Programmation Logique avec contraintes - TD/TP 2

tiré partiellement de `www710.univ-lyon1.fr/~csolnon/Site-PPC/e-miage-ppc-som.htm`

Le but de ce TP est de programmer en Prolog "pur" un solutionneur basique de contraintes binaires sur un domaine fini.

Description de CSPs binaires en Prolog

Pour concevoir un solutionneur de contraintes, on fixe d'abord une convention pour décrire le CSP à résoudre. On se limite ici à des CSPs binaires, c'est-à-dire des CSPs dont toutes les contraintes portent sur 2 variables exactement, et on propose de décrire un CSP binaire (C,D) en définissant les 2 prédicats Prolog suivants :

- Le prédicat `variables/1` décrit les variables du CSP, avec leurs domaines associés: `variables(L)` donne dans `L` la liste des noms des variables du CSP, chaque nom de variable étant suivi du symbole ':' puis du domaine de la variable (la liste des valeurs que la variable peut prendre). Par exemple pour les 4 reines:

  ```
  variables([x(1):[1,2,3,4], x(2):[1,2,3,4], x(3):[1,2,3,4], x(4):[1,2,3,4]]).
  ```

- Le prédicat `consistants/2` décrit les contraintes binaires du CSP : `consistants((Xi,Vi),(Xj,Vj))` réussit si l'affectation partielle `{(Xi,Vi),(Xj,Vj)}` est consistante, autrement dit, si la contrainte binaire entre les variables `Xi` et `Xj` est vérifiée pour les valeurs `Vi` et `Vj`
  - Par exemple pour les 4 reines `?- consistants((x(1),2),(x(4),2))` donne `no`
    et `?- consistants((x(1),1),(x(2),3))` donne `yes`

- La définition pour les 4 reines en général :

  ```
  consistants((x(I),VI),(x(J),VJ)) :- VI =\= VJ, I+VI =\= J+VJ, I-VI =\= J-VJ.
  ```

Programmation de l'algorithme "génère et teste" en Prolog

- Définir le prédicat `genere(V,A)` qui réussit si `V` contient la liste des noms des variables du CSP, chaque nom de variable étant suivi du domaine de la variable, et `A` s'unifie avec une affectation totale des variables de `V`, c-à-d. une liste de couples `(variable,valeur)`. Indication : Modifiez le fichier `~habermeh/csp.pl`. Utilisez `member`.

  Exemple : `?- variables(V), genere(V,A).` Prolog doit énumérer toutes les affectations totales (il y en a 4 puissance 4...).

  ```
  A = [(x(1),1),(x(2),1),(x(3),1),(x(4),1)] ? ;
  A = [(x(1),1),(x(2),1),(x(3),1),(x(4),2)] ? ;
  A = [(x(1),1),(x(2),1),(x(3),1),(x(4),3)] ? ;
  A = [(x(1),1),(x(2),1),(x(3),1),(x(4),4)] ? ;
  A = [(x(1),1),(x(2),1),(x(3),2),(x(4),1)] ? ; etc.
  ```

- Définir le prédicat `teste(Sol)` qui réussit si `Sol` est une affectation consistante.

- Définir le prédicat `genereEtTeste(Sol)` qui unifie `Sol` avec une solution du CSP décrit par les prédicats `variables/1` et `consistants/2`

simple retour en arrière

On peut s'inspirer du prédicat `genere/2` : il suffit d'intégrer au cours de la génération des affectations un test qui vérifie, à chaque fois que l'on instancie une nouvelle variable, que cette instanciation est consistante avec les affectations déjà effectuées.

