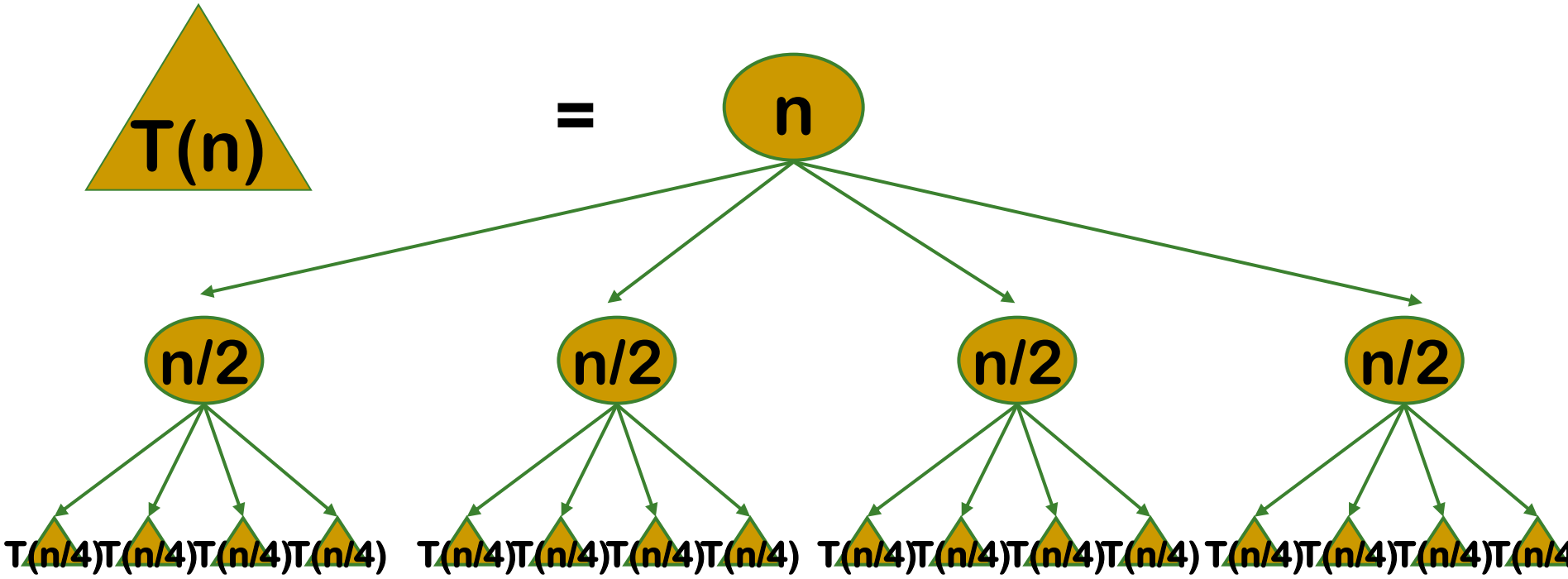
Analysis and Design of Algorithms

Chapter 8: Dynamic Programming

School of Software Engineering © Ye LUO

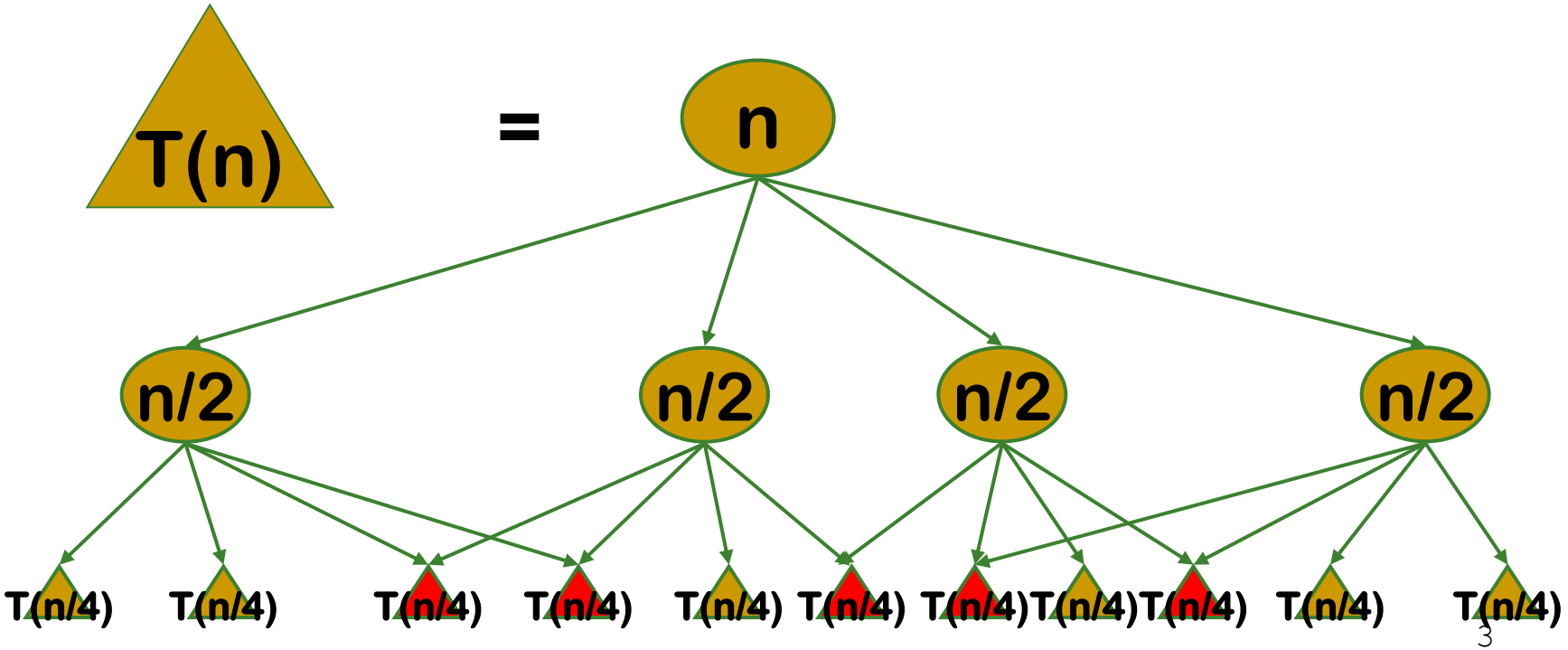
Dynamic Programming

- *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*

动态规划算法的两种形式

上面已经知道动态规划算法的核心是记住已经求过的解，记住求解的方式有两种：①**自顶向下的备忘录法** ②**自底向上。**

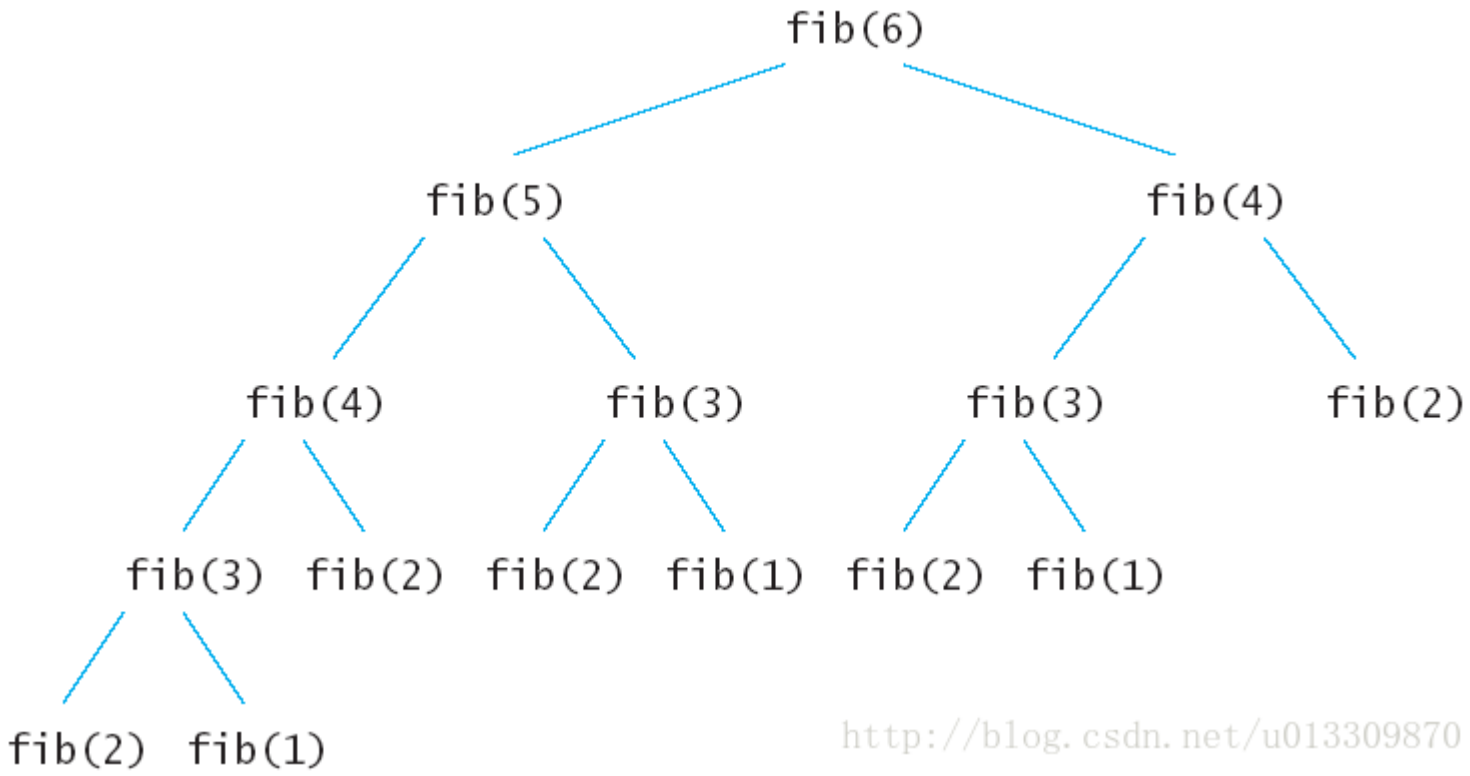
求斐波拉契数列**Fibonacci** 的例子：

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

求斐波拉契数列Fibonacci的递归版本

```
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
```

求斐波拉契数列Fibonacci的递归算法



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

自顶向下的备忘录DP算法

```
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     }
```

自底向上的DP

```
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 }
```

进一步压缩空间

```
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     }
```

