Instituția Publică Liceul Teoretic “Spiru Haret”

PROIECT
LA INFORMATICĂ

TEMA: **”**Tehnici de elaborare a algoritmilor.

Metoda reluării.(BackTracking) **“**

Efectuat de: Rîșneanu Ilinca

Profesor: Guțu Maria

Chișinău, 2019

Cuprins

Informații generale

La dispoziția celor care rezolvă probleme cu ajutorul calculatorului există mai multe metode. Dintre acestea cel mai des utilizate sunt:

- metoda Greedy;
- metoda Divide et impera;
- metoda Branch and Bound;
- metoda **Backtracking**;

Metoda Reluării (Backtracking) se aplică problemelor în care soluția poate fi reprezentată sub forma unui vector:

$X = (x_1, x_2, x_3, \ldots, x_k, \ldots, x_n)$.

Cerința problemei este, de obicei, găsirea tuturor soluțiilor posibile sau găsirea numărului de soluții care satisfac anumite condiții specifice problemei.

Fiecare component x_k a vectorului X poate lua valori dintr-o anumită mulțime A_k, k=1,2,…, n. Se consideră că cele m_k elemente ale fiecărei mulțimi A_k sunt **ordonate** conform unui criteriu bine stabilit. Deci, în metoda reluării componentele vectorului X primesc valori pe rând, în sensul că lui x_k i se atribuie o valoare numai dacă au fost deja atribuite valori lui $x_1, x_2, \ldots, x_{k-1}$.

Anume micșorarea lui k dă nume metodei, cuvântul *"reluare"*, semnificând revenirea la alte variante de alegere a variabilelor. Aceeaşi semnificaţie o are și denumirea engleză a metodei – *backtracking*

(back-înapoi, track - urmă).

Schema generală

Schema generală a unui algoritm recursive bazat pe metoda reluării este redată cu ajutorul procedurii ce urmează:

```
procedure Reluare(k:integer);
 begin
 if k<=n then
 begin
 X[k] :=PrimulElement (k);
 if Continuare (k) then Reluare (k+1);
 while  ExistaSuccesor (k) do
 begin
 X[k] :=Succesor (k);
 if Continuare (k) then Reluare (k+1)
 end;
 end
 else PrelucrareaSolutiei;
 end;
```

Procedura Reluare comunică cu programul apelant şi subprogramele apelate prin variabilele globale ce reprezintă vectorul X şi multimile A_1, A_2,...,A_n. Subprogramele apelate execută următoarele operații:

PrimulElement (k) – returnează primul element din mulțimea A_k;

Continuare (k) – returnează valoare *true* dacă elementele înscrise în primele k componente ale vectorului X satisfac condițiile de continuare şi *false* în caz contrar;

ExistaSuccesor (k) – returnează valoarea *true* dacă elementul memorat în componenta x_k are un succesor în multimea A_k şi *false* în caz contrar;

Succesor (k) – returnează succesorul elementului memorat în componenta x_k;

PrelucrareaSolutiei – de obicei, în această procedură soluția reținută în vectorul X este afişată la ecran;

Probleme rezolvate

1. ***Generarea produsului cartezian a n multimi***

Se consideră n mulțimi finite S_1, S_2, … S_n, de forma {1,2..,S_n}. Să se genereze produsul cartezian al acestor mulțimi.

Indicatii de rezolvare:

Am considerat mulțimile de forma {1,2..,S_n} pentru a simplifica problema, în special la partea de citire și afişare, algoritmul de generare rămânând nemodificat.

ldentificăm următoarele particularități şi condiții:

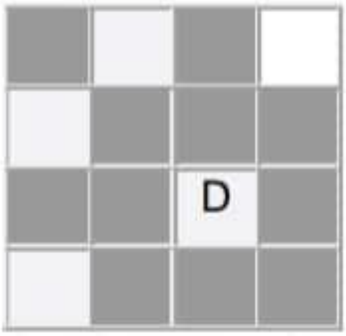
- Vectorul soluție: X=(x_1, x_2,... x_n) ∈ S1xS2x...Sn
- Fiecare element $X_k \in S_k$
- Nu există condiții interne pentru vectorul soluție. Nu are funcție de validare.
- Obținem soluția când s-au generat n valori.
- Avem soluție dacă k=n+1

```
//generare prod cartezian
#include <iostream>
#include <fstream>
using namespace std;
int x[50], n,k,s[50][100], card[50];
ifstream f("fis.in");
void citeste()
{  int i,j; f>>n;
   for(i=1;i<=n;i++)
   {   f>>card[i];
       for(j=1;j<=card[i];j++) f>>s[i][j];}}
void scrie()
{    int i;cout<<endl;
     for(i=1;i<=n;i++) cout<<s[i][x[i]]<<" ";}
int solutie(int k) { return(k==n+1);}

void back(int k)
{    if(solutie(k))scrie();
     else
      for(int i=1;i<=card[k];i++)
        {x[k]=i;
          // if(valid(k)) nu este cazul
        back(k+1);}
}
int main()
{citeste(); back(1); return 0;}
```

2. *Aranjarea a n regine pe o tablă de șah de dimensiune nxn fără ca ele să se atace.*

Dându-se o tablă de şah de dimensiune nxn (n>3) să se aranjeze pe ea n regine fără ca ele să se atace. Reamintim că o regină atacă linia, coloana și cele 2 diagonale pe care se află. În figura de mai jos celulele colorate mai închis sunt atacate de regina poziționată unde indică litera “D”.



Se plaseaza câte o regină pe fiecare linie.
Condiţia de a putea plasa o regină pe poziţia k presupune verificarea ca să nu se atace cu nici una dintre celelalte k-1 regine deja plasate pe tabla. Dacă pe poziţia k din vectorul X punem o valoare ea va reprezenta coloana pe care se plasează pe tablă regina k.

- x[k] ϵ { 1,2,......,n };
- Validare(k): x[i]≠x[k] şi |k-i|≠|x[k]-x[i]| cu i=2,...,k-1.
- Soluție: k=n+1

```
function cont(x:stiva;
k:integer):boolean;
var i:integer;
begin
 cont:=true;
 for i:=1 to k-1 do
  if (x[i]=x[k]) or
     (abs(x[k]-x[i])=abs(k-i))
  then
     cont:=false;
end;
```

```
int cont(int x[], int k)
{



 for(int i=0;i<k;i++)
  if ((x[i]==x[k]) ||
   (abs(x[k]-x[i])== k-i))
       return 0;
 return 1;

}
```

3. *Generarea permutărilor*

Se citește un număr natural n. Să se genereze toate permutările mulțimii {1,2,…,n}.

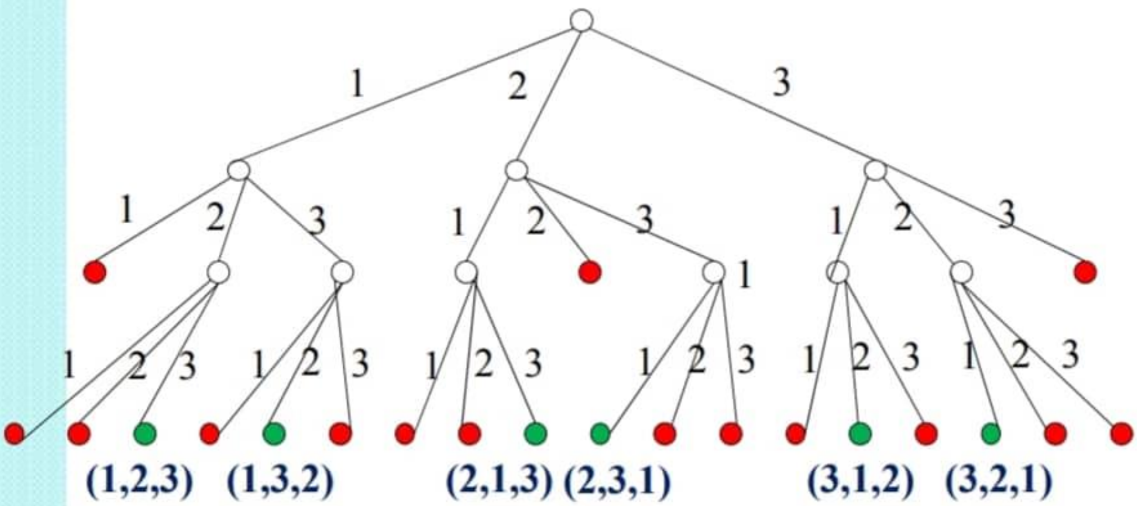
- Reprezentarea soluţiei ?
- Condiţii interne ?
- Condiţii de continuare ?

