Metoda Backtracking

• Generarea elementelor unui produs cartezian

Fie $X=X_0 \times \ldots \times X_{n-1}$ cu $|X_i|=s_i$. Atunci, printr-o bijecţie, putem presupune că $X_i=\{0,1,2,\ldots,s_{i-1}\}$. Rezultă că $|X| = s_0 \times s_1 \times \ldots \times s_{n-1}$.

Algoritmul de generare, în ordine lexicografică, a elementelor lui X este:

```
k←0; xi←-1, ∀i=0,...,n-1
while k>=0
   if k=n
   then prel_sol(x); k←k-1;
   else if xk<sk-1
        then xk←xk+1; k←k+1
        else xk←-1; k←k-1;
```

k	x_0	x_1	x_2

Complexitatea în timp a algoritmilor joacă un rol esenţial. Un algoritm este considerat "acceptabil" numai dacă timpul său de executare este polinomial, adică de ordinul $O(n^k)$ pentru un anumit k; n reprezintă numărul datelor de intrare.

Pentru a ne convinge de acest lucru, vom considera un calculator vechi, capabil să efectueze doar un milion de operaţii pe secundă, o secundă corespunzând prelucrării unei soluţii.

	n=20	n=40	n=60
n^3	–	–	0,2 sec
2^n	1 sec	12,7 zile	366 secole
3^n	58 min	3855 secole	10^{13} secole

Deci pentru $|X_i|=s=3$ pentru orice i, şi n=60, timpul va fi inacceptabil chiar dacă viteza calculatoarelor creşte (de exemplu) de un milion de ori!

Într-adevăr, tabelul de mai sus arată că algoritmii exponenţiali nu sunt acceptabili. Aceasta chiar dacă în 2007 cel mai rapid supercomputer din lume era Blue Gene, lansat de IBM, care foloseşte 131.000 de procesoare şi poate efectua 280 de trilioane de operaţii pe secundă, adică în prezent calculatoarele performante sunt capabile de a efectua zeci de miliarde de operaţii pe secundă, deci este de aproximativ 3×10^8 ori mai puternic.

• Descrierea metodei

Fie $X=X_0 \times \ldots \times X_{n-1}$. Caut $x \in X$ cu $\varphi(x)$, unde $\varphi: X \to \{0,1\}$ este o proprietate definită pe X.

Din cele de mai sus rezultă că generarea tuturor elementelor produsului cartezian X nu este acceptabilă.

Metoda backtracking încearcă, dar nu reuşeşte totdeauna, micşorarea timpului de calcul. X este numit *spaţiul soluţiilor posibile*, iar φ sintetizează *condiţiile interne*, numite şi *condiţii de continuare*.

Vectorul x este construit progresiv, începând cu prima componentă.

Se ajunge la poziţia k numai dacă $x_0,...,x_{k-1}$ au valori necontradictorii (relativ la proprietatea φ). Se creşte cu o unitate valoarea curentă a x_k creşte cu o unitate numai dacă este satisfăcută *condiţia de continuare* $\varphi_k(x_0,...,x_k)$ prin care se verifică dacă x_k „se înţelege bine" (relativ la această condiţie) cu $x_0,...,x_{k-1}$.

Condiţiile de continuare rezultă de obicei din φ. *Ele sunt strict necesare, ideal fiind să fie şi suficiente.*

Distingem următoarele cazuri posibile la alegerea lui x_k:

1) *"Atribuie şi avansează"*: mai sunt valori neanalizate din X_k şi noua valoare x_k aleasă satisface $\varphi_k \Rightarrow$ se măreşte k.
2) *"Încercare eşuată"*: mai sunt valori neanalizate din X_k şi noua valoarea x_k aleasă nu satisface $\varphi_k \Rightarrow$ se va relua pentru acelaşi k, încercându-se alegerea unei noi valori pentru x_k.
3) *"Revenire"*: nu mai există valori neanalizate din X_k (X_k epuizată) $\Rightarrow$ întreaga X_k devine disponibilă şi $k \leftarrow k-1$.
4) *"Revenire după determinarea unei soluţii"*: este reţinută soluţia.

