1 Laborator 9: Backtracking

1.1 Obiective

Scopul acestei lucrari este de a va familiariza cu tehnica backtracking-ului. Se prezinta pe scurt fundamentele teoretice, si se prezinta doua exemple de probleme rezolvate prin aceasta tehnica, utilizand doua sabloane usor diferite (unul dintre ele fiind sablonul general prezentat la curs). Problemele obligatorii au scopul de a fixa notiunile teoretice si de a stimula exercitiul practic. Va recomandam de asemenea abordarea problemelor optionale. La finalul lucrarii aveti propuse si doua probleme extra credit.

1.2 Notiuni teoretice

Tehnica backtracking-ului este utilizata in dezvoltarea de algoritmi pentru urmatorul tip de probleme: avem n multimi diferite, S_1, S_2,...,S_n, fiecare avand n_i componente. Solutia unei astfel de probleme se poate reprezenta printr-un vector $X = (x_1, x_2, ..., x_n) \in$S=S$_1 \times S_2 \times ... \times S_n$ care respecta o relatie $\varphi(x_1, x_2, ..., x_n)$ intre componentele vectorului X (data). Relatia φ se numeste *relatie interna*, multimea $S = S_1 \times S_2 \times ... \times S_n$ se numeste *multimea/spatiul solutiilor posibile*, vectorul $X = (x_1, x_2, ..., x_n)$ se numeste rezultat.

Backtracking genereaza toate solutiile posibile (fezabile, corecte) ale problemei. Dintre acestea, se poate selecta una care satisface o conditie aditionala - minimizeaza/maximizeaza o *functie obiectiv* (e.g. gaseste drumul de cost minim intr-un graf, sau numarul minim de monezi prin care se poate genera o suma data).

Backtracking-ul elimina necesitatea de a genera toate cele $\prod_{i=1}^{n} n_S$ solutii posibile. Pentru a realiza aceasta reducere a spatiului, in algoritm se respecta urmatoarele conditii:

- x_k primeste valori doar daca $x_1, x_2, ..., x_{k-1}$ au primit deja valori
- dupa ce x_k primeste o valoare, se verifica relatia de continuare $\varphi(x_1, x_2, ..., x_k)$, stabilindu-se daca are sens sa se evalueze x_{k+1}. Daca conditia $\varphi(x_1, x_2, ..., x_k)$ nu este satisfacuta, se alege o noua valoare pentru $x_k \in$S$_k$ si se testeaza φ din nou. Daca multimea de valori posibile pentru x_k devine vida, se reincepe prin selectia urmatoarei valori pentru x_{k-1}, si asa mai departe. Aceasta revenire la pasi anteriori pentru k da si numele metodei de fapt: cand nu este posibila explorarea spatiului de cautare pe directia curenta, se revine (*back-track*) pe calea de construire a solutiei curente, si se incearca noi valori. Exista o relatie puternica intre conditia de continuare si relatia interna - stabilirea optima a conditiei de continuare reduce mult numarul de stari generate.

O versiune iterativa a strategiei si una recursiva sunt prezentate schematic mai jos:

```
#define MAXN ? /* suitable value */

void nonRecBackTrack( int n )
/* sets S_i and the corresponding number of elements in each set,
  n_S, are assumed global */
{
        int x[ MAXN ];
        int k, v;

        k=1;
        while ( k > 0 )
        {
                v = 0;
                while ( ∃ untested α ∈ S_k && v == 0 )
                {
                        x[ k ] = α;
                        if ( φ( x[ 1 ], x[ 2 ],..., x[ k ] )
                                v = 1;
                }
                if ( v == 0 )
                        k--;
                else
                if ( k == n )
                        listOrProcess( x, n )
                        /* list or process solution */
                else
                        k++;
        }
```

```
}

#define MAXN ? /* suitable value */
int x[ MAXN ];

/* n, the number of sets S_k, sets S_k and the corresponding number of elements
   in each set, nS[k], are assumed global */
void recBackTrack( int k )
{
        int j;

        for ( j = 1; j <= nS[ k ]; j++ )
        {
                x[ k ] = S_k[ j ];
                /* the j^th element of set S_k */
                if ( φ( x[1], x[2],..., x[k] )
                        if ( k < n )
                                recBackTrack( k + 1 );
                        else
                          listOrProcess( x, n ) /* list or process solution */
        }
}
```

Exemplu. Problema plasarii reginelor pe tabla de sah

Problema se poate formula in felul urmator:

> Gasiti toate aranjamentele de n regine pe o tabla de sah de dimensiune $n \times n$, astfel incat nici o regina sa nu ameninte o alta regina (nu se afla pe aceeasi linie sau diagonala).

