Lucrarea 5. Backtracking.

Prezentarea Tehnicii Backtracking

Această tehnică poate fi utilizată pentru rezolvarea problemelor de căutare. Căutarea efectuată este exhaustivă, putând fi folosită pentru generarea tuturor soluţiilor posibile.

Se foloseşte în rezolvarea problemelor care îndeplinesc simultan următoarele condiţii:

- Soluţia lor poate fi pusă sub forma unui vector $V = x_1x_2x_3 \ldots\ldots x_n$, cu $x_1 \in A_1 \ldots. X_n \in A_n$
- Mulţimile $A_1, A_2, \ldots A_n$ sunt mulţimi finite, iar elementele lor se află într-o ordine bine stabilită
- Nu se dispune de o metodă de rezolvare mai rapidă

La întâlnirea unei astfel de probleme suntem tentaţi să generăm toate elementele produsului cartezian $A_1 \times A_2 \times \ldots. \times A_n$. Aceasta nu este totuşi o rezolvare foarte rezonabilă. Timpul de execuţie este atât de mare, încât poate fi considerat infinit.

Dacă de exemplu dorim să generăm toate permutările unei mulţimi, generând produsul cartezian, am putea obţine 1,1,1,….1, pentru ca apoi să constatăm că nu am obţinut o permutare (cifrele nu sunt distincte), deşi chiar de la a 2-a cifră se poate observa că cifrele nu sunt distincte.

Tehnica Backtracking are la bază un principiu extrem de simplu: se construieşte soluţia pas cu pas. Dacă se constată că pentru o anumită valoare nu se poate ajunge la soluţie, se renunţă la acea valoare şi se reia căutarea din punctul unde am rămas.

Prezentarea algoritmului

Pentru uşurarea înţelegerii metodei vom prezenta o rutină unică, aplicabilă oricărei probleme, rutină care utilizează noţiunea de stivă.

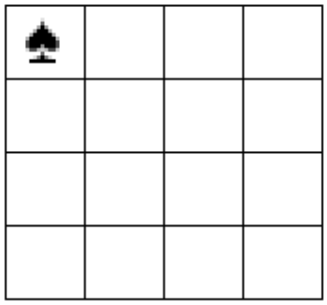
- se alege primul element x_1, ce aparţine lui A_1;
- presupunând generate elementele $x_1 \ldots x_k$, vom avea 2 posibilităţi:
  - nu s-a găsit un astfel de element, caz în care se reia căutarea, considerând generate elementele $x_1 \ldots x_{k-1}$, iar aceasta se reia de la următorul element A_k, rămas netestat.
  - a fost găsit, caz în care se testează dacă acesta îndeplineşte anumite condiţii (răspunsul poate fi verificat de o funcţie *Validare* care returnează true sau false).
  - dacă le îndeplineşte, se testează dacă s-a ajuns la soluţie (se utilizează funcţia *Soluţie*, care returnează true dacă s-a ajuns la soluţie)
    - dacă s-a ajuns la soluţie, se tipăreşte şi se continuă algoritmul pentru găsirea unei alte soluţii, considerându-se generate $x_1 \ldots x_k$, iar elementul x_{k+1} se caută printre elementele mulţimii A_{k+1} rămase netestate
    - dacă nu s-a ajuns la soluţie se reia algoritmul considerându-se generate elementele $x_1, \ldots x_k, x_{k+1}$
  - dacă nu le îndeplineşte se reia algoritmul considerând generate $x_1 \ldots x_k$, iar elementul x_{k+1} se caută printre elementele mulţimii A_{k+1} rămase netestate

Problema Reginelor.

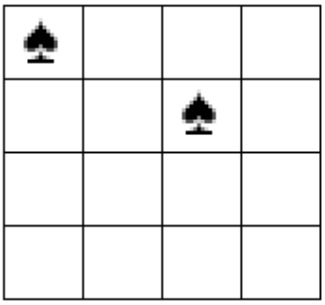
O să exemplificăm tehnica căutării cu reveniri cu ajutorul unei probleme clasice: cum putem amplasa N regine pe o tablă de şah cu dimensiuni N x N astfel încât acestea să nu se atace două câte două.

Este evident că nu putem amplasa decât o singură regină pe linie sau pe coloană şi în final fiecare linie va avea amplasată câte o regină. De aceea vom încerca amplasarea linie cu linie, începând cu prima linie. Atunci când încercăm să amplasăm o regină pe o linie începem cu prima coloană şi continuăm cu următoarele, până când găsim o poziţie validă. În momentul în care nu mai găsim nici o poziţie validă pe o linie vom reveni la linia anterioară, unde vom căuta următoarea poziţie. Să rezolvăm, de exemplu, problema pentru N = 4.

