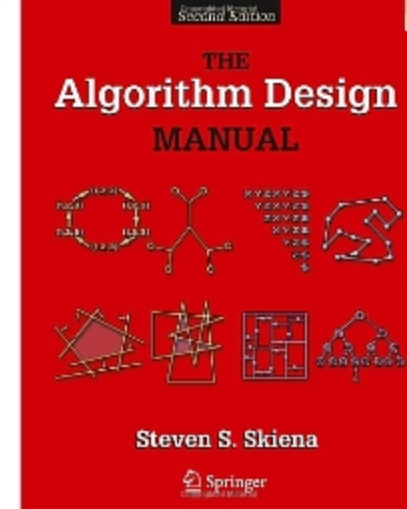


Curs 13:

Tehnica căutării cu revenire (Backtracking)

Motivație

S. Skiena – The Algorithm Design Manual



Interview Problems

- 7-14. [4] Write a function to find all permutations of the letters in a particular string.
- 7-15. [4] Implement an efficient algorithm for listing all k -element subsets of n items.
- 7-16. [5] An anagram is a rearrangement of the letters in a given string into a sequence of dictionary words, like *Steven Skiena* into *Vainest Knees*. Propose an algorithm to construct all the anagrams of a given string.

Structura

- Ce este tehnica căutării cu revenire ?
- Structura generală a algoritmilor proiectați folosind această tehnică
- Aplicații:
 - Generarea permutărilor
 - Problema plasării reginelor pe tabla de șah
 - Colorarea hărților
 - Găsirea traseelor care unesc două locații
 - Problema labirintului

Ce este tehnica căutării cu revenire?

- Este o strategie de căutare sistematică în spațiul soluțiilor unei probleme
- Se utilizează în special pentru rezolvarea problemelor a căror cerință este de a determina configurații care satisfac anumite restricții (**probleme de satisfacere a restricțiilor**).
- Majoritatea problemelor ce pot fi rezolvate folosind tehnica căutării cu revenire se încadrează în următorul șablon:

“ Să se găsească o submulțime S a produsului cartezian $A_1 \times A_2 \times \dots \times A_n$ (A_k – mulțimi finite) având proprietatea că fiecare element $s=(s_1,s_2,\dots,s_n)$ satisface anumite restricții”

Exemplu: generarea tuturor permutărilor mulțimii $\{1,2,\dots,n\}$

$A_k = \{1,2,\dots,n\}$ pentru fiecare $k=1..n$

$s_i \neq s_j$ pentru orice $i \neq j$ (restricția: componente distincte)

Ce este tehnica căutării cu revenire?

Ideea de bază:

- Soluțiile sunt construite în manieră **incrementală** prin găsirea succesivă a valorilor potrivite pentru componente (la fiecare etapă se completează o componentă; o soluție în care doar o parte dintre componente sunt completate este denumită **soluție parțială**)
- Fiecare soluție parțială este evaluată cu scopul de a stabili dacă este **validă (promițătoare)**. O soluție parțială validă poate conduce la o soluție finală pe când una invalidă încalcă restricțiile parțiale, însemnând că nu va conduce niciodată la o soluție care să satisfacă toate restricțiile problemei
- Dacă nici una dintre valorile corespunzătoare unei componente nu conduce la o soluție parțială validă atunci **se revine la componenta anterioară** și se încearcă altă valoare pentru aceasta.

