Année 2022/2023
Résolution de Problèmes
Master 1, MIV

Série de Travaux Pratiques N°2
Simulation des algorithmes de recherche pour les jeux

L'objectif de ce projet est de concevoir une interface graphique pour simuler le déroulement des algorithmes de recherche pour les jeux (MiniMax, NegaMax et NegaMax with Alpha-Beta Pruning).

Etape 1 : Construction de l'arbre de recherche

La première étape consiste à créer un arbre binaire (chaque nœud possède seulement deux fils : fils gauche et fils droit) de 5 niveaux. Pour cela, vous devez définir les objets et les fonctions suivants :

- La classe *Node* qui va représenter un nœud de l'arbre et qui va contenir les attributs suivants :
  - Un lien vers le nœud parent. On va appeler cette variable *parent ;*
  - Un lien vers les deux nœuds fils. On va appeler ces deux variables *leftChild et rightChild ;*
  - La valeur du nœud qui est Initialement vide sauf pour les nœuds feuilles. On va l'appeler *value ;*
  - Si besoin, la position du nœud dans l'interface graphique (cette information va vous faciliter la mise à jour de l'interface graphique lors du déroulement des algorithmes de recherche). On va appeler cette variable *position ;*
  - Si besoin, un lien vers le nœud fils à partir duquel on a obtenu la valeur lors de l'application de l'algorithme de recherche. On va appeler cette variable *path ;*
- La fonction qui va créer un arbre binaire dont tous les nœuds sont initialement vides (n'ont pas de valeur) sauf pour les nœuds feuilles dont les valeurs sont des nombres aléatoires.
- La fonction qui va afficher l'arbre initial dans l'interface graphique, comme le montre la Figure 1.

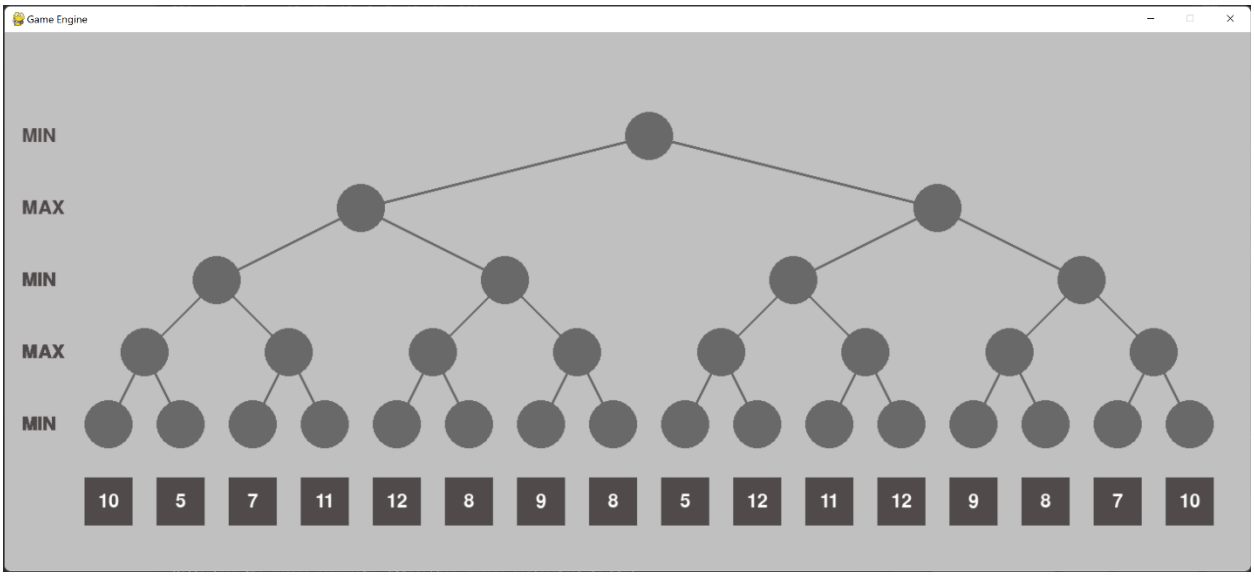

Figure 1 Arbre de recherche initial

Etape 2 : Implémentation de l'algorithme de recherche MiniMax

La deuxième étape consiste à implémenter une fonction récursive qui va à la fois, permettre d'appliquer l'algorithme de recherche MiniMax sur l'arbre défini dans l'étape 1, mais aussi d'afficher le déroulement de l'algorithme dans l'interface graphique. Cette fonction est définie par l'Algorithme -1-. Une simulation de la fonction MiniMax est représentée dans la Figure 2.

