Chương 4
Qui hoạch động và giải thuật tham lam

Qui hoạch động
Giải thuật tham lam

1. Qui hoạch động

Quy hoạch động (dynamic programming) **giải các bài toán bằng cách kết hợp các lời giải của các bài toán con của bài toán đang xét.**

Phương pháp này khả dụng khi các bài toán con không độc lập đối với nhau, tức là khi các bài toán con có dùng chung những bài toán "cháu" (subsubproblem).

Qui hoạch động giải các bài toán "cháu" dùng chung này một lần và lưu lời giải của chúng trong một bảng và sau đó khỏi phải tính lại khi gặp lại bài toán cháu đó.

Qui hoạch động được áp dụng cho những bài toán tối ưu hóa (optimization problem).

Bốn bước của qui hoạch động

Sự xây dựng một giải thuật qui hoạch động có thể được chia làm bốn bước:

3. Đặc trưng hóa cấu trúc của lời giải tối ưu.

2. Định nghĩa giá trị của lời giải tối ưu một cách đệ quy.

3. Tính trị của lời giải tối ưu theo kiểu từ dưới lên.

4. Cấu tạo lời giải tối ưu từ những thông tin đã được tính toán.

Thí dụ1: Nhân xâu ma trận

Cho một chuỗi $<A_1, A_2, \ldots, A_n>$ gồm *n* matrận, và ta muốn tính tích các ma trận.

$$A_1 A_2 \ldots A_n \qquad (5.1)$$

Tích của xâu ma trận này được gọi là *mở-đóng-ngoặc-đầy-đủ* (*fully parenthesized*) nếu nó là một ma trận đơn hoặc là tích của hai xâu ma trận mở-đóng-ngoặc-đầy-đủ .

<u>Thí dụ</u>: $A_1 A_2 A_3 A_4$ có thể được mở-đóng-ngoặc-đầy-đủ theo 5 cách:

$$(A_1(A_2(A_3A_4)))$$
$$(A_1((A_2A_3)A_4)$$
$$((A_1A_2)(A_3A_4))$$
$$(A_1(A_2A_3))A_4)$$
$$(((A_1A_2)A_3)A_4)$$

Cách mà ta mở đóng ngoặc một xâu ma trận có ảnh hưởng rất lớn đến chi phí tính tích xâu ma trận.

Thí dụ: A_1 10×100

A_2 100×5

A_3 5×50

$((A_1A_2)A_3)$ thực hiện

$10.100.5 + 10.5.50 = 5000 + 2500$

$= 7500$ phép nhân vô hướng.

$(A_1(A_2A_3))$ thực hiện

$100.5.50 + 10.100.50 = 25000 + 50000 = 75000$ phép nhân vô hướng.

Hai chi phí trên rất khác biệt nhau.

Phát biểu bài toán nhân xâu ma trận

Bài toán tính tích xâu ma trận:

"Cho một chuỗi $<A_1, A_2, \ldots, A_n>$ gồm *n* matrận, với mỗi $i = 1, 2, \ldots, n$, ma trận A_i có kích thước $p_{i-1} \times p_i$, ta mở-đóng-ngoặc tích này sao cho tối thiểu hóa tổng số phép nhân vô hướng".

Đây là một bài toán tối ưu hóa thuộc loại khó.

Cấu trúc của một cách mở đóng ngoặc tối ưu

<u>Bước 1</u>: Đặc trưng hóa cấu trúc của một lời giải tối ưu.
Dùng $A_{i..j}$ để ký hiệu ma trận kết quả của việc tính $A_i A_{i+1} \ldots A_j$.
Một sự mở đóng ngoặc tối ưu của tích xâu ma trận $A_1.A_2 \ldots A_n$ Tách xâu ngay tại vị trí nằm giữa A_k và A_{k+1} với một trị nguyên *k*, $1 \le k < n$. Nghĩa là, trước tiên ta tính các chuỗi ma trận $A_{1..k}$ and $A_{k+1..n}$ và rồi nhân chúng với nhau để cho ra $A_{1.n}$.

Chi phí của sự mở đóng ngoặc tối ưu này = chi phí tính $A_{l..k}$ + chí phí tính $A_{k+1..n}$, + chi phí nhân chúng lại với nhau.

Diễn tả lời giải một cách đệ quy

Ở đây, những bài toán con của ta là bài toán xác định chi phí tối ưu ứng với sự mở đóng ngoặc cho chuỗi $A_i.A_{i+1}\ldots A_j$ với $1 \le i \le j \le n$.

Đặt m[i, j] là tổng số tối thiểu các phép nhân vô hướng được đòi hỏi để tính ma trận $A_{i..j}$. Chi phí của cách rẻ nhất để tính $A_{1..n}$ sẽ được ghi ở m[1, n].

Giả sử rằng sự mở đóng ngoặc tối ưu *tách đôi* tích chuỗi $A_i A_{i+1}\ldots A_j$ tại giữa A_k and A_{k+1}, với $i \le k < j$. Thì m[i, j] bằng với chí phí tối thiểu để tính $A_{i..k}$ và $A_{k+1..j}$, cọng với chi phí để nhân hai ma trận này lại với nhau.

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

Một công thức đệ quy

Như vậy, định nghĩa đệ quy cho chi phí tối thiểu của một sự mở đóng ngoặc cho $A_i A_{i+1} \ldots A_j$ là như sau:

$$m[i, j] = 0 \qquad \text{nếu } i = j,$$
$$= \min \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j .\}$$
$$\text{nếu } i < j. \qquad (5.2)$$

Để giúp theo dõi cách tạo một lời giải tối ưu, hãy định nghĩa:

s[i, j]: trị của *k* tại đó chúng ta tách tích xâu ma trận $A_i A_{i+1} \ldots A_j$ để đạt đến một sự mở đóng ngoặc tối ưu.

Một nhận xét quan trọng

Một nhận xét quan trọng là
"*Sự mở đóng ngoặc của xâu con $A_1A_2....A_k$ bên trong sự mở đóng ngoặc tối ưu của xâu $A_1A_2…A_n$ cũng phải là một sự mở đóng ngoặc tối ưu*".

Như vậy, một lời giải tối ưu cho bài tóan tích xâu ma trận chứa đựng trong nó những lời giải tối ưu của những bài toán con.

Bước thứ hai của phương pháp qui hoạch động là định nghĩa trị của lời giải tối ưu một cách đệ quy theo những lời giải tối ưu của những bài toán con.

Tính những chi phí tối ưu

Thay vì tính lời giải dựa vào công thức cho ở (5.2) bằng một giải thuật đệ quy, chúng ta đi thực hiện Bước 3 của qui hoạch động: tính chi phí tối ưu bằng cách tiếp cận *từ dưới lên*.

Giả sử ma trận A_i có kích thước $p_{i-1} \times p_i$ với
$i = 1, 2, ..., n$.
Đầu vào là chuỗi trị số $<p_0, p_1, \ldots, p_m>$.

Thủ tục dùng một bảng m[1..n, 1..n] để lưu các chi phí m[i, j] và bảng s[1..n, 1..n] để lưu giá trị nào của vị trí *k* mà thực hiện được chi phí tối ưu khi tính m[i, j].

Thủ tục MATRIX-CHAIN-ORDER trả về hai mảng *m* và *s*.

Thủ tục tính hai bảng m và s

```
procedure MATRIX-CHAIN-ORDER(p, m, s);
begin
    n:= length[p] - 1;
    for i: = 1 to n do m[i, i] := 0;
    for l:= 2 to n do   /* l: length of the chain */
      for i:= 1 to n – l + 1 do
      begin
         j:= i + l – 1;
         m[i, j]:= ∞;                    /* initialization   */
         for k:= i to j-1 do
         begin
            q:= m[i, k] + m[k + 1, j] + p_{i-1}p_k p_j;
             if q < m[i, j] then
             begin  m[i, j]: = q; s[i, j]: = k end
         end
     end
end
```

Một thí dụ: Tính tích xâu ma trận

Vì ta định nghĩa m[i, j] chỉ cho i < j, chỉ phần của bảng *m* ở trên đường chéo chính mới được dùng.

Cho các ma trận với kích thước như sau:

A_1 30×35

A_2 35×15

A_3 15×5

A_4 5×10

A_5 10×20

A_6 20×25

Hình 4.1 trình bày bảng *m* và *s* được tính bởi thủ tục MATRIX-CHAIN-ORDER với *n* = 6.

Một thí dụ về tính tích xâu ma trân (tt.)

Mảng m

		i				
		1	*2*	*3*	*4*	*5*	*6*
	6	15125	10500	51375	3500	5000	0
	5	11875	7125	2500	1000	0	
j	*4*	9357	4375	750	0		
	3	7875	2625	0			
	2	15750	0				
	1	0					

Mảng s

Hình 5.1

		i				
		1	*2*	*3*	*4*	*5*
	6	3	3	3	5	5
	5	3	3	3	4	
j	*4*	3	3	3		
	3	1	2			
	2	1				

Một thí dụ về tính tích xâu ma trân (tt.)

$$\mathbf{m[2,5] = \min} \begin{cases} m[2,2] + m[3,5] + p.p_2p_5 = 0 + 2500 + 35.15.20 = 13000 \\ m[2,3] + m[4,5] + p_1p_2p_5 = 2625 + 100 + 35.5.30 = 7125 \\ m[2,4] + m[5m5] + p_1p_4p_5 = 4375 + 0 + 35.10.20 = 11375 \end{cases}$$

$$= 7125$$

$$\Rightarrow k = 3 \text{ for } A_{2..5}$$

Bước 4 của phương pháp qui hoạch động là tạo một lời giải tối ưu từ những thông tin đã tính toán.