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**。

动态规划原理

①最优子结构

用动态规划求解最优化问题的第一步就是刻画最优解的结构，**如果一个问题的解结构包含其子问题的最优解，就称此问题具有最优子结构性质**。因此，某个问题是否适合应用动态规划算法，它是否具有最优子结构性质是一个很好的线索。使用动态规划算法时，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。

动态规划原理

②重叠子问题

在斐波拉契数列中，可以看到大量的重叠子问题，比如说在求fib（6）的时候，fib（2）被调用了5次。如果使用递归算法的时候会反复的求解相同的子问题，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，就称为具有重叠子问题（overlapping subproblems）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。

Dynamic Programming

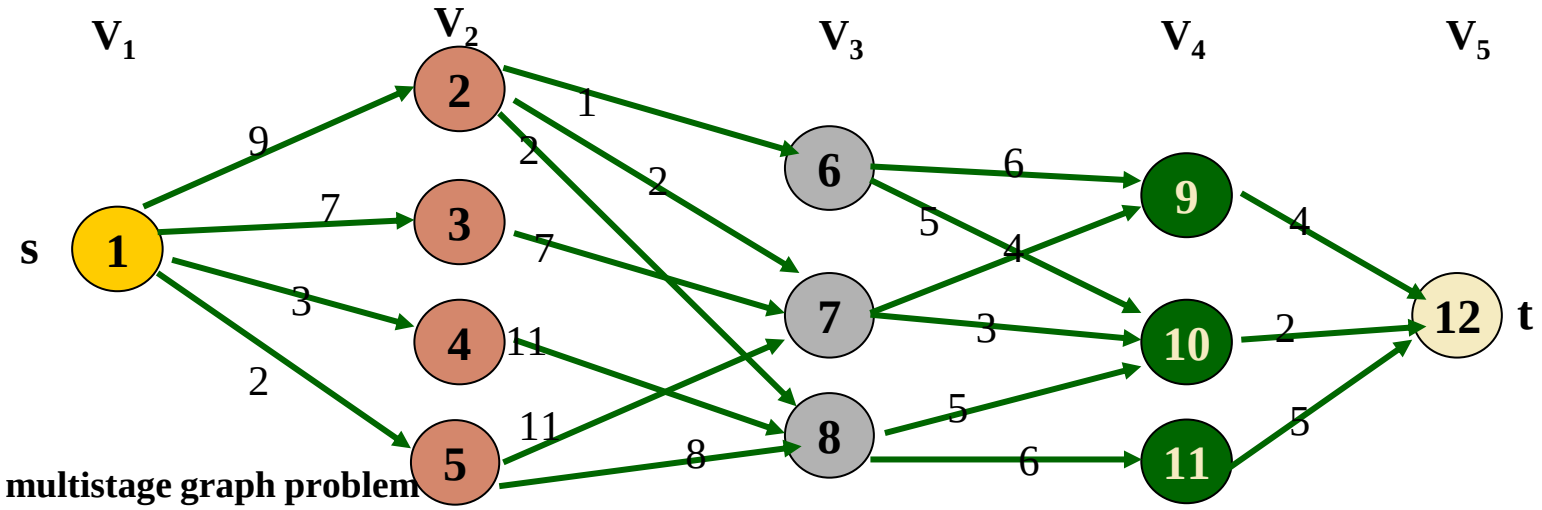
Main idea

- *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

DP and MDP

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



Dynamic Programming

- ***Applications of dynamic programming***

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*
8. *Some instances of difficult discrete optimization problems:*

