

第7章 动态规划方法

本章首先介绍动态规划方法的基本思想，包括最优化原理、使用动态规划方法的基本要素以及动态规划方法的基本步骤。然后介绍矩阵连乘问题、多段图问题、最短路径问题和最长公共子序列问题等问题的动态规划算法。

- 动态规划方法的基本思想
- 矩阵连乘问题的动态规划算法
- 使用动态规划方法的基本步骤
- 多段图问题的动态规划算法*
- 最短路径问题的动态规划算法
- 最长公共子序列问题的动态规划算法*

7.1 动态规划方法的基本思想

动态规划方法是处理分段过程最优化问题的一类及其有效的方法。在实际生活中，有一类问题的活动过程可以分成若干个阶段，而且在任一阶段后的行为依赖于该阶段的状态，而与该阶段之前的过程如何达到这种状态的方式无关。这类问题的解决是多阶段的决策过程。在50年代，贝尔曼（Richard Bellman）等人提出了解决这类问题的“最优化原理”，创建了求解最优化问题的一种新的算法设计方法——动态规划方法。用动态规划方法设计的算法称为动态规划算法。

7.1.1 最优化原理

多阶段过程的最优决策序列应当具有这样的性质：无论过程的初始状态和初始决策是什么，其余决策相对于初始决策所产生的状态都构成一个最优决策序列。

对于一个多阶段过程问题，最优决策是否存在依赖于该问题是否具有最优子结构性质（原问题的最优解包含了子问题的最优解）。而是否采用动态规划方法还要看该问题的子问题是否有重叠。问题的最优子结构性质和子问题的重叠性是使用动态规划方法的两个基本要素。先看2个例子。

7.1.2 多段图问题的最优子结构性质

1. 问题描述

多段图是一种简单而经典的模型，在实际问题中应用较多。设 $G=(V,E)$ 是一个有向加权图，顶点集 V 划分成 $m>2$ 个不相交的子集 V_i ， $0\leq i<m$ ，其中， V_0 和 V_{m-1} 只有一个顶点（分别称为源和汇），所有边的始点和终点都在相邻的两个子集中。

多段图问题：求由源到汇的最小成本路径。例如，对于图7-1所示的多段图，由源到汇的一条最小成本路径为0-1-6-9-11。

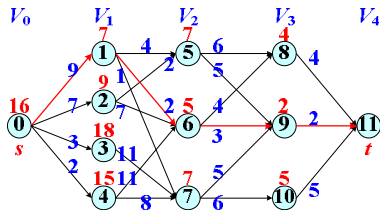
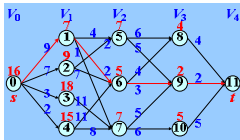


图7-1 一个多段图

2. 最优子结构性质

对于每一条由源 s 到汇 t 的路径，可以看作在 $m-2$ 个阶段做出的某个决策序列，第 i ($0 \leq i < m-2$) 步决策就是确定 V_{i+1} 中哪个顶点属于

该路径。假设路径 $s, v_1, v_2, \dots, v_{m-2}, t$ 是一条由 s 到 t 的最短路径。若从 s (初始状态) 开始，已经做出了到顶点 v_1 的决策 (初始决策)，则 v_1 就是初始决策产生的状态。若将 v_1 看成某个子问题的初始状态，则解这个子问题就是找一条由 v_1 到 t 的最短路径。路径 $v_1, v_2, \dots, v_{m-2}, t$ 一定是一条由 v_1 到 t 的最短路径。否则，设路径 $v_1, q_2, \dots, q_{m-2}, t$ 比 $v_1, v_2, \dots, v_{m-2}, t$ 更短，则路径 $s, v_1, q_2, \dots, q_{m-2}, t$ 比 $s, v_1, v_2, \dots, v_{m-2}, t$ 更短，与 $s, v_1, v_2, \dots, v_{m-2}, t$ 是 s 到 t 的最短路径矛盾。这说明多段图问题具有最优子结构性质。



7.1.3 矩阵连乘问题的最优子结构性质

1. 问题描述

给定 n 个数字矩阵 A_0, A_1, \dots, A_{n-1} ，其中 A_i 与 A_{i+1} 是可乘的， $0 \leq i < n-1$ 。求矩阵连乘 $A_0 A_1 \dots A_{n-1}$ 的加括号方法，使得所用的数乘次数最少。

如果矩阵 A ， B 分别是 $p \times r$ 和 $r \times q$ 矩阵，则它们的乘积 $C = AB$ 是 $p \times q$ 矩阵，其 (i, j) -元素为

$$c_{ij} = \sum_{k=0}^{r-1} a_{ik} b_{kj} \quad (i = 0, \dots, p-1, \quad j = 0, \dots, q-1)$$

因而 AB 所用的数乘次数是 prq 。

3个以上矩阵连乘需要考虑乘积次序问题，即加括号方法。例如，3个矩阵连乘的加括号方法有两种： $((A_0 A_1) A_2)$ 和 $(A_0 (A_1 A_2))$ 。设 A_0, A_1, A_2 分别是 $10 \times 100, 100 \times 5, 5 \times 50$ 矩阵，则两种乘法次序所用的数乘次数分别为：
 $10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$ 和 $10 \times 100 \times 50 + 100 \times 5 \times 50 = 750000$ 。

对于 n 个矩阵的连乘积，如果用 $P(n)$ 表示连乘积的完全加括号方法数（计算次序总数），则有如下递归关系

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & n > 1 \end{cases}$$

从而

$$\begin{aligned} P(n) &\geq 2P(1)P(n-1) = 2P(n-1) \geq 2^2P(n-2) \\ &\geq \dots \geq 2^{n-1}P(1) = 2^{n-1} = 0.5 \times 2^n \end{aligned}$$

