Metoda Backtracking

1 Consideraţii teoretice

1.1 Aplicabilitatea algoritmilor de tip Backtracking

Algoritmii de tip backtracking se aplică la problemele unde spaţiul soluţiilor S este de forma unui produs cartezian $S = S_0 \times S_1 \times ... \times S_{n-1}$. Orice element x din spaţiul soluţiilor S va fi un vector de forma $x = (x_0, x_1, ..., x_{n-1})$, cu $x_i \in S_i$, $0<i<n$.

Nu toate elementele $x \in S$ sunt soluţii valide ale problemei. Doar acele elemente x care satisfac anumite condiţii impuse de problemă vor fi soluţii valide. Definim condiţiile care trebuie satisfăcute sub forma unei funcţii booleene *Solutie*$(x_0,x_1,...,x_{n-1})$. Un element $x=(x_0, x_1, ..., x_{n-1}) \in S$ este soluţie a problemei dacă funcţia *Solutie* aplicată componentelor lui x va returna valoarea *true*.

Scopul este de a găsi acei vectori $x \in S$ pentru care funcţia *Solutie* returnează *true*.

1.2 Modul de lucru al algoritmilor de tip Backtracking

Modul de lucru al algoritmului este următorul: se alege un element x_0^0 din S_0, apoi un element x_1^0 din S_1, apoi un element x_2^0 din S_2, ş.a.m.d, până când se va fi ales un element x_{n-1}^0 din S_{n-1}. În acest moment vom avea un vector $x = (x_0^0, x_1^0, ..., x_{n-2}^0, x_{n-1}^0) \in S$. Evaluăm rezultatul funcţiei *Solutie* pentru acest vector. Dacă obţinem rezultatul *true* atunci avem o soluţie corectă a problemei. Dacă obţinem rezultatul *false*, soluţia nu este corectă şi continuăm căutarea.

Aplicăm principiul revenirii pe cale, adică ne întoarcem cu un pas în urmă, înainte de alegerea elementului $x_{n-1}^0 \in S_{n-1}$. De aici continuăm alegând un alt element $x_{n-1}^1 \in S_{n-1}$, $x_{n-1}^1 \neq x_{n-1}^0$. Vom obţine un nou vector $x' = (x_0^0,x_1^0,...,x_{n-2}^0,x_{n-1}^1) \in S$ asupra căruia aplicăm din nou funcţia *Solutie*. Dacă rezultatul este *false*, revenim din nou cu un pas în urmă şi continuăm cu alegerea unui alt element $x_{n-1}^2 \in S_{n-1}$, $x_{n-1}^2 \neq x_{n-1}^0$ şi $x_{n-1}^2 \neq x_{n-1}^1$. Repetăm aceşti paşi până la epuizarea tuturor elementelor din S_{n-1}.

După ce am epuizat toate elementele mulţimii S_{n-1}, vom fi la faza în care am ales elementele $x_0^0, x_1^0, ..., x_{n-2}^0$. Ar trebui să alegem un element din S_{n-1}, dar cum toate elementele din S_{n-1} au fost deja alese, revenim cu încă un pas în urmă pe cale, şi alegem un alt element $x_{n-2}^1 \in S_{n-2}$. Pe urmă reîncepem alegerea elementelor din S_{n-1} cu primul dintre ele, $x_{n-1}^0 \in S_{n-1}$. Vom avea vectorul $x = (x_0^0, x_1^0,..., x_{n-3}^0, x_{n-2}^1, x_{n-1}^0)$.

Procedând în acest mod, vom ajunge practic să construim toţi vectorii $x \in S$, adică vom explora întreg spaţiul soluţiilor posibile. Există în principiu două categorii de probleme: unele care cer găsirea unei singure soluţii şi unele care cer găsirea tuturor soluţiilor. Atunci când se cere găsirea unei singure soluţii, putem opri căutarea după găsirea primului vector x pentru care funcţia *Solutie* returnează valoarea true. La cealaltă categorie de probleme este nevoie să parcurgem întreg spaţiul soluţiilor pentru a găsi toate soluţiile.

