CHION
Romain

21/12/2012

INGENIERIE DE LA CONNAISSANCE ET PROGRAMMATION LOGIQUE

TNE n°1

Le Compte est Bon !

Objectifs

Nous nous intéressons dans ce TP à la recherche de solutions pour le jeu « le compte est bon », toujours diffusé dans le cadre de l'émission la plus ancienne du PAF (sans interruption depuis 1972 !) : « des chiffres et des lettres ». Nous souhaitons mettre en œuvre une recherche exhaustive des solutions, pour garantir que la solution trouvée est bien la plus proche possible du nombre cherché. Nous souhaitons également contrôler la durée d'exécution de notre algorithme de manière à renvoyer une réponse dans les 45 secondes imparties pour trouver le résultat.

Notre approche de recherche exhaustive se fondera sur un parcours à la volée en profondeur d'abord dans un arbre contenant toutes les permutations possibles des chiffres de départ combinées avec les 4 opérateurs arithmétiques possibles.

Chacune de ces permutations correspondra à l'expression d'un calcul en notation Polonaise inverse. Il s'agit d'une évolution de la notation polonaise inventée par Jan Lukasiewicz en 1920 ; proposée par Charles Leonard Hamblin au milieu des années 1950 pour permettre les calculs sans adresses mémoire.

Ce type de notation permet d'éviter de manipuler des parenthèses dans l'écriture d'une expression à calculer. Tout calcul susceptible d'être solution de notre problème admet une représentation sans parenthèses en notation Polonaise inverse. Ainsi, si nous garantissons que notre algorithme parcoure effectivement toutes les expressions en notation Polonaise inverse possibles que l'on peut construire à partir des chiffres de départ, on est sûr qu'on aura bien testé tous les calculs possibles.

Par exemple, l'expression 4 3 2 1 - * + s'évalue à la valeur 4 + (3 * (2 -1)) = 7. Cette notation a notamment été utilisée par la série de calculatrices scientifiques produites par HP depuis la HP-35 à partir de 1972. Son intérêt est qu'elle nécessite moins de mémoire pour évaluer un calcul puisqu'elle ne nécessite pas d'empiler les opérateurs de l'expression, qui sont consommés dès qu'ils sont entrés par l'utilisateur.

Attention cependant : écrire en notation Polonaise inverse n'induit pas que tous les opérateurs se trouvent à la fin de l'expression. Par exemple, pour exprimer le calcul (3+4) * (7-2), il faut écrire 3 4 + 7 2 - * : le symbole + est placé au milieu de l'expression.

Partie 1 : Génération du problème du Compte est Bon

Nous commençons par générer une instance du problème du compte est bon. Le jeu commence par le tirage au sort de 6 « cartons » choisis aléatoirement parmi les valeurs suivantes : 1;2;3;4;5;6;7;8;9;10;25;50;75;100. Le nombre à trouver doit être compris entre 100 et 999.

On suggère de structurer le problème en utilisant une liste (cible, listeCartons) qui contiendra une instance du problème. Pour éviter de régénérer un problème à chaque exécution d'un but, et de devoir passer l'instance du problème aux prédicats de résolution, nous souhaitons enregistrer l'instance générée comme un nouveau fait de la base de connaissances.

Génération dynamique de faits

Dans certains cas, il peut être intéressant de faire enregistrer par prolog de nouvelles connaissances, suite à l'exécution d'un but. C'est une manière de se rapprocher des langages de programmation impératifs comme le C dans lequel les effets de bord sont permis, ou encore de construire progressivement une base de données. C'est aussi une manière d'éviter de manipuler des structures de données trop longues en les découpant en morceaux qui seront autant de faits. Mais il ne faut pas en abuser !

Les deux prédicats qui gèrent cela s'appellent assert et retract : assert/1 ajoute un nouveau terme dans la base de connaissances de prolog (à la fin des faits), retract le supprime. Il existe aussi les prédicats recorda, recorded et erase qui permettent de stocker des termes dans une base de données associative, où chaque terme est associé à une clé.

Attention : le prédicat ***retract*** ne permet pas de supprimer des faits statiques (définis dans le fichier consulté), il ne peut agir que sur des faits établis à l'aide d'un ***assert*** .

1. Développer un prédicat genInstance(Cible :,ListeCartons:) permettant de définir une instance d'un problème du compte est bon. Ce prédicat instancie les variables Cible et Cartons, et crée un nouveau fait baptisé ***instance([Cible,Cartons])*** avec un assert, qui pourra être utilisé par la suite. Ce prédicat supprime une éventuelle instance précédemment définie avec un retract.

