Chương 5

Các kỹ thuật thiết kế giải thuật

Nội dung

1. **Qui hoạch động**
2. **Giải thuật tham lam**
3. **Giải thuật quay lui**

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

$(A_1(A_2A_3))$ thực hiện

$10.000.5 + 10.5.50 = 5000 + 2500$

$= 7500$ 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 5.1 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

		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				

Hình 5.1

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

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

$$= 7125$$

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

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

Định lý 6.1

Cho $X = \langle x_1, x_2, \ldots, x_m \rangle$ và $Y = \langle y_1, y_2, \ldots, y_n \rangle$ là những chuỗi, và $Z = \langle z_1, z_2, \ldots, z_k \rangle$ 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 5.2 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."

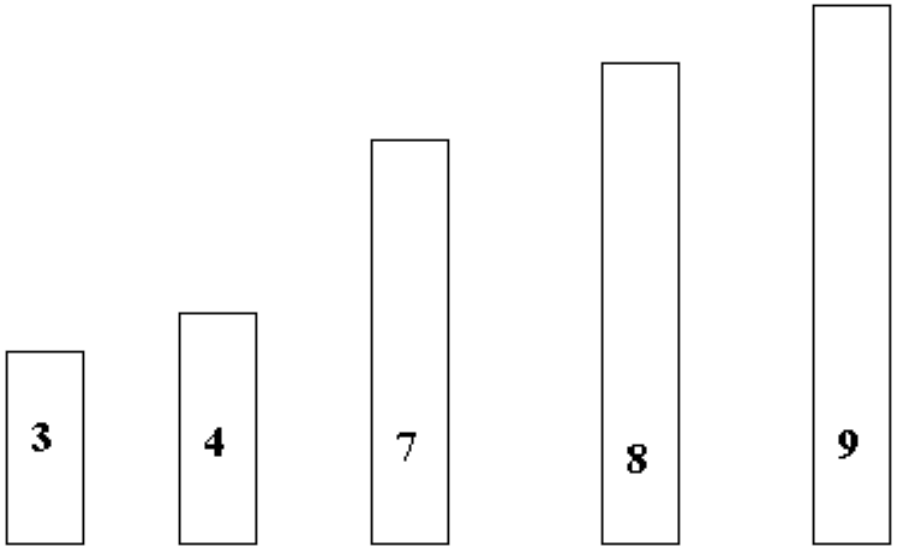
Bài toán này có thể giải bằng *qui hoạch động* bằng cách dùng hai bảng *cost* và *best* sau đây:

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**

Hình 5.3 Một thí dụ của bài toán cái túi

Giải thuật quy hoạch động cho bài toán 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 5.1.1 ***Giải thuật qui hoạch động để giải bài toán cái túi có thời gian chạy tỉ lệ với NM*****.**

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:

- **Giải thuật Prim để tính cây bao trùm tối thiểu**
- **Giải thuật Dijkstra để giải bài tóan những lối đi ngắn nhất từ một đỉnh nguồn (single-source shortest paths problem).**

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:
$f_1 \leq f_2 \leq \ldots \leq f_n$.

