Laborator 11. Programare dinamică

1 Obiective

Studierea metodei de programare dinamică, o altă metodă de dezvoltare a algoritmilor, precum şi un exemplu de aplicare a metodei.

2 Noţiuni teoretice

2.1 Programare dinamică

Programarea dinamică, la fel cu metoda divide et impera, rezolvă probleme prin descompunerea lor în subprobleme de acelaşi tip, rezolvarea subproblemelor şi combinarea soluţiilor subproblemelor pentru a obţine soluţia problemei iniţiale. Pentru metoda *divide et impera*, e necesar ca subproblemele să fie disjuncte. La *programarea dinamică*, subproblemele se suprapun parţial, adică sub-subprobleme ale lor pot să coincidă. În astfel de cazuri, dacă am aplica un algoritm divide et impera, acesta ar face mai multe prelucrări decât e necesar, rezolvând în mod repetat sub-subproblemele comune. Un exemplu extrem de simplu ar fi calculul elementelor şirului lui Fibonacci: $F_n = F_{n-1} + F_{n-2}$, $n > 1$, $F_0 = F_1 = 1$, unde dacă descompunem problema iniţială (calculul lui F_n) în cele două subprobleme, calculul lui F_{n-1} şi calculul lui F_{n-2}, divide et impera ar calcula de multe ori valorile pentru indici mai mici, deoarece sunt multe subprobleme comune. O soluţie în acest caz ar fi să calculăm valorile pornind de la indicii mici, să memorăm valori calculate anterior, iar acestea ne ajută să calculăm valoarea curentă.

Un algoritm de programare dinamică rezolvă fiecare sub-subproblemă o singură dată şi îi salvează răspunsul, astfel încât se evită recalcularea răspunsului de fiecare dată când e necesar.

Programarea dinamică se aplică de regulă pentru probleme de optimizare, adică probleme care pot avea mai multe soluţii posibile, şi se caută o soluţie optimă. Pentru a se aplica metoda, problema trebuie să îndeplinească două condiţii: să aibă substructură optimală şi subproblemele în care se descompune să se suprapună parţial. Substructură optimală înseamnă că o soluţie optimală a problemei e conţinută în soluţiile optimale ale subproblemelor.

Ca urmare, prin programare dinamică o soluţie optimă este găsită printr-o secvenţă de decizii care depind de decizii luate anterior şi care satisfac principiul optimalităţii. Acest principiu poate fi formulat astfel:

Fie $\langle s_0, s_1, \ldots, s_n \rangle$ o secvenţă de stări, unde s_0 este starea iniţială şi s_n este starea finală. Starea finală se atinge prin luarea unei secvenţe de decizii $d_1, d_2, \ldots, d_n$ (decizia d_i transformă starea s_{i-1} în starea s_i).

Dacă secvenţa de decizii $\langle d_i, d_{i+1}, \ldots, d_j \rangle$ care schimbă starea s_{i-1} în starea s_j (cu stările intermediare $s_i, s_{i+1}, \ldots, s_{j-1}$) este optimă, şi dacă pentru orice $i \leq k \leq j-1$, atât $\langle d_i, d_{i+1}, \ldots, d_k \rangle$ cât şi $\langle d_{k+1}, d_{k+2}, \ldots, d_j \rangle$ sunt secvenţe de decizii optime care transformă starea s_{i-1} în s_k, respectiv starea s_k în s_j, atunci este satisfăcut principiul optimalităţii.

Pentru a aplica această metodă putem proceda astfel:

- Verificăm că principiul optimalităţii este satisfăcut.
- Definim recursiv valoarea asociată soluţiei optime (Scriem relaţiile de recurenţă care rezultă din regulile ce guvernează schimbările de stare)
- Calculăm valoarea pentru soluţia optimă, de obicei în manieră bottom-up (rezolvăm relaţiile de recurenţă)
- Construim o soluţie optimă pe baza valorii calculate pentru soluţia optimă.

2.2 Exemplu rezolvat folosind programarea dinamică

Ordinea de înmulţire a n matrici

Considerăm înmulţirea unei secvenţe de matrici

$$R = A_1 \times A_2 \times \ldots \times A_n,$$

unde A_i cu $1 \leq i \leq n$ este de dimensiune $d_{i-1} \times d_i$. Matricea rezultată, R, va fi de dimensiune $d_0 \times d_n$. Se ştie că la înmulţirea a două matrici, A_i şi A_{i+1}, trebuie să se calculeze $d_{i-1} \times d_i \times d_{i+1}$ înmulţiri. Dacă matricile de înmulţit au număr diferit de linii/coloane, atunci numărul de operaţii necesar pentru a obţine R depinde de ordinea de efectuare (altfel spus, de asociere) a înmulţirilor celor n matrici. Vrem să găsim ordinea de efectuare a acestor înmulţiri astfel încât numărul de operaţii să fie minim.
De exemplu, pentru înmulţirea a 3 matrici A, B, C de dimensiuni 2×3, 3×5 şi 5×2, asocierea $(A \cdot B) \cdot C$ va avea de efectuat $2 \cdot 3 \cdot 5 + 2 \cdot 5 \cdot 2 = 50$ înmulţiri, iar $A \cdot (B \cdot C)$ va avea $3 \cdot 5 \cdot 2 + 2 \cdot 3 \cdot 2 = 42$ înmulţiri.