NB : La fonction ***random(+IntExpr)*** *permet de générer un entier entre 0 et une valeur passée en paramètre (non comprise).*

```
:-dynamic instance/2.

genInstance2(Cible,[A,B,C,D,E,F]):-

    Cible is random(900)+100,

    genCarton(A),

    genCarton(B),

    genCarton(C),

    genCarton(D),
```

```
    genCarton(E),
    genCarton(F),
    retractall(instance(_,_)),
    assert(instance(Cible,[A,B,C,D,E,F])).

genCarton(A):-extraireR(A,[1,2,3,4,5,6,7,8,9,10,25,50,75,100],_).
```

Partie 2 : Manipulation d'expressions en notation Polonaise inverse

Dans cette section, nous définissons les prédicats de manipulation d'expressions en notation Polonaise inverse. Une expression pourra contenir un nombre arbitraire d'opérandes et d'opérateurs (que nous nommons dans la suite des « piv »), placés dans n'importe quel ordre. Nous utiliserons une liste pour stocker une expression contenant des piv.

Nous souhaitons rédiger une fonction d'évaluation d'expression en notation Polonaise inverse représentée sous forme de liste. Cette fonction utilisera une pile pour stocker les opérandes de l'expression qu'elle rencontre. Lorsqu'un opérateur est trouvé, on dépile deux opérandes et on ré- empile le résultat de l'opération. A la fin, si la liste représente une expression valide, tous les éléments de la liste ont été consommés et la pile ne contient plus qu'une opérande : le résultat du calcul, que l'on renvoie.

```
/*Partie 2*/

iso(*).

iso(+).

iso(-).

iso(/).

isp(+).

ism(-).

isf(*).

isd(/).

isande(X):-integer(X).

ispiv(X):-iso(X).

ispiv(X):-isande(X).
```

2. Rédiger des prédicats de gestion de pile permettant d'empiler ou dépiler des PIV d'une pile passée en paramètre : pivPush(:PileInitiale,PileResultat :,:PivAEmpiler) et pivPop(:Pile,PivDepile:).

```
/*Question 2*/

pivPush(L,[O|L],O):-ispiv(O).

pivPop([O|_],O):-ispiv(O).
```

3. Rédiger le prédicat EvaluerExpression(:Pile,:ListePiv,Resultat:). Ce prédicat évalue une expression en notation Polonaise inverse en suivant la démarche décrite ci-dessus. Le premier argument est la pile intermédiaire, le second la liste des piv restants dans l'expression, le dernier est le résultat de l'évaluation. Pour simplifier le travail, on suggère de créer les prédicats isOperande(:Piv) et isOperateur(:Piv) permettant de savoir si un PIV qui vient d'être extrait d'une expression est une opérande ou un opérateur.

```
/*Question 3*/

evaluerExpression([],[X],X):-!.

evaluerExpression([T|Q],L,R):-
        isande(T),
        evaluerExpression(Q,[T|L],R),!.
evaluerExpression([T|Q],[X|[Y|L]],R):-
        isp(T),S is (X+Y),
        evaluerExpression(Q,[S|L],R),!.
evaluerExpression([T|Q],[X|[Y|L]],R):-
        ism(T),S is (Y-X),
        evaluerExpression(Q,[S|L],R),!.
evaluerExpression([T|Q],[X|[Y|L]],R):-
        isf(T),S is (X*Y),
        evaluerExpression(Q,[S|L],R),!.
evaluerExpression([T|Q],[X|[Y|L]],R):-
        isd(T),X =\= 0,Y mod X =:= 0,S is (Y/X),
        evaluerExpression(Q,[S|L],R),!.
```

Il est difficile de formaliser un critère simple permettant de détecter si une expression en notation Polonaise est valide ou non. Comme nous générons exhaustivement toutes les suites de symboles possibles, de nombreuses expressions générées seront invalides. Il faudra dans ce cas renvoyer comme résultat l'atome res_erreur. C'est le cas par exemple lorsqu'il n'est pas possible de dépiler deux opérandes lorsque l'on rencontre un opérateur, lorsque l'on effectue une division par zéro ou une division dont le résultat n'est pas entier.

Un cas particulier doit cependant renvoyer comme résultat l'atome res_ok : lorsque l'expression est vide, ce qui caractérise la toute première étape de notre procédure de génération.

