INF8215 – Intelligence artificielle : méthodes et algorithmes

TP2 : Prolog, programmation logique, programmation par contraintes

Introduction

Dans une première partie, nous allons nous intéresser à la programmation logique. Pour cela nous ferons usage de *Prolog*, qui est un langage de programmation fonctionnel logique. On a affaire ici à des bases de connaissances avec des prédicats et des règles sur ces prédicats. *Prolog* nous permet d'interroger ces bases afin d'en tirer des résultats.

Dans une seconde partie, nous allons nous intéresser à la programmation par contraintes (CP). Celle-ci permet de résoudre des problèmes de satisfaction ou d'optimisation combinatoires de façon efficace. Pour cela, nous ferons usage de *MiniZinc*, avec le solveur *Gecode*, afin de modéliser et résoudre nos problèmes.

Installation

Prolog et *MiniZinc* sont tous les deux disponibles sur vos postes de travail. Afin de les lancer, utilisez le terminal et exécutez une de ces commandes :

- `swipl`
- `MiniZincIDE`

Pour installer *Prolog* sur vos ordinateurs personnels, visitez le site officiel de SWI-Prolog. Pour installer *MiniZinc*, visitez la repository GitHub officielle de MiniZinc.

Astuce pour macOS et Windows :
Ajoutez…

- `%PROGRAMFILES%\swipl\bin` (Windows)
- `/Applications/SWI-Prolog.app/Contents/MacOS` (macOS)

… à votre `PATH` pour utiliser Prolog plus facilement à partir de *Powershell* ou *Command Line* sur Windows ou *Terminal* sur macOS.

De l'aide pour écrire vos programmes Prolog :
Installez *Sublime Text 3* sur vos ordinateurs, il peut reconnaitre la syntaxe de Prolog. Pour cela, appuyez sur Ctrl+Shift+P (ou Cmd+Shift+P sur macOS) et exécutez, dans l'ordre :

- `Install Package Control`
- `Package Control: Install Package`
- `Prolog`

Pour activer la syntaxe de Prolog, allez à *View* > *Syntax* > *Prolog* > *SWI-Prolog*.

Importants rappels

Utilisation de MiniZinc

Référez-vous aux exemples sur Moodle ou alors au MiniZinc Tutorial pour vous aider à utiliser MiniZinc et à comprendre sa syntaxe.

Les bases de Prolog

Prolog est un langage de programmation fonctionnel logique. Le code ne s'exécute pas nécessairement comme une suite d'instructions. Au contraire, il s'exécute comme une suite d'appels de fonctions. Dans le cas de Prolog, on manipule des prédicats qui se trouvent dans une base de connaissances (knowledge base).

Tout ce qui se trouve dans une base de connaissances est supposé **vrai**. Par contraste, tout ce qui ne s'y trouve pas est automatiquement **faux**.

Une base de connaissances est composée de prédicats. Ces prédicats sont définis par une liste de littéraux. Un littéral prend la forme `symbole_predicat(constante)`.

Soit la base de connaissances suivante :

```
humain(scott).
humain(ramona).
humain(envy).
humain(wallace).
fille(ramona).
ennemis(scott, envy).
```

Ouvrez un fichier texte et mettez-y ces 3 lignes. Enregistrez-le avec le nom `exemple.pl`.

Sur le Terminal, mettez-vous sur le dossier où figure votre fichier et tapez :

```
swipl –s exemple.pl
```

Alternativement, vous pouvez lancer `swipl` sur le terminal sans charger de fichier et le faire subséquemment (utile aussi pour recharger un fichier modifié) à l'aide d'une des commandes suivantes (pour notre exemple) :

```
?– [exemple].
?– consult(exemple).
```

Notre base des connaissances contient un seul prédicat : `humain`. Ce prédicat est défini sur 3 littéraux. Les éléments situés entre parenthèses sont appelés « *atoms* » (des constantes). Toutes les instructions, de n'importe quelle nature sur Prolog, peuvent être écrites sur plusieurs lignes. Elles finissent cependant obligatoirement par un point (`.`).

