BACKTRACKING
Dimitriu Gabriel gr 21

I)Prezentarea tehnicii backtracking

Tehnica backtracking se foloseste in rezolvarea problemelor care indeplinesc simultan conditiile:

- Solutia poate fi codificata prin vectori cu n elemente, $S=(x_1,x_2,..,x_n)$ cu $x_i \in A_i$, pentru i=1,n;
- multimile $A_1,A_2,...,A_n$ sunt multimi finite, iar elementele lor pot fi ordonate dupa o relatie de ordine. Pentru fiecare A_i trebuie sa se poata preciza primul element, succesiunea elementelor si ultimul element
- nu se dispune de o alta metoda de rezolvare, mai rapida sau se doreste baleierea tuturor solutiilor

Principiul care sta la baza algoritmului backtracking este:

- se construieste solutia pas cu pas, generind pe rind elementele vectorului S, in ordine.
- daca se constata ca, pentru o valoare aleasa corespunzatoare pozitiei k, nu se poate ajunge la solutie, se renunta la acea valoare si se continua cautarea din punctul in ca care am ramas (se evita generarea a cit mai multe elemente din produsul cartezian $A_1xA_2x...xA_n$).

Multimea $A_1xA_2x...xA_n$ se numeste spatiul solutiilor. In fiecare problema care se rezolva cu ajutorul tehnicii backtracking trebuie sa se precizeze anumite realtii intre elementele $(x_1,x_2,..,x_n)$ si contii pentru x_i, i=1,n, aceastea formind conditiile interne. Solutiile ce satisfac conditiile interne se numesc solutii rezultat.

Fie multimea $C_k \subset A_1xA_2x...xA_k$, a elementelor de forma $(x_1,x_2,..,x_k)$ care satisfac anumite conditii interne, deci sunt numite solutii rezultat.

Fie $(x_1,x_2,..,x_{k-1})$ componentele solutiei determina pina la pasul k-1. Pentru a determina componenta $x_k \in A_k$, cautam in A_k primul element disponibil care satisface conditiile interne. Daca un astfel de element exista se selecteaza si se continua cu pozitia k+1. Daca nu mai exista nici un element din multimea A_k astfel incit sa poata fi determinat un element al multimii C_k atunci se revine la poztia k-1 (considerind generate elementele $x_1,x_2,..,x_{k-2}$ si cautam o noua posibilitate pentru k-1).

Versiuna iterativa a programului este cea prezentata de Grigore Albeanu in “Algoritmi si Limbaje de Programare”:

```
procedure back(n:integer);
var s:stack_Array;x:Tip_element;
   k:integer; inainte,found: boolean;
begin
k:=1;s[1]:=init(1);
while(k>0) do
   begin
   repeat
         inainte:=false;succesor(k,x,found);
         if found then
               begin
               s[k]:=x;
               if Conditii_interne(k) then inainte:=true
               end;
   until inainte or (not found);
```

```
   if( not found) then k:=k-1
   else if k=n then Final
            else begin k:=k+1;s[k]:=init(k) end
   end;
end;
```

In care functiile care se particularizeaza dupa aplicatie sunt urmatoare:

- init(k): funtie care returneaza o valoare ce va permite identificarea primului element al multimii A_k.
- Succesor(jk,x,ind): procedura care atribuie variabilei ind valoarea logica true si in x depune urmatoarul element din multimea A_k, in caz ca un astfel de elemente exista; respectiv atribuie variabilei ind valaorea logica falsi si x este nedefinit atunci cind au fost deja investigate toate elementele multimii A_k.
- Conditii_interne(k): functie care returneaza true cind elementul plasat pe pozitia k verifica conditiile interne; respectiv false, in caz contrar.
- Final(): subprogram pentru postprocesarea solutiei rezultat.

II)Aplicatii la tehnica backtracking:

Problema 1. Se da n:1..100. Sa se tipareasca toate posibilitatile de a fi asezate pe o table de sa, de dimensiune nxn, n dame care sa nu se atace reciproc.

