Correction de l'examen final
Partie modèles stables

Jean-François Baget `baget@lirmm.fr`

16 janvier 2025

1 Modèles stables (9 points)

Dans ce document, vous trouverez à la fois le sujet du contrôle continu (sur fond blanc), le corrigé lui-même (dans des boîtes jaunes), des versions alternatives du corrigé (dans des boîtes oranges) et de nombreux commentaires (dans des boîtes vertes telles que celle-ci). Si vous avez des questions, n'hésitez pas à me contacter par mail.

A l'issue de la correction de la partie modèles stables, je suis très déçu par vos copies. La moyenne (ramenée à une note sur 20) est de $3,35$, seules 5 copies (sur 35) atteignent une note supérieure à 8, et 9 copies ont obtenu un zéro.

1.1 Application des algorithmes vus en cours (3 points)

Comme le titre l'indique, cette partie est une application directe du cours. Seule une étudiante a eu le maximum de points sur cette partie, félicitations à elle. Ce type d'exercice avait été vu plusieurs fois en cours, proposé au précédent contrôle continu, et j'en avais fait une correction détaillée. Le résultat est donc très décevant. J'avais insisté pour que vous fassiez plusieurs exercices de ce type, jusqu'à ce que ça devienne un automatisme. Visiblement, ça n'a pas été fait.

On se donne le programme de règles existentielles avec négation suivant:

```
p(1). p(2). p(3).
c1(X) :- p(X), not o1(X).
o1(X) :- c1(Y), p(X), X != Y.
c2(X) :- p(X), not c1(X), not o2(X).
o2(X) :- c2(Y), p(X), X != Y.
```

Question 1: mettre les deux premières règles de ce programme sous forme propositionnelle. Attention, dans votre programme propositionnel, *le symbole de différence devra disparaître*![1]

Les règles du programme sont déjà skolémisées (pas de variable existentielle) et sous forme normale (toutes les variables des corps négatifs sont dans le corps positif). Nous passons directement à l'instanciation, avec $\mathcal{H} = \{1, 2, 3\}$ pour domaine de Herbrand. Nous notons R, S, T, U les 4 règles du programme.

```
p(1). p(2). p(3).
```

[1] En effet, supposons une règle `q(X) :- p(X), X != b.` Si vous instanciez avec {`X: a`}, alors l'atome `X != b` sera évalué à vrai et inutile dans la règle qui deviendra `q(a) :- p(a).` Par contre, si vous instanciez avec {`X: b`}, alors l'atome `X != b` sera évalué à faux et c'est toute la règle qui deviendra inutile, car elle n'est jamais déclenchable.

```
[R1]   c1(1) :- p(1), not o1(1). % instanciation de R avec {X:1}
[R2]   c1(2) :- p(2), not o1(2).
[R3]   c1(3) :- p(3), not o1(3).

[S12] o1(1) :- c1(2), p(1). % instanciation de S avec {X:1, Y:2}
[S13] o1(1) :- c1(3), p(1).
[S21] o1(2) :- c1(1), p(2).
[S23] o1(2) :- c1(3), p(2).
[S31] o1(3) :- c1(1), p(3).
[S32] o1(3) :- c1(2), p(3).
```

Dans la règle `S12` (par exemple), nous n'avons pas rajouté l'atome $1 \neq 2$ car il est toujours vérifié. La règle `S11` (comme les règles `S22` et `S33`) est absente car l'atome $1 \neq 1$ n'est jamais vérifié.

J'ai commencé à m'inquiéter quand j'ai vu des horreurs pour cette première question.

- certains ont plus ou moins recopié l'énoncé, en laissant des variables dans les règles obtenues. Si il y a des variables, ce n'est pas un programme propositionnel, ce qui était demandé. Variation sur le même thème, j'ai vu une recopie de la règle `S` en enlevant l'atome $X \neq Y$, ce qui change complètement la sémantique du programme.

