Programação Dinâmica

Wladimir Araújo Tavares

16 de abril de 2008

1 Introdução

Programação dinâmica é um método para a construção de algoritmos para a resolução de problemas de otimização. Ela é aplicável à problemas no qual a solução ótima pode ser computada a partir da solução ótima previamente calculada e memorizada - de forma a evitar recálculo - de outros subproblemas que, sobrepostos, compõem o problema original.

O que um problema de otimização deve ter para que a programação dinâmica seja aplicável são duas principais características: subestrutura ótima e sobreposição de subproblemas. Um problema apresenta uma subestrutura ótima quando uma solução ótima para o problema contém em seu interior soluções ótimas para subproblemas. A sobreposição de subproblemas acontece quando um algoritmo recursivo reexamina o mesmo problema muita vezes.

O exemplo clássico para o entedimento de programação dinâmica é o problema da sequência de fibonacci:

$$\text{f(n)} = \begin{cases} 1 & \text{se n=0 e n=1} \\ f(n-1) + f(n-2) & \text{se } n \geq 2 \end{cases}$$

Podemos fazer um programa que calcule um termo da sequência de fibonacci recursivamente da seguinte maneira:

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

Como podemos fazer um programa que calcule um termo da sequência de fibonacci de maneira iterativa *bottom-up*:

```
int fibo(int n) {
    memo[1] = memo[2] = 1;
    for (int i=3; i<=n; i++)
        memo[i] = memo[i-1] + memo[i-2];
```

```
    return memo[n];
}
```

A cada passo da iteração para calcular um termo da sequência de fibonacci, os termos do qual ele depende já foram previamente calculados. Utilizando a técnica de *memorização* podemos resover o problema rapidamente.

2 Problema da Programação de linha de montagem

Nesse primeiro exemplo temos uma fábrica com duas linhas de montagens. Um chassi de automóvel entra em cada linha de montagem, tem as peças adicionadas a ele em uma série de estações, e um automóvel pronto sai no final da linha. Cada linha de montagem possui n estações, numeradas com j = 1, 2, ..., n. Cada linha de montagem é numerada com i = 1 ou 2. Com isso temos $S_{i,j}$ como sendo a j-ésima estação na linha i. A j-ésima estação da linha 1 ($S_{1,j}$) executa a mesma função que a j-ésima estação na linha 2 ($S_{2,j}$). Porém, as estações foram construídas em épocas diferentes e com tecnologias diferentes, de forma que o tempo exigido em cada estação varia, até mesmo as estações na mesma posição nas duas linhas. Denotamos o tempo de montagem exigido na estação $S_{i,j}$ por $a_{i,j}$. Além disso, temos um tempo de entrada e_i, um tempo de saída x_i e um tempo para transferir um chassi de uma linha de montagem para outra depois da passagem pela estação $S_{i,j}$, que é $t_{i,j}$. É importante ressaltar que uma transferência pode ocorrer até quando a estação for n-1, pois após a n-ésima estação, a montagem se completa.

Etapa 1: Utilizar a subestrutura ótima do problema

Se utilizarmos a subestrutura ótima do problema podemos construir uma solução ótima para um problema a partir de soluções ótimas para subproblemas. No caso da programação de linha de montagem, o raciocínio é dado a seguir. Se examinarmos um caminho mais rápido através da estação $S_{1,j}$ ele tem de passar pela estação j-1 na linha 1 ou na linha 2. Desse modo, o caminho mais rápido pela estação $S_{1,j}$ é:

- O caminho mais rápido pela estação $S_{1,j-1}$ e depois diretamente pela estação $S_{1,j}$.
- O caminho mais rápido pela estação $S_{2,j-1}$, uma transferência da linha 2 para a linha 1, e depois através da estação $S_{1,j}$.

Usando um raciocínio simétrico podemos definir o caminho mais rápido através da estações $S_{2,j}$.

Etapa 2: Uma solução recursiva

Nessa etapa buscamos uma solução recursiva para o problema em questão. Seja $f_i[j]$ o menor tempo possível para levar um chassi desde o ponto de

partida desde o ponto de partida até a estação $S_{i,j}$. O nosso obejtivo final é descobrir o menor tempo para levar um chassi por todo o percuso na fábrica, que denotaremos por f^*.

$f^* = \min(\ f_1[n] + x_1\ ,\ f_2[n] + x_2)$.

Podemos obter equações recursivas para $f_1[j]$:

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{se j=1} \\ min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{se } j \geq 2 \end{cases}$$

Usando um raciocínio simétrico podemos obter as equações recursivas para $f_2[j]$.

