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 (Requêtes du premier ordre) - 2 pts

Considérez la base de données suivante :

Films

Titre	Réalisateur	Acteur
The Imitation Game	Tyldum	Cumberbatch
The Imitation Game	Tyldum	Knightley
...	...	...
Internet's Own Boy	Knappenberger	Swartz
Internet's Own Boy	Knappenberger	Lessig
Internet's Own Boy	Knappenberger	Berners-Lee
...	...	...
Dogma	Smith	Damon
Dogma	Smith	Affleck

Cinémas

Cinéma	Addresse	Téléphone
UFA	St. Petersburger Str. 24	4825825
Schauburg	Königsbrücker Str. 55	8032185
...	...	...

Programme

Cinéma	Titre	Heure
Schauburg	The Imitation Game	19:30
Schauburg	Dogma	20:45
UFA	The Imitation Game	22:45

Écrivez les requêtes suivantes en tant que "first-order queries" (un demi-point chacune) :

1. Qui sont les acteurs qui ont travaillé avec le réalisateur "Tyldum" ?
2. Lister les acteurs qui ont joué dans un film projeté à "Schauburg".
3. Découvrez tous les realisateurs qui n'ont pas travaillé avec le realisateur "Smith".
4. Ecrivez une requête booléenne (i.e., a Boolean query) pour déterminer si deux réalisateurs différents ont réalisé le même film.

Exercice 2 (Core et règles Datalog) - 2 pts

Question 1. Donner un *core* de l'ensemble d'atomes $\mathcal{A}$ ci-dessous (où la seule constante est a), ainsi qu'un homomorphisme de $\mathcal{A}$ dans ce core :

$$\mathcal{A} = \{p(a,x), p(a,y), p(x,x), p(y,z), r(x,t), r(z,t)\}$$

Question 2. Soit une base de faits F sans variables. Soit $\mathcal{R}$ un ensemble de règles positives Datalog. La saturation de F par $\mathcal{R}$ est-elle nécessairement un *core* ?

Question 3. Même question avec une base de faits F qui peut comporter des variables mais qui est un *core*.

Exercice 3 (Contraintes négatives) - 4,5 pts

On considère des *contraintes négatives*, dont on rappelle qu'elles sont de la forme $\forall \vec{x}\ (B[\vec{x}] \rightarrow \bot)$, où $\vec{x}$ est une liste de variables, $B[\vec{x}]$ est une conjonction d'atomes dont les variables sont exactement celles de $\vec{x}$, et $\bot$ est le symbole absurde (ayant la valeur faux dans toute interprétation). Dans les exemples, on ne mentionne pas les quantificateurs universels (comme pour les règles).

Question 1. Soit une base de connaissances $\mathcal{K} = (F, \mathcal{C})$ où F est une base de faits et $\mathcal{C}$ un ensemble de contraintes négatives. On n'a donc pas de règles, hormis les contraintes. Comment vérifier si $\mathcal{K}$ est *satisfiable* ?

Question 2 En considérant toujours une base de connaissances $\mathcal{K} = (F, \mathcal{C})$ où F est une base de faits et $\mathcal{C}$ un ensemble de contraintes négatives, donner une requête booléenne q (UCQ) telle que la réponse à q sur F est *oui* si et seulement si $\mathcal{K}$ est *insatisfiable*.

Illustrez votre réponse sur l'ensemble $\mathcal{C} = \{C_1, C_2, C_3\}$, avec :
$C_1 : p(x, y) \wedge p(y, x) \rightarrow \bot$
$C_2 : p(x, x) \rightarrow \bot$
$C_3 : s(x) \wedge p(x, y) \rightarrow \bot$

Question 3. Etant données deux contraintes négatives C_i et C_j, on dit que C_i est *plus forte* que C_j si toute base de faits qui satisfait C_i satisfait aussi C_j. Comment vérifier si une contrainte est plus forte qu'une autre ? Etant donné un ensemble de contraintes négatives $\mathcal{C}$, comment définiriez-vous les contraintes de $\mathcal{C}$ qui sont inutiles ?