- plus de la moitié des étudiants ont normalisé les règles qui, rappelons-le, étaient déjà sous forme normale. Déjà, ça vous donne plus de boulot, ce qui n'est pas une bonne idée quand vous êtes pressé par le temps. Mais voyons un peu ce qui se passe:

  ```
  [R]    c1(X) :- p(X), not o1(X).
  % Normalisation, reecriture de R en Ra et rb
  [Ra]   c1(X) :- p(X), not aux1(X).
  [Rb]   aux1(X) :- o1(X).
  % Maintenant instanciation
  ...
  [Rb2] aux1(2) :- o1(2).
  ...
  ```

  J'ai quand même compté juste pour ceux qui m'ont (bien) fait cette normalisation inutile, mais ça pose problème à la question 2, puisque la règle `Rb2` sera dans le programme réduit, que la saturation devrait donc générer l'atome `aux1(2)`, et donc l'ensemble que j'ai donné ne sera pas un modèle stable. Et là, curieusement, votre saturation ignore délibérément cet atome, pour pouvoir dire que vous avez bien trouvé un modèle stable. Ceci me met en colère: 1) parce que vous êtes supposé être des scientifiques, et qu'en sciences on ne maquille pas ses résultats; et 2) parce que vous supposez que je ne vais pas le remarquer.

- à propos de la normalisation, j'ai vu des trucs absolument incompréhensibles comme:

  ```
  [R]    c1(X) :- p(X), not o1(X).
  % Normalisation, reecriture de R en Ra et rb
  [Ra]   c1(X) :- p(X), aux1(X).
  [Rb]   aux1(X) :- not o1(X).
  ```

  Ici, la règle `Rb` n'est pas une règle valide car il y a une variable de la tête qui n'apparait que dans les corps négatifs, ce qu'on avait explicitement interdit dans le cours. Pour faire une normalisation inutile, vous avez introduit un machin qui n'est pas une règle.

- enfin, il y a souvent eu une gestion très approximative de la différence, dont le principe était exposé dans une note de bas de page de l'énoncé. Une autre façon de faire aurait été de réécrire $X \neq Y$ en `not X = Y`, puis de faire comme indiqué dans la version suivante de la correction.

Ici, nous réécrivons chaque atome de la forme $X \neq Y$ en `not X = Y`. L'instanciation de la règle `S` est:

```
[S11] o1(1) :- c1(1), p(1), not 1 = 1. % avec {X:1, Y:1}
[S12] o1(1) :- c1(2), p(1), not 1 = 2.
[S13] o1(1) :- c1(3), p(1), not 1 = 3.
[S21] o1(2) :- c1(1), p(2), not 2 = 1.
[S22] o1(2) :- c1(2), p(2), not 2 = 2.
[S23] o1(2) :- c1(3), p(2), not 2 = 3.
[S31] o1(3) :- c1(1), p(3), not 3 = 1.
[S32] o1(3) :- c1(2), p(3), not 3 = 2.
[S33] o1(3) :- c1(3), p(3), not 3 = 3.
```

L'instanciation de `S` donne ici 9 règles au lieu de 6 dans la version précédente (ce qui donne plus de travail quand on calcule le programme réduit à la question 2). Voir que la version précédente est une simplification de celle-ci, en remarquant par exemple que `not 1 = 1` ne sera jamais vérifié (on peut donc supprimer la règle `S11`) ou que `not 1 = 2` sera toujours vérifié (on peut donc supprimer cet atome du corps de `S12`).

Question 2: en utilisant le programme de la question 1 et la définition par point fixe, prouvez que l'ensemble d'atomes:

`{p(1), p(2), p(3), c1(1), o1(2), o2(3), c2(2), o2(1), o2(3)}`

est un modèle stable du programme. Vous justifierez soigneusement votre réponse, notamment en exhibant le programme réduit qui, comme indiqué à la question 1, ne devra pas contenir de symbole de différence.

Curieusement, cette question a été mieux traitée que la précédente. Et si je dis curieusement, c'est qu'il me semble difficile de construire un programme réduit correct à partir d'un programme instancié faux. Je suspecte donc que l'algorithme utilisé est "construire le programme qui donne le résultat voulu", ce qui n'est pas vraiment ce que je voulais. Mais j'ai quand même accordé les points pour un programme réduit correct, indépendemment de ce qui avait été fait à la question précédente (et également pour une bonne réduction du programme de la question 1, même quand il était faux).

