CÁC CHIẾN LƯỢC THIẾT KẾ THUẬT TOÁN

Với một vấn đề đặt ra, làm thế nào chúng ta có thể đưa ra thuật toán giải quyết nó? Trong chương này, chúng ta sẽ trình bày các chiến lược thiết kế thuật toán, còn được gọi là các kỹ thuật thiết kế thuật toán. Mỗi chiến lược này có thể áp dụng để giải quyết một phạm vi khá rộng các bài toán. Mỗi chiến lược có các tính chất riêng và chỉ thích hợp cho một số dạng bài toán nào đó. Chúng ta sẽ lần lượt trình bày các chiến lược sau: chia-để-trị (divide-and-conquer), quy hoạch động (dynamic programming), quay lui (backtracking) và tham ăn (greedy method). Trong mỗi chiến lược chúng ta sẽ trình bày ý tưởng chung của phương pháp và sau đó đưa ra một số ví dụ minh họa.

Cần nhấn mạnh rằng, ta không thể áp dụng máy móc một chiến lược cho một vấn đề, mà ta phải phân tích kỹ vấn đề. Cấu trúc của vấn đề, các đặc điểm của vấn đề sẽ quyết định chiến lược có khả năng áp dụng.

1. CHIA - ĐỂ - TRỊ

1.1. Phương pháp chung

Chiến lược thiết kế thuật toán được sử dụng rộng rãi nhất là chiến lược chia-để-trị. Ý tưởng chung của kỹ thuật này là như sau: Chia vấn đề cần giải thành một số vấn đề con cùng dạng với vấn đề đã cho, chỉ khác là cỡ của chúng nhỏ hơn. Mỗi vấn đề con được giải quyết độc lập. Sau đó, ta kết hợp nghiệm của các vấn đề con để nhận được nghiệm của vấn đề đã cho. Nếu vấn đề con là đủ nhỏ có thể dễ dàng tính được nghiệm, thì ta giải quyết nó, nếu không vấn đề con được giải quyết bằng cách áp dụng đệ quy thủ tục trên (tức là lại tiếp tục chia nó thành các vấn đề con nhỏ hơn,…). Do đó, các thuật toán được thiết kế bằng chiến lược chia-để-trị sẽ là các thuật toán đệ quy.

Sau đây là lược đồ của kỹ thuật chia-để-trị:

```
DivideConquer (A,x)
// tìm nghiệm x của bài toán A.
{
  if (A đủ nhỏ)
      Solve (A);
  else {
      Chia bài toán A thành các bài toán con
          A1, A2,…, Am;
      for (i = 1; i <= m ; i ++)
          DivideConquer (Ai , xi);
      Kết hợp các nghiệm xi của các bài toán con Ai (i=1, …, m) để nhận
      được nghiệm x của bài toán A;
      }
```

}

"Chia một bài toán thành các bài toán con" cần được hiểu là ta thực hiện các phép biến đổi, các tính toán cần thiết để đưa việc giải quyết bài toán đã cho về việc giải quyết các bài toán con cỡ nhỏ hơn.

Thuật toán tìm kiếm nhị phân (xem mục 4.4.2) là thuật toán được thiết kế dựa trên chiến lược chia-để-trị. Cho mảng A cỡ n được sắp xếp theo thứ tự tăng dần: A[0] ≤ … ≤ A[n-1]. Với x cho trước, ta cần tìm xem x có chứa trong mảng A hay không, tức là có hay không chỉ số 0 ≤ i ≤ n-1 sao cho A[i] = x. Kỹ thuật chia-để-trị gợi ý ta chia mảng A[0…n-1] thành 2 mảng con cỡ n/2 là A[0…k-1] và A[k+1…n-1], trong đó k là chỉ số đứng giữa mảng. So sánh x với A[k]. Nếu x = A[k] thì mảng A chứa x và i = k. Nếu không, do tính được sắp của mảng A, nếu x < A[k] ta tìm x trong mảng A[0…k-1], còn nếu x > A[k] ta tìm x trong mảng A[k+1…n-1].

Thuật toán Tháp Hà Nội (xem mục 15.5), thuật toán sắp xếp nhanh (QuickSort) và thuật toán sắp xếp hoà nhập (MergeSort) sẽ được trình bày trong chương sau cũng là các thuật toán được thiết kế bởi kỹ thuật chia-để-trị. Sau đây chúng ta đưa ra một ví dụ đơn giản minh hoạ cho kỹ thuật chia-để-trị.

1.2. Tìm max và min

Cho mảng A cỡ n, chúng ta cần tìm giá trị lớn nhất (max) và nhỏ nhất (min) của mảng này. Bài toán đơn giản này có thể giải quyết bằng các thuật toán khác nhau.

Một thuật toán rất tự nhiên và đơn giản là như nhau. Đầu tiên ta lấy max, min là giá trị đầu tiên A[0] của mảng. Sau đó so sánh max, min với từng giá trị A[i], 1 ≤ i ≤ n-1, và cập nhật max, min một cách thích ứng. Thuật toán này được mô tả bởi hàm sau:

```
SiMaxMin (A, max, min)
{
        max = min = A[0];
        for ( i = 1 ;  i < n , i ++)
                if (A[i] > max)
                        max = A[i];
                else if (A[i] < min)
                        min = A[i];
}
```

Thời gian thực hiện thuật toán này được quyết định bởi số phép so sánh x với các thành phần A[i]. Số lần lặp trong lệnh lặp for là n-1. Trong trường hợp xấu nhất (mảng A được sắp theo thứ tự giảm dần), mỗi lần lặp ta cần thực hiện 2 phép so sánh.

Như vậy, trong trường hợp xấu nhất, ta cần thực hiện 2(n-1) phép so sánh, tức là thời gian chạy của thuật toán là O(n).

Bây giờ ta áp dụng kỹ thuật chia-để-trị để đưa ra một thuật toán khác. Ta chia mảng A[0..n-1] thành các mảng con A[0..k] và A[k+1..n-1] với k = [n/2]. Nếu tìm được max, min của các mảng con A[0..k] và A[k+1..n-1], ta dễ dàng xác định được max, min trên mảng A[0..n-1]. Để tìm max, min trên mảng con ta tiếp tục chia đôi chúng. Quá trình sẽ dừng lại khi ta nhận được mảng con chỉ có một hoặc hai phần tử. Trong các trường hợp này ta xác định được dễ dàng max, min. Do đó, ta có thể đưa ra thuật toán sau:

```
MaxMin (i, j, max, min)
// Biến max, min ghi lại giá trị lớn nhất, nhỏ nhất trong mảng A[i..j]
{
    if (i = = j)
        max = min = A[i];
    else if (i = = j-1)
        if (A[i] < A[j])
            {
                max = A[j];
                min = A[i];
            }
        else    {
                max = A[i];
                min = A[j];
            }
    else {
        mid = (i+j) / 2;
        MaxMin (i, mid, max1, min1);
        MaxMin (mid + 1, j, max2, min2);
        if (max 1< max2)
                max = max2;
        else    max = max1;
        if (min1 < min2)
                min = min1;
        else    min = min2;
       }
```

}

Bây giờ ta đánh giá thời gian chạy của thuật toán này. Gọi T(n) là số phép so sánh cần thực hiện. Không khó khăn thấy rằng, T(n) được xác định bởi quan hệ đệ quy sau.

$T(1) = 0$

$T(2) = 1$

$T(n) = 2T(n/2) + 2$ với $n > 2$

Áp dụng phương pháp thế lặp, ta tính được T(n) như sau:

$T(n) = 2\,T(n/2) + 2$

$= 2^2T(n/2^2) + 2^2 + 2$

$= 2^3T(n/2^3) + 2^3 + 2^2 + 2$

……

$= 2^kT(n/2^k) + 2^k + 2^{k-1} + \ldots + 2$

Với k là số nguyên dương sao cho $2^k \le n < 2^{k+1}$, ta có

$T(n) = 2^kT(1) + 2^{k+1} - 2 = 2^{k+1} - 2 \le 2(n-1)$

Như vậy, T(n) = O(n).

2. THUẬT TOÁN ĐỆ QUY

Khi thiết kế thuật toán giải quyết một vấn đề bằng kỹ thuật chia-để-trị thì thuật toán thu được là thuật toán đệ quy. Thuật toán đệ quy được biểu diễn trong các ngôn ngữ lập trình bậc cao (chẳng hạn Pascal, C/C++) bởi các hàm đệ quy: đó là các hàm chứa các lời gọi hàm đến chính nó. Trong mục này chúng ta sẽ nêu lên các đặc điểm của thuật toán đệ quy và phân tích hiệu quả (về không gian và thời gian) của thuật toán đệ quy.

Đệ quy là một kỹ thuật đặc biệt quan trọng để giải quyết vấn đề. Có những vấn đề rất phức tạp, nhưng chúng ta có thể đưa ra thuật toán đệ quy rất đơn giản, sáng sủa và dễ hiểu. Cần phải hiểu rõ các đặc điểm của thuật toán đệ quy để có thể đưa ra các thuật toán đệ quy đúng đắn.

Giải thuật đệ quy cho một vấn đề cần phải thoả mãn các đòi hỏi sau:

1. Chứa lời giải cho các trường hợp đơn giản nhất của vấn đề. Các trường hợp này được gọi là các trường hợp cơ sở hay các trường hợp dừng.
2. Chứa các lời gọi đệ quy giải quyết các vấn đề con với cỡ nhỏ hơn.
3. Các lời gọi đệ quy sinh ra các lời gọi đệ quy khác và về tiềm năng các lời gọi đệ quy phải dẫn tới các trường hợp cơ sở.

