UNIVERSITATEA BABEŞ-BOLYAI
Facultatea de Matematică şi Informatică

INTELIGENȚĂ ARTIFICIALĂ

Rezolvarea problemelor de căutare

Strategii de căutare neinformată

Laura Dioşan

Sumar

A. Scurtă introducere în Inteligenţa 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 învaţă singure
  - Arbori de decizie
  - Reţele neuronale artificiale
  - Maşini cu suport vectorial
  - Algoritmi evolutivi
- Sisteme bazate pe reguli
- Sisteme hibride

Sumar

- Probleme

- Rezolvarea problemelor
  - Paşi în rezolvarea problemelor

- Rezolvarea problemelor prin căutare
  - Paşi în rezolvarea problemelor prin căutare
  - Tipuri de strategii de căutare

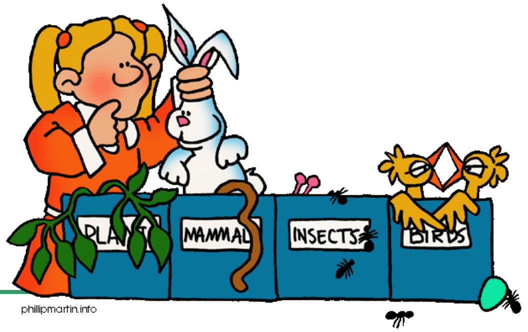
Materiale de citit şi legături utile

- capitolele I.1, I.2, II.3 şi II.4 din *S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Prentice Hall, 1995*
- capitolele 1 - 4 din *C. Groşan, 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

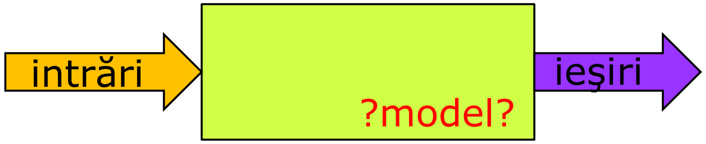
- Două mari categorii de probleme:
  - Rezolvabile în mod determinist
    - Calculul sinusului unui unghi sau a rădăcinii pătrate dintr-un număr
  - Rezolvabile în mod stocastic
    - Probleme din lumea reală → proiectarea unui ABS
    - Presupun căutarea unei soluţii → metode ale IA

Probleme

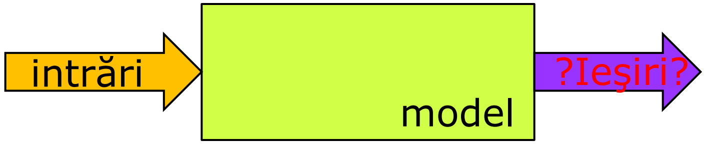


- Tipologie
  - Probleme de căutare/optimizare

    - Planificare, proiectarea sateliţilor
  - Probleme de modelare



    - Predicţii, clasificări
  - Probleme de simulare



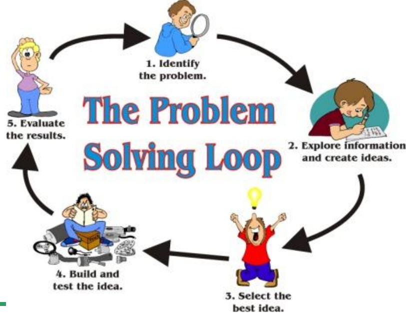
    - Teoria jocurilor economice

Rezolvarea problemelor

- Constă în identificarea unei soluţii
  - În informatică (IA) → proces de căutare
  - În inginerie şi matematică → proces de optimizare

- Cum?
  - Reprezentarea soluţiilor (parţiale) → puncte în spaţiul de căutare
  - Proiectarea unor operatori de căutare → transformă o posibilă soluţie în altă soluţie

Paşi în rezolvarea problemelor

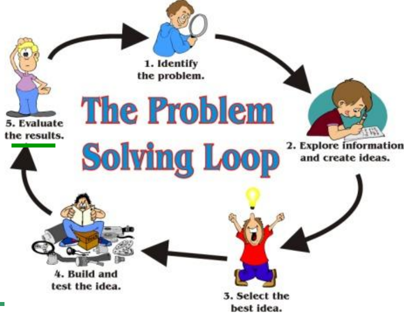


- Definirea problemei
- Analiza problemei
- Alegerea unei tehnici de rezolvare
  - căutare
  - reprezentarea cunoştinţelor
  - abstractizare

Rezolvarea problemelor prin căutare

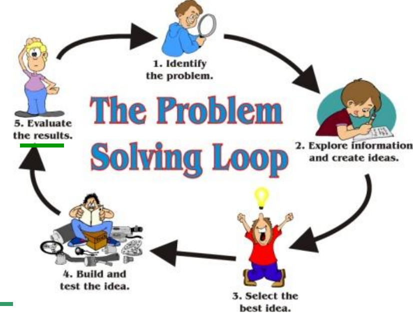
- bazată pe urmărirea unor obiective
- compusă din acţiuni care duc la îndeplinirea unor obiective
  - fiecare acţiune modifică o anumită stare a problemei
- succesiune de acţiuni care transformă starea iniţială a problemei în stare finală

Paşi în rezolvarea problemelor prin căutare – Definirea problemei



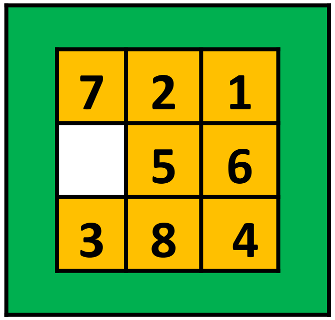
- Definirea problemei implică stabilirea:
  - unui spaţiu de stări
    - toate configuraţiile posibile, fără enumerarea obligatorie a tuturor configuraţiilor
    - reprezentare
      - explicită – construirea (în memorie) a tuturor stărilor posibile
      - implicită – utilizarea unor structuri de date şi a unor funcţii (operatori)
  - unei/unor stări iniţiale
  - unei/unor stări finale - obiectiv
  - unui/unor drum(uri)
    - succesiuni de stări
  - unui set de reguli (acţiuni)
    - funcţii succesor (operatori) care precizează starea următoare a unei stări
    - funcţii de cost care evaluează
      - trecerea dintr-o stare în alta
      - un întreg drum
    - funcţii obiectiv care verifică dacă s-a ajuns într-o stare finală

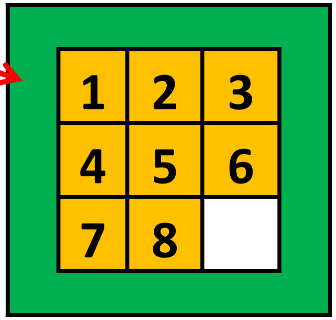
Paşi în rezolvarea problemelor prin căutare – Definirea problemei



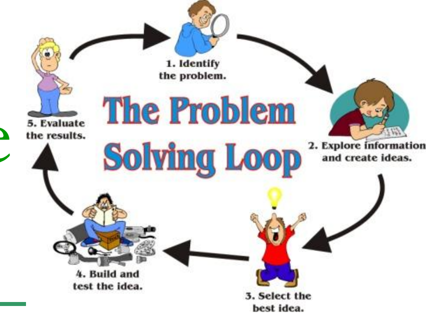
- Exemple

  - Joc puzzle cu 8 piese
    - Spaţiul stărilor – configuraţii ale tablei de joc cu 8 piese
    - Starea iniţială – o configuraţie oarecare
    - Starea finală – o configuraţie cu piesele aranjate într-o anumită ordine
    - Reguli → acţiuni albe
      - Condiţii: mutarea în interiorul tablei
      - Transformări: spaţiul alb se mişcă în sus, în jos, la stânga sau la dreapta
    - Soluţia – secvenţa optimă de acţiuni albe

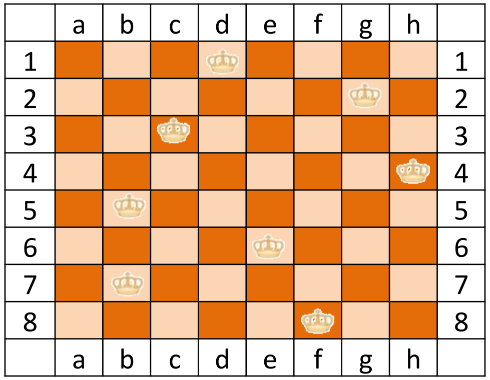




Paşi în rezolvarea problemelor prin căutare
Definirea problemei

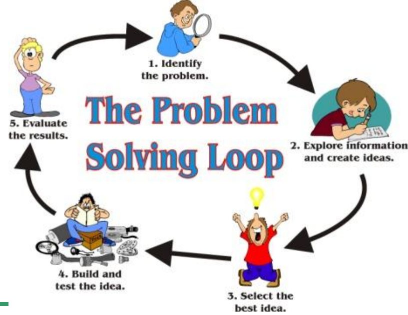


- Exemple
  - Joc cu *n* dame
    - Spaţiul stărilor – configuraţii ale tablei de joc cu *n* regine



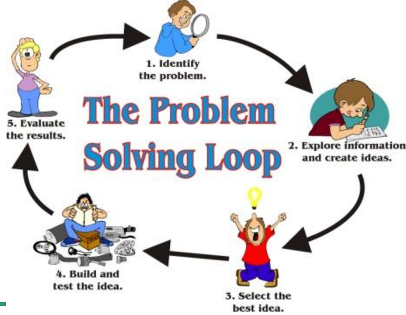
    - Starea iniţială – o configuraţie fără regine
    - Starea finală – o configuraţie cu *n* regine care nu se atacă
    - Reguli → amplasarea unei regine pe tablă
      - Condiţii: regina amplasată nu este atacată de nici o regină existentă pe tablă
      - Transformări: amplasarea unei noi regine într-o căsuţă de pe tabla de joc
    - Soluţia – amplasarea optimă a reginelor pe tablă

Paşi în rezolvarea problemelor prin căutare
Analiza problemei



- Se poate descompune problema?
  - Sub-problemele sunt independente sau nu?
- Universul stărilor posibile este predictibil?
- Se doreşte obţinerea oricărei soluţii sau a unei soluţii optime?
- Soluţia dorită constă într-o singură stare sau într-o succesiune de stări?
- Sunt necesare multiple cunoştinţe pentru a limita căutarea sau chiar pentru a identifica soluţia?
- Problema este conversaţională sau solitară?
  - Este sau nu nevoie de interacţiune umană pentru rezolvarea ei?

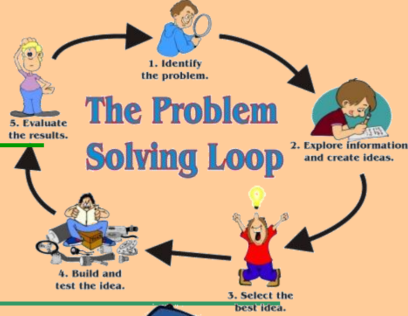
Paşi în rezolvarea problemelor prin căutare Alegerea unei tehnici de rezolvare



- Rezolvarea prin utilizarea regulilor (în combinaţie cu o strategie de control) de deplasare în spaţiul problemei până la găsirea unui drum între starea iniţială şi cea finală

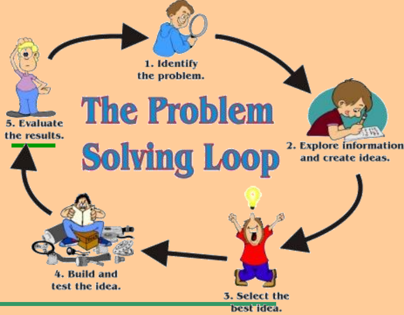
- Rezolvare prin căutare
  - Examinarea sistematică a stărilor posibile în vederea identificării
    - unui drum de la o stare iniţială la o stare finală
    - unei stări optime
  - Spaţiul stărilor = toate stările posibile + operatorii care definesc legăturile între stări

Paşi în rezolvarea problemelor prin căutare – Alegerea unei tehnici de rezolvare



- Rezolvare prin căutare
  - Strategii de căutare multiple ➔ cum alegem o strategie?
    - Complexitatea computaţională (temporală şi spaţială)
    - Completitudine → algoritmul se sfârşeşte întotdeauna şi găseşte o soluţie (dacă ea există)
    - Optimalitate → algoritmul găseşte soluţia optimă (costul optim al drumului de la starea iniţială la starea finală)

