3)+W(v_1v_3v_4)+W(v_1v_4v_6)+W(v_4v_5v_6)$$

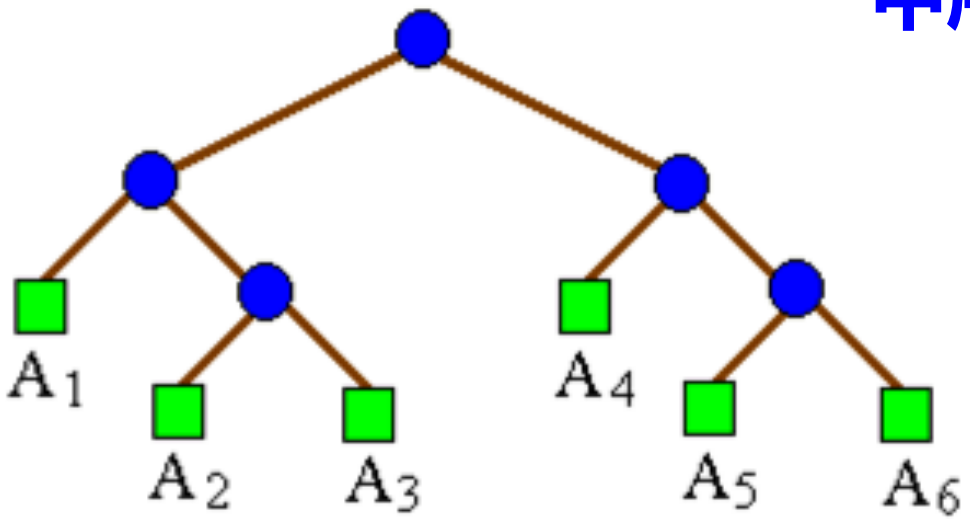
凸多边形最优三角剖分

口三角剖分的结构及其相关问题

- 一个表达式的完全加括号方式相应于一棵完全二叉树，称为**表达式的语法树**；

- 例如，完全加括号的矩阵连乘积$((A_1(A_2A_3))(A_4(A_5A_6)))$对应的语法树如图 (a)所示。

中序遍历算法



(a)

