Projet d'algorithmitique : Jeu avec une IA

C. RAHMANI, A.WASBAUER, Z. HOUSNI, D. MAM

May 9, 2023

POUR JOUER C'EST ICI ! CLIQUEZ !

GitHub

Contents

1 Introduction

1.1 Objectifs du projet

L'objectif de ce projet est de développer un jeu et une intelligence artificielle (IA) capable de jouer à un jeu à deux joueurs avec information complète. Pour cela, nous allons nous baser sur l'algorithme de min-max, éventuellement amélioré par l'approche alpha-bêta. Les jeux de stratégie combinatoires abstraits se prêtent particulièrement bien à cette approche, c'est pourquoi nous avons choisi Othello comme jeu de référence pour notre projet.

1.2 Types de jeux

Un jeu est une station où des individus (les joueurs) sont conduits à faire des choix(action) parmi un certain nombre de cas possibles dans un cadre défini à l'avance (les règles de jeu).

- jeux plateaux à deux joueurs
- l'ensemble des stratégies des joueurs est finis
- joue à tour de rôle (séquentiel)
- informations complètes : chaque joueur connaît ses possibilités d'actions, celle des autres joueurs, les gains associés aux actions et les motivations des autres joueurs.

1.3 Brève explication de la règle du jeu Othello

L'Othello est un jeu de stratégie combinatoire abstrait pour deux joueurs. Il se joue sur un plateau de 8x8 cases, sur lequel chaque joueur dispose de 32 pions bicolores, l'un étant noir et l'autre blanc. Le but du jeu est de capturer un maximum de pions de la couleur adverse en retournant les pions adverses qui sont situés entre deux pions du joueur. Le joueur ayant le plus de pions de sa couleur à la fin de la partie est déclaré vainqueur.

Le joueur doit également, en posant son pion, encadrer un ou plusieurs pions adverses entre le pion qu'il pose et un pion à sa couleur, déjà placé sur l'othellier. Cette prise en sandwich peut se faire aussi bien horizontalement ou verticalement, qu'en diagonale.

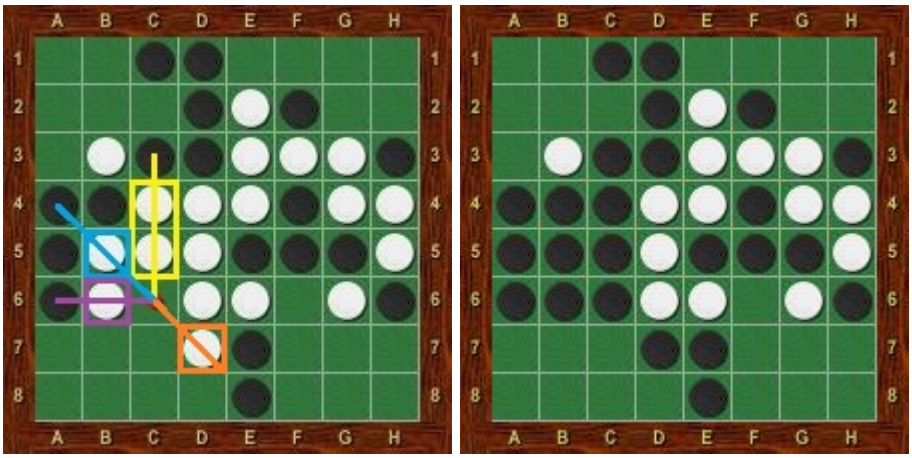


Figure 1: Les pions impactés par le placement d'un pion noir en c6.

La partie se termine lorsque tous les espaces de la grille ont été remplis, ou lorsque l'un des joueurs ne peut plus jouer de coups légaux. Le joueur ayant le plus de pions de sa couleur sur le plateau est déclaré vainqueur.

2 Conception du jeu Othello

2.1 Langage de programmation utilisé

Dans ce projet, nous avons choisi Javascript comme langage de programmation car il est largement utilisé dans le développement Web, c'est un langage flexible qui comme Python permet de travailler sur toutes les parties d'un projet, pas seulement le développement de l'IA.

Javascript offre la possibilité de faire de la programmation orientée objets, de créer des graphismes attrayants en utilisant Three.js, d'accélérer les performances en utilisant WebGL, et d'accroître la portabilité de notre projet en utilisant des technologies Web modernes.

Quant au déploiement et à l'hébergement de notre site, nous avons utilisé une plateforme cloud, Vercel.

2.2 Implémentations du jeu

En résumé, nous avons utilisé JavaScript, Three.js et WebGL pour développer le jeu d'Othello. Voici une explication détaillée en plusieurs partie de notre Othello, sans l'interface.

2.2.1 Création du plateau de jeu

```
class Board {
  constructor() {
    // group creation
    this.board = new THREE.Group();
    this.boardChunk = new THREE.Group();
    this.disks = new THREE.Group();

    //Creation of tokens

    this.currentPlayer = 2;
    this.gridLogic = [
      [0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0]
    ];

    this._createBoard();
    this._createInitialDisks();
    this.board.add(this.boardChunk, this.disks)

  }
}
```

La classe nommée "Board"' permet d'initialiser, de stocker et d'interagir à l'aide de méthode, le plateau de jeu et/ou les pions. C'est notamment le constructeur de la classe "Board"' initialise plusieurs groupes pour stocker les éléments du jeu, comme le plateau et les disques des joueurs.

