Analysis and Design of Algorithms

Chapter 9: Dynamic Programming

School of Software Engineering © Ye LUO

Content

1. **Introduction of Dynamic Programming**
2. **Applications of DP**
   - ① *Computing binomial coefficients*
   - ② *the longest common subsequence (LCS)*
   - ③ *Dynamic Matrix Multiplication*
   - ④ *0-1 Knapsack Problem*
   - ⑤ *Multistage Decision Processes*
   - ⑥ *Warshall's algorithm for transitive closure*
   - ⑦ *Floyd's algorithms for all-pairs shortest paths*
3. **Examples of Checking the Principle of Optimality**

1. Introduction of DP

- Basic Concepts of DP
- The Principle of Optimality
- Basic Idea of DP
- Basic Solution Steps of DP
- Conditions of Applying DP

1. Introduction of DP-- basic concepts

➢**动态规划(Dynamic Programming,DP)**是运筹学的一个分支。

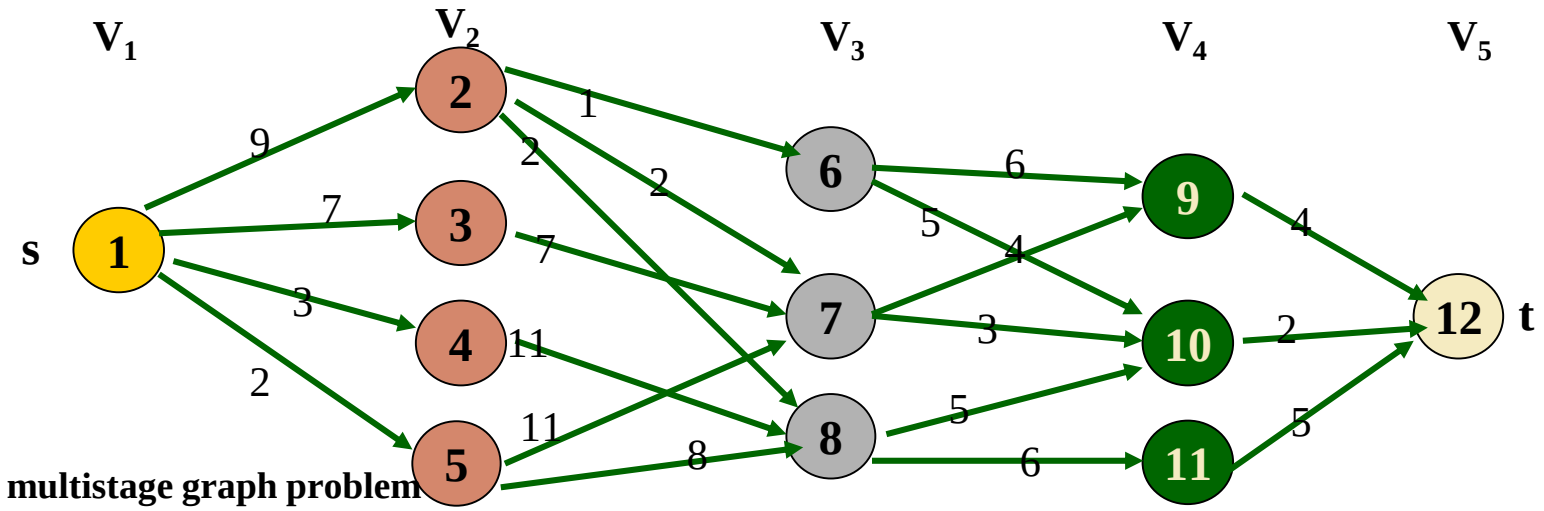
➢20世纪50年代初美国数学家R.E.Bellman等人在研究多阶段决策过程(Multistep Decision Process，MDP)的优化问题时，提出了著名的**最优化原理(Principle of Optimality)**，把多阶段过程转化为一系列单阶段问题，逐个求解，创立了解决这类过程优化问题的新方法—动态规划。

➢**多阶段决策问题(MDP)**：求解的问题可以划分为一系列相互联系的阶段，在每个阶段都需要作出决策，且一个阶段决策的选择会影响下一个阶段的决策，从而影响整个过程的活动路线，求解的目标是选择各个阶段的决策使整个过程达到最优。

1. Introduction of DP-- basic concepts

DP and MDP

- *Dynamic Programming for optimizing Multistage decision processes, [1950s]*



1. Introduction of DP-- basic concepts

- 动态规划主要用于求解以时间划分阶段的动态过程的优化问题，但是一些与时间无关的静态规划(如线性规划、非线性规划)，可以人为地引进时间因素，把它视为多阶段决策过程，也可以用动态规划方法方便地求解。

- 动态规划是考察问题的一种途径，或是求解某类问题的一种方法。

- 动态规划问世以来，在经济管理、生产调度、工程技术和最优控制等方面得到了广泛的应用。例如最短路线、库存管理、资源分配、设备更新、排序、装载等问题，用动态规划方法比用其它方法求解更为方便。

1. Introduction of DP-- basic concepts

➢ **阶段（Stage）：** 把所给的问题的求解过程恰当地划分为若干个相互联系的阶段。

➢**状态 (State)：** 表示每个阶段开始时，问题或系统所处的客观状况。状态既是该阶段的某个起点，又是前一个阶段的某个终点。通常一个阶段有若干个状态。

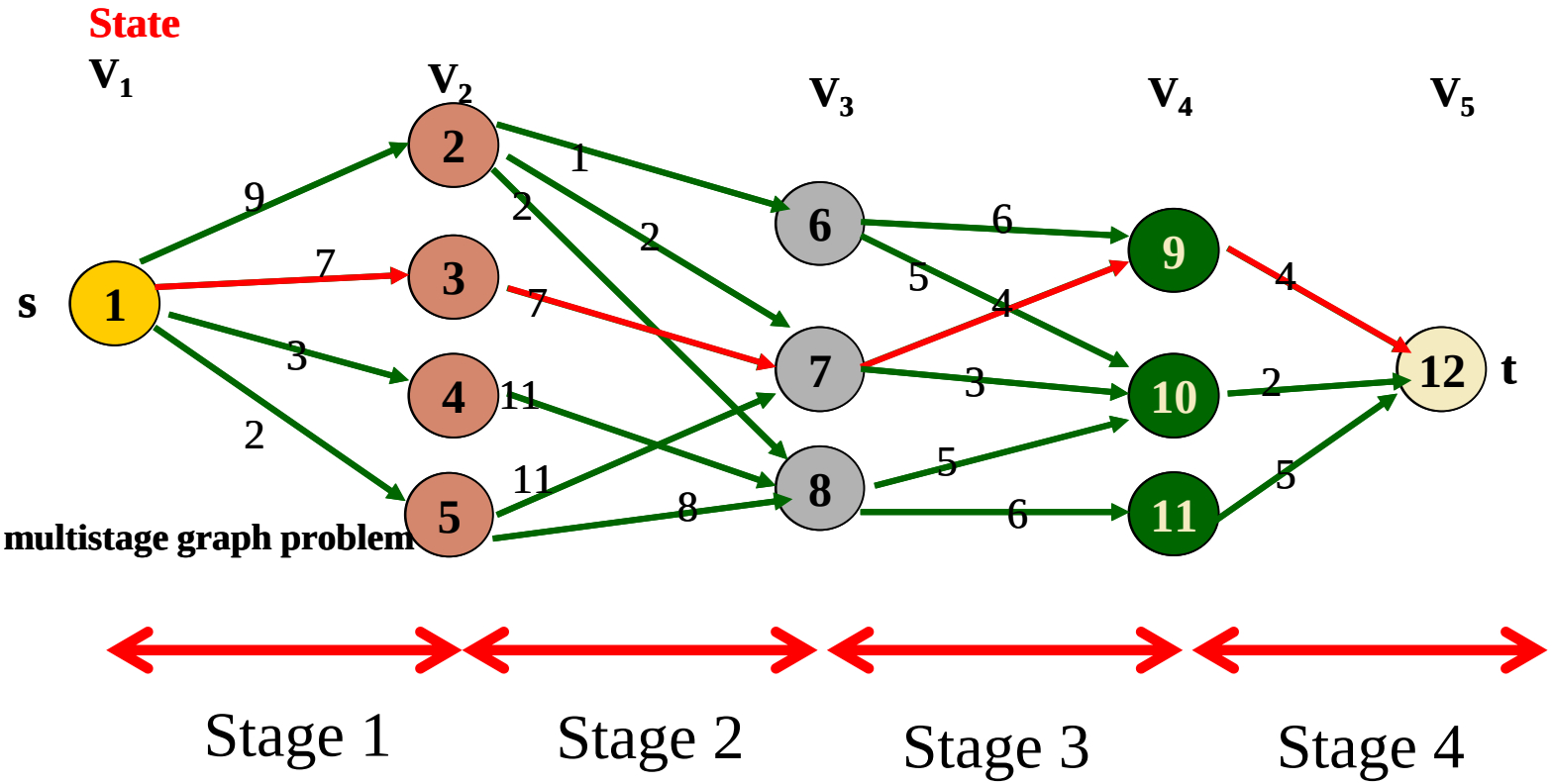
状态的无后效性：如果某阶段状态给定后，则该阶段以后过程的发展不受该阶段以前各阶段状态的影响，也就是说状态具有**马尔科夫性**。

注：适于动态规划法求解的问题具有状态的无后效性。

➢**策略（Policy）：** 各个阶段决策的确定后，就组成了一个决策序列，该序列称之为一个策略。由某个阶段开始到终止阶段的过程称为子过程，其对应的某个策略称为子策略。

1. Introduction of DP-- basic concepts

- ***MDP*** *: Multistage decision processes,* [*1950s*]



1. Introduction of DP-the principle of optimality

➢ **Bellman 的原定义**

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

➢**Bellman 最优性原理**

求解问题的一个最优策略序列时，该最优策略序列的子策略序列总是最优的，则称该问题满足最优性原理。

注：对具有最优性原理性质的问题而言，如果有一决策序列包含有非最优的决策子序列，则该决策序列一定不是最优的。

1. Introduction of DP-- the basic idea

➢动态规划的思想实质：分治和解决冗余。

➢与分治法相同点：

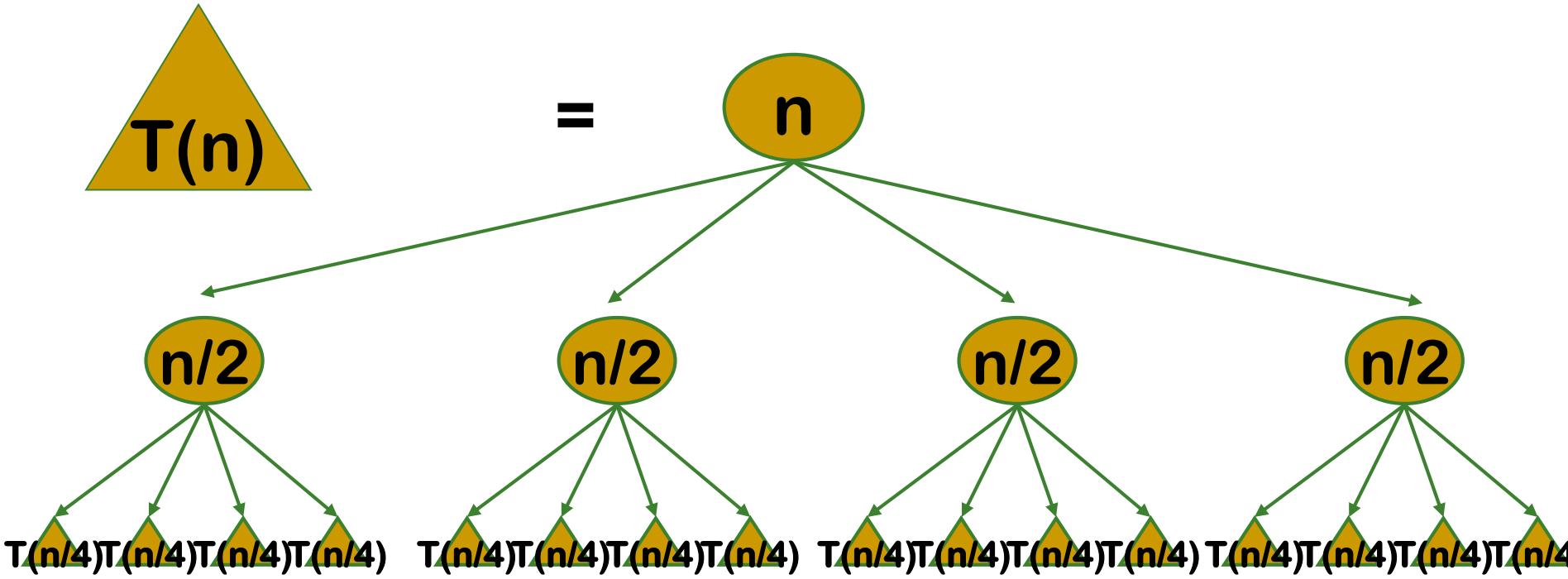
将原问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。

➢与分治法不相同点：

经分解的子问题往往不是互相独立的。若用分治法来解，有些共同部分（子问题或子子问题）被重复计算了很多次。

1. Introduction of DP-- the basic idea

- *Main idea of Divide-and-Conquer*



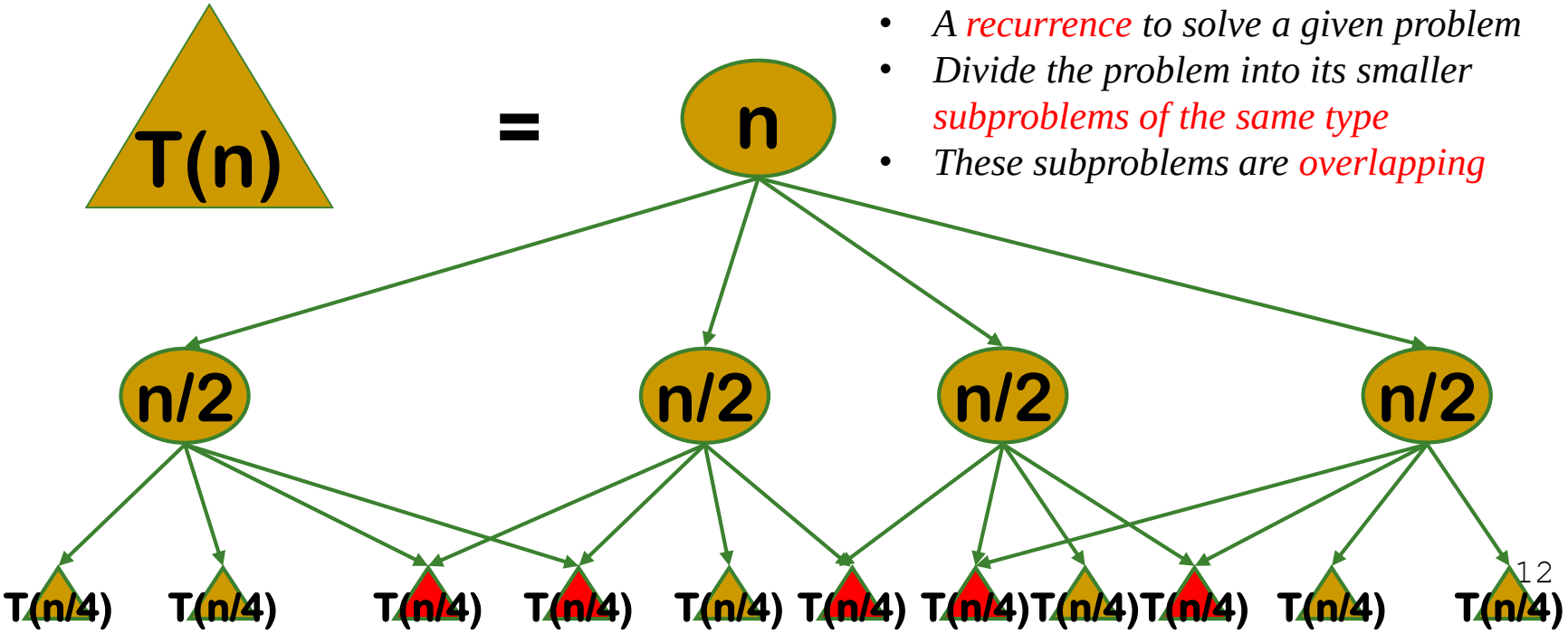
divide-and-conquer

1. Introduction of DP-- the basic idea

- ***Main idea of Dynamic-Programming***

Dynamic Programming

- *A recurrence to solve a given problem*
- *Divide the problem into its smaller subproblems of the same type*
- *These subproblems are overlapping*