凸多边形最优三角剖分

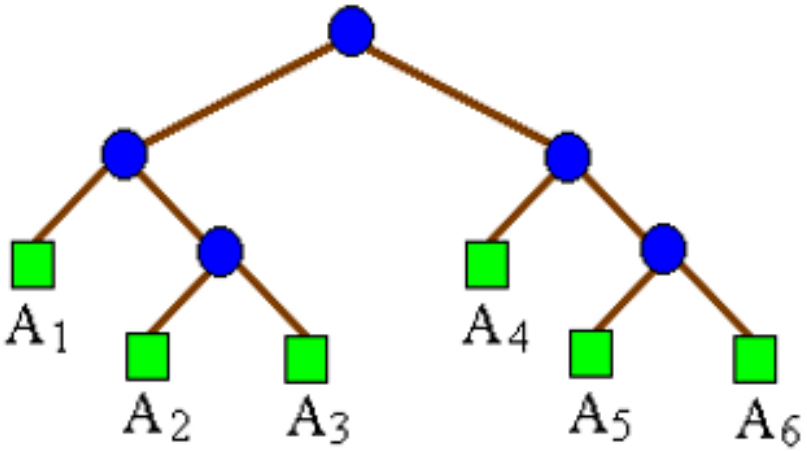
口三角剖分的结构及其相关问题

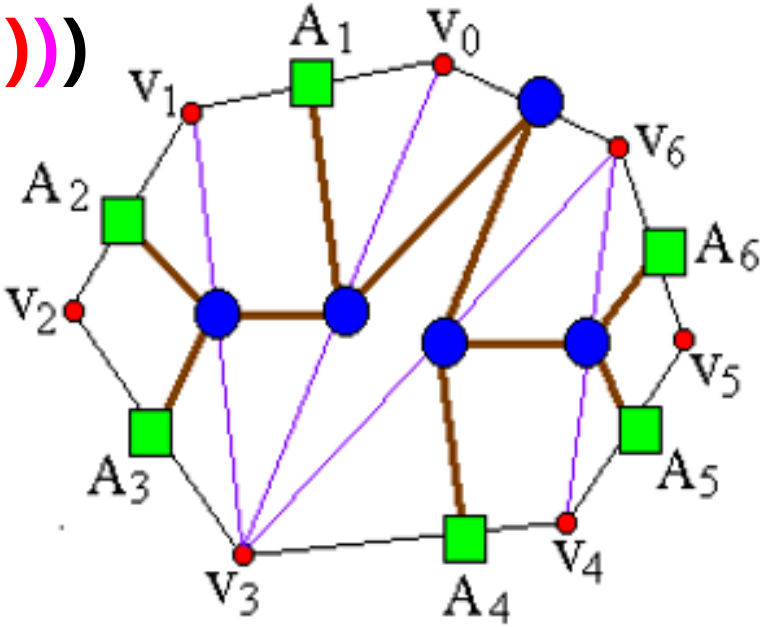
1）凸多边形$\{v_0, v_1, \ldots, v_{n-1}\}$的三角剖分也可以用语法树表示。图 (b)中凸多边形的三角剖分可用图 (a)所示的语法树表示；

2）矩阵连乘积中的每个矩阵A_i对应于凸(n+1)边形中的一条边$v_{i-1}v_i$；

3）三角剖分中的一条弦v_iv_j，i<j，对应于矩阵连乘积A[i+1:j]。

矩阵连乘积 $((A_1(A_2A_3))(A_4(A_5A_6)))$





凸多边形最优三角剖分

口三角剖分的结构及其相关问题（同构性）

- 矩阵连乘积的最优计算次序问题是凸多边形最优三角剖分问题的特殊情形;

① 对于给定的矩阵链$A_1A_2\ldots A_n$，定义与之对应的凸多边形P=$\{v_0, v_1,\ldots, v_n\}$，使得矩阵A_i与凸多边形的边$v_{i-1}v_i$——对应；

② 若矩阵A_i的维数为$p_{i-1}p_i$ (i=1,2,...,n)，则定义三角形$v_iv_jv_k$上的权函数为 w($v_iv_jv_k$)=$p_ip_jp_k$；

③ 则凸多边形P的最优三角剖分对应的语法树给出矩阵链$A_1A_2\ldots A_n$的最优完全加括号方式。

口三角剖分的最优子结构性质

凸多边形的最优三角剖分问题有最优子结构性质。

分析：

- 若凸(n+1)边形P={$v_0, v_1, \ldots, v_n$}的最优三角剖分T包含三角形$v_0 v_k v_n$，1≤k≤n-1，则T的权为3部分权的和：三角形$v_0 v_k v_n$的权，子多边形{$v_0, v_1, \ldots, v_k$}和{$v_k, v_{k+1}, \ldots, v_n$}的权之和。
- 可以断言，由T所确定的这2个子多边形的三角剖分也是最优的。因为若有比{$v_0, v_1, \ldots, v_k$}或{$v_k, v_{k+1}, \ldots, v_n$}更小权的三角剖分将导致T不是最优三角剖分的矛盾（反证法）。

凸多边形最优三角剖分

口三角剖分的递归结构

分析：

- 定义t[i][j] (1≤i<j≤n) 为凸子多边形{v_{i-1}, v_i,..., v_j}的最优三角剖分所对应的权函数值，即其最优值。设退化的多边形{v_{i-1},v_i}具有权值0, 据此定义，要计算的凸(n+1)边形P的最优权值为t[1][n]；
- t[i][j]的值可以利用最优子结构性质递归地计算。当j-i≥1时，凸子多边形至少有3个顶点。由最优子结构性质，t[i][j]的值应为t[i][k]的值加上t[k+1][j]的值，再加上三角形$v_{i-1}v_kv_j$的权值，其中i≤k≤j-1；
- 由于在计算时还不知道k的确切位置，而k的所有可能位置只有j-i个，因此可以在这j-i个位置中选出使t[i][j]值达到最小的位置(切点)。

口三角剖分的递归结构

- 由此，t[i][j]可递归地定义为：

$$t[i][j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\{t[i][k]+t[k+1][j]+w(v_{i-1}v_kv_j)\} & i<j\end{cases}$$