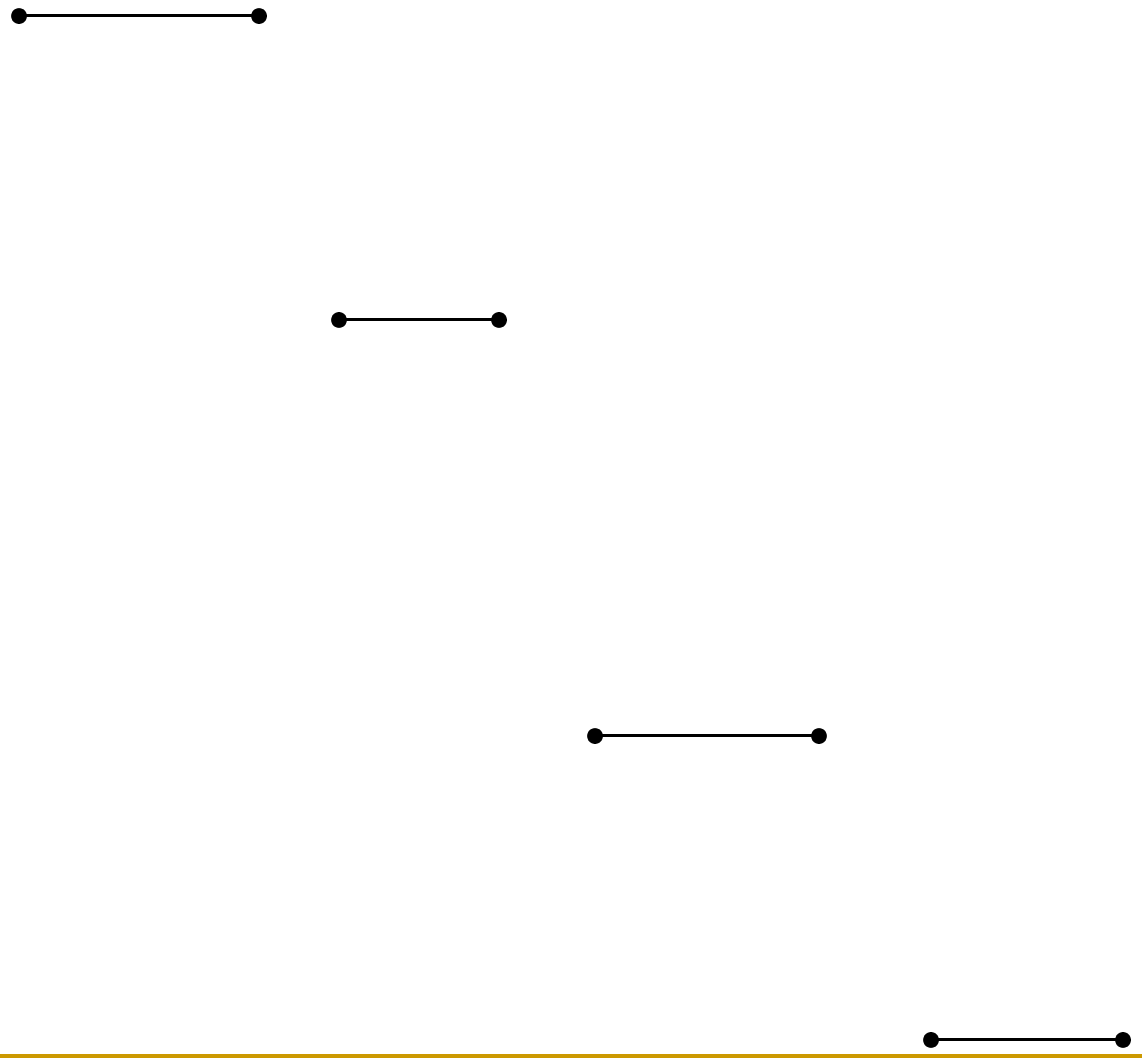
```
procedure GREED-ACTIVITY-SELECTOR(S, f) ;        /* s is the array keeping the set of activities and f is the array keeping the finishing times */
begin
   n := length[s];   A := {1};  j: = 1;
   for i: = 2 to n do
      if s_i >= f_j then /* i is compatible with all  activities in A */
      begin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

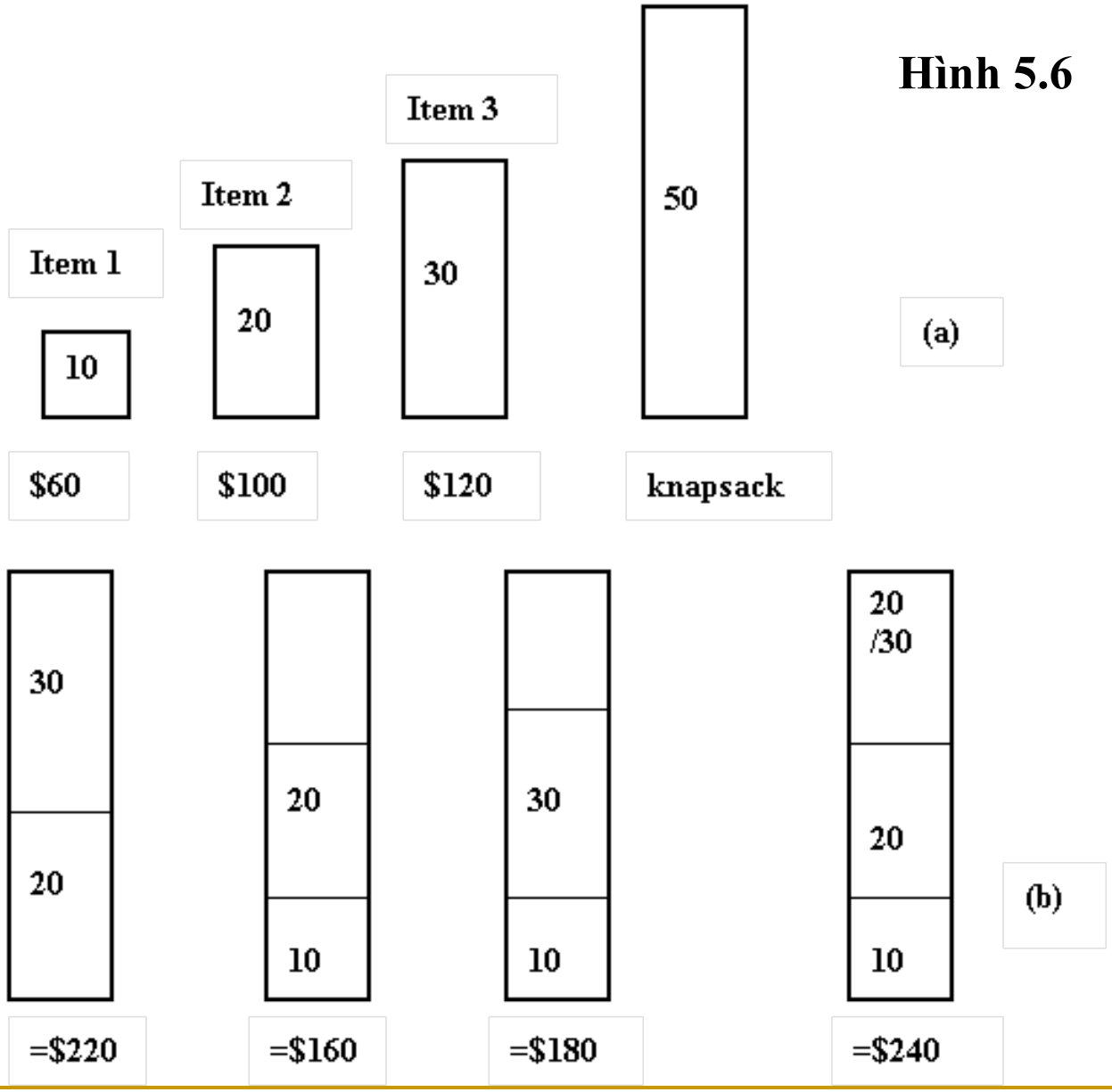
Bài toán cái túi dạng phân số (tt.)

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

```
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 chúng ta có một tập tin 100000 ký tự mà chúng 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

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

Nếu chúng 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 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.