1. Introduction of DP-- the basic idea

- 动态规划的思想实质：分治和解决冗余。

- 动态规划法用一个表来记录所有已解的子问题的答案。在需要时再查找，这样就可以避免重复计算、节省时间。

- 具体的动态规划算法多种多样，但它们具有相同的填表方式。

1. Introduction of DP– basic solution steps

- 动态规划的基本解题步骤：

① 找出最优解的性质，并刻画其结构特征；
② 递归的定义最优值（写出动态规划方程）；
③ 以*自底向上或自顶向下*的方式计算出最优值；
④ 根据计算最优值时的记录信息，构造最优解。

注：步骤1-3是动态规划算法的基本步骤。如果只需要求出最优值，步骤4可以省略。

若需要求出问题的一个最优解，则必须执行步骤4，步骤3中记录的信息是构造最优解的基础。

1. Introduction of DP-- *examples of solution steps*

动态规划的两种求解方式：

①**自顶向下的备忘录法** ；

②**自底向上。**

求斐波拉契数列**Fibonacci** 的例子：

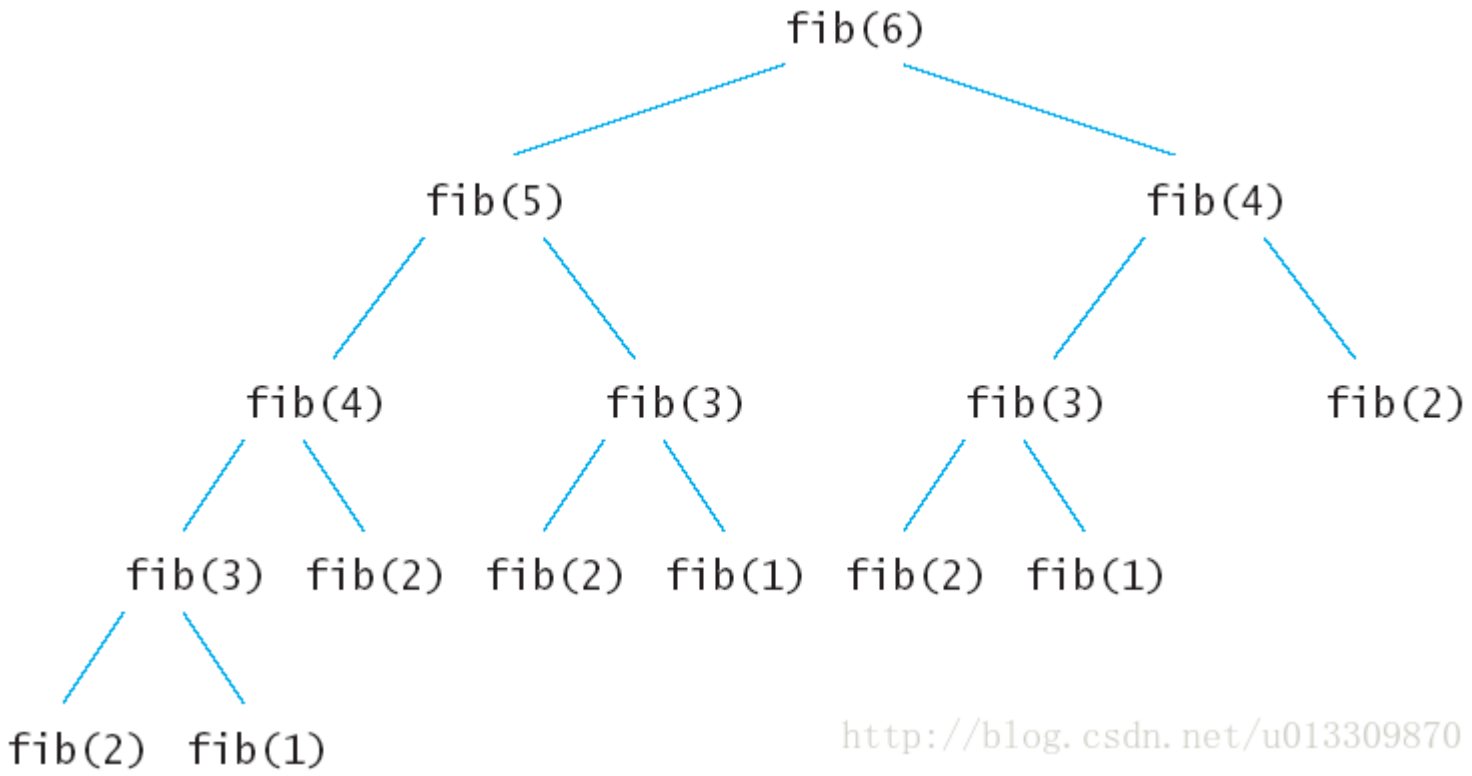
```
1  Fibonacci (n) = 1;    n = 0
2
3  Fibonacci (n) = 1;    n = 1
4
5  Fibonacci (n) = Fibonacci(n-1) + Fibonacci(n-2)
```

1. Introduction of DP-- *examples of solution steps*

求斐波拉契数列Fibonacci数列的递归伪代码：

```
1  public int fib(int n)
2  {
3      if(n<=0)
4          return 0;
5      if(n==1)
6          return 1;
7      return fib( n-1)+fib(n-2);
8  }
9  //输入6
10 //输出: 8
```

1. Introduction of DP-- examples of solution steps



上面的递归树中的每一个子节点都会执行一次，很多重复的节点被执行，fib(2)被重复执行了5次。由于调用每一个函数的时候都要保留上下文，所以空间上开销也不小。这么多的子节点被重复执行，如果在执行的时候把执行过的子节点保存起来，后面要用到的时候直接查表调用的话可以节约大量的时间

1. Introduction of DP-- *examples of solution steps*

```
1  public static int Fibonacci(int n)
2  {
3          if(n<=0)
4              return n;
5          int []Memo=new int[n+1];
6          for(int i=0;i<=n;i++)
7              Memo[i]=-1;
8          return fib(n, Memo);
9      }
10     public static int fib(int n,int []Memo)
11     {
12
13         if(Memo[n]!=-1)
14             return Memo[n];
15     //如果已经求出了fib（n）的值直接返回，否则将求出的值保存在Memo备忘录中。
16         if(n<=2)
17             Memo[n]=1;
18
19         else Memo[n]=fib( n-1,Memo)+fib(n-2,Memo);
20
21         return Memo[n];
22     }
```

自顶向下的备忘录DP算法

1. Introduction of DP-- examples of solution steps

自底向上的DP算法

```
1  public static int fib(int n)
2  {
3          if(n<=0)
4              return n;
5          int []Memo=new int[n+1];
6          Memo[0]=0;
7          Memo[1]=1;
8          for(int i=2;i<=n;i++)
9          {
10             Memo[i]=Memo[i-1]+Memo[i-2];
11         }
12         return Memo[n];
13 }
```

1. Introduction of DP-- *examples of solution steps*

```
1  public static int fib(int n)
2      {
3          if(n<=1)
4              return n;
5
6          int Memo_i_2=0;
7          int Memo_i_1=1;
8          int Memo_i=1;
9          for(int i=2;i<=n;i++)
10         {
11             Memo_i=Memo_i_2+Memo_i_1;
12             Memo_i_2=Memo_i_1;
13             Memo_i_1=Memo_i;
14         }
15         return Memo_i;
16     }
```

自底向上的DP算法进一步压缩空间

一般来说由于备忘录方式的动态规划方法使用了递归，递归的时候会产生额外的开销，**使用自底向上的动态规划方法要比备忘录方法好**。

1. Introduction of DP-- Conditions of DP

动态规划的有效性依赖于问题本身具有的**两个重要的适用性质：最优子结构和重叠子问题。**

① 最优子结构

如果问题的最优解是由其子问题的最优解来构造，则称该问题具有最优子结构性质。

使用动态规划算法时，要用子问题的最优解来构造原问题的最优解，因此必须考查最优解中用到的所有子问题。

② 重叠子问题

使用递归算法的时反复求解相同的子问题，不停的调用函数，而不是生成新的子问题。如果递归算法反复求解相同的子问题，就称具有重叠子问题（overlapping subproblems）的性质。

在动态规划算法中使用数组来保存子问题的解，这样子问题多次求解的时候可以直接查表不用调用函数递归。E.g. 斐波拉契数列中fib(2)的5次调用。

Content

1. Introduction of Dynamic Programming
2. **Applications of DP**
   - ① Computing binomial coefficients
   - ② the longest common subsequence (LCS)
   - ③ Dynamic Matrix Multiplication
   - ④ 0-1 Knapsack Problem
   - ⑤ Multistage Decision Processes
   - ⑥ Warshall's algorithm for transitive closure
   - ⑦ Floyd's algorithms for all-pairs shortest paths
3. Examples of Checking the Principle of Optimality

Computing Binomial Coefficients

Definition

binomial coefficient

- *A binomial coefficient, denoted C(n, k), is the number of combinations of k elements from an n-element set (0 ≤ k ≤ n).*
- *its participation in the binomial formula*

$$(a+b)^n = C(n,0)a^n + \ldots + C(n,k)a^{n-k}b^k + \ldots + C(n,n)b^n$$

Recurrence relation *(a problem → 2 overlapping subproblems)*

C(n, k) = C(n-1, k-1) + C(n-1, k), for n > k > 0,

C(n, 0) = C(n, n) = 1

Computing Binomial Coefficients

Dynamic Programming for Computing Binomial Coefficients

- *Record the values of the binomial coefficients in a table of n+1 rows and k+1 columns, numbered from 0 to n and 0 to k respectively.*
- *to compute C(n,k), fill the table from row 0 to row n, row by row*
- *each row i (0<=i <=n) from left to right, starting with C(n, 0)= 1,*
- *row 0 through k, end with 1 on the table's diagonal, C(i, i)= 1*
- *other elements, C(n, k) = C(n-1, k-1) + C(n-1, k), using the contents of the cell in the preceding row and the pervious column and the cell in the preceding row and the same column*

	0	1	2	3	4	5	……	k-1	k
0	1								
1	1	1							
2	1	2	1						
3	1	3	3	1					
4	1	4	6	4	1				
5	1	5	10	10	5	1			
k	1								1
…									
n-1	1							C(n-1,k-1)	C(n-1,k)
n	1								C(n,k)

Computing Binomial Coefficients

DP for Computing Binomial Coefficients

ALGORITHM *Binominal* (n,k)

// computes $C(n,k)$ by dynamic programming alg.

for $i = 0$ **to** n **do**

  for $j = 0$ **to** min (i, k) **do**

    if $j = 0$ **or** $j = i$

      BiCoeff [i, j] = 1

    else

      BiCoeff [i, j] = *BiCoeff* [i-1, j-1] + *BiCoeff* [i-1, j]

return *BiCoeff* [n, k]

basic operation: addition

Computing Binomial Coefficients

Efficiency

- *the table can be split into two parts, the first k+1 rows form a triangle, the remaining n-k rows form a rectangle*
- *total number of addition in computing C(n,k)*

$$A(n,k) = \sum_{i=1}^{k}\sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n}\sum_{j=1}^{k} 1 = \sum_{i=1}^{k}(i-1) + \sum_{i=k+1}^{n} k$$

$$= \frac{k(k-1)}{2} + k(n-k) \qquad \in \theta(nk)$$

The Longest Common Subsequence (LCS)

Algorithm 1

Enumerate all subsequences of S_1, and check if they are subsequences of S_2.

Questions:

- How do we implement this?
- How long does it take?

The Longest Common Subsequence (LCS)

Optimal Substructure

Theorem Let $X =< x_1, x_2, \ldots, x_m >$ and $Y =< y_1, y_2, \ldots, y_n >$ be sequences, and let $Z =< z_1, z_2, \ldots, z_k >$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

The Longest Common Subsequence (LCS)

Proof

Let $X =< x_1, x_2, \ldots, x_m >$ and $Y =< y_1, y_2, \ldots, y_n >$ be sequences, and let $Z =< z_1, z_2, \ldots, z_k >$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Proof

1. If $z_k \neq x_m$, then we could append $x_m = y_n$ to Z to obtain a common subsequence of X and Y of length $k+1$, contradicting the supposition that Z is a *longest* common subsequence of X and Y. Thus, we must have $z_k = x_m = y_n$. Now, the prefix Z_{k-1} is a length-$(k-1)$ common subsequence of X_{m-1} and Y_{n-1}. We wish to show that it is an LCS. Suppose for the purpose of contradiction that there is a common subsequence W of X_{m-1} and Y_{n-1} with length greater than $k-1$. Then, appending $x_m = y_n$ to W produces a common subsequence of X and Y whose length is greater than k, which is a contradiction.

The Longest Common Subsequence (LCS)

Proof

Let $X =< x_1, x_2, \ldots, x_m >$ and $Y =< y_1, y_2, \ldots, y_n >$ be sequences, and let $Z =< z_1, z_2, \ldots, z_k >$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Proof

2. If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y. If there were a common subsequence W of X_{m-1} and Y with length greater than k, then W would also be a common subsequence of X_m and Y, contradicting the assumption that Z is an LCS of X and Y.

3. The proof is symmetric to the previous case.

The Longest Common Subsequence (LCS)

Recursion for length

$$c[i,j] = \begin{cases} 0 & \textbf{if } i = 0 \textbf{ or } j = 0\ , \\ c[i-1,j-1]+1 & \textbf{if } i,j > 0 \textbf{ and } x_i = y_j\ , \\ \max(c[i,j-1], c[i-1,j]) & \textbf{if } i,j > 0 \textbf{ and } x_i \neq y_j\ . \end{cases} \quad (1)$$

Last characters match: Suppose $x_i = y_j$. Example: Let $X_i = \langle ABCA \rangle$ and let $Y_j = \langle DACA \rangle$. Since both end in A, we claim that the LCS must also end in A. (We will explain why later.) Since the A is part of the LCS we may find the overall LCS by removing A from both sequences and taking the LCS of $X_{i-1} = \langle ABC \rangle$ and $Y_{j-1} = \langle DAC \rangle$ which is $\langle AC \rangle$ and then adding A to the end, giving $\langle ACA \rangle$ as the answer. (At first you might object: But how did you know that these two A's matched with each other. The answer is that we don't, but it will not make the LCS any smaller if we do.)

Thus, if $x_i = y_j$ then $c[i,j] = c[i-1,j-1]+1$.

Last characters do not match: Suppose that $x_i \neq y_j$. In this case x_i and y_j cannot both be in the LCS (since they would have to be the last character of the LCS). Thus either x_i is *not* part of the LCS, or y_j is *not* part of the LCS (and possibly *both* are not part of the LCS).

In the first case the LCS of X_i and Y_j is the LCS of X_{i-1} and Y_j, which is $c[i-1,j]$. In the second case the LCS is the LCS of X_i and Y_{j-1} which is $c[i,j-1]$. We do not know which is the case, so we try both and take the one that gives us the longer LCS.

Thus, if $x_i \neq y_j$ then $c[i,j] = \max(c[i-1,j], c[i,j-1])$.

