Programare dinamică

Marius Minea

10 martie 2010

Cum să ne planificăm examenele

după Michael A. Trick, Tutorial on Dynamic Programming, CMU, 1997

Zile	Franceză	Engleză	Statistică
0	0.8	0.75	0.9
1	0.7	0.7	0.7
2	0.65	0.67	0.6
3	0.62	0.65	0.55
4	0.6	0.62	0.5

Se dă probabilitatea de a pica o materie dupa k zile de studiu.

Cum trebuie învățat, având în total 4 zile,
minimizând probabilitatea de a pica toate 3 examenele?

Descompunerea problemei

Notăm: $p_i(t) =$ probabilitatea de a pica materia i învățând t zile.

Căutăm $\min p_1(t_1) * p_2(t_2) * p_3(t_3)$ unde $t_1 + t_2 + t_3 = 4$

Descompunerea problemei

Notăm: $p_i(t)$ = probabilitatea de a pica materia i învățând t zile.

Căutăm $\min p_1(t_1) * p_2(t_2) * p_3(t_3)$ unde $t_1 + t_2 + t_3 = 4$

Reformulăm problema: presupunem că ne aflăm în etapa i,
și mai avem t ore disponibile pentru materiile $i..n$.
Care e probabilitatea $f(i, t)$ de a le pica pe cele rămase ?

Cazul de bază: $i = n$ (a rămas materia n). Trivial, $f(n, t) = p_n(t)$

Soluția recurentă:

$$f(i, t) = \min_{k \leq t} p_i(k) \cdot f_{i+1}(t - k) \qquad (1 \leq i < n)$$

Problema dată: Află $f(1, t)$ (n = 3, t = 4).

Rezolvarea: completarea unui tabel

```
double solve(int n, int tot, double p[n+1][tot+1])
{
  double f[n+1][tot+1];
  for (int t = 0; t <= tot; ++t) f[n][t] = p[n][t];
  for (int i = 1; i < n; ++i)   // etapele, de la coadă
    for (t = 0; t <= tot; ++t) {
      double min = 1;   // prob. e sigur <= 1
      for (k = 0; k <= t; ++k)
        if (p[i][k]*f[i+1][t-k] < min)
          min = p[i][k]*f[i+1][t-k];
    }
  return f[1][tot];
}
```

Dacă vrem să și determinăm *combinația optimă*, declarăm `int opt[n][tot+1];` și completăm `opt[i][t] = k;` în ciclu

Problemă: cea mai lungă subsecvență comună a două șiruri

O *subsecvență* a șirului $x_0 x_1 \ldots x_{m-1}$ e o secvență de elemente *nu neapărat consecutive* cu indici crescători $z_j = x_{i_j}$, $0 \leq j < k$.

Cea mai lungă subsecvență comună pt. $x_0 x_1 \ldots x_{m-1}$ și $y_0 y_1 \ldots y_{n-1}$:

– pt. $x_{m-1} = y_{n-1}$, adăugăm la secvența comună a capetelor rămase
– altfel: maximul după eliminarea ultimului element din câte un șir

```
int l[M][N], d[M][N]; // !=0: care şir e scurtat; ==0: ambele
for (i = 0; i < M; ++i)
  for (j = 0; j < N; ++j)
    if (x[i] == y[j]) { l[i][j] = 1+l[i-1,j-1]; d[i][j] = 0; }
    else if (l[i-1][j]>l[i][j-1])
      { l[i][j]=l[i-1][j]; d[i][j]=1; }
    else { l[i][j] = l[i][j-1]; d[i][j] = 2; }
```

Cum recunoaștem o problemă de programare dinamică

Soluția problemei e structurată în *etape* (aici: materii),
cu o *decizie* la fiecare etapă (cât învățăm la o materie)

Cum recunoaștem o problemă de programare dinamică

Soluția problemei e structurată în *etape* (aici: materii),
cu o *decizie* la fiecare etapă (cât învățăm la o materie)