Bước 4: Tạo một lời giải tối ưu

Ta dùng mảng s[1..n, 1..n] để xác định cách tốt nhất để tính tích xâu ma trận. Mỗi phần tử s[i, j] ghi trị of *k* sao cho tại đó sự mở đóng ngoặc tối ưu tách đôi xâu $A_iA_{i+1}\ldots A_j$ thành hai đoạn tại A_k và A_{k+1}.

Cho trước chuỗi ma trận $A = <A_1, A_2\ldots, A_n>$, bảng *s* và các chỉ số *i* và *j*, thủ tục đệ quy MATRIX-CHAIN-MULTIPLY sau đây tính tích xâu ma trận $A_{i..j,}$. Thủ tục trả về kết quả qua tham số AIJ.

Vơi lệnh gọi ban đầu là

MATRIX-CHAIN-MULTIPLY(A,s, 1, n, A1N)

Thủ tục sẽ trả về kết quả ma trận tích sau cùng với mảng A1N.

Tính lời giải

```
procedure MATRIX-CHAIN-MULTIPLY(A, s, i, j, AIJ);
begin
   if j > i then
   begin
      MATRIX-CHAIN-MULTIPLY(A, s, i, s[i, j], X);
      MATRIX-CHAIN-MULTIPLY(A,s, s[i, j]+1, j, Y);
      MATRIX-MULTIPLY(X, Y, AIJ);
   end
   else
     assign Ai to AIJ;
end;
```

Các thành phần của quy hoạch động

Có hai thành phần then chốt mà một bài toán tối ưu hóa phải có để có thể áp dụng qui hoạch động:

(1) *tiểu cấu trúc tối ưu* (optimal substructure) và

(2) *các bài toán con trùng lắp* (overlapping subproblems).

<u>Tiểu cấu trúc tối ưu</u>

Một bài toán có tính chất tiểu cấu trúc tối ưu nếu lời giải tối ưu chứa trong nó những lời giải tối ưu của những bài toán con.

Những bài toán con trùng lắp

Khi một giải thuật đệ quy gặp lại cùng một bài toán con nhiều lần, ta bảo rằng bài toán tối ưu hóa có những bài toán con trùng lắp.

Giải thuật quy hoạch động lợi dụng những bài toán con trùng lắp bằng cách giải mỗi bài toán con một lần, cất lời giải vào trong một bảng mà bảng này sẽ được tham khảo đến khi cần.

Các giải thuật đệ quy *làm việc từ trên xuống* trong khi các giải thuật quy hoạch động *làm việc từ dưới lên*, Cách sau hữu hiệu hơn .

Thí dụ 2: Bài toán chuỗi con chung dài nhất

Một *chuỗi con* (*subsequence*) của một chuỗi (sequence) là chuỗi ấy sau khi bỏ đi một vài phần tử.

Thí dụ: Z = <B, C, D, B> là một chuỗi con của X = <A, B, C, B, D, A, B> với chuỗi chỉ số <2, 3, 5, 7>.

Cho hai chuỗi X và Y, ta bảo Z là *chuỗi con chung* (*common subsequence*) của X và Y nếu Z là một chuỗi con của cả hai chuỗi X và Y.

Trong bài toán chuỗi con chung dài nhất, ta được cho hai chuỗi $X = \langle x_1, x_2, \ldots, x_m \rangle$ và $Y = \langle y_1, y_2, \ldots, y_n \rangle$ và muốn tìm *chuỗi con chung dài nhất* (LCS) của X và Y.

Tiểu cấu trúc tối ưu của bài toán chuỗi con chung dài nhất

Thí dụ: X = <A, B, C, B, D, A, B> và Y = <B, D, C, A, B, A>
<B, D, A, B> là LCS của X and Y.

Cho chuỗi X = $<x_1, x_2, \ldots, x_m>$, ta định nghĩa *tiền tố thứ i* của X, với $i = 0, 1, \ldots, m$, là $X_i = <x_1, x_2, \ldots, x_i>$.

Định lý 4.1

Cho $X = <x_1, x_2, \ldots, x_m>$ và $Y = <y_1, y_2, \ldots, y_n>$ là những chuỗi, và $Z = <z_1, z_2, \ldots, z_k>$ là LCS của X và Y.

1. Nếu $x_m = y_n$ thì $z_k = x_m = y_n$ và Z_{k-1} là LCS của X_{m-1} và Y_{n-1}.
2. Nếu $x_m \neq y_n$, thì $z_k \neq x_m$ hàm ý Z là LCS của X_{m-1} và Y.
3. Nếu $x_m \neq y_n$, thì $z_k \neq y_n$ hàm ý Z là LCS của X và Y_{n-1}.

Lời giải đệ quy

Để tìm một LCS của X và Y, ta có thể cần tìm LCS của X và Y_{n-1} và LCS của X_{m-1} và Y. Nhưng mỗi trong hai bài toán con này có những bài toán "cháu" để tìm X_{m-1} và Y_{n-1}.

Gọi c[i, j] là chiều dài của LCS của hai chuỗi X_i và Y_j. Nếu *i* = 0 hay *j* = 0, thì LCS có chiều dài 0. Tính chất tiểu cấu trúc tối ưu của bài toán LCS cho ra công thức đệ quy sau:

$$c[i,j] = \begin{cases} 0 & \text{nếu } i = 0 \text{ hay } j = 0 \\ c[i-1, j-1]+1 & \text{nếu } i, j > 0 \text{ và } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{nếu } i, j > 0 \text{ và } x_i \neq y_j \end{cases} \quad (5.3)$$

Tính chiều dài của một LCS

Dựa vào phương trình (5.3), ta có thể viết một giải thuật đệ quy để tìm chiều dài của một LCS của hai chuỗi. Tuy nhiên, chúng ta dùng qui hoạch động để tính lời giải theo cách *từ dưới lên*.

Thủ tục LCS-LENGTH có hai chuỗi $X = \langle x_1, x_2, \ldots, x_m \rangle$ và $Y = \langle y_1, y_2, \ldots, y_n \rangle$ là đầu vào.

Thủ tục lưu các trị c[i, j] trong bảng c[0..m, 0..n]. Nó cũng duy trì bảng b[1..m, 1..n] để đơn giản hóa việc tạo lời giải tối ưu.

```
procedure LCS-LENGTH(X, Y)
begin
    m: = length[X];  n: = length[Y];
    for i: = 1 to m do c[i, 0]: = 0;   for j: = 1 to n do c[0, j]: = 0;
    for i: = 1 to m do
      for j: = 1 to n do
        if x_i = y_j then
        begin   c[i, j]: = c[i-1, j-1] + 1; b[i, j]: = "↖"   end
        else if c[i – 1, j] > = c[i, j-1] then
        begin  c[i, j]: = c[i – 1, j]; b[i, j]: = "↑"   end
        else
        begin c[i, j]: = c[i, j-1]; b[i, j]: = "←"    end
end;
```

Hình 4.2 sau đây trình bày ma trận *c* của thí dụ.

x_i \ y_j		B	D	C	A	B	A
	0	0	0	0	0	0	0
A	0	0 ↑	0 ↑	0 ↑	1 ↖	1 ←	1 ↖
B	0	1 ↖	1 ←	1 ←	1 ↑	2 ↖	2 ←
C	0	1 ↑	1 ↑	2 ↖	2 ←	2 ↑	2 ↑
B	0	1 ↖	1 ↑	2 ↑	2 ↑	3 ↖	3 ←
D	0	1 ↑	2 ↖	2 ↑	2 ↑	3 ↑	3 ↑
A	0	1 ↑	2 ↑	2 ↑	3 ↖	3 ↑	4 ↖
B	0	1 ↖	2 ↑	2 ↑	3 ↑	4 ↖	4 ↑

Hình 5.2

Tạo chuỗi con chung dài nhất

Bảng *b* có thể được dùng để tạo một LCS của

$X = \langle x_1, x_2, \ldots x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$

Thủ tuc đệ quy sau đây in ra một LCS của X và Y. Lệnh gọi đầu tiên là PRINT-LCS(b, X, n, m).

```
procedure PRINT-LCS(b, X, i, j)
 begin
   if i <> 0 and j <> 0 then
      if b[i, j] = " ↖ " then
      begin PRINT-LCS(b, X, i- 1 , j - 1 ) ;
       print x_i
      end
      else if b[i,j] = "↑" then
            PRINT-LCS (b, X, i-1, j)
       else PRINT-LCS(b, X, i, j-1)
end;
```

Thời gian tính toán của thủ tục PRINT-LCS là O(m+n), vì ít nhất *i* hay *j* giảm một đơn vị trong mỗi chặng của đệ quy.

Thí dụ 3. Bài toán cái túi (Knapsack)

"Một kẻ trộm đột nhập vào một cửa hiệu tìm thấy có *n* mặt hàng có trọng lượng và giá trị khác nhau, nhưng y chỉ mang theo một cái túi có sức chứa về trọng lượng tối đa là M. Bài toán cái túi là tìm một tổ hợp các mặt hàng mà kẻ trộm nên bỏ vào cái túi để đạt một giá trị cao nhất với những món hàng mà y mang đi."

