CUPRINS

CAPITOLUL I. NOŢIUNI TEORETICE

În căutarea unor principii fundamentale ale elaborării algoritmilor, backtraking este una dintre cele mai generale tehnici. Multe probleme care necesită căutarea unui set de soluţii sau o soluţie optimă care să satisfacă anumite restricţii pot fi rezolvate utilizând această metodă. Denumirea backtraking (căutarea cu revenire) a fost utilizată pentru prima oară în anii '50 de către D. H. Lehmer. Cei care au studiat procesul înaintea sa au fost: R. J. Walker (care i-a dat o interpretare algoritmică în 1960), S. Golomb şi L. Bammert (care au prezentat o descriere generală a metodei, precum şi varietatea de aplicabilitate a acesteia).

1. Prezentarea metodei

Această metodă se aplică problemelor a căror soluţie poate fi pusă sub forma unui vector $\mathbf{x_1, x_2, \dots, x_n}$ în care $x_1 \in A_1, \dots, x_n \in A_n$ sunt mulţimi finite şi nevide şi ale căror elemente se află într-o relaţie de ordine bine stabilită. De multe ori mulţimile $A_1, \dots, A_n$ coincid.

Metoda backtraking construieşte soluţia problemei progresiv, obţinând pe rând elementele $x_1, x_2, \dots, x_n$, netrecând la componenta x_k până când nu s-au stabilit celelate componente, $x_1, \dots, x_{k-1}$.

Presupunem că am generat deja elementele $x_1, \dots, x_k$. Următorul pas este generarea lui $x_{k+1} \in A_{k+1}$. În acest scop se caută în mulţimea A_{k+1} următorul element disponibil. Există două posibilităţi:

1. Să nu existe un astfel de element în mulţimea A_{k+1}. În această situaţie se renunţă la ultimul x_k ales, deci s-a făcut un pas înapoi şi în continuare se caută generarea unui alt element x_k din mulţimea A_k.
2. Există un x_{k+1} disponibil în mulţimea A_{k+1}. În această situaţie se verifică dacă acest element îndeplineşte condiţiile (condiţii de continuare) cerute de enunţul problemei de a face parte din soluţie.

Apar alte posibilităţi.

Dacă x_{k+1} ales îndeplineşte condiţiile de continuare şi poate face parte din soluţie, atunci apar alte două situaţii:

2.1. Să se fi ajuns la soluţia finală a problemei şi aceasta trebuie tipărită;

2.2.Să nu se fi ajuns la soluţia finală şi atunci se consideră generat x_{k+1}, făcând un pas înainte la $x_{k+2} \in A_{k+2}$.

Dacă x_{k+1} nu îndeplineşte condiţiile de continuare atunci se caută următorul element disponibil în mulţimea A_k. Dacă şi ea se epuizează se face un pas înapoi renunzându-se la x_k.

Observaţie: Problema se termină, cu tipărirea tuturor soluţiilor, când a fost epuizată mulţimea A_1.

Pentru a implementa această metodă într-un limbaj de programare, respectiv în C++ , se foloseşte o structură de date de tip special (LIFO – *last in first out*), numită *stivă*, caracterizată prin faptul că operaţiile permise asupra ei se pot realiza la un singur capăt numit vârful stivei. Stiva se asimilează cu un vector pe verticală şi în ea se construieşte soluţia problemei.

Această structură are nevoie de o variabilă k care precizează în permanenţă nivelul stivei.

Dacă k = 0 se spune că stiva este vidă.

Dacă k = n se spune că stiva este plină.

Orice adăugare a unui element în stivă presupune creşterea nivelului stivei: k=k+1.

Orice eliminare a unui element din stivă presupune scăderea nivelului stivei : k = k – 1.

Stiva pe care o folosim este o stivă simplă, având capacitatea de a memora pe un anumit nivel o literă sau o cifră.

Metoda de programare backtracking admite atât o implementare interativă cât şi una recursivă, folosind o serie de subprograme cu rol bine determinat, corpul lor de instrucţiuni depinzând de natura şi complexitatea problemei.