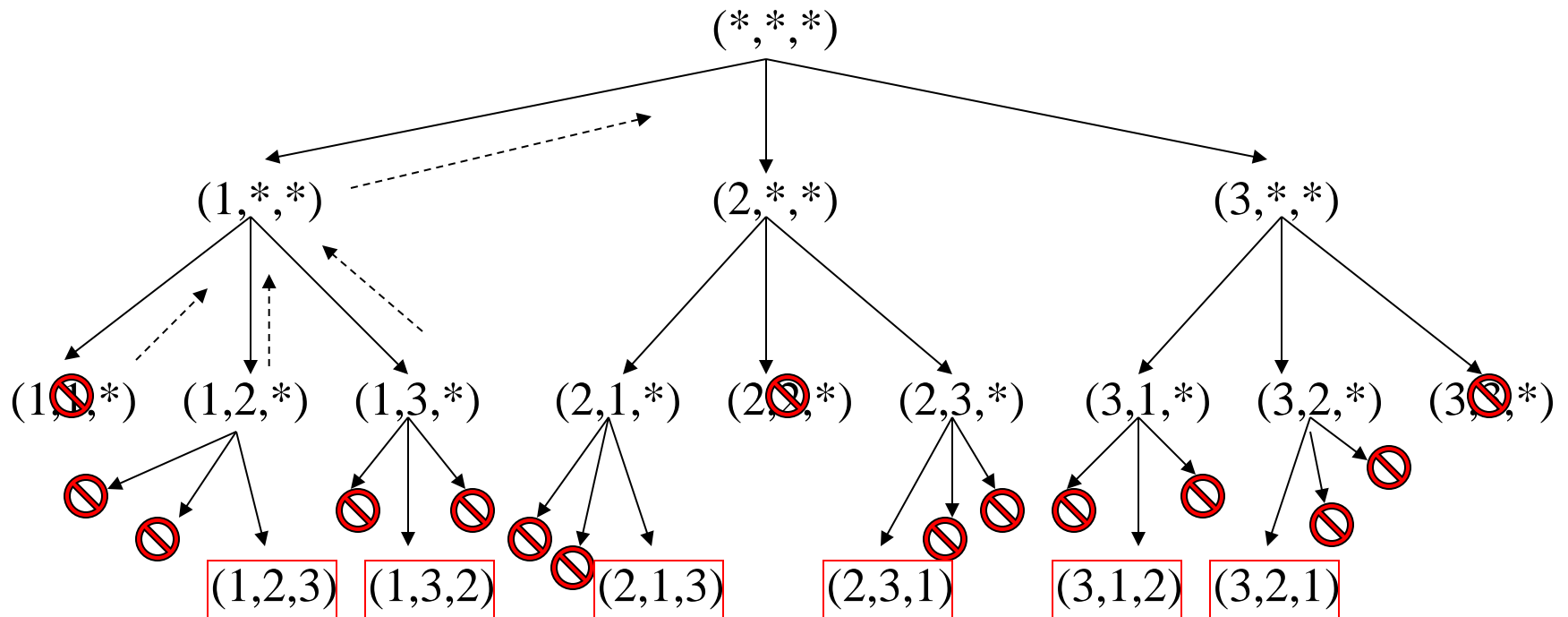
Ce este tehnica căutării cu revenire?

Ideea de bază:

- Principiul tehnicii poate fi vizualizat folosind un **arbore de căutare** care ilustrează parcurgerea spațiului soluțiilor:
 - Rădăcina arborelui corespunde stării inițiale (înainte de a începe completarea componentelor)
 - Un nod intern corespunde unei soluții parțiale valide
 - Un nod extern (frunză) corespunde fie unei soluții parțiale invalide fie unei soluții finale

Ce este tehnica căutării cu revenire?

Exemplu: arborele de parcurgere a spațiului soluțiilor în cazul problemei de generare a permutărilor ($n=3$)



Structura generală a algoritmului

Etape în proiectarea algoritmului:

1. Se alege modul de **reprezentare a soluțiilor**
2. **Se identifică spațiul de căutare și se stabilesc mulțimile A_1, \dots, A_n și ordinea în care acestea sunt parcurse**
3. Din restricțiile problemei se deduc condițiile pe care trebuie să le satisfacă soluțiile parțiale pentru a fi valide. Aceste condiții sunt denumite **condiții de continuare**
4. Se stabilește criteriul în baza căruia se decide dacă o **soluție parțială este soluție finală**

Structura generală a algoritmului

Exemplu: generarea permutărilor

1. **Reprezentarea soluției:** fiecare permutare este un vector $s=(s_1,s_2,\dots,s_n)$ care satisface $s_i \neq s_j$ pentru orice $i \neq j$
2. **Mulțimile A_1,\dots,A_n :** $\{1,2,\dots,n\}$. Fiecare mulțime este parcursă în ordinea naturală a elementelor (de la 1 la n)
3. **Condiții de continuare:** o soluție parțială (s_1,s_2,\dots,s_k) trebuie să satisfacă $s_k \neq s_i$ pentru orice $i < k$
4. **Criteriu pentru a decide când o soluție parțială este soluție finală:** $k=n$ (au fost completate toate cele n componente)

Structura generală a algoritmului

Notății:

(s_1, s_2, \dots, s_k) soluție parțială

k – indice în s

$$A_k = \{a_{1k}^k, \dots, a_{m_k k}^k\}$$

$$m_k = \text{card}\{A_k\}$$

i_k - indice in A_k

Structura generală a algoritmului

Se caută o valoare pt componenta k care conduce la o soluție parțială validă

Dacă o astfel de valoare există atunci se verifică dacă nu s-a ajuns la o soluție finală

