Prolog: Probleme de căutare în spaţiul stărilor

- Responsabil: Andrei Olaru [mailto:cs@andreiolaru.ro]

- Data publicării: 06.05.2019
- Data ultimei modificări: 06.05.2019

Obiective

Scopul acestui laborator este învăţarea unor tehnici de rezolvare a problemelor de căutare în spaţiul stărilor.

Aspectele urmărite sunt:

- metodele de reprezentare a datelor
- particularităţile diverselor tehnici de rezolvare
- eficienţa
- abstractizarea procesului de rezolvare

Căutare în spaţiul stărilor

Fie un sistem dinamic care se poate afla într-un număr finit de stări. Definim un graf orientat în care nodurile sunt stări, iar arcele reprezintă posibilitatea de a evolua direct dintr-o stare în alta. Graful conţine un set de stări iniţiale şi un set de stări finale (scopuri). Problema descrisă de sistem va fi rezolvată prin parcurgerea spaţiului stărilor până când este găsită o cale de la o stare iniţială la o stare finală, în cazul în care problema admite soluţie. Căutarea este un mecanism general, recomandat atunci când o metodă directă de rezolvare a problemei nu este cunoscută.

Exemple de strategii de căutare în spaţiul stărilor:

- generare şi testare
- backtracking
- căutare în adâncime/lăţime
- algoritmul A*

Generare şi testare

Fie problema colorării a 7 țări de pe o hartă folosind 3 culori. Scopul este acela de a atribui câte o culoare fiecărei țări, astfel încât nicio țară să nu aibă niciun vecin de aceeași culoare cu aceasta. Soluția problemei va fi o listă de atribuiri din domeniul `[„r“, „g“, „b“]`, care desemnează culorile atribuite fiecărei țări `(1, 2, 3, 4, 5, 6, 7)`.

Această strategie se traduce în următorul cod Prolog:

```
all_members([], _). % predicat care verifică că toate elementele din prima listă sunt prezente în a doua
all_members([X|Rest], In) :- member(X, In), all_members(Rest, In).
solve(S) :-
        L = [_|_], length(L, 7), all_members(L, ["r", "g", "b"]),
        safe(S). % predicat care verifică faptul că țările nu au culori identice cu niciun vecin
```

Programul anterior este foarte ineficient. El construieşte extrem de multe atribuiri, fără a le respinge pe cele „ilegale“ într-un stadiu incipient al construcţiei.

Backtracking atunci când cunoaştem lungimea căii către soluţie

Mecanismul de backtracking ne oferă o rezolvare mai eficientă. Știm că orice soluţie pentru problema colorării hărților constă într-o listă de atribuiri a 3 culori, de genul `[X1/C1,X2/C2, ... X7/C7]`, scopul programului de rezolvare fiind să

instanţieze adecvat variabilele X1, C1, X2, C2 etc. Vom considera că orice soluţie este de forma `[1/C1,2/C2, ... 7/C7]`, deoarece ordinea ţărilor nu este importantă. Fie problema mai generală a colorării a N ţări folosind M culori. Definim soluţia pentru N=7 ca o soluţie pentru problema generală a colorării hărţilor, care în plus respectă template-ul `[1/Y1,2/Y2, ... 7/Y7]`. Semnul „/“ este în acest caz un separator, fără nicio legătură cu operaţia de împărţire.

În Prolog vom scrie:

```
%% Lungimea soluției este cunoscută și fixă.
template([1/_, 2/_, 3/_, 4/_, 5/_, 6/_, 7/_]).

correct([]):-!.
correct([X/Y|Others]):-
        correct(Others),
        member(Y, ["r", "g", "b"]),
        safe(X/Y, Others).

solve_maps(S):-template(S), correct(S).
```

Regulă: Atunci când calea către soluţie respectă un anumit template (avem de instanţiat un număr finit, predeterminat, de variabile), este eficient să definim un astfel de template în program.

Observaţie: În exemplul de mai sus am reţinut explicit ordinea celor 7 ţări. Redundanţa în reprezentarea datelor ne asigură un câştig în viteza de calcul (câştigul se observă la scrierea predicatului safe).

