PLC – Cours 6

Thi-Bich-Hanh Dao

Université d'Orléans

M1 Informatique

Plan

Programmation non-déterministe

Algorithmes "générer et tester"

Algorithmes "générer et tester"

- Un algorithme de "génération et test" consiste en un processus qui engendre l'ensemble des candidats solutions au problème et un autre processus, qui teste les candidats, essayant de trouver un (ou plusieurs) candidats qui sont solutions.
- Le "backtracking" (retour en arrière) de Prolog est très bien adapté à ces algorithmes.
- Ces algorithmes ne sont pas efficaces mais sont faciles à construire.
- Des exemples de problèmes : *n* reines, coloriage d'un plan, carré magique, ...

Algorithmes "générer et tester"

Forme typique

Typiquement, ces programmes sont sous forme

```
trouver(X) :- generer(X), tester(X).
```

Lorsqu'on appelle `find(X)` :

- `generer(X)` réussit, retournant un X avec lequel `tester(X)` est appelé,
- si le but `tester(X)` échoue, le retour en arrière remonte à `generer(X)`, qui engendre l'élément suivant,
- ceci continue jusqu'à ce que le testeur trouve avec succès une solution, ou que le générateur épuise les autres candidats,
- lorsqu'une solution a été trouvée, le mécanisme d'exploration de l'arbre de dérivation permet de trouver les solutions suivantes.

Algorithmes "générer et tester"

Exemples étudiés

- Trouver un élément pair d'une liste

```
trouver(X,L) :- member(X,L), pair(X).
pair(X) :- Y is X mod 2, Y==0.
```

- Tri lent : générer les permutations et tester si une permutation est triée

```
tri_lent(L,Lt) :- permuter(L,Lt), est_triee(Lt).
```

- Problème des n reines
- Ali Baba et les 40 voleurs

Algorithmes "générer et tester"

Les n reines

- Problème : placer n reines sans prises sur un échiquier $n \times n$.
- Pas de solution pour $n = 2$ et $n = 3$, 2 solutions pour $n = 4$, 92 solutions pour $n = 8$.
- Représentation des configurations : une liste de n entiers, inclus entre 1 et n.
- Le i-ème entier donne la position (numéro de colonne) de la reine sur la i-ème ligne.
- Exemple : deux solutions avec $n = 4$
  [2,4,1,3] et [3,1,4,2]
- Formulation `reines(N,X)` ?

Algorithmes "générer et tester"

Les n reines : programme amélioré

- Au lieu d'engendrer la permutation complète, c-à-d placer tous les dames puis les tester, chaque reine peut être testée au moment où elle est placée.
- On commence avec la liste $L = [1, 2, \ldots, N]$ des reines à placer et la liste vide [] des reines déjà placées.
- Jusqu'à l'épuisement de la liste L, on sélectionne un numéro et le sortir de L. Ce numéro sera la colonne de le première reine s'il n'y a pas d'attaque avec les reines déjà placées. En ce cas la liste des reines déjà placées est modifiée.

Algorithmes "générer et tester"

Ali Baba et les 40 voleurs

- Problème : voir feuille attribuée
- Idée simple :
  Générer les permutations et tester si une permutation représente une solution. Pour cela, il faut entre autres :
  - trouver les permutations de [1,2,...,n] avec 1 au début de la liste,
  - un prédicat représentant la libération par avancement circulaire de n pas, lorsque une place est libérée, le nombre de ce place est remplacé par 0, dans ce prédicat, lorsque 2 est libéré, au moins la moitié de la liste doit être déjà libéré,
  - un prédicat représentant la libération tant qu'elle est encore possible (on ne se trouve pas sur une place déjà libérée),
  - un prédicat qui détermine si une permutation est une solution, c-à-d lorsque la libération se termine, toutes les places sont libérées.

Plan

Programmation non-déterministe

1. Algorithmes "générer et tester"
2. Non-déterminisme par indifférence et par ignorance

Non-déterminisme par indifférence et par ignorance

Différentiés par la nature du choix à faire entre différentes possibilités

- par indifférence : le choix se fait arbitrairement,
- par ignorance : le choix est important mais on ne sais lequel est le choix correct au moment où on le fait.

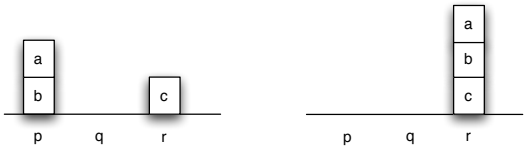
Le fonctionnement de programmes Prolog comportant du non-déterminisme par ignorance est connu. Ex. : tester si deux arbres binaires sont isomorphes

```
isotree(void,void).
isotree(tree(X,L1,R1),tree(X,L2,R2)) :-
        isotree(L1,L2), isotree(R1,R2).
isotree(tree(X,L1,R1),tree(X,L2,R2)) :-
        isotree(L1,R2), isotree(R1,L2).
```


Non-déterminisme par indifférence et par ignorance

Le monde de blocs (1/4)

- Trois blocs a, b, c, trois places p, q, r
- Actions possibles :
  - transférer un bloc libre vers une place libre
  - transférer un bloc libre sur un autre bloc libre
- Trouver un plan pour arriver de l'état initial à l'état final



Non-déterminisme par indifférence et par ignorance

Le monde de blocs (2/4)

- L'algorithme non déterministe : tant que l'état désiré n'est pas atteint
  - trouver une action légale,
  - mettre à jour l'état courant,
  - vérifié qu'il n'a pas été visité auparavant.
- Deux actions possibles, le choix n'est pas déterministe.
- Pour chaque action, il convient de spécifier les conditions la rendant légales et comment mettre à jour l'état.

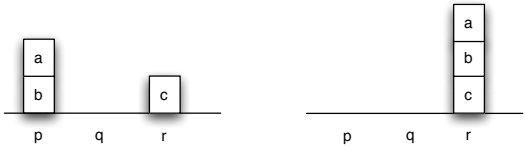
Non-déterminisme par indifférence et par ignorance

Le monde de blocs (3/4)

- Un état est représenté par une liste de description, ex. l'état initial `[sur(a,b), sur(b,p), sur(c,r)]`
- Les actions :
  - `placer_sur_place(B1,B2,P)` transférer bloc `B1` étant sur bloc `B2` vers la place `P`
  - `placer_sur_bloc(B1,X,B2)` transférer bloc `B1` étant sur `X` vers le dessus du bloc `B2`
- Pour ne pas revenir à un état, on mémorise les états déjà visités.

Non-déterminisme par indifférence et par ignorance

Le monde de blocs (4/4)




```
?- test_plan(P).

P = [placer_sur_place(a,b,q),placer_sur_bloc(b,p,c),
placer_sur_bloc(a,q,b)]
```