Université de Montpellier - Master informatique Janvier 2023

Théorie des bases de données et de connaissances (HAI933I)

Examen

Durée totale : 2 heures
Document autorisé : 1 feuille A4 manuscrite recto-verso

Toutes vos réponses doivent être justifiées. Une réponse sans justification ne sera pas prise en compte. Le barème est donné à titre indicatif et peut varier légèrement.

Exercice 1 (Answer Set Programming) - 8 pts

Remarques préliminaires Dans ce devoir, j'utilise (et vous utiliserez) la syntaxe et la sémantique de l'extension des règles existentielles qu nous avons vue en cours. Si elle ressemble à la syntaxe ASP que vous avez vue avec Clingo, il existe de petites différences que je souligne ici.

- il n'y a dans notre variante du langage aucun moyen d'exprimer la disjonction de façon syntaxique. En Clingo, `p(X), q(X).` est une disjonction et `p(X). q(X).` une conjonction. Nous n'utilisons *que* la conjonction et adoptons ici la seconde notation pour éviter les ambiguités. De la même manière, Clingo considère les atomes de la tête de `p(X), q(X) :- r(X).` comme une disjonction, alors que nous les considérons comme une conjonction.
- une règle peut pour avoir plusieurs corps négatifs, chacun composé de plusieurs atomes, comme cela a été défini dès le début du cours (voir règle de la question 1).
- contrairement à Clingo, nous autorisons dans un corps négatif des variables qui n'apparaissent pas dans le corps positif (mais ces variables ne doivent pas apparaitre en tête de règle). Voir par exemple la règle en question 3.

Question 1 (Application de règles) - 1 pt On considère le programme ASP suivant:

```
p(a, b). q(b).
p(a, c). r(c, a).
p(a, d). s(d).
p(a, e). r(e, a). s(e).
t(Y) :- p(X, Y), not q(Y), not r(Y, X), s(Y).
```

En justifiant pourquoi la règle est déclenchable / applicable, construisez une dérivation complète de ce programme.

Question 2 (Un titre aiderait trop) - 1 pt En utilisant le programme de la question 1, et les théorèmes vus en cours, justifiez pourquoi le résultat de votre dérivation est le seul modèle stable de ce programme.

Question 3 (Propositionalisation) - 1 pt Mettez sous forme propositionnelle le programme suivant. Vous justifierez chaque étape de la transformation, suivant l'algorithme vu en cours.

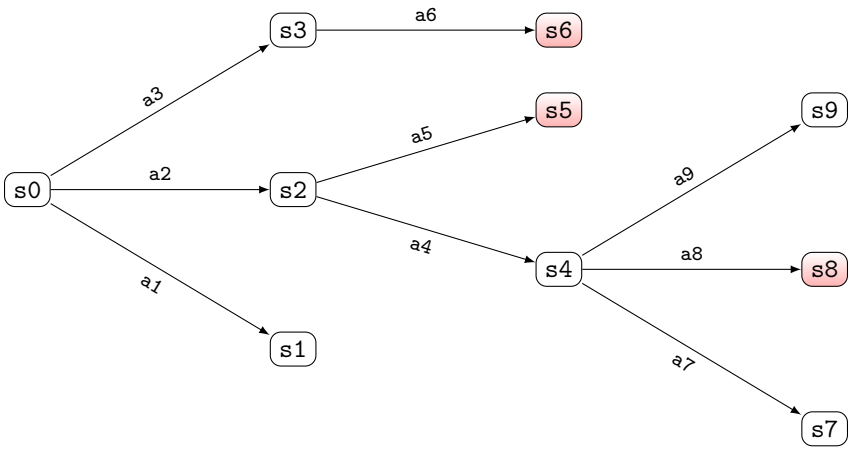
```
p(a). q(a, b).
s(X) :- p(X), not q(X, Y), r(Y), not r(X).
```

Question 4 (Point fixe) - 1 pt En utilisant le programme réduit et la définition par point fixe, dire si {`a, c`} est un modèle stable du programme suivant. Vous justifierez tout particulièrement la construction du programme réduit.

```
a.
c :- a, not b, c.
b :- a, not c.
c :- a, not b.
c :- b.
```

Question 5 (Asperix) - 1 pt En utilisant l'algorithme ASPERIX cu en cours, donnez tous les modèles stables du programme de la question 4.