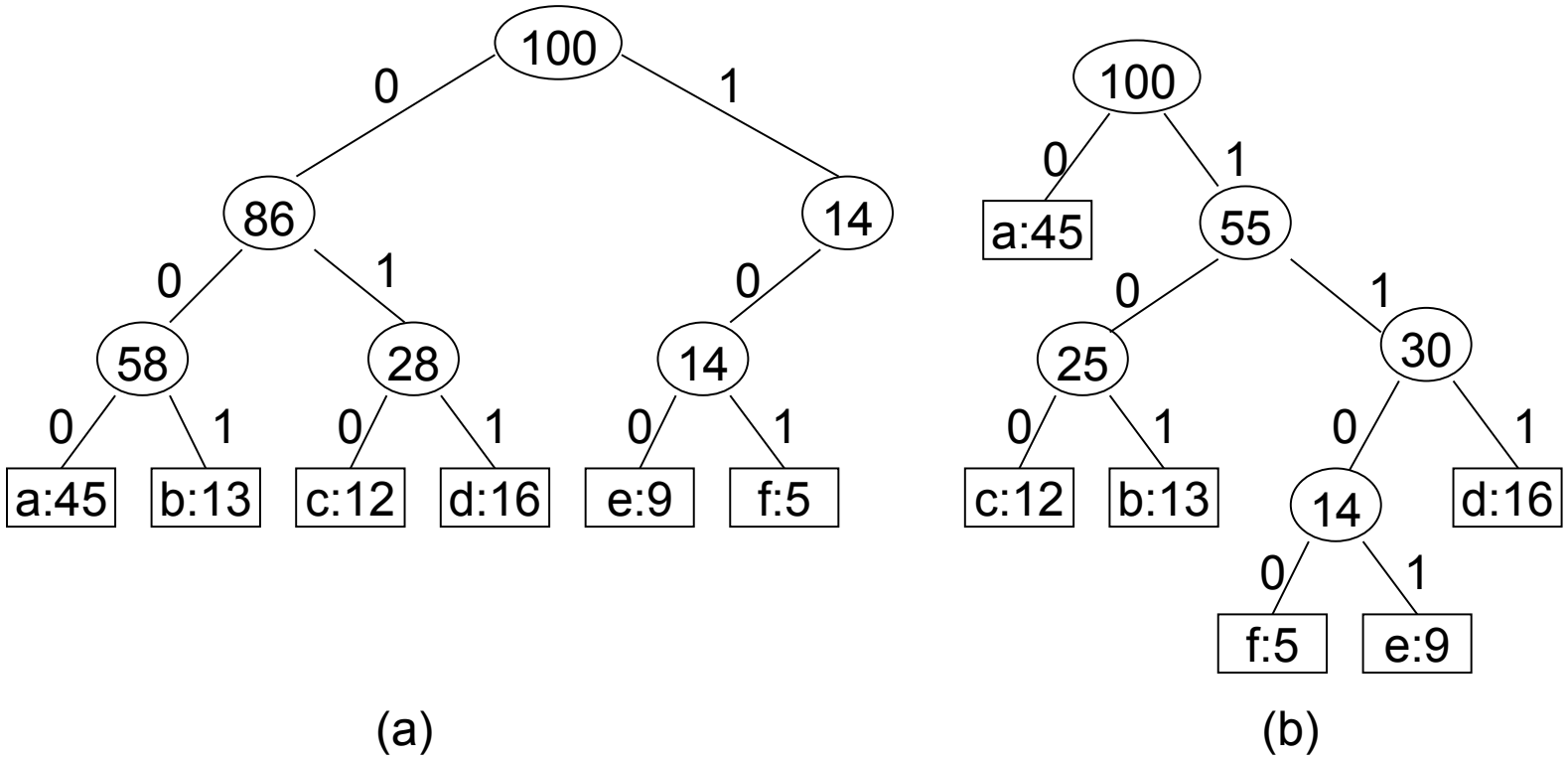
Mã phi tiền tố và cây nhị phân

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.