$P(n)$ 随 n 指数增长，不能希望列举所有可能次序的连乘积，从中找到数乘次数最少的次序。

2. 最优子结构性质

事实上, 矩阵连乘积问题具有最优子结构性质, 可以采用动态规划方法, 在多项式时间内找到最优的连乘积次序。

用 $A[i..j]$ 表示连乘积 $A_i A_{i+1} \dots A_j$ 。假设计算 $A[0..(n-1)]$ 的一个最优次序 α 在矩阵 A_k 和 A_{k+1} 之间 ($0 \leq k < n-1$) 将连乘积分开, 则完全加括号方式为 $((A_0 A_1 \dots A_k)(A_{k+1} \dots A_{n-1}))$ 。依此次序, 首先分别计算 $A[0..k]$ 和 $A[(k+1)..(n-1)]$, 然后将计算结果相乘得到 $A[0..(n-1)]$ 。易知, $A[0..(n-1)]$ 的一个最优次序所包含的子链 $A[0..k]$ 和 $A[(k+1)..(n-1)]$ 的次序也一定是最优的。否则, $A[0..(n-1)]$ 存在另一个计算次序比 α 更优, 与 α 是计算 $A[0..(n-1)]$ 的一个最优次序矛盾。所以, 矩阵连乘问题具有最优子结构性质。

7.1.4 动态规划方法的基本思路

首先确定原问题的最优值和其子问题的最优值之间的递推关系，然后递归地构造出最优解。最优子结构性质是最优化原理得以采用的先决条件。一般说来，分阶段选择策略确定最优解的问题往往会形成一个决策序列。Bellman认为，利用最优化原理以及所获得的递推关系式去求解最优决策序列，可以使枚举数量急剧下降。这里有一个问题值得注意：最优子结构性质提示我们使用最优化原理产生的算法是递归算法，很可能增加时间与空间复杂度。例如，用递归式直接计算矩阵连乘积 $A[0..(n-1)]$ 的算法的时间复杂度将是 $\Omega(2^n)$ 。可以利用子问题的重叠性想办法避免重复计算而得到效率更高的算法。

7.2 矩阵连乘问题与动态规划的基本步骤

7.2.1 矩阵连乘问题的递归算法

首先递归定义最优值，并建立该问题与其子问题最优值之间的递归关系，然后依据递归关系式用自顶向下的方式进行计算，在计算过程中记录构造最优解的相关信息。

1. 建立递归关系

设 $c(i, j)$ 为用最优法计算 $A[i..j]$ 的消耗, $\text{kay}(i, j)$ 为用最优法计算 $A[i..j]$ 加括号方式的断点。矩阵 A_0, A_1, \dots, A_{n-1} 的行数和列数分别是 (r_0, r_1) , (r_1, r_2) , \dots , (r_{n-1}, r_n) 。 $A[i..j]$ 的最优算法包括用最优算法计算 $A[i..k]$ 和 $A[(k+1)..j]$ 以及计算 $A[i..k] \times A[(k+1)..j]$ 。因为 $A[i..k]$ 和 $A[(k+1)..j]$ 的行数与列数分别是 (r_i, r_{k+1}) 和 (r_{k+1}, r_{j+1}) , 所以计算 $A[i..k] \times A[(k+1)..j]$ 需要的数乘次数为 $r_i r_{k+1} r_{j+1}$ 。根据最优化原理, 可得到如下递归式。

$$\begin{cases} c(i, i) = 0 & 0 \leq i < n \\ c(i, j) = \min_{i \leq k < j} \{ c(i, k) + c(k+1, j) + r_i r_{k+1} r_{j+1} \} & 0 \leq i < j < n \\ \text{kay}(i, j) = \text{取得上述最小值的 } k \end{cases}$$

2. 计算最优值

// 用最优化法计算A[i..j]的消耗

// 一维数组r为各矩阵的行数和列数, 二维数组kay用于构造最优次序

function C(i, j)

{ if(j <= i) return 0; // 只有一个矩阵

// 有多个矩阵, 从k = i开始计算最小项, 并更新kay

u = C(i, i) + C(i + 1, j) + r[i] * r[i + 1] * r[j + 1];

kay[i, j] = i;

for(k = i + 1 to j - 1)

{ t = C(i, k) + C(k + 1, j) + r[i] * r[k + 1] * r[j + 1];

if(t < u) u = t, kay[i, j] = k; // 找到更小的项

}

return u;

}

$$\begin{cases} c(i, i) = 0 & 0 \leq i < n \\ c(i, j) = \min_{i \leq k < j} \{ c(i, k) + c(k+1, j) + r_i r_{k+1} r_{j+1} \} & 0 \leq i < j < n \\ \text{kay}(i, j) = \text{取得上述最小值的 } k \end{cases}$$

3. 复杂性分析

如果用 $T(n)$ 表示该算法的计算 $A[0..(n-1)]$ 的时间, 则

$$T(n) = \begin{cases} C_1 & n = 2 \\ C_2 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + C_3) & n > 2 \end{cases}$$

当 $n > 2$ 时, 有 $T(n) \geq 2T(n-1) \geq \dots \geq 2^{n-1}T(2) = C \times 2^n$, 从而, $T(n) = \Omega(2^n)$ 。

7.2.2 动态规划方法

实际上,在用递归算法自顶向下求解具有最优子结构的问题时,每次产生的子问题并不总是新的,有些问题反复计算多次。动态规划方法正是利用了这种子问题的重叠性质,对每一个问题只解一次,将其解保存在一个表格中,当需要再次求解时,只需用常数时间简单查看一下结果。

通常,不同的子问题个数随输入问题的大小呈多项式增长。因此,用动态规划方法通常只需要多项式时间,从而获得较高的效率。所以,可用动态规划方法求解的问题应具备的另一个基本要素是子问题的重叠性。