Paşi în rezolvarea problemelor prin căutare – Alegerea unei tehnici de rezolvare

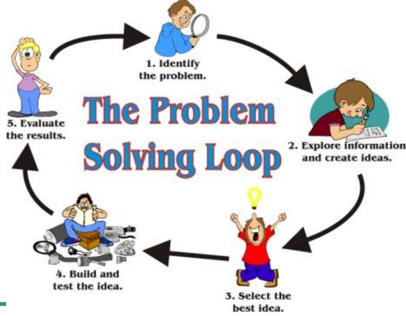


- Rezolvare prin căutare
  - Strategii de căutare multiple ➔ cum alegem o strategie?
    - Complexitatea computaţională (temporală şi spaţială)
      - Performanţa strategiei depinde de:
        - Timpul necesar rulării
        - Spaţiul (memoria) necesară rulării

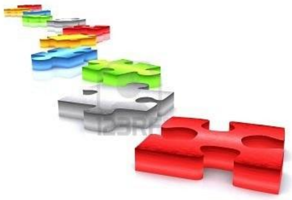
          Factori interni
        - Mărimea intrărilor algoritmului
        - Viteza calculatorului
        - Calitatea compilatorului

          Factori externi
    - Se măsoară cu ajutorul complexităţii → Eficienţă computaţională
      - Spaţială → memoria necesară identificării soluţiei
        - *S(n)* – cantitatea de memorie utilizată de cel mai bun algoritm A care rezolvp o problemă de decizie *f* cu *n* date de intrare
      - Temporală → timpul necesar identificării soluţiei
        - *T(n)* – timpul de rulare (numărul de paşi) al celui mai bun algoritm A care rezolvă o problemă de decizie *f* cu *n* date de intrare

Paşi în rezolvarea problemelor prin căutare
Alegerea unei tehnici de rezolvare

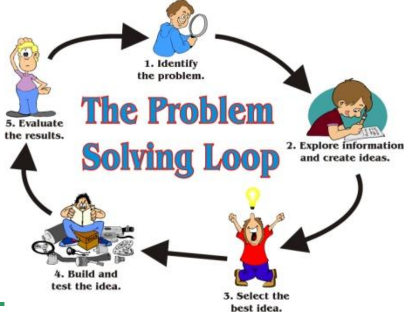


- Rezolvarea problemelor prin căutare poate consta în:
  - Construirea progresivă a soluţiei

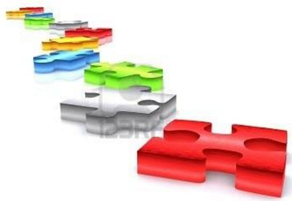


  - Identificarea soluţiei potenţiale optime

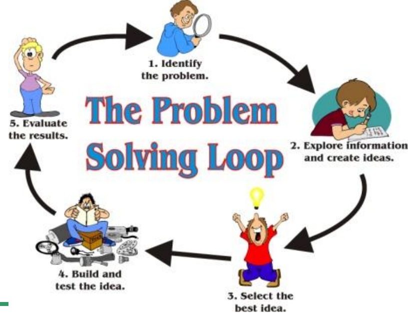
Paşi în rezolvarea problemelor prin căutare Alegerea unei tehnici de rezolvare



- Rezolvarea problemelor prin căutare poate consta în:
  - Construirea progresivă a soluţiei
    - Componentele problemei
      - Stare iniţială
      - Operatori (funcţii succesor)
      - Stare finală
      - Soluţia = un drum (de cost optim) de la starea iniţială la starea finală
    - Spaţiul de căutare
      - Mulţimea tuturor stărilor în care se poate ajunge din starea iniţială prin aplicarea operatorilor
      - stare = o componentă a soluţiei
    - Exemple
      - Problema comisului voiajor
    - Algoritmi
      - Ideea de bază: se începe cu o componentă a soluţiei şi se adaugă noi componente până se ajunge la o soluţie completă
      - Recursivi → se re-aplică până la îndeplinirea unei condiţii
      - Istoricul căutării (drumul parcurs de la starea iniţială la starea finală) este reţinut în liste de tip LIFO sau FIFO
    - Avantaje
      - Nu necesită cunoştinţe (informaţii inteligente)

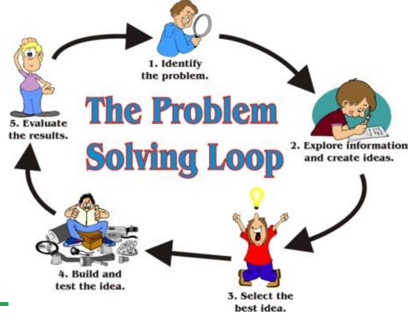


Paşi în rezolvarea problemelor prin căutare
Alegerea unei tehnici de rezolvare

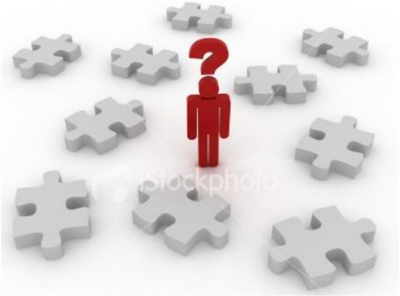


- Rezolvarea problemelor prin căutare poate consta în:
  - Identificarea soluţiei potenţiale optime
    - Componentele problemei
      - Condiţii (constrângeri) pe care trebuie să le satisfacă (parţial sau total) soluţia
      - Funcţie de evaluare a unei soluţii potenţiale → identificareaa optimului
    - Spaţiul de căutare
      - mulţimea tuturor soluţiilor potenţiale complete
      - Stare = o soluţie completă
    - Exemple
      - Problema celor 8 regine
    - Algoritmi
      - Ideea de bază: se începe cu o stare care nu respectă anumite constrângeri pentru a fi soluţie optimă şi se efectuează modificări pentru a elimina aceste violări
      - Iterativi → se memorează o singură stare şi se încearcă îmbunătăţirea ei
      - Istoricul căutării nu este reţinut
    - Avantaje
      - Simplu de implementat
      - Necesită puţină memorie
      - Poate găsi soluţii rezonabile în spaţii de căutare (continue) foarte mari pentru care alţi algoritmi sistematici nu pot fi aplicaţi

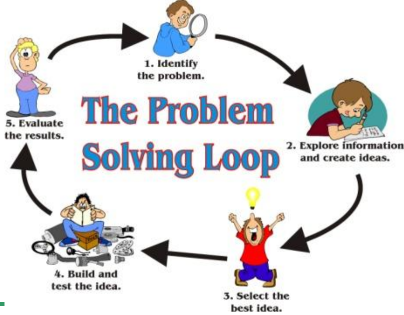
Paşi în rezolvarea problemelor prin căutare Alegerea unei tehnici de rezolvare



- Rezolvarea problemelor prin căutare presupune
  - algoritmi cu o complexitate ridicată (probleme NP-complete)
  - căutarea într-un spaţiu exponenţial

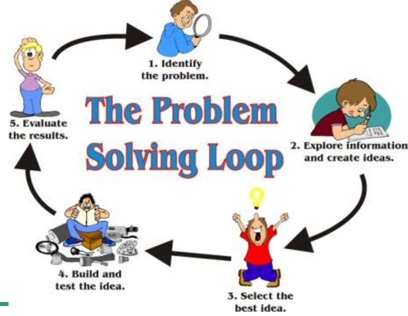


Paşi în rezolvarea problemelor prin căutare
Alegerea unei tehnici de rezolvare



- Tipologia strategiilor de căutare
  - În funcţie de modul de **generare** a soluţiei
    - Căutare **constructivă**
      - Construirea progresivă a soluţiei
      - Ex. TSP
    - Căutare **perturbativă**
      - O soluţie candidat este modificată în vederea obţinerii unei noi soluţii candidat
      - Ex. SAT - Propositional Satisfiability Problem
  - În funcţie de modul de **traversare** a spaţiului de căutare
    - Căutare **sistematică**
      - Traversarea completă a spaţiului
        - Ideintificarea soluţiei dacă ea există → algoritmi compleţi
    - Căutare **locală**
      - Traversarea spaţiului de căutare dintr-un punct în alt punct vecin → algoritmi incompleţi
      - O stare a spaţiului poate fi vizitată de mai multe ori
  - În funcţie de elementele de **certitudine** ale căutării
    - Căutare **deterministă**
      - Algoritmi de identificare exactă a soluţiei
    - Căutare **stocastică**
      - Algoritmi de aproximare a soluţiei
  - În funcţie de stilul de **explorare** a spaţiului de căutare
    - Căutare **secvenţială**
    - Căutare **paralelă**

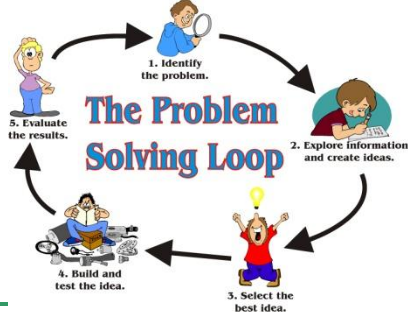
Paşi în rezolvarea problemelor prin căutare
Alegerea unei tehnici de rezolvare



Tipologia strategiilor de căutare

- În funcţie de **scopul** urmărit
  - Căutare **uni-obiectiv**
    - Soluţia trebuie să satisfacă o signură condiţie/constrângere
  - Căutare **multi-obiectiv**
    - Soluţia trebuie să satisfacă mai multe condiţii (obiective)
- În funcţie de **numărul de soluţii optime**
  - Căutare **uni-modală**
    - Există o singură soluţie optimă
  - Căutare **multi-modală**
    - Există mai multe soluţii optime
- În funcţie de tipul de **algoritm** folosit
  - Căutare de-a lungul unui **număr finit de paşi**
  - Căutare **iterativă**
    - Algoritmii converg către soluţie
  - Căutare **euristică**
    - Algoritmii oferă o aproximare a soluţiei
- În funcţie de **mecanismul** căutării
  - Căutare **tradiţională**
  - Căutare **modernă**
- În funcţie de **locul** în care se desfăşoară căutarea în spaţiul de căutare
  - Căutare **locală**
  - Căutare **globală**

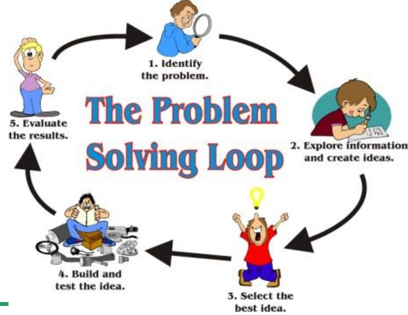
Paşi în rezolvarea problemelor prin căutare
Alegerea unei tehnici de rezolvare



Tipologia strategiilor de căutare

- În funcţie de tipul **(linearitatea) constrângerilor**
  - Căutare **liniară**
  - Căutare **neliniară**
    - Clasică (deterministă)
      - Directă – bazată doar pe evaluarea funcţiei obiectiv
      - Indirectă – bazată şi pe derivata (I si/sau II) a funcţiei obiectiv
    - Enumerativă
      - În funcţie de modul de stabilire a soluţiei
        - Ne-informată → soluţia se ştie doar când ea coincide cu starea finală
        - Informată → se lucrează cu o funcţie de evaluare a unei soluţii parţiale
      - În funcţie de tipul spaţiului de căutare
        - Completă – spaţiu finit (dacă soluţia există, ea poate fi găsită)
        - Incompletă – spaţiu infinit
    - Stocastică
      - Bazată pe elemente aleatoare
- În funcţie de **agenţii** implicaţi în căutare
  - Căutare realizată de un **singur agent** → fără piedici în atingerea obiectivelor
  - Căutare **adversială** → adversarul aduce o incertitudine în realizarea obiectivelor

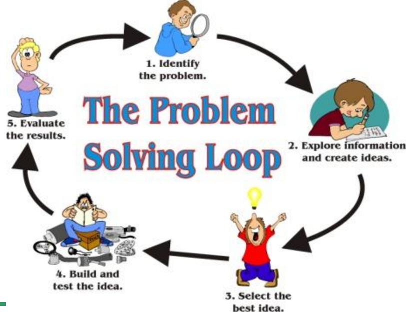
Paşi în rezolvarea problemelor prin căutare
Alegerea unei tehnici de rezolvare



Exemplu

- Tipologia strategiilor de căutare
  - În funcţie de modul de generare a soluţiei
    - **Căutare constructivă**
    - Căutare perturbativă
  - În funcţie de modul de traversare a spaţiului de căutare
    - **Căutare sistematică**
    - Căutare locală
  - În funcţie de elementele de certitudine ale căutării
    - **Căutare deterministă**
    - Căutare stocastică
  - În funcţie de stilul de explorare a spaţiului de căutare
    - **Căutare secvenţială**
    - Căutare paralelă