2. Strategia generală backtraking în varianta iterativă

Se folosesc următoarele funcţii:

►***Funcţia*** **init (), de tip void –** cu rolul de a iniţializa fiecare nivel k al stivei, cu o valoare neutră, care nu face parte din soluţie, dar care permite, pentru fiecare componentă a soluţiei, trecerea la primul element al domeniului său de valori, ca şi în cazul celorlalte, conform relaţiei de ordine în care acestea se află.

►***Funcţia*** **succesor (), de tip int** – cu rolul de a găsi un succesor, adică de a căuta un nou element disponibil în mulţimea de valori a unei componete din soluţie. Aceată funcţie va returna valoarea 1 dacă există succesor şi valoarea 0 în caz contrar.

► ***Funcţia*** **valid (), de tip int –** cu rolul de a verifica dacă succesorul ales îndeplineşte condiţiile de continuare. Aceată funcţie va returna valoarea 1 dacă succesorul este valid (respectă condiţiile de continuare) şi valoarea 0 în caz contrar.

► ***Funcţia*** **tipar(), de tip void –** cu rolul de a tipări o soluţie determinată.

► ***Funcţia*** **soluţie(), de tip int** – cu rolul de a verifica, la nivelul k al stivei, dacă s-a ajuns la soluţie sau nu, returnând, în mod corespunzător valoarea 1 sau 0.

Strategia generală backtraking se implementează în C, într-o functie de tip void, în felul următor:

```
void back()
        {
         k=1 ;
         init();
         while  (k>0)
                    {as=1; ev=0;
                      while ( as && !ev)
                              { as=succesor ();
                              if (as)
                                      ev= valid ();
```

```
                    }

        if (as)
                if soluţie( )  tipar();
                        else
                                {   k:= k+1;
                                  init ();
                                 };
                else k:=k-1;
          }
    }
```

Se folosesc variabilele întregi as şi ev, cu semnificaţia “am succesor”, respectiv “este valid”.

3. Recursivitatea

Recursivitatea este una din noţiunile fundamentale ale informaticii. Utilizarea frecventă a recursivităţii s-a făcut după anii ’80. Multe din limbajele de programare evoluate şi mult utilizate nu permiteau scrierea programelor recursive. În linii mari, recursivitatea este un mecanism general de elaborare a programelor. Ea a apărut din necesităţi practice (transcrierea directă a formulelor matematice recursive) şi reprezintă acel mecanism prin care un subprogram se autoapelează.

Un algoritm recursiv are la bază un mecanism de gândire diferit de cel cu care ne-am obişnuit deja. Atunci când scriem un algoritm recursiv este suficient să gândim ce se întâmplă la un anumit nivel pentru că la orice nivel se întâmplă exact acelaşi lucru.

Un algoritm recursiv corect trebuie să se termine, contrar programul se va termina cu eroare şi nu vom primi rezultatul aşteptat. Condiţia de terminare va fi pusă de programator.

Un rezultat matematic de excepţie afirmă că pentru orice algoritm iterativ există şi unul recursiv echivalent (rezolvă aceaşi problemă) şi invers, pentru orice algoritm recursiv există şi unul iterativ echivalent.

În continuare, răspundem la întrebarea: care este mecanismul intern al limbajului care permite ca un algoritm recursiv să poată fi implementat? Pentru a putea implementa recursivitatea, se foloseşte structura de date numită stivă.

Mecanismul unui astfel de program poate fi generalizat cu uşurinţă pentru obţinerea recursivităţii. Atunci când o funcţie se autoapelează se depun în stivă:

- valorile parametrilor transmişi prin valoare;
- adresele parametrilor transmişi prin referinţă;
- valorile tuturor variabilelor locale (declarate la nivelul funcţiei);
- adresa de revenire la instrucţinea imediat următoare autoapelului.