例如,在矩阵连乘积问题中,若用 $c(i, j)$ 表示由第 i 个矩阵到第 j 个矩阵的连乘积所用的最少数乘次数,则计算 $c(0, n-1)$ 的子问题只有 $\Theta(n^2)$ 个。这是因为,对于 $0 \leq i < j < n$, 不同的有序对 (i, j) 对应于不同的子问题,而不同有序对个数为 $n(n-1)/2 = \Theta(n^2)$ 。用动态规划方法解此问题时,在计算过程中,保存已解决的子问题答案,而在需要用到时只是简单检查一下,保证每个子问题只计算一次,从而避免了大量的重复计算。

动态规划可以使用自顶向下或自底向上的方法。

自顶向下方法也称备忘录方法，通常需要使用递归，在递归过程中检查子问题的重叠性。

自底向上方法通常称为动态规划方法，通常使用迭代方法实现，在迭代过程中避免子问题重复计算。

一般来说，当所有子问题都至少要解一次时，用动态规划方法比用备忘录方法要好，因为动态规划方法没有任何多余的计算，通常可以利用规则表格来减少计算时间和空间需求。当有一部分子问题不必求解时，则用备忘录方法比较有利，因为备忘录方法使用递归结构，对于不要求解的子问题，没有必要再进行下一层递归，从而使得该方法只解那些确实要求解的子问题。

7.2.3 矩阵连乘问题的备忘录方法

1. 算法描述

用递归方法求 c 的值，在递归过程中检查已经计算过的 $c(i, j)$ 。

// 用 $c[i][j]$ 记录计算 $A[i..j]$ 的消耗, 初始值为0
// 一维数组 r 保存各矩阵的行数和列数, 二维数组 kay 用于构造最优次序

```
function C(i, j)
{
    if( $c[i][j] > 0$ ) return  $c[i][j]$ ; //  $c[i][j]$  已经确定
    if( $j \leq i$ ) return 0; // 只有一个矩阵
    // 有多个矩阵, 从  $k = i$  开始计算最小项, 并更新  $kay$ 
     $c[i, j] = C(i, i) + C(i + 1, j) + r[i] * r[i + 1] * r[j + 1]$ ;
     $kay[i, j] = i$ ;
    for( $k = i + 1$  to  $j - 1$ )
    {
         $t = C(i, k) + C(k + 1, j) + r[i] * r[k + 1] * r[j + 1]$ ;
        if( $t < c[i, j]$ )  $c[i, j] = t$ ,  $kay[i, j] = k$ ; // 找到更小的项
    }
    return  $c[i, j]$ ;
};
```

$$\left\{ \begin{array}{l} c(i, i) = 0 \\ c(i, j) = \min_{i \leq k < j} \{ c(i, k) + c(k + 1, j) + r_i r_{k+1} r_{j+1} \} \\ kay(i, j) = \text{取得上述最小值的 } k \end{array} \right. \quad \begin{array}{l} 0 \leq i < n \\ 0 \leq i < j < n \end{array}$$

2. 复杂性分析

在该算法中，每个 c 和 kay 只需计算一次，每次计算 c 和 kay 的主要计算量取决于程序中的for循环，for循环的计算量为 $O(n)$ ，共有 $O(n^2)$ 个 c 和 kay ，所以，该算法的计算时间为 $O(n^3)$ 。

```
for(k = i + 1 to j - 1)
{
    t = C(i, k) + C(k + 1, j) + r[i] * r[k + 1] * r[j + 1];
    if(t < c[i, j]) c[i, j] = t, kay[i, j] = k;
}
```

7.2.4 矩阵连乘问题的动态规划算法

1. 算法描述

用迭代方法求解 c 的递归式，按照“ $i = n - 2, n - 3, \dots, 0$ ， $j = i + 1, i + 2, \dots, n - 1$ ”的顺序计算 $c(i, j)$ ，每个 c 和 kay 只计算一次。

```
function C(r, n) // 动态规划方法
{
    // 用c[i, j]记录计算A[i..j]的消耗, 初始值为0
    // 一维数组r保存各矩阵的行数和列数, 二维数组kay用于构造最优次序
    for(i = n - 2 to 0) // 计算c[i, j]和kay[i, j], 其中j>i
    {
        for(j = i + 1 to n - 1)
        {
            c[i, j] = c[i, i] + c[i + 1, j] + r[i] * r[i + 1] * r[j + 1];
            kay[i, j] = i; // 从k = i开始计算最小项, 并更新kay
            for(k = i + 1 to j - 1)
            {
                t = c[i, k] + c[k + 1, j] + r[i] * r[k + 1] * r[j + 1];
                if(t < c[i, j]) c[i, j] = t, kay[i, j] = k; // 找到更小的项
            }
        }
    }
}
```

$$\begin{cases} c(i, i) = 0 & 0 \leq i < n \\ c(i, j) = \min_{i \leq k < j} \{ c(i, k) + c(k + 1, j) + r_i r_{k+1} r_{j+1} \} & 0 \leq i < j < n \\ kay(i, j) = \text{取得上述最小值的 } k \end{cases}$$

2. 复杂性分析

该算法的主要计算量取决于程序中对 i, j, k 的三重循环，循环体内的计算量为 $O(1)$ ，三重循环的总次数是 $O(n^3)$ ，所以，算法的计算时间为 $O(n^3)$ 。

```
for(i = n - 2 to 0)
{
  for(j = i + 1 to n - 1)
  {
    c[i, j] = c[i, i] + c[i + 1, j] + r[i] * r[i + 1] * r[j + 1];
    kay[i, j] = i;
    for(k = i + 1 to j - 1)
    {
      t = c[i, k] + c[k + 1, j] + r[i] * r[k + 1] * r[j + 1];
      if(t < c[i, j]) c[i, j] = t, kay[i, j] = k;
    }
  }
}
```

