

TP2: Adversarial search

L'objectif de ce TP est d'utiliser les algorithmes d'exploration d'arbres de jeu (*miniMax*, *alphaBeta* et *DeepningAlphaBeta*) dans des jeux à deux adversaires.

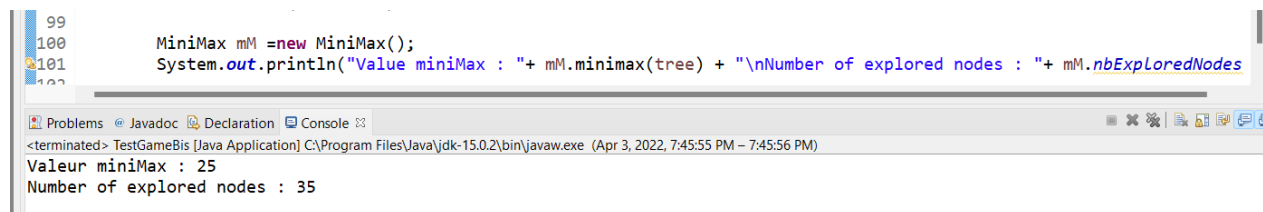
Exercice 1 : Algorithme Minimax

Dans l'algorithme *miniMax* on cherche à minimiser la perte du joueur dans le pire des scénarios. Il se base sur deux fonctions *minVal* et *maxVal* qui s'appellent récursivement jusqu'à l'atteinte d'une feuille. L'une cherche à minimiser l'utilité maximale des nœuds fils et l'autre cherche à maximiser l'utilité minimale des nœuds fils.

```
public Integer minVal( SimpleTwoPlyGameTree<Integer> node ){
    Integer min=Integer.MAX_VALUE;
    if (node.isLeaf() ){
        return node.getValue();
    }else {
        ArrayList<SimpleTwoPlyGameTree<Integer>> children=node.getChildren();
        for( SimpleTwoPlyGameTree<Integer> succ: children) {
            (MiniMax.nbExploredNodes)++;
            Integer maxMinVal=maxVal(succ);
            if ( maxMinVal <= min)
                min=maxMinVal;
        }
        return min;
    }
}
```

```
public Integer maxVal( SimpleTwoPlyGameTree<Integer> node ){
    Integer max=Integer.MIN_VALUE;
    if (node.isLeaf() ){
        return node.getValue();
    }else {
        ArrayList<SimpleTwoPlyGameTree<Integer>> children=node.getChildren();
        for( SimpleTwoPlyGameTree<Integer> succ: children) {
            Integer minMaxVal=minVal(succ);
            (MiniMax.nbExploredNodes)++;
            if ( minMaxVal >= max)
                max=minMaxVal;
        }
        return max;
    }
}
```

Nous testons l'algorithme sur l'arbre de jeu fourni grâce à une instance de la classe *SimpleTwoPlayerGameTree*. Voici le résultat :



The screenshot shows a Java IDE with a code editor and a console window. The code in the editor is as follows:

```
99
100 MiniMax mM =new MiniMax();
101 System.out.println("Value miniMax : "+ mM.minimax(tree) + "\nNumber of explored nodes : "+ mM.nbExploredNodes
102
```

The console window shows the output of the program:

```
<terminated> TestGameBis [Java Application] C:\Program Files\Java\jdk-15.0.2\bin\javaw.exe (Apr 3, 2022, 7:45:55 PM – 7:45:56 PM)
Valeur miniMax : 25
Number of explored nodes : 35
```

L'algorithme renvoie bien la valeur MiniMax attendue (25) et a exploré 35 nœuds. Pour un problème de si petite taille on peut se demander s'il ne serait pas possible de trouver la solution en explorant moins de nœuds.

Exercice 2 : Algorithme AlphaBeta

AlphaBeta est une variation de miniMax où on n'explore pas certains nœuds inutiles en faisant des coupes Alpha ou Beta. Voici le résultat obtenu :

```

102
103     AlphaBeta ab=new AlphaBeta();
104     System.out.println( "\nValue alphaBeta : "+ ab.alphaBetaAlgo(tree, Integer.MIN_VALUE, Integer.MAX_VALUE)
105         +"\nNumber of explored nodes : "+ ab.nbExploredNodes );
106 }
107 }
108

```

Problems Javadoc Declaration Console

<terminated> TestGameBis [Java Application] C:\Program Files\Java\jdk-15.0.2\bin\javaw.exe (Apr 3, 2022, 8:08:46 PM – 8:08:46 PM)

Value alphaBeta : 25
Number of explored nodes : 21

Comme attendu, on retrouve le même résultat qu’avec MiniMax. Mais on explore seulement 21 nœuds au lieu de 35. Pour un problème de plus grande taille, on pourrait s’attendre à avoir un gain plus important comme on le verra avec le tic-tac-toe.

Exercice 3 : Tic-tac-toe

Le tic-tac-toe est un jeu célèbre à deux joueurs. Les deux joueurs placent à tour de rôle des ronds/croix sur une grille 3*3 et chacun cherche à aligner trois croix/ronds.

