Fiche TP02-A

marc-michel.corsini@u-bordeaux.fr

10 mars 2015

Le but de ce TP est de construire plusieurs méthodes permettant de parcourir un arbre d'évaluations. Pour cela vous allez travailler dans le fichier python **arbres.py**. Ce fichier contient les différentes méthodes obligatoires à développer (en plusieurs étapes) pour les deux classes suivantes :

1. classe **Parcours**
2. classe **IA**

1 Parcours

Cette classe possède une méthode principale `minmax` qui va, grâce à ses paramètres spécifier l'algorithme (parmi 6 dont 3 optionnels) et la profondeur d'exploration, permettre de récupérer le meilleur coup possible en fonction de l'évaluation attribuée. Deux méthodes secondaires `positionGagnante` qui va déterminer s'il existe un coup permettant d'assurer la victoire et `positionPerdante` qui va déterminer si la configuration courante conduit irrémédiablement à une perte de la partie.
Dans cette première étape nous allons développer uniquement la méthode `_minmax` qui est la version de code 0 de la méthode `minmax` ; ainsi que les deux méthodes secondaires.

1.1 Algorithme minmax

On suppose que l'on reçoit en entrée un nœud de l'arbre à explorer et une profondeur maximale à ne pas dépasser. On suppose de plus l'existence d'une fonction **h** capable d'évaluer un nœud. Enfin, comme vu en cours on considère que les niveaux dans l'arbre sont alternativement de niveau **MAX** et **MIN**. L'algorithme en simili-python sera donc :

```
def minmax(n,pf,pfmax):
    """ parcours récursif en profondeur d'abord """
    if feuille(n) or pf == pfmax :
        return h(n)
    elif n de niveau MAX :
        soit f_1, ... f_k les fils de n
        rep = -∞
        for i in range(1,k+1):
            rep = max(rep,minmax(f_i,pf + 1,pfmax))
        return rep
    else: # n de niveau MIN
        soit f_1, ... f_k les fils de n
        rep = +∞
        for i in range(1,k+1):
            rep = min(rep,minmax(f_i,pf + 1,pfmax))
        return rep
```

1.2 Mise en oeuvre

L'algorithme décrit section 1.1 ne répond qu'imparfaitement à notre problème. En effet le but est non pas de récupérer la meilleure évaluation mais bien le meilleur coup à jouer. Nous allons regarder les méthodes à notre disposition :

1. Un nœud de l'arbre sera un état (configuration) du jeu.
2. Le nombre de fils d'un nœud sera le nombre de coups que l'on peut effectuer, la configuration et le joueur ayant le trait étant connus.
3. Un fils d'un nœud sera la configuration résultante de l'application du coup à l'état courant. Il est important de ne pas modifier la valeur du nœud père.
4. Il est inutile de disposer de l'information pf et pfmax, il suffit d'initialiser la variable pf à la valeur pfmax, et de décrémenter à chaque changement de niveau.
5. La fonction **h** est la fonction d'évaluation du TP01.
6. Une feuille de l'arbre correspond à une fin de partie
7. Le niveau dans l'arbre (**MAX** ou **MIN**) peut-être déterminé en comparant le joueur qui a le trait avec le joueur qui a été à l'origine du calcul.
8. La valeur ∞ est représentée par la variable `BIGVALUE` du fichier `arbres.py`

La signature de `_minmax` est : Etat $\times$ Joueur $\times \mathbb{N}^+ \rightarrow$ Coup $\times \mathbb{R}$
C'est-à-dire qu'en entrée la fonction reçoit l'état du jeu (variable `self`, le joueur par rapport à qui l'évaluation est faite, une profondeur strictement positive, en retour la fonction renvoie le meilleur coup possible et l'estimation de la valeur associée à ce coup. Lorsqu'il n'y a pas de coup disponible, le coup renvoyé sera **None**.

1.3 positionGagnante,positionPerdante

Ces deux méthodes sont mutuellement récursives.

1. Une position (une configuration) sera considérée comme **Gagnante** si et seulement si il existe un coup qui conduise à une position **Perdante** pour son adversaire.
2. Une position (une configuration) sera considérée comme **Perdante** si et seulement si quelque soit le coup choisi il conduit à une position **Gagnante** pour son adversaire

Les signatures sont :

1. `positionPerdante` : Etat $\times$ Joueur $\rightarrow \mathbb{B}$
2. `positionGagnante` : Etat $\times$ Joueur $\rightarrow$ (Coup $\cup$ {`None`}) $\times \mathbb{B}$
   Axiomes :
   si le coup est `None` alors soit c'est une fin de partie, soit le booléen est `False`
   si le coup n'est pas `None` alors le booléen est `True`

2 IA

Il s'agit d'une classe dérivée de **Player** qui va permettre d'exploiter un parcours spécifique afin de trouver le meilleur coup à jouer pour une situation de jeu particulière. La seule méthode qu'il vous est demandée de développer et la méthode **choixCoup** dont l'algorithme est :

```
def choixCoup(self,unJeu,joueur):
    creation d'une variable instance de Parcours
    recuperation du coup et de son evaluation par minmax
    affichage du jeu (facultatif)
    affichage du coup et de sa valeur
    return coup
```

Remarque ne marchera pas tant qu'au moins une des méthodes de **Parcours** parmi `_minmax, _negamax, _alphabeta` ne soit opérationnelle.

3 Matériel

Pour réaliser ce TP, 3 fichiers sont fournis :

1. `tp02_abstract` description abstraite des méthodes dans les classes `Base` et `IAPlayer` ; ce fichier **ne doit pas** être altéré.
2. `arbres` le fichier dans lequel vous allez travailler, les paramètres sont explicités, pour toutes les méthodes à développer, de même des commentaires ont été ajoutés pour vous permettre d'accéder aux différentes informations nécessaires pour la réalisation.
3. `arbres_doo` est un exemple d'utilisation de arbres pour faire jouer une IA au jeu de **Doo**

A venir un fichier de validation02