1 Laborator 9: Metoda Greedy si metoda 'divide et impera'

1.1 Obiective

În lucrare sunt prezentate principiile metodelor Greedy şi 'divide et impera', variantele lor de aplicare şi exemple.

1.2 Notiuni teoretice

1.2.1 Metoda greedy

Metoda greedy se aplică la rezolvarea problemelor de optimizare. Soluţiile acestor probleme sunt fie submulţimi, fie elemente ale unor produse carteziene pentru care se atinge optimul (minimul sau maximul) funcţiei de scop.

Metoda greedy determină întotdeauna o singură soluţie a problemei. Ea construieşte soluţia treptat: iniţial soluţia este vidă, iar apoi se alege pe rând elementul cel mai promiţător corespunzător situaţiei de la momentul respectiv. Datorită acestui mod de alegere a elementelor metoda se numeşte greedy (lacom). Alegând mereu elementul cel mai promiţător în situaţia curentă (locală) se aleg elemente care asigură un optim local ceea ce nu garantează optimalitatea soluţiei globale astfel construite. Asadar metoda greedy nu determină întotdeauna soluţia optimă, iar optimalitatea rezolvarii folosind greedy trebuie demonstrată.

Practic, metoda greedy este potrivită în cazul când dintr-o mulţime A de n elemente se cere determinarea unei submulţimi B care să îndeplinească anumite condiţii pentru a fi acceptată.

Există două variante de rezolvare a unei probleme cu ajutorul metodei Greedy:

Greedy varianta 1

Se pleacă de la mulţimea B vidă. Se alege din mulţimea A un element neales în paşii precedenţi. Se cercetează dacă adăugarea la soluţia parţială B conduce la o soluţie posibilă. În caz afirmativ se adaugă elementul respectiv la B.

Descrierea variantei este următoarea:

```
#define MAXN ? /* suitable value */
void greedy1( int A[ MAXN ], int n, int B[ MAXN ], int *k )
/* A = set of candidate n elements
         B = set of k elements solution */
{
        int x, v, i;
        *k = 0; /* empty solution set */
        for ( i = 0; i < n; i++ )
        {
                select( A, B, i, x );
                /* select x, the first of A[ i ], A[ i + 1 ], ..., A[ n - 1 ],
                        and swap it with element at position i */
                v = checkIfSolution( B, x );
                /* v = 1 if by adding x we get a solution and v = 0 otherwise */
                if ( v == 1 )
                        add2Solution( B, x, *k );
                /* add x to B, specifying the number of elements in B */
        }
}
```

În varianta 1 a metodei, funcţia *select* stabileşte criteriul care duce la soluţia finală.

Greedy varianta 2

Se stabileşte de la început ordinea în care trebuie considerate elementele mulţimii A. Apoi se ia pe rând câte un element în ordinea stabilită şi se verifică dacă prin adăugare la soluţia parţială B anterior construită, se ajunge la o soluţie posibilă. În caz afirmativ se face adăugarea.

Descrierea variantei este următoarea:

```
#define MAXN ? /* suitable value */
void greedy2( int A[ MAXN ], int n, int B[ MAXN ], int *k )
/* A is the set of candidate n elements; B is the set of k elements solution */
{
        int x, v, i;
        *k = 0; /* empty solution set */
```

```
        process( A, n );  /* rearrange A */
        for ( i = 0; i < n; i++ )
        {
                x = A[ i ];
                checkIfSolution( B, x, v );
                /* v = 1 if by adding x we get a solution and v = 0 otherwise */
                if ( v == 1 )
                        add2Solution( B, x, *k );
                        /* add x to B, specifying the number of elements in B */
        }
}
```

1.2.2 Exemplu rezolvat cu greedy

Algoritmul lui Prim

Determinarea arborelui de acoperire de cost minim prin algoritmul lui Prim. Fie $G = (N, R)$ un graf neorientat conex. Fiecărei muchii $(i, j) \in R$ i se asociază un cost $c[i][j] > 0$. Problema constă în a determina un graf parţial conex $A = (N, T)$, astfel încât suma costurilor muchiilor din T să fie minimă. Se observă imediat că acest graf parţial este chiar arborele de acoperire. Algoritmul constă în următoarele:

- Iniţial se ia arborele ce conţine un singur vârf. S-a demonstrat că nu are importanţă cu care vârf se începe; ca urmare se ia vârful 1. Mulţimea arcelor este vidă.
- Se alege arcul de cost minim, care are un vârf în arborele deja construit, iar celălalt vârf nu aparţine arborelui. Se repetă în total acest pas de $n - 1$ ori.

Pentru evitarea parcurgerii tuturor arcelor grafului la fiecare pas, se ia vectorul v având n componente definit astfel:

$$U_i = \begin{cases} 0 & \text{if vertex } i \in T \\ k & \text{if vertex } i \notin T; \\ & k \in T \text{ is a node such that} \\ & (i, k) \text{ is a minimum cost edge} \end{cases}$$

Initial, $v[1] = 0$, si $v[2] = v[3] = \ldots = v[n] = 1$, adica initial arborele este $A = (\{1\}, \emptyset)$.

Implementarea algoritmului este data mai jos:

```
#include <stdio.h>
#define MAXN 10
#define INFTY 0x7fff

void Prim2( int n, int c[ MAXN ][ MAXN ], int e[ MAXN ][ 2 ], int *cost )
/* n = number of vertices;
   c = cost matrix;
   e = edges of the MST;
   c = cost of MST */
{
        int v[ MAXN ];
        /* v[ i ] = 0 if i is in the MST;
                 v[ i ] = j if i is not in the MST;
                 j is a node of the tree such that (i, j) is a minimum cost edge */
        int i, j, k, min;

        *cost = 0;
        v[ 1 ]=0;
        for ( i = 2; i <= n; i++ )
                v[ i ] = 1; /* tree is ({1},{}) */
        /* find the rest of edges */
        for ( i = 1; i <= n - 1; i++ )
        { /* find an edge to add to the tree */
                min = INFTY;
                for ( k  = 1; k <= n; k++ )
                        if ( v[ k ] != 0 )
                                if ( c[ k ][ v[ k] ] < min )
                                {
                                        j = k;
                                        min = c[ k ][ v[ k ]];
                                }
                e[ i ][ 0 ] = v[ j ];
                e[ i ][ 1 ] = j;
```

```
                *cost += c[ j ][ v[ j ]];
                /* update vector v */
                v[ j ]= 0;
                for ( k = 1; k <= n; k++ )
                        if ( v[ k ] != 0 &&
                                         c[ k ][ v[ k ]] > c[ k ][ j ])
                                v[ k ] = j;
        }
}
int main( int argc, char *argv[] )
{
        int n; /* number of nodes */
        int c[ MAXN ][ MAXN     ]; /* costs */
        int e[ MAXN ][ 2 ]; /* tree edges */
        int i, j, k, cost;

        printf( "\nNumber of nodes in graph G: " );
        scanf( "%d", &n );
  while ( '\n' != getchar() );
        for ( i = 1; i <= n; i++ )
                for ( j = 1; j <= n; j++ )
                        c[ i ][ j ] = INFTY;
        /* read cost matrix (integers) */
        for ( i = 1; i < n; i++ )
        {
                do
                {
                        printf(
                        "\nNode adjacent to %2d [0=finish]:",
                                                        i );
                        scanf( "%d", &j );
      while ( '\n' != getchar() );
                        if (j > 0 )
                        {
                                printf("\nCost c[%d][%d]:", i, j);
                                scanf( "%d", &c[ i ][ j ] );
        while ( '\n' != getchar() );
                          c[ j ][ i ] = c[ i ][ j ];
                                /* c is symmetric */
                        }
                }
                while ( j > 0 );
        }
        Prim2( n, c, e, &cost );
        printf("\nThe cost of MST is %d", cost );
        printf("\nTree edges\tEdge cost\n" );
        for ( i = 1; i <= n - 1; i++ )
                printf("%2.2d - %2.2d\t%10d\n",
                        e[ i ][ 0 ], e[ i ][ 1 ],
                        c[ e[ i ][ 0 ] ][ e[ i ][ 1 ] ] );
  return 0;
}
```

