Tăierea de backtracking

1. Obiective

În lucrarea de față vom prezenta o metodă de îmbunătățire a performanței unui predicat prin reducerea spațiului de căutare. De asemenea vom continua cu operații pe liste folosind cele 2 tipuri de recursivitate: înainte și înapoi.

2. Considerații teoretice

2.1. Operatorul „!”

Operatorul **!** de tăiere este o proprietate a Prolog-ului care șterge ramurile alternative de căutare a soluției, astfel acele ramuri nu sunt explorate de backtracking. O dată ce s-a trecut peste operator, apelurile din fața operatorului și predicatul inițial *NU* mai pot căuta o altă clauză cu care să se unifice, deci nu mai pot căuta o altă soluție.

Acest operator poate îmbunătăți eficiența programelor Prolog; însă, predicactele care conțin „!” sunt mai greu de urmărit.

În caz general putem scrie o clauză care folosește operatorul de tăiere în felul următor:

p :- a_1, a_2, …, a_k, !, a_{k+1}, …, a_n.

Daca a_{k+1} eșuează, nu se va mai căuta o altă clauză care să se unifice cu a_k, a_{k-1}, …, a_1 și p. Un caz simplu de utilizare ar fi atunci când avem 2 clauze cu apeluri inițiale mutual exclusive.

Exemplu:

```
% Varianta 1 fără !
p(X) :- X>0, a, … .
p(X) :- X=<0, b, … .

% Varianta 2 cu !
p(X) :- X>0, !, a, … .
p(X) :- b, … .
```

În concluzie, nodul p și nodurile a_1, …, a_k nu pot să facă backtrack. Ce trebuie să retineți despre operatorul de tăiere de backtracking este:

- Tăierea de backtracking se termina mereu cu succes
- Nicio clauză apelată înainte de acest operator nu va fi reconsiderată
- Nicio clauză ulterioară la începutul regulii curente nu va fi reconsiderată
- Orice variabilă unificată înainte de operator nu va putea lua alte valori

O potentiala aplicatie a operatorului „!" reiese din acestea. Amintiți-vă exercițiul din laboratorul anterior. Vom folosi ca exemplu predicatul separate_parity/3:

```
separate_parity([], [], []).
separate_parity([H|T], [H|E], O):-
   0 is H mod 2,
   separate_parity(T, E, O).
separate_parity([H|T], E, [H|O]):-
   separate_parity(T, E, O).
```

Dacă interogăm acest predicat, primul răspuns va fi corect, iar următoarele răspunsuri vor muta elemente pare în lista celor impare prin backtracking.

De exemplu, la prima întrebare a *?-separate_parity([1,2,3,4], E, O)*, 4 a trecut prin *0 is H mod 2* și a fost evaluat ca true. Dacă repetăm întrebarea, prin backtracking, va fi evaluat ca false și astfel, va fi mutat în a treia clauză unde 4 va fi adăugat la lista celor impare.

```
?- separate_parity([1,2,3,4], E, O).
E = [2, 4],
O = [1, 3];
E = [2],
O = [1, 3, 4];
…
```

Cum putem opri acest comportament? O primă opțiune este să adăugăm condiția pentru a treia clauză, dar care? Condiția *opusă*, negarea lui *0 is H mod 2*, care în acest caz ar fi 1 is H mod 2.

```
separate_parity([], [], []).
separate_parity([H|T], [H|E], O):-
   0 is H mod 2,
   separate_parity(T, E, O).
separate_parity([H|T], E, [H|O]):-
   1 is H mod 2,
   separate_parity(T, E, O).
```

```
?- separate_parity([1,2,3,4], E, O).
E = [2, 4],
O = [1, 3];
false
```

Prin adăugarea acestei condiții, primul răspuns va rămâne cel corect, dar la repetarea întrebării vom primi drept răspuns *false*. Această modificare transformă predicatul dintr-unul *nondeterminist* într-unul **determinist**.

Același efect poate fi obținut prin adăugarea operatorului „!” dupa condiția *0 is H mod 2* (condiția opusă nemaifiind necesară în a treia clauză) :

```
separate_parity([], [], []).
separate_parity([H|T], [H|E], O):-
   0 is H mod 2, !,
   separate_parity(T, E, O).
separate_parity([H|T], E, [H|O]):-
   separate_parity(T, E, O).
```

```
?- separate_parity([1,2,3,4], E, O).
E = [2, 4],
O = [1, 3].
```

Urmează să reluăm predicate din laboratorul anterior: *member/2* și *delete/3*. După cum ați observat, ambele predicate permit comportament nondeterminist. Prin adiția operatorului „!”, putem să le transformăm în predicate deterministe.

