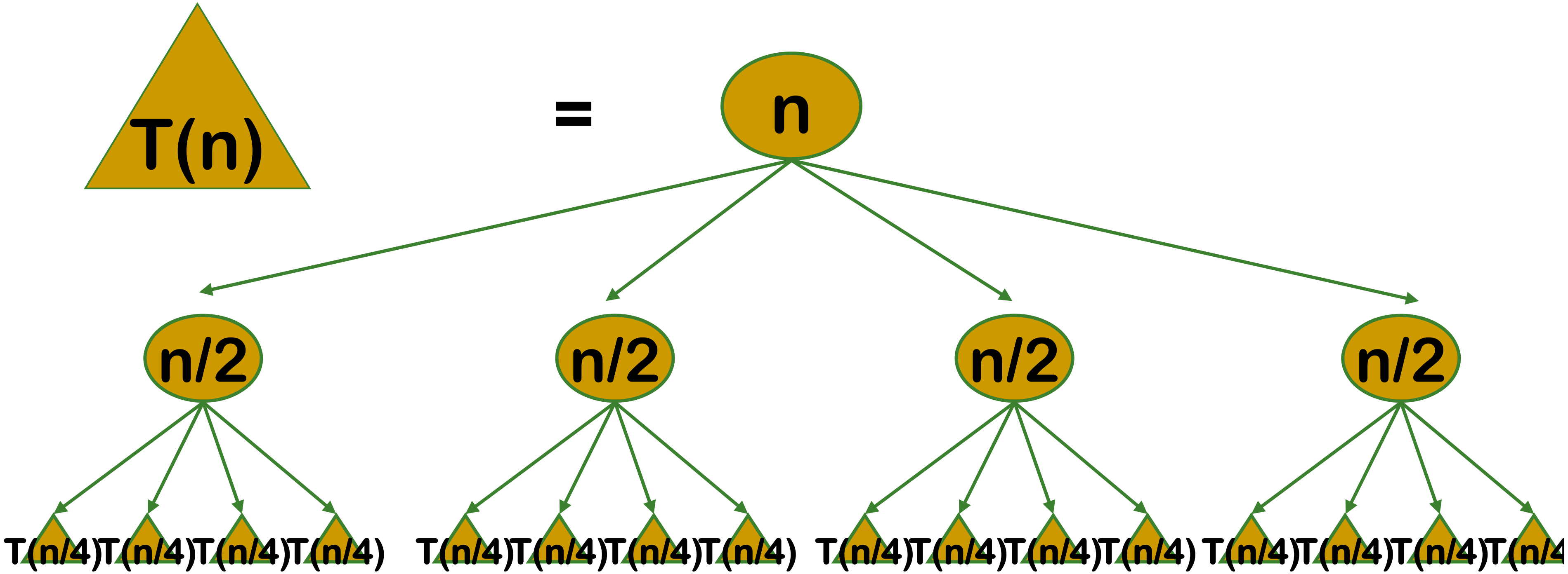
Analysis and Design of Algorithms

Chapter 8: Dynamic Programming

School of Software Engineering © Yanling Xu

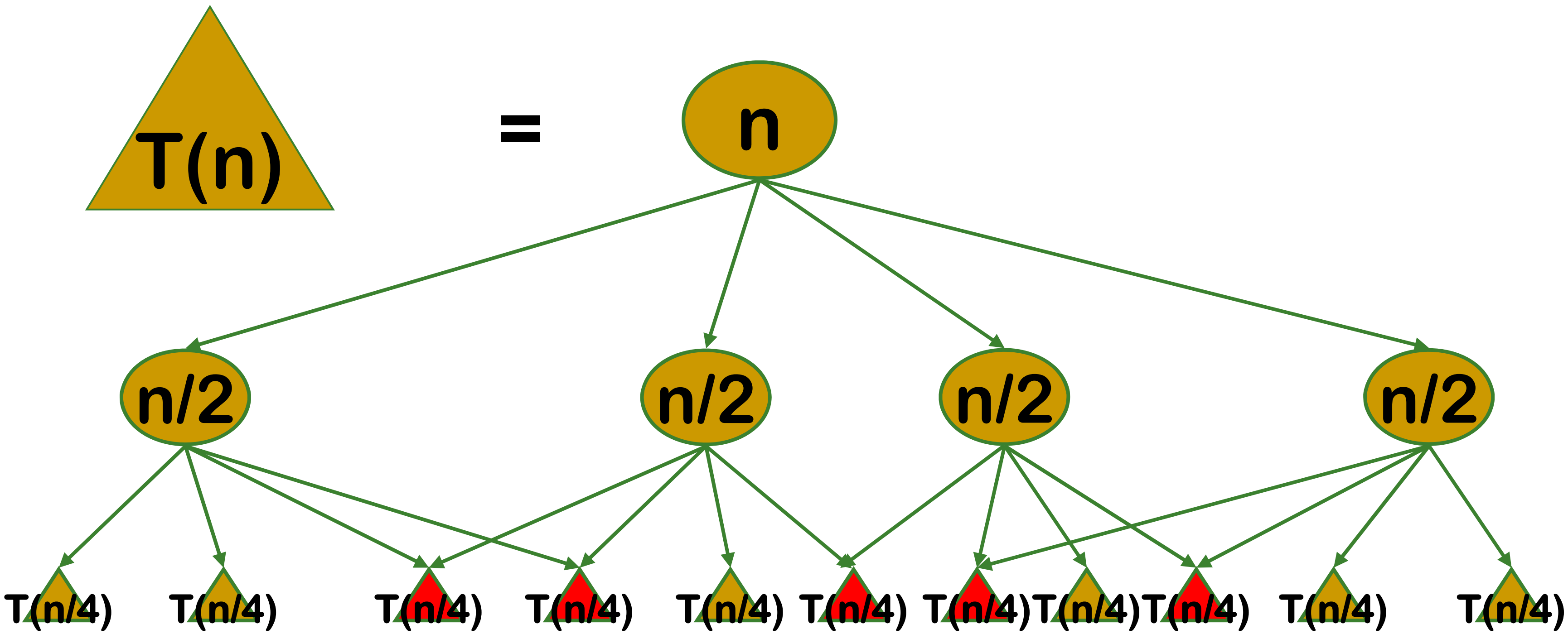
Dynamic Programming

- *Main idea*



divide-and-conquer

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*

Dynamic Programming

Main idea

- *solve several smaller (overlapping) subproblems*
- *record solutions in a table so that each subproblem is only solved once*
- *final state of the table will be (or contain) solution*
- *Problem solved*
  - ***Solution can be expressed in a recursive way***
  - ***Sub-problems occur repeatedly***
  - ***Subsequence of optimal solution is an optimal solution to the sub-problem***

Dynamic Programming

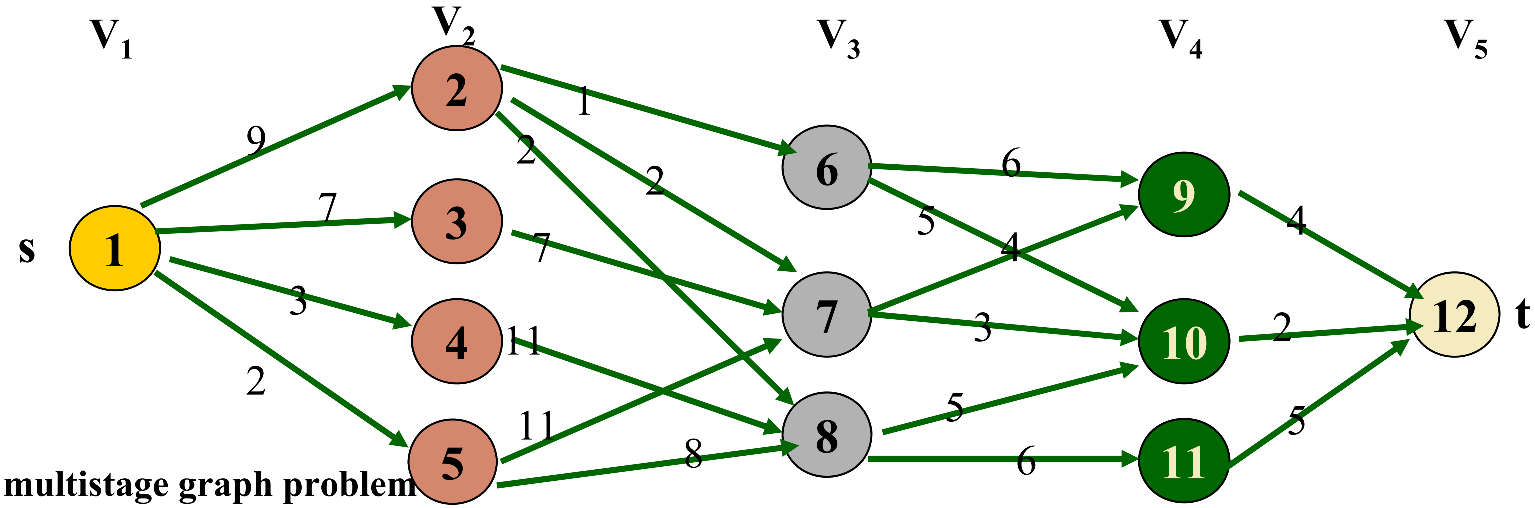
Dynamic programming vs. divide-and-conquer

- *dividing a problem into small subproblems*
  - DP: partition a problem into overlapping subproblems
  - D&C: partition the problem into independent subproblems
- *store and not store solutions to subproblems*
  - DP: solves every subsubproblem just ONCE and then saves its answer in a table,
  - D&C: repeatedly solving the common subproblems

Dynamic Programming

DP and MDP

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



multistage graph problem

Dynamic Programming

application of dynamic programming

- *Computing binomial coefficients*
- *Compute the longest common subsequence*
- *Compute the shortest common supersquence*
- *Warshall's algorithm for transitive closure*
- *Floyd's algorithms for all-pairs shortest paths*
- *Some instances of difficult discrete optimization problems:*
- *knapsack*

Dynamic Programming

- ***Frame***

  - *Characterize the structure of an optimal solution*
  - *Recursively define the value of an optimal solution*
  - *Compute the value of an optimal solution in a bottom-up fashion*
  - *Construct an optimal solution from computed information*

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,* $C(n, k) = C(n\text{-}1, k\text{-}1) + C(n\text{-}1, k)$*, 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							$C(n\text{-}1,k\text{-}1)$	$C(n\text{-}1,k)$
n	1								$C(n,k)$

