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 **!** șterge ramurile alternative de căutare a soluției. 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.

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

```
p :- a1, a2, …, ak, !, ak+1, …, an.
```

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, … .
```

2.1.1 Predicatul „member” cu tăiere de backtracking

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

Exemplu de urmărire a execuției (cu trace):

```
[trace] 12 ?- 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
```

Urmărește 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

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

Exemplu de urmărire a execuției (cu trace):

```
[trace] 15 ?- 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.
```

Urmărește 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:

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

În Prolog se va scrie:

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

% Varianta 2 (recursivitate înainte)
length2([], Acc, Len) :- Len=Acc.
length2([H|T], Acc, Len) :- Acc1 is Acc + 1, length2(T, Acc1, Len).
length2_pretty(L, R) :- length2(L, 0, R).
```

Urmărește execuția la:

```
?- length1([a, b, c, d], Len).
?- length1([1, [2], [3|[4]]], Len).
?- length2([a, b, c, d], 0, Res).
?- length2_pretty([a, b, c, d], Len).
?- length2_pretty([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:

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

În Prolog se va scrie:

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

% Varianta 2 (recursivitate înainte)
reverse2([], Acc, R) :- Acc=R.
reverse2([H|T], Acc, R) :- Acc1=[H|Acc], reverse2(T, Acc1, R).
reverse2_pretty(L, R) :- reverse2(L, [], R).
```

Urmărește execuția la:

```
?- reverse1([a, b, c, d], R).
?- reverse1([1, [2], [3|[4]]], R).
?- reverse2_pretty([a, b, c, d], R).
?- reverse2_pretty([1, [2], [3|[4]]], R).
?- reverse2([a, b, c, d], [1, 2], R).
```

2.4 Predicatul minim

Predicatul *minim(L,M)* determina 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 înapoi)
min1([H|T], M) :- min1(T, M), M<H, !.
min1([H|_], H).
```

```
% Varianta 2 (recursivitate înainte)
min2([], Mp, M) :- M=Mp.
min2([H|T], Mp, M) :- H<Mp, !, min2(T, H, M).
min2([H|T], Mp, M) :- min2(T, Mp, M).
min2_pretty([H|T], M) :- min2(T, H, M). % la început inițializăm
minimul cu primul element
```

Urmărește execuția la:

```
?- min1([], M).
?- min1([3, 2, 6, 1, 4, 1, 5], M).
?- min2_pretty([], M).
?- min2_pretty([3, 2, 6, 1, 4, 1, 5], M).
```

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).
union([H|T], L2, [H|R]) :- union(T, L2, R).
```

Urmărește 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 *inters(L1,L2,R)* care realizează intersecția între două mulțimi.
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).
3. Scrieți predicatele *del_min(L,R)* și *del_max(L,R)* care șterg toate aparițiile minimului, respectiva ale maximului din lista *L*.
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
```

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

```
?- 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 funcția random(valoare_maxima).*

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