Etape 3 : Implémentation de l'algorithme de recherche NegaMax

La troisième étape consiste à implémenter une fonction récursive qui va à la fois, permettre d'appliquer l'algorithme de recherche NegaMax sur l'arbre défini dans l'étape 1, mais également d'afficher le déroulement de l'algorithme dans l'interface graphique. Cette fonction est définie par l'Algorithme 2. Une simulation de la fonction NegaMax est représentée dans la Figure 3.

Etape 4 : Implémentation de l'algorithme de recherche NegaMax with Alpha-Beta Pruning

La dernière étape consiste à inclure le mécanisme d'élagage Alpha-Beta dans la fonction NegaMax précédente et à afficher le déroulement de l'algorithme dans l'interface graphique. Cette fonction est définie par l'Algorithme 3. Une simulation de la fonction NegaMax with Alpha-Beta Pruning est représentée dans la Figure 4.

```
MiniMax (node, player, depth) // Initial depth is 5
Begin
  If (depth == 1)
  {
    // Display the current node's value and mark it as explored
  }
  Else
  {
    // Mark the current node as explored
    listChildren = [node.leftChild, node.rightChild]
    If (player == MAX) // MAX = +1
    {
          bestValue = -inf
          bestPath = None
          For each child in listChildren
          {
             // Mark the link between the current node and the child node as explored
             MiniMax (child, -player, depth-1)  // Apply the MiniMax function on each child
             If child.value > bestValue
             {
                   bestValue = child.value
                   bestPath = child
             }
          }
    Else  // If the player is MIN (MIN = -1)
    {
          bestValue = +inf
          bestPath = None
          For each child in listChildren
          {
             // Mark the link between the current node and the child node as explored
             MiniMax (child, -player, depth-1)  // Apply the MiniMax function on each child
             If child.value < bestValue
             {
                   bestValue = child.value
                   bestPath = child
             }
          }
    }
    node.value = bestValue
    node.path = bestPath
    // Display the best path and the current node's value
  }
End
```