4. Strategia generală backtraking în varianta recursivă

Metoda de programare backtraking are şi o variantă recursivă. Prelucrările care se fac pentru elementul **k** al soluţiei se fac şi pentru elementul **k+1** al soluţiei şi aceste prelucrări se pot face prin apel recursiv. În algoritmul backtracking implementat iterativ, revenirea la nivelul **k-1** trebuie să se facă atunci când pe nivelul **k** nu se găseşte o valoare care să îndeplinească **condiţiile interne**. În cazul implementării recursive, condiţia de bază este ca pe nivelul **k** să nu se găsească o valoare care să îndeplinească condiţiile interne. Când se ajunge la condiţia de bază, încetează apelul recursiv şi se revine la subprogramul apelant, adică la la subprogramul care prelucrează elementul k-1 al soluţieie, iar în stivă se vor regăsi valorile prelucrate anterior în acest subprogram.

Întreaga rutină care implementează algoritmul backtracking recursiv se va transforma într-o funcţie de tip void (procedurală) ce se va apela din programul principal prin ***back(1)***, având ca parametru nivelul ***k*** al stivei; aceasta se va autoapela pe nivelele următoare ale stivei.

```
void back (int k)
{
  init(k);
  while (succesor(k))
        if ( valid(k))
              if (solutie(k))
                    tipar (k);
                    else
                    back(k+1);
  }
```

Această procedură funcţionează în felul următor:

- se testează dacă s-a ajuns la soluţie; dacă da, aceasta se tipăreşte şi se revine la nivelul anterior;

-în caz contrar, se iniţiază nivelul stivei şi se caută un succesor;

-dacă am găsit un succesor, se verifică dacă este valid şi dacă da se autoapelează procedura pentru (k+1); în caz contrar, se continuă căutarea succesorului;

-dacă nu există succesor se revine la nivelul anterior (k+1) prin ieşirea din procedura back.

CAPITOLUL II. EXEMPLU DE APLICARE A METODEI

PROBLEMA CELOR N DAME

O problemă clasică este plasarea a n dame pe o tablă de şah astfel încât să nu se atace reciproc, altfel spus, să nu se afle două dame pe aceeaşi linie, pe aceeaşi coloană sau diagonală.

Se numerotează liniile şi coloanele de pe tabla de şah de la 1 la n. De asemenea, damele se numerotează de la 1 la n. Dacă fiecare damă trebuie să se afle pe rânduri diferite, putem plasa dama k pe linia k, deci toate soluţiile problemei pot fi reprezentate ca vectori $(x_1,x_2,\ldots,x_n)$, unde x_k este coloana unde este plasată dama k.

Rezultă că domeniul soluţiilor este format din n^n vectori. Restricţiile problemei impun ca toate elementele x_k să fie diferite între ele (toate damele să se afle pe coloane diferite) şi nici o damă să nu fie pe aceeaşi diagonală cu o altă damă. Prima din aceste două restricţii presupune că toate soluţiile sunt permutări ale vectorului (1,2,3,…,n). Acest lucru reduce domeniul soluţiilor de la n^n la n! posibilităţi.

A doua restricţie: pentru ca dama de pe linia i coloana x_i să fie pe aceeaşi diagonală cu dama de pe linia k şi coloana x_k, triunghiul care se formează ar trebui să aibă unghiurile de 45^0 , deci să fie isoscel. Prin urmare catetele de lungimi i-k ,respectiv $x_1 - x_k$ trebuie să fie egale; condiţia $i - k \neq x_i - x_k$ exprimă faptul că două dame nu pot fi plasate pe aceeaşi diagonală. Faptul că nu pot fi plasate două dame pe aceeaşi coloană poate fi exprimat prin condiţia $x_1 \neq x_k$.

			D				
					D		
							D
D							
						D	
D							
		D					
				D			

```
#include<iostream.h>
#include<math.h>
 int stiva[100];
int n,k,ev,as;
stiva st;
void init ()
{
st[k]=0;
}

int succesor ()
{
if (st[k]<n)
        {
        st[k]=st[k]+1;
        return 1;
        }
        else
        return 0;
}

int valid ()
{
for (int i=1; i<=n; i++)
        if (st[k]= =st[i]) | | abs(st[k]-st[i] = = k-i)
                return 0;
return 1;
}
```

```
int solutie ()
{
return k= = n;
}
void tipar ()
{
for (int i=1; i<=n; i++)
        cout<<st[i]<< “  “;
        cout<< endl;
}
```

Observaţie: Problemele rezolvate prin această metodă necesită un timp îndelungat. Din acest motiv, este bine să utilizăm metoda numai atunci când nu avem la dispoziţie un alt algoritm mai eficient. Cu toate că există probleme pentru care nu se pot elabora alţi algoritmi mai eficienţi, tehnica ***backtracking*** trebuie aplicată numai în ultimă instanţă.