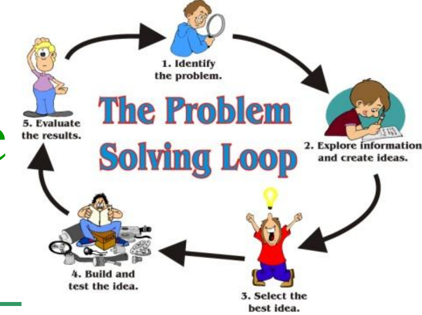
Paşi în rezolvarea problemelor prin căutare
Alegerea unei tehnici de rezolvare

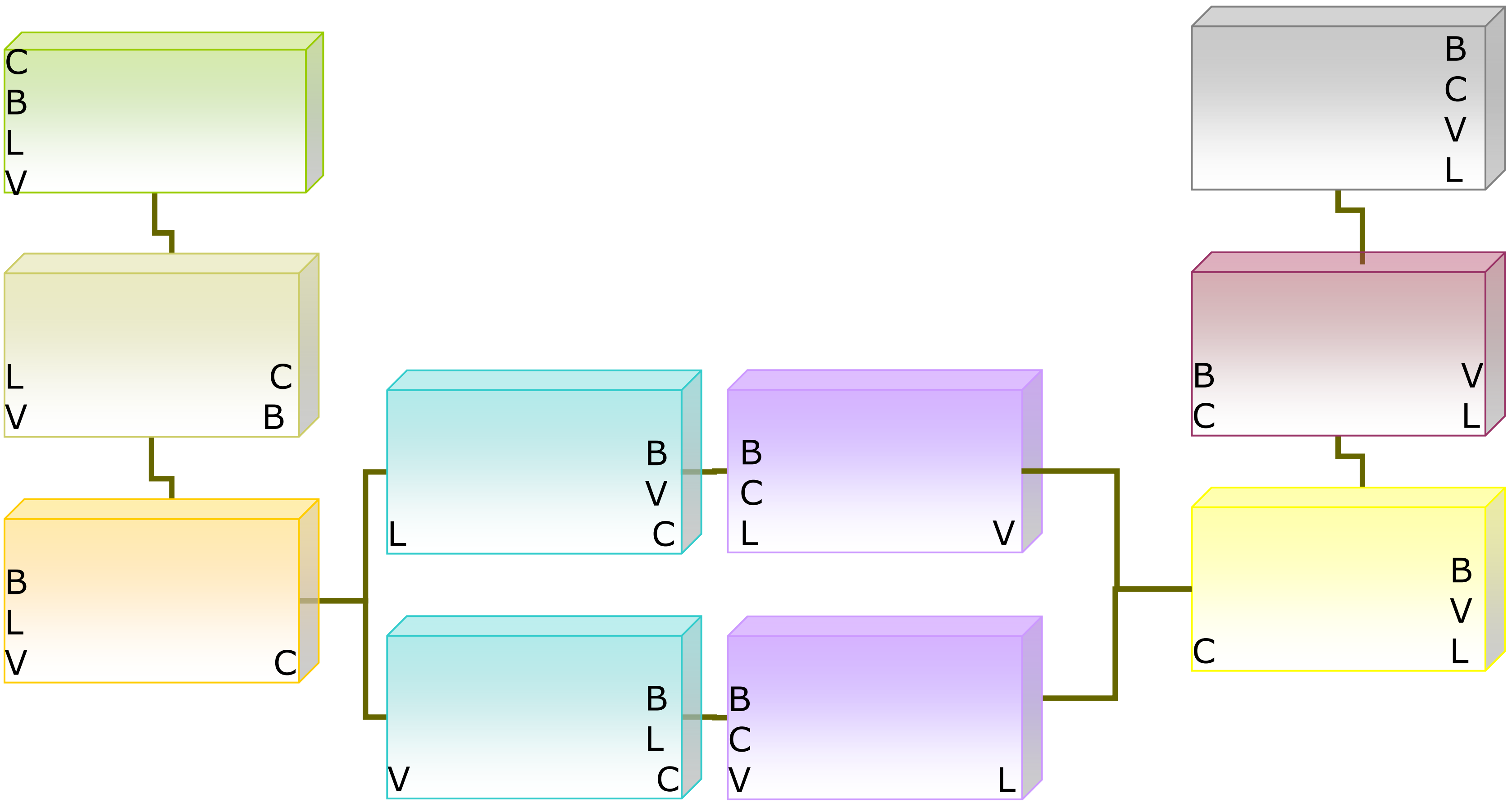


Exemplu

- Problema "capra, varza şi lupul"
  - Se dau:
    - o capră, o varză şi un lup pe malul unui râu
    - o barcă cu barcagiu
  - Se cere
    - Să se traverseze toţi pasagerii pe malul celălalt al râului
    - cu următoarele condiţii
      - în barcă există doar 2 locuri
      - nu pot rămâne pe acelaşi mal
        - capra şi varza
        - lupul şi capra

Paşi 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 şi operaţii
  - Reprezentare

Strategii de căutare – Elemente fundamentale – TAD Listă

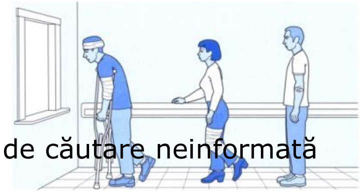
- Domeniu
  - *D = {l | l = (el1, el2, …)*, unde el_i, *i=1,2,3…*, sunt de acelaşi tip *TE* (*tip element*) şi fiecare element el_i, *i=1,2,3…*, are o poziţie unică în *l* de tip *TP* (*TipPoziţie*)}
- Operaţii
  - •Creare(l)
  - •Prim(l)
  - •Ultim(l)
  - •Următor(l,p)
  - •Anterior(l,p)
  - •Valid(l,p)
  - •getElement(l,p)
  - •getPoziţie(l,e)
  - •Modifică(l,p,e)
  - •AdăugareLaÎnceput(l,e)
  - •AdăugareLaSfârşit(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) înlănţuite, etc
- Cazuri particulare
  - Stivă – LIFO
  - Coadă – FIFO
  - Coadă cu priorităţi

Strategii de căutare – Elemente fundamentale – TAD Listă

- Domeniu
  - *D = {l | l = (el1, el2, …), unde* el_i*, i=1,2,3…*, sunt de acelaşi tip *TE* (*tip element*) şi fiecare element el_i, *i=1,2,3…*, are o poziţie unică în *l* de tip *TP* (*TipPoziţie*)}
- Operaţii
  - •Creare(l)
  - •Prim(l)
  - •Ultim(l)
  - •Următor(l,p)
  - •Anterior(l,p)
  - •Valid(l,p)
  - •getElement(l,p)
  - •getPoziţie(l,e)
  - •Modifică(l,p,e)
  - •AdăugareLaÎnceput(l,e)
  - •AdăugareLaSfârşit(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) înlănţuite, etc
- Cazuri particulare
  - Stivă – LIFO
  - Coadă – FIFO
  - Coadă cu priorităţi

Strategii de căutare – Elemente fundamentale – TAD Listă

- Domeniu
  - *D = {l | l = (el1, el2, …),* unde *el_i, i=1,2,3…,* sunt de acelaşi tip *TE* (*tip element*) şi fiecare element *el_i, i=1,2,3…,* are o poziţie unică în *l* de tip *TP* (*TipPoziţie*)}
- Operaţii
  - •Creare(l)
  - •Prim(l)
  - •Ultim(l)
  - •Următor(l,p)
  - •Anterior(l,p)
  - •Valid(l,p)
  - •getElement(l,p)
  - •getPoziţie(l,e)
  - •Modifică(l,p,e)
  - •AdăugareLaÎnceput(l,e)
  - •AdăugareLaSfârşit(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) înlănţuite, etc
- Cazuri particulare
  - Stivă – LIFO
  - Coadă – FIFO
  - Coadă cu priorităţi



Strategii de căutare – Elemente fundamentale – TAD Graf

- Domeniu – container de noduri si legături între noduri
  - $D = \{nod_1, nod_2, \ldots, nod_n,\ leg_1,\ leg_2,\ \ldots, leg_m$, unde nod_i, cu $i=1,2,\ldots,n$ sunt noduri, iar leg_i, cu $i=1,2,\ldots,m$ sunt muchii între noduri}

- Operaţii
  - creare
  - creareNod
  - traversare
  - getIterator
  - distrugere

- Reprezentare
  - Lista muchilor
  - Lista de adiacenţă (Tradiţională şi Modernă)
  - Matricea de adiacenţă (Tradiţională şi Modernă)
  - Matricea de incidenţă

- Cazuri particulare
  - Grafuri orientate şi 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
  - $D = \{nod_1, nod_2, \ldots, nod_n, leg_1, leg_2, \ldots, leg_m$, unde nod_i, cu $i=1,2,\ldots,n$ sunt noduri, iar leg_i, cu $i=1,2,\ldots,m$ sunt muchii între noduri astfel încât să nu se formeze cicluri$\}$

- Operaţii
  - creare
  - creareFrunză
  - adăugareSubarbore
  - getInfoRădăcină
  - getSubarbore
  - traversare
  - getIterator
  - distrugere

- Reprezentare
  - Vectorială
  - Liste înlănţuite ale descendenţilor

- 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 restricţii
  - drum închis
    - nodul iniţial = nodul final
  - circuit
    - un *trail* închis
  - ciclu
    - un *path* închis

Strategii de căutare neinformate (SCnI)

- Caracteristici
  - nu se bazează pe informaţii specifice problemei
  - sunt generale
  - strategii oarbe
  - strategii de tipul forţei brute
- Topologie
  - În funcţie de ordinea expandării stărilor în spaţiul de căutare:
    - SCnI în structuri liniare
      - căutare liniară
      - căutare binară
    - SCnI în structuri ne-liniare
      - căutare în lăţime (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 bidirecţională

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] => O(1)*
    - Cel mai slab caz: *elem ∉ list => T(n) = n +1 => O(n)*
    - Cazul mediu: *T(n) = (1 + 2 + … + n + (n+1))/(n+1) => O(n)*
  - Complexitate spaţială
    - *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

- **Aplicaţii**
  - 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**

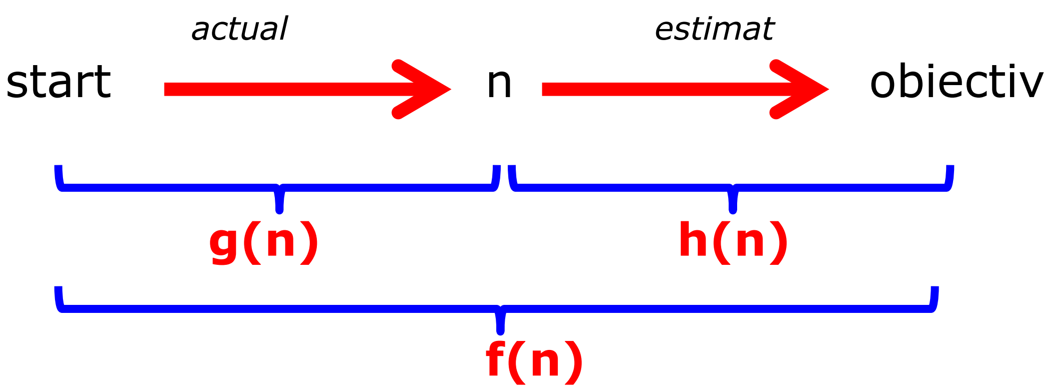
```
bool BS(elem, list){
      found = false;
      left = 1;
      right = list.length;
      while((left < right) && (!found)){
            middle = left + (right - left)/2;
            if (element == list[middle])
                found = true;
            else
                  if (element < list[middle])
                      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 şi $T(n) = T(n/2) + 1$, altfel
    - Pp. că $n = 2k \Rightarrow k = \log_2 n$
    - Pp. că $2k < n < 2k+1 \Rightarrow k < log2n < k + 1$
    - $T(n) \quad = T(n/2) + 1$
    - $T(n/2) \quad = T(n/22) + 1$
    - …
    - $T(n/2k-1) = T(n/2k) + 1$
    - ------------------------------
    - $T(n) = k + 1 = \log_2 n + 1$
  - Complexitate spaţială – $S(n) = n$
  - Completitudine – da
  - Optimalitate – da
- **Avantaje**
  - Complexitate temporală redusă faţă de căutarea liniară
- **Dezavantaje**
  - Lucrul cu vectori (trebuie accesate elemente indexate) sortaţi
- **Aplicaţii**
  - Jocul ghicirii unui număr
  - Căutare într-o carte de telefon/dicţionar

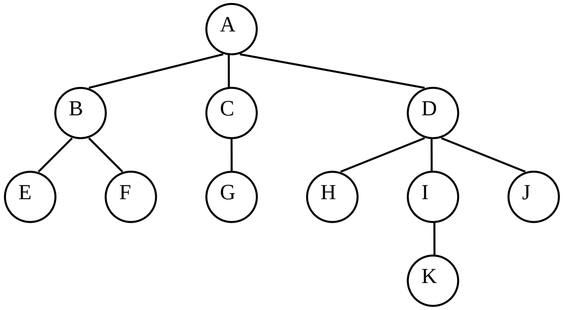
SC în structuri arborescente

- Noţiuni necesare
  - *f(n)* – funcţie de evaluare pentru estimarea costului soluţiei prin nodul (starea) *n*
  - *h(n)* – funcţie euristică pentru estimarea costului drumului de la nodul *n* la nodul obiectiv
  - *g(n)* – funcţie 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 lăţime (breadth-first search – BFS)

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



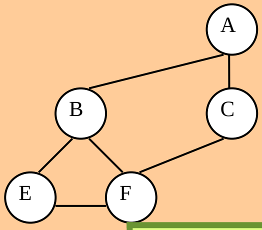
  - Ordinea vizitării: A, B, C, D, E, F, G, H, I, J, K
- **Algoritm**

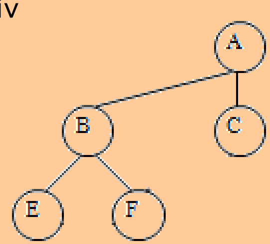
```
bool BFS(elem, list){
        found = false;
        visited = Φ;
        toVisit = {start};          //FIFO list
        while((toVisit != Φ) && (!found)){
                node = pop(toVisit);
                visited = visited U {node};
                if (node == elem)
                    found = 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 lăţime (breadth-first search – BFS)