Parcurgerea întregului spaţiu al soluţiilor posibile este foarte mare consumatoare de timp. În general se poate accelera această parcurgere prin următoarea optimizare: pornind de la funcţia *Solutie*$(x_0, x_1, ..., x_{n-1})$, definim o funcţie *Continuare*$(x_0, x_1, ..., x_k)$ care să ne spună dacă, având alese la un moment dat elementele $x_0, x_1, ..., x_k$ există o şansă de a se ajunge la o soluţie. Această optimizare este foarte utilă deoarece de multe ori după alegerea câtorva elemente $x_0, x_1, ..., x_k$ ne putem da seama că nu vom putea găsi nici o soluţie validă care conţine elementele respective. În asemenea situaţii nu mai are rost să continuăm alegerea de elemente până la x_{n-1}, ci putem direct să revenim cu un pas în urmă pe cale şi să încercăm cu un alt element x_k'.

2 Implementare

Algoritmul backtracking redactat sub formă de pseudocod arată în felul următor:

```
k = 0;

while (k >= 0)
{
    do
    {
        * alege urmatorul x[k] din multimea S[k]
        * evalueaza Continuare(x[1], x[2], ..., x[k])
    }
    while ( !Continuare(x[1], x[2], ..., x[k]) &&
            (* mai sunt elemente de ales din multimea S[k]) )

    if (Continuare(x[1], x[2], ..., x[k]))
    {
        if (k == n-1)
        {
            if (Solutie(x[1], x[2], ..., x[n]))
                * afiseaza solutie
        }
        else
        {
            k = k + 1;
        }
    }
    else
    {
        k = k - 1;
    }
}
```

Dacă analizăm modul de funcţionare al algoritmului backtracking, vom vedea ca la orice moment de timp ne este suficient un tablou de *n* elemente pentru a memora elementele x_0, x_1, ..., x_{n-1} alese. Ca urmare în implementare declarăm un tablou x de dimensiune *n*. Tipul de date al tabloului depinde de problemă, de tipul mulţimilor S.

Variabila k ne indică indicele mulţimii din care urmează să alegem un element. La început de tot trebuie să alegem un element din mulţimea S_0, de aceea îl iniţializăm pe k cu valoarea 0. Pe urmă k se modifică în funcţie de modul în care avansăm sau revenim pe calea de căutare.

Pentru alegerea următorului x_k din mulţimea S_k, ne bazăm pe faptul că între elementele fiecărei mulţimi S_k există o relaţie de ordine. Dacă încă nu s-a ales nici un element din mulţimea S_k atunci îl alegem pe primul conform relaţiei de ordine. Dacă deja s-a ales cel puţin un element, atunci îl alegem pe următorul neales, conform relaţiei de ordine.

3 Exemple

3.1 Săritura calului pe tabla de şah

Enunţ Avem o tablă de şah de dimensiune 8x8. Să se găsească toate modalităţile de a deplasa un cal pe această tablă, astfel încât calul să treacă prin toate căsuţele de pe tablă exact o dată.

Rezolvare Pentru a parcurge fiecare căsuţă de pe tabla de şah exact o dată, calul va trebui să facă exact $8 \times 8 = 64$ de paşi. La fiecare pas el poate alege oricare din cele 64 de căsuţe de pe tablă. Să codificăm căsuţele de pe tabla de şah în modul următor: căsuţa de la linia i şi coloana j o notăm prin

perechea (i,j). Să notăm mulţimea tuturor căsuţelor de pe tablă cu C: C = {(0,0), (0,1), ..., (0,7), (1,0), ..., (7,7)}.

O soluţie a problemei o putem nota printr-un vector $x = (x_0, x_1, ..., x_{63})$, unde $x \in S = C \times C \times C \times ... \times C$ (produs cartezian în care mulţimea C apare de 64 de ori), iar $x_i \in C$, $\forall\, i \in \{0, 1, ..., 63\}$.

