Probleme rezolvabile prin backtracking Problema taranului

Mecanismul implicit de functionare a limbajului Prolog este bazat pe **backtracking**. De aceea problemele care se rezolva prin backtracking se implementeaza mai usor în Prolog decât în alte limbaje.

Fie urmatoarea problema: Pe malul unui râu se afla un taran cu un lup, o capra si o varza. Ţăranul doreste sa traverseze cu ele râul. Taranul poate face traversari înot ale râului împreuna doar cu unul din cele trei personaje de transportat, sau singur. Daca lupul ramâne pe acelasi mal împreuna cu capra si taranul este pe celalalt mal, capra va fi mâncata de lup. Similar se întâmpla cu varza si capra. Să se scrie un program Prolog care sa îi dea taranului toate solutiile de traversare.

Vom prezenta o rezolvare pe baza de backtracking. Definim starea problemei la un moment dat ca fiind:

stare(Taran, Lup, Capra, Varza)

unde **Taran**, **Lup**, **Capra** si **Varza** indică poziţia (malul) fiecarui element şi pot avea valorile **stang** sau **drept**. În acest fel stim exact pe ce mal se afla fiecare dintre personaje. De remarcat ca aceasta este o structura si nu un predicat. De aici rezulta ca starea intiala este:stare(stang, stang, stang, stang), iar starea finala este:stare(drept, drept, drept, drept).

Rezolvarea problemei prin backtracking presupune încercarea unei miscari posibile în starea curenta si apoi reluarea algoritmului. O miscare genereaza o noua stare, din care se vor încerca apoi alte miscari. Daca se ajunge într-o stare finala, s-a gasit o solutie. În acest caz trebuie facuta o revenire pentru a încerca gasirea altor solutii. Trebuie specificate, evident, starea iniţială şi starea sau stările finale.

Daca se ajunge într-o stare *ilegala*, cum ar fi de exemplu, în cazul acestei probleme, stare(stang,stang,drept,drept), se executa tot o revenire în starea anterioara. Acelasi lucru trebuie să se întâmple şi daca starea curenta este una în care s-a mai ajuns odată. Nu are rost sa trecem de mai multe ori prin aceeasi configuratie deoarece acest lucru nu face decât sa lungeasca numarul de miscari necesare şi poate conduce la bucle infinite.

Deci vom avea nevoie de un predicat care, dându-se o stare, sa stie sa execute miscarile posibile din acea stare. Fie acest predicat:

miscare(Stare, StareUrmatoare).

Sa vedem cum se defineste predicatul **miscare** în cazul acestei probleme:

1) Taranul poate lua lupul cu el pe celalat mal:

miscare(stare(Taran, Taran, Capra,Varza), stare(Taran1, Taran1, Capra,Varza)) **:-**opus((Taran, Taran1).

2) Taranul poate lua capra cu el pe celalat mal:

miscare(stare(Taran, Lup, Taran,Varza), stare(Taran1, Lup, Taran1, Varza)) **:-** opus(Taran, Taran1).

3) Taranul poate lua varza cu el pe celalat mal:

miscare(stare(Taran, Lup, Capra, Taran), stare(Taran1, Lup, Capra, Taran1)) **:-** opus(Taran, Taran1).

4) Taranul poate traversa singur râul:

```
miscare(stare(Taran, Lup, Capra, Varza), stare(Taran1, Lup, Capra, Varza)) :-
        opus(Taran, Taran1).
```

Se observa ca am folosit predicatul **opus(Mal1, Mal2)**, care determină malul opus unui mal dat. Definitia lui este:

```
opus(stang, drept).
opus(drept, stang).
```

Predicatele **initiala(S)** si **finala(S)** specifica daca o stare este sau nu initiala, respectiv finala. Definitiile lor sunt:

```
initiala(stare(stang, stang, stang, stang)).
finala(stare(drept, drept, drept, drept)).
```

Predicatul **ilegala(Stare)** spune daca o stare este ilegala:

1) Daca lupul este pe acelasi mal cu capra si taranul este pe celalalt mal:

```
ilegala(stare(Taran, Lup, Lup, _ )) :- opus(Taran, Lup).
```

2) Daca varza este pe acelasi mal cu capra si taranul este pe celalalt mal:

```
ilegala(stare(Taran, _ ,Capra, Capra)) :- opus(Taran, Capra).
```

Se poate scrie acum predicatul de gasire a unei succesiuni de mutari care rezolva problema. Deoarece nu trebuie sa repetam stari, predicatul va primi, pe lânga starea curenta, si o lista de stari deja parcurse, întorcând lista de stări (Solutie) care constituie rezolvarea:

```
lcv(+Stare, -Solutie, -Vizitate).
```

unde **Stare** este starea curenta, **Solutie** este solutia ce trebuie gasita, iar **Vizitate** este lista de stări parcurse. Acest predicat se scrie ținând cont de:

1) Daca starea curenta este o stare finala, atunci soluție este lista de stări parcurse împreuna cu starea curenta:

```
lcv(Stare, [Stare|Vizitate], Vizitate) :- finala(Stare).
```

2) Altfel genereaza o noua mutare, testeaza dacă este legala, testeaza dacă nu a mai fost parcursa si relanseaza cautarea din noua stare:

```
lcv(Stare, Solutie, Vizitate) :-
    miscare(Stare, StareUrmatoare),                  % o noua mutare
    not(ilegala(StareUrmatoare)),                    % este ilegala?
    not(member(StareUrmatoare, [Stare | Vizitate])), % este deja parcursa?
lcv(StareUrmatoare, Solutie, [Stare | Vizitate]).    % reia cautarea
```

Predicatul **rezolva** rezolva problema, întorcând lista stărilor parcurse din starea inițială în starea finală, deci soluția problemei.

```
rezolva(Solutie) :- initiala(Stare), lcv(Stare, Solutie, [] ).
```

Programul complet este prezentat mai jos:

```
initiala(stare(stang, stang, stang, stang)).
finala(stare(drept, drept, drept, drept)).

opus(stang, drept).
opus(drept, stang).

ilegala(stare(Taran, Lup, Lup, _)):- opus(Taran, Lup).
ilegala(stare(Taran, _, Capra, Capra)):- opus(Taran, Capra).


miscare(stare(Taran, Lup, Capra, Varza),
        stare(TaranN, Lup, Capra, Varza)):- opus(Taran, TaranN).

miscare(stare(Taran, Taran, Capra, Varza),
        stare(TaranN, TaranN, Capra, Varza)):- opus(Taran, TaranN).

miscare(stare(Taran, Lup, Taran, Varza),
        stare(TaranN, Lup, TaranN, Varza)):- opus(Taran, TaranN).

miscare(stare(Taran, Lup, Capra, Taran),
        stare(TaranN, Lup, Capra, TaranN)):- opus(Taran, TaranN).

lcv(Stare, [Stare|Vizitate], Vizitate):- finala(Stare).
lcv(Stare, Solutie, Vizitate):- miscare(Stare, StareUrm),
                                not(ilegala(StareUrm)),
                                not(member(StareUrm, [Stare|Vizitate])),
                                lcv(StareUrm, Solutie, [Stare|Vizitate]).

rezolva(Solutie):-initiala(Stare), lcv(Stare, Solutie, []).

afis([]):- nl.
afis([X|R]):- afis(R), nl, write(X).

% Am scris astfel predicatul de afisare pentru a obtine solutia ca o
% succesiune de stari, pornind de la starea initiala pana la starea
% finala. Predicatul rezolva intoarce succesiunea de stari, pornind de
% la starea finala si terminand cu starea initiala. De aceea, predicatul
% de afisare intoarce aceasta lista, dar in ordine inversa, pornind de
% la starea initiala si terminand cu starea finala.

go:-rezolva(L), afis(L), fail.
```