Fiind dată o tablă de şah, de dimensiune n, x_n, se cer toate soluţiile de aranjare a n dame, astfel încât să nu se afle două dame pe aceeaşi linie, coloană sau diagonală (damele să nu se atace reciproc).

Exemplu: Presupunând că dispunem de o tablă de dimensiune 4x4, o soluţie ar fi următoarea:

	D		
			D
D			
		D	

Observăm că o damă trebuie să fie plasată singură pe linie. Plasăm prima damă pe linia 1, coloana 1.

D			

A doua damă nu poate fi aşezată decât în coloana 3.

D			
		D	

Observăm că a treia damă nu poate fi plasată în linia 3. Încercăm atunci plasarea celei de-a doua dame în coloana 4.

D			
			D

A treia damă nu poate fi plasată decât în coloana 2.

D			
			D
	D		

În această situaţie dama a patra nu mai poate fi aşezată.

Încercând să avansăm cu dama a treia, observăm că nu este posibil să o plasăm nici în coloana 3, nici în coloana 4, deci o vom scoate de pe tablă. Dama a doua nu mai poate avansa, deci şi ea este scoasă de pe tablă. Avansăm cu prima damă în coloana 2.

	D		

A doua damă nu poate fi aşezată decât în coloana 4.

	D		
			D

Dama a treia se aşează în prima coloană.

	D		
			D
D			

Acum este posibil să plasăm a patra damă în coloana 3 si astfel am obţinut o soluţie a problemei.

	D		
			D
D			
		D	

Algoritmul continuă în acest mod până când trebuie scoasă de pe tablă prima damă.

Pentru reprezentarea unei soluţii putem folosi un vector cu n componente (având în vedere că pe fiecare linie se găseşte o singură damă).

Exemplu pentru soluţia găsită avem vectorul ST ce poate fi asimilat unei stive.

Două dame se găsesc pe aceeaşi diagonală dacă si numai dacă este îndeplinită condiţia: |st(i)-st(j)|=|i-j| (diferenţa, în modul, între linii si coloane este aceeaşi).

ST(4)

ST(3) În general ST(i)=k semnifică faptul că pe linia i dama ocupa poziţia k. ST(2)

ST(1)

Exemplu: în tabla 4 x4 avem situaţia:

	D		
			D
D			
		D	

st(1)= 1 i = 1

st(3)= 3 j = 3

|st(1) - st(3)| = |1 – 3| = 2

|i – j| = |1 – 3| = 2

sau situaţia:

	D		
			D
D			
		D	

st(1) = 3 i = 1

st(3) = 1 j = 3

|st(i) - st(j)| = |3 – 1| = 2

|i – j| = |1 – 3| = 2

Întrucât doua dame nu se pot găsi în aceeaşi coloană, rezultă că o soluţie este sub formă de permutare. O primă idee ne conduce la generarea tuturor permutărilor si la extragerea soluţiilor pentru problema ca două dame să nu fie plasate în aceeaşi diagonală. A proceda astfel, înseamnă că lucrăm conform strategiei ***backtracking***. Aceasta presupune ca imediat ce am găsit două dame care se atacă, să reluăm căutarea.

lată algoritmul, conform strategiei generate de ***backtracking***:

- În prima poziţie a stivei se încarcă valoarea 1, cu semnificaţia că în linia unu se aşează prima damă în coloană.

- Linia 2 se încearcă aşezarea damei în coloana 1, acest lucru nefiind posibil întrucât avem doua dame pe aceeaşi coloană.