Attention, typo: comme Marie-Laure vous l'a signalé au cours de l'examen, il y avait une typo dans l'ensemble d'atomes de la question. Un des 2 `o2(3)` devait être `o1(3)`. Toutes mes excuses à ce sujet. Je n'ai pas enlevé de points à ceux qui avaient correctement répondu à la question, quel que soit l'ensemble d'atomes qu'ils avaient utilisé. Dans ma correction, j'utilise l'ensemble d'atomes corrigé, c'est à dire:

`E = {p(1), p(2), p(3), c1(1), o1(2), o1(3), c2(2), o2(1), o2(3)}`

Nous utilisons ici la version forte de la définition du programme réduit pour calculer le programme réduit par l'ensemble d'atomes E.

Programme propositionnel	Programme réduit	Justification
p(1), p(3), p(3).	p(1), p(2), p(3).	Fait
[R1] c1(1) :- p(1), not o1(1).	c1(1) :- p(1).	p(1) $\in E$ et o1(1) $\notin E$
[R2] c1(2) :- p(2), not o1(2).	–	o1(2) $\in E$
[R3] c1(3) :- p(3), not o1(3).	–	o1(3) $\in E$
[S12] o1(1) :- c1(2), p(1).	–	c1(2) $\notin E$
[S13] o1(1) :- c1(3), p(1).	–	c1(3) $\notin E$
[S21] o1(2) :- c1(1), p(2).	o1(2) :- c1(1), p(2).	c1(1) $\in E$ et p(2) $\in E$
[S23] o1(2) :- c1(3), p(2).	–	c1(3) $\notin E$
[S31] o1(3) :- c1(1), p(3).	o1(3) :- c1(1), p(3).	c1(1) $\in E$ et p(3) $\in E$
[S32] o1(3) :- c1(2), p(3).	–	c1(2) $\notin E$
[T1] c2(1) :- p(1), not c1(1), not o2(1).	–	c1(1) $\in E$
[T2] c2(2) :- p(2), not c1(2), not o2(2).	c2(2) :- p(2).	p(2) $\in E$ et c1(2) $\notin E$ et o2(2) $\notin E$
[T3] c2(3) :- p(3), not c1(3), not o2(3).	–	c1(3) $\in E$
[U12] o2(1) :- c2(2), p(1).	o2(1) :- c2(2), p(1).	c2(2) $\in E$ et p(1) $\in E$
[U13] o2(1) :- c2(3), p(1).	–	c2(3) $\notin E$
[U21] o2(2) :- c2(1), p(2).	–	c2(1) $\notin E$
[U23] o2(2) :- c2(3), p(2).	–	c2(3) $\notin E$
[U31] o2(3) :- c2(1), p(3).	–	c2(1) $\notin E$
[U32] o2(3) :- c2(2), p(3).	o2(3) :- c2(2), p(3).	c2(2) $\in E$ et p(3) $\in E$

La saturation du programme réduit donne l'ensemble d'atomes {p(1), p(2), p(3), c1(1), o1(2), o1(3), c2(2), o2(1), o2(3)} $= E$. L'ensemble E est donc bien un modèle stable du programme.

Parmi les erreurs les plus courantes:

- j'ai encore vu des programmes réduits contenant des corps négatifs. Or le programme réduit est toujours un programme positif. Il s'agit d'un point sur lequel j'avais déjà insisté lors de la précédente correction.
- j'ai trop souvent vu des programmes réduits dont la saturation ne donnait trivialement pas E, mais l'étudiant affirme cependant avec applomb que E est bien le résultat. Ce n'est pas honnête (voir remarque précédente). Je préfère voir "la saturation devrait donner E, ce n'est pas le résultat que j'obtiens, il doit y avoir une erreur que je ne vois pas quelque part".
- même la saturation est parfois mal calculée: j'ai vu 2 ou 3 étudiants qui réussissaient à appliquer S21 avant R1...

Pour ceux qui ont utilisé la version faible de la définition du programme réduit (ce qui n'est pas une bonne idée puisque ça donne plus de règles dans le programme réduit), la correction devrait ressembler à ça:

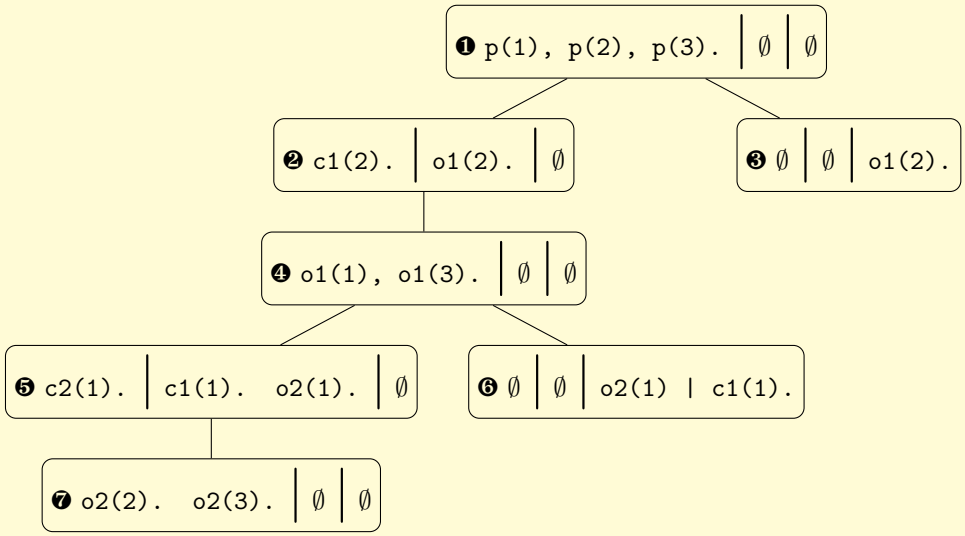
Nous utilisons ici la version faible de la définition du programme réduit pour calculer le programme réduit par l'ensemble d'atomes E.

Programme propositionnel	Programme réduit	Justification
p(1), p(3), p(3).	p(1), p(2), p(3).	Fait
[R1] c1(1) :- p(1), not o1(1).	c1(1) :- p(1).	o1(1) $\notin E$
[R2] c1(2) :- p(2), not o1(2).	–	o1(2) $\in E$
[R3] c1(3) :- p(3), not o1(3).	–	o1(3) $\in E$
[S12] o1(1) :- c1(2), p(1).	o1(1) :- c1(2), p(1).	Règle positive
[S13] o1(1) :- c1(3), p(1).	o1(1) :- c1(3), p(1).	Règle positive
[S21] o1(2) :- c1(1), p(2).	o1(2) :- c1(1), p(2).	Règle positive
[S23] o1(2) :- c1(3), p(2).	o1(2) :- c1(3), p(2).	Règle positive
[S31] o1(3) :- c1(1), p(3).	o1(3) :- c1(1), p(3).	Règle positive
[S32] o1(3) :- c1(2), p(3).	o1(3) :- c1(2), p(3).	Règle positive
[T1] c2(1) :- p(1), not c1(1), not o2(1).	–	c1(1) $\in E$
[T2] c2(2) :- p(2), not c1(2), not o2(2).	c2(2) :- p(2).	c1(2) $\notin E$ et o2(2) $\notin E$
[T3] c2(3) :- p(3), not c1(3), not o2(3).	–	c1(3) $\in E$
[U12] o2(1) :- c2(2), p(1).	o2(1) :- c2(2), p(1).	Règle positive
[U13] o2(1) :- c2(3), p(1).	o2(1) :- c2(3), p(1).	Règle positive
[U21] o2(2) :- c2(1), p(2).	o2(2) :- c2(1), p(2).	Règle positive
[U23] o2(2) :- c2(3), p(2).	o2(2) :- c2(3), p(2).	Règle positive
[U31] o2(3) :- c2(1), p(3).	o2(3) :- c2(1), p(3).	Règle positive
[U32] o2(3) :- c2(2), p(3).	o2(3) :- c2(2), p(3).	Règle positive

La saturation du programme réduit donne l'ensemble d'atomes {`p(1), p(2), p(3), c1(1), o1(2), o1(3), c2(2), o2(1), o2(3)`} $= E$. L'ensemble E est donc bien un modèle stable du programme.

Question 3: Dessinez (une partie de) l'arbre de recherche ASPERIX utilisant le programme non instancié initial. Vous pourrez arrêter l'arbre de recherche quand il vous permettra de répondre à la question suivante: *le programme initial est-il stratifiable?* Vous justifierez soigneusement pourquoi vous vous êtes arrêté, mais n'avez pas droit au graphe de dépendance des prédicats pour justifier que le programme n'est pas stratifiable.