口三角剖分的具体实现

- 参考矩阵连乘（略）


SICHUAN UNIVERSITY
1896
DiCA
Data Intelligence & Computing Art Laboratory

3.3 最长公共子序列

口问题描述

- 若给定序列X=$\{x_1, x_2, \ldots, x_m\}$，序列Z=$\{z_1, z_2, \ldots, z_k\}$；Z是X的子序列是指存在一个严格递增下标序列$\{i_1, i_2, \ldots, i_k\}$使得对于所有j=1,2,…,k有：$z_j = x_{i_j}$；

- 例如，序列Z={B，C，D，B}是序列X={A，B，C，B，D，A，B}的子序列，相应的递增下标序列为{2，3，5，7}。

A，B，C，B，D，A，B

$z_1 = x_2$　$z_2 = x_3$　$z_3 = x_5$　$z_4 = x_7$

最长公共子序列

口问题描述

- 给定2个序列X和Y，当另一序列Z既是X的子序列又是Y的子序列时，称Z是序列X和Y的公共子序列;

- 特别地， 给定2个序列: X=$\{x_1, x_2, \ldots, x_m\}$
  Y=$\{y_1, y_2, \ldots, y_n\}$，
  找出X 和 Y 的最长公共子序列Z=$\{z_1, z_2, \ldots, z_k\}$；

即找出最长且严格递增的下标序列$\{a_1, a_2, \ldots, a_k\}$（$a_k \leq m$）和严格递增的下标子序列$\{b_1, b_2, \ldots, b_k\}$（$b_k \leq n$），使得$x_{a_i} = y_{b_i}$，对任意i=1,2,...,k 成立，则
Z=$\{x_{a_1}, x_{a_2}, \ldots, x_{a_k}\}$ =$\{y_{b_1}, y_{b_2}, \ldots, y_{b_k}\}$ 为X 和 Y 的最长公共子序列。

口最优子结构分析

分析：设序列X=$\{x_1, x_2, \ldots, x_m\}$和Y=$\{y_1, y_2, \ldots, y_n\}$的最长公共子序列为Z=$\{z_1, z_2, \ldots, z_k\}$，$X_{m-1}=\{x_1, x_2, \ldots, x_{m-1}\}$,则:
1)若$x_m=y_n$，则$z_k=x_m=y_n$，且Z_{k-1}是X_{m-1}和Y_{n-1}的最长公共子序列；

证明：i) 假设$z_k \neq x_m$，那么$\{z_1, z_2, \ldots, z_k, x_m\}$ 是 X 和 Y 长度为 k+1的公共子序列，这与Z是X和Y的最长公共子序列矛盾，因此必有$z_k=x_m=y_n$；
ii) 假设 X_{m-1}和Y_{n-1}有长度大于k-1的公共子序列W，则将x_m加在W尾部产生X和Y的长度大于k的公共子序列，由此产生矛盾

口最优子结构分析

分析：设序列X=$\{x_1, x_2, \ldots, x_m\}$和Y=$\{y_1, y_2, \ldots, y_n\}$的最长公共子序列为Z=$\{z_1, z_2, \ldots, z_k\}$，则:

2)若$x_m \neq y_n$且$z_k \neq x_m$，则Z是X_{m-1}和Y的最长公共子序列

证明：若X_{m-1}和Y有长度大于k的公共子序列W，则W也是X和Y的长度大于k的公共子序列，这与Z是X和Y的最长公共子序列矛盾。

3)若$x_m \neq y_n$且$z_k \neq y_n$，则Z是X和Y_{n-1}的最长公共子序列（同上可证）。

2个序列的最长公共子序列包含了2个序列的前缀的最长公共子序列

口递归结构分析

- 由最长公共子序列问题的最优子结构性质建立子问题最优值的递归关系；
- 用c[i][j]记录序列X_i和Y_j的最长公共子序列的长度。其中，$X_i=\{x_1, x_2, \ldots, x_i\}$和$Y_j=\{y_1, y_2, \ldots, y_j\}$；
- 当i=0或j=0时，空序列是X_i和Y_j的最长公共子序列。故此时c[i][j]=0。其它情况下，由最优子结构性质可建立递归关系如下：

$$c[i][j]=\begin{cases} 0 & i=0, j=0 \\ c[i-1][j-1]+1 & i, j>0; x_i=y_j \\ \max\{c[i][j-1], c[i-1][j]\} & i, j>0; x_i\neq y_j \end{cases}$$

