Examen HAI710I – Fondements de l'IA symbolique

Session 2 - 29 mars 2022

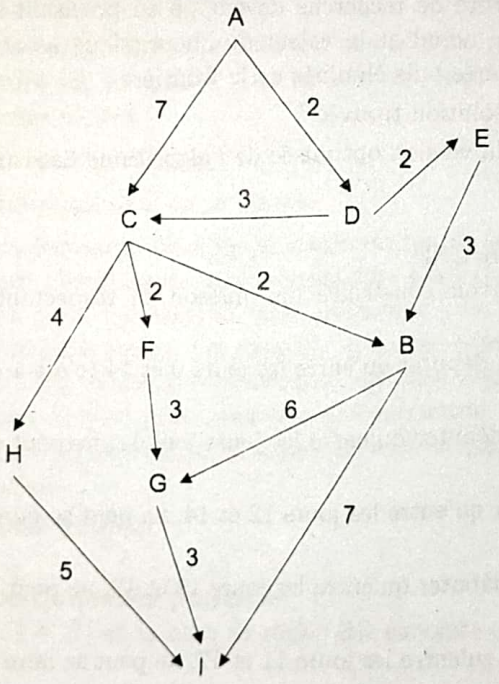
Durée 2h - Aucun document autorisé

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 (4 points), ex. 3 (3 points), ex. 4 (5 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. Appliquez l'algorithme de recherche en largeur à ce problème. Vous représenterez l'arbre de recherche développé par cette recherche en largeur en faisant figurer sur chaque sommet généré le coût du chemin de la racine au nœud.
2. Combien de nœuds ont été générés par cette recherche ?
3. Combien ont été explorés ?
4. Quel est le coût de la solution trouvée ? Est-elle optimale ?

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` optimisé (c'est-à-dire avec mémorisation des états déjà rencontrés).

1. On choisit de prendre comme heuristique la longueur du chemin minimal pour atteindre le but. Faites un tableau donnant pour chaque point son heuristique et la valeur du coût optimal vers l'état but.

Point	A	B	C	D	E	F	G	H	I
h									
g^*									

2. Cette heuristique est elle admissible ? Justifier la réponse.
3. Appliquez la recherche gloutonne sur ce problème en utilisant la version 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. Vous rayerez les éventuels sommets générés puis éliminés de la frontière.
4. La solution trouvée est elle optimale ?
5. La recherche gloutonne avec l'algorithme `Explorer` optimisé est elle toujours complète quelque soit le problème considéré ? Justifier la réponse.
6. Appliquez la recherche A* sur ce problème avec la version 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. Vous rayerez les éventuels sommets générés puis éliminés de la frontière.
7. Quel est le coût de la solution trouvée ?
8. La recherche A* avec la version optimisée de l'algorithme `Explorer` est elle toujours optimale ? Justifier votre réponse

Exercice 3. Modélisation CSP

Un architecte pense pouvoir construire une maison en respectant les contraintes suivantes des différents prestataires :

1. La maçonnerie ne peut débuter qu'entre les jours 0 et 14 (c'est-à-dire y compris 0 et 14), et dure 7 jours
2. La charpente ne peut débuter qu'entre les jours 7 et 11, ne peut se faire qu'après la maçonnerie et dure 3 jours
3. Le toit ne peut débuter qu'entre les jours 12 et 14, ne peut se faire qu'après la charpente et dure 1 jour
4. La plomberie ne peut débuter qu'entre les jours 12 et 19, ne peut se faire qu'après la maçonnerie et dure 8 jours
5. Le sol ne peut débuter qu'entre les jours 11 et 17, ne peut se faire qu'après la maçonnerie et dure 3 jours
6. Le fenêtrage ne peut débuter qu'entre les jours 11 et 19, ne peut se faire qu'après le toit et dure 1 jour
7. La façade ne peut débuter qu'entre les jours 17 et 22, ne peut se faire qu'après le toit et la plomberie et dure 2 jours
8. Le jardin intérieur ne peut débuter qu'entre les jours 17 et 23, ne peut se faire qu'après le toit et la plomberie et dure 1 jour
9. La peinture ne peut débuter qu'entre les jours 8 et 18, ne peut se faire qu'après le sol et dure 2 jours
10. L'aménagement intérieur ne peut débuter qu'entre les jours 12 et 23, ne peut se faire qu'après le fenêtrage, la façade, le jardin et la peinture et dure 1 jour

Modéliser ce problème comme un problème de satisfaction de contraintes.