Aceasta problema se poate rezolva prin metoda backtracking daca fiecare element din vector inseamna pozitia pe coloana a damei de pe rindul k, in cazul matrice[k] si final[k]; Pentru aceasta implementare vom genera clasa backtracking:

```
class backtracking
{
public:
   backtracking();
   backtracking(int nr);
   virtual ~backtracking();
private:
   bool solutie(int k);
   int * final;
   int init(int k);
   void calcul(int k);
   int backup;
   bool conditii_interne(int k);
   int nr_dame;
   bool succesor(int *a,int k);
   void Final(void);
};
```

In care functia de calcul este functia principala care implementeaza tehica backtraking:

```
void backtracking::calcul(int k)
{
   bool found,inainte;
   k=0;
   backup=init(0);
   while(k>=0)
   {
            while(1)
```

```
        {
                found=succesor(&backup,k);
                inainte=false;
                if(found) inainte=conditii_interne(k);
                if(inainte) final[k]=backup;
                if(inainte || !found) break;
        }
        if(found)
        {
                if(solutie(k))  Final();
                else { k++; backup=init(k); }
        }
        else { k--; backup=final[k]; }
  }
}
```

Se observa ca s-au facut modificari fata de algoritmul prezentat in sectiunea anterioara si aceasta deoarece nu exista in C –C++ implementarea constructiei repeat until si in momentul in care se gereaza valoarea pentru k in cazul in care nu este buna ea trebuie schimbata.

Implementarea constructorilor si desstructorilor clasei:

```
backtracking::backtracking()
{
  cout<<"Introduceti numarul de dame=";
  cin>>nr_dame;
  final=new int[nr_dame];
  for(int i=0;i<nr_dame;i++) final[i]=0;
  calcul(0);
}

backtracking::backtracking(int nr)
{
  nr_dame=nr;
  final=new int[nr_dame];
  for(int i=0;i<nr_dame;i++) final[i]=0;
  calcul(0);
}

backtracking::~backtracking()
{
  delete final;
}
```

Implementarile functiilor specifice sunt:

```
bool backtracking::succesor(int *a,int k)
{
  if((*a)<nr_dame-1) { (*a)++; return true; }
  else return false;
}

bool backtracking::conditii_interne(int k)
{
  int i;
  bool OK;
  i=0;
  OK=true;
  while(i<k && OK)
```

```
            if((backup==final[i]) || (abs(k-i)==abs(final[i]-backup))) OK=false;
            else i++;
   return OK;
}
int backtracking::init(int k)
{
   return -1;
}
bool backtracking::solutie(int k)
{
   if(k==nr_dame-1) return true;
   else return false;
}
void backtracking::Final()
{
   int i,j,**mat,*tmat;
   cout<<"Saving data\n";
   ofstream fp("dame.dat",ios::app);
   if(!fp) { cout<<"Eroare la deschidere fisier"; exit(-1); }
   mat=(int **)calloc(nr_dame,sizeof(int *));
   tmat=(int *)calloc(nr_dame*nr_dame,sizeof(int));
   for(i=0;i<nr_dame;i++) { mat[i]=tmat; tmat+=nr_dame; }
   for(i=0;i<nr_dame;i++) mat[i][final[i]]=1;
   for(i=0;i<nr_dame;i++) { for(j=0;j<nr_dame;j++) fp<<mat[i][j]<<' '; fp<<endl; }
   fp<<endl;
   free(*mat);
   free(mat);
   fp.close();
}
```

Problema 2. Se dau n, K_0:1..100, cu $K_0 \leq n$. Sa se tipareasca si sa se numere toate posibilitatile de a fi aranjate pe o tabla de sah de dimensiune nxn, minim /maxim K_0 dame care sa nu se atace reciproc.

Aceasta problema este similara cu problema anterioara doar ca se face pentru un K_0 diferit de fiecare data. Implementarea mai are o mica diferenta si anume : se lucreaza cu o tabla de nxn+1 in care daca dama spune ca se afla pe pozitia n+1 (in C++ pe pozitia n) inseamna ca acea linie este goala deoarece nu avem n+1 coloane.

