Université de Monastir

Cours: Programmation déclarative

Chapitre 4: Backtracking et prog. dynamique en Prolog

Réalisé par:
Dr. Taoufik Sakka Rouis
https://github.com/srtaoufik/Cours-Prog-Declarative/

Introduction au backtracking

Le backtracking est la méthode de recherche principale utilisée par Prolog pour trouver des solutions.

- Lorsque Prolog rencontre une impasse dans une règle, il revient en arrière pour explorer d'autres possibilités.

- Ce processus est essentiel pour résoudre des problèmes avec plusieurs solutions.

Introduction au backtracking

- Recherche en profondeur (Depth-First Search) : Prolog explore un chemin en profondeur jusqu'à ce qu'une solution soit trouvée ou qu'une impasse soit atteinte.

- Recherche des solutions multiples : Prolog peut générer toutes les solutions possibles en revenant en arrière (backtracking).

- Le backtracking permet de tester toutes les alternatives en cas d'échec d'une règle ou condition.

Chapitre 4: Backtracking et prog. dynamique en Prolog

Introduction au backtracking

Exemple - Recherche avec Backtracking
Considérons une base de données de relations familiales :

parent (john, mary).
parent (john, alex).
parent (mary, susan).

Requête : Trouver tous les enfants de John.
?- parent (john, X).

Résultat : X = mary ; X = alex (grâce au backtracking).

Introduction au backtracking

• L'opérateur **findall** permet de collecter toutes les solutions d'une requête en une liste.

Exemple :
?- findall(X, parent(john, X), Liste).

% rep. Liste = [mary, alex].

• **findall** permet de gérer efficacement les résultats multiples.

Exercices sur le backtracking

Exercice: Écrire un prédicat permettant de trouver tous les nombres pairs dans une liste donnée.

Solution :
pair(X) :- X mod 2 =:= 0.
-- l'opérateur =:= est utilisé pour **vérifier l'égalité arithmétique** entre deux **expressions**.
pairs([], []).
pairs([T|Q], [T|P]) :- pair(T), pairs(Q, P).
pairs([_|Q], P) :- pairs(Q, P).

Requête :
?- pairs([1, 2, 3, 4], P).   % rep. P = [2, 4].

Exercices sur le backtracking

Exercice : Écrire un prédicat permettant de trouver un chemin entre deux nœuds dans un graphe avec backtracking.

Solution :

```
arc(a, b).
arc(b, c).
arc(a, c).
chemin(X, Y, [X,Y]) :- arc(X, Y).
chemin(X, Y, [X|P]) :- arc(X, Z), chemin(Z, Y, P).
```

Requête :

```
?- chemin(a, c, P).        % rep.  P = [a, b, c] ; P = [a, c].
```


Introduction à la prog. dynamique

- La programmation dynamique est une technique d'optimisation.

- Elle repose sur la décomposition d'un problème en sous-problèmes plus petits.

- En Prolog, elle se base sur :
  - La mémorisation des résultats intermédiaires.
  - L'utilisation de faits dynamiques.

Introduction à la prog. dynamique

Avantages:

• Éviter le recalcule inutile de sous-problèmes déjà résolus.

• Approprié pour les problèmes avec des sous-problèmes qui se chevauchent.

• Optimiser les algorithmes récursifs coûteux.

• Résoudre des problèmes combinatoires comme :
- La suite de Fibonacci.
- Le problème du sac à dos.

Introduction à la prog. dynamique

Exemple : Fibonacci (Solution naïve)

```
fibonacci(0, 0).
fibonacci(1, 1).
fibonacci(N, F) :-   N > 1, N1 is N - 1, N2 is N - 2,
                     fibonacci(N1, F1), fibonacci(N2, F2),
                     F is F1 + F2.
```

• Limite : Beaucoup de calculs redondants.
• Complexité exponentielle.

Introduction à la prog. dynamique

La programmation dynamique en Prolog repose sur des outils permettant d'optimiser le calcul et de gérer les connaissances dynamiques :

- Directives dynamiques (:- dynamic)
- Utilisation des arités
- Gestion de faits dynamiques (asserta, retract)

Chapitre 4: Backtracking et prog. dynamique en Prolog

Introduction à la prog. dynamique

La directive :- dynamic permet de rendre un prédicat modifiable durant l'exécution du programme.

Syntaxe :

:- dynamic Predicat/Arity.

Exemple :

:- dynamic memo_fibonacci/2.

Applications :

- Ajouter dynamiquement des faits (asserta/1, assertz/1).
- Retirer des faits (retract/1).

Introduction à la prog. dynamique

L'arité correspond au nombre d'arguments d'un prédicat.

Exemple :
- memo_fibonacci/2 a une arité de 2 (deux arguments : N et F).

Utilisation :
Les prédicats dynamiques doivent être déclarés avec leur arité exacte.

Introduction à la prog. dynamique

Prolog permet de modifier la base de faits durant l'exécution :

- asserta(Fait) : Ajoute un fait au début de la base.
- assertz(Fait) : Ajoute un fait à la fin de la base.
- retract(Fait) : Supprime un fait existant.

Exemple :

```
asserta(memo_fibonacci(5, 8)).
retract(memo_fibonacci(0, 0)).
```

Exercices sur la prog. dynamique

Exercice 1:

1. Implémentez une fonction factorial/2 avec mémorisation
3. Testez votre programme avec les requêtes :

?- factorial(5, F).
?- factorial(3, F).