Cu aceste elemente putem vedea că se poate aplica o rezolvare de tip backtracking. Funcţia *Solutie* va verifica să nu existe două elemente c_i şi c_j care au aceeaşi valoare, deoarece asta ar însemna că s-a trecut de două ori prin aceeaşi căsuţă. În plus funcţia mai trebuie să verifice faptul că $\forall i \in \{0, 1, ..., 61, 62\}$ calul poate sări de la căsuţa c_i la căsuţa c_{i+1}. Asta înseamnă că fie c_i şi c_{i+1} se află la două linii distanţă şi la o coloană distanţă, fie ele se află la o linie distanţă şi la două coloane distanţă.

Funcţia *Continuare* trebuie să facă exact aceleaşi verificări ca şi funcţia Solutie, dar nu pentru toate 64 de căsuţe ci pentru cele k căsuţe care au fost alese până la un moment dat.

Cod sursă În continuare prezentăm codul sursă pentru rezolvarea acestei probleme.

```
#include <stdio.h>
#include <stdlib.h>

/* Dimensiunea tablei de sah definita ca si constanta.
   Pentru o tabla de dimensiune 8x8 gasirea solutiilor
   dureaza foarte mult, de aceea lucram pe o tabla de
   5x5 unde solutiile sunt gasite mult mai repede. */
#define N 5

#define INVALID -1

int main(void)
{
    /* Pentru o tabla de dimensiune N vom memora
       solutiile intr-un vector de dimensiune N*N.
       Fiecare element din vector va fi la randul lui
       un vector cu doua elemente, primul element va
       memora linia de pe tabla, iar al doilea element
       va memora coloana de pe tabla. */
    int c[N*N][2];

    int k, i;
    int pe_tabla, continuare;
    int delta_l, delta_c;

    /* Numaram si cate solutii sunt gasite. */
    int count = 0;

    /* Pentru inceput marcam toate elementele
       vectorului "c" cu INVALID, semn ca nu am
       ales nici un element din multimile
       produsului cartezian. */
    for (i=0; i<N*N; i++)
    {
        c[i][0] = INVALID;
        c[i][1] = INVALID;
    }

    k = 0;
    while (k >= 0)
    {
          /* Incercam sa plasam mutarea "k" a
```

```
    calului in fiecare casuta a tablei
    de joc, pe rand. Evaluam la fiecare
    alegere functia "Continuare". Ne
    oprim fie atunci cand am incercat
    toate casutele de pe tabla, fie
    atunci cand gasim o casuta unde
    functia "Continuare" returneaza
    "true". */
do
{
     /* Aici alegem urmatorul element
        din multimea "C[k]". Daca elementul
        "c[k]" este setat pe INVALID,
        inseamna ca inca nu am ales nici
        un element din multimea curenta,
        deci alegem primul element (plasam
        calul in casuta de la linia 0 si
        coloana 0). */
     if (c[k][0] == INVALID)
     {
         c[k][0] = 0;
         c[k][1] = 0;
         pe_tabla = 1;
     }
     /* Daca elementul "c[k]" nu este setat
        pe invalid, inseamna ca deja am ales
        o casuta din multimea "C[k]". Acum
        alegem urmatoarea casuta de pe tabla.
        Cu alte cuvinte incercam sa plasam
        calul in urmatoarea casuta. Daca este
        posibil incercam sa ramanem pe aceeasi
        linie si sa ne deplasam cu o coloana
        spre dreapta. */
     else if (c[k][1] < N-1)
     {
         c[k][1]++;
         pe_tabla = 1;
     }
     /* Daca cumva eram chiar la ultima casuta
        din linie, atunci alegem prima casuta
        din linia urmatoare. Ne asiguram ca nu
        eram cumva si pe ultima linie a
        tablei, caz in care am epuizat toate
        casutele. */
     else if (c[k][0] < N-1)
     {
         c[k][1] = 0;
         c[k][0]++;
         pe_tabla = 1;
     }
     /* Daca eram pe ultima linie a tablei,
        atunci am epuizat toate casutele.
        Marcam acest lucru setand variabila
        "pe_tabla" pe 0. */
     else
     {
         pe_tabla = 0;
     }
```

```
        /* Daca casuta "c[k]" aleasa este valida
           (se afla pe tabla de joc), atunci
           trecem la calculul functiei
           "Continuare". */
        if (pe_tabla)
        {
            /* Daca suntem la prima mutare a
               calului, atunci mutarea este
               valida oriunde ar fi ea pe
               tabla. */
            if (k == 0)
                continuare = 1;
            /* Daca nu suntem la prima mutare,
               atunci trebuie sa facem o serie
               de verificari. */
            else
            {
              /* In primul rand verificam daca
                 de la pozitia precedenta a
                 calului pe tabla ("c[k-1]")
                 se poate ajunge in pozitia
                 aleasa acum printr-o mutare.
              */
              delta_l = abs(c[k-1][0]-c[k][0]);
              delta_c = abs(c[k-1][1]-c[k][1]);
              continuare = (((delta_l == 1) &&
                             (delta_c == 2)) ||
                            ((delta_l == 2) &&
                             (delta_c == 1)));

              /* Si apoi verificam daca nu
                 cumva s-a mai trecut prin
                 casuta aleasa acum. */
              for (i=0; continuare && (i<k); i++)
              {
                  if ((c[i][0] == c[k][0]) &&
                      (c[i][1] == c[k][1]))
                      continuare = 0;
              }
            }
        }
        /* Daca casuta "c[k]" aleasa este in
           afara tablei de sah, atunci functia
           "Continuare" va returna automat
           "false". */
        else
        {
            continuare = 0;
        }
    }
    while (!continuare && pe_tabla);

    /* Daca am obtinut rezultat pozitiv in urma
       verificarilor de "Continuare", atunci
       consideram piesa asezata la pozitia "c[k]"
       si continuam cautarea. */
    if (continuare)
```

```
            {
                /* Daca s-a parcurs toata tabla de sah
                   atunci afisam solutia. */
                if (k == N*N - 1)
                {
                      for (i=0; i<N*N; i++)
                          printf("(%d,%d) ", c[i][0],
                                             c[i][1]);
                      printf("\n");
                      count++;
                }
                /* Daca nu s-a parcurs inca toata tabla
                   atunci trecem cu un pas inainte pe
                   calea de cautare. */
                else
                {
                    k++;
                }
            }

            /* Daca casuta aleasa nu este valida, atunci
               marcam elementul "c[k]" cu INVALID si
               revenim cu un pas inapoi pe calea de
               cautare. */
            else
            {
                c[k][0] = INVALID;
                c[k][1] = INVALID;
                k--;
            }
        }

        printf("%d solutii\n", count);
        return 0;
}
```