On utilise comme structure de données un tableau multidimensionnel "gridLogic" qui définit une matrice contenant des entiers. Cette matrice est une grille de 8 rangées et 8 colonnes initialisée à 0. Une méthode appelée "_createInitialDisks()" ajoute les quatre premiers jetons aux positions de départ dans le plateau de jeu et modifie dans la matrice les 0 en 1 ou en 2, selon la position où les deux premiers pions noirs et blancs sont placés, soient {(3,3); (3,4); (4,3); (4,4)}.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

2.2.2 Le joueur clique !

Plusieurs fonctions permettent le bon placement des jetons mais avant que nos jetons soient placés l'utilisateur doit cliquer sur la case.

```
function addClickListenerToBoard(othelloBoard, camera, onSquareClick) {
    raycaster = new THREE.Raycaster();
    mouse = new THREE.Vector2();
  function onClick(event) {
    mouse.x = (event.clientX / window.innerWidth) * 2 - 1;
    mouse.y = -(event.clientY / window.innerHeight) * 2 + 1;

    // update the picking ray with the camera and mouse position
    raycaster.setFromCamera(mouse, camera);
    // calculate objects intersecting the picking ray
    const intersects = raycaster.intersectObjects(othelloBoard.boardChunk.children,
    true);

    if (intersects.length > 0) {
      const cellPosition = {x: intersects[0].object.position.x, z: intersects[0].
    object.position.z};
      onSquareClick(cellPosition.x, cellPosition.z);
    }
  }

  //desactivation de l'event click si c'est le tour de l'IA

  if (othelloBoard.currentPlayer === 2){
    canvas.removeEventListener('click', onClick)
  }
  if (othelloBoard.currentPlayer === 1){
    canvas.addEventListener('click', onClick)
  }
}
```

La fonction 'addClickListenerToBoard' écoute le joueur lorsqu'il clique sur une cellule du plateau. La fonction calcule la positon de la souris avec la caméra et met à jour le raycatser. Elle calcule ensuite les objets qui intersectent avec le raycaster et détermine quelle cellule a été cliquée. Enfin, elle appelle la fonction onSquareClick avec la ligne et la colonne de la cellule cliquée.

Fonction onSquareClick(row, column) en pseudo-code :

```
fonction onSquareClick(row, column) {
  si canMove(1) == false && canMove(2) == false alors
    gameOver(othelloBoard.gridLogic);
  fin si

   //si la case a deja un disque

  si gridLogic[row][column] != 0  alors
    afficher un message dans la console indiquant la case cliquee;
    retourner;
  fin si

    // si le joueur peut cliquer sur la case alors obtenir tous les jetons affectes
    par le coup

  si canClickSquare(playerID, row, column) alors
    affectedDisks = getAffectedDisk(playerID, row, column)
    flipDisks(affectedDisks)
  fin si

    //  si aucun joueur ne peut plus jouer alors appeler la fonction gameOver avec l'
    etat actuel de la grille de jeu

  si canMove(1) == false et canMove(2) == false alors
    gameOver()
  fin si

    //si le joueur actuel est le joueur 1 et qu'il peut jouer alors placer un disque a
     la position indiquee et mettre a jour le score

  si playerID == 1 && CanMove(1) alors
    placeDisk(row, column)
    setScore()
  fin si

    // si le joueur actuel est le joueur 2 et qu'il peut jouer

  si playerID == 2 && canMove(2) alors
      attendre 2 seconde
      turnAI() // on appelle une fonction qui fait tourner l'IA
  fin si
```

La fonction onSquareClick est appelée lorsque le joueur clique sur une cellule. La fonction vérifie premièrement si la cellule est vide ou non. Si la cellule n'est pas vide, la fonction renvoie un message dans la console et retourne rien.

Si la cellule est vide, la fonction appelle la fonction canClickSquare pour déterminer si le joueur peut placer un jeton dans cette cellule en appelant la fonction getAffectedDisk(playerID, row, column). Si le joueur peut placer un jeton, la fonction récupère les jetons affectés et les retourne.

La fonction vérifie ensuite si le jeu est terminé à l'aide de la fonction gameOver() en conséquence.

Si le jeu n'est pas terminé, la fonction vérifie si le joueur en cours est l'IA. Si c'est le cas, la fonction appelle la fonction turnAI après un délai de 2 secondes. Sinon, la fonction place un jeton sur le plateau et met à jour le score.

```
function canClickSquare(id, row,column){
  var affectedDisks = getAffectedDisk(id, row, column);
```

```
  if (affectedDisks.length == 0) {return false;}
  else {return true;}
}
```

La fonction canClickSquare vérifie si le joueur peut placer un jeton sur le plateau à la ligne et la colonne spécifiées. La fonction récupère les jetons affectés dans un tableau et vérifie si la longueur du tableau est supérieure à zéro. Si c'est le cas, la fonction renvoie vrai. Sinon, la fonction renvoie faux.

Ainsi si le joueur clique sur une case cette fonction permet de déterminer si le joueur à le droit d'y placer un pion.

2.2.3 Placement des jetons

La méthode placeDisk(row, column) définit dans la classe 'Board' prend une ligne et une colonne et crée un nouveau disque sur le plateau à cette position. La fonction crée notamment le nouveau jeton en 3D à l'aide de Three.js et ajoute ces jetons à un groupe nommés disks où l'on pourra retrouver tous les jetons et appelle la fonction setCurrentTurn(). Une méthode qui permet de changer de joueur.

La fonction vérifie aussi si la cellule de la grille est vide avant de placer un disque. S'il y a déjà un disque à cette position, la fonction afffiche un message d'erreur.

2.2.4 Trouver les jetons affectés

Avant de retourner les jetons, on appelle dans onSquareClick(row, column) la fonction getAffected-Disk(id, row, column) qui va nous donner les positions des jetons à retourner.

La fonction "getAffectedDisk" utilisé dans plusieurs fonctions comme canClickSquare(id, row,column) permet de déterminer les jetons affectés par la pose d'un nouveau jeton à une position donnée sur le plateau de jeu.

