



UNIVERSITATEA BABEȘ-BOLYAI  
Facultatea de Matematică și Informatică



# INTELIGENȚĂ ARTIFICIALĂ

**Rezolvarea problemelor de căutare**

Strategii de căutare neinformată

Laura Dioșan

# Sumar

---

## A. Scurtă introducere în Inteligența Artificială (IA)

## B. Rezolvarea problemelor prin căutare

- Definirea problemelor de căutare
- Strategii de căutare
  - Strategii de căutare neinformate
  - Strategii de căutare informate
  - Strategii de căutare locale (Hill Climbing, Simulated Annealing, Tabu Search, Algoritmi evolutivi, PSO, ACO)
  - Strategii de căutare adversială

## C. Sisteme inteligente

- Sisteme care învață singure
  - Arbori de decizie
  - Rețele neuronale artificiale
  - Mașini cu suport vectorial
  - Algoritmi evolutivi
- Sisteme bazate pe reguli
- Sisteme hibride

# Sumar

---

- Probleme
- Rezolvarea problemelor
  - Pași în rezolvarea problemelor
- Rezolvarea problemelor prin căutare
  - Pași în rezolvarea problemelor prin căutare
  - Tipuri de strategii de căutare

# Materiale de citit și legături utile

---

- ❑ capitolele I.1, I.2, II.3 și II.4 din *S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Prentice Hall, 1995*
- ❑ capitolele 1 - 4 din *C. Groșan, A. Abraham, Intelligent Systems: A Modern Approach, Springer, 2011*
- ❑ capitolele 2.1 – 2.5 din <http://www-g.eng.cam.ac.uk/mmg/teaching/artificialintelligence/>

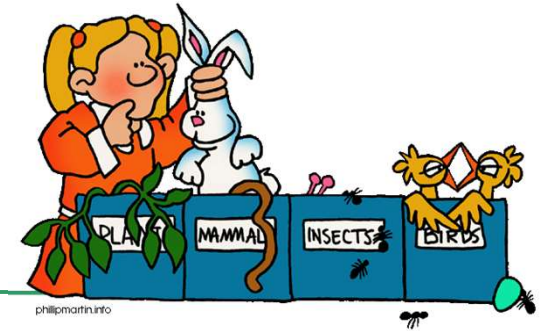
# Probleme

---



- Două mari categorii de probleme:
  - Rezolvabile în mod determinist
    - Calculul sinusului unui unghi sau a rădăcinii pătrate dintr-un număr
  - Rezolvabile în mod stocastic
    - Probleme din lumea reală → proiectarea unui ABS
    - Presupun căutarea unei soluții → metode ale IA

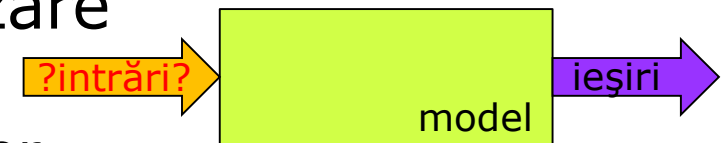
# Probleme



## □ Tipologie

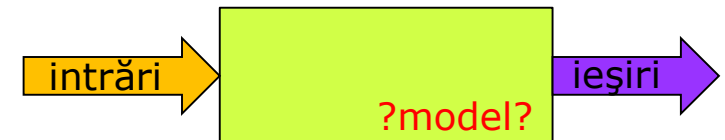
### ■ Probleme de căutare/optimizare

- Planificare, proiectarea sateliților



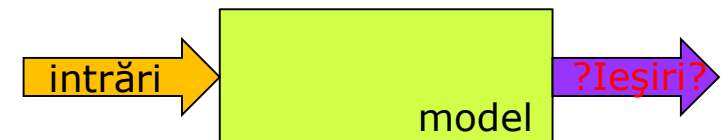
### ■ Probleme de modelare

- Predicții, clasificări



### ■ Probleme de simulare

- Teoria jocurilor economice



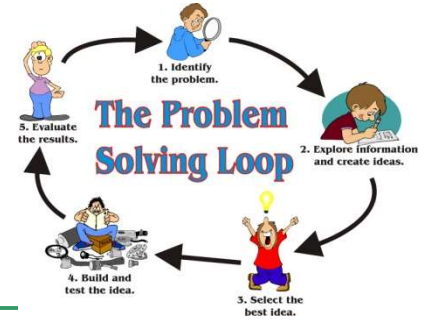
# Rezolvarea problemelor

---



- Constă în identificarea unei soluții
  - În informatică (IA) → proces de căutare
  - În inginerie și matematică → proces de optimizare
  
- Cum?
  - Reprezentarea soluțiilor (parțiale) → puncte în spațiul de căutare
  - Proiectarea unor operatori de căutare → transformă o posibilă soluție în altă soluție

# Pași în rezolvarea problemelor



- ❑ Definirea problemei
- ❑ Analiza problemei
- ❑ Alegerea unei tehnici de rezolvare
  - căutare
  - reprezentarea cunoștințelor
  - abstractizare



# Rezolvarea problemelor prin căutare

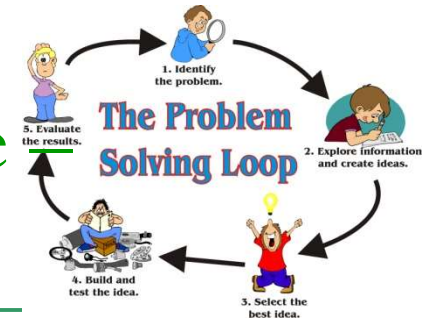
---



- bazată pe urmărirea unor obiective
- compusă din acțiuni care duc la îndeplinirea unor obiective
  - fiecare acțiune modifică o anumită stare a problemei
- succesiune de acțiuni care transformă starea inițială a problemei în stare finală

# Pași în rezolvarea problemelor prin căutare

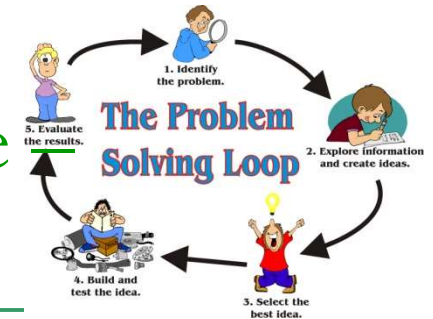
## Definirea problemei



- Definirea problemei implică stabilirea:
  - unui spațiu de stări
    - toate configurațiile posibile, fără enumerarea obligatorie a tuturor configurațiilor
    - reprezentare
      - explicită – construirea (în memorie) a tuturor stărilor posibile
      - implicită – utilizarea unor structuri de date și a unor funcții (operatori)
  - unei/unor stări inițiale
  - unei/unor stări finale - obiectiv
  - unui/unor drum(uri)
    - succesiuni de stări
  - unui set de reguli (acțiuni)
    - funcții succesori (operatori) care precizează starea următoare a unei stări
    - funcții de cost care evaluează
      - trecerea dintr-o stare în alta
      - un întreg drum
    - funcții obiectiv care verifică dacă s-a ajuns într-o stare finală

# Pași în rezolvarea problemelor prin căutare

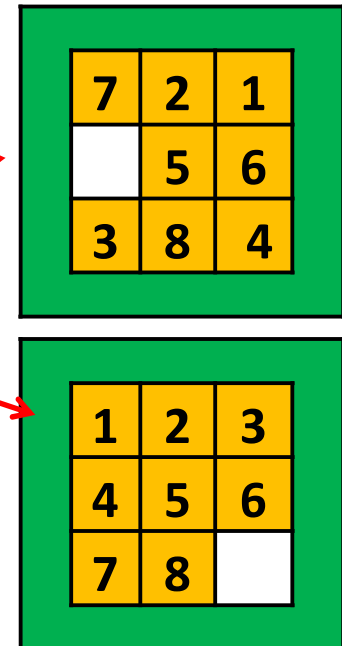
## Definirea problemei



### ■ Exemple

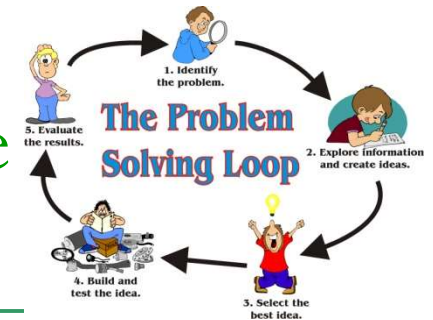
#### ■ Joc puzzle cu 8 piese

- Spațiul stărilor – configurații ale tablei de joc cu 8 piese
- Starea inițială – o configurație oarecare
- Starea finală – o configurație cu piesele aranjate într-o anumită ordine
- Reguli → acțiuni albe
  - Condiții: mutarea în interiorul tablei
  - Transformări: spațiul alb se mișcă în sus, în jos, la stânga sau la dreapta
- Soluția – secvența optimă de acțiuni albe



# Pași în rezolvarea problemelor prin căutare

## Definirea problemei



### Exemple

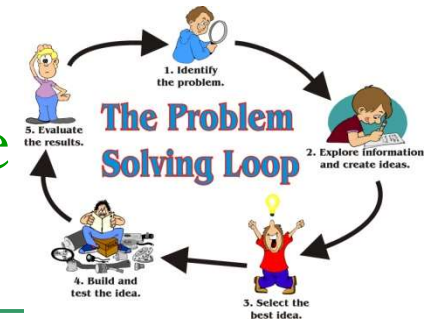
#### Joc cu $n$ dame

- Spațiul stărilor – configurații ale tablei de joc cu  $n$  regine
- Starea inițială – o configurație fără regine
- Starea finală – o configurație cu  $n$  regine care nu se atacă
- Reguli → amplasarea unei regine pe tablă
  - Condiții: regina amplasată nu este atacată de nici o regină existentă pe tablă
  - Transformări: amplasarea unei noi regine într-o căsuță de pe tabla de joc
- Soluția – amplasarea optimă a reginelor pe tablă

	a	b	c	d	e	f	g	h	
1				♔					1
2							♔		2
3			♔						3
4								♔	4
5		♔							5
6					♔				6
7		♔							7
8						♔			8
	a	b	c	d	e	f	g	h	

# Pași în rezolvarea problemelor prin căutare

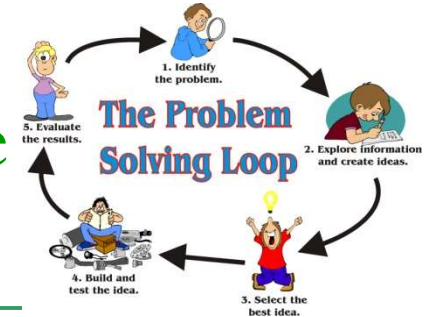
## Analiza problemei



- Se poate descompune problema?
  - Sub-problemele sunt independente sau nu?
- Universul stărilor posibile este predictibil?
- Se dorește obținerea oricărei soluții sau a unei soluții optime?
- Soluția dorită constă într-o singură stare sau într-o succesiune de stări?
- Sunt necesare multiple cunoștințe pentru a limita căutarea sau chiar pentru a identifica soluția?
- Problema este conversațională sau solitară?
  - Este sau nu nevoie de interacțiune umană pentru rezolvarea ei?

# Pași în rezolvarea problemelor prin căutare

## Alegerea unei tehnici de rezolvare

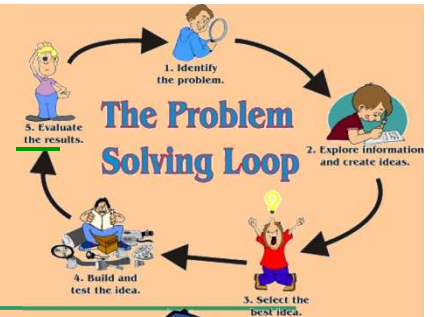


- Rezolvarea prin utilizarea regulilor (în combinație cu o strategie de control) de deplasare în spațiul problemei până la găsirea unui drum între starea inițială și cea finală
- Rezolvare prin căutare
  - Examinarea sistematică a stărilor posibile în vederea identificării
    - unui drum de la o stare inițială la o stare finală
    - unei stări optime
  - Spațiul stărilor = toate stările posibile + operatorii care definesc legăturile între stări



# Pași în rezolvarea problemelor prin căutare

## Alegerea unei tehnici de rezolvare

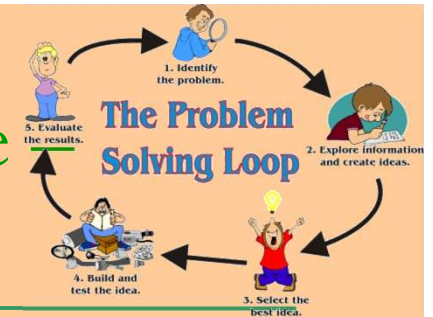


### □ Rezolvare prin căutare

- Strategii de căutare multiple → cum alegem o strategie?
  - Complexitatea computațională (temporală și spațială)
  - Completitudine → algoritmul se sfârșește întotdeauna și găsește o soluție (dacă ea există)
  - Optimalitate → algoritmul găsește soluția optimă (costul optim al drumului de la starea inițială la starea finală)

# Pași în rezolvarea problemelor prin căutare

## Alegerea unei tehnici de rezolvare



### □ Rezolvare prin căutare

#### ■ Strategii de căutare multiple → cum alegem o strategie?

- Complexitatea computațională (temporală și spațială)
  - Performanța strategiei depinde de:
    - Timpul necesar rulării
    - Spațiul (memoria) necesară rulării
    - Mărimea intrărilor algoritmului
    - Viteza calculatorului
    - Calitatea compilerului
- Se măsoară cu ajutorul complexității → Eficiență computațională
  - Spațială → memoria necesară identificării soluției
    - $S(n)$  – cantitatea de memorie utilizată de cel mai bun algoritm A care rezolvă o problemă de decizie  $f$  cu  $n$  date de intrare
  - Temporală → timpul necesar identificării soluției
    - $T(n)$  – timpul de rulare (numărul de pași) al celui mai bun algoritm A care rezolvă o problemă de decizie  $f$  cu  $n$  date de intrare

} Factori interni

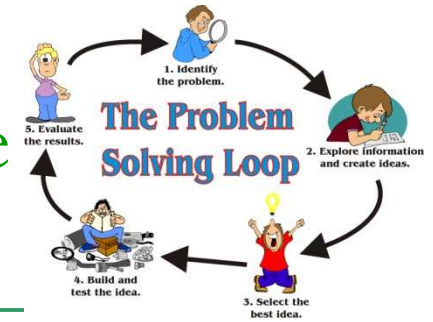
} Factori externi





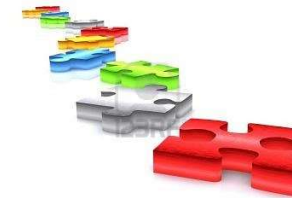
# Pași în rezolvarea problemelor prin căutare

## Alegerea unei tehnici de rezolvare



□ Rezolvarea problemelor prin căutare poate consta în:

■ Construirea progresivă a soluției

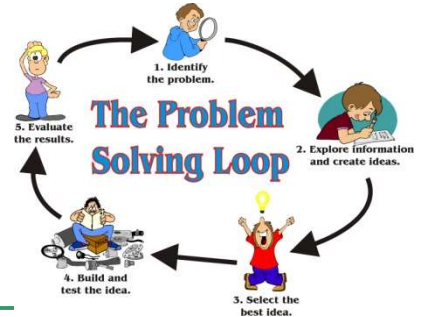


■ Identificarea soluției potențiale optime

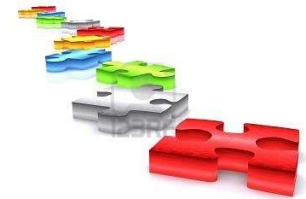


# Pași în rezolvarea problemelor prin căutare

## Alegerea unei tehnici de rezolvare

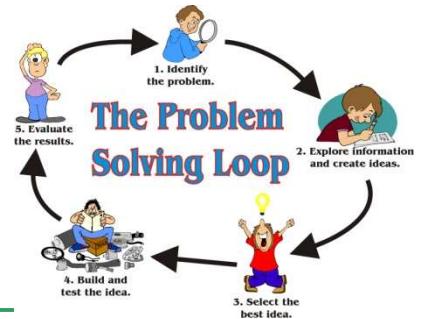


- Rezolvarea problemelor prin căutare poate consta în:
  - Construirea progresivă a soluției
    - Componentele problemei
      - Stare inițială
      - Operatori (funcții succesori)
      - Stare finală
      - Soluția = un drum (de cost optim) de la starea inițială la starea finală
    - Spațiul de căutare
      - Mulțimea tuturor stărilor în care se poate ajunge din starea inițială prin aplicarea operatorilor
      - stare = o componentă a soluției
    - Exemple
      - Problema comisului voiajor
    - Algoritmi
      - Ideea de bază: se începe cu o componentă a soluției și se adaugă noi componente până se ajunge la o soluție completă
      - Recursivi → se re-aplică până la îndeplinirea unei condiții
      - Istoricul căutării (drumul parcurs de la starea inițială la starea finală) este reținut în liste de tip LIFO sau FIFO
    - Avantaje
      - Nu necesită cunoștințe (informații inteligente)



# Pași în rezolvarea problemelor prin căutare

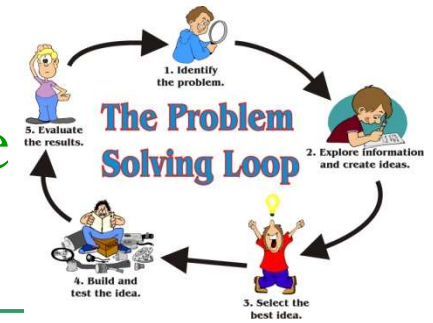
## Alegerea unei tehnici de rezolvare



- Rezolvarea problemelor prin căutare poate consta în:
  - Identificarea soluției potențiale optime
    - Componentele problemei
      - Condiții (constrângeri) pe care trebuie să le satisfacă (parțial sau total) soluția
      - Funcție de evaluare a unei soluții potențiale → identificarea optimului
    - Spațiul de căutare
      - mulțimea tuturor soluțiilor potențiale complete
      - Stare = o soluție completă
    - Exemple
      - Problema celor 8 regine
    - Algoritmi
      - Ideea de bază: se începe cu o stare care nu respectă anumite constrângeri pentru a fi soluție optimă și se efectuează modificări pentru a elimina aceste violări
      - Iterativi → se memorează o singură stare și se încearcă îmbunătățirea ei
      - Istoricul căutării nu este reținut
    - Avantaje
      - Simplu de implementat
      - Necesită puțină memorie
      - Poate găsi soluții rezonabile în spații de căutare (continue) foarte mari pentru care alți algoritmi sistematici nu pot fi aplicați

# Pași în rezolvarea problemelor prin căutare

## Alegerea unei tehnici de rezolvare



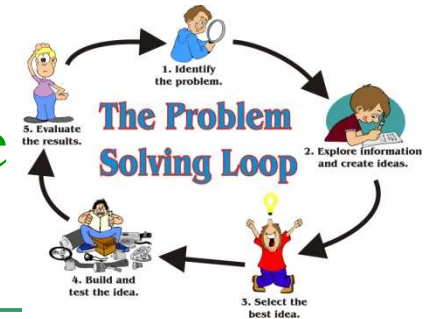
### □ Rezolvarea problemelor prin căutare presupune

- algoritmi cu o complexitate ridicată (probleme NP-complete)
- căutarea într-un spațiu exponențial



# Pași în rezolvarea problemelor prin căutare

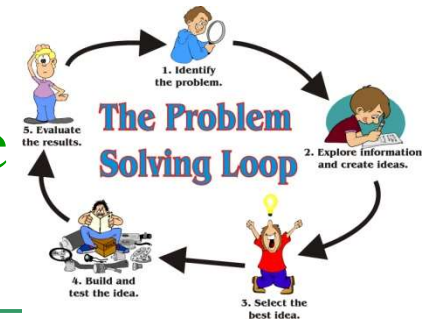
## Alegerea unei tehnici de rezolvare



- Tipologia strategiilor de căutare
  - În funcție de modul de **generare** a soluției
    - Căutare **constructivă**
      - Construirea progresivă a soluției
      - Ex. TSP
    - Căutare **perturbativă**
      - O soluție candidat este modificată în vederea obținerii unei noi soluții candidat
      - Ex. SAT - Propositional Satisfiability Problem
  - În funcție de modul de **traversare** a spațiului de căutare
    - Căutare **sistematică**
      - Traversarea completă a spațiului
        - Identificarea soluției dacă ea există → algoritmi compleți
    - Căutare **locală**
      - Traversarea spațiului de căutare dintr-un punct în alt punct vecin → algoritmi incompleți
      - O stare a spațiului poate fi vizitată de mai multe ori
  - În funcție de elementele de **certitudine** ale căutării
    - Căutare **deterministă**
      - Algoritmi de identificare exactă a soluției
    - Căutare **stocastică**
      - Algoritmi de aproximare a soluției
  - În funcție de stilul de **explorare** a spațiului de căutare
    - Căutare **secvențială**
    - Căutare **paralelă**

# Pași în rezolvarea problemelor prin căutare

## Alegerea unei tehnici de rezolvare

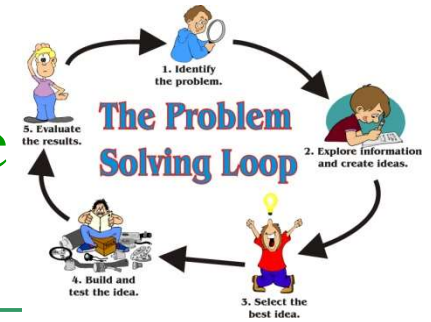


### Tipologia strategiilor de căutare

- În funcție de **scopul** urmărit
  - Căutare **uni-obiectiv**
    - Soluția trebuie să satisfacă o singură condiție/constrângere
  - Căutare **multi-obiectiv**
    - Soluția trebuie să satisfacă mai multe condiții (obiective)
- În funcție de **numărul de soluții optime**
  - Căutare **uni-modală**
    - Există o singură soluție optimă
  - Căutare **multi-modală**
    - Există mai multe soluții optime
- În funcție de tipul de **algoritm** folosit
  - Căutare de-a lungul unui **număr finit de pași**
  - Căutare **iterativă**
    - Algoritmii converg către soluție
  - Căutare **euristică**
    - Algoritmii oferă o aproximare a soluției
- În funcție de **mecanismul** căutării
  - Căutare **tradițională**
  - Căutare **modernă**
- În funcție de **locul** în care se desfășoară căutarea în spațiul de căutare
  - Căutare **locală**
  - Căutare **globală**

# Pași în rezolvarea problemelor prin căutare

## Alegerea unei tehnici de rezolvare



### Tipologia strategiilor de căutare

#### □ În funcție de tipul **(linearitatea) constrângerilor**

##### ■ Căutare **liniară**

##### ■ Căutare **neliniară**

###### □ Clasică (deterministă)

- Directă – bazată doar pe evaluarea funcției obiectiv
- Indirectă – bazată și pe derivata (I și/sau II) a funcției obiectiv

###### □ Enumerativă

- În funcție de modul de stabilire a soluției
  - Ne-informată → soluția se știe doar când ea coincide cu starea finală
  - Informată → se lucrează cu o funcție de evaluare a unei soluții parțiale
- În funcție de tipul spațiului de căutare
  - Completă – spațiu finit (dacă soluția există, ea poate fi găsită)
  - Incompletă – spațiu infinit

###### □ Stocastică

- Bazată pe elemente aleatoare

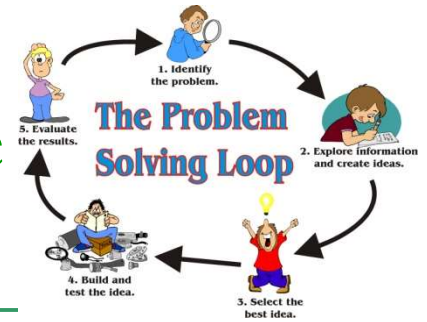
#### □ În funcție de **agenții** implicați în căutare

##### ■ Căutare realizată de un **singur agent** → fără piedici în atingerea obiectivelor

##### ■ Căutare **adversială** → adversarul aduce o incertitudine în realizarea obiectivelor

# Pași în rezolvarea problemelor prin căutare

## Alegerea unei tehnici de rezolvare



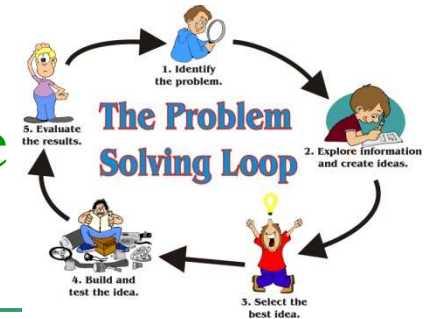
### Exemplu

- Tipologia strategiilor de căutare
  - În funcție de modul de generare a soluției
    - **Căutare constructivă**
    - Căutare perturbativă
  - În funcție de modul de traversare a spațiului de căutare
    - **Căutare sistematică**
    - Căutare locală
  - În funcție de elementele de certitudine ale căutării
    - **Căutare deterministă**
    - Căutare stocastică
  - În funcție de stilul de explorare a spațiului de căutare
    - **Căutare secvențială**
    - Căutare paralelă



# Pași în rezolvarea problemelor prin căutare

## Alegerea unei tehnici de rezolvare



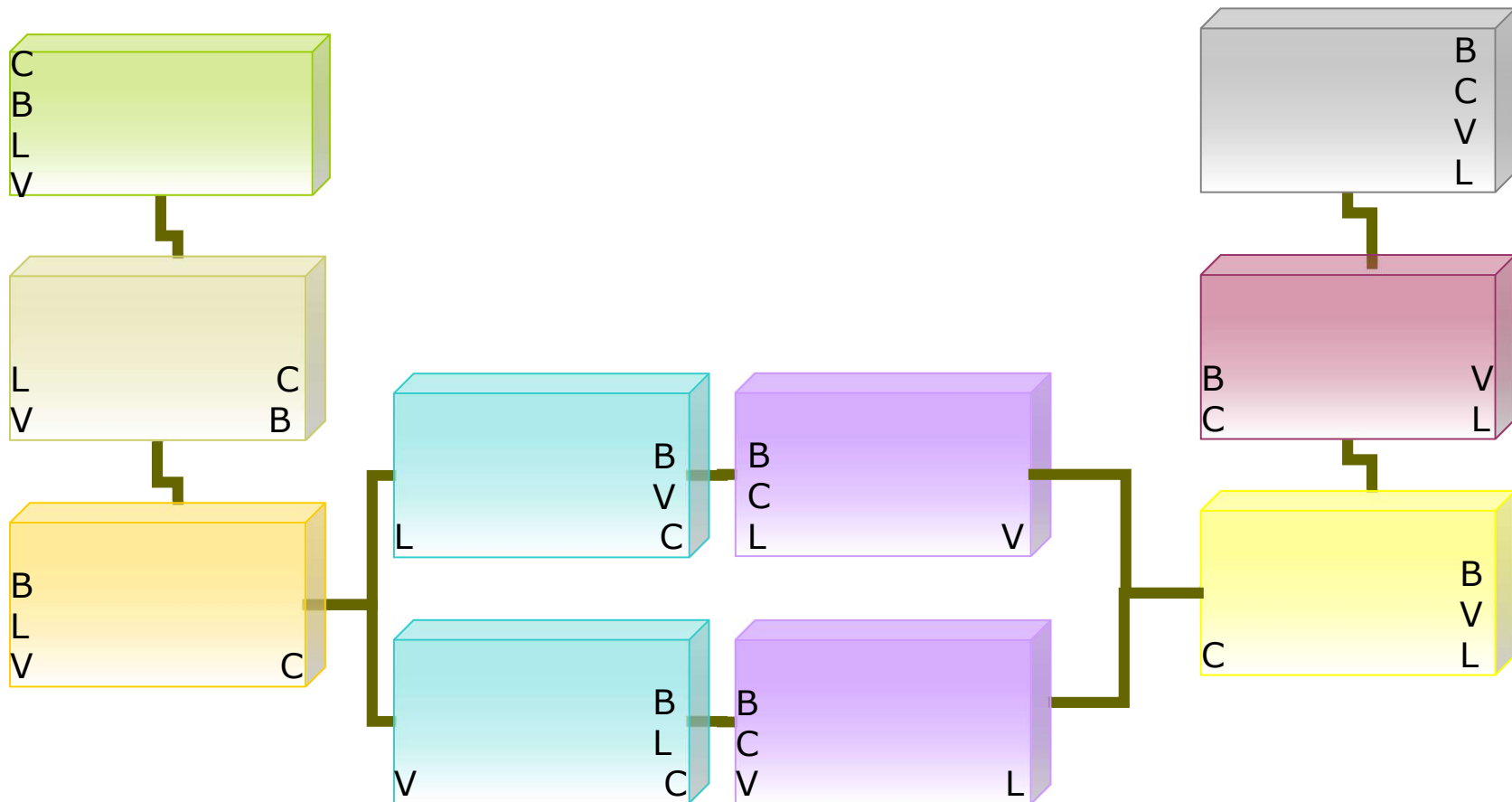
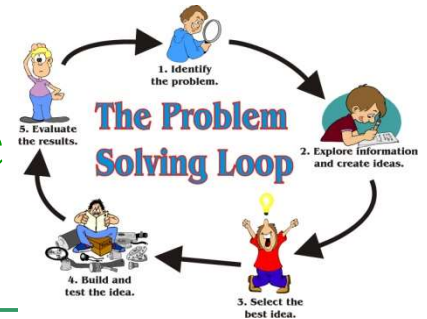
### Exemplu

#### ❑ Problema "capra, varza și lupul"

- Se dau:
  - ❑ o capră, o varză și un lup pe malul unui râu
  - ❑ o barcă cu barcagiu
- Se cere
  - ❑ Să se traverseze toți pasagerii pe malul celălalt al râului
  - ❑ cu următoarele condiții
    - în barcă există doar 2 locuri
    - nu pot rămâne pe același mal
      - capra și varza
      - lupul și capra

# Pași în rezolvarea problemelor prin căutare

## Alegerea unei tehnici de rezolvare



# Strategii de căutare – Elemente fundamentale

---



- Tipuri abstracte de date (TAD)
  - TAD listă → structură liniară
  - TAD arbore → structură arborescentă (ierarhică)
  - TAD graf → structură de graf
  