Dacă s-a găsit o soluție atunci aceasta se procesează și se trece la următoarea valoare

Dacă nu e soluție finală se trece la următoarea componentă

Dacă nu mai există nici o valoare neanalizată pt componenta curentă se revine la componenta anterioară

```
Backtracking( $A_1, A_2, \dots, A_n$ )
```

```
 $k \leftarrow 1; i_k \leftarrow 0$ 
```

```
WHILE  $k > 0$  DO
```

```
   $i_k \leftarrow i_k + 1$ 
```

```
   $v \leftarrow \text{False}$ 
```

```
  WHILE  $v = \text{False}$  AND  $i_k \leq m_k$  DO
```

```
     $s_k \leftarrow a_{i_k}^k$ 
```

```
    IF “ $(s_1, \dots, s_k)$  e validă” THEN  $v \leftarrow \text{True}$ 
```

```
    ELSE  $i_k \leftarrow i_k + 1$  ENDIF ENDWHILE
```

```
  IF  $v = \text{True}$  THEN
```

```
    IF “ $(s_1, \dots, s_k)$  este soluție finală”
```

```
      THEN “prelucrează soluția”
```

```
    ELSE  $k \leftarrow k + 1; i_k \leftarrow 0$  ENDIF
```

```
  ELSE  $k \leftarrow k - 1$  ENDIF
```

```
ENDWHILE
```

Structura generală a algoritmului

Varianta recursivă:

- Presupunem ca A_1, \dots, A_n și s sunt variabile globale
- Fie k componenta care urmează a fi completată

Algoritmul se apelează prin
BT_rec(1)

Se încearcă fiecare valoare posibilă

```
BT_rec(k)
IF "(s1, ..., sk-1) este soluție"
  THEN "se prelucrează soluția"
  ELSE
    FOR j ← 1, mk DO
      sk ← akj
      IF "(s1, ..., sk) este validă"
        THEN BT_rec(k+1) ENDIF
    ENDFOR
  ENDIF
```

Se completează următoarea componentă

Aplicație 1: generarea permutărilor

```
Backtracking( $A_1, A_2, \dots, A_n$ )
 $k \leftarrow 1$ ;  $i_k \leftarrow 0$ 
WHILE  $k > 0$  DO
   $i_k \leftarrow i_k + 1$ 
   $v \leftarrow \text{False}$ 
  WHILE  $v = \text{False}$  AND  $i_k \leq m_k$  DO
     $s_k \leftarrow a_{ik}^k$ 
    IF “( $s_1, \dots, s_k$ ) e valida” THEN  $v \leftarrow \text{True}$ 
    ELSE  $i_k \leftarrow i_k + 1$  ENDIF ENDWHILE
  IF  $v = \text{True}$  THEN
    IF “( $s_1, \dots, s_k$ ) e solutie finala”
      THEN “prelucreaza solutia”
    ELSE  $k \leftarrow k + 1$ ;  $i_k \leftarrow 0$  ENDIF
  ELSE  $k \leftarrow k - 1$  ENDIF
ENDWHILE
```

```
permutari(n)
 $k \leftarrow 1$ ;  $s[k] \leftarrow 0$ 
WHILE  $k > 0$  DO
   $s[k] \leftarrow s[k] + 1$ 
   $v \leftarrow \text{False}$ 
  WHILE  $v = \text{False}$  AND  $s[k] \leq n$  DO
    IF  $\text{valid}(s[1..k])$ 
      THEN  $v \leftarrow \text{True}$ 
      ELSE  $s[k] \leftarrow s[k] + 1$ 
    ENDWHILE
  IF  $v = \text{True}$  THEN
    IF  $k = n$ 
      THEN  $\text{WRITE } s[1..n]$ 
      ELSE  $k \leftarrow k + 1$ ;  $s[k] \leftarrow 0$ 
    ELSE  $k \leftarrow k - 1$ 
  ENDIF ENDIF ENDWHILE
```

Aplicație 1: generarea permutărilor

Funcție care verifică dacă o soluție parțială este validă

```
valid(s[1..k])
FOR i ← 1,k-1 DO
  IF s[k]=s[i]
    THEN RETURN FALSE
  ENDIF
ENDFOR
RETURN TRUE
```

Varianta recursiva:

```
perm_rec(k)
IF k=n+1 THEN WRITE s[1..n]
ELSE
  FOR i ← 1,n DO
    s[k] ← i
    IF valid(s[1..k])=True
      THEN perm_rec(k+1)
    ENDIF
  ENDFOR
ENDIF
```