Optimizare Putem face o optimizare la programul prezentat, pornind de la faptul că se cunosc regulile după care se poate deplasa calul pe tabla de şah. La alegerea din mulţimea C_k, în loc să alegem pe rând fiecare căsută şi apoi să verificăm dacă s-ar putea face mutare în căsuţa respectivă, e mai eficient să alegem direct dintre căsuţele în care se poate face mutare. Pentru a putea determina aceste căsuţe, folosim doi vectori care definesc mutările posibile ale calului (numărul de căsuţe cu care se poate deplasa pe orizontală şi pe verticală). Prezentăm mai jos codul sursă care implementează această optimizare. Implementarea este una recursivă, pentru a arăta că modul de lucru al algoritmului backtracking se pretează în mod natural la implementări recursive.

```
#include <stdio.h>

#define N 5

int dx[8] = {-1, -2, -2, -1,  1,  2,  2,  1};
int dy[8] = {-2, -1,  1,  2,  2,  1, -1, -2};

int c[N*N][2];
int count = 0;

void back(int pas)
{
     int i, j, continuare;
```

```
    if (pas == N*N)
    {
       for (i=0; i<pas; i++)
           printf("(%d,%d) ", c[i][0], c[i][1]);
       printf("\n");
       count++;
    }
    else
    {
        for (i=0; i<8; i++)
        {
            c[pas][0] = c[pas-1][0] + dy[i];
            c[pas][1] = c[pas-1][1] + dx[i];

            if ((c[pas][0]>=0) && (c[pas][0]<N) &&
                (c[pas][1]>=0) && (c[pas][1]<N))
            {
                continuare = 1;
                for (j=0; continuare && (j<pas); j++)
                {
                    if ((c[j][0] == c[pas][0]) &&
                        (c[j][1] == c[pas][1]))
                       continuare = 0;
                }

                if (continuare)
                   back(pas+1);
            }
        }
    }
}

int main(void)
{
    int i,j;
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
        {
            c[0][0] = i;
            c[0][1] = j;
            back(1);
        }
    printf("%d solutii\n", count);
    return 0;
}
```