- TAD
  - Domeniu și operații
  - Reprezentare

# Strategii de căutare –

## Elemente fundamentale – TAD Listă



- Domeniu
  - $D = \{l \mid l = (el_1, el_2, \dots), \text{ unde } el_i, i=1,2,3,\dots, \text{ sunt de același tip } TE \text{ (tip element) și fiecare element } el_i, i=1,2,3,\dots, \text{ are o poziție unică în } l \text{ de tip } TP \text{ (TipPoziție)}\}$
- Operații
  - Creare( $l$ )
  - Prim( $l$ )
  - Ultim( $l$ )
  - Următor( $l, p$ )
  - Anterior( $l, p$ )
  - Valid( $l, p$ )
  - getElement( $l, p$ )
  - getPozitie( $l, e$ )
  - Modifică( $l, p, e$ )
  - AdăugareLaÎnceput( $l, e$ )
  - AdăugareLaSfârșit( $l, e$ )
  - AdăugareDupă( $l, p, e$ )
  - AdăugareÎnainte( $l, p, e$ )
  - Eliminare( $l, p$ )
  - Căutare( $l, e$ )
  - Vidă( $l$ )
  - Dimensiune( $l$ )
  - Distrugere( $l$ )
  - getIterator( $l$ )
- Reprezentare
  - Vectorială
  - Liste (simplu sau dublu) înlanțuite, etc
- Cazuri particulare
  - Stivă – LIFO
  - Coadă – FIFO
  - Coadă cu priorități



# Strategii de căutare –

## Elemente fundamentale – TAD Listă



- Domeniu
  - $D = \{I \mid I = (el_1, el_2, \dots), \text{ unde } el_i, i=1,2,3,\dots, \text{ sunt de același tip } TE \text{ (tip element) și fiecare element } el_i, i=1,2,3,\dots, \text{ are o poziție unică în } I \text{ de tip } TP \text{ (TipPozitie)}\}$
- Operații
  - Creare(I)
  - Prim(I)
  - Ultim(I)
  - Următor(I,p)
  - Anterior(I,p)
  - Valid(I,p)
  - getElement(I,p)
  - getPozitie(I,e)
  - Modifică(I,p,e)
  - AdăugareLaÎnceput(I,e)
  - AdăugareLaSfârșit(I,e)
  - AdăugareDupă(I,p,e)
  - AdăugareÎnainte(I,p,e)
  - Eliminare(I,p)
  - Căutare(I,e)
  - Vidă(I)
  - Dimensiune(I)
  - Distrugere(I)
  - getIterator(I)
- Reprezentare
  - Vectorială
  - Liste (simplu sau dublu) înlanțuite, etc
- Cazuri particulare
  - Stivă – LIFO
  - Coadă – FIFO
  - Coadă cu priorități

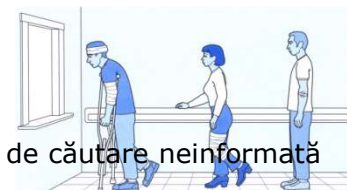


# Strategii de căutare –

## Elemente fundamentale – TAD Listă



- Domeniu
  - $D = \{I \mid I = (el_1, el_2, \dots), \text{ unde } el_i, i=1,2,3,\dots, \text{ sunt de același tip } TE \text{ (tip element) și fiecare element } el_i, i=1,2,3,\dots, \text{ are o poziție unică în } I \text{ de tip } TP \text{ (TipPozitie)}\}$
- Operații
  - Creare(I)
  - Prim(I)
  - Ultim(I)
  - Următor(I,p)
  - Anterior(I,p)
  - Valid(I,p)
  - getElement(I,p)
  - getPozitie(I,e)
  - Modifică(I,p,e)
  - AdăugareLaÎnceput(I,e)
  - AdăugareLaSfârșit(I,e)
  - AdăugareDupă(I,p,e)
  - AdăugareÎnainte(I,p,e)
  - Eliminare(I,p)
  - Căutare(I,e)
  - Vidă(I)
  - Dimensiune(I)
  - Distrugere(I)
  - getIterator(I)
- Reprezentare
  - Vectorială
  - Liste (simplu sau dublu) înlanțuite, etc
- Cazuri particulare
  - Stivă – LIFO
  - Coadă – FIFO
  - Coadă cu priorități



# Strategii de căutare –

## Elemente fundamentale – TAD Graf

---



- Domeniu – container de noduri si legături între noduri
  - $D = \{nod_1, nod_2, \dots, nod_n, leg_1, leg_2, \dots, leg_m\}$ , unde  $nod_i$ , cu  $i=1,2,\dots,n$  sunt noduri, iar  $leg_i$ , cu  $i=1,2,\dots,m$  sunt muchii între noduri}
- Operații
  - creare
  - creareNod
  - traversare
  - getIterator
  - distrugere
- Reprezentare
  - Lista muchilor
  - Lista de adiacență (Tradițională și Modernă)
  - Matricea de adiacență (Tradițională și Modernă)
  - Matricea de incidență
- Cazuri particulare
  - Grafuri orientate și neorientate
  - Grafuri simple sau multiple
  - Grafuri conexe sau nu
  - Grafuri complete sau nu
  - Grafuri cu sau fără cicluri (aciclice → păduri, arbori)

# Strategii de căutare –

## Elemente fundamentale – TAD Arbore

---



- Domeniu – container de noduri si legături între noduri
  - $D = \{nod_1, nod_2, \dots, nod_n, leg_1, leg_2, \dots, leg_m\}$ , unde  $nod_i$ , cu  $i=1,2,\dots,n$  sunt noduri, iar  $leg_i$ , cu  $i=1,2,\dots,m$  sunt muchii între noduri astfel încât să nu se formeze cicluri}
- Operații
  - creare
  - creareFrunză
  - adăugareSubarbore
  - getInfoRădăcină
  - getSubarbore
  - traversare
  - getIterator
  - distrugere
- Reprezentare
  - Vectorială
  - Liste înlănțuite ale descendenților
- Cazuri particulare
  - Arbori binari (de căutare)
  - Arbori n-ari



# Strategii de căutare – Elemente fundamentale – parcurgerea grafelor

---



- Drumuri
  - drum (*path*)
    - nodurile nu se pot repeta
  - *trail*
    - muchiile nu se pot repeta
  - *walk*
    - fără restricții
  - drum închis
    - nodul inițial = nodul final
  - circuit
    - un *trail* închis
  - ciclu
    - un *path* închis

# Strategii de căutare neinformate (SCnI)



- Caracteristici
  - nu se bazează pe informații specifice problemei
  - sunt generale
  - strategii oarbe
  - strategii de tipul forței brute
- Topologie
  - În funcție de ordinea expandării stărilor în spațiul de căutare:
    - SCnI în structuri liniare
      - căutare liniară
      - căutare binară
    - SCnI în structuri ne-liniare
      - căutare în lățime (breadth-first)
        - căutare de cost uniform (branch and bound)
      - căutare în adâncime (depth-first)
        - căutare în adâncime limitată (limited depth-first)
        - căutare în adâncime iterativă (iterative deepening depth-first)
      - căutare bidirecțională

# SCnI în structuri liniare

## Căutare liniară



### ❑ Aspecte teoretice

- Se verifică fiecare element al unei liste până la identificarea celui dorit
- Lista de elemente poate fi ordonată sau nu

### ❑ Exemplu

- Lista = ( 2, 3, 1, ,7, 5)
- Elem = 7

### ❑ Algoritm

```
bool LS(elem, list){
    found = false;
    i = 1;
    while ((!found) && (i <= list.length)){
        if (elem == list[i])
            found = true;
        else
            i++;
    } //while
    return found;
}
```

# SCnI în structuri liniare

## Căutare liniară



### □ Analiza căutării

- Complexitate temporală
  - Cel mai bun caz:  $elem = list[1] \Rightarrow O(1)$
  - Cel mai slab caz:  $elem \notin list \Rightarrow T(n) = n + 1 \Rightarrow O(n)$
  - Cazul mediu:  $T(n) = (1 + 2 + \dots + n + (n+1))/(n+1) \Rightarrow O(n)$
- Complexitate spațială
  - $S(n) = n$
- Completitudine
  - da
- Optimalitate
  - da

### □ Avantaje

- Simplitate, complexitate temporală bună pentru structuri mici
- Structura nu trebuie sortată în prealabil

### □ Dezavantaje

- complexitate temporală foarte mare pentru structuri mari

### □ Aplicații

- Căutări în baze de date reale

# SCnI în structuri liniare

## Căutare binară



### □ Aspecte teoretice

- Localizarea unui element într-o listă ordonată
- Strategie de tipul *Divide et Conquer*

### □ Exemplu

- List = ( 2, 3, 5, 6, 8, 9, 13, 16, 18), Elem = 6
  - List = ( 2, 3, 5, 6, 8, 9, 13, 16, 18)
  - List = ( 2, 3, 5, 6)
  - List = ( 5, 6)
  - List = ( 6 )

### □ Algoritm

```
bool BS(elem, list){
    found = false;
    left = 1;
    right = list.length;
    while((left < right) && (!found)){
        middle = left + (right - left)/2;
        if (element == list[middle])
            found = true;
        else
            if (element < list[middle])
                right = middle - 1;
            else
                left = middle + 1;
    } //while
    return found;
}
```

# SCnI în structuri liniare

## Căutare binară



### □ Analiza căutării

- Complexitate temporală  $T(n) = 1$ , pt  $n = 1$  și  $T(n) = T(n/2) + 1$ , altfel  
Pp. că  $n = 2k \Rightarrow k = \log_2 n$   
Pp. că  $2k < n < 2k+1 \Rightarrow k < \log_2 n < k + 1$   
$$\begin{array}{lcl} T(n) & = & T(n/2) + 1 \\ T(n/2) & = & T(n/2^2) + 1 \\ & \dots & \\ T(n/2^{k-1}) & = & T(n/2^k) + 1 \\ \hline T(n) & = & k + 1 = \log_2 n + 1 \end{array}$$
- Complexitate spațială –  $S(n) = n$
- Completitudine – da
- Optimalitate – da

### □ Avantaje

- Complexitate temporală redusă față de căutarea liniară

### □ Dezavantaje

- Lucrul cu vectori (trebuie accesate elemente indexate) sortați

### □ Aplicații

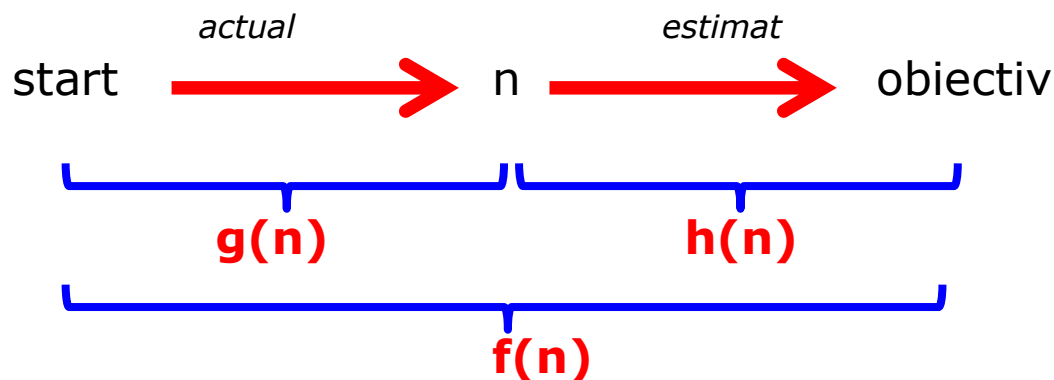
- Jocul ghicirii unui număr
- Căutare într-o carte de telefon/dicționar

# SC în structuri arborescente



## ■ Noțiuni necesare

- $f(n)$  – funcție de evaluare pentru estimarea costului soluției prin nodul (starea)  $n$
- $h(n)$  – funcție euristică pentru estimarea costului drumului de la nodul  $n$  la nodul obiectiv
- $g(n)$  – funcție de cost pentru estimarea costului drumului de la nodul de start până la nodul  $n$
- $f(n) = g(n) + h(n)$



# SCnI în structuri arborescente

## căutare în lățime (breadth-first search – BFS)

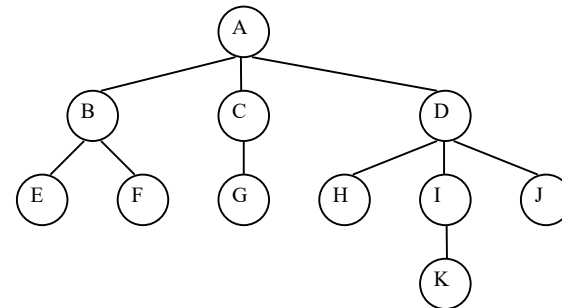


### Aspecte teoretice

- Toate nodurile aflate la adâncimea  $d$  se expandează înaintea nodurilor aflate la adâncimea  $d+1$
- Toate nodurile fii obținute prin expandarea nodului curent se adaugă într-o *listă* de tip **FIFO** (*coadă*)

### Exemplu

- Ordinea vizitării: A, B, C, D, E, F, G, H, I, J, K



### Algoritm

```

bool BFS(elem, list){
    found = false;
    visited = Φ;
    toVisit = {start};          //FIFO list
    while((toVisit != Φ) && (!found)){
        node = pop(toVisit);
        visited = visited U {node};
        if (node == elem)
            found = true;
        else{
            aux = Φ;
            for all (unvisited) children of node do
                aux = aux U {child};
            }
            toVisit = toVisit U aux;
        }
    } //while
    return found;
}
    
```

Vizitate deja	De vizitat
Φ	A
A	B, C, D
A, B	C, D, E, F
A, B, C	D, E, F, G
A, B, C, D	E, F, G, H, I, J
A, B, C, D, E	F, G, H, I, J
A, B, C, D, E, F	G, H, I, J
A, B, C, D, E, F, G	H, I, J
A, B, C, D, E, F, G, H	I, J
A, B, C, D, E, F, G, H, I	J, K
A, B, C, D, E, F, G, H, I, J	K
A, B, C, D, E, F, G, H, I, J, K	Φ



# SCnI în structuri arborescente

## căutare în lățime (breadth-first search – BFS)



### Analiza căutării:

- Complexitate temporală:
  - $b$  – factor de ramificare (nr de noduri fii ale unui nod)
  - $d$  – lungimea (adâncimea) soluției
  - $T(n) = 1 + b + b^2 + \dots + b^d \Rightarrow O(b^d)$
- Complexitate spațială
  - $S(n) = T(n)$
- Completitudine
  - Dacă soluția există, atunci BFS o găsește
- Optimalitate
  - nu

### Avantaje

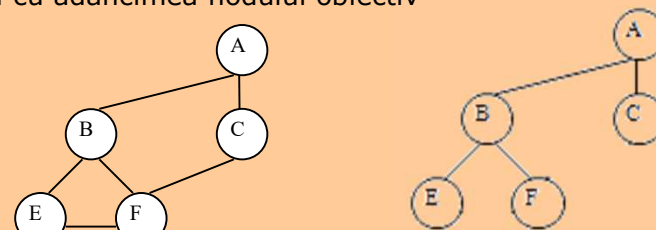
- Găsirea drumului de **lungime** minimă până la nodul obiectiv (soluția cea mai puțin adâncă)

### Dezavantaje

- Generarea și stocarea unui arbore a cărui mărime crește exponențial cu adâncimea nodului obiectiv
- Complexitate temporală și spațială exponențială
- Experimentul Russel&Norvig????
- Funcțional doar pentru spații de căutare mici