În fiecare etapă sunt posibile mai multe *stări* (ex. timp disponibil)

Cum recunoaștem o problemă de programare dinamică

Soluția problemei e structurată în *etape* (aici: materii),
cu o *decizie* la fiecare etapă (cât învățăm la o materie)

În fiecare etapă sunt posibile mai multe *stări* (ex. timp disponibil)

O *decizie* duce dintr-o stare în altă stare în etapa următoare

Cum recunoaștem o problemă de programare dinamică

Soluția problemei e structurată în *etape* (aici: materii),
cu o *decizie* la fiecare etapă (cât învățăm la o materie)

În fiecare etapă sunt posibile mai multe *stări* (ex. timp disponibil)

O *decizie* duce dintr-o stare în altă stare în etapa următoare

Decizia optimă într-o stare *nu depinde* de stările/deciziile *anterioare*
(aici: doar de bugetul de timp rămas)

Cum recunoaștem o problemă de programare dinamică

Soluția problemei e structurată în *etape* (aici: materii),
cu o *decizie* la fiecare etapă (cât învățăm la o materie)

În fiecare etapă sunt posibile mai multe *stări* (ex. timp disponibil)

O *decizie* duce dintr-o stare în altă stare în etapa următoare

Decizia optimă într-o stare *nu depinde* de stările/deciziile *anterioare*
(aici: doar de bugetul de timp rămas)

Există o *relație de recurență* pentru deciziile în etape consecutive
(inclusiv un caz de bază, cu soluție directă)

după M. A. Trick, CMU

Despre programarea dinamică (cont.)

Termenul vine de la *înlănțuirea* unor decizii optime una după alta
[Bellmann, anii '40-'50]
mai precis: decizii descompuse în decizii pentru subprobleme mai mici

Folosită la probleme care pot fi *descompuse în subprobleme*, dar:
– divide and conquer: subproblemele generate sunt disjuncte;
la fiecare pas, alegem o singură descompunere
– programare dinamică: la fiecare pas, > 1 descompuneri posibile

Domeniu de aplicație: în special *probleme de optimizare*
– cu proprietatea de descompunere optimală *optimal substructure*
(soluția optimă a problemei conține soluții optime la subprobleme)

– de regulă un tablou pentru cost + unul pentru a reține soluțiile
– consumul de memorie: adesea semnificativ (pătratic sau mai mult)

Comparații cu alte tehnici de soluție

Metoda Greedy

– pentru probleme de optimizare

– alegerea optimului local în scopul de a obține un optim global

(nu e valabilă universal; uneori, e doaer o euristică)

Exemple: arborele minim de cuprindere într-un graf

Căutarea cu revenire

– când nu se poate determina sigur pasul care conduce la succes

– e necesară o căutare exhaustivă în spațiul stărilor

– căutare recursivă (în adâncime), cu revenire în caz de eșec

Tehnica divizării

– rezolvarea unei probleme prin descompunerea în probleme mai mici

– tipic: structură recursivă (exemplu: quicksort)

– de regulă: alegem *o singură* împărțire

Exemplu: înmulțirea a N matrici

Fie matricile $A_{m,n}$ (m linii, n coloane) și $B_{n,p}$ (n linii, p coloane).
Pentru a calcula produsul avem nevoie de $m \cdot n \cdot p$ înmulțiri

```
for (i = 0; i < m; ++i)
  for (k = 0; k < p; ++k) {
    c[i][k] = 0;
    for (j = 0; j < n; ++j)
      c[i][k] += a[i][j]*b[j][k];
  }
```

Înmulțirea matricilor e asociativă $\Rightarrow$ calcul în mai multe feluri
ex. cu $A_{10,100}$, $B_{100,5}$, $C_{5,50}$:
$(AB)C$: $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 5000 + 2500 = 7500$ înmulțiri
$A(BC)$: $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 25000 + 50000 = 75000$ înmulțiri

Problemă: Care e ordinea calculelor cu număr minim de înmulțiri ?