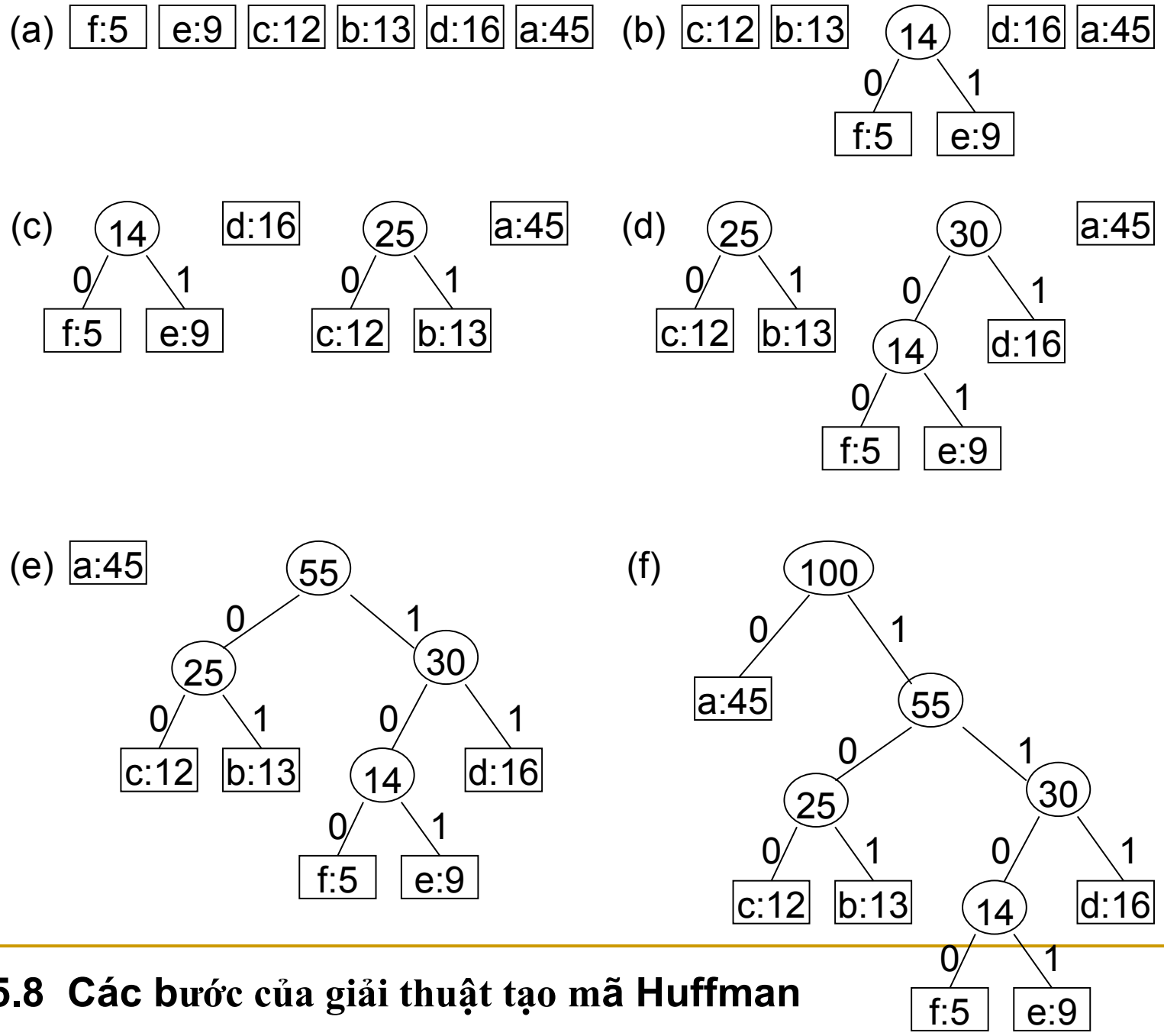
Cấu tạo mã Huffman

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
```

Độ phức tạp của giải thuật Huffman

Giả sử Q được hiện thực hóa bởi một heap nhị phân.

Cho một tập C gồm *n* ký tự, việc khởi tạo của 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, thời gian tính toán của giải thuật HUFFMAN trên tập *n* ký tự sẽ là O(nlgn).

Giải thuật quay lui

Một phương pháp tổng quát để giải quyết vấn đề: thiết kế giải thuật tìm lời giải cho bài tóan không phải là bám theo một tập qui luật tính tóan được xác định mà là bằng cách *thử và sửa sai* (*trial and error*).

Khuôn mẫu thông thường là phân rã quá trình thử và sửa sai thành những công tác bộ phận. Thường thì những công tác bộ phận này được diễn tả theo lối đệ quy một cách thuận tiện và bao gồm việc *thăm dò một số hữu hạn những công tác con*.

Ta có thể coi toàn bộ quá trình này như là một *quá trình tìm kiếm* (search process) mà dần dần cấu tạo và duyệt qua một cây các công tác con.

Bài toán đường đi của con hiệp sĩ (The Knight's Tour Problem)

Cho một bàn cờ $n \times n$ với n^2 ô. Một con hiệp sĩ – được di chuyển tuân theo luật chơi cờ vua – được đặt trên bàn cở tại ô đầu tiên có tọa độ x_0, y_0.

Vấn đề là tìm một lộ trình gồm $n^2 - 1$ bước sao cho phủ toàn bộ bàn cờ (mỗi ô được viếng đúng một lần).

Cách rõ ràng để thu giảm bài toán phủ n^2 ô là xét bài toán, hoặc là

- **thực hiện bước đi kế tiếp, hay**
- **phát hiện rằng không kiếm được bước đi hợp lệ nào.**

```
procedure try next move;
begin  initialize selection of moves;
   repeat
      select next candidate from list of next moves;
       if acceptable then
  begin
        record move;
         if board not full then
              begin
                   try next move;                          (5.3.1)
               if not successful then erase previous recording
              end
        end
   until (move was successful) ∨ (no more  candidates)
end
```

Cách biểu diễn dữ liệu

Chúng ta diễn tả bàn cờ bằng một ma trận *h*.

```
type index = 1..n ;
var h: array[index, index] of integer;
```

h[x, y] = 0: ô <x,y> chưa hề được viếng

h[x, y] = i: ô <x,y> đã được viếng tại bước chuyển thứ *i* ($1 \le i \le n^2$)

Điều kiện "board not full" có thể được diễn tả bằng "$i < n^2$".

u, v: tọa độ của ô đến.

Điều kiện "acceptable" có thể được diễn tả bằng

$(1 \le u \le n) \wedge (1 \le v \le n) \wedge (h[u,v]=0)$

```
procedure try(i: integer; x,y : index; var q: boolean);
var u, v: integer; q1 : boolean;
begin  initialize selection for moves;
   repeat  let u, v be the coordinates of the next move ;
     if (1≤u≤n) ∧ (1≤v≤n) ∧ (h[u,v]=0) then
     begin  h[u,v]:=i;
        if i < sqr(n) then                                    (5.3.2)
        begin
           try(i + 1, u, v, q1);  if ¬ q1 then h[u,v]:=0
        end
        else q1:= true
      end
    until q1 ∨ (no more candidates);
   q:=q1
end
```

Cho tọa độ của ô hiện hành <x, y>, có 8 khả năng để chọn ô kế tiếp <u, v> để đi tới. Chúng được đánh số từ 1 đến 8 như sau:

	3		2	
4				1
		⊕		
5				8
	6		7	

Sự tinh chế sau cùng

Cách đơn giản nhất để đạt được tọa độ *u, v* từ *x, y* là bằng cách cọng độ sai biệt toạ độ tại hai mảng *a* và *b*.

Và *k* được dùng để đánh số ứng viên (candidate) kế tiếp.

```
program knightstour (output);
const n = 5; nsq = 25;
type index = 1..n
var  i,j: index;  q: boolean;
s: set of index;
a,b: array [1..8] of integer;
h: array [index, index] of integer;
```

```
procedure try (i: integer; x, y: index; var q:boolean);
var k,u,v :  integer; q1: boolean;
begin    k:=0;
   repeat
      k:=k+1; q1:=false; u:=x+a[k]; v:=y+b[k];
      if (u in s) ∧ (v in s) then
        if h[u,v]=0 then
        begin
             h[u,v]:=i;
             if i < nsq then
             begin
                 try(i+1, u,v,q1);
                  if ¬ q1 then h[u,v]:=0
             end
             else q1:=true
       end
   until q1 ∨ (k =8);
   q:=q1
end {try};
```

```
begin
      s:=[1,2,3,4,5];
a[1]:=  2;  b[1]:=  1;
a[2]:=  1; b[2]:= 2;
a[3]:=  –1; b[3]:= 2;
a[4]:=  –2;   b[4]:=1;
a[5]:=  –2;   b[5]:= –1;
a[6]:=  –1;  b[6]:= –2;
a[7]:=  1;     b[7]:= –2;
a[8]:=  2;     b[8]:= –1;
  for i:=1 to n do
for j:=1 to n do h[i,j]:=0;
```

```
h[1,1]:=1; try (2,1,1,q);
 if q then
     for i:=1 to n do
     begin
          for j:=1 to n do
               write(h[i,j]:5);
          writeln
      end
   else writeln ('NO
SOLUTION')
end.
```

Thủ tục đệ quy được khởi động bằng lệnh gọi với tọa độ khởi đầu x_0, y_0 , từ đó chuyến đi bắt đầu.

$$H[x_0,y_0]:= 1; try(2, x_0, y_0, q)$$

Hình 5.3.1 trình bày một lời giải đạt được với vị trí <1,1> với n = 5.

1	6	15	10	21
14	9	20	5	16
19	2	7	22	11
8	13	24	17	4
25	18	3	12	23

Từ thí dụ trên ta đi đến với một kiểu "giải quyết vấn đề" mới:

Đặc điểm chính là

"bước hướng về lời giải đầy đủ và ghi lại thông tin về bước này mà sau đó nó có thể bị tháo gỡ và xóa đi khi phát hiện rằng bước này đã không dẫn đến lời giải đầy đủ, tức là một bước đi dẫn đến "tình thế bế tắc"(*dead-end*). (Hành vi này được gọi là *quay lui -bactracking*.)

Khuôn mẫu tổng quát của giải thuật quay lui

```
procedure try;
begin intialize selection of candidates;
repeat
    select next;
        if acceptable then
        begin
            record it;
            if solution incomplete then
            begin
                try next step;                          (5.3.3)
                if not successful then cancel recording
            end
        end
until successful ∨ no more candidates
end
```

Nếu tại mỗi bước, số ứng viên phải thử là cố định thì kiểu mẫu trên có thể biến đổi như :

⇒

Thủ tục được gọi bằng lệnh gọi

try(1).