### Aplicații

- Identificarea tuturor componentelor conexe într-un graf
- Identificarea celui mai scurt drum într-un graf
- Optimizări în rețele de transport → algoritmul Ford-Fulkerson
- Serializarea/deserializarea unui arbore binar (vs. serializarea în mod sortat) permite reconstrucția eficientă a arborelui
- Copierea colecțiilor (garbage collection) → algoritmul Cheney



Vizitate deja	De vizitat
$\Phi$	B
B	A, E, F
B, A	E, F, C
B, A, E	F, C
B, A, E, F	C
B, A, E, F, C	$\Phi$

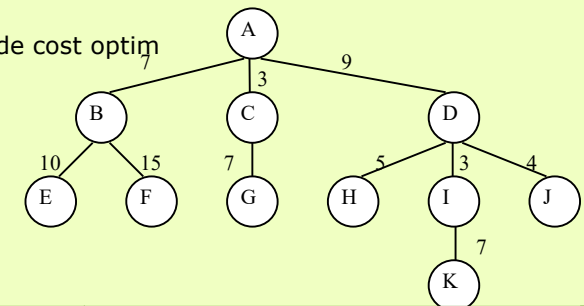
# SCnI în structuri arborescente

## căutare de cost uniform (uniform cost search – UCS)



### Aspecte teoretice

- BFS + procedură specială de expandare a nodurilor (bazată pe costurile asociate legăturilor dintre noduri)
- Toate nodurile de la adâncimea  $d$  sunt expandate înaintea nodurilor de la adâncimea  $d+1$
- Toate nodurile fii obținute prin expandarea nodului curent se adaugă într-o **listă ORDONATĂ** de tip **FIFO**
  - Se expandează mai întâi nodurile de cost minim
  - Odată obținut un drum până la nodul țintă, acesta devine candidat la drumul de cost optim
- Algoritmul *Branch and bound*



### Exemplu

- Ordinea vizitării nodurilor: A, C, B, D, G, E, F, I, H, J, K

### Algoritm

```
bool UCS(elem, list){
    found = false;
    visited = Φ;
    toVisit = {start};    //FIFO sorted list
    while((toVisit != Φ) && (!found)){
        node = pop(toVisit);
        visited = visited U {node};
        if (node== elem)
            found = true;
        else
            aux = Φ;
            for all (unvisited) children of node do{
                aux = aux U {child};
            } // for
            toVisit = toVisit U aux;
            TotalCostSort(toVisit);
    } //while
    return found;
}
```

visited	toVisit
Φ	A
A	C(3), B(7), D(9)
A, C	B(7), D(9), G(3+7)
A, C, B	D(9), G(10), E(7+10), F(7+15)
A, C, B, D	G(10), I(9+3), J(9+4), H(9+5), E(17), F(22)
A, C, B, D, G	I(12), J(13), H(14), E(17), F(22)
A, C, B, D, G, I	J(13), H(14), E(17), F(22), <b>K(9+3+7)</b>
A, C, B, D, G, I, J	H(14), E(17), F(22), <b>K(19)</b>
A, C, B, D, G, I, J, H	E(17), F(22), <b>K(19)</b>
A, C, B, D, G, I, J, H, E	F(22), <b>K(19)</b>
A, C, B, D, G, I, J, H, E, F	K(19)
A, C, B, D, G, I, J, H, E, F, K	Φ

# SCnI în structuri arborescente

## căutare de cost uniform (uniform cost search – UCS)

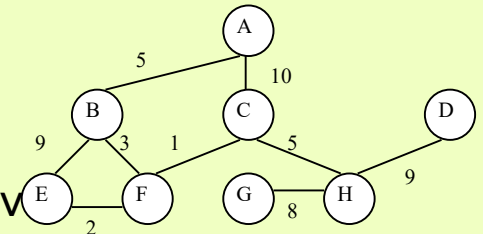


### ■ Analiza complexității

- Complexitate temporală:
  - $b$  – factor de ramificare (nr de noduri fii ale unui nod)
  - $d$  - lungimea (adâncimea) soluției
  - $T(n) = 1 + b + b^2 + \dots + b^d \Rightarrow O(b^d)$
- Complexitate spațială
  - $S(n) = T(n)$
- Completitudine
  - Da – dacă soluția există, atunci UCS o găsește
- Optimalitate
  - Da

### ■ Avantaje

- Găsirea drumului de **cost** minim până la nodul obiectiv



### ■ Dezavantaje

- Complexitate temporală și spațială exponențială

### ■ Aplicații

- Cel mai scurt drum → algoritmul Dijkstra

Vizitate deja	De vizitat
$\Phi$	A(0)
A(0)	B(5), C(10)
A(0), B(5)	F(8), C(10), E(14)
A(0), B(5), F(8)	C(9), E(10)
A(0), B(5), F(8), C(9)	E(10), H(14)
A(0), B(5), F(8), C(9), E(10)	H(14)

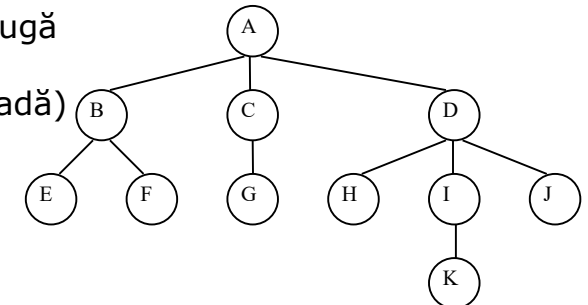
# SCnI în structuri arborescente

## căutare în adâncime (depth-first search – DFS)



### Aspecte teoretice

- Expandarea într-un nod fiu și înaintarea în adâncime până când
  - Este găsit un nod țintă (obiectiv) sau
  - Nodul atins nu mai are fii
- Cu revenirea în cel mai recent nod care mai poate fi explorat
- Toate nodurile fii obținute prin expandarea nodului curent se adaugă într-o listă de tip **LIFO (stivă)**
- Similar cu BFS, dar nodurile se plasează într-o stivă (în loc de coadă)



### Exemplu

- Ordinea vizitării nodurilor: A, B, E, F, C, G, D, H, I, K, J

### Algoritm

```

bool DFS(elem, list){
    found = false;
    visited =  $\Phi$ ;
    toVisit = {start};          //LIFO list
    while((toVisit !=  $\Phi$ ) && (!found)){
        node = pop(toVisit);
        visited = visited U {node};
        if (node== elem)
            found = true;
        else{
            aux =  $\Phi$ ;
            for all (unvisited) children of node do{
                aux = aux U {child};
            }
            toVisit = aux U toVisit;
        }
    }
    //while
    return found;
}
    
```

Vizitate deja	De vizitat
$\Phi$	A
A	B, C, D
A, B	E, F, C, D
A, B, E	F, C, D
A, B, E, F	C, D
A, B, E, F, C	G, D
A, B, E, F, C, G	D
A, B, E, F, C, G, D	H, I, J
A, B, E, F, C, G, D, H	I, J
A, B, E, F, C, G, D, H, I	K, J
A, B, E, F, C, G, D, H, I, K	J
A, B, E, F, C, G, D, H, I, K, J	$\Phi$

# SCnI în structuri arborescente

## căutare în adâncime (depth-first search – DFS)



### Analiza complexității

- Complexitate temporală:
  - $b$  – factor de ramificare (nr de noduri fii ale unui nod)
  - $d_{max}$  – lungimea (adâncimea) maximă a unui arbore explorat
  - $T(n) = 1 + b + b^2 + \dots + b^{d_{max}} \Rightarrow O(b^{d_{max}})$
- Complexitate spațială
  - $S(n) = b * d_{max}$
- Completitudine
  - Nu → algoritmul nu se termină pt drumurile infinite (neexistând suficientă memorie pt reținerea nodurilor deja vizitate)
- Optimalitate
  - Nu → căutarea în adâncime poate găsi un drum soluție mai lung decât drumul optim

### Avantaje

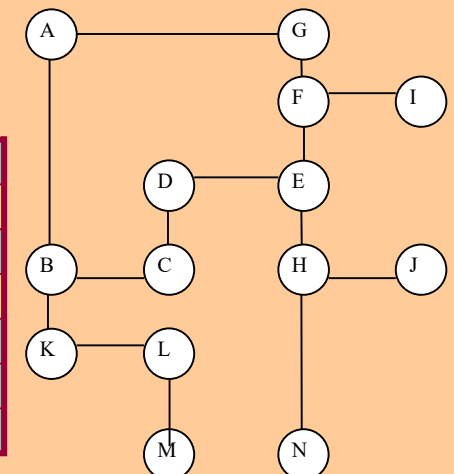
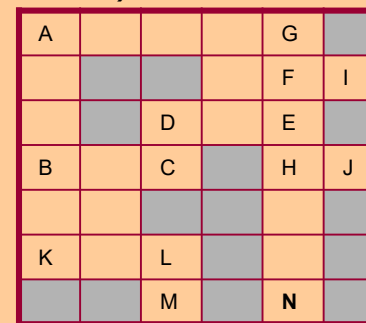
- Găsirea drumului de lungime minimă până la nodul obiectiv cu consum minim de memorie
  - versiunea recursivă

### Dezavantaje

- Se poate bloca pe anumite drumuri greșite (nenorocoase) fără a putea reveni
  - Ciclu infinit
  - Găsirea unei soluții mai "lungi" decât soluția optimă

### Aplicații

- Problema labirintului (maze)
- Identificarea componentelor conexe
- Sortare topologică
- Testarea planarității



# SCnI în structuri arborescente

## căutare în adâncime (depth-first search – DFS)



```
bool DFS_edges(elem, list){
    discovered =  $\Phi$ ;
    back =  $\Phi$ ;
    toDiscover =  $\Phi$ ;    //LIFO
    for (all neighbours of start) do
        toDiscover = toDiscover U {(start, neighbour)}
    found = false;
    visited = {start};
    while((toDiscover !=  $\Phi$ ) && (!found)){
        edge = pop(toDiscover);
        if (edge.out !e visited){
            discovered = discovered U {edge};
            visited = visited U {edge.out}
            if (edge.out == end)
                found = true;
            else{
                aux =  $\Phi$ ;
                for all neighbours of edge.out do{
                    aux = aux U {(edge.out, neighbour)};
                }
                toDiscover = aux U toDiscover;
            }
        }
        else
            back = back U {edge}
    } //while
    return found;
}
```

Muchia	Muchii vizitate deja	Muchii de vizitat	înapoi	Noduri vizitate
	$\Phi$	AB, AF	$\Phi$	A
AB	AB	BC, BK, AF	$\Phi$	A, B
BC	AB, BC	CD, BK, AF	$\Phi$	A, B, C
CD	AB, BC, CD	DE, BK, AF	$\Phi$	A, B, C, D
DE	AB, BC, CD, DE	EF, EH, BK, AF	$\Phi$	A, B, C, D, E
EF	AB, BC, CD, DE, EF	FI, FG, EH, BK, AF	$\Phi$	A, B, C, D, E, F
FI	AB, BC, CD, DE, EF, FI	FG, EH, BK, AF	$\Phi$	A, B, C, D, E, F, I
FG	AB, BC, CD, DE, EF, FI, FG	GA, EH, BK, AF	$\Phi$	A, B, C, D, E, F, I, G
GA	AB, BC, CD, DE, EF, FI, FG	EH, BK, AF	GA	A, B, C, D, E, F, I, G
EH	AB, BC, CD, DE, EF, FI, FG	HJ, HN, BK, AF	GA	A, B, C, D, E, F, I, G, H
HJ	AB, BC, CD, DE, EF, FI, FG, HJ	HN, BK, AF	GA	A, B, C, D, E, F, I, G, H, J
HN	AB, BC, CD, DE, EF, FI, FG, HJ, HN	BK, AF	GA	A, B, C, D, E, F, I, G, H, N

# SCnI în structuri arborescente

## căutare în adâncime limitată (depth-limited search – DLS)

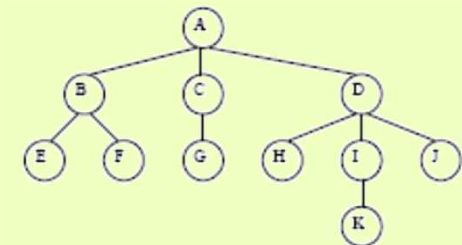


### Aspecte teoretice

- DFS + adâncime maximă care limitează căutarea ( $d_{lim}$ )
- Se soluționează problemele de completitudine ale căutării în adâncime (DFS)

### Exemplu

- $d_{lim} = 2$
- Ordinea vizitării nodurilor: A, B, E, F, C, G, D, H, I, J



### Algoritm

```

bool DLS(elem, list, dlim){
    found = false;
    visited = Φ;
    toVisit = {start}; //LIFO list
    while((toVisit != Φ) && (!found)){
        node = pop(toVisit);
        visited = visited U {node};
        if (node.depth <= dlim){
            if (node == elem)
                found = true;
            else{
                aux = Φ;
                for all (unvisited) children of node do{
                    aux = aux U {child};
                }
                toVisit = aux U toVisit;
            }
        }
    }
    return found;
}
    
```

Vizitate deja	De vizitat
Φ	A
A	B, C, D
A, B	E, F, C, D
A, B, E	F, C, D
A, B, E, F	C, D
A, B, E, F, C	G, D
A, B, E, F, C, G	D
A, B, E, F, C, G, D	H, I, J
A, B, E, F, C, G, D, H	I, J
A, B, E, F, C, G, D, H, I	J
A, B, E, F, C, G, D, H, I, K, J	Φ

# SCnI în structuri arborescente

## căutare în adâncime limitată (depth-limited search – DLS)



### □ Analiza complexității

- Complexitate temporală:
  - $b$  – factor de ramificare (nr de noduri fii ale unui nod)
  - $d_{lim}$  – limita lungimii (adâncimii) permisă pentru un arbore explorat
  - $T(n) = 1 + b + b^2 + \dots + b^{d_{lim}} \Rightarrow O(b^{d_{lim}})$
- Complexitate spațială
  - $S(n) = b * d_{lim}$
- Completitudine
  - Da, dar  $\Leftrightarrow d_{lim} > d$ , unde  $d$  = lungimea (adâncimea) soluției optime
- Optimalitate
  - Nu  $\rightarrow$  căutarea în adâncime poate găsi un drum soluție mai lung decât drumul optim

### □ Avantaje

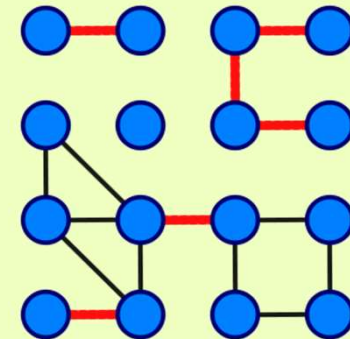
- Se soluționează problemele de completitudine ale căutării în adâncime (DFS)

### □ Dezavantaje

- Cum se alege o limită  $d_{lim}$  bună?