Interrogation d'une base de connaissances sur Prolog :
On peut consulter notre base de connaissances une fois celle-ci chargée en écrivant des littéraux à la suite de « ?– ».

Essayez de consulter la base avec `humain(scott)` ensuite avec `humain(ramona)`. Vous devez recevoir la réponse `true`. Ce sont des faits qui existent dans la base et qui sont par conséquent vrais.

Maintenant, essayez avec `humain(envy)`. Vous devez recevoir la réponse `false`. En effet, ce littéral n'existe pas et `envy`, dans le cadre de cette exécution, n'est pas un `humain`.

Maintenant essayez `chanteur(envy)`. Vous devez recevoir une erreur vous informant que `chanteur/1` est non-défini. En effet, à aucun moment est-ce que le prédicat singer n'a été défini. Cela est différent de `false`. Alors que `false` est le résultat d'une requête qui peut être valide, cette erreur est une situation inattendue et est donc une erreur de programmation à éviter.

Information : les prédicats et règles sont désignés par Prolog comme `symbole/n`. Le n ici est le nombre de paramètres que prend symbole. `symbole` peut être défini pour être compatible avec un nombre variable de paramètres et le prototype correspondant est désigné par le n approprié.

Variables :
Il est possible d'utiliser des variables afin d'augmenter les requêtes à la recherche de plus d'informations. Pour indiquer une variable, on fait usage d'une majuscule au début de son nom.

Supposons que nous voulons savoir quels sont tous les humains dans notre base de connaissances. Il suffira de formuler une des requêtes suivantes :

```
?- humain(X).
?- humain(CeciEstUneVar).
```

La réponse de Prolog va être (si on utilise X) :

```
X = scott ;
X = ramona ;
X = envy ;
X = wallace.
```

(Appuyez sur `;` à chaque ligne pour afficher la ligne suivante. Appuyez sur `.` pour arrêter la recherche)

Prolog va chercher tous les *atoms* à quoi X peut correspondre et les affiche un par un.

Opérateurs booléens :
Il est possible de formuler des requêtes qui sont la conjonction ou la disjonction d'un ou plusieurs littéraux. Pour cela on aura besoin de *ET* et *OU*. *ET* sur Prolog est une virgule (`,`). *OU* est un point-virgule (`;`).

On peut alors chercher tous les humains qui sont aussi des filles avec la requête suivante :

```
?- humain(X), fille(X).
```

On aura alors :

```
X = ramona ;
false.
```

(`false` ici marque la fin de la recherche)

Notez que même `envy` est une fille dans la vraie vie mais vu qu'on ne l'a pas indiqué dans la base de connaissances, celle-ci ne l'est pas dans ce cas de figure.

Notez aussi par ailleurs que notre base de connaissances a été rédigée de telle sorte que l'appel à `fille/1` seulement aurait suffi à avoir toutes les filles sans avoir besoin de `humain/1`.

Règles et procédures :
Armés de nos connaissances sur le fonctionnement de Prolog jusqu'à maintenant, nous pouvons définir des règles qu'on peut consulter sur Prolog afin de réaliser des actions ou obtenir de l'information.

Une règle est définie comme ceci :

```
symboleRegle(parametres) :- litteraux.
```

Soit le littéral `ennemis(scott, envy).` qui dit que Scott et Envy sont ennemis. Si on interroge notre base comme ceci :
`?- ennemis(envy, scott).`

Nous aurons `false`. En effet, le prédicat `ennemis/2` n'a jamais été défini de cette façon. Ceci est contraire à l'idée que Scott et Envy sont effectivement ennemis. Nous augmenterons alors `ennemis/2` avec une règle qui contournera cela :

```
ennemis(X, Y) :- ennemis(Y, X).
```

Avec cette règle, quand Prolog ne trouve pas le littéral avec l'ordre des *atoms* indiqué dans la base de connaissances, il ira voir la règle en haut qui inverse leur ordre. Prolog refait alors sa recherche avec les paramètres inversés et trouve le bon littéral.