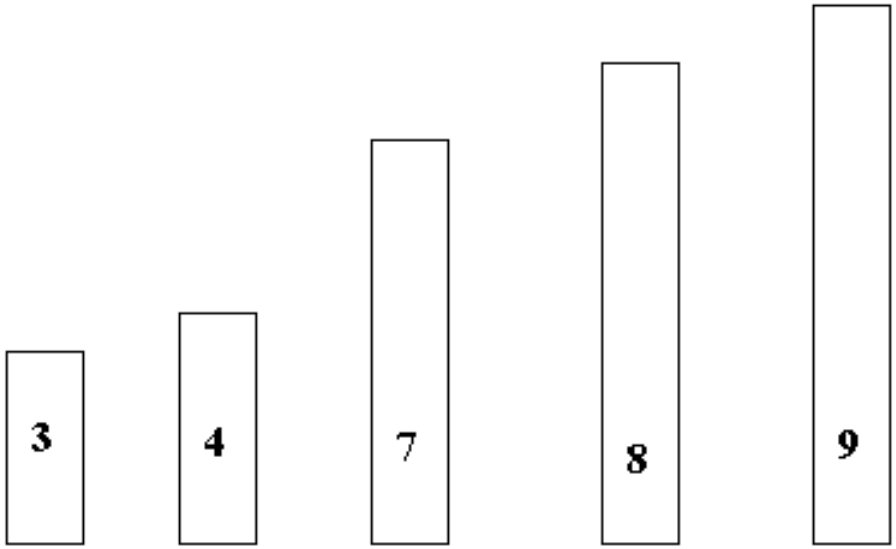
Bài toán này có thể giải bằng *qui hoạch động* bằng cách dùng hai bảng *cost* và *best* sau đây:

cost[i]* chứa giá trị tối đa mà có thể thực hiện được với một cái túi có sức chứa *i

$$cost[i] = cost[i - size[j]] + val[j]$$

***best[i]* chứa mặt hàng cuối cùng bỏ vào túi nhằm đạt được giá trị tối đa.**

Một thí dụ của bài toán cái túi



value	**4**	**5**	**10**	**11**	**13**
name	**A**	**B**	**C**	**D**	**E**

M = 17

Hình 5.3 Một thí dụ của bài toán cái túi

Giải thuật quy hoạch động cho bài toán cái túi

M: sức chứa tối đa của cái túi

```
for i: = 0 to M do cost[i]: = 0;
for j: = 1 to N do   /* each of item type   */
begin
   for i:= 1 to M do  /* i means capacity  */
      if i – size[j] > = 0 then
         if cost[i] < (cost[i – size[j]] + val[j]) then
         begin
              cost[i]: = cost[i – size[j]] + val[j];      best[i]: = j
          end;
end;
```

Một thể hiện của cái túi

K	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
j=1																	
cost[k]	0	0	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
best[k]			A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
j=2																	
cost[k]	0	0	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
best[k]			A	B	B	A	B	B	A	B	B	A	B	B	A	B	B
j=3																	
cost[k]	0	0	4	5	5	8	10	10	12	14	15	16	18	18	20	22	24
best[k]			A	B	B	A	C	B	A	C	C	A	C	C	A	C	C
j=4																	
cost[k]	0	0	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
best[k]			A	B	B	A	C	D	A	C	C	A	C	C	D	C	C
j=5																	
cost[k]	0	0	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24
best[k]			A	B	B	A	C	D	E	C	C	E	C	C	D	E	C

Hình 5.4 Các mảng *cost* và *best* của một thí dụ bài toán cái túi

Ghi Chú:

Bài toán cái túi có thể dễ dàng giải được nếu M không lớn, nhưng khi M lớn thì thời gian chạy trở nên không thể chấp nhận được.

Phương pháp này không thể làm việc được khi M và trọng lượng/kích thước là những số thực thay vì số nguyên.

Tính chất 4.1.1 ***Giải thuật qui hoạch động để giải bài toán cái túi có thời gian chạy tỉ lệ với NM*****.**

Thí dụ 4: Giải thuật Warshall và giải thuật Floyd

Tính bao đóng truyền

Trong đồ thị có hướng, chúng ta quan tâm đến tập đỉnh mà *đến được* từ một đỉnh nào đó bằng cách duyệt các cạnh trong đồ thị theo một hướng đã được ấn định.

Một tác vụ mà ta muốn thực hiện là "thêm một cạnh từ *x* đến *y* nếu tồn tại một cách nào đó để đi từ *x* đến *y*"

Đồ thị tạo ra bằng cách thêm tất cả các cạnh có tính chất trên được gọi là *bao đóng truyền* của đồ thị.

Vì đồ thị bao đóng truyền thì thường là đồ thị dày, do đó ta nên dùng cách biểu diễn ma trận kế cận.

Giải thuật Warshall

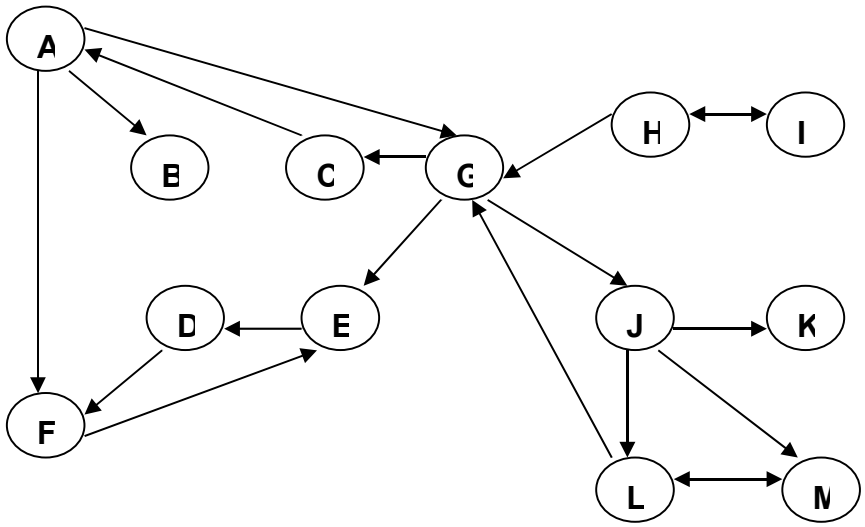
Có một giải thuật đơn giản để tính bao đóng truyền của một đồ thị được biểu diễn bằng ma trận kế cận.

```
for y : = 1 to V do
   for x : = 1 to V do
      if a[x, y] then
         for j: = 1 to V do
            if a[y, j] then a[x, j]: = true;
```

S. Warshall đề ra giải thuật này năm 1962, dựa trên một quan sát đơn giản: "*Nếu tồn tại một cách để đi từ nút x đến nút y và cách để đi từ nút y đến nút j, thì sẽ có cách để đi từ nút x đến nút j*."

Một thí dụ tính bao đóng truyền

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	1	0	0	0	1	1	0	0	0	0	0	0
B	0	1	0	0	0	0	0	0	0	0	0	0	0
C	1	0	1	0	0	0	0	0	0	0	0	0	0
D	0	0	0	1	0	1	0	0	0	0	0	0	0
E	0	0	0	1	1	0	0	0	0	0	0	0	0
F	0	0	0	0	1	1	0	0	0	0	0	0	0
G	0	0	1	0	1	0	1	0	0	1	0	0	0
H	0	0	0	0	0	0	1	1	1	0	0	0	0
I	0	0	0	0	0	0	0	1	1	0	0	0	0
J	0	0	0	0	0	0	0	0	0	1	1	1	1
K	0	0	0	0	0	0	0	0	0	0	1	0	0
L	0	0	0	0	0	0	0	0	0	0	0	1	1
M	0	0	0	0	0	0	0	0	0	0	0	1	1



Ma trận kế cận ứng với bước khởi đầu của giải thuật Warshall

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	1	1	1	1	1	1	0	0	1	1	1	1
B	0	1	0	0	0	0	0	0	0	0	0	0	0
C	1	1	1	1	1	1	1	0	0	1	1	1	1
D	0	0	0	1	1	1	0	0	0	0	0	0	0
E	0	0	0	1	1	1	0	0	0	0	0	0	0
F	0	0	0	1	1	1	0	0	0	0	0	0	0
G	1	1	1	1	1	1	1	0	0	1	1	1	1
H	1	1	1	1	1	1	1	1	1	1	1	1	1
I	1	1	1	1	1	1	1	1	1	1	1	1	1
J	1	1	1	1	1	1	1	0	0	1	1	1	1
K	0	0	0	0	0	0	0	0	0	0	1	0	0
L	1	1	1	1	1	1	1	0	0	1	1	1	1
M	1	1	1	1	1	1	1	0	0	1	1	1	1

Ma trận kế cận ứng với bước cuối cùng của giải thuật Warshall

<u>Tính chất 5</u>.3.1 ***Giải thuật Warshall tính bao đóng truyền với chi phí $O(V^3)$.***

Giải thuật Warshall thể hiện sự áp dụng chiến lược quy hoạch động vì sự tính toán căn cứ vào một hệ thức truy hồi (5.4) nhưng lại không xây dựng thành giải thuật đệ quy. Thay vào đó là một giải thuật lặp với sự hỗ trợ của một ma trận để lưu trữ các kết quả trung gian.