```
procedure try (i: integer);
var k : integer;
begin k:=0;
   repeat
      k:=k+1; select k-th candidate;
      if acceptable then
      begin
         record it;
         if i<n then
         begin
            try (i+1);                                    (5.3.4)
            if not successful then
                              cancel recording
         end
      end
    until successful ∨ (k=m)
end
```

Bài toán 8 con hậu

Bài toán này đã được C.F. Gauss khảo sát năm 1850, nhưng ông ta không hoàn toàn giải quyết được.

"Tám con hậu được đặt vào bàn cờ sao cho không có con hậu nào có thể tấn công con hậu nào".

Dùng khuôn mẫu ở hình 5.3.1, ta sẽ có được một thủ tục sau cho bài toán 8 con hậu:

```
procedure try (i: integer);
begin
   initialize selection of positions for i-th queen;
   repeat
      make next selection;
      if safe then
      begin
         setqueen;
         if i < 8 then
         begin
            try (i + 1);
            if not successful then remove queen
         end
       end
    until successful ∨ no more positions
end
```

Luật cờ: ***Một con hậu có thể tấn công các con hậu khác nằm trên cùng một hàng, cùng một cột hay là cùng đường chéo trên bàn cờ.***

Cách biểu diễn dữ liệu

Làm cách nào để diễn tả 8 con hậu trên bàn cờ?

```
var x: array[1..8] of integer;
    a: array[1..8] of Boolean;
    b: array[b1..b2] of Boolean;
    c: array[c1..c2] of Boolean;
```

với

x[i] chỉ vị trí của con hậu trên cột thứ *i*;

a[j] cho biết không có con hậu trên hàng thứ *j*;

b[k] cho biết không có con hậu trên đường chéo ↙ thứ *k*;

c[k] cho biết không có con hậu trên đường chéo ↘ thứ *k*.

Việc chọn trị cho các mốc *b1*, *b2*, *c1*, *c2* được xác định bởi cách mà các chỉ số của các mảng *b* và *c* được tính. Hãy chú ý rằng trên cùng một đường chéo chiều ↙ tất cả các ô sẽ có cùng giá trị của tổng hai tọa độ *i* +*j*, và trên cùng một đường chép chiều ↘ diagonal, tất cả các ô sẽ có cùng giá trị của hiệu hai tọa độ (*i* – *j*).

Như vậy, phát biểu *setqueen* được tinh chế như sau:

x[i]:=j; a[j]:=false; b[i+j]:=false;c[i-j]:=false;

Phát biểu *removequeen* được chi tiết hóa như sau:

a[j] = true; b[i+j] = true ; c[i-j] := true

Điều kiện *safe* được diễn tả như sau:

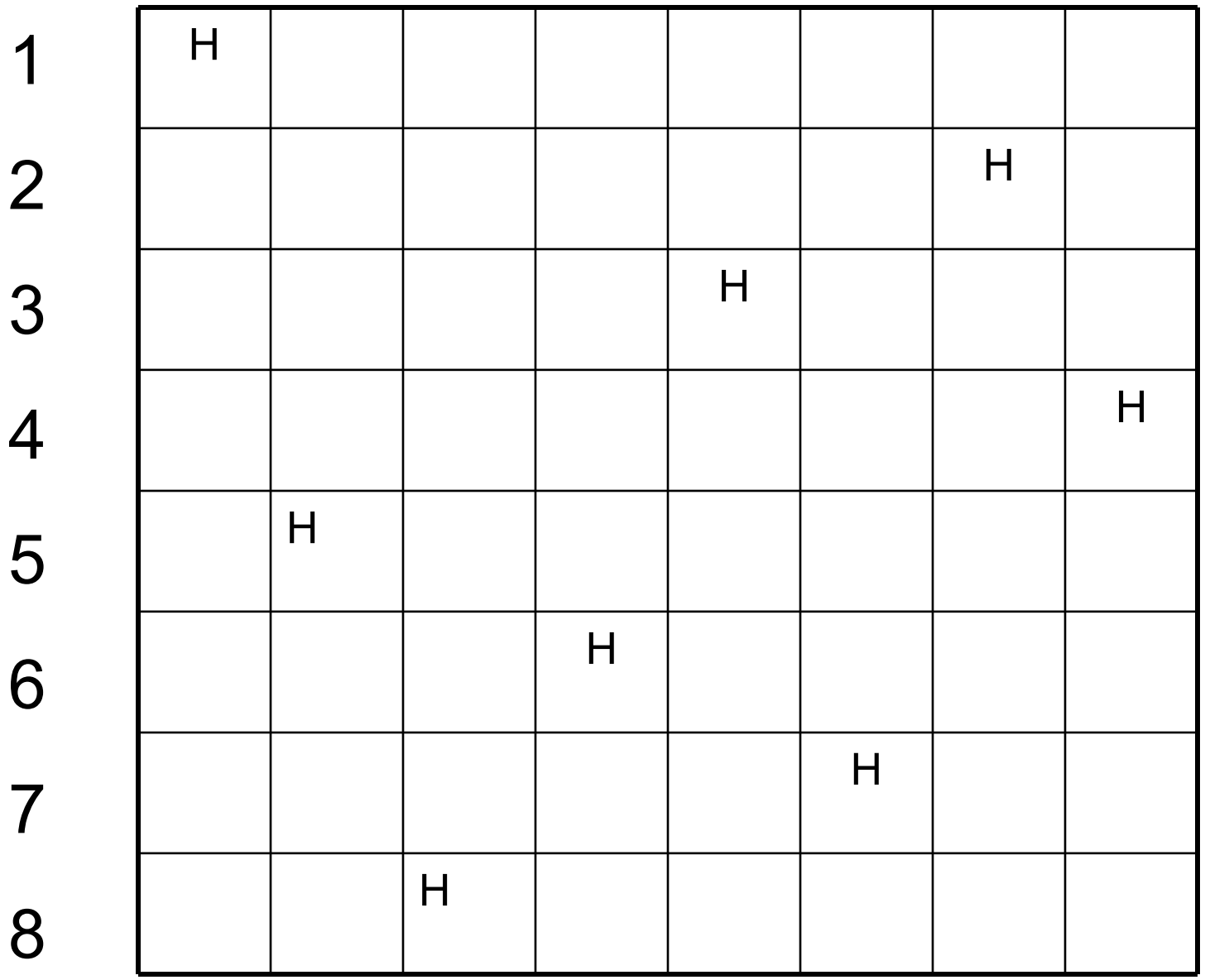
a[j] ∧ b[i+j] ∧ c[i-j]

