Backtracking în Prolog

Backtracking este practic o formă de căutare.

- În contextul Prolog, să presupunem că interpretul Prolog încearcă să satisfacă o secvență de obiective goal_1, goal_2.
- Când interpretul Prolog găsește un set de legături variabile care permit îndeplinirea obiectivului_1, se angajează să respecte acele legături și apoi încearcă să satisfacă obiectivul_2.
- În cele din urmă, se întâmplă unul din două lucruri:
    1. obiectivul_2 este satisfăcut și s-a încheiat;
    2. obiectivul_2 nu poate fi îndeplinit.
    - În ambele cazuri, Prolog dă înapoi. Adică, se „renunță” la legările variabilelor pe care le-a făcut în satisfacerea obiectivului_1 și caută un set diferit de legături variabile care să permită îndeplinirea obiectivului_1.
    - Dacă găsește un al doilea set de astfel de legături, se angajează față de ele și continuă să încerce să satisfacă obiectivul_2 din nou, cu noile legături.
    - În cazul 1., interpretul Prolog caută soluții suplimentare, în timp ce în cazul 2. încă caută prima soluție.
    - Deci, backtracking poate servi pentru :
    1. a găsi soluții suplimentare la o problemă
    2. pentru a continua căutarea unei prime soluții, atunci când un prim set de ipoteze (adică legături variabile) se dovedește a nu conduce la o soluție.

Definiția predicatului built-in Prolog: member

```
1 member(X, [X|_]).
2     % X is a member if its the first element
3 member(X, [_|Rest]) :-
4     member(X, Rest).   % otherwise, check if X is in the Rest
```

Este posibil să nu vă gândiți la **member** ca un predicat de backtracking, dar backtracking este încorporat în Prolog, așa că, în circumstanțe adecvate, **member** va face backtracking:

```
?- member(X, [a, b, c]).
```

Aici, **member** revine pentru a găsi toate soluțiile posibile la interogarea care îi este dată. Luați în considerare și:

```
?- member(X, [a, a, a]).
```

Aici **member** revine, chiar dacă continuă să găsească același răspuns. Ce ziceti

```
?-
```

```
member(a, [a, a, a]).
```

- Termenul de backtracking se aplică și căutării mai multor seturi de legături variabile pentru a satisface un singur obiectiv.
- În unele circumstanțe, poate fi de dorit să se inhibe înapoi, ca atunci când este necesară o singură soluție.
- Scopul **cut** Prolog permite acest lucru

TRACE - Urmărirea execuției unei bucăți de cod Prolog cu **backtracking** poate fi o modalitate bună de a afla ce se întâmplă în timpul execuției.

Tăieturi în Prolog

- Tăierea (cut), în Prolog, este un goal, scris ca !, care întotdeauna reușește, dar nu se poate face backtrack înapoi.
- Este folosit pentru a preveni backtracking nedorit, de exemplu, pentru a preveni găsirea de soluții suplimentare de către Prolog.
- Tăierea trebuie folosită cu moderație.
- Există tentația de a introduce tăieturi experimental în codul care nu funcționează corect. Dacă faceți acest lucru, rețineți că, atunci când se face debug este completă, ar trebui să înțelegeți efectul fiecărei tăieturi pe care o utilizați și să puteți explica necesitatea fiecărei tăieturi.
- Utilizarea unei tăieturi ar trebui astfel comentată.

Empty markdown cell. Double click to edit

```
1 teaches(dr_fred, history).
2 teaches(dr_fred, english).
3 teaches(dr_fred, drama).
4 teaches(dr_fiona, physics).
5
6 studies(alice, english).
7 studies(angus, english).
8 studies(amelia, drama).
9 studies(alex, physics).
```

```
?- teaches(dr_fred, Course), studies(Student, Course).
```

- Backtracking nu este inhibat aici.
- Cursul este inițial legat de istorie, dar nu există studenți la istorie, așa că al doilea obiectiv eșuează, are loc backtracking,
- Cursul este re-legat la engleză, al doilea obiectiv este încercat și sunt găsite două soluții (Alice și Angus), apoi prin backtracking din nou, iar Course este legat de dramă, și este se găsește studentul Amelia.

```
?- teaches(dr_fred, Course), !, studies(Student, Course).
```