Tính chất 3 là đặc biệt quan trọng, nếu không thoả mãn, hàm đệ quy sẽ chạy mãi không dừng. Ta xét hàm đệ quy tính giai thừa:

```
int     Fact(int  n)
{
        if (n = 0)
                return   1;
        else
                return   n * Fact(n-1);  // gọi đệ quy.
        }
```

Trong hàm đệ quy trên, trường hợp cơ sở là n = 0. Để tính Fact(n) cần thực hiện lời gọi Fact(n-1), lời gọi này lại dẫn đến lời gọi F(n-2),…, và cuối cùng dẫn tới lời gọi F(0), tức là dẫn tới trường hợp cơ sở.

Đệ quy và phép lặp. Đối với một vấn đề, có thể có hai cách giải: giải thuật đệ quy và giải thuật dùng phép lặp. Giải thuật đệ quy được mô tả bởi hàm đệ quy, còn giải thuật dùng phép lặp được mô tả bởi hàm chứa các lệnh lặp, để phân biệt với hàm đệ quy ta sẽ gọi là hàm lặp. Chẳng hạn, để tính giai thừa, ngoài hàm đệ quy ta có thể sử dụng hàm lặp sau:

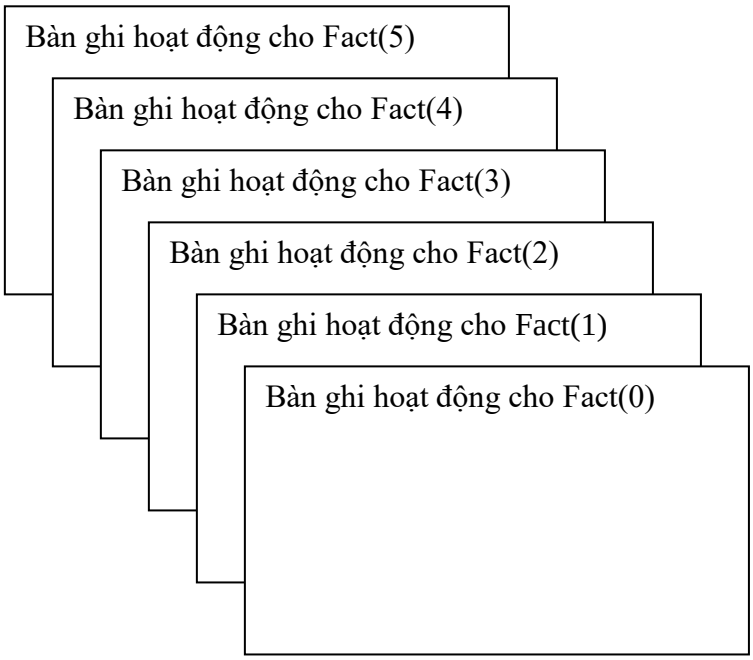
```
int     Fact(int n)
{
        if (n = = 0)
                return   1;
        else    {
                        int      F= 1;
                        for (int i = 1; i <= n ; i + +)
                                F = F * i;

                return   F;
                }
}
```

Ưu điểm nổi bật của đệ quy so với phép lặp là đệ quy cho phép ta đưa ra giải thuật rất đơn giản, dễ hiểu ngay cả đối với những vấn đề phức tạp. Trong khi đó, nếu không sử dụng đệ quy mà dùng phép lặp thì thuật toán thu được thường là phức tạp hơn, khó hiểu hơn. Ta có thể thấy điều đó trong ví dụ tính giai thừa, hoặc các thuật toán tìm kiếm, xem, loại trên cây tìm kiếm nhị phân (xem mục 8.4). Tuy nhiên, trong nhiều trường hợp, các thuật toán lặp lại hiệu quả hơn thuật toán đệ quy.

Bây giờ chúng ta phân tích các nhân tố có thể làm cho thuật toán đệ quy kém hiệu quả. Trước hết, ta cần biết cơ chế máy tính thực hiện một lời gọi hàm. Khi gặp

một lời gọi hàm, máy tính tạo ra một bản ghi hoạt động (activation record) ở ngăn xếp thời gian chạy (run-time stack) trong bộ nhớ của máy tính. Bản ghi hoạt động chứa vùng nhớ cấp cho các tham biến và các biến địa phương của hàm. Ngoài ra, nó còn chứa các thông tin để máy tính trở lại tiếp tục hiện chương trình đúng vị trí sau khi nó đã thực hiện xong lời gọi hàm. Khi hoàn thành thực hiện lời gọi hàm thì bản ghi hoạt động sẽ bị loại bỏ khỏi ngăn xếp thời gian chạy.

Khi thực hiện một hàm đệ quy, một dãy các lời gọi hàm được sinh ra. Hậu quả là một dãy bản ghi hoạt động được tạo ra trong ngăn xếp thời gian chạy. Cần chú ý rằng, một lời gọi hàm chỉ được thực hiện xong khi mà các lời gọi hàm mà nó sinh ra đã được thực hiện xong và do đó rất nhiều bản ghi hoạt động đồng thời tồn tại trong ngăn xếp thời gian chạy, chỉ khi một lời gọi hàm được thực hiện xong thì bản ghi hoạt động cấp cho nó mới được loại ngăn xếp thời gian chạy. Chẳng hạn, xét hàm đệ quy tính giai thừa, nếu thực hiện lời gọi hàm Fact(5) sẽ dẫn đến phải thực hiện các lời họi hàm Fact(4), Fact(3), Fact(2), Fact(1), Fact(0). Chỉ khi Fact(4) đã được tính thì Fact(5) mới được tính, … Do đó trong ngăn xếp thời gian chạy sẽ chứa các bản ghi hoạt động như sau:



Trong đó, bản ghi hoạt động cấp cho lời gọi hàm Fact(0) ở đỉnh ngăn xếp thời gian chạy. Khi thực hiện xong Fact(0) thì bản ghi hoạt động cấp cho nó bị loại, rồi bản ghi hoạt động cho Fact(1) bị loại,…

Vì vậy, việc thực hiện hàm đệ quy có thể đòi hỏi rất nhiều không gian nhớ trong ngăn xếp thời gian chạy, thậm chí có thể vượt quá khả năng của ngăn xếp thời gian chạy trong bộ nhớ của máy tính.

Một nhân tố khác làm cho các thuật toán đệ quy kém hiệu quả là các lời gọi đệ quy có thể dẫn đến phải tính nghiệm của cùng một bài toán con rất nhiều lần. Số Fibonacci thứ n, ký hiệu là F(n), được xác định đệ quy như sau:

F(1) = 1

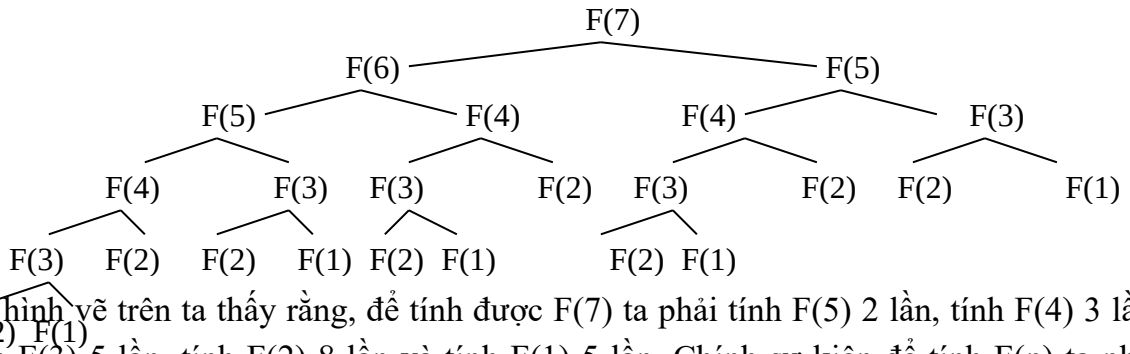
F(2) = 1

F(n) = F(n-1) + F(n-2) với n>2

Do đó, ta có thể tính F(n) bởi hàm đệ quy sau.

```
int      Fibo(int n)
{
        if ((n = = 1) // (n = = 2))
                return  1;
        else
                return  Fibo (n-1) + Fibo(n-2);
}
```

Để tính F(7), các lời gọi trong hàm đệ quy Fibo dẫn ta đến phải tính các F(k) với k<7, như được biểu diễn bởi cây trong hình dưới đây; chẳng hạn để tính F(7) cần tính F(6) và F(5), để tính F(6) cần tính F(5) và F(4), …



Từ hình vẽ trên ta thấy rằng, để tính được F(7) ta phải tính F(5) 2 lần, tính F(4) 3 lần, tính F(3) 5 lần, tính F(2) 8 lần và tính F(1) 5 lần. Chính sự kiện để tính F(n) ta phải tính các F(k), với k<n, rất nhiều lần đã làm cho hàm đệ quy Fibo kém hiệu quả. Có thể đánh giá thời gian chạy của nó là $O(\phi^n)$, trong đó = $(1 + \sqrt{5})/2$.

