

Recursivitate. Programare dinamică

10 ianuarie 2006

Recursivitate: calculul expresiilor

Expresiile pot fi definite recursiv, cu diagrame de sintaxă sau *gramatici*

$expresie ::= termen \mid expresie + expresie \mid expresie - expresie$

$termen ::= identificator \mid numar$

– simboluri *neterminale*: definite în funcție de altele, prin *reguli*

– simboluri *terminale*: ex. +, numar

Dorim:

– să recunoaștem în intrare o secvență care corespunde tiparului dat de gramatică

– să calculăm valoarea expresiei

Idee de prelucrare:

– câte o funcție (cu recursivitate (in)directă) pentru fiecare neterminal

– consumă din intrare terminale, sau apelează funcții pt. neterminale după structura gramaticii, și returnează o valoare (pt. expresie)

Structura recursivă a expresiilor

Problemă: gramatica dată e *ambiguă*: fie $5 - 4 + 2$.

Considerăm ca $expr + expr$ $5-4 + 2$ sau $expr - expr$ $5-4 + 2$?

⇒ rescriem gramatica pentru a explicita asociativitatea la stânga:

$expresie ::= termen \mid expresie + termen \mid expresie - termen$

– fiecare expresie admite acum o unică *derivare* din gramatică

Problemă: în prelucrarea expresiei, dorim să știm citind un singur simbol din intrare pe care din ramuri (termen, +, -) să o alegem.

Dar orice dezvoltare de expresie începe până la urmă tot cu un termen!

⇒ rescriem gramatica, factorizând și eliminând recursivitatea la stânga

$expresie ::= termen \text{ rest-expr}$

$\text{rest-expr} ::= \epsilon \mid + \text{ termen rest-expr} \mid - \text{ termen rest-expr}$

Expresii prefix si postfix

Operatorii sunt de fapt un caz particular de funcții, cu scriere infix.

⇒ putem rescrie $5 - 4 + 2$ ca fiind $plus(minus(5, 4), 2)$

In scrierea funcțiilor, parantezele si virgulele sunt doar o convenție; nu sunt necesare dacă știm câte argumente are fiecare funcție

⇒ obținem scrierea *prefix* a unei expresii: $+ - 5 4 2$

Obținem formatul *postfix* scriind fiecare funcție (operator) *după* argumentele sale: $5 4 - 2 +$

Metode de proiectare a algoritmilor

Metoda Greedy

- pentru probleme de optimizare
 - alegerea optimului local în scopul de a obține un optim global (nu e valabilă universal; uneori, e doar 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)

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 pentru N matrici $A_0 \cdot A_1 \cdot \dots \cdot A_{N-1}$ produsele individuale pot fi calculate în diverse ordini

ex. pt. $A_{10,100}$, $B_{100,5}$, $C_{5,50}$:

$(AB)C$: $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 5000 + 2500 = 7500$ înmulțiri

$A(BC)$: $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 25000 + 50000 = 75000$ înmulțiri

Problemă: Care e ordinea calculelor pt. număr minim de înmulțiri ?

Înmulțirea a N matrici: gruparea cu paranteze

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

Soluție: relație recursivă: 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 merită calculat optimul, n fiind relativ mic)

\Rightarrow nr. minim de înmulțiri în $A_i \cdot \dots \cdot A_k$ e (cu separare înainte de A_j):

$m_{ii} = 0$, $m_{ik} = \min_{i < j \leq k} m_{i,j-1} + m_{j,k} + d_i d_j d_k$ (A_i e matrice $d_i \times d_{i+1}$)

\Rightarrow calculul se poate face 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) /* succesiv pt. subșir de c+1 matrici */
  for (i = 0, k = c; k < n; ++i, ++k) /* matricile de la i la k */
    m[i][k] = DBL_MAX; /* valoare maximă, se modifică sigur */
    for (j = k; j > i; ) { /* pt. toate pozițiile intermediare */
      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; }
    }
  }
```

Programarea dinamică - generalități

Denumirea: “programare” se referă la găsirea formulei și la procedura de calcul prin completarea unui tablou

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)

Programare dinamică și recursivitate

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

Adesea: ordine naturală (ex. de la indici mici) \Rightarrow program ușor de scris
Dar, nu întotdeauna e evident \Rightarrow abordarea recursivă e mai naturală

Calculul combinărilor: $C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$ pt. $0 < k < n$, $C_n^0 = C_n^n = 1$.

