

Lab.2 Inteligență artificială– Generarea ciclărilor

Semnificația interogărilor

S-a pus în evidență pe parcursul primului laborator că polimorfismul predicatelor este nativ în Prolog (este o replicare, la nivel computațional a polimorfismului gândirii umane). S-a arătat deja cum interogând pe baza aceluiași predicat se obțin răspunsuri ale căror semnificații diferă fundamental de la un caz la altul. De aceea este momentul să facem acum câteva clarificări privind semantica interogărilor. Vom referi următoarele tipuri de interogări sub numele asociate lor în tabelul de mai jos:

Interogare Prolog:	Interogare în limbaj natural:	Numele operațiunii semantice:
predicat(_)	Există măcar o soluție? Există o valoare care face proprietatea adevărată? Există o ipostază în care proprietatea se verifică?	testul satisfiabilității minimale sau căutarea unei soluții
predicat	Este adevărat ca ...?	testul validității
predicat(X) cu X nelegată	Care sunt soluțiile problemei? Care sunt ipostazele în care proprietatea se verifică?	testul satisfiabilității maximele sau căutarea tuturor soluțiilor
predicat(X) cu X legată	Verifică valoarea / individul / ipostaza X proprietatea data?	testul satisfiabilității punctuale sau verificarea unei soluții particulare

Predicate de aritate nulă

Se consideră predicatul:

```
run1:-X=3,nl,write(X),nl.
```

Rulându-l ca scop extern se constată că are ca efect tipărirea numărului 3 și returnează "True" (Yes). Îi vom adăuga acum o instrucțiune contradictorie cu instanțierea X=3:

```
run2:-X=3,nl,write(X),nl,X=4.
```

Ca efect, la rularea ca scop extern a predicatului se printează 3 și se returnează "False" (No). Acum se va schimba ordinea instrucțiunilor după cum urmează:

```
run3:-X=3,nl,X=4,write(X),nl.
```

La rulare se constată că printarea lui X nu mai are loc ci doar se returnează "False". Predicatul de mai sus ilustrează următoarea regulă care funcționează în cazul tuturor predicatelor (inclusiv cele de aritate nulă) definite printr-o conjuncție de instrucțiuni (de alte predicate):

R1: Dacă un predicat este definit prin conjuncția a n predicate atunci predicatul de indice k ($1 < k \leq n$) se execută dacă și numai dacă toate predicatele care-l precedă au returnat "True". Dacă predicatul de indice k ($1 \leq k < n$) returnează "False" atunci toate predicatele care îi succed în conjuncție nu se mai execută. Pe scurt, o conjuncție ratează testul satisfiabilității/validității odată cu prima componentă falsă.

Acest comportament se explică prin faptul că valoarea logică a unei conjuncții este "False" de îndată ce s-a găsit cel puțin o propoziție falsă în conjuncție, ceea ce conduce la inutilitatea parcurgerii următoarelor propoziții din conjuncție. Se poate spune deci că

predicatul “run3” de mai sus iese din execuție la capăt sau după prima instrucțiune care returnează “False”.

Se implementează acum predicatul “run4” prin două alternative (implicite, respectiv explicite):

```
run4:-X=3,nl,write(X),nl;           (1)
      X=4,nl,write(X),nl.
```

```
run44:-xeste(X),nl,write(X),nl.    (2)
```

unde:

```
xeste(X):-X=3;                     (3)
          X=4.
```

Rulând predicatele se constată că, în ambele variante, are loc printarea lui 3 și că predicatul returnează “True”. Este evident deci că a doua alternativă a fost total ignorată.

Exemplele (1), (2) date mai sus ilustrează următoarea regulă:

R2: Predicatele de aritate nulă definite printr-o disjuncție nu generează backtracking-ul alternativelor sale și ies din execuție odată cu prima alternativă adevărată. Pe scurt, un predicat de aritate definit printr-o disjuncție trece testul validității odată cu prima componentă adevărată.