Etapa 3: Transformando uma solução recursiva em programação dinâmica

Utilizaremos l_i[j] para indicar a linha de montagem que precedeu a estação j em $f_i[j]$.

```
MONTAGEM(a, t, e, x, n)
1 f1[1] <- e1 + a1,1
2 f2[1] <- e2 + a2,2
3 for j = 2 to n do
4     if f1[j − 1] + a1, j <= f2[j − 1] + t2, j − 1 + a1, j then
5         f1[j] = f1[j − 1] + a1, j
6         l1[j] = 1
7     else f1[j] = f2[j − 1] + t2,j−1 + a1,j
8         l2[j] = 2
9     if f2[j − 1] + a2,j <= f1[j − 1] + t1,j−1 + a2,j then
10        f2[j] =f2[j − 1] + a2,j
11        l2[j] = 2
12    else f2[j] = f1[j − 1] + t1, j − 1 + a2,j
13        l2[j] = 1
14 if f1[n] + x1 <= f2[n] + x2 then
15    f* = f1[n] + x1
16    l* = 1
17 else f∗ = f2[n] + x2
18    l* = 2
```

Etapa 4: Mostrando a solução ótima

Temos que utilizar um procedimento para mostrar a solução ótima. A chamada para o procedimento é PRINT(l,l^*,n).

```
PRINT(l,i,j)
      if i >= 2 then
        PRINT(li[j],j-1)
      imprimir "“linha"" , i , "“estação"" , j.
```

3 Subsequência Comum mais longa

Uma subsequência de uma determinada sequência é apenas a sequência dada com zero ou mais elementos omitidos. Formalmente, dada uma sequência X $= < x_1, \ldots, x_m >$, outra sequência Z $= < z_1, \ldots, z_k >$ é uma subsequência de X se existe uma sequência estritamente crescente $< i_1, \ldots, i_k >$ de índices de X tais que para todo j= 1,2,...,k temos que $x_{i_j} = z_j$. Por exemplo, Z=$< B, C, D, B >$ é uma subsequência de X=$< A, B, C, B, D, A, B >$ com sequência de índice correspondente $< 2, 3, 5, 7 >$.

Dada duas sequencias X e y, dizemos que uma sequencia Z é uma subsequencia comum de X e Y se Z é subsequencia de X e Y ao mesmo tempo. Por exemplo, se X=$< A, B, C, B, D, A, B >$ E Y=$< B, D, C, A, B, A >$, a sequência $< B, C, A >$ é uma subsequência comum das sequencias X e Y e $< B, C, B, A >$ é a subsequencia comum mais longa.

No problema da subsequencia comum mais longa, temos duas sequencias X $= < x_1, \ldots, x_m >$ e Y $= < y_1, \ldots, y_n >$, e desejamos encontrar uma subsequencia comum de comprimento máximo(longest comum subsequence-LCS) de X e Y.

Etapa 1: Caracterizar um subsequencia comum mais longa

Para encontrar uma LCS de X $= < x_1, \ldots, x_m >$ e Y $= < y_1, \ldots, y_n >$ temos que analisar dois subproblemas:

- se $x_m = y_n$, devemos encontrar uma LCS de X_{m-1} e Y_{n-1}. A anexação de $x_m = y_n$ a essa LCS produz uma LCS de X e Y.
- se $x_m \neq y_n$, então temos que resolver dois subproblemas: encontrar uma LCS de X_{m-1} e Y e encontrar uma LCS de X e Y_{n-1}. A maior entre as duas LCSs é uma LCS de X e Y.

Podemos ver que muitos subproblemas compartilham subproblemas. A superposição de problemas pode ser vista de imediato.

Etapa 2: Definição recursiva

Vamos definir c[i,j] como o comprimento de uma LCS das sequências X_i e Y_i. Se i=0 ou j=0, uma das sequencias tem o comprimento 0; assim, a LCS tem comprimento 0. A subestrutura ótima do problema da LCS fornece a fórmula recursiva

$$\text{c[i,j]} = \begin{cases} 0 & \text{se i=0 ou j=0} \\ c[i-1, j-1] + 1 & \text{se i,j} > 0 \text{ e} x_i = y_j \\ max(c[i, j-1], c[i-1, j]) & \text{se i,j} > 0 \text{ e} x_i \neq y_j \end{cases}$$

Etapa 3: Transformando a recursão em programação dinâmica

A matriz b[i,j] indica de onde veio a solução ótima para c[i,j].

```
char X[MAX],Y[MAX];
int i,j,m,n,c[MAX][MAX],b[MAX][MAX];
int LCSlength() {
    m=strlen(X);
    n=strlen(Y);
    for (i=1;i<=m;i++) c[i][0]=0;
```

```
    for (j=0;j<=n;j++) c[0][j]=0;

     for (i=1;i<=m;i++)
         for (j=1;j<=n;j++) {
             if (X[i-1]==Y[j-1]) {
                c[i][j]=c[i-1][j-1]+1;
                b[i][j]=1;
             }
             else if (c[i-1][j]>=c[i][j-1]) {
                c[i][j]=c[i-1][j];
                b[i][j]=2;
             }
             else {
                c[i][j]=c[i][j-1];
                b[i][j]=3;
             }
         }
    return c[m][n];
}
```

Etapa 4: Imprimindo a solução ótima

```
void printLCS(int i,int j) {
      if (i==0 || j==0) return;
      if (b[i][j]==1) {
         printLCS(i-1,j-1);
         printf("%c",X[i-1]);
      } else if (b[i][j]==2)
         printLCS(i-1,j);
      else
         printLCS(i,j-1);
}
```