- Pour les 8 directions (haut, bas, gauche, droite et les 4 diagonales).
- La fonction parcourt la matrice dans chaque direction jusqu'à atteindre une case vide ou un jeton de la même couleur.
- Elle garde une trace de tous les emplacements des jetons de couleur opposée rencontrés sur la route.
- Si la case terminale est de la même couleur que le joueur actuel, ajoutez ces emplacements à la liste affectedTokens.

```
fonction getAffectedDisk(id, row, column) :

    affectedTokens = tableau vide

    // Detecte s'il y a des jetons a retourner a la droite
    couldBeAffected = tableau vide
    columnIterator = column

    Tant que columnIterator < 7 :
        columnIterator += 1
        valueAtSpot = gridLogic[row][columnIterator]
        Si valueAtSpot == 0 ou valueAtSpot == id :
            Si valueAtSpot == id :
                affectedTokens = affectedTokens.concat(couldBeAffected)
            Pause
        Sinon :
            tokenLocation = {row: row, column: columnIterator}
```

```
            couldBeAffected.push(tokenLocation)

    // Detecte s'il y a des jetons a retourner a la gauche
    couldBeAffected = tableau vide
    columnIterator = column

    Tant que columnIterator > 0 :
        columnIterator -= 1
        valueAtSpot = gridLogic[row][columnIterator]
        Si valueAtSpot == 0 ou valueAtSpot == id :
            Si valueAtSpot == id :
                affectedTokens = affectedTokens.concat(couldBeAffected)
            Pause
        Sinon :
            tokenLocation = {row: row, column: columnIterator}
            couldBeAffected.push(tokenLocation)

    // Detecte s'il y a des jetons a retourner au dessus
    couldBeAffected = tableau vide
    rowIterator = row

    Tant que rowIterator > 0 :
        rowIterator -= 1
        valueAtSpot = gridLogic[rowIterator][column]
        Si valueAtSpot == 0 ou valueAtSpot == id :
            Si valueAtSpot == id :
                affectedTokens = affectedTokens.concat(couldBeAffected)
            Pause
        Sinon :
            tokenLocation = {row: rowIterator, column: column}
            couldBeAffected.push(tokenLocation)

   // Detecte s'il y a des jetons a retourner en bas
    couldBeAffected = tableau vide
    rowIterator = row

    Tant que rowIterator < 7 :
        rowIterator += 1
        valueAtSpot = gridLogic[rowIterator][column]
        Si valueAtSpot == 0 ou valueAtSpot == id :
            Si valueAtSpot == id :
                affectedTokens = affectedTokens.concat(couldBeAffected)
            Pause
        Sinon :
            tokenLocation = {row: rowIterator, column: column}
            couldBeAffected.push(tokenLocation)

   // Detecte s'il y a des jetons a retourner en bas a droite en diagonale
    couldBeAffected = tableau vide
    rowIterator = row

    columnIterator = column
    Tant que rowIterator < 7 et columnIterator < 7 :
        rowIterator += 1
        columnIterator += 1
        valueAtSpot = gridLogic[rowIterator][columnIterator]
        Si valueAtSpot == 0 ou valueAtSpot == id :
            Si valueAtSpot == id :
                affectedTokens = affectedTokens.concat(couldBeAffected)
            Pause
        Sinon :
            tokenLocation = {row: rowIterator, column: column}
            couldBeAffected.push(tokenLocation)

   // Detecte s'il y a des jetons a retourner en bas a gauche en diagonale
    couldBeAffected = tableau vide
    rowIterator = row

    columnIterator = column
    Tant que rowIterator > 0 et columnIterator > 0 :
        rowIterator -= 1
```

```
        columnIterator -= 1
        valueAtSpot = gridLogic[rowIterator][columnIterator]
        Si valueAtSpot == 0 ou valueAtSpot == id :
            Si valueAtSpot == id :
                affectedTokens = affectedTokens.concat(couldBeAffected)
            Pause
        Sinon :
            tokenLocation = {row: rowIterator, column: column}
            couldBeAffected.push(tokenLocation)

    // Detecte s'il y a des jetons a retourner en haut a droite en diagonale
     couldBeAffected = tableau vide
     rowIterator = row

     columnIterator = column
     Tant que rowIterator > 0 et columnIterator < 7 :
        rowIterator -= 1
        columnIterator += 1
        valueAtSpot = gridLogic[rowIterator][columnIterator]
        Si valueAtSpot == 0 ou valueAtSpot == id :
            Si valueAtSpot == id :
                affectedTokens = affectedTokens.concat(couldBeAffected)
            Pause
        Sinon :
            tokenLocation = {row: rowIterator, column: column}
            couldBeAffected.push(tokenLocation)

    // Detecte s'il y a des jetons a retourner en haut a gauche en diagonale
     couldBeAffected = tableau vide
     rowIterator = row

     columnIterator = column
     Tant que rowIterator < 7 et columnIterator > 0 :
        rowIterator += 1
        columnIterator -= 1
        valueAtSpot = gridLogic[rowIterator][columnIterator]
        Si valueAtSpot == 0 ou valueAtSpot == id :
            Si valueAtSpot == id :
                affectedTokens = affectedTokens.concat(couldBeAffected)
            Pause
        Sinon :
            tokenLocation = {row: rowIterator, column: column}
            couldBeAffected.push(tokenLocation)

    return affectedTokens;
```

Pour expliquer plus précisément une étape de cette fonction :

La fonction prend en entrée l'identifiant du joueur et la position de ligne et de colonne d'un mouvement potentiel et renvoie une liste de tous les jetons de l'adversaire qui seraient affectés par ce mouvement.

La fonction commence par créer un tableau vide affectedTokens qui stockera ultérieurement toutes les positions des jetons de l'adversaire qui seront affectées par le mouvement du joueur.

Ensuite, la fonction vérifie dans toutes les huit directions, à partir de l'emplacement actuel, et se déplace dans cette direction jusqu'à ce qu'elle atteigne une case vide ou la couleur du joueur. En cours de route, la fonction garde une trace de toutes les positions des jetons de l'adversaire qu'elle rencontre, qui sont stockées dans un tableau couldBeAffected.

Si la case terminale dans une direction est de la couleur du joueur, la fonction ajoute tous les jetons de l'adversaire dans cette direction à la liste affectedTokens.

Enfin, la fonction renvoie la liste affectedTokens contenant tous les jetons de l'adversaire qui seront affectés par le mouvement du joueur.

Enfin, pour retourner les jetons, on utilise flipDisks(affectedDisks). Cette fonction permet de retourner les pions affectés en inversant la couleur des pions de l'adversaire avec ceux du joueurs actifs.

```
function flipDisks(affectedDisks){
   /*
    for all the items in the list: affectedTokens:
        if the token at that has spot as value
          make it a 2
        else
          make it a 1
    */

  Pour i allant de 0 a affectedDisks.length
    let spot = affectedDisks[i]
    si gridLogic[spot.row][spot.column] === 1 alors
        gridLogic[spot.row][spot.column] = 2;
    sinon
        gridLogic[spot.row][spot.column] = 1;
    fin si

    animateRotationDisk(spot.row,spot.column)

  fin pour
}
```

Cette fonction prend en entrée une liste d'emplacements de pions affectés et ne retourne rien. La fonction parcourt la liste des emplacements affectés et pour chaque emplacement, elle vérifie si le pion sur cet emplacement est actuellement une pièce du joueur actif ou non. Si le pion est une pièce du joueur actif, il est transformé en une pièce de l'adversaire. Sinon, il est transformé en une pièce du joueur actif. La fonction utilise ensuite la méthode foundChild de l'objet othelloBoard pour mettre à jour la position de chaque pion et applique une animation de rotation pour chaque pion modifié.

