Examen HAI710I – Fondements de l'IA symbolique

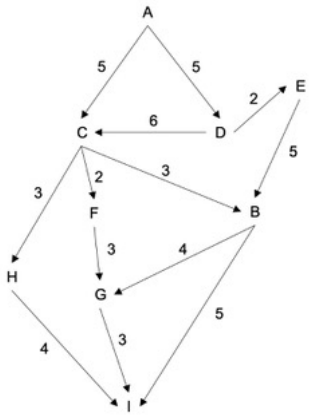
Session 1 - 13 janvier 2021

Durée 2h - Aucun document autorisé

Annexe : des rappels de cours concernant les exercices 1 et 2 à la fin du sujet.
Barème indicatif (sur 21) : ex. 1 (3 points), ex. 2 (5 points), ex. 3 (3 points), ex. 4 (4 points), ex. 5 (2 points), ex. 6 (4 points).

Exercice 1. Recherche Aveugle

Soit le problème de calcul d'un chemin d'un point A à un point I sur le graphe orienté suivant (les entiers indiquent les coûts de chaque arc). On rappelle qu'une fonction de coût associe un entier strictement positif à toute action permettant de passer d'un état à un autre.



1. La recherche en profondeur sur ce problème est-elle complète ? Justifiez votre réponse.
2. Appliquez l'algorithme de recherche de coût min à ce problème. Vous représenterez l'arbre de recherche développé par coût min en précisant sur chaque nœud le coût du chemin de la racine au nœud.
3. Quel est le coût de la solution trouvée ?
4. La recherche par coût min est elle complète quelque soit la fonction de coût considérée ? Justifiez votre réponse.

Exercice 2. Recherche Informée

On se propose maintenant d'utiliser des heuristiques pour mettre en œuvre des stratégies de recherche informée sur le problème précédent. On considère ici l'algorithme `Explorer` non optimisé (c'est-à-dire sans mémorisation des états déjà rencontrés). On rappelle qu'une fonction heuristique est une fonction qui associe un entier positif ou nul à tout état de l'espace de recherche. On envisage les trois fonctions heuristiques suivantes :

Point	A	B	C	D	E	F	G	H	I
h_1	10	5	5	10	10	3	3	4	0
h_2	10	2	8	11	6	2	1	5	0
h_3	10	1	4	9	9	3	3	3	0

1. Rappelez ce que cherche à approximer une fonction heuristique.

2. Appliquez la recherche gloutonne sur ce problème en utilisant l'heuristique h_1 et la version non optimisée de l'algorithme `Explorer`. Vous représenterez l'arbre de recherche développé en précisant sur chaque nœud le coût g du chemin de la racine au nœud et la valeur de l'heuristique associée au nœud.
3. On rappelle qu'une heuristique h est admissible si pour tout état e, $h(e) \leq$ coût minimal d'un chemin de e à l'état but le plus proche. Que garantit la propriété d'admissibilité d'une heuristique ?
4. Les 3 heuristiques précédentes sont-elles admissibles ? Justifiez votre réponse.
5. On rappelle qu'une heuristique h_i domine une heuristique h_j si pour tout état e, $h_i(e) \geq h_j(e)$. Quelles relations de dominance existent entre ces 3 heuristiques ?
6. Finalement quelle heuristique choisiriez-vous pour la recherche A* avec l'algorithme `Explorer` non optimisé ?
7. Appliquez la recherche A* sur ce problème avec cette heuristique et la version non optimisée de l'algorithme `Explorer`. Vous représenterez l'arbre de recherche développé en précisant sur chaque nœud le coût g du chemin de la racine au nœud et la valeur de l'heuristique associée au nœud.

Exercice 3. Modélisation CSP d'un problème d'homomorphisme

Une recherche d'homomorphismes de E_1 dans E_2, où E_1 et E_2 sont deux ensembles d'atomes représentant des formules existentielles conjonctives de la logique du premier ordre, consiste à chercher des substitutions s des variables de E_1 dans les termes de E_2 telle que $s(E_1) \subseteq E_2$.

Un réseau de contraintes est un triplet (X, D, C) où $X = \{x_1, \ldots, x_n\}$ est un ensemble de variables, D une fonction qui associe un domaine D_i à chaque variable x_i, et C un ensemble de contraintes portant sur les variables de X. On appelle CSP (problème de satisfaction de contraintes) la recherche de solutions à un tel réseau.