Enunþul sintetizeazã de fapt douã probleme, ambele rezolvabile cu metoda backtracking :

1: - aranjarea a minim K_0 dame pe o tablã de dimensiune nxn
2: - aranjarea a maxim K_0 dame pe o tablã de dimensiune nxn

Asadar clasa va fi:

```
class backtracking
{
public:
   backtracking();
   backtracking(int nr,int nr1);
   virtual ~backtracking();
private:
   int m_p;
   int k0;
   bool solutie(int k);
```

```
   bool first;
   int * final;
   int init(int k);
   void calcul(int k);
   int * matrice;
   bool conditii_interne(int k);
   int nr_dame;
   bool succesor(int *a,int k);
   void Final(void);
};
```
Implementarea constructorilor clasei:
```
backtracking::backtracking()
{
   cout<<"Introduceti dimensiunea tablei=";cout.flush();
   cin>>nr_dame;
   cout<<"Introduceti pragul ";cout.flush();
   cin>>k0;
   matrice=new int[nr_dame];
   final=new int[nr_dame];
   for(int i=0;i<nr_dame;i++) { final[i]=nr_dame; matrice[i]=0; }
   m_p=1; //pentru prima problema
   m_p=k0;//pentru a doua problema
   ofstream fp("dame.dat",ios::trunc);
   fp.close();
   fp.flush();
   calcul(0);
}

backtracking::backtracking(int nr,int nr1)
{
   nr_dame=nr;
   k0=nr1;
   matrice=new int[nr_dame];
   final=new int[nr_dame];
   for(int i=0;i<nr_dame;i++) { final[i]=nr_dame; matrice[i]=0; }
   m_p=1; //pentru prima problema
   m_p=k0;//pentru a doua problema
   ofstream fp("dame.dat",ios::trunc);
   fp.close();
   fp.flush();
   calcul(0);
}

backtracking::~backtracking()
{
   delete matrice;
   delete final;
}
```
Implementarea functiei de calcul (functia principala) de backtraking:
```
void backtracking::calcul(int k)
{
   bool found,inainte;
   while(m_p<=k0) //pentru prima problema
   while(m_p>=k0 && m_p<nr_dame) //pentru a doua problema
   {
            k=0;
```

```
        for(int i=0;i<nr_dame;i++) { final[i]=nr_dame; matrice[i]=0; }
        first=true;
        matrice[0]=init(0);
        while(k>=0)
        {
                while(1)
                {
                        found=succesor(&matrice[k],k);
                        first=false;
                        inainte=false;
                        if(found) inainte=conditii_interne(k);
                        if(inainte) final[k]=matrice[k];
                        if(inainte || !found) break;
                }
                if(found)
                {
                        if(solutie(k))
                        {
                                Final();
                                m_p++;
                                break;
                        }
                        else { k++; matrice[k]=init(k); }
                }
                else { matrice[k]=final[k]; k--; }
        }
   }
}
```

Implementarea functiilor specifice aplicatiei:

```
bool backtracking::succesor(int *a,int k)
{
   if(k==0 && first) return true;
   if((*a)<nr_dame) { (*a)++; return true; }
   else return false;
}

bool backtracking::conditii_interne(int k)
{
   int i;
   bool OK;
   i=0;
   OK=true;
   while(i<nr_dame && OK)
   {
        if(final[i]==nr_dame || i==k) i++;
        else if((matrice[k]==final[i]) || (abs(k-i)==abs(final[i]-matrice[k]))) OK=false;
                else i++;
   }
   return OK;
}

int backtracking::init(int k)
{
   return 0;
}
```

```
void backtracking::Final()
{
   int i,j,**mat,*tmat;
   cout<<"Saving data\n";cout.flush();
   ofstream fp("dame.dat",ios::app);
   if(!fp) { cout<<"Eroare la deschidere fisier"; exit(-1); }
   mat=(int **)calloc(nr_dame,sizeof(int *));
   tmat=(int *)calloc(nr_dame*nr_dame,sizeof(int));
   for(i=0;i<nr_dame;i++) { mat[i]=tmat; tmat+=nr_dame; }
   for(i=0;i<nr_dame;i++) if(final[i]!=nr_dame) mat[i][final[i]]=1;
   fp<<"Solutia pentru "<<m_p<<"dame"<<endl;fp.flush();
   for(i=0;i<nr_dame;i++)
   {
         for(j=0;j<nr_dame;j++) fp<<mat[i][j]<<' ';
         fp<<final[i]<<endl;
   }
   fp.flush();
   fp.close();
   fp.flush();
   free(*mat);
   free(mat);
}

bool backtracking::solutie(int k)
{
   int nrdame;
   nrdame=0;
   for(int i=0; i<nr_dame;i++) if(final[i]!=nr_dame) nrdame++;
   if(nrdame==m_p) return true;
   else if(nrdame<m_p) return false;
   else { cout<<"Ciudat!!"; cout.flush(); return false; }
}
```