7.2.5 矩阵连乘问题的最优解

注意，上述三个算法只是明确给出了矩阵最优连乘次序所用的数乘次数，并未明确给出最优连乘次序，即完全加括号方法。但是以 $kay(i, j)$ 为元素的2维数组却给出了足够的信息。事实上， $kay(i, j) = k$ 说明计算连乘积 $A[i..j]$ 的最佳方式应该在矩阵 A_k 和 A_{k+1} 之间断开，即最优加括号方式为 $(A[i..k])(A[(k+1)..j])$ 。下列算法按照上述算法计算出的断点信息 kay 指示的加括号方式输出计算 $A[i..j]$ 的最优次序。

```
function Trace(i, j) // 通过kay[i, j]得到最优方案
{
    if(j <= i) return; // 只有一个矩阵
    k = kay[i, j];
    Trace(i, k), Trace(k + 1, j);
    cout << "M(" << i << ", " << k << ") * ";
    cout << "M(" << k + 1 << ", " << j << ") = ";
    cout << "M(" << i << ", " << j << ")" << endl;
}
```

7.2.6 动态规划方法的基本步骤

通常分为4个阶段。这4个阶段是一个整体，必要时才分步完成（如矩阵连乘问题的动态规划算法），很多时候是统一完成的（如多段图问题的动态规划算法）。

（1）分析最优解的结构。确定要解决的问题具有最优子结构性质，这是使用动态规划方法的基础。

（2）建立递归关系。递归定义最优值，并建立该问题与其子问题最优值之间的递归关系。例如在矩阵连乘问题中，递归关系为

$$\left\{ \begin{array}{l} c(i, i) = 0 \\ c(i, j) = \min_{i \leq k < j} \{ c(i, k) + c(k+1, j) + r_i r_{k+1} r_{j+1} \} \\ \text{kay}(i, j) = \text{取得上述最小值的 } k \end{array} \right. \quad \begin{array}{l} 0 \leq i < n \\ 0 \leq i < j < n \end{array}$$

（3）计算最优值。依据递归关系式用自底向上或自顶向下的方式进行计算，在计算过程中保存已经解决的子问题答案。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，最终获得多项式时间的算法。

（4）构造最优解。依据求最优值时记录的信息，构造出最优解。

7.2.7 矩阵连乘问题动态规划算法的C++描述*

1. 直接递归

```
#include "algorithm.h"

auto MatrixChain1(int r[], int n) // 递归方法
{ Matrix<int> kay(n, n); // 用于构造最优次序

function<int(int, int)> C = [&](int i, int j)
{ if(j <= i) return 0; // 只有一个矩阵
  // 有多个矩阵, 从k = i开始计算最小项, 并更新kay
  int u = C(i, i) + C(i + 1, j) + r[i] * r[i + 1] * r[j + 1];
  kay[i][j] = i;
  for(int k = i + 1; k < j; ++k)
  { int t = C(i, k) + C(k + 1, j) + r[i] * r[k + 1] * r[j + 1];
    if(t < u) u = t, kay[i][j] = k; // 找到更小的项
  }
  return u;
};

int i = 0, j = n - 1;
return pair{C(i, j), kay};
}
```

2. 备忘录方法

```
auto MatrixChain2(int r[], int n) // 备忘录方法
{
    Matrix<int> c(n, n, 0); // c[i][j]初始为0
    Matrix<int> kay(n, n); // 用于构造最优次序

    function<int(int, int)> C = [&](int i, int j)
    {
        if(c[i][j] > 0) return c[i][j]; // c[i][j]已经确定
        if(j <= i) return 0; // 只有一个矩阵
        // 有多个矩阵, 从k = i开始计算最小项, 并更新kay
        c[i][j] = C(i, i) + C(i + 1, j) + r[i] * r[i + 1] * r[j + 1];
        kay[i][j] = i;
        for(int k = i + 1; k < j; ++k)
        {
            int t = C(i, k) + C(k + 1, j) + r[i] * r[k + 1] * r[j + 1];
            if(t < c[i][j]) c[i][j] = t, kay[i][j] = k; // 找到更小的项
        }
        return c[i][j];
    };

    int i = 0, j = n - 1;
    return pair{C(i, j), kay};
}
```

3. 动态规划方法

```
auto MatrixChain3(int r[], int n) // 动态规划方法
{
    Matrix<int> c(n, n, 0); // c[i][j]初始为0
    Matrix<int> kay(n, n); // 用于构造最优次序
    for(int i = n - 2; i >= 0; --i) // 计算c[i][j]和kay[i][j], 其中j>i
    {
        for(int j = i + 1; j < n; ++j)
        {
            c[i][j] = c[i][i] + c[i + 1][j] + r[i] * r[i + 1] * r[j + 1];
            kay[i][j] = i; // 从k=i开始计算最小项, 并更新kay
            for(int k = i + 1; k < j; ++k)
            {
                int t = c[i][k] + c[k + 1][j] + r[i] * r[k + 1] * r[j + 1];
                if(t < c[i][j]) c[i][j] = t, kay[i][j] = k; // 找到更小的项
            }
        }
    }

    int i = 0, j = n - 1;
    return pair{c[i][j], kay};
}
```


4. 最优解

```
void Traceback(Matrix<int> &kay) // 通过kay[i][j]得到最优方案
{
    int n = kay.Rows(); // 矩阵个数

    function<void(int, int)> Trace = [&](int i, int j)
    {
        if(j <= i) return; // 只有一个矩阵
        int k = kay[i][j];
        Trace(i, k), Trace(k + 1, j);
        cout << "M(" << i << ", " << k << ")*";
        cout << "M(" << k + 1 << ", " << j << ")=";
        cout << "M(" << i << ", " << j << ")" << endl;
    };

    int i = 0, j = n - 1;
    Trace(i, j);
}
```