2.1.1. Predicatul „member” cu tăiere de backtracking

Versiunea determinstă a predicatului *member/2*:

```
member1(X, [X|_]) :- !.
member1(X, [_|T]) :- member1(X, T).
```

Exemplu de urmărire a execuției (cu trace):
[trace] ?- member1(X,[1,2,3]).
  Call: (7) member1(_G1657, [1, 2, 3]) ?
  Exit: (7) member1(1, [1, 2, 3]) ?
X = 1. **% nu mai putem repeta întrebarea pentru că nu mai poate să caute o altă clauză cu care să se unifice**

După cum se poate observa din urmărirea apelului, tăierea de backtracking nu permite nodului corespunzător apelului de *member1(_G1657,[1, 2, 3])* să fie unificat cu alte clauze și variabila _G1657 să fie unificată cu altă valoare. Astfel, când repetăm întrebarea, primim fail.

Urmăriți execuția la:
?- X=3, member1(X, [3, 2, 4, 3, 1, 3]).
?- member1(X, [3, 2, 4, 3, 1, 3]).

2.1.2. Predicatul „delete” cu tăiere de backtracking

Predicatul delete/3 din lucrarea anterioară ștergea o apariție (defapt, chiar prima) a unui element (primul argument) la un apel. Cum am face dacă am vrea versiunea determinstă acestui predicat, predicatul care șterge prima si doar prima apariție a unui element în listă? Această versiune a predicatului delete este prezentată mai jos:

```
delete1(X, [X|T], T) :- !.
delete1(X, [H|T], [H|R]) :- delete1(X, T, R).
delete1(_, [], []).
```

Exemplu de urmărire a execuției (cu trace):
[trace] ?- delete1(3,[4,3,2,3,1],R).
  Call: (7) delete1(3, [4, 3, 2, 3, 1], _G1701) ? **% se unifică cu clauza 2 și apoi apel recursiv**
  Call: (8) delete1(3, [3, 2, 3, 1], _G1786) ? **% se unifică cu clauza 1**
  Exit: (8) delete1(3, [3, 2, 3, 1], [2, 3, 1]) ? **% succes și iese din apelul recursiv**

Exit: (7) delete1(3, [4, 3, 2, 3, 1], [4, 2, 3, 1]) ?
R = [4, 2, 3, 1] ; **% repetăm întrebare**
Redo: (7) delete1(3, [4, 3, 2, 3, 1], _G1701) ? **% nu se poate anula unificarea cu clauza 1 și atunci se urca în stiva, la apelul precedent, se anulează unificarea cu clauza 2 și se încearcă unificarea cu clauza 3**
Fail: (7) delete1(3, [4, 3, 2, 3, 1], _G1701) ? **% eșec**
false.

Astfel, această versiunea a predicatului delete șterge doar prima apariție a elementului.

Urmăriți execuția la:
?- delete1(X, [3, 2, 4, 3, 1, 3], R).

2.2. Predicatul „length”

Predicatul *length(L,R)* calculează numărul de elemente din lista *L* și pune rezultatul în *R*. Recurența matematică poate fi formulată astfel:

$$lungime(L) = \begin{cases} 0, & L = [] \\ 1 + lungime\big(coada(L)\big), & altfel \end{cases}$$

Sau, redactat în cuvinte:

- Lungimea listei vide este 0
- Lungimea unei liste nonvide [H|T] este lungimea lui T plus 1

În Prolog se va scrie:

```
% Varianta 1 (recursivitate înapoi)
length1([], 0).
length1([_|T], Len) :- length1(T, Lcoada), Len is 1+Lcoada.
```

Această versiune a predicatului length folosește o abordare de recursivitate înapoi. Astfel, rezultatul la nivelul *i*, are nevoie de rezultatul de la nivelul *i-1*. Lungimea (al doilea argument) este inițializată când apelul recursiv ajunge să fie unificat cu condiția de oprire (prima clauză) și este construit progresiv la fiecare întoarcere dintr-un apel recursiv prin incremente de unu.

```
% Varianta 2 (recursivitate înainte -> acumulator = al doilea argument)
length2([], Acc, Len) :- Len=Acc.
length2([_|T], Acc, Len) :- Acc1 is Acc + 1, length2(T, Acc1, Len).

length2(L, R) :- length2(L, 0, R).
% length2/2 = wrapper al predicatului length2/3 care folosește un acumulator
```