Nous avons représenté ici une partie de l'arbre de recherche ASPERIX. Les sommets contiennent un identifiant (par exemple ❶) suivi des trois champs IN, OUT et MBT. Pour des raisons de place, nous n'avons indiqué dans chaque sommet que ce qui a été rajouté à la construction de ce sommet.



Cet arbre de recherche a été construit de la façon suivante:

- le sommet racine ❶ est initialisé avec les faits du programme.
- on évalue la règle `R` sur le sommet ❶ avec l'homomorphisme {`X: 2`} pour obtenir les sommets ❷ (application) et ❸ (non application).
- on évalue la règle `S` sur le sommet ❷, tout d'abord avec l'homomorphisme {`X: 1, Y: 2`} puis avec l'homomorphisme {`X: 3, Y: 2`} pour obtenir le sommet ❹. En toute rigueur, lorsqu'on évalue cette règle avec le premier homomorphisme, il faudrait un successeur **a** pour l'application, dont le champ OUT contient `1 = 2` et un successeur **b** pour la non application, dont le champ MBT contient `1 = 2`. Comme on n'arrivera jamais à prouver `1 = 2`, le sommet **b** ne peut mener à aucun modèle stable, donc on n'a pas besoin de le représenter. Pour la même raison, on n'a pas besoin de mettre `1 = 2` dans le champ OUT du sommet **a**, d'où notre simplification.
- on évalue la règle `T` sur le sommet ❹ avec l'homomorphisme {`X: 1`} pour obtenir les sommets ❺ (application) et ❻ (non application).
- on évalue la règle `U` sur le sommet ❺, tout d'abord avec l'homomorphisme {`X: 2, Y: 1`} puis avec l'homomorphisme {`X: 3, Y: 1`} pour obtenir le sommet ❼. Même remarque pour la gestion de la différence.

La branche de gauche est complète (plus aucune règle n'est applicable sans être bloquée), aucune contrainte de OUT n'est violée, et toutes les contraintes de MBT sont satisfaites. Le résultat de cette branche (l'union de tous ses champs IN) est donc un modèle stable du programme.

`E'={p(1), p(2), p(3), c1(2), o1(1), o1(3), c2(1), o2(2), o2(3)}` est un modèle stable.

Ce modèle stable est distinct de celui trouvé à la question 2, donc notre programme admet au moins deux modèles stables, et il n'est donc pas stratifiable.

Cet exercice a dans l'ensemble été très mal fait, ce qui est surprenant puisque j'avais insisté sur cet algorithme qui tombe tous les ans à l'examen. Ce n'est pas qu'une question de temps, vous pouvez voir que l'arbre que j'obtiens est petit (7 sommets). Je ne demandais pas non plus d'expliquer la construction de cet arbre, comme je l'ai fait dans ma correction pour vous aider à la comprendre.

Point positif: même sans avoir calculé l'arbre de recherche, plusieurs étudiants ont affirmé que "si l'arbre nous donne un modèle stable différent de celui de la question 2, alors on peut arrêter puisqu'on aura prouvé que le programme n'est pas stratifiable." Malin, ça permettait de gratter un demi point sans perdre de temps sur l'arbre de recherche.

Une question qui n'a finalement pas été posée à l'examen était celle-ci:

Question: Expliquez précisément en langue naturelle et en 2 à 3 lignes ce que font les règles du programme.

Il fallait comprendre que, si on a les faits `p(1), p(2), ..., p(n).` qui représentent n choix possibles, le programme génère $n(n-1)$ modèles stables contenant un unique premier choix `c1(i)` (avec $1 \leq i \leq n$) et un unique second choix distinct `c2(j)` (avec $1 \leq j \leq n$ et $i \neq j$).

Comprendre ceci permettait d'écrire facilement la réponse à la question la plus compliquée de la partie modélisation.

1.2 Modélisation: le problème du pont (4 points)

Nous nous proposons dans cette partie de faire résoudre à Clingo un petit problème combinatoire. Je rappelle que si (1) les règles n'ont qu'un seul atome en tête et (2) les règles ne contiennent pas de variable existentielle, alors les sémantiques de Clingo et des règles existentielles avec négation coïncident. Dans les questions suivantes, vous vous conformerez à ces restrictions.

Il est dommage que si peu d'étudiants aient abordé cette partie modélisation, puisque ceux qui l'ont fait ont pu y grapiller pas mal de points.