### □ Aplicații

- Determinarea „podurilor” într-un graf





# SCnI în structuri arborescente – căutare în adâncime iterativă (iterative deepening depth search – IDDS)



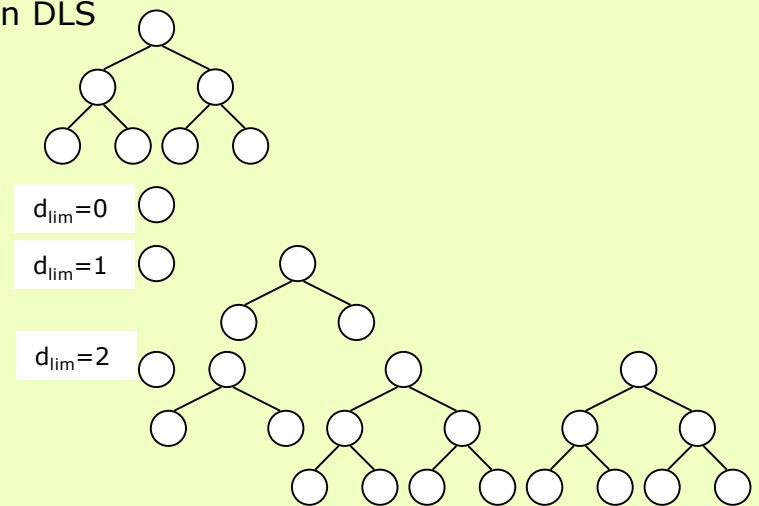
## Aspecte teoretice

- U DLS( $d_{lim}$ ), unde  $d_{lim} = 1, 2, 3, \dots, d_{max}$
- Se soluționează problema stabilirii limitei  $d_{lim}$  optime din DLS
- De obicei, se aplică acolo unde:
  - spațiul de căutare este mare și
  - se cunoaște lungimea (adâncimea) soluției

## Exemplu

## Algoritm

```
bool IDS(elem, list){
    found = false;
    dlim = 0;
    while ((!found) && (dlim < dmax)){
        found = DLS(elem, list, dlim);
        dlim++;
    }
    return found;
}
```



# SCnI în structuri arborescente – căutare în adâncime iterativă (iterative deepening depth search – IDDS)



## □ Analiza complexității

### ■ Complexitate temporală:

- Nodurile situate la adâncimea  $d_{max}$  (în număr de  $b^{d_{max}}$ ) se expandează o singură dată  $\Rightarrow 1 * b^{d_{max}}$
- Nodurile situate la adâncimea  $d_{max}-1$  (în număr de  $b^{d_{max}-1}$ ) se expandează de 2 ori  $\Rightarrow 2 * (b^{d_{max}-1})$
- ...
- Nodurile situate la adâncimea 1 (în număr de  $b$ ) se expandează de  $d_{max}$  ori  $\Rightarrow d_{max} * b^1$
- Nodurile situate la adâncimea 0 (în număr de 1 - rădăcina) se expandează de  $d_{max}+1$  ori  $\Rightarrow (d_{max}+1)*b^0$

$$T(n) = \sum_{i=0}^{d_{max}} (i+1)b^{d_{max}-i} \Rightarrow O(b^{d_{max}})$$

### ■ Complexitate spațială

- $S(n) = b * d_{max}$

### ■ Completitudine

- Da

### ■ Optimalitate

- Da

## □ Avantaje

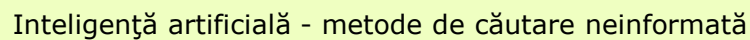
- Necesită memorie liniară
- Asigură atingerea nodului țintă urmând un drum de lungime minimă
- Mai rapidă decât BFS și DFS

## □ Dezavantaje

- Necesită cunoașterea "adâncimii" soluției

## □ Aplicații

- Jocul Tic tac toe



# SCnI în structuri arborescente

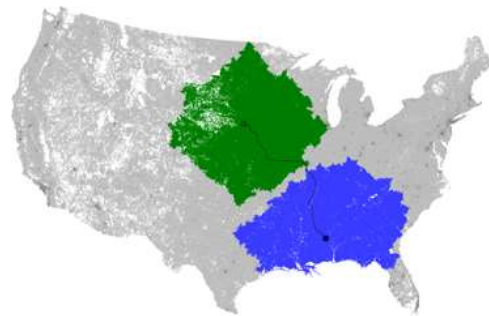
## căutare bidirecțională (bi-directional search – BDS)



### □ Aspecte teoretice

- 2 căutări simultane
  - Înainte (*forward*): de la rădăcină spre frunze
  - Înapoi (*backward*): de la frunze spre rădăcinăcare se opresc atunci când ajung la un nod comun
- Într-o direcție pot fi folosite oricare dintre strategiile de căutare anterioare
- necesită
  - stabilirea succesorilor, respectiv a predecesorilor unui nod
  - stabilirea locului de întâlnire

### □ Exemplu



### □ Algoritm

- În funcție de strategia de căutare folosită

# SCnI în structuri arborescente

## căutare bidirecțională (bi-directional search – BDS)



### ■ Analiza complexității

- Complexitate temporală
  - $b$  – factor de ramificare (nr de noduri fii ale unui nod)
  - $d$  - lungimea (adâncimea) soluției
  - $O(b^{d/2}) + O(b^{d/2}) \Rightarrow O(b^{d/2})$
- Complexitate spațială
  - $S(n) = T(n)$
- Completitudine
  - da
- Optimalitate
  - da

### ■ Avantaje

- Complexitate spațială și temporală redusă

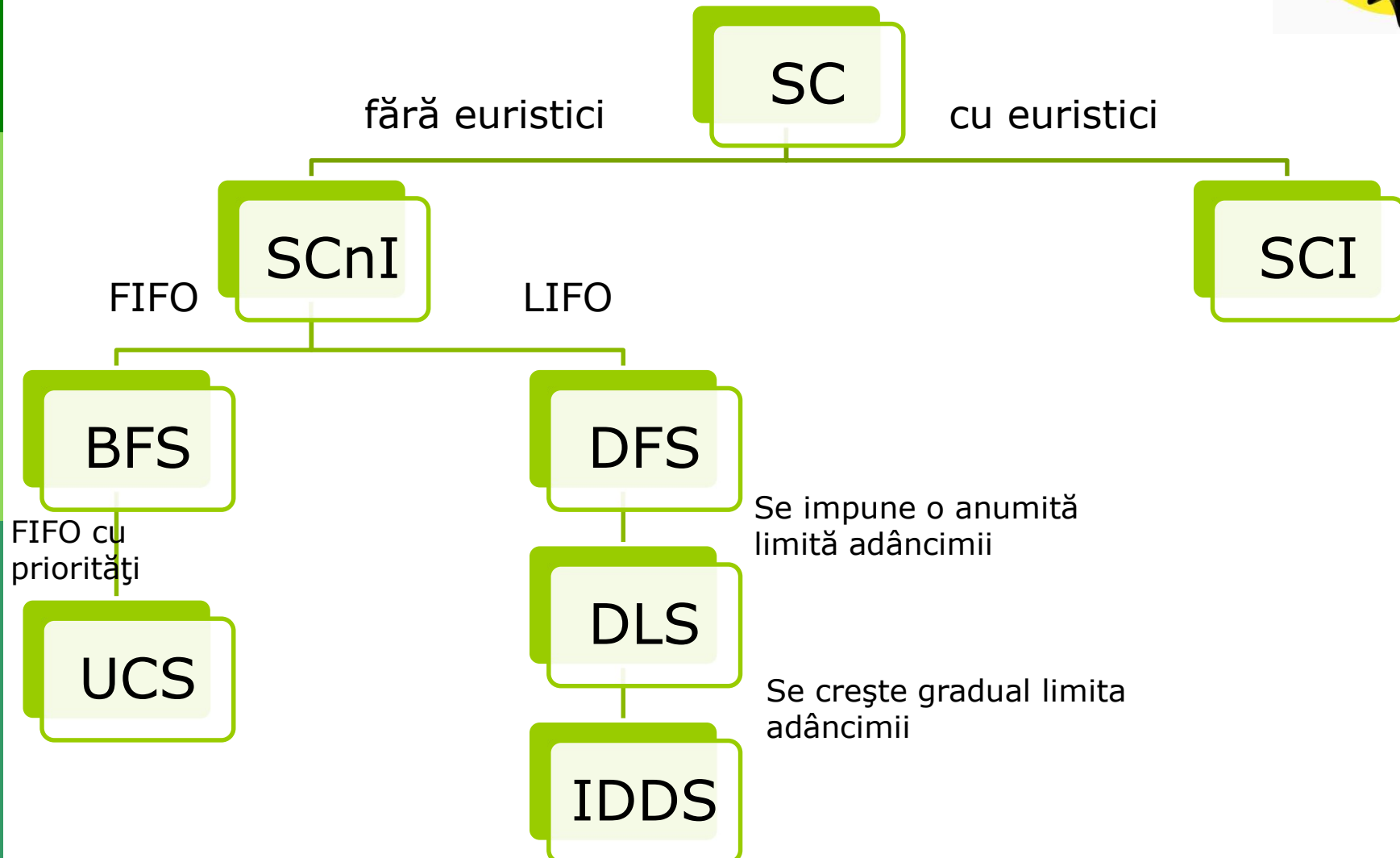
### ■ Dezavantaje

- Dificultăți în formularea problemei astfel încât fiecare stare să poată fi inversată
  - Parcurgere dinspre cap spre coadă
  - Parcurgere dinspre coadă spre cap
- Implementare dificilă
- Trebuie determinați succesorii și predecesorii tuturor stărilor
- Trebuie cunoscută starea finală (obiectiv)

### ■ Aplicații

- Problema partiționării
- Cel mai scurt drum

# SCnI în structuri arborescente



# SCnI în structuri arborescente



## Compararea performanțelor

Metoda de căutare	Complexitate temporală	Complexitate spațială	Completitudin e	Optimalitate
BFS	$O(b^d)$	$O(b^d)$	Da	Da
UCS	$O(b^d)$	$O(b^d)$	Da	Da
DFS	$O(b^{d_{\max}})$	$O(b * d_{\max})$	Nu	Nu
DLS	$O(b^{d_{\lim}})$	$O(b * d_{\lim})$	Da dacă $d_{\lim} > d$	Nu
IDS	$O(b^d)$	$O(b * d)$	Da	Da
BDS	$O(b^{d/2})$	$O(b^{d/2})$	Da	Da

# Rezolvarea problemelor prin căutare

---



- Strategii de căutare (SC)
  - Topologie
    - În funcție de informația disponibilă
      - SC ne-informate (oarbe)
      - SC informate (euristice)



# Strategii de căutare informate (SCI)



## □ Caracteristici

- se bazează pe informații specifice problemei încercând să restrângă căutarea prin alegerea inteligentă a nodurilor care vor fi explorate
- ordonarea nodurilor se face cu ajutorul unei funcții (euristici) de evaluare
- sunt particulare

## □ Topologie

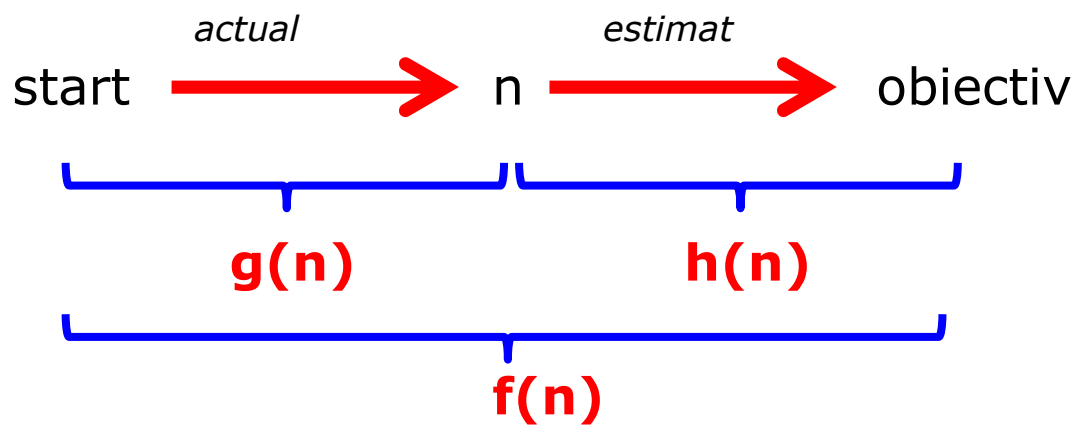
- Strategii globale
  - Best-first search
    - Greedy best-first search
    - A\* + versiuni ale A\*
- Strategii locale
  - Căutare tabu
  - *Hill climbing*
  - *Simulated annealing*

# SC în structuri arborescente



## □ Noțiuni necesare

- $f(n)$  – funcție de evaluare pentru estimarea costului soluției prin nodul (starea)  $n$
- $h(n)$  – funcție euristică pentru estimarea costului drumului de la nodul  $n$  la nodul obiectiv
- $g(n)$  – funcție de cost pentru estimarea costului drumului de la nodul de start până la nodul  $n$
- $f(n) = g(n) + h(n)$



# SCI – Best first search



## □ Aspecte teoretice

- Best first search = mai întâi cel mai bun
- Se determină costul fiecărei stări cu ajutorul funcției de evaluare  $f$
- Nodurile de expandant sunt reținute în structuri (cozi) ordonate
- Pentru expandare se alege starea cu cea mai bună evaluare
  - Stabilirea nodului care urmează să fie expandat
- Exemple de SC care depind de funcția de evaluare
  - Căutare de cost uniform (SCnI)
    - $f$  = costul drumului
  - În SCI se folosesc funcții euristice
- Două categorii de SCI de tip best first search
  - SCI care încearcă să expandeze nodul cel mai apropiat de starea obiectiv
  - SCI care încearcă să expandeze nodul din soluția cu costul cel mai mic

## □ Exemplu

- Detalii în slide-urile următoare

# SCI – Best first search



## □ Algorithm

```
bool BestFS(elem, list){
    found = false;
    visited =  $\Phi$ ;
    toVisit = {start};    //FIFO sorted list (priority queue)
    while((toVisit !=  $\Phi$ ) && (!found)){
        if (toVisit ==  $\Phi$ )
            return false
        node = pop(toVisit);
        visited = visited U {node};
        if (node == elem)
            found = true;
        else
            aux =  $\Phi$ ;
        for all unvisited children of node do{
            aux = aux U {child};
        }
        toVisit = toVisit U aux; //adding a node into the FIFO list based on its
                                // evaluation (best one in the front of list)
    } //while
    return found;
}
```

# SCI – best first search



## □ Analiza căutării

- Complexitate temporală:
  - $b$  – factor de ramnificare (nr de noduri fii ale unui nod)
  - $d$  - lungimea (adâncimea) maximă a soluției
  - $T(n) = 1 + b + b^2 + \dots + b^d \Rightarrow O(b^d)$
- Complexitate spațială
  - $S(n) = T(n)$
- Completitudine
  - Nu- drumuri infinite dacă euristica evaluează fiecare stare din drum ca fiind cea mai bună alegere
- Optimalitate
  - Posibil – depinde de euristica

## □ Avantaje

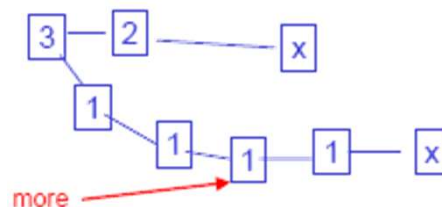
- Informațiile despre domeniul problemei ajută căutarea (SCI)
- Viteză mai mare de a ajunge la starea obiectiv

## □ Dezavantaje

- Necesită evaluarea stărilor → efort computațional, dar nu numai
- Anumite path-uri pot “arăta” ca fiind bune conform funcției euristice