Il est bien entendu possible d'écrire une conjonction ou une disjonction de littéraux dans les règles pour les rendre plus efficaces. Par exemple, si ne on cherche que les ennemis humains de Scott, on écrira la règle suivante :

```
ennemisHumainScott(X) :- (ennemis(X, scott); ennemis(scott, X)), humain(X).
```

Variables anonymes :
Parfois, dans certaines règles, on a besoin de stocker un élément dans une variable temporaire qu'on ne va pas utiliser. Au lieu de choisir une variable normale (qui déclencherait un avertissement de variable singleton), on utilise des variables anonymes pour cela. Les variables anonymes sont un underscore (_). Chacune de ces variables anonymes est indépendante.

Débogage :
Prolog propose un outil d'aide au débogage qui détaille la procédure de recherche de Prolog à chaque étape (la trace). Cela permet de déceler les erreurs de façon plus efficace. Afin d'activer cette fonction exécuter la commande suivante :

```
trace.
```

Pour désactiver ce mode :

```
notrace, nodebug.
```

Commentaires :
Il est possible d'insérer des commentaires dans les fichiers *pl*. Il suffit de les faire précéder par %.

Tests et unification

Tests entre variables :
On peut réaliser des tests entre variables de la même façon qu'on ferait sur n'importe quel langage de programmation. Leur résultat est `true` ou false et leur `syntaxe` est la suivante :

- X==Y : `true` si X et Y sont égaux (même *atom*).
- X\==Y : `true` si X et Y ne sont **pas** égaux.
- X<Y ou X=<Y : évalue X et Y et renvoie `true` si X est plus petit (resp. plus petit ou égal) à Y.
- X>Y ou X>=Y : évalue X et Y et renvoie `true` si X est plus grand (resp. plus grand ou égal) à Y.

Unification :
L'unification fait correspondre une expression à une autre qui lui est similaire. Par exemple X=Y est une expression qui peut s'unifier à 2=3 car les deux expressions sont similaires. Après unification, X sera désormais 2 et Y sera désormais 3.

L'unification ne peut avoir lieu que si les éléments peuvent être changés pour faire la correspondance. Ainsi, 3+3 et 2+4 ne peuvent être unifiés même si les expressions sont similaires.

L'unification permet de faire un *Pattern Matching*, exemple :

- `X+Y` unifié à `m(p)*6+5/9` fait correspondre `m(p)*6` à `X` et `5/9` à `Y`.
- `X+Y` unifié à `6+7+8+9` fait correspondre `6` à `X` et `7+8+9` à `Y`.
- `Y*4` unifié à `0*X` fait correspondre `4` à `X` et `0` à `Y`.

Soient deux expressions `M` et `N` :

- `M=N` : `true` si `M` et `N` sont unifiables et, si oui, effectue l'unification.
- `M\=N` : `true` si `M` et `N` ne sont **pas** unifiables. S'ils le sont, l'unification ne se produit pas.
- `M is N` : évalue `N` et unifie son résultat à `M` (ie, stocke le résultat dans `M`).

Récursivité et listes

Listes :
Prolog nous permet de manipuler des listes. Leur syntaxe générale est la suivante :

```
[item1, item2, item3, …, itemM, itemP, …, itemN]
```

Pour travailler avec des listes il est important de savoir comment les diviser. Chaque liste peut en effet être unifiée à un élément de cette allure :

```
[Var1, Var2, …, VarM | B]
```

Si on venait à effectuer l'unification des deux listes en haut, nous auront la correspondance suivante :

- `Var1` prend la valeur `item1`, `Var2` prend la valeur `item2`, …, `VarM` prend la valeur `itemM`.
- `B` se voit faire assigner la liste `[itemP, …, itemN]`.