Prelucrarea unei soluţii constă în apelarea unei proceduri `prel_sol` care prelucrează soluţia (o tipăreşte, o compară cu alte soluţii etc.) şi fie opreşte procesul (dacă se doreşte o singură soluţie), fie prevede $k \leftarrow k-1$ (dacă dorim să determinăm toate soluţiile.

```
k←0; xi←0, ∀i=0,...,n-1
while k>=0
   if k=n
   then prel_sol(x); k←k-1;     { revenire după obţinerea unei soluţii }
   else if xk<sk-1
        then xk←xk+1;
             if φk(x0,…,xk)
             then k←k+1;         { atribuie şi avansează }
             else                { încercare eşuată }
        else xk←-1; k←k-1;       { revenire }
```

- **Exemple**

În exemplele care urmează, φ_k va fi notată în continuare prin `cont(k)`. Se aplică algoritmul de mai sus pentru diferite forme ale funcţiei de continuare.

1) Problema celor n dame

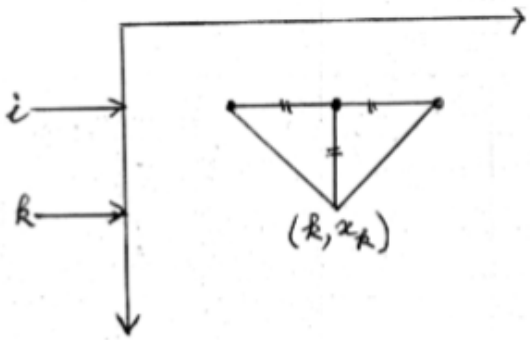
Se consideră un caroiaj $n \times n$. Prin analogie cu o tablă de şah (n=8), se doreşte plasarea a n dame pe pătrăţelele caroiajului, astfel încât să nu existe două dame una în bătaia celeilalte (adică să nu existe două dame pe aceeaşi linie, coloană sau diagonală).

Evident, pe fiecare linie vom plasa exact o damă. Fie x_k coloana pe care este plasată dama de pe linia k.

Damele de pe liniile i şi k sunt în conflict dacă:

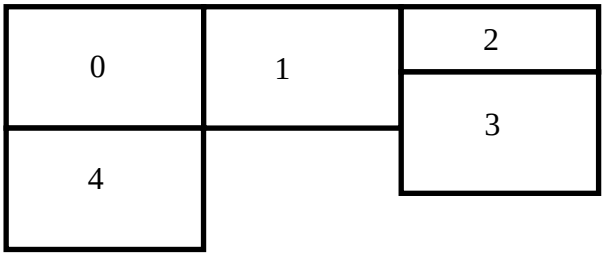
- pe aceeaşi coloană: dacă $x_i=x_k$;
- pe aceeaşi diagonală: dacă $|x_i-x_k|=k-i$.

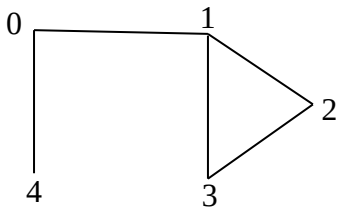
```
function cont(k)
   i ← 0;
   while i<k
      if |xi-xk|=k-i or xi=xk
      then return false
      else i ← i+1;
   return true
end;
```



2) Colorarea hărţilor. Se consideră o hartă. Se cere colorarea ei folosind cel mult `s` culori, astfel încât oricare două ţări vecine (cu frontieră comună de lungime strict pozitivă) să fie colorate diferit. Culorile disponibile sunt `0,1,..,s-1`.

Fie x_k culoarea curentă cu care este colorată ţara `k`. Trebuie ca nicio ţară vecină, din cele precedente, să nu fie colorată la fel.





0	1	0	0	1
	0	1	1	0
		0	1	0
			0	0
				0

```
function cont(k)
   i ← 0;
   while i<k
      if vecin(i,k) & xi=xk
      then return false
      else i ← i+1
   return true
end;
```

3) Problema ciclului hamiltonian

Se consideră un graf neorientat. Un ciclu hamiltonian este un ciclu care trece exact o dată prin fiecare vârf al grafului.

Pentru orice ciclu hamiltonian putem presupune că el pleacă din vârful 0. Fie x_i vârful cu indicele i din ciclu.

Observăm că $x=(x_0,...,x_{n-1})$ este *soluţie* dacă:

$x_0=0$

$\{x_1,...,x_{n-1}\}=\{1,...,n-1\}$ &

x_{k-1},x_k vecine, $\forall\ k=1,...,n-2$

x_{n-1},x_0 vecine.