- **Analiza căutării:**
  - Complexitate temporală:
    - *b* – factor de ramificare (nr de noduri fii ale unui nod)
    - *d* - lungimea (adâncimea) soluţiei
    - $T(n) = 1 + b + b^2 + \ldots + b^d \Rightarrow O(b^d)$
  - Complexitate spaţială
    - $S(n) = T(n)$
  - Completitudine
    - Dacă soluţia există, atunci BFS o găseşte
  - Optimalitate
    - nu
- **Avantaje**
  - Găsirea drumului de lungime minimă până la nodul obiectiv (soluţia cea mai puţin adâncă)
- **Dezavantaje**
  - Generarea şi stocarea unui arbore a cărui mărime creşte exponenţial cu adâncimea nodului obiectiv
  - Complexitate temporală şi spaţială exponenţială
  - Experimentul Russel&Norvig????
  - Funcţional doar pentru spaţii de căutare mici



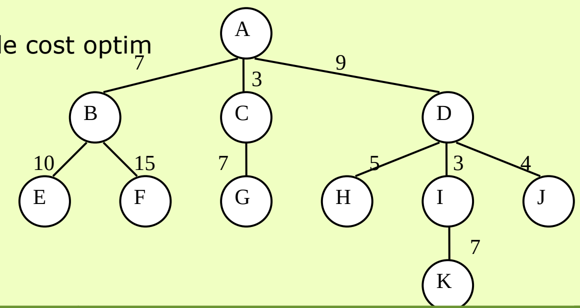


- **Aplicaţii**
  - Identificarea tuturor componentelor conexe într-un graf
  - Identificarea celui mai scurt drum într-un graf
  - Optimizări in reţele de transport → algoritmul Ford-Fulkerson
  - Serializarea/deserializarea unui arbore binar (vs. serializarea în mod sortat) permite reconstrucţia eficientă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

Vizitate deja	De vizitat
Φ	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	Φ

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 obţinute prin expandarea nodului curent se adaugă într-o *listă ORDONATĂ* de tip *FIFO*
    - Se expandează mai întâi nodurile de cost minim
    - Odată obţinut un drum până la nodul ţintă, 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

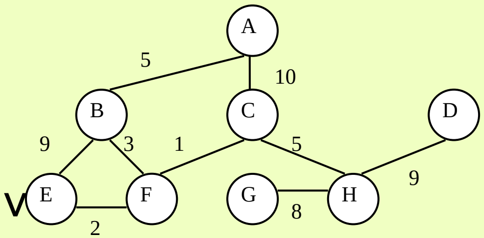
- **Algoritm**

```
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), **K(9+3+7)**
A, C, B, D, G, I, J	H(14), E(17), F(22), **K(19)**
A, C, B, D, G, I, J, H	E(17), F(22), **K(19)**
A, C, B, D, G, I, J, H, E	F(22), **K(19)**
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 complexităţii**
  - Complexitate temporală:
    - *b* – factor de ramificare (nr de noduri fii ale unui nod)
    - *d* - lungimea (adâncimea) soluţiei
    - $T(n) = 1 + b + b^2 + \ldots + b^d \Rightarrow O(b^d)$
  - Complexitate spaţială
    - $S(n) = T(n)$
  - Completitudine
    - Da – dacă soluţia există, atunci UCS o găseşte
  - Optimalitate
    - Da



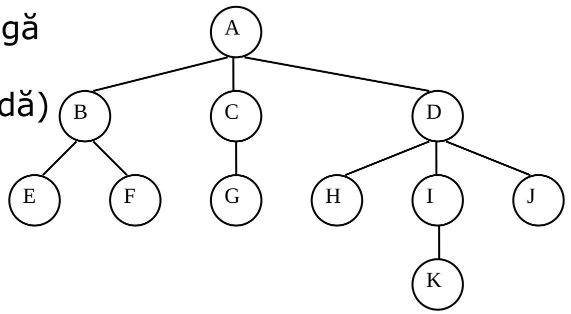
- **Avantaje**
  - Găsirea drumului de cost minim până la nodul obiectiv

Vizitate deja	De vizitat
Φ	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)

- **Dezavantaje**
  - Complexitate temporală şi spaţială exponenţială
- **Aplicaţii**
  - Cel mai scurt drum → algoritmul Dijkstra

SCnI în structuri arborescente căutare în adâncime (depth-first search – DFS)

- **Aspecte teoretice**
  - Expandarea intr-un nod fiu şi înaintarea în adâncime până când
    - Este găsit un nod ţintă (obiectiv) sau
    - Nodul atins nu mai are fii
  - Cu revenirea în cel mai recent nod care mai poate fi explorat
  - Toate nodurile fii obţinute 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 = Φ;
        toVisit = {start};       //LIFO 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 = aux U toVisit;
              }
        }  //while
        return found;
}
```

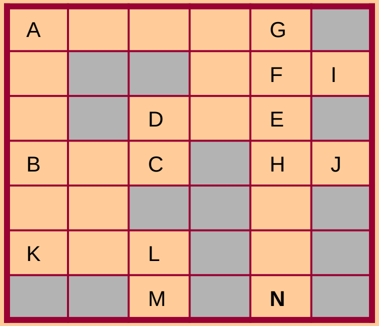
Vizitate deja	De vizitat
Φ	A
A	B, C, D
A, B	E, F, C, D
A, B, E	F, C, D
A, B, E, F	C, D
A, B, E, F, C	G, D
A, B, E, F, C, G	D
A, B, E, F, C, G, D	H, I, J
A, B, E, F, C, G, D, H	I, J
A, B, E, F, C, G, D, H, I	K, J
A, B, E, F, C, G, D, H, I, K	J
A, B, E, F, C, G, D, H, I, K, J	Φ

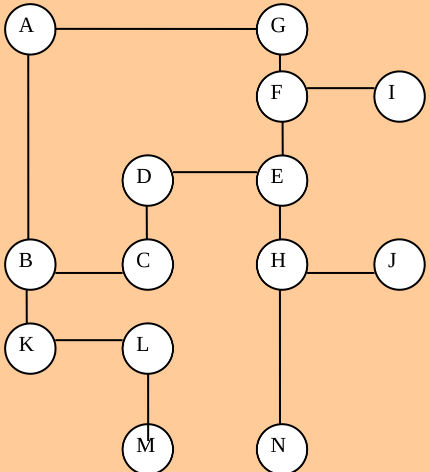
SCnI în structuri arborescente
căutare în adâncime (depth-first search – DFS)

- **Analiza complexităţii**
  - Complexitate temporală:
    - *b* – factor de ramificare (nr de noduri fii ale unui nod)
    - d^{max} - lungimea (adâncimea) maximă a unui arbore explorat
    - $T(n) = 1 + b + b^2 + \ldots + b^{dmax} \Rightarrow O(b^{dmax})$
  - Complexitate spaţială
    - $S(n) = b * d_{max}$
  - Completitudine
    - Nu → algoritmul nu se termină pt drumurile infinite (neexistând suficientă memeorie pt reţinerea nodurilor deja vizitate)
  - Optimalitate
    - Nu → căutarea în adâncime poate găsi un drum soluţie 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 greşite (nenorocoase) fără a putea reveni
    - Ciclu infinit
    - Găsirea unei soluţii mai "lungi" decât soluţia optimă





- **Aplicaţii**
  - Problema labirintului (maze)
  - Identificarea componentelor conexe
  - Sortare topologică
  - Testarea planarităţii

SCnI în structuri arborescente căutare în adâncime (depth-first search – DFS)