1.2.3 Divide et impera

Metoda “Divide et Impera” constă în împărţirea repetată a unei probleme în două sau mai multe probleme de acelaşi tip şi apoi combinarea subproblemelor rezolvate, în final obţinându-se soluţia problemei iniţiale.

Fie vectorul $A = (a_1, a_2, \ldots, a_n)$ ale carui elemente se proceseaza. Metoda “Divide et impera” este aplicabila daca pentru orice numere naturale p si q, astfel incat $1 \leq p < q \leq n$ exista un numar $m \in [p + 1, q - 1]$ astfel incat prelucrarea secventei $a_p, a_{p+1}, \ldots, a_q$ se poate face prelucrand secventele $a_p, a_{p+1}, \ldots, a_m$ si $a_m, a_{m+1}, \ldots, a_q$, şi apoi prin combinarea rezultatelor se obţine prelucrarea dorită.

Metoda “Divide et Impera” poate fi descrisă astfel:

```
void DivideAndConquer( int p, int q, SolutionT α )
/* p and q are indices in the processed sequence; α is the solution */
{
        int ε, m;
        SolutionT β, γ;

        if ( abs( q - p ) ≤ ε ) process( p, q, α );
        else
        {
```

```
                Divide( p, q, m );
                DivideAndConquer( p, m, β );
                DivideAndConquer( m + 1, q, γ );
                Combine( β, γ, α );
        }
}
```

Apelul funcţiei "DivideAndConquer" se face astfel::

```
DivideAndConquer( 1, n, α )
```

Semnificatia variabilelor din functiile definite mai sus este urmatoarea:

ϵ este lungimea maximă a unei secvenţe $a_p, a_{p+1}, \ldots, a_q$ pentru care prelucrarea se poate face direct;
m este indicele intermediar în care secvenţa $a_p, a_{p+1}, \ldots, a_q$ este împărţită în două subsecvenţe de funcţia $divide$;
$beta$ si γ reprezintă rezultatele intermediare obţinute în urma prelucrării subsecvenţelor $a_p, a_{p+1}, \ldots, a_m$ si respectiv $a_m, a_{m+1}, \ldots, a_q$;
α reprezintă rezultatul combinării rezultatelor intermediare β si γ;
$divide$ împarte secvenţa $a_p, a_{p+1}, \ldots, a_q$ in secventele $a_p, a_{p+1}, \ldots, a_m$ si $a_m, a_{m+1}, \ldots, a_q$;
$combine$ combina solutiile β si γ obţinând rezultatul $alpha$ a prelucrării secvenţei iniţiale.