Chúng ta có thể đưa ra thuật toán lặp để tính dãy số Fibonacci. Ý tưởng của thuật toán là ta tính lần lượt các F(1), F(2), F(3), …, F(n -2), F(n-1), F(n) và sử dụng hai biến để lưu lại hai giá trị vừa tính. Hàm lặp tính dãy số Fibonacci như sau:

```
int      Fibo1(int n)
{
        if ((n= = 1)//(n= = 2)
                return 1;
        else {
```

```
            int   previous = 1;
            int   current   = 1;
        for (int k = 3 ; k <=  n ; k ++)
            {
                current + = previous;
                previous = current – previous;
            }
        return   current;
        }
}
```

Dễ dàng thấy rằng, thời gian chạy của hàm lặp Fibo1 là O(n). Để tính F(50) thuật toán lặp Fibo1 cần 1 micro giây, thuật toán đệ quy Fibo đòi hỏi 20 ngày, còn để tính F(100) thuật toán lặp cần 1,5 micro giây, trong khi thuật toán đệ quy cần 10^9 năm!

Tuy nhiên, có rất nhiều thuật toán đệ quy cũng hiệu quả như thuật toán lặp, chẳng hạn các thuật toán đệ quy tìm, xem, loại trên cây tìm kiếm nhị phân (xem mục 8.4). Các thuật toán đệ quy: sắp xếp nhanh (QuickSort) và sắp xếp hoà nhập (MergeSort) mà chúng ta sẽ nghiên cứu trong chương 17 cũng là các thuật toán rất hiệu quả.

Trong mục 6.6 chúng ta đã nghiên cứu kỹ thuật sử dụng ngăn xếp để chuyển thuật toán đệ quy thành thuật toán lặp. Nói chung, chỉ nên sử dụng thuật toán đệ quy khi mà không có thuật toán lặp hiệu quả hơn.

3. QUY HOẠCH ĐỘNG

3.1. Phương pháp chung

Kỹ thuật quy hoạch động giống kỹ thuật chia-để-trị ở chỗ cả hai đều giải quyết vấn đề bằng cách chia vấn đề thành các vấn đề con. Nhưng chia-để-trị là kỹ thuật top-down, nó tính nghiệm của các vấn đề con từ lớn tới nhỏ, nghiệm của các vấn đề con được tính độc lập bằng đệ quy. Đối lập, quy hoạch động là kỹ thuật bottom-up, tính nghiệm của các bài toán từ nhỏ đến lớn và ghi lại các kết quả đã tính được. Khi tính nghiệm của bài toán lớn thông qua nghiệm của các bài toán con, ta chỉ việc sử dụng các kết quả đã được ghi lại. Điều đó giúp ta tránh được phải tính nhiều lần nghiệm của cùng một bài toán con. Thuật toán được thiết kế bằng kỹ thuật quy hoạch động sẽ là thuật toán lặp, trong khi thuật toán được thiết kế bằng kỹ thuật chia-để-trị là thuật toán đệ quy. Để thuận tiện cho việc sử dụng lại nghiệm của các bài toán con, chúng ta lưu lại các nghiệm đã tính vào một bảng (thông thường là mảng 1 chiều hoặc 2 chiều).

Tóm lại, để giải một bài toán bằng quy hoạch động, chúng ta cần thực hiện các bước sau:

- Đưa ra cách tính nghiệm của các bài toán con đơn giản nhất.
- Tìm ra các công thức (hoặc các quy tắc) xây dựng nghiệm của bài toán thông qua nghiệm của các bài toán con.
- Thiết kế bảng để lưu nghiệm của các bài toán con.
- Tính nghiệm của các bài toán con từ nhỏ đến lớn và lưu vào bảng.
- Xây dựng nghiệm của bài toán từ bảng.

Một ví dụ đơn giản của thuật toán được thiết kế bằng quy hoạch động là thuật toán lặp tính dãy số Fibonacci mà ta đã đưa ra trong mục 16.2. Trong hàm lặp Fibo1, ta đã tính tuần tự F(1), F(2),…, đến F(n). Và bởi vì để tính F(k) chỉ cần biết F(k-1) và F(k-2), nên ta chỉ cần lưu lại F(k-1) và F(k-2).

Kỹ thuật quy hoạch động thường được áp dụng để giải quyết các **bài toán tối ưu** (optimization problems). Các bài toán tối ưu thường là có một số lớn nghiệm, mỗi nghiệm được gắn với một giá, và mục tiêu của chúng ta là tìm ra nghiệm có giá nhỏ nhất : **nghiệm tối ưu** (optimization solution). Chẳng hạn, bài toán tìm đường đi từ thành phố A đến thành phố B trong bản đồ giao thông, có nhiều đường đi từ A đến B, giá của một đường đi đó là độ dài của nó, nghiệm tối ưu là đường đi ngắn nhất từ A đến B. Nếu nghiệm tối ưu của bài toán được tạo thành từ nghiệm tối ưu của các bài toán con thì ta có thể sử dụng kỹ thuật quy hoạch động.

Sau đây, chúng ta sẽ đưa ra một số thuật toán được thiết kế bằng kỹ thuật quy hoạch động.

3.2. Bài toán sắp xếp các đồ vật vào ba lô

Giả sử ta có chiếc ba lô có thể chứa được một khối lượng w, chúng ta có n loại đồ vật được đánh số i,…, n. Mỗi đồ vật loại i (i = 1,…, n) có khối lượng a_i và có giá trị c_i. Chúng ta muốn sắp xếp các đồ vật vào ba lô để nhận được ba lô có gía trị lớn nhất có thể được. Giả sử mỗi loại đồ vật có đủ nhiều đề xếp vào ba lô.

Bài toán ba lô được mô tả chính xác như sau. Cho trước các số nguyên dương w, a_i, và c_i (i = 1,…,n). Chúng ta cần tìm các số nguyên không âm x_i (i = 1,…, n) sao cho

$$\sum_{n=1}^{n} x_i a_i \le w \text{ và}$$

$$\sum_{n=1}^{n} x_i c_i \text{ đạt giá trị lớn nhất.}$$

Xét trường hợp đơn giản nhất: chỉ có một loại đồ vật (n = 1). Trong trường hợp này ta tìm được ngay lời giải: xếp đồ vật vào ba lô cho tới khi nào không xếp được nữa thì thôi, tức là ta tìm được ngay nghiệm $x_i = w/a_i$.

Bây giờ ta đi tìm cách tính nghiệm của bài toán "xếp n loại đồ vật vào ba lô khối lượng w" thông qua nghiệm của các bài toán con "xếp k loại đồ vật ($1 \le k \le n$)

vào ba lô khối lượng v ($1 \le v \le w$)" Ta gọi tắt là bài toán con (k,w), gọi cost (k,v) là giá trị lớn nhất của ba lô khối lượng v ($1 \le v \le w$) và chỉ chứa các loại đồ vật 1, 2,….,k. Ta tìm công thức tính cost (k,v).Với k = 1 và $1 \le v \le w$, ta có

$x_i = v / a_i$ và

cost (1,v) = $x_i c_i$ (1)

Giả sử ta đã tính được cost (s,u) với $1 \le s < k$ và $1 \le u \le v$, ta cần tính cost (k,v) theo các cost (s,u) đã biết đó. Gọi $y_k = v / a_k$, ta có

cost (k,v) = max[cost (k-1,u) + $x_k c_k$] (2)

Trong đó, max được lấy với tất cả x_k = 0, 1,…, y_k và u = v - $x_k a_k$ (tức là được lấy với tất cả các khả năng xếp đồ vật thứ k). Như vậy, tính cost (k,v) được quy về tính cost (k-1,u) với $u \le v$. Giá trị của x_k trong (2) mà cost (k-1,u) + $x_k c_k$ đạt max chính là số đồ vật loại k cần xếp. Giá trị lớn nhất của ba lô sẽ là cost(n, w).

Chúng ta sẽ tính nghiệm của bài toán từ cỡ nhỏ đến cỡ lớn theo các công thức (1) và (2). Nghiệm của các bài toán con sẽ được lưu trong mảng 2 chiều A[0..n-1][0..w-1], cần lưu ý là nghiệm của bài toán con (k,v) được lưu giữ trong A[k-1][v-1], vì các chỉ số của mảng được đánh số từ 0.

Mỗi thành phần A[k-1][v-1] sẽ chứa cost(k,v) và số đồ vật loại k cần xếp. Từ các công thức (1) và (2) ta có thể tính được các thành phần của mảng A lần lượt theo dòng 0, 1,…n-1.

Từ bảng A đã làm đầy, làm thế nào xác định được nghiệm của bài toán, tức là xác định được số đồ vật loại i (i = 1,2,…,n) cần xếp vào ba lô? Ô A[n-1][w-1] chứa giá trị lớn nhất của ba lô cost (n,w) và số đồ vật loại n cần xếp x_n. Tính v = w – $x_n a_n$. Tìm đến ô A[n-2][v-1] ta biết được cost(n-1,v) và số đồ vật loại n-1 cần xếp x_{n-1}. Tiếp tục quá trình trên, ta tìm được x_{n-2},..,x_2 và cuối cùng là x_1.

3.3. Tìm dãy con chung của hai dãy số

Xét bài toán sau: Cho hai dãy số nguyên a = (a_1,…, a_m) và b = (b_1,…b_n), ta cần tìm dãy số nguyên c = (c_1,…, c_k) sao cho c là dãy con của cả a và b, và c là dài nhất có thể được. Ví dụ, nếu a = (3, 5, 1, 3, 5, 5, 3) và b = (1,5,3,5,3,1) thì dãy con chung dài nhất là c = (5,3,5,3) hoặc c = (1,3,5,3) hoặc c = (1,5,5,3).