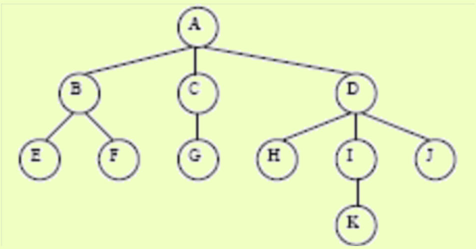
```
bool DFS_edges(elem, list){
   discovered = Φ;
   back = Φ;
   toDiscover = Φ;     //LIFO
   for (all neighbours of start) do
        toDiscover = toDiscover U {(start, neighbour)}
   found = false;
   visited = {start};
   while((toDiscover != Φ) && (!found)){
        edge = pop(toDiscover);
        if (edge.out !∈ visited){
           discovered = discovered U {edge};
           visited = visited U {edge.out}
           if (edge.out == end)
                found = true;
           else{
                aux = Φ;
                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
	Φ	AB, AF	Φ	A
AB	AB	BC, BK, AF	Φ	A, B
BC	AB, BC	CD, BK, AF	Φ	A, B, C
CD	AB. BC, CD	DE, BK, AF	Φ	A, B, C, D
DE	AB, BC, CD, DE	EF, EH, BK, AF	Φ	A, B, C, D, E
EF	AB, BC, CD, DE, EF	FI, FG, EH, BK, AF	Φ	A, B, C, D, E, F
FI	AB, BC, CD, DE, EF, FI	FG, EH, BK, AF	Φ	A, B, C, D, E, F, I
FG	AB, BC, CD, DE, EF, FI, FG	GA, EH, BK, AF	Φ	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, HI, 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 soluţionează 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;
                   }//if found
             }//if dlim
       } //while
       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

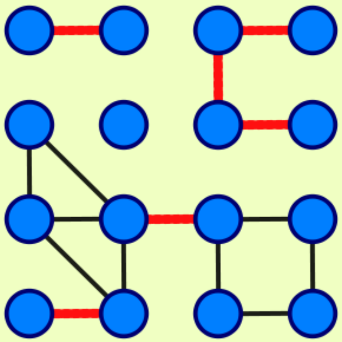
SCnI în structuri arborescente
căutare în adâncime limitată (depth-limited search – DLS)

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

- **Avantaje**
  - Se soluţionează problemele de completitudine ale căutării în adâncime (DFS)

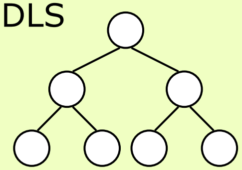
- **Dezavantaje**
  - Cum se alege o limită d_{lim} bună?

- **Aplicaţii**
  - Determinarea "podurilor" într-un graf

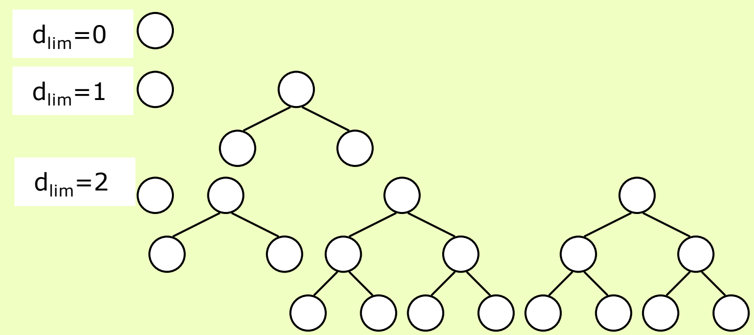


SCnI în structuri arborescente – căutare în adâncime iterativă (iterative deepening depth search – IDDS)

- **Aspecte teoretice**
  - U DLS(d_{lim}), unde d_{lim} = 1, 2, 3, ..., d_{max}
  - Se soluţionează problema stabilirii limitei d_{lim} optime din DLS
  - De obicei, se aplică acolo unde:
    - spaţiul de căutare este mare şi
    - se cunoaşte lungimea (adâncimea) soluţiei



- **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 complexităţii**
  - Complexitate temporală:
    - Nodurile situate la adâncimea d_{max} (în număr de b^{dmax}) se expandează o singură dată=> 1 * b^{dmax}
    - Nodurile situate la adâncimea d_{max}-*1* (în număr de b^{dmax-1}) se expandează de 2 ori => *2 * ($b^{dmax - 1}$)*
    - ...
    - Nodurile situate la adâncimea 1 (în număr de *b*) se expandează de d_{max} ori => d_{max} * b^1
    - Nodurile situate la adâncimea 0 (în număr de 1 - rădăcina) se expandează de d_{max}+1 ori => *(d_{max}+1)*b^0*

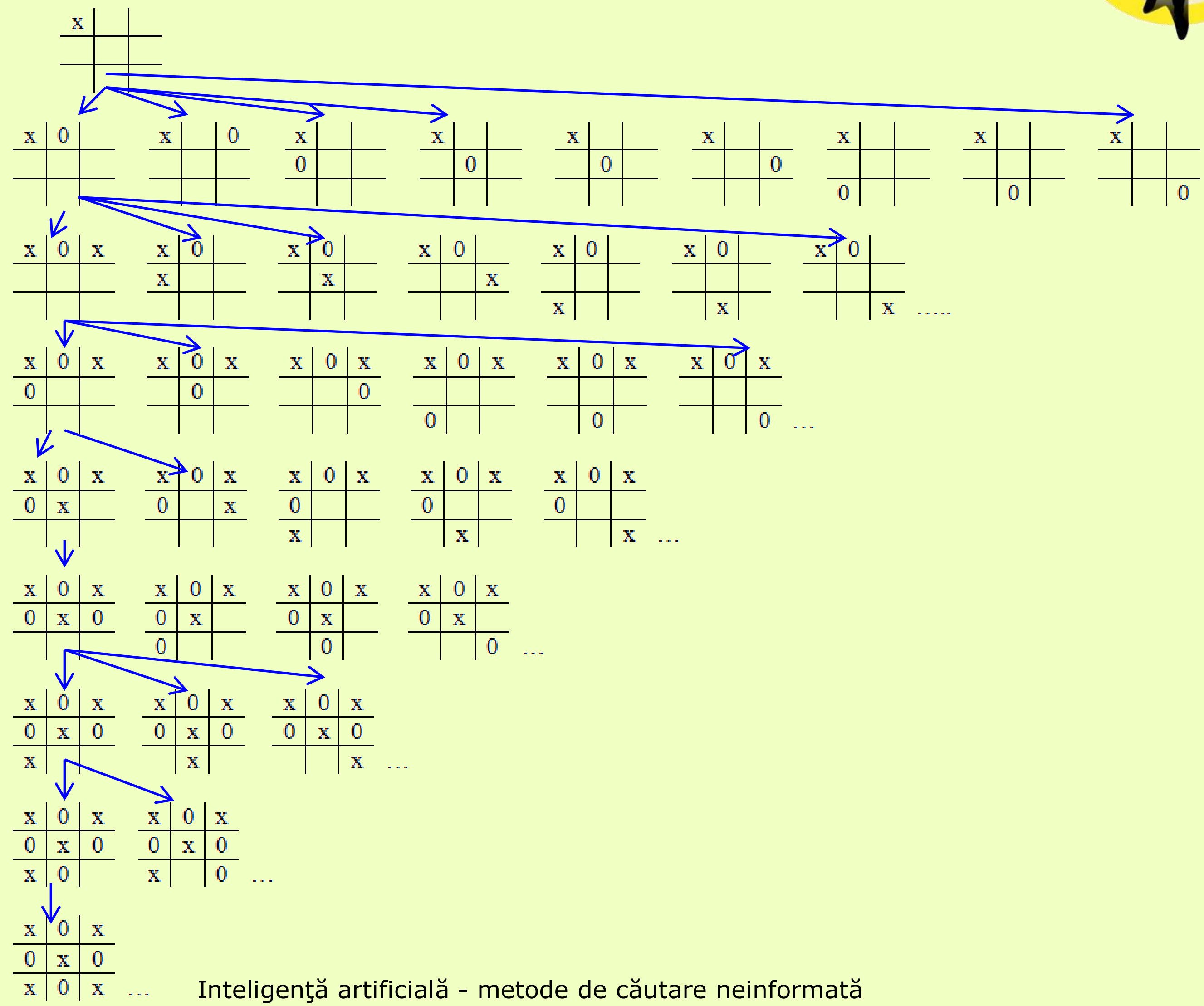
$$T(n) = \sum_{i=0}^{d_{\max}} (i+1) b^{d_{\max}-1} \Rightarrow O(b^{d_{\max}})$$

  - Complexitate spaţială
    - $S(n) = b * d_{max}$
  - Completitudine
    - Da
  - Optimalitate
    - Da

- **Avantaje**
  - Necesită memorie liniară
  - Asigură atingerea nodului ţintă urmând un drum de lungime minimă
  - Mai rapidă decât BFS şi DFS
- **Dezavantaje**
  - Necesită cunoaşterea "adâncimii" soluţiei

- **Aplicaţii**
  - Jocul Tic tac toe

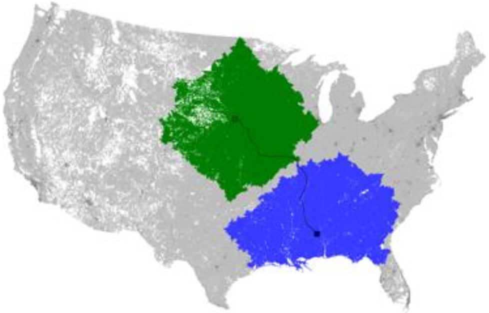
SCnI în structuri arborescente – căutare în adâncime iterativă (iterative deepening depth search – IDDS)



SCnI în structuri arborescente
căutare bidirecţională (bi-directional search – BDS)

- **Aspecte teoretice**
  - 2 căutări simultane
    - Înainte (*forward*): de la rădăcină spre frunze
    - Înapoi (*backward*): de la frunze spre rădăcină

    care se opresc atunci când ajung la un nod comun
  - într-o direcţie pot fi folosite oricare dintre streategiile de căutare anterioare
  - necesită
    - stabilirea succesorilor, respectiv a predecesorilor unui nod
    - stabilirea locului de întâlnire
- **Exemplu**



- **Algoritm**
  - În funcţie de strategia de căutare folosită

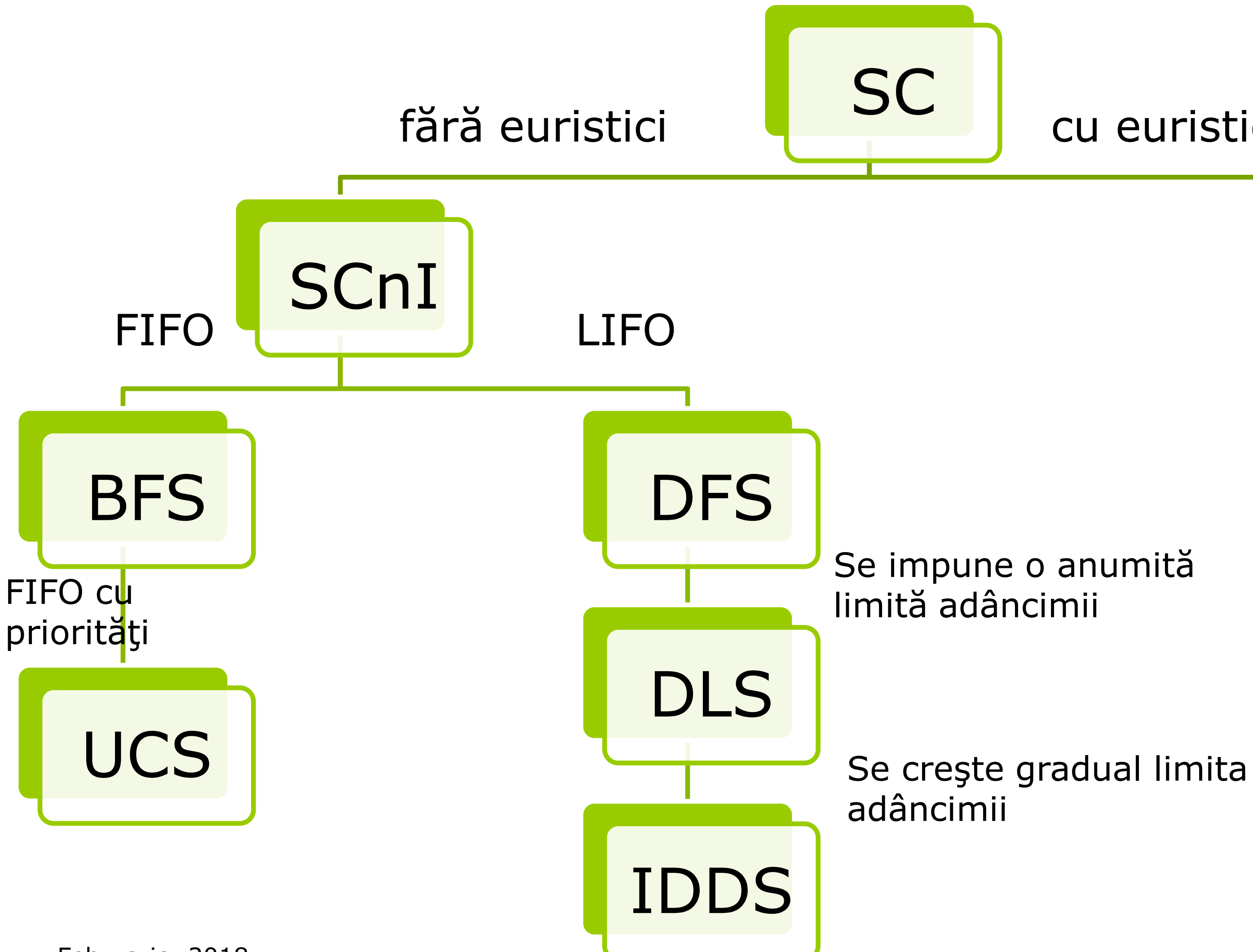
SCnI în structuri arborescente căutare bidirecţională (bi-directional search – BDS)

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

- **Avantaje**
  - Complexitate spaţială şi temporală redusă
- **Dezavantaje**
  - Dificultăţi în formularea problemei astfel încât fiecare stare să poată fi inversată
    - Parcurgere dinspre cap spre coadă
    - Parcurgere dinspre coadă spre cap
  - Implementare dificilă
  - Trebuie determinaţi succesorii şi predecesorii tuturor stărilor
  - Trebuie cunoscută starea finală (obiectiv)

- **Aplicaţii**
  - Problema partiţionării
  - Cel mai scurt drum

SCnI în structuri arborescente



SCnI în structuri arborescente

Compararea performanţelor

Metoda de căutare	Complexitate temporală	Complexitate spaţială	Completitudine	Optimalitate
BFS	$O(b^d)$	$O(b^d)$	Da	Da
UCS	$O(b^d)$	$O(b^d)$	Da	Da
DFS	$O(b^{dmax})$	$O(b*d_{max})$	Nu	Nu
DLS	$O(b^{dlim})$	$O(b*d_{lim})$	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

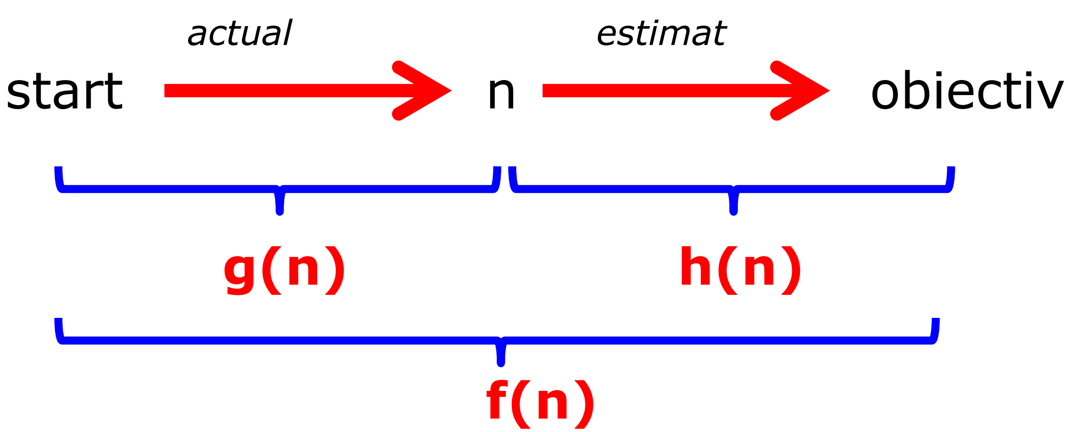
- Strategii de căutare (SC)
  - Topologie
    - În funcţie de informaţia disponibilă
      - SC ne-informate (oarbe)
      - SC informate (euristice)

Strategii de căutare informate (SCI)

- Caracteristici
  - se bazează pe informaţii specifice problemei încercând să restrângă căutarea prin alegerea inteligentă a nodurilor care vor fi explorate
  - ordonarea nodurilor se face cu ajutorul unei funcţii (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

- Noţiuni necesare
  - *f(n)* – funcţie de evaluare pentru estimarea costului soluţiei prin nodul (starea) *n*
  - *h(n)* – funcţie euristică pentru estimarea costului drumului de la nodul *n* la nodul obiectiv
  - *g(n)* – funcţie 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 funcţiei de evaluare *f*
  - Nodurile de expandant sunt reţinute 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 funcţia de evaluare
    - Căutare de cost uniform (SCnI)
      - *f* = costul drumului
    - În SCI se folosesc funcţii 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 soluţia cu costul cel mai mic

- **Exemplu**
  - Detalii în slide-urile următoare

SCI – Best first search

- **Algoritm**