1.2.4 Exemplu rezolvat cu "divide et impera"

Sortarea prin interclasare a unui vector de n elemente:

```c
#include <stdio.h>
#define MAXN 100
int A[ MAXN ]; /* vector to sort */

void printVector(int n)
/* print vector elements - 10 on one line */
{
  int i;

  printf( "\n" );
  for( i = 0; i < n; i++ )
  {
     printf("%5d", A[ i ]);
     if( (i + 1) % 10 == 0 )
       printf("\n");
  }
  printf("\n");
}
void merge(int lBound, int mid, int rBound)
{
  int i, j, k, l;
  int B[ MAXN ]; /* B = auxiliary vector */

  i = lBound;
  j = mid + 1;
  k = lBound;
  while( i <= mid && j <= rBound )
  {
    if( A[ i ] <= A[ j ])
    {
      B[ k ] = A[ i ];
      i++;
    }
    else
    {
      B[ k ] = A[ j ];
      j++;
    }
    k++;
  }
  for ( l = i; l <= mid; l++ )
  { /* there are elements on the left */
    B[ k ] = A[ l ];
    k++;
  }
  for ( l = j; l<= rBound; l++ )
  { /* there are elements on the right */
    B[ k ] = A[ l ];
```

```
    k++;
  }
  /* sequence from index lBound to rBound is now sorted */
  for( l = lBound; l <= rBound; l++ )
    A[ l ] = B[ l ];
}

void mergeSort( int lBound, int rBound)
{
  int mid;

  if( lBound < rBound)
  {
    mid= ( lBound + rBound ) / 2;
    mergeSort( lBound, mid );
    mergeSort( mid + 1, rBound);
    merge( lBound, mid, rBound);
  }
}
int main()
{
  int i, n;

  printf("\nNumber of elements in vector=");
  scanf( "%d", &n );
  while ( '\n' != getchar() );
  printf("\nPlease input vector elements\n");
  for( i = 0; i < n; i++ )
  {
    printf( "a[%d]=", i );
    scanf( "%d", &A[ i ] );
  }
  printf("\nUnsorted vector\n");
  printVector( n );
  mergeSort( 0, n-1 );
  printf("\nSorted vector\n");
  printVector( n );
  while ( '\n' != getchar() );
  while ( '\n' != getchar() );
  return 0;
}
```

In programul de mai sus combinarea solutiilor se face folosind $merge$.

1.3 Mersul lucrarii

1.3.1 Probleme Obligatorii

1. Algoritmul lui Prim

2. Colorarea hartilor. Harta unei regiuni a lumii contine n tari. Fiecare tara are mai multe tari vecine. Se dau m culori pentru a realiza colorarea hartii. Sa se determine toate posibilitatile de colorare a hartii folosind cele m culori astfel incat oricare doua tari vecine sunt colorate diferit. .
   I/O description. Input: numarul tarilor pe o line, urmata de tarile care sunt vecine, date ca perechi de nume – fiecare pereche de tari vecine fiind data pe o linie, apoi numarul de culori pe linia urmatoare, si numele culorilor date ca stringuri.

   ```
   9   # numar de tari
   Romania Ungaria
   Romania Serbia
   Romania Bulgaria
   ...
   5   # numar de culori
   rosu
   verde
   galben
   alb
   ...
   ```

Nota: tot ce urmeaza dupa # pe o linie de intrare este un comentariu si este ignorat. La iesire se vor afisa tara si culoarea cu care a fost colorata – pe cate o linie:

```
Romania␣galben
Ungaria␣verde
Serbia␣rosu
Ucraina␣alb
...
```

Folosind metoda "Divide et impera" rezolvati urmatoarele probleme:

3. Fiind dat un vector ce conţine elemente de tip întreg ordonate crescător, să se scrie o funcţie de căutare a unui element dat în vector, returnându-se poziţia sa.

4. Problema turnurilor din Hanoi. Se dau trei tije. Pe una dintre ele sunt aşezate n discuri de mărimi diferite, discurile de diametre mai mici fiind aşezate peste discurile cu diametre mai mari. Se cere să se mute aceste discuri pe o altă tijă, utilizând tija a treia ca intermediar, cu condiţia mutării a câte unui singur disc şi fără a pune un disc de diametru mai mare peste unul cu diametru mai mic.

1.3.2 Probleme Pentru Puncte în Plus

Folosind metoda Greedy rezolvati urmatoarele probleme:

1. Ciclu Hamiltonian: Un graf conex $G = (V, E)$ este dat de matricea sa de costuri, toate costurile fiind pozitive si ≤ 65534. Sa se determine un ciclu simplu care trece prin toate nodurile (Ciclu Hamiltonian) si are costul minim. .
I/O description. Input: Numarul de noduri pe o linie, urmat de matricea de costuri data rand cu rand (exemplu - vedeti Figura 1.1).