Intrucat pe fiecare linie trebuie sa avem cate o singura regina, solutia se poate reprezenta ca un vector $X = (x_1, x_2, \ldots, x_n)$, unde x_i este coloana pe care este plasata regina de pe linia i.

Conditiile de continuare sunt:

- doua regine nu se pot afla pe aceeasi coloana, i.e. $X[i] \neq X[j] \forall i \neq j$
- doua regine nu se pot afla pe aceeasi diagonala, i.e. $|k - i| \neq |X[k] - X[i]|$ for $i = 1, 2, \ldots, k - 1$

O solutie nerecursiva:

```
#include <stdio.h>
#include <stdlib.h>
#define MAXN 10

void nonRecQueens( int n )
/* find all possible arrangements of n queens on a chessboard such that no queen
   threatens another */
{
        int x[ MAXN ];
        int v;
        int i, j, k, solNb;

  solNb = 0;
        k = 1;
        x[ k ] = 0;
        while( k > 0)
        { /* find a valid arrangement on line k */
                v=0;
                while( v==0 && x[ k ] <= n - 1 )
                {
                        x[ k ]++;
                        v = 1;
                        i = 1;
                        while ( i <= k - 1 &&
                                                        v == 1 )
                                if ( x[ k ] == x[ i ] ||
                                                abs( k - i ) ==
                                                abs( x[ k ] - x[ i ] ))
                                        v=0;
                                else
```

```
                                i++;
        }
              if ( v == 0 )
                k = k - 1;
              else
              {
                      if ( k== n )
                      { /* display chessboard */
                              solNb++;
                              printf( "\nSolution %d\n", solNb );
                              for ( i = 1; i <= n; i++ )
                              {
                                      for ( j = 1; j <= n; j++ )
                                              if ( x[ i ] == j )
                                                      printf( "1" );
                                              else
                                                printf( "0" );
                                              printf( "\n" );
                              }
                              while ( '\n' != getchar() );
                      }
                      else
                      {
                              k++;
                              x[ k ]=0;
                      }
              }
      }
}
int main(void)
{
  int n;

  printf( "\nNumber of queens=" );
  scanf( "%d", &n );
        while ( '\n' != getchar() );
  nonRecQueens( n );
  printf( "\nEND\n" );
  return 0;
}
```

O solutie recursiva:

```
#include <stdio.h>
#include <stdlib.h>
#define MAXN 10
int x[ MAXN ];
int n; /* chessboard size */
int solNb;  /* solution number */
enum { FALSE=0, TRUE=1 };

int phi( int k )
/* test continuation conditions */
{
  int p;

  for ( p = 1; p <= k - 1; p++ )
    if ( x[ k ] == x[ p ] ||
         abs( k - p ) ==
         abs( x[ k ] - x[ p ] ))
      return FALSE;
  return TRUE;
}
void recQueens( int k )
/* find all possible arrangements of n queens on a chessboard such that no queen
   threatens another */
{
  int i, j, p;

  for ( j = 1; j <= n; j++ )
  {
    x[ k ] = j;
    if ( phi( k ) == TRUE )
      if ( k < n )
```

```
        recQueens( k + 1 );
      else
      { /* list solution */
        solNb++;
        printf( "\nSolution %d\n", solNb );
        for ( i = 1; i <= n; i++ )
        {
          for ( p = 1; p <= n; p++ )
            if ( x[ i ] == p )
              printf( "1" );
            else
              printf( "0" );
          printf( "\n" );
        }
        while ( '\n' != getchar() );
      }
  }
}
int main(void)
{
  printf( "\nNumber of queens=" );
  scanf( "%d", &n );
  while ( '\n' != getchar() );
  solNb = 0;
  recQueens( 1 );
  printf( "\nEND\n" );
  return 0;
}
```

Exemplu. Numararea restului

Un alt exemplu de problema care poate fi abordata cu tehnica backtracking este problema numararii restului. Problema are mai multe enunturi posibile, redam aici unul dintre ele:

> Se va o multime de bancnote si monede, fiecare avand o anumita valoare (valorile se pot repeta). Se cere sa se determine numarul minim de monezi si bancnote necesar pentru a returna un anumit rest.

Valorile diferitelor unitati monetare disponibile le vom stoca intr-un vector de dimensiune n (numarul de bancnote/-monede disponibile) - $values[n]$ pe fiecare pozitie vom avea valoarea respectivei monede/bancnote. Avand in vedere ca avem un anumit numar de monede/bancnote disponibile de o anumita valoare, vectorul poate contine valori duplicate. O sa reprezentam o solutie ca un vector de dimensiune n - $x[n]$ - cu valori binare (pe pozitia i avem 0 daca acel elementul de valoare $values[i]$ nu este utilizat in solutia curenta, respectiv 1 daca este utilizat).