Avec cette manipulation, couplée à une formulation récursive, il est possible de manipuler les listes de la façon que l'on veut. Il n'y a aucun moyen d'indexer les listes ou d'itérer sur celles-ci.

Récursivité :
Une règle peut s'appeler elle-même dans son exécution. Dans la mesure où il n'y a pas de moyen de faire des structures itératives sur Prolog, la récursivité reste le seul moyen d'effectuer des opérations itératives.

Les règles récursives ont souvent besoin d'au moins deux définitions : une formulation récursive et un *état de base*. La récursivité s'arrête quand cet état de base est atteint.

Soit l'exemple suivant où on veut chercher l'*atom* 3 dans la liste `[0, 6, 2, 3, 9, 4]` et le mettre dans une variable :

```
lookFor3([3|_], 3).
lookFor3([L|B], R):- lookFor3(B, R).
```

La deuxième définition va, à chaque fois, enlever un élément à la liste et s'appeler elle-même sur le restant de la liste B. A chaque appel subséquent, Prolog consulte la base de connaissances du haut vers le bas. Quand `lookFor3` va s'appeler sur une liste qui débute par 3, le premier littéral va être vrai. Celui-ci marque la fin de la branche dans l'arbre de recherche et assigne au deuxième paramètre (quand celui-ci est une variable) l'*atom* 3.

Notez que l'ordre ici est **important** : avec l'ordre inverse, `lookFor3` va vider la liste, arriver à une impasse et backtrack à la recherche d'un autre littéral qui peut être satisfait. Dans ce cas, le résultat reste le même,

mais la recherche est inutilement plus longue (vous pouvez essayer de visualiser cela sur vos machines avec `trace`). Dans d'autres cas, cela peut produire des erreurs. Le code, tel que présenté en haut, trouve une réponse aussitôt qu'une liste avec un 3 au début est rencontrée.

Cut :
Quelquefois, dans certaines formulations récursives (ou non d'ailleurs), on désire arrêter la recherche dans l'arbre aussitôt qu'une branche de celui-ci est explorée à terme. En d'autres mots, on ne veut pas qu'il y ait de backtracking à la recherche d'une autre solution. On utilise le prédicat `!/0`, aussi appelé *cut*.

Soit l'exemple suivant d'une règle qui est vraie si un X est membre d'une liste :

```
isMember(X, [X|_]).
isMember(X, [Y|B]):- isMember(X, B).
```

Écrit comme cela, notre formulation marche. Mais elle est inutilement inefficace, une fois de plus, car toute la liste va être testée à la recherche de l'élément X. Pour arrêter la recherche au bon moment, on reformule la première règle :

```
isMember(X, [X|_]):- !. % !/0 est un littéral comme les autres, vrai par déf
isMember(X, [Y|B]):- isMember(X, B).
```

Exercice 1 : Détective

Soit les affirmations suivantes :

- L'Anglais habite à la maison rouge.
- L'Espagnol a un chien.
- Dans la maison verte, on boit du café.
- L'Ukrainien boit du thé.
- La maison verte est immédiatement à droite de la maison blanche.
- Le sculpteur élève des escargots.
- Le diplomate habite la maison jaune.
- Dans la maison du milieu, on boit du lait.
- Le Norvégien habite à la première maison à gauche.
- Le médecin habite dans une maison voisine de celle où demeure le propriétaire du renard.
- La maison du diplomate est à côté de celle où il y a un cheval.
- Le violoniste boit du jus d'orange.
- Le Japonais est acrobate.
- Le Norvégien habite à côté de la maison bleue.

Répondez aux questions suivantes…

- Qui boit de l'eau ?
- Qui possède le zèbre ?

… en considérant le problème comme un CSP et en créant un modèle adéquat sur *MiniZinc*.