The Longest Common Subsequence (LCS)

Code

```
LCS − Length(X, Y)
 1  m ← length[X]
 2  n ← length[Y]
 3  for i ← 1 to m
 4        do c[i, 0] ← 0
 5  for j ← 0 to n
 6        do c[0, j] ← 0
 7  for i ← 1 to m
 8        do for j ← 1 to n
 9              do if x_i = y_j
10                    then c[i, j] ← c[i − 1, j − 1] + 1
11                         b[i, j] ← "↖"
12                    else if c[i − 1, j] ≥ c[i, j − 1]
13                           then c[i, j] ← c[i − 1, j]
14                                b[i, j] ← "↑"
15                           else c[i, j] ← c[i, j − 1]
16                                b[i, j] ← "←"
17  return c and b
```

c[i, j] denote the length of the longest common subsequence of Xi and Yj

b[i,j] some help pointer

The Longest Common Subsequence (LCS)

Example: Calculate the length of LCS

		0	1	2	3	4 = n
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
m= 5	B	0	1	2	2	3

X = BACDB

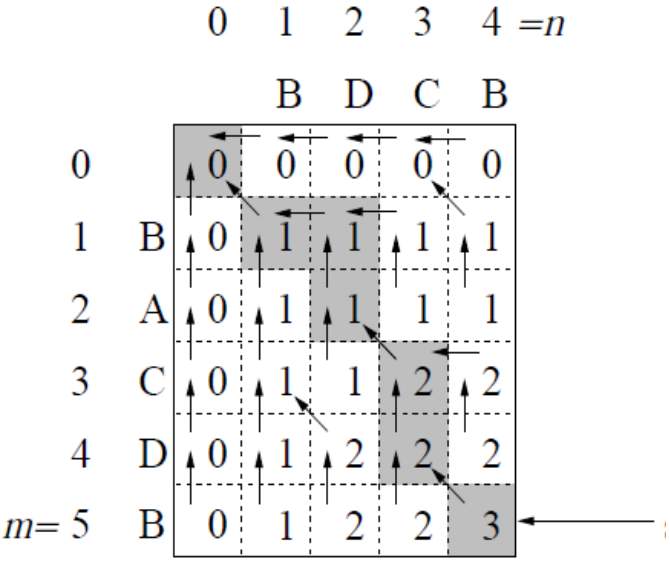
Y = BDCB

LCS = BCB

LCS Length Table

The Longest Common Subsequence (LCS)

Example: Get the LCS



with back pointers included

```
getLCS(char x[1..m], char y[1..n], int b[0..m,0..n]) {
    LCS = empty string
    i = m
    j = n
    while(i != 0 && j != 0) {
        switch b[i,j] {
        case ADDXY:
            add x[i] (or equivalently y[j]) to front of LCS
            i--;   j--;   break
        case SKIPX:
            i--;  break
        case SKIPY:
            j--;  break
        }
    }
    return LCS
}
```

ADDXY: 对角线箭头， SKIPX:向上的箭头， SKIPY: 向左的箭头

Dynamic Matrix Multiplication

Example1: given four matrices A,B,C,D,

- with sizes:

  A: 50*10, B: 10*40, C:40*30, D: 30*5

- different orders of multiplication of these matrices

$(A((BC)D))$	16000
$(A(B(CD)))$	10500
$((AB)(CD))$	36000
$(((AB)C)D)$	87500
$((A(BC))D)$	34500

⟶ *If we have a series of matrices of different sizes that we need to multiply, the order we do it can have a big impact on the number of operations*

Dynamic Matrix Multiplication

- **Problem:** Given a series of matrices A_1, A_2, …, A_n with different sizes $s_1 \times s_2$, $s_2 \times s_3$, …, $s_N \times s_{N+1}$, find the order of multiplication of these matrices such that the total number of multiplications is minimized
- **Exhaustive Search**

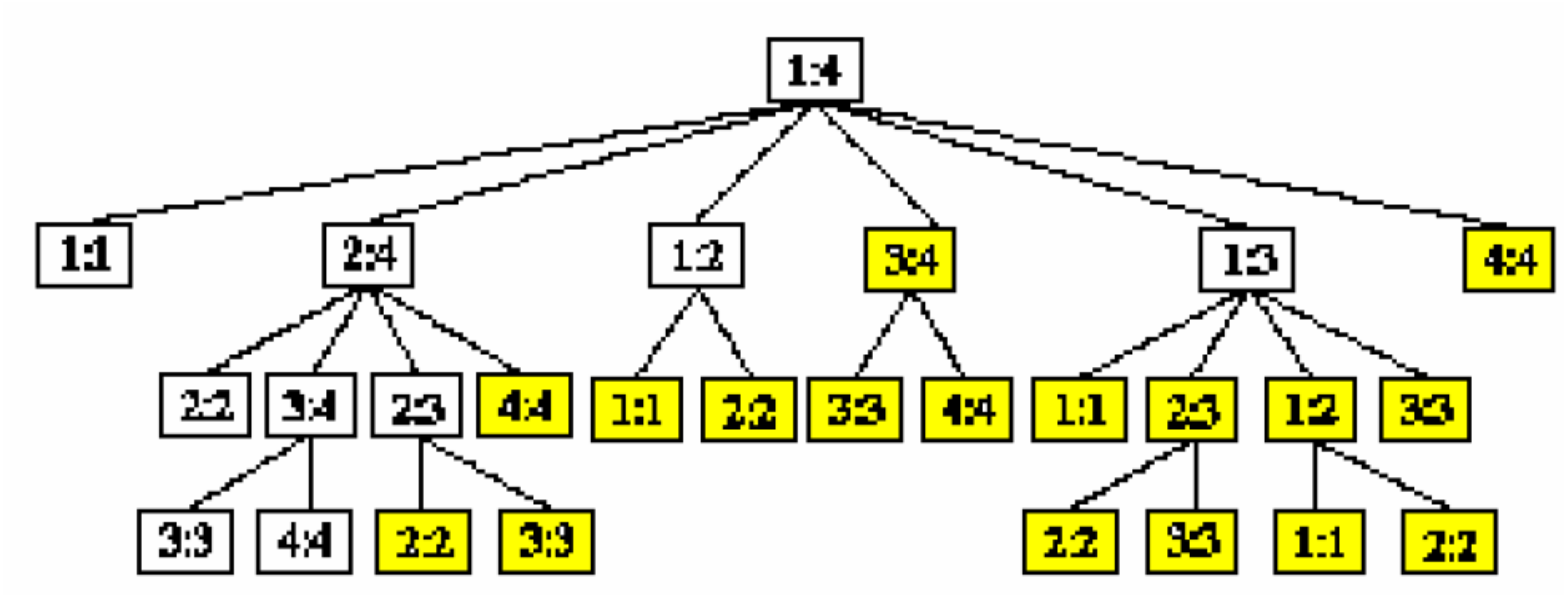
  Try all possibilities, calculate the number of multiplications for each condition, and find out the order with the minimum.

  - Efficiency

  - for a series of n matrices , it could be divided as two sub-sequences, and thus two sub-problems of matrices series $(A_1 \ldots A_k)(A_{k+1} \ldots A_n)$。
  - the number of all possibilities

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

Dynamic Matrix Multiplication



Dynamic Matrix Multiplication

Dynamic Programming

- *Characterize the structure of an optimal solution*
  - *A[i:j]: a series of matrices multiplication* $A_iA_{i+1}\ldots A_j$, $i \le j$
  - *to find optimal multiplication order for A[i:j], suppose the matrix series are divided into two sub-series between* A_k and A_{k+1}, $i \le k < j$, *i.e.* $(A_iA_{i+1}\ldots A_k)(A_{k+1}A_{k+2}\ldots A_j)$
  - *total number of multiplications : The optimal value for A[i:j] must be the minimum one of the optimal values for A[i:k] plus for A[k+1:j] plus the numbers of multiplications for matrix A[i:k]*A[k+1:j]*, i.e. $s_is_{k+1}s_{j+1}$ *for* $i <= k < j$.

Dynamic Matrix Multiplication

- *A key property*
  - *If A[i:j] is divided into A[i:k] and A[k+1:j], then the optimal order for A[i:j] contains the optimal order for A[i:k] and A[k+1:j],*
  - *principle of optimality, an optimal solution to any instance of an optimization problem is composed of optimal solutions to its subinstances*
  - *the challenge lies in deriving an equation relating a solution to any instance to solutions to its smaller subinstances*

Dynamic Matrix Multiplication

- *Recursively define the value of an optimal solution*
  - ***m[i,j]:*** *suppose the minimum number of multiplications for A[i:j]，1≤i≤j≤n, then optimal value for original problem is m[1,n]*
  - *if i=j，A[i:j]=A_i，so* ***m[i,j]=0***，*i=1,2,…,n*
  - *if i<j, based on the principle of optimality, suppose the optimal order divides A[i:j] between A_k and A_{k+1}, then*

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

the size of A_i is $p_{i-1} \times p_i$

  - *the position of k has j-i possibilities, {i, i+1, …, j-1}，among which the position with the minimal number of multiplications min{m[i,j]} is the optimal solution, and the minimal m[i,j] is optimal value*

Dynamic Matrix Multiplication

- *Recursively define the value of an optimal solution (con't)*
  - *Recurrence equation*:

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

  - *s[i,j]: the position at which the matrix sequence is divided corresponding to the optimal number of multiplication m[i,j],*

    --- using s[i,j] could construct the optimal solution recursively

$$((A_i A_{i+1} \cdots A_{i+b})(A_{i+b+1} \cdots A_k))(A_{k+1} A_{k+2} \cdots A_{k+c} A_{k+c+1} \cdots A_j)$$

Dynamic Matrix Multiplication

- *Compute the value of optimal solution in bottom-up fashion*
  - *dynamic programming, compute recursively in bottom-up manner*
    - *all the entries in the table are initialized with null to indicate that they have not been calculated*
    - *whenever a new value needs to be calculated, the method checks the corresponding entry in the table first,*

      - if this entry in not null, it is simply retrieved from the table

      - otherwise, it is computed by the recursive call, and the result is recorded in the table

Dynamic Matrix Multiplication

```
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++)              //矩阵链长度r
        for (int i = 1; i <= n - r+1; i++) {
          int j=i+r-1;
          m[i][j] = 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;}
          }
       }
}
```

按矩阵链长递增的方式依次计算
m[i][i+1],i=1,2,…,n-1
m[i][i+2],i=1,2,…,n-2
m[i][i+3],i=1,2,…,n-3

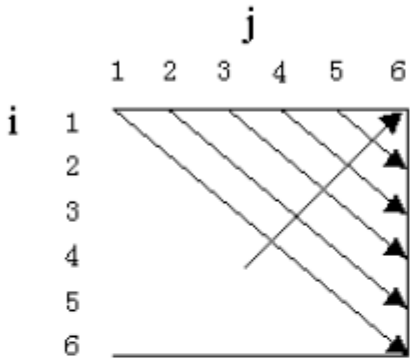
Dynamic Matrix Multiplication

- *Efficiency*

  $O(n^3)$

Dynamic Matrix Multiplication

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



(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]

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

$$=7125$$

$$\therefore\ s[2][5]=3$$

Dynamic Matrix Multiplication

- *using s[i,j] could construct the optimal solution recursively*
  from s[1][n], the optimal position for A[1:n] is

  (A[1 : s[1][n]]) (A[s[1][n]+1: n])

  the optimal position for A[1 : s[1][n]] *is*

  (A[1 : s[1][s[1][n]]])(A[s[1][s[1][n]]+1 : s[1][n]])

```
void Traceback(int i,  int j,  int **s)
{
      if(i==j) return;
      Traceback (i, s[i][j], s);
      Traceback (s[i][j]+1, j, s);
      count<<"multiply A"<< i << "," << s[i][j];
      count<<"and A"<< (s[i][j]+1) << "," << j<< endl;
}
```

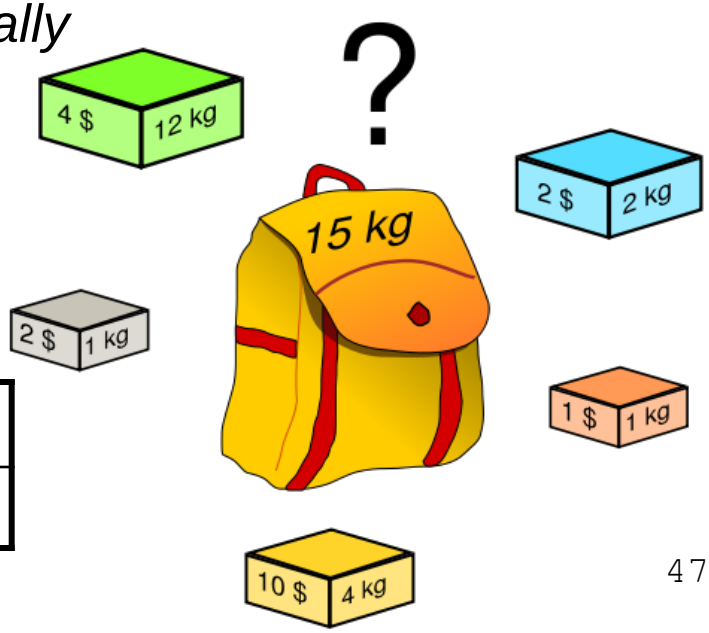
0-1 *Knapsack Problem*

Problem

Given n items of known weights $w_1,\ldots, w_n$ and values $v_1,\ldots, v_n$ and a knapsack of capacity W. Find the most valuable subset of the given n items to fit into the knapsack W?

- *two possibilities for item i, totally included in the knapsack, or else, not included in the knapsack*
- *not permitted to be included partially*

—— **0-1 *Knapsack Problem***



weights	w_1	w_2	...	w_n
values	v_1	v_2	...	v_n

0-1 *Knapsack Problem*

- *mathematical model*

  0-1 *Knapsack Problem is a kind of integer linear programming problem,*

  given $W>0$, $w_i>0$, $v_i>0$, $1 \le i \le n$, *find a n-ary* 0-1 *vector* $(x_1,x_2,\ldots,x_n)$ *to satisfy*

  Objective

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

  under the condition

  Constraints

$$\sum_{i=1}^{n} w_i x_i \le W \qquad x_i \in \{0,1\},\ 1 \le i \le n$$

0-1 *Knapsack Problem*

- ***Dynamic Programming -*** *recurrence* **from item1 to n**
  - ***Main Idea:*** *to derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller subinstances*.
  - ***Some Notations:***
    - *a subinstance of the first* ***i*** *items*, $0 \le i < n$, *of known weights* $w_1, \ldots, w_i$ *and values* $v_1, \ldots, v_i$;
    - *a knapsack of capacity* ***j***, $1 \le j < W$;
    - *Let* $V[i,j]$ *be the value of an optimal solution to this instance, i.e., the value of the most valuable subset of the first i items that fit into the knapsack of capacity j.*
  - *recurrence computing to get* ***V[n, W], the maximum value*** *of a subset of the n given items to fit into the knapsack of capacity* W

0-1 *Knapsack Problem*

- *compute from* ***item*1 to *item n*** *('cont)*

  - ***If*** *the i^{th} item does not fit into the knapsack, the value of an optimal subset selected from the first i items is same as the value of an optimal subset selected from the first i-1 items.*

  - ***else,***

    - *among the subsets that do not include the i^{th} item, the value of an optimal solution is V[i-1,j];*
    - *among the subsets that do include the i^{th} item (hence, $j-w_i \geq 0$), an optimal solution is made up of this item and an optimal subset of the first i-1 items that fit into the knapsack of capacity $j-w_i$, the value of such an optimal subset is $v_i + V[i-1, j-w_i]$;*

0-1 *Knapsack Problem*

- *compute* ***from item1 to item n*** *('cont)*
  - *Recursively define the value of an optimal solution in terms of solutions to smaller problems.*

Initial Settings:

$V[0, j] = 0 \quad \text{for } j \geq 0; \qquad V[i,0] = 0 \quad \text{for } i \geq 0;$

Recursive Step:

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

0-1 *Knapsack Problem*

- *compute* ***from item*1 to *item n*** *(‘cont)*
  - *Bottom-up computation of V[i,j] using an Iterative way instead of a recursive way.*

Bottom-up computation: Computing the table using

$$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w-w_i])$$

row by row.

