Université de Montpellier - Master informatique Janvier 2023

Théorie des bases de données et de connaissances (HAI933I)

Correction partie "modèles stables"

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

Cette partie, notée sur 8 points, comprenait 5 points d'application directe des définitions et algorithmes vus en cours et 3 points sur une petite modélisation. Plus de la moitié des étudiants n'ont pas eu la moyenne sur la partie définition/algorithmes, ce qui dénote un manque de travail évident. Deux d'entre vous ont fait carton plein sur cette partie: bravo à eux. La partie modélisation n'a été que très peu abordée (manque de temps?).

Dans cette correction, les cadres verts sont mes commentaires sur les erreurs les plus répandues dans les copies, tandisque les cadres jaunes sont la correction en elle-même. Notez que cette correction est plus longue que nécessaire dans une copie: j'ai souhaité y apporter le plus de détails possibles.

Enfin, pour ceux qui passeront le rattrapage, je ne peux que vous encourager à travailler le cours et les exercices qui ont été donnés: aucune question posée dans cet examen n'aurait du vous prendre par surprise.

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.

Pour beaucoup d'entre vous, il aurait été utile de lire ces remarques préliminaires. En particulier, le deuxième point rappelle que nous considérons ici (comme défini dans le cours, par exemple dans Cours 1 Slide 12) des corps négatifs comprenant plusieurs atomes. La règle `p(X) :- q(X), not r(X), s(X)` doit donc se lire comme `p(X) :- q(X), not (r(X), s(X))` et non comme `p(X) :- q(X), s(X) not r(X).`

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.

La réponse ci-dessous est un peu verbeuse, afin de bien expliquer le blocage dans le cas de corps négatifs avec de multiples atomes, qui n'a pas été bien compris.

Notons F l'ensemble des faits du programme. Il y a 4 déclencheurs de la règle dans F: $d_1 = \{\texttt{X: a, Y: b}\}$, $d_2 = \{\texttt{X: a, Y: c}\}$, $d_3 = \{\texttt{X: a, Y: d}\}$ et $d_4 = \{\texttt{X: a, Y: e}\}$.

Le déclencheur d_1 est bloqué dans F par le premier corps négatif (`q(Y)`) car `q(b)` est dans F. La règle n'est donc pas applicable suivant d_1.

Le déclencheur d_2 n'est bloqué dans F ni par le premier corps négatif (car `q(c)` n'est pas dans F) ni par le second `r(Y, X), s(Y)`, car, même si on trouve `r(c, a)`, il faudrait trouver à la fois trouver `r(c, a)` et `s(c)` pour bloquer.

Pour les mêmes raisons, le déclencheur d_3 n'est pas bloqué dans F.

Le déclencheur d_4 est bloqué dans F par le second corps négatif car on trouve `r(e, a)` et `s(e)` dans F.

On peut donc trouver une dérivation $\mathcal{D}$ à partir de F; on applique la règle suivant d_2 pour générer $F_1 = F \cup \{t(c)\}$ puis suivant d_3 pour générer $F_2 = F \cup \{t(c), t(d)\}$. Aucune autre application de règle n'étant possible (l'ajout de `t(x)` et `t(d)` n'y change rien), la dérivation $\mathcal{D}$ est complète.

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.

Comme les déclencheurs d_2 et d_3 ne sont pas bloqués dans F_2, la dérivation $\mathcal{D}$ est persistante. Puisqu'elle est aussi complète (voir question 1), son résultat F_2 est donc un modèle stable du programme.

Il est immédiat de voir que le programme est stratifiable (on pourrait faire le dessin). Il n'admet donc qu'un seul modèle stable: c'est F_2.