```
program eightqueeen1(output);
{find one solution to eight queens
problem}
var       i : integer; q: boolean;
a : array [1..8]   of boolean;
b : array [2..16] of boolean;
c : array [–7..7] of boolean;
x : array [1..8]   of integer;
procedure try(i: integer; var q:
boolean);
var j: integer;
begin
   j:=0;
   repeat
      j:=j+1; q:=false;
      if a[j] ∧ b[i+j] ∧ c[i-j] then
```

```
begin
         x[i]:=j;
         a[j]:=false; b[i+j]:=false;
         c[i-j]:=false;
         if i<8 then
         begin
            try (i+1, q);
            if ¬ q then
            begin
               a[j]:=true; b[i+j]:=true;
               c[i-j]:=true
             end
          end
          else q:=true
       end
    until q ∨ (j=8)
end {try};
```

```
begin
    for i:= 1 to 8 do a[i]:=true;
    for i:= 2 to 16 do b[i]:=true;
    for i:= –7 to 7 do c[i]:=true;
    try (1,q);
    if q then
      for i:=1 to 8 do
                    write (x[i]:4);
    writeln
end
```

Một lời giải của bài toán 8 con hậu được cho ở hình vẽ sau:


1
2
3
4
5
6
7
8
H
H
H
H
H
H
H
H

Sự mở rộng: Tìm tất cả các lời giải

Sự mở rộng là tìm không chỉ một lời giải mà tất cả những lời giải của bài toán đã cho.

<u>Phương pháp</u>: ***Một khi một lời giải được tìm thấy và ghi lại, ta tiếp tục xét ứng viên kế trong quá trình chọn ứng viên một cách có hệ thống*****.**

Khuôn mẫu tổng quát được dẫn xuất từ (5.3.4) và được trình bày như sau:

```
procedure try(i: integer);
var k: integer;
begin
    for k:=1 to m do
    begin
        select k-th candidate;
        if acceptable then
        begin
            record it;
            if i<n then try (i+1) else print solution;
            cancel recording
        end
    end
end
```

Trong giải thuật mở rộng, để đơn giản hóa điều kiện dừng của quá trình chọn, phát biểu *repeat* được thay thế bằng phát biểu *for*

```
program eightqueens(output);
var i: integer;
a: array [1.. 8]   of boolean;
b: array [2.. 16] of boolean;
c: array [–7.. 7] of boolean;
x: array [1.. 8]   of integer;
procedure print;
var k : integer;
begin
   for k : 1 to 8 do write(x[k]:4);
   writeln
end {print};
```

```
procedure try (i:integer);
var j: integer;
begin
   for j:=1 to 8 do
      if a[j] ∧ b[i+j] ∧ c[i-j] then
      begin
         x[i]:=j;
         a[j]:=false; b[i+j]:= false;
         c[i-j]:=false;
         if i < 8 then try(i+1) else print;
         a[j]:=true; b[i+j]:= true;
         c[i-j]:= true;
      end
end {try};
```