Algorithme - 1 -

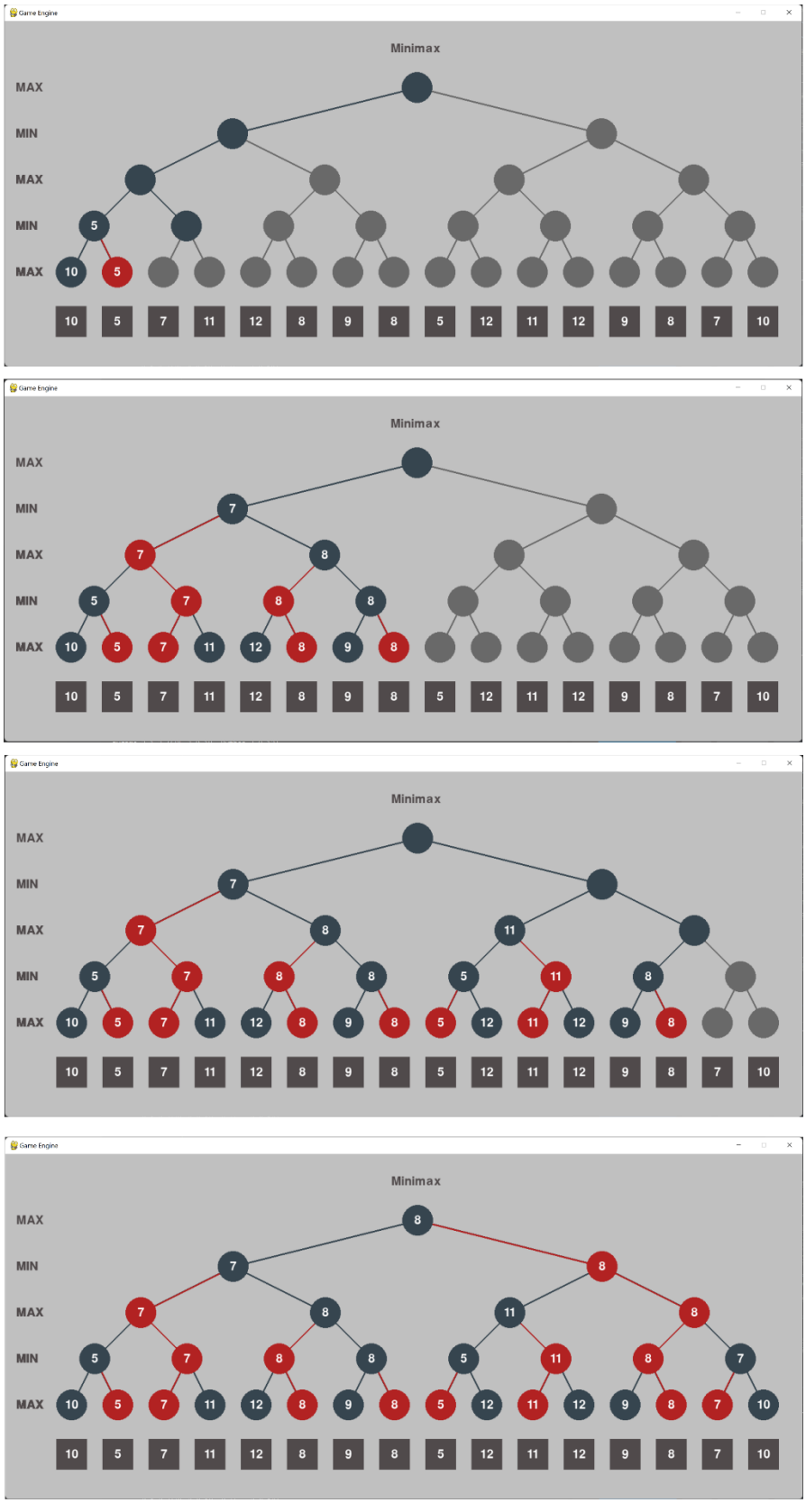

Figure 2 Simulation de l'algorithme MiniMax

```
NegaMax (node, player, depth) // Initial depth is 5
Begin
  If depth == 1
  {
    If player == MIN
    {
          node.value = -node.value
    }
    // Display the current node's value and mark it as explored
  }
  Else
  {
    // Mark the current node as explored
    listChildren = [node.leftChild, node.rightChild]
    bestValue = -inf
    bestPath = None
    For each child in listChildren
    {
        // Mark the link between the current node and the child node as explored
        NegaMax (child, -player, depth-1)  // Apply the NegaMax function on each child
        child.value = -child.value
        If child.value > bestValue
        {
              bestValue = child.value
              bestPath = child
        }
     }
     node.value = bestValue
     node.path = bestPath
     // Display the best path and the current node's value
  }
End
```