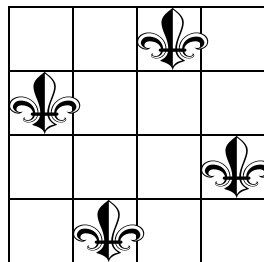
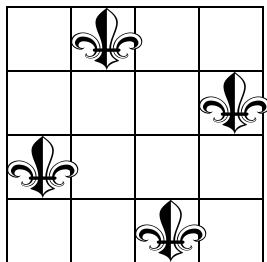
Aplicație 2: problema reginelor

Să se determine toate posibilitățile de a plasa n regine pe o tablă de șah de dimensiune $n \times n$ astfel încât acestea să nu se atace reciproc:

- fiecare **linie** conține exact o regină
- fiecare **coloană** conține exact o regină
- fiecare **diagonală** conține cel mult o regină

Obs. Este o problemă clasică propusă de către Max Bezzel (1850) și studiată de către mai mulți matematicieni ai vremii (Gauss, Cantor)

Exemplu: dacă $n \leq 3$ nu are soluție; dacă $n=4$ sunt 2 soluții



Pe măsură ce n crește, numărul soluțiilor crește (pt $n=8$ sunt 92 soluții)

Aplicație 2: problema reginelor

1. **Reprezentarea soluției:** vom considera că regina k este întotdeauna plasată pe linia k . Astfel pentru fiecare regină este suficient să se identifice coloana pe care va fi plasată. Soluția va fi astfel reprezentată printr-un tablou (s_1, \dots, s_n) unde $s_k =$ indicele coloanei pe care se plasează regina k
2. **Mulțimile A_1, \dots, A_n :** $\{1, 2, \dots, n\}$. Fiecare mulțime va fi prelucrată în **ordinea naturală** a elementelor (de la 1 la n)
3. **Condiții de continuare:** o soluție parțială (s_1, s_2, \dots, s_k) trebuie să satisfacă restricțiile problemei (nu mai mult de o regină pe fiecare linie, coloană sau diagonală)
4. **Criteriu pentru a decide dacă o soluție parțială este finală:** $k = n$ (toate reginele au fost plasate)

Aplicație 2: problema reginelor

Condiții de continuare: Fie (s_1, s_2, \dots, s_k) o soluție parțială. Aceasta este validă dacă satisface:

- Reginele se află pe linii diferite – condiția este implicit satisfăcută datorită modului de reprezentare utilizat (regina i este întotdeauna plasată pe linia i)

- Reginele se afla pe coloane diferite:

$$s_i \neq s_j \text{ pentru orice } i \neq j$$

(este suficient să se verifice că $s_k \neq s_i$ pentru orice $1 \leq i \leq k-1$)

- Reginele se află pe diagonale diferite:

$$|i-j| \neq |s_i - s_j| \text{ pentru orice } i \neq j$$

(este suficient să se verifice $|k-i| \neq |s_k - s_i|$ pentru orice $1 \leq i \leq k-1$)

Aplicație 2: problema reginelor

Obs:

Două regine i și j se află pe aceeași diagonală dacă

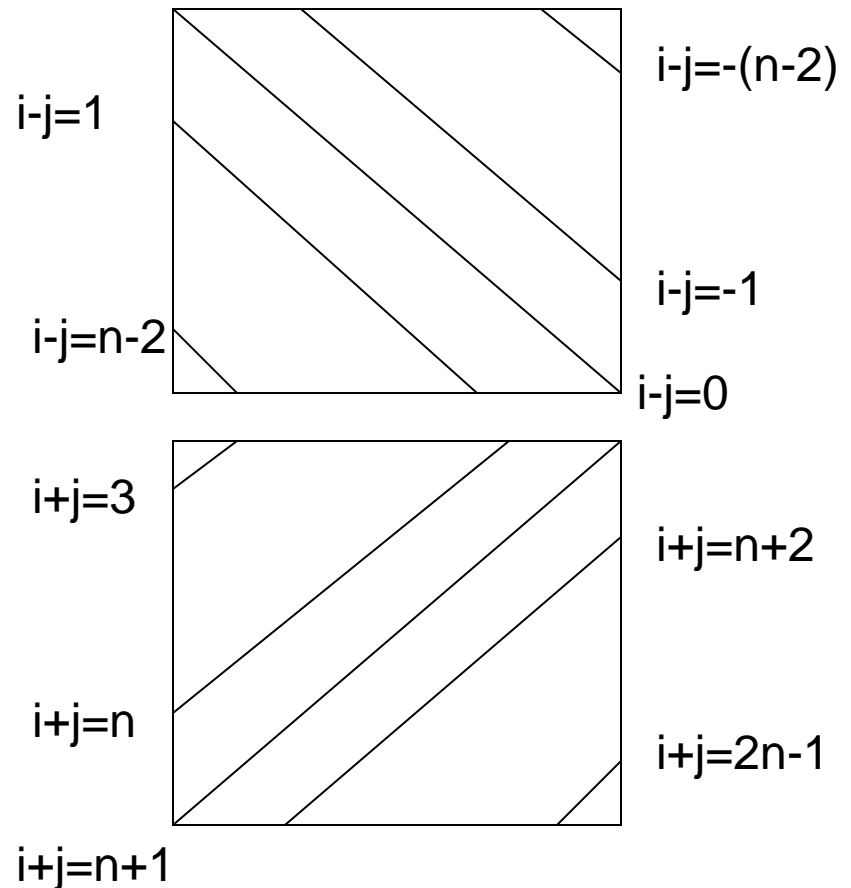
$$i-s_i=j-s_j \Leftrightarrow i-j = s_i-s_j$$