Trường hợp đơn giản nhất khi một trong hai dãy a và b rỗng (m = 0 hoặc n = 0), ta thấy ngay dãy con chung dài nhất là dãy rỗng.

Ta xét các đoạt đầu của hai dãy a và b, đó là các dãy (a_1,a_2,…,a_i) và (b_1,b_2,…,a_j) với $0 \le i \le m$ và $0 \le j \le n$. Gọi L(i,j) là độ dài lớn nhất của dãy con chung của hai dãy (a_1,a_2,…,a_i) và (b_1,b_2,…,a_j). Do đó L(n,m) là độ dài lớn nhất của dãy con chung của a và b. Bây giờ ta đi tìm cách tính L(i,j) thông qua các L(s,t) với $0 \le s \le i$ và $0 \le t \le j$. Dễ dàng thấy rằng:

L(0,j) = 0 với mọi j

L(i,0) = 0 với mọi i (1)

Nếu i > 0 và j > 0 và $a_i \# b_j$ thì

L(i,j) = max [L(i,j-1), L(i-1,j)] (2)

Nếu i > 0 và j > 0 và $a_i = b_j$ thì

L(i,j) = 1 + L(i-1,j-1) (3)

Sử dụng các công thức đệ quy (1), (2), (3) để tính các L(i,j) lần lượt với i = 0,1,…,m và j = 0,1,…,n. Chúng ta sẽ lưu các giá trị L(i,j) vào mảng L[0..m][0..n].

Công việc tiếp theo là từ mảng L ta xây dựng dãy con chung dài nhất của a và b. Giả sử k = L[m][n] và dãy con chung dài nhất là c = (c_1,…c_{k-1}, c_k). Ta xác định các thành phần của dãy c lần lượt từ phải sang trái, tức là xác định c_k, rồi c_{k-1},…,c_1. Ta xem xét các thành phần của mảng L bắt từ L[m,n]. Giả sử ta đang ở ô L[i][j] và ta đang cần xác định c_r, (1 <= r <= k). Nếu $a_i = b_j$ thì theo (3) ta lấy $c_r = a_i$, giảm r đi 1 và đi đến ô L[i-1][j-1]. Còn nếu $a_i \# b_j$ thì theo (2) hoặc L[i][j] = L[i][j-1], hoặc L[i][j] = L[i-1][j]. Trong trường hợp L[i][j] = L[i][j-1] ta đi tới ô L[i][j-1], còn nếu L[i][j] = L[i-1][j] ta đi tới ô L[i-1][j]. Tiếp tục quá trình trên ta xác định được tất cả các thành phần của dãy con dài nhất.

4. QUAY LUI

4.1. Tìm kiếm vét cạn

Trong thực tế chúng ta thường gặp các câu hỏi chẳng hạn như “có bao nhiêu khả năng...?”, “hãy cho biết tất cả các khả năng...?”, hoặc “có tồn tại hay không một khả năng...?”. Ví dụ, có hay không một cách đặt 8 con hậu vào bàn cờ sao cho chúng không tấn công nhau. Các vấn đề như thế thông thường đòi hỏi ta phải xem xét tất cả các khả năng có thể có. Tìm **kiếm vét cạn** (exhaustive search) là xem xét tất cả các ứng cử viên nhằm phát hiện ra đối tượng mong muốn. Các thuật toán được thiết kế bằng tìm kiếm vét cạn thường được gọi là **brute-force algorithms**. Ý tưởng của các thuật toán này là sinh-kiểm, tức là sinh ra tất cả các khả năng có thể có và kiểm tra mỗi khả năng xem nó có thoả mãn các điều kiện của bài toán không. Trong nhiều vấn đề, tất cả các khả năng mà ta cần xem xét có thể quy về các đối tượng tổ hợp (các tập con của một tập), hoặc các hoán vị của n đối tượng, hoặc các tổ hợp k đối tượng từ n đối tượng. Trong các trường hợp như thế, ta cần phải sinh ra, chẳng hạn, tất cả các hoán vị, rồi kiểm tra xem mỗi hoán vị có là nghiệm của bài toán không. Tìm kiếm vét cạn đương nhiên là kém hiệu quả, đòi hỏi rất nhiều thời gian. Nhưng cũng có vấn đề ta không có cách giải quyết nào khác tìm kiếm vét cạn.

Ví dụ 1(Bài toán 8 con hậu). Chúng ta cần đặt 8 con hậu vào bàn cờ 8x8 sao cho chúng không tấn công nhau, tức là không có hai con hậu nào nằm cùng hàng, hoặc cùng cột, hoặc cùng đường chéo.

Vì các con hậu phải nằm trên các hàng khác nhau, ta có thể đánh số các con hậu từ 1 đến 8, con hậu i là con hậu đứng ở hàng thứ i (i=1,...,8). Gọi x_i là cột mà con hậu thứ i đứng. Vì các con hậu phải đứng ở các cột khác nhau, nên (x_1, x_2, ...,x_8) là một

hoán vị của 8 số 1, 2,..., 8. Như vậy tất cả các ứng cử viên cho nghiệm của bài toán 8 con hậu là tất cả các hoán vị của 8 số 1, 2,..., 8. Đến đây ta có thể đưa ra thuật toán như sau: sinh ra tất cả các hoán vị của (x_1, x_2, ...,x_8), với mỗi hoán vị ta kiểm tra xem hai ô bất kì (i,x_i) và (j,x_j) có cùng đường chéo hay không.

Đối với bài toán tổng quát: đặt n con hậu vào bàn cờ nxn, số các hoán vị cần xem xét là n!, và do dó thuật toán đặt n con hậu bằng tìm kiếm vét cạn đòi hỏi thời gian O(n!). Trong mục sau, chúng ta sẽ đưa ra thuật toán hiệu quả hơn được thiết kế bằng kỹ thuật quay lui.

Ví dụ 2(Bài toán người bán hàng).

Bài toán người bán hàng (saleperson problem) được phát biểu như sau. Một người bán hàng, hàng ngày phải đi giao hàng từ một thành phố đến một số thành phố khác rồi quay lại thành phố xuất phát. Anh ta muốn tìm một tua qua mỗi thành phố cần đến đúng một lần với độ dài của tua là ngắn nhất có thể được. Chúng ta phát biểu chính xác bài toán như sau. Cho đồ thị định hướng gồm n đỉnh được đánh số 0,1,...,n-1. Độ dài của cung (i,j) được kí hiệu là d_{ij} và là một số không âm. Nếu đồ thị không có cung (i,j) thì ta xem $d_{ij} = +\infty$. Chúng ta cần tìm một đường đi xuất phát từ một đỉnh qua tất cả các đỉnh khác của đồ thị đúng một lần rồi lại trở về đỉnh xuất phát (tức là tìm một chu trình Hamilton) sao cho độ dài của tua là nhỏ nhất có thể được. Mỗi tua như tế là một dãy các đỉnh (a_0, a_1,..., a_{n-1}, a_0), trong đó các a_0, a_1,..., a_{n-1} là khác nhau. Không mất tính tổng quat, ta có thể xem đỉnh xuất phát là đỉnh 0, $a_0 = 0$. Như vậy, mỗi tua tương ứng với một hoán vị (a_1,..., a_{n-1}) của các đỉnh 1, 2, ..., n-1. Từ đó ta có thuật toán sau: sinh ra tất cả các hoán vị của n-1 đỉnh 1, 2, ..., n-1; với mỗi hoán vị ta tính độ dài của tua tương ứng với hoán vị đó và so sánh các độ dài ta sẽ tìm được tua ngắn nhất. Lưu ý rằng, có tất cả (n-1)! hoán vị và mỗi tua cần n phép toán để tính độ dài, do đó thuật toán giải bài toán người bán hàng với n thành phố bằng tìm kiếm vét cạn cần thời gian O(n!).

Bài toán người bán hàng là bài toán kinh điển và nổi tiếng. Ngoài cách giải bằng tìm kiếm vét cạn, người ta đã đưa ra nhiều thuật toán khác cho bài toán này. Thuật toán quy hoạch động cho bài toán người bán hàng đòi hỏi thời gian $O(n^2 2^n)$. Cho tới nay người ta vẫn chưa tìm ra thuật toán có thời gian đa thức cho bài toán người bán hàng.

4.2. Quay lui

Quay lui (backtracking) là kỹ thuật thiết kế thuật toán có thể sử dụng để giải quyết rất nhiều vấn đề khác nhau. Ưu điểm của quay lui so với tìm kiếm vét cạn là ở chỗ có thể cho phép ta hạn chế các khả năng cần xem xét.

Trong nhiều vấn đề, việc tìm nghiệm của vấn đề được quy về tìm một dãy các trạng thái (a_1, a_2,…, a_k,…), trong đó mỗi a_i (i = 1,2,…) là một trạng thái được chọn ra từ một tập hữu hạn A_i các trạng thái, thoả mãn các điều kiện nào đó. Tìm kiếm vét cạn

đòi hỏi ta phải xem xét tất cả các dãy trạng thái đó để tìm ra dãy trạng thái thoả mãn các yêu cầu của bài toán.

Chúng ta sẽ gọi dãy các trạng thái (a_1, a_2,…, a_n) thoả mãn các yêu cầu của bài toán là vectơ nghiệm. Ý tưởng của kỹ thuật quay lui là ta xây dựng vectơ nghiệm xuất phát từ vectơ rỗng, mỗi bước ta bổ xung thêm một thành phần của vectơ nghiệm, lần lượt a_1,a_2,…

Đầu tiên, tập S_1 các ứng cử viên có thể là thành phần đầu tiên của vectơ nghiệm chính là A_1.