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

- **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)*

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

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

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)

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

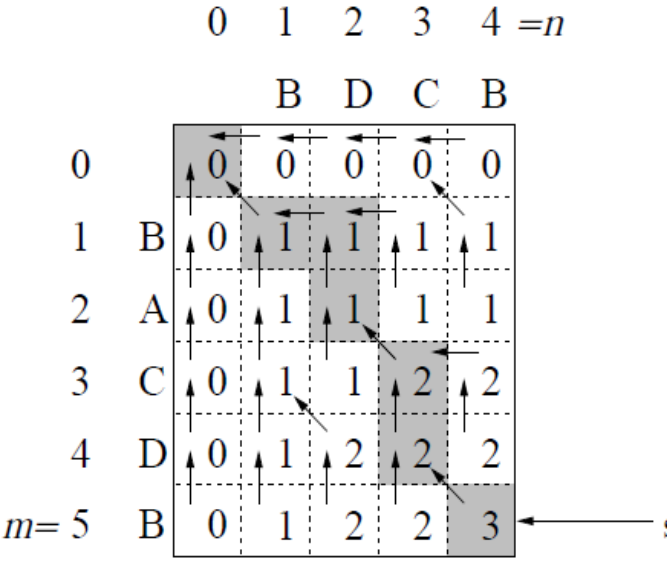
Y = BDCB

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
}
```

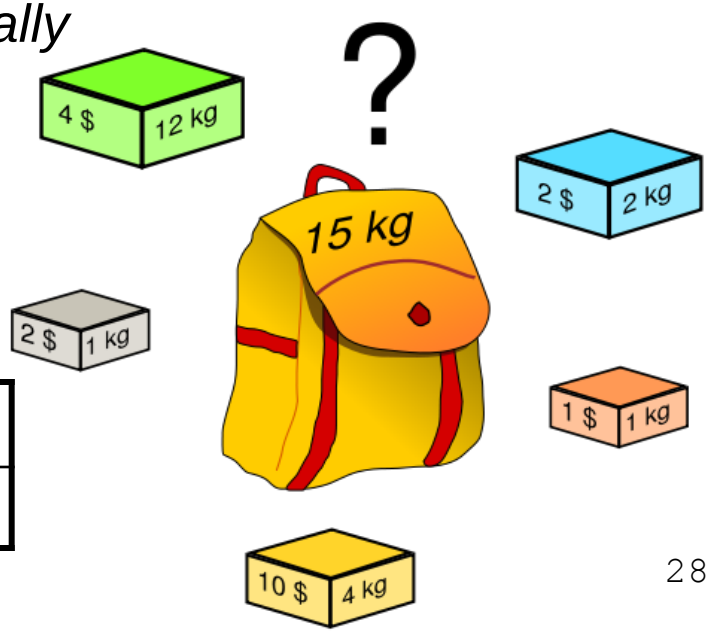
0-1 *Knapsack Problem*

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-*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]*

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

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

0-1 *Knapsack Problem*

Ex. recurrence from the first item

		0	$j-w_i$	……	j	W
	0	0	0		0	0
w_i v_i	i-1	0	$V[i-1, j-w_1]$ + v_i		$V[i-1, j]$	
	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	$V(i-1, j-w_1)+v_1$		$V(i-1, j)$
w_1=2. v_1=12	1	0	0	12	12	12	12	$V(i-1, j-w_2)+v_2$	$V(i-1, j)$	$V(i, j)$