2.2.5 Fonctionnement du système de score

À chaque fois qu'un jeton est placé et qu'on renverse les jetons affectés, on appelle la fonction setScore(score):

```
fonction setScore(score) :
Debut
  initialiser le score pour le joueur noir et le joueur blanc a 0;

  pour chaque case dans la grille de jeu faire
     player = l'entier dans la matrice affecte a la case

    si player = noir alors
      incrementer le score du joueur noir;
    sinon si player = blanc alors
        incrementer le score du joueur blanc;
    fin si
  fin pour

  mettre a jour l'affichage du score sur la page web en utilisant les valeurs du score
     pour le joueur noir et le joueur blanc.
fin
```

La fonction parcourt toutes les cases de la grille de jeu et incrémente les scores du joueur noir ou du joueur blanc si un disque correspondant est présent dans la case. Enfin, la fonction met à jour l'affichage du score sur la page web en utilisant les nouvelles valeurs du score.

2.3 Bibliothèques, Frameworks utilisés et Interfaces

Nous avons utilisé les bibliothèques Three.js et WebGL pour faciliter le développement de l'interface utilisateur et pour accélérer les rendus graphiques en utilisant le GPU.

```
import * as THREE from 'three';
```

Three.js est une bibliothèque et un framework JavaScript open-source qui facilite la création de scènes et d'objets 3D en utilisant la technologie WebGL pour le rendu graphique. Il offre une interface simplifiée pour la manipulation d'objets 3D, de l'éclairage et des textures, ainsi que pour la gestion des animations et des interactions avec l'utilisateur. Pygame(python), pour comparer, est davantage axé sur les jeux 2D, c'est pour cela que nous avons choisi d'implémenter notre jeu en javascript.

WebGL est une API de bas niveau pour le rendu graphique en 3D dans les navigateurs web. Elle permet d'utiliser la puissance du GPU de l'ordinateur pour le traitement des graphismes pour accélérer les performances et est supportée par la plupart des navigateurs modernes. Pygame, quant à lui, utilise principalement le CPU pour le rendu des graphismes, ce qui peut être moins performant pour les jeux en 3D. WebGL permet de créer des jeux et des applications 3D complexes directement dans le navigateur web, sans avoir besoin de plugins supplémentaires.

```
import { OrbitControls } from 'three/examples/jsm/controls/OrbitControls.js'
```

OrbitControls est une fonctionnalité de la bibliothèque Three.js qui permet de contrôler la caméra en utilisant la souris ou les touches du clavier. Elle permet à l'utilisateur de se déplacer autour d'un objet 3D en faisant pivoter la caméra autour de celui-ci.

```
import TWEEN from '@tweenjs/tween.js'
```

TWEEN est une bibliothèque JavaScript qui permet de créer des animations fluides et réactives pour les objets HTML et WebGL. Elle facilite la création d'animations complexes en utilisant une syntaxe simple et intuitive. TWEEN permet également de contrôler l'animation en modifiant des propriétés comme la durée, la fonction d'interpolation et le délai.

La méthode _createBoard() permet de créer le plateau en 3D, il est composé de 64 cubes. On a choisit d'alterner une couleur clair et foncé contrairement au plateau original de couleur uniforme pour différencier les cases.

```
_createBoard() {
   let i = 0;
   // Create square for the platform
   for (let row = 0; row < 8; row++) {
     for (let column = 0; column < 8; column++) {
       let cube;
       var lightGreen =  new THREE.MeshStandardMaterial({color: colors.color2})
       var darkGreen =  new THREE.MeshStandardMaterial({color: colors.color1})
       lightGreen.roughness = 0.1
       darkGreen.roughness = 0.1
       if (column % 2 == 0)
       {
         cube = new THREE.Mesh(cubeGeometry, row % 2 == 0 ? lightGreen : darkGreen);
       }
       else
       {
         cube = new THREE.Mesh(cubeGeometry, row % 2 == 0 ? darkGreen : lightGreen);
       }
     cube.userData.idnum = i;
     i++;
     cube.position.set(row, 0, column);
     this.boardChunk.add(cube);
     }
   }
 }
```