Cealaltă abordare este să numărăm elementele din lista în timp ce lista este descompusă (prin șablonul [H|T]) și să construim lungimea (în acumulator) în timp ce lista este parcursă prin recursivitate. Această abordare este cea de recursivitate înainte. Pentru a folosi această abordare, acumulatorul trebuie să fie inițializat cu 0 la fiecare întrebare. Cu fiecare element nou descoperit, lungimea crește cu 1. Pentru a face rezultatul disponibil, avem nevoie să unificăm acumulatorul cu o varibilă neinstanțiată (*al treilea argument*) care a fost pasat prin toate apelurile recursive fără a fi instanțiată.

Retineți. În contrast cu alte limbaje de programare, în Prolog expresiile nu sunt evaluate implicit. Pentru a evalua o expresie trebuie să folosim operatorul **is.** Mod de folosire: <Variabilă> is <expresie>.

Urmăriți execuția la:

```
?- length1([a, b, c, d], Len).
?- length1([1, [2], [3|[4]]], Len).
?- length2([a, b, c, d], 0, Res).
?- length2([a, b, c, d], Len).
?- length2([1, [2], [3|[4]]], Len).
?- length2([a, b, c, d], 3, Len).
```

2.3. Predicatul „reverse”

Predicatul *reverse(L,R)* inversează ordinea elementelor din lista *L*. Recurența matematică poate fi formulată astfel:

$$invers(L) = \begin{cases} [], & L = [] \\ invers\big(coada(L)\big) \oplus \{primul(L)\}, & altfel \end{cases}$$

Sau, redactat în cuvinte:

- Inversul listei vide [] este [], și
- Inversul unei liste non-vide [H|T] poate fi obținut prin inversarea T-ului și prin adăugarea H-ului la finalul listei rezultate

În Prolog se va scrie:

```
% Varianta 1 (recursivitate înapoi)
reverse1([], []).
reverse1([H|T], R) :- reverse1(T, Rcoada), append1(Rcoada, [H], R).
```

În primul rând, obținem inversul cozii T, denominat *Rcoada* și construim inversul listei L=[H|T], prin concatenarea primul element H la finalul lui *Rcoada*.

```
% Varianta 2 (recursivitate înainte -> acumulator = al doilea argument)
reverse2([], Acc, R) :- Acc=R.
reverse2([H|T], Acc, R) :- Acc1=[H|Acc], reverse2(T, Acc1, R).

reverse2(L, R) :- reverse2(L, [], R).
% reverse2/2 = wrapper a predicatului reverse2/3 care
% folosește un acumulator
% În contrast cu acumulatoarele de până acum, aici avem operații cu liste,
% astfel va fi instanțiată cu o listă vidă
```

În a doua clauză, elementele vor fi adăugate în fața acumulatorului când sunt descoperite. Acest fapt poate să pară contra-intuitiv, ne vom uita la un exemplu:

```
?- reverse2([1,2,3], [], R).

reverse2([1,2,3], [], R).
        Acc1 = [1 | []] = [1]
        reverse2([2,3], [1], R).
                Acc1 = [2 | [1]] = [2,1]
                reverse2([3], [2,1], R).
                        ...
```

Astfel, când lista de intrare devine lista vidă (prima clauză), lista inversată este deja în acumulator. Ultimul lucru care trebuie făcut este unificarea acumulatorului (clauza 1) cu variabila (până atunci) neinstanțiată (al treilea argument).

Urmăriți execuția la:
?- reverse1([a, b, c, d], R).
?- reverse1([1, [2], [3|[4]]], R).
?- reverse2([a, b, c, d], R).
?- reverse2([1, [2], [3|[4]]], R).
?- reverse2([a, b, c, d], [1, 2], R).