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			

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*

- ***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*

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

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

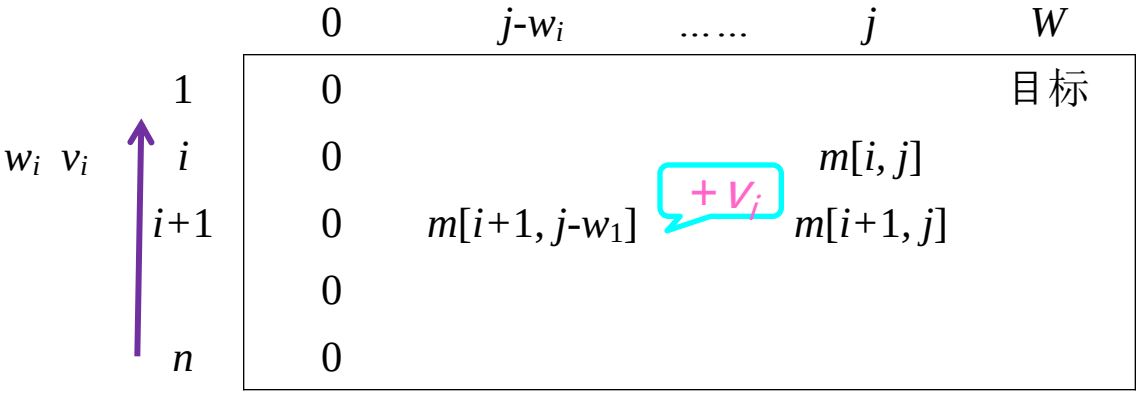
  - *Based on the principle of optimality of* 0-1 *Knapsack problem, we can construct the recurrence equation for m(i, j)*

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

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

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 — 阶跃法

- 考察0-1背包问题的一个具体实例

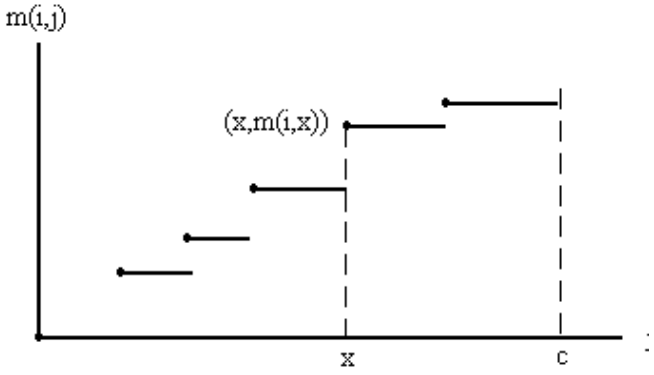
  $n = 5,\ W = 10,\ w = \{2,2,6,5,4\},\ v = \{6,3,5,4,6\}$

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

- 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$i(1 \leq i \leq n)$，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*

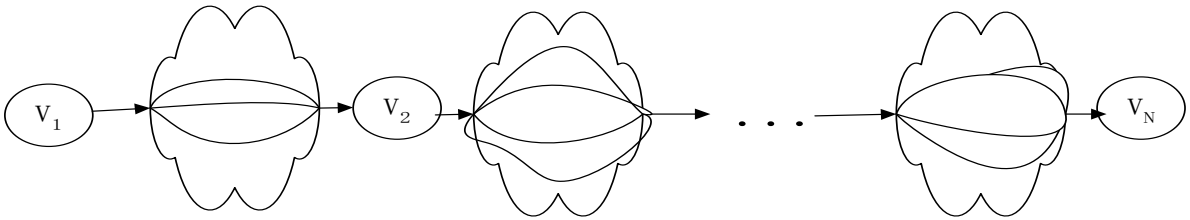
- $p[i]$