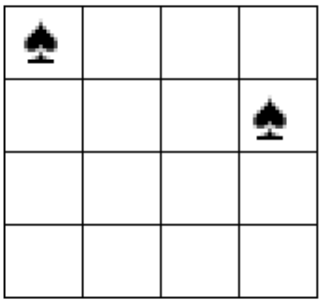
Începem cu prima linie şi prima coloană:



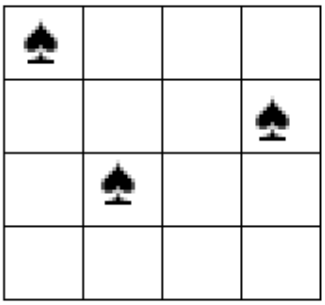
Pe cea de-a doua linie nu putem amplasa nici o damă până pe cea de-a treia coloană:



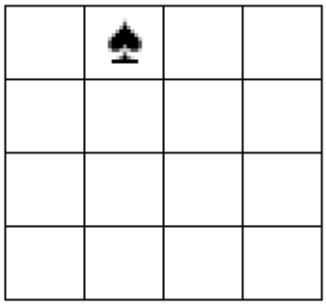
Pe linia a treia nu putem amplasa nici o damă, de aceea vom reveni la linia anterioară pentru următoarea opţiune:



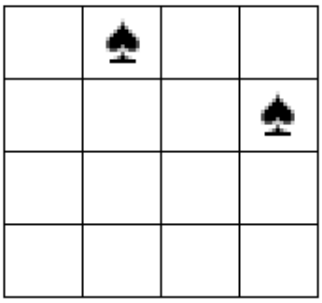
Pe linia a treia prima opţiune validă este pe coloana a doua:



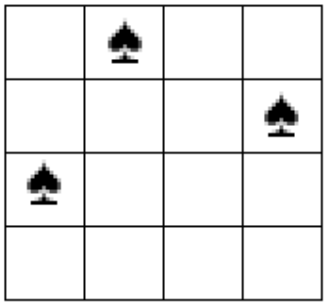
Dar acum nu mai avem nici o opţiune validă pentru linia a patra. Revenim la linia a treia. Dar nici aici nu mai avem nici o opţiune validă. De aceea trebuie să revenim la linia a doua. Este evident că aici nu mai avem ce face aşa că se revine până la linia a I-a:



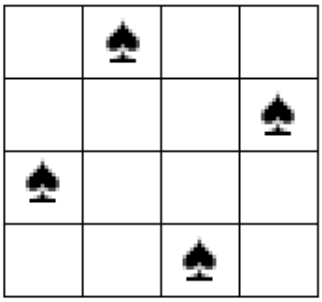
Acum putem trece la linia a II-a:



Pe linia a treia putem amplasa o regină chiar pe prima coloană:



În fine, pe ultima linie avem următoarea opţiune validă:



Backtracking-ul se poate implementa mai uşor recursiv. Programul complet pentru varianta recursivă este următorul:

```
#include "stdio.h"
#include "conio.h"
#include "math.h"

int ataca(int linie, int memorie[])
//aceasta funcţie testează daca regina de pe linie  este atacata de reginele poziţionate anterior. Este de fapt funcţia de Validare
{
      int i;
//luam pe rând toate damele poziţionate anterior
      for(i=0;i<linie;i++)
//verificam daca cele 2 dame sunt pe aceeaşi coloana sau pe diagonală
            if((memorie[linie]==memorie[i]) ||
                  (abs(memorie[i]-memorie[linie])==linie-i))
// se ataca, deci nu sunt corect poziţionate.
                        return 1;
      return 0; //daca am ajuns aici, înseamnă că nu se atacă
}



//căile de afişare sunt infinite
void afiseaza(int dim,int memorie[])
{
      int l,c;
      for(l=0;l<=dim;l++)
            printf("_");
      for(l=0;l<dim;l++)
      {
            printf("\n|");
            for(c=0;c<dim;c++)
                  if(memorie[l]==c)
                        printf("%c",6);
                  else
                        if((l+c)&1)
                              printf(" ");
                        else
                              printf("%c",219);
            printf("|");
      }
      printf("\n");
      for(l=0;l<=dim;l++)
```

```
                printf("-");
            printf("\n");
            printf("o tasta, va rog!\n");
            getch();
        }


        void dame(int dim,int linie, int memorie[])
    // aici este funcţia care face efectiv backtracking-ul (suntem la
    // linia(nivelul) linie.
    // dim ne spune cât de mare-i tabla
    // memorie tine minte damele deja poziţionate. Ea este stiva
        {
          if(linie==dim)// daca am terminat
                afiseaza(dim,memorie);//afişăm
          else //altfel avem de lucru
            for(memorie[linie]=0;memorie[linie]<dim;memorie[linie]++)
    //încercam toate coloanele de la stânga la dreapta
                        if(!ataca(linie,memorie))
    //daca e o poziţie valida trecem la nivelul următor
                                dame(dim,linie+1,memorie);
    //când ne întoarcem din apelul recursiv înseamnă că am revenit
    // la acest nivel şi încercăm următoarea coloana
        }

        void main()
        {
            int n,memorie[100];
            printf("dimensiunea tablei de joc=");
            scanf("%d",&n);
            dame(n,0,memorie);// prima linie are numărul zero
        }
```