Giải thích giải thuật Warshall

- Giải thuật Warshall lặp *V* bước trên ma trận kế cận *a,* tạo ra một loại những ma trận:

$$a^{(0)}, a^{(y-1)}, a^{(y)}, \ldots, a^{(V)} \qquad (5.4)$$

- Ý tưởng chính của giải thuật là ta có thể tính tất cả các phần tử trong mỗi ma trận $a^{(y)}$ từ ma trận đi trước nó $a^{(y-1)}$ trong loạt ma trận (4.1)
- Sau bước lặp thứ *y*, a[x, j] sẽ bằng 1 nếu và chỉ nếu có bất kỳ lối đi nào từ đỉnh *x* đến đỉnh *j* với những đỉnh trung gian mang chỉ số không lớn hơn *y*. Nghĩa là, *x* và *j* có thể là bất kỳ đỉnh nào nhưng những đỉnh trung gian trên lối đi phải nhỏ hơn hay bằng *y*.

- Tại bước lặp thứ *y*, ta tính các phần tử của ma trận *a* bằng công thức sau:

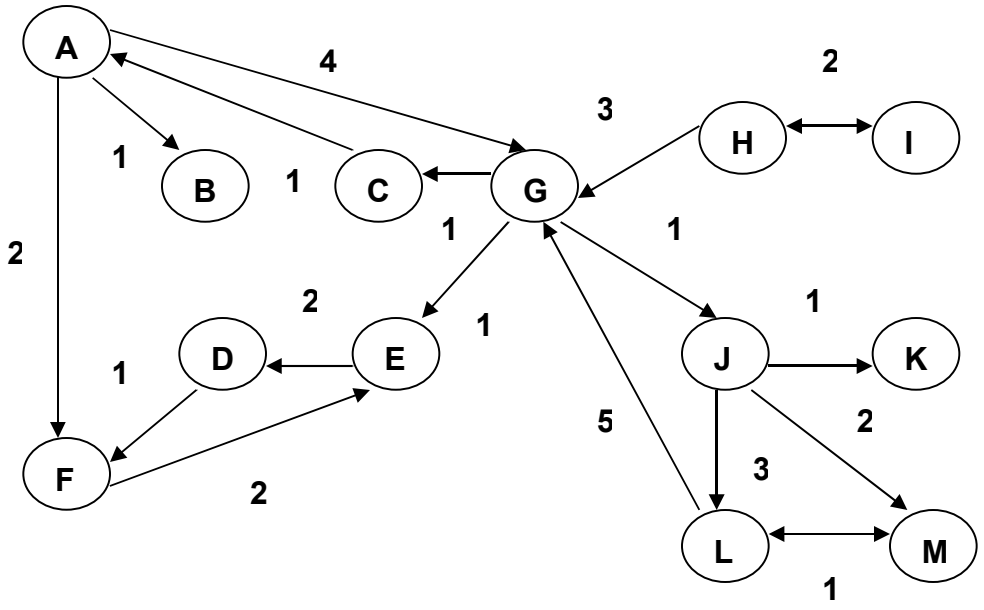
$$a^{y}[x,j] = a^{y-1}[x,j] \text{ or } (a^{y-1}[x, y] \text{ and } a^{y-1}[y, j]) \qquad (5.5)$$

Chỉ số *y* chỉ trị của một phần tử trong ma trận *a* sau bước lặp thứ *y*.

Giải thuật Floyd cho bài toán các lối đi ngắn nhất

Đối với đồ thị có trọng số (có hướng hoặc không) ta có thể muối xây dựng một ma trận cho phép người ta tìm được lối đi ngắn nhất từ *x* đến *y* đối với mọi cặp đỉnh. Đấy là bài toán *những lối đi ngắn nhất cho mọi cặp đỉnh* (*all-pairs shortest path problem*).

Hình 5.7



Giải thuật Floyd

Có thể dùng một phương pháp tương tự như phương pháp Warshall, mà được đưa ra bởi R. W. Floyd:

```
for y : = 1 to V do
   for x : = 1 to V do
      if a [x, y] > 0 then
         for j: = 1 to V do
            if a [y, j] > 0 then
               if (a[x, j] = 0) or (a[x, y] + a[y, j] < a [x, j])
               then
                  a[x, j] = a[x, y] + a[y, j];
```

Một thí dụ dùng giải thuật Floyd (cho đồ thi hình 5.7)

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	0	1	0	0	0	2	4	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0	0	0	0	0	0
C	1	0	0	0	0	0	0	0	0	0	0	0	0
D	0	0	0	0	0	1	0	0	0	0	0	0	0
E	0	0	0	2	0	0	0	0	0	0	0	0	0
F	0	0	0	0	2	0	0	0	0	0	0	0	0
G	0	0	1	0	1	0	0	0	0	1	0	0	0
H	0	0	0	0	0	0	3	0	1	0	0	0	0
I	0	0	0	0	0	0	0	1	0	0	0	0	0
J	0	0	0	0	0	0	0	0	0	0	1	3	2
K	0	0	0	0	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	5	5	0	0	0	0	0	1
M	0	0	0	0	0	0	0	0	0	0	1	0	

Ma trận kế cận ứng với bước khởi đầu của giải thuật Floyd

Chú ý: Các phần tử trên đường chéo đều bằng 0.

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	6	1	5	6	4	2	4	0	0	5	6	8	7
B	0	0	0	0	0	0	0	0	0	0	0	0	0
C	1	2	6	7	5	3	5	0	0	6	7	9	8
D	0	0	0	5	3	1	0	0	0	0	0	0	0
E	0	0	0	2	5	3	0	0	0	0	0	0	0
F	0	0	0	4	2	5	0	0	0	0	0	0	0
G	2	3	1	3	1	4	6	0	0	1	2	4	3
H	5	6	4	6	4	7	3	2	1	4	5	7	6
I	6	7	5	7	5	8	4	1	2	5	6	8	7
J	10	11	9	11	9	12	8	0	0	9	1	3	2
K	0	0	0	0	0	0	0	0	0	0	0	0	0
L	7	8	6	8	6	9	5	0	0	6	7	2	1
M	8	9	7	9	7	10	6	0	0	7	8	1	2

Ma trận kế cận ứng với bước cuối của giải thuật Floyd

Tính chất 5.3.2 ***Giải thuật Floyd để giải bài toán những lối đi ngắn nhất giữa những cặp có độ phức tạp tính toán $O(V^3)$.***

Giải thích giải thuật Floyd

Giải thuật Floyd lặp *V* bước trên ma trận kế cận *a*, tạo ra một loại những ma trận:

$$a^{(0)}, a^{(y-1)}, a^{(y)}, \ldots, a^{(V)} \qquad (5.6)$$

Ý tưởng chính của giải thuật là ta có thể tính tất cả các phần tử trong mỗi ma trận $a^{(y)}$ từ ma trận đi trước nó, $a^{(y-1)}$ trong loạt ma trận.

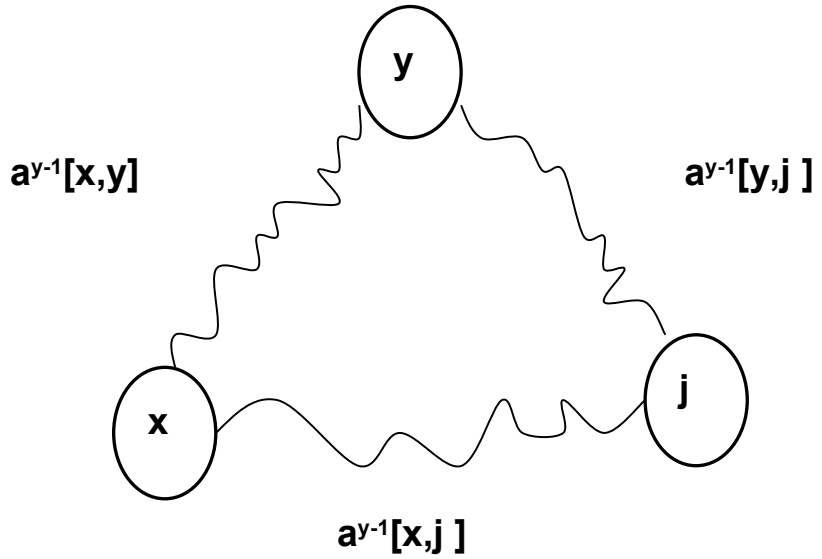
Sau bước lặp thứ *y*, a[x, j] sẽ chứa chiều dài nhỏ nhất của bất kỳ lối đi nào từ đỉnh *x* đến đỉnh *j* mà đi qua những đỉnh trung gian không mang chỉ số lớn hơn *y*. Nghĩa là, *x* và *j* có thể có là bất kỳ đỉnh nào nhưng những đỉnh trung gian trên lối đi phải nhỏ hơn hay bằng *y*.

Tại bước lặp thứ *y*, ta tính các phần tử của ma trận *a* bằng công thức sau:

$$a^{y}[x,j] = \min(a^{y-1}[x,j], a^{y-1}[x, y] + a^{y-1}[y, j]) \quad (5.7)$$

Chỉ số *y* chỉ trị của một phần tử trong ma trận *a* sau bước lặp thứ *y*.

Công thức này được minh họa bằng hình vẽ sau đây.