Computing Binomial Coefficients

- ***Dynamic Programming 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)*

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

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

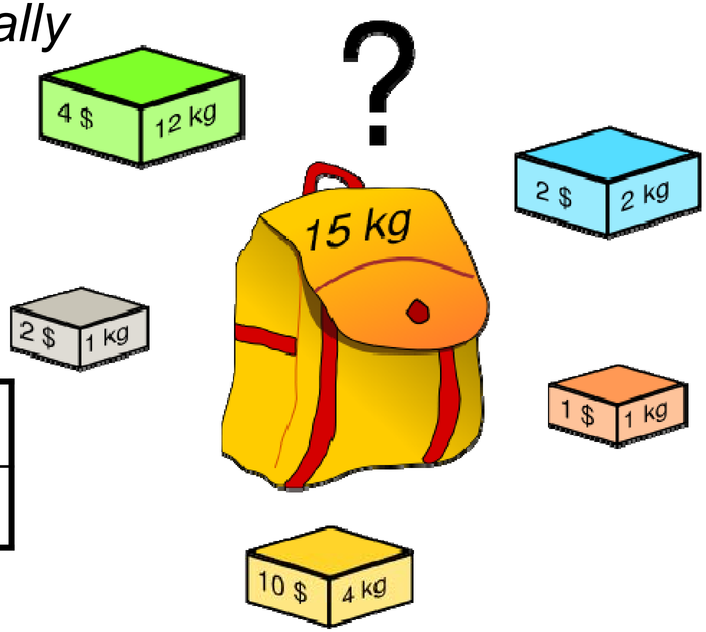
0-1 *Knapsack Problem*

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 item* 1 to *n*
  - *to derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller subinstances*.
  - *a subinstance of the first i items*, $0 \le i < n$, *of known weights* $w_1, \ldots, w_i$ *and values* $v_1, \ldots, v_i$ *and 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* $V[i\text{-}1,j]$
    - *among the subsets that do include the i^{th} item (hence, $j\text{-}w_i \geq 0$), 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 $j\text{-}w_i$, the value of such an optimal subset is* $v_i + V[i\text{-}1, j\text{-}w_i]$

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

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

0-1 *Knapsack Problem*

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

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

- **proof by contradiction:**

suppose $(z_2,z_3,\ldots,z_n)$ is the optimal solution for above subproblem, and $(y_2,y_3,\ldots,y_n)$ 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 $(y_1,z_2,\ldots,z_n)$ is a **more** optimal solution for the original 0-1 knapsack problem, and $(y_1,y_2,\ldots,y_n)$ 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*

- *optimal value* $m[i][j]$

T(n) = O(nc)

```
void Knapsack(Type v, int w, int c, int n, Type **m )
{
    int jMax=min(w[n]-1,c);
    for(int j=0; j<=jMax; j++)  m[n][j]=0; //由于j比w[n]小，第n号物品不放入
    for(j=w[n] ; j<=c; j++)      m[n][j]=v[n];     //表示第n号物品放入
    for(int i=n-1; i>1; i--){      //利用递归函数，从后往前计算
            jMax=min(w[i]-1,c);
            for(j=0; j<= jMax; j++)     m[i][j]=m[i+1][j];
            for(j=w[i]; j<=c; j++)      m[i][j]=max(m[i+1][j], m[i+1][j-w[i]]+v[i]); }
    m[1][c]=m[2][c]; //第一行不计算，减少计算量
    if(c>=w[1])    m[1][c]=max(m[1][c],m[2][c-w[1]]+v[1]);
}
```

0-1 *Knapsack Problem*

- *optimal solution* ***O(n)***

```
void Traceback (Type **m, int w, int c, int n, int x)
{
    for ( int i=1; i < n; i++)
        if (m[i][c]==m[i+1][c]) x[i]=0;
        else {  x[i]=1;
                c-=w[i];  }
    x[n]=(m[n][c]) ? 1:0;
}
```

0-1 *Knapsack Problem*

Ex. recurrence from the first item

		0	$j\text{-}w_i$	……	j	W
	0	0	0		0	0
w_i v_i	i-1	0	$V[i\text{-}1, j\text{-}w_1]$ $+v_i$		$V[i\text{-}1, j]$	
	i	0	0		$V[i, j]$	
		0				
	n	0				目标

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
$w_1=2.$ $v_1=12$	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	$\boxed{37}$

$V(i\text{-}1, j\text{-}w_1)+v_1$		$V(i\text{-}1, j)$
$V(i\text{-}1, j\text{-}w_2)+v_2$	$V(i\text{-}1, j)$	$V(i, j)$
	$V(i, j)$	

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

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

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

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

$V[1,2] \neq V[0,2]$, *item* 1 *is included in an optimal subset . So, optimal solution is* {1,1,0,1}, *i.e. item{1,2,4}*

0-1 *Knapsack Problem*

Ex. recurrence from the last item

		0	$j\text{-}w_i$	……	j	W
	1	0				目标
w_i v_i	i	0			$m[i, j]$	
	$i+1$	0	$m[i+1, j\text{-}w_1]$	$+v_i$	$m[i+1, j]$	
		0				
	n	0				

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
w_3=3 v_3=20	3	0	0	15	20	20	35
w_4=2. v_4=15	4	0	0	15	15	15	15

	$m(i, j)$	
$m(i+1, j\text{-}w_2)+v_2$	$m(i+1, j)$	$m(i, j)$
$m(i+1, j\text{-}w_3)+v_3$		$m(i+1, j)$

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

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

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

$m[4,2] \neq 0$, *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*

Memory functions for 0-1 Knapsack Problem

- *dynamic programming deals with problems whose solutions satisfy a recurrence relation with overlapping subproblems*

  → top-down approach to such a recurrence solves common subproblems more than once, and hence inefficient

- *bottom-up dynamic programming fills a table with solutions to **all** smaller subproblems, each of them solved only once*

  → solutions to some of the subproblems are not necessary for getting a solution to the given problem

0-1 *Knapsack Problem*

- *Memory functions*
  - *only solve necessary subproblems, only once*
  - *top-down manner*
  - *maintains a table as in bottom-up dynamic programming,*
    - *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

0-1 *Knapsack Problem*

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 指出先考虑的物品数量，一个非负整数 j 指出了背包的承重量
//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 V[0..n,0..W]作为全局变量，除了行 0 和
  列 0 用 0 初始化以外，V 的所有单元都用-1 做初始化。
if V[i, j] < 0
    if j < Weights[i]
        value ← MFKnapsack(i − 1, j)
    else
        value ← max(MFKnapsack(i − 1), j),
                    Value[i] + MFKnapsack(i − 1, j − Weights[i]))
    V[i, j] ← value
return V[i, j]
```

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
$w_2 = 1, v_2 = 10$	2	0	–	12	22	–	22
$w_3 = 3, v_3 = 20$	3	0	–	–	22	–	32
$w_4 = 2, v_4 = 15$	4	0	–	–	–	–	37

0-1 *Knapsack Problem*

算法改进 — 阶跃法

- 考察0-1背包问题的一个具体实例

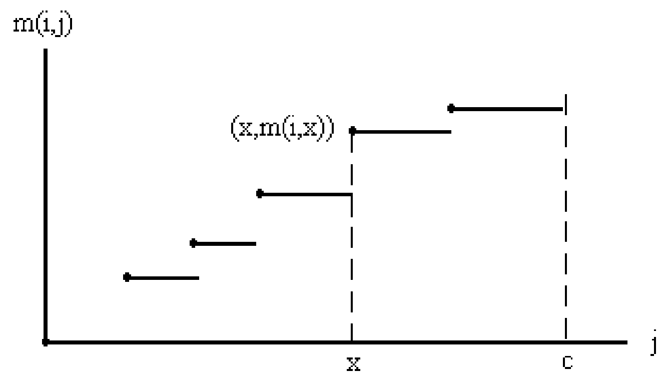
  $n = 5, W = 10, w = \{2,2,6,5,4\}, v = \{6,3,5,4,6\}$

  由$m(i,j)$的递归式，当$i = 5$时， $m(5,j)=\begin{cases}6 & j \ge 4\\ 0 & 0 \le j < 4\end{cases}$

- 由$m(i,j)$的递归式容易证明，在一般情况下，对每一个确定的$i(1 \le i \le n)$，函数$m(i,j)$是关于变量j的阶梯状单调不减函数。

- 跳跃点是这一类函数的描述特征。

  在一般情况下，函数$m(i,j)$由其全部跳跃点唯一确定。

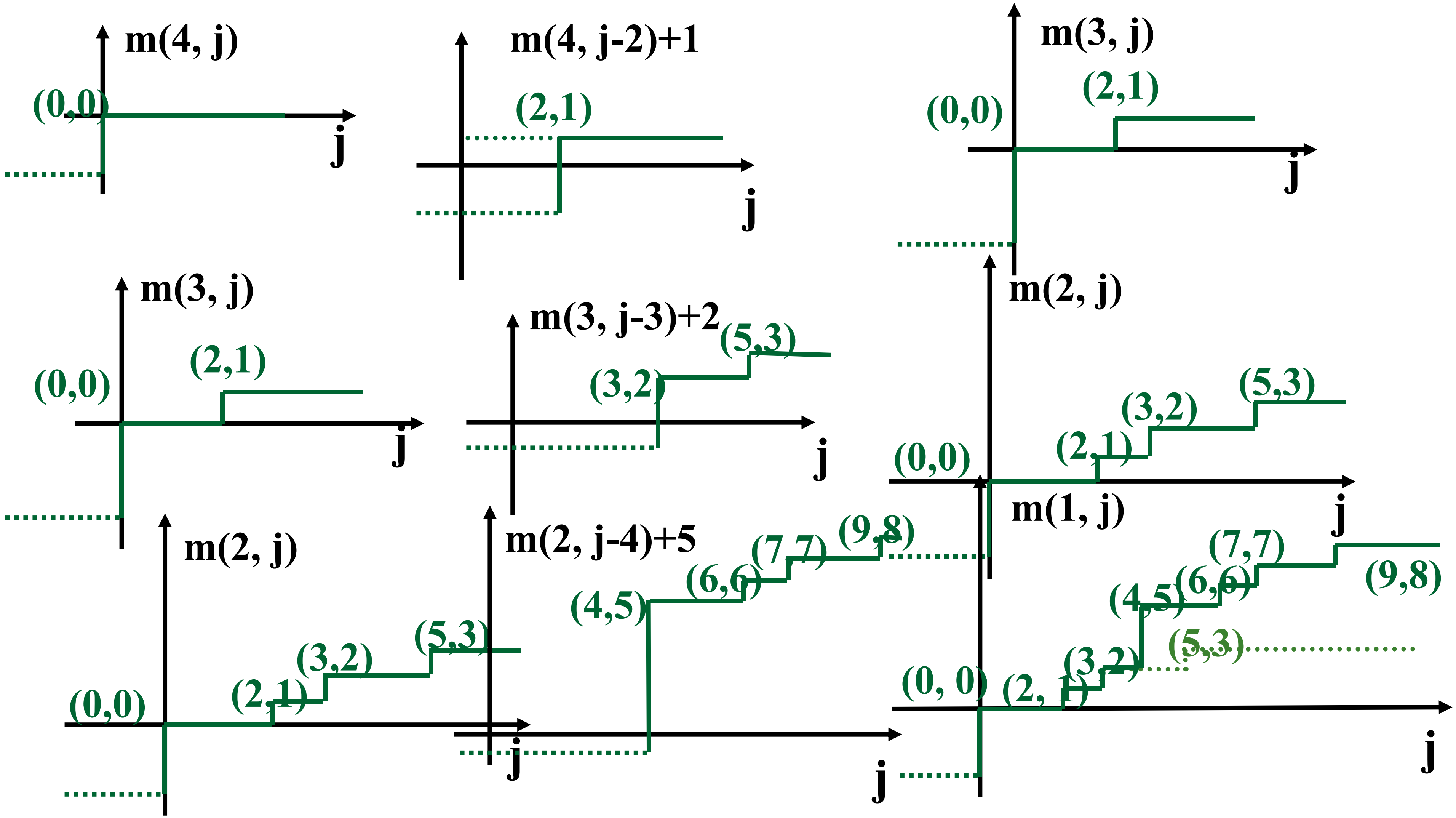


0-1 *Knapsack Problem*

- $p[i]$
- $p[i+1]$
- $q[i+1]=p[i+1]\oplus(w_i,v_i)=\{(j+w_i,m(i,j)+v_i)|(j,m(i,j))\in p[i+1]\}$
- $p[i+1]\cup q[i+1]$
- 并清除其中的受控跳跃点

0-1 *Knapsack Problem*

- ***Example:*** n=3，W=6，w={4，3，2}，v={5，2，1}



0-1 *Knapsack Problem*

- ***Example:*** **n=5，W=10，w={2，2，6，5，4}，v={6，3，5，4，6}**

初始时p[6]={(0,0)}，(w_5,v_5)=(4,6)。因此，q[6]=p[6]⊕(w_5,v_5)={(4,6)}。

p[5]= p[6]∪q[6]={({(0,0),(4,6)}。

q[5]=p[5]⊕(w_4,v_4)={(5,4),(9,10)}。

从跳跃点集p[5]与q[5]的并集p[5]∪q[5]={(0,0),(4,6),(5,4),(9,10)}中看到跳跃点(5,4)受控于跳跃点(4,6)。将受控跳跃点(5,4)清除后，得到

p[4]={(0,0),(4,6),(9,10)}

q[4]=p[4]⊕(6，5)={(6，5)，(10，11)}

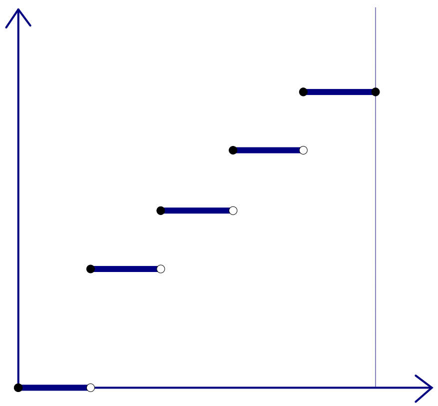
p[3]={(0，0)，(4，6)，(9，10)，(10，11)}

q[3]=p[3]⊕(2，3)={(2，3)，(6，9)}

p[2]={(0，0)，(2，3)，(4，6)，(6，9)，(9，10)，(10，11)}

q[2]=p[2]⊕(2，6)={(2，6)，(4，9)，(6，12)，(8，15)}

p[1]={(0，0)，(2，6)，(4，9)，(6，12)，(8，15)}



p[1]的最后的那个跳跃点(8,15)给出所求的最优值为m(1,W)=15。

0-1 *Knapsack Problem*

- 0-1背包问题的改进的动态规划算法 ***O(n)***