- În linia 2 se încearcă aşezarea damei în coloana 2 , însă acest lucru nu este posibil, pentru că damele se găsesc pe aceiaşi diagonală (|st(1)-st(2)|=|1-2|);

- Aşezarea damei 2 în coloana 3 este posibilă.

- Nu se poate plasa dama 3 în coloana 1, întrucât în liniile 1-3 damele ocupa acelaşi coloană.

- Şi această încercare eşuează întrucât damele de pe 2 şi 3 sunt pe aceeaşi diagonală.

- Damele de pe 2-3 se găsesc pe aceeaşi coloană.

- Damele de pe 2-3 se găsesc pe aceeaşi diagonală.

- Am coborât în stivă mutând dama de pe linia 2 şi coloana 3 în coloana 4.

Algoritmul se încheie atunci când stiva este vidă. Semnificaţia procedurilor utilizate este următoarea:

INIT - nivelul k al stivei este iniţializat cu 0;

SUCCESOR - măreşte cu 1 valoarea aflată pe nivelul k al stivei în situaţia în care aceasta este mai mică decât n şi atribuie variabilei EV valoarea TRUE, în caz contrar, atribuie variabilei EV valoarea FALSE;

VALID - validează valoarea pusă pe nivelul k al stivei, verificând dacă nu avem două dame pe aceeaşi linie (st(k)=st(i)), sau dacă nu avem două dame pe aceeaşi diagonală (st(k)-st(i)=|k-i|)caz în care variabilei EV i se atribuie FALSE; în caz contrar, variabilei EV i se atribuie TRUE;

SOLUTIE - verifică dacă stiva a fost completată până la nivelul n inclusiv;

TIPAR - tipăreşte o soluţie.

CAPITOLUL III. APLICAŢIE

Se consideră n piese de domino, memorate ca n perechi de numere naturale în fisierul text domino1.txt. Primul număr al perechii reprzintă valoarea jumătăţii superioare a pisei, iar al doilea număr reprezintă valoarea jumătăţii inferioare.

Se cere să se afişeze, în fişierul text domino2.txt, toate soluţiile de aşezare a acestor piese într-un lanţ domino de lungime a, fără a roti piesele.

Un lanţ domino se alcătuieşte din pise domino astfel încât o piesă este urmată de o alta a cărei primă jumătate coincide cu a doua jumătate a piesei curente.

Programul C++ corespunzător este:

```
#include<iostream.h>
struct piese
{int p,u;};
 piese domino[20];
int st[20],n,sol,k,as,ev,a,i;
void init()
{
st[k]=0;
}
int succesor()
{
if (st[k]<n)
        {st[k]=st[k]+1;
        return 1;
        }
        else return 0;
}
```

```
int valid()
{
if(k>1)
        {
        if (domino[st[k]].p!=domino[st[k-1]].u)
        return 0;
        for (i=1;i<=k-1;i++)
                if(st[k]==st[i])
                return 0;
        }
return 1;
}
int solutie()
{
if(k==a)
        return 1;
else
        return 0;
}
void tipar ()
{

sol++;
cout<<"Lant domino, nr."<<sol<<endl;
for (i=1;i<=a;i++)
        {
        cout<<domino[st[i]].p<<" "<<domino[st[i]].u<<"  ";
        }
        cout<<endl;
}
void lant()
{

        k=1;
        while (k>0)
        {
        as=1;ev=0;
        while(as && !ev)
                {
                as=succesor();
             if(as)
                        ev=valid();
                }
        if (as)
                if(solutie())
                        tipar();
```

```
                        else
                        {
                        k++;
                        init();
                        }
        else k--;
        }
}
void main()
{
cout <<"dati numarul total de piese";
cin>>n;
cout<<"dati perechile de numere care formeaza piesele";
for(i=1;i<=n;i++)
        {
        cout<<"primul numar:";
        cin>>domino[i].p;
        cout<<"al doilea numar:";
        cin>>domino[i].u;
        }
cout<<"dati lungimea lantului de piese:";
cin>>a;
lant();
cout<<sol;
}
```