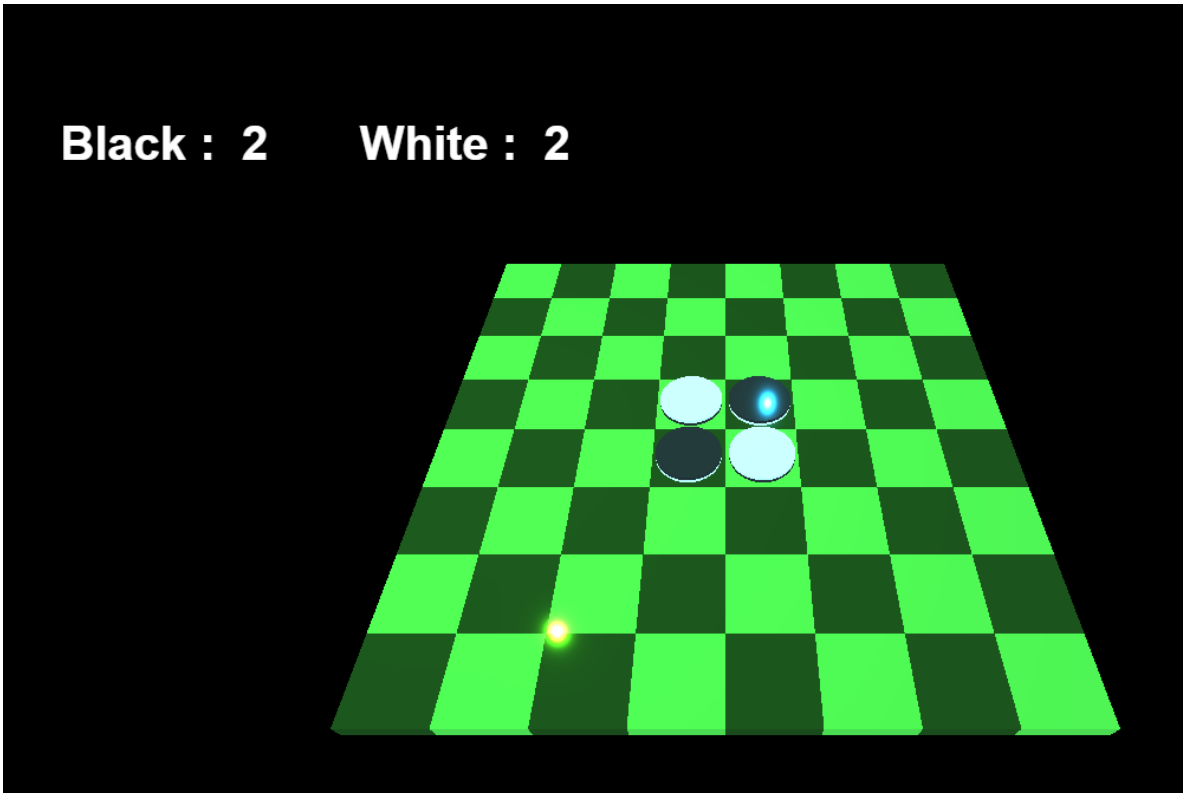


Figure 2: Le plateau de jeu en 3D

Les jetons eux sont composés de 2 cylindres collés deux à deux de couleurs blanc et noir, stockés dans un groupe disk afin de créer un objet manipulable et comme indiqué plus haut stockés à leur tour dans le groupe disks faisant partie des attributs de la classe Board qu'on peut par la suite parcourir pour retrouver les enfants ou un enfant de l'objet disks.

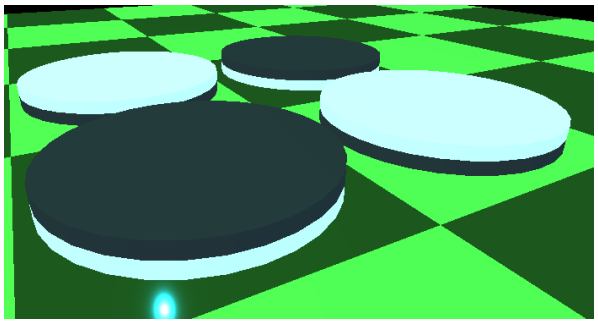


Figure 3: Le plateau de jeu en 3D

Certaines fonctions aident en background pour faire le lien entre interface et logique de jeu.

Par exemple, on définit dans notre classe la méthode foundChild(row, column) pour retrouver des objets (jetons) en fonction de leur position sur le plateau dans le groupe disks pour pouvoir par la suite exécuter des modifications sur l'objet ou juste effectuer une rotation comme avec la fonction flipDisks (affectedDisks).

```
foundChild(row, column) {
    if (this.getgridLogic(row, column)!= 0)
    {
      var childToFind = null;
      for (var i = 0; i < this.disks.children.length; i++)
      {
        var child = this.disks.children[i];

        if (child.position.x === row && child.position.z === column) {
          childToFind = child;
        }
      }
      return childToFind;
    } else console.error('No disk placed to found')

}
```

3 Conception de l'IA

Les algorithmes de recherche de jeux, nous aident à gagner du temps lorsque l'on cherche une solution. Ils sont souvent utilisés quand on a un système de jeux avec adversaire, c'est-à-dire, deux ou plusieurs personnes en train de jouer. Il s'agira dans ce cadre à chercher à maximiser les gains et à minimiser celui de l'adversaire en étudiant toutes les possibilités de jeux pour empêcher le joueur de jouer, en maximisant les gains.

En ce qui concerne l'IA, nous avons implémenté l'algorithme Minimax avec élagage alpha-beta comme demandé dans l'énoncé.

3.1 Description des algorithmes d'IA implémentés : Minimax, Alpha-beta pruning

Problème de recherche : l'adversaire est imprévisible, on doit donc choisir un mouvement pour chaque réponse de l'adversaire.

3.1.1 Algorithme Minimax

L'algorithme MinMax est une technique d'exploration de l'arbre de jeu utilisée dans les jeux à deux joueurs avec information complète, tels que les échecs, le jeu de dames, etc. Il consiste à considérer que chaque joueur va jouer de manière à maximiser son score, tout en supposant que l'adversaire va jouer de manière à minimiser ses pertes.

Ainsi, l'algorithme va simuler tous les coups possibles pour les deux joueurs jusqu'à une certaine profondeur de l'arbre de jeu, en attribuant une valeur heuristique à chaque position finale (victoire, défaite ou match nul).

Principe :

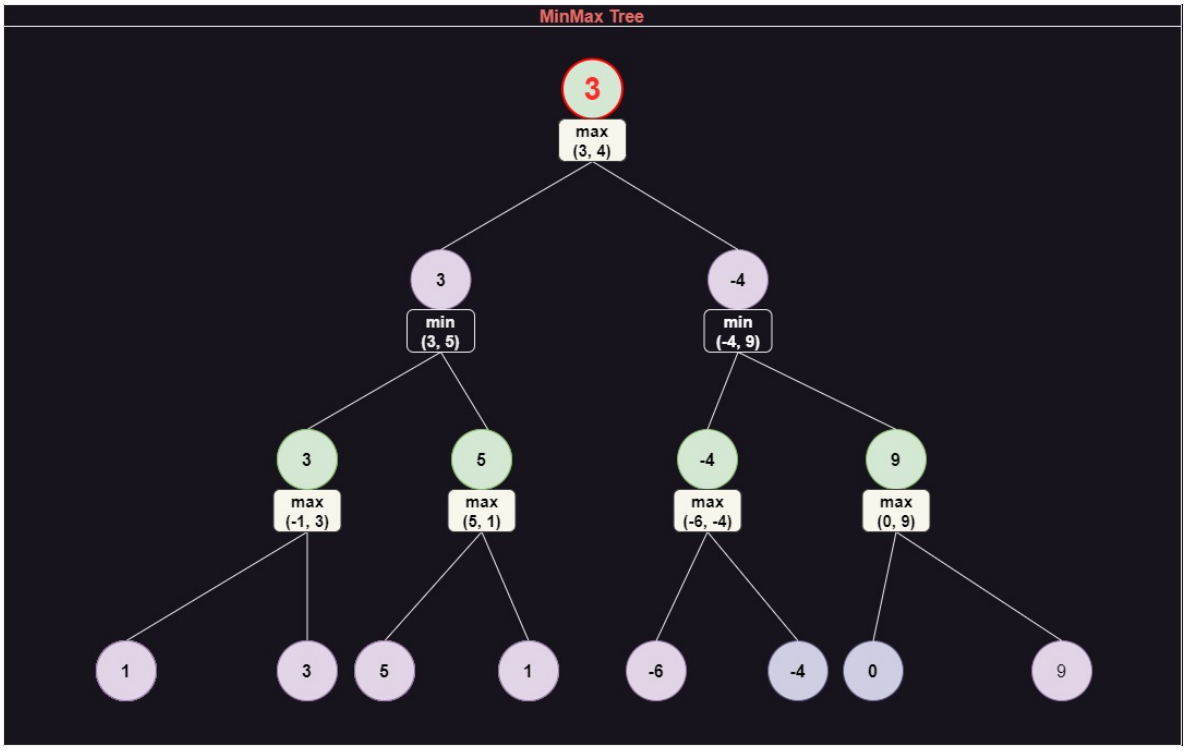
L'algorithme minimax visite l'arbre de jeu pour faire remonter à la racine une valeur qui est calculée récursivement de la façon suivante :