5. 测试*

(1) 测试程序。使用5个矩阵进行测试，行列数分别是 10×5 、 5×1 、 1×10 、 10×2 和 2×10 ，测试程序和测试结果如下。

```
int main()
{   int n = 5; // 矩阵个数
    int r[n + 1] = {10, 5, 1, 10, 2, 10};
    auto [c, kay] = MatrixChain1(r, n);
    cout << "递归方法:\n", Traceback(kay);
    tie(c, kay) = MatrixChain2(r, n);
    cout << "备忘录方法:\n", Traceback(kay);
    tie(c, kay) = MatrixChain3(r, n);
    cout << "动态规划方法:\n", Traceback(kay);
}
```

递归方法:

$M(0,0) * M(1,1) = M(0,1)$

$M(2,2) * M(3,3) = M(2,3)$

$M(2,3) * M(4,4) = M(2,4)$

$M(0,1) * M(2,4) = M(0,4)$

备忘录方法:

$M(0,0) * M(1,1) = M(0,1)$

$M(2,2) * M(3,3) = M(2,3)$

$M(2,3) * M(4,4) = M(2,4)$

$M(0,1) * M(2,4) = M(0,4)$

动态规划方法:

$M(0,0) * M(1,1) = M(0,1)$

$M(2,2) * M(3,3) = M(2,3)$

$M(2,3) * M(4,4) = M(2,4)$

$M(0,1) * M(2,4) = M(0,4)$

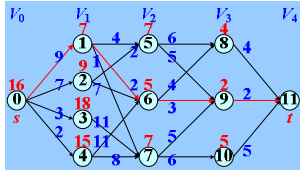
(2) 测试结果分析。三种方法结果相同，测试结果分析如表7-1所示。

表7-1 矩阵连乘的测试结果分析

方法	递归			备忘录			动态规划
Cost matrix				0	50	150	90 190
				0	0	50	30 90
				0	0	0	20 40
				0	0	0	0 200
				0	0	0	0 0
kay matrix		*	0	1	1	1	
		*	*	1	1	1	
		*	*	*	2	3	
		*	*	*	*	3	
		*	*	*	*	*	
Answer	$M(0, 0) * M(1, 1) = M(0, 1)$			$kay(0, 1)=0$			
	$M(2, 2) * M(3, 3) = M(2, 3)$			$kay(2, 3)=2$			
	$M(2, 3) * M(4, 4) = M(2, 4)$			$kay(2, 4)=3$			
	$M(0, 1) * M(2, 4) = M(0, 4)$			$kay(0, 4)=1$			

7.3 多段图问题*

7.3.1 递归关系



7.3.2 最小成本

1. 算法描述

如果使用邻接矩阵表示多段图，不明确指定每一段包含哪些顶点，则可以将

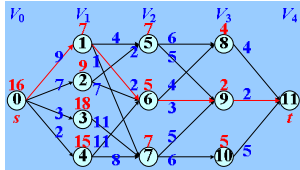
$$C(j) = \min\{c(j, r) + C(r) \mid r \in \text{NextSect}(j)\}$$

改为

$$C(j) = \min\{c(j, r) + C(r) \mid j < r < n\}$$

为了方便获得最小成本路径，用 $\text{Next}(j)$ 记录顶点 j 到汇点的最短路径中 j 的后继顶点。由此，可得下列计算最小成本的算法。

```
#include "algorithm.h"
auto MultiGraph(Matrix<double> &G)
{ // 邻接矩阵G, 顶点按段序统一编号
  int n = G.Rows(), t = n - 1; // 顶点数, 汇点
  vector<double> C(n, 0); // C[j]是j到汇点的最小成本, 初始为0
  vector<int> Next(n); // Next[j]是j到汇点的最短路径中j的后继顶点
```



```
for(int j = t - 1; j >= 0; --j) // 从最后一个顶点开始往前计算
{ // 从j的后一个阶段中选择顶点r, 使G[j][r]+C[r]最小
    int r = j + 1;
    for(int i = r; i < n; ++i)
        if(G[j][i] + C[i] < G[j][r] + C[r])
            r = i;
    C[j] = G[j][r] + C[r], Next[j] = r;
}
return pair {C[0], Next};
}
```

2. 复杂性分析

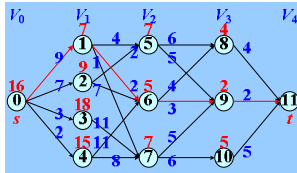
r 的确定可以在 $O(n)$ 时间内完成（内层for循环）。因此，完成向前处理的for循环需耗时 $O(n^2)$ （因为对任何顶点 j ，均需对 $r \in [(j+1), (n-1)]$ 进行处理）。

7.3.3 最短路径

使用记录下来的决策生成最短路径。Next(j) 记录了顶点 j 到汇点的最短路径中 j 的后继顶点，由此可以很容易地求出从源点到汇点的最小成本路径。方法如下。

```
auto MultiGraph(vector<int> &Next, int m)
{ // 记录下来的决策, 段数
    vector<int> X(m); // X[i]是最短路径中第i阶段选取的顶点
    X[0] = 0;
    for(int i = 0; i < m - 1; ++i)
        X[i + 1] = Next[X[i]];
    return X;
}
```

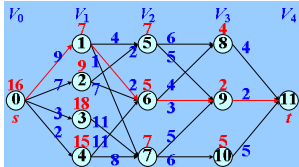
生成路径的for循环需耗时 $O(m)$ 。因为 $m < n$ ，所以也可以说生成路径的for循环需耗时 $O(n)$ 。