De data aceasta, Cursul este inițial legat de istorie,

- apoi obiectivul **cut** este executat,
- iar apoi obiectivul **studies** este încercat și eșuează (pentru că nimeni nu studiază istoria).
- Din cauza tăierii, nu putem reveni la obiectivul **teaches** pentru a găsi o altă legătură pentru curs, așa că întreaga interogare eșuează.

```
?- teaches(dr_fred, Course), studies(Student, Course), !.
```

- Aici scopul **teaches** este încercat ca până acum, iar cursul este legat de istorie, din nou ca până acum.
- În continuare, obiectivul studiilor este încercat și nu reușește, astfel încât să nu ajungem la tăierea de la sfârșitul interogării în acest moment și poate apărea backtracking.
- Astfel, scopul **teaches** este reîncercat, iar cursul este legat de limba engleză.
- Apoi scopul studiilor este încercat din nou, și reușește, cu Student = alice.
- După aceea, obiectivul **cut** este încercat și bineînțeles reușește, astfel încât nu mai este posibilă nicio altă întoarcere și se găsește astfel o singură soluție.

```
?- !, teaches(dr_fred, Course), studies(Student, Course).
```

În acest exemplu final, se găsesc aceleași soluții ca și cum nu ar fi fost prezentă nicio tăietură, deoarece nu este niciodată necesar să se întoarcă dincolo de tăietură pentru a găsi următoarea soluție, astfel încât backtracking-ul nu este niciodată inhibat.

Tăieturi în interiorul regulilor

- În practică, tăierea este folosită în reguli, mai degrabă decât în interogări cu mai multe obiective, iar în astfel de cazuri se aplică anumite expresii.
- Neînţelegerea mecanismului de tăiere este o sursă de greşeli chiar şi pentru programatorii experimentaţi. Greşelile pot fi de două feluri:

1. se taie ramuri ale arborelui de căutare ce nu trebuie tăiate, deoarece conţin soluţii
2. nu se taie acele ramuri ce nu conţin soluţii

- Prin urmare, există două contexte diferite în care se poate utiliza predicatul !/0:

1. Tăietură verde - predicatul se introduce numai pentru creşterea eficienţei programului
2. Tăietura roșie - modifică semnificaţia procedurală a programului

Tăietura Verde

- Folosind predicatul !/0 se poate recurge la stoparea căutărilor, dacă răspunsul deja este găsit.
- De exemplu, luăm în considerare următorul cod pentru `mai_mare(X, Y, Z)` , care se presupune că leagă Z la cel mai mare număr dintre X și Y, care se presupune că sunt și ele numere.

```
1 %clauza 1
2 mai_mare(X,X,X).
3 % clauza 2
4 mai_mare(X,Y,Y):-X<Y.
5 % clauza 3
6 mai_mare(X,Y,X):-X>Y.
```

```
?- trace, mai_mare(4,1,P).
```

- În clauza 2 apare o condiţie care ajută să se stabilească care este mai mic.
- Definiţia are o semnificaţie declarativă deosebită.
- Procedural, însă, conţine o redundanţă. Clauzele sunt mutual exclusive, dar a doua şi a treia conţin câte o condiţie de comparare în calitate de subscopuri.
- De aceea, dacă a doua clauză reuşeşte, o eventuală tentativă de resatisfacere va apela a treia clauză, dar subscopul respectiv va eşua. Astfel, tentativa este inutilă.
- Această redundanţă nu este relevantă, dar devine semnificativă când se consideră în contextul altui predicat, cum ar fi predicatul `maximum(X,L)` : “X este elementul maximum al listei de numere L”.

```
1 maximum(M,[M]).
2 maximum(Min,[T1,T2|C]):-
3     maximum(M,[T2|C]), mai_mare(T1,M,Min).
```