minimax(p) = f(p) si p est une feuille de l'arbre où f est une fonction d'évaluation de la position du jeu ;
minimax(p) = max(minimax(O_1), ..., minimax(O_n)) si p est un nœud Joueur avec fils $O_1, \ldots, O_n$;
minimax(p) = min(minimax(O_1), ..., minimax(O_n)) si p est un nœud Opposant avec fils $O_1, \ldots, O_n$.

Dans le schéma ci-dessous, les nœuds verts représentent les nœuds joueurs et les lilas les nœuds opposants. Pour déterminer la valeur de la racine (encerclée en rouge) on choisit la valeur maximum de l'ensemble des fils de la racine. Il faut donc déterminer les valeurs des fils qui reçoivent chacun la valeur minimum stockée dans leurs fils (lilas = opposants). Ainsi de suite récursivement jusqu'aux feuilles de chaque branche.

Modélisation de l'arbre de jeu :

- L'arbre de jeu : une représentation explicite de toutes les actions possibles de jeux
- Nœud racine : position initiale du jeu
- Nœud niveau 1 : position que le 1er joueur peut atteindre en un déplacement
- Nœud niveau 2 : positions résultant de la réplique du second joueur
- Nœud terminant : représente un nœud gagnant, un nœud perdant
- Chemin racine ⇒ nœud terminale une partie complète de jeu.



Pseudocode :

```
fonction minMax(position, profondeur, joueur_max):
Debut
    si profondeur = 0 ou position est une position finale alors
        retourner une valeur qui indique a quel point la position est favorable ou
    defavorable
    fin si

    si joueur_max faire
        valeur_maximale = -infini
        pour chaque coup dans les coups possibles de la position faire
            effectuer le coup sur la position
            valeur = minMax(position, profondeur-1, faux)
            defaire le coup sur la position
            valeur_maximale = max(valeur_maximale, valeur)
        fin pour
        retourner valeur_maximale

    sinon
        valeur_minimale = +infini
        pour chaque coup dans les coups possibles de la position faire
            effectuer le coup sur la position
            valeur = minMax(position, profondeur-1, vrai)
            defaire le coup sur la position
            valeur_minimale = min(valeur_minimale, valeur)
        fin pour
        retourner valeur_minimale
    fin si
fin
```

3.1.2 Algorithme Minimax avec élagage Alpha-Beta

L'algorithme MinMax est efficace pour les jeux à deux joueurs, mais peut être très coûteux en temps de calculs pour les jeux complexes tels que les échecs.

Une extension courante de l'algorithme MinMax pour l'optimiser est l'algorithme Alpha-Bêta, qui élimine les branches de l'arbre de jeu qui ne peuvent pas conduire à une meilleure solution que celles déjà trouvées. Cette méthode utilise le fait que tous les autres niveaux de l'arbre seront maximisés et que tous les autres niveaux seront minimisés.

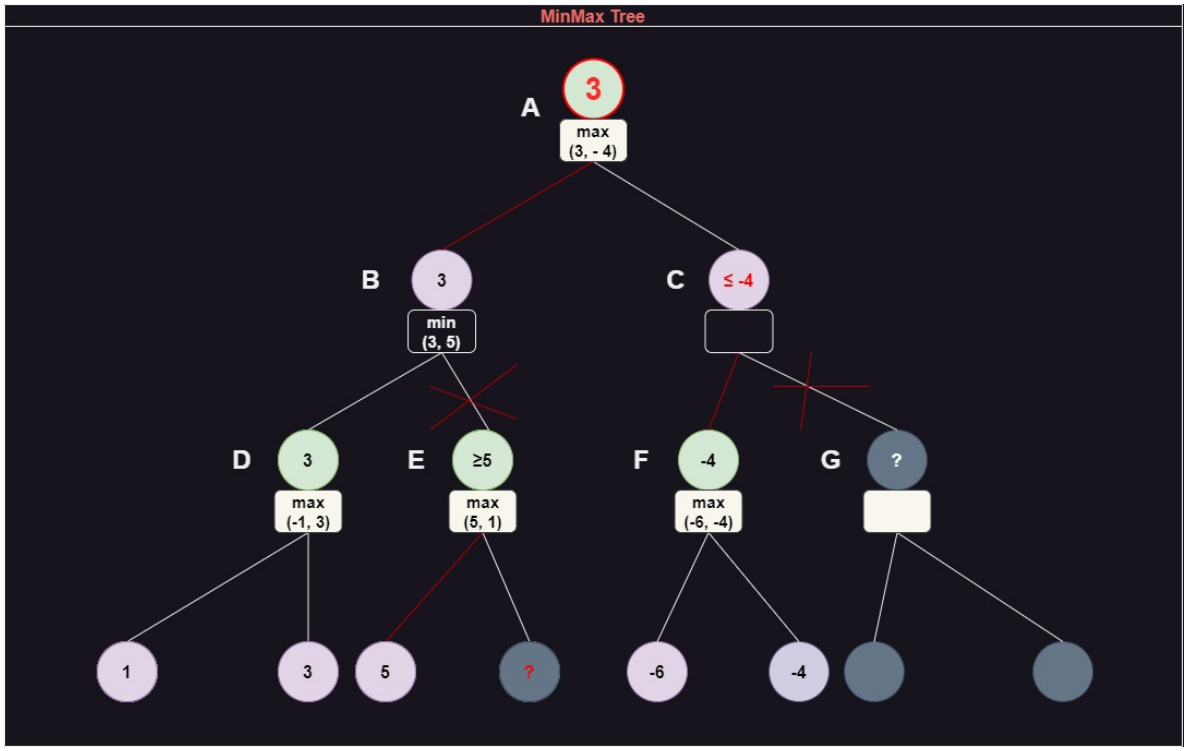
Comme cette méthode réduit le nombre de nœuds évalués par l'algorithme MinMax, cela nous permet de rechercher beaucoup plus rapidement, et même d'accéder à des niveaux plus profonds dans l'arborescence du jeu.