În opoziție cu comportamentul descris mai sus, predicatele:

```
run5(X):- X=3,nl,write(X),nl;      (4)
          X=4,nl,write(X),nl.
```

```
run55(X):-xeste(X),nl,write(X),nl. (5)
```

relate ca scop extern sub forma “run5(X)”, respectiv, “run55(X)”, returnează două soluții, ceea ce dovedește că, indiferent de caracterul implicit/explicit al alternativelor, și a doua alternativă a ajuns să fie executată. Se întâmplă astfel pentru că existența alternativelor conduce în acest caz la backtracking.

Exemplele date mai sus ilustrează următoarea regulă:

R3: Dacă este definit printr-o disjuncție, un scop care returnează valori ale unor variabile din lista de apel (o căutare, un test de satisfiabilitate) generează întotdeauna backtracking-ul alternativelor.

Predicatele care returnează prin nume valori de adevăr și nu valori ale variabilelor din lista de apel (adică testele de validitate), chiar dacă sunt definite prin alternative (implicite sau explicite), nu generează obligatoriu backtracking.

Obs:

- R1 funcționează identic în cazul predicatelor de aritate nenulă (exemplificați!) în timp ce R2 și R3 postulează o diferențiere între căutări și teste de validitate.
- Pentru a forța parcurgerea celei de a doua alternative în cazul (1), se va adăuga în prima alternativă predicatul “fail” care garantează că prima alternativă va returna „False”.

```
run444:-X=3,nl,write(X),nl,fail; .  (6)
        X=4,nl,write(X),nl.
```

Rularea lui ca scop extern va evidenția parcurgerea ambelor alternative. Situația va rămâne neschimbată dacă alternativele sunt implicite:

```
run4444:-xeste(X),nl,write(X),nl,fail.
xeste(X):-X=3;
X=4. (7)
```

cu diferența că, la terminare, “run” returnează “False”.

- Comportamentul descris în situația (1) este motivat de faptul că valoarea logică a unei disjuncții este „True” de îndată ce s-a găsit o propoziție din disjuncție care returnează „True”.

Exemplele (6) și (7) ilustrează următoarele regulă:

R4: Pentru ca un predicat de aritate nulă definit alternativ (implicit sau explicit) să genereze backtracking-ul alternativelor sale, el trebuie să-și rateze cu consecvență alternativele (lucru realizabil prin folosirea predicatului „fail”).

Ciclarea infinită

De importanță practică este realizarea unei proceduri care să ruleze un scop de maniera unei ciclări infinite până când utilizatorul decide să se oprească. Cum blocul de instrucțiuni care va fi ciclat va diferi de la o problemă practică la alta, mecanismul de ciclare trebuie conceput independent de problema practică, deci independent de numărul și natura variabilelor și parametrilor particulari corespunzători fiecărei probleme. Ca urmare, un candidat de luat în seamă pentru rezolvarea acestei cerințe este un predicat din clasa celor de aritate nulă.

```
run:-repetă,blocdeinstrucțiuni.
```

În care ciclarea infinită va fi garantată de definirea cu alternative implicite (predicatul recursiv „repetă” va aduce alternativele). Aplicând R4 deducem că se impune cu necesitate ca predicatul „run” să-și rateze alternativele. Ca urmare acesta va conține un predicat „fail”. Blocul de instrucțiuni ciclat îl vom înlocui, de exemplu, cu două instrucțiuni simple: o citire și o scriere.

```
run:-repetă,nl,readreal(X),nl,write(X),fail.
```