Exercice 4. Résolution de CSP

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

— $X = \{v, w, x, y, z\}$

— $D(v) = D(w) = D(x) = D(y) = D(z) = \{1, 2, 3\}$

— $C = \{c_1, c_2, c_3, c_4, c_5, c_6\}$

c_1

x	y
1	2
~~1~~	~~3~~
~~2~~	~~1~~
3	2

c_3

w	z
2	1
~~3~~	~~2~~
~~3~~	~~3~~

c_5

x	v
1	2
~~2~~	~~3~~
3	1
3	2

c_2

y	z
~~1~~	~~2~~
~~1~~	~~3~~
2	1
~~2~~	~~2~~
~~3~~	~~2~~

c_4

v	w
1	2
2	2
~~2~~	~~3~~
~~3~~	~~1~~

c_6

w	y
~~1~~	~~3~~
~~2~~	~~1~~
2	2
~~3~~	~~1~~

1. Dessinez le graphe de ce réseau de contraintes
2. Appliquez l'algorithme de backtrack à la recherche d'une solution en prenant l'ordre alphabétique pour les variables et l'ordre sur les entiers pour les valeurs. Vous dessinerez l'arbre de recherche en indiquant les contraintes violées.
3. Quelle est cette première solution trouvée ?
4. Calculez la fermeture arc-consistante de ce réseau.
5. 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 sur les entiers 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*.
6. Donnez toutes les solutions trouvées.

Exercice 5. Règles propositionnelles positives

Soit la base de faits BF $= \{A, B\}$ et la base de règles BR suivante :

$R1 : B \wedge C \rightarrow D$
$R2 : D \wedge A \rightarrow G$
$R3 : B \wedge H \rightarrow I$
$R4 : G \wedge F \rightarrow I$
$R5 : A \wedge I \rightarrow H$
$R6 : A \wedge G \rightarrow I$
$R7 : F \rightarrow E$
$R8 : A \rightarrow C$

1. Le symbole I appartient-il à la base de faits saturée ? Si oui, donnez une dérivation (une suite d'applications de règles) produisant I. Si non, donnez la base de faits saturée.
2. Votre réponse à la question 1 vous permet-elle de dire si le symbole I est prouvable en chaînage arrière ? Pourquoi ?

3. Essayez de prouver I en chaînage arrière. Dessinez l'arbre de recherche construit par un algorithme qui considère les règles par numéro croissant et les symboles par ordre d'apparition dans l'hypothèse de la règle considérée. Vous indiquerez sur chaque feuille traitée : échec, (appartient à) BF, déjà prouvé, boucle. I est-il finalement prouvé ?

Exercice 6. Règles propositionnelles avec négation du monde clos

On considère maintenant des règles avec négation du monde clos. On rappelle qu'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 $H^- \cap BF^i = \emptyset$).

Soit la base de règles BR suivante :

$R_1 : O \wedge P \rightarrow T$
$R_2 : T \wedge not\ S \rightarrow Q$
$R_3 : S \rightarrow R$
$R_4 : T \wedge not\ P \rightarrow S$
$R_5 : O \rightarrow P$
$R_6 : R \wedge P \rightarrow S$

1. Soit la base de connaissances (BF, BR) avec $BF = \{T\}$. Quelles sont toutes les bases de faits saturées possibles ? Parmi celles-ci, dire lesquelles peuvent être obtenues par des dérivations satisfaisantes ?
2. Quelle est la propriété fondamentale assurée par une base de règles stratifiable ?
3. Calculer le graphe de précédence des symboles associé à BR. On rappelle que seuls les symboles apparaissant en conclusion de règle sont pertinents. En déduire une stratification possible.
4. Partant de BF $= \{O\}$, quelle base de faits saturée est obtenue par une dérivation satisfaisante ? Comment la calculez-vous ?

Annexe

Soient les structures et fonctions générales de recherche permettant de parcourir les états d'un espace d'états par priorité croissante. On donne ici la version optimisée de l'algorithme `Explorer` qui ne peut pas 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, c0, p0) ;
frontiere ← inserer(racine, []) ;
explore ← ∅ ;
tant que non frontiere.vide ?() faire
    n ← frontiere.oterTete() ;
    explore.ajouter(n.etat) ;
    si p.but ?(n.etat) alors retourner n;
    pour toute action a dans p.actions(n.etat) faire
        se ← resultat(n.etat, a) ;
        si non explore.contient ?(se) alors
            sn ← new Noeud(n, se, a, csn, psn) ;
            sosie ← frontiere.rechercher(se) ;
            si sosie = null alors
                frontiere.insererCroissant(sn) ;
            sinon si sn.priorite < sosie.priorite alors
                frontiere.oterNoeud(sosie) ;
                frontiere.insererCroissant(sn) ;
retourner null ;
```