```
void Knapsack(int n, Type c, Type v[ ], Type w[ ], Type **p, int x[ ] )
{   int *head=new int [n+2]；
    head[n+1]=0；p[0][0]=0；p[0][1]=0；
    int left=0, right=0, next=1；head[n]=1；
    for(int i=n； i>=1；i--){ int k=left；
       for (int j=left；j<=right； j++){
         if (p[j][0]+w[i]>c) break；
         Type y=p[j][0]+w[i]；
              m=p[j][1]+v[i];
         while(k <= right&&p[k][0]<y){
            p[next][0]=p[k][0]；
            p[next++][1]=p[k++][1]； }
         if ( k<=right&&p[k][0]==y){
             if(m<p[k][1]) m=p[k][1]； k++； }
         if ( m > p[next-1][1]) { p[next][0]=y；p[next++][1]=m；}
         while(k<=right&&p[k][1]<=p[next-1][1]) k++；           }
       while(k<=right){ p[next][0]=p[k][0]；p[next++][1]p[k++][1]；}
       left=right+1； right=next-1； head[i-1]=next；     }
    Traceback(n, w, v, p, head, x)；return p[next-1][1]；     }
```

0-1 *Knapsack Problem*

```
void Traceback(int n, Type w[], Type v[], Type**p, int*head, int x[])
{
    Type j = p[head[0]-1][0],
          m=p[head[0]-1][1]；
    for( int i=1 ； i<=n ； i++){
       x[i]=0 ；
       for ( int k=head[i+1] ； k<=head[i]-1 ； k++){
         if (p[k][0]+w[i]==j&&p[k][1]+v[i]==m){
            x[i]=1 ；
            j=p[k][0] ；
            m=p[k][1] ；
            break ；
            }
         }
    }
}
```

0-1 *Knapsack Problem*

- 算法复杂度分析
  - 上述算法的主要计算量在于计算跳跃点集$p[i]$$(1\le i\le n)$。由于$q[i+1]=p[i+1]\oplus(w_i，v_i)$，故计算$q[i+1]$需要O(|$p[i+1]$|)计算时间。合并$p[i+1]$和$q[i+1]$并清除受控跳跃点也需要O(|$p[i+1]$|)计算时间。从跳跃点集$p[i]$的定义可以看出，$p[i]$中的跳跃点相应于$x_i,\ldots,x_n$的0/1赋值。因此，$p[i]$中跳跃点个数不超过$2^{n-i+1}$。由此可见，算法计算跳跃点集$p[i]$所花费的计算时间为

$$O\left(\sum_{i=2}^{n}|\,p[i+1]\,|\right)=O\left(\sum_{i=2}^{n}2^{n-i}\right)=O\left(2^{n}\right)$$

  - 从而，改进后算法的计算时间复杂性为$\boldsymbol{O(2^n)}$。当所给物品的重量$w_i(1\le i\le n)$是整数时，$|p[i]|\le c+1$，$(1\le i\le n)$。在这种情况下，改进后算法的计算时间复杂性为$\boldsymbol{O(\min\{nc,2^n\})}$。

Longest Common Subsequence (LCS)

Definition

subsequence

- *A subsequence of a sequence S is obtained by deleting zero or more symbols from S. For example, the following are all subsequences of "president": pred, sdn, predent.*
- *Sequence* $Z=\{z_1,z_2,\ldots,z_k\}$ *is subsequence of* $X=\{x_1,x_2,\ldots,x_m\}$

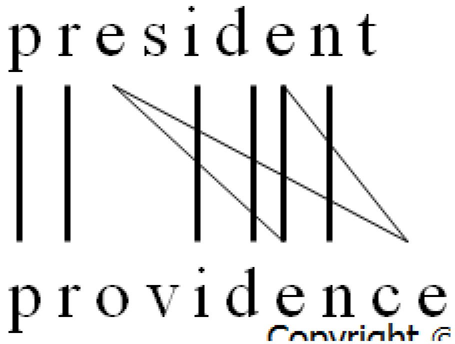
  $\leftrightarrow$ *there exits an increasing sequence* $\{i_1,i_2,\ldots,i_k\}$, *with* $z_j=x_{i_j}$ *for all* $j = 1,2,\ldots,k$.

- E.g. *sequence* Z={B，C，D，B} *is subsequence of* X={A，B，C，B，D，A，B} *with increasing sequence* {2，3，5，7}。

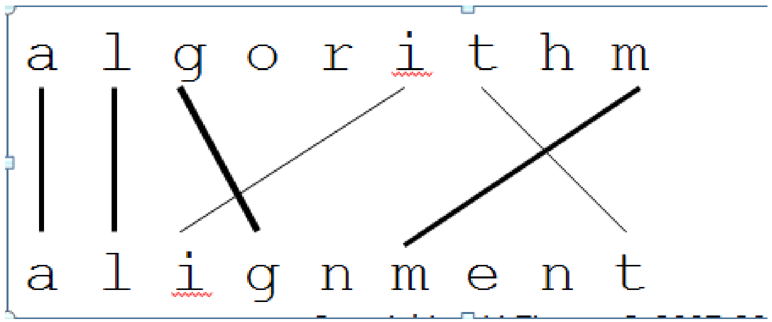
Longest Common Subsequence (LCS)

- Common subsequence
  - *sequence Z is common subsequence of X and Y : Z is subsequence of X and Z is subsequence of Y*
  - *Longest common subsequence problem is to find a maximum length common subsequence between two sequences.*

- *Example:*
  Sequence 1: president
  Sequence 2: providence
  Its LCS : priden.



- *Example:*
  Sequence 1: algorithm
  Sequence 2: alignment
  One of its LCS is algm.



Longest Common Subsequence (LCS)

Dynamic Programming to solve LCS

- *Problem*

  find the longest common subsequence for $X=\{x_1,x_2,\ldots,x_m\}$ and $Y=\{y_1,y_2,\ldots,y_n\}$

- *Characterize the structure of an optimal solution to LCS*

  for sequence $X=\{x_1,x_2,\ldots,x_m\}$ *and* $Y=\{y_1,y_2,\ldots,y_n\}$*, their longest common subsequence is* $Z=\{z_1,z_2,\ldots,z_k\}$ *, then*

  (1) *if* $x_m=y_n$*, then* $z_k=x_m=y_n$*, and* Z_{k-1} *is LCS for* X_{m-1} *and* Y_{n-1}*;*

  (2) *if* $x_m \neq y_n$ *and* $z_k \neq x_m$*, then Z is LCS for* X_{m-1} *and Y;*

  (3) *if* $x_m \neq y_n$ *and* $z_k \neq y_n$*, then Z is LCS for X and* Y_{n-1}*.*

  ➔ *the longest common subsequence problem has the* ***principle of optimality***

Longest Common Subsequence (LCS)

- *Recursively define the value of an optimal solution*
  - *Based on the **principle of optimality of** longest common subsequence problem, we can define the recursive structure of its subproblems,*
    - *if $x_m=y_n$, find LCS for X_{m-1} and Y_{n-1}, then add $x_m(=y_n)$ to get LCS for X_m and Y_n*
    - *if $x_m \neq y_n$, solve two subproblems,*
      - *LCS for X_{m-1} and Y_n*
      - *LCS for X_m and Y_{n-1}*

      then the longer one between the above two, is the LCS for X_m and Y_n

➔ *the longest common subsequence problem has the **overlapping subproblems***

- ✓ *overlapping subproblem between subproblem LCS for X_{m-1} and Y_n, and subproblem LCS for X_m and Y_{n-1} is subproblem LCS for $X_{m-1}$and Y_{n-1}*

Longest Common Subsequence (LCS)

- *Recurrence equation*:

  c[*i*][*j*]: *length of the LCS for* $X_i = \{x_1, x_2, \ldots, x_i\}$ *and* $Y_j = \{y_1, y_2, \ldots, y_j\}$

  - *if* i=0 or j=0, *null sequence is the Longest Common Subsequence for* X_i *and* Y_j, *so* c[*i*][*j*]=0.
  - c[*i*][*j*] *can be computed as follows:*

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

Longest Common Subsequence (LCS)

- *Compute the value of optimal solution in bottom-up fashion*

