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

5 Oct 2022

Inteligență artificială
Cursul 2

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 stare dorește să ajungă în

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 => 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 reduse la probleme NP-complete
- **NP-hard** = —"— NP-hard.

P = NP (poate fi orice alg. nedeterminist rezolvat și pentru-un alg. determinist)?
└ dilemă în informatică

ex. probleme NP-hard → rezolvabile: QSAT
nerezolvabile: Turing Halting problem

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

ex. Pt. a rezolva problema turnurilor Hanoi:

→ pentru 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 calcula o soluție

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

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

→ Inițial, piesele 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 următoarele).

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

Deci nu dorim
- ambiguitate
- prea multe informații
- informații neexplicite și irelevante

Stări speciale → **starea inițială**: (3, 1, 1, 1, 1, 1, 1, 1, 1) (trebuie să avem cel puțin o stare inițială)

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

starea finală: (n, n, n, n, n, n, n, n, n) (cel puțin o stare finală)

```
Boolean IsFinal(State s) {
    if s = (k, k, ..., k) then return true;   // m+1
    else return false;
}
```

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

Spațiul problemei include toate stările posibile.

Alte întrebări → câte stări sunt în spațiul problemei?
→ ce mutări putem face în acel spațiu?
→ cât de dificilă e problema?

Simon's Ant → avem o furnică care vrea să ajungă dintr-o stare inițială într-o stare finală. Trebuie să determinăm **tranzițiile**, întrucât furnica poate întâlni obstacole pe drum și nu suntem siguri că poate merge în linie dreaptă.

Complexitatea unei soluții este dată de problemă, nu de ordinul de rezolvare.

Tranzițiile pt. turnul Hanoi: pt. a schimba starea curentă, putem doar să mutăm o piesă pe alt turn.

$(n, t_{11}, t_{12}, \ldots, t_{1m}) \to (n, t_{21}, t_{22}, \ldots, t_{2m})$, $t_{1i} = t_{2i}$ $\forall$ $1 \le i \le m$, excepând $i = k$.

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

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

2

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 (validate (s, piece, tower)) {

s = transition (s, piece, tower);

} } }

Strategii de căutare
- neinformate (= nu poate face distincția dintre stări)
  - Random
  - BFS
  - DFS
  - Backtracking
  - Bidirectional
- informate (= euristice, pt a putea distinge între stări)
  - Greedy
  - Hill Climbing
  - Beam
  - A și IDA

DFS (Depth-First Search) și IDS (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 => nu există drumuri infinite

BFS (Breadth-First Search) și Costul Uniform

BFS - vizitează stările în ordinea distanței (adică a numărului de tranziții) față de starea inițială.
se oprește atunci când a găsit starea finală sau când nu mai sunt vecini

La fel ca DFS - trebuie să reţină fiecare stare => poate explozie
- poate vizita aceeași stare de mai multe ori.

Totuși, BFS găsește cel mai scurt drum

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

Random și DFS sunt la fel dacă alg. random reține vecinii vizitați.

Backtracking și DFS nu sunt la fel!
- BKT evaluează o ordine pt. vecinii stărilor.
- BKT nu trebuie să reţină stările vizitate
- BKT poate evita bucle fără a reţine stările

Căutarea bidirectională
- începe să exploreze de la starea inițială și cea finală în mod simultan
- trebuie să reţină fiecare stare generată, dar nu sunt la fel de multe ca la BFS sau DFS
- găsește soluția optimă

Comparație.

Metoda	BFS	DFS	IDS	Bidirecțional	BKT	Random
Complexitate	N^0	N^A	N^0	N^0	N^A	N^A
Optim	Da	Nu	Da	Da	Nu	Nu
Concluzie	Da	Nu	Da	Da	Da	Nu

N - numărul mediu de stări accesibile

O - lungimea soluției optime

A - lungimea soluției medii.