Soit le problème de recherche d'homomorphismes de Q dans F suivant :

$$Q = \{p(x,y), q(x,w), q(y,z), q(w,w), r(z,y)\}$$
$$F = \{p(a,b), p(a,c), p(c,a), p(c,c), p(f,e),$$
$$q(a,d), q(b,b), q(c,d), q(c,e), q(d,d), q(e,e), q(f,b),$$
$$r(b,c), r(d,c), r(e,a)\}$$

où w, x, y, z sont des variables et a, b, c, d, e, f des constantes.

Définissez un réseau de contraintes (X, D, C), dont les contraintes C sont toutes binaires et en extension, et tel que les solutions à ce réseau correspondent exactement aux solutions au problème de recherche d'homomorphismes de Q dans F.

Exercice 4. Résolution de CSP

Soit le réseau de contraintes (X, D, C) binaires où :

— $X = \{t, u, v, w\}$

— $D(t) = D(u) = D(v) = D(w) = \{1, 2, 3, 4, 5, 6\}$

— $C = \{c_1, c_2, c_3, c_4\}$

c_1

u	w
1	2
1	3
3	1
3	3
6	4

c_2

t	u
2	2
2	6
4	1
4	3
4	4
5	3
5	5

c_3

v	w
2	2
2	6
4	1
4	3
4	4
5	3
5	5

c_4

v	w
2	3
4	3
5	1

1. Calculez la fermeture arc-consistante de ce réseau.

2. Appliquez l'algorithme de Forward Checking pour trouver toutes les solutions sur cette fermeture arc-consistante en utilisant l'heuristique (dynamique) $dom + deg + alpha$ pour l'ordre des variables et l'ordre alphabétique pour les valeurs. Vous préciserez bien à chaque étape comment les domaines de chaque variable évoluent. On rappelle que *dom* consiste à sélectionner en priorité la variable ayant le plus petit domaine (dynamique), que *deg* consiste à sélectionner en priorité la variable impliquée dans le plus grand nombre de contraintes et *alpha* correspond à l'ordre alphabétique ; $dom + deg + alpha$ consiste à utiliser *dom* et en cas d'égalité *deg* pour départager puis, si encore égalité, *alpha*.
3. Donnez les solutions trouvées.

Exercice 5. Chaînage avant

1. Le chaînage avant sur les règles positives en logique des propositions est **complet**. Choisir dans les phrases ci-dessous celle(s) qui tradui(sen)t cette propriété.
   (a) Pour tout symbole A, si $A \in BF^*$ alors $BF, BR \models A$
   (b) Pour tout symbole A, si $BF, BR \models A$ alors $A \in BF^*$
   (c) Pour tout symbole A, si $A \notin BF^*$ alors $BF, BR \not\models A$
   (d) Pour tout symbole A, si $BF, BR \not\models A$ alors $A \notin BF^*$
   (e) Pour tout symbole A, si $A \notin BF^*$ alors $BF, BR \models \neg A$
2. Montrer par un exemple simple que le chaînage avant n'est plus complet si les règles comportent des littéraux négatifs (même si la base de faits ne comporte que des littéraux positifs). Expliquez bien pourquoi le chaînage avant n'est pas complet sur votre exemple.

Exercice 6. Règles avec négation du monde clos

On considère maintenant des règles en logique des propositions avec négation du monde clos. On rappelle qu'une suite d'application de règles (une dérivation) menant d'une base de faits BF à une certaine base de faits BF^i est dite *satisfaisante* si aucune règle de la dérivation n'est bloquée dans BF^i (autrement dit, pour toute règle $H+, H^- \rightarrow C$, on a $H^- \cap BF^i = \emptyset$). Une dérivation de BF à BF^i est dite *complète* si aucune règle applicable sur BF^i n'apporte de nouvel atome.
Soit la base de règles suivante :

$BR = \{$
$R_1 : A, not\ B \rightarrow E$
$R_2 : A, not\ B \rightarrow C$
$R_3 : A, not\ C \rightarrow B$
$R_4 : not\ B, not\ E \rightarrow D$
$R_5 : D \rightarrow F$
$R_6 : C, E \rightarrow F$
$\}$