- ✓ $x = (x_1, x_2, \ldots, x_n)$
- ✓ Condiţii interne: pentru orice $i \neq j$, $x_i \neq x_j$
- ✓ Condiţii de continuare: pentru orice $\mathbf{i<k}$, $x_i \neq x_k$

Exemplu: *n* = 3

= {1,2,3} × {1,2,3} × {1,2,3}

$x = (x_1, x_2, x_3)$



4. *Generarea șirurilor de n paranteze ce se închid corect*

Se citeşte de la tastatură un număr natural n, $n \leq 30$. Să se genereze şi să se afişeze pe ecran toate combinaţiile de n paranteze rotunde care se închid corect.

✓ De exemplu, pentru n = 4 se obţin următoarele combinaţii:

(( )), ( )( )

✓ Pentru n = 6 se obţin combinaţiile:

((( ))), (()()), ( )( )( ), ( )(( )), (( ))( ).

- Există soluţii ⇔ n este par.
- **Reprezentarea soluţiei**: $x = (x_1, x_2, \ldots, x_n)$, unde $x_k \in \{'(', ')'\}$
- Notăm *dif = nr(- nr)* la pasul k
- **Condiţii interne (finale)**

  ◦ dif = 0

  ◦ dif ≥ 0 pentru orice secvență $\{x_1, x2, \ldots, x_k\}$

- **Condiţii de continuare**

  ◦ dif ≥ 0 → doar necesar

  ◦ dif ≤ n – k → şi suficient

- **Observaţie**. În implementarea următoare backtracking-ul este **optimal**: se avansează dacă şi numai dacă suntem siguri că vom obţine cel puţin o soluţie. Cu alte cuvinte, **condiţiile de continuare nu sunt numai necesare, dar şi suficiente.**

```
procedure backrec(k:integer);
 begin
  if(k=n+1) then
    tipar(x)
  else
   begin
     x[k]:='(';
     dif:=dif+1;
     if(dif <= n-k+1) then
          backrec(k+1);
     dif:=dif-1;
     x[k]:=')';
     dif:=dif-1;
     if(dif >= 0) then
          backrec(k+1);
     dif:=dif+1;
   end;
end;
```

```
void backrec(int k)
{
 if(k==n)
     tipar(x);
 else
  {
     x[k]='(';
        dif++;
        if (dif <= n-k)
            backrec(k+1);
        dif--;
        x[k]=')';
        dif--;
        if (dif >= 0)
             backrec(k+1);
        dif++;
  }
}
```

```
if(n mod 2=0) then
 begin
   dif:=0;
   backrec(1);
 end;
```

```
if (n%2==0)
    {
      dif=0;
      backrec(0);
    }
```

5. *Partițiile unui număr natural*

Dat un număr natural n, să se genereze toate partițiile lui n ca sumă de numere pozitive (x_1, x_2,…, $x_k \in \{1,2,\ldots,n\}$ cu proprietatea $x_1 + x_2 + \ldots + x_k = n$).

