9 Laborator: Backtracking

9.1 Obiective

Scopul acestei lucrari este de a va familiariza cu tehnica de programare *backtracking*. Se prezinta pe scurt fundamentele teoretice, si se propun cateva probleme care pot fi rezolvate utilizand aceasta tehnica.

9.2 Notiuni teoretice

Tehnica de programare *backtracking* este utilizata in general pentru dezvoltarea de algoritmi pentru urmatorul tip de probleme: se dau n multimi diferite, S_1, S_2,...,S_n, fiecare avand n_i componente si se cer una sau mai multe solutii care se pot reprezenta sub forma unui 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. 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 *solutie*.

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* (de exemplu gaseste drumul de cost minim intr-un graf, sau numarul minim de monezi prin care se poate plati o suma data).

Backtracking-ul elimina necesitatea de a genera toate cele $\prod_{i=1}^{n} n_i$ solutii posibile. Pentru a realiza aceasta reducere a spatiului a spatiului de cautare, 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 conditia 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 de fapt si numele metodei *backtracking*: 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 $\varphi(x_1, x_2, ..., x_k)$ si relatia interna $\varphi(x_1, x_2, ..., x_n)$: stabilirea optima a conditiei de continuare reduce mult numarul de stari generate (spatiul de cautare).

9.3 Algoritmul *backtracking* nerecursiv

In cele ce urmeaza este prezentata o versiune iterativa (pseudocod) a tehnicii de programare *backtracking*.

```
/* define the number of sets n; the sets S_i, i = 1,n (overline); the number of elements in each set, n_i; the
    maximum number of elements in solution vector, MAXN */
enum {INVALID, VALID};
void nonRecursiveBackTracking() {
        int x[MAXN]; /* solution vector, integer elements assumed */
        int k; /* current position in solution vector */
        int v; /* boolean-value for storing if a partial solution is found */
        int k = 1; /* starts with the first element in solution vector */
        while ( k > 0 ) {
                v = INVALID; /* the partial solution is not yet valid */
                while ( ∃untested α ∈ S_k && v == INVALID )
                {
                        x[k] = α; /* try this element α ∈ S_k */
                        if ( φ(x[1], x[2], ..., x[k]) )
                                v = VALID; /* a valid partial solution was found */
                }
                if ( v == INVALID )
                        k--; /* all elements α ∈ S_k were tried with no success; go back to k-1 */
                else
                        if ( k == n )
                                listOrProcessSolution(x, n); /* a solution was found; list or
                                    process it */
                        else
                                k++; /* the partial solution is valid, go next to k+1 towards
                                    fulfilling the solution vector */
        }
}
```

9.4 Algoritmul *backtracking* recursiv

In cele ce urmeaza este prezentata o versiune recursiva (pseudocod) a tehnicii de programare *backtracking*. Spre deosebire de varianta nerecursiva, aceasta are avantajul ca nu trebuie sa se implementeze explicit mutarea inapoi in construirea vectorului solutie. Acest lucru se datoreaza faptului ca fiecare apel recursiv este plasat automat pe stiva programului si este automat eliminat in momentul in care se termina executia lui (se revine inapoi in locul de unde a fost initiat apelul), realizandu-se astfel automat mutarea inapoi in construirea vectorului solutie.

```
/* define the number of sets n; the sets S_i, i = 1,n; the number of elements in each set, n_i; the
    maximum number of elements in solution vector, MAXN */
int x[MAXN]; /* solution vector, integer elements assumed */

void recursiveBackTracking(int k) { /* k is the current position in solution vector */
        for ( each element α ∈ S_k ) {
                x[k] = α; /* try this element α ∈ S_k */
                if ( φ( x[1], x[2], ..., x[k]) ) /* check if the partial solution is valid */
                        if ( k < n )
                                recursiveBackTracking(k + 1); /* a valid partial solution was found,
                                                        go next to k+1 towards fulfilling the
                                                            solution vector */
                        else
                                listOrProcessSolution(x, n); /* a solution was found;
                                                               list or process it */
        }
}
```

9.5 Mersul lucrarii

9.5.1 Probleme obligatorii

1. Problema generarii permutarilor

Problema se poate formula in felul urmator:

> Sa se genereze toate permutarile multimii de numere naturale $1 \ldots n$. Afisati intreg spatiul explorat si marcati cand s-a gasit cate o solutie.

Exemplu - Rezultate afisate pentru $n = 3$:

```
1
1 1
1 2
1 2 1
1 2 2
1 2 3 - solution
1 3
1 3 1
1 3 2 - solution
1 3 3
2
2 1
2 1 1
2 1 2
2 1 3 - solution
2 2
2 3
2 3 1 - solution
2 3 2
2 3 3
3
3 1
3 1 1
3 1 2 - solution
3 1 3
3 2
```

```
3 2 1 - solution
3 2 2
3 2 3
3 3
```

2. Problema generarii combinarilor

Problema se poate formula in felul urmator:

> Sa se genereze toate combinarile de n elemente luate cate k ale multimii de numere naturale $1...n$. Afisati toate solutiile si numarul total al acestora.

