Seminar 4 – Backtracking în Prolog

- Prolog folosește backtracking pentru a găsi toate soluțiile. Până acum, am rezolvat probleme care aveau o singură soluție (de ex., să găsim maximul unei liste) și trebuia fie să adaugăm în clauze tăieturi care să oprească backtracking-ul, fie să adăugăm condiții pentru a ne asigura că avem o singură soluție.
- La acest seminar abordăm o problemă care are mai multe soluții, faptul că Prolog folosește backtracking devenind de mare ajutor. Vom implementa predicate care au mai multe soluții (predicate **nedeterministe**).

1. Se dă șirul $a_1,..., a_n$ format din numere întregi distincte. Se cere să se genereze lista tuturor submulțimilor ordonate cu aspect de ”vale” (o secvență are aspect de vale dacă este formată dintr-o secvență strict descrescătoare, urmată de o secvență strict crescătoare).

   De exemplu, să considerăm șirul [5,4,3,6,7,-1,-2,-3]. Câteva exemple de submulțimi cu aspect de vale sunt: [5,4,6], [6,4,5], [5,4,3,6,7], [-3,-2,-1,3,4,5], [-1,-3,-2], etc.

- O primă variantă – foarte ineficientă – de a rezolva problema ar fi să folosim 2 predicate: unul care generează toate submulțimile ordonate (aranjamente de cel puțin 3 elemente) și altul care verifică dacă o listă are sau nu aspect de vale. Problema este că o listă are foarte multe astfel de submulțimi diferite (mai ales că în cazul nostru ordinea elementelor în listă contează) și este probabil ca o mare parte dintre ele să nu aibă aspect de vale.

- Observăm că dacă avem lista [5,3,7] (care este o vale) la această listă nu mai putem adăuga elemente din lista originală (4,-1,-2,-3) deci nu e nevoie să mai continuăm să generăm liste adăugând aceste elemente.
- De aceea, pentru a avea o soluție mai eficientă, va trebui să combinăm generarea listelor cu verificarea condiției. Vom folosi o listă candidat, care reprezintă o soluție candidat, o soluție parțială. Vom adăuga elemente pe rând în această listă candidat, dar numai elemente cu care putem completa soluția parțială a.î. să se poată obține o soluție finală (o listă cu aspect de vale).
- Pentru a genera o listă cu aspect de vale avem nevoie de o secvență de elemente descrescătoare urmată de o secvență de elemente crescătoare. Când adăugăm elemente în lista candidat va trebui să ținem cont dacă suntem pe secvența de elemente descrescătoare sau dacă suntem pe secvența de elemente crescătoare. Pentru acest lucru vom reține un parametru în plus, care ne indică direcția pe care suntem. De exemplu:
  - Valoarea 0 indică că suntem pe partea descrescătoare (partea stângă)
  - Valoarea 1 indică că suntem pe partea crescătoare (partea dreaptă)
- Lista candidat este o listă colectoare. Pentru a evita adăugarea la finalul listei colectoare (pe considerentul eficienței), vom opta să adăugăm elemente la începutul listei candidat.

Dar aceasta presupune să generăm soluția în ordinea inversă (adică de la dreapta la stânga). De exemplu, soluția [5,4,3,6,7] va fi generată astfel:
- [7]
- [6,7]
- [3,6,7]
- [4,3,6,7]
- [5,4,3,6,7]

- Când adăugăm un element nou în lista candidat va trebui să ținem cont de 2 valori - direcția și primul element din lista candidat (care este practic ultimul element adăugat în lista candidat) - astfel:
  - Direcție 0 și element de adăugat mai mare decât primul element din candidat => mergem mai departe cu partea descrescătoare (ex. 8, [7,5,6], candidatul devine [8,7,5,6])
  - Direcție 1 și element de adăugat mai mic decât primul element din candidat => mergem mai departe cu partea crescătoare (ex. 5, [7,8], candidatul devine [5,7,8])
  - Direcția 1 și element de adăugat mai mare decât primul element din candidat => începem să generăm partea descrescătoare (ex. 9, [5,7,8], candidatul devine [9,5,7,8]).
  - Direcția 0 și element de adăugat mai mic decât primul element din candidat => nu putem adăuga (ex. 4, [7,6,5,8]).