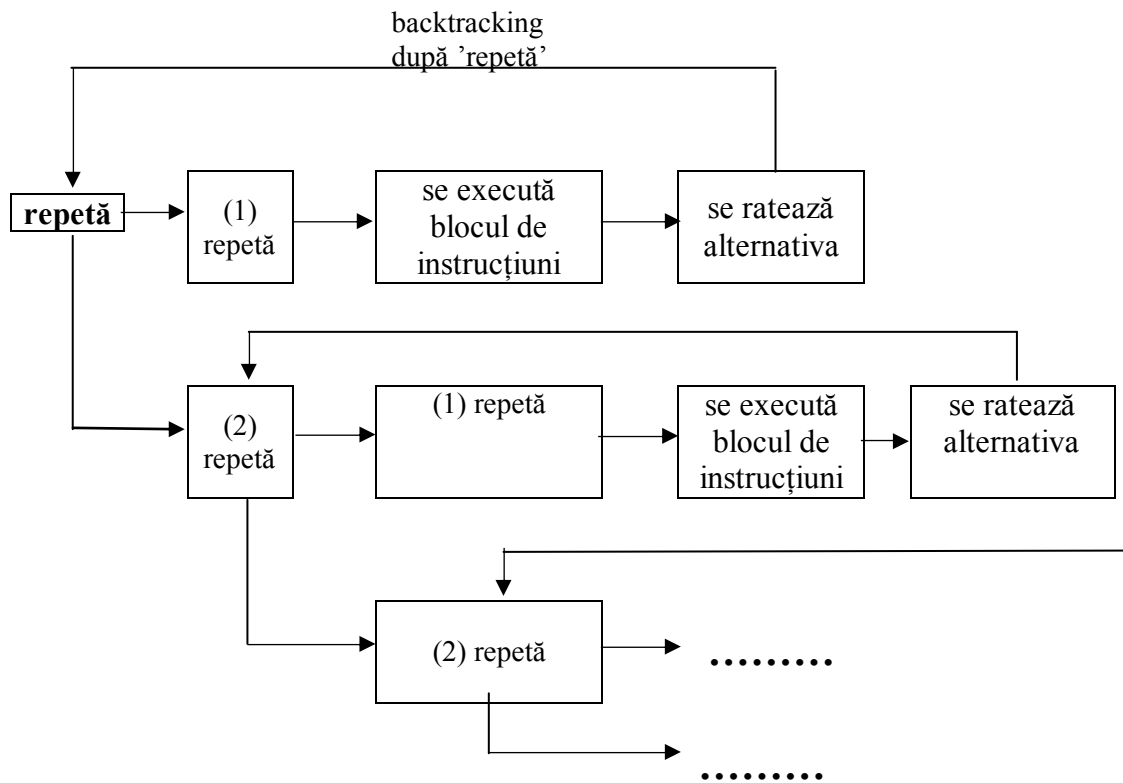
Cum blocul de instrucțiuni va fi parcurs cel puțin o dată (vezi și R1) predicatul „repetă” trebuie să returneze „True”. Ca urmare prima clauză care îl implementează este un fapt:

```
run:-repetă,nl,readreal(X),nl,write(X),fail.
repetă.
```

După prima parcurgere a blocului de instrucțiuni, conjuncția care a funcționat ca primă alternativă (implicită) a lui „run” este forțată (prin „fail”) să returneze „False”. Ca urmare acum se declanșează backtracking-ul după alternativele predicatului „repetă” (care sunt, de fapt, alternativele implicite ale lui „run”). În consecință, a doua alternativă a lui „repetă” se execută acum și trebuie să returneze, ca și prima „True”, și să genereze următoarea repetiție:

```
run6:-repetă,nl,readreal(X),nl,write(X),fail.
repetă. (1)
repetă:-repetă (2)
```

Rularea predicatului „run” ca scop extern va produce ciclarea infinită a blocului de instrucțiuni:



Ciclarea cu test final

Pentru a ameliora comportamentul predicatului „repetă” astfel încât să nu mai cicleze la infinit ci să continue ciclarea doar după acceptul utilizatorului vom schimba regula „repetă” de la „Repetă orice ar fi / Repeta necondiționat” la „Dacă vrei să repeți, atunci repetă”:

```

run7:-repetă2,nl,readreal(X),nl,write(X),fail.
repetă2.
repetă2:-vreisarepeti,repetă2.
vreisarepeti:-nl,write(„vrei să repeți?(y/n)”),readchar(Y),Y=’y’.
  