Le problème du pont: *Quatre personnes doivent traverser un pont en 17 minutes. Chacune d'entre elles marche à une vitesse maximale donnée. Appelons 1, la personne qui peut traverser le pont en 1 minute, 2 celle qui le traverse en 2 minutes, 5 celle qui le fait en 5 minutes et 10 celle qui le traverse en 10 minutes. Ces quatre personnes n'ont en tout qu'une torche et il est impossible de traverser le pont sans torche. Le pont ne peut supporter que le poids de 2 personnes. Dans quel ordre doivent traverser ces quatre personnes ?*

Vous ne répondrez pas à la question "*Dans quel ordre doivent traverser ces quatre personnes ?*", mais écrirez un programme qui permet à Clingo de le faire.

La base de faits et le vocabulaire: la base de faits est la suivante:

```
position(1, side1, 0). position(2, side1, 0).
position(5, side1, 0). position(10, side1, 0).
torch(side1, 0).
opposite(side1, side2). opposite(side2, side1).
```

Ici, `position(1, side1, 0).` signifie que la personne `1` (celle qui met 1 minute à traverser le pont) se trouve du côté 1 (`side1`) du pont au temps `0`; `torch(side1, 0).` signifie que la torche est du côté 1 au temps `0`, et les prédicats `opposite` vous aideront par la suite à écrire moins de règles.

Question 4: Afin de lancer la partie du programme qui va choisir la ou les personnes qui vont traverser avec la torche, nous avons la règle suivante:

```
init(S, N) :- torch(S, N), not win(N).
```

qui dit que si la torche est sur le côté `S` au temps `N` et qu'on n'a pas encore gagné au temps `N`, alors on en déduit l'atome `init(S, N)` qui sera utilisé pour indiquer qu'il faut faire traverser 1 ou 2 personnes depuis le côté `S` au temps `N`. Mais pour celà, il nous faut écrire une règle qui encode les conditions de succès.

Ecrire la règle dont la tête est `win(N)` et qui est applicable lorsque les conditions de succès sont vérifiées au temps `N`.

```
win(N) :- position(1, side2, N), position(2, side2, N),
          position(5, side2, N), position(10, side2, N),
          N <= 17.
```

La condition `N <= 17` a souvent été oubliée.
On aurait pu écrire ceci d'une autre manière, plus élégante quand on a des milliers de personnes à faire traverser.

```
continue(N) :- position(X, side1, N).
win(N) :- position(X, side2, N), N <= 17, not continue(N).
```

Question 5: En adaptant le plus simplement possible le programme de la question 1, compléter le programme écrit jusqu'à présent avec des règles qui ont pour effet, lorqu'on a déduit `init(W, N)`, de générer, pour chaque paire {`P1, P2`} de personnes situées sur le côté `W` un candidat modèle stable contenant les atomes `chose1(P1, N)` et `chose2(P2, N)`. Vous discuterez de l'opportunité de spécifier, comme à la question 1, que `P1` doit être différent de `P2`. Vous pourrez utiliser les prédicats auxiliaires `otherchoice1` et `otherchoice2`. *Attention*, vous vous faciliterez grandement les questions suivantes si vous réussissez à imposer que `P1` soit plus lent que `P2`.

```
chose1(P, N) :- init(W, N), position(P, W, N), not otherchoice1(P, N).
otherchoice1(P1, N) :- position(P1, W, N), chose1(P2, N), P1 != P2.
chose2(P2, N) :- init(W, N), chose1(P1, N),
                 position(P2, W, N), P1 <= P2, not otherchoice2(P2, N).
otherchoice2(P1, N) :- position(P1, W, N), chose2(P2, N), P1 != P2.
```

C'est vraiment le schéma de règle de la première partie de cet examen, et ceux qui ont abordé cette question l'ont pas mal fait. Quelques remarques:

- la première règle sélectionne `P` le premier candidat à la traversée au temps `N`. Il doit être du même côté que la torche pour être un candidat possible (`init(W, N)` et `position(P, W, N)`)

- la troisième règle selectionne `P2` le second candidat à la traversée au temps `N`. Notons que, contrairement au premier exercice, on n'impose pas au second candidat d'être différent du premier, ce qui est voulu. Sélectionner deux fois le même candidat permet de simuler la traversée d'une seule personne. Par contre, nous imposons `P1 <= P2`, ce qui a deux avantages. Tout d'abord, ceci permet d'éviter des symétries et accélère le calcul, puisque nous pouvons avoir `choice1(1), choice2(5)`, mais pas `choice1(5), choice2(1)`. Ensuite, puisque le premier candidat est le plus lent, il sera plus facile à la question suivante de mettre à jour le temps de la traversée.