/*Res_ok et Res_erreur sont superflus, car lorsque ok, cela renvoie un résultat sinon, cela renvoie false*/

Partie 3 : Recherche à la volée en profondeur d'abord

Nous souhaitons maintenant construire toutes les expressions possibles formées à partir des 6 cartons de départ et des opérateurs arithmétiques. Pour cela, nous mettons en œuvre une méthodologie de recherche à la volée en profondeur d'abord.

Chaque carton ne pouvant être utilisé qu'une seule fois, nous développons une boucle qui parcourt les cartons restants et les 4 opérateurs possibles (qui – au contraire des cartons - peuvent être réutilisés à chaque fois). Pour chaque opérateur ou opérande (ie chaque piv), nous l'ajoutons à l'expression en cours de construction (enregistrée sous forme de liste) et appelons récursivement la procédure de recherche en lui passant cette nouvelle expression ainsi que les cartons restants.

Au début de chaque appel récursif, nous testons si l'expression passée en paramètre est valide de manière à ne pas parcourir le sous-arbre correspondant, et éviter par exemple de produire des expressions construites uniquement à partir de suites d'opérateurs. Lorsqu'elle est valide, on enregistre le résultat intermédiaire (résultat de l'évaluation, expression polonaise associée représentée sous forme de liste de piv), si il améliore la meilleure solution connue. L'enregistrement de la meilleure solution connue et du meilleur calcul correspondant peuvent être faits à l'aide d'appels au prédicat assert comme dans la partie 1.

Parcourir à la volée signifie que nous n'avons pas besoin de stocker en mémoire l'ensemble des expressions possibles, c'est la pile d'exécution de notre processus, qui contient les arguments passés en paramètre, qui nous sert de stockage temporaire, à la manière d'un parcours DFS dans un graphe.

Le parcours en profondeur est pertinent dans la mesure où nos expressions ont une taille maximale connue, puisqu'on ne peut assembler au maximum que 6 opérandes et 5 opérateurs, soit une profondeur maximale de 11 appels récursifs.

4. Pour nous aider, nous développons un prédicat renvoyant le ième élément d'une liste et la liste privée de cet élément : extraire(:I,:ListeCartons,ResteCartons:).

```
/*Partie 3*/

/*Question 4*/

extraire(1,T,[T|Q],Q):-!.

extraire(N,X,[T|Q],[T|R]):-N1 is N-1,extraire(N1,X,Q,R).


extraireR(CN1,C,RC):-length(C,N),N>=1,N1 is random(N)+1,

        extraire(N1,CN1,C,RC).
```

5. Cette fonction nous permettra de parcourir la liste de cartons dans notre procédure de recherche : recherche(:ResteCartons,:ListePiv). Cette fonction n'est appelée qu'avec des variables instanciées, car elle produit des nouveaux faits (par des assert) lorsqu'elle améliore une solution. Elle s'interrompt lorsque le résultat cherché dans l'instance du problème à résoudre a été trouvé.

```
/*Question 5*/

:- dynamic solution/1.


solveur(M):-instance(Cible,_),solution(M),evaluerExpression(M,[],Cible).

solveur(Piv):-

        instance(Cible,Cartons),

        recherche(Cartons,[],Piv),

        evaluerExpression(Piv,[],R),

        solution(M),

        evaluerExpression(M,[],S),E is abs(Cible-S)-1,

        E>=abs(Cible-R),

        retractall(solution(_)),assert(solution(Piv)).



recherche(_,S,S):-evaluerExpression(S,[],_).

recherche(C,Piv,S):-length(C,N),length(Piv,M),M=<N-1,
```

```
        recherche(C,[+|Piv],S).
recherche(C,Piv,S):-length(C,N),length(Piv,M),M=<N-1,
        recherche(C,[-|Piv],S).
recherche(C,Piv,S):-length(C,N),length(Piv,M),M=<N-1,
        recherche(C,[*|Piv],S).
recherche(C,Piv,S):-length(C,N),length(Piv,M),M=<N-1,
        recherche(C,[/|Piv],S).
recherche(C,Piv,S):-
        extraireR(CN1,C,RC),
        recherche(RC,[CN1|Piv],S).
recherche(C,Piv,S):-
        extraireR(_,C,RC),
        recherche(RC,Piv,S).
/*recherche(C,Piv,S):-extraire(1,CN1,C,RC),recherche(RC,[CN1|Piv],S).
recherche(C,Piv,S):-extraire(1,_,C,RC),recherche(RC,Piv,S).
recherche(C,Piv,S):-extraire(2,CN1,C,RC),recherche(RC,[CN1|Piv],S).
recherche(C,Piv,S):-extraire(2,_,C,RC),recherche(RC,Piv,S).
recherche(C,Piv,S):-extraire(3,CN1,C,RC),recherche(RC,[CN1|Piv],S).
recherche(C,Piv,S):-extraire(3,_,C,RC),recherche(RC,Piv,S).
recherche(C,Piv,S):-extraire(4,CN1,C,RC),recherche(RC,[CN1|Piv],S).
recherche(C,Piv,S):-extraire(4,_,C,RC),recherche(RC,Piv,S).
recherche(C,Piv,S):-extraire(5,CN1,C,RC),recherche(RC,[CN1|Piv],S).
recherche(C,Piv,S):-extraire(5,_,C,RC),recherche(RC,Piv,S).
recherche(C,Piv,S):-extraire(6,CN1,C,RC),recherche(RC,[CN1|Piv],S).
recherche(C,Piv,S):-extraire(6,_,C,RC),recherche(RC,Piv,S).*/
```