```
void LCSLength(int m， int n， char *x， char *y， int **c， int **b)
{// c[i][j]记录Xi和Yj 的最长公共子序列的长度; b[i][j]记录是由哪个子问题的解得到的
    int i， j;
    for (i = 1; i <= m; i++) c[i][0] = 0;
    for (i = 1; 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; }      // ←
        }
}
```

$T(n) = \theta(mn)$

Longest Common Subsequence (LCS)

- *Construct LCS from computed information*

*using array **b** computed in* LCSLength, *we can easily construct LCS for* X_m *and* Y_n, *and begin with* $b[m][n]$,

- *if* b[i][j]=1, *LCS for* X_i *and* Y_j *could be achieved by adding* x_i *in the end of* X_{i-1} *and* Y_{j-1}
- *if* b[i][j]=2, *LCS for* X_i and Y_j *could be achieved by LCS for* X_{i-1} *and* Y_j
- *if* b[i][j]=3, *LCS for* X_i *and* Y_j *could be achieved by LCS for* X_i *and* Y_{j-1}

```
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);
}
```

$T(n) = O(m+n)$

Longest Common Subsequence (LCS)

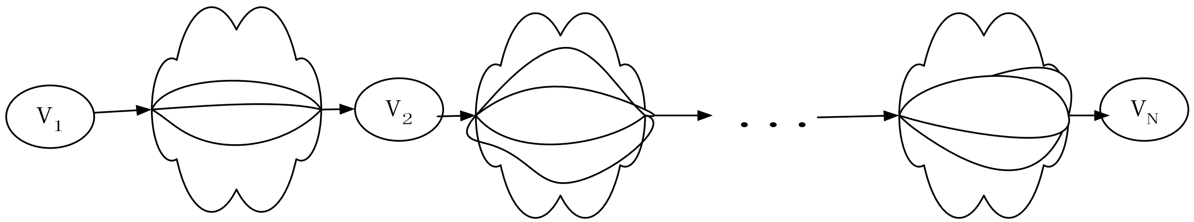
Example

i	*j* *0*	*1* *p*	*2* *r*	*3* *o*	*4* *v*	*5* *i*	*6* *d*	*7* *e*	*8* *n*	*9* *c*	*10* *e*
0	0	0	0	0	0	0	0	0	0	0	0
1 *p*	0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
2 *r*	0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2
3 *e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
4 *s*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3
5 *i*	0	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3	← 3	↑ 3	↑ 3	↑ 3	↑ 3
6 *d*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↖ 4	← 4	← 4	← 4	← 4
7 *e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↖ 5	← 5	← 5	↖ 5
8 *n*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↖ 6	← 6	← 6
9 *t*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 6	↑ 6

Multistage Decision Processes

多阶段决策问题

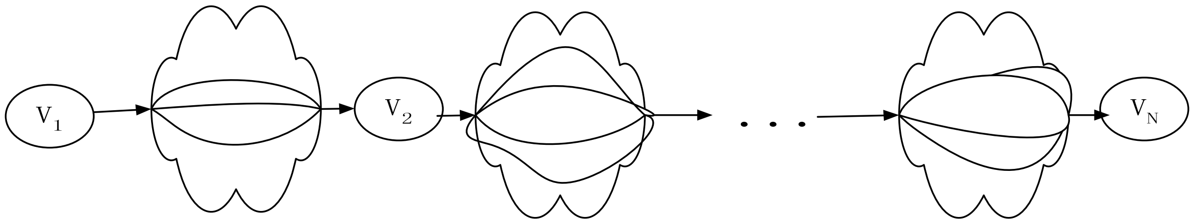
- **多阶段决策过程** multistep decision process
  - 问题的活动过程分为若干相互联系的阶段
  - 在每一个阶段都要做出决策，这决策过程称为多阶段决策过程
  - 任一阶段 i 以后的行为仅依赖于i 阶段的过程状态，而与i 阶段之前的过程如何达到这种状态的方式无关



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 Processes

多阶段决策过程的求解策略

枚举法

- 穷举可能的决策序列，从中选取可以获得最优解的决策序列

动态规划

- 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，创立了解决这类过程优化问题的新方法——动态规划。
- 动态规划是运筹学的一个分支，是求解决策过程最优化的数学方法。

Multistage Decision Processes

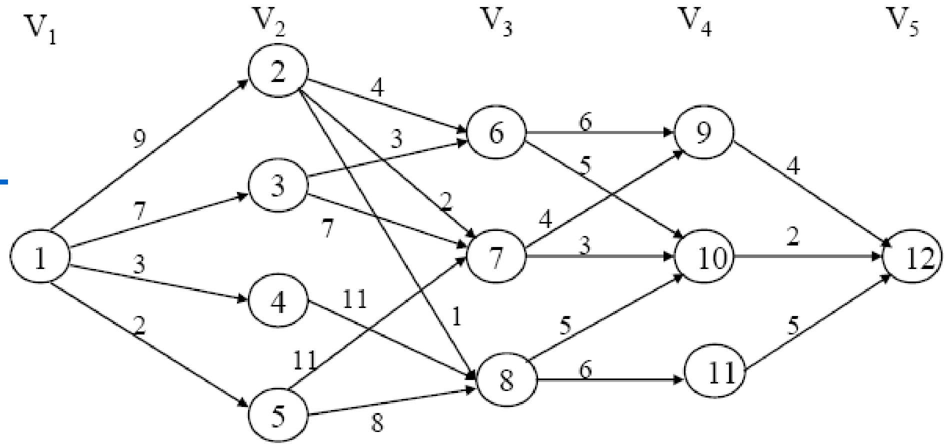
最优性原理

- 过程的最优决策序列具有如下性质：无论过程的初始状态和初始决策是什么，其余的决策都必须相对于初始决策所产生的状态构成一个最优决策序列。
- 利用动态规划求解问题的前提
  - 证明问题满足最优性原理

    如果对所求解问题证明满足最优性原理，则说明用动态规划方法有可能解决该问题
  - 获得问题状态的递推关系式