Exercice 2 : Round-Robin

Soit *N* équipes de football dans un certain tournoi. Lors de ce tournoi, chaque équipe joue contre toutes les autres une seule fois exactement. On appelle ce genre de tournoi un *Round-Robin tournament*.

Lors de ce tournoi, chaque tour est marqué par la confrontation de toutes les équipes, ie, le tournoi se fait sur *N* – 1 tours.

En plus de cela, chaque équipe a son propre stade de football. Une équipe est dite jouer à domicile si elle accueille son adversaire sur son stade. Sinon, on dit que l'équipe joue à l'extérieur.

On fournit le fichier `N14a.dzn` avec une variable `N` qui donne la taille du problème (ici 14) et un tableau `location` tel que `location[i,j] = 1` si `i` joue contre `j` à domicile ou `location[i,j] = 0` si `i` joue contre `j` à l'extérieur (ie, `j` joue à domicile).

→ Trouvez un calendrier adéquat pour ce tournoi si, en plus, **on exige qu'aucune équipe ne peut jouer 4 matchs successifs (ou plus) à domicile ni 4 matchs successifs (ou plus) à l'extérieur**.

Définissez un modèle adéquat sur *MiniZinc* pour ce CSP et résolvez-le.

Expliquez une symétrie dans le problème et ajouter une contrainte redondante afin de la briser. Qu'observez-vous quant au temps de résolution ? Pourquoi une si grande différence ?

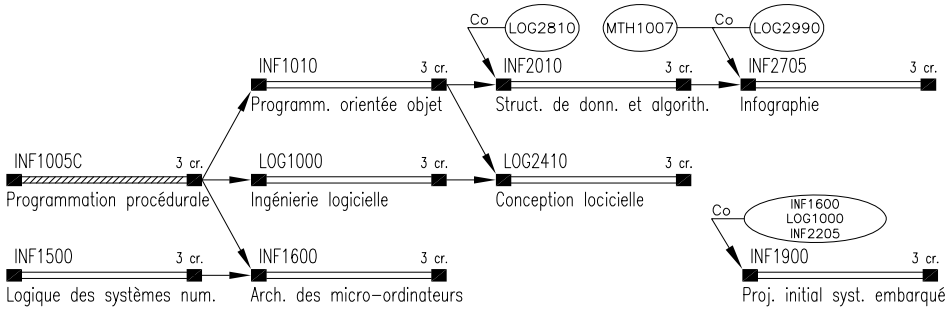
Bonus

Cherchez une **contrainte globale** qu'on peut appliquer afin de faire respecter l'exigence du nombre de matchs à domicile/à l'extérieur. Expliquez ce qu'elle fait et pourquoi son usage serait meilleur. **Vous n'êtes pas tenus de l'implanter** pour recevoir les points alloués à ce bonus mais la qualité et la complétude de votre explication influencera le nombre de points donnés.

Exercice 3 : Cours à prendre

Description

Soit l'agencement de cours suivant :



On désire développer un programme *Prolog* qui donne, pour un cours donné, tous les cours à suivre avant d'être éligible de le prendre.

Exemple

Pour le cours INF1900, le résultat devra être : INF1600, INF2205, LOG1000, INF1005C, INF1500.

Procédure

Définissez une base de connaissances adéquate et efficace. Celle-ci ne doit comporter que des prédicats de **deux paramètres au maximum**.

Écrivez une règle `coursAPrendreComplet/?` qui permet d'avoir l'énumération complète de tous les cours prérequis et corequis à un cours donné. Libre à vous d'implanter cela comme bon vous semble. Vous avez le droit d'utiliser des fonctions intégrées à Prolog. **Pas d'usage de libraires externes**.

A la fin, on exige une sortie claire dont l'interprétation sera facile. **On ne permet pas d'avoir des doublons ou autres artéfacts**.