```
begin
    for i:= 1 to 8 do a[i]:=true;
    for i:= 2 to 16 do b[i]:=true;
    for i:= –7 to 7 do c[i]:=true;
    try(1);
end.
```

Giải thuật mở rộng có thể sản sinh tất cả 92 lời giải cho bài toán 8 con hậu.

Nhưng thật ra chỉ có 12 lời giải thật sự khác biệt nhau.

Mười hai lời giải đó được liệt kê trong bảng sau:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	N
1	5	8	6	3	7	2	4	876
1	6	8	3	7	4	2	5	264
1	7	4	6	8	2	5	3	200
1	7	5	8	2	4	6	3	136
2	4	6	8	3	1	7	5	504
2	5	7	1	3	8	6	4	400
2	5	7	4	1	8	6	3	72
2	6	1	7	4	8	3	5	280
2	6	8	3	1	4	7	5	240
2	7	3	6	8	5	1	4	264
2	7	5	8	1	4	6	3	160
2	8	6	1	3	5	7	4	336

Những giá trị ở cột N chỉ số lần thử để tìm một ô an toàn. Trung bình cần 161 phép thử trong 92 lời giải này.

Cây không gian trạng thái

- **Để tiện diễn tả giải thuật quay lui, ta xây dựng cấu trúc cây ghi những lựa chọn đã được thực hiện. Cấu trúc cây này được gọi là *cây không gian trạng thái* (state space tree) hay *cây tìm kiếm* (search tree).**
- **Nút rễ của cây diễn tả *trạng thái đầu tiên* trước khi quá trình tìm kiếm lời giải bắt đầu.**
- **Các nút ở mức đầu tiên trong cây diễn tả những lựa chọn được làm ứng với thành phần đầu tiên của lời giải.**
- **Các nút ở mức thứ hai trong cây diễn tả những lựa chọn được làm ứng với thành phần thứ hai của lời giải và các mức kế tiếp tương tự như thế.**

Một nút trên cây KGTT được gọi là *triển vọng* nếu nó tương ứng với lời giải bộ phận mà sẽ có thể dẫn đến lời giải đầy đủ; trái lại, nó được gọi là một lời giải *không triển vọng*.

Các nút lá diễn tả những trường hợp *bế tắc* (dead end) hay những *lời giải đầy đủ*.

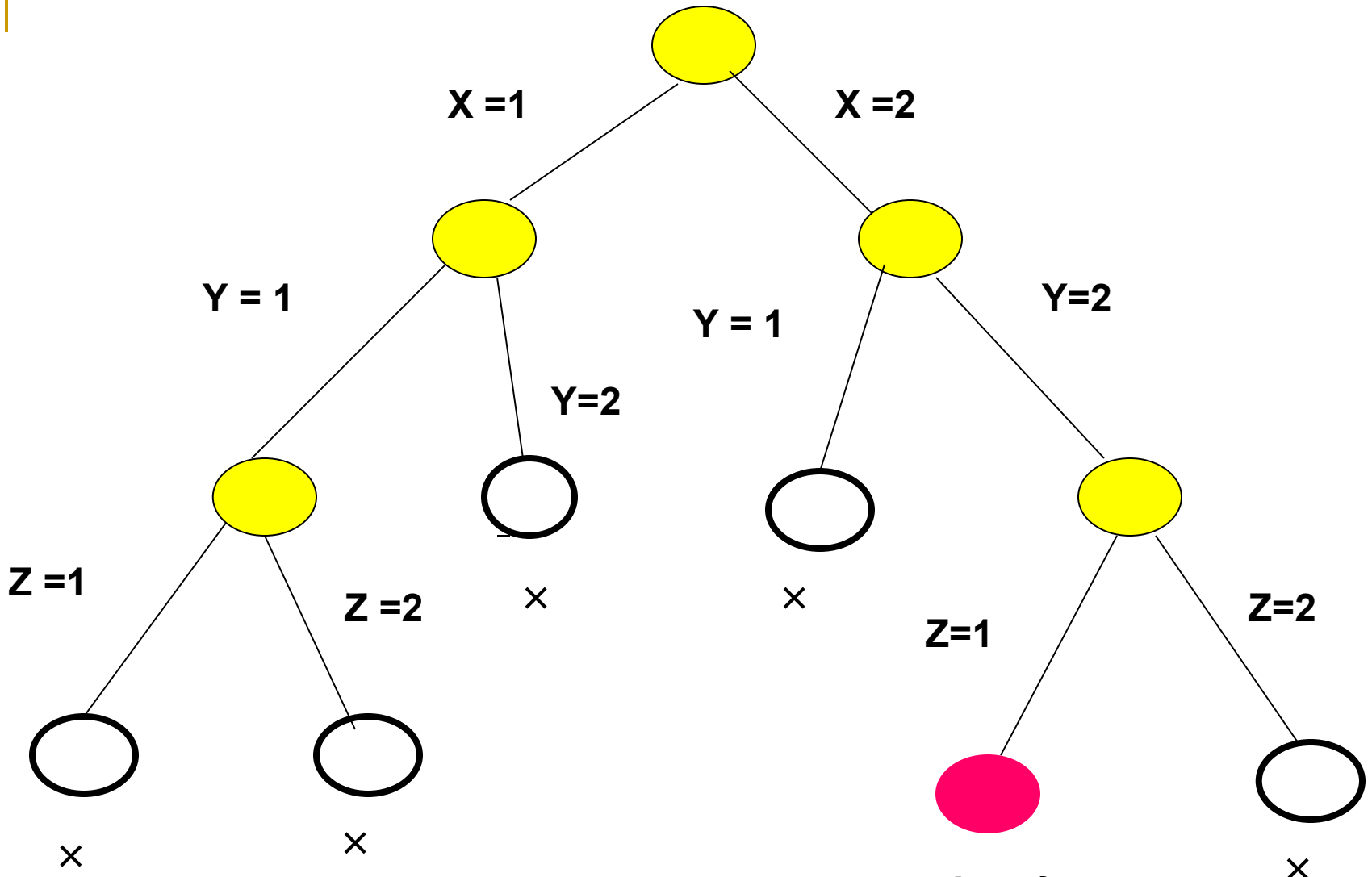
<u>Thí dụ</u>: Cho một bài toán như sau:

Tập biến: X, Y, Z.

Gán trị từ tập {1,2} vào các biến sao cho thỏa mãn các ràng buộc: X = Y, X ≠ Z, Y > Z.

Hãy giải bài toán bằng một giải thuật quay lui.

Cây không gian trạng thái của bài toán này được cho ở hình vẽ sau:



Hình 5.9 Thí dụ về cây Không Gian Trạng Thái

Độ phức tạp của giải thuật quay lui

Thời gian tính toán của các giải thuật quay lui thường là hàm mũ (*exponential*).

Nếu mỗi nút trên cây không gian trạng thái có trung bình α nút con, và chiều dài của lối đi lời giải là N, thì số nút trên cây sẽ tỉ lệ với α^N.