    获得各阶段间的递推关系式是解决问题的关键。

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。

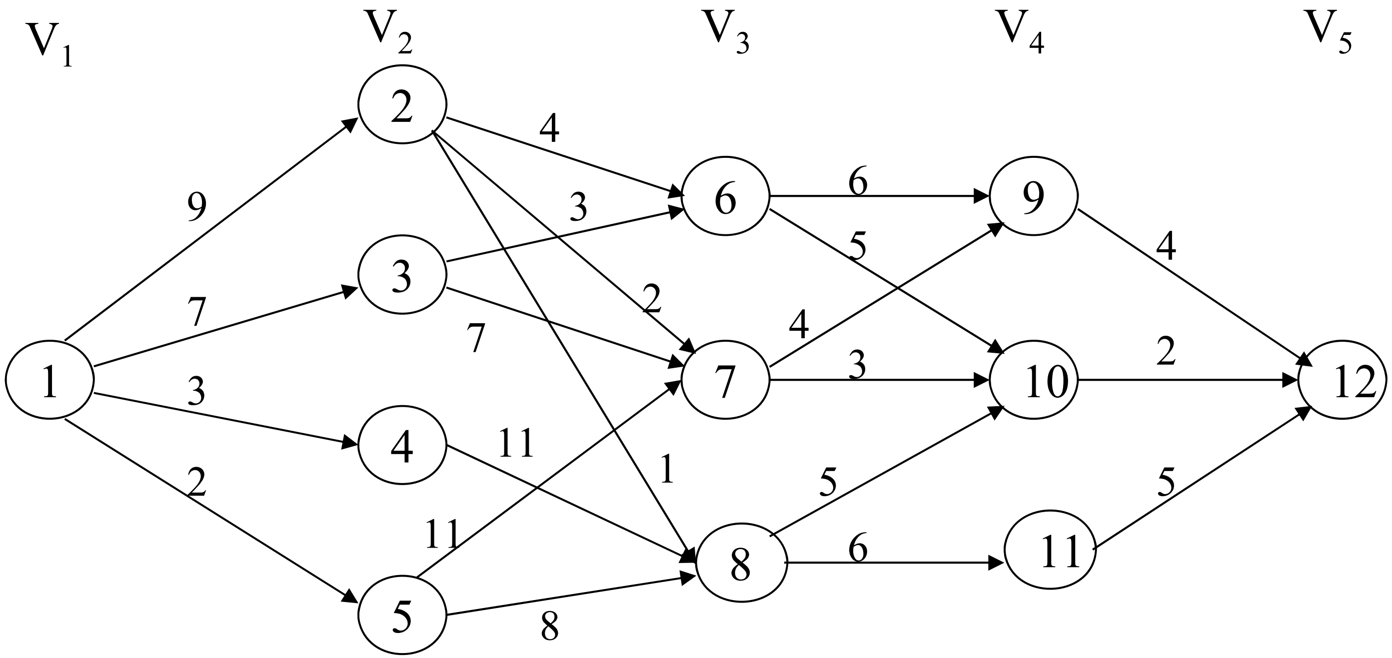
成本：每条边(u,v)均附有成本c(u,v)。

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

最优性原理对多段图问题成立

- 假设$s,v_2,v_3,\ldots,v_{k-1},t$ 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$v_2,v_3,\ldots,v_{k-1},t$ 即

  其余的决策：$v_3,\ldots,v_{k-1}$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$v_2,q_3,\ldots,q_{k-1},t$是一条由v_2到t的更短的路径，则$s,v_2,q_3,\ldots,q_{k-1},t$ 将是比$s,v_2,v_3,\ldots,v_{k-1},t$更短的从s到t的路径。与假设矛盾。

Multistage Decision Processes

Problem:

- 在多段图中求从s到t的一条最小成本的路径，可以看作是在k-2个阶段作出某种决策的结果。
- 第i次决策决定V_{i+1}中的哪个结点在这条路径上，这里$1 \le i \le k-2$；
- 最优性原理对多段图问题成立

Multistage Decision Processes

向前处理策略求解多段图

- 设P(i, j)是一条从V_i中的结点 j 到汇点 t 的最小成本路径，

  COST(i, j)是这条路径的成本。

向前递推式

$$COST(i,j)=\min_{\substack{l\in V_{i+1}\\ (j,l)\in E}}\{c(j,l)+COST(i+1,l)\}$$

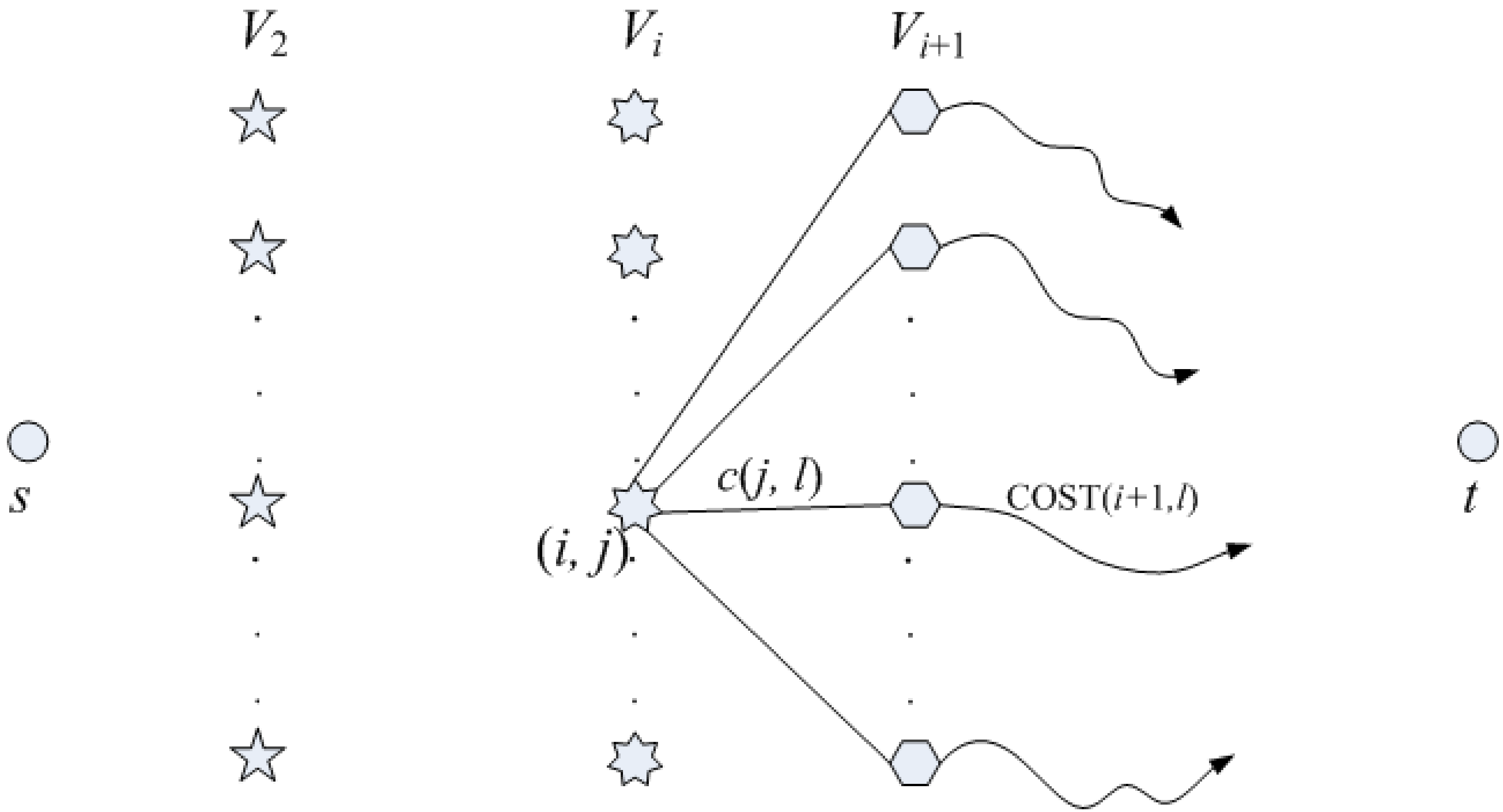
递推过程

第k−1段

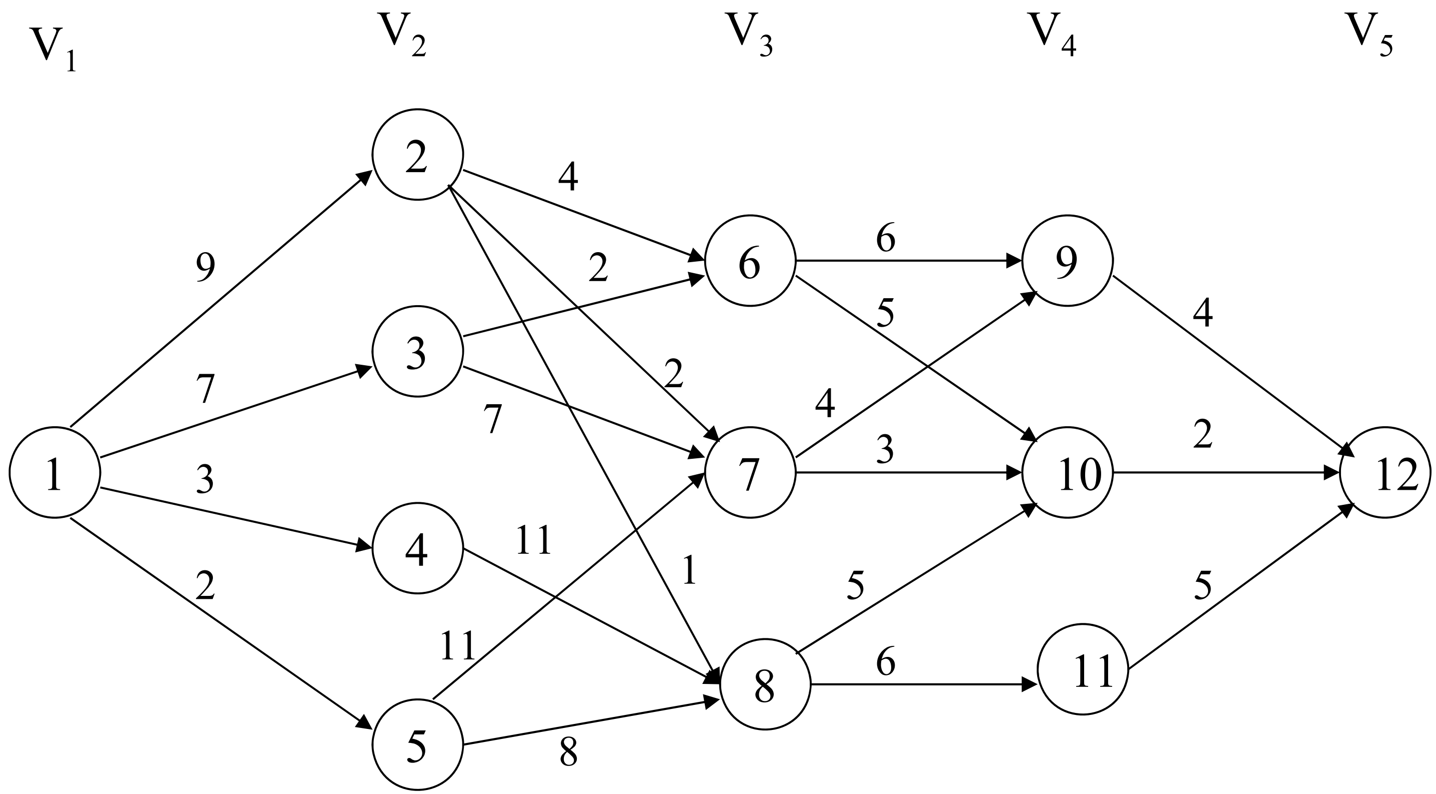
$$COST(k-1,j)=\begin{cases} c(j,t) & <j,t>\in E \\ \infty & <j,t>\notin E \end{cases}$$

对所有 $j\in V_{k-2}$ 计算COST（k-2, j）;然后对所有 $j\in V_{k-3}$ 计算COST（k-3, j）

Multistage Decision Processes

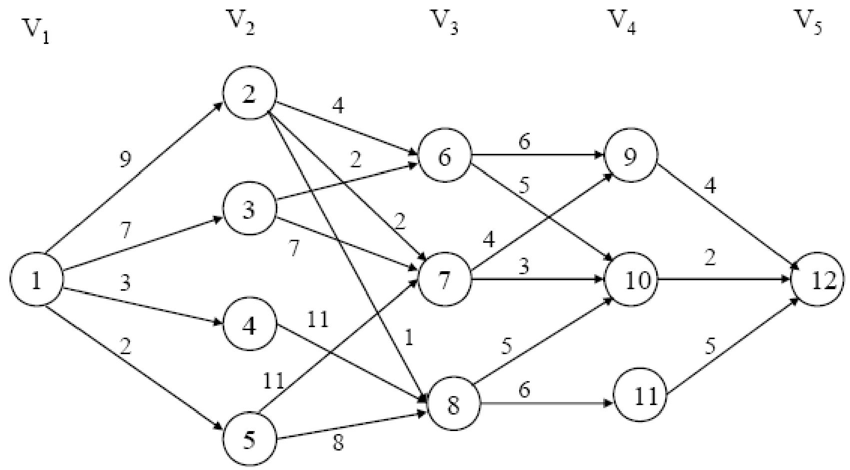


Multistage Decision Processes



5段图

Multistage Decision Pr



- 向前递推结果

第4段　COST(4,9) = c(9,12) = 4
COST(4,10) = c(10,12) = 2
COST(4,11) = c(11,12) = 5

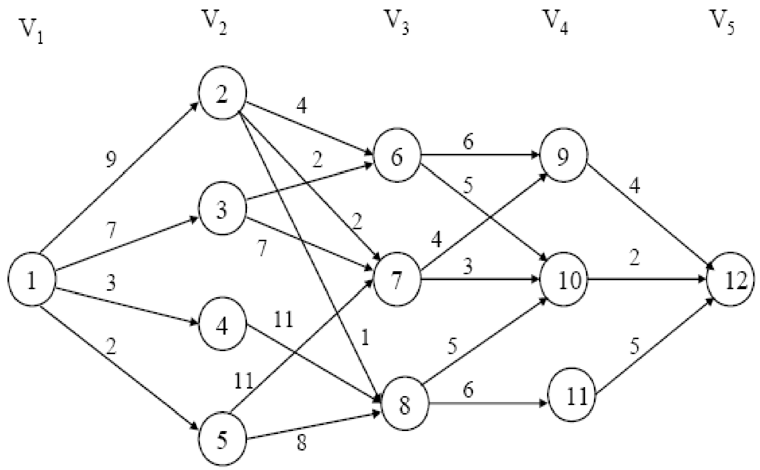
第3段　COST(3,6) = min{6+COST(4,9),5+COST(4,10)} = 7
COST(3,7) = min{4+COST(4,9),3+COST(4,10)} = 5
COST(3,8) = min{5+COST(4,10),6+COST(4,11)} = 7

第2段　COST(2,2) = min{4+COST(3,6) , 2+COST(3,7), 1+COST(3,8)} = 7
COST(2,3) = 9
COST(2,4) = 18
COST(2,5) = 15

第1段　COST(1,1) = min{9+COST(2,2),7+COST(2,3),
3+COST(2,4),2+COST(2,5)} = 16

S到t的最小成本路径的成本 ＝ 16

Multistage Decision Proc



- 最小成本路径的求取

$$COST(i,j) = \min_{\substack{l \in V_{i+1} \\ (j,l) \in E}} \{c(j,l) + COST(i+1,l)\}$$

记 D(*i*,*j*)＝每一COST(*i*,*j*)的决策

即，使c(*j*,*l*)+COST(*i*+1,*l*)取得最小值的*l* 值。

例：D(3,6) = 10, D(3,7)= 10 D(3,8) = 10

D(2,2) = 7, D(2,3) = 6, D(2,4) = 8, D(2,5) = 8

D(1,1) = 2

根据D(1,1)的决策值向后递推求取最小成本路径：

v_2 = D(1,1)=2

v_3 = D(2,D(1,1)) = 7

v_4 = D(3,D(2,D(1,1))) = D(3,7) = 10

故由s到t的最小成本路径是：1→2→7→10→12

Multistage Decision Processes

算法描述

结点的编号规则

源点s 编号为1,然后依次对V_2、V_3…V_{k-1}中的结点编号，汇点 t 编号为n。

目的：使对COST和D的计算仅按n-1,n-2,…,1的次序计算即可，无需考虑标示结点所在段的第一个下标。

时间复杂度 $\Theta(n+e)$

Multistage Decision Processes