## □ Aplicații

- Web crawler (automatic indexer)
- Jocuri



# SCI – Funcții euristice



- Etimologie: *heuriskein (gr)*
  - *a găsi, a descoperi*
  - *studiul metodelor și regulilor de descoperire și invenție*
  
- Utilitate
  - Evaluarea potențialului unei stări din spațiul de căutare
  - Estimarea costului drumului (în arborele de căutare) din starea curentă până în starea finală (cât de aproape de țintă a ajuns căutarea)
  
- Caracteristici
  - Depind de problema care trebuie rezolvată
  - Pentru probleme diferite trebuie proiectate sau învățate diferite euristici
  - Se evaluează o anumită stare (nu operatorii care transformă o stare în altă stare)
  - Funcții pozitive pentru orice stare  $n$ 
    - $h(n) \geq 0$  pentru orice stare  $n$
    - $h(n) = 0$  pentru starea finală
    - $h(n) = \infty$  pentru o stare din care începe un drum mort (o stare din care nu se poate ajunge în starea finală)

# SCI – Funcții euristice



## □ Exemple

- Problema misionarilor și canibalilor
  - $h(n)$  – nr. persoanelor aflate pe malul inițial
- 8-puzzle
  - $h(n)$  – nr. pieselor care nu se află la locul lor
  - $h(n)$  – suma distanțelor Manhattan (la care se află fiecare piesă de poziția ei finală)
- Problema comisului voiajor
  - $h(n)$  – cel mai apropiat vecin !!!
- Plata unei sume folosind un număr minim de monezi
  - $h(n)$  – alegerea celei mai valoroase monezi mai mică decât suma (rămasă) de plată

# SCI - Greedy



## Aspecte teoretice

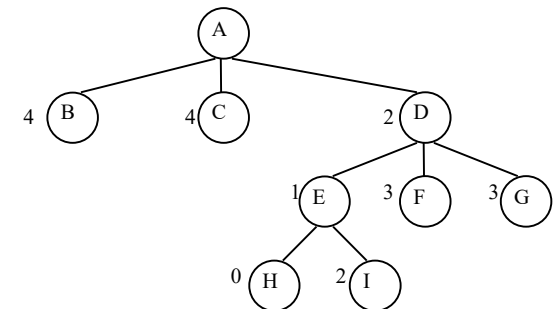
- Funcția de evaluare  $f(n) = h(n)$ 
  - estimarea costului drumului de la starea curentă la starea finală –  $h(n)$
  - minimizarea costului drumului care mai trebuie parcurs

## Exemplu

- A,D,E,H

## Algoritm

```
bool BestFS(elem, list){
    found = false;
    visited =  $\Phi$ ;
    toVisit = {start};          //FIFO sorted list (priority queue)
    while((toVisit !=  $\Phi$ ) && (!found)){
        if (toVisit ==  $\Phi$ )
            return false
        node = pop(toVisit);
        visited = visited U {node};
        if (node == elem)
            found = true;
        else
            aux =  $\Phi$ ;
        for all unvisited children of node do{
            aux = aux U {child};
        }
        toVisit = toVisit U aux; //adding a node onto the FIFO list based on its evaluation  $h(n)$ 
                                //(best one in the front of list)
    } //while
    return found;
}
```



Vizitate deja	De vizitat
$\Phi$	A
A	D, B, C
A, D	E, F, G, B, C
A, D, E	H, I, F, G, B, C
A, D, E, H	$\Phi$



# SCI - Greedy



## □ Analiza căutării:

- Complexitate temporală → DFS
  - $b$  - factor de ramnificare (nr de noduri fii ale unui nod)
  - $d^{max}$  - lungimea (adâncimea) maximă a unui arbore explorat
  - $T(n) = 1 + b + b^2 + \dots + b^{d^{max}} \Rightarrow O(b^{d^{max}})$
- Complexitate spațială → BFS
  - $d$  - lungimea (adâncimea) soluției
  - $S(n) = 1 + b + b^2 + \dots + b^d \Rightarrow O(b^d)$
- Completitudine
  - Nu (există posibilitatea intrării în cicluri infinite)
- Optimalitate
  - posibil

## □ Avantaje

- Găsirea rapidă a unei soluții (dar nu neapărat soluția optimă), mai ales pentru probleme mici

## □ Dezavantaje

- Suma alegerilor optime de la fiecare pas nu reprezintă alegerea globală optimă
  - Ex. Problema comisului voiajor

## □ Aplicații

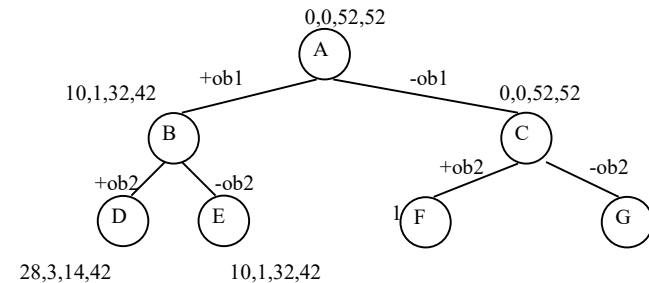
- Probleme de planificare
- Probleme de sume parțiale
  - Plata unei sume folosind diferite tipuri de monezi
  - Problema rucsacului
- Puzzle-uri
- Drumul optim într-un graf

# SCI – A\*



## Aspecte teoretice

- Combinarea aspectelor pozitive ale
  - căutării de cost uniform
    - optimalitate și completitudine
    - utilizarea unei cozi ordonate
  - căutării Greedy
    - viteza mare
    - ordonare pe baza unei funcții de evaluare
- Funcția de evaluare  $f(n)$ 
  - estimarea costului celui mai bun drum care trece prin nodul  $n$
  - $f(n) = g(n) + h(n)$
  - $g(n)$  – funcție folosită pentru stabilirea costului drumului de la starea inițială până la starea curentă  $n$
  - $h(n)$  – funcție euristică folosită pentru estimarea costului drumului de la starea curentă  $n$  la starea finală
- Minimizarea costului total al unui drum



	$o_1$	$o_2$	$o_3$	$o_4$
$p_i$	10	18	32	14
$w_i$	1	2	4	3

## Exemplu

- Problema rucsacului de capacitate  $W$  în care pot fi puse  $n$  obiecte ( $o_1, o_2, \dots, o_n$ ) fiecare având o greutate  $w_i$  și aducând un profit  $p_i, i=1,2,\dots,n$ 
  - Soluția: pentru un rucsac cu  $W = 5 \rightarrow$  alegerea obiectelor  $o_1$  și  $o_3$
- $g(n) = \sum p_i$ , pentru acele obiecte  $o_i$  care au fost selectate
- $h(n) = \sum p_j$ , pentru acele obiecte care nu au fost selectate și  $\sum w_j \leq W - \sum w_i$
- Fiecare nod din graf este un tuplu:  $(p, w, p^*, f)$ , unde:
  - $p$  – profitul adus de obiectele deja selectate (funcția  $g(n)$ )
  - $w$  – greutatea obiectelor selectate
  - $p^*$  – profitul maxim care se poate obține pornind din starea curentă și ținând cont de locul disponibil în rucsac (funcția  $h(n)$ )

# SCI – A\*



## □ Algorithm

```
bool BestFS(elem, list){
    found = false;
    visited =  $\Phi$ ;
    toVisit = {start}; //FIFO sorted list (priority queue
    while((toVisit !=  $\Phi$ ) && (!found)){
        if (toVisit ==  $\Phi$ )
            return false
        node = pop(toVisit);
        visited = visited U {node};
        if (node == elem)
            found = true;
        else
            aux =  $\Phi$ ;
        for all unvisited children of node do{
            aux = aux U {child};
        }
        toVisit = toVisit U aux; //adding a node onto the FIFO list
                                // based on its evaluation  $f(n) = g(n) + h(n)$ 
                                // (best one in the front of list)

    } //while
    return found;
}
```

# SCI – A\*



## □ Analiza căutării:

- Complexitate temporală:
  - $b$  – factor de ramnificare (nr de noduri fii ale unui nod)
  - $d^{max}$  - lungimea (adâncimea) maximă a unui arbore explorat
  - $T(n) = 1 + b + b^2 + \dots + b^{d^{max}} \Rightarrow O(b^{d^{max}})$
- Complexitate spațială
  - $d$  - lungimea (adâncimea) soluției
  - $T(n) = 1 + b + b^2 + \dots + b^d \Rightarrow O(b^d)$
- Completitudine
  - Da
- Optimalitate
  - Da

## □ Avantaje

- Algoritmul care expandează cele mai puține noduri din arborele de căutare

## □ Dezavantaje

- Utilizarea unei cantități mari de memorie

## □ Aplicații

- Probleme de planificare
- Probleme de sume parțiale
  - Plata unei sume folosind diferite tipuri de monezi
  - Problema rucsacului
- Puzzle-uri
- Drumul optim într-un graf

# SCI – A\*



## □ Variante

- iterative deepening A\* (IDA\*)
- memory-bounded A\* (MA\*)
- simplified memory bounded A\* (SMA\*)
- recursive best-first search (RBFS)
- dynamic A\* (DA\*)
- real time A\*
- hierarchical A\*

## □ Bibliografie suplimentară

- 02/A\_IDA.pdf
- 02/A\_IDA\_2.pdf
- 02/SMA\_RTA.pdf
- 02/Recursive Best-First Search.ppt
- 02/IDS.pdf
- 02/IDA\_MA.pdf
- [http://en.wikipedia.org/wiki/IDA\\*](http://en.wikipedia.org/wiki/IDA*)
- [http://en.wikipedia.org/wiki/SMA\\*](http://en.wikipedia.org/wiki/SMA*)

# Rezolvarea problemelor prin căutare



- Tipologia strategiilor de căutare
  - În funcție de modul de **generare** a soluției
    - Căutare **constructivă**
      - Construirea progresivă a soluției
      - Ex. TSP
    - Căutare **perturbativă**
      - O soluție candidat este modificată în vederea obținerii unei noi soluții candidat
      - Ex. SAT - Propositional Satisfiability Problem
  - În funcție de modul de **traversare** a spațiului de căutare
    - Căutare **sistematică**
      - Traversarea completă a spațiului
        - Ideintificarea soluției dacă ea există → algoritmi compleți
    - Căutare **locală**
      - Traversarea spațiului de căutare dintr-un punct în alt punct vecin → algoritmi incompleți
      - O stare a spațiului poate fi vizitată de mai multe ori
  - În funcție de elementele de **certitudine** ale căutării
    - Căutare **deterministă**
      - Algoritmi de identificare exactă a soluției
    - Căutare **stocastică**
      - Algoritmi de aproximare a soluției
  - În funcție de stilul de **explorare** a spațiului de căutare
    - Căutare **secvențială**
    - Căutare **paralelă**

# Rezolvarea problemelor prin căutare



Poate consta în:

- ❑ Construirea progresivă a soluției
- ❑ Identificarea soluției potențiale optime
  - Componentele problemei
    - ❑ Condiții (constrângeri) pe care trebuie să le satisfacă (parțial sau total) soluția
    - ❑ Funcție de evaluare a unei soluții potențiale → identificarea optimului
  - Spațiul de căutare
    - ❑ mulțimea tuturor soluțiilor potențiale complete
    - ❑ Stare = o soluție completă
    - ❑ Stare finală (scop) → soluția optimă
  - Exemple
    - ❑ Problema celor 8 regine,
      - Stările posibile: configurații ale tablei de sah cu câte 8 regine
      - Operatori: modificarea coloanei în care a fost plasată una din regine
      - Scopul căutării: configurația în care nu există atacuri între regine
      - Funcția de evaluare: numărul de atacuri
    - ❑ probleme de planificare,
    - ❑ proiectarea circuitelor digitale, etc



# Rezolvarea problemelor prin căutare



Poate consta în:

- ❑ Construirea progresivă a soluției
- ❑ Identificarea soluției potențiale optime
  - Algoritmi
    - ❑ Algoritmii discutați până acum explorau în mod **sistematic** spațiul de căutare
      - De ex.  $A^* \rightarrow 10^{100}$  stări  $\approx 500$  variabile binare
    - ❑ Problemele reale pot avea 10 000 – 100 000 variabile  $\rightarrow$  nevoia unei alte categorii de algoritmi care explorează **local** spațiul de căutare (algoritmi iterativi)
    - ❑ Ideea de bază:
      - se începe cu o stare care nu respectă anumite constrângeri pentru a fi soluție optimă și
      - se efectuează modificări pentru a elimina aceste violări (se deplasează căutarea într-o soluție vecină cu soluția curentă) astfel încât căutarea să se îndrepte spre soluția optimă
    - ❑ Iterativi  $\rightarrow$  se memorează o singură stare și se încearcă îmbunătățirea ei
      - versiunea inteligentă a algoritmului de forță brută
    - ❑ Istoricul căutării nu este reținut

```
bool LS(elem, list){
    found = false;
    crtState = initState
    while ((!found) && timeLimitIsNotExceeded){
        toVisit = neighbours(crtState)
        if (best(toVisit) is better than crtState)
            crtState = best(toVisit)
        if (crtState == elem)
            found = true;
    } //while
    return found;
}
```



# Rezolvarea problemelor prin căutare



Poate consta în:

- ❑ Construirea progresivă a soluției
- ❑ Identificarea soluției potențiale optime
  - Avantaje
    - ❑ Simplu de implementat
    - ❑ Necesită puțină memorie
    - ❑ Poate găsi soluții rezonabile în spații de căutare (continue) foarte mari pentru care alți algoritmi sistematici nu pot fi aplicați
  - E utilă atunci când:
    - ❑ Se pot genera soluții complete rezonabile
    - ❑ Se poate alege un bun punct de start
    - ❑ Există operatori pentru modificarea unei soluții complete
    - ❑ Există o măsură pentru a aprecia progresul (avansarea căutării)
    - ❑ Există un mod de a evalua soluția completă (în termeni de constrângeri violate,

# Strategii de căutare locală



## □ Tipologie

- Căutare locală simplă - se reține o singură stare vecină
  - Hill climbing → alege cel mai bun vecin
  - Simulated annealing → alege probabilistic cel mai bun vecin
  - Căutare tabu → reține lista soluțiilor recent vizitate
  
- Căutare locală în fascicol (*beam local search*) – se rețin mai multe stări (o populație de stări)
  - Algoritmi evolutivi
  - Optimizare bazată pe comportamentul de grup (*Particle swarm optimisation*)
  - Optimizare bazată pe furnici (*Ant colony optimisation*)



# Strategii de căutare locală

## □ Căutare locală simplă

### ■ elemente de interes special:

- Reprezentarea soluției
- Funcția de evaluare a unei potențiale soluții
- Vecinătatea unei soluții
  - Cum se definește/generează o soluție vecină
  - Cum are loc căutarea soluțiilor vecine
    - Aleator
    - Sistematic

} dependente  
de problemă

### □ Criteriul de acceptare a unei noi soluții

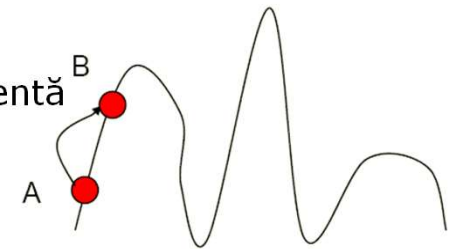
- Primul vecin mai bun decât soluția curentă
- Cel mai bun vecin al soluției curente mai bun decât soluția curentă
- Cel mai bun vecin al soluției curente mai slab decât soluția curentă
- Un vecin aleator