- $p[i+1]$
- $q[i+1]=p[i+1]\oplus(w_i,v_i)=\{(j+w_i,m(i,j)+v_i)|(j,m(i,j))\in p[i+1]\}$
- $p[i+1]\cup q[i+1]$
- 并清除其中的受控跳跃点

Multistage Decision Processes

多阶段决策问题

多阶段决策过程 multistep decision process

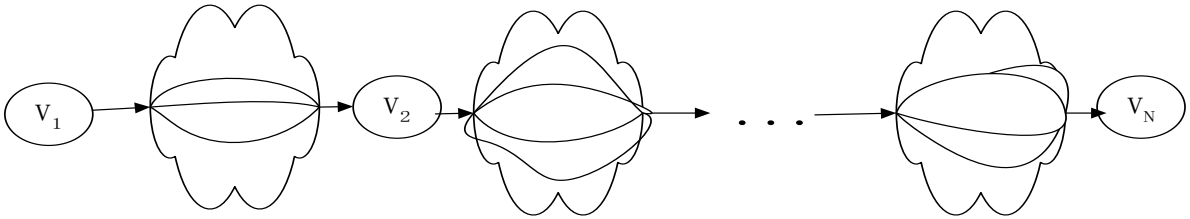
- 问题的活动过程分为若干相互联系的阶段
- 在每一个阶段都要做出决策，这决策过程称为多阶段决策过程
- 任一阶段 i 以后的行为仅依赖于i 阶段的过程状态，而与i 阶段之前的过程如何达到这种状态的方式无关



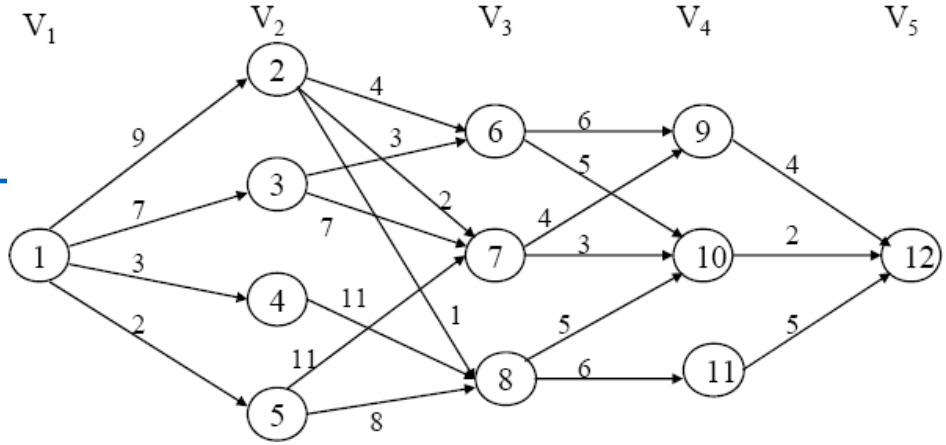
Multistage Decision Processes

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。

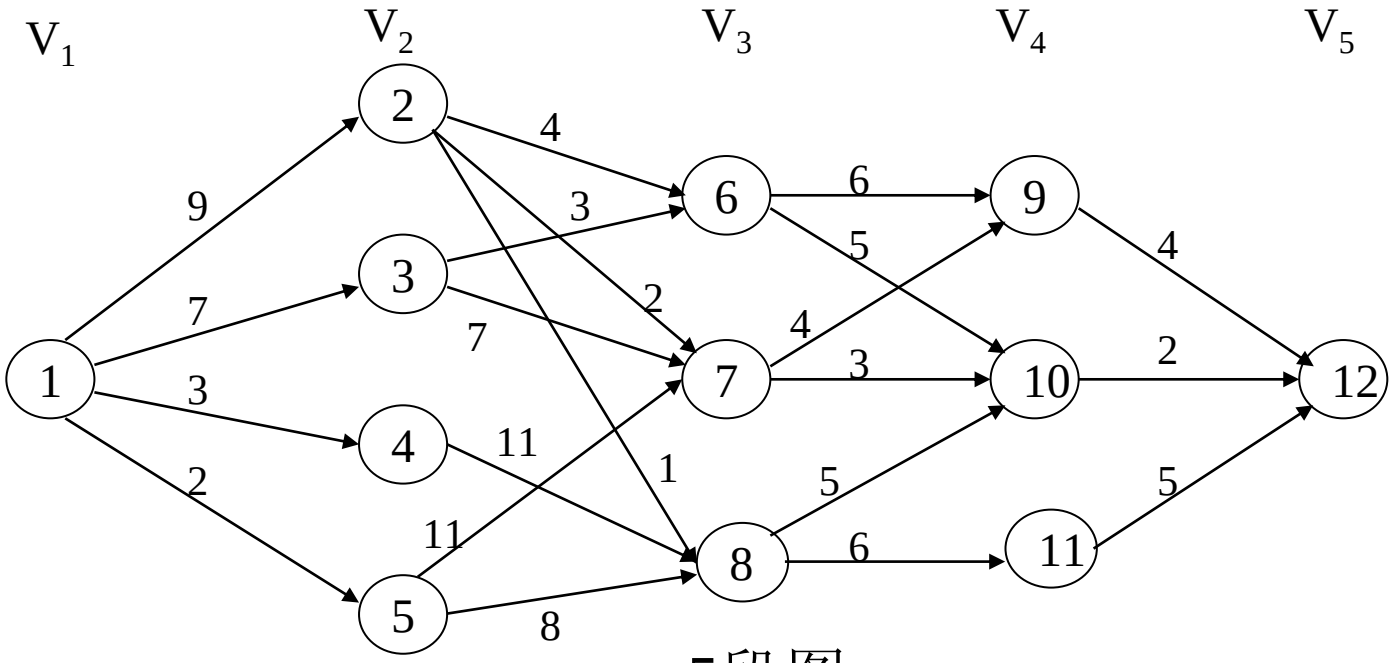
边： 所有的边(u,v)，若<u,v>∈E，则
若u∈V_i，则v∈V_{i+1}，即该边是从某段 *i* 指向*i*+1段，1≤*i*≤k - 1。

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,…,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

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

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

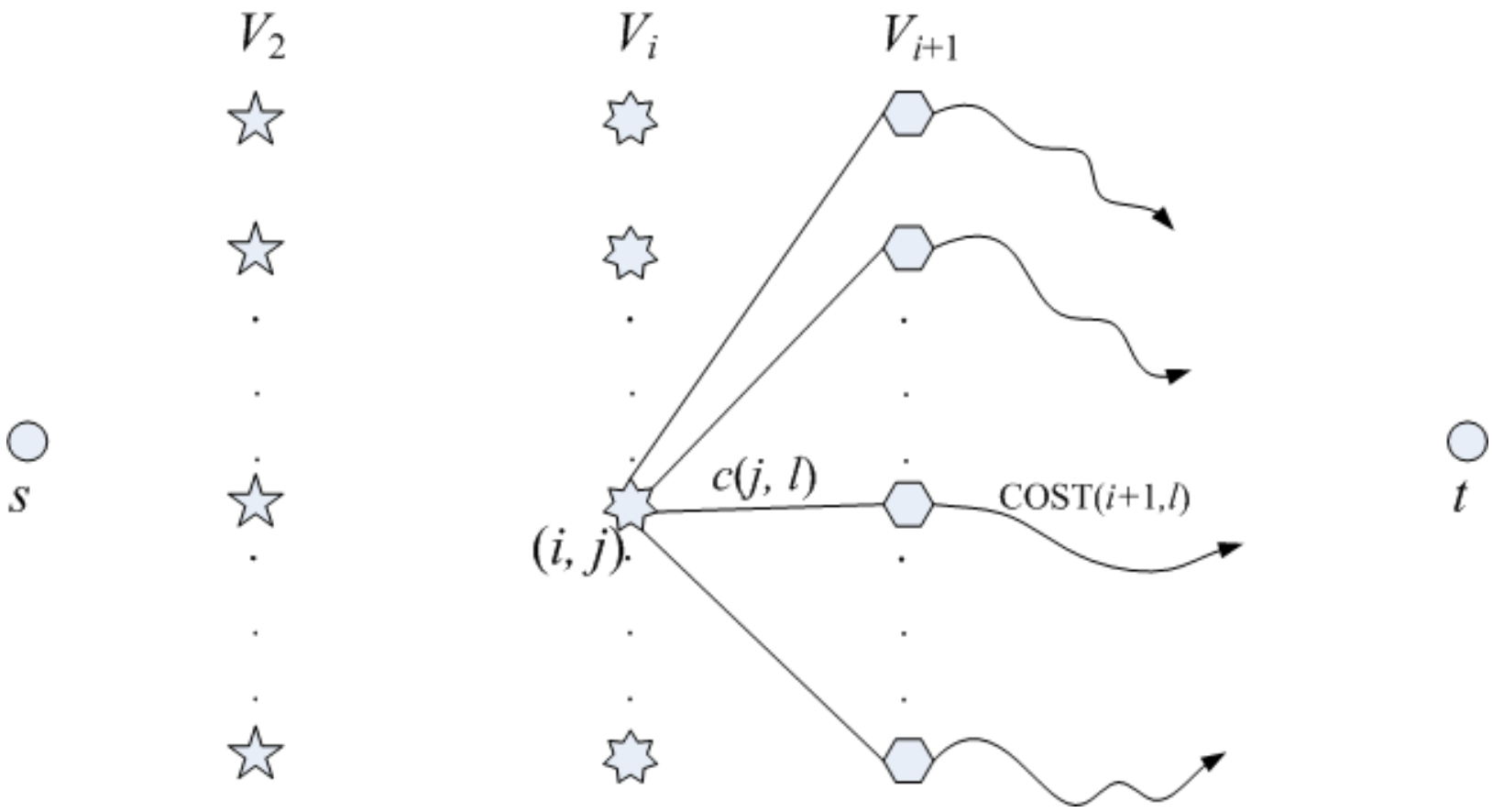
递推过程

第k−1段

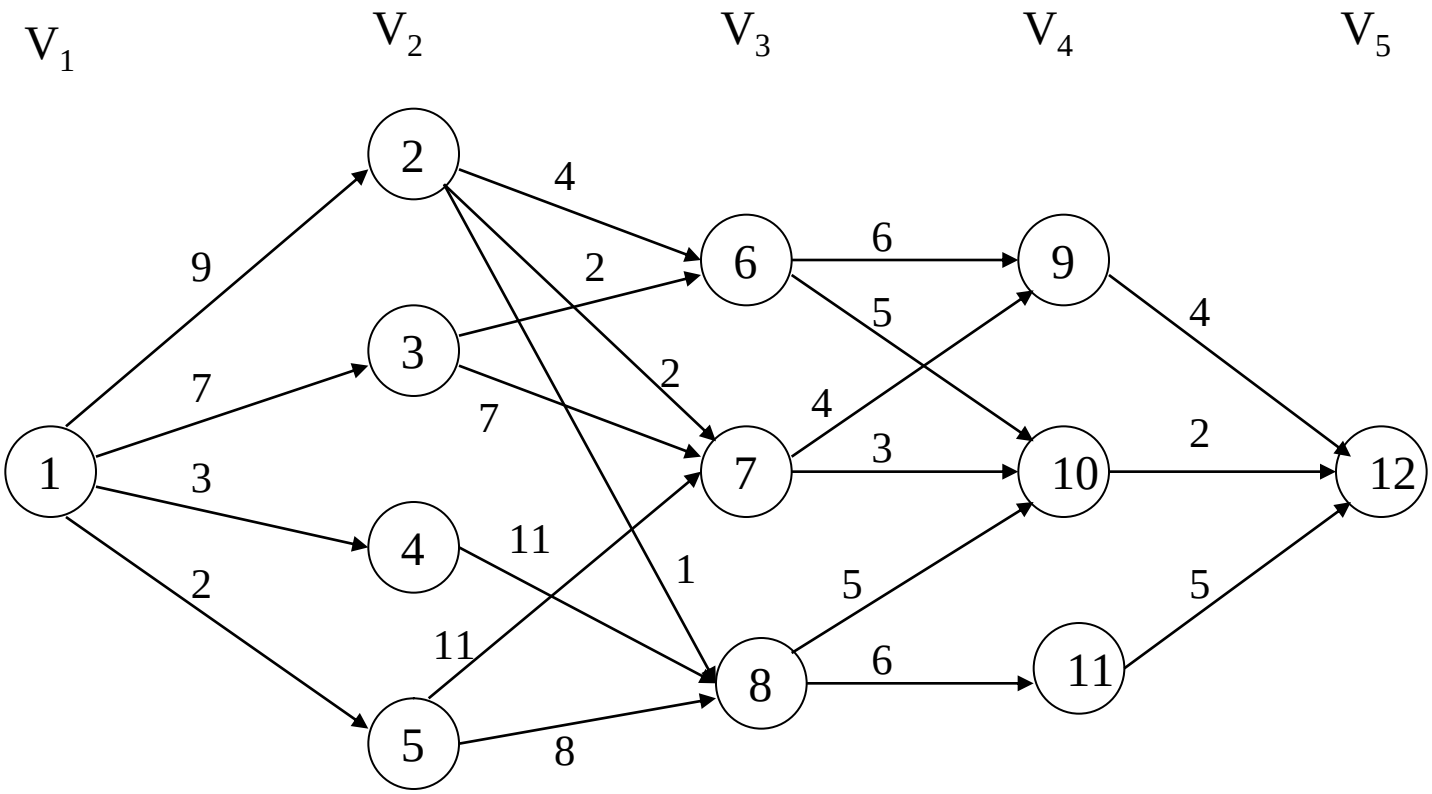
$$COST(k\text{-}1, j) = \begin{cases} c(j,t) & \langle j,t \rangle \in E \\ \infty & \langle j,t \rangle \notin E \end{cases}$$