```
procedure FGRAPH(E,k,n,P)
    // 输入是按段的顺序给结点编号的，有n个结点的k段图。E是边
     集，c(i,j)是边<i,j>的成本。P(1:k)带出最小成本路径
    real COST(n); integer D(n-1),P(k),r,j,k,n
    COST(n)←0
    for j←n-1 to 1 by -1 do //计算COST(j)//
      设r是具有性质：<j,r>∈E且使c(j,r)+COST(r)取最小值的结点
      COST(j)←c(j,r) + COST(r)
      D(j) ←r    //记录决策值
    repeat
    P(1)←1; P(k)←n
    for j←2 to k-1 do     //找路径上的第j个结点
       P(j) ←D(P(j-1))    //回溯求出该路径
    repeat
  end FGRAPH
```

Multistage Decision Processes

向后处理策略求解多段图

- 设BP(i, j)是一条从源点s到V_i中的结点j的最小成本路径，

  BCOST(i, j)是这条路径的成本。

向后递推式

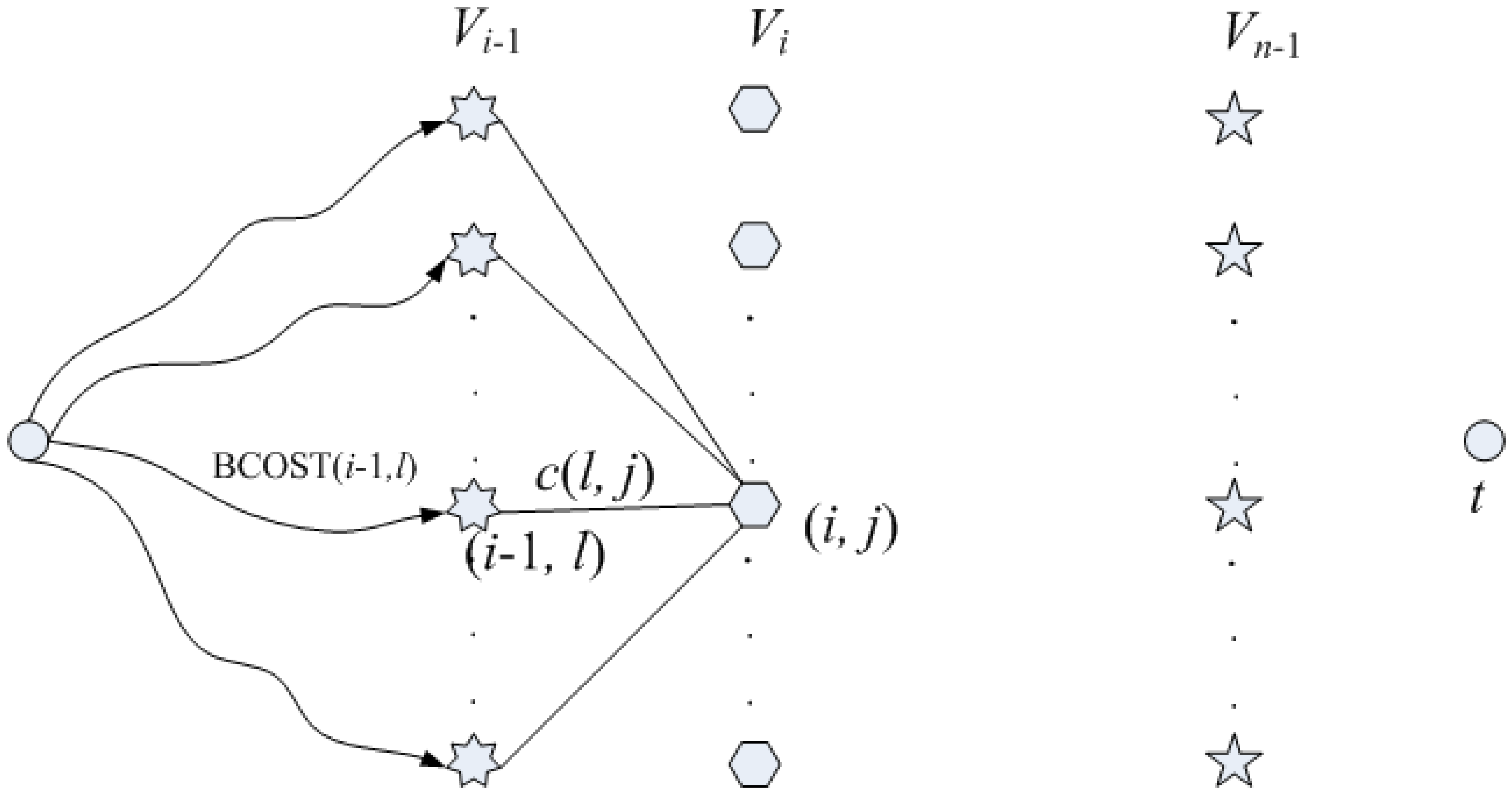
$$BCOST(i,j) = \min_{\substack{l \in v_{i-1} \\ (l,j) \in E}} \{BCOST(i-1,l) + c(l,j)\}$$

递推过程

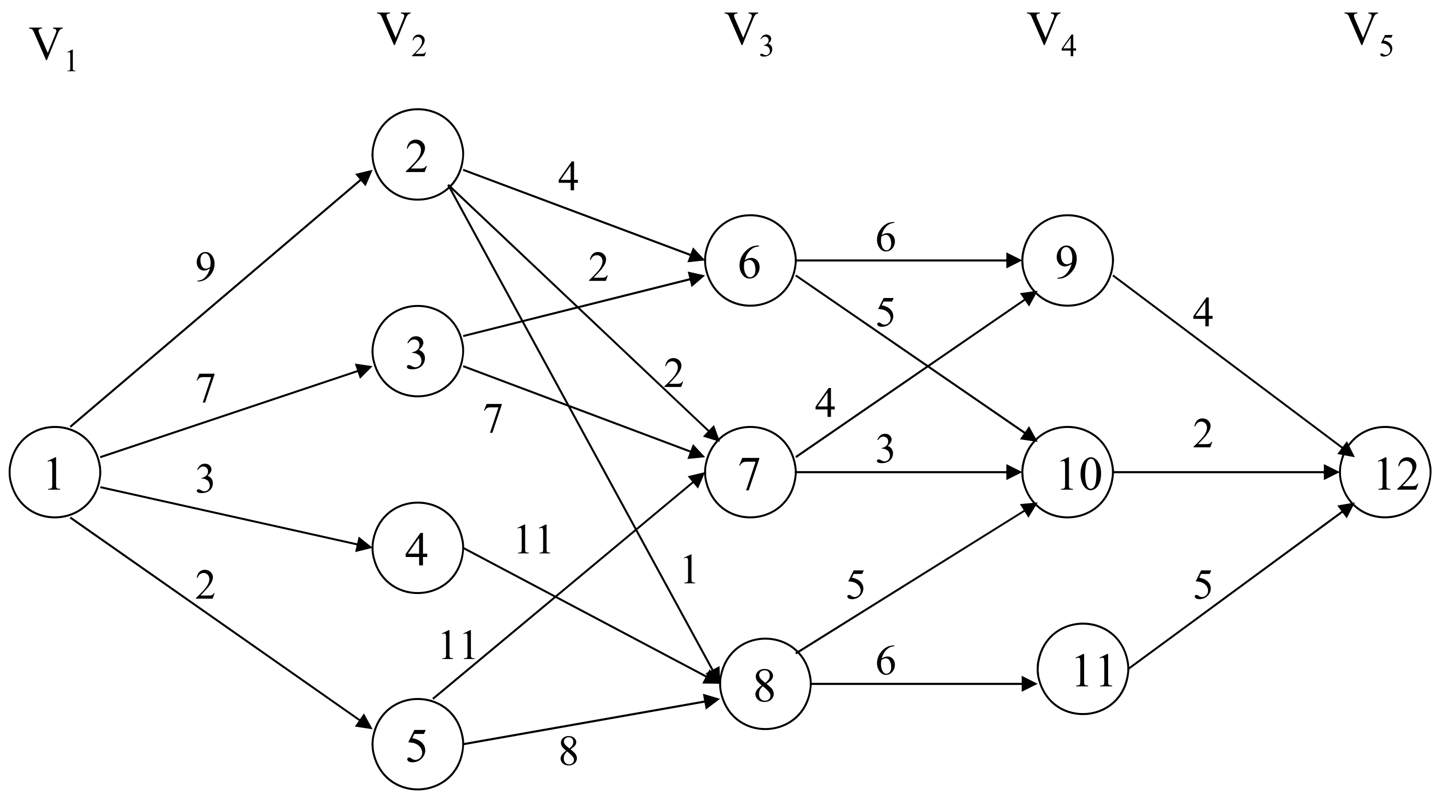
第2段

$$BCOST(2,j) = \begin{cases} c(1,j) & <1,j> \in E \\ \infty & <1,j> \notin E \end{cases}$$

Multistage Decision Processes

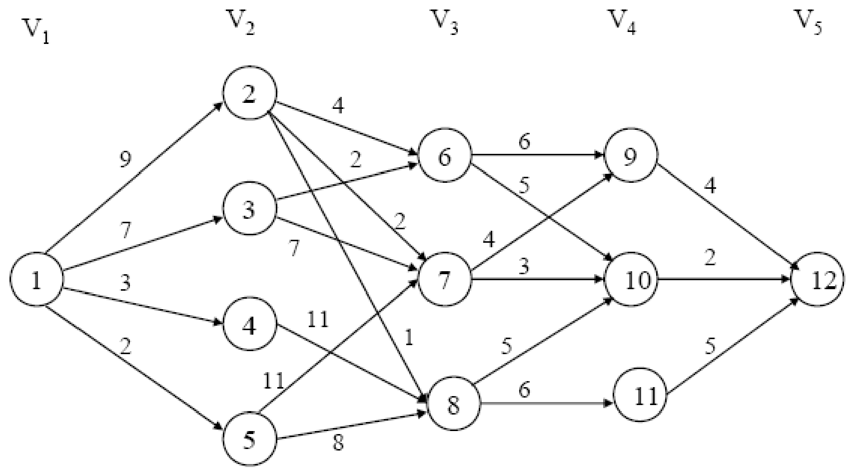


Multistage Decision Processes



5段图

Multistage Decision Pr



- 向后递推结果

第2段 BCOST(2,2) = 9
BCOST(2,3) = 7
BCOST(2,4) = 3
BCOST(2,5) = 2

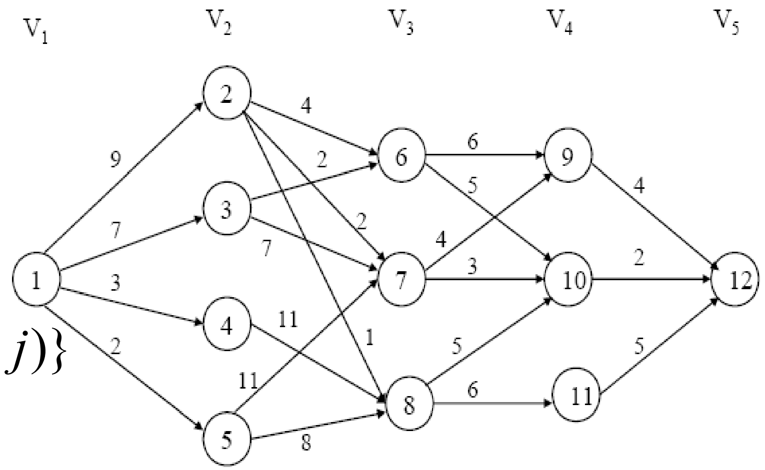
第3段 BCOST(3,6) = min{BCOST(2,2)+4,BCOST(2,3)+2} = 9
BCOST(3,7) = min{BCOST(2,2)+2,BCOST(2,3)+7,BCOST(2,5)+11} = 11
BCOST(3,8) = min{BCOST(2,4)+11, BCOST(2,5)+8} = 10

第4段 BCOST(4,9) = min{BCOST(3,6)+6,BCOST(3,7)+4} = 15
BCOST(4,10) = min{BCOST(3,6)+5, BCOST(3,7)+3,BCOST(3,8)+5}= 14
BCOST(4,11) = min{BCOST(3,8)+6} = 16

第5段 BCOST(5,12) = min{BCOST(4,9)+4, BCOST(4,10)+2, BCOST(4,11)+5}
= 16

S到t的最小成本路径的成本 ＝ 16

Multistage Decision Proc



- 最小成本路径的求取

$$BCOST(i,j) = \min_{\substack{l \in v_{i-1} \\ (l,j) \in E}} \{COST(i-1,l) + c(l,j)\}$$

记 BD(i,j)＝每一BCOST(i,j)的决策

即，使 COST(i-1,l)+c(l,j) 取得最小值的l 值。

例：BD(3,6) = 3, BD(3,7)= 2 ,BD(3,8) = 5

BD(4,9) = 6, BD(4,10) = 7, BD(4,11) = 8

BD(5,12) = 10

根据D(5,12)的决策值向前递推求取最小成本路径：

v_4 = BD(5,12)=10

v_3 = BD(4,BD(5,12)) = 7

v_2 = BD(3,BD(4,BD(5,12))) = BD(3,7) = 2

故由s到t的最小成本路径是：1→2→7→10→12

Multistage Decision Processes

- 算法描述