V[i,w]	w=0	1	2	3	…	…	W
i= 0	0	0	0	0	…	…	0
1	→	→	→	→	→	→	→
2	→	→	→	→	→	→	→
⋮	→	→	→	→	→	→	→
n	→	→	→	→	→	→	→

bottom

↓

up

0-1 *Knapsack Problem*

- *compute* ***from item1 to item n*** *('cont)*
  - ***DP Code: here V[i,w] is equal to V[i, j] in our slides.***

```
KnapSack(v, w, n, W)
{
  for (w = 0 to W) V[0, w] = 0;
  for (i = 1 to n)
    for (w = 0 to W)
      if (w[i] ≤ w)
        V[i, w] = max{V[i − 1, w], v[i] + V[i − 1, w − w[i]]};
      else
        V[i, w] = V[i − 1, w];
  return V[n, W];
}
```

Time complexity: Clearly, $O(nW)$.

0-1 *Knapsack Problem*

- ***Based on the Optimal Result How to Construct an Optimal Solution.***

  - The algorithm for computing $V[i, w]$ described in the previous slide does not keep record of which subset of items gives the optimal solution.

  - To compute the actual subset, we can add an auxiliary boolean array $keep[i, w]$ which is 1 if we decide to take the i-th file in $V[i, w]$ and 0 otherwise.

0-1 *Knapsack Problem*

- ***Based on the Optimal Result How to Construct an Optimal Solution.***

  Question: How do we use the values $keep[i, w]$ to determine the subset T of items having the maximum computing time?

  If keep$[n, W]$ is 1, then $n \in T$. We can now repeat this argument for keep$[n-1, W-w_n]$.
  If keep$[n, W]$ is 0, the $n \notin T$ and we repeat the argument for keep$[n-1, W]$.

- ***Code:***

```
K = W;
for (i = n downto 1)
    if (keep[i, K] == 1)
    {
        output i;
        K = K - w[i];
    }
```

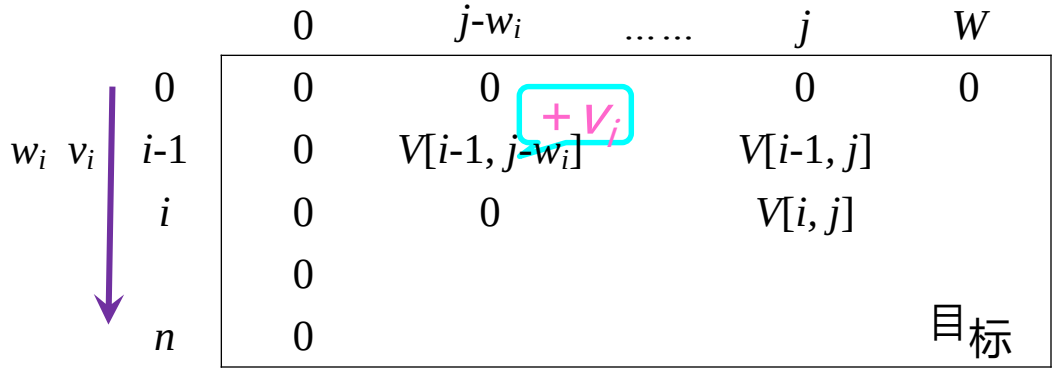
0-1 *Knapsack Problem*

- ***Complete DP Code for bottom-up 0-1 Knapsack Problem***

```
KnapSack(v, w, n, W)
{
    for (w = 0 to W) V[0, w] = 0;
    for (i = 1 to n)
        for (w = 0 to W)
            if ((w[i] ≤ w) and (v[i] + V[i − 1, w − w[i]] > V[i − 1, w]))
            {
                V[i, w] = v[i] + V[i − 1, w − w[i]];
                keep[i, w] = 1;
            }
            else
            {
                V[i, w] = V[i − 1, w];
                keep[i, w] = 0;
            }
    K = W;
    for (i = n downto 1)
        if (keep[i, K] == 1)
        {
            output i;
            K = K − w[i];
        }
    return V[n, W];
}
```

0-1 *Knapsack Problem Example*

Ex. recurrence from the first item



item	weight	value
1	2	12 ¥
2	1	10 ¥
3	3	20 ¥
4	2	15 ¥

	i	0	1	2	3	4	5			
	0	0	0	0	0	0	0	$V(i\text{-}1, j\text{-}w_1)+v_1$		$V(i\text{-}1, j)$
w_1=2. v_1=12	1	0	0	12	12	12	12	$V(i\text{-}1, j\text{-}w_2)+v_2$	$V(i\text{-}1, j)$	$V(i, j)$
w_2=1. v_2=10	2	0	10	12	22	22	22		$V(i, j)$	
w_3=3 v_3=20	3	0	10	12	22	30	32			
w_4=2. v_4=15	4	0	10	15	25	30	37			

0-1 *Knapsack Problem Example*

Ex. recurrence from the first item

	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
w_1=2. v_1=12	1	0	0	12	12	12	12
w_2=1. v_2=10	2	0	10	12	22	22	22
w_3=3 v_3=20	3	0	10	12	22	30	32
w_4=2. v_4=15	4	0	10	15	25	30	37

$V(i\text{-}1, j\text{-}w_1)+v_1$ | | $V(i\text{-}1, j)$

$V(i\text{-}1, j\text{-}w_2)+v_2$ | $V(i\text{-}1, j)$ | $V(i, j)$

$V(i, j)$

Composition of an optimal solution, through tracing back the computations of the last entry V [4,5].

① $V[4,5] \neq V[3,5]$, *item* 4 *is included in an optimal solution, with an optimal subset for* $V[3,3]$;

② $V[3,3] = V[2,3]$, *item* 3 *not included in an optimal subset,*

③ $V[2,3] \neq V[1,3]$, *item* 2 *is included in an optimal subset*

④ $V[1,2] \neq V[0,2]$, *item* 1 *is included in an optimal subset .*

So, optimal solution is x={1,1,0,1}, i.e. item{1,2,4}

0-1 *Knapsack Problem*

- ***Dynamic Programming -*** *recurrence from* ***item n*** **to 1**
  - *principle of optimality* (*recurrence from item n* to 1)
    - *suppose* $(y_1,y_2,...,y_n)$ *is an optimal solution for a given* 0-1 *knapsack, then* $(y_2,y_3,...,y_n)$ *is an optimal solution for its subproblem*

$$\max \sum_{i=2}^{n} v_i x_i \quad \sum_{i=2}^{n} w_i x_i \le W - w_1 y_1 \qquad x_i \in \{0,1\}, \quad 2 \le i \le n$$

- **proof by contradiction:**

suppose $(z_2,z_3,...,z_n)$ is the optimal solution for above subproblem, and $(y_2,y_3,...,y_n)$ is not its the optimal solution, then we can get

$$\sum_{i=2}^{n} v_i z_i > \sum_{i=2}^{n} v_i y_i \text{ , then } v_1 y_1 + \sum_{i=2}^{n} v_i z_i > \sum_{i=1}^{n} v_i y_i$$

$$\sum_{i=2}^{n} w_i z_i \le W - w_1 y_1 \text{ , then } w_1 y_1 + \sum_{i=2}^{n} w_i z_i \le W$$

so $(y_1,z_2,...,z_n)$ is a **more** optimal solution for the original 0-1 knapsack problem, and $(y_1,y_2,...,y_n)$ is not its optimal solution → **contradiction**

0-1 *Knapsack Problem*

- *recurrence equation from item n* to 1

  - ***m(i, j):*** *optimal value for the following* 0-1 *knapsack subproblem*

$$\max \sum_{k=i}^{n} v_k x_k$$
$$\sum_{k=i}^{n} w_k x_k \le j \qquad x_k \in \{0,1\}, \quad i \le k \le n$$

i.e. m(i, j) is the optimal value when selected from i，i+1，…，n for knapsack W

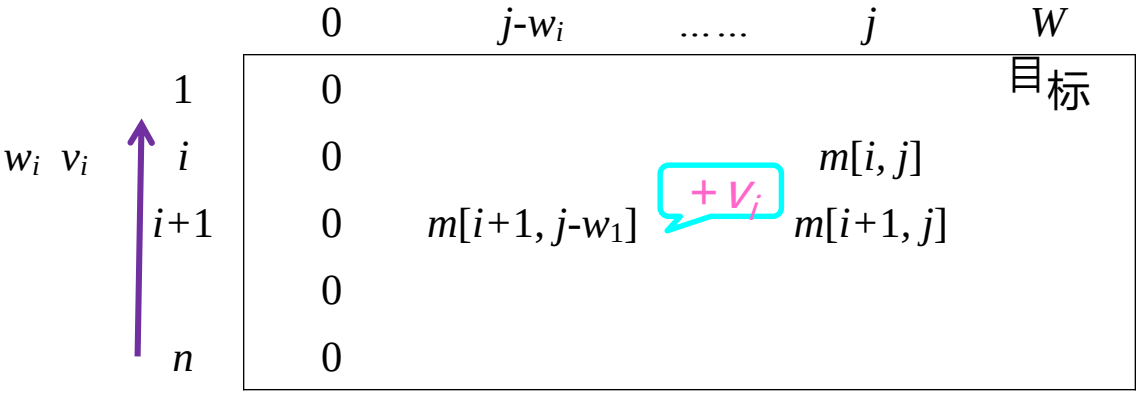
  - *Based on the principle of optimality of* 0-1 *Knapsack problem, we can construct the recurrence equation for m(i, j)*

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

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

0-1 *Knapsack Problem*

Ex. recurrence from the last item



item	weight	value
1	2	12 ¥
2	1	10 ¥
3	3	20 ¥
4	2	15 ¥

	i	0	1	2	3	4	5			
w_1=2. v_1=12	1	0	10	15	25	30	37			
w_2=1. v_2=10	2	0	10	15	25	30	35		$m(i, j)$	
w_3=3 v_3=20	3	0	0	15	20	20	35	$m(i+1, j-w_2)+v_2$	$m(i+1, j)$	$m(i, j)$
w_4=2. v_4=15	4	0	0	15	15	15	15	$m(i+1, j-w_3)+v_3$		$m(i+1, j)$

m[1,5]≠*m*[2,5], *item* 1 *is included in an optimal solution, with an optimal subset for* m[2,3]

m[2,3] ≠ *m*[3,3], *item* 2 *is included in an optimal subset* ,

m[3,2] = *m*[4,2], *item* 3 *not included in an optimal subset* ,

m[4,2] ≠ *0*, *item* 4 *is included in an optimal subset* . *So, optimal solution is* {1,1,0,1}, *i.e. item{1,2,4}*

0-1 *Knapsack Problem*

算法改进 — 记忆法

```
算法  MFKnapsack (i, j)
//对背包问题实现记忆功能方法
//输入：一个非负整数 i 表示先考虑的物品数量，一个非负整数 j 表示背包的承重量
//输出：前 i 个物品的最优可行子集的价值
//注意：我们把输入数组 Weights[1..n]，Values[1..n]和表格 F[0..n, 0..W]作为全局变量
//除了行 0 和列 0 用 0 初始化以外，F 的所有单元格都用-1 初始化
if  F[i, j] < 0
    if  j < Weights[i]
        value ← MFKnapsack(i − 1, j)
    else
        value ← max(MFKnapsack(i − 1, j),
                    Values[i] + MFKnapsack(i − 1, j − Weights[i]))
    F[i, j] ← value
return  F[i, j]
```

0-1 *Knapsack Problem*

算法改进 — 记忆法

例 2　对例 1 中的实例应用记忆功能法。图 8.6 给出了结果。只计算了 20 个有效值(也就是不在行 0 和列 0 上的)中的 11 个。只有一个有效单元 $V(1,2)$ 的值是从表中取到的，而不是计算得来的。对于较大的实例，这种单元的比例会显著地增加。

		承重量 j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1=2,\ v_1=12$	1	0	0	12	12	12	12
$w_2=1,\ v_2=10$	2	0	—	12	22	—	22
$w_3=3,\ v_3=20$	3	0	—	—	22	—	32
$w_4=2,\ v_4=15$	4	0	—	—	—	—	**37**

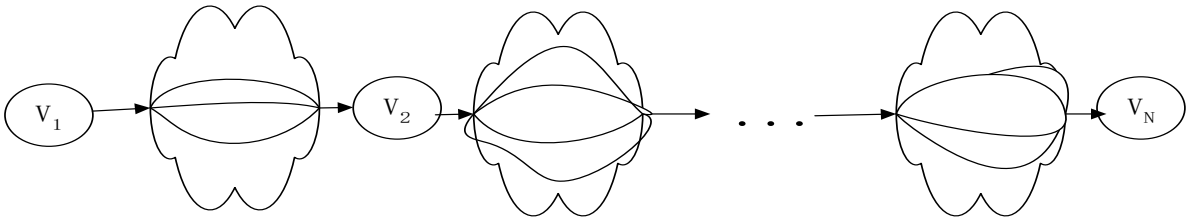
图 8.6　用记忆功能算法解背包问题的一个实例

Multistage Decision Processes

多阶段决策问题

多阶段决策过程 multistep decision process

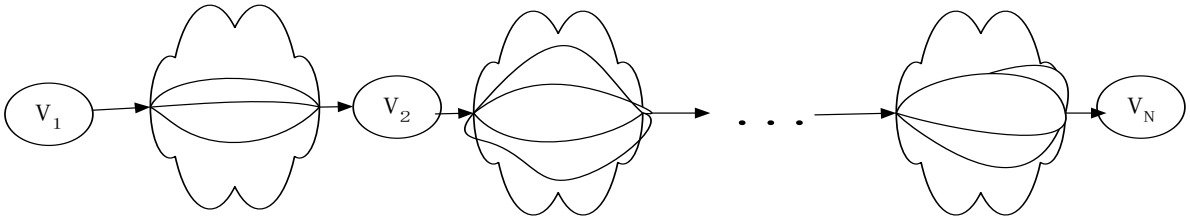
- 问题的活动过程分为若干相互联系的阶段
- 在每一个阶段都要做出决策，这决策过程称为多阶段决策过程
- 任一阶段 i 以后的行为仅依赖于i 阶段的过程状态，而与i 阶段之前的过程如何达到这种状态的方式无关



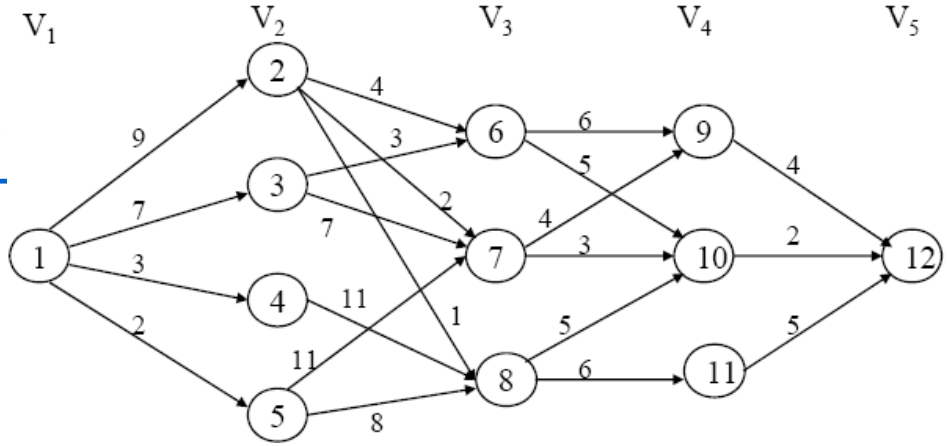
Multistage Decision Processes

最优化问题

- 问题的每一阶段可能有多种可供选择的决策，必须从中选择一种决策。
- 各阶段的决策构成一个决策序列。
- 决策序列不同，所导致的问题的结果可能不同。
- 多阶段决策的最优化问题就是：在所有容许选择的决策序列中选择能够获得问题最优解的决策序列——最优决策序列。