7.3.4 测试*

1. 测试程序

使用图7-1所示的图。



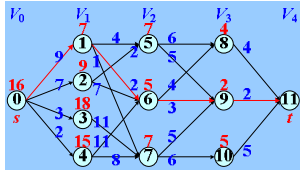
```
int main()
{   int n = 12, m = 5;
    Matrix<double> G(n, n, inf); // 空图(权值无穷大)
    G[0][1] = 9, G[0][2] = 7, G[0][3] = 3, G[0][4] = 2; // V0-V1
    G[1][5] = 4, G[1][6] = 2, G[1][7] = 1, G[2][5] = 2, G[2][6] = 7,
        G[3][7] = 11, G[4][6] = 11, G[4][7] = 8; // V1-V2
    G[5][8] = 6, G[5][9] = 5, G[6][8] = 4, G[6][9] = 3,
        G[7][9] = 5, G[7][10] = 6; // V2-V3
    G[8][11] = 4, G[9][11] = 2, G[10][11] = 5; // V3-V4
    auto [C, Next] = MultiGraph(G);
    cout << C << endl;
    auto X = MultiGraph(Next, m);
    cout << to_string(X, m, "-") << endl;
}
```

16

0-1-6-9-11

2. 计算步骤

最小成本计算步骤如下。



$$C(8) = 4, C(9) = 2, C(10) = 5$$

$$C(5) = \min\{6 + C(8), 5 + C(9)\} = 7$$

$$C(6) = \min\{4 + C(8), 3 + C(9)\} = 5$$

$$C(7) = \min\{5 + C(9), 6 + C(10)\} = 7$$

$$C(1) = \min\{4 + C(5), 2 + C(6), 1 + C(7)\} = 7$$

$$C(2) = \min\{2 + C(5), 7 + C(6)\} = 9$$

$$C(3) = 11 + C(7) = 18$$

$$C(4) = \min\{11 + C(6), 8 + C(7)\} = 15$$

$$C(0) = \min\{9 + C(1), 7 + C(2), 3 + C(3), 2 + C(4)\} = 16$$

决策记录步骤如下。

$$\text{Next}(8) = \text{Next}(9) = \text{Next}(10) = 11$$

$$\text{Next}(5) = \text{Next}(6) = \text{Next}(7) = 9$$

$$\text{Next}(1) = 6, \text{Next}(2) = 5, \text{Next}(3) = 7, \text{Next}(4) = 7$$

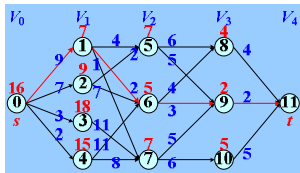
$$\text{Next}(0) = 1$$

设这条最小成本路径是 $s = 0, v_1, v_2, v_3, t = 11$ ，则

$$v_1 = \text{Next}(s) = \text{Next}(0) = 1$$

$$v_2 = \text{Next}(v_1) = \text{Next}(1) = 6$$

$$v_3 = \text{Next}(v_2) = \text{Next}(6) = 9$$



7.4 每对顶点之间的最短路径

7.4.1 问题的描述

已知 $G=(V,E)$ 是一个有 n 个顶点的加权图(有向或无向), 用邻接矩阵表示。求图中每对顶点之间最短路径的长度, 即求矩阵 A , A 中的元素 $A(i,j)$ 表示 G 中的顶点 i 和 j 之间最短路径的长度。

这里首先针对权值非负的情况分析最优子结构性质和递推关系, 然后放宽该条件构造动态规划算法。

7.4.2 最优子结构性质

考察 G 中一条由 i 到 j 的最短路径 ($i \neq j$)。该路径由 i 出发, 通过一些中间顶点 (也可能没有), 在 j 处结束。可以假定这条路径不含圈, 如果有圈, 则可去掉这个圈, 这是因为去掉圈后不会增加路径长度。如果 k 是这条路径上的一个中间顶点, 那么由 i 到 k 和由 k 到 j 的两条子路径分别是由 i 到 k 和由 k 到 j 的最短路径。否则, 这条路径就不是由 i 到 j 的最短路径。

7.4.3 递推关系

由上面的分析可知,图中每对顶点之间的最短路径是一个递推的过程。分析递推过程的关键有以下几点。

① 将顶点编号(从0开始)。若 k 是 i 到 j 的最短路径上编号最高的中间顶点,则 i 到 k 和 j 到 k 的最短路径上不会有比编号 $k-1$ 还大的中间顶点。

② 用 $A_{ij}^{[k]}$ 表示从 i 到 j 的中间顶点最大编号不超过 k 的最短路径 α 的长度。如果 α 不经过顶点 k ,那么 α 的中间顶点最大编号不超过 $k-1$,所以 $A_{ij}^{[k]} = A_{ij}^{[k-1]}$;如果 α 经过顶点 k ,那么 $A_{ij}^{[k]} = A_{ik}^{[k-1]} + A_{kj}^{[k-1]}$ 。

③ 综合上述两种情况,得到递推公式 $A_{ij}^{[k]} = \min(A_{ij}^{[k-1]}, A_{ik}^{[k-1]} + A_{kj}^{[k-1]})$, $0 \leq k < n$ 。它表示从 i 到 j 的中间顶点最大编号不超过 k 的最短路径的长度,需考察不经过 k 和经过 k 两种情况。 $A_{ij}^{[k-1]}$ 表示不经过 k , $A_{ik}^{[k-1]} + A_{kj}^{[k-1]}$ 表示经过 k ,选取较小的。

综上所述, 要求任意顶点间的最短路径, 可以先建立一个初始矩阵 $A^{[-1]}$, 它等于成本矩阵COST。然后求中间顶点最大编号为0的任意顶点间最短路径矩阵 $A^{[0]}$, 再求中间顶点最大编号为1的最短路径矩阵 $A^{[1]}$, 以此类推, 直到求出 $A^{[n-1]}$ 。