Mai jos prezentam o solutie care gaseste **TOATE** solutiile posibile pentru problema numararii restului (codul urmeaza sablonul prezentat la curs; s-a mentinut structura generala de acolo, comentandu-se elementele care nu sunt necesare in solutia curenta):

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

int solNb=0; /* solution number */
enum { FALSE=0, TRUE=1 };

void construct_candidates(int c[], int *ncandidates, int change){
   if(change > 0){
        c[0] = 1;
        c[1] = 0;
        *ncandidates = 2;
   }else *ncandidates = 0;

}

bool is_a_solution(int change)
{
    return (change == 0);
}

void backtrack(int a[], int k, int n, int change, int values[])
{
    int c[2]; /* candidates for next position */
    int ncandidates; /* next position candidate count */
    int i; /* counter */
```

```
    if(k == n) //am parcurs toate valorile posibile de monezi
        return;


    if (is_a_solution(change)){
        printf("\n---- FOUND SOLUTION %d??\n", ++solNb);
        //must process solution
         for(int i=0; i<n; i++)
            printf("%d ", a[i]*values[i]);
        printf("\n");
    }
    else {
        k = k+1;
        construct_candidates(c,&ncandidates,change);
        for (i=0; i<ncandidates; i++) {
            a[k] = c[i];
            // make_move(a,k,input);
            backtrack(a,k,n,change-a[k]*values[k], values);
            //unmake_move(a,k,input);
            //if (finished) return; /* terminate early */
        }

    }
}


int main()
{
    int n, change;
    int* values = NULL;
    int* a = NULL;
    printf( "\nNumber of coins=\n" );
    scanf( "%d", &n );
    printf("Change to be returned:\n");
    scanf( "%d", &change );
    printf("Input coin values:\n");
    values = (int*)malloc(n*sizeof(int));
    a=(int*)malloc(n*sizeof(int));
    for(int i=0; i<n; i++){
        a[i] = values[i] = 0;
    }
    for(int i=0; i<n; i++)
        scanf("%d", values+i);

    backtrack(a, -1, n, change, values);

    printf( "\nEND\n" );
    return 0;
}
```

1.3 Mersul lucrarii

1.3.1 Probleme obligatorii

1. Rulati exemplele prezentate in sectiunea 1.2
2. Modificati solutia problemei pentru numararea restului astfel incat sa returneze doar solutia optima (i.e. solutia care contine numarul minim de bancnote/monede). Algoritmul vostru ar trebui sa nu exploreze ramuri ale spatiului de cautare care nu pot oferi solutii mai bune decat cea mai buna solutie gasita pana la acel moment).
3. Dati o solutie bazata pe backtracking pentru generarea tuturor submultimilor unei multimi de n intregi, $S = \{1, 2, ..., n\}$.
4. Dati o solutie bazata pe backtracking pentru generarea tuturor permutarilor unei multimi de n intregi, $S = \{1, 2, ..., n\}$.

1.3.2 Probleme optionale

Rezolvati urmatoarele probleme folosind tehnica backtracking-ului. Datele de intrare/iesire se citesc/scriu din/in fisier.

1. *Colorarea hartilor*. O harta a lumii contine n tari. Fiecare tara se invecineaza cu una sau mai multe tari. Se dau m culori diferite, si ce cere sa se gaseasca toate colorarile posibile utilizand cele m culori, astfel incat oricare 2 tari

vecine au culori diferite .
I/O description. Intrare: numarul de tari pe o linie, urmat de relatiile de vecinatate − fiecare pe o linie; apoi numarul de culori, urmat de cele m culori, cate 1 pe fiecare linie, date ca si string.
E.g.

```
9␣␣␣#␣number␣of␣countries
Romania␣Hungary
Romania␣Serbia
Romania␣Bulgaria
...
5␣␣␣#␣number␣of␣colors
red
green
yellow
...
```

Orice urmeaza dupa # pe o linie este comentariu, deci se ignora Iesirea sunt perechi de varf − culoare pairs, cate 1 pe linie, e.g:

```
Romania␣yellow
Hungary␣green
Serbia␣red
Ukraine␣white
...
```

2. *Ciclu Hamiltonian*. Un graf conex $G = (V, E)$ este reprezentat printr-o matrice de costuri, toate costurile fiind pozitive, ≤ 65534. Se cere sa se determine ciclul simplu care trece prin toate nodurile (un ciclu Hamiltonian) de cost minim .
I/O description. Intrare: numarul de noduri pe o linie, urmat de matricea de costuri, linie cu linie.