Problema 3. Se dau n,p:1..100, p≤n. Sa se genereze toate posibilitatile de a alege p elemente din {1,2,...,n} formind multimi ordonate de p elemente.

Observatii:

- Din enunt deducem ca doua multimi cu aceleasi elemente dar la care ordinea acestora difera, sunt considerate distince.
- Pentru cazul p=n se obtin permutarile multimii {1,2,...,n}, deci n! posibilitati.
- Numarul posibilitatilor cerute este A_n^p, adica numarul aplicatiilor injective definite pe {1,2,...,p} cu valori in {1,2,...,n}.
- Deoarece stiim ca fiecare valoare va aparea de A_n^{p-1} daca p este numarul de elemente din aranjament deci daca generam prima cifra de numarul de ori necesar vom avea un sisteme in care sa avem incredere in prima valoare.

Ca in problema precedenta clasa va fi:

```
class backtracking
{
public:
   backtracking();
   virtual ~backtracking();
private:
   int val_partial;
   int backup;
```

```
   int nr_reusite;
   int contor;
   int nr_combinatii;
   int aranjamente(int n,int p);
   int Nr_elem;
   bool solutie(int k);
   bool first;
   int ** final;
   int init(int k);
   void calcul(int k);
   int * matrice;
   bool conditii_interne(int k);
   int nr_max;
   bool succesor(int *a);
   void Final(void);
};
```

In care functiile au aceeasi specificatie ca inainte:

```
backtracking::backtracking()
{
   cout<<"Introduceti dimensiunea permutarii=";
   cin>>nr_max;
   cout<<"Introduceti dimensiunea aranjementelor=";
   cin>>Nr_elem;
   nr_combinatii=aranjamente(nr_max,Nr_elem);
   val_partial=aranjamente(nr_max-1,Nr_elem-1);
   matrice=new int[nr_max];
   int *temp;
   final=(int **)calloc(nr_combinatii,sizeof(int *));
   temp=(int *)calloc(nr_combinatii*Nr_elem,sizeof(int));
   for(int i=0;i<nr_combinatii;i++) { final[i]=temp; temp+=Nr_elem; }
   for(i=0;i<nr_max;i++) matrice[i]=0;
   cout<<"total "<<nr_combinatii<<" partial "<<val_partial<<endl;cout.flush();
   calcul(0);
}

backtracking::~backtracking()
{
   free(final);
   delete matrice;
}

int backtracking::aranjamente(int n, int p)
{
   int nr1;
   int i;
   for(i=n,nr1=1;i>=(n-p+1);i--) nr1*=i;
   return nr1;
}

void backtracking::Final()
{
   int j;
   cout<<"Saving data\n";cout.flush();
   ofstream fp("data.dat",ios::app);
   if(!fp) { cout<<"Eroare la deschidere fisier"; exit(-1); }
   for(j=0;j<Nr_elem;j++) fp<<final[contor][j]<<' ';
```

```
   fp<<endl;
   fp.flush();
   fp.close();
}

int backtracking::init(int k)
{
   if(k==0)
   {
          while(1)
          {
                 if(matrice[nr_reusite]<val_partial) { matrice[nr_reusite]++; return nr_reusite; }
                 else
                 {
                        nr_reusite++;
                        if(nr_reusite==nr_max)
                        {
                               cout<<"Something is wrong!!!"<<endl;cout.flush();
                               for(int i=0;i<nr_max;i++) matrice[i]=0;
                               for(i=0;i<contor;i++) matrice[final[i][0]]++;
                               nr_reusite=0;
                        }
                 }
          }
   }
   else return 0;
}


bool backtracking::solutie(int k)
{
   if(k==Nr_elem)
   {
          int i,j;
          for(i=0;i<contor;i++)
          {
                 for(j=0;j<Nr_elem;j++) if(final[i][j]!=final[contor][j]) break;
                 if(j==Nr_elem) return false;
          }
          return true;
   }
   else return false;
}

bool backtracking::conditii_interne(int k)
{
   int i;
   for(i=0;i<k;i++) if(final[contor][i]==backup) return false;
   return true;
}

bool backtracking::succesor(int *a)
{
   if(first) return true;
   if((*a)<nr_max-1) { (*a)++; return true; }
   else return false;
```

```
}

void backtracking::calcul(int k)
{
   bool found,inainte;
   contor=0;
   nr_reusite=0;
   while(contor<nr_combinatii)
   {
          backup=init(0);
          first=true;
          final[contor][0]=backup;
          k=1;
          while(k>0)
          {
                 while(1)
                 {
                        found=succesor(&backup);
                        first=false;
                        inainte=false;
                        if(found) inainte=conditii_interne(k);
                        if(inainte)          final[contor][k]=backup;
                        if(inainte || !found) break;
                 }
                 if(found)
                 {
                        if(k<Nr_elem) { k++; backup=init(k); first=true; }
                        else
                        {
                               if(solutie(k))
                               {
                                      Final();
                                      contor++;
                                      if(contor==nr_combinatii) return;
                                      k=-1;
                               }
                               else
                               {
                                      k--;
                                      if(k==0) { k=1; backup=0; first=true; }
                                      else { backup=final[contor][k]; first=false; }
                               }
                        }
                 }
                 else
                 {
                        k--;
                        if(k==0) { k=1; backup=0; first=true; }
                        else { backup=final[contor][k]; first=false; }
                 }
          }
   }
}
```

