– Algorithmique & Programmation – D.E.S.S. Imoi –

Projet : jeu de stratégie à 2 joueurs

Rappel : vous pouvez toujours trouver les compléments sur les règles de ce jeu à l'adresse `http://www.loria.fr/~scheuer/fr/Enseignement/AP/awele.html`.

Partie programmation

La partie algorithmique étant maintenant terminée, il me semble judicieux de préciser quelques points qui pourraient poser problème pour la programmation.

La hiérarchie des classes

Tout d'abord, la meilleure hiérarchie des classes me semble être la suivante :

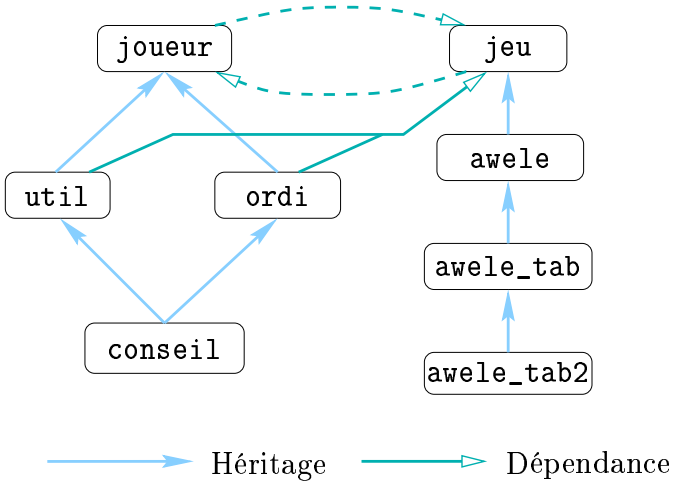

FIG. 1 – *Hiérarchie des classes proposée*

Le fonctionnement sous-tendu par cette hiérarchie repose sur les points suivants :

- on utilise l'héritage pour les jeux de manière à définir les parties communes aux classes héritières : on peut ainsi définir plusieurs jeux de stratégie, de type `jeu`, sans avoir à recopier les attributs ou les fonctions déjà donnés dans `jeu`, ou bien définir plusieurs implantations du jeu de l'awélé, de type `awele`, en minimisant de même le recopiage ;
- l'héritage est aussi utilisé au niveau des joueurs afin de permettre de considérer de la même manière (comme de type `joueur`) un joueur-ordinateur (de type `ordi`), un joueur-utilisateur simple (de type `util`) ou un joueur-utilisateur conseillé par

l'ordinateur (de type `conseil`) ; ce dernier est d'ailleurs défini (grâce à l'héritage) très simplement à partir de la définition des types `ordi` et `util` ;

- le type `joueur` n'a pas besoin de connaître la définition du type `jeu`, mais seulement son existence ; de même, le type `jeu` se satisfait de l'existence du type `joueur`, et n'a pas besoin de connaître sa définition ;
- par contre, les classes `ordi` et `util` ont besoin de connaître la définition du type `jeu`, afin de pouvoir manipuler ce jeu pour choisir un coup à jouer.

L'algorithme Alpha-Bêta

D'autre part, l'algorithme, tel qu'il vous a été donné dans l'article joint à l'énoncé initial, était non seulement faux (une petite coquille s'y était glissée) mais surtout peu clair. L'algorithme suivant, équivalent à celui donné, me semble pourtant plus clair :

si la profondeur maximale de recherche est atteinte
alors **retourner** l'évaluation statique du jeu
sinon si on est à un niveau minimisant
alors **pour** chaque évolution possible du jeu **et**
tant que alpha est strictement inférieur à bêta
bêta reçoit le minimum entre sa valeur et la valeur retournée
par l'alpha-bêta pour cette évolution du jeu
retourner la valeur de bêta
sinon (on est à un niveau maximisant)
pour chaque évolution possible du jeu **et**
tant que alpha est strictement inférieur à bêta
alpha reçoit le maximum entre sa valeur et la valeur retournée
par l'alpha-bêta pour cette évolution du jeu
retourner la valeur de alpha