```

În momentul în care utilizatorul introduce ’n’, predicatul „vreisarepeti” returnează „False” și ca urmare ultima alternativă (cea actuală în execuție) a lui „repetă” returnează și ea „False” iar autoapelul nu se mai produce (vezi și R1), ieșindu-se astfel din ciclare. Se observă în ultima formă a predicatului „run” că parcurgerea blocului de instrucțiuni are loc cel puțin o dată. Ca urmare, forma echivalentă în pseudocod a predicatului „run” este o **ciclare cu test final**:

Repeat bloc de instrucțiuni
Until răspuns negativ

Ciclarea cu test inițial

Dacă se dorește ca până și prima ciclare să fie condiționată (caz în care blocul de instrucțiuni ar putea să nu se execute nici măcar o dată), atunci prima alternativă a lui „run” trebuie să fie și ea condiționată. Cum punctul de backtracking al predicatului „run” este predicatul „repeat”, deducem că orice instrucțiune plasată înaintea lui nu va avea nici un efect asupra celei de a doua ciclări și nici a celor ulterioare (instrucțiunile situate înaintea punctului de backtracking se vor executa o singură dată și anume în prima ciclare):

```
run8:vreisarepeti,repeta2,nl,readreal(X),nl,write(X),fail.
vreisarepeti:-nl, write("executați?(y/n)"),readchar(Y),Y='y'.
repeta2.
repeta2:-vreisarepeti,repeta2.
```

Predicatul „run8” este echivalent în pseudocod cu o **ciclare cu test inițial**:

```
While răspuns afirmativ
  Do
    bloc de instrucțiuni
  EndDo
EndWhile
```

Redefinirea predicatelor. Ciclarea (finită) iterativă.

Calculul factorialului.

Înainte de a trece mai departe, scrieți predicatul Prolog care să calculeze factorialul:

- recursiv de la 0 la N (forward recursion),
- recursiv de la N la 0 (backward recursion),
- recursiv pe două ramuri: de la $[N/2]$ la 0 și de la $[N/2]+1$ la N.

Porniți de la formulările pseudocod sau C/C++.

Transpunerea în Prolog a unei ciclări iterative de contor i ,

```
For i = vi to vf
  Do
    bloc de instrucțiuni
  EndDo
EndFor
```

(cu i între o valoare inițială vi și o valoare finală vf) și simularea iterațiilor în cadrul unui predicat sunt posibile prin redefinirea predicatului în sensul măririi numărului de argumente și prin definirea sa recursivă. După caz, se vor adăuga argumente care să stocheze indicele de ciclare, rezultatul sau rezultatele intermediare și valoarea finală a indicelui de ciclare. Întoarcerea din recursivitate se va produce atunci când indicele de ciclare ajunge la valoarea finală vf și va fi asigurată prin implementarea unui fapt.

Adăugarea argumentelor suplimentare nu este întotdeauna o necesitate, dar, pentru că permite definirea predicatelor recursive la urmă, este binevenită.

Să reconsiderăm exemplul factorialului:

În Prolog:	$\text{factorial}(N, \text{Rez}) :- \text{factorial}(N, 0, 1, \text{Rez}).$
În limbaj natural:	Factorial de N este Rez dacă Rez este rezultatul aplicării unei proceduri recursive la urmă care simulează ciclarea iterativă de la 0 la N și calculează factorialul (până la cel mai interior autoapel) prin actualizarea și propagarea unui rezultat intermediar (inițializat aici cu 1).

unde: 0 este valoarea inițială a indicelui de ciclare; 1 este primul rezultat intermediar (0!); $\text{factorial}(, , ,)$ este predicat recursiv la urmă implementat printr-un fapt (care asigură întoarcerea din recursivitate în momentul când indicele de ciclare a ajuns la N) și o regulă care propagă autoapelul (simulând ciclarea):

$\text{factorial}(N, \text{Rez}) :- \text{factorial}(N, 0, 1, \text{Rez}).$

$\text{factorial}(N, N, \text{Rez}, \text{Rez}).$

$\text{factorial}(N, I, R_i, \text{Rez}) :- I < N, I_a = I + 1, R_{i_a} = R_i * I_a, \text{factorial}(N, I_a, R_{i_a}, \text{Rez})$

Se observă că predicatul $\text{factorial}(, , ,)$ găsește rezultatul Rez în cel mai interior autoapel, îl propagă neschimbat la închiderea autoapelurilor (la întoarcerea din recursivitate), nu conține puncte de backtracking și este implementat printr-o regulă terminată în autoapel. În concluzie predicatul $\text{factorial}(, , ,)$ este recursiv la urmă.