Multistage Decision



多段图问题

多段图

- 多段图G=(V,E)是一个有向图，且具有特性:

结点：结点集V被分成 k≥2个不相交的集合V_i，1≤i≤k，
其中V_1和V_k分别只有一个结点：s (源结点)和 t (汇点)。

段：　每一集合V_i定义图中的一段——共*k*段。

边：　所有的边(u,v)，若<u,v>∈E，则
若u∈V_i，则v∈V_{i+1}，即该边是从某段 *i* 指向*i*+1段，1≤*i*≤k - 1。

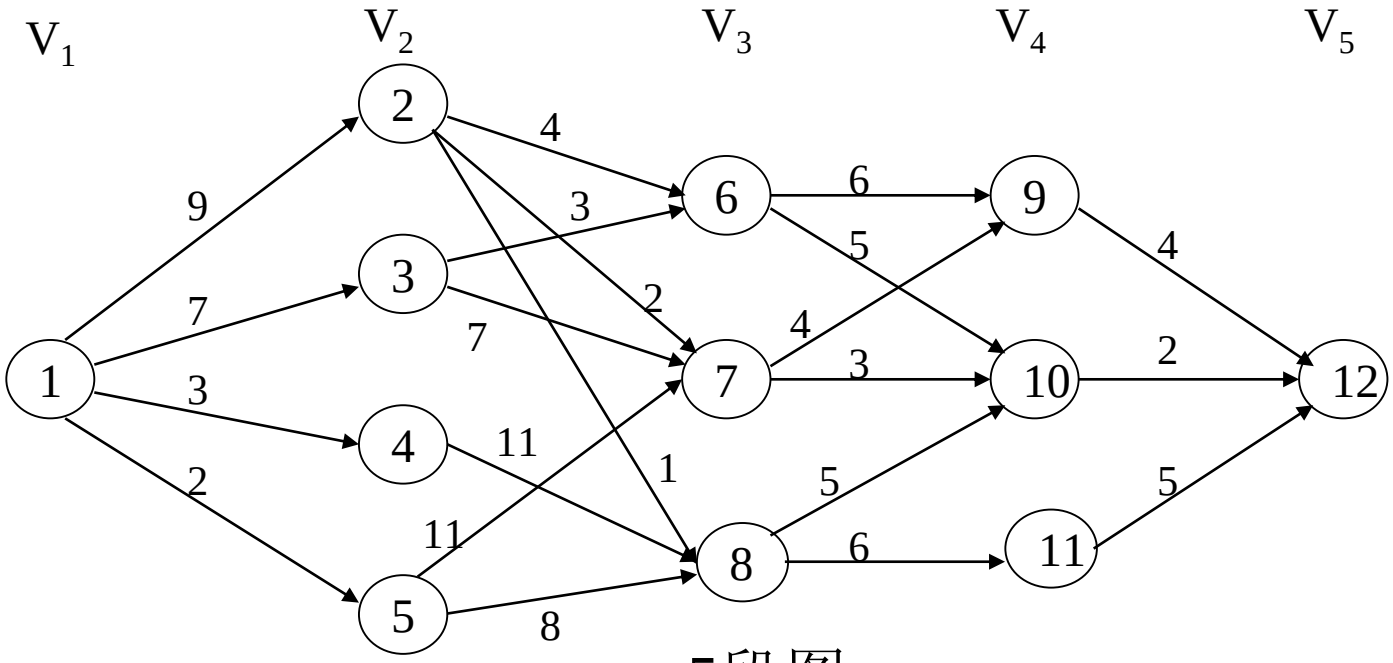
成本：每条边(u,v)均附有成本**w(u,v)**。

s到t的路径：是一条从第1段的源点s出发，依次经过第2段的某结点$v_{2,i}$，
经第3段的某结点$v_{3,j}$、…、最后在第k段的汇点t结束的路径。

该路径的成本是这条路径上边的成本和。

- 多段图问题：求由s到t的最小成本路径。

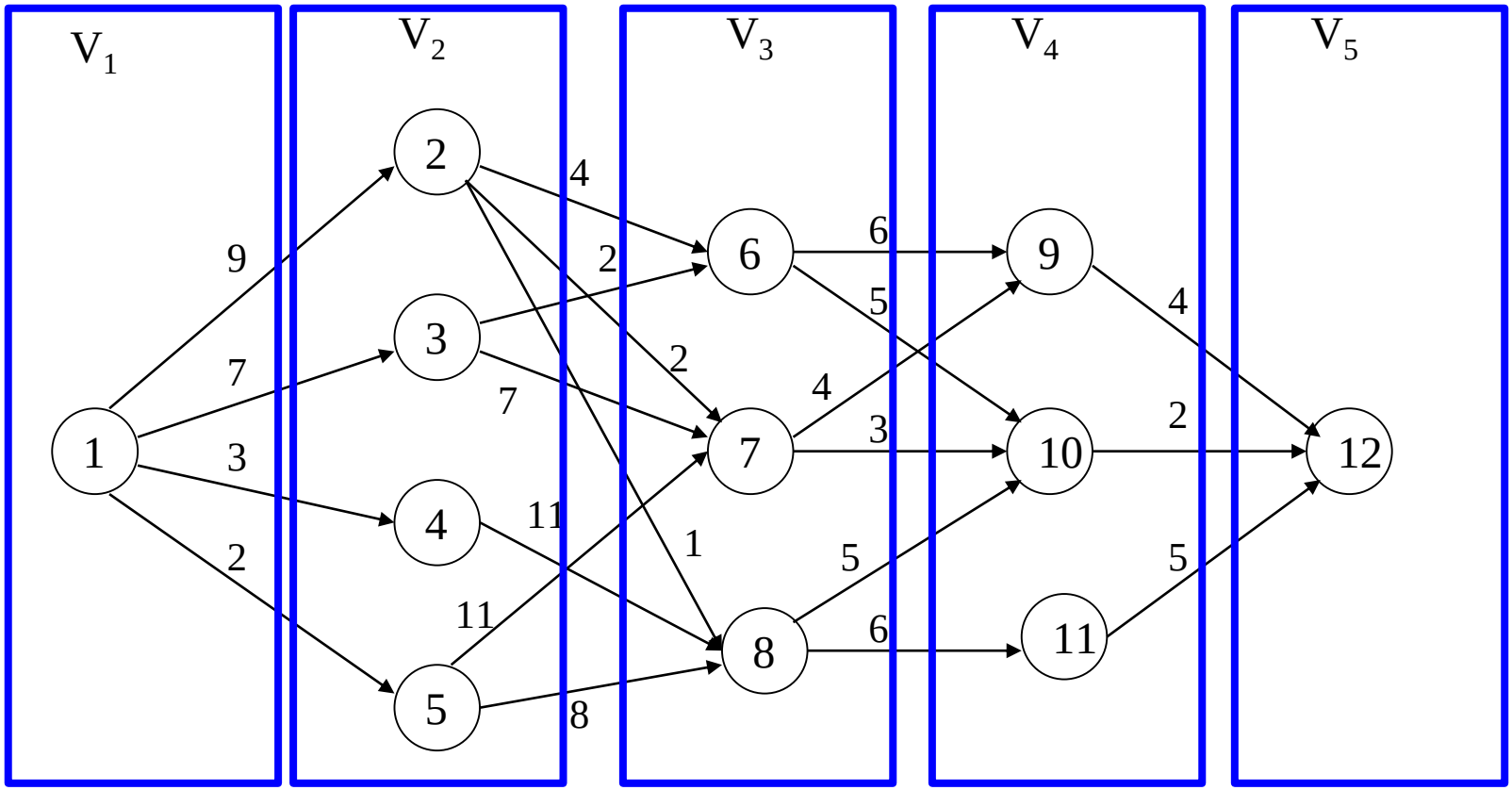
Multistage Decision Processes



5段图

- **多段图问题的多阶段决策过程：**
  - 生成从s到t的最小成本路径
  - 在k-2个阶段（除s和t外）进行某种决策的过程：
  - 从s开始，第i次决策决定V_{i+1}(1≤i≤k-2)中的哪个结点在从s到t的最短路径上。

Multistage Decision Processes



5段图

Multistage Decision Processes

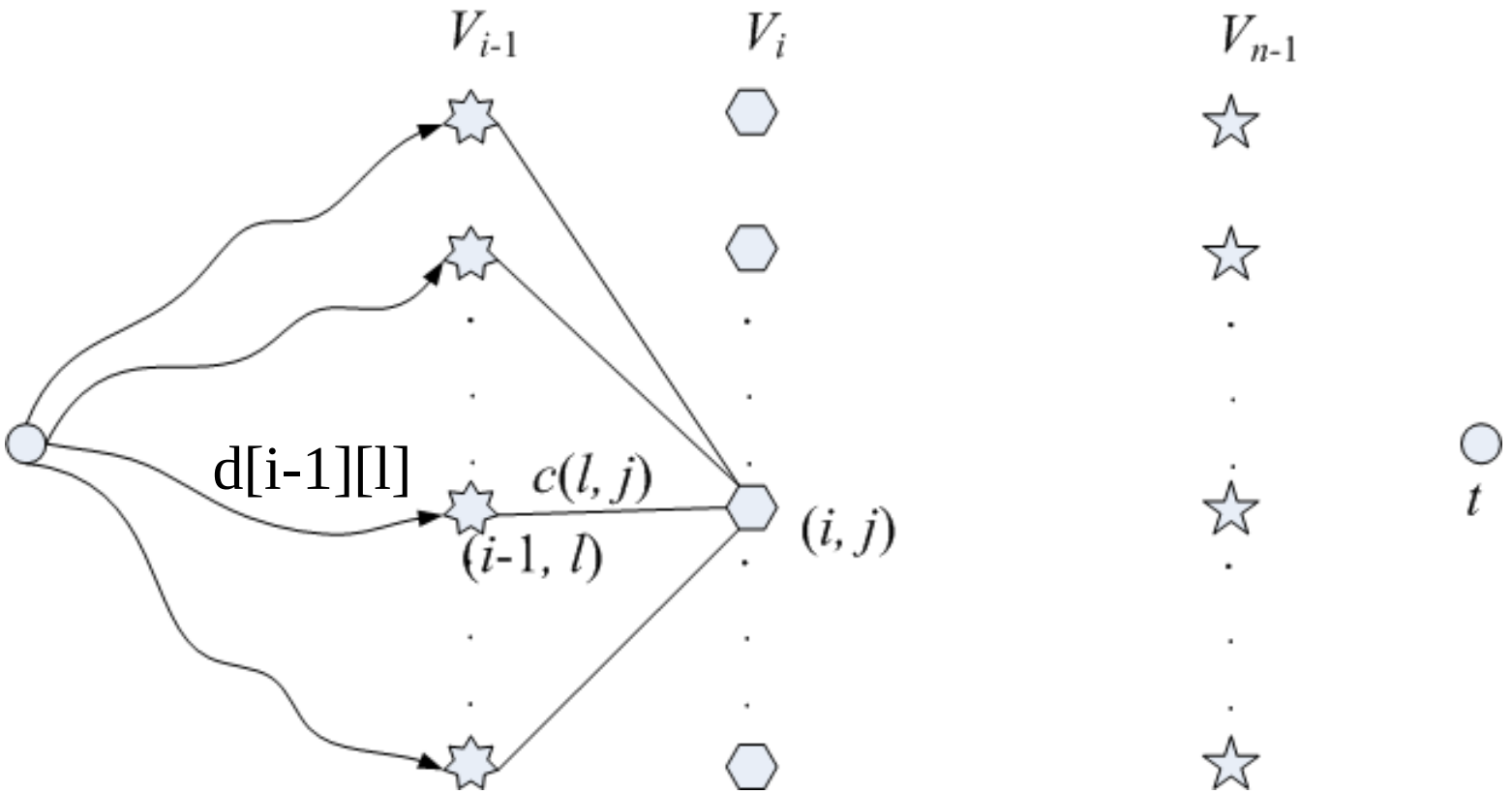
最优性原理对多段图问题成立

- 假设s,v_2,v_3,…,v_{k-1},t 是一条由s到t 的最短路径。
  - 初始状态：源点 s
  - 初始决策：从s到结点v_2 (s,v_2)， $v_2 \in V_2$
  - 初始决策产生的状态：v_2
- 如果把v_2看作原问题的一个子问题的初始状态，则解这个子问题就是找出一条由v_2到t 的最短路径，显然就是v_2,v_3,…,v_{k-1},t 即

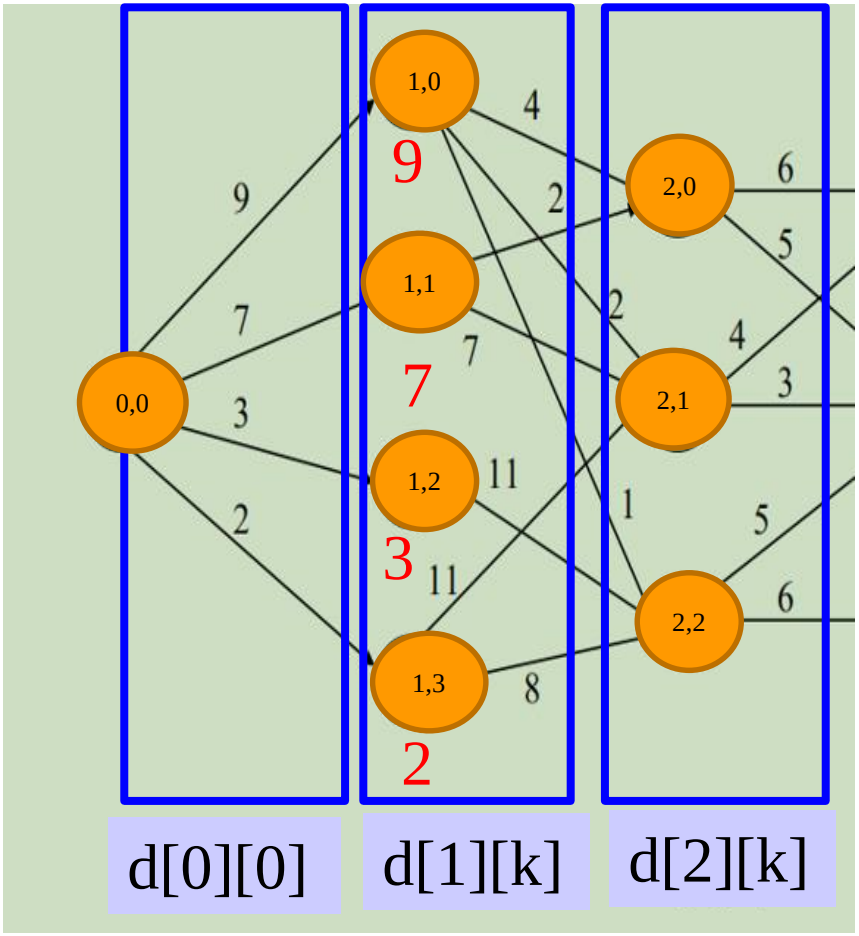
其余的决策：v_3,…,v_{k-1}相对于v_2将构成一个最优决策序列——最优性原理成立。

- 反证：若不然，设v_2,q_3,…,q_{k-1},t是一条由v_2到t的更短的路径，则s, v_2,q_3,…,q_{k-1},t 将是比s,v_2,v_3,…,v_{k-1},t更短的从s到t的路径。与假设矛盾。