Exemplu - Rezultate pentru $n = 5$ si $k = 4$:

```
1 2 3 4
1 2 3 5
1 2 4 5
1 3 4 5
2 3 4 5
5 total solutions.
```

3. 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. $|k - i| \neq |X[k] - X[i]|$ pentru $i = 1, 2, \ldots, k - 1$

Pentru implementare, dezvoltati scheletul de cod existent!
Exemplu - Rezultate pentru $n = 4$ ale pozitionarii reginelor pe tabla de sah: regine(R) si pozitii libere(.)

```
Solution 1:
.R..
...R
R...
..R.

Solution 2:
..R.
R...
...R
.R..
```

4. Problema numararii restului

Problema se poate formula in felul urmator:

> Se da 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 plati un anumit rest de bani.

Valorile diferitelor unitati monetare disponibile le vom stoca intr-un vector $values$ de dimensiune n (numarul de bancnote/monede disponibile) - fiecare element $values[i]$, $i = \overline{1, n}$ va 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 x de dimensiune n cu valori binare (pe pozitia i avem 0 daca acea bancnota/moneda de valoare $values[i]$ nu este utilizat in solutia curenta, respectiv 1 daca este utilizata).

Pentru implementare, imbunatatiti algoritmul existent care urmareste sablonul prezentat la curs si care deja determina toate modalitatile de plata a restului avand la dispozitie respectivele bancnote/monede. Algoritmul pe care il dezvoltati trebuie sa nu exploreze ramuri ale spatiului de cautare care nu pot oferi solutii mai bune decat cea mai buna solutie gasita pana in acel moment.
Exemplu:
Intrare:
```
Number of coins=6
Change to be returned=30
Input coin values:
10
10
5
20
5
5
```
Iesire:
Programul existent afiseaza deja toate modalitatile de plata a restului de bani:
```
Found solution 1:
10 10 5 0 5 0
Found solution 2:
10 10 5 0 0 5
Found solution 3:
10 10 0 0 5 5
Found solution 4:
10 0 0 20 0 0
Found solution 5:
0 10 0 20 0 0
Found solution 6:
0 0 5 20 5 0
Found solution 7:
0 0 5 20 0 5
Found solution 8:
0 0 0 20 5 5
```
Dupa dezvoltarea algoritmului care reduce spatiul de cautare si care determina solutia optima, programul ar trebui sa afiseze:
```
Found solution 1:
10 10 5 0 5 0
Found solution 2:
10 0 0 20 0 0
Optimal solution:
10 0 0 20 0 0
```

9.5.2 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 doua tari vecine sa aiba culori diferite
   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 una pe fiecare linie, date ca si sir de caractere.

   ```
   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 reprezinta perechi varf − culoare, cate una 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 un ciclul simplu care trece prin toate nodurile (ciclu Hamiltonian).
   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: o secventa de noduri, separate prin spatiu, reprezentand un ciclu Hamiltonian.

```
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
   Intrare: n si m pe o linie, urmate de matricea A, coordonatele iesirii, si coordonatele persoanei.

```
25␣30
000000000000000000000000000000
001111110111111111011111111100
001000010100000000000001000100
...

...
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
   Intrare: enumerarea elementelor multimii, pe o linie, si suma pe a doua linie

```
1␣-3␣5␣-7␣2␣6
6
```

   Iesire: enumerarea elementelor care au suma ceruta (cate o solutie pe linie)

```
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
   Intrare: enumerarea elementelor din multime, pe o linie, suma pe a doua linie.

```
1␣3␣5␣7␣2␣6
0
```

Iesire: enumerarea elementelor care dau suma data (expresiile rezultate), cate o solutie pe linie

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

9.5.3 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.
   Intrare: n si m pe o linie, urmate de randurile matricei A, si coordonatele initiale ale bilei, de exemplu

   ```
   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, de exemplu

   ```
   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.
   In implementare tineti cont de optimizarea operatiilor necesare si reducerea pe cat posibil a spatiului de cautare a solutiilor.
3. *Ciclu Hamiltonian de cost minim*. Sa se determine ciclul Hamiltonian de cost minim dintr-un graf complet neorientat (vezi problema optionala 2 pentru ciclu Hamiltonian simplu!). Se va optimiza implementarea astfel incat se va reduce spatiul de cautare prin evitarea generarii de cicluri duplicate - de exemplu: $a - b - c - d - a$ este un ciclu identic cu $a - d - c - b - a$.