sau

$$i+s_i=j+s_j \Leftrightarrow i-j = s_j-s_i$$

Aceasta înseamnă că

$$|i-j|=|s_i-s_j|$$



Aplicație 2: problema reginelor

Algoritm:

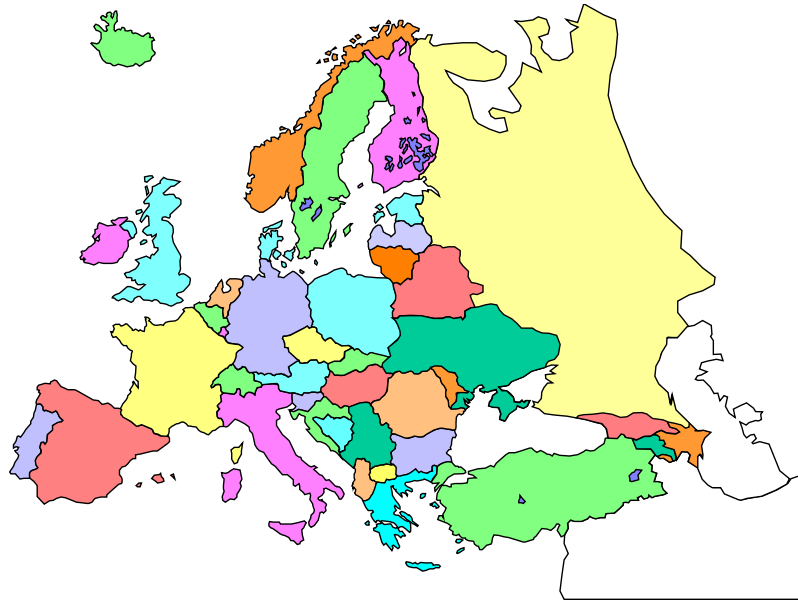
```
Validare(s[1..k])
FOR i ← 1,k-1 DO
  IF s[k]=s[i] OR |i-k|=|s[i]-s[k]|
    THEN RETURN False
  ENDIF
ENDFOR
RETURN True
```

Apel: regine(1)

```
regine(k)
IF k=n+1 THEN WRITE s[1..n]
ELSE
  FOR i ← 1,n DO
    s[k] ← i
    IF Validare(s[1..k])=True
      THEN regine(k+1)
    ENDIF
  ENDFOR
ENDIF
```

Aplicație 3: colorarea hărților

Problema: Considerăm o hartă geografică care conține n țări. Având la dispoziție $4 \leq m < n$ culori să se asigneze o culoare fiecărei țări astfel încât oricare două țări vecine să fie colorate diferit.



Problema matematică: orice hartă poate fi colorată utilizând cel mult 4 culori (rezultat demonstrat în 1976 de către Appel and Haken – unul dintre primele rezultate obținute folosind tehnica demonstrării asistate de calculator)

Aplicație 3: colorarea hărților

Problema: Considerăm o hartă geografică care conține n țări. Având la dispoziție $4 \leq m < n$ culori să se asigneze o culoare fiecărei țări astfel încât oricare două țări vecine să fie colorate diferit.