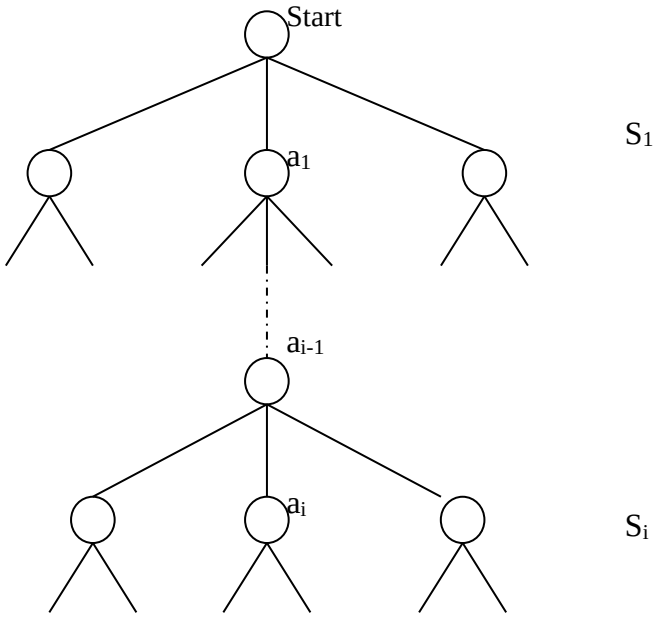
Chọn $a_1 \in S_1$, ta có vectơ (a_1). Giả sử sau bước thứ i-1, ta đã tìm được vectơ (a_1,a_2,…,a_{i-1}). Ta sẽ gọi các vectơ như thế là nghiệm một phần (nó thoả mãn các đòi hỏi của bài toán, những chưa "đầy đủ"). Bây giờ ta mở rộng nghiệm một phần (a_1,a_2,…,a_{i-1}) bằng cách bổ xung thêm thành phần thứ i. Muốn vậy, ta cần xác định tập S_i các ứng cử viên cho thành phần thứ i của vectơ nghiệm. Cần lưu ý rằng, tập S_i được xác định theo các yêu cầu của bài toán và các thành phần a_1,a_2,…,a_{i-1} đã chọn trước, và do đó S_i là tập con của tập A_i các trạng thái. Có hai khả năng

- Nếu S_i không rỗng, ta chọn $a_i \in S_i$ và thu được nghiệm một phần (a_1,a_2,…,a_{i-1},a_i), đồng thời loại a_i đã chọn khỏi S_i. Sau đó ta lại tiếp tục mở rộng nghiệm một phần (a_1,a_2,…,a_i) bằng cách áp dụng đệ quy thủ tục mở rộng nghiệm.
- Nếu S_i rỗng, điều này có nghĩa là ta không thể mở rộng nghiệm một phần (a_1,a_2,…,a_{i-2},a_{i-1}), thì ta quay lại chọn phần tử mới a'_{i-1} trong S_{i-1} làm thành phần thứ i-1 của vectơ nghiệm. Nếu thành công (khi S_{i-1} không rỗng) ta nhận được vectơ (a_1,a_2,…,a_{i-2},a'_{i-1}) rồi tiếp tục mở rộng nghiệm một phần này. Nếu không chọn được a'_{i-1} thì ta quay lui tiếp để chọn a'_{i-2}… Khi quay lui để chọn a'_1 mà S_1 đã trở thành rỗng thì thuật toán dừng.

Trong quá trình mở rộng nghiệm một phần, ta cần kiểm tra xem nó có là nghiệm không. Nếu là nghiệm, ta ghi lại hoặc in ra nghiệm này. Kỹ thuật quay lui cho phép ta tìm ra tất cả các nghiệm của bài toán.

Kỹ thuật quay lui mà ta đã trình bày thực chất là kỹ thuật đi qua cây tìm kiếm theo độ sâu (đi qua cây theo thứ tự preorder). Cây tìm kiếm được xây dựng như sau

- Các đỉnh con của gốc là các trạng thái của S_1
- Giả sử a_{i-1} là một đỉnh ở mức thứ i-1 của cây. Khi đó các đỉnh con của a_{i-1} sẽ là các trạng thái thuộc tập ứng cử viên S_i. Cây tìm kiếm được thể hiện trong hình 16.1.



Hình 16.1. Cây tìm kiếm vectơ nghiệm

Trong cây tìm kiếm, mỗi đường đi từ gốc tới một đỉnh tương ứng với một nghiệm một phần.

Khi áp dụng kỹ thuật quay lui để giải quyết một vấn đề, thuật toán được thiết kế có thể là đệ quy hoặc lặp. Sau đây ta sẽ đưa ra lược đồ tổng quát của thuật toán quay lui.

Lược đồ thuật toán quay lui đệ quy. Giả sử vector là nghiệm một phần $(a_1,a_2,\ldots,a_{i-1})$. Hàm đệ quy chọn thành phần thứ i của vector nghiệm là như sau:

```
Backtrack(vector , i)
// Chọn thành phần thứ i của vector.
{
        if (vector là nghiệm)
                viết ra nghiệm;
       Tính Si;
        for (mỗi ai∈Si)
                Backtrack(vector + (ai) , i+1);
}
```

Trong hàm trên, nếu vector là nghiệm một phần $(a_1,\ldots,a_{i-1})$ thì vector + (a_i) là nghiệm một phần $(a_1,a_2,\ldots,a_{i-1},a_i)$. Để tìm ra tất cả các nghiệm, ta chỉ cần gọi Backtrack(vector,1), với vector là vector rỗng.

Lược đồ thuật toán quay lui không đệ quy

```
Backtrack
{
```

```
k = 1;
Tính   S1;
while (k>0)
 {
    if (Sk không rỗng)
      {
          chọn ak ∈ Sk;
          Loại ak khỏi Sk;
          if ((a1,…,ak) là nghiệm)
                viết ra nghiệm;
          k++;
          Tính   Sk;
      }
    else   k-- ;    //Quay lui
  }
}
```

Chú ý rằng, khi cài đặt thuật toán theo lược đồ không đệ quy, chúng ta cần biết cách lưu lại vết của các tập ứng viên S_1, S_2,…,S_k để khi quay lui ta có thể chọn được thành phần mới cho vectơ nghiệm.

Ví dụ 3. Thuật toán quay lui cho bài toán 8 con hậu. Hình 16.2. mô tả một nghiệm của bài toán 8 con hậu.

	0	1	2	3	4	5	6	7
0	x							
1							x	
2					x			
3								x
4		x						
5				x				
6						x		
7			x					

Hình 16.2. Một nghiệm của bài toán 8 con hậu

Như trong ví dụ 1, ta gọi cột của con hậu ở dòng i (i = 0,1,..,7) là x_i. Nghiệm của bài toán là vectơ (x_0,x_1,...,x_7), chẳng hạn nghiệm trong hình 16.2 là (0,6,4,7,1,3,5,2). Con hậu 0 (ở dòng 0) có thể được đặt ở một trong tám cột. Do đó S_0={0,1,…,7}. Khi ta đã đặt con hậu 0 ở cột 0 (x_0=0), con hậu 1 ở cột 6 (x_1=6), như trong hình 16.2, thì con hậu 2 chỉ có thể đặt ở một trong các cột 1,3,4. Tổng quát, khi ta đã đặt các con hậu 0,1,2,…,k-1 thì con hậu k (con hậu ở dòng k) chỉ có thể đặt ở một trong các cột khác với các cột mà các con hậu 0,1,2,…,k-1 đã chiếm và không cùng đường chéo với chúng. Điều đó có nghiã là khi đã chọn được nghiệm một phần (x_0,x_1,…,x_{k-1}) thì x_k chỉ có thể lấy trong tập ứng viên S_k được xác định như sau

$S_k = \{x_k \in \{0,1,\ldots,7\} \mid x_k \neq x_i$ và $| i-k | \neq | x_k - x_i |$ với mọi $i < k\}$

Từ đó ta có thể đưa ra thuật toán sau đây cho bài toán 8 hậu

```
void      Queen(int   x[8])
{
   int   k = 0;
   x[0] = -1;
   while (k>0)
    {
        x[k]++;
        if (x[k]<=7)
          {
                int i;
                for (i = 0 ; i < k ; i++)
                if ((x[k] == x[i]) | | (fabs(i-k) == fabs(x[k] - x[i])))
                        break;     // kiểm tra xem x[k] có thuộc Sk
               if (i == k)        // chỉ khi x[k] ∈Sk
                    if (k == 7)
                            viết ra mảng x;
                    else
                        {
                            k++;
                            x[k] = -1;
                         }
          }
        else    k--;     //quay lui
    }     // Hết vòng lặp while
```

}

Ví dụ 4. Các dãy con có tổng cho trước

Cho một dãy số nguyên dương (a_0,a_1,…,a_{n-1}) và một số nguyên dương M. Ta cần tìm các dãy con của dãy sao cho tổng của các phần tử trong dãy con đó bằng M. Chẳng hạn, với dãy số (7,1,4,3,5,6) và M=11, thì các dãy con cần tìm là (7,1,3), (7,4), (1,4,6) và (5,6).