Définissons les paramètres alpha et beta :

Alpha est la meilleure valeur que le maximiseur peut actuellement garantir à ce niveau ou au-dessus. Bêta est la meilleure valeur que le minimiseur peut actuellement garantir à ce niveau ou au-dessus.

La coupure alpha se fait aux niveaux Min. Elle est basée sur l'observation que si la valeur d'un niveau Min est plus petite que la valeur du niveau Max supérieur, quelles que soient les valeurs des nœuds suivants le niveau Min, ils ne changeront pas la valeur du niveau Max supérieur.

Exemple :



Pseudocode :

```
fonction minMaxAlphaBeta(position, profondeur, alpha, beta, joueur_max):
Debut
    si profondeur = 0 ou position est une position finale alors
        retourner la valeur heuristique de la position
    fin si

    si joueur_max alors
        valeur_maximale = -infini
        pour chaque coup dans les coups possibles de la position faire
            effectuer le coup sur la position
            valeur = minMaxAlphaBeta(position, profondeur-1, alpha, beta, faux)
            defaire le coup sur la position
            valeur_maximale = max(valeur_maximale, valeur)
            alpha = max(alpha, valeur_maximale)
            si beta <= alpha :
                break
        fin pour
        retourner valeur_maximale

    sinon
        valeur_minimale = +infini
        pour chaque coup dans les coups possibles de la position faire
            effectuer le coup sur la position
            valeur = minMaxAlphaBeta(position, profondeur-1, alpha, beta, vrai)
            defaire le coup sur la position
            valeur_minimale = min(valeur_minimale, valeur)
            beta = min(beta, valeur_minimale)
            si beta <= alpha :
                break
        fin pour
        retourner valeur_minimale
    fin si

fin
```

Maintenant, que nous avons décrit l'algorithme utilisé dans cet Othello, il est temps de l'implémenter.

3.2 Implémentations de l'IA

```
export class ComputerPlayer {
    constructor(depth) {
        this.depth = depth;
      }
  }
```

La classe ComputerPlayer a plusieurs méthodes qui travaillent ensemble pour déterminer le meilleur mouvement pour l'IA. Le constructeur initialise la profondeur donnée pour l'algorithme Minimax.

```
getGridCopy(board){
        var newBoard = JSON.parse(JSON.stringify(board));
        return newBoard;
      }
```

La méthode getGridCopy crée une copie du plateau de jeu en utilisant JSON.parse et JSON.stringify. Cette méthode est utilisée sur des tableaux multidimensionnels pour créer des copies du plateau de jeu pour que l'IA puisse évaluer les mouvements possibles sans modifier le plateau de jeu d'origine.

La méthode principale qui va permettre à l'IA de renvoyer les positions x et y de son prochain coup est :

```
1
2  function getBestMove(player, board):
3
4  Debut
5      // Copie le tableau de jeu
6      gridCopy = getGridCopy(board);
7
8      // Initialisation des variables
9      bestMove = null;
10     bestScore = -infty;
11     alpha = -infty;
12     beta = infty;
13
14     // Genere tous les coups possibles pour le joueur actuel
15     moves = generateMoves(gridCopy, player);
16
17     // Evalue chaque coup en utilisant l'algorithme Minimax avec elagage Alpha-Beta
18     Pour tout move dans moves faire
19         // Applique le coup a une copie du tableau de jeu
20         newBoard = applyMove(gridCopy, player, move.x, move.y);
21
22         // Evalue le coup en utilisant l'algorithme Minimax avec elagage Alpha-Beta
23         score = minimax(newBoard, player, depth - 1, alpha, beta, false);
24
25         // Met a jour le meilleur coup et le meilleur score
26         Si score > bestScore alors
27             bestMove = move;
28             bestScore = score;
29         fin si
30
31         // Met a jour alpha pour l'elagage Alpha-Beta
32         alpha = max(alpha, bestScore);
33
34         // Elagage si beta <= alpha
35         Si beta <= alpha alors
36             break;
37         fin si
38     fin pour
39
40     // Retourne le meilleur coup
41     return bestMove;
42
43 fin
```

Décrivons pas à pas getBestMove(player, board) ainsi que les méthodes appellées.

- gridCopy est une copie du plateau de jeu, créée avec la méthode getGridCopy(), qui est utilisée pour évaluer les coups possibles sans modifier le plateau de jeu d'origine.

- bestMove est initialisé à null et bestScore à -Infini.

- Les valeurs alpha et bêta sont également initialisées à -Infini et Infini respectivement. Ces valeurs sont utilisées pour l'élagage Alpha-Bêta.