根据上述分析, 可以构造出下列求任意顶点间最短距离的Floyd算法。其中 $A(i, j)$ 存储 i 到 j 的最短距离, $Next(i, j)$ 存储 i 到 j 的最短路径中 i 的后继顶点。

7.4.4 任意顶点间最短距离（最优值）

1. 计算结果

```
#include "algorithm.h"
auto Floyd(Matrix<double> &G)
{   int n = G.Rows(); // n是顶点数
    auto A = G; // A[-1]=G, A(i, j)是i到j的最短距离
    Matrix<int> Next(n, n); // Next(i, j)是i到j的最短路径中i的后继顶点
    for(int i = 0; i < n; ++i) // 建立初始Next
        for(int j = 0; j < n; ++j)
            Next[i][j] = j;

    for(int k = 0; k < n; ++k) // A[0]到A[n-1]的循环控制
        for(int i = 0; i < n; ++i)
            for(int j = 0; j < n; ++j)
                {   auto t = A[i][k] + A[k][j];
                    if(t < A[i][j]) A[i][j] = t, Next[i][j] = Next[i][k];
                }
    return pair{A, Next};
}
```

2. 检查结果的正确性

如果图中有负权值的边，则需要检查结果的正确性。要判断结果是否正确，只需要判断“对任何 $0 \leq i, j, k < n$ 都有 $A_{ij}^{[n-1]} \leq A_{ik}^{[n-1]} + A_{kj}^{[n-1]}$ ”是否成立^①。

```
bool ValidFloyd(Matrix<double> &A) // 判断结果的正确性
{
    int n = A.Rows();
    for(int i = 0; i < n; ++i)
        for(int j = 0; j < n; ++j)
            for(int k = 0; k < n; ++k)
                if(A[i][j] > A[i][k] + A[k][j]) return false;
    return true;
}
```

^① Wang Zhixi, Wang Yana, Hu Binliang, Liu Yu. An Algorithm of Plus - Closures of Loop - Nonnegative Matrices over Idempotent Semirings and its Applications[J]. Applied Mechanics and Materials, 2013, 347-350: 2208 - 2213

3. 复杂性分析

过程Floyd计算 $A^{[n-1]}$ 所需要的时间是 $\Theta(n^3)$ 。这是因为最内层循环体迭代了 n^3 次，而且整个循环的耗时与矩阵 A 中的数据无关，耗时 $\Theta(n^3)$ 。过程ValidFloyd 耗时 $O(n^3)$ 。

7.4.5 任意顶点间最短路径（最优解）

```
auto Path(int i, int j, Matrix<int> &Next) // 最短路径(i,j)的所有顶点
{
    vector<int> X; // 记录最短路径中的顶点
    for(; i != j; i = Next[i][j])
        X.push_back(i);
    X.push_back(j);
    return X;
}
```

7.4.6 一个实例*

1. 实例分析

对于如图7-2所示的赋权有向图，Floyd算法得到的Dist、Next和最短路径如表7-2所示。

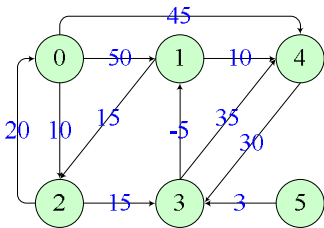
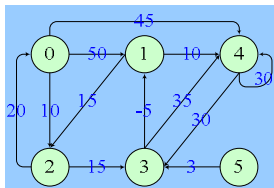


图7-2 一个赋权有向图

表7-2 Floyd算法实例分析

Dist						Next					
0	20	10	25	30	inf	2	2	2	2	2	5
35	0	15	30	10	inf	2	2	2	2	4	5
20	10	0	15	20	inf	0	3	3	3	3	5
30	-5	10	0	5	inf	1	1	1	1	1	5
60	25	40	30	0	inf	3	3	3	3	4	5
33	-2	13	3	8	0	3	3	3	3	3	5

AllPath					
0	0-2-3-1	0-2	0-2-3	0-2-3-1-4	0
1-2-0	1	1-2	1-2-3	1-4	1
2-0	2-3-1	2	2-3	2-3-1-4	2
3-1-2-0	3-1	3-1-2	3	3-1-4	3
4-3-1-2-0	4-3-1	4-3-1-2	4-3	4	4
5-3-1-2-0	5-3-1	5-3-1-2	5-3	5-3-1-4	5

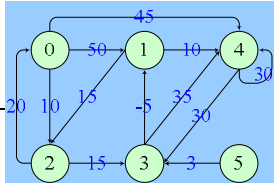


2. 显示所有最短路径

```
void Path(Matrix<int> &Next) // 显示所有最短路径
{
    int n = Next.Rows();
    for(int i = 0; i < n; ++i)
    {
        for(int j = 0; j < n; ++j)
        {
            auto X = Path(i, j, Next);
            cout << to_string(begin(X), end(X), "-") << " ";
        }
        cout << endl;
    }
}
```

3. 测试程序

使用如图7-2所示的赋权有向图。



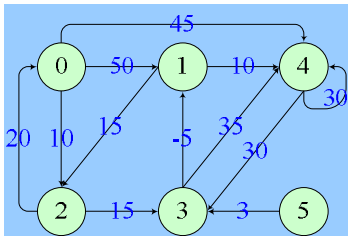
```
int main()
{ Matrix<double> G =
  { {0, 50, 10, inf, 45, inf},
    {inf, 0, 15, inf, 10, inf},
    {20, inf, 0, 15, inf, inf},
    {inf, -5, inf, 0, 35, inf},
    {inf, inf, inf, 30, 0, inf},
    {inf, inf, inf, 3, inf, 0}
  };

  auto [A, Next] = Floyd(G);
  if(!ValidFloyd(A)) return -1;
  Path(Next);
}
```