Thời gian tính toán của giải thuật đệ quy tương ứng với số nút trên cây không gian trạng thái nên có độ phức tạp hàm mũ.

Giải thuật nhánh và cận (branch-and-bound)

- **Bài toán người thương gia du hành (TSP): cho một tập các thành phố và khoảng cách giữa mỗi cặp thành phố, tìm một lộ trình đi qua tất cả mọi thành phố sao cho tổng khoảng cách của lộ trình nhỏ hơn M.**

Điều này dẫn đến một bài toán khác: cho một đồ thị vô hướng, có cách nào để nối tất cả các nút bằng một chu trình đơn hay không. Đây chính là bài toán Chu trình Hamilton (HCP).

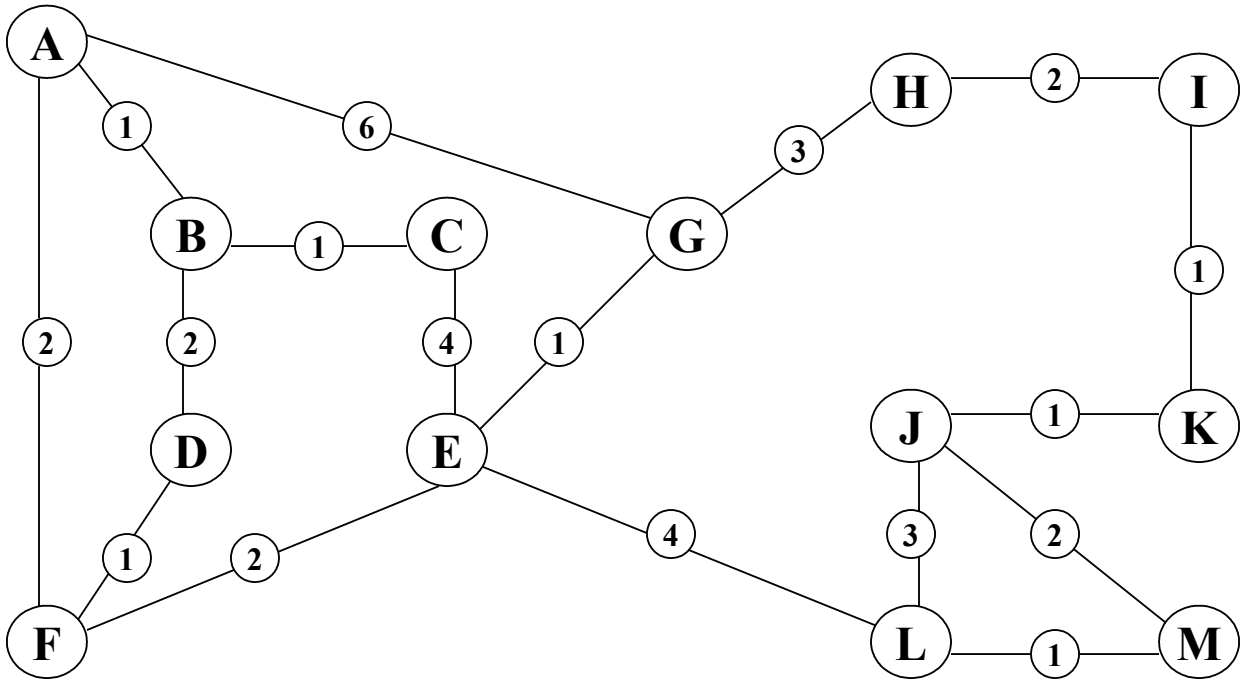
Để giải bài toán (HCP), ta có thể cải biên giải thuật tìm kiếm theo chiều sâu trước (DFS) để giải thuật này có thể sinh ra mọi lối đi đơn mà đi qua mọi đỉnh trong đồ thị.

Giải thuật DFS cải biên

Điều này có thể thực hiện được bằng cách sửa lại thủ tục *visit* như sau:

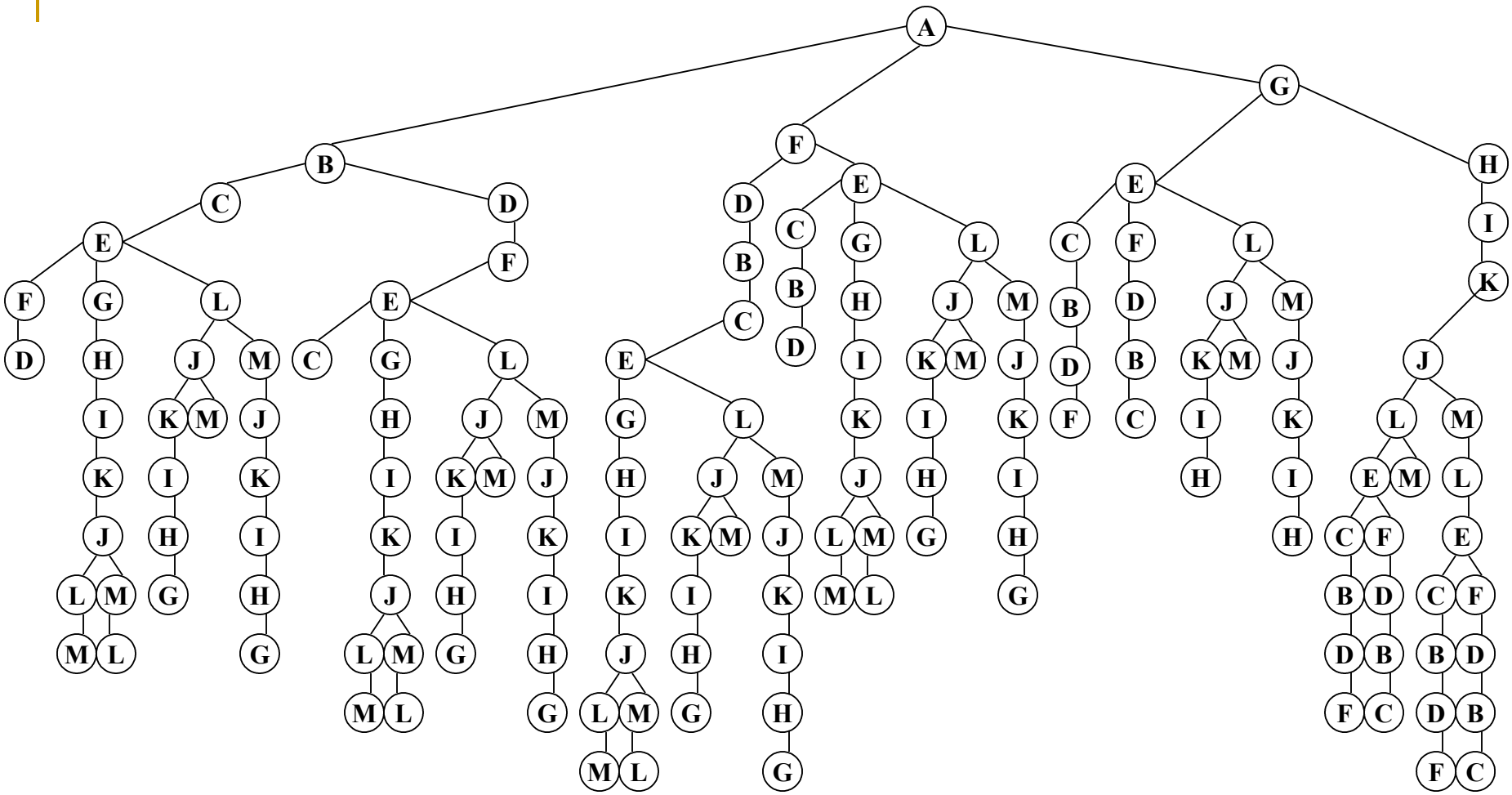
```
procedure visit( k: integer);
var t: integer;
begin
   id := id +1; val[k]:= id;
    for t:= 1 to V do
        if a[k, t] = 0 then visit(t);
   id := id –1; val[k] := 0
end;
```

Một thí dụ về bài toán TSP



Hình 5.10

Tìm kiếm vét cạn các lối đi đơn



Hình 5.11

Ý tưởng nhánh và cận

Khi áp dụng giải thuật DFS cải biên để sinh ra mọi lối đi đơn, trong quá trình tìm kiếm một lối đi tốt nhất (tổng trọng số nhỏ nhất) cho bài toán TSP, có một kỹ thuật *tỉa nhánh* quan trọng là kết thúc sự tìm kiếm ngay khi thấy rằng nó không thể nào thành công được.

Giả sử một lối đi có chí phí *x* đã được tìm thấy. Thì thật vô ích để duyệt tiếp trên lối đi chưa-đầy-đủ nào mà chi phí cho đến hiện giờ đã *lớn hơn* *x*. Điều này có thể được thực hiện bằng cách không gọi đệ quy thủ tục *visit* nếu lối đi chưa-đầy-đủ hiện hành đã lớn hơn chi phí của *lối đi đầy đủ tốt nhất* cho đến bây giờ.

Ý tưởng nhánh và cận (tt.)

Rõ ràng ta sẽ không *bỏ sót* lối đi chi phí nhỏ nhất nào nếu ta bám sát một chiến lược như vậy.

Kỹ thuật tính cận (bound) của các lời giải chưa-đầy-đủ để hạn chế số lời giải phải dò tìm được gọi là *giải thuật nhánh và cận*.

Giải thuật này có thể áp dụng khi có chi phí được gắn vào các lối đi.