```
int comb(int n, int k)
{ return k==0 || k==n ? 1 : comb(n,k-1)+comb(n-1,k-1); }
```

soluție recursivă ineficientă (exponențială), recalculează inutil valorile față de soluția care completează pe rând tabelul pt. n crescător

```
for (n = 0; n <= N; ++n) {
    C[n][n] = C[n][0] = 1;
    for (k = 1; k < n; ++k) C[n][k] = C[n-1][k-1] + C[n-1][k];
} /* ar trebui păstrate de fapt doar 2 rânduri consecutive */
```

Tehnica de *memoizare*

```
#define N    10
int C[N+1][N+1]; /* implicit zero */
int comb(int n, int k)
{
    return C[n][k] ? C[n][k] :
        (C[n][k] = k==0 || k==n ? 1 : comb(n,k-1)+comb(n-1,k-1));
}
```

Soluție cu structură recursivă, dar cu memorarea valorilor calculate

⇒ 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)

Problemă: 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:

$$c_{ik} = \sum_{j=i}^k p_j + \min_{i \leq j \leq k} (c_{i,j-1} + c_{j+1,k}), \quad c_{ii} = p_i, \quad c_{i,i-1} = 0 \text{ (arb. vid)}$$
 (suma $\sum_{j=i}^k p_j$ pt. că fiecare nod e cu 1 mai jos decât în subprobleme)

Completarea tabloului: în ordine crescătoare a diferenței $k-i$ (numărul de noduri), în paralel cu 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*.

Dacă dimensiunile sunt reale, problema e NP-completă (exponențială).

Varianta 1: număr nelimitat de obiecte de fiecare tip

```
for (d = 1; d <= D; ++d) /* succesiv pt. 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) /* succesiv pt. dimensiuni crescătoare */
  for (i = 0; i < n; ++i) /* încearcă includerea obiectului i */
    c[d][i] = s[i]<=d ? max(c[d][i-1], c[d-s[i]][i-1]+v[i]) : c[d][i-1];
```

Problemă: cea mai lungă subsecvență comună a două șiruri

Fiind dat un șir $x_1x_2\dots x_m$, o subsecvență e un șir $z_1z_2\dots z_k$ pt. care există șirul de indici $i_1 < \dots < i_k$ a.î. $z_j = x_{i_j}$, $0 \leq j < k$. Găsiți cea mai lungă subsecvență comună a șirurilor $x_1x_2\dots x_m$ și $y_1y_2\dots y_n$.

- dacă $x_m = y_n$, se adaugă la secvența comună a capetelor rămase
- altfel: maximul după eliminarea ultimului element din câte un șir

```
int l[M+1][N+1], d[M+1][N+1]; // != 0: șirul scurtat; == 0: ambele
for (i = 0; i <= M; ++i) l[i][0] = 0;
for (j = 1; j <= N; ++j) l[0][j] = 0;
for (i = 1; i <= M; ++i)
    for (j = 1; 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; }
```

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+1][N+1]; // costul pt. primele i din x -> primele j din y
for (i = 0; i <= M; ++i) c[i][0] = i * C_DELETE;
for (j = 1; j <= N; ++j) c[0][j] = j * C_INSERT;
for (i = 1; i <= M; ++i)
    for (j = 1; 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, \dots, v_k cu $0 \leq i < k < n$ și v_j ($i < j < k$) vârful care formează triunghi cu latura $v_i v_k$, și c_{ik} costul de triangulare. Atunci: $c_{ik} = \min_{i < j < k} (w(i, j, k) + c_{ij} + c_{jk})$, $c_{i, i+1} = 0$. Căutăm $c_{0, n-1}$.

```
double c[N][N]; unsigned v[N][N];
for (l = 2; l < N; ++l)          /* l == k - i */
  for (k = N - 1, i = k - 1; 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] ține minte soluția */
  }
```