În continuare prezentăm şi o implementare iterativă:

```
        #include "stdio.h"
        #include "conio.h"
        #include "math.h"

        int ataca(int linie, int memorie[])
    //aceleaşi explicaţii ca mai sus, dar nu de alta dar e aceeaşi funcţie
        {
            int i;
            for(i=0;i<linie;i++)
                if((memorie[linie]==memorie[i]) ||
                    (abs(memorie[i]-memorie[linie])==linie-i))
                        return 1;
            return 0;
        }

        void afiseaza(int dim,int memorie[])
    //la fel ca mai sus
        {
            int l,c;
            for(l=0;l<=dim;l++)
                printf("_");
            for(l=0;l<dim;l++)
            {
```

```
                        printf("\n|");
                        for(c=0;c<dim;c++)
                              if(memorie[l]==c)
                                    printf("%c",6);
                              else
                                    if((l+c)&1)
                                          printf(" ");
                                    else
                                          printf("%c",219);
                        printf("|");
                  }
                  printf("\n");
                  for(l=0;l<=dim;l++)
                        printf("-");
                  printf("\n");
                  printf("o tasta, va rog!\n");
                  getch();
            }

            void main()
            {
                  int n,memorie[100],nivel;
                  printf("dimensiunea tablei de joc=");
                  scanf("%d",&n);
                  nivel=0;
  //datorită C-ului primul nivel are numărul zero
  //punem prima damă să se încălzească pe marginea tablei
                  memorie[nivel]=-1;
   //cât timp nu încercăm să poziţionăm dama de linia -1
                  while(nivel>=0)
                  {
                        if(nivel==n)    //avem o soluţie
                        {
                              afiseaza(nivel,memorie);
     //ne întoarcem la ultima linie
                              nivel--;
                        }
    //încercăm următoarea poziţie de pe linia curentă
                        memorie[nivel]++;
     // dacă dama a epuizat coloanele
                        if(memorie[nivel]==n)
     //revenim la linia anterioară
                              nivel--;
                        else  // altfel
     //dacă este o poziţie validă
                              if(!ataca(nivel,memorie))
                                    memorie[++nivel]=-1;
    //trecem la următorul nivel şi scoatem încă o damă la încălzire
            }
         }
```

Această tehnică este relativ uşor de folosit, dar fiind foarte lentă şi consumatoare de memorie este recomandabilă pentru cazurile în care resursele nu sunt o problemă.

Probleme propuse.

1. Se dă un labirint cu N x M celule având forma unui pătrat, fiecare celulă având maxim o uşă pe fiecare latură (în total maxim 4 uşi). Uşile pot funcţiona într-un

singur sens. Se cere să se determine calea de la o celulă oarecare către ieşirea din labirint.
2. Se dă o hartă cu n oraşe. Pe această sunt trasate m drumuri, care unesc oraşele două câte două. Să se determine calea care leagă două oraşe oarecare.
3. Să se determine toate mutările pe care trebuie să le efectueze un cal pe o tablă de şah astfel încât să treacă prin toate poziţiile cel puţin o dată, pornind dintr-o poziţie oarecare.
4. Să se determine toate mutările pe care trebuie să le efectueze un cal pe o tablă de şah astfel încât să treacă prin toate poziţiile o singură dată, pornind dintr-o poziţie oarecare.
5. Se dau x plăci de gresie de dimensiuni 1x1, fiecare placă având fiecare margine vopsită într-o culoare. Să se determine cum poate fi pavată o încăpere de dimensiuni N x M astfel încât marginile vecine ale plăcuţelor să aibă aceeaşi culoare.
6. Se dau x plăci de gresie de dimensiuni 1x1, fiecare placă fiind vopsită într-o culoare. Să se determine cum poate fi pavată o încăpere de dimensiuni N x M astfel încât două plăcuţe vecine să nu aibă aceeaşi culoare.