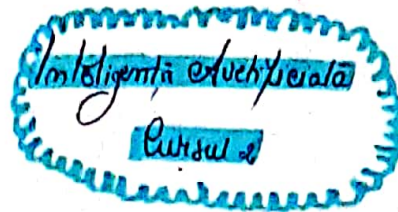


- concepte / formule
- metode / clasificare
- important pt. notare



Ne așteptăm de la un AI să se comporte în următorul fel: pentru o problemă definită, computerul o rezolvă și returnează un output. În funcție de tipul de rezultat, putem avea:

↙ explainable AI  
 ↘ unexplainable AI

Modele bazate pe stări: calculatorul știe starea inițială și ce <sup>în</sup> stare dorește să ajungă.

Rezolvarea unei probleme: pt. 2 numere A și B, trebuie să determinăm care instanță este mai mare. Totuși, calculatorul nu va mai putea compara A și B de la o anumită dimensiune  $\Rightarrow$  trebuie să ne asigurăm că am rezolvat problema pt. TOATE instanțele problemei.

Tipuri de probleme

- P = set de probleme care poate fi rezolvat în timp polinomial de o mașină Turing deterministă
- NP = nedeterministă. Soluția unei probleme NP este o P-problemă

probleme NP

- NP-complete = subset al problemelor NP care pot fi reducă la probleme NP-complete
- NP-hard = NP-hard.

$P = NP$  (poate fi orice alg. nedeterminist rezolvat și printr-un alg. determinist)?  
 ↳ dilema în informatică

ex. probleme NP-hard irrezolvabile: QSAT  
irrezolvabile: Turing Halting problem

Dacă rezolvăm o problemă NP, atunci rezolvăm orice problemă de tip NP pentru că le vom putea reduce la acea problemă.

ex. Pt. a rezolva problema stărilor Hamilton:

→ propunem un model bazat pe stări: {

- vom alege reprezentarea unei stări, a.î. să nu avem ambiguități, și să fie destul de expresivă pentru a include toate datele
- o stare trebuie să includă toate datele necesare pt. a putea căuta o soluție

→ O imagine nu este destul pt. o reprezentare a stării pt. că nu știm care a fost stăruțul inițial.

→ O altă variantă pt. reprezentare este oferirea unei liste cu ordinea în care au fost puse, după mărime: (3, 3, 3, 3, 1, 1, 2, 2)

→ Inițial, pusele sunt reprezentate astfel: (3, 1, 1, 1, 1, 1, 1, 1)

Pentru multe probleme, este greu de găsit o reprezentare bună a stării (ex. vremea pe 8 opt. viitoare).

Totuși, nu putem include toate informațiile pt. o reprezentare pt. că asta ar presupune mult spațiu pt. alocare.

Aici nu dorim { ambiguitate  
puțin multe informații  
informații neexplicite și nerelevante

Stări speciale

starea inițială:  $(0, 1, 1, 1, 1, 1, 1, 1, 1)$  (trebuie să avem cel puțin o stare inițială)

State Initialize (int m, int m) {  
return m(1, ..., 1);  
}

starea finală:  $(m, m, m, m, m, m, m, m, m)$  (cel puțin o stare finală)

Boolean IsFinal (State s) {  
if  $s = (k, k, \dots, k)$  then return true;  
else return false;  
}

Starea finală este mai greu de descris pt. că trebuie descrise toate variantele în care poate apărea problema la final.

Spațiul/problemei include toate stările posibile.

Alte întrebări

- câte stări sunt în spațiul problemei?
- ce mutații putem face în acel spațiu?
- cât de dificilă e problema?

Simon's Ant → avem o furmică care vrea să ajungă dintr-o stare inițială într-o stare finală. Trebuie să determinăm tranzitiile, înțelegând furmici poate întâlni obstacole pe drum și nu suntem siguri că poate merge în linie dreaptă.

Complexitatea unei soluții este dată de problemă, nu de editorul modului de rezolvare.

Tranzitiile pt. turmul Harmoni: pt. a schimba starea curentă, putem doar să mutăm o piesă pe alt turm.