Introduction à la prog. dynamique

```
%Rép. Question 1 (Solution 1):
:- dynamic memo_factorial/2.
% Cas de base
factorial(0, 1) :- !.
% Le cut (!) permet de couper l'exploration une fois qu'une solution est trouvée.

% Cas récursif avec mémorisation
factorial(N, F) :-    N > 0,    memo_factorial(N, F), !.
  % Si le résultat est déjà mémorisé, on l'utilise directement.

factorial(N, F) :-    N > 0,    N1 is N - 1,
   factorial(N1, F1),  % Calcul de factorial(N-1)
   F is N * F1,        % Calcul de factorial(N)
   asserta(memo_factorial(N, F)).  % Mémorisation du résultat pour N

% c'est une bonne solution
```

Chapitre 4: Backtracking et prog. dynamique en Prolog

Introduction à la prog. dynamique

```
%Rép. Question 1 (Solution 2):
:- dynamic factorial/2.
% Cas de base
factorial(0, 1) :- !.
% Le cut (!) permet de couper l'exploration une fois qu'une solution est trouvée.

factorial(N, F) :-    N > 0,    N1 is N - 1,
   factorial(N1, F1),  % Calcul de factorial(N-1)
    F is N * F1,         % Calcul de factorial(N)
    asserta(factorial(N, F)).  % Mémorisation du résultat pour N


% C'est  pas une bonne solution
```

17

Chapitre 4: Backtracking et prog. dynamique en Prolog

Introduction à la prog. dynamique

```
%Rép. Question 1 (Solution 3):
:- dynamic factorial/2.
% Cas de base
factorial(0, 1):- !.
% Cas général avec gestion explicite de la mémorisation
factorial(N, F) :-
    N > 0,
    (   clause(factorial(N, F), true)   % Vérifie si le résultat est déjà mémorisé
         ->  true                        % Si oui, on l'utilise tel quel
     ;   % Sinon, le calculer et le mémoriser
        N1 is N - 1,
        factorial(N1, F1),  % Calcul de factorial(N-1)
        F is N * F1,        % Calcul de factorial(N)
        asserta(factorial(N, F))  % Mémorisation du résultat pour N
    ).
% C'est une bonne solution
```

18

Introduction à la prog. dynamique

Rép. Question 2:

?- factorial(5, F).

% Rep. F = 120 (calculé et mémorisé)

?- factorial(3, F).

% Rep. F = 6 (calculé et mémorisé après le premier appel)

Exercices sur la prog. dynamique

Exercice 2:

Implémentez une version optimisée de la suite de Fibonacci en Prolog en utilisant la programmation dynamique.

Exercices sur la prog. dynamique

Solution 1 pour la suite de Fibonacci:

```
:- dynamic memo_fibonacci/2.
fibonacci(0, 0).
fibonacci(1, 1).
fibonacci(N, F) :-   memo_fibonacci(N, F), !.
fibonacci(N, F) :-   N > 1, N1 is N - 1, N2 is N - 2,
   fibonacci(N1, F1), fibonacci(N2, F2),
   F is F1 + F2,   asserta(memo_fibonacci(N, F)).
% c'est une bonne solution
```

NB Le cut (!) est utilisé pour empêcher le backtracking une fois que la condition avant le cut a été satisfaite.

➔Dans notre exemple, cela signifie que si **memo_fibonacci(N, F)** est trouvé dans la base de faits ou de règles, Prolog n'essaiera **pas d'exécuter les clauses suivantes** de la définition de memo_fibonacci/2.

```
1 ?- fibonacci(3,X).          %rep. X = 2.
2 ?- memo_fibonacci(3,X).     %rep. X = 2.
3 ?- memo_fibonacci(4,X).     %rep. false.
4 ?- fibonacci(4,X).          %rep. X = 3.
```


Chapitre 4: Backtracking et prog. dynamique en Prolog

Exercices sur la prog. dynamique

Solution 2 pour la suite de Fibonacci:

```
:- dynamic fibonacci/2.

fibonacci(0, 0).
fibonacci(1, 1).
fibonacci(N, F) :-   N > 1, N1 is N - 1, N2 is N - 2,
   fibonacci(N1, F1), fibonacci(N2, F2),
   F is F1 + F2,   asserta(fibonacci(N, F)).

% C'est  pas une bonne solution
```

Exercices sur la prog. dynamique

Solution 3 pour la suite de Fibonacci:

```
:- dynamic fibonacci/2.
% Cas de base
fibonacci(0, 0).
fibonacci(1, 1).
% Cas général avec gestion explicite de la mémorisation
fibonacci(N, F) :-    N > 1,
    % Vérifie si le résultat est déjà mémorisé
    (   clause(fibonacci(N, F), true)
        ->  true          % Si déjà mémorisé, utiliser le résultat
    ;   % Sinon, le calculer et le mémoriser
        N1 is N - 1,
        N2 is N - 2,
        fibonacci(N1, F1),
        fibonacci(N2, F2),
        F is F1 + F2,
        asserta(fibonacci(N, F))  % Mémorisation
    ). % c'est une bonne solution
```


Chapitre 4: Backtracking et prog. dynamique en Prolog

Exercices sur la prog. dynamique

Exercice: Suite

Soit la suite U_n définit par :

$U_n = 2 * U_{n-1}$ si n est un entier impair

$U_n = U_{n-1} + U_{n-2}$ si n est un entier pair

$U_0 = 1$

Écrire un prédicat dynamique qui calcule le terme U_n

Exercices sur la prog. dynamique

Exercice: Triangle de Pascal

On veut calculer les coefficients binomiaux $C_n^k = \begin{pmatrix} n \\ k \end{pmatrix} = \frac{n!}{k!(n-k)!}$

Rappelant les propriétés suivantes :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \; pour \; 0 < k < n$$
$$\binom{n}{n} = 1 \; et \; \binom{n}{0} = 1$$

Proposer un prédicat prolog qui qui permet de calculer $\binom{n}{k}$ en utilisant la technique de la programmation