4 Probleme propuse

4.1 Ieşirea din labirint

Să se scrie un program care găseşte toate căile de ieşire dintr-un labirint.

Configuraţia labirintului se citeşte din fişierul text “labirint.dat”. Labirintul are forma unei matrici de dimensiune NxM. Un element al matricii poate fi perete sau spaţiu liber. În fişierul de intrare

labirintul este descris pe N linii de câte M caractere. Peretele se codifică prin litera `'P'`, iar spaţiul liber prin caracterul `'.'`.

Un exemplu de fişier de intrare care descrie un labirint este:

```
PPPP.PPP
P....PPP
PP..PPSP
PP.PP..P
P..PP.PP
P.PPP..P
P...PP.P
PPP.PP.P
PPP....P
PPPPPPPP
```

În afară de caracterele `'P'` şi `'.'`, în fişierul de intrare va mai apare şi caracterul `'S'`, o singură dată. Caracterul `'S'` reprezintă un spaţiu liber în labirint şi marchează punctul de unde începem căutarea ieşirilor.

Deplasarea se poate face doar pe verticală şi pe orizontală între oricare două celule învecinate, cu condiţia ca ambele celule să fie spaţii libere. Nu este permis să se treacă de mai multe ori prin aceeaşi celulă.

Pentru fiecare cale de ieşire găsită, programul trebuie să afişeze această cale pe ecran, după care va aştepta apăsarea unei taste. După apăsarea unei taste se va trece la următoarea cale de ieşire găsită, ş.a.m.d. Pentru afişarea unei căi de ieşire se va folosi aceeaşi codificare ca şi cea din fişierul de intrare, cu diferenţa că se va marca drumul de ieşire prin caracterul `'x'`.

Spre exemplu un drum de ieşire posibil pentru labirintul de mai sus este:

```
PPPPxPPP
P.xxxPPP
PPx.PPSP
PPxPPxxP
PxxPPxPP
PxPPPxxP
PxxxPPxP
PPPxPPxP
PPPxxxxP
PPPPPPPP
```

Dacă nu există nici un drum de ieşire din labirint, programul va afişa mesajul “Nu avem nici un drum de ieşire.”

Un labirint va avea maxim 24 de linii şi 80 de coloane. Valorile pentru N şi M nu sunt date explicit, va trebui să le deduceţi din structura fişierului de intrare. Se garantează că datele de intrare sunt corecte.

4.2 Ieşirea cea mai rapidă din labirint

Pentru problema anterioară să se determine şi să se afişeze pe ecran doar cel mai scurt drum de ieşire din labirint.

4.3 Problema celor 8 regine

Avem o tablă de şah de dimensiune 8x8. Să se aşeze pe această tablă de şah 8 regine astfel încât să nu existe două regine care se atacă între ele.

Implementarea trebuie să fie făcută folosind recursivitatea.