```
bool BestFS(elem, list){
     found = false;
     visited = Φ;
     toVisit = {start};   //FIFO sorted list (priority queue)
     while((toVisit != Φ) && (!found)){
          if (toVisit == Φ)
            return false
          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; //adding a node into the FIFO list based on its
                                   // evaluation (best one in the front of list)

     }  //while
     return found;
}
```

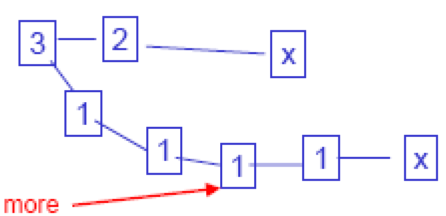
SCI – best first search

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

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

- **Dezavantaje**
  - Necesită evaluarea stărilor → efort computaţional, dar nu numai
  - Anumite path-uri pot "arăta" ca fiind bune conform funcţiei euristice

- **Aplicaţii**
  - *Web crawler (automatic indexer)*
  - Jocuri



SCI – Funcții euristice

- Etimologie: *heuriskein (gr)*
  - *a găsi, a descoperi*
  - *studiul metodelor şi regulilor de descoperire şi invenţie*

- Utilitate
  - Evaluarea potenţialului unei stări din spaţiul de căutare
  - Estimarea costului drumului (în arborele de căutare) din starea curentă până în starea finală (cât de aproape de ţintă a ajuns căutarea)

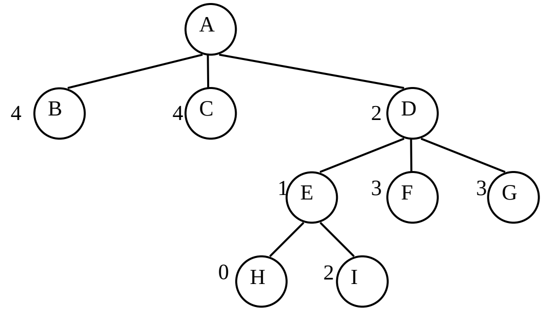
- Caracteristici
  - Depind de problema care trebuie rezolvată
  - Pentru probleme diferite trebuie proiectate sau învăţate diferite euristici
  - Se evaluează o anumită stare (nu operatorii care transformă o stare în altă stare)
  - Funcţii pozitive pentru orice stare *n*
    - $h(n) \geq 0$ pentru 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 – Funcţii euristice

- Exemple
  - Problema misionarilor şi canibalilor
    - *h(n)* – nr. persoanelor aflate pe malul iniţial
  - 8-puzzle
    - *h(n)* – nr pieselor care nu se află la locul lor
    - *h(n)* – suma distanţelor Manhattan (la care se află fiecare piesă de poziţia 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**
  - Funcţia de evalaure ***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**



Vizitate deja	De vizitat
Φ	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	Φ

```
bool BestFS(elem, list){
        found = false;
        visited = Φ;
        toVisit = {start};           //FIFO sorted list (priority queue
        while((toVisit != Φ) && (!found)){
                if (toVisit == Φ)
                    return false
                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; //adding a node onto the FIFO list based on its evaluation h(n)
                                         //(best one in the front of list)

        }  //while
        return found;
}
```

SCI - Greedy

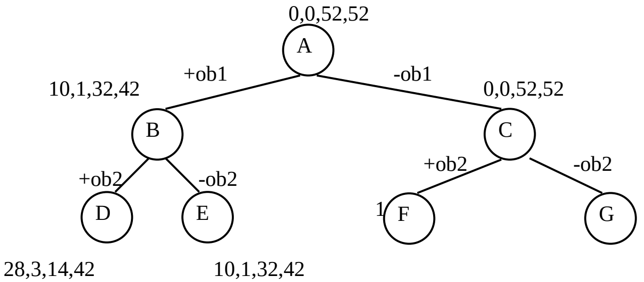
- **Analiza căutării:**
  - Complexitate temporală → DFS
    - *b* – factor de ramnificare (nr de noduri fii ale unui nod)
    - d^{max} - lungimea (adâncimea) maximă a unui arbore explorat
    - $T(n) = 1 + b + b^2 + \ldots + b^{dmax} \Rightarrow O(b^{dmax})$
  - Complexitate spaţială → BFS
    - *d*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

- **Avantaje**
  - Găsirea rapidă a unei soluţii (dar nu neapărat soluţia optimă), mai ales pentru probleme mici
- **Dezavantaje**
  - Suma alegerilor optime de la fiecare pas nu reprezintă alegerea globală optimă
    - Ex. Problema comisului voiajor

- **Aplicaţii**
  - Probleme de planificare
  - Probleme de sume parţiale
    - 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 şi completitudine
      - utilizarea unei cozi ordonate
    - căutării Greedy
      - viteza mare
      - ordonare pe baza unei funcţii de evaluare
  - Funcţia de evalaure ***f(n)***
    - estimarea costului celui mai bun drum care trece prin nodul n
    - ***f(n) = g(n) + h(n)***
    - ***g(n)*** – funcţie folosită pentru stabilirea costului drumului de la starea iniţială până la starea curentă *n*
    - ***h(n)*** – funcţie 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 $(o_1, o_2, \ldots, o_n)$ fiecare având o greutate w_i şi aducând un profit p_i, $i=1,2,\ldots,n$
    - Soluţia: pentru un rucsac cu W = 5 ➔ alegerea obiectelor o_1 şi o_3
  - $g(n) = \Sigma p_i$, pentru acele obiecte o_i care au fost selectate
  - $h(n) = \Sigma p_j$, pentru acele obiecte care nu au fost selectate şi $\Sigma w_j <= W - \Sigma w_i$
  - Fiecare nod din graf este un tuplu: *(p, w, p*, f)*, unde:
    - *p* – profitul adus de obiectele deja selectate (funcţia *g(n)*)
    - *w* – greutatea obiectelor selectate
    - *p** - profitul maxim care se poate obţine pornind din starea curentă şi ţinând cont de locul disponibil în rucsac (funcţia *h(n)*)

SCI – A*

- **Algoritm**

```
bool BestFS(elem, list){
     found = false;
     visited = Φ;
     toVisit = {start};  //FIFO sorted list (priority queue
     while((toVisit != Φ) && (!found)){
          if (toVisit == Φ)
            return false
          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; //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
    - $T(n) = 1 + b + b^2 + \ldots + b^{dmax} \Rightarrow O(b^{dmax})$
  - Complexitate spaţială
    - *d* - lungimea (adâncimea) soluţiei
    - $T(n) = 1 + b + b^2 + \ldots + b^d \Rightarrow O(b^d)$
  - Completitudine
    - Da
  - Optimalitate
    - Da

- **Avantaje**
  - Algoritmul care expandează cele mai puţine noduri din arborele de căutare
- **Dezavantaje**
  - Utilizarea unei cantităţi mari de memorie

- **Aplicaţii**
  - Probleme de planificare
  - Probleme de sume parţiale
    - 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/SMA*

Rezolvarea problemelor prin căutare

- Tipologia strategiilor de căutare
  - În funcţie de modul de **generare** a soluţiei
    - Căutare **constructivă**
      - Construirea progresivă a soluţiei
      - Ex. TSP
    - Căutare **perturbativă**
      - O soluţie candidat este modificată în vederea obţinerii unei noi soluţii candidat
      - Ex. SAT - Propositional Satisfiability Problem
  - În funcţie de modul de **traversare** a spaţiului de căutare
    - Căutare **sistematică**
      - Traversarea completă a spaţiului
        - Ideintificarea soluţiei dacă ea există ➔ algoritmi compleţi
    - Căutare **locală**
      - Traversarea spaţiului de căutare dintr-un punct în alt punct vecin ➔ algoritmi incompleţi
      - O stare a spaţiului poate fi vizitată de mai multe ori
  - În funcţie de elementele de **certitudine** ale căutării
    - Căutare **deterministă**
      - Algoritmi de identificare exactă a soluţiei
    - Căutare **stocastică**
      - Algoritmi de aproximare a soluţiei
  - În funcţie de stilul de **explorare** a spaţiului de căutare
    - Căutare **secvenţială**
    - Căutare **paralelă**

Rezolvarea problemelor prin căutare

Poate consta în:

- Construirea progresivă a soluţiei
- Identificarea soluţiei potenţiale optime
  - Componentele problemei
    - Condiţii (constrângeri) pe care trebuie să le satisfacă (parţial sau total) soluţia
    - Funcţie de evaluare a unei soluţii potenţiale → identificareaa optimului
  - Spaţiul de căutare
    - mulţimea tuturor soluţiilor potenţiale complete
    - Stare = o soluţie completă
    - Stare finală (scop) → soluţia optimă
  - Exemple
    - Problema celor 8 regine,
      - Stările posibile: configuraţii ale tablei de sah cu câte 8 regine
      - Operatori: modificarea coloanei în care a fost plasată una din regine
      - Scopul căutării: configuraţia în care nu existe atacuri între regine
      - Funcţia de evaluare: numărul de atacuri
    - probleme de planificare,
    - proiectarea circuitelor digitale, etc

Rezolvarea problemelor prin căutare

Poate consta în:

- Construirea progresivă a soluţiei
- Identificarea soluţiei potenţiale optime
  - Algoritmi
    - Algoritmii discutaţi până acum explorau în mod **sistematic** spaţiul de căutare
      - De ex. A* ➔ 10^{100} stări ≈ 500 variabile binare
    - Problemele reale pot avea 10 000 – 100 000 variabile ➔ nevoia unei alte categorii de algoritmi care explorează **local** spaţiul de căutare (algoritmi iterativi)
    - Ideea de bază:
      - se începe cu o stare care nu respectă anumite constrângeri pentru a fi soluţie optimă şi
      - se efectuează modificări pentru a elimina aceste violări (se deplasează căutarea într-o soluţie vecină cu soluţia curentă) astfel încât căutarea să se îndrepte spre soluţia optimă
    - Iterativi → se memorează o singură stare şi se încearcă îmbunătăţirea ei
      - versiunea inteligentă a algoritmului de forţă brută
    - Istoricul căutării nu este reţinut

```
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 soluţiei
- Identificarea soluţiei potenţiale optime
  - Avantaje
    - Simplu de implementat
    - Necesită puţină memorie
    - Poate găsi soluţii rezonabile în spaţii de căutare (continue) foarte mari pentru care alţi algoritmi sistematici nu pot fi aplicaţi
  - E utilă atunci când:
    - Se pot genera soluţii complete rezonabile
    - Se poate alege un bun punct de start
    - Există operatori pentru modificarea unei soluţii complete
    - Există o măsură pentru a aprecia progresul (avansarea căutării)
    - Există un mod de a evalua soluţia completă (în termeni de constrângeri violate)

Strategii de căutare locală

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

Strategii de căutare locală

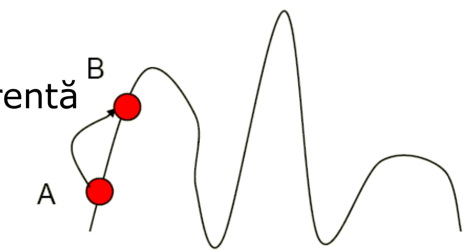
- Căutare locală simplă
  - elemente de interes special:
    - Reprezentarea soluţiei
    - Funcţia de evaluare a unei potenţiale soluţii
    - Vecinătatea unei soluţii
      - Cum se defineşte/generează o soluţie vecină
      - Cum are loc căutarea soluţiilor vecine
        - Aleator
        - Sistematic

dependente de problemă

    - Criteriul de acceptare a unei noi soluţii
      - Primul vecin mai bun decât soluţia curentă
      - Cel mai bun vecin al soluţiei curente mai bun decât soluţia curentă
      - Cel mai bun vecin al soluţiei curente mai slab decât soluţia curentă
      - Un vecin aleator

Strategii de căutare locală – Hill climbing (HC)

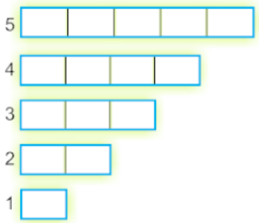
- **Aspecte teoretice**
  - Urcarea unui munte în condiţii de ceaţă şi amnezie a excursionistului :D
  - Mişcarea continuă spre valori mai bune (mai mari ➔ urcuşul pe munte)
  - Căutarea avansează în direcţia îmbunătăţirii valorii stărilor succesor până când se atinge un optim
  - Criteriul de acceptare a unei noi soluţii
    - cel mai bun vecin al soluţiei curente mai bun decât soluţia curentă

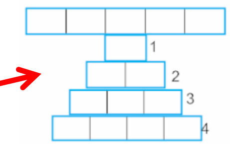


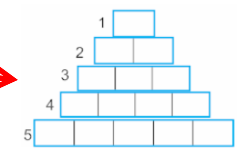
  - Îmbunătăţire prin
    - Maximizarea calităţii unei stări ➔ *steepest ascent HC*
    - Minimizarea costului unei stări ➔ *gradient descent HC*
  - HC $\neq$ *steepest ascent/gradient descent* (SA/GD)
    - HC optimizează f(x) cu $x \in R^n$ prin modificarea unui element al lui x
    - SA/GD optimizează f(x) cu $x \in R^n$ 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 aceeaşi lăţime, dar de lungimi diferite) aşezate unele peste altele formând un turn (stivă). Să se re-aşeze piesele astfel încât să se formeze un nou turn ştiind că la o mutare se poate mişca doar o piesă din vârful stivei, piesă care poate fi mutată pe una din cele 2 stive ajutătoare.



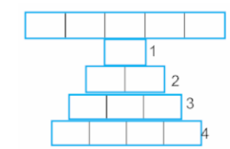




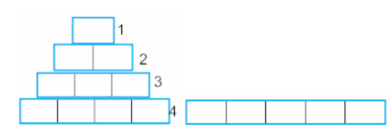
- Reprezentarea soluţiei
  - Stare *x* – vectori de *n* perechi de forma *(i,j)*, unde *i* reprezintă indexul piesei (*i=1,2,...,n*), iar *j* indexul stivei pe care se află piesa (*j=1,2,3*)
  - Starea iniţială – vectorul corespunzător turnului iniţial
  - Starea finală – vectorul corespunzător turnului final
- Funcţia de evaluare a unei stări
  - *f1* = numărul pieselor corect amplasate → maximizare
    - conform turnului final – *f1* = *n*
  - *f2* = numărul pieselor greşit amplasate → minimizare
    - conform turnului final – *f2* = *0*
  - *f* = *f1* – *f2* → maximizare
- Vecinătate
  - Mutări posibile
    - Mutarea unei piese *i* din vârful unei stive *j1* pe altă stivă *j2*
- Criteriul de acceptare a unei noi soluţii
  - Cel mai bun vecin al soluţiei curente mai bun decât soluţia curentă

Strategii de căutare locală – Hill climbing (HC)

- **Exemplu**
  - Iteraţia 1
    - Starea curentă x = starea iniţială:
      - $x = s_1 = ((5,1), (1,1), (2,1), (3,1), (4,1))$
      - Piesele 1, 2 şi 3 sunt corect aşezate
      - Piesele 4 şi 5 nu sunt corect aşezate
      - $f(s_1) = 3 - 2 = 1$



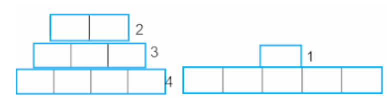
    - $x^* = x$
    - Vecinii stării curente x – un singur vecin → piesa 5 se mută pe stiva 2 ➔
      - $s_2 = ((1,1), (2,1), (3,1), (4,1), (5,2))$
      - $f(s_2) = 4-1=3 > f(x)$ ➔ $x = s_2$



Strategii de căutare locală – Hill climbing (HC)

- **Exemplu**
  - Iteraţia 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 ➔ s_3 = ((2,1), (3,1), (4,1), (1,2), (5,2)) ➔ $f(s_3)$ = 3-2=1 < f(x)

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



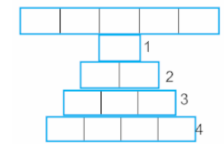
      - nu există vecin de-al lui x mai bun ca x ➔ stop
  - x* = x = ((1,1), (2,1), (3,1), (4,1), (5,2))
  - Dar x* este doar optim local, nu global

Strategii de căutare locală – Hill climbing (HC)

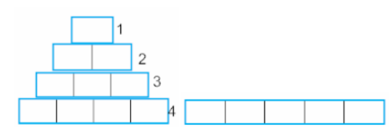
- **Exemplu**
  - Construirea unor turnuri din diferite forme geometrice
    - Funcţia de evaluare a unei stări
      - f1 = suma înălţimilor stivelor pe care sunt amplasate corect piese (conform turnului final – f1 = 10) → maximizare
      - f2 = suma înălţimilor stivelor pe care sunt amplasate incorect 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**



  - Iteraţia 1
    - Starea curentă x = starea iniţială s_1 = ((5,1), (1,1), (2,1), (3,1), (4,1))
      - Toate piesele nu sunt corect aşezate ➔ f1 = 0, f2 = 3+2 + 1 + 0 + 4 = 10
      - $f(s_1) = 0 - 10 = -10$
    - x* = x
    - Vecinii stării curente x– un singur vecin → piesa 5 se mută pe stiva 2 ➔ s_2 = ((1,1), (2,1), (3,1), (4,1), (5,2))

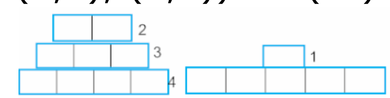


      - $f(s_2) = 0 - (3+2+1+0) = -6 > f(x)$ ➔ $x = s_2$

Strategii de căutare locală – Hill climbing (HC)

- **Exemplu**
  - Iteraţia 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 ➔ s3 = ((2,1), (3,1), (4,1), (1,2), (5,2)) ➔ f(s3) = 0 – (0+2+3+0)=-5 > f(x)

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



      - cel mai bun vecin al lui x este s4 ➔ x = s4
  - Iteraţia 3
    - ...
  - Se evită astfel blocarea în optimele locale

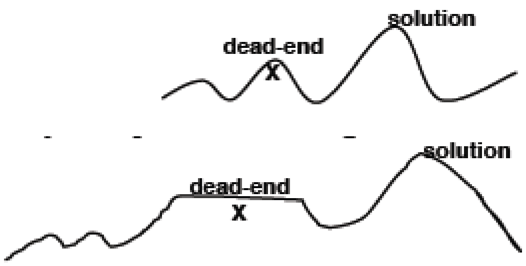
Strategii de căutare locală – Hill climbing (HC)

- **Algoritm**

```
Bool HC(S){
    x = s1    //starea iniţială
    x*=x      // cea mai bună soluţie găsită
   (până la un moment dat)
    k = 0     // numarul de iteraţii
    while (not termiantion 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)

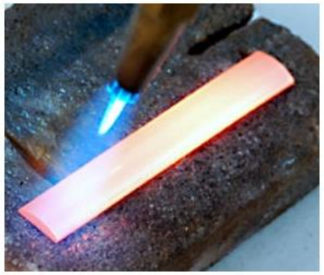
- **Analiza căutării**
  - Convergenţa spre optimul local
- **Avantaje**
  - Simplu de implementat → se poate folosi usor pentru a aproxima soluţia unei probleme când soluţia exactă este dificil sau imposibil de găsit
    - Ex. TSP cu forate multe oraşe
  - Nu necesită memorie (nu se revine în starea precedentă)
- **Dezavantaje**
  - Funcţia de evaluare (eurisitcă) poate fi dificil de estimat
  - Dacă se execută foarte multe mutări algoritmul devine ineficient
  - Dacă se execută prea puţine mutări algoritmul se poate bloca
    - Într-un optim local (nu mai poate "coborî" din vârf)
    - Pe un platou – zonă din spaţiul de căutare în care funcţia de evaluare este constantă
    - Pe o creastă – saltul cu mai mulţi paşi ar putea ajuta căutarea
- **Aplicaţii**
  - Problema canibalilor
  - 8-puzzle, 15-puzzle
  - TSP
  - Problema damelor



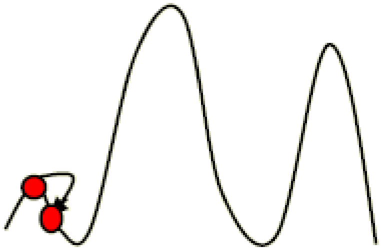
Strategii de căutare locală – Hill climbing (HC)

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

Strategii de căutare locală – Simulated Annealing

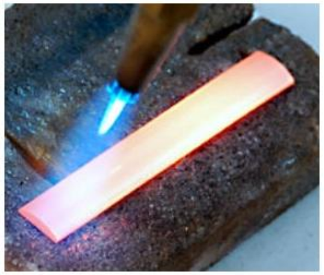


- **Aspecte teoretice**



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

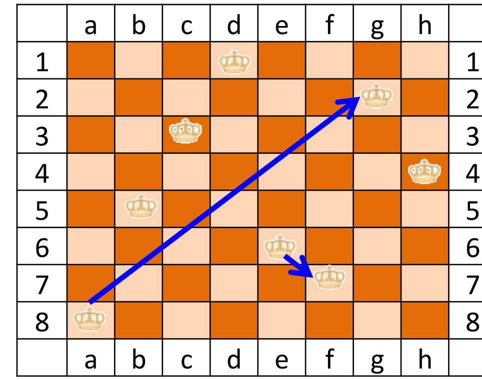
Strategii de căutare locală – Simulated Annealing



- **Exemplu** – Problema celor 8 regine
  - Enunţ
    - Să se amplaseze pe o tablă de şah 8 regine astfel încât ele să nu se atace reciproc
  - Reprezentarea soluţiei
    - Stare *x* – permutare de n elemente $x = (x_1,x_2,..,x_n)$, unde xi – linia pe care este plasată regina de pe coloana j
      - Nu există atacuri pe verticală sau pe orizontală
      - Pot exista atacuri pe diagonală
    - Starea iniţială – o permutare oarecare
    - Starea finală – o permutare fără atacuri de nici un fel
  - Funcţia 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 soluţii
    - Un vecin oarecare al soluţiei curente
      - mai bun decât soluţia curentă
      - mai slab decât soluţia curentă – cu o anumită probabilitate $P(\Delta E) = e^{-\frac{\Delta E}{T}}$ unde:
        - ΔE – diferenţa de energie (evaluare) între cele 2 stări (vecină şi curentă)
        - T – temperatura, T(k) = 100/k, unde k este nr iteraţiei

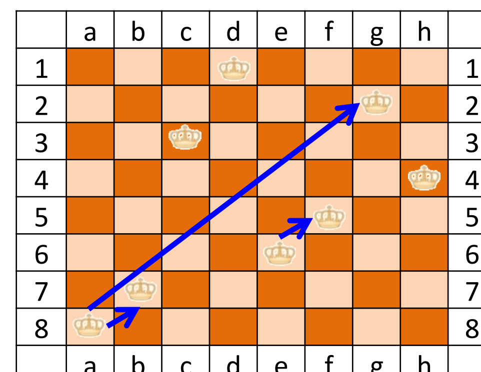
Strategii de căutare locală – Simulated Annealing