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

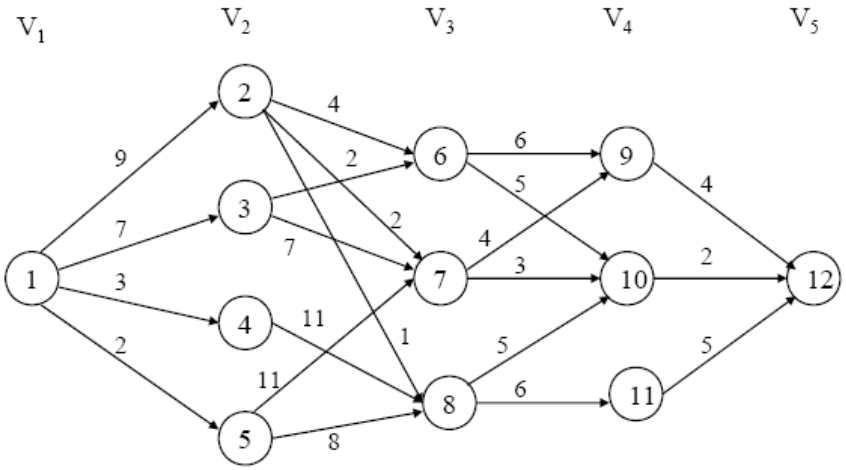


Multistage Decision Processes



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

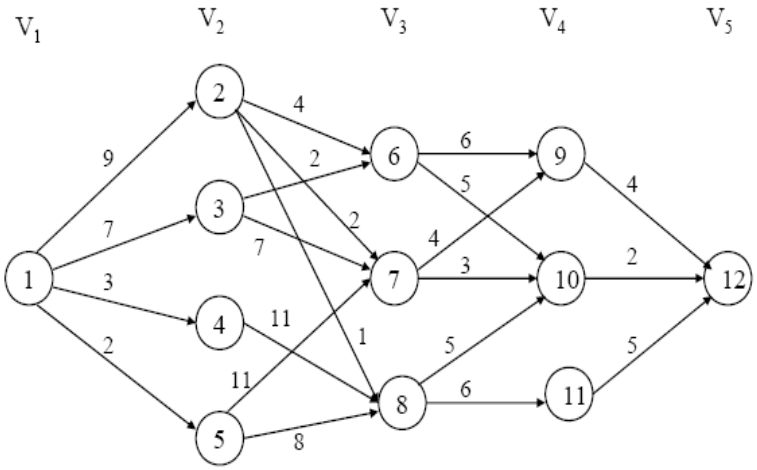
第3段 COST(3,6) = min{6+COST(4,9),5+COST(4,10)} = 7
COST(3,7) = min{4+COST(4,9),3+COST(4,10)} = 5
COST(3,8) = min{5+COST(4,10),6+COST(4,11)} = 7

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

向后处理策略求解多段图

- 设BP(i, j)是一条从源点s到V_j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

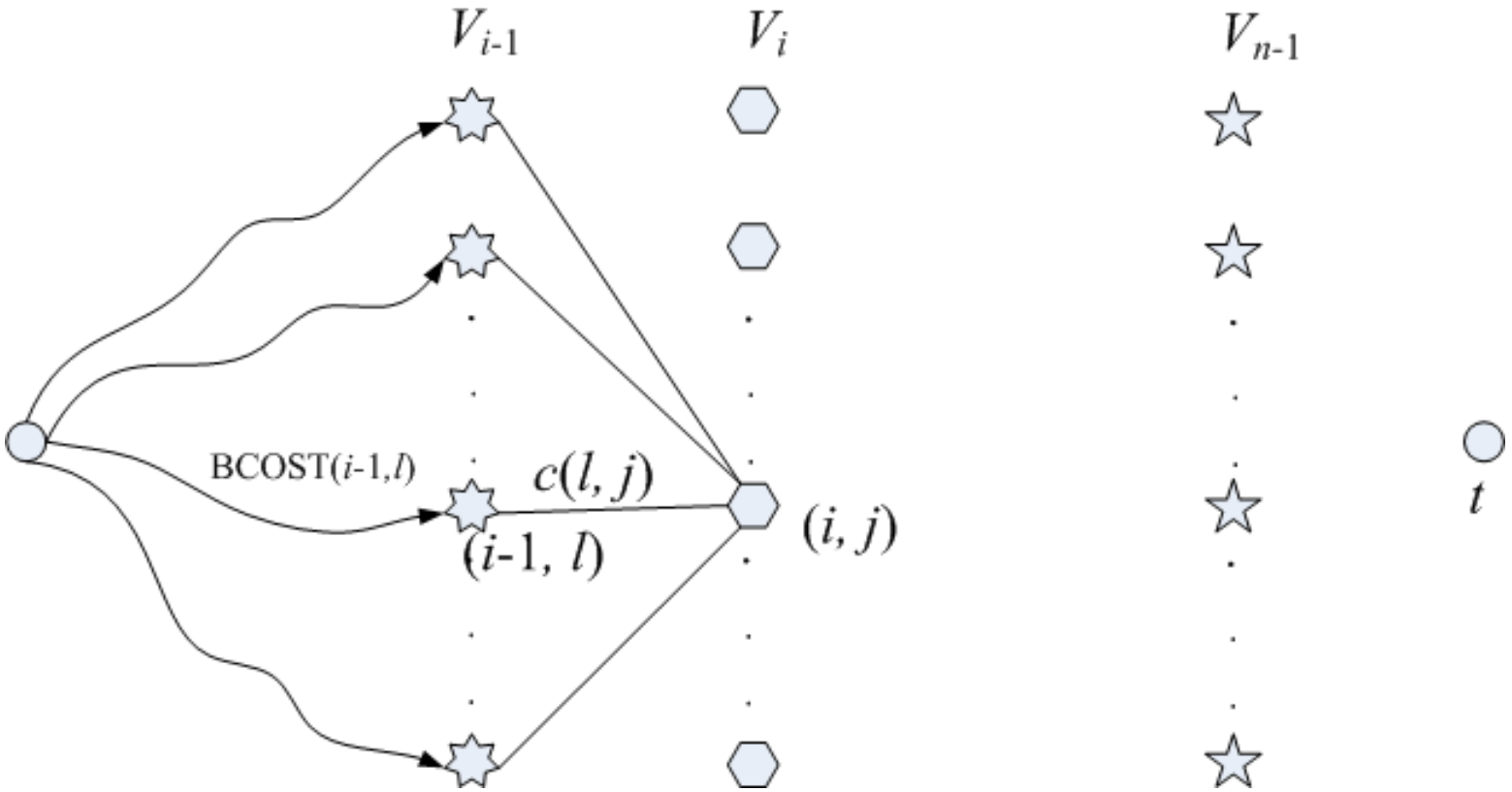
$$BCOST(i, j) = \min_{\substack{l \in v_{i-1} \\ (l, j) \in E}} \{BCOST(i-1, l) + c(l, j)\}$$

递推过程

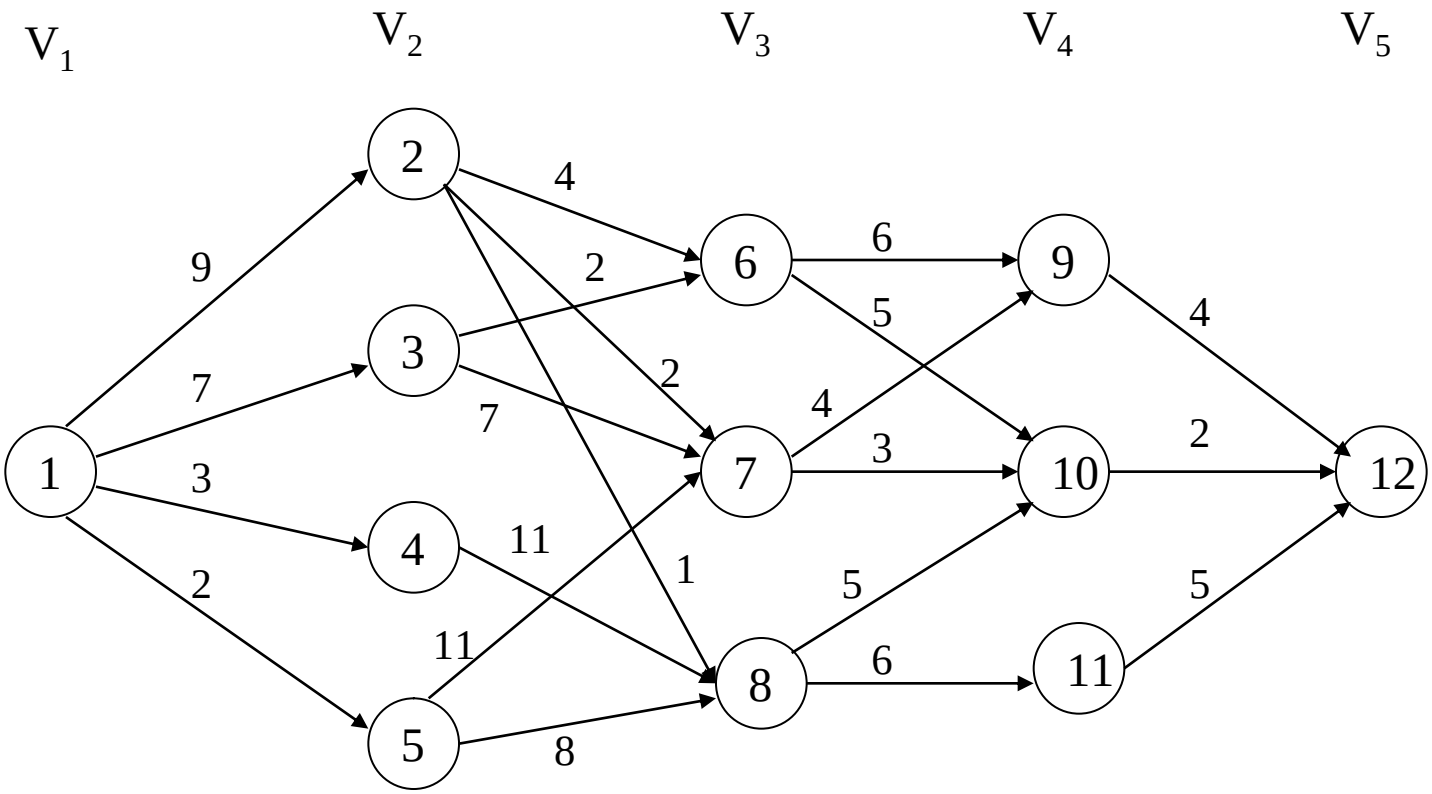
第2段

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

Multistage Decision Processes

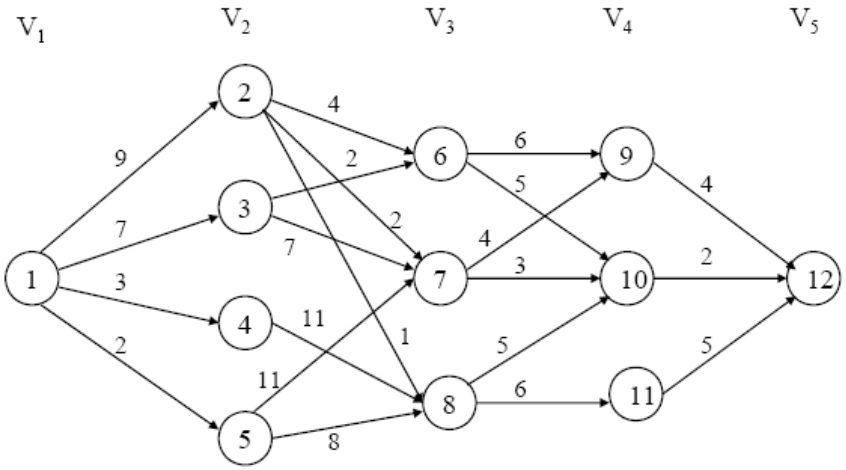


Multistage Decision Processes



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

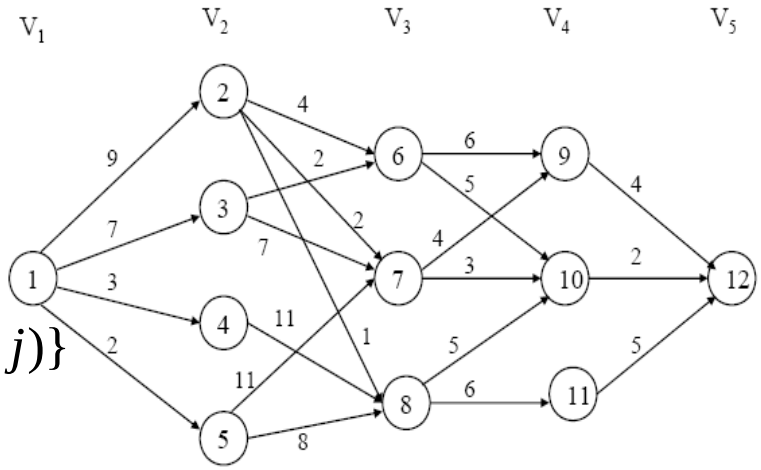