Sử dụng kỹ thuật quay lui, ta xác định dãy con (a_{i0},a_{i1},…,a_{ik}) sao cho $a_{i0}+a_{i1}+…+a_{ik}$ = M bằng cách chọn lần lượt a_{i0},a_{i1},…Ta có thể chọn a_{i0} là một trong a_0,a_1,…,a_{n-1} mà nó <= M, tức là có thể chọn a_{i0} với i_0 thuộc tập ứng viên S_0 = {i ∈ {0,1,…,n-1} | a_i <= M}. Khi đã chọn được (a_{i0},a_{i1},…,a_{ik-1}) với S = a_{i0} + a_{i1} + … + a_{ik-1} < M thì ta có thể chọn a_{ik} với i_k là một trong các chỉ số bắt đầu từ i_{k-1}+1 tới n-1 và sao cho S+a_{ik} <= M. Tức là, ta có thể chọn a_{ik} với i_k thuộc tập S_k = {i ∈ {i_{k-1} +1,…, n-1} | S+a_i <= M}. Giả sử dãy số đã cho được lưu trong mảng A. Lưu dãy chỉ số {i_0,i_1,…,i_k} của dãy con cần tìm vào mảng I, ta có thuật toán sau

```
void    SubSequences(int A[n], int M, int I[n])
{
        k  =  0;
        I[0]  =  -1;
        int   S  =  0;
        while (k > 0)
        {
           I[k]++;
           If (I[k] < n)
                {
                  if (S + A[i[k]] <= M)
                       if (S + A[i[k]] == M)
                                viết ra mảng I[0..k];
                       else
                       {
                                S  =  S + A[i[k]];
                                I[k+1]  =  I[k];
                                k++;
                       }

                }
           else
```

```
                {
                    k --;
                    S = S - A[i[k]];
                }
            }
        }
```

4.3. Kỹ thuật quay lui để giải bài toán tối ưu

Trong mục này chúng ta sẽ áp dụng kỹ thuật quay lui để tìm nghiệm của bài toán tối ưu.

Giả sử nghiệm của bài toán có thể biểu diễn dưới dạng $(a_1,..,a_n)$, trong đó mỗi thành phần a_i (i = 1,…,n) được chọn ra từ tập S_i các ứng viên. Mỗi nghiệm $(a_1,..,a_n)$ của bài toán có một giá $cost(a_1,..,a_n) >= 0$, và ta cần tìm nghiệm có giá thấp nhất (nghiệm tối ưu).

Giả sử rằng, giá của các nghiệm một phần là không giảm, tức là nếu $(a_1,..,a_{k-1})$ là nghiệm một phần và $(a_1,..,a_{k-1},a_k)$ là nghiệm mở rộng của nó thì

$cost(a_1,..,a_{k-1}) <= cost(a_1,..,a_{k-1},a_k)$

Trong quá trình mở rộng nghiệm một phần (bằng kỹ thuật quay lui), khi tìm được nghiệm một phần $(a_1,..,a_k)$, nếu biết rằng tất cả các nghiệm mở rộng của nó $(a_1,..,a_k,a_{k+1},...)$ đều có giá lớn hơn giá của nghiệm tốt nhất đã biết ở thời điểm đó, thì ta không cần mở rộng nghiệm một phần $(a_1,..,a_k)$ đó.

Giả sử $cost^*(a_1,..,a_k)$ là cận dưới của giá của tất cả các nghiệm $(a_1,..,a_k,a_{k+1},...)$ mà nó là mở rộng của nghiệm một phần $(a_1,..,a_k)$. Giả sử giá của nghiệm tốt nhất mà ta đã tìm ra trong quá trình tìm kiếm là lowcost. (Ban đầu lowcost được khởi tạo là $+\infty$ và giá trị của nó được cập nhật trong quá trình tìm kiếm). Khi ta đạt tới nghiệm một phần $(a_1,..,a_k)$ mà $cost^*(a_1,..,a_k) >$ lowcost thì ta không cần mở rộng nghiệm một phần $(a_1,..,a_k)$ nữa; điều đó có nghĩa là, trong cây tìm kiếm hình 16.1 ta cắt bỏ đi tất cả các nhánh từ đỉnh a_k.

Từ các điều trình bày trên, ta đưa ra lược đồ thuật toán tìm nghiệm tối ưu sau. Thuật toán này thường được gọi là **thuật toán nhánh–và--cận** (branch – and – bound).

```
                BranchBound
{
   lowcost  =  +∞;
   cost*  =  0;
   k  =  1;
```

```
tính  S1;
while (k > 0)
 {
    if (Sk không rỗng và cost* <= lowcost)
    {
            chọn ak ∈ Sk;
            Loại ak ra khỏi Sk;
            cost* = cost*(a1,..,ak);
            if ((a1,..,ak) là nghiệm)
                    if (cost(a1,..,ak) <  lowcost)
                            lowcost  = cost(a1,..,ak);
            k++;
            tính   Sk;
    }
 else
    {
           k--;
            cost* = cost(a1,..,ak);
     }
  }
}
```

Ưu điểm của thuật toán nhánh – và - cận là ở chỗ nó cho phép ta không cần phải xem xét tất cả các nghiệm vẫn có thể tìm được nghiệm tối ưu. Cái khó nhất trong việc áp dụng kỹ thuật nhánh và cận là xây dựng hàm đánh giá cận dưới cost* của các nghiệm là mở rộng của nghiệm một phần. Đánh giá cận dưới có chặt mới giúp ta cắt bỏ được nhiều nhánh không cần thiết phải xem xét tiếp, và do đó thuật toán nhận được mới nhanh hơn đáng kể so với thuật toán tìm kiếm vét cạn.

5. CHIẾN LƯỢC THAM ĂN

5.1. Phương pháp chung

Các bài toán tối ưu thường là có một số rất lớn nghiệm, việc tìm ra nghiệm tối ưu (nghiệm có giá thấp nhất) đòi hỏi rất nhiều thời gian. Điển hình là bài toán người bán hàng, thuật toán quy hoạch động cũng đòi hỏi thời gian $O(n^2 2^n)$, và cho tới nay người ta vẫn chưa tìm ra thuật toán có thời gian đa thức cho bài toán này.

Một cách tiếp cận khác để giải quyết các bài toán tối ưu là chiến lược tham ăn (greedy strategy).

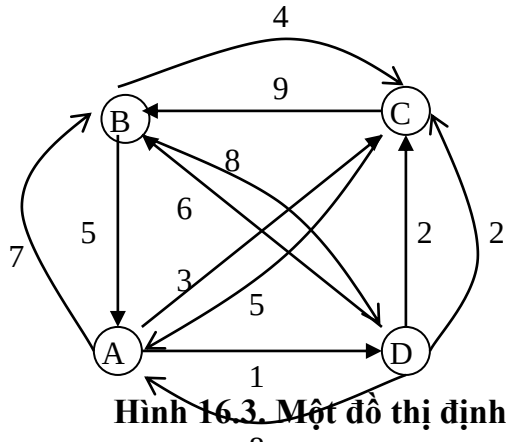
Trong hầu hết các bài toán tối ưu, để nhận được nghiệm tối ưu chúng ta có thể đưa về sự thực hiện một dãy quyết định. Ý tưởng của chiến lược tham ăn là, tại mỗi bước ta sẽ lựa chọn quyết định để thực hiện là quyết định được xem là tốt nhất trong ngữ cảnh nào đó được xác định bởi bài toán. Tức là, quyết định được lựa chọn ở mỗi bước là quyết định tối ưu địa phương. Tùy theo từng bài toán mà ta đưa ra tiêu chuẩn lựa chọn quyết định cho thích hợp.

Các thuật toán tham ăn (greedy algorithm) nói chung là đơn giản và hiệu quả (vì các tính toán để tìm ra quyết định tối ưu địa phương thường là đơn giản). Tuy nhiên, các thuật toán tham ăn có thể không tìm được nghiệm tối ưu, nói chung nó chỉ cho ra nghiệm gần tối ưu, nghiệm tương đối tốt. Nhưng cũng có nhiều thuật toán được thiết kế theo kỹ thuật tham ăn cho ta nghiệm tối ưu, chẳng hạn thuật toán Dijkstra tìm đường đi ngắn nhất từ một đỉnh tới các đỉnh còn lại trong đồ thị định hướng, các thuật toán Prim và Kruskal tìm cây bao chùm ngắn nhất trong đồ thị vô hướng, chúng ta sẽ trình bày các thuật toán này trong chương 18.

5.2. Thuật toán tham ăn cho bài toán người bán hàng

Giả sử đồ thị mà ta xét là đồ thị định hướng n đỉnh được đánh số 0,1,2,…,n-1, và là đồ thị đầy đủ, tức là với mọi $0 \le i , j \le$ n-1 đều có cung đi từ i đến j với độ dài là số thực không âm d(i,j). Giả sử đỉnh xuất phát là đỉnh 0, và đường đi ngắn nhất mà ta cần tìm là (0, a_1, a_1,… a_{n-1}, 0) trong đó $a_k \in \{1,2,\ldots,n-1\}$. Để nhận được nghiệm tối ưu trên, tại mỗi bước k (k = 1,…, n-1) chúng ta cần chọn một đỉnh a_k để đi tới trong số các đỉnh chưa thăm (tức là $a_k \ne a_i$, i = 1,…, k-1). Với mong muốn đường đi nhận được là ngắn nhất, ta đưa ra tiêu chuẩn chọn đỉnh a_k ở mỗi bước là đỉnh gần nhất trong số các đỉnh chưa thăm.