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).
```

La tête de la règle ne contenant aucune variable existentielle, il n'y a pas besoin de skolémiser.

Le premier corps négatif `q(X, Y), r(Y)` contient une variable `Y` qui n'apparait pas dans le corps positif (ce qui n'est pas le cas du second corps négatif). Il y a donc besoin de normaliser. Nous obtenons:

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

Le domaine de Herbrand de notre programme étant $\mathcal{H} = \{a, b\}$, en instanciant le programme de toutes es façons possibles suivant $\mathcal{H}$, nous obtenons:

```
p(a). q(a, b).

aux(a) :- q(a, a), r(a). % avec {X:a, Y:a}
aux(a) :- q(a, b), r(b). % avec {X:a, Y:b}
aux(b) :- q(b, a), r(a). % avec {X:b, Y:a}
aux(b) :- q(b, b), r(b). % avec {X:b, Y:b}

s(a) :- p(a), not aux(a), not r(a).  % avec {X:a}
```

```
s(b) :- p(b), not aux(b), not r(b).  % avec {X:b}
```

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.
```

J'ai vu ici de trop nombreuses fois des programmes réduits dont les règles contenanient des corps négatifs! Je dois donc (encore) rappeler que le programme réduit est un programme *positif*, qui ne contient pas de `not`, ce qui est la seule raison de pouvoir assurer l'unicité du résultat de toutes les dérivations possibles.

Construisons le programme réduit par $E = \{a, c\}$ de notre programme Π.

Ce programme contient tous les faits et règles positives, c'est à dire `a` et `c :- b`. Examinons maintenant les règles avec corps négatif.

La règle `c :- a, not b, c` n'est pas bloquée dans E (on ne trouve pas *à la fois* `b` et `c` dans E). La forme positive de cette règle, `c :- a`, apparaît donc dans le programme réduit $\Pi_{|E}$.

La règle `b :- a, not c` est boquée dans E car `c` appartient à E. Cette règle n'apparaît donc pas dans $\Pi_{|E}$.

La règle `c :- a, not b` n'est pas bloquée dans E car `b` n'appartient pas à E. Sa forme positive `c :- a` apparaît donc dans $\Pi_{|E}$.

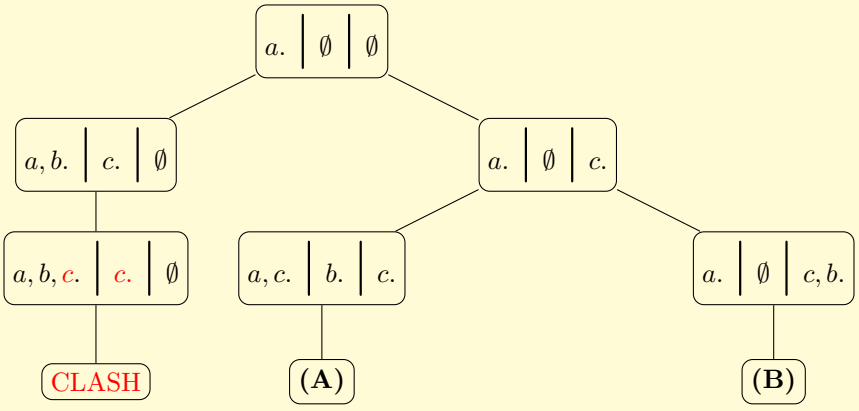
Le programme réduit $\Pi_{|E}$ est donc:

```
a.
c :- a.
c :- b.
```

La saturation de ce programme réduit produit $\Pi^*_{|E} = \{a, c\} = E$. Donc E est bien un modèle stable du programme.