Vom considera că graful este dat prin matricea sa de adiacenţă.

```
function cont(k)
   if a(x_{k-1},x_k)=0
   then return false
   else i ← 0;
        while i<k
          if x_k=x_i then return false
                   else i ← i+1;
        if k=n-1 then return a(x_{n-1},x_0)=1;
end;
```

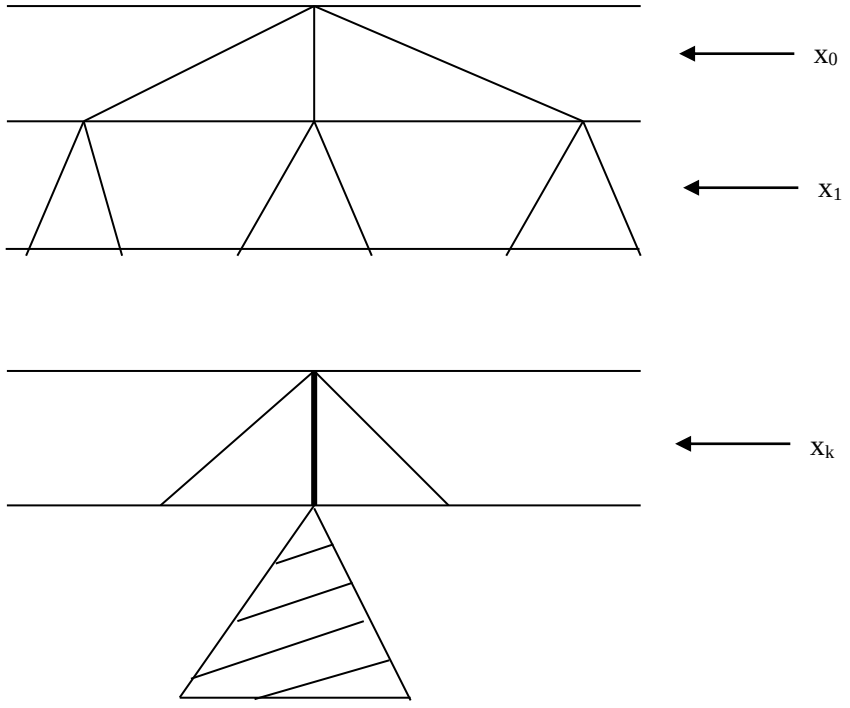
- **O descriere succintă a metodei backtracking**

Metoda backtracking poate fi descrisă astfel:

Backtracking = parcurgerea limitată *) în adâncime a unui arbore

*) conform condiţiilor de continuare

Rolul condiţiilor de continuare este ilustrat în figura ce urmează. Dacă pentru x_k este aleasă o valoare ce nu satisface condiţiile de continuare, atunci la parcurgerea în adâncime este evitată parcurgerea unui întreg subarbore.



- **Variante**

Variantele cele mai uzuale întâlnite în aplicarea metodei backtracking sunt următoarele:

- soluţia poate avea un număr variabil de componente *şi/sau*
- dintre ele alegem una care optimizează o funcţie dată.

Exemplu. $a=(a_1,...,a_n)\in \mathbf{Z}^n$. Caut un subşir crescător de lungime maximă.

Deci caut $1\le x_1<...<x_k\le n$ cu $\begin{cases} k \text{ maxim} \\ a_{x1} < a_{x2} <...< a_{xk} \end{cases}$

Pentru $n=8$ şi $a=(1,4,2,3,7,5,8,6)$ va rezulta $k=5$.

Aici, *soluţia posibilă*: care nu poate fi continuată. Exemplu: $(4,7,8)$.

Notăm prin `xf` şi `kf` soluţia optimă şi lungimea sa.

- Completez cu $-\infty$ şi $+\infty$: $a_0 \leftarrow -\infty$; $n\leftarrow n+1$; $a_n \leftarrow +\infty$;

- Funcţia `cont` are următoarea formă:

```
function cont(k)
  cont ← a(x[k-1])<a(x[k])
end;
```

- Procedura `prel_sol` are forma:

```
procedure prel_sol(k)
   if k>kf then xf←x; kf←k;
end;
```

- Algoritmul backtracking se modifică astfel:

```
k←1; x0←0; x1←0; kf←0;
while k>0
      if xk<n
      then  xk←xk+1;
            if cont(k)
            then if xk=n                { an=+∞ }
                 then prel_sol(k); k←k-1
                 else k←k+1; xk←xk-1
            else
      else k←k-1;
```

Observaţie. Se face tot o parcurgere limitată în adâncime a unui arbore.

- **Varianta recursivă**

Descriem varianta recursivă pentru $X_0=...=X_{n-1}=\{0,...,s-1\}$.
Apelul iniţial este: `back(0)`.

```
procedure back(k)
   if k=n
   then prel_sol(x)
   else for i=0,s-1
           x_k←i;
           if cont(k)
           then back(k+1); revenire din recursivitate
           else
end.
```

Exemplu. Dorim să producem toate şirurile de `n` paranteze ce se închid corect.

