TP numéro 5

Intelligence artificielle, ENSIIE

Semestre 4, 2022–23

Votre code devra être déposé d'ici au 26 avril 2023 sur `exam.ensiie.fr` dans le dépôt `ia-fisa-tp5`.

L'objectif de ce TP est de proposer une intelligence artificielle pour résoudre le jeu MasterMind. Le but de ce jeu est de deviner une combinaison de 4 couleurs (bleu ●, rouge ●, vert ●, jaune ●, blanc ○ ou noir ●) en moins de dix essais. Pour chaque essai, l'adversaire indique le nombre de couleurs à la bonne place, et le nombre de couleurs présentes mais pas à la bonne place. (Chaque couleur de la solution ne pouvant compter qu'une fois.)

Par exemple, si la combinaison choisie par l'adversaire est ● ● ● ●,
si on joue ● ○ ○ ●, l'adversaire renverra 0 et 0,
si on joue ● ● ● ●, l'adversaire renverra 2 et 1,
si on joue ● ● ● ●, l'adversaire renverra 1 et 1.

Dans tout ce travail, nous considérerons des combinaisons de taille quelconque et pas nécessairement 4, de façon à rester générique. On définira un prédicat `color` qui indique qu'une constante est une couleur. Les combinaisons seront représentées par des listes de constantes qui vérifient toutes le prédicat `color`.

1. Définir le prédicat d'arité 1 `color` qui sera vrai pour les constantes `blue`, `red`, `green`, `yellow`, `white` et `black`.
2. Définir un prédicat `combination(L)` qui sera vrai quand tous les éléments de la liste `L` vérifient `color`.

Exercice 1 : Calcul des réponses

Avant d'écrire l'intelligence artificielle permettant d'essayer de trouver la solution, nous allons d'abord écrire un programme permettant de calculer les nombres de couleurs bien et mal placées, en connaissant la solution. Il s'agit donc de définir un prédicat `answer(P, S, G, B)` tel que si `P` est une combinaison constituant une proposition, `S` est la solution qui est une combinaison supposée de même longueur, alors `G` est le nombre de couleurs bien placées, et `B` celles mal placées. Pour définir ce prédicat, nous allons procéder par plusieurs étapes.

3. Définir un prédicat `well_placed(P, S, N)` qui sera vrai quand `N` est le nombre de positions où les listes `P` et `S` contiennent les mêmes valeurs.
4. Définir un prédicat `remove_well_placed(P, S, NP, NS)` qui sera vrai quand les listes `NP` et `NS` seront égales aux listes `P` et `S` sans les positions où ces dernières contiennent les mêmes valeurs.

5. Définir un prédicat `mem_remove(X, L, Q)` tel que `Q` soit la liste `L` dans laquelle on a retirée la première occurrence de `X`. Le prédicat ne sera pas défini si `X` n'est pas dans `L`. On aura par exemple `mem_remove(a, [b,a,c,a,d], [b,c,a,d])`.
6. Définir un prédicat `wrong_placed(P, S, N)` qui sera vrai quand `N` est le nombre d'éléments de `P` qui apparaissent dans `S`, chaque élément de `S` ne pouvant compter qu'une seule fois.

   On utilisera donc `mem_remove` pour tester l'appartenance dans `S`, pour enlever de `S` l'élément trouvé le cas échéant.
7. Définir le prédicat `answer/4` en utilisant `well_placed/3` et `remove_well_placed/4` et `wrong_placed/3`.

Exercice 2 : Suggestion de propositions

Le but de cette partie est d'écrire un prédicat qui suggère des propositions en fonction des propositions précédentes et de leurs réponses associées. On ne suppose donc plus qu'on connaît la solution. Ici aussi on va y aller par étapes. L'idée générale est qu'on va considérer n'importe quelle combinaison valide (de façon non-déterministe), puis on va vérifier que cette combinaison est en accord avec les contraintes données par les réponses des propositions précédentes. Les tentatives précédentes avec leurs réponses associées seront stockées dans une liste dont la taille sera un multiple de 3, par exemple `[[yellow, white, white, black], 0, 0, [blue, blue, green, green], 2, 1]`.

8. Définir un prédicat `check_proposition(P, L)` qui sera vrai si la proposition `P` est compatible avec les contraintes données par la liste de tentatives précédentes avec leurs réponses associées.

   Par exemple, avec la liste ci-dessus, la combinaison `[blue, yellow, green, blue]` n'est pas valide car il ne peut pas y avoir du jaune, et la combinaison `[blue, blue, blue, green]` n'est pas valide car si c'était la solution, la deuxième tentative aurait eu comme réponse 3 et 0.
9. Définir un prédicat `suggest(P, L)` qui donne la première liste `P` qui est une combinaison valide et qui vérifie les tentatives précédentes `L`. On utilisera une coupure pour garantir de n'avoir qu'une solution.
10. Étant donnée une liste de propositions avec leurs réponses associées, quelle requête faire pour obtenir une nouvelle suggestion ?

Exercice 3 : Simulation de partie

On va maintenant écrire un prédicat qui va nous permettre de voir comment notre intelligence artificielle fonctionne par rapport à une solution donnée.

11. Définir un prédicat `play(L, S, R)` tel que pour une solution `S`, à partir des tentatives précédentes avec leurs réponses associées dans `L`, on ait dans `R` les tentatives qui amène à la solution `S`.

Pour cela, à partir de `L` on suggère une nouvelle proposition `P`. Si cette proposition est la solution, alors `R` vaudra `L` suivie de `[P, 4, 0]`. Sinon, on calcule les réponses `G` et `B` de `P` par rapport à `S`, et on continue de jouer avec `L` suivie de `P, G, B`.

12. Définir le prédicat `score(S, N)` qui calcule le nombre de tentatives `N` que met l'algorithme précédent pour trouver la solution `S`.
13. Trouver une combinaison pour laquelle ce prédicat est maximum.

    L'algorithme est-il suffisant pour gagner à tous les coups ? (Rappelons qu'on dispose de dix tentatives.)
14. Est-il possible d'améliorer l'algorithme pour trouver les solutions en moins de tentatives ? Si oui, proposer une méthode.