```
procedure BGRAPH(E,k,n,P)
    //输入是按段的顺序给结点编号的，有n个结点的k段图。E是边
     集，c(i,j)是边<i,j>的成本。P(1:k)带出最小成本路径
    real BCOST(n); integer BD(n-1),P(k),r,j,k,n
    BCOST(1)←0
    for j←2  to n  do //计算BCOST(j)
      设r是具有<r,j>∈E且使BCOST(r)+ c(r,j)取最小值性质的结点
      BCOST(j)← BCOST(r)+ c(r,j)
      BD(j) ←r    //记录决策值
    repeat
    P(1)←1; P(k)←n
    for j←k-1 to 2 by -1 do     //找路径上的第j个结点
       P(j) ←D(P(j+1))    //回溯求出该路径
    repeat
  end BGRAPH
```

Multistage Decision Processes

- **多段图问题的应用实例 - 资源的分配问题**
  - **将n个资源分配给r个项目的问题：**
    - 如果把 j 个资源，$0 \le j \le n$，分配给项目 i，可以获得 $N(i,j)$ 的净利。
    - 问题：如何将这n个资源分配给r个项目才能使各项目获得的净利之和达到最大。
    - 转换成一个r+1段图问题求解。

Multistage Decision Processes

- 用r＋1段图描述该问题：

段：1到r之间的每个段 i 表示项目 i。$2 \le i \le r$

结点：

源点s =V(1, 0);　　汇点t=V(r+1, n)

当$2 \le i \le r$时，每段有n+1个结点，j =0, 1, …, n

每个结点V(i, j) 表示到项目 i 之前为止，共把j个资源分配给了前i-1个项目，

边的一般形式：<V(i, j), V(i+1, l)>, $j \le l$, $1 \le i \le r$

成本：

- 当 $j \le l$ 且$1 \le i < r$时，边<V(i, j), V(i+1, l)>赋予成本N(i, l-j),表示给项目i分配l-j个资源所可能获得的净利。
- 当$j \le n$且i=r时，此时的边为：<V(r,j),V(r+1,n)>,该边赋予成本：

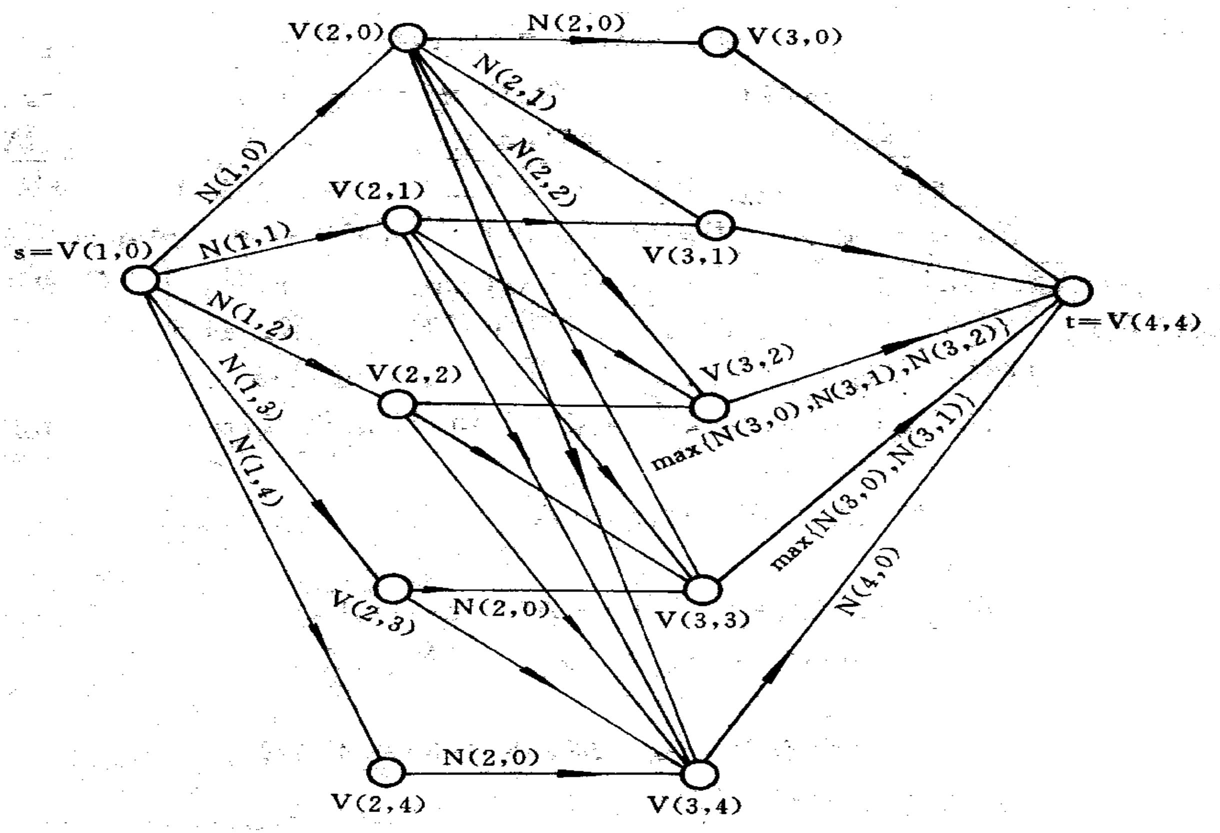
$$\max_{0 \le p \le n-j} \{N(r, p)\}$$

指向汇点的边

Multistage Decision Processes

- 实例：将n=4个资源分配给r=3个项目。构成一个4段图。

  问题的解：资源的最优分配方案由一条s到t的最大成本路径给出——改变边成本的符号，从而将问题转换成为求取最小成本路径问题。



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*

$\leftrightarrow$ *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 $T = \{t_{ij}\}$,

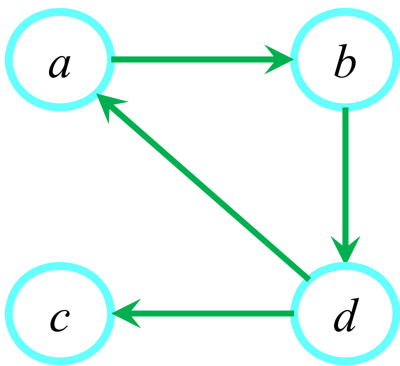
t_{ij} *in the* i^{th} *row and* j^{th} *column* $= 1$

$\leftrightarrow$ *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***

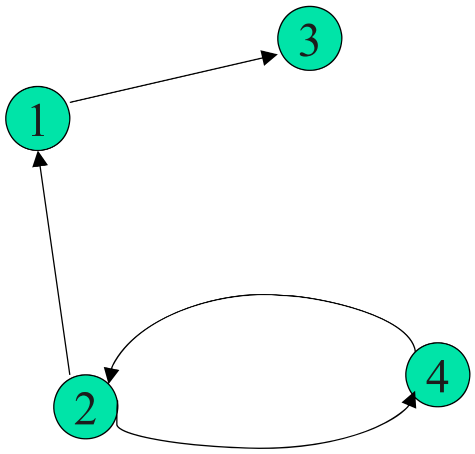


adjacent matrix

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

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*

$$v_i, \text{ vertices numbered} \le k\text{-}1, v_k, \text{ vertices numbered} \le k\text{-}1, v_j \quad (**)$$

- *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*

  → $r_{kj}^{(k-1)} = 1$

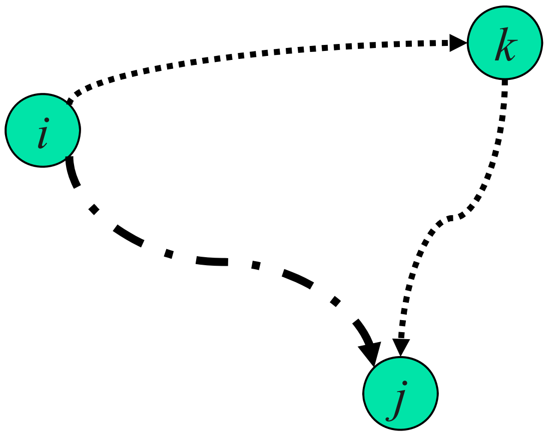
Warshall's Algorithm

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

$$
r_{ij}^{(k)} = 1: \begin{cases} r_{ij}^{(k-1)} = 1 & \text{(path using just } 1, \ldots, k-1\text{)} \\ \text{or} & \\ (r_{ik}^{(k-1)} = 1 \text{ and } r_{kj}^{(k-1)} = 1) & \text{(path from } i \text{ to } k \text{ and from } k \text{ to } i \text{ using just } 1, \ldots, k-1\text{)} \end{cases}
$$



Warshall's Algorithm

Warshall's Algorithm

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

Rule to determine whether $r_{ij}^{(k)}$ should be 1 in $R^{(k)}$:

- *If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$.*
- *If an element r_{ij} is 0 in $R^{(k-1)}$,it has to be changed to 1 in $T^{(k)}$ iff the element in its row i and column k and the element in its column j and row k are both 1's in $R^{(k-1)}$.*

$$R^{(k-1)} = \begin{array}{c|cccc|} & j & \cdots\cdots & k & \\ \hline & & & & \\ k & 1 & & & \\ & \uparrow & & & \\ i & 0 & \rightarrow & 1 & \\ & & & & \\ \hline \end{array}$$

$$R^{(k)} = \begin{array}{c|cccc|} & j & \cdots\cdots & k & \\ \hline & & & & \\ k & 1 & & & \\ i & 1 & & 1 & \\ & & & & \\ \hline \end{array}$$

Warshall's Algorithm

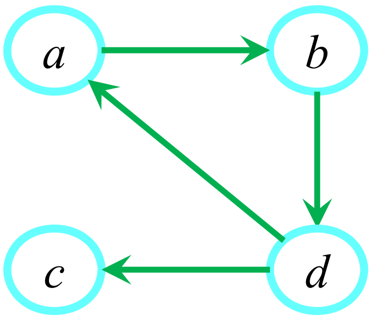
Warshall's Algorithm

TRANSITIVE-CLOSURE(G)

$n \leftarrow |V[G]|$
for $i \leftarrow 1$ **to** n
    do for $j \leftarrow 1$ **to** n
        do if $i = j$ or $(i, j) \in E[G]$
            then $t_{ij}^{(0)} \leftarrow 1$
            else $t_{ij}^{(0)} \leftarrow 0$
for $k \leftarrow 1$ **to** n
    do for $i \leftarrow 1$ **to** n
        do for $j \leftarrow 1$ **to** n
            do $t_{ij}^{(k)} \leftarrow t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$
return $T^{(n)}$

Warshall's Algorithm

- ***Example*1**



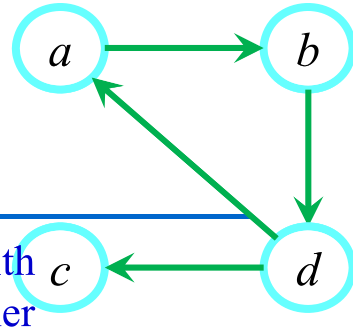
$$R^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

Ones reflect the existence of paths with no intermediate vertices.
$R^{(0)}$ is adjacent matrix;
boxed row and column are used for getting $R^{(1)}$

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

Ones reflect the existence of paths with intermediate vertices numbered no higher than 1, i.e. just vertex a,
note a new path from d to b.
boxed row and column are used for getting $R^{(2)}$

Warshall's Algorithm



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

Ones reflect the existence of paths with intermediate vertices numbered no higher than 2, i.e. vertex *a* and *b*,
note two new paths.
boxed row and column are used for getting $R^{(3)}$

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

Ones reflect the existence of paths with intermediate vertices numbered no higher than 3, i.e. vertex *a*, *b* and *c*.
no new path.
boxed row and column are used for getting $R^{(4)}$

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

Ones reflect the existence of paths with intermediate vertices numbered no higher than 4, i.e. vertex *a*, *b, c* and *d.*
note five new paths.
boxed row and column are used for getting $R^{(4)}$

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.

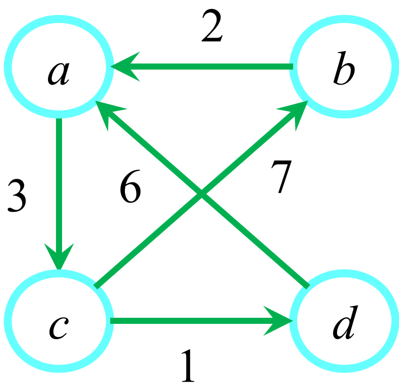
- *distance matrix:*

  *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***

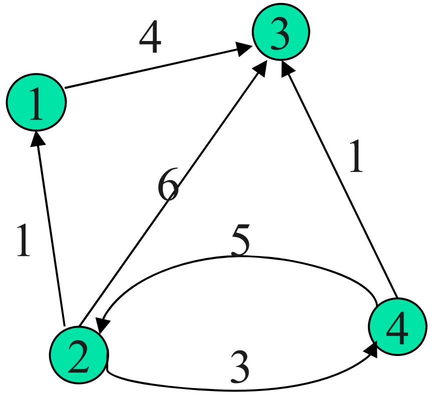


weight matrix

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

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 $D^{(k-1)}$?*

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

$\leftrightarrow$ *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

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

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

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* $\le k$-1, $v_{k,}$ *vertices numbered* $\le k$-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

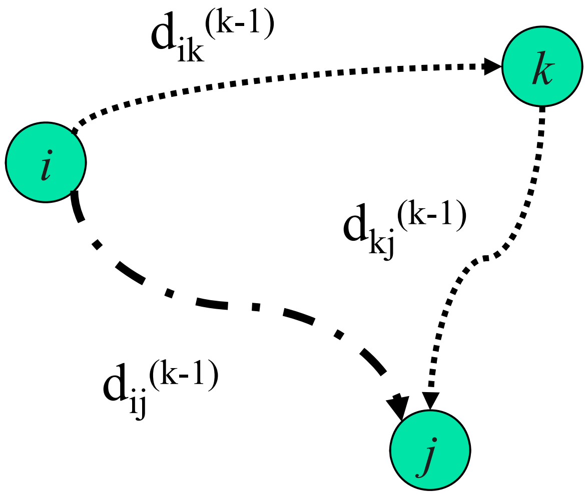
Floyd's Algorithm

- *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

Floyd's Algorithm

FLOYD(W)

$n \leftarrow rows[W]$
$D^{(0)} \leftarrow W$
for $k \leftarrow 1$ **to** n
  do for $i \leftarrow 1$ **to** n
    do for $j \leftarrow 1$ **to** n
      do $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

Time Complexity: $O(n^3)$, Space ?

Floyd's Algorithm: All pairs shortest paths

Floyd's Algorithm

Less space