- În predicatul mai_mare/3 se poate evita redundanţa, utilizând pentru a confirma clauza:

```
1 mai_mare(X,X,X):-!.
2 mai_mare(X,Y,Y):-X<Y,!.
3 mai_mare(X,Y,X):-X>Y.
```

- Astfel, relaţia este constituită dintr-o mulţime de clauze, fiecare din ele tratând un tip particular de intrări
- În aceste circumstanţe se poate utiliza **tăitura verde** pentru a semnaliza sistemului despre inaplicabilitatea alternativelor clauzei selectate
- Dacă aceste condiţii sunt satisfăcute, clauza care le conţine va fi considerată ultima din pachet, chiar dacă sunt şi altele, şi chiar dacă subscopurile, ce urmează după tăiere, eşuează.

```
?- trace, mai_mare(4,1,P).
```

```
?- trace, mai_mare(4,1,P).
```

Tăietura roșie

- adăugarea în programele de mai sus a predicatului de tăiere nu afectează mulţimea de soluţii.
- Tăierea **verde** înlătură ramurile arborelui care nu duc spre găsirea noilor soluţii => tăierea verde este o expresie a **determinismului**.

- Iar tăierile ce nu sunt verzi, se numesc **roşii**.

Putem observa ca pentru predicatul **mai_mare/3** din secţiunea precedentă, poate fi satisfăcută doar o condiţie din X=Y, X>Y şi X<Y pentru orice valori ale lui X şi Y

```
1 mai_mare(X,X,X):-!.
2 mai_mare(X,Y,Y):-X<Y,!.
3 mai_mare(X,Y,X).
```

Mai succint,

```
1 mai_mare(X,Y,Y):-X<=Y,!.
2 mai_mare(X,_,X).
```

Dacă X este mai mic sau egal cu Y, atunci Y e mai mare; în caz contrar mai mare e X.

=> Adică nu e nevoie de compararea lui X şi Y în a doua clauză.

```
1 mai_mare(X,Y,Z):-X<=Y,!,Z=Y.
2 mai_mare(X,Y,X).
```

```
1 mai_mare(X,Y,Z):-X<=Y,!,Z=Y.
2 mai_mare(X,Y,X).
```

```
?- mai_mare(1,4,1).
```

- Interogarea de mai sus reușește, deci ultima clauză are, de fapt, o altă interpretare.
- Utilizarea tăierii, de fapt, a substituit prezenţa unei condiţii, din punct de vedere declarativ, esenţiale.
- Pe lângă aceasta, efectul dorit există, numai dacă se respectă aceeaşi ordine a clauzelor.
- Tăierea prezentă este **roşie**.

Tăietura verde => tăietura roșie

- În anumite cazuri efectul introducerii predicatului !/0 poate fi mai subtil: => tăierea inclusă în program drept verde, adică pentru ridicarea eficienţei, este de fapt roşie.
  De exemplu, fie predicatul cu mai multe moduri de utilizare, cum ar fi predicatul apartine/2:

```
1 eliminare(Elem,[Elem|Coada],Rez):- eliminare(Elem,Coada,Rez).
2 eliminare(Elem,[Cap|Coada1],[Cap|Coada2]):-
3           Elem=\=Cap,
4    eliminare(Elem,Coada1,Coada2).
```

```
5 eliminare(_,[],[]).
```

```
?- eliminare(1,[1,2,3,1,1],P).
```

```
1 eliminare(Elem,[Elem|Coada],Rez):-
2     !,eliminare(Elem,Coada,Rez).
3 eliminare(Elem,[Cap|Coada1],[Cap|Coada2]):-
4           Elem=\=Cap,!, eliminare(Elem,Coada1,Coada2).
5
6 eliminare(_,[],[]).
```

```
?- eliminare(1,[1,2,3,1,1],P).
```

Toate tăieturile sunt verzi.

- prezenţa tăierii în prima clauză exclude alternativa exprimată de clauza a doua => verificarea explicită a inegalităţii din corpul clauzei doi este în plus.
- Caracterul mutual eliminatoriu al acestor două condiţii este exprimat de cele două predicate !/0 => coincide sau nu capul listei cu elementul destinat eliminării.