Giải thuật Floyd thể hiện sự áp dụng chiến lược *quy hoạch động* cho một bài toán tối ưu hóa vì sự tính toán căn cứ vào một hệ thức truy hồi (5.6) nhưng lại không xây dựng thành giải thuật đệ quy. Thay vào đó là một giải thuật lặp với sự hỗ trợ của một ma trận để lưu trữ các kết quả trung gian.

Cải tiến giải thuật Floyd

- **Ta thường muốn biết lối đi ngắn nhất từ một đỉnh đến một đỉnh khác bao gồm những đỉnh trung giannào.**
- **Một cách để thực hiện điều này là dùng thêm một ma trận *P*, với *P[i,j]* chứa đỉnh *k* mà khiến giải thuật Floyd tìm được giá trị nhỏ nhất cho *a[i,j].***
- **Giải thuật Floyd cải tiến sẽ như sau:**

```
for i := 1 to V do
   for j:= 1 to V do
       P[i,j] := 0;
for i := 1 to V do
      a[i,i]:= 0;
```

```
for y : = 1 to V do
  for x : = 1 to V do
    if a [x, y] > 0 then
      for j: = 1 to V do
        if a [y, j] > 0 then
          if (a[x, j] = 0) or (a[x, y] + a[y, j] < a [x, j]) then
            begin
              a[x, j] = a[x, y] + a[y, j];
              P[x,j] := k;
            end
```

Để in ra những đỉnh trung gian trên lối đi ngắn nhất từ đỉnh *x* đến đỉnh *j*, ta gọi thủ tục *path(x,j)* với *path* là một thủ tục đệ quy được cho ở hình bên.

```
procedure path(x, j: int)
var k : int;
begin
   k := P[x,j];
   if k = 0 then return;
   path(x,k); writeln(k); path(k,j);
end
```

2. Giải thuật tham lam

Các giải thuật tối ưu hóa thường đi qua một số bước với một tập các khả năng lựa chọn tại mỗi bước. Một giải thuật tham lam thường chọn một khả năng mà xem như tốt nhất tại lúc đó.

Tức là, giải thuật chọn một khả năng tối ưu cục bộ với hy vọng sẽ dẫn đến một lời giải tối ưu toàn cục.

Vài thí dụ của giải thuật tham lam:

- **Bài toán xếp lịch cho các hoạt động**
- **Bài toán cái túi dạng phân số**
- **Bài toán mã Huffman**
- **Giải thuật Prim để tính cây bao trùm tối thiểu**

Bài toán xếp lịch cho các hoạt động (Activity-Selection Problem)

Giả sử ta có một tập S = {1, 2, …, n} gồm *n* hoạt động mà cùng muốn sử dụng cùng một *tài nguyên*, thí dụ như một giảng đường, mà chỉ có thể được dùng bởi một hoạt động tại một lúc.

Mỗi hoạt động *i* có *thời điểm bắt đầu* s_i và một *thời điểm kết thúc* f_i, mà $s_i \leq f_i$. Nếu được lựa chọn, hoạt động *i* diễn ra trong thời khoảng $[s_i, f_i)$. Hoạt động *i* và *j* là *tương thích* nếu thời khoảng $[s_i, f_i)$ và $[s_j, f_j)$ không phủ lấp lên nhau (tức là, *i* và *j* là tương thích nếu $s_i >= f_j$ hay $s_j >= f_i$).

Bài toán xếp lịch các hoạt động là chọn ra một chuỗi các hoạt động tương thích với nhau và có số hoạt động nhiều nhất.

Giải thuật tham lam cho bài toán xếp lịch các hoạt động

Trong thủ tục áp dụng giải thuật tham lam để giải bài toán xếp lịch các hoạt động, ta giả sử rằng các hoạt động nhập vào được sắp theo thứ tự tăng của thời điểm kết thúc:
$f_1 \leq f_2 \leq \ldots \leq f_n$.

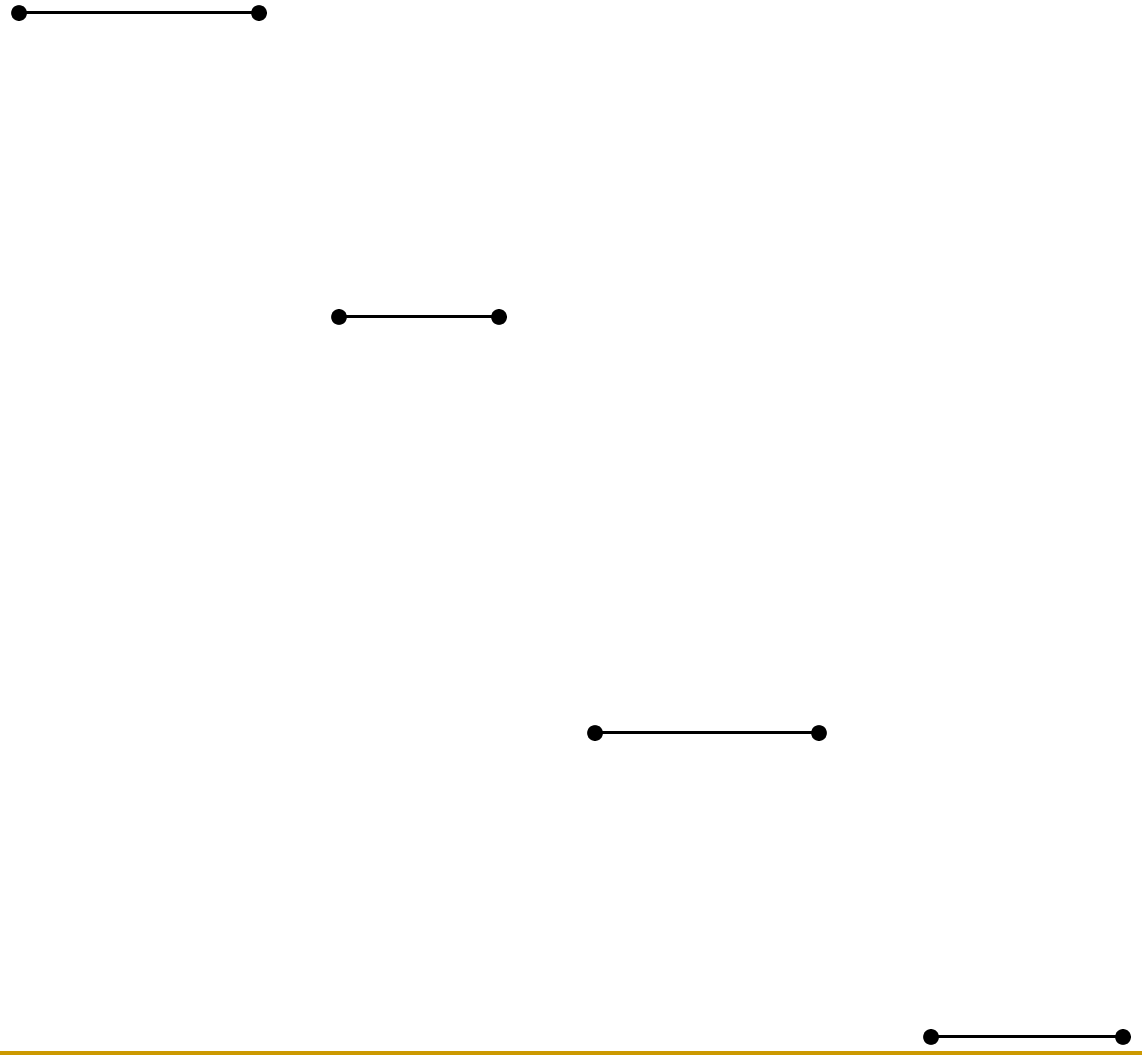
```
procedure GREED-ACTIVITY-SELECTOR(S, f) ;        /* s is the array keeping the set of activities and f is the array keeping the finishing times */
begin
   n := length[s];   A := {1};  j: = 1;
   for i: = 2 to n do
      if s_i >= f_j then /* i is compatible with all  activities in A */
      begin  A: = A ∪ {i}; j: = i  end
end
```

Thủ tục Greedy-activity-selector

Hoạt động được chọn bởi thủ tục GREEDY-ACTIVITY-SELECTER thường là hoạt động với *thời điểm kết thúc sớm nhất* mà có thể được xếp lịch một cách hợp lệ. Hoạt động được chọn theo cách "tham lam" theo nghĩa nó sẽ để lại cơ hội để xếp lịch cho được nhiều hoạt độngkhác.

Giải thuật tham lam không nhất thiết đem lại lời giải tối ưu. Tuy nhiên thủ tục GREEDY-ACTIVITY-SELECTOR thường tìm được một lời giải tối ưu cho một thể hiện của bài toán xếp lịch các hoạt động.

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



Hình 5.5 Một thí dụ của bài toán xếp lịch

Hai thành phần chính của giải thuật tham lam

Có hai tính chất mà các bài toán phải có để có thể áp dụng giải thuật tham lam là: (1) tính chất lựa chọn tham lam và (2) tiểu cấu trúc tối ưu.

Lựa chọn được thực hiện bởi giải thuật tham lam tùy thuộc vào những lựa chọn đã làm cho đến bây giờ, nhưng nó không tùy thuộc vào bất kỳ lựa chọn trong tương lai hay những lời giải của những bài toán con. Như vậy, một giải thuật tham lam tiến hành theo kiểu *từ trên xuống*, thực hiện mỗi lúc một lựa chọn tham lam.