Multistage Decision Processes: Forward



Multistage Decision Processes: FMDP



d[0][0]=0

d[1][k]:

d[1][0]= w[0][0][0]= 9
d[1][1]= w[0][0][1]= 7
d[1][2]= w[0][0][2]= 3
d[1][3]= w[0][0][3]= 2

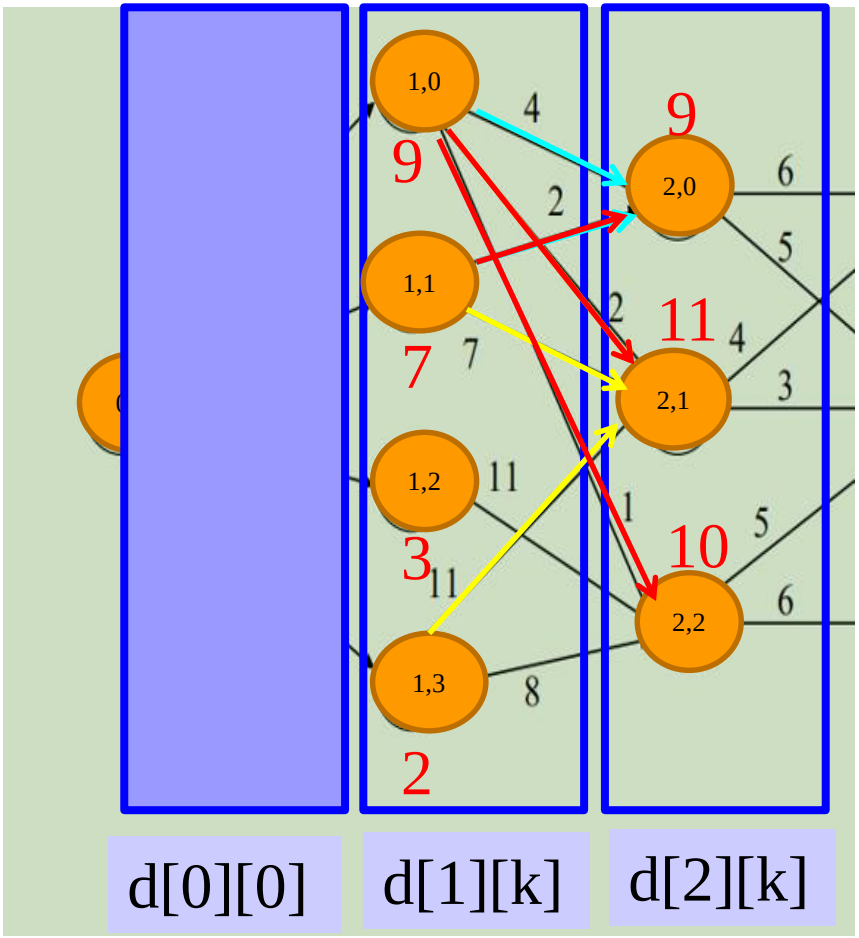
w[i][j][k] 第i个阶段第j个点到下一阶段第k个点的边权值
d[i][j] 第i个阶段到第j个点积累的代价

dij 是贪心
多阶段决策动态规划
TSP也是单源最短路径

两种方式 自上而下和自下而上
求完所有的中间状态再求解
需要的时候再算

w[0][0][0]=9 w[1][3][2]=8

Multistage Decision Processes: FMDP



d[1][k]:

d[1][0]= 9; d[1][1]= 7

d[1][2]= 3; d[1][3]= 2

d[2][k]:

d[2][0]=min{d[1][0]+w[1][0][0], d[1][1] +w[1][1][0]}}

=min{d[1][k]+w[1][k][0]}

=min{9+4;7+2} =9

d[2][1]=min{d[1][k]+w[1][k][1]}

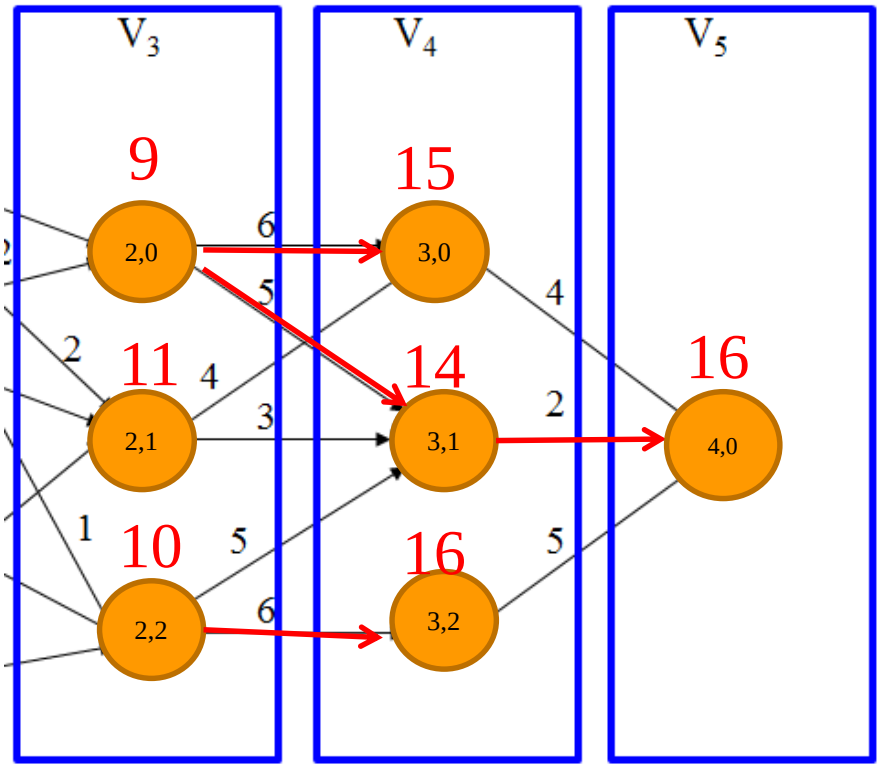
=min{9+2;7+7;2+11} =11

d[2][2]=min{d[1][k]+w[1][k][2]}

=min{9+1;3+11;2+8}=10

w[0][0][0]=9 w[1][3][2]=8

Multistage Decision Processes: FMDP



d[3][k]:

d[3][0]=min{d[2][k]+w[2][k][0]}
=min{9+6;11+4} =15

d[3][1]=min{d[2][k]+w[2][k][1]}
=min{9+5;11+3;10+5}=14

d[3][2]=min{d[2][k]+w[2][k][2]}
=min{10+6}=16

d[4][k]:

d[4][0]=min{d[3][k]+w[3][k][0]}
=min{15+4; 14+2; 16+5}=16

General form: d[i][j]=min{d[i-1][k]+w[i-1][k][j]}

Multistage Decision Processes

Forward MDP

- 设 *path[i, j]* 是一条从V_0 到V_i中的结点 j 的最小成本路径，d*[i, j]* 是这条路径的成本。

Recursive Equation

$$d[\mathrm{i}][\mathrm{j}] = \min_{\substack{j \in V_i \\ (\mathrm{k},\mathrm{j}) \in \mathrm{E}}} \{\mathrm{d}[\mathrm{i}-1][\mathrm{k}] + \mathrm{w}[\mathrm{i}-1][\mathrm{k}][\mathrm{j}]\}$$

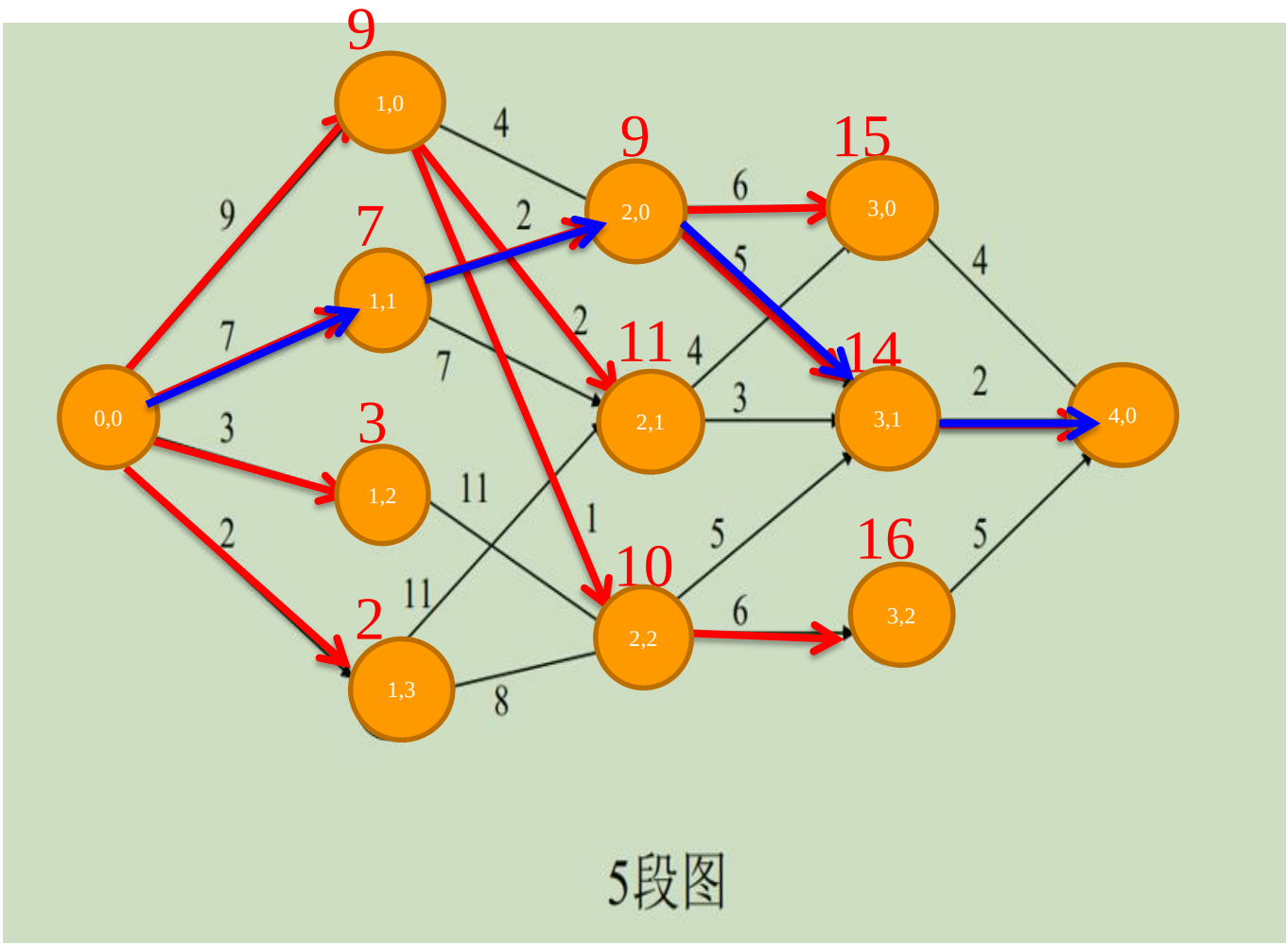
$$path[\mathrm{i}][\mathrm{j}] = \arg\min_{k} \{\mathrm{d}[\mathrm{i}-1][\mathrm{k}] + \mathrm{w}[\mathrm{i}-1][\mathrm{k}][\mathrm{j}]\}$$

Initialization

第1段

$$\mathrm{d}[0,\mathrm{j}] = \begin{cases} \mathrm{w}[0,0,\mathrm{j}] & <0,j> \in E \\ \infty & <0, j> \notin E \end{cases}$$

Forward Multistage Decision Processes



path[2][0]=1
path[2][1]=0
path[2][2]=0

path[3][0]=0
path[3][1]=0
path[3][2]=2

path[4][0]=1

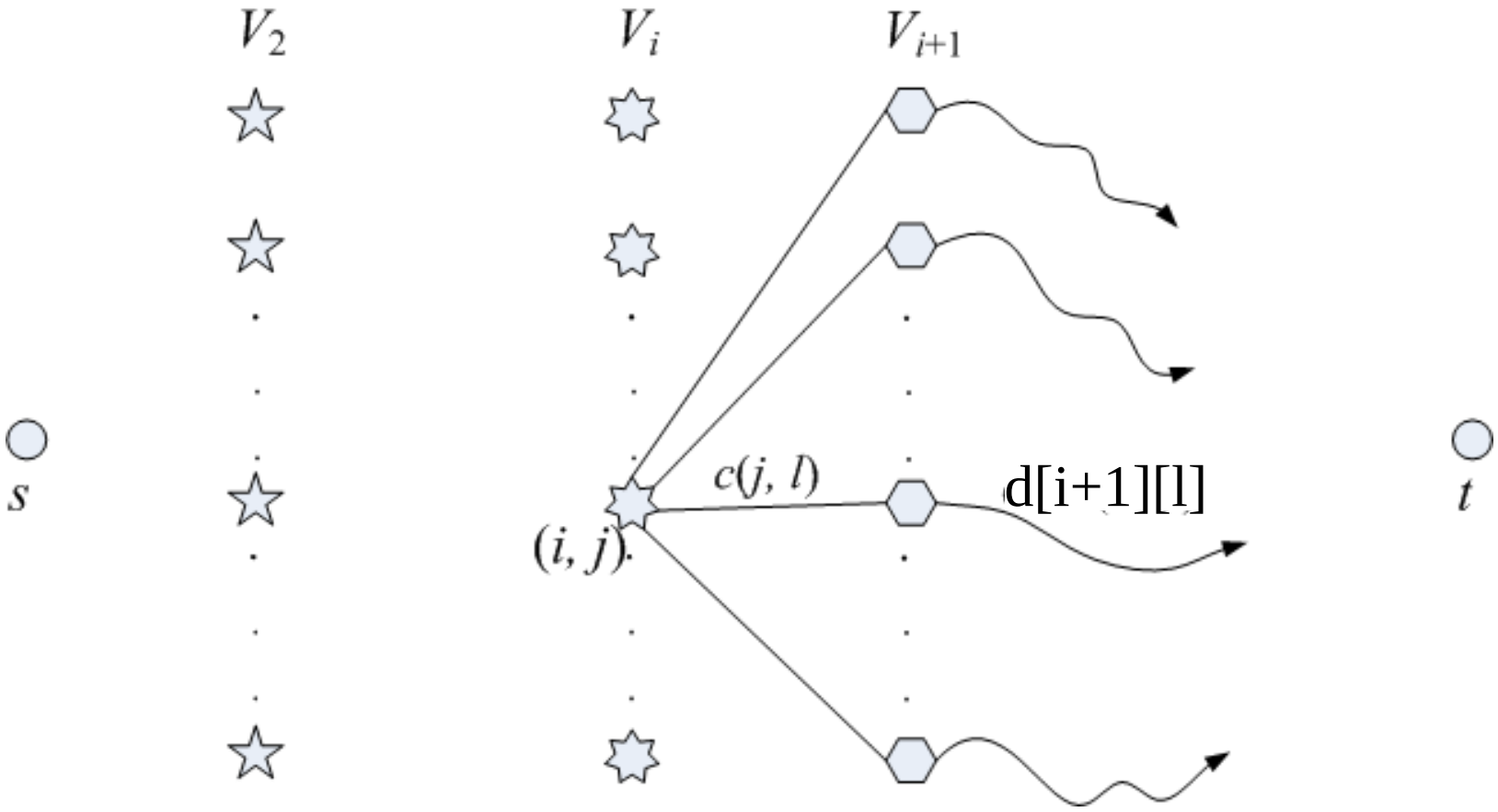
policy=[0,1,0,1,0]