Partie 4 : Affichage des résultats et Contrôle du délai de réponse

Nous souhaitons afficher le calcul qui mène au meilleur résultat de la manière habituelle, en notation infixe.

6. Développer un prédicat afficher(:ListePiv) permettant d'afficher à l'aide de prédicats 'write' le calcul correspondant à une expression donnée en notation polonaise inverse, sous forme de liste de piv.

```
/*Partie 4*/

/*Question 6*/

afficher(L):-afficherCalc(L,[]). /*Lance l'affichage sans l'accumulateur*/


/*Calcule et affiche le calcul du résultat de la polonaise en argument 1, argument 2 accumulateur*/

afficherCalc([],[_]):-!.

afficherCalc([T|Q],L):-isande(T),afficherCalc(Q,[T|L]),!.

afficherCalc([T|Q],[X|[Y|L]]):-

        isp(T),S is (X+Y),

        write(Y),write(' '),write(T),write(' '),write(X),write(' '),write('='),write(' '),write(S),nl,

        afficherCalc(Q,[S|L]),!.

afficherCalc([T|Q],[X|[Y|L]]):-

        ism(T),S is (Y-X),

        write(Y),write(' '),write(T),write(' '),write(X),write(' '),write('='),write(' '),write(S),nl,

        afficherCalc(Q,[S|L]),!.

afficherCalc([T|Q],[X|[Y|L]]):-

        isf(T),S is (X*Y),

        write(Y),write(' '),write(T),write(' '),write(X),write(' '),write('='),write(' '),write(S),nl,

        afficherCalc(Q,[S|L]),!.

afficherCalc([T|Q],[X|[Y|L]]):-
```

```
    isd(T),S is (Y/X),

    write(Y),write(' '),write(T),write(' '),write(X),write(' '),write('='),write(' '),write(S),nl,

    afficherCalc(Q,[S|L]),!.
```

7. Pour garantir que la machine s'arrête après le délai imposé dans le jeu du compte est bon (45 secondes), on peut utiliser le prédicat call_with_time_limit(+Time, :Goal)

```
/*Question 7*/

solveurTime(S):-

    call_with_time_limit(45,solveur(S)).
```

8. Développer un prédicat permettant de rechercher puis d'afficher la meilleure solution dans un délai imposé. Si le résultat est trouvé plus tôt, nous souhaitons que la durée de la recherche soit affichée.

```
/*Question 8*/

lecompteestbon(_):-

    assert(solution([0,0,+])),

    time(solveurTime(S)),

    solution(S),

    afficher(S),

    instance(X,_),write('le resultat attendu était '),write(X).
```

Partie 5 : Recherche de solutions par la programmation par contraintes

Réfléchir et proposer des approches permettant d'utiliser la puissance de la programmation par contraintes pour résoudre le compte est bon, sans forcément les développer.

Conclusion

Des erreurs non résolues se trouvent à la fin. En effet, le programme s'arrête avant d'avoir trouvé la meilleure solution du premier coup et il s'arrête parfois lorsqu'il en trouve une meilleure (même sans la limite de temps et en enlevant les « ! »). Il finit par trouver la meilleure solution au bout de quelques lancements.