Problema 4. Se dau n,p:1..100, p≤n. Sa se genereze toate submultimile cu p elemente ale multimii {1,2,...,n}. Doua submultimi se considera egale daca si numai daca au aceleasi elemente (nu conteaza ordinea elementelor in cadrul solutiei).

Solutia problemei este similara cu cea dinainte doar ca se modifica urmatoarele: aranjamente devin combinari, functia solutie, init si calcul se chimba deoare nu mai putem accelera procesul de convergenta prin initializare, de asemenea se scoate tot ce apartine de variabila matrice deoarece ea era utilizata la initializare.

Asadar functiile de initialiazare si solutie sunt :

```
int backtracking::init(int k)
{
  return 0;
}

bool backtracking::solutie(int k)
{
  if(k==Nr_elem)
  {
        int i,j,k;
        for(i=0;i<contor;i++)
        {
              int test;
              for(j=0,test=0;j<Nr_elem;j++)
              {
                    for(k=0;k<Nr_elem;k++) if(final[i][j]==final[contor][k]) test++;
              }
              if(test==Nr_elem) return false;
        }
        return true;
  }
  else return false;
}
```

Functia de calcul a combinarilor este:

```
int backtracking::combinari(int n, int p)
{
  int nr1,nr2;
  int i;
  for(i=n,nr1=1;i>=(n-p+1);i--) nr1*=i;;
  for(i=1,nr2=1;i<=p;i++) nr2*=i;
  return nr1/nr2;
}
```

In functia de calcul modificam urmatoarele: se elimina tot ceea ce este if(k==0) cu aferent, se inlocuieste in primul while conditia k>0 cu k>=0 se adauga inainte de while(k>=0) se pune first=true si se elimina k=-1 care este dupa Final();

Problema 5. Pe o sârmã se pot aºeza 9 bile ce au scrise câte un numãr natural între 1 ºi 9. Toate bilele au numere distincte. Dupã aºezarea bilelor (egale ca dimensiune), cu sârma se formeazã un triunghi echilateral, în vârfuri fiind cîte o bilã ºi pe laturi, câte douã bile, dupã cum se vede în desenul alãturat:

În câte moduri distincte se pot aºeza cele 9 bile pe poziþiile P1, P2,…,P9, figurate în