2.4. Predicatul minim

Predicatul *min(L,M)* determină elementul minim din lista *L*. Dacă lista este vidă, va returna *false*. Recurența matematică poate fi formulată astfel:

$$min(L) = \begin{cases} min\big(coada(L)\big), & min\big(coada(L)\big) < primul(L) \\ primul(L), & altfel \end{cases}$$

În Prolog se va scrie:

```
% Varianta 1 (recursivitate înainte -> acumulator = al doilea argument)
min1([], Mp, M) :- M=Mp.
min1([H|T], Mp, M) :- H<Mp, !, min1(T, H, M).
min1([_|T], Mp, M) :- min1(T, Mp, M).

min1([H|T], M) :- min1(T, H, M).
% În contrast cu acumulatoarea folosite până acum
% pentru predicatul min1/3,
% acumulatorul (al doilea argument) va fi inițializat cu primul element
% Într-un mod similar, min1/2 este un wrapper.
```

O primă soluție pentru determinarea elementului minim într-o listă este să parcurgem elementele unul câte unul și să păstrăm, la fiecare pas, elementul minim curent. Când lista devine vidă, minimul parțial devine minim global. Această abordare corespunde unei strategii de recursivitate înainte.

Ultimele 2 clauze a predicatului traversează lista: a doua clauză acoperă cazul în care minimul parțial trebuie să fie actualizat (un nou minim parțial a fost găsit ca primul element curent), pe când în ultima clauză minimul este pasat mai

departe fără să fie modificat. Prima clauză reprezintă condiția de oprire: lista devine vidă, astfel minimul parțial este unificat cu variabila (până atunci) neinstanțiată (al treilea argument).

```
% Varianta 2 (recursivitate înapoi)
min2([H], H).
min2([H|T], M):-min2(T, M), H>=M.
min2([H|T], H):-min2(T, M), H<M.
```

Această abordare este de recursivitate înapoi, minimul este actualizat la întoarcerea din recursivitate (clauzele 2-3 actualizează minimul la întoarcere). Minimul este inițializat în condiția de oprire (clauza 1) cu ultimul element. *Rețineți*: nu este nevoie de un argument adițional (ca la recursivitatea înainte).

Studiați (folosind trace) execuția la:
?- min2 ([1, 2, 3, 4], M).
?- min2([4, 3, 2, 1], M).
?- min2 ([3, 2, 6, 1, 4, 1, 5], M).
?- min2([], M).
Repetați întrebările. Câte răspunsuri are fiecare întrebare? Care este ordinea răspunsurilor? (dacă se aplică).

Predicatul *min2/2* poate fi îmbunătățit prin următoarele considerente:

- Cele două condiții din clauzele 2 și 3 (H<M și H>=M) sunt complementare
- Deoarece actualizarea minimul se face la întoarcerea din recursivitate, nu este niciun scop în a descompune lista iar când condiția din a doua clauză va da fail; este suficient să actualizăm minimul până în acel punct, astfel predicat poate fi îmbunătățit:

```
min2([H], H).
min2([H|T], M):- min2(T, M), H>=M, !.
min2([H|T], H).
```

- Prin analiza clauzelor 1 și 3, putem observa că acestea se pot combina într-o singură clauză, astfel simplificând predicatul. Este important să observăm că această clauză combinată trebuie să fie plasată după a doua clauză curentă. De ce? Versiunea finală a predicatului minim folosind recursivitate înapoi:

```
min2([H|T], M) :- min2(T, M), M<H, !.
min2([H|_], H).
```

Urmăriți execuția la:
?- min1([], M).
?- min1([3, 2, 6, 1, 4, 1, 5], M).
?- min2([], M).
?- min2([3, 2, 6, 1, 4, 1, 5], M).

Puteți identifica diferența dintre implementarea inițială si cea finală a predicatului?

Observație. Predicatul *min1/3* care folosește recursivitate înainte inițializează minimul ca primul element în wrapper, pe când la recursivitatea înapoi minimul este inițializat cu ultimul element. Există o diferență totuși, până acum am văzut unificarea cu o condiție de oprire la care se ajunge când o condiție este adevărată (ex. un counter devine 0, o listă devine []). În cazul minimului, când lista ajunge la listă vidă va da fail și nu se va putea unifica cu niciuna din clauze și doar prin backtracking va ajunge să se unifice cu a doua clauză și astfel inițializează minimul (al doilea argument).