# Strategii de căutare locală – Hill climbing (HC)



## □ Aspecte teoretice

- Urcarea unui munte în condiții de ceață și amnezie a excursionistului :D
- Mișcarea continuă spre valori mai bune (mai mari → urcușul pe munte)
- Căutarea avansează în direcția îmbunătățirii valorii stărilor succesore până când se atinge un optim
- Criteriul de acceptare a unei noi soluții
  - cel mai bun vecin al soluției curente mai bun decât soluția curentă
- Îmbunătățire prin
  - Maximizarea calității unei stări → *steepest ascent HC*
  - Minimizarea costului unei stări → *gradient descent HC*
- $HC \neq \textit{steepest ascent/gradient descent (SA/GD)}$ 
  - HC optimizează  $f(x)$  cu  $x \in \mathbb{R}^n$  prin modificarea unui element al lui  $x$
  - SA/GD optimizează  $f(x)$  cu  $x \in \mathbb{R}^n$  prin modificarea tuturor elementelor lui  $x$

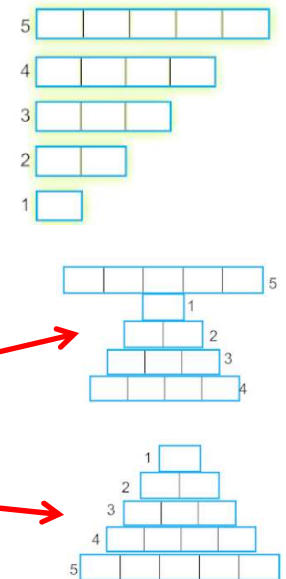


# Strategii de căutare locală – Hill climbing (HC)



## Exemplu

- Construirea unor turnuri din diferite forme geometrice
  - Se dau  $n$  piese de formă dreptunghiulară (de aceeași lățime, dar de lungimi diferite) așezate unele peste altele formând un turn (stivă). Să se re-așeze piesele astfel încât să se formeze un nou turn știind că la o mutare se poate mișca doar o piesă din vârful stivei, piesă care poate fi mutată pe una din cele 2 stive ajutătoare.
- Reprezentarea soluției
  - Stare  $x$  – vectori de  $n$  perechi de forma  $(i,j)$ , unde  $i$  reprezintă indexul piesei ( $i=1,2,\dots,n$ ), iar  $j$  indexul stivei pe care se află piesa ( $j=1,2,3$ )
  - Starea inițială – vectorul corespunzător turnului inițial
  - Starea finală – vectorul corespunzător turnului final
- Funcția de evaluare a unei stări
  - $f1$  = numărul pieselor corect amplasate  $\rightarrow$  maximizare
    - conform turnului final –  $f1 = n$
  - $f2$  = numărul pieselor greșit amplasate  $\rightarrow$  minimizare
    - conform turnului final –  $f2 = 0$
  - $f = f1 - f2 \rightarrow$  maximizare
- Vecinătate
  - Mutări posibile
    - Mutarea unei piese  $i$  din vârful unei stive  $j1$  pe altă stivă  $j2$
- Criteriul de acceptare a unei noi soluții
  - Cel mai bun vecin al soluției curente mai bun decât soluția curentă



# Strategii de căutare locală – Hill climbing (HC)

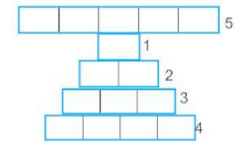


## □ Exemplu

### ■ Iterația 1

- Starea curentă  $x$  = starea inițială:

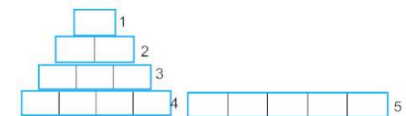
- $x = s_1 = ((5,1), (1,1), (2,1), (3,1), (4,1))$
- Piesele 1, 2 și 3 sunt corect așezate
- Piesele 4 și 5 nu sunt corect așezate
- $f(s_1) = 3 - 2 = 1$



- $x^* = x$

- Vecinii stării curente  $x$  – un singur vecin  $\rightarrow$  piesa 5 se mută pe stiva 2  $\rightarrow$

- $s_2 = ((1,1), (2,1), (3,1), (4,1), (5,2))$
- $f(s_2) = 4 - 1 = 3 > f(x) \rightarrow x = s_2$



# Strategii de căutare locală – Hill climbing (HC)



## ■ Exemplu

### ■ Iterația 2

■ Starea curentă  $x = ((1,1), (2,1), (3,1), (4,1), (5,2))$

■  $f(x) = 3$

■ Vecinii stării curente – doi vecini:

■ piesa 1 se mută pe stiva 2  $\rightarrow s_3 = ((2,1), (3,1), (4,1), (1,2), (5,2)) \rightarrow f(s_3) = 3-2=1 < f(x)$



■ piesa 1 se mută pe stiva 3  $\rightarrow s_4 = ((2,1), (3,1), (4,1), (5,2), (1,3)) \rightarrow f(s_4) = 3-2=1 < f(x)$



■ nu există vecin de-al lui  $x$  mai bun ca  $x \rightarrow$  stop

■  $x^* = x = ((1,1), (2,1), (3,1), (4,1), (5,2))$

■ Dar  $x^*$  este doar optim local, nu global

# Strategii de căutare locală – Hill climbing (HC)



## □ Exemplu

- Construirea unor turnuri din diferite forme geometrice
  - Funcția de evaluare a unei stări
    - $f1$  = suma înălțimilor stivelor pe care sunt amplasate corect piese (conform turnului final –  $f1 = 10$ ) → maximizare
    - $f2$  = suma înălțimilor stivelor pe care sunt amplasate incorect piese (conform turnului final –  $f2 = 0$ ) → minimizare
    - $f = f1 - f2$  → maximizare
  - Vecinătate
    - Mutări posibile
      - Mutarea unei piese  $i$  din vârful unei stive  $j1$  pe altă stivă  $j2$



# Strategii de căutare locală – Hill climbing (HC)

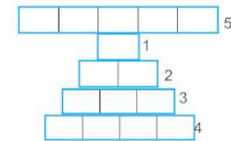


## □ Exemplu

### ■ Iterația 1

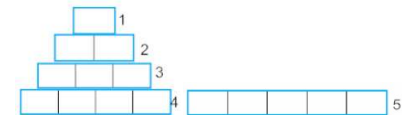
- Starea curentă  $x$  = starea inițială  $s_1 = ((5,1), (1,1), (2,1), (3,1), (4,1))$

- Toate piesele nu sunt corect așezate  $\rightarrow f_1 = 0, f_2 = 3+2+1+0+4 = 10$
- $f(s_1) = 0 - 10 = -10$



- $x^* = x$

- Vecinii stării curente  $x$ – un singur vecin  $\rightarrow$  piesa 5 se mută pe stiva 2  $\rightarrow s_2 = ((1,1), (2,1), (3,1), (4,1), (5,2))$



- $f(s_2) = 0 - (3+2+1+0) = -6 > f(x) \rightarrow x = s_2$

# Strategii de căutare locală – Hill climbing (HC)



## ■ Exemplu

### ■ Iterația 2

- Starea curentă  $x = ((1,1), (2,1), (3,1), (4,1), (5,2))$

- $f(x) = -6$

- Vecinii stării curente – doi vecini:

- piesa 1 se mută pe stiva 2  $\rightarrow s3 = ((2,1), (3,1), (4,1), (1,2), (5,2)) \rightarrow f(s3) = 0 - (0+2+3+0) = -5 > f(x)$



- piesa 1 se mută pe stiva 3  $\rightarrow s4 = ((2,1), (3,1), (4,1), (5,2), (1,3)) \rightarrow f(s4) = 0 - (1+2+1) = -4 > f(x)$



- cel mai bun vecin al lui x este s4  $\rightarrow x = s4$

### ■ Iterația 3

- ...

- Se evită astfel blocarea în optimele locale

# Strategii de căutare locală – Hill climbing (HC)



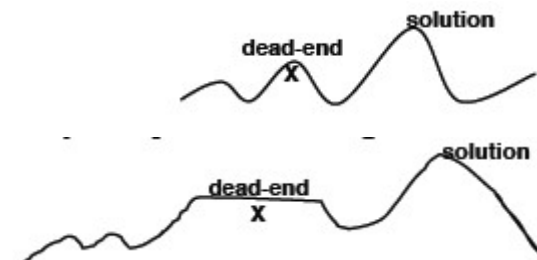
## □ Algorithm

```
Bool HC(S) {  
    x = s1      //starea inițială  
    x*=x        // cea mai bună soluție găsită  
                (până la un moment dat)  
    k = 0       // numarul de iterații  
    while (not termination criteria) {  
        k = k + 1  
        generate all neighbours of x (N)  
        Choose the best solution s from N  
        if f(s) is better than f(x) then x = s  
        else stop  
    } //while  
    x* = x  
    return x*;  
}
```

# Strategii de căutare locală – Hill climbing (HC)



- **Analiza căutării**
  - Convergența spre optimul local
- **Avantaje**
  - Simplu de implementat → se poate folosi ușor pentru a aproxima soluția unei probleme când soluția exactă este dificil sau imposibil de găsit
    - Ex. TSP cu foarte multe orașe
  - Nu necesită memorie (nu se revine în starea precedentă)
- **Dezavantaje**
  - Funcția de evaluare (euristică) poate fi dificil de estimat
  - Dacă se execută foarte multe mutări algoritmul devine ineficient
  - Dacă se execută prea puține mutări algoritmul se poate bloca
    - Într-un optim local (nu mai poate "coborî" din vârf)
    - Pe un platou – zonă din spațiul de căutare în care funcția de evaluare este constantă
    - Pe o creastă – saltul cu mai mulți pași ar putea ajuta căutarea
- **Aplicații**
  - Problema canibalilor
  - 8-puzzle, 15-puzzle
  - TSP
  - Problema damelor



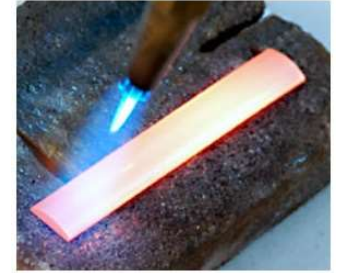
# Strategii de căutare locală – Hill climbing (HC)



## □ Variante

- HC stocastic
  - Alegerea aleatoare a unui succesor
- HC cu prima opțiune
  - Generarea aleatoare a succesorilor până la întâlnirea unei mutări neefectuate
- HC cu repornire aleatoare → *beam local search*
  - Repornirea căutării de la o stare inițială aleatoare atunci când căutarea nu progresează

# Strategii de căutare locală – Simulated Annealing

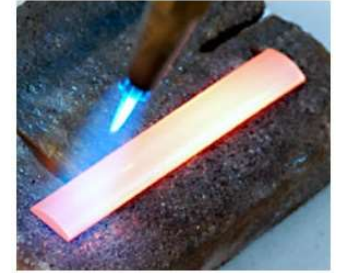


## □ Aspecte teoretice

- Inspirată de modelarea proceselor fizice
  - Metropolis et al. 1953, Kirkpatrick et al. 1982;
- Succesorii stării curente sunt aleși și în mod aleator
  - Dacă un succesori este mai bun decât starea curentă
    - atunci el devine noua stare curentă
    - altfel succesorul este reținut doar cu o anumită probabilitate
- Se permit efectuarea unor mutări "slabe" cu o anumită probabilitate  $p$ 
  - → evadarea din optimele locale
- Probabilitatea  $p = e^{\Delta E/T}$ 
  - Proporțională cu valoarea diferenței (energia)  $\Delta E$
  - Modelată de un parametru de temperatură  $T$
- Frecvența acestor mutări "slabe" și mărimea lor se reduce gradual prin scăderea temperaturii
  - $T = 0 \rightarrow$  hill climbing
  - $T \rightarrow \infty \rightarrow$  mutările "slabe" sunt tot mai mult executate
- Soluția optimă se identifică dacă temperatura se scade treptat ("slowly")
- Criteriul de acceptare a unei noi soluții
  - Un vecin aleator mai bun sau mai slab (cu probabilitatea  $p$ ) decât soluția curentă



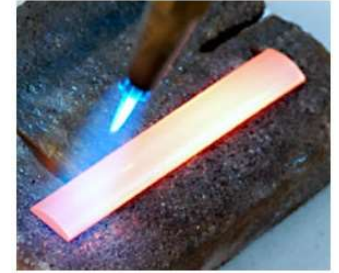
# Strategii de căutare locală – Simulated Annealing



## ■ Exemplu – Problema celor 8 regine

- Enunț
  - Să se amplaseze pe o tablă de șah 8 regine astfel încât ele să nu se atace reciproc
- Reprezentarea soluției
  - Stare  $x$  – permutare de  $n$  elemente  $x = (x_1, x_2, \dots, x_n)$ , unde  $x_i$  – linia pe care este plasată regina de pe coloana  $j$ 
    - Nu există atacuri pe verticală sau pe orizontală
    - Pot exista atacuri pe diagonală
  - Starea inițială – o permutare oarecare
  - Starea finală – o permutare fără atacuri de nici un fel
- Funcția de evaluare a unei stări
  - $f$  – suma reginelor atacate de fiecare regină → minimizare
- Vecinătate
  - Mutări posibile
    - Mutarea unei regine de pe o linie pe alta (interschimbarea a 2 elemente din permutare)
- Criteriul de acceptare a unei noi soluții
  - Un vecin oarecare al soluției curente
    - mai bun decât soluția curentă
    - mai slab decât soluția curentă – cu o anumită probabilitate  $P(\Delta E) = e^{-\frac{\Delta E}{T}}$  unde:
      - $\Delta E$  – diferența de energie (evaluare) între cele 2 stări (vecină și curentă)
      - $T$  – temperatura,  $T(k) = 100/k$ , unde  $k$  este nr iterației

# Strategii de căutare locală – Simulated Annealing



## □ Exemplu – Problema celor 8 regine

### ■ Iterația 1 ( $k = 1$ )

□ Starea curentă  $x$  = starea inițială

$s_1 = (8, 5, 3, 1, 6, 7, 2, 4)$

■  $f(s_1) = 1 + 1 = 2$

□  $x^* = x$

□  $T = 100/1 = 100$

□ Un vecin al stării curente  $x \rightarrow$  regina de pe linia 5 se interschimbă cu regina de pe linia 7

$\rightarrow s_2 = (8, 7, 3, 1, 6, 5, 2, 4)$

■  $f(s_2) = 1 + 1 + 1 = 3 > f(x)$

■  $\Delta E = f(s_2) - f(s_1) = 1$

■  $P(\Delta E) = e^{-1/100}$

■  $r = \text{random}(0, 1)$

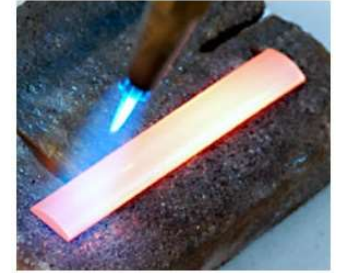
■ Dacă  $r < P(\Delta E) \rightarrow x = s_2$

	a	b	c	d	e	f	g	h	
1				♔					1
2							♔		2
3			♔						3
4								♔	4
5		♔							5
6					♔				6
7							♔		7
8	♔								8
	a	b	c	d	e	f	g	h	

	a	b	c	d	e	f	g	h	
1				♔					1
2							♔		2
3			♔						3
4								♔	4
5		♔							5
6					♔				6
7							♔		7
8	♔								8
	a	b	c	d	e	f	g	h	