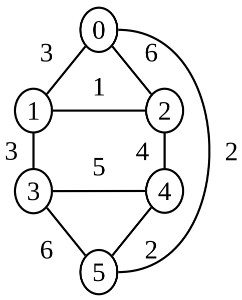

Figure 1.1: Exemplu pentru problema Ciclu hamiltonian

```
6
␣␣␣␣␣␣0␣␣␣␣␣1␣␣␣␣␣2␣␣␣␣␣3␣␣␣␣␣4␣␣␣␣␣5
0␣␣␣␣␣0␣␣␣␣␣3␣␣␣␣␣6␣65535␣65535␣␣␣␣␣2
1␣␣␣␣␣3␣␣␣␣␣0␣␣␣␣␣1␣␣␣␣␣3␣65535␣65535
2␣␣␣␣␣6␣␣␣␣␣1␣␣␣␣␣0␣65535␣␣␣␣␣4␣65535
3␣65535␣␣␣␣␣3␣65535␣␣␣␣␣0␣␣␣␣␣5␣␣␣␣␣6
4␣65535␣65535␣␣␣␣␣4␣␣␣␣␣5␣␣␣␣␣0␣␣␣␣␣2
5␣␣␣␣␣2␣65535␣65535␣␣␣␣␣6␣␣␣␣␣2␣␣␣␣␣0
```

Mai sus valoarea 65535 inseamna ca nu este arc intre cele doua noduri si este folosita pentru a marca$+\infty$. Nodurile sunt numerotate de la zero.
Output: pe o line se va afisa o secventa de noduri separate prin spatiu de exemplu: [a]

```
0␣2␣1␣3␣4␣5␣0
```

[a] Iesirea (outputul) este doar un exemplu de ciclu Hamiltonian - nu este un Ciclu Hamiltonian de lungime minima

2. Fiind data o matrice A de dimensiune $n \times n$, sa se determine cea mai mica diferenta a n numere intregi luate din diagonale diferite paralele cu diagonala secundara. Numerele considerate pot face parte si din diagonala secundara a matricii. .
I/O description. Input: numar de linii si coloane din matricea A, urmata de valorile din matrice date rand cu rand:

```
3
1␣␣␣␣7␣␣␣2
4␣␣␣␣5␣␣␣3
10␣␣-2␣␣␣0
```

Output:

```
-19
```

Elementele considerate sunt: -2, 7, si 10 (i.e. $-2 - 7 - 10$).

3. Un labirint este codificat folosind o matrice de dimensiune $n \times m$. In matrice sunt date niste coridoare codificate cu valori de 1 situate consecutiv pe linii sau coloane, restul elementelor fiind zero. O persoana se afla la pozitia (i, j) in labirint. Afisati toate traseele pe unde poate iesi persoana din labirint, fara ca sa treaca de doua ori prin acelasi loc. .
I/O description. Input: n si m pe o linie urmate de randurile matricii A, apoi sunt date coordonatele (rand, coloana) iesirii, si coordonatele la care se afla persoana (rand, coloana), ca in exemplul de mai jos:

```
25␣30
000000000000000000000000000000
001111110111111111011111111100
001000010100000000000001000100
...
```

Iesirea este o secventa de perechi rand $-$ coloana care indica pozitiile succesive din traseele pe unde poate iesi persoana din labirint.

Folosind metoda ”Divide et impera” rezolvati urmatoarele probleme:

1. Varianta cu 4 tije a problemei turnurilor din Hanoi: Se dau patru tije: A, B, C si D. Initial pe tija A sunt plasate un numar de discuri, avand discul cel mai mare jos si discul cel mai mic sus, asa cum arata figura 1.2. Se cere să

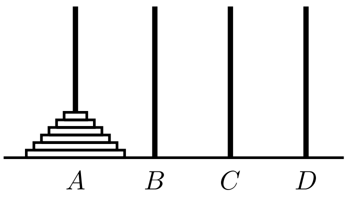

Figure 1.2: Pozitia initiala pentru problema ”turnurilor din Hanoi”.

se mute aceste discuri cate unul pe rand, din tija in tija, fara a pune un disc de diametru mai mare peste un disc cu diametru mai mic, astfel incat la final toate discurile sa ajunga pe tija D. .
I/O description. Input: numarul initial de discuri de pe tija A. Output: configuratiile consecutive.
E.g. pentru 7 discuri configuratia initiala ar fi:

```
A:␣1␣2␣3␣4␣5␣6␣7
B:
C:
D:
```

Numerele reprezinta diametrele discurilor, numarul cel mai mic indicand discul cu diametrul cel mai mic.