Înmulțirea a N matrici: gruparea cu paranteze

Subproblemă: Care e numărul $P(n)$ de grupări posibile de paranteze ?

Soluție recurentă: pe ce poziție $1 \leq k \leq n-1$ e ultima înmulțire?

$$P(1) = 1;\ P(n) = \sum_{j=1}^{n-1} P(j) \cdot P(n-j)$$

Soluția: *numerele lui Catalan*, $P(n) = C(n-1)$, $C(n) = C_{2n}^{n}/(n+1)$

(exponențial în n, dar la n relativ mic merită calculat optimul)

Înmulțirea a N matrici: gruparea cu paranteze

Subproblemă: Care e numărul $P(n)$ de grupări posibile de paranteze ?
Soluție recurentă: pe ce poziție $1 \le k \le n-1$ e ultima înmulțire?

$$P(1) = 1;\ P(n) = \sum_{j=1}^{n-1} P(j) \cdot P(n-j)$$

Soluția: *numerele lui Catalan*, $P(n) = C(n-1)$, $C(n) = C_{2n}^{n}/(n+1)$
(exponențial în n, dar la n relativ mic merită calculat optimul)

$\Rightarrow$ nr. minim de înmulțiri în $A_i \cdot \ldots \cdot A_k$ e (cu separare înainte de A_j):
$m_{ii} = 0$, $m_{ik} = \min_{i<j\le k} m_{i,j-1} + m_{j,k} + d_i d_j d_k$ (A_i e de $d_i \times d_{i+1}$)

$\Rightarrow$ calculăm completând tabloul global `double m[N][N];`
și tabloul `int p[N][N];` cu poziția la care se face ultima înmulțire

```
for (c = 1; c < n; ++c)          // subşir de c+1 matrici
  for (i = 0, k = c; k < n; ++i, ++k) // mat. de la i la k
    m[i][k] = DBL_MAX;                      // sigur maxim
    for (j = k; j > i; ) {          // j = poz. intermediară
      double v = d[i]*d[j]*d[k]+m[j][k]; v+=m[i][--j];
      if (v < m[i][k]) { m[i][k] = v; p[i][k] = j; }
    }
  }
```

Programare dinamică, recursivitate, memoizare

Una din sarcinile de rezolvat: în ce ordine se rezolvă subproblemele (se completează elementele tabelului) ?

Adesea: ordine naturală (ex. indicii cresc) $\Rightarrow$ program ușor de scris
Dar, nu întotdeauna e evident $\Rightarrow$ abordarea recursivă e mai naturală

Dar: în programarea dinamică apar *subprobleme comune*
Recursivitatea e ineficientă (exponențială), recalculează inutil valorile
(la fel ca și calculul naiv pentru Fibonacci)

Soluție: tehnica de *memoizare*

Soluție cu structură recursivă, dar cu *memorarea valorilor calculate*
$\Rightarrow$ dacă valoarea a fost calculată, o returnează;
dacă nu, o calculează recursiv și o memorează înainte de a o returna

– Calculul recursiv cu memoizare: mai natural de scris; mai eficient dacă pentru rezultatul cerut nu trebuie rezolvate *toate* subproblemele
– Calculul de jos în sus: cod mai eficient (fără apeluri de funcții), dar rezolvă exhaustiv toate subproblemele (chiar nenecesare)

Exemplu: longest common subsequence pt. $x_1 \dots x_m$ și $y_1 \dots y_n$:
dacă $x_m = y_n$ trebuie calculat $len_{m-1,n-1}$ (și nu $len_{m,n-1}$, $len_{m-1,n}$
dacă $x_m \neq y_n$ NU trebuie calculat $len_{m-1,n-1}$

Arbore binar de căutare optim

O mulțime de chei trebuie aranjate într-un arbore binar de căutare
(cheile din subarborele stâng $<$ cheia din nod $<$ cheile din s. drept)
Fiecare cheie are o *probabilitate* de apariție. (tablou `double p[N];`)
Care e arborele binar cu nr. minim (probabilistic) de pași de căutare ?

