TP4 Intelligence Artificielle

L'objectif de ce TP est de créer une intelligence artificielle capable de joueur au jeu de dames (plusieurs variantes de ce jeu existent, mais nous allons travailler sur le jeu de dames anglaises). Je vous laisse consulter les régles sur Wikipedia :https://en.wikipedia.org/wiki/English_draughts.

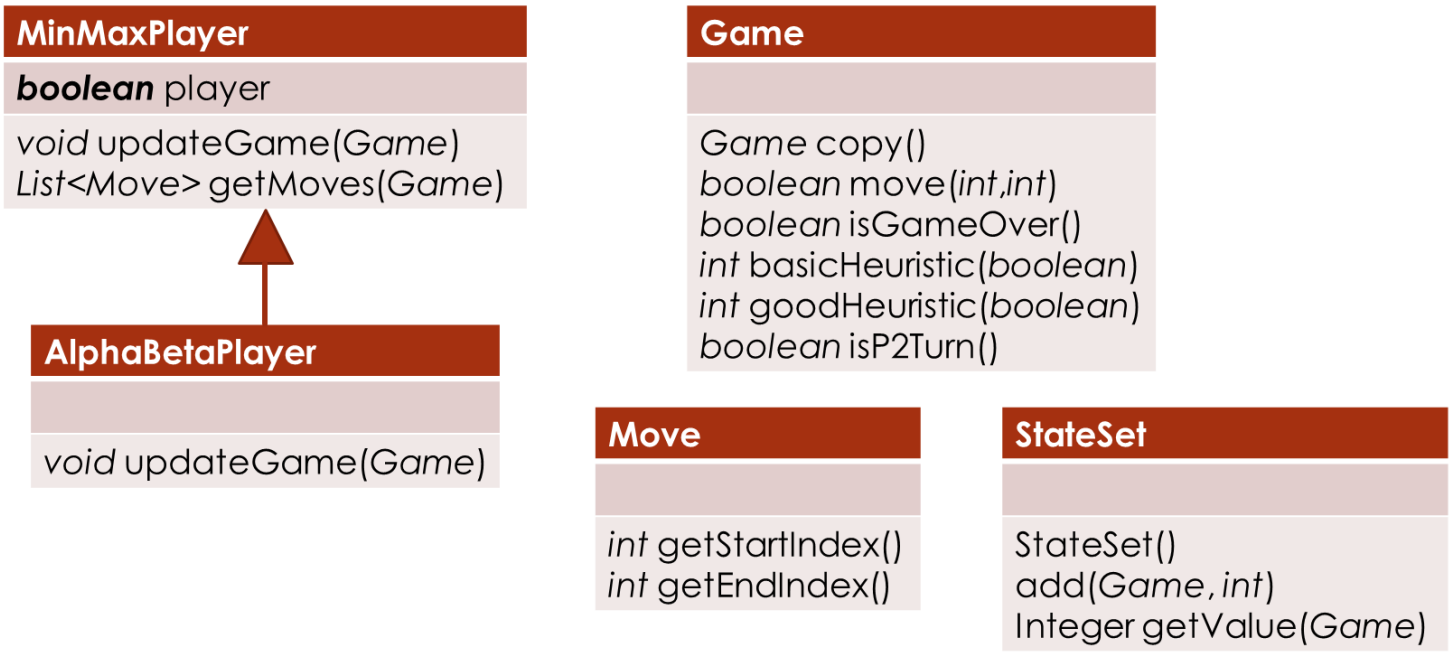
I) Moteur du jeu et affichage

J'ai utilisé le code de Devon McGrath (et je l'en remercie) pour le jeu et l'affichage graphique du jeu. J'ai très peu modifié le code qui était relativement bien écrit.

Je me suis également inspiré du code en C++ de John Wiseman (et je l'en remercie) pour écrire une des heuristiques (j'en parles plus tard dans le TP).