Există soluţii ⇔ `n` par.
Fie `dif=nr( - nr)`.
Condiţiile sunt :

`0 ≤ dif ≤ n-k` pentru orice `0 ≤ k ≤ n-2`; (*)
`dif=0` pentru `k=n-1` .

Pornirea algoritmului backtracking se face prin:

```
a_0←'( '; dif←1; back(1);
```

Procedura `back` are următoarea formă:

```
procedure back(k)
   if k=n
   then prel_sol        {scrie soluţia}
   else a_k←'( '; dif++;
              if dif ≤ n-k then back(k+1)
              dif--;
        a_k←')'; dif--;
              if dif≥0 then back(k+1)
              dif++;
end.
```

Observaţie. Condiţiile (*) ne asigură că secvenţa iniţială curentă se poate completa la o soluţie astfel:

- continuăm cu `n-k` paranteze închise;
- pentru noua secvenţă iniţială avem `dif`=0, deci lungimea ei este un număr par;
- poziţiile următoare (tot în număr par) pot fi completate (de exemplu) cu perechi „`()`” pentru a ajunge la o soluţie.

În concluzie, condiţiile de continuare nu sunt numai necesare, dar şi suficiente. Ca urmare **în acest caz backtracking-ul este optimal!**

• Metoda backtracking în plan

Se consideră un caroiaj (matrice) `A` cu `m` linii şi `n` coloane. Poziţile pot fi:

- libere: $a_{ij}=0$;
- ocupate: $a_{ij}=1$.

Se mai dă o poziţie (i_0,j_0). Se caută toate drumurile care ies în afara matricii, trecând numai prin poziţii libere.

- Mişcările posibile sunt date printr-o matrice `depl` cu două linii şi `ndepl` coloane. De exemplu, dacă deplasările permise sunt cele către poziţiile vecine situate la Est, Nord, Vest şi Sud, matricea are forma:
  $\begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix}$
- Bordez matricea cu 2 pentru a nu studia separat ieşirea din matrice.
- Pentru refacerea drumurilor, pentru fiecare poziţie atinsă memorăm legătura la precedenta.
- Dacă poziţia e liberă şi pot continua, pun $a_{ij}=-1$ (a fost atinsă), continuăm şi apoi repunem $a_{ij} \leftarrow 0$ (întoarcere din recursivitate).

Programul în Java are următoarea formă:

```
import java.util.*;

class elem {
  int i,j; elem prec;
  static int m,n,i0,j0,ndepl=4;
  static int[][] mat;
  static int[][] depl = { {1,0,-1,0}, {0,-1,0,1} };

  elem() {
     int i,j;
     Scanner sc = new Scanner(System.in);
     System.out.print("m,n = "); m = sc.nextInt();
     n = sc.nextInt();     // de fapt m+2,n+2
     mat = new int[m][n];
     for(i=1; i<m-1; i++)
        for(j=1; j<n-1; j++) mat[i][j] = sc.nextInt();
     for (i=0; i<n; i++) {mat[0][i] = 2; mat[m-1][i] = 2; }
     for (j=0; j<m; j++) {mat[j][0] = 2; mat[j][n-1] = 2; }
     System.out.print("i0,j0 = ");
        i0 = sc.nextInt(); j0 = sc.nextInt();
  }

  elem(int ii, int jj, elem x) { i = ii; j = jj; prec = x; }

  String print() {
     if (prec == null) return "(" + i + "," + j + ")";
     else return prec.print() + " " + "(" + i + "," + j + ")";
  }

  void back() { // backtracking pentru celula curenta
    elem x; int ii,jj;
    for (int k=0; k<ndepl; k++) {
      ii = i+depl[0][k]; jj = j+depl[1][k];
      if (mat[ii][jj] == 1);
      else if (mat[ii][jj] == 2)
        System.out.println(print());
      else if (mat[ii][jj] == 0) {
        mat[i][j] = -1;
        x = new elem(ii,jj,this); x.back();
        mat[i][j] = 0;
      }
    }
  }
}
```

```
class DrumPlan {
  public static void main(String[] args) {
    new elem(); elem start = new elem(elem.i0,elem.j0,null);
    start.back();
  }
}
```

Exemplu de test. Se introduc:

```
m,n = 5 6
1 1 0 1
0 0 0 1
1 0 0 1
i0,j0 = 2 2
```

Variante:

- cum pot ajunge într-o poziţie (i_1, j_1) dată?
- se cere determinarea componentelor conexe.