Illustrez votre réponse sur l'ensemble $\mathcal{C} = \{C_1, C_2, C_3\}$.

Question 4. On considère maintenant une base de connaissances de la forme $\mathcal{K} = (F, \mathcal{R}, \mathcal{C})$, où $\mathcal{R}$ est un ensemble de règles existentielles. Donnez deux façons de vérifier si $\mathcal{K}$ est satisfiable, en vous appuyant soit sur le chainage avant (chase) soit sur le chaînage arrière (réécriture de requête).

Question 5. Les deux façons de procéder données à la question précédente sont-elles toujours applicables (c'est-à-dire quel que soit l'ensemble $\mathcal{R}$) ?

Indiquer notamment comment procéder avec l'ensemble $\mathcal{C} = \{C_1, C_2, C_3\}$ donné au-dessus et l'ensemble $\mathcal{R} = \{R_1, R_2\}$:
$R_1 : p(x, y) \rightarrow \exists z\ p(y, z)$
$R_2 : q(x, y) \rightarrow p(y, x)$
Détaillez les étapes de calcul.

Exercice 4 (Chase) - 3,5 pts

Soit la base de connaissances $\mathcal{K} = (F, \mathcal{R})$ avec :
$F = \{q(a), q(b), p(a, b)\}$
$R_1 : r(x) \rightarrow \exists z\ p(x, z) \wedge q(z)$
$R_2 : q(x) \rightarrow r(x)$

Question 1. Quel est le résultat du chainage avant simple (oblivious chase) sur $\mathcal{K}$?

Question 2. Quel est le résultat du chainage avant restreint (restricted chase) sur $\mathcal{K}$?

Rappel : le restricted chase n'effectue l'application d'une règle $R = B \rightarrow H$ selon un homomorphisme h de B dans la base de faits courante F_i que si h ne *s'étend pas* à un homomorphisme de H dans F_i.

Question 3. La base de faits F' ci-dessous (où z_0 est une variable) correspond-elle à un modèle universel de $\mathcal{K}$? (Justifier)
$F' = \{q(a), q(b), r(a), r(b), p(a, b), p(b, z_0), q(z_0), r(z_0), p(z_0, z_0)\}$

Question 4. $\mathcal{K}$ possède-t-elle un modèle universel fini ? (Justifier)

Exercice 5 (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.

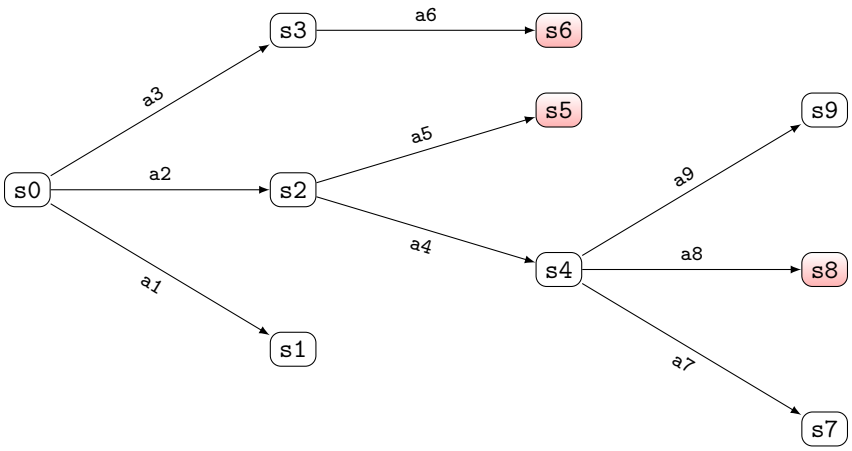
```
p(a). q(a, b).
s(X) :- p(X), not q(X, Y), r(Y), not r(X).
```

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)`.