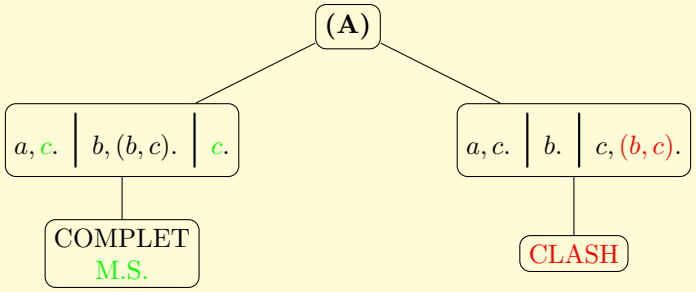
Lors de la construction du programme réduit, nous aurions pu également utiliser a définition alternative disant qu'on ne garde une règle (sous sa forme positive) que si aucun corps négatif n'est pas bloqué dans E *et son corps positif appartient à* E. Dans ce cas, la règle `c :- b` n'aurait pas fait partie du programme réduit (et ça n'aurait rien changé à sa saturation).

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

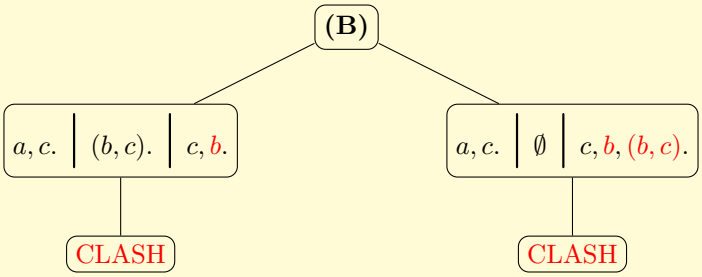


Cette partie de l'arbre ASPERIX est obtenue en appliquant la règle 2 (`b :- a, not c`) sur la racine, la règle positive 4 (`c :- b`) sur le fils gauche pour obtenir un CLASH et la règle 3 (`c :- a, not b`) sur le fils droit.

Beaucoup se sont arrêté là, concluant que **(A)** était un modèle stable tandisque **(B)** violait la partie MBT. J'ai généreusement accordé quelques points, mais pourtant on ne peut pas encore conclure. En effet, il reste la règle 1 (`c :- a, not (b, c)` – je rajoute les parenthèses pour plus de lisibilité) à évaluer sur **(A)** et sur **(B)**, ce qui n'a que trop rarement été fait.

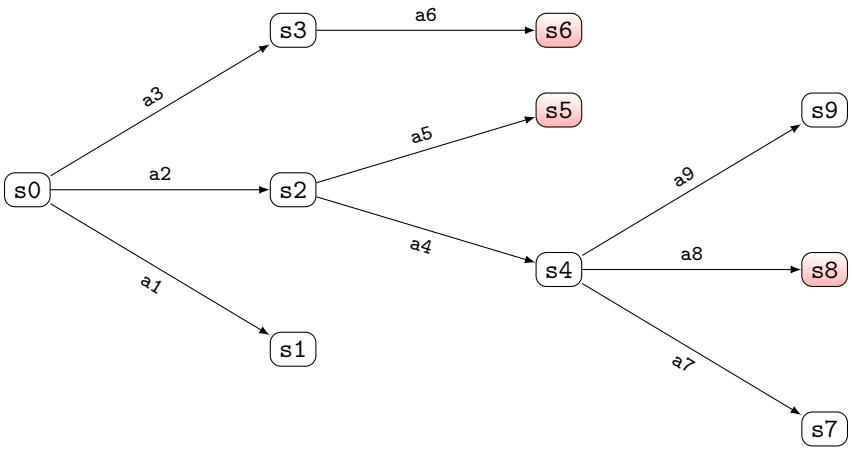


Nous avons ici évalué la règle 1 sur le noeud **(A)**. *Remarquez le parenthésage, par exemple dans la partie OUT du fils gauche. Il y aura violation du OUT si on trouve* `b` *dans le IN ou si on trouve à la fois* `b` *et* `c` *dans le IN.* Cette fois-ci, les branches sont complètes, et on n'a pas eu de violation du OUT. Le fils gauche satisfait tous les éléments du MBT, et donc correspond à un modèle stable, tandisque le fils droit viole le MBT: on ne trouve pas `(b, c)` – c'est à dire à la fois `b` et `c` – dans le IN.



