Seminar 4 – Backtracking în Prolog

- Prolog folosește backtracking pentru a găsi toate soluțiile pentru o anumită problemă. Până acum am lucrat cu probleme care aveau o singură soluție (de ex. să găsim maximul unei liste) și trebuia să adaugăm în clauze tăietura care să oprească backtracking-ul, sau trebuia să adăugăm condiții pentru a ne asigura că avem o singură soluție.
- La acest seminar vom vorbi de o problemă care are mai multe soluții, și unde faptul că Prolog folosește backtracking este un mare ajutor. Vom lucra cu predicate care au mai multe soluții (și care se numesc predicate nedeterministe).

1. Sa dă șirul $a_1...a_n$ format din numere întregi distincte. Se cere să se afișeze toate submulțimile cu aspect de ”vale” (o secvență se spune că are aspect de vale dacă elementele descresc până la un moment dat, apoi cresc).

   - O primă variantă – foarte ineficientă – ar fi să rezolvăm problema folosind 2 predicate: unul care generează toate submulțimile și altul care verifică dacă o submulțime are sau nu aspect de vale. Problema este că o listă are foarte multe submulțimi diferite (mai ales că în cazul nostru ordinea elementelor în submulțime contează) și s-ar putea ca o mare parte dintre ele să nu aibă aspect de vale.
     De exemplu, să considerăm lista [5,4,3,6,7,-1,-2,-3]. Cateva 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. (sunt în total 2778 de submulțimi cu aspect de vale si in jur de 109601 de submultimi).
     Dacă avem submulțimea [5,3,7] (care este o vale) la această submulțime nu mai putem adăuga o serie de elemente din lista originală (4,-1,-2,-3) deci nu e nevoie să mai continuăm să generăm subliste adăugând aceste elemente.
   - De aceea pentru a avea o soluție mai eficientă, va trebui să combinăm generarea submulțimilor cu verificare. Vom folosi o listă candidat, care reprezintă un candidat de soluție, o soluție parțială. Vom adăuga elemente pe rând în această listă candidat, dar numai elemente cu care putem completa soluția noastră ca să avem o vale.
   - Pentru a genera o listă cu aspect de vale avem nevoie de o secvență de elemente descrescătoare urmat 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 arată direcția pe care suntem:
     - Valoare 0 înseamnă că suntem pe partea descrescătoare
     - Valoare 1 înseamnă că suntem pe partea crescătoare
   - Lista candidat este de fapt o listă colectoare. În general când adăugăm elemente într-o listă candidat trebuie să le adăugăm la finalul listei, deci ne trebuie predicatul adaugaSf. În cazul nostru putem evita folosirea predicatului adaugaSf, dacă adăugăm elemente la începutul

listei candidat. Dar asta presupune să generăm elementele în ordinea inversă. De exemplu soluția [5,4,3,6,7] va fi generată așa:

  - [7]
  - [6,7]
  - [3,6,7]
  - [4,3,6,7]
  - [5,4,3,6,7]

- Când adaug un element nou în lista candidat va trebui să țin cont de 2 valori: direcția și primul element din lista candidat (care este practic ultimul element adăugat în lista candidat)
  - 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 => încep să generez 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 pot adăuga, așa nu pot face vale (ex. 4, [7,6,5,8]).
- Când este o listă candidat o soluție? Ca să am o listă cu aspect de vale, îmi trebuie și secvență descrescătoare și secvență crescătoare. Dintre aceste secvențe prima dată generez cea crescătoare, deci dacă lista candidat are o secvență descrescătoare (adică parametrul pentru direcție a devenit 0), am o soluție. Dar în același timp, e posibil ca această listă candidat să fie extinsă să genereze alte soluții.
- Cum generez elementele care vor fi adăugate în lista candidat? Îmi trebuie un predicat care să îmi dea un element din lista originală, care poate este adăugat în candidat, poate nu (în funcție de condițiile discutate mai sus), după care să-mi dea un alt element, etc. Este primul predicat nedeterminist de care avem nevoie, unul care îmi dă pe rând toate elementele dintr-o listă.

$$candidat(l_1 l_2 \dots l_n) = \begin{matrix} 1.\ l_1 \\ 2.\ candidat(l_2 \dots l_n) \end{matrix}$$

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

- După ce am generat un element din listă, trebuie să verific 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 genereaza 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)

$$generare(L, c_1 \dots c_n, D)$$

$$= \begin{array}{l} 1.\, c_1 \dots c_n,\ dacă\ D = 0 \\ 2.\, generare(L, E \cup c_1 \dots c_n, 0), dacă\ E = candidat(L), E \notin c_1 \dots c_n, E > c_1\ şi\ D = 0 \\ 3.\, generare(L, E\ \cup c_1 \dots c_n, 1), dacă\ E = candidat(L), E \notin c_1 \dots c_n, E < c_1\ şi\ D = 1 \\ 4.\, generare(L, E \cup c_1 \dots c_n, 0), dacă\ E = candidat(L), E\ \notin c_1 \dots c_n, E > c_1\ şi\ D = 1 \end{array}$$

```
%generare(L:list, C:list, Directie:intreg, R:list)
%model de flux (i,i,i,o)
% L - lista initiala din care vom folosi elementele
% C - solutie candidat, in aceasta lista construim element cu element solutia
% Directie - 0, daca sunt pe partea de scadere, 1 pt partea care creste
% R - rezultat

generare(_, Cand, 0, Cand).
generare(L, [H|Cand], 0, 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),
      not(candidat([H|Cand], E)),
      E < H,
      generare(L, [E,H|Cand], 1, R).
generare(L, [H|Cand], 1, R):-
      candidat(L, E),
      not(candidat([H|Cand], E)),
      E > H,
      generare(L, [E,H|Cand], 0, 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. Dacă tot va trebui să mai scriem un predicat pentru primul apel, în acest predicat vom genera primele 2 elemente din candidat, și apelăm generare doar dacă aceste 2 elemente sunt în ordine potrivită.

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

```
%start(L:list, Rez: list)
%model de flux (i,o) (i,i)
%L - lista initiala
%Rez - o solutie
start(L, Rez):-
	candidat(L, E1),
	candidat(L, E2),
	E1 < E2,
	generare(L, [E1,E2], 1, Rez).
```

- Predicatul start generează sublistele 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.

$$startAll(l_1 l_2 \dots l_n) = \bigcup start(l_1 l_2 \dots l_n)$$

```
%startAll(L:list, Rez: list)
%model de flux (i,o) (i,i)
%L - lista initiala
%Rez - o solutie
startAll(L, Rez):-
	findall(R, start(L, R), Rez).
```

- Cum ar trebui modificată soluția noastră dacă vrem ca în soluție să avem elementele în aceeași ordine ca în lista originală?
- 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, ca să returneze și această listă.

$$candidat2(l_1 \dots l_n, E, R) = \begin{matrix} 1.\ E = l_1 şi\ R = \emptyset \\ 2.\ candidat2(l_2 \dots l_n, E, l_1 \cup R) \end{matrix}$$

```
%candidat2(L:list, E:element, R:list)
%model de flux (i,o,o), (i,i,i), (i,o,i), (i,i,o)
%L - lista initiala
%E - un element din lista L
%R - lista pana la elementul E
candidat2([E|_], E, []).
candidat2([H|T], E, [H|R]):-
	candidat2(T,E,R).
```

- candidat2([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]
- Dacă folosim predicatul candidat2, la predicatul generare apelul recursiv se face cu lista returnată de candidat2, nu cu lista originală L.