Multistage Decision Processes

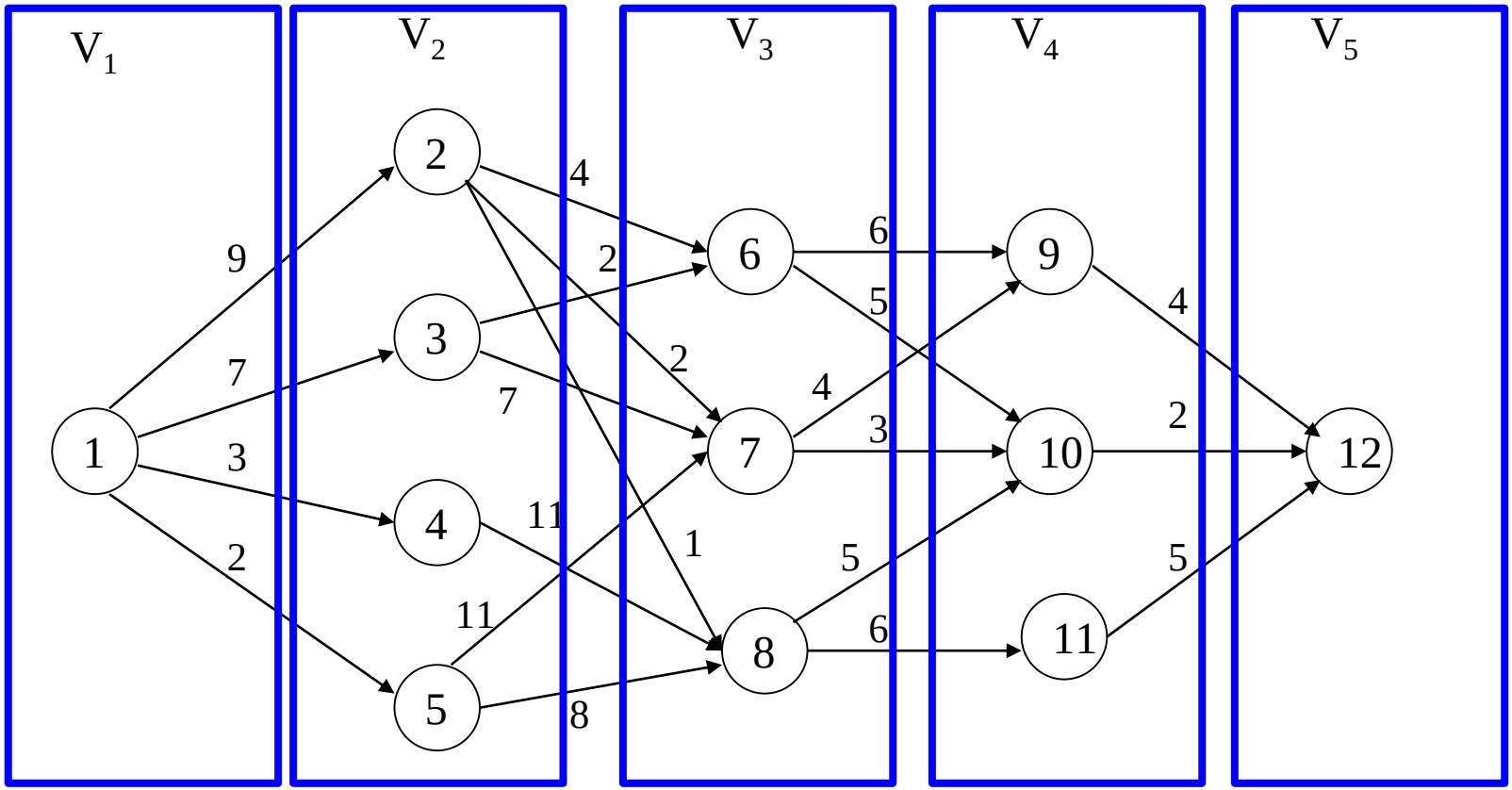
Algorithm FMDP(structure s,int w)

```
//input 表示多段图的结构体数组 s, 以及路径矩阵 w;
//output the cost matrix d, and the path

d[0][0]=0;  path[0][0]=0;
n=the number of stages;
for(int i=1;i<n;i++)
            for(int j=0;j<s[i].no;j++){
                min_d = d[i-1][0]+w[i-1][0][j];
                 min_k = 0;
                for(int k=1;k<s[i-1].no;k++){
                     d = d[i-1][k]+w[i-1][k][j] ;
                    if (d <=  min_d)    min_d = d;  min_k =k;}
                  d[i][j]=min_d;   path[i][j]=k;}

policy[]=0; a=0;
for(int i=n-1;i>0;i--) {
        a=path[i][a];   policy[i-1]=a;
}
```

Multistage Decision Processes: Backward

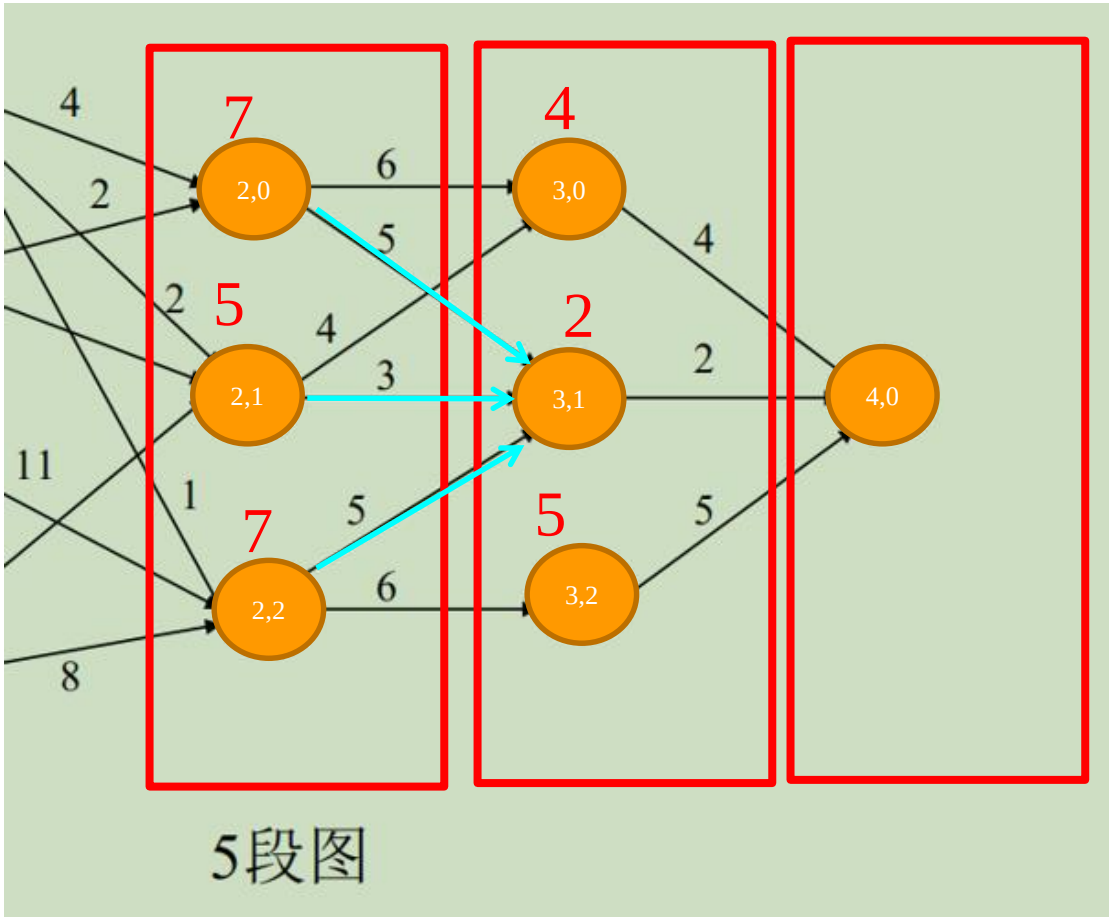


Multistage Decision Processes: Backward



5段图

Multistage Decision Processes: Backward



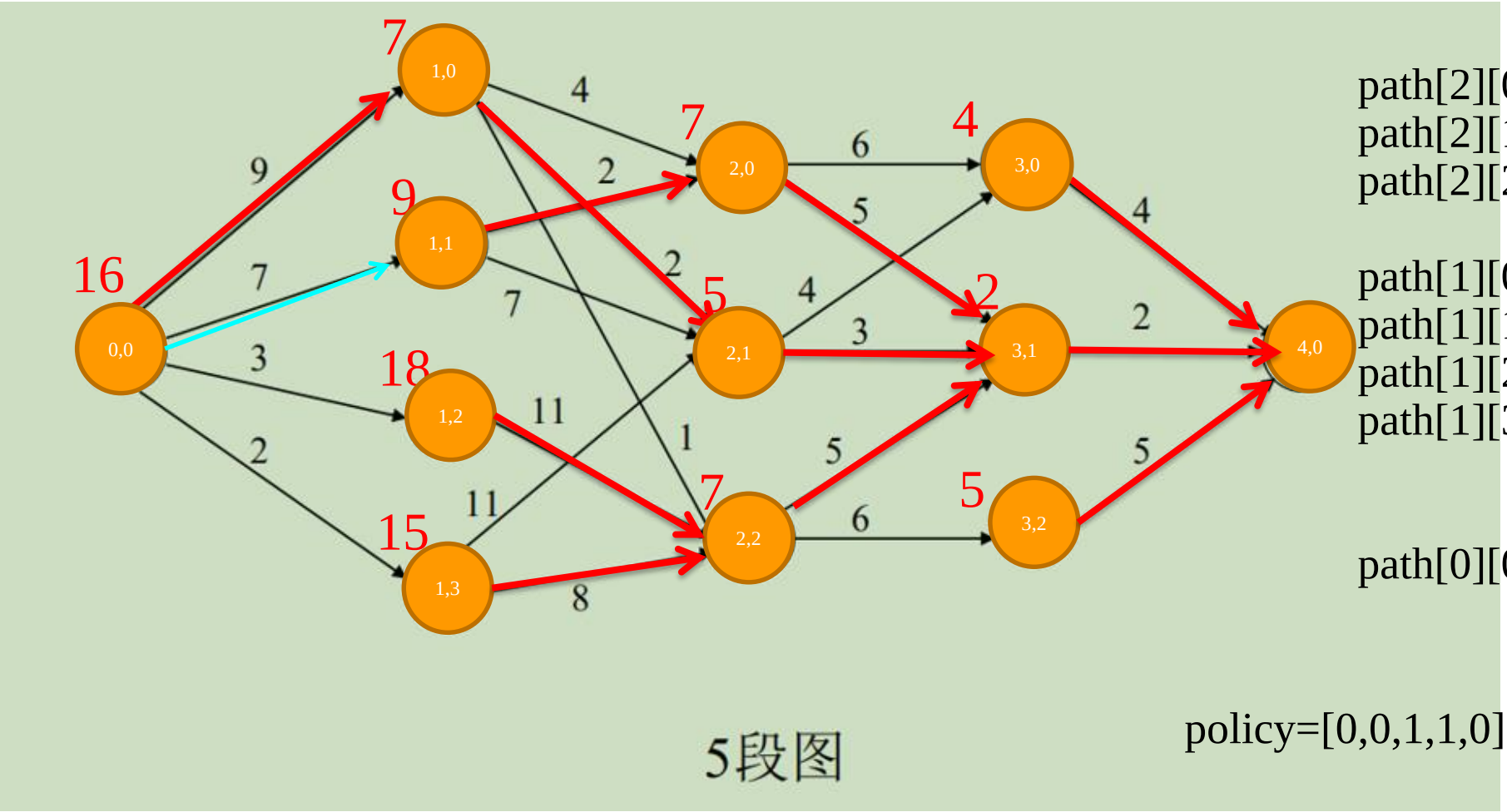
d[3][0]=4
d[3][1]=2
d[3][2]=5

d[2][0]=min{d[3][k]+w[2][k][0]}
d[2][1]=min{d[3][k]+w[2][k][1]}
d[2][2]=min{d[3][k]+w[2][k][2]}

d[1][0]=min{d[2][k]+w[1][k][0]}

General form: d[i][j]=min{d[i+1][k]+w[i][k][j]}

Multistage Decision Processes: Backward



path[2][0]=1
path[2][1]=1
path[2][2]=1

path[1][0]=1
path[1][1]=0
path[1][2]=2
path[1][3]=2

path[0][0]=0

policy=[0,0,1,1,0]

policy2=[0,1,0,1,0]

Multistage Decision Processes

Backward MDP

- 设 *path[i, j]* 是一条从V_n 到V_i中的结点 j 的最小成本路径，d*[i, j]* 是这条路径的成本。

Recursive Equation

$$d[\mathrm{i}][\mathrm{j}] = \min_{\substack{j \in V_i \\ (\mathrm{k},\mathrm{j}) \in \mathrm{E}}} \{\mathrm{d}[\mathrm{i}+1][\mathrm{k}] + \mathrm{w}[\mathrm{i}][\mathrm{k}][\mathrm{j}]\}$$

$$path[\mathrm{i}][\mathrm{j}] = \arg\min_{k} \{\mathrm{d}[\mathrm{i}+1][\mathrm{k}] + \mathrm{w}[\mathrm{i}][\mathrm{k}][\mathrm{j}]\}$$

Initialization

第1段

d[n-1,j] =

$$\begin{cases} \mathrm{w}[\mathrm{n}-1,0,\mathrm{j}] & <0,j> \in E \\ \infty & <0, j> \notin E \end{cases}$$

Multistage Decision Processes: Backward

Algorithm BMDP(structure s, int w)

```
//input 表示多段图的结构体数组 s, 以及路径矩阵 w;
//output the cost matrix d, and the path

d[n-1][0]=0; path[0][0]=0;  n=the number of stages;
for(int i=n-2;i>=0;i--)
            for(int j=0;j<s[i].no;j++){
                 min_d = d[i+1][0]+w[i][0][j];
                 min_k = 0;
                for(int k=0;k<s[i-1].no;k++){
                   d = d[i+1][k]+w[i][k][j] ;
                   if (d <=  min_d)    min_d = d;  min_k =k;}
                  d[i][j]=min_d;   path[i][j]=k;}
policy[]=0; a=0;
for(int i=0;i<n-1;i++) {
        a=path[i][a] ;  policy[i+1]=a;
}
```

Warshall's Algorithm

Definitions

Adjacent matrix

adjacent matrix $A=\{a_{ij}\}$ *of a directed graph is the boolean matrix,*

1 *in* i^{th} *row and* j^{th} *column*

↔ *a directed edge from* i^{th} *vertex to* j^{th} *vertex*

Transitive Closure

transitive closure of a directed graph with n vertices can be defined as the nxn matrix $T = \{t_{ij}\}$,

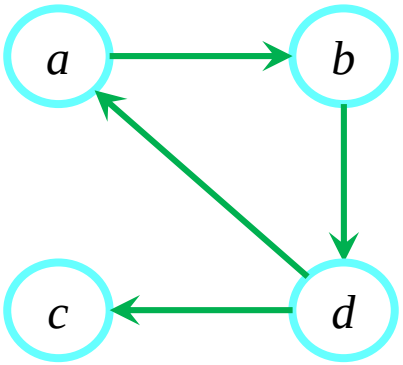
t_{ij} *in the* i^{th} *row and* j^{th} *column* = 1

↔ *if there exists a nontrivial directed path (i.e., a directed path of a positive length) from the* i^{th} *vertex to the* j^{th} *vertex;*

otherwise, $t_{ij} = 0$.