⊕ *P*1

*P*9 ⊕ ⊕ *P*2 desen, a.î. sumele corespunzãtoare laturilor
*P*8 ⊕ ⊕ *P*3 triunghiului sã fie egale? Mai exact, dacã s
*P*7 ⊕ ⊕ ⊕ ⊕ *P*4 este vectorul soluþie, atunci:
P6 P5

s[1]+s[2]+s[3]+s[4] = s[4]+s[5]+s[6]+s[7] =s[7]+s[8]+s[9]+s[1] (R1)

Definitia clasei este:

```
class backtracking
{
public:
   backtracking();
   virtual ~backtracking();
private:
   int backup;
   int nr_reusite;
   bool solutie(int k);
   int *final;
   int init(int k);
   void calcul(int k);
   bool conditii_interne(int k);
   bool succesor(int *a);
   void Final(void);
};
```

Rutina de calcul este identica cu cea de la problema 1.

Restul functiilor in afara de functia de Final in care se inlocuieste final[contor][i] cu final[i] si functia init care ramine aceeasi.

```
backtracking::backtracking()
{
   final=new int[9];
   for(int i=0;i<9;i++) final[i]=0;
   nr_reusite=0;
   calcul(0);
}

backtracking::~backtracking()
{
   delete final;
}

bool backtracking::solutie(int k)
{
   if(k==8)
   {
            int sx,so,sz;
            int i;
            for(i=0,so=0;i<4;i++) so+=final[i];
            for(i=3,sx=0;i<7;i++) sx+=final[i];
            for(i=6,sz=final[0];i<9;i++) sz+=final[i];
            if(so!=sx) return false;
            if(sz!=so) return false;
            nr_reusite++;
            cout<<nr_reusite;cout.flush();
            return true;
```

```
   }
   else return false;
}

bool backtracking::conditii_interne(int k)
{
   int i;
   for(i=0;i<k;i++) if(final[i]==backup) return false;
   return true;
}

bool backtracking::succesor(int *a)
{
   if((*a)<9)
   {
          (*a)++;
          return true;
   }
   else return false;
}
```

III) Backtracking generalizat

In cazul in care elementele lui A_i sunt vectori (cu doua, trei,... elemente) stiva in care se construieste solutia este o stiva dubla, tripla,...etc si problema se incadreaza in capitolul de backtracking generalizat.

Problema canibalilor si misionarilor

Se dau c,m: integer; $c,m \in N^*$. Pe malul apei se gasesc c canibali si m misionari. Ei doresc sa treaca apa folosind o barca cu doua locuri. In situatia in care, pe oricare mal, exista la un moment dat mai multi canibali decit misionari, misionarii sunt mincati. Sa se tipareasca toate variantele de trecere a apei astfel incit nici un misionar sa nu fie mincat.

Sa analizam problema!

Exista 5 modalitati de a traversa riul cu barca.

1.trec doi misionari;
2.trec doi canibali;
3.trec un misionar si un canibal;
4.trece un misionar;
5.trece un canibal;

Conform strategiei backtracking se va folosi o stiva care insa va avea drept elemete 5 vectori. Daca s[k] este elementul de pe pozitia k din stiva solutie atunci:
s[k][0] retine codul celei de a k-a traversare cu barca;
s[k][1] retine numarul de misionari de pe malul 1 la inceputul traversarii k;
s[k][2] retine numarul de canibali de pa malul 1 la inceputul traversarii k;
s[k][3] retine numarul de misionari de pe malul 2 la inceputul traversarii k;
s[k][4] retine numarul de canibali de pe malul 2 la inceputul traversarii k.

Vom conveni ca pentru k par barca sa traverseze riul de la malul 1 la malul 2 iar pentru k impar barca trece de la malul 2 la malul 1. O varianta de trecere a apei va consta in tiparirea, una sub alta, a mai multor configuratii retinute in s[0],s[1],..; in care s[0] este configuratia initiala este s[0]=(cond_traversare,m,c,0,0) iar configuratia finala va fi de forma (cod_traversare,0,0,m,c).