Algorithme - 2 -

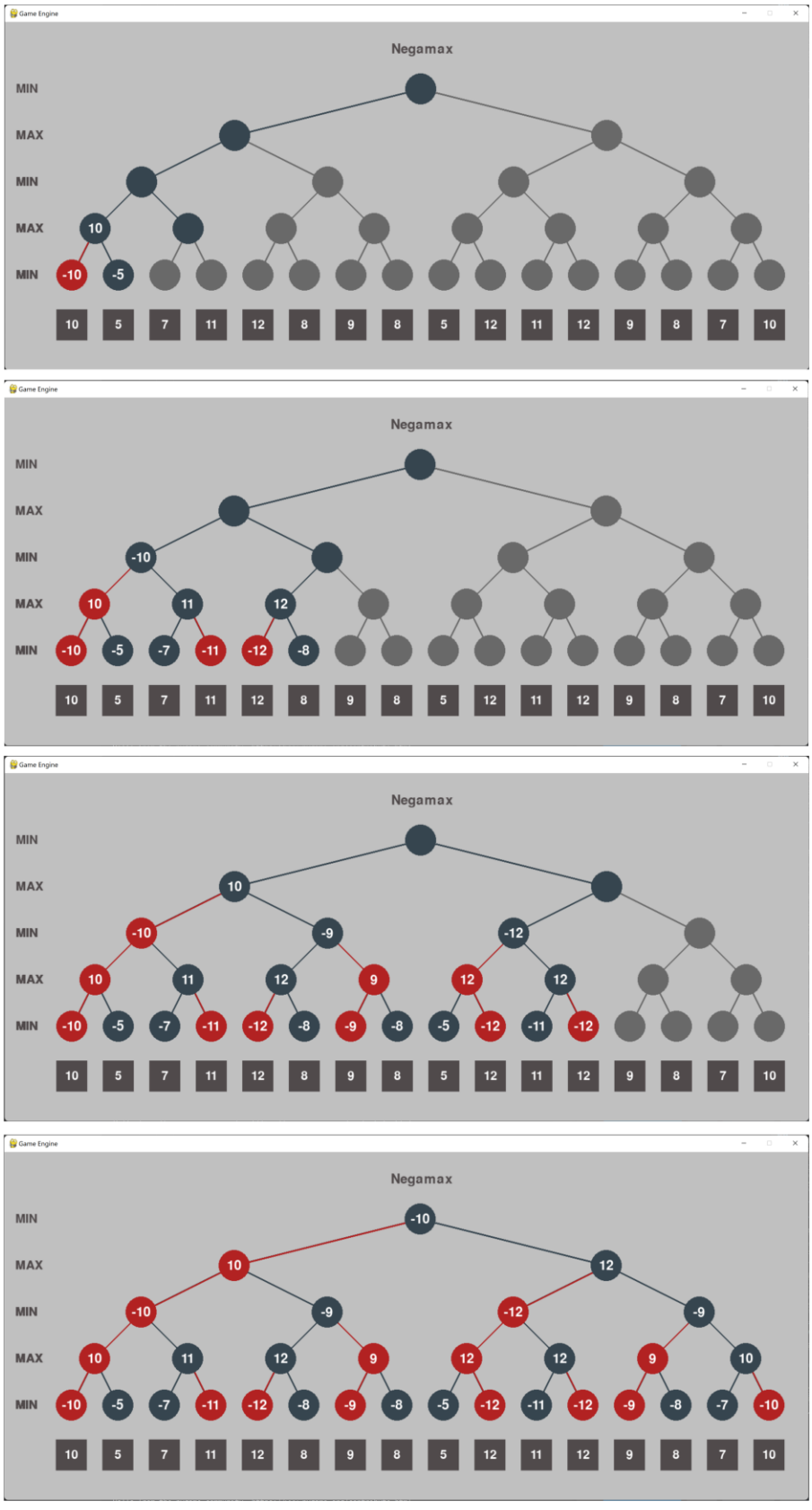

Figure 3 Simulation de l'algorithme NegaMax

```
NegaMaxAlphaBetaPruning (node, player, depth, alpha, beta)
// Initially, depth=5, alpha=-inf and beta=+inf
Begin
  If depth == 1
  {
    If player == MIN
    {
          node.value = - node.value
    }
    // Display the current node's value and mark it as explored
    // Display the values of alpha and beta
  }
  Else
  {
    // Mark the current node as explored
    // Display the values of alpha and beta
    listChildren = [node.leftChild, node.rightChild]
    bestValue = -inf
    bestPath = None
    For each child in listChildren
    {
       // Mark the link between the current node and the child node as explored
       NegaMaxAlphaBetaPruning (child, -player, depth-1, -beta, -alpha)
       child.value = - child.value
       If child.value > bestValue
       {
           bestValue = child.value
           bestPath = child
       }
       If bestValue > alpha
       {
           alpha = bestValue
           // Display the new value of alpha
       }
       If beta <= alpha
           Break
    }
    node.value = bestValue
    node.path = bestPath
    // Display the best path and the current node's value
  }
End
```