最长

$$c[i][j]=\begin{cases}0 & i=0 \text{ 或 } j=0\\ c[i-1][j-1]+1 & i,j>0;x_i=y_j\\ \max\{c[i][j-1],c[i-1][j]\} & i,j>0;x_i\neq y_j\end{cases}$$

口 计算最优值: 动态规划算法自底向上地计算最优值

```
void LCSLength(int m，int n，char *x ，char *y，int **c，int **b)
{
    int i，j;
    for (i = 0; i <= m; i++) c[i][0] = 0;
    for (i = 0; i <= n; i++) c[0][i] = 0;
    for (i = 1; i <= m; i++)
      for (j = 1; j <= n; j++) {
        if (x[i]==y[j]) {
              c[i][j]=c[i-1][j-1]+1; b[i][j]=1;}
        else if ( c[i-1][j]>=c[i][j-1] )
              { c[i][j]=c[i-1][j]; b[i][j]=2;}
        else { c[i][j]=c[i][j-1]; b[i][j]=3; }
    }
}
```

$x_m=y_n$，则$z_k=x_m=y_n$，且Z_{k-1}是X_{m-1}和Y_{n-1}的最长公共子序列

b[i][j] 用于标记c[i][j]的值是由哪个子问题的解得到，用于构造最长公共子序列

$x_{m-1}=y_n$，则$z_k=x_{m-1}=y_n$，原问题等价于求是X_{m-1}和Y_n的最长公共子序列

$x_m=y_{n-1}$，则$z_k=x_m=y_{n-1}$，原问题等价于求是X_m和Y_{n-1}的最长公共子序列

最长公共子序列

$$c[i][j]=\begin{cases}0 & i=0 \text{ 或 } j=0\\ c[i-1][j-1]+1 & i,j>0;x_i=y_j\\ \max\{c[i][j-1],c[i-1][j]\} & i,j>0;x_i\neq y_j\end{cases}$$

- 例：求 X= “ABCBDAB” Y= “BDCABA” 最长公共子序列。

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

注：左图中当遇到 ↖ 时表示最长公共子序列是由X_{i-1}和Y_{j-1}的最长公共子序列在尾部加上X_i所得到的子序列。

X_7，Y_5；X_6，Y_4

X_5，Y_3；{X_4，Y_3或

X_5，Y_2}；

X_5，Y_2 -> X_4，Y_1

-> X_3，Y_0

最长公共子序列

口计算最优值: 构造最长公共子序列(参见上页图例)

$$c[i][j]=\begin{cases}0 & i=0 \text{ 或 } j=0\\ c[i-1][j-1]+1 & i,j>0;x_i=y_j\\ \max\{c[i][j-1],c[i-1][j]\} & i,j>0;x_i\neq y_j\end{cases}$$

```
void LCS(int i，int j，char *x，int **b){
    if (i ==0 || j==0)  return;
    if (b[i][j]== 1) { LCS(i-1，j-1，x，b); cout<<x[i];  }
    else if (b[i][j]== 2)   LCS(i-1，j，x，b);
    else LCS(i，j-1，x，b);
}
```

最长公共子序列

口算法的改进:可进一步将数组b省去

分析：

- 事实上，数组元素c[i][j]的值仅由c[i-1][j-1]，c[i-1][j]和c[i][j-1]这3个数组元素的值所确定;
- 对于给定的数组元素c[i][j]，可以不借助于数组b而仅借助于c本身在时间内确定c[i][j]的值是由c[i-1][j-1]，c[i-1][j]和c[i][j-1]中哪一个值所确定的;
- 如果只需要计算最长公共子序列的长度，则算法的空间需求可大大减少;
- 事实上，在计算c[i][j]时，只用到数组c的第i行和第i-1行。因此，用2行的数组空间就可以计算出最长公共子序列的长度。进一步的分析还可将空间需求减至O(min(m,n))。


四川大学
1896
SICHUAN UNIVERSITY


DiCA
Data Intelligence & Computing Art Laboratory

3.4 0-1背包问题

0-1背包问题

问题描述

- 给定n种物品和一背包。物品i的重量是w_i，其价值为v_i，背包的容量为C。问应如何选择装入背包的物品，使得装入背包中物品的总价值最大?