Cela étant, l'illustration du fonctionnement de l'alpha-bêta donné par la figure 4.18 de l'article correspond bien au principe de l'algorithme, mais pas à sa formalisation. En effet, les valeurs successives de alpha et de bêta ne sont pas indiquées, pas plus que les valeurs retournées par la fonction à chaque niveau d'évaluation. La figure 2 (page suivante) me semble combler cet oubli : l'ordre de succession des évènements étant indiqué entre parenthèses, on peut voir selon les endroits l'intervalle de recherche de la meilleure solution (sous la forme [alpha,bêta]) ou la valeur retournée par l'appel de l'alpha-bêta à un niveau donné.

Ainsi, les valeurs initiales de alpha et bêta sont propagées lors des premiers appels (étape 1 à 3), avant la première évaluation statique (étape 4). Cette évaluation change la valeur de alpha (étape 5), qui ne change plus pendant les autres évaluations statiques (étapes 6 et 7). L'évaluation de ce niveau rend donc comme valeur 8 (étape 8), ce qui modifie au niveau supérieur la valeur de bêta (étape 9)... Et ainsi de suite, jusqu'au retour de la meilleure valeur (étape 50).

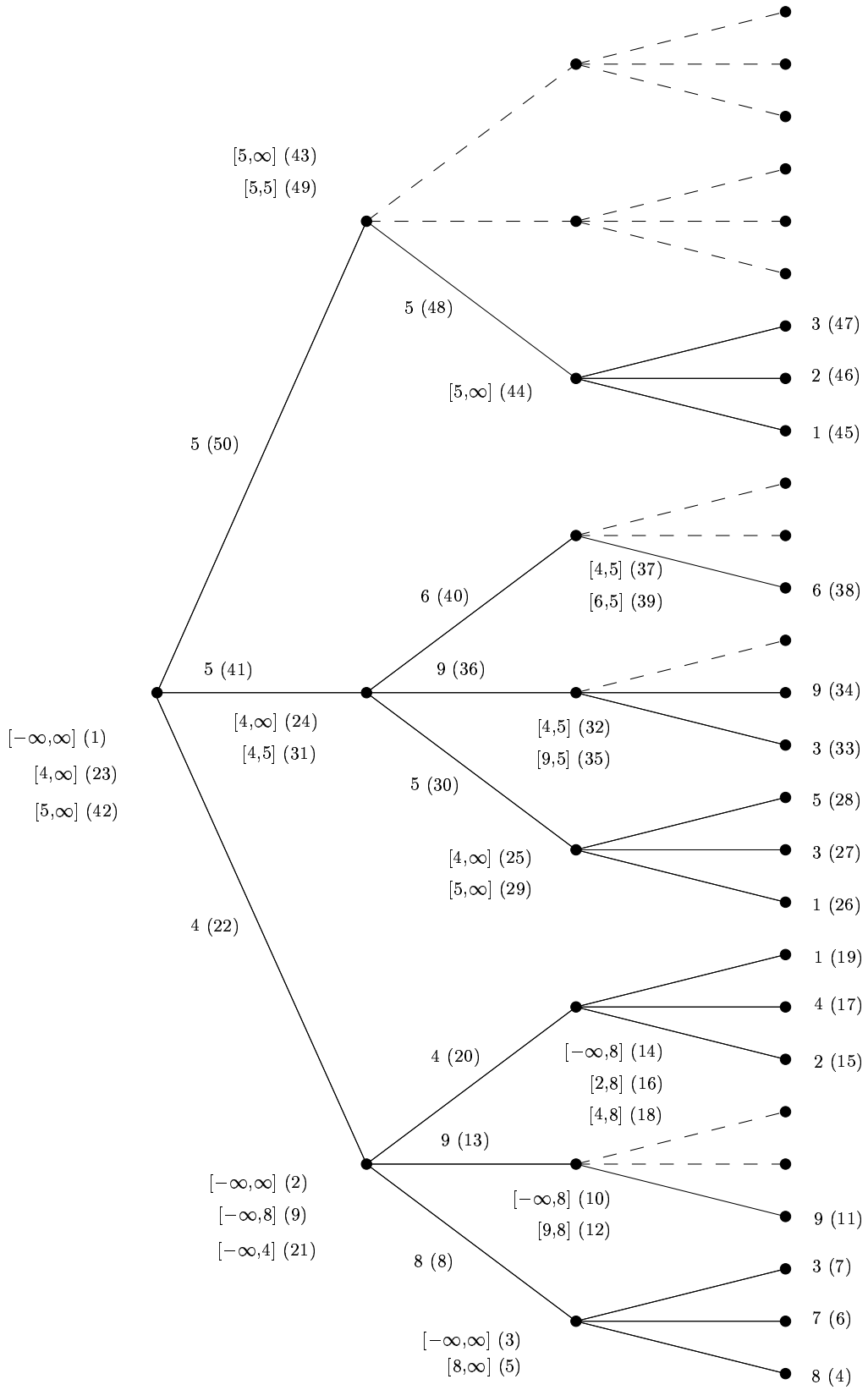

FIG. 2 – *Exemple de recherche par l'alpha-bêta*

La copie d'un jeu

Enfin, dans l'algorithme de l'alpha-bêta, on souhaite évaluer chaque évolution possible du jeu à partir d'une configuration donnée. Or, pour pouvoir évaluer deux évolutions distinctes, il faut pouvoir revenir à la configuration initiale. Pour cela, on a deux solutions : soit copier la configuration initiale (autant de fois qu'il y a d'évolutions possibles), soit définir une méthode pour revenir un coup en arrière (afin de retrouver la configuration initiale). La seconde solution me semblant plus compliquée à mettre en œuvre (les classes définies en standard, comme la classe `list`, ne fonctionnant pas correctement), j'ai préféré me limiter à la première.