2.5. Operații pe mulțimi

Vom reprezenta o mulțime folosind o listă fără elemente duplicate. Reuniunea între 2 mulțimi poate fi realizată prin concatenarea listei a doua cu elementele din prima listă care nu apar în a doua listă. Recurența matematică poate fi formulată astfel:

$$L_1 \cup L_2 = \begin{cases} \{primul(L_1)\} \oplus (coada(L_1) \cup L_2), & primul(L_1) \notin L_2 \\ coada(L_1) \cup L_2, & altfel \end{cases}$$

În Prolog se va scrie:

```
union([], L, L).
union([H|T], L2, R) :- member(H, L2), !, union(T, L2, R).
% Prin folosirea predicatului member și recursivitate, putem verifica fiecare
% element (H) a listei L1 dacă este un element și a L2:
% dacă este -> nu va fi adăugat în rezultatul R
% Altfel, îl adăugăm in R.
union([H|T], L2, [H|R]) :- union(T, L2, R).
```

Urmăriți execuția la:
?-union([1,2,3], [4,5,6], R).
?-union([1,2,5], [2,3], R).

?-union(L1,[2,3,4],[1,2,3,4,5]).

3. Exerciții

1. Scrieți predicatul *intersect(L1,L2,R)* care realizează intersecția între două mulțimi.
   Sugestie: Gândiți-vă la cum a fost implementat predicatul union. Rețineți – vom lua doar elementele care apar în ambele liste.

```
?- intersect([1,2,3], [1,3,4], R).
R = [1, 3] ;
false
```

2. Scrieți predicatul *diff(L1,L2,R)* care realizează diferența între două mulțimi (elementele care apar în prima mulțime și nu apar în a doua mulțime).
   Sugestie: Gândiți-vă la cum a fost implementat predicatul union și *intersect*. Rețineți – vom lua doar elementele care apar în prima listă și nu și in a doua.

```
?- diff([1,2,3], [1,3,4], R).
R = [2] ;
false
```

3. Scrieți predicatele *del_min(L,R)* și *del_max(L,R)* care șterg **toate** aparițiile minimului, respectiva ale maximului din lista *L*.

```
?- del_min([1,3,1,2,1], R).
R = [3, 2] ;
false

?- del_max([3,1,3,2,3], R).
R = [1, 2] ;
false
```

Exercițiu greu: Încercați să implementați fiecare din aceste predicate folosind o singură traversare a listei *(del_min1/2* și *del_max1/2).*

4. Scrieți un predicat care inversează ordinea elementelor dintr-o listă începând cu al K-lea element.

```
?- reverse_k([1, 2, 3, 4, 5], 2, R).
R = [1, 2, 5, 4, 3] ;
false
```

5. Scrieți un predicat care codifică șirul de elemente folosind algoritmul RLE (Run-length encoding). Un șir de elemente consecutive și egale se vor înlocui cu perechi **[element, număr de apariții].**

```
?- rle_encode([1, 1, 1, 2, 3, 3, 1, 1], R).
R = [[1, 3], [2, 1], [3, 2], [1, 2]] ;
false
```

Opțional. Puteți modifica acest predicat astfel încât dacă un element este singular (nu are valori consecutive egale), să păstram acel element în loc să adăugam o pereche?

```
?- rle_encode2([1, 1, 1, 2, 3, 3, 1, 1], R).
R = [[1, 3], 2, [3, 2], [1, 2]] ;
false
```

6. Scrieți un predicat care rotește o listă K poziții la dreapta.

   Sugestie: încercați să implementați rotate_left mai întâi, este mai ușor.

```
?- rotate_right([1, 2, 3, 4, 5, 6], 2, R).
R = [5, 6, 1, 2, 3, 4] ;
false
```

7. Scrieți un predicat care extrage aleatoriu K element din lista L și le pune în lista rezultat R.

   Sugestie: folosiți predicatul predefinit *random_between/3(min_value, max_value, result)*.

```
?- rnd_select([a, b, c, d, e, f, g, h], 3, R).
R = [e, d, a] ;
false
```

8. Scrieți un predicat care decodifică șirul de elemente folosind algoritmul RLE (Run-length encoding). O pereche **[element, număr de apariții]** va fi înlocuită de un șir de elemente consecutive și egale.

```
?- rle_decode([[1, 3], [2, 1], [3, 2], [1, 2]], R).
R = [1, 1, 1, 2, 3, 3, 1, 1];
false
```

Observație. Nu uitați că toate predicatele returnează răspunsul corect la prima intrebare, iar la repetarea întrebarii -> primim rezultatul: **false**.