Tính chất tiểu cấu trúc tối ưu (Optimal Substructure)

Một bài tóan có tính chất tiểu cấu trúc tối ưu nếu một lời giải tối ưu chứa trong nó những lời giải tối ưu cho những bài toán con.

Giải thuật tham lam so sánh với quy hoạch động

Sự khác biệt giữa qui hoạch động và giải thuật tham lam khi dùng để giải bài toán tối ưu là rất tế nhị.

Bài toán cái túi dạng 0-1 được định nghiã như sau:

'‘Một kẻ trộm đột nhập vào một cửa hiệu tìm thấy *n* loại món hàng có trọng lượng và giá trị khác nhau (món hàng thứ *i* có giá trị v_i đô la và trọng lượng w_i), nhưng chỉ có một cái túi với sức chứa về trọng lượng là *M* để mang các món hàng. Bài toán cái túi là tìm một tổ hợp các món hàng mà kẻ trộm nên chọn bỏ vào cái túi để đạt được một giá trị tối đa với những món hàng mà y lấy đi.”.

Bài toán này được gọi là *bài toán cái túi dạng 0-1* vì mỗi món hàng thì hoặc là lấy đi hoặc là bỏ lại, kẻ trộm không thể lấy đi chỉ một phần của món hàng.

Bài toán cái túi dạng phân số (Fractional knapsack problem)

Trong bài toán cái túi dạng phân số, tình tiết cũng như vậy, nhưng kẻ trộm có thể lấy đi một phần của một món hàng.

Cả hai bài toán đều có tính chất tiểu cấu trúc tối ưu.

- **Đối với bài toán cái túi dạng 0-1, xét một tổ hợp nặng M ký mà đem lại giá trị cực đại. Nếu ta lấy món hàng thứ *j* ra khỏi túi, những món hàng còn lại cũng là tổ hợp đem lại giá trị lớn nhất ứng với trọng lượng tối đa M - w_j mà kẻ trộm có thể lấy đi từ *n-1* loại mặt hàng trừ mặt hàng thứ *j*.**

- **Đối với bài toán cái túi dạng phân số, xét trường hợp khi ta lấy ra khỏi túi w_j -*w* ký của mặt hàng thứ *j*, những món hàng còn lại cũng là tổ hợp đem lại giá trị lớn nhất ứng với trọng lượng M – (w_j –*w*) mà kẻ trộm có thể lấy đi từ *n-1* loại mặt hàng trừ mặt hàng thứ *j*.**

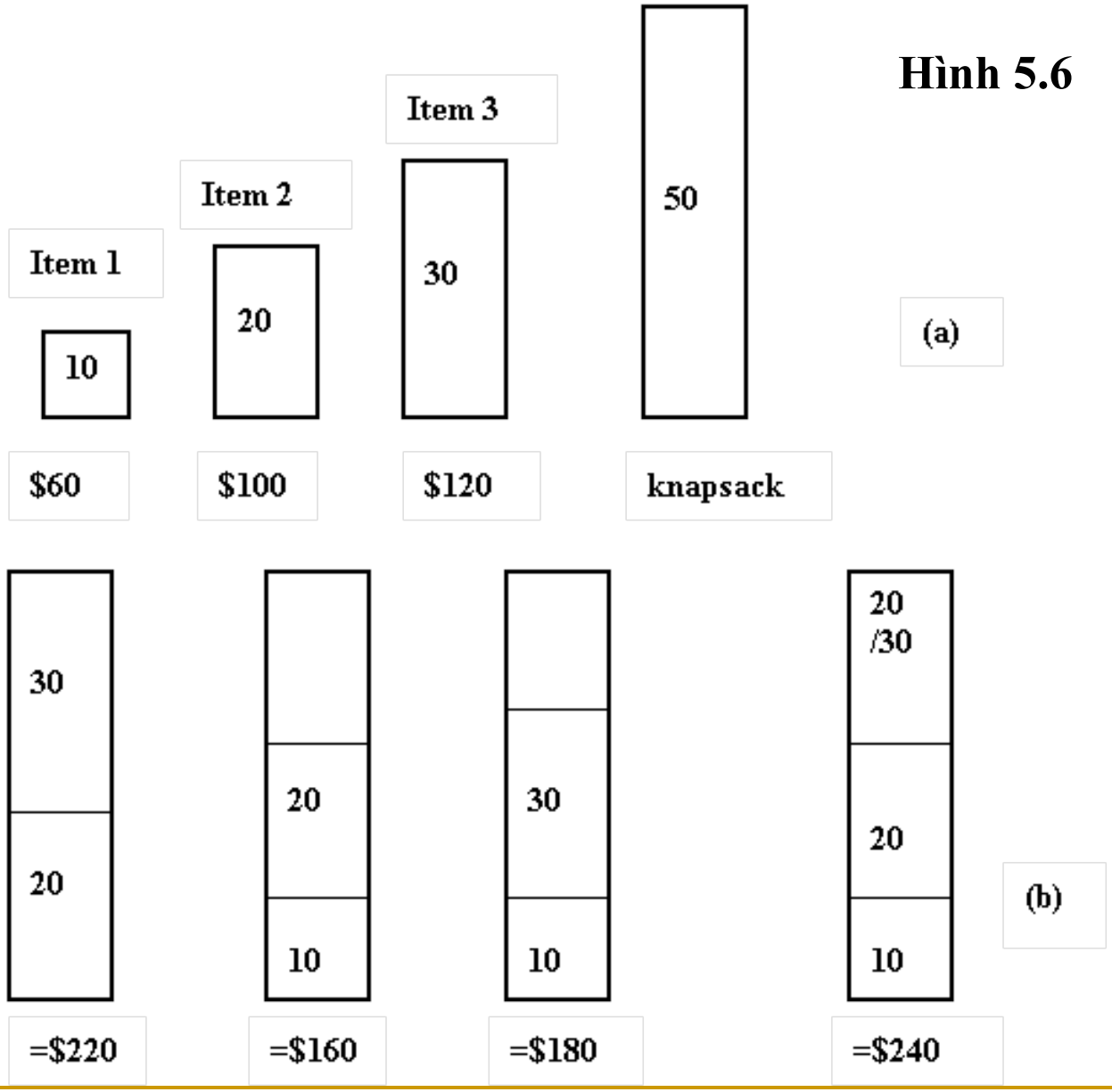
Bài toán cái túi dạng phân số (tt.)

Ta dùng *giải thuật tham lam* cho bài toán cái túi dạng phân số và *qui hoạch động* cho bài toán cái túi dạng 1-0.

Để giải bài toán cái túi dạng phân số, trước tiên ta tính hệ số giá trị tiền trên một đơn vị trọng lượng (v_i/w_i) của từng mặt hàng.

Kẻ trộm bắt đầu bằng cách lấy càng nhiều càng tốt mặt hàng có hệ số v_i/w_i lớn nhất. Khi loại mặt hàng này đã cạn mà kẻ trộm còn có thể mang thêm được nữa thì y sẽ lấy càng nhiều càng tốt mặt hàng có hệ số v_i/w_i lớn nhì và cứ như thể cho đến khi y không còn có thể mang thêm nữa.

Hình 5.6


Item 1
10
$60
Item 2
20
$100
Item 3
30
$120
50
knapsack
(a)
30
20
=$220
20
10
=$160
30
10
=$180
20
/30
20
10
=$240
(b)

```
procedure GREEDY_KNAPSACK(V, W, M, X, n);
/* V, W are the arrays contain the values and weights of n objects
ordered so that V_i/W_i ≥ V_{i+1}/W_{i+1}. M is the knapsack capacity and X is
solution vector */
var rc: real; i: integer;
begin
   for i:= 1 to n do  X[i]:= 0;
   rc := M ;  // rc = remaining knapsack
                                  capacity  //
   for i := 1 to n do
   begin
      if W[i] > rc  then  exit;
      X[i] := 1;  rc := rc – W[i]
   end;
   if i ≤ n then X[i] := rc/W[i]
end
```

Bỏ qua thời gian sắp thứ tự các món hàng, giải thuật này có độ phức tạp O(n).

Mã Huffman

Chủ đề này liên quan đến vấn đề *nén file* (*file compression*). Các mã Huffman là kỹ thuật được dùng phổ biến và rất hữu hiệu cho việc nén dữ liệu, tiết kiệm từ 20% đến 90% là điển hình.

Bước đầu tiên của việc xây dựng mã Huffman là đếm *tần số xuất hiện* (frequency) của mỗi ký tự trong tập tin được mã hóa.

Giả sử ta có một tập tin 100000 ký tự mà ta muốn lưu trữ ở dạng nén.

	a	b	c	d	e	f
Tần số	45	13	12	16	9	5
Mã có chiều dài cố định	000	001	010	011	100	101
Mã có chiều dài thay đổi	0	101	100	111	1101	1100

Xét bài toán thiết kế *một mã nhị phân cho ký tự* (binary character code) theo đó mỗi ký tự được diễn tả bằng một tràng bit nhị phân.

Nếu ta dùng một *mã có chiều dài cố định* (3 bit) để diễn tả 6 ký tự:

a = 000, b = 001, . . . , f = 101

Thì cần tất cả 300000 bit để mã hóa toàn tập tin.

