Paradigme de programare

Laboratorul 11

Mihai Nan - (AI-MAS)

Laboratory of Artificial Intelligence and Multi-Agent Systems (AI-MAS)
Facultatea de Automatică și Calculatoare
Universitatea Politehnica din București

Anul universitar 2020-2021

Cuprins

Probleme de căutare în spațiul stărilor

Introducere

- În practică, există multe probleme care se pot rezolva aplicând o căutare în spațiul stărilor.

Formalism

- Pornim dintr-o **stare inițială** și încercăm să atingem o **stare scop**.
- **Secvența de acțiuni** care sunt aplicate din **starea inițială** pentru a ajunge în **starea scop** reprezintă **soluția** problemei de căutare în spațiul stărilor.

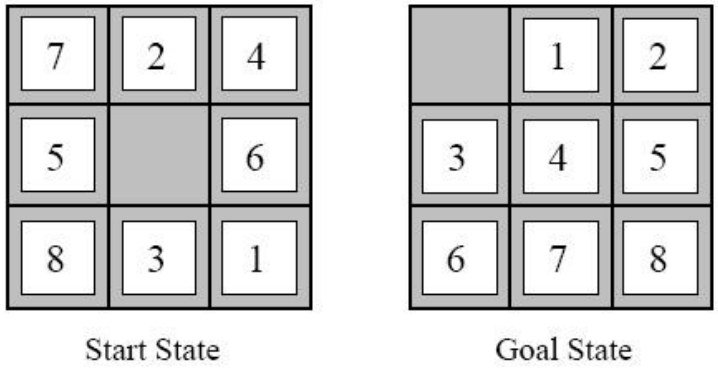
Inconvenient

- Pentru majoritatea problemelor, spațiul de căutare este unul foarte mare (multe stări posibile și multe acțiuni care se pot aplica în fiecare stare).

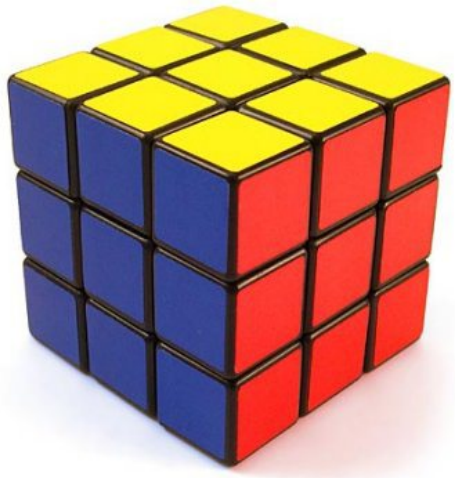
Probleme de căutare în spațiul stărilor

Exemple de probleme

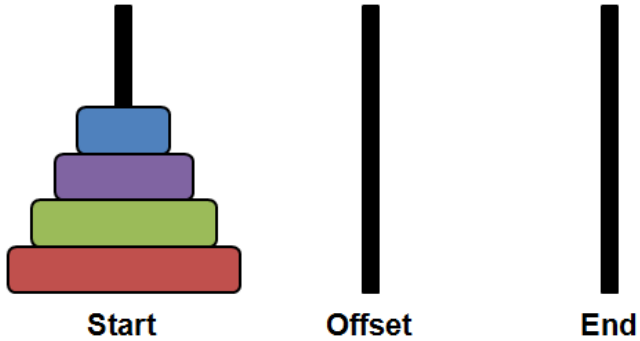
1. Problema 8-puzzle



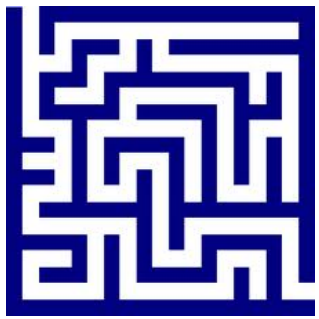
2. Problema cubului rubik



3. Problema turnurilor din Hanoi



4. Problema labirintului



Probleme de căutare în spațiul stărilor

Descrierea formală a problemei – I

- Fie S mulțimea stărilor posibile ale problemei (**spațiul stărilor**).
- Pentru a defini problema avem nevoie de o **stare inițială** (s_0), **tranzițiile** dintre stări ($succ$) și una sau mai multe **stări finale** (stări scop) ($goal$) $\Rightarrow problem = (s_0, succ, goal)$.

Problema de căutare

1. $s_0 \in S$ – starea inițială;
2. $succ : S \to \mathcal{P}(S)$ – funcția de tranziție;
3. $goal : S \to \{0, 1\}$ – o asertiune care verifică dacă o stare este sau nu stare scop.