- **Exemplu** – Problema celor 8 regine
  - Iteraţia 1 (k = 1)
    - Starea curentă x = starea iniţială $s_1 = (8,5,3,1,6,7,2,4)$
      - $f(s_1) = 1+1 = 2$

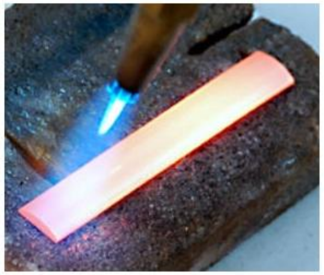


    - $x^* = x$
    - $T = 100/1 = 100$

    - Un vecin al stării curente x → regina de pe linia 5 se interschimbă cu regina de pe linia 7 ➔ $s_2 = (8,7,3,1,6,5,2,4)$
      - $f(s_2) = 1+1+1=3 > f(x)$
      - $\Delta E = f(s_2) - f(s_1) = 1$
      - $P(\Delta E)=e^{-1/100}$
      - $r = random(0,1)$
      - Dacă $r < P(\Delta E)$ ➔ $x = s_2$



Strategii de căutare locală – Simulated Annealing

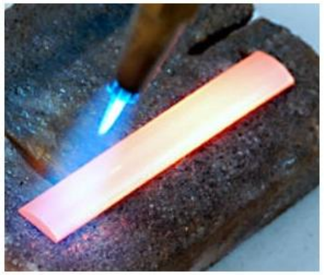


- **Algoritm**

```
bool SA(S){
        x = s1            //starea iniţială
        x*=x  // cea mai bună soluţie găsită (până la un moment dat)
        k = 0 // numarul de iteraţii
        while (not termiantion 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(ΔE) then x = s
        }  //while
        x* = x
        return x*;
}
```

- Criterii de oprire
  - S-a ajuns la soluţie
  - S-a parcurs un anumit număr de iterţii
  - S-a ajuns la temepratura 0 (îngheţ)
- Cum se alege o probabilitate mică?
  - p = 0.1
  - p scade de-a lungul iteraţiilor
  - p scade de-a lungul iteraţiilor şi pe măsură ce "răul" |f(s) – f(x)| creşte
    - p = exp(- |f(s) – f(x)|/T)
    - Unde T – temepratura (care creşte)
      - Pentru o T mare se admite aproape orice vecin v
      - Petnru 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

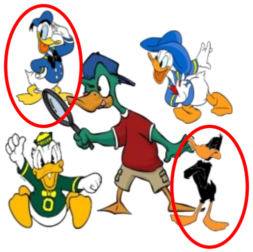


- **Analiza căutării**
  - Convergenţa (complet, optimal) lentă spre optimul global

- **Avantaje**
  - Algoritm fundamentat statistic ➔ garantează găsirea soluţiei optime, dar necesită multe iteraţii
  - Uşor de implementat
  - În general găseşte o soluţie relativ bună (optim global)
  - Poate rezolva probleme complexe (cu zgomot şi multe constrângeri)
- **Dezavantaje**
  - Algoritm încet – convergenţa la soluţie durează foarte mult timp
    - Compromis între calitatea soluţiei şi timpul necesar calculării ei
  - Depinde de anumiţi parametri (temperatura) care trebuie reglaţi
  - Nu se ştie dacă soluţia oferită este optimă (local sau gloabal)
  - Calitatea soluţiei găsite depinde de precizia variabilelor implicate în algoritm

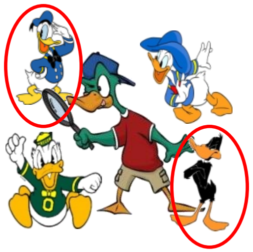
- **Aplicaţii**
  - Probleme de optimizare combinatorială → problema rucsacului
  - Probleme de proiectare → Proiectarea circuitelor digitale
  - Probleme de planificare → Planificarea producţiei, planificarea meciurilor în turniruirle de tenis US Open

Strategii de căutare locală – Căutare tabu



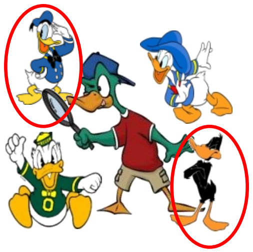
- **Aspecte teoretice**
  - "Tabu" → interdicţie socială severă cu privire la activităţile umane sacre şi interzise
  - Propusă în anii 1970 de către F. Glover
  - Ideea de bază
    - se începe cu o stare care nu respectă anumite constrângeri pentru a fi soluţie optimă şi
    - se efectuează modificări pentru a elimina aceste violări (se deplasează căutarea în cea mai bună soluţie vecină cu soluţia curentă) astfel încât căutarea să se îndrepte spre soluţia optimă
    - se memorează
      - Starea curentă
      - Stările vizitate până la momentul curent al căutării şi 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 soluţii
    - Cel mai bun vecin al soluţiei curente mai bun decât soluţia curentă şi nevizitat încă
  - 2 elemente importante
    - Mutări tabu (T) – determinate de un proces non-Markov care se foloseşte de informaţiile obţinute în timpul căutării de-a lungul ultimelor generaţii
    - Condiţii tabu – pot fi inegalităţi liniare sau legături logice exprimate în funcţie de soluţia curentă
      - Au rol în alegerea mutărilor tabu

Strategii de căutare locală – Căutare tabu



- **Exemplu**
  - Enunţ
    - Plata sumei *S* folosind cât mai multe dintre cele *n* monezi de valori v_i (din fiecare monedă există b_i bucăţi)
  - Reprezentarea soluţiei
    - Stare *x* – vector de *n* întregi $x = (x_1, x_2, ..., x_n)$ cu $xi \in \{0, 1, 2, ..., b_i\}$
    - Starea iniţială – aleator
  - Funcţia de evaluare a unei stări
    - f_1 = Diferenţa între S şi valaorea monezilor alese → minimă
      - Dacă valoarea monezilor depăşeşte *S* ➔ penalizare de 500 unităţi
    - f_2 = Numărul monezilor selectate → maxim
    - $f = f_1 - f_2$ → minimizare
  - Vecinătate
    - Mutări posibile
      - Includerea în sumă a monezii *i* în *j* exemplare ($plus_{i,j}$)
      - Excluderea din sumă a monezii *i* în *j* exemplare ($minus_{i,j}$)
    - Lista tabu reţine mutările efectuate într-o iteraţie
      - Mutare – moneda adăugată/eliminată din sumă

Strategii de căutare locală – Căutare tabu



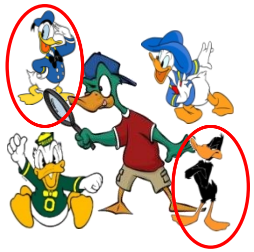
- **Exemplu**
  - S = 500, penalizare = 500, n = 7

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	∅	2 0 1 3 0 2 1	$plus_{4,3}$	321
			2 0 1 0 0 3 1	$plus_{6,1}$	348
			0 0 1 0 0 2 1	$minus_{1,2}$	406
2 0 1 3 0 2 1	321	$plus_{4,3}$	2 0 1 3 5 2 1	$plus_{5,5}$	316
			2 0 1 1 0 2 1	$minus_{4,2}$	363
			2 1 1 3 0 2 1	$plus_{2,1}$	270
2 1 1 3 0 2 1	270	$plus_{4,3}$ $plus_{2,1}$	...		

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

Strategii de căutare locală – Căutare tabu



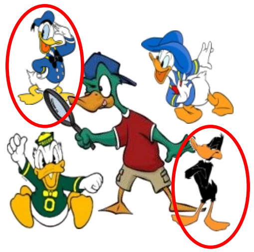
- **Exemplu**
  - S = 500, penalizare = 500, n = 7

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	∅	1 0 1 4 0 2 1	$\text{minus}_{1,1}$,$\text{plus}_{4,4}$	311
			2 0 4 0 1 2 1	$\text{plus}_{3,3}$,$\text{minus}_{5,1}$	235
			2 0 1 0 4 2 6	$\text{plus}_{5,4}$, $\text{plus}_{7,5}$	450
2 0 4 0 1 2 1	235	$\text{plus}_{3,3}$, $\text{minus}_{5,1}$	2 0 5 0 5 2 1	$\text{plus}_{3,1}$, $\text{plus}_{5,4}$	315
			5 0 4 0 4 2 1	$\text{plus}_{1,3}$, $\text{plus}_{5,3}$	399
			2 2 4 0 5 2 1	$\text{plus}_{2,2}$, $\text{plus}_{5,4}$	739
2 0 4 0 1 2 1	235	$\text{plus}_{3,3}$, $\text{minus}_{5,1}$	...		

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

Strategii de căutare locală – Căutare tabu

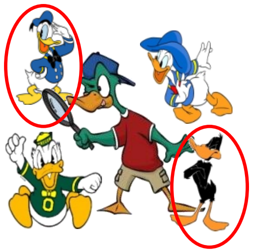


- **Algoritm**

```
bool TS(S){
    Select x∈S   //S - spaţiul de căutare
    x*=x         //cea mai bună soluţie (până la un mom. dat)
    k = 0        // numarul de iteraţii
    T = ∅        // listă de mutări tabu
    while (not termiantion 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*;
}
```

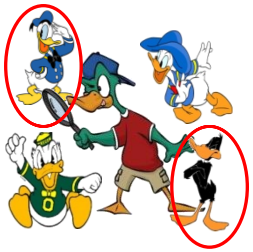
- Criterii de terminare
  - Număr fix de iteraţii
  - Număr de iteraţii fără îmbunătăţiri
  - Apropierea suficientă de soluţie (dacă aceasta este cunoscută)
  - Epuizarea elementelor nevizitate dintr-o vecinătate

Strategii de căutare locală – Căutare tabu



- **Analiza căutării**
  - Convergenţa rapidă spre optimul global
- **Avantaje**
  - Algoritm general şi simplu de implementat
  - Algoritm rapid (poate oferi soluţia optimă globală în scurt timp)
- **Dezavantaje**
  - Stabilirea stărilor vecine în spaţii continue
  - Număr mare de iteraţii
  - Nu se garantează atingerea optimului global

Strategii de căutare locală – Căutare tabu



- **Aplicaţii**
  - Determinarea structurii tridimensionale a proteinelor în secvenţe de aminoacizi (optimizarea unei funcţii de potenţial energetic cu multiple optime locale)
  - Optimizarea traficului în reţele de telecomunicaţii
  - Planificare în sisteme de producţie
  - Proiectarea reţelelor de telecomunicaţii optice
  - Ghidaj automat pentru vehicule
  - Probleme în grafuri (partiţionări)
  - Planificări în sistemele de audit
  - Planificări ale task-urilor paralele efectuate de procesor (multiprocesor)
  - Optimizarea structurii electromagnetice (imagistica rezonanţei magnetice medicale)
  - Probleme de asignare quadratică (proiectare VLSI)
  - Probleme de combinatorică (ricsac, plata sumei)
  - Problema tăierii unei bucăţi în mai multe părţi
  - Controlul structurilor spaţiale (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 iniţială la starea curentă – g(n) – şi a costului de la starea curentă la starea obiectiv – h(n)
    - Evitarea repetării stărilor
    - Fără supraestimarea lui h(n)
    - Timp şi spaţiu de căutare mare → în funcţie de euristica folosită
    - Complet
    - Optimal

Recapitulare

- SC locale
  - Algoritmi iterativi
    - Lucrează cu o soluţie potenţială → soluţia optimă
    - Se pot bloca în optime locale

	Alegerea stării următoare	Criteriul de acceptare	Convergenţa
HC	Cel mai bun vecin	Vecinul este mai bun decât strarea 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 strarea curentă	Optim global (rapidă)

Cursul următor

A. Scurtă introducere în Inteligenţa 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 învaţă singure
  - Arbori de decizie
  - Reţele neuronale artificiale
  - Maşini cu suport vectorial
  - Algoritmi evolutivi
- Sisteme bazate pe reguli
- Sisteme hibride

Cursul următor – Materiale de citit şi legături utile

- capitolul 14 din *C. Groşan, 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*

- Informaţiile prezentate au fost colectate din diferite surse bibliografice, precum şi din cursurile de inteligenţă artificială ţinute în anii anteriori de către:
  - Conf. Dr. Mihai Oltean – www.cs.ubbcluj.ro/~moltean
  - Lect. Dr. Crina Groşan - www.cs.ubbcluj.ro/~cgrosan
  - Prof. Dr. Horia F. Pop - www.cs.ubbcluj.ro/~hfpop