- Când este o listă candidat o soluție? Pentru avea o listă cu aspect de vale, ne trebuie atât secvență descrescătoare, cât și secvență crescătoare. Dintre aceste secvențe, prima dată se generează cea crescătoare, deci dacă lista candidat are o secvență descrescătoare (adică parametrul pentru direcție a devenit 0), avem o soluție. Dar, în același timp, este posibil ca această listă candidat să fie extinsă pentru a genera alte soluții.
- Cum generăm elementele care vor fi adăugate în lista candidat? Ne trebuie un predicat care să ne dea un element din lista inițială, care eventual este adăugat în candidat (în funcție de condițiile discutate mai sus), după care să ne dea un alt element ș.a.m.d.. Este primul predicat nedeterminist de care avem nevoie, unul care ne furnizează pe rând toate elementele dintr-o listă.

$$candidat(l_1 ... l_n) = \begin{cases} 1.\ l_1 \ dacă\ n > 0 \\ 2.\ candidat(l_2 ... l_n)\ dacă\ n > 0 \end{cases}$$

```
% candidat(L:list, E: element)
% model de flux: (i,o), (i,i)
% L - lista inițială, din care se generează elemente candidat
% E - un element din lista L
candidat([H|_], H).
candidat([_|T], E):-
      candidat(T, E).
```

- După ce am generat un element din listă, trebuie să verificăm ca acest element să nu mai apară în lista candidat. Pentru această verificare putem folosi tot predicatul candidat, dar cu un alt model de flux: (i,i).
- Acum avem de scris predicatul care generează soluțiile. Acesta va trebui să aibă următorii parametri:
    - Lista din care luăm elemente – L
    - Lista candidat în care adăugăm elemente unul câte unul - C
    - Direcția (dacă suntem pe partea cu elemente descrescătoare sau elemente crescătoare)
    - Rezultatul (doar în Prolog, nu și în model matematic)

$generare(L,\ c_1 ... c_n,\ D)\ =$

1. $c_1 ... c_n,\ dacă\ D\ =\ 0$
2. $generare(L,\ E\ \cup c_1 ... c_n,\ 0)\ dacă\ E\ =\ candidat(L),\ E\ \notin c_1 ... c_n,\ E\ >\ c_1,\ D\ =\ 0$
3. $generare(L,\ E\ \cup\ c_1 ... c_n,\ 1)\ dacă\ E\ =\ candidat(L),\ E\ <\ c_1,\ D\ =\ 1$
4. $generare(L,\ E\ \cup\ c_1 ... c_n,\ 0)\ dacă\ E\ =\ candidat(L),\ E\ \notin c_1 ... c_n,\ E\ >\ c_1,\ D\ =\ 1$

Observăm că ramurile 2 și 4 sunt foarte asemănătoare, singura diferență fiind valoarea variabilei D. Ele pot fi condensate într-o singură ramură. Astfel, obținem următorul model matematic:

$generare(L,\ c_1 ... c_n,\ D)\ =$

1. $c_1 ... c_n,\ dacă\ D\ =\ 0$
2. $generare(L,\ E\ \cup\ c_1 ... c_n,\ 0)\ dacă\ E\ =\ candidat(L),\ E\ \notin c_1 ... c_n,\ E\ >\ c_1$
3. $generare(L,\ E\ \cup\ c_1 ... c_n,\ 1)\ dacă\ E\ =\ candidat(L),\ E\ <\ c_1,\ D\ =\ 1$

```
%generare(L:list, C:list, Directie:intreg, R:list)
%model de flux (i,i,i,o) nedeterminist, (i, i, i, i) determinist
% L - lista inițiala pentru care vom genera listele având aspect de
vale
% C - soluție candidat în care construim element cu element soluția
% Directie - 0, daca suntem pe partea de descreștere, 1 dacă suntem pe
% partea de creștere
% R - rezultat
generare(_, Cand, 0, Cand).
generare(L, [H|Cand], _, R):-
      candidat(L, E),
      not(candidat([H|Cand], E)),
      E > H,
      generare(L, [E,H|Cand], 0, R).
generare(L, [H|Cand], 1, R):-
      candidat(L, E),
      E < H,
```

```
	generare(L, [E,H|Cand], 1, R).
```

- Ne mai trebuie un predicat care să apeleze predicatul generare. Din moment ce împărțim lista candidat în primul element și restul listei, nu putem începe candidatul cu lista vidă. Va trebui să generăm primul element (folosind predicatul candidat), și să apelăm predicatul generare având ca listă candidat lista cu acest element.
- Predicatul generare are o problemă: generează ca soluție și liste care au doar secvență descrescătoare. Deci, dacă tot va trebui să mai scriem un predicat pentru primul apel, în acest predicat vom genera primele 2 elemente din candidat, apelând generare doar dacă aceste 2 elemente sunt în ordinea potrivită.