Question 6: Maintenant que vous avez choisi la ou les personnes qui traversent au temps `N`, écrivez une ou plusieurs règles qui mettent à jour les positions des personnes et de la torche au temps `N + V`, où `V` est la vitesse de la plus lente des personnes qui ont traversé au temps `N`.

```
% mise à jour de la torche
torch(W2, N+P):- chose1(P, N), torch(W1, N), opposite(W1, W2).
% mise à jour du premier traverseur
position(P, W2, N+P) :- chose1(P, N), position(P, W1, N), opposite(W1, W2).
% mise à jour du second traverseur
position(P2, W2, N+P1) :- chose1(P1, N), chose2(P2, N),
                          position(P2, W1, N), opposite(W1, W2).
% mise à jour de ceux qui n'ont pas traversé
position(P2, W, N+P1) :- chose1(P1, N), position(P2, W, N),
                         not chose1(P2, N), not chose2(P2, N).
```

Peu d'étudiants ont abordé cette question. Et parmi eux, je ne me souviens pas en avoir trouvé un qui a mis à jour ceux qui n'ont pas traversé. C'est indispensable pour que le programme tourne correctement, et c'est ce qu'on appelle traditionnellement le *frame problem* en IA: "In the logical context, actions are typically specified by what they change, with the implicit assumption that everything else (the frame) remains unchanged."

Question 7: Si vous avez correctement suivi les spécifications du programme jusqu'à présent, vous pouvez remarqué que votre programme ne s'arrête pas. Justifiez cette affirmation. Proposez des modifications simples de votre programme pour que l'algorithme ASPERIX s'arrête. Etes-vous certain que l'algorithme de Clingo s'arrête également?

La construction d'une branche de l'arbre de recherche peut être décrite de la façon suivante:

```
Config <- Config(0) % configuration initiale (au temps 0)
Temps <- 0
TANT QUE Pas gagné
  Choisir 1 ou 2 traverseurs du côté de la torche
  % le plus lent met T mins pour traverser
  Temps += T
  Config <- Config(T)
```

La branche de l'arbre ASPERIX qui explore une bonne solution (tout le monde a traversé en moins de 17 mins) est finie, mais toutes les autres branches sont infinies. Afin de s'assurer que toutes les branches sont finies, nous pouvons faire la modification suivante:

```
lose(N) :- torch(W, N), N > 17.
% Remplacement de la regle init par:
```

```
init(W, N) :- torch(W, N), not win(N), not lose(N).
```

Ainsi, on ne continuera pas l'exploration d'une branche de l'arbre de recherche ASPERIX lorsque le temps aura dépassé 17, et on assurera la terminaison de toutes les branches. Par contre, les règles de mise à jour des positions créent de nouveaux temps, et rien n'assure que le grounding effectué par Clingo sera fini.

Pour assurer l'arrêt du grounding par Clingo, nous avons pris l'habitude de borner les étapes/temps disponibles, par exemple en énumérant tous les temps possibles et en utilisant ces temps disponibles dans les corps des règles qui devraient générer de nouveaux temps. Par exemple:

```
time(1..17).
% changement dans la mise à jour de la torche
torch(W2, N+P):- chose1(P, N), torch(W1, N), opposite(W1, W2), time(N+P).
```

Cette solution permet à la fois d'assurer l'arrêt d'ASPERIX et du grounding de Clingo. Mais, à ma grande surprise, il n'y a pas eu besoin de faire ça dans le cas de ce programme. Le grounding effectué par clingo avec la modification précédente est fini, grâce à une optimisation que j'ignore.

Question 8: Rajoutez maintenant une règle pour que les seuls modèles stables énumérés par Clingo encodent une solution du problème.

```
:- not win(N).
```

La question la plus facile, avec une solution courte qu'on avait déjà vue dans 2-3 exemples en cours. Mais je ne me rappelle pas l'avoir vu dans une copie...