Fie C_{ij} numărul minim de înmulţiri pentru calcularea produsului $A_i \times A_{i+1} \times \ldots \times A_j$ pentru $1 \leq j \leq n$. Se observă că:

- $C_{i,i} = 0$, adică "înmulţirea" unei singure matrici are cost 0.
- $C_{i,i+1} = d_{i-1} \times d_i \times d_{i+1}$.
- $C_{1,n}$ va fi valoarea minimă.
- Principiul optimalităţii este respectat, adică
  $C_{i,j} = \min\{C_{i,k} + C_{k+1,j} + d_{i-1} \times d_k \times d_j$, pentru $i \leq k < j\ \}$, şi
  asocierile sunt $(A_i \times A_{i+1} \times \ldots \times A_k) \times (A_{k+1} \times A_{k+2} \times \ldots \times A_j)$.

Trebuie să calculăm $C_{i,i+d}$ pentru fiecare nivel d până când obţinem $C_{1,n}$. Pentru rezolvarea problemei, construim un arbore binar care descrie ordinea înmulţirilor. Pentru construirea arborelui, vom reţine şi valoarea lui k pentru care s-a obţinut minimul, ceea ce ne va permite să obţinem modul în care au fost asociate matricile. Nodurile arborelui vor conţine limitele subsecvenţei de

matrici pentru care se calculează produsul. Rădăcina va fi $(1, n)$, iar un subarbore cu rădăcina (i, j) va avea ca fii nodurile (i, k) şi $(k + 1, j)$, unde k este valoarea pentru care se obţine optimul.

O implementare posibilă în C pentru algoritmul de aflare a înmulţirii optime a n matrici este dată în continuare.
Observaţie. În matricea costurilor, C, elementele de deasupra diagonalei principale (inclusiv) sunt costurile calculate, C[1,n] fiind ultimul calculat şi reprezentând numărul minim de înmulţiri. Elementele de sub diagonala principală, C[j,i] (care nu sunt utilizate pentru calculul costurilor), sunt utilizate pentru memorarea valorii lui k pentru care se atinge minimul $C_{i,j} = C_{i,k} + C_{k+1,j} + d_{i-1} \times d_k \times d_j$.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define  MAXN 10

typedef struct _NodeT
{
  long ind1, ind2;
  struct _NodeT *left,*right;
}
NodeT;

void matMulOrder( long C[ MAXN ][ MAXN ], int D[ MAXN + 1 ], int n )
/* Determina ordinea optima de inmultire pentru o secventa de matrici
   A1 x A2 x ... x An de dimensiuni D0 x D1, D1 x D2, ..., Dn-1 x Dn */
{
  int i, j, k, l, pos;
  long min, q;

  /* costul inmultirii unei singure matrici e 0: */
  for( i = 1; i <= n; i++ ) C[ i ][ i ] = 0;

  for(l = 2; l <= n; l++ )  // l - lungimea unei secvente
    for( i = 1; i <= n - l + 1; i++ ) {
      j = i + l - 1;
      min = LONG_MAX;
      for( k = i; k <= j - 1; k++ ) {
        q = C[ i ][ k ] + C[ k + 1 ][ j ] + (long) D[ i - 1 ] * D[ k ] * D[ j ];
        if( q < min)  {
          min = q;
          pos = k;
        }
      }
      C[ i ][ j ] = min;
      C[ j ][ i ] = pos;
    }
```

```
}

NodeT *buildTree( NodeT *p, int i, int j, long C[ MAXN ][ MAXN ] ) {
  p = ( NodeT * )malloc( sizeof( NodeT ));
  p->ind1 = i;
  p->ind2 = j;
  if( i < j ) {
    p->left = buildTree( p->left, i, C[ j ][ i ], C ) ;
    p->right = buildTree( p->right, C[ j ][ i ] + 1, j, C );
  }
  else
    p->left = p->right= NULL;
  return p;
}

void postOrder( NodeT *p, int level ) {
  int i;
  if( p != NULL ) {
    postOrder( p->left, level + 1 );
    postOrder( p->right, level + 1 );
    for( i = 0; i <= level; i++) printf("  ");
      printf( "(%ld, %ld)\n", p->ind1, p->ind2 );
  }
}

int main(void) {
  int i, j, n;
  long C[ MAXN ][ MAXN ];
  int D[MAXN+1]; /* dimensiunile matricilor */
  NodeT *root = NULL;

  printf("\nIntroduceti nr. de matrici (maxim %d): ", MAXN-1);
  scanf( "%d", &n );
  while ( getchar() != '\n' );
  printf("\nDimensiunile matricilor:\n");
  for( i = 0; i <= n; i++ ) {
    printf("\nDati dimensiunea D[%d]=", i );
    scanf( "%d", &D[ i ] );
    while ( getchar() != '\n' );
  }
  matMulOrder( C, D, n );
  /* Matricea de cost calculata C */
  printf("\nMatricea de costuri C\n");
  for(i = 1; i <= n; i++ ) {
    for( j = 1; j <= n; j++)
      printf( "%6ld", C[ i ][ j ] );
    printf( "\n" );
  }
  printf("\nNumarul minim de inmultiri = %ld", C[ 1 ][ n ] );
  while ( getchar() != '\n' );
```