第3段 BCOST(3,6) = min{BCOST(2,2)+4,BCOST(2,3)+2} = 9
BCOST(3,7) = min{BCOST(2,2)+2,BCOST(2,3)+7,BCOST(2,5)+11} = 11
BCOST(3,8) = min{BCOST(2,4)+11, BCOST(2,5)+8} = 10

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

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

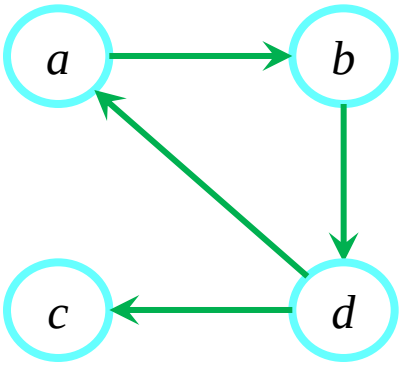
t_{ij} *in the* i^{th} *row and* j^{th} *column* = 1

↔ *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***

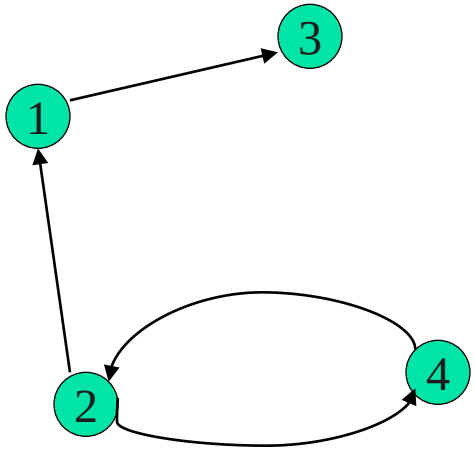


adjacent matrix

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

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

    → $r_{kj}^{(k-1)} = 1$

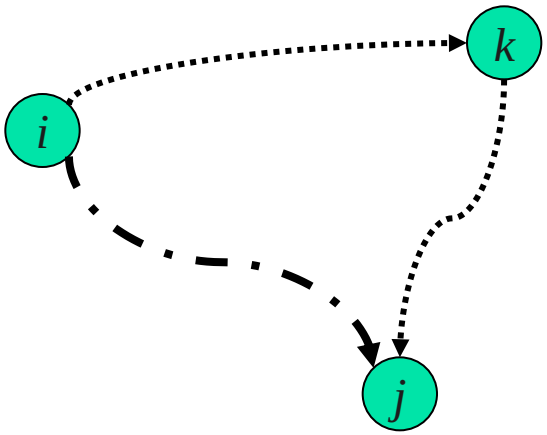
Warshall's Algorithm

Warshall's Algorithm

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

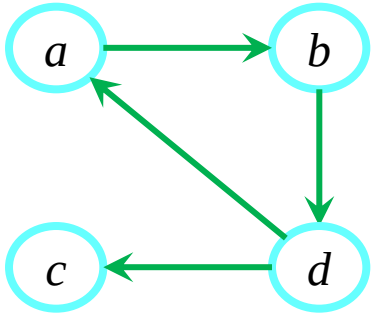
In the k^{th} stage: to determine $R^{(k)}$ is to determine if a path exists between two vertices i, j using just vertices among 1,…,k

$r_{ij}^{(k)} = 1$: { $r_{ij}^{(k-1)} = 1$ *(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