2. Fiind data o multime de n puncte (x_i, y_i) se cere sa se determine care este distanta dintre cele mai apropiate 2 puncte. O abordare neoptima ar determina distanta dintre oricare doua puncte dar aceasta ar lua foarte mult timp. O rezolvare folosind ”divide et impera” ar ordona punctele de-a lungul axei Ox, ar imparti regiunea in doua parti R_{left} si R_{right} avand un numar egal de puncte si apoi ar aplica recursiv algoritmul pe sub-regiuni, si ar calcula distanta minima in regiunea originala.

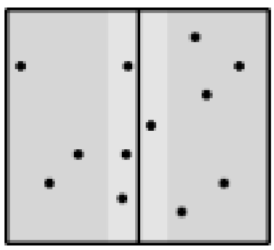

Figure 1.3: Problema celor mai apropiate 2 puncte

Perechea cu cele mai apropiate 2 puncte poate fi in regiunea din stanga (left), in regiunea dreapta (right) sau pe linia de separare a celor doua regiuni. Ultimul caz trateaza doar punctele la distanta $d = \min(d_{left}, d_{right})$ de linia de

separare unde d_{left} si d_{right} sunt distantele minime pentru regiunea stanga si regiunea dreapta.
Punctele din regiunea care este in zona liniei de separare sunt ordonate de-a lungul coordonatei y, si procesate in acea ordine. Procesarea consta in compararea fiecarul punct din aceasta zona cu punctele care se afla inainte lor cu cel mult distanta d, pe axa y. Deoarece o fereastra de dimensiune $d \times 2d$ poate contine cel mult 6 puncte, va fi necesar sa se calculeze cel mult 5 distante pentru fiecare din aceste puncte. Vedeti figura 1.3.
Ordonarea punctelor dupa coordonatele x si y se poate face inainte de a aplica algoritmul "divide et impera".
.
I/O description. Input: n, numarul de puncte, urmat de coordonatele punctelor (x_i, y_i) pe o linie. Output: coordonatele celor mai apropiate 2 puncte, $(x_i, y_i)(x_j, y_j)$ pe o linie. Folositi paranteze pentru a marca o pereche de coordonate.

1.3.3 Probleme Optionale

1. **Minimizarea timpului mediu de asteptare**: La un cabinet stomatologic se prezinta simultan n pacienti. Sa se determine ordinea in care medicul stomatolog va trata pacientii astfel incat sa se minimizeze timpul mediu de asteptare, daca se cunosc duratele tratamentelor celor n pacienti. (Durata tratamentului d_j pentru pacientul j se citeste de la tastatura). Timpul mediu de asteptare este media aritmetica a timpilor de asteptare a celor n pacienti. Astfel minimizarea timpului mediu de asteptare revine la minimizarea timpului total de asteptare.

2. **Inchirieri auto**: o companie de transporturi inchiriaza autovehicule. Unul din vehicule este extrem de solicitat; de aceea solicitarile pentru anul urmator se aduna pe parcursul anului curent. Solicitarea se precizeaza printr-o pereche de numere, a, b reprezentand numerele de ordine ale zilelor din an intre care se doreste inchirierea. Determinati o solutie de inchiriere care sa asigure inchirierea autovehiculului unui numar maxim de persoane stiind ca exista n persoane care au solicitat inchirierea ($n \leq 100$). Se vor afisa numarul maxim de persoane si perioadele de inchiriere.