Exercice 4 : Akinator

Description

On développe ici une base de connaissances avec *Prolog* qui sera utilisée pour un jeu de devinettes. Lorsque ce programme s'exécute, il demande de penser à une personne connue ou à un objet pouvant se trouver dans une maison. En posant des questions auxquelles on peut répondre par *oui* ou par *non*, ce programme essaye de deviner de quelle personne ou de quel objet il s'agit.

En fait, plus précisément, il s'agira de deux programmes distincts, chacun ayant sa base de connaissances : un programme pour les personnes et un autre pour les objets.

Liste des personnes :

Michael Jackson
Mikhail Gorbachev
Jennifer Lawrence
Hideo Kojima
Banksy
Lara Croft
Mario
J. K. Rowling
Lady Gaga
Quentin Tarantino
Joseph Staline
Dwight D. Eisenhower
Cléopâtre
Victor Hugo
Jésus
Ayrton Senna
Moise
Fernando Alonso
Pape François
James Bond
Denzel Washington
Richard Nixon

Objets :

Aspirateur
Ordinateur
Téléphone
Fourchette
Balai
Cactus
Assiette
Four
Cuisinière
Cafetière
Grille-pain
Table
Casserole
Shampooing
Détergent à vaisselle
Lit
Clé
Portefeuille
Sac à dos
Piano
Lampe
Papier

Considérations importantes

Vous devrez vous assurer que votre base de connaissances soit suffisamment bien construite et flexible pour faciliter l'ajout de nouvelles entités par la suite.

Par ailleurs, votre base de connaissances doit être compacte et votre programme devra poser un nombre minimal de questions afin d'identifier votre personne/objet.

Exemple

Pour vous aider à démarrer, voici un petit exemple, comprenant la description de 4 personnes :

```
1  ask(gouverne, Y) :-
2        format('~w gouverne ? ', [Y]),
3        read(Reponse),
4        Reponse = 'oui'.
5  ask(musicien, X) :-
6        format('~w est un musicien ? ', [X]),
7        read(Reponse),
8        Reponse = 'oui'.
9  ask(chanteur, X) :-
10       format('~w est un chanteur ? ', [X]),
11       read(Reponse),
12       Reponse = 'oui'.
13
14 personne(X) :- politicien(X).
15 personne(X) :- artiste(X).
16
17 artiste(X) :- ask(chanteur, X), chanteur(X).
18 artiste(X) :- ask(musicien, X), musicien(X).
19
20 politicien(X) :- gouverne(X, Y), pays(Y), ask(gouverne, Y).
21
22 chanteur(celine_dion).
23 musicien(john_lewis).
24 gouverne(stephen_harper,canada).
25 gouverne(barack_obama,usa).
26 pays(canada).
27 pays(usa).
```

Exécution

Pour exécuter le programme, il suffira de l'interroger comme ceci, selon le cas de figure :

```
?- personne(X).
?- objet(X).
```

Aussi, on devrait pouvoir déterminer l'appartenance d'une entité à une certaine classe en interrogeant directement la base de connaissances. Par exemple, on peut formuler la requête suivante :

```
?- appareil_electromenager(four_micro_onde).
```

Et celle-ci devrait être vraie dans ce cas de figure.

Instructions de remise

Rédigez un rapport PDF détaillant votre méthode, vos motivations, vos choix et vos implantations. **N'y incluez pas de code** : le rapport doit servir de compagnon à celui-ci. **Soyez les plus brefs et les plus précis possible**.

Remettez ce rapport, ainsi que vos fichiers *pl* et *mzn*, sur Moodle, dans une archive.

→ *Date limite de remise : 11 novembre 2018, 23h55.*

Barème

- Exercice 1 : 2.5 pts
- Exercice 2 : 6 pts + 2 pts
- Exercice 3 : 6.5 pts
- Exercice 4 : 5 pts

** Ce TP a été inspiré en partie par le travail de Gilles Pesant, de Pierre Hulot et de Rodrigo Randel.*