1. Soit la base de connaissances $K = (BF, BR)$ avec $BF = \{A\}$. La dérivation (R_1, R_3) partant de BF est-elle satisfaisante ? Est-elle complète ?
2. Soit la même base de connaissances $K = (BF, BR)$ avec $BF = \{A\}$. Donner toutes les dérivations satisfaisantes et complètes de cette base de connaissances, ainsi que le résultat de chacune (c'est-à-dire la base de faits saturée obtenue).
3. Construire le graphe de précédence des symboles associé à BR. Qu'en concluez-vous quant-à la stratifiabilité de la base de règles ? Justifier votre réponse.
4. Quelle est la propriété fondamentale assurée par une base de règles stratifiable ?
5. Votre réponse à la question 2 vous permettait-elle de conclure directement à propos de la stratifiabilité de BR ? Pourquoi ?
6. Donner un exemple de règle qui ne peut jamais apparaître dans une dérivation satisfaisante.

7. La notion de base de règles stratifiable est indépendante de toute base de faits. Lorsqu'une base de règles n'est pas stratifiable, il se peut pourtant qu'elle assure la propriété de la question 4 pour certaines bases de faits. Supposons que l'on connaisse la base de faits BF qui nous intéresse : comment prendre en compte cette information pour assouplir la condition de stratifiabilité de BR tout en garantissant la propriété de la question 4 pour (BF, BR) ?

Annexe

Soient les structures et fonctions générales de recherche permettant de parcourir les états d'un espace d'états par priorité croissante. On ne donne ici que la version non optimisée de l'algorithme **??** qui peut ré-explorer plusieurs fois un même état. 3 différentes stratégies de recherche sont obtenues selon les valeurs données à $\mathbf{c_0}$, $\mathbf{c_{sn}}$, $\mathbf{p_0}$ et $\mathbf{p_{sn}}$:

- Recherche par coût min :
  - $c_0 = 0$ et $p_0 = 0$
  - $c_{sn} = n.cout + p.cout(a)$ et $p_{sn} = n.cout + p.cout(a)$
- Recherche gloutonne :
  - $c_0 = 0$ et $p_0 = p.heuristique(p.etatInitial)$
  - $c_{sn} = n.cout + p.cout(a)$ et $p_{sn} = p.heuristique(se)$
- Recherche A* :
  - $c_0 = 0$ et $p_0 = p.heuristique(p.etatInitial)$
  - $c_{sn} = n.cout + p.cout(a)$ et $p_{sn} = n.cout + p.cout(a) + p.heuristique(se)$

```
Interface Etat {
};

Interface Action {
};

Interface Probleme {
    etatInitial : Etat ;
    actions(e : Etat) : Ensemble d'Action ;     // actions possibles pour un état donné
    resultat(e : Etat, a : Action) : Etat ;
    but?(e : Etat) : Booleen ;
    cout(a : Action) : Reel ;
    heuristique(e : Etat) : Reel ;

};

Interface Noeud {
    parent : Noeud ;
    etat : Etat ;
    action : Action ;
    cout : Reel ;
    priorite : Reel;
};

Interface Liste<Noeud> {
    vide?() : Booleen ;
    oterTete() : Noeud ;
    oterNoeud(n : Noeud) : void ;
    insererTete(n : Noeud) : void ;
    insererQueue(n : Noeud) : void ;
    insererCroissant(n: Noeud) : void ;
    rechercher(e : Etat) : Noeud (ou null) ;
```

```
};

Interface Ensemble<Etat> {
    contient?(e : Etat) : Booleen;
    ajouter(e : Etat) : void ;
};
```

Fonction Explorer(p : Probleme) : Noeud (ou null)

racine ← new Noeud(null, p.etatInitial, null, $\mathbf{c_0}$, $\mathbf{p_0}$) ;
frontiere ← inserer(racine, []) ;
tant que non *frontiere.vide ?()* **faire**
    n ← frontiere.oterTete() ;
    si *p.but ?(n.etat)* **alors retourner** *n*;
    pour toute *action a* **dans** *p.actions(n.etat)* **faire**
        se ← resultat(n.etat, a) ;
        sn ← new Noeud(n, se, a, $\mathbf{c_{sn}}$, $\mathbf{p_{sn}}$) ;
        frontiere.insererCroissant(sn) ;
retourner *null* ;