$(m, t_{11}, t_{12}, \dots, t_{1m}) \rightarrow (m, t_{21}, t_{22}, \dots, t_{2m}), t_{i1} = t_{i2} \forall 1 \leq i \leq m, \text{ exceptând } i=k$  (3)

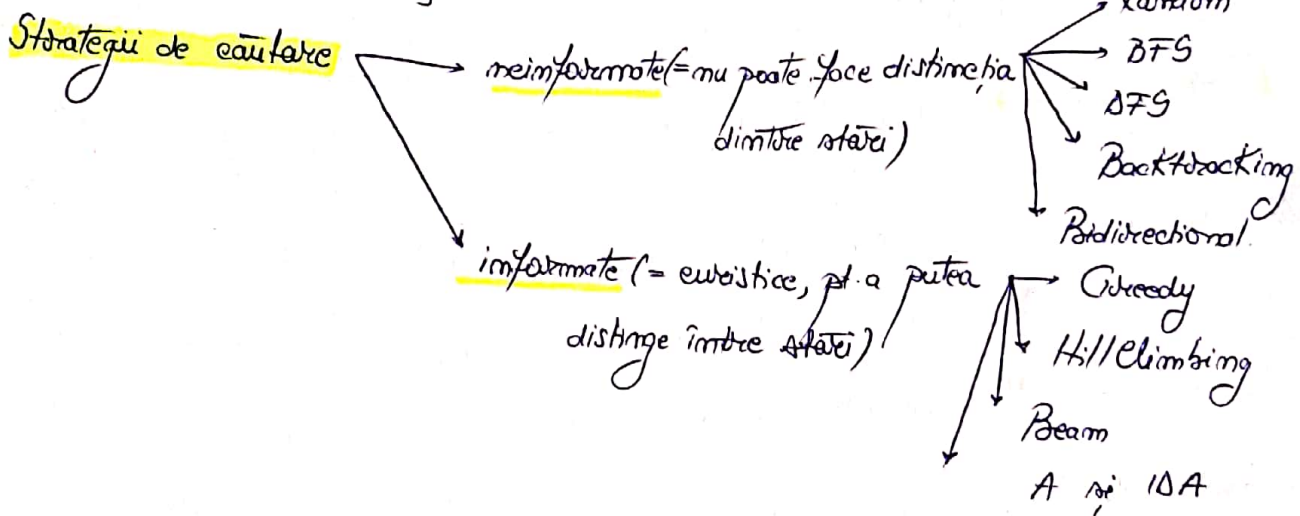


Validarea tranzițiilor (se reprezintă sub forma unei funcții Boolean Valide (State s, piece, tower))

1. mică piesă mai mică nu se află sub piesa k:  $t_{a1} \neq t_{a2}, \forall i \leq d \leq k$

Good practice → implementarea tranzițiilor și a validării se face în funcții diferite

Strategia de căutare: void strategy (state s) {  
 while (!isFinal(s)) {  
 choose piece, tower;  
 if (!valid(s, piece, tower)) {  
 s = transition(s, piece, tower);  
 }  
 }  
}



DFS (Depth-First Search) și IDFS (Iterative Deepening Search)

DFS → vizitează un vecin al stării curente până când se ajunge la starea finală; se întoarce la starea precedentă care nu a fost vizitată până când nu mai este niciun vecin nevizitat.

Alg. este costisitor pt. că trebuie să țină fiecare stare vizitată, iar unele stări să fie vizitate de mai multe ori. Există riscul ca alg. să nu se termine din cauza instrucțiunilor repetitive.

**IDS** - explorează doar până la o anumită distanță față de starea inițială  $\Rightarrow$  nu există drumuri infinite

**BFS** (Breadth-First Search / **Costul Uniform**)

**BFS** vizitează stările în ordinea distanței (adică a numărului de tranziții) față de starea inițială.

se oprește atunci când s-a găsit starea finală sau când nu mai sunt vecini

La fel ca DFS trebuie să viziteze fiecare stare  $\Rightarrow$  foarte costisitor.

poate vizita aceeași stare de mai multe ori.

Totuși, BFS găsește cel mai scurt drum

**Costul Uniform** = dacă costurile au costuri diferite, atunci le va vizita mai întâi pe cele de cost mai mic

**Random** și **DFS** sunt la fel dacă alg. randomează vecinii vizitați.

**Backtracking** și **DFS** nu sunt la fel!

BKT creează o ordine pt. vecinii stărilor.

BKT nu trebuie să viziteze stările vizitate

BKT poate evita buclele față de starea inițială.

**Căutarea bidirecțională**

începe să exploreze de la starea inițială și cea finală în mod

simultan

trebuie să viziteze fiecare stare generată, dar nu sunt la fel de multe ca la BFS sau DFS

găsește soluția optimă

**Comparativ**

Metoda	BFS	DFS	IDS	Bidirecțional	BKT	Random
Complexitate	$N^0$	$N^N$	$N^0$	$N^0$	$NA$	$NA$
Optim	Aa	Nu	Aa	Aa	Nu	Nu
Concluzie	Aa	Nu	Aa	Aa	Aa	Nu

$N$  - numărul mediu de Afări observate

$O$  - lungimea soluției optime

$A$  - lungimea soluției medii.