Backtracking atunci când calea către soluţie admite un număr nedeterminat de stări intermediare

În această situaţie nu este posibil să definim un template care descrie forma soluţiei problemei. Vom defini o căutare mai generală, după modelul următor:

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

`search(+StăriVizitate,-Soluţie)` defineşte mecanismul general de căutare astfel:

- căutarea începe de la o stare inițială dată (predicatul `initial_state/1`)
- dintr-o stare curentă se generează stările următoare posibile (predicatul `next_state/2`)
- se testează că starea în care s-a trecut este nevizitată anterior (evitând astfel traseele ciclice)
- căutarea continuă din noua stare, până se întâlneşte o stare finală (predicatul `final_state/1`)

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

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

Căutare în lăţime

Căutarea în lăţime este adecvată situaţiilor în care se doreşte drumul minim între o stare iniţială şi o stare finală. La o căutare în lăţime, expandarea stărilor „vechi“ are prioritate în faţa expandării stărilor „noi“.

```
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 lăţime astfel:

- căutarea începe de la o stare inițială dată care n-are predecesor în spaţiul stărilor (StartNode cu părintele nil)
- se generează toate stările următoare posibile

- se adaugă toate aceste stări la coada de stări încă nevizitate
- căutarea continuă din starea aflată la începutul cozii, până se întâlneşte o stare finală

Căutare A*

A* este un algoritm de căutare informată de tipul best-first search, care caută calea de cost minim (distanță, cost, etc.) către scop. Dintre toate stările căutate, o alege pe cea care pare să conducă cel mai repede la soluție. A* selectează o cale care minimizează `f(n) = g(n) + h(n)`, unde `n` este nodul curent din cale, `g(n)` este costul de la nodul de start până la nodul `n` și `h(n)` este o euristică ce estimează cel mai mic cost de la nodul `n` la nodul final.

```
astar_search(Start, End, Grid, Path) :-
      manhattan(Start, End, H),
      astar(End, [H:Start], [Start:("None", 0)], Grid, Discovered),
      get_path(Start, End, Discovered, [End], Path).
```

`astar(+End, +Frontier, +Discovered, +Grid, -Result)` va defini mecanismul general de căutare A*, astfel:

- căutarea începe de la o stare inițială dată care n-are predecesor în spaţiul stărilor (StartNode cu părintele „None“) și distanța estimată de la acesta până la nodul de final printr-o euristică (exemplu: distanța Manhattan)
- se generează toate stările următoare posibile și se calculează costul acestora adăugând costul acțiunii din părinte până în starea aleasă cu costul real calculat pentru a ajunge în părinte (costul părintelui în Discovered)
- dacă starea aleasă nu este în Discovered sau dacă noul cost calculat al acesteia este mai mic decât cel din Discovered, se adaugă în acesta, apoi va fi introdusă în coada de priorități (Frontier) cu prioritatea fiind costul cu care a fost adaugată în Discovered + valoarea dată de euristică din starea curentă până în cea finală
- căutarea continuă din starea aflată la începutul cozii, până se întâlneşte o stare finală

Aflarea tuturor soluțiilor pentru satisfacerea unui scop

Prolog oferă un set special de predicate care pot construi liste din toate soluțiile de satisfacere a unui scop. Acestea sunt extrem de utile deoarece altfel este complicată culegerea informațiilor la revenirea din backtracking (o alternativă este prezentată în secțiunea următoare, mai mult în laboratorul viitor).

findall(+Template, +Goal, -Bag)

Predicatul `findall` pune în Bag câte un element Template pentru fiecare soluție a expresiei `Goal`. Desigur, predicatul este util atunci când `Goal` și `Template` au variabile comune. De exemplu:

```
even(Numbers, Even):-
    findall(X,(member(X, Numbers), X mod 2 =:= 0), Even).
```

```
?- even([1,2,3,4,5,6,7,8,9], Even).
Even = [2, 4, 6, 8].
```

Resurse

- Exerciții
- Cheatsheet

Referinţe

- „Prolog Programming for Artificial Intelligence“, Ivan Bratko