Formalizarea problemei: Considerăm că relația de vecinătate între țări este specificată printr-o matrice N având elementele:

$$N(i,j) = \begin{cases} 0 & \text{dacă } i \text{ și } j \text{ nu sunt țări vecine} \\ 1 & \text{dacă } i \text{ și } j \text{ sunt țări vecine} \end{cases}$$

Scopul problemei este să se determine un tablou $S=(s_1, \dots, s_n)$ cu s_k în $\{1, \dots, m\}$ specificând culoarea asociată țării k și astfel încât pentru toate perechile (i,j) cu $N(i,j)=1$ elementele s_i și s_j sunt diferite ($s_i \neq s_j$)

Aplicație 3: colorarea hărților

1. Reprezentarea soluției

$S=(s_1, \dots, s_n)$ unde s_k culoarea asociată țării k

2. Mulțimile $A_1, \dots, A_n : \{1, 2, \dots, m\}$. Fiecare mulțime va fi prelucrată în ordinea naturală a elementelor (de la 1 la m)

3. Condiții de continuare: o soluție parțială (s_1, s_2, \dots, s_k) trebuie să satisfacă $s_i \neq s_j$ pentru toate perechile (i, j) pentru care $N(i, j)=1$

Pentru fiecare k este suficient să se verifice ca $s_k \neq s_j$ pentru toate valorile i în $\{1, 2, \dots, k-1\}$ satisfacând $N(i, k)=1$

4. Criteriu pentru a decide dacă o soluție parțială este soluție finală: $k = n$ (toate țările au fost colorate)

Aplicație 3: colorarea hărților

Algoritm recursiv

```
Colorare(k)
IF k=n+1 THEN WRITE s[1..n]
ELSE
  FOR j ← 1,m DO
    s[k] ← j
    IF valid(s[1..k])=True
      THEN Colorare(k+1)
    ENDIF
  ENDFOR
ENDIF
```

Algoritm de validare

```
valid(s[1..k])
FOR i ← 1,k-1 DO
  IF N[i,k]=1 AND s[i]=s[k]
    THEN RETURN False
  ENDIF
ENDFOR
RETURN True
```

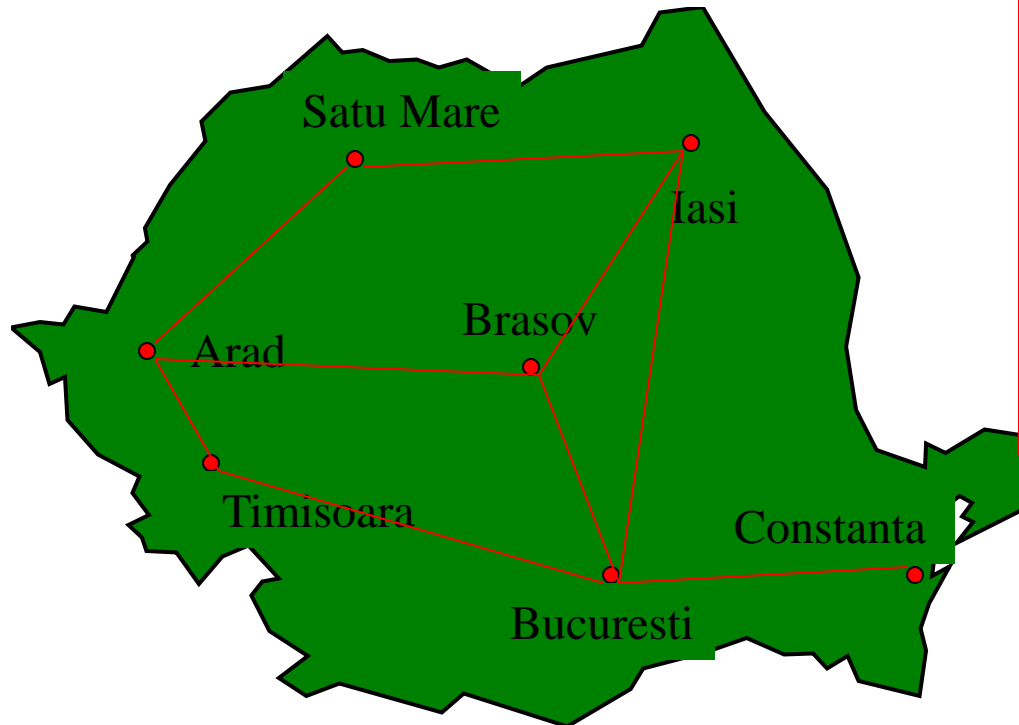
Apel: Colorare(1)

Aplicație 4: găsirea traseelor

Considerăm un set de n orașe și o rețea de drumuri care le conectează. Să se determine toate traseele care conectează două orașe date (un traseu nu poate trece de două ori prin același oraș).