- ✓ De exemplu, pentru n = 4, partițiile sunt

  1+1+1+1; 1+1+2; 1+3; 2+2; 4 ;

- ✓ Pentru n = 6, partițiile sunt

  1+1+1+1+1+1; 1+1+1+1+2; 1+1+1+3; 1+1+2+2; 1+1+4; 1+2+3; 1+5; 2+2+2; 2+4; 3+3; 6;

- **Reprezentarea soluției:** $x = (x_1, x_2, \ldots, x_k)$ (stivă de lungime variabilă!), unde $x_i \in \{1,2,\ldots,n\}$
- **Condiții interne (finale)**

  ◦ $x_1 + x_2 + \ldots + x_k = n$

  ◦ pentru unicitate: $x_1 \leq x_2 \leq \ldots \leq xk_k$

- **Condiții de continuare**

  ◦ $x_{k-1} \leq x_k$ (avem $x_k \in \{x_{k-1},\ldots,n\}$)

  ◦ $x_1 + x_2 + \ldots + x_k \leq n$

```pascal
begin
 k:=1; s:=0;
 while(k>=1) do
  begin
  if(x[k]<n) then
    begin
     x[k]:=x[k]+1; s:=s+1;
     if (s<=n)
       then begin
         if(s=n) then begin
                   tipar(x,k);
                   s:=s-x[k];
                   k:=k-1;
                   end
           else begin
                k:=k+1;
                x[k]:=x[k-1]-1;
                s:=s+x[k];
                end;
       end
     else begin
          s:=s-x[k];
          k:=k-1;
          end;
    end;
  end;
end;
```

```c
void back(){
 int k=0, s=0;
 while(k>=0){
  if(x[k]<n){
    x[k]++; s++;
    if(s<=n) { //cond.cont.
      if(s==n){ // solutie
        tipar(x,k);
        s=s-x[k]; k--;
         //revenire
      }
      else
       { //avansare
        k++;
        x[k]=x[k-1]-1;
        s+=x[k];
       }
    }
   else {//revenire
     s=s-x[k]; k--;
   }
  }
 }
}
```

Concluzie

Metoda backtracking se aplică problemelor în care soluția se poate prezenta sub forma unui vector $x=\{x_1, x_2, \ldots, x_n\}$ unde x1 aparține unei mulțimi S1, x2 aparține mulțimii S2 s.a.m.d. Cerința problemei este, de obicei, găsirea tuturor soluțiilor posibile sau găsirea numărului de soluții care satisfac anumite conditii specifice problemei. De multe ori metoda se foloseste si pentru gasirea unei singure solutii (dupa gasirea acesteia se intrerupe executia programului), a unei solutii maxime/minime.

Metoda de generare a tuturor solutiilor posibile si apoi de determinare a solutiilor rezultat prin verificarea îndeplinirii conditiilor interne necesita foarte mult timp.Metoda backtracking evita aceasta generare si este mai eficienta. Metoda backtracking urmăreşte evitarea generării tuturor solutiilor posibile, micşorându-se astfel complexitatea algoritmului şi scurtându-se timpul de executie. Trebuie precizat de la bun început că la nici un caz (exceptând eventual câteva cazuri particulare) nu vom reduce complexitatea algoritmului de la una exponentială la una polinomială, ci doar la una mai putin exponentială (de exemplu, cum am spus şi mai sus, de la un 3^n la un 2^n).

Tipuri de probleme ce pot fi rezolvate cu această metodă:

1. Generarea permutărilor, aranjamentelor, combinarilor
2. Problema celor n regine
3. Colorarea hărtilor
4. Siruri de paranteze ce se închid corect
5. Partitiile unui număr natural
6. Generarea unor numere cu proprietăti specificate
7. Generarea produsului cartezian
8. Generarea tuturor solutiilor unei probleme, pentru a alege dintre acestea o solutie care minimizează sau maximizeaza o expresie

Bibliografie

- http://www.scritub.com/stiinta/informatica/METODA-BACKTRACKING1055131414.php
- https://www.slideshare.net/BalanVeronica/catalinametoda-relurii
- https://sites.google.com/site/matriciinformaticapascal/home/metoda-backtracking
- http://www.lbi.ro/~carmen/vineri/teorie_backtraking.pdf?fbclid=IwAR1iEmemm7YAir0wLwbQQXHjUEVOdMWoTM5Wo8hTq5lTSt6qNrQTlxZUXxA
- http://fmi.unibuc.ro/ro/pdf/2017/admitere/licenta/FMI_Backtracking_2017.pdf?fbclid=IwAR0AXHuKy7G-01428sOGiSc0toi1VX9UNoWgYGg8vIhcnaVNqes5SIzXwrQ