L'évaluation de la règle 1 à partir du sommet **(B)** donne 2 autres branches complètes qui violent le MBT. Il n'y a donc qu'un seul modèle stable, qui est $\{a, c\}$.

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.

```
% Le premier joueur joue à l'état racine.
player1(J1), player2(J2) :- players(J1, J2).
root(S) :- state(S), not next(T, A, S).
turn(S, J) :- root(S), player1(J).

% Puis on alterne. J'ai choisi ici de ne pas jouer sur les feuilles.
leaf(S) :- state(S), not next(S, A, T).
turn(T, J2) :- turn(S, J1), players(J1, J2), next(S, A, T), not leaf(T).
turn(T, J1) :- turn(S, J2), players(J1, J2), next(S, A, T), not leaf(T).
```

Beaucoup de personnes ont ici adapté le programme du choix multiple vu en cours pour coder "*pour chaque état, il y a un des 2 joueurs qui joue*", menant à 2^s modèles stables (où s est le nombre d'états) alors que ce programme doit être complètement déterministe. Il fallait ici *(1)* faire jouer le joueur 1 à la racine – et pour cela détecter la racine; et *(2)* propager les joueurs en alternant. Parmi ceux qui ont fait *(1)*, beaucoup ont repéré la racine par `state(s0)`, or `s0` n'est le nom de la racine que dans l'exemple ...

J'ai volontairement écrit ma réponse ci-dessus de façon assez verbeuse, mais le programme pouvait être aussi simple que:

```
turn(S, J1) :- state(S), players(J1, J2), not next(T, A, S).
turn(T, J2) :- turn(S, J1), players(J1, J2), next(S, A, T).
turn(T, J1) :- turn(S, J2), players(J1, J2), next(S, A, T).
```

Attention, dans cette version simplifiée, on joue sur les feuilles, ce qui peut poser des problèmes.

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)`.

```
% cas de gain quand c'est le tour du premier joueur.
win(S) :- turn(S, J), player1(J), next(S, A, T), win(T).

% cas de gain quand c'est le tour du second joueur.
maynotwin(S) :- turn(S, J), player2(J), next(S, A, T), not win(T).
win(S) :- turn(S, J), player2(J), not maynotwin(S).
```

Le deuxième cas était le demi-point le plus compliqué de cette modélisation. Voir que si on a adopté la version du programme où on joue sur les feuilles, ce deuxième cas doit devenir:

```
% cas de gain quand c'est le tour du second joueur.
maynotwin(S) :- turn(S, J), player2(J), next(S, A, T), not win(T).
win(S) :- turn(S, J), player2(J), next(S, A, T), not maynotwin(S).
```

Car on ne veut pas rajouter de `win` sur les feuilles quand c'est au tour du joueur2 de jouer.

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)`.

```
% on suppose qu'il y a un coup gagnant pour joueur1
:- root(S), not win(S).

% on récupère les choix possibles.
possiblechoice(A) :- root(S), next(S, A, T), win(T).

% dans chaque MS, on choisit 1 parmi les possiblechoice
otherchoice(A) :- choice(B), possiblechoice(A), B != A.
choice(A) :- possiblechoice(A), not otherchoice(A).
```

Il suffisait ici de récupérer les choix possibles, puis d'adapter le sous-programme de disjonction n-aire vu de nombreuses fois pendant le cours.

Une dernière remarque, que je ne devrai pas avoir à faire à des étudiants: certaines copies sont pratiquement illisibles! J'ai vu des réponses sans indication du numéro de la question auxquelles elles étaient censé répondre, et même dans un cas un changement d'exercice qui n'était pas signalé ... Mention spéciale à celui qui a répondu deux fois à la même question, à deux endroits différents de la copie, en trouvant deux résultats différents ... Par bonheur, les deux versions étaient fausses, j'aurai été bien embêté pour la correction ...