```
6
␣␣␣␣␣␣0␣␣␣␣␣1␣␣␣␣␣2␣␣␣␣␣3␣␣␣␣␣4␣␣␣␣␣5
0␣␣␣␣␣0␣␣␣␣␣3␣␣␣␣␣6␣65535␣65535␣␣␣␣␣2
1␣␣␣␣␣3␣␣␣␣␣0␣␣␣␣␣1␣␣␣␣␣3␣65535␣65535
2␣␣␣␣␣6␣␣␣␣␣1␣␣␣␣␣0␣65535␣␣␣␣␣4␣65535
3␣65535␣␣␣␣␣3␣65535␣␣␣␣␣0␣␣␣␣␣5␣␣␣␣␣6
4␣65535␣65535␣␣␣␣␣4␣␣␣␣␣5␣␣␣␣␣0␣␣␣␣␣2
5␣␣␣␣␣2␣65535␣65535␣␣␣␣␣6␣␣␣␣␣2␣␣␣␣␣0
```

Aici valoarea 65535 inseamna ca nu exista arc ($+\infty$). Nodurile se numeroteaza de la 0.
Iesire: secventa de noduri, separate prin spatiu (in exemplu nu se da ciclul de cost minim).

```
0␣2␣1␣3␣4␣5␣0
```

3. Un labirint este codificat folosind o matrice $n \times m$, cu coridoarele reprezentate de valori de 1 la pozitii consecutive pe aceeasi linie sau coloana, restul elementelor fiind 0. O persoana se afla la pozitia (i, j) in interiorul labirintului. Gasiti toate rutele de iesire din labirint care nu trec prin acelasi loc de doua ori .
I/O description. Intrare: n si m pe o linie, urmate de matricea A,coordonatele iesirii, si coordonatele persoanei, e.g.

```
25␣30
00000000000000000000000000000
00111111011111111101111111100
00100001010000000000001000100
...

...
24␣3
2␣1
```

Iesirea este o secventa de perechi rand−coloana care indica locatiile succesive ale persoanei.

4. Se da o multime de numere intregi. Sa se genereze toate subseturile ale caror suma este egala cu S .
   I/O description. Intrare: enumerarea elementelor multimii, pe o linie, si suma pe a doua linie, e.g.

   ```
   1␣-3␣5␣-7␣2␣6
   6
   ```

   Iesire: enumerarea elementelor care au suma ceruta (cate 1 solutie pe linie), e.g.

   ```
   1␣-3␣2␣6
   5␣-7␣2␣6
   ...
   ```

5. Se da o multime de numere naturale. Generati toate submultimile acestei multimi, careia daca i s-ar atasa operatorii + sau − alternativ se obtine suma S .
   I/O description. Intrare: enumerarea elementelor din multime, pe o linie, suma pe a doua linie e.g.

   ```
   1␣3␣5␣7␣2␣6
   0
   ```

   Iesire: enumerarea elementelor care dau suma data (expresiile rezultate), cate 1 solutie pe linie e.g.

   ```
   1-3+2
   1+3-6
   5-7+2
   1+5-6
   ...
   ```

1.3.3 Probleme extra credit

1. O bila este plasata pe o duna de nisip de inaltime variabila, situata intr-o regiune plana. Inaltimea dunei (numar natural ≤ 255) este stocata intr-o matrice $n \times m$ de inaltimi discrete - numere naturale. Inaltimea regiunii plane este cu 1 mai mica decat cea a celui mai jos punct de duna. Pozitia initiala a bilei este data de perechea rand−coloana, i, j din matrice. Generati toate posibilitatile ca bila sa coboare din duna pe suprafata plana fara a trece de doua ori prin aceeasi locatie. Doar daca bila se afla in miscare bila poate trece prin puncte de aceeasi inaltime. Bila se poate misca doar pe linii sau pe coloane. .
   I/O description. Intrare: n si m pe o linie, urmate de randurile matricei A, si coordonatele initiale ale bilei, e.g.

   ```
   5␣4␣␣#␣dune␣size
   15␣15␣11␣22
   15␣10␣11␣15
   10␣␣2␣16␣16
   ␣7␣␣8␣15␣33
   11␣11␣11␣11
   3,3␣␣#␣ball␣position
   ```

   Iesirea este o secventa de perechi rand−coloana indicand pozitiile succesive ale bilei, e.g.

   ```
   3,3␣2,3␣1,3
   3,3␣4,3␣4,2␣4,1
   ```

2. *Sudoku*. Se da o matrice 9×9 partial completata cu numere naturale intre 1 si 9. Se cere sa se completeze matricea cu numere intre 1 si 9 astfel incat fiecare rand, coloana si sub-matrice 3×3 sa contina toate numerele de la 1 la 9, o singura data.