Formula de recurenta este:

-s[k+1][1] la inceput se initializeaza cu zero si eventul se modifica cu succesor
-s[k+1][i], se initilizeaza cu valorile calculate anterior, verificate cu conditii_interne ca fiind bune si transmise ca variabile globale in program: nc,nm1,nc2,nm2 cu semnificatia:

- nm1=numarul curent al misionarilor de pe malul 1
- nc1=numarul curent al canibalilor de pe malul 1
- nm2=numarul curent al misionarilor de pe malul 2
- nc2=numarul curent al canibalilor de pe malul 2

Declaratia clasei:

```
class backtracking
{
public:
   backtracking();
   virtual ~backtracking();
private:
   int backup;
   int nm1,nm2,nc1,nc2,m,c;
   bool solutie(int k);
   int final[100][5];
   void init(int k);
   void calcul(int k);
   bool conditii_interne(int k);
   bool succesor(int *a);
   void Final(int k);
};
```

Functia de calcul:

```
void backtracking::calcul(int k)
{
   bool found,inainte;
   k=0;
   while(k>=0)
   {
            while(1)
            {
                     found=succesor(&backup);
                     inainte=false;
                     if(found) inainte=conditii_interne(k);
                     if(inainte) final[k][0]=backup;
                     if(inainte || !found) break;
            }
            if(found)
            {
                     if(solutie(k)) Final(k);
                     else { k++; init(k);}
            }
            else { k--;backup=final[k][0];}
   }
}
```

Functiile de initializare si constructori:

```
void backtracking::init(int k)
{
```

```
   backup=-1;
   final[k][1]=nm1;
   final[k][2]=nc1;
   final[k][3]=nm2;
   final[k][4]=nc2;
}
backtracking::backtracking()
{
   cout<<"Misionari ";cout.flush();
   cin>>m;
   cout<<"Canibali ";cout.flush();
   cin>>c;
   final[0][0]=-1;
   backup=-1;
   final[0][1]=m;
   final[0][2]=c;
   final[0][3]=0;
   final[0][4]=0;
   calcul(0);
}

backtracking::~backtracking()
{
}
```

Functiile pentru conditii interne si solutie:

```
bool backtracking::solutie(int k)
{
   return ((nc1==0)&&(nm1==0));
}

bool backtracking::conditii_interne(int k)
{
   nm1=final[k][1];
   nc1=final[k][2];
   nm2=final[k][3];
   nc2=final[k][4];
   if (k%2==0)
   {
          switch(backup)
          {
      case 0:
            nm1=nm1-2;
            nm2=nm2+2;
            break;
      case 1:
            nc1=nc1-2;
            nc2=nc2+2;
            break;
          case 2:
                  nm1--;
                  nc1--;
                  nm2++;
                  nc2++;
                  break;
          case 3:
```

```
                nm1--;
                nm2++;
                break;
        case 4:
                nc1--;
                nc2++;
                break;
        }
  }
  else
  {
        switch(backup)
        {
        case 0:
                nm2=nm2-2;
                nm1=nm1+2;
                break;
        case 1:
                nc2=nc2-2;
                nc1=nc1+2;
                break;
        case 2:
                nm2--;
                nc2--;
                nm1++;
                nc1++;
                break;
        case 3:
                nm2--;
                nm1++;
                break;
        case 4:
                nc2--;
                nc1++;
                break;
        }
  }
  bool ev;
  ev=true;
  if((nm1<0 || nm2<0 || nc1<0 || nc2<0)
        || (nc1>nm1 && nm1>0) || (nc2>nm2 && nm2>0)) ev=false;
  if (k!=0) if(backup==final[k-1][0]) ev=false;
  for(int i=0;i<k;i++) if((final[i][1]==nm1) &&(final[i][2]==nc1)
        && ((k-i)%2==1)) ev=false;
  return ev;
}
```

Functia pentru succesor si functia de printare:

```
bool backtracking::succesor(int *a)
{
  if((*a)<4)
  {
        (*a)++;
        return true;
  }
```

```
   else return false;
}

void backtracking::Final(int k)
{
   int i;
   cout<<"Saving data\n";cout.flush();
   ofstream fp("data.dat",ios::app);
   if(!fp)
   {
            cout<<"Eroare la deschidere fisier";
            exit(-1);
   }
   for(i=0;i<=k;i++) fp<<final[i][1]<<' '<<final[i][2]<<' '<<final[i][3]<<' '<<final[i][4]<<endl;
   fp<<nm1<<' '<<nc1<<' '<<nm2<<' '<<nc2<<endl;
   fp.flush();
   fp.close();
}
```