

Prolog CheatSheet

Laborator 11

Liste și funcții utile pe liste

```
[ ] - lista vida
[a,b,c] - elementele a, b, c
[Prim|Rest] - element concatenat la rest
[X1,X2,...,XN|Rest] - n elemente concatenate la restul listei

append(?List1, ?List2, ?List1AndList2) % List1AndList2 este
concatenarea între List1 și List2
?- append([1,2,3], [4,5], X).
X = [1, 2, 3, 4, 5].
?- append([1,2,3], X, [1,2,3,4]).
X = [4].
?- append([1,2,3], X, [2,3,4]).
false.

member(?Elem, ?List) % Adevărat dacă Elem esre prezent în List
?- member(1, [1,2,3]).
true ;
false.
?- member(X, [1,2,3]).
X = 1 ;
X = 2 ;
X = 3.

length(?List, ?Int) % Adevărat dacă Int reprezintă numărul de
elemente din List
?- length([1,2,3], X).
X = 3.

reverse(?List1, ?List2) % Adevărat atunci când elementele din
List2 sunt în ordine inversă față de List1
?- reverse([1, 2, 3], X).
X = [3, 2, 1].

sort(+List, -Sorted) % Adevărat atunci când Sorted conține toate
elementele din List (fără duplicate), sortate în ordinea
standard
?- sort([2, 3, 1, 2], X).
X = [1, 2, 3].
```

Backtracking când se cunoaște lungimea căii către soluție

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

De exemplu, pentru problema celor opt regine putem scrie astfel:

```
template([1/_ , 2/_ , 3/_ , 4/_ , 5/_ , 6/_ , 7/_ , 8/_]).
```

```
correct([ ]):-!.
correct([X/Y|Others]):-
    correct(Others),
    member(Y, [1, 2, 3, 4, 5, 6, 7, 8]),
    safe(X/Y, Others). % predicat care verifică faptul că
                        reginele nu se atacă între ele
```

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

Backtracking când calea are un număr nedeterminat de stări

În această situație 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).
```

search(+StăriVizitate,-Soluție) definește 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 întâlnește 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).
```

Mecanism BFS

bfs(+CoadăStărilorNevizitate,+StăriVizitate,-Soluție) va defini mecanismul general de căutare în lățime, astfel:

- căutarea începe de la o stare inițială dată care n-are predecesor în spațiul 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 întâlnește o stare finală

```
bfs([(StartNode,nil)], [], Discovered).
```

Mecanism A*

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 spațiul stărilor (StartNode cu părintele nil) 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 adăugată în Discovered + valoarea dată de euristică din starea curentă până în cea finală
- căutarea continuă din starea aflată la începutul cozii, până se întâlnește o stare finală

```
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).
```