TP9. Puissance 4 en OCaml

Dans ce TP, on développe un algorithme de min-max capable de jouer une partie de puissance 4. Plus précisément, l'algorithme doit simuler un joueur en décidant les coups à jouer pour répondre aux coups du joueur adverse, qui eux sont ceux renseignés manuellement à travers d'une interface graphique. Autrement dit on permet la réalisation de partie humain contre machine.

Représentation du jeu en machine

Afin de représenter le jeu du puissance 4 en machine, on se munit des types suivants :

- Le type couleur, admettant les deux valeurs Rouge et Jaune, permet de représenter la couleur des pions, mais aussi les joueurs car chacun a une couleur

```
type couleur = Rouge | Jaune
```

- - Le type `game` permet de représenter un état du plateau de jeu par une matrice d'éléments de type `couleur option`. En effet une case vide (sans jeton) est représentée par la valeur `None`, une case avec un jeton rouge (resp. jaune) par la valeur `Some(Rouge)` (resp. `Some(Jaune)`).

```
type game = couleur option array array
```

  On suppose fixées dans toute la suite deux grandeurs `nb_colonnes` et `nb_lignes` valant respectivement 7 et 6. Ainsi dans la suite on supposera que les éléments de type `game` sont de dimensions `nb_ligne` $\times$ `nb_colonne`.

- Le type `etat` permet de représenter un état du jeu par l'état du plateau de jeu et le joueur dont c'est le tour de jouer.

```
type etat = { game : game ; joueur : couleur }
```

Dans le fichier `tp9.ml` on fournit plusieurs fonctions utilitaires, dont une fonction d'affichage `draw_game : game -> unit` à utiliser pour les tests, et une fonction `main : unit -> unit` qui permettra de lancer une partie contre l'ordinateur.

I. Graphe d'états

On se munit maintenant d'une représentation implicite du graphe du jeu du puissance 4.

1. Écrire une fonction `place_jeton : etat -> int -> etat option` prenant en arguments un état du jeu et un indice `k` de colonne et retournant l'état de jeu après que le joueur (dont c'était le tour) a placé son jeton dans la colonne `k`. Dans l'éventualité où un tel coup n'est pas possible, si la colonne est déjà pleine, on retourne `None`.
On veillera à ne pas modifier la grille de jeu prise en entrée, mais une copie de cette dernière, à l'aide de la fonction `copy_game` fournie.

2. En déduire une fonction `next : etat -> etat list` prenant en argument un état du jeu et retournant la liste de tous les états accessibles en un coup.
Cette fonction `next` nous fournit donc une représentation en machine du graphe d'états du jeu. Le graphe n'est cependant jamais explicitement construit dans la mémoire.

II. MinMax et élagage α-β

On commence par se munir d'une fonction heuristique d'évaluation de la qualité d'un état du jeu pour un joueur j. L'heuristique de score d'un plateau pour un joueur j est définie de la manière suivante :

- Si le joueur j est gagnant, le score est $+\infty$;
- Si son adversaire est gagnant, le score est $-\infty$
- Sinon, on somme les valeurs des cases occupées sur le plateau, en comptant positivement celles de j et négativement celles de son adversaire. La valeur de chaque case représente le nombre d'alignements de 4 cases qui la couvre. Ainsi les cases près du centre ont une plus grande valeur. L'idée est que la situation est d'autant plus favorable pour un joueur qu'il a de jetons en position de former des alignement, mais c'est approximatif car on compte les possibilités comme si le pions était seul sur le plateau, alors que certains des alignements potentiels comptabilisés sont peut-être déjà impossibles. On fournit, dans `tp9.ml`, une matrice `val_cases` donnant les valeurs des cases.

3. Définir une fonction `heuristique : etat -> couleur -> int` prenant en argument un état du jeu, un joueur et retournant le score heuristique de l'état du jeu pour ce joueur. La fonction testant si un joueur a gagné est fournie.

4. Définir une fonction `minmax : int -> etat -> couleur -> int` prenant en arguments une profondeur `p` de recherche maximale, un état du jeu `e`, un joueur `c` et retournant le score de l'état du jeu `e` pour le joueur `c` à horizon `p` coups.

5. En déduire une fonction `joue : etat -> int -> etat` prenant en argument un état du jeu, une profondeur de recherche maximale et retournant l'état successeur de l'état courant ayant le plus grand score obtenu par l'algorithme de minmax.

6. La fonction `main` fournie dans `tp9.ml` lance une partie de puissance 4 contre l'algorithme codé dans la fonction `joue`. Tester cet algorithme en jouant contre lui au puissance 4.

7. Remplacer l'algorithme `minmax` la question 4 par un algorithme de minmax avec élagage α-β.