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

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

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

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.

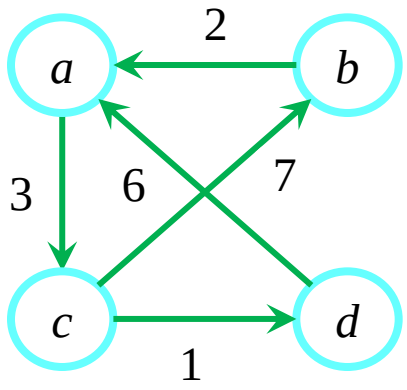
- *distance matrix:*

  *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***

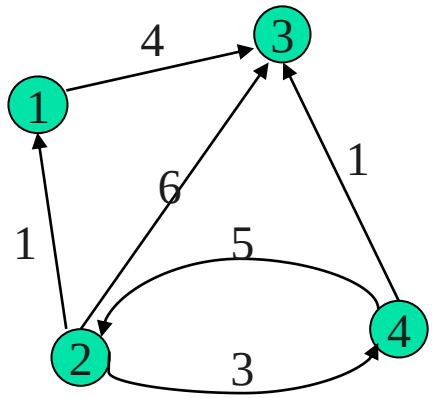


weight matrix

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

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

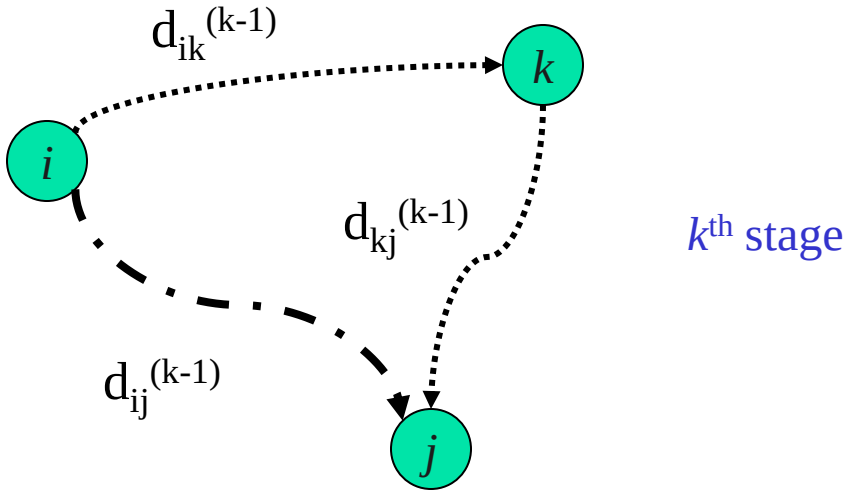
Floyd's Algorithm

- *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.

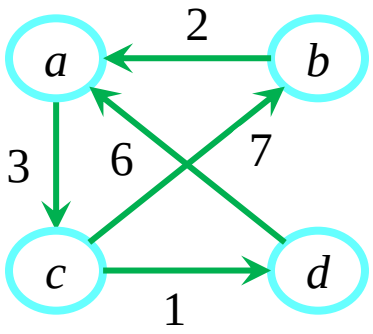
$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1, d_{ij}^{(0)} = w_{ij}$$



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*

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

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

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