1.3 Enumération des modèles stables (2 points)

La partie "théorique", que seul un étudiant a abordée. Et il l'a fait parfaitement, donc toutes mes félicitations.

Comme lors du contrôle précédent, nous cherchons ici à réduire le nombre de sous-ensembles du vocabulaire d'un programme à explorer pour énumérer tous les modèles stables de ce programme.

Question 9: Prouvez la propriété suivante:

Propriété A: *Si* Π *est un programme propositionnel et* $E \subseteq F$ *sont deux ensembles d'atomes, alors* $(\Pi_{|F})^* \subseteq (\Pi_{|E})^*$.

Nous notons ici $\Pi_{|E}$ le programme positif obtenu en réduisant le programme Π par l'esnemble d'atomes Π et Π^* l'ensemble d'atomes obtenu par saturation d'un programme positif Π.

Attention, vous justifierez quelle version (faible ou forte) du programme réduit vous utilisez pour arriver à ce résultat, et montrerez que cette affirmation est fausse si on utilise l'autre version.

Cette propriété avait été énoncée et démontrée dans la correction du contrôle continu que je vous avais envoyée. Aussi, je suis déçu que seul un étudiant ait réussi cette question.

Soit Π un programme propositionnel, et $E \subseteq F$ deux ensembles d'atomes. Soit R une règle de Π dont la forme positive R^+ est dans $\Pi_{|F}$, en utilisant la *définition faible* du programme réduit. Ceci veut dire que pour tout corps négatif N de R, $N \not\subseteq F$. A fortiori, et puisque $E \subseteq F$, $N \not\subseteq E$. Donc R^+ est également une règle de $\Pi_{|E}$. Nous avons donc montré que $\Pi_{|F} \subseteq \Pi_{|E}$, et il s'ensuit que $(\Pi_{|F})^* \subseteq (\Pi_{|E})^*$, par monotonie de la logique des propositions. □

Utilisons maintenant la *définition forte* du programme réduit avec le programme Π suivant:

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

On prend $E = \emptyset$ et $F = \{a\}$, on a bien $E \subseteq F$. Le programme $\Pi_{|E}$ contient uniquement le fait `a.` (la règle est supprimée car $a \notin E$) et donc $(\Pi_{|E})^* = \{a\}$. Par contre, le programme $\Pi_{|F}$ contient le fait `a.` et la règle `b:- a.`, et on a $(\Pi_{|F})^* = \{a, b\}$. On voit que $(\Pi_{|F})^* \not\subseteq (\Pi_{|E})^*$. □

Le contrexemple donné ici est beaucoup plus simple que celui de la correction du contrôle continu.

Question 10: Utilisez la propriété A de la question 10 pour prouver les propriétés suivantes:

Propriété B: *Si* $(\Pi_{|E})^* \subsetneq E$*, alors aucun super-ensemble de* E *ne peut être un modèle stable de* Π.

Propriété C: *Si* $E \subsetneq (\Pi_{|E})^*$*, alors aucun sous-ensemble de* E *ne peut être un modèle stable de* Π.

Propriété B: On suppose $(\Pi_{|E})^* \subsetneq E$. Soit F un super-ensemble de E, c'est à dire $E \subseteq F$. De par la propriété A (et avec la définition faible du programme réduit), on a $(\Pi_{|F})^* \subseteq (\Pi_{|E})^*$. En résumé on a:

$$(\Pi_{|F})^* \subseteq (\Pi_{|E})^* \subsetneq E \subseteq F$$

ce qui veut dire $(\Pi_{|F})^* \subsetneq F$ et donc $(\Pi_{|F})^* \neq F$, et F n'est donc pas un modèle stable de Π. □

Propriété C: On suppose $E \subsetneq (\Pi_{|E})^*$. Soit F un sous-ensemble de E, c'est à dire $F \subseteq E$. De par la propriété A (et avec la définition faible du programme réduit), on a $(\Pi_{|E})^* \subseteq (\Pi_{|F})^*$. En résumé on a:

$$F \subseteq E \subsetneq (\Pi_{|E})^* \subseteq (\Pi_{|F})^*$$

ce qui veut dire $F \subsetneq (\Pi_{|F})^*$ et donc $F \neq (\Pi_{|F})^*$, et F n'est donc pas un modèle stable de Π. □