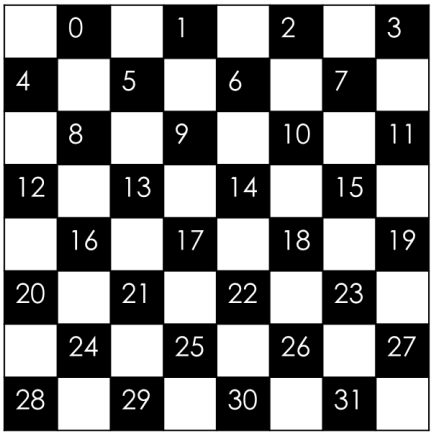
Le code source se trouve sur mon site web (sur la même page que le sujet de TP). Vous allez devoir compléter ce code afin d'implémenter l'IA du joueur MinMax et du joueur Alpha-Beta.

Voici le diagramme UML simplifié des classes du TP que vous allez utiliser:



a) Classe Game

La classe *Game* modélise le moteur du jeu. Il nous permettra également de représenter un état du jeu lors de la recherche.



La fonction *move* permet de déplacer une pièce (pion ou dame) sur le plateau. Elle prend en paramètres deux entiers qui correspondent à des positions sur le damier. La première position est celle de la pièce à déplacer et la seconde position correspond à la case où va être déplacée cette pièce. Elle renvoie un booléen qui indique si le déplacement a pu être effectué. A titre indicatif, j'ai indiqué les entiers correspondant aux indices des différentes cases.

La fonction *isGameOver* indique si le jeu est fini, c'est-à-dire que le joueur courant n'a plus de déplacement possible (ou plus de pièces). La fonction *isP2Turn* renvoie vrai si c'est au deuxième joueur de jouer, et faux sinon.

La fonction *copy* renvoie une copie du jeu (pour être plus précis de la position des pièces sur le plateau). Vous allez utiliser cette fonction pour conserver les positions des pièces avant de tester des actions possibles.

Les fonctions *basicHeuristic* et *goodHeuristic* revoie une valeur qui est une estimation des chances de victoire du joueur courant. Si ce joueur a de bonnes chances de gagner alors la valeur sera positive, et s'il a de bonnes chances de perdre alors cette valeur sera négative. Plus cette valeur sera grande, et plus les chances de victoires seront importantes. Inversement, plus cette valeur est petite et plus les chances de défaite seront grandes. A titre d'exemple, le fonction *basicHeuristic* revoie :

$$nombre\_pions + 3 \times nombre\_dames - nombre\_pions\_adverse - 3 \times nombre\_dames\_adverse$$

b) Classe MinMaxPlayer

Il s'agit d'une des classes qui implémente une intelligence artificielle capable de jouer aux dames. Plus précisément, l'algorithme utilisé pour cette classe est l'algorithme MinMax. L'attribut *player* indique quelle est la position de l'IA (joueur 1 ou joueur 2). Sa valeur est vraie si l'IA joue comme joueur 2, et faux sinon.

La méthode *getMoves* revoie l'ensemble des déplacements possibles pour le joueur courant, c'est-à-dire une liste d'objets de type *Move* qui décrivent les déplacements sur le plateau que les pions du joueur courant peuvent effectuer.

Un objet de la classe *Move* décrit un déplacement d'une pièce. La méthode *getStartIndex* renvoie la position de départ de la pièce (voir plus haut), et la méthode *getEndIndex* revoie la position d'arrivée de la pièce. Ces deux méthodes sont à combiner avec la méthode *move* de la classe *Game*.

La méthode *updateGame* est celle qui est appelée lorsque c'est à l'IA de jouer (et que vous allez donc implémenter). Elle doit donc décider du meilleur déplacement à effectuer dans la situation décrite par l'objet de la classe *Game* pris en paramètre, et effectuer ce déplacement en utilisant la méthode *move* sur ce jeu. Pour cela, elle va faire tourner l'algorithme MinMax (voir plus bas pour plus de détails).

c) Classe AlphaBetaPlayer

Il s'agit d'une autre classe qui implémente une intelligence artificielle capable de jouer aux dames. Plus précisément, l'algorithme utilisé pour cette classe est l'algorithme MinMax utilisant l'élagage alpha-beta.

Cette classe hérite de la classe *MinMaxPlayer*, ce qui lui permet d'utiliser la fonction *getMoves* décrite plus haut.