Fie c_{ik} costul minim pentru arborele cu cheile $i..k$, și r_{ik} rădăcina sa:
$c_{ik} = \sum_{j=i}^{k} p_j + \min_{i \le j \le k}(c_{i,j-1} + c_{j+1,k})$, $c_{ii} = p_i$, $c_{i,i-1} = 0$ (arb. $\emptyset$)
($\sum_{j=i}^{k} p_j$ pt. că fiecare nod e cu 1 mai jos decât în subprobleme)

Soluție: în ordine crescătoare a diferenței $k - i$ (nr. de noduri)
În paralel, completăm tabloul r_{ik} pentru rădăcina arborelui

Problema rucsacului pentru întregi

Se dau N (tipuri de) obiecte cu dimensiuni s_i și valori v_i întregi.
Care e valoarea maximă a unor obiecte de dimensiune totală dată D ?

Problema rucsacului are multe variante !
Dacă se permit fragmente de obiecte, problema are soluție *greedy*.
Pentru dimensiuni reale, problema e NP-completă (exponențială).

Varianta 1: număr nelimitat de obiecte de fiecare tip

```
for (d = 1; d <= D; ++d)          // dimensiuni crescătoare
  for (c[d] = i = 0; i < n; ++i) // încearcă fiecare obiect i
    if (s[i] < d && (m = c[d-s[i]] + v[i]) > c[i]) c[i] = m;
```

Varianta 2: obiecte unice; `c[d][i]`: cost max. pt. dim. d, obiecte 0..i

```
for (d = 1; d <= D; ++d)          // dimensiuni crescătoare
  for (i = 0; i < n; ++i)         // încearcă includerea obiectulu
    if (s[i] < d) c[d][i] = max(c[d][i-1], c[d-s[i]]+v[i])
```

Problemă: distanța minimă de editare între două șiruri

Se dau două șiruri de lungimi m și n. Să se determine costul minim al operațiilor de editare necesare pentru a transforma un șir în celălalt, prin adăugarea / ștergerea / schimbarea unui caracter.
(Obs: schimbarea are sens dacă costul $<$ adăugare $+$ ștergere)

– dacă ultimele caractere sunt egale, transformăm restul șirurilor;
– altfel, minimul celor 3 modificări posibile ale ultimelor caractere

```
int c[M][N]; // costul pt. primele i din x -> primele j din y
for (i = 0; i < M; ++i)
  for (j = 0; j < N; ++j)
    if (x[i] == y[j]) c[i][j] = c[i-1][j-1];
    else c[i][j] = min3(c[i-1][j]+C_DELETE,
                        c[i][j-1]+C_INSERT,
                        c[i-1][j-1]+C_CHANGE);
```

Problemă: triangularea optimă a unui poligon

Să se împartă un poligon convex în triunghiuri, prin coarde de lungime totală minimă.
Mai general: cu minimizarea unei funcții de cost $w(\Delta)$

Fie vârfurile consecutive $v_i, \ldots, v_k$ cu $0 \leq i < k < n$
v_j $(i < j < k)$ vârful care formează triunghi cu latura $v_i v_k$,
și c_{ik} costul de triangulare.
$\Rightarrow c_{ik} = \min_{i<j<k}(w(i,j,k) + c_{ij} + c_{jk})$, $c_{i,i+1} = 0$. Vrem $c_{0,n-1}$.

```
double c[N][N]; unsigned v[N][N];
for (l = 2; l < N; ++l)          // l == k - i
  for (k = N - 1, i = k - l; i >= 0; --k, --i) {
    c[i][k] = DBL_MAX;
    for (j = k; --j > i; ) {
      double m = w(i,j,k) + c[i][j] + c[j][k];
      if (m < c[i][k]) { c[i][k] = m; v[i][k] = j; }
    }                  // v[i][j] ţine minte soluţia
  }
```