Nous représentons chaque état **Ex** par un vecteur de **3*3+1=10** éléments :

- $Ex[0] \in \{-1, 1\}$ indique le joueur qui doit jouer
- Les 9 autres entiers décrivent l’occupation de la grille :
 - $Ex[k] == 0$: la cellule (i,j) est inoccupée (avec $k = 3*i + j$, où i désigne le numéro de ligne et j le numéro de colonnes, $i, j \in \{0, 1, 2\}$)
 - $Ex[k] == 1$: la cellule k est occupée par un rond (O)
 - $Ex[k] == -1$: la cellule k est occupée par une croix (X)

Au début du jeu on $Ex[0] = 1$ (le joueur avec les ronds commence la partie) et pour $k \geq 1$ $Ex[k] = 0$ (grille vide).

Une **action** consiste à choisir la position $k \in \{1, \dots, 9\}$ de son prochain pion dans l’une des cases inoccupées de la grille 3*3. Voici une représentation d’un état au cours de la partie. Le tiret (« - ») indique que la case est inoccupée.

```

Machine player, what is your action?
Metrics for Minimax : {expandedNodes=59704}
Metrics for AlphaBeta : {expandedNodes=2223}
Metrics for IDAlphaBetaSearch : {expandedNodes=7525, maxDepth=8}
Chosen action is -5

=====
Current state:
O      -      -
-      X      -
-      -      -

```

Dans cet exemple, nous voyons aussi la différence de performance des algorithmes. MiniMax explore plus de 59700 nœuds alors que AlphaBeta en explore environ 2200 soit 50 fois moins.

Nous avons choisi comme fonction d'utilité une fonction h telle que :

- $H(n) = +\infty$ si le joueur gagne (3 de ses pions alignés)
- $H(n) = -\infty$ si l'adversaire gagne
- $H(n) = \text{NombrePionsAlignésJoueur} - \text{NombrePionsAlignésAdversaire}$ dans tous les autres cas

Cela permet notamment d'avoir un joueur qui maximise ses nombres de pions alignés tout en minimisant celui de l'adversaire. (Voir le code pour l'implémentation). L'IA ainsi construite s'avère être très efficace (elle obtient au moins un match nul dans plus de 20 parties jouées).

```

O      X      O
O      X      -
X      O      X
Human player, what is your action?
6
Chosen action is 6

<--- GAME OVER:   A draw: no winner ! --->

Final grid state:
Current state:
O      X      O
O      X      O
X      O      X

```

Exercice 4 : ConnectFour

Dans le ConnectFour, l'objectif de chaque joueur est d'aligner au moins 4 de ses pions (en horizontal, vertical ou diagonale). Nous représentons les états de la même manière que pour le tic-tac-toe et utilisons la **même fonction d'utilité**. Chaque état Ex par un vecteur de $6*7+1=43$ éléments :

- $Ex[0] \in \{-1, 1\}$ indique le joueur qui doit jouer
- Les 42 autres entiers décrivent l'occupation de la grille :
 - $Ex[k] == 0$: la cellule (i, j) est inoccupée (avec $k = 3*i + j$, où i désigne le numéro de ligne et j le numéro de colonnes, $i, j \in \{0, 1, 2\}$)
 - $Ex[k] == 1$: la cellule k est occupée par un rond (O)
 - $Ex[k] == -1$: la cellule k est occupée par une croix (X)

La grande différence ici est l'intervention de la gravité. Le joueur ne pas choisir n'importe quelle case de la grille. Ainsi une action consiste à choisir la colonne $j \in \{1, 2, \dots, 7\}$.

Comme l'espace de recherche pour ce jeu est très grand, nous n'utiliserons que l'algorithme *DeepningAlphaBeta* en limitant le temps de recherche maximal à 7 secondes.

```

Machine player, what is your action?
Metrics for IDAlphaBetaSearch : {expandedNodes=12678165, maxDepth=13}
Chosen action is 1

=====
Current state:
- - - - -
- - - - -
- - - - -
- - - - -
- - - - -
X - - - -
O - - - -

```

```

=====
Current state:
X  O  -  -  -  -  -
O  X  -  -  -  -  -
X  X  -  -  -  -  -
X  X  X  O  O  -  -
X  O  O  O  X  -  -
O  O  O  X  O  -  -
Machine player, what is your action?
Metrics for IDAlphaBetaSearch : {expandedNodes=48335, maxDepth=12}
Chosen action is 5

```

```

Machine player, what is your action?
Metrics for IDAlphaBetaSearch : {expandedNodes=5, maxDepth=1}
Chosen action is 4

----- <GAME OVER : Machine wins! > -----

Final grid state:
X  O  -  -  -  -  -
O  X  X  X  X  -  -
X  X  O  O  X  X  -
X  X  X  O  O  O  -
X  O  O  O  X  O  -
O  O  O  X  O  O  X

```