4. 测试结果

如表7-2所示。

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



7.5 最长公共子序列*

7.5.1 问题

1. 相关定义

给定序列 $X = x_1x_2\dots x_m$, $Z = z_1z_2\dots z_k$ 。若存在一个严格递增的下标序列 (i_1, i_2, \dots, i_k) , 使得对所有 $j = 1, 2, \dots, k$, 都有 $z_j = x_{i_j}$, 则称 Z 是 X 的子序列。

例如, 序列 $Z = BCDB$ 是序列 $X = ABCBDAB$ 的子序列, 相应的递增下标序列为 $(2, 3, 5, 7)$ 。给定2个序列 X 和 Y , 若序列 Z 既是 X 的子序列又是 Y 的子序列, 则称 Z 是序列 X 和 Y 的公共子序列。

2. 问题描述

给定2个序列 $X = x_1x_2\dots x_m$ 和 $Y = y_1y_2\dots y_n$, 找出 X 和 Y 的最长公共子序列。

7.5.2 最优子结构性质

为了方便, 将序列 $X = x_1x_2 \dots x_m$ 的前缀 $x_1x_2 \dots x_i$ 记作 X_i ($0 \leq i \leq m$)。设序列 $X = x_1x_2 \dots x_m$ 和 $Y = y_1y_2 \dots y_n$ 的最长公共子序列为 $Z = z_1z_2 \dots z_k$, 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。

(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。

(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。

由此可见, 2个序列的最长公共子序列包含了这2个序列前缀的最长公共子序列。因此, 最长公共子序列问题具有最优子结构性质。

7.5.3 递归关系

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 $X_i = x_1x_2\dots x_i$ 和 $Y_j = y_1y_2\dots y_j$ 的最长公共子序列的长度。当 $i=0$ 或 $j=0$ 时，空序列是 X_i 和 Y_j 的最长公共子序列，所以 $c[i][j]=0$ 。由最优子结构性质可知

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

X\Y		1	2	1
	0	0	0	0
1	0	1	1	1
2	0	1	2	2
2	0	1	2	2
1	0	1	2	3

7.5.4 计算最长公共子序列的长度

```
#include "algorithm.h"
template<class Seq>
auto LCSSize(const Seq &X, const Seq &Y)
{   int m = X.size(), n = Y.size();
    Matrix<int> C(m + 1, n + 1); // (m+1) * (n+1) 矩阵C
    for(int i = 0; i <= m; ++i)
        for(int j = 0; j <= n; ++j)
            if(i <= 0 || j <= 0) C[i][j] = 0;
            else if(X[i - 1] == Y[j - 1]) C[i][j] = C[i - 1][j - 1] + 1;
            else C[i][j] = max(C[i - 1][j], C[i][j - 1]); // 上或左
    return C; // C[m][n] 为最长公共子序列的长度
}
```

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

7.5.5 构造最长公共子序列

```
template<class Seq>
auto LCS(const Seq &X, const Seq &Y)
{   Matrix<int> C = LCSSize(X, Y);
    int i = X.size(), j = Y.size(), k = C[i][j];
    Seq Z;
    while(k > 0) // 逆向记录结果序列
        if(X[i - 1] == Y[j - 1])
            Z.push_back(X[i - 1]), --i, --j, --k;
        else if(C[i][j] == C[i - 1][j]) --i; // 上
        else --j; // 左
    reverse(begin(Z), end(Z)); // 反转结果序列
    return Z;
}
```

7.5.6 生成过程演示

设 $X = \{1, 2, 2, 1\}$, $Y = \{1, 2, 1, 2\}$, 则LCS(X, Y)的生成过程如图7-3所示。

X\Y		1	2	1	2
	0	0	0	0	0
1	0	1	1	1	1
2	0	1	2	2	2
2	0	1	2	2	3
1	0	1	2	3	3

图7-3 LCS的生成过程

X\Y		1	2	1
	0	0	0	0
1	0	1	1	1
2	0	1	2	2
2	0	1	2	2
1	0	1	2	3

7.5.7 测试程序*

```
int main()
{
    string X = "ABCBADAB", Y = "BDCABA";
    cout << "LCS(" << X << ", " << Y << ") = " << LCS(X, Y) << endl;
    vector<int> U = {1, 2, 2, 1}, V = {1, 2, 1, 2};
    cout << "LCS(" << U << ", " << V << ") = " << LCS(U, V) << endl;
}
```

LCS(ABCBADAB, BDCABA) = BCBA

LCS(1 2 2 1, 1 2 1 2) = 1 2 2

7.6 练习题

- 1、使用一种程序设计语言或伪代码描述计算矩阵连乘最优次序的递归算法，并分析其时间复杂性。
- 2、使用一种程序设计语言或伪代码描述求矩阵连乘最优次序的备忘录方法（使用递归方法），并分析其时间复杂性。
- 3、使用一种程序设计语言或伪代码描述求矩阵连乘最优次序的动态规划算法（使用循环结构），并分析其时间复杂性。
- 4、使用一种程序设计语言或伪代码描述多段图的动态规划算法，并分析其时间复杂性。
- 5、使用一种程序设计语言或伪代码描述求任意顶点间最短路径的动态规划算法，并分析其时间复杂性。
- 6、请使用非递归方法输出矩阵连乘积最优次序，并分析其时间复杂性。
- 7、改写Floyd算法的程序，从一个有向图的邻接矩阵开始计算其自反传递闭包矩阵RTC。若从顶点 i 到顶点 j 有通路，则 $RTC[i][j]=1$ ，否则 $RTC[i][j]=0$ 。要求代码的复杂性为 $O(n^3)$ ，其中 n 为图的顶点数。

8、使用一种程序设计语言或伪代码描述求任意两个有限序列的最长公共子序列的动态规划算法，并分析其时间复杂性。