```
  root = buildTree( root, 1, n, C );
  printf("\nTraversarea arborelui in postordine\n");
  postOrder( root, 0 );
  while ( getchar() != '\n' );
  return 0;
}
```

3 Mersul lucrării

3.1 Probleme obligatorii

1. Pentru problema ordinii de înmulţire a matricilor, să se afişeze şi ordinea (asocierile) înmulţirilor matricilor astfel încât să se efectueze numărul minim de operaţii.

2. Să se rezolve problema de tăiere a unei tije astfel încât costul total al bucăţilor obţinute să fie maxim (problemă descrisă la curs).

3.2 Probleme opţionale

1. Se dă un graf orientat ponderat, în care ponderile pot fi şi negative, dar care să nu aibă cicluri cu cost negativ. Algoritmul lui Floyd-Warshall se bazează pe programarea dinamică pentru a găsi drumurile de cost minim între toate perechile de noduri. (https://en.wikipedia.org/wiki/Floyd-Warshall_algorithm#Algorithm). Să se implementeze acest algoritm.

2. Se dă o pereche de şiruri de litere $A = a_1 \dots a_m$ şi $B = b_1 \dots b_n$. Se cere să se convertească A în B cu un cost cât mai mic, respectând următoarele reguli:

   - ştergerea unei litere are costul 3.
   - inserarea unei litere pe orice poziţie are costul 4.
   - înlocuirea unei litere cu alta are costul 5.

   De exemplu, se poate converti $A =$ `abcabc` în $B =$ `abacab` astfel: `abcabc` poate fi convertit în `abaabc` cu costul 5, care poate fi convertit la `ababc` cu costul 3, care poate fi convertit la `abac` cu costul 3, care cu costul 4 poate fi convertit la `abacb`, care cu costul 4 poate fi convertit la `abacab`. Costul total al acestor transformări este deci 19. Acesta nu e neapărat cea mai puţin costisitoare conversie.
   Intrare: cele două şiruri pe linii consecutive.
   Ieşire: secvenţa de şiruri care transformă primul şir în al doilea cu un cost minim. Fiecare linie intermediară trebuie să conţină operaţia efectuată (`s`=şterge, `i`=inserează, `r`=înlocuieşte, şi şirul modificat. Ultima linie trebuie să conţină costul total.
   Exemplu Pentru exemplul anterior, intrarea va fi:

   ```
   abcabc
   abacab
   ```

   Ieşire (exemplu, nu e neapărat soluţia optimă):

```
r abaabc
s ababc
s abac
i abacb
i abacab
19
```

3. Se dă o secvenţă $R = \langle R_0, \ldots, R_n \rangle$ de numere pozitive şi un număr întreg k. Numărul R_i reprezintă numărul de utilizatori care cer o anumită informaţie la momentul i (de exemplu de la un server de web). Dacă serverul transmite această informaţie la un moment t, atunci sunt satisfăcute cererile tuturor utilizatorilor care cer informaţia strict înainte de momentul t. Serverul poate transmite această informaţie de cel mult k ori. Se cere să se aleagă cele k momente pentru a transmite, astfel încât să se minimizeze timpul total (pentru toate cererile) cât trebuie să aştepte toţi utilizatorii pentru a li se satisface cererile.
   Exemplu. Fie $R = 3, 4, 0, 5, 2, 7$ (deci $n = 6$) şi $k = 3$. O soluţie posibilă (nu neapărat optimă) ar fi să se transmită la momentele $1, 3$, şi 6. Cele 3 cereri de la momentul 0 vor aştepta o unitate de timp. Cele 4 cereri la momentul 1 vor aştepta 2 unităţi de timp. Cele 5 cereri la momentul 3 vor aştepta 3 unităţi de timp. Cele 2 cereri de la momentul 4 vor aştepta 2 unităţi de timp. Cele 7 cereri de la momentul 5 vor aştepta o unitate de timp. Astfel, timpul total de aşteptare pentru această soluţie ar fi $3 \times 1 + 4 \times 2 + 5 \times 3 + 2 \times 2 + 7 \times 1$.
   Intrare: n şi k pe prima linie, vectorul R pe linia a doua.
   Ieşire: Secvenţa celor k momente de timp.