Algorithme - 3 –

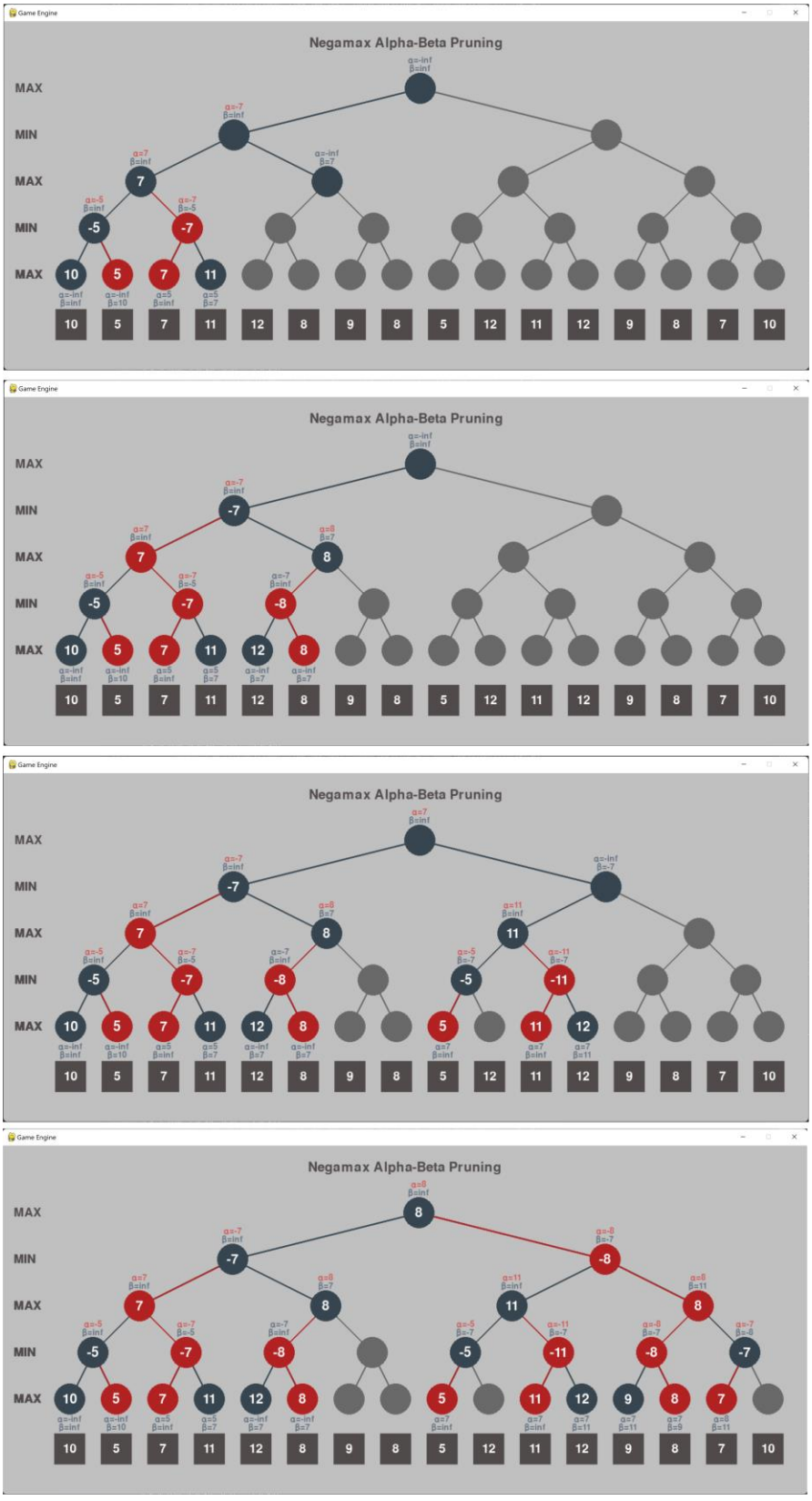

Figure 4 Simulation de l'algorithme NegaMax with Alpha-Beta Pruning