- Pour cela, définir d'abord le prédicat `teste/2` suivant : `teste((X,V),A)` réussit si l'affectation de `X` à la valeur `V` est consistante avec les affectations déjà effectuées dans `A`. Exemples :

  ```
  | ?- teste((x(3),1),[(x(1),1),(x(2),3)]).
  no
  | ?- teste((x(3),1),[(x(1),2),(x(2),4)]).
  yes
  ```

- Ensuite définir le prédicat `simpleretourenarriere(V,Solpartielle,Sol)` qui étant donné un ensemble de variables (avec leurs domaines) `V` pas encore affectées et une solution partielle `Solpartielle` (une affectation des autres variables) donne dans `Sol` une affectation complète qui est une solution. Par exemple :

```
?- simpleretourenarriere([x(3): [1,2,3,4], x(4): [1,2,3,4]], [(x(1),2),(x(2),4)],A).
A = [(x(3),1),(x(4),3)] ? ;
no
?- simpleretourenarriere( [x(3): [1,2,3,4], x(4): [1,2,3,4]], [(x(1),1),(x(2),3)],A).
no
?- simpleretourenarriere([x(1): [1,2,3,4], x(2): [1,2,3,4],x(3): [1,2,3,4], x(4): [1,2,3,4]],[],A).
A = [(x(1),2),(x(2),4),(x(3),1),(x(4),3)] ? ;
A = [(x(1),3),(x(2),1),(x(3),4),(x(4),2)] ? ;
no
```

- Définir ensuite le prédicat `simpleretourenarriere(Sol)` qui donne une solution dans `Sol` en utilisant la methode du simple retour en arrière.

Consistance de noeud

Pour utiliser la consistance de noeud, définir d'abord un prédicat `filtre(X,V,Vars,VarsFiltrees)` qui étant donné une variable `X`, une valeur `V` pour cette variable et une liste de variables `Vars` (avec leur domaines) pas encore affectées donne dans `VarFiltrees` la même liste de variables avec leur domaines filtrés. Chaque domaine filtré est obtenu en enlevant du domaine toutes les valeures incompatibles avec l'affection de la valeur `V` à la variable `X`. Chaque domaine doit contenir au moins une variable, sinon le prédicat échoue.

Indication : Pour connaître toutes les réponse à une question on peut utiliser le prédicat prédéfini `bagof`, par exemple `bagof(Vj, (member(Vj,[1,2,3,4]),consistants((x(2),3),(x(3),Vj))),L)` donne dans L une liste de toutes les valeurs pour `x(3)` qui sont compatibles. Exemples de `filtre` :

```
?- filtre(x(1),1,[x(2):[1,2,3,4], x(3):[1,2,3,4], x(4):[1,2,3,4]],VarFiltrees).
VarFiltrees = [x(2):[3,4], x(3):[2,4], x(4):[2,3]] ?
yes
?- filtre(x(1),1,[x(2):[1,2], x(3):[1,2], x(4):[1,2]], VarFiltrees).
no
```

Ensuite en utilisant `filtre` définir `retourconsnoeud(V,Solpartielle,Sol)` qui étant donné un ensemble de variables (avec leurs domaines) `V` pas encore affectées et une solution partielle `Solpartielle` (une affectation des autres variables) donne dans `Sol` une affectation complète en utilisant la méthode du retour en arrière avec noeud consistance.

Ensuite, définir `retourconsnoeud(Sol)`.

Heuristique : choisir la variable la plus contrainte

Pour utiliser cette heuristique, définir un prédicat `extraitMin(Var,Xmin,VarsansXmin)` qui étant donnée une liste de variables (avec leurs domaines) `Var` donne dans `Xmin` la variable (avec son domaine) qui a le plus petit domaine et dans `VarsansXmin` la liste des variables sans `Xmin`. Exemple :

```
?- extraitMin([a:[1,2,3,4], b:[1,2], c:[1,2,3]], X, VarsansXmin).
X = b:[1,2]
VarsansXmin = [a:[1,2,3,4],c:[1,2,3]]
```

Ensuite, définir comme pour les autres solutionneurs les prédicats permettant d'utiliser cette heuristique avec le retour en arrière.

Comparaison

Vous pouvez comparer vos différents solutionneurs sur différents problèmes de n reines qui peuvent être générés en utilisant le fichier `~habermeh/reinesN.pl` avec `genereReines(8)` par exemple.

Le prédicat predéfini `bagof` permet d'obtenir toutes les solutions à une question (voir ci-dessus). Le prédicat predéfini `statistics(runtime,X)` donne le temps de calcul en millisecondes depuis le lancement de Prolog et depuis le dernier appel de `statistics`. Comparer le temps d'éxécution des différents algorithmes pour trouver toutes les solutions du problème des 6 reines.