La fonction *updateGame* est une redéfinition de celle de la classe *MinMaxPlayer*. Elle va faire tourner l'algorithme MinMax en utilisant l'élagage alpha-beta (voir plus bas pour plus de détails) afin de choisir le meilleur déplacement à jouer dans le jeu pris en paramètre, et va effectuer ce déplacement sur ce jeu.

d) Classe StateSet

Il s'agit d'une classe permettant de gérer une table de transposition. Cette table va contenir l'ensemble des états dont on a déjà calculé la valeur minimax. Les valeurs minimax de ces états sont conservées afin que les algorithmes (MinMax et Alpha-Beta) n'aient pas à recalculer la valeur minmax d'un même état plusieurs fois.

Le constructeur permet de construire une table vide.

Lorsque la valeur minmax d'un état est calculé, cette valeur sera stockée dans la table de transposition, en l'associant à l'état correspondant, en utilisant la méthode *add*.

Lorsqu'un nouvel état est rencontré, avant de chercher à calculer la valeur minmax de cet état, l'algorithme (MinMax ou Alph-Beta) va vérifier que cette valeur n'a pas déjà été calculée plus tôt lors du parcours. Pour cela, il va utiliser la méthode *getValue* avec comme paramètre l'état dont on cherche à savoir si il a déjà été visité. Si cet état n'a pas été visité, on donc qu'aucune entrée associée à cet état n'existe dans la table de transposition, alors valeur renvoyée est *null*. Sinon, la valeur renvoyée est un objet de type *Integer* contenant la valeur minmax de l'état (il suffit d'appeler la méthode *intValue* pour obtenir l'*int* correspondant).

II) Travail à effectuer

Comme indiqué plus haut, vous devez compléter les méthodes *updatesGame* des classes *MinMaxPlayer* et *AlphaBetaPlayer*. Vous pouvez vous inspirer du pseudocode présenté dans les slides de cours (vous pouvez créer de nouvelles fonctions dans les classes *MinMaxPlayer* et *AlphaBetaPlayer* pour coller un maximum à ce pseudocode). N'oubliez pas d'utiliser une table de transpositions pour accélérer la recherche (l'utilisation de la table de transposition n'apparait pas dans le pseudo-code du cours). Attention : Une nouvelle table de transposition doit être créée à chaque fois qu'un nouvelle recherche est lancée (donc après chaque nouveau coup).

Les états dans l'arbre de jeu seront des objets de la classe *Game*. N'oubliez pas d'utiliser la méthode *copy* de la classe *Game* avant d'effectuer un mouvement si vous ne voulez pas perdre la configuration initiale. Petite subtilité : lorsqu'un joueur a capturé une pièce et que cette pièce est en position de capturer une autre pièce, alors le joueur « rejoue » (en fait, il s'agit d'un même coup ou une pièce en capture plusieurs). Donc il se peut qu'un joueur joue deux fois (voir plus) de suite. Vous

devez donc vérifier qui est le prochain joueur à jouer avant de faire l'appel récursif à *MinValue* ou *MaxValue* (voir les slides de cours). Pour cela vous devez utiliser la méthode *isP2Turn* et comparer son résultat à l'attribut *player* de la classe Game (si ces deux booléens sont égaux, c'est à l'IA de jouer, sinon c'est à l'autre joueur).

Il ne vous sera pas possible de construire l'arbre en entier car le nombre de coups avant la fin de la partie est très souvent trop grand (c'est le cas dans beaucoup de jeux). Une approche dans ce cas là est d'arrêter la recherche à partir d'une certaine profondeur d'arbre (de manière similaire à depth-limited search). Arrivé à cette profondeur, les états du jeu sont considérés comme des feuilles de l'arbre, même si le jeu n'est pas terminé. Etant donné que le jeu n'est pas terminé, l'issue du jeu n'est pas connue et il n'est pas réellement possible de donner une valeur minmax à l'état. On peut néanmoins utiliser une heuristique, qui va « prédire » la victoire ou la défaite de l'IA (comme décrit plus haut pour les méthodes *basicHeuristic* et *goodHeuristic* de la classe *Game*). La valeur renvoyée dans ce cas là sera donc la valeur retournée par l'heuristique. Si le jeu est réellement terminé, alors la valeur renvoyée sera *Integer.MIN_VALUE* si l'IA a perdu, et *Integer.MAX_VALUE* si elle a gagné.