Celle-ci n'est pourtant pas évidente. En effet, la classe `jeu` ne peut contenir de constructeur de copie abstrait. Ainsi, au niveau de l'algorithme de l'alpha-bêta, on ne peut pas demander de copier un jeu, en laissant au compilateur le soin d'utiliser le constructeur de copie de la *vraie* classe de ce jeu (dans notre cas, celui de la classe `awele_tab2`, *cf.* fig. 1). Par contre, il est possible de créer une fonction abstraite de *clonage* d'un jeu. Cette fonction renverra toujours une référence sur un `jeu` (pour rester cohérent avec la définition dans la classe `jeu`), mais elle construira en réalité un objet du même type que celui sur lequel elle est appelée (dans notre cas, un `awele_tab2`, qui peut bien être considéré comme un `jeu`, dont il hérite). La forme générale de cette fonction de clonage est la suivante :

```
jeu_strat2& awele_tab2::clone() const
{
  // On crée le clone, avec les références des joueurs
  awele_tab2& res = *( new awele_tab2( le_joueur(1),
                                       le_joueur(2) ) );
  // on copie les attributs
  ...
  // et on retourne le clone
  return res;
} // end of jeu_strat2& clone() const
```

Bien sûr, il ne faut pas oublier de définir un opérateur `delete`, de prototype `static void operator delete(void *ptr)`, afin de libérer la mémoire réservée pour le clone. Cet opérateur devra utiliser entre autres l'opérateur `delete` par défaut (`::delete`) et indiquer le vrai type du pointeur passé en argument. Ainsi, il sera défini de la manière suivante :

```
static void operator awele_tab2::delete(void *ptr)
{
  // on libère les attributs
  ...
  // on libère la structure
  ::delete (awele_tab2 *)(ptr);
} // end of operator delete(void *)
```

où `...` correspond aux instructions nécessaires à la libération de mémoire qui n'est pas automatiquement gérée par la libération standard (effectuée par l'opérateur `delete` par défaut, à la dernière ligne).