Ví dụ. Xét đồ thị định hướng trong hình 16.3



Hình 16.3. Một đồ thị định hướng

Giả sử ta cần tìm tua ngắn nhất xuất phát từ A. Vì d(A,B) = 7, d(A,C) = 3 và d(A,D) = 1, nên ta chọn đỉnh D để đi tới, ta có đường đi (A, D). Từ D, các đỉnh chưa thăm là B và C, ta chọn C để đi tới vì C gần D hơn là B. Ta thu được đường đi (A, D,

C). Từ C ta chỉ có một khả năng là đi tới B. Do đó ta nhận được tua (A, D, C, B, A). Độ dài của nó là 1 + 2 + 9 + 5 = 17. Đây không phải là đường đi ngắn nhất, vì đường đi ngắn nhất là (A, C, D, B, A) có độ dài 3 + 2 + 6 + 5 = 16.

5.3. Thuật toán tham ăn cho bài toán ba lô

Chúng ta trở lại bài toán ba lô đã đưa ra trong mục 16.3.2. Chúng ta cần nhận được chiếc ba lô chứa đồ vật có giá trị lớn nhất. Một cách tiếp cận khác để có chiếc ba lô đó là mỗi bước xếp một loại đồ vật vào ba lô. Vấn đề đặt ra là tại bước k (k = 1,2,…) ta cần chọn loại đồ vật nào để xếp và xếp bao nhiêu đồ vật loại đó. Từ các đòi hỏi của bài toán, ta đưa ra tiêu chuẩn chọn như sau: tại mỗi bước ta sẽ chọn loại đồ vật có giá trị lớn nhất trên một đơn vị khối lượng (gọi tắt là tỷ giá) trong số các loại đồ vật chưa được xếp vào ba lô. Khi đã chọn một loại đồ vật thì ta xếp tối đa có thể được.

Ví dụ, giả sử ta có ba lô chứa được khối lượng w = 20. Chúng ta có 4 loại đồ vật có khối lượng a_i và giá trị c_i (i = 1,2,3,4) được cho trong bảng sau:

Loại	1	2	3	4
Khối lượng a_i	5	7	8	3
Giá trị c_i	21	42	20	9
Tỷ giá c_i/a_i	4,2	6	2,5	3

Đầu tiên trong 4 loại đồ vật thì loại có tỷ giá lớn nhất là loại 2. Ta có thể xếp được tối đa 2 đồ vật loại 2 vào ba lô, và khối lượng còn lại của ba lô là 20 – 2 . 7 = 6. Đến đây số loại đồ vật chưa xếp: 1 , 3 , 4 , loại có tỷ giá lớn nhất là loại 1. Khối lượng còn lại của ba lô là 6, nên chỉ xếp được 1 đồ vật loại 1. Bước tiếp theo ta chọn loại 4, nhưng khối lượng còn lại của ba lô là 6-1.5 = 1, nên không xếp được đồ vật loại 4 (vì đồ vật loại 4 có khối lượng 3 > 1). Chọn đồ vật loại 3, cũng không xếp được, và dừng lại. Như vậy ta thu được ba lô chứa 2 đồ vật loại 2 và 1 đồ vật loại 1, với giá trị là 2.42 + 1.21 = 105.

Thuật toán tham ăn xếp các đồ vật vào ba lô như đã trình bày cũng không tìm ra nghiệm tối ưu, mà chỉ cho ra nghiệm tốt. Thế thì tại sao ta lại cần đến các thuật toán tham ăn mà nó chỉ cho ra nghiệm tốt, gần đúng với nghiệm tối ưu. Vấn đề là ở chỗ, đối với nhiều bài toán, chẳng hạn bài toán người bán hàng, bài toán sơn đồ thị,…, các thuật toán tìm ra nghiệm chính xác đòi hỏi thời gian mũ, không sử dụng được trong thực tế khi cỡ bài toán khá lớn. Còn có những bài toán để tìm ra nghiệm tối ưu ta chỉ còn có cách là tìm kiếm vét cạn. Trong các trường hợp như thế, sử dụng các thuật toán tham ăn là cần thiết, bởi vì các thuật toán tham ăn thường là đơn giản, rất hiệu quả, và thực tế nhiều khi có được một nghiệm tốt cũng là đủ.

6. THUẬT TOÁN NGẪU NHIÊN

Khi trong một bước nào đó của thuật toán, ta cần phải lựa chọn một trong nhiều khả năng, thay vì phải tiêu tốn thời gian xem xét tất cả các khả năng để có sự lựa chọn

tối ưu, người ta có thể chọn ngẫu nhiên một khả năng. Sự lựa chọn ngẫu nhiên lại càng thích hợp cho các trường hợp khi mà hầu hết các khả năng đều "tốt" ngang nhau. Các thuật toán chứa sự lựa chọn ngẫu nhiên được gọi là các thuật toán ngẫu nhiên (randomized algorithm hay probabilistic algorithm). Đặc trưng của thuật toán ngẫu nhiên là, kết quả của thuật toán không chỉ phụ thuộc vào giá trị đầu vào của thuật toán mà còn phụ thuộc vào giá trị ngẫu nhiên được sinh ra bởi hàm sinh số ngẫu nhiên. Nếu ta cho chạy thuật toán ngẫu nhiên hai lần trên cùng một dữ liệu vào, thuật toán có thể cho ra kết quả khác nhau.

Trong các thuật toán ngẫu nhiên, ta cần sử dụng các hàm sinh số ngẫu nhiên (random number generator). Trong các thuật toán ngẫu nhiên sẽ đưa ra sau này, ta giả sử đã có sẵn các hàm sinh số ngẫu nhiên sau. Hàm RandInt(i,j), trong đó i, j là các số nguyên và 0 <= i <= j, trả về một số nguyên ngẫu nhiên k, i <= k <= j. Hàm RandReal(a,b), trong đó a, b là các số thực và a < b, trả về một số thực ngẫu nhiên x , a <= x <= b.

Các thuật toán ngẫu nhiên hay gặp thường là thuộc một trong các lớp sau:

* Các thuật toán tính nghiệm gần đúng của các bài toán số.

* Các thuật toán Monte Carlo. Đặc điểm của các thuật toán này là nó luôn cho ra câu trả lời , song câu trả lời có thể không đúng. Xác suất thành công (tức là nhận được câu trả lời đúng) sẽ tăng, khi ta thực hiện lặp lại thuật toán.

* Các thuật toán Las Vegas. Các thuật toán này không bao giờ cho ra câu trả lời sai, song có thể nó không tìm ra câu trả lời. Xác suất thất bại (không tìm ra câu trả lời) có thể là nhỏ tùy ý, khi ta lặp lại thuật toán một số lần đủ lớn với cùng một dữ liệu vào.

Các thuật toán ngẫu nhiên rất đa dạng và phong phú, và có trong nhiều lĩnh vực khác nhau. Sau đây ta đưa ra một số ví dụ minh họa.

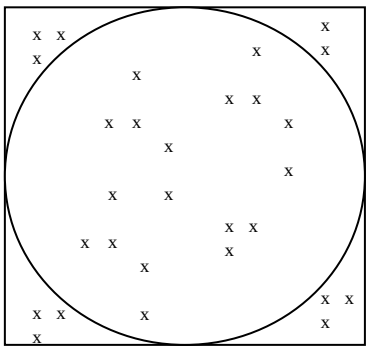
Ví dụ 1. Tính gần đúng số $\prod$

Ta có một hình vuông ngoại tiếp một hình tròn bán kính r (xem hình 16). Ta tiến hành thực nghiệm sau. Ném n hạt vào hình vuông này, giả sử rằng, mọi điểm trong hình vuông này "là điểm rơi khi ta ném một hạt vào hình vuông" với xác suất là như nhau. Diện tích của hình tròn là $\prod r^2$, và diện tích của hình vuông là $4r^2$, do đó $\prod r^2 /4r^2 = \prod/4$

Giả sử số hạt rơi vào trong hình tròn là k, ta có thể đánh giá $\prod$ = 4k/n. Thực nghiệm trên được mô tả bởi thuật toán sau:

```
k = 0;
for (i = 0 ; i < n ; i++)
{
   x  =  RandReal(-r,r);
   y  =  RandReal(-r,r);
```

```
    if ( điểm (x,y) nằm trong hình tròn )
                    k++
}
∏ = 4k/n;
```



Hình 16. Ném các hạt để tính ∏

Ví dụ 2. Tính gần đúng tính phân xác định

Giả sử ta cần tính tích phân xác định

$$\int_a^b f(x)dx$$

Giả sử tích phân này tồn tại. Ta chọn ngẫu nhiên n điểm trên đoạn [a,b]. Khi đó giá trị của tích phân có thể đánh giá là trung bình cộng các giá trị của hàm f(x) trên các điểm đã chọn nhân với độ dài của đoạn lấy tích phân. Ta có thuật toán sau:

```
Integral(f, a, b)
{
    sum = 0;
    for ( i = 0 ; i < n ; i++)
     {
        x  = RandReal(a, b);
        sum  =  sum+f(x);
      }
    return     (b-a) * (sum / n);

}
```

Ví dụ 3. Phần tử đa số.