Mã có chiều dài thay đổi

Một *mã có chiều dài thay đổi* (*variable-length code*) có thể làm việc tốt hơn một mã có chiều dài cố định, nó cho những ký tự hay xuất hiện những mã ngắn và những ký tự ít hay xuất hiện những mã dài hơn.

a = 0, b = 101, . . . f = 1100

Mã này đòi hỏi:

(45. *1* + 13 .3 + 12.3 + 16.3 + 9.4 + 5.4).1000 = 224000 bits

để biểu diễn tập tin, tiết kiệm được ≈ 25 %.

Và đấy cũng chính là mã tối ưu cho tập tin này.

Mã phi-tiền tố (Prefix-code)

Ở đây ta chỉ xét những cách mã hóa mà không có mã của ký tự nào là *tiền tố* (prefix) của mã của một ký tự khác. Những cách mã hóa như vậy được gọi là *mã phi tiền tố* (*prefix-free-code*) hay *mã tiền tố* (*prefix-code*).

Có thể chứng minh được rằng sự nén tin tối ưu được thực hiện bởi một cách mã hóa ký tự và đó là mã phi tiền tố.

Mã phi tiền tố được ưa chuộng vì nó làm đơn giản sự mã hóa và giải mã.

- Sự mã hóa là đơn giản; ta chỉ cần ghép kề các mã của các ký tự lại với nhau thì sẽ biểu diễn được mọi ký tự trong tập tin.

- Sự giải mã cần một sự biểu diễn thuận tiện cho mã phi tiền tố sao cho *phần đầu* của mã được nhặt ra một cách dễ dàng.

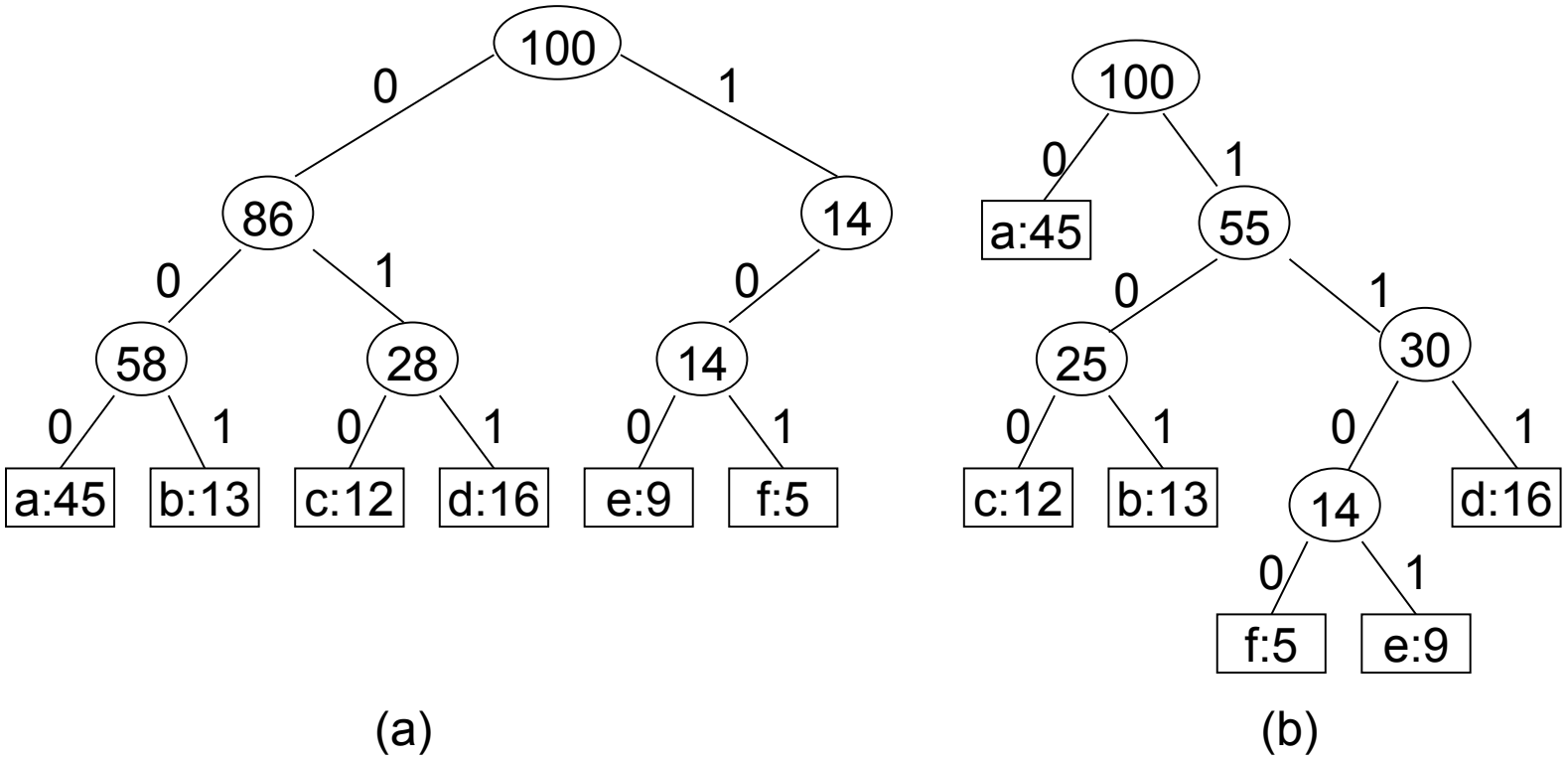
Mã phi tiền tố và cây nhị phân

Biểu diễn cho một mã phi tiền tố là một cây nhị phân với mỗi nút lá tương ứng với các ký tự được cho.

Chúng ta phân giải một mã nhị phân cho một ký tự như là một lối đi từ nút rễ đến nút lá của ký tự ấy, mà 0 ứng với "rẽ sang con bên trái" và 1 nghĩa là "rẽ sang con bên phải".

Mã tối ưu của một tập tin thường được biểu diễn bằng một *cây nhị phân đầy đủ* (*full binary tree*). Một cây nhị phân đầy đủ là một cây nhị phân mà mỗi nút không phải lá có đủ hai con.

Nếu C là tập ký tự mà từ đó các ký tự lấy ra, thì cây nhị phân cho mã phi tiền tố tối ưu có đúng |C| nút lá, mỗi nút lá cho một ký tự, và đúng |C|-1 nút nội.



Hình 5.7 So sánh hai cách mã hóa

Mã phi tiền tố và cây nhị phân (tt.)

Cho một cây T tương ứng với một mã phi tiền tố, chúng ta có thể tính tổng số bit cần để mã hóa một tập tin.

Với mỗi ký tự *c* trong tập ký tự C, dùng *f(c)* để ký hiệu tần số xuất hiện của *c* trong tập tin và $d_T(c)$ là chiều dài của mã cho ký tự *c*. Thì số bit đòi hỏi để mã hóa tập tin là

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

Mà chúng ta coi là *chi phí* của cây nhị phân T.