Question 6 (Modélisation) - 3 pts L'objectif de cette question est de construire, en ASP, des éléments d'un moteur de jeu tour par tour à 2 joueurs. Les données initiales du jeu sont un atome `players(j1, j2)` indiquant que `j1` et `j2` sont les deux joueurs et on suppose qu'on a pu construire un arbre des états du jeu: l'atome `state(s)` indique que `s` est l'identificateur d'un état du jeu, et l'atome `next(s, a, s')` indique que l'on peut passer de l'état `s` à l'état `s'` par l'action d'identificateur `a`. Vous ne vous occuperez pas ici de la construction de cet arbre, puisqu'on ne sait même pas à quel jeu on joue.



Vous pouvez voir ci-dessus un exemple de la représentation graphique d'un tel arbre d'états (il s'agit bien d'un arbre, avec en particulier unicité de la racine et unicité du chemin allant de la racine à un état quelconque). Il sera encodé dans notre programme par les atomes suivants:

```
state(s0). state(s1). state(s2). state(s3). state(s4). state(s5). state(s6).
state(s7). state(s8). state(s9).
next(s0, a1, s1). next(s0, a2, s2). next(s0, a3, s3).
next(s2, a4, s4). next(s2, a5, s5).
next(s3, a6, s6).
next(s4, a7, s7). next(s4, a8, s8). next(s4, a9, s9).
```

a) Si les joueurs ont été donnés par l'atome `players(j1, j2)`, c'est le joueur `j1` qui joue en premier (qui doit jouer à l'état racine de l'arbre). Si un joueur joue à un état `s`, alors c'est l'autre joueur qui doit jouer à tout état `s'` successeur de `s` (c'est à dire tel que `next(s, a, s')`).

Ecrivez les règles permettant de générer tous les atomes `turn(s, j)` indiquant que le joueur `j` doit jouer quand on est dans l'état `s`. Attention, vous devrez utiliser dans vos règles une caractérisation de la racine.

Dans notre exemple, le premier joueur (`j1`) doit jouer aux états `s0` et `s4`, tandisque le second doit jouer aux états `s2` et `s3`. Il n'est pas très important de savoir qui joue sur les états feuilles, mais votre réponse devra être cohérente avec les réponses aux questions suivantes.

b) On suppose maintenant que certaines feuilles de l'arbre ont été étiquetées par un atome `win(s)` indiquant que l'état `s` est gagnant pour le premier joueur (et on suppose que les autres feuilles sont un état perdant pour lui). Il faut maintenant inférer quels sont les états (autres que les feuilles) qui sont gagnants pour le premier joueur (c(est à dire pour lesquels on est certain qu'il y a une stratégie gagnante)). Nous utiliserons la propriété suivante:

- si c'est au tour du premier joueur de jouer à l'état `s`, alors cet état est gagnant si il a un successeur gagnant.
- sinon, cet état est gagnant si tous ses successeurs sont gagnants.

Ecrivez les règles permettant de générer tous les atomes `win(s)` indiquant que l'état `s` est gagnant pour le premier joueur.

Dans notre exemple, les états `s5`, `s6` et `s8` ont été évalués gagnants par une autre partie de notre programme de jeu, qui a généré les atomes suivants:

```
win(s5). win(s6). win(s8).
```

Votre programme devra alors générer les atomes `win(s4)`, `win(s2)`, `win(s3)` et `win(s0)`.

c) Notons que, pour l'instant, notre programme est entièrement déterministe et ne génère qu'un seul modèle stable. Ecrire les règles qui permettent de générer les modèles stables contenant chacun un des coups gagnants: si un tel modèle stable contient l'atome `choix(a)`, cela veut dire qu'il y a un état gagnant successeur de la racine accessible par l'action `a`.

Toujours sur notre exemple, votre programme devra générer 2 modèles stables, le premier contenant `choix(a2)` (car `a2` est un "coup gagnant" quand on est dans l'état `s0`) et le second contenant `choix(a3)`.