```
FLOYD'(W)
n ← rows[W]
D ← W
for k ← 1 to n
    do for i ← 1 to n
        do for j ← 1 to n
            do d_ij ← min(d_ij, d_ik + d_kj)
return D
```

Floyd's Algorithm: All pairs shortest paths

Floyd's Algorithm

- *Constructing a shortest path*

for k=0

for k≥1

Floyd's Algorithm: *All pairs shortest paths*

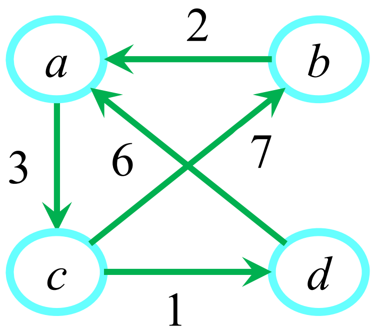
Floyd's Algorithm

Print all-pairs shortest paths

```
PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j)
if i = j
   then print i
   else if π_ij = NIL
           then print "no path from" i "to" j "exists"
           else PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, π_ij)
                print j
```

Floyd's Algorithm: *All pairs shortest paths*

- ***Example*1**



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

length of the shortest paths with no intermediate vertices.
$D^{(0)}$ is weight matrix;

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

length of the shortest paths with intermediate vertices numbered no higher than 1, i.e. just vertex *a*,
note two new shortest paths from *b* to *c*, *d* to *c*.

Floyd's Algorithm: All pairs shortest paths

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

length of the shortest paths with intermediate vertices numbered no higher than 2, i.e. vertex *a* and *b*,
note a new shortest path from *c* to *a*.

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

length of the shortest paths with intermediate vertices numbered no higher than 3, i.e. vertex *a b,* and *c*,
note four new shortest paths.

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

length of the shortest paths with intermediate vertices numbered no higher than 4, i.e. vertex *a b, c*, and *d*.
note a new shortest path.