Pentru următorul set de date de intrare:

n=**4**;
piesa 1: **1 2**
piesa 2: **2 3**
piesa 3: **3 1**
piesa 4: **1 3**
a=**3**
pe ecran vor fi afişate următoarele soluţii:

Lant domino, nr.1
1 2 2 3 3 1
Lant domino, nr.2
2 3 3 1 1 2
Lant domino, nr.3
2 3 3 1 1 3
Lant domino, nr.4
3 1 1 2 2 3

Lant domino, nr.5
1 3 3 1 1 2
5 solutii

Pentru comoditatea folosirii programului şi pentru teste diverse ale datelor, se pot folosi fişierele text atât pentru citirea datelor de intrare cât şi pentru afişarea rezultatelor.

În programul următor se foloseşte ca fişier de intrare domino1.txt, cu următorul conţinut:

```
4
1  2
2  3
3  1
1  3
3
```

unde pe primul rând se află n, numărul total de piese, pe următoarele n rânduri se află perechile de numere ce alcătuies pisele, separate prin spaţiu, iar pe ultimul rând se află a, lunimea lanţurilor cerute.

```
#include<iostream.h>
#include<fstream.h>

struct piese
{int p,u;};
 piese domino[20];
int st[20],n,sol,k,as,ev,a,i;
ifstream f("domino1.txt");
ofstream g("domino2.txt");
void init()
{
st[k]=0;
}
int succesor()
{
if (st[k]<n)
        {st[k]=st[k]+1;
        return 1;
        }
        else return 0;
}

int valid()
{
if(k>1)
        {
        if (domino[st[k]].p!=domino[st[k-1]].u)
        return 0;
```

```
        for (i=1;i<=k-1;i++)
                if(st[k]==st[i])
                return 0;
        }
return 1;
}
int solutie()
{
if(k==a)
        return 1;
else
        return 0;
}
void tipar ()
{

sol++;
g<<"Lant domino, nr."<<sol<<endl;
for (i=1;i<=a;i++)
        {
        g<<domino[st[i]].p<<" "<<domino[st[i]].u<<"  ";
        }
        g<<endl;
}
void lant()
{

        k=1;
        while (k>0)
        {
        as=1;ev=0;
        while(as && !ev)
                {
                as=succesor();
             if(as)
                        ev=valid();
                }
        if (as)
                if(solutie())
                        tipar();
                        else
                        {
                        k++;
                        init();
                        }
        else k--;
```

```
        }
}
void main()
{
f>>n;

for(i=1;i<=n;i++)
        {

        f>>domino[i].p>>domino[i].u;
        }
f>>a;
lant();
g<<sol<<"solutii";
f.close(); g.close();
}
```

În urma rulării acestui program se va crea fişierul text domino2.txt, cu următorul conţinut:

```
Lant domino, nr.1
1 2   2 3   3 1
Lant domino, nr.2
2 3   3 1   1 2
Lant domino, nr.3
2 3   3 1   1 3
Lant domino, nr.4
3 1   1 2   2 3
Lant domino, nr.5
1 3   3 1   1 2
5 solutii
```

Bilbliografie

Balanescu, T., Gavrila, S., Georgescu, H., Gheorghe, M., Sofonea, L., Vaduva, I., *Limbajul C++. Concepte fundamentale*-volumul I, Editura Tehnica, Bucuresti, 1992;

Cristian Udrea, *Informatica(Teorie si aplicatii)*-Vol II (Clasa a X-a), Editura Arves, 2002;

Tudor, Sorin, *Informatica (Tehnici de programare*)– Manual pentru clasa a X- a,Varianta C++, Editura L&S Infomat, Bucuresti, 2002;

Tudor, Sorin, *Informatica. Varianta C++* – Manual pentru clasa a XI- a, Editura L&S Infomat, Bucuresti, 2002;

Stelian Niculescu**,** Emanuela Cerchez, *Bacalaureat si atestat* - Editura L&S Informat, 1999;

Doina Roncea, *Limbajul C++*, Editura Libris, 2004;

Atanasiu Adrian, Pintea Rodica, *Culegere de probleme Pascal/C++*, Editura Petrion, Bucuresti 2006.