O altă variantă poate fi:

```
1 eliminare(Elem,[Elem|Coada],Rez):-
2     !,eliminare(Elem,Coada,Rez).
3 eliminare(Elem,[Cap|Coada1],[Cap|Coada2]):-
4           !,eliminare(Elem,Coada1,Coada2).
5 eliminare(_,[],[]).
```

```
?- eliminare(1,[1,2,3,1,1],P).
```

DAR Tăietura din clauza întâi este roşie, iar din clauza a doua este verde.

- Tăietura roșie funcționează doar dacă se respectă o anumită ordine de executare a clauzelor.
- Tăietura roșie îngreunează lizibilitatea codului.

Revenind la exemplul implementării predicatului member/2 (/pldoc/man?predicate=member/2) din Prolog, prezentat la începutul acestui notebook:

```
1 member(X, [X|_]).
2     % X is a member if its the first element
3 member(X, [_|Rest]) :-
4     member(X, Rest).   % otherwise, check if X is in the Rest
```

Putem îmbunătăți aceast predicat:

```
member(X, [X|_]):-!.
    % X is a member if its the first element
member(X, [_|Rest]) :-
    member(X, Rest).   % otherwise, check if X is in the Rest
```

```
?- member(Elem,[1,2,3]).
```

Tăietura din prima clauză, deși se dorea să fie un **verde**, care să eficientizeze implemnetarea predicatului, se dovedește a fi una **roșie**, deoarece schimbă comportamentul programului:

Interpretare

- Semnificația **declarativă** a predicatului `apartine(E,L)` este “Elementul E aparţine listei L”, independent cum sunt instanţiate argumentele.
- Semnificaţia **procedurală**, invers, depinde de instanţierea argumentelor şi este diferită în cele două versiuni. Versiunea fără tăiere are semnificaţia procedurală: “Să se verifice apartenenţa unui element unei liste, dacă ambii sunt date sau să se furnizeze elementele unei liste date”. Versiunea cu tăiere, dimpotrivă, are semnificaţia procedurală: “Să se verifice apartenenţa unui element unei liste, dacă ambii sunt date sau să se furnizeze primul element al listei date”. Deci, tăierea în această definiţie este roşie.

Concluzii

- ! acționează ca un marker, dincolo de care prologul nu va merge. Toate alegerile făcute pentru a tăia sunt stabilite și tratate ca și cum acestea ar fi doar căile posibile.
- Dacă un utilizator ajunge să se deplaseze departe în clauză, nu trebuie să se întoarcă și să încerce o altă alegere pentru a demonstra obiectivul sau să încerce o altă modalitate de a satisface sub-obiectivele care au fost dovedite pentru acest obiectiv, deoarece obiectivul reduce „!” satisface imediat.
- **cut/0** poate fi folosit pentru a îmbunătăți eficiența căutării în Prolog prin reducerea spațiului de căutare.
- Tăierile Prolog pot fi folosite pentru a elimina răspunsurile nedorite.
- Prolog cut poate fi folosit în general pentru a-i spune programului că a găsit regula potrivită pentru un anumit scop.
- Îi spune sistemului Prolog să eșueze imediat un anumit obiectiv, fără măcar să încerce soluții alternative.
- Se va încheia generarea de soluții alternative.

Tăierea poate fi clasificată în 2 tipuri:

1. **Green cut**: va afecta sensul procedural al programului, dar nu afectează sensul declarativ. Prin urmare, nu afectează lizibilitatea programului și este utilizat în principal pentru optimizarea căutării, evitându- se căutările inutile. Cu alte cuvinte, se poate recurge la stoparea căutărilor, dacă răspunsul deja este găsit.
2. **Red cut**: afectează semnificația declarativă a programului și semnificația procedurală a programului. Va afecta lizibilitatea programului, este folosit pentru a evita orice calcule irosite și introduce semantică. Dacă tăierea roșie nu este utilizată cu precauție, poate afecta ieșirea într-un mod arbitrar.