Orașe:

1. Arad
2. Brașov
3. București
4. Constanța
5. Iași
6. Satu-Mare
7. Timișoara



Trasee de la Arad
la Constanța

1->7->3->4

1->2->3->4

1->6->5->3->4

1->6->5->2->3->4

Aplicație 4: găsirea traseelor

Formalizarea problemei: Rețeaua de șosele este specificată prin matricea C:

$$C(i,j) = \begin{cases} 0 & \text{nu există drum direct de la orașul } i \text{ la orașul } j \\ 1 & \text{există drum direct de la orașul } i \text{ la orașul } j \end{cases}$$

Să se găsească toate traseele $S=(s_1, \dots, s_m)$ - unde s_k în $\{1, \dots, n\}$ specifică orașul vizitat la etapa k - astfel încât

s_1 este orașul sursă

s_m este orașul destinație

$s_i \neq s_j$ pentru orice $i \neq j$ (printr-un oraș se trece o singură dată)

$C(s_k, s_{k+1})=1$ (există drum direct între orașele vizitate la momente consecutive de timp)

Aplicație 4: găsirea traseelor

1. Reprezentarea soluției

$S=(s_1,\dots,s_m)$ cu s_k reprezentând orașul vizitat la etapa k
(m = numărul de etape; nu este cunoscut de la început)

2. Mulțimile $A_1,\dots,A_n : \{1,2,\dots,n\}$. Fiecare mulțime va fi prelucrată în ordinea naturală a elementelor (de la 1 la n)

3. Condiții de continuare: o soluție parțială (s_1,s_2,\dots,s_k) trebuie să satisfacă:

$s_k \neq s_j$ pentru orice j în $\{1,2,\dots,k-1\}$ (orașele sunt distincte)

$C(s_{k-1},s_k)=1$ (se poate trece direct de la s_{k-1} la s_k)

4. Criteriul pentru a decide dacă o soluție parțială este soluție finală:

$s_k =$ oraș destinație

Aplicație 4: găsirea traseelor

Algoritm recursiv

```
trasee(k)
IF s[k-1]="oras destinatie" THEN
    WRITE s[1..k-1]
ELSE
    FOR j ← 1,n DO
        s[k] ← j
        IF valid(s[1..k])=True
            THEN trasee(k+1)
        ENDIF
    ENDFOR
ENDIF
```

Algoritm de validare

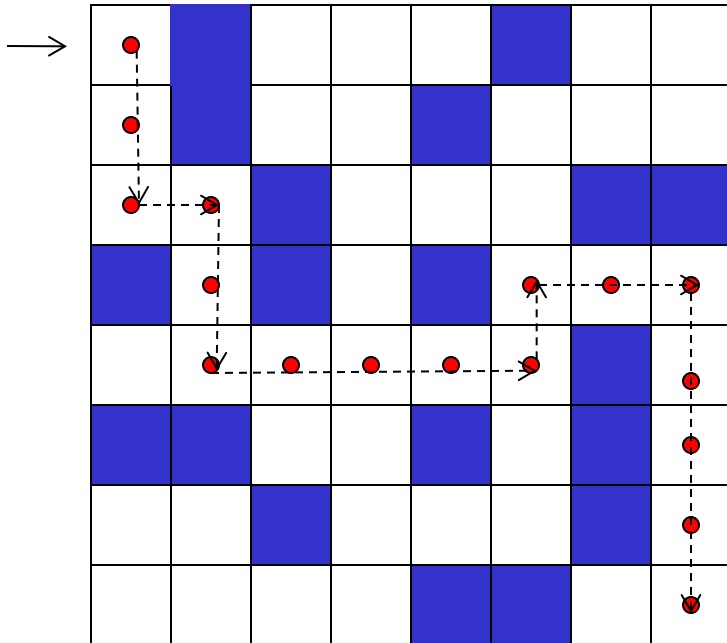
```
valid(s[1..k])
IF C[s[k-1],s[k]]=0 THEN
    RETURN False
ENDIF
FOR i ← 1,k-1 DO
    IF s[i]=s[k]
        THEN RETURN False
    ENDIF
ENDFOR
RETURN True
```

Apel:

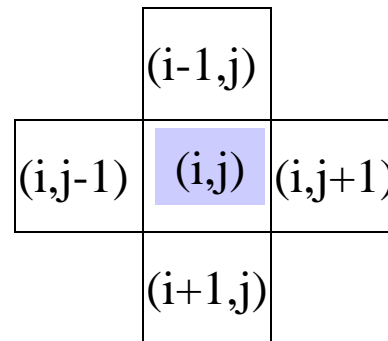
```
s[1] ← oraș sursă
trasee(2)
```

Aplicație 5: problema labirintului

Problema. Se consideră un labirint definit pe o grilă $n \times n$. Să se găsească toate traseele care pornesc din $(1,1)$ și se termină în (n,n)



Obs. Doar celulele libere pot fi vizitate.
Dintr-o celulă (i,j) se poate trece în una dintre celulele vecine:



Obs: celulele de pe frontieră au mai puțini vecini

Aplicație 5: problema labirintului

Formalizarea problemei. Labirintul este stocat într-o matrice $n \times n$

$$M(i,j) = \begin{cases} 0 & \text{celulă liberă} \\ 1 & \text{celulă ocupată} \end{cases}$$

Să se găsească $S=(s_1, \dots, s_m)$ cu s_k în $\{1, \dots, n\} \times \{1, \dots, n\}$ indicii corespunzători celulei vizitate la etapa k

- s_1 este celula de start: $(1,1)$
- s_m este celula destinație: (n,n)
- $s_k \neq s_q$ pentru orice $k \neq q$ (o celulă este vizitată cel mult o dată)
- $M(s_k)=0$ (doar celulele libere pot fi vizitate)
- s_{k-1} și s_k sunt celule vecine

Aplicație 5: problema labirintului

1. Reprezentarea soluției

$S=(s_1, \dots, s_n)$ cu s_k reprezentând indicii celulei vizitate la etapa k

2. Mulțimile A_1, \dots, A_n sunt submulțimi ale mulțimii

$\{1, 2, \dots, n\} \times \{1, 2, \dots, n\}$. Pentru fiecare celulă (i, j) există un set de 4 vecini: $(i, j-1)$, $(i, j+1)$, $(i-1, j)$, $(i+1, j)$

3. Condiții de continuare: o soluție parțială (s_1, s_2, \dots, s_k) trebuie să satisfacă:

$s_k \neq s_q$ pentru orice q în $\{1, 2, \dots, k-1\}$

$M(s_k)=0$

s_{k-1} și s_k sunt celule vecine

4. Condiția ca o soluție parțială (s_1, \dots, s_k) să fie soluție finală:

$s_k = (n, n)$

Aplicatie 5: problema labirintului

labirint(k)

IF s[k-1].i=n and s[k-1].j=n THEN WRITE s[1..k]

ELSE // se încearcă toți vecinii

s[k].i ← s[k-1].i-1; s[k].j ← s[k-1].j

IF valid(s[1..k])=True THEN labirint (k+1) ENDIF

s[k].i ← s[k-1].i+1; s[k].j ← s[k-1].j

IF valid(s[1..k])=True THEN labirint (k+1) ENDIF

s[k].i ← s[k-1].i; s[k].j ← s[k-1].j-1

IF valid(s[1..k])=True THEN labirint (k+1) ENDIF

s[k].i ← s[k-1].i-1; s[k].j ← s[k-1].j+1

IF valid(s[1..k])=True THEN labirint (k+1) ENDIF

ENDIF

Obs:

s[k] este o structură
cu două câmpuri:

s[k].i reprezintă
prima componentă a
perechii de indici

s[k].j reprezintă a
doua componentă a
perechii de indici

Aplicație 5: problema labirintului

```
valid(s[1..k])
IF s[k].i<1 OR s[k].i>n OR s[k].j<1 OR s[k].j>n // în afara grilei
    THEN RETURN False
ENDIF
IF M[s[k].i,s[k].j]=1 THEN RETURN False ENDIF // celulă ocupată
FOR q ← 1,k-1 DO // celulă deja vizitată
    IF s[k].i=s[q].i AND s[k].j=s[q].j THEN RETURN False ENDIF
ENDFOR
RETURN True
```

Apel algoritm labirint:

s[1].i:=1; s[1].j:=1

labirint(2)

Curs următor: recapitulare

Tematica examen:

- Algoritmi iterativi: proiectare și analiza complexitate
- Algoritmi recursivi: descriere și analiza complexitate
- Tehnici de reducere și divizare: descriere și analiza complexitate; aplicații
- Algoritmi de sortare:
 - algoritmi elementari (inserție, selecție, interschimbare)
 - algoritmi eficienți (sortare prin interclasare, sortare rapidă)
- Căutare local optimală (greedy) – aplicații
- Programare dinamică – aplicații
- Căutare cu revenire (backtracking) – aplicații