- Funcția de tranziție poate să fie definită **explicit** sau într-un mod **implicit**.

Probleme de căutare în spațiul stărilor

Algoritmul general de căutare

```
def search(s0, succ, goal):
        open = [s0]
        while len(open) > 0:
                current = remove(open)
                if goal(current):
                        return current
                for next in expand(current, succ):
                        insert(next, open)
        raise Exception("Fail")
```

- `remove(open)` – extrage o stare din lista `open`;
- `expand(current, succ)` – expandează starea `current` folosind funcția de tranziție `succ`;
- `insert(next, open)` – inserează starea `next` în lista `open`.

- Strategii de căutare neinformate (căutare oarbă)
  1. Backtracking
  2. Breadth-first search (BFS)
- Strategii de căutare informate (căutare euristică)
  1. Algoritmul A*

Cuprins

Backtracking

- Atunci când calea către soluție admite un număr nedeterminat de stări intermediare nu este posibil să definim un template care descrie forma soluției problemei.
- Vom defini o căutare mai generală, după modelul următor:

```
solve(Solution):-
    initial_state(State),
    search([State], Solution).
```

- unde `search(+StăriVizitate,-Soluţie)` definește mecanismul general de căutare astfel:
  1. căutarea începe de la o stare inițială dată (predicatul `initial_state/1`)
  2. dintr-o stare curentă se generează stările următoare posibile (predicatul `next_state/2`)
  3. se testează că starea în care s-a trecut este nevizitată anterior (evitând astfel traseele ciclice)
  4. căutarea continuă din noua stare, până se întâlnește o stare finală (predicatul `final_state/1`).

Backtracking

- Verificăm dacă am ajuns într-o stare scop. Dacă da, oprim căutarea (folosim predicatul !) și inversăm soluția.

```
search([CurrentState|Other], Solution):-
    final_state(CurrentState), !,
    reverse([CurrentState|Other], Solution).
```

- Dacă nu am ajuns într-o stare scop:
  1. explorăm stările următoare posibile;
  2. verificăm dacă starea nu a fost deja vizitată;
  3. continuăm căutarea din noua stare.

```
search([CurrentState|Other], Solution):-
    next_state(CurrentState, NextState),
    \+ member(NextState, Other),
    search([NextState,CurrentState|Other], Solution).
```

Căutarea în lățime

- Căutarea în lățime este adecvată situațiilor în care se dorește drumul minim între o stare inițială și o stare finală.
- La o căutare în lățime, expandarea stărilor "vechi" are prioritate în fața expandării stărilor "noi" (folosim ca structură de date auxiliară pentru a reține nodurile intermediare o coadă).

```
do_bfs(Solution):-
    initial_node(StartNode),
    bfs([(StartNode,nil)], [], Discovered),
    extract_path(Discovered, Solution).
```

- `bfs(+CoadaStărilorNevizitate,+StăriVizitate,-Soluţie)` va defini mecanismul general de căutare în lățime astfel:
  1. căutarea începe de la o stare inițială dată care n-are predecesor în spațiul stărilor (StartNode cu părintele nil);
  2. se generează toate stările următoare posibile;
  3. se adaugă toate aceste stări la coada de stări încă nevizitate;
  4. căutarea continuă din starea aflată la începutul cozii, până se întâlnește o stare finală.

Cuprins

Problema țăranului, a lupului, a caprei și a verzei

- Un țăran, ducând la târg un lup, o capră și o varză ajunge în dreptul unui râu pe care trebuie să-l treacă. Cum va proceda el, știind că:
  - lupul mănâncă capra și capra mănâncă varza;
  - el nu poate să-i treacă pe toți o dată și nici câte doi.
- Vom reprezenta o stare astfel:
  `state(MalTaran, Lista-cu-cine-este-pe-malul-cu-taranul).`
- Baza de cunoștinte:

```
opus(est, vest).
opus(vest, est).
allTaran([capra, lup, varza]).
initial_state(taran, state(est, Cine)) :- allTaran(Cine).
final_state(taran, state(vest, Cine)) :- allTaran(All), sort(Cine, All).
```

- Configurații posibile:
  1. Pot rămâne pe același mal `lup` și `varza`;
  2. Poate rămâne pe un mal doar `lup` sau doar `capra` sau doar `varza`.
  3. Rămâne un mal fără nimic.
- Avem predicatul `safeTaran(+Lista-cu-cine-este-pe-malul-opus-taranului)`
- **Exemplu:** `safeTaran([lup]).` (poate rămâne pe malul opus țăranului doar lupul).