分析：

- 0-1背包问题是一个特殊的整数规划问题;
- 特点：物品i只有装入、不装入两种状态。

目标

$$\max \sum_{i=1}^{n} v_i x_i$$

条件

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \le C \\ x_i \in \{0,1\}, 1 \le i \le n \end{cases}$$

口最优子结构分析

分析：设$\{y_1, y_2, \ldots, y_n\}$是0-1背包问题的一个最优解，则:

- $\{y_2, \ldots, y_n\}$ 是下列相应子问题的一个最优解；

$$\max \sum_{i=2}^{n} V_i X_i \quad \text{s.t.} \begin{cases} \sum_{i=2}^{n} w_i x_i \le c - w_1 \\ x_i \in \{0,1\}, 2 \le i \le n \end{cases}$$

反证法：假设$\{z_2, \ldots, z_n\}$是上述子问题的最优解，那么

$\Sigma_{i=2}^{n} v_i z_i > \Sigma_{i=2}^{n} v_i y_i$ 且 $v_1 y_1 + \Sigma_{i=2}^{n} v_i z_i > \Sigma_{i=1}^{n} v_i y_i$

说明$\{y_1, z_2, \ldots, z_n\}$是问题的最优解，与假设矛盾

0-1背包问题

口递归结构分析

分析：

- 设0-1背包问题的子问题

$$\max \sum_{k=i}^{n} v_k x_k \quad \text{s.t.} \quad \begin{cases} \sum_{k=i}^{n} w_k x_k \le j \\ x_k \in \{0,1\}, i \le k \le n \end{cases}$$

的最优解为m(i，j)，即 m(i，j)是背包容量为j，可选择物品为i，i+1，…，n时0-1背包问题的最优解。

1）对于不包括第i个物品的子问题，最优解是m(i+1,j)；

2）在包括第i个物品的子问题（j-w_i≥0），最优解是由该物品和从第i+1, i+2, …, n个物品中能够放进重量为j-w_i的背包的最优解组成，这种最优解等于v_i+m(i+1,j-w_i)。

口递归结构分析

分析：

- 由0-1背包问题的最优子结构性质，可以建立计算m(i，j)的递归式如下：

$$m(i,j)=\begin{cases}\max\{m(i+1,j),m(i+1,j-w_i)+v_i\} & j\geq w_i\\ m(i+1,j) & 0\leq j<w_i\end{cases}$$

$$m(n,j)=\begin{cases}v_n & j\geq w_n\\ 0 & 0\leq j<w_n\end{cases}$$

3.5 TSP问题（略）

TSP问题

口问题描述

- 旅行家要旅行n个城市，要求各个城市经历且仅经历一次然后回到出发城市，并要求所走的路程最短;
- 城市间距离用邻接矩阵表示

$$C=\begin{pmatrix} \infty & 3 & 6 & 7 \\ 5 & \infty & 2 & 3 \\ 6 & 4 & \infty & 2 \\ 3 & 7 & 5 & \infty \end{pmatrix}$$

口最优子结构分析

- 设s, s_1, s_2, …, s_p,s 是从s出发的一条路径长度最短的简单回路。假设从s到下一个城市s_1已经求出，则问题转化为求从s_1到s的最短路径。

- 显然s_1, s_2, …, s_p,s一定构成一条从s_1到s的最短路径。

- 如若不然，设s_1, r_1, r_2, …, r_q, s是一条从s_1到s的最短路径且经过n-1个不同城市，则s, s_1, r_1, r_2, …, r_q, s将是一条从s出发的路径长度最短的简单回路且比s, s_1, s_2, …, s_p,s要短，从而导致矛盾。

口递归结构分析

- 假设从顶点s出发，令d(i, V′)表示从顶点i出发经过V′中各个顶点一次且仅一次，最后回到出发点s的最短路径长度。

- 开始时，V′ =V - {s}，于是，TSP问题的递归函数为：

$$\begin{cases} d(i,V') = \min\{c_{ik} + d(k,V' - \{k\})\} & (k \in V') \\ d(i,\{\}) = c_{is} & (i \neq s) \end{cases}$$


SICHUAN UNIVERSITY
1896


DiCA
Data Intelligence & Computing Art Laboratory

End