$start(L) = generare(L, [E_1, E_2], 1)\ dacă\ E_1 = candidat(L),\ E2 = candidat(L),\ E_1 < E_2$

```
%start(L:list, Rez: list)
%model de flux (i,o) (i,i)
%L - lista inițială pentru care generăm listele cu aspect de vale
%Rez - o submulțime ordonata cu aspect de vale a șirului L
start(L, Rez):-
	candidat(L, E1),
	candidat(L, E2),
	E1 < E2,
	generare(L, [E1,E2], 1, Rez).
```

Predicatul `start` generează listele cu aspect de vale pe rând și putem să le vedem apăsând ; după fiecare soluție. Dacă vrem să avem o listă cu toate soluțiile (fără să trebuiască să apăsăm ;), putem folosi predicatul `findall` din Prolog.

$startAll(l_1 ... l_n) = \cup\ start(l_1 ... l_n)$

```
%startAll(L:list, Rez: list)
%model de flux (i,o) (i,i)
%L - lista inițială pentru care generăm listele cu aspect de vale
%Rez - lista tuturor submultimilor ordonate cu aspect de vale

startAll(L, Rez):- findall(R, start(L, R), Rez).
```

O varianta alternativă de rezolvare constă în implementarea predicatului candidat astfel încât să returneze atât elementul generat, cât și lista obținută prin eliminarea elementului generat din lista originală.

$$candidat2(l_1 ... l_n) = \begin{cases} 1.\ (l_1,\ l_2 ... l_n)\ dacă\ n > 0 \\ 2.\ (E,\ l_1 \cup R)\ unde\ (E,\ R) = candidat2(l_2 ... l_n)\ dacă\ n > 0 \end{cases}$$

```
% candidat2(L: list, E: element, R: list)
```

```
% model de flux: (i,o,o), (i,i,i), (i,o,i), (i,i,o)
% L - lista inițiala, din care se generează elemente candidat
% E - un element din lista L
% R - lista inițială din care elementul generat a fost șters

candidat2([E|T], E, T).
candidat2([H|T], E, [H|R]) :-
      candidat2(T, E, R).
```

Dacă folosim predicatul candidat2, în predicatul `generare` apelul recursiv se face cu lista returnată de candidat2, nu cu lista originală L.

- Cum ar trebui modificată soluția noastră dacă dorim ca în soluție să avem elementele în aceeași ordine precum în lista originală (subșiruri ale listei originale)?
- Din moment ce generăm candidatul în ordine inversă, după ce am adăugat un element în candidat vom putea adăuga doar elemente care în lista originală sunt în fața elementului adăugat. De exemplu din lista [5,4,3,6,7,-1,-2,-3], dacă am generat deja candidatul [6,7], următorul candidat trebuie generat doar din lista [5,4,3]. Pentru a realiza acest lucru, cea mai simplă variantă este să modificăm predicatul candidat, pentru ca să returneze și această listă.

$$candidat3(l_1 \ldots l_n) = \begin{cases} 1.\ (l_1,\ \emptyset)\ dacă\ n > 0 \\ 2.\ (E,\ l_1 \cup R)\ unde\ (E,\ R) = candidat3(l_2 \ldots l_n)\ dacă\ n > 0 \end{cases}$$

```
%candidat3(L:list, E:element, R:list)
%model de flux: (i,o,o), (i,i,i), (i,o,i), (i,i,o)
%L - lista inițială
%E - un element din lista L
%R - sublista listei L până la elementul E
candidat3([E|_], E, []).
candidat3([H|T], E, [H|R]):-
      candidat3(T, E, R).
```

`candidat3([1,2,3,4,5], E, R)` ne dă următoarele soluții

- E = 1, R = []
- E = 2, R = [1]
- E = 3, R = [1,2]
- E = 4, R = [1,2,3]
- E = 5, R = [1,2,3,4]