Warshall's Algorithm

- ***Example***

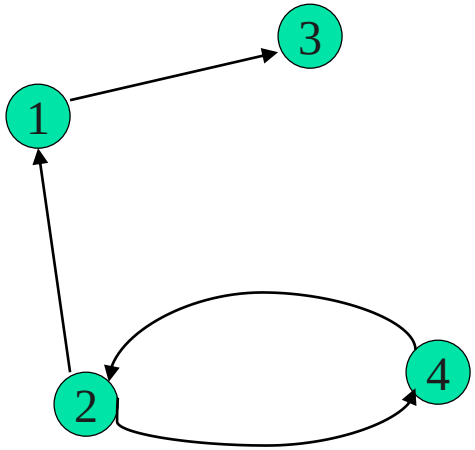


adjacent matrix

$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Transitive Closure

$$T = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$



adjacent matrix

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

Transitive Closure

$$T = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Warshall's Algorithm

Warshall's Algorithm

Idea

Use a bottom-up method to construct the transitive closure of a given digraph with n vertices, through a series of nxn boolean matrices: $R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots, R^{(n)}$

- *element* $r_{ij}^{(k)}$ *in the i* th *row and j* th *column of matrix* $R^{(k)} = 1$

  $\leftrightarrow$ *exists a directed path (of a positive length) from i* th *vertex to j* th *vertex with each intermediate vertex, if any, numbered not higher than k*

- *Each matrix provides certain information about directed paths in the digraph*
- *each subsequent matrix in the series has one more vertex to use as intermediate vertex for its paths than its predecessor matrix and hence may, but does not have to, contain more ones*

$R^{(0)}$ *is adjacent matrix,* $R^{(n)}$ *is transitive closure*

Warshall's Algorithm

Warshall's Algorithm

- *Key point: how to obtain $R^{(k)}$ from $R^{(k-1)}$?*

  $r_{ij}^{(k)} = 1$

  $\leftrightarrow$ *exists a directed path from i th vertex v_i to j th vertex v_j with each intermediate vertex numbered not higher than k*

  v_i, *a list of intermediate vertices each numbered not higher than k,* v_j **(*)**

  - ***Situation*1**, *list of intermediate vertices does not contain vertex* v_k

    $\rightarrow$ *this path from* v_i *to* v_j *has intermediate vertices numbered not higher than k*-1

    $\rightarrow$ $r_{ij}^{(k-1)} = 1$

Warshall's Algorithm

Warshall's Algorithm

- *Key point: how to obtain $R^{(k)}$ from $R^{(k-1)}$? ('cont)*
  - ***Situation2***, *list of intermediate vertices does contain k th vertex v_k*

    → *v_k occurs once in the path, (if not, we can create a new path from v_i to v_j by simply eliminating all vertices between the first and the last occurrences of v_k in it)*

    → *path * be turned into*

    v_i, vertices numbered $\leq k$-1, v_k, vertices numbered $\leq k$-1, v_j **(**)**

    - ***first part***, *there exists a path from v_i to v_k, with each intermediate vertex numbered not higher than k-1*

      → $r_{ik}^{(k-1)} = 1$

    - ***second part***, *there exists a path from v_k to v_j, with each intermediate vertex numbered not higher than k-1*

      → $r_{kj}^{(k-1)} = 1$

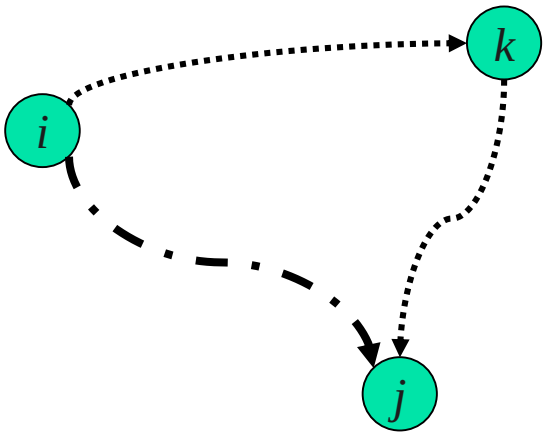
Warshall's Algorithm

Warshall's Algorithm

- *Key point: how to obtain $R^{(k)}$ from $R^{(k-1)}$? ('cont)*

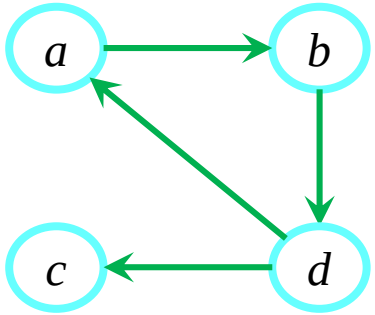
In the k^{th} stage: to determine $R^{(k)}$ is to determine if a path exists between two vertices i, j using just vertices among 1,…,k

$r_{ij}^{(k)} = 1$: { $r_{ij}^{(k-1)} = 1$ *(path using just 1 ,…,k-1)*
or
$(r_{ik}^{(k-1)} = 1$ *and* $r_{kj}^{(k-1)} = 1)$ *(path from i to k and from k to i using just 1 ,…,k-1)*



Warshall's Algorithm

- ***Example1***



$$R^{(0)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

$$R^{(1)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Warshall's Algorithm

$$R^{(2)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & \mathbf{1} \end{bmatrix} \end{array}$$

$$R^{(3)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

$$R^{(4)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} \mathbf{1} & 1 & \mathbf{1} & 1 \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Warshall's Algorithm

- ***Warshall's Algorithm***
  - *Time Efficiency ?*

Floyd's Algorithm: All pairs shortest paths

Problems

- *All pairs shortest paths problem:*

  In a weighted graph, find the distances (lengths of the shortest paths) from each vertex to all other vertices.

  Applicable to: undirected and directed weighted graphs; no negative weight.

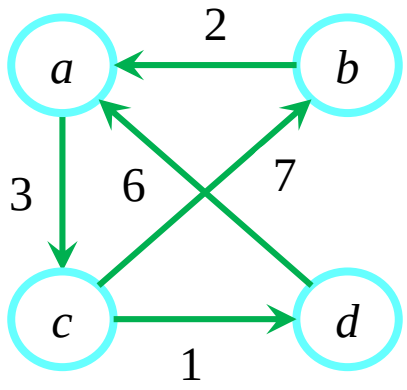
- *distance matrix:*

  *record the lengths of the shortest paths in an n*n matrix*

  d_{ij} in the i^{th} row and j^{th} column: length of the shortest path from vertex i to j.

Floyd's Algorithm

- ***Example***

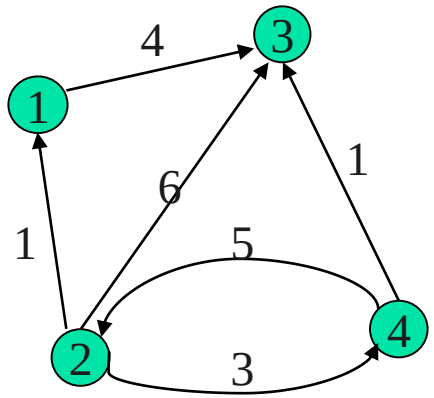


weight matrix

$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

Distance Matrix

$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$



weight matrix

$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 4 & \infty \\ 1 & 0 & 6 & 3 \\ \infty & \infty & 0 & \infty \\ \infty & 5 & 1 & 0 \end{bmatrix} \end{array}$$

Distance Matrix

$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 4 & \infty \\ 1 & 0 & \mathbf{4} & 3 \\ \infty & \infty & 0 & \infty \\ \mathbf{6} & 5 & 1 & 0 \end{bmatrix} \end{array}$$

Floyd's Algorithm: All pairs shortest paths

Floyd's Algorithm

Idea

find the distance matrix of a weighted graph with n vertices through series of nxn matrices $D^{(0)}$, $D^{(1)}$, …, $D^{(n)}$

- *element $d_{ij}^{(k)}$ in the i th row and j th column of matrix $D^{(k)}$*

  ↔ equals the length of the shortest path among all paths from the i th vertex to j th vertex, with each intermediate vertex, if any, numbered not higher than k

- *Each matrix provides the lengths of shortest paths with certain constraints*
- *$D^{(0)}$ is weight matrix, not allow any intermediate vertices in its paths*
- *$D^{(n)}$ is distance matrix, contains the lengths of the shortest paths among all paths that can use all n vertices as intermediate*

Floyd's Algorithm: All pairs shortest paths

Floyd's Algorithm

- *Key point: how to obtain $D^{(k)}$ from $D^{(k-1)}$?*

  $d_{ij}^{(k)}$

  ↔ *the length of the shortest path among all paths from the i th vertex to j th vertex, with each intermediate vertex, if any, numbered not higher than k*

  v_i, *a list of intermediate vertices each numbered not higher than k,* v_j **(*)**

  - ***Situation*1**, *list of intermediate vertices does not contain vertex* v_k

    → *this path from* v_i *to* v_j *has intermediate vertices numbered not higher than k*-1

    → $d_{ij}^{(k-1)}$

Floyd's Algorithm: All pairs shortest paths

Floyd's Algorithm

Key point: how to obtain $D^{(k)}$ from $D^{(k-1)}$? ('cont)

- **Situation2**, *list of intermediate vertices does contain k th vertex* v_k

  → v_k *occurs once in the path,* (*visiting* v_k *more than once, can only increase the path's length; and we limit our discussion to the graph not contain a cycle of a negative length*)

  → *path * be turned into*

  v_i, *vertices numbered* $\leq k$-1, v_k, *vertices numbered* $\leq k$-1, v_j **(**)**

  - ***first part***, *a path from* v_i *to* v_k, *with each intermediate vertex numbered not higher than k*-1, *the shortest among these is*

    → $d_{ik}^{(k-1)}$

  - ***second part***, *a path from* v_k *to* v_j, *with each intermediate vertex numbered not higher than k*-1, *the shortest among these*

    → $d_{kj}^{(k-1)}$

Floyd's Algorithm: All pairs shortest paths

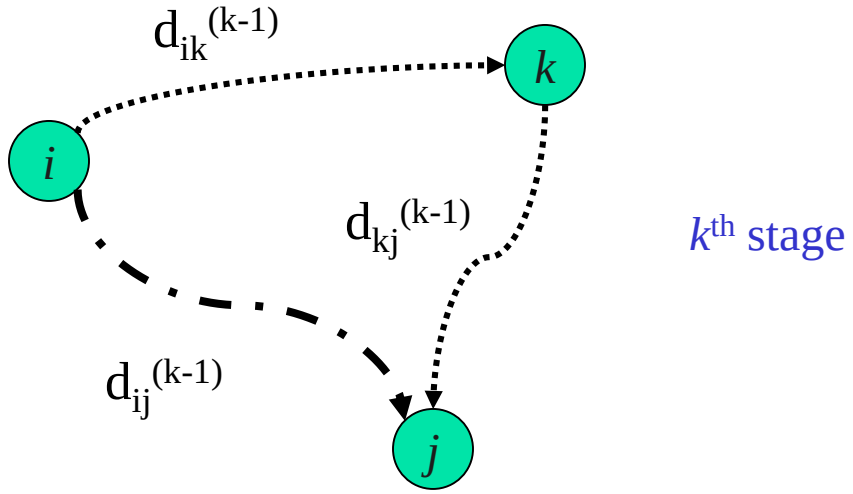
Floyd's Algorithm

- *Key point: how to obtain $D^{(k)}$ from $D^{(k-1)}$? ('cont)*

  $D^{(k)}$: allow 1, 2, …, k to be intermediate vertices.

  In the k^{th} stage, determine whether the introduction of k as a new eligible intermediate vertex will bring about a shorter path from i to j.

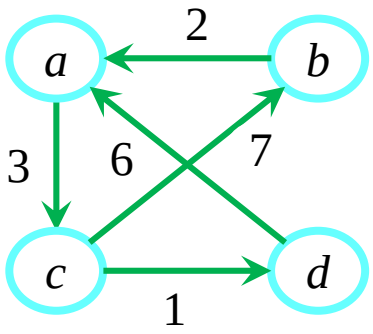
$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1, d_{ij}^{(0)} = w_{ij}$$



k^{th} stage

Floyd's Algorithm: All pairs shortest paths

- ***Example1***



$$D^{(0)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

$$D^{(1)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{bmatrix} \end{array}$$

Floyd's Algorithm: *All pairs shortest paths*

$$D^{(2)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array}$$

$$D^{(3)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{bmatrix} \end{array}$$

$$D^{(4)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ \mathbf{7} & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

Content

1. Introduction of Dynamic Programming
2. Applications of DP
   - ① Computing binomial coefficients
   - ② the longest common subsequence (LCS)
   - ③ Dynamic Matrix Multiplication
   - ④ 0-1 Knapsack Problem
   - ⑤ Multistage Decision Processes
   - ⑥ Warshall's algorithm for transitive closure
   - ⑦ Floyd's algorithms for all-pairs shortest paths
3. **Examples of Checking the Principle of Optimality**

Checking the Principle of Optimality

- 例1：设G是一个有向加权图，则G从顶点i到顶点j之间的最短路径问题满足最优性原理。

  证明：（反证）

  设$i\sim i_p\sim i_q\sim j$是一条最短路径，但其中子路径$i_p\sim i_q\sim j$不是最优的，

  假设最优的路径为$i_p\sim i_q'\sim j$

  则我们重新构造一条路径：$i\sim i_p\sim i_q'\sim j$

  显然该路径长度小于$i\sim i_p\sim i_q\sim j$，与$i\sim i_p\sim i_q\sim j$是顶点i到顶点j的最短路径相矛盾.

  所以，原问题满足最优性原理。 □

Checking the Principle of Optimality

- 例2：0-1背包问题Knap(1,n,c)满足最优性原理

形式化表达为数学表达式

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

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \le c \\ x_i \in \{0,1\} \quad 1 \le i \le n \end{cases} \xrightarrow{记} \Rightarrow \quad Knap\ (1,n,c)$$

证明：设$(y_1,y_2,\ldots,y_n)$是Knap(1,n,c)的一个最优解，下证$(y_2,\ldots,y_n)$是Knap(2,n,c-w_1y_1)子问题的一个最优解。

若不然，设$(z_2,\ldots,z_n)$是Knap(2,n,c-w_1y_1)的最优解，因此有

$$\sum_{i=2}^{n} v_i z_i > \sum_{i=2}^{n} v_i y_i \text{ 且 } \sum_{i=2}^{n} w_i z_i \le c - w_1 y_1$$

$$\Rightarrow \quad v_1 y_1 + \sum_{i=2}^{n} v_i z_i > \sum_{i=1}^{n} v_i y_i \text{ 又有 } w_1 y_1 + \sum_{i=2}^{n} w_i z_i \le c$$

说明$(y_1,z_2,\ldots,z_n)$是Knap(1,n,c)的一个更优解，矛盾。 □

Checking the Principle of Optimality

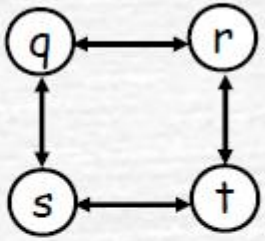
- 例3：最长路径问题不满足最优性原理。

证明：

p: q→r→t是q到t的最长路径，

而q→r的最长路径是q→s→t→r

r→t的最长路径是r→q→s→t



但q→r和r→t的最长路径合起来并不是q到t的最长路径。所以，原问题并不满足最优性原理。 □

注：因为q→r和r→t的子问题都共享路径s→t，组合成原问题解时，有重复的路径对原问题是不允许的。

Summary

设计技巧

- 动态规划的设计技巧：阶段的划分、状态的表示和存储表的设计；
- 在动态规划的设计过程中，阶段的划分和状态的表示是其中重要的两步，这两步会直接影响该问题的计算复杂性和存储表设计，有时候阶段划分或状态表示的不合理还会使得动态规划法不适用。
- 记忆型递归——动态规划的变种
  - 每个子问题的解对应一表项；
  - 每表项初值为一特殊值，表示尚未填入；
  - 递归时，第一次遇到子问题进行计算并填表，以后查表取值；

Summary

存在问题

- 问题的阶段划分和状态表示，需要具体问题具体分析，没有一个清晰明朗的方法；
- 空间溢出的问题，是动态规划解决问题时一个普遍遇到的问题；
  - 动态规划需要很大的空间以存储中间产生的结果，这样可以使包含同一个子问题的所有问题共用一个子问题解，从而体现动态规划的优越性，但这是以牺牲空间为代价的，为了有效地访问已有结果，数据也不易压缩存储，因而空间矛盾是比较突出的。