# Strategii de căutare locală – Simulated Annealing



## □ Algoritm

```
bool SA(S){  
    x = s1 //starea inițială  
    x*=x // cea mai bună soluție găsită (până la un moment dat)  
    k = 0 // numărul de iterații  
    while (not termination criteria){  
        k = k + 1  
        generate a neighbour s of x  
        if f(s) is better than f(x) then x = s  
        else  
            pick a random number r (in (0,1) range)  
            if r < P( $\Delta E$ ) then x = s  
        } //while  
    x* = x  
    return x*;  
}
```

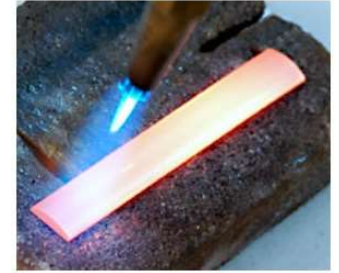
## □ Criterii de oprire

- S-a ajuns la soluție
- S-a parcurs un anumit număr de iterații
- S-a ajuns la temperatura 0 (îngheț)

## □ Cum se alege o probabilitate mică?

- $p = 0.1$
- $p$  scade de-a lungul iterațiilor
- $p$  scade de-a lungul iterațiilor și pe măsură ce "răul"  $|f(s) - f(x)|$  crește
  - $p = \exp(-|f(s) - f(x)|/T)$
  - Unde  $T$  – temperatura (care crește)
    - Pentru o  $T$  mare se admite aproape orice vecin  $v$
    - Pentru o  $T$  mică se admite doar un vecin mai bun decât  $s$
  - Dacă "răul" e mare, atunci probabilitatea e mică

# Strategii de căutare locală – Simulated Annealing



## ■ Analiza căutării

- Convergența (complet, optimal) lentă spre optimul global

## ■ Avantaje

- Algoritm fundamentat statistic → garantează găsirea soluției optime, dar necesită multe iterații
- Ușor de implementat
- În general găsește o soluție relativ bună (optim global)
- Poate rezolva probleme complexe (cu zgomot și multe constrângeri)

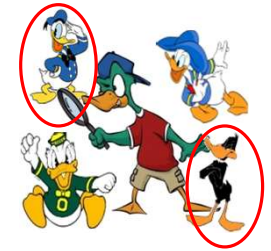
## ■ Dezavantaje

- Algoritm încet – convergența la soluție durează foarte mult timp
  - Compromis între calitatea soluției și timpul necesar calculării ei
- Depinde de anumiți parametri (temperatura) care trebuie reglați
- Nu se știe dacă soluția oferită este optimă (local sau global)
- Calitatea soluției găsite depinde de precizia variabilelor implicate în algoritm

## ■ Aplicații

- Probleme de optimizare combinatorială → problema rucsacului
- Probleme de proiectare → Proiectarea circuitelor digitale
- Probleme de planificare → Planificarea producției, planificarea meciurilor în turnirurile de tenis US Open

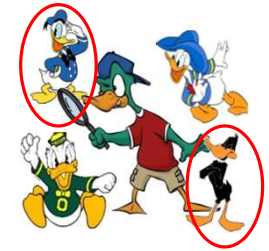
# Strategii de căutare locală – Căutare tabu



## ■ Aspecte teoretice

- “Tabu” → interdicție socială severă cu privire la activitățile umane sacre și interzise
- Propusă în anii 1970 de către F. Glover
- Ideea de bază
  - se începe cu o stare care nu respectă anumite constrângeri pentru a fi soluție optimă și
  - se efectuează modificări pentru a elimina aceste violări (se deplasează căutarea în **cea mai bună soluție vecină** cu soluția curentă) astfel încât căutarea să se îndrepte spre soluția optimă
  - se memorează
    - **Starea curentă**
    - **Stările vizitate** până la momentul curent al căutării și **mutările efectuate** pentru a ajunge în fiecare din acele stări de-a lungul căutării (se memorează o **listă limitată de stări care nu mai trebuie revizitate**)
- Criteriul de acceptare a unei noi soluții
  - Cel mai bun vecin al soluției curente mai bun decât soluția curentă și nevizitat încă
- 2 elemente importante
  - Mutări tabu (T) – determinate de un proces non-Markov care se folosește de informațiile obținute în timpul căutării de-a lungul ultimelor generații
  - Condiții tabu – pot fi inegalități liniare sau legături logice exprimate în funcție de soluția curentă
    - Au rol în alegerea mutărilor tabu

# Strategii de căutare locală – Căutare tabu



## ■ Exemplu

### ■ Enunț

- Plata sumei  $S$  folosind cât mai multe dintre cele  $n$  monezi de valori  $v_i$  (din fiecare monedă există  $b_i$  bucăți)

### ■ Reprezentarea soluției

- Stare  $x$  – vector de  $n$  întregi  $x = (x_1, x_2, \dots, x_n)$  cu  $x_i \in \{0, 1, 2, \dots, b_i\}$
- Starea inițială – aleator

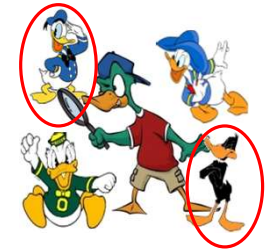
### ■ Funcția de evaluare a unei stări

- $f_1$  = Diferența între  $S$  și valoarea monezilor alese  $\rightarrow$  minimă
  - Dacă valoarea monezilor depășește  $S \rightarrow$  penalizare de 500 unități
- $f_2$  = Numărul monezilor selectate  $\rightarrow$  maxim
- $f = f_1 - f_2 \rightarrow$  minimizare

### ■ Vecinătate

- Mutări posibile
  - Includerea în sumă a monezii  $i$  în  $j$  exemplare ( $\text{plus}_{i,j}$ )
  - Excluderea din sumă a monezii  $i$  în  $j$  exemplare ( $\text{minus}_{i,j}$ )
- Lista tabu reține mutările efectuate într-o iterație
  - Mutare – moneda adăugată/eliminată din sumă

# Strategii de căutare locală – Căutare tabu



## □ Exemplu

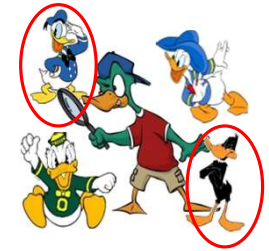
- $S = 500$ , penalizare = 500,  $n = 7$

$S=500$	$m_1$	$m_2$	$m_3$	$m_4$	$m_5$	$m_6$	$m_7$
$v_i$	10	50	15	20	100	35	5
$b_i$	5	2	6	5	5	3	10

Stare curentă	Val. f	Listă tabu	Stări vecine	Mutări	Val. f
2 0 1 0 0 2 1	384	$\emptyset$	2 0 1 3 0 2 1	plus <sub>4,3</sub>	321
			2 0 1 0 0 3 1	plus <sub>6,1</sub>	348
			0 0 1 0 0 2 1	minus <sub>1,2</sub>	406
2 0 1 3 0 2 1	321	plus <sub>4,3</sub>	2 0 1 3 5 2 1	plus <sub>5,5</sub>	316
			2 0 1 1 0 2 1	minus <sub>4,2</sub>	363
			2 1 1 3 0 2 1	plus <sub>2,1</sub>	270
2 1 1 3 0 2 1	270	plus <sub>4,3</sub> plus <sub>2,1</sub>	...		

- Soluția finală: 4 1 5 4 1 3 10 ( $f = -28$ )

# Strategii de căutare locală – Căutare tabu



## □ Exemplu

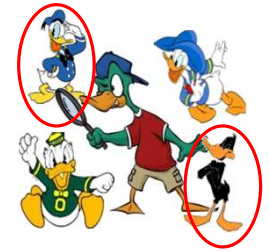
- $S = 500$ , penalizare = 500,  $n = 7$

$S=50$	$m_1$	$m_2$	$m_3$	$m_4$	$m_5$	$m_6$	$m_7$
$v_i$	10	50	15	20	100	35	5
$b_i$	5	2	6	5	4	3	10

Stare curentă	Val. f	Listă tabu	Stări vecine	Mutări	Val. f
2 0 1 0 0 2 1	384	$\emptyset$	1 0 1 4 0 2 1	minus <sub>1,1</sub> , plus <sub>4,4</sub>	311
			2 0 4 0 1 2 1	plus <sub>3,3</sub> , minus <sub>5,1</sub>	235
			2 0 1 0 4 2 6	plus <sub>5,4</sub> , plus <sub>7,5</sub>	450
2 0 4 0 1 2 1	235	plus <sub>3,3</sub> , minus <sub>5,1</sub>	2 0 5 0 5 2 1	plus <sub>3,1</sub> , plus <sub>5,4</sub>	315
			5 0 4 0 4 2 1	plus <sub>1,3</sub> , plus <sub>5,3</sub>	399
			2 2 4 0 5 2 1	plus <sub>2,2</sub> , plus <sub>5,4</sub>	739
2 0 4 0 1 2 1	235	plus <sub>3,3</sub> , minus <sub>5,1</sub>	...		

- Soluția finală: 4 1 5 4 1 3 10 ( $f = -28$ )

# Strategii de căutare locală – Căutare tabu



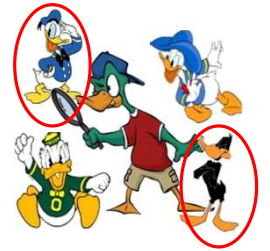
## □ Algoritm

```
bool TS(S) {  
    Select x∈S    //S - spațiul de căutare  
    x*=x          //cea mai bună soluție (până la un mom. dat)  
    k = 0         // numărul de iterații  
    T = ∅         // listă de mutări tabu  
    while (not termination criteria) {  
        k = k + 1  
        generate a subset of solutions in the neighborhood N-T of x  
        choose the best solution s from N-T and set x=s.  
        if f(x)<f(x*) then x*=x  
        update T with moves of generating x  
    } //while  
    return x*;  
}
```

## ■ Criterii de terminare

- Număr fix de iterații
- Număr de iterații fără îmbunătățiri
- Apropierea suficientă de soluție (dacă aceasta este cunoscută)
- Epuizarea elementelor nevizitate dintr-o vecinătate

# Strategii de căutare locală – Căutare tabu



## □ Analiza căutării

- Convergența rapidă spre optimul global

## □ Avantaje

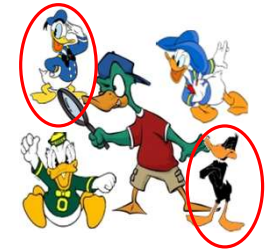
- Algoritm general și simplu de implementat
- Algoritm rapid (poate oferi soluția optimă globală în scurt timp)

## □ Dezavantaje

- Stabilirea stărilor vecine în spații continue
- Număr mare de iterații
- Nu se garantează atingerea optimului global



# Strategii de căutare locală – Căutare tabu



## □ Aplicații

- Determinarea structurii tridimensionale a proteinelor în secvențe de aminoacizi (optimizarea unei funcții de potențial energetic cu multiple optime locale)
- Optimizarea traficului în rețele de telecomunicații
- Planificare în sisteme de producție
- Proiectarea rețelelor de telecomunicații optice
- Ghidaj automat pentru vehicule
- Probleme în grafuri (partiționări)
- Planificări în sistemele de audit
- Planificări ale task-urilor paralele efectuate de procesor (multiprocesor)
- Optimizarea structurii electromagnetice (imagistica rezonanței magnetice medicale)
- Probleme de asignare quadratică (proiectare VLSI)
- Probleme de combinatorică (ricsac, plata sumei)
- Problema tăierii unei bucăți în mai multe părți
- Controlul structurilor spațiale (NASA)
- Optimizarea proceselor cu decizii multi-stagiu
- Probleme de transport
- Management de portofoliu
- Chunking



# Recapitulare

- ❑ SCI best first search
  - Nodurile mai bine evaluate (de cost mai mic) au prioritate la expandare
  - SCI de tip greedy
    - ❑ minimizarea costului de la starea curentă la starea obiectiv –  $h(n)$
    - ❑ Timp de căutare < SCnI
    - ❑ Ne-completă
    - ❑ Ne-optimală
  - SCI de tip A\*
    - ❑ minimizarea costului de la starea inițială la starea curentă –  $g(n)$  – și a costului de la starea curentă la starea obiectiv –  $h(n)$
    - ❑ Evitarea repetării stărilor
    - ❑ Fără supraestimarea lui  $h(n)$
    - ❑ Timp și spațiu de căutare mare → în funcție de euristica folosită
    - ❑ Complet
    - ❑ Optimal



# Recapitulare

## □ SC locale

### ■ Algoritmi iterativi

- Lucrează cu o soluție potențială → soluția optimă
- Se pot bloca în optime locale

	Alegerea stării următoare	Criteriul de acceptare	Convergența
HC	Cel mai bun vecin	Vecinul este mai bun decât starea curentă	Optim local sau gloabl
SA	Un vecin oarecare	Vecinul este mai bun sau mai slab (acceptat cu probabilitatea $p$ ) decât starea curentă	Optim global (lentă)
TS	Cel mai bun vecin nevizitat încă	Vecinul este mai bun decât starea curentă	Optim global (rapidă)

# Cursul următor

---

## A. Scurtă introducere în Inteligența Artificială (IA)

## B. Rezolvarea problemelor prin căutare

- Definirea problemelor de căutare
- Strategii de căutare
  - Strategii de căutare neinformate
  - Strategii de căutare informate
  - Strategii de căutare locale (Hill Climbing, Simulated Annealing, Tabu Search, Algoritmi evolutivi, PSO, ACO)
  - Strategii de căutare adversială

## C. Sisteme inteligente

- Sisteme care învață singure
  - Arbori de decizie
  - Rețele neuronale artificiale
  - Mașini cu suport vectorial
  - Algoritmi evolutivi
- Sisteme bazate pe reguli
- Sisteme hibride

# Cursul următor –

## Materiale de citit și legături utile

---

- ❑ capitolul 14 din *C. Groșan, A. Abraham, Intelligent Systems: A Modern Approach, Springer, 2011*
- ❑ *M. Mitchell, An Introduction to Genetic Algorithms, MIT Press, 1998*
- ❑ capitolul 7.6 din *A. A. Hopgood, Intelligent Systems for Engineers and Scientists, CRC Press, 2001*
- ❑ Capitolul 9 din *T. M. Mitchell, Machine Learning, McGraw-Hill Science, 1997*

- 
- Informațiile prezentate au fost colectate din diferite surse bibliografice, precum și din cursurile de inteligență artificială ținute în anii anteriori de către:
    - Conf. Dr. Mihai Oltean – [www.cs.ubbcluj.ro/~moltean](http://www.cs.ubbcluj.ro/~moltean)
    - Lect. Dr. Crina Groșan - [www.cs.ubbcluj.ro/~cgrosan](http://www.cs.ubbcluj.ro/~cgrosan)
    - Prof. Dr. Horia F. Pop - [www.cs.ubbcluj.ro/~hfpop](http://www.cs.ubbcluj.ro/~hfpop)