Chúng ta gọi phần tử đa số trong một mảng n phần tử A[0..n-1] là phần tử mà số phần tử bằng nó trong mảng A lớn hơn n/2. Với mảng A cho trước ta cần biết mảng A có chứa phần tử đa số hay không. Ta đưa ra thuật toán đơn giản sau. Chọn ngẫu nhiên một phần tử bất kỳ trong mảng A và kiểm tra xem nó có là phần tử đa số hay không.

```
bool     Majority( A[0..n-1] )
{
     i  =  RandInt(0,n-1);
    x  =  A[i];
    k  =  0;
    for ( j = 0 ; j < n ; j++)
        if (A[j] = = x)
                k++;
    return   (k > n / 2);
}
```

Nếu mảng A không chứa phần tử đa số, thuật toán trên luôn trả về false (tức là luôn luôn cho câu trả lời đúng). Giả sử mảng A chứa phần tử đa số. Khi đó thuật toán có thể cho câu trả lời sai. Nhưng vì phần tử đa số chiếm quá nửa số phần tử trong mảng, nên xác suất chọn ngẫu nhiên được phần tử đa số là p > 1/2, tức là xác suất thuật toán cho câu trả lời đúng là p > 1/2. Bây giờ cho chạy thuật toán trên hai lần và thử tính xem xác suất để "lần đầu nhận được câu trả lời đúng hoặc lần đầu nhận được câu trả lời sai và lần hai nhận được câu trả lời đúng" là bao nhiêu. Xác suất này bằng

$p + (1 – p)\, p = 1- (1 - p)^2 > 3 / 4$

Như vậy có thể kết luận rằng, nếu mảng A chứa phần tử đa số, thì thực hiện lặp lại thuật toán trên một số lần đủ lớn, ta sẽ tìm được phần tử đa số. Thuật toán trên là thuật toán Monte Carlo.

Ví dụ 4. Thuật toán Las Vegas cho bài toán 8 con hậu

Chúng ta nhìn lại bài toán 8 con hậu đã đưa ra trong mục 16.4. Nhớ lại rằng, nghiệm của bài toán là vectơ $(x_0, x_1, \ldots, x_7)$, trong đó x_i là cột của con hậu ở dòng thứ i (i = 0, 1, …,7). Trong thuật toán quay lui, x_i được tìm bằng cách xem xét lần lượt các cột 0, 1, …,7 và quan tâm tới điều kiện các con hậu không tấn công nhau. Nhưng quan sát các nghiệm tìm được ta thấy rằng không có một quy luật nào về các vị trí của các con hậu. Điều đó gợi ý ta đưa ra thuật toán ngẫu nhiên sau. Để đặt con hậu thứ i, ta đặt nó ngẫu nhiên vào một trong các cột có thể đặt (tức là khi đặt con hậu thứ i vào cột đó, thì nó không tấn công các con hậu đã đặt). Việc đặt ngẫu nhiên như thế có thể không dẫn tới nghiệm, bởi vì các con hậu đã đặt có thể khống chế mọi vị trí và do đó không thể đặt con hậu tiếp theo.

BÀI TẬP

Thiết kế thuật toán bằng kỹ thuật chia - để - trị cho các bài toán sau:

1. (Trao đổi hai phần của một mảng). Cho mảng A[0…n-1], ta cần trao đổi k phần tử đầu tiên của mảng ($1 \le k < n$) với n – k phần tử còn lại, nhưng không được sử dụng mảng phụ. Chẳng hạn, với k = 3 và A là mảng như sau:

A :

a	b	c	d	e	f	g

Sau khi trao đổi ta cần nhận được mảng:

A :

d	e	f	g	a	b	c

2. (Dãy con không giảm dài nhất). Cho một dãy số nguyên được lưu trong mảng A[0 … n-1], ta cần tìm dãy chỉ số $0 \le i_1 < i_2 < \ldots < i_k \le n - 1$ sao cho $A[i_0] \le A[i_1] \le \ldots \le A[i_k]$ và k là lớn nhất có thể được. Ví dụ, nếu a = (8, 3, 7, 4, 2, 5, 3, 6) thì dãy con không giảm dài nhất là (3, 4, 5, 6).

3. Cho hai dãy không giảm $A = (a_1, a_2, \ldots, a_m)$ trong đó $a_1 \le a_2 \le \ldots \le a_m$ và $B = (b_1, b_2, \ldots, b_n)$ trong đó $b_1 \le b_2 \le \ldots \le b_n$. Dãy hoà nhập của hai dãy không giảm A và B là dãy không giảm $C = (c_1, c_2, \ldots, c_{m+n})$, trong đó mỗi phần tử của dãy A hoặc dãy B xuất hiện trong dãy C đúng một lần.

   Hãy tìm phần tử thứ k của dãy C. Chẳng hạn, nếu A = (1, 3, 5, 9) và B = (3, 6, 8) thì dãy C = (1, 3, 5, 6, 8, 9).

Thiết kế thuật toán bằng kỹ thuật quy hoạch động cho các bài toán sau:

4. Bài toán tìm dãy con không giảm dài nhất đã nói trong bài toán 2.
5. (Đổi tiền). Cho một tập A các loại tiền $A = \{a_1, a_2, \ldots, a_n\}$, trong đó mỗi a_i là mệnh giá của một loại tiền, a_i là số nguyên dương. Vấn đề đổi tiền được xác định như sau. Cho một số nguyên dương c (số tiền cần đổi), hãy tìm số ít nhất các tờ tiền với các mệnh giá trong A sao cho tổng của chứng bằng c. Giả thiết rằng, mỗi loại tiền có đủ nhiều, và có một loại tiền có mệnh giá là 1.
6. Cho u và v là hai xâu ký tự bất kỳ. Ta muốn biến đổi xâu u thành xâu v bằng cách sử dụng các phép toán sau:
   - Xoá một ký tự.
   - Thêm một ký tự.
   - Thay đổi một ký tự.

Chẳng hạn, ta có thể biến đổi xâu abbac thành xâu abcbc bằng 3 phép toán như sau:

abbac → abac (xoá b)

→ ababc (thêm b)

→ abcbc (thay a bằng c)

Có thể thấy rằng, cách trên không tối ưu, vì chỉ cần 2 phép toán. Vấn đề đặt ra là: hãy tìm số ít nhất các phép toán cần thực hiện để biến xâu u thành xâu v, và cho biết đó là các phép toán nào.

7. Cho n đối tượng, ta muốn sắp xếp n đối tượng đó theo thứ tự được xác định bởi các quan hệ " < " và " = ". Chẳng hạn, với 3 đối tượng A, B, C chúng ta có 13 cách sắp xếp như sau:

   A = B = C, A = B < C, A < B = C, A < B < C, A < C < B, A = C < B, B < A = C, B < A < C, B < C < A, B = C < A, C < A = B, C < A < B, C < B < A.

   Hãy tính số cách sắp xếp n đối tượng.

Trong các bài toán sau, hãy đưa ra thuật toán được thiết kế bằng kỹ thuật quy lui:

8. Mê lộ là một lưới ô vuông gồm n dòng và n cột, các dòng và các cột được đánh số từ 0 đến n-1. Một ô vuông có thể bị cấm đi vào hoặc không. Từ một ô vuông có thể đi đến ô vuông kề nó theo dòng hoặc theo cột, nếu ô đó không bị cấm đi vào. Cần tìm đường đi từ ô vuông ở góc trên bên trái tới ô ở góc dưới bên phải.
9. Cho số tự nhiên n, hãy cho biết tất cả các dãy số tự nhiên tăng, có tổng bằng n. Chẳng hạn, với n = 6, ta có các dãy sau:

   1 , 2 , 3

   1 , 5

   2 , 4

   6

10. (Bài toán cặp đôi). Cho n đối tượng được đánh số 0, 1, …, n-1. Cho P[i] [j] là số đo sự ưa thích của đối tượng i với đối tượng j, P[i] [j] là số không âm. Trọng số của cặp đôi (i, j) là tích P[i] [j] * P[j] [i]. Chúng ta cần tìm một cách cặp đôi sao cho mỗi đối tượng phải được cặp đôi với một đối tượng khác (giả sử n chẵn) và hai đối tượng khác nhau cần phải cặp đôi với hai đối tượng khác nhau, và sao cho tổng các trọng số cặp đôi là lớn nhất.
11. Cho bàn cờ n x n và một vị trí xuất phát bất kỳ trên bàn cờ. Tìm đường đi của con mã từ vị trí xuất phát sao cho nó thăm tất cả các vị trí của bàn cờ đúng một lần.

Thiết kế thuật toán giải các bài toán sau đây bằng **kỹ thuật tham ăn**:

12. Quay lại bài toán đổi tiền trong bài tập 5. Hãy đưa ra một thuật toán khác dựa vào ý tưởng sau. Tại mỗi bước, với số tiền còn lại ta sử dụng loại tiền có mệnh giá lớn nhất trong các loại tiền còn lại, và sử dụng số tờ tối đa với mệnh giá đó. Hãy chỉ ra rằng, thuật toán có thể không cho ra cách đổi với số tờ tiền là ít nhất.
13. Cho đồ thị vô hướng G = (V, E), trong đó V là tập đỉnh, còn E là tập cạnh. Một tập U các đỉnh được gọi là một phủ, nếu cạnh (u, v) $\in$ E thì hoặc đỉnh u hoặc đỉnh v phải thuộc U. Phủ có số đỉnh ít nhất được gọi là phủ nhỏ nhất. Có thể xây dựng tập U dần từng bước xuất phát từ U rỗng, tại mỗi bước ta thêm vào U một đỉnh v là đỉnh có bậc lớn nhất trong các đỉnh không có trong U. Hãy viết ra thuật toán dựa theo ý tưởng trên. Thuật toán có cho ra phủ nhỏ nhất không?
14. Hãy đưa ra một thuật toán ngẫu nhiên để tạo ra một mê lộ (xem định nghĩa mê lộ trong bài tập 8).