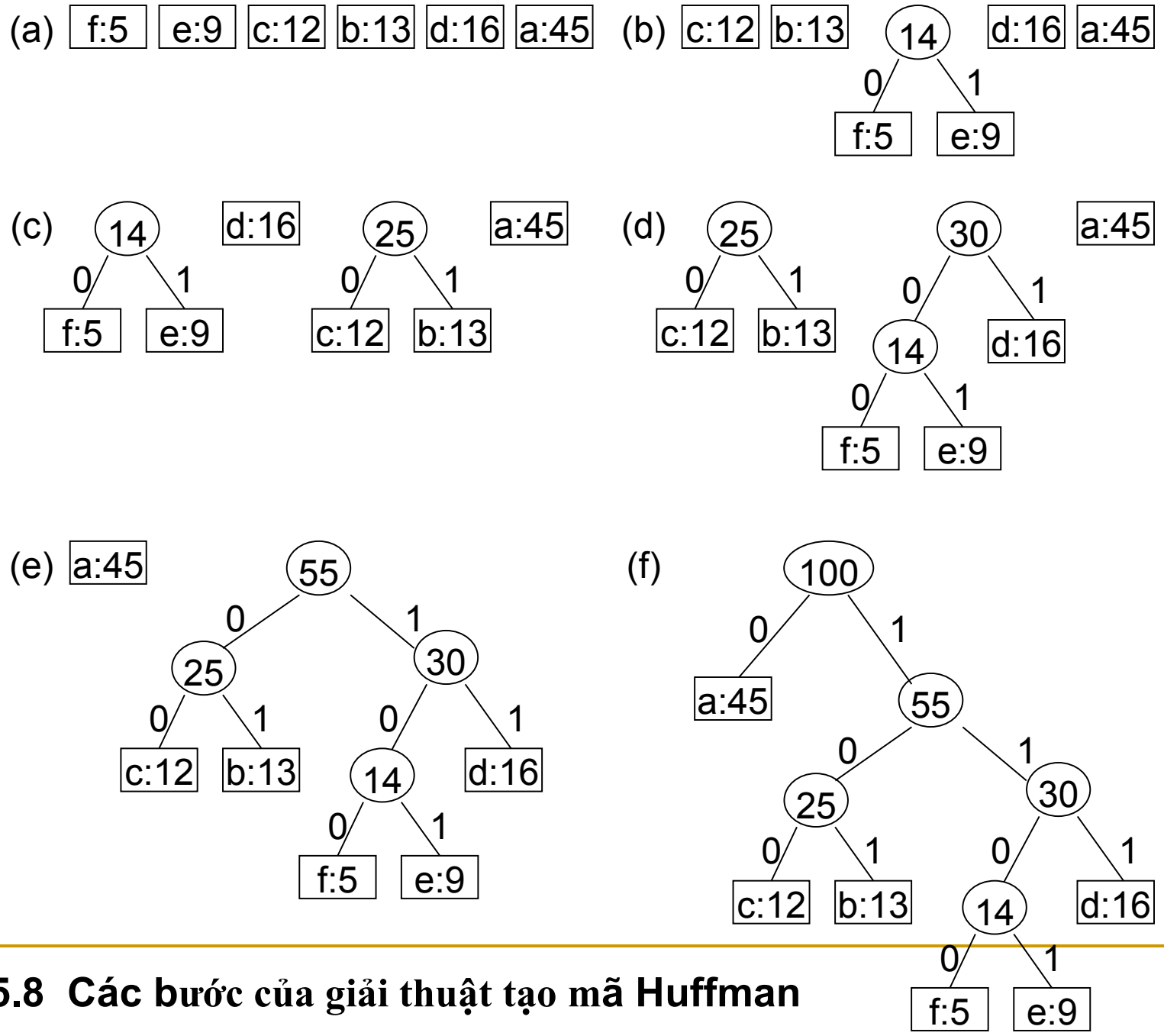
Cấu tạo mã Huffman

Huffman đã đề xuất một giải thuật tham lam để cấu tạo một mã phi tiền tố tối ưu được gọi là mã Huffman (*Huffman code*).

Giải thuật tạo một cây nhị phân T tương ứng với mã tối ưu theo kiểu từ dưới lên. Giải thuật bắt đầu với một tập gồm |C| nút lá và thực hiện một chuỗi gồm |C| tác vụ *trộn* để tạo ra cây cuối cùng.

Một *hàng đợi có độ ưu tiên* *Q*, lấy trị khóa theo tần số *f*, được dùng để nhận diện hai đối tượng có tần số nhỏ nhất để trộn lại với nhau.

Kết quả của việc trộn hai đối tượng là một đối tượng mới mà tần số là tổng tần số của hai đối tượng mà đã được trộn.



Hình 5.8 Các bước của giải thuật tạo mã Huffman

Giải thuật Huffman

```
procedure HUFFMAN(C) ;
begin
    n := |C| ; Q := C ;
    for i := 1 to n -1 do
    begin
        z: = ALLOCATE-NODE( );
        left[z]: = EXTRACT-MIN(Q);
        right[z]: = EXTRACT-MIN(Q);
        f[z] := f[left[z]] + f[right[z]];
        INSERT(Q, z);
    end
end
```

Giả sử Q được hiện thực bởi một heap.

Cho một tập C gồm *n* ký tự, việc khởi tạo Q được thực thi với thời gian O(n).

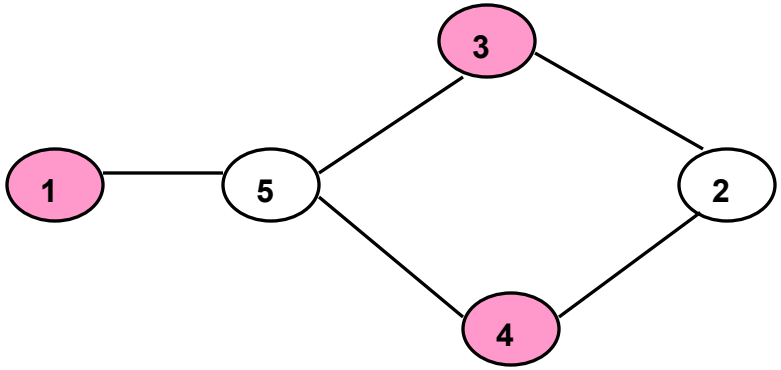
Vòng lặp *for* được thực thi chính xác gồm n-1 lần, và vì mỗi tác vụ làm việc trên heap đòi hỏi O(lgn), vòng lặp này đóng góp chi phí O(nlgn) vào thời gian tính toán.

Như vậy, độ phức tạp của giải thuật HUFFMAN trên tập *n* ký tự sẽ là O(nlgn).

Thí dụ 4: Bài toán tô màu đồ thị

- **Cho một đồ thị vô hướng, tìm số màu tối thiểu để tô các đỉnh của đồ thị sao cho không có hai đỉnh nào có cạnh nối lại được tô cùng màu.**
- **Đây là một bài toán tối ưu hóa.**
- **Một chiến lược để giải quyết bài toán này là dùng giải thuật "tham lam".**
- **Ý tưởng: *Đầu tiên ta cố tô cho được nhiều đỉnh với màu đầu tiên, và rồi dùng một màu mới tô các đỉnh chưa tô sao cho tô được càng nhiều đỉnh càng tốt. Và quá trình này được lặp lại với những màu khác cho đến khi mọi đỉnh đều được tô màu*.**
- **<u>Chú ý</u>: Giải thuật tham lam không đem lại lời giải tối ưu cho bài toán.**

- **Để tô màu các đỉnh trong đồ thị với một màu mới, ta thực hiện các bước sau:**
  - Chọn một đỉnh chưa tô nào đó và tô nó với màu mới.
  - Duyệt qua danh sách các đỉnh còn chưa tô. Với mỗi đỉnh chưa tô, xét xem nó có cạnh nối đến bất cứ đỉnh nào đã được tô với màu mới không. Nếu không có một cạnh như thế, ta tô đỉnh đó với màu mới.
- **<u>Thí dụ</u>: Trong hình vẽ, ta tô đỉnh 1 với màu đỏ, rồi thì ta có thể tô các đỉnh 3 và 4 với cùng màu đỏ.**



Thủ tục SAME_COLOR

- **Thủ tục SAME_COLOR xác đinh một tập các đỉnh (biến *newclr*), mà tất cả những đỉnh đó có thể tô với cùng màu mới. Thủ tục này được gọi nhiều lần cho đến khi mọi đỉnh đều đã được tô màu.**

```
procedure SAME_COLOR(G, newclr);
  /* SAME_COLOR assigns to newclr a set
       of vertices of G that may be given the
       same color  */
  begin
   newclr := ∅;
     for each uncolored vertex of G do
       if v is not adjacent to any vertex in newclr
       then
            mark v colored and add v to newclr.
   end;
```

```
procedure G_COLORING(G);
  procedure SAME_COLOR(G, newclr);
  /* SAME_COLOR assigns to newclr a set of
       vertices of G that may be given the same color ;
       a: adjacency matrix for graph G */
  begin
   newclr := ∅;
     for each uncolored vertex v of G do
     begin
       found := false;
       for each vertex w X newclr do
          if a[v,w] = 1    /*there is an edge between v and w in G */
          then
                    found := true;
       if  not found  then
          mark v colored and add v to newclr
     end
   end;
```

```
for each vertex in G do mark uncolored;
while there is any vertex marked uncolored do
begin
  SAME_COLOR(G, newclr);
  print newclr
end.
```

- ***Bậc của một đỉnh*: số cạnh nối đến đỉnh đó.**

Định lý: Nếu $\chi(G)$ là số màu tối thiểu để tô đồ thị G và ΔG là bậc lớn nhất trong đồ thị G thì $\chi(G) \leq \Delta G + 1$

Độ phức tạp của giải thuật tô màu đồ thị

- **Tác vụ căn bản: kiểm tra hai đỉnh có cạnh nối hay không.**

  Độ phức tạp của thủ tục SAME_COLOR: O(n).

  Nếu m là số màu được dùng để tô đồ thị thì thủ tục SAME_COLOR được gọi tất cả m lần. Do đó, độ phức tạp của toàn giải thuật: m* O(n). Vì m thường là một số nhỏ, ta có thể nói

  $\Rightarrow$ Giải thuật có độ phức tạp tuyến tính.

Ứng dụng: Xếp lịch thi học kỳ

- Mỗi môn thi được biểu diễn bằng một đỉnh trong đồ thị.
- Xếp lịch thi là gán ca thi vào các môn thi. Các ca thi chính là các màu dùng để tô cho các đỉnh.
- Một cạnh nối giữa hai đỉnh nếu có tồn tại ít nhất một sinh viên lấy cả hai môn và phải thi cả hai môn, do đó không thể xếp hai môn thi biểu thị bằng hai đỉnh đó vào cùng một ca thi.

Ứng dụng: Gán tần số trong lãnh vực vô tuyến, điện thoại di động (Frequency assignment problem)

Một Heuristic cho bài toán Tô Màu đồ thị

***Màu có bậc lớn nhất được tô trước*. (Welsh and Powell)**

- ***Bậc của một đỉnh*: số cạnh nối đến đỉnh đó.**
- **Lý do: Những đỉnh có càng nhiều cạnh nối tới thì càng khó tô nếu ta đợi cho đến khi những đỉnh láng giềng của nó đã được tô.**
- **Giải thuật**
  - Arrange the vertices by decreasing order of degrees.
  - Color a vertex with maximal degree with color 1.
  - Choose an uncolored vertex with a maximum degree. If there is another vertex with the same maximum vertex, choose either of them.
  - Color the chosen vertex with the least possible (lowest numbered) color.
  - If all vertices are colored, stop. Otherwise, return to 3.