- La méthode generateMoves(gridCopy, player) est utilisée pour générer tous les coups possibles pour le joueur courant, l'IA.

```
1
2  fonction generateMoves(boards, player) :
3
4  // Initialiser un tableau vide pour stocker les mouvements possibles
5  moves = [];
6
7  Debut
8      // Parcourir chaque case du plateau de jeu
9      pour row allant de 0 a 7 faire
10         pour column allant de 0 a 7 faire
11             // Si la case est deja occupee, passer a la suivante
12             si boards[row][column] != 0 alors
13                 continuer;
14             fin si
15
16             // Rechercher dans toutes les directions a partir de la case actuelle
17             pour i allant de -1 a 1 faire
18                 pour j allant de -1 a 1  faire
19                     // Ignorer la case actuelle
20                     si i == 0 et j == a 0 alors
21                         continuer;
22                     fin si
23
24                     x = row + i;
25                     y = column + j;
26                     adversaireTrouve = faux;
27
28                     // Parcourir la ligne dans la direction i,j jusqu'a ce qu'une
29                         piece soit trouvee
30                     tant que x \in {0,7} et y \in {0,7} :
31                         si boards[x][y] == 0 alors
32                             break;
33                         fin si
34
35                         si boards[x][y] == player alors
36                             // Si une piece de joueur est trouvee apres une ou
37                                 plusieurs pieces adverses
38                             // ajouter la case actuelle aux mouvements possibles
39                             si adversaireTrouve == vrai alors
40                                 ajouter {x: row, y: column} a moves;
41                             fin si
42                             break;
43                         fin si
44
45                         adversaireTrouve = vrai
46                         x += i;
47                         y += j;
48                     fn tant que
49                 fin pour
50             fin pour
51         fin pour
52     fin pour
53
54     retourner moves
55
56 fin
```

- Une boucle est utilisée pour évaluer chaque coup possible à l'aide de l'algorithme Minimax avec l'élagage Alpha-Beta.

  Ci-dessous la version implémentée de l'algorithme Minimax avec l'élagage Alpha-Beta :

```
      minimax(board, player, depth, alpha, beta, isMaximizingPlayer) {
        if (depth === 0 || gameOver(board)) {
          return this.evaluateBoard(board, player);
        }

        if (isMaximizingPlayer) {
          // Maximise le score de l'IA
          let maxEval = -Infinity;
          const availableMoves = this.generateMoves(board, player);

          for (let i = 0; i < availableMoves.length; i++) {
            var x = availableMoves[i].x;
            var y = availableMoves[i].y;
            const newBoard = this.applyMove(board, player, x, y);
            const evaluation = this.minimax(newBoard, player, depth - 1, alpha,
    beta, false);
            maxEval = Math.max(maxEval, evaluation);
            alpha = Math.max(alpha, evaluation);

            if (beta <= alpha) {
              break; // Beta cut-off
            }
          }
          return maxEval;
        } else {
          // Minimise le score de l'adversaire
          let minEval = Infinity;
          const opponent = 3 - player; // Assuming the players are labeled as 1
    and 2

          const availableMoves = this.generateMoves(board, opponent);

          for (let i = 0; i < availableMoves.length; i++) {
            var x = availableMoves[i].x;
            var y = availableMoves[i].y;
            const newBoard = this.applyMove(board, opponent, x, y);
            const evaluation = this.minimax(newBoard, player, depth - 1, alpha,
    beta, true);
            minEval = Math.min(minEval, evaluation);
            beta = Math.min(beta, evaluation);

            if (beta <= alpha) {
              break; // Alpha cut-off
            }
          }
          return minEval;
        }
    }
}

```

- Pour chaque coup possible, une copie du plateau de jeu est créée à l'aide de la méthode applyMove(gridCopy, player, move.x, move.y) et le score est évalué à l'aide de la méthode minimax(newBoard, player, this.depth - 1, alpha, beta, false).

```
applyMove(board, player, x, y):
    /*
    Faire une copie du plateau de jeu
    Placer le pion du joueur sur la case specifiee
    Retourner la copie du plateau de jeu avec le coup applique
    */

```

```
Debut
    newBoard = une copie de board;
    newBoard[x][y] = player;

    return newBoard;
fin

```

- Le score est comparé au meilleur score trouvé jusqu'à présent, et si le score est meilleur, bestMove et bestScore sont mis à jour.

- La valeur d'alpha est mise à jour en prenant le maximum de alpha et bestScore.

- Si bêta est inférieur ou égal à alpha, la recherche est écourtée et la boucle s'arrête.

- Une fois tous les coups possibles évalués, la méthode retourne le meilleur coup possible pour le joueur informatique.

3.3 Intégration de l'IA dans le script du jeu

```
var computer = new ComputerPlayer(4)
```

Pour initaialiser l'IA avec une profondeur de 4

```
function turnAI(currentPlayer){

  if (canMove(1) == false && canMove(2) == false){
    gameOver(othelloBoard.gridLogic);
  }

  if (canMove(2) == true && othelloBoard.currentPlayer == 2) {
    var aiMove = computer.getBestMove(currentPlayer, othelloBoard.gridLogic);
    console.log(aiMove)
    console.log(othelloBoard.gridLogic)
    if (aiMove == null && canMove(2)){
      aiMove = computer.getBestMove(currentPlayer, othelloBoard.gridLogic);
      if (aiMove == null) {
        console.log("AI cannot make a move");
        othelloBoard.setCurrentTurn(); // switch to the other player
      }
    }

    othelloBoard.placeDisk(aiMove.x, aiMove.y);
    let affectedDisks = getAffectedDisk(currentPlayer,aiMove.x,aiMove.y);
    flipDisks(affectedDisks);
    setScore(score)

  }

  console.log('turn', othelloBoard.currentPlayer, canMove(othelloBoard.currentPlayer))
  if(canMove(1) == false && othelloBoard.currentPlayer == 1 && canMove(2) == true){
    othelloBoard.setCurrentTurn()
    setTimeout(() => {
        turnAI(othelloBoard.currentPlayer);
      }, 2000);
  }
}
```

On récupère les positions x et y du jeton à placer sur le plateau et la fonction placeDisk(x,y) place le jeton sur le plateau.
La fonction est exécutée dans la fonction onSquareClick(row,column) deux secondes après chaque fin de tour du joueur 1 (les noirs) si le joueur 2 peut insérer un pion.

A la fin du tour du joueur 2, si par contre le joueur 1 ne peut pas jouer mais le joueur 2 peut, on réexecute turnAI() une nouvelle fois.

TurnAI() est une fonction récursive.

3.4 Complexité

Pour les complexités temporelles de chaque fonction dans le fichier ai.js :

- **gameOver(board)** - $O(n^2)$, où n est la longueur d'un côté du plateau (dans ce cas, n=8).
- **ComputerPlayer.getGridCopy(board)** - $O(n^2)$,où n est la longueur d'un côté du plateau (dans ce cas, n=8).
- **ComputerPlayer.getBestMove(player, board)** - $O(b^m)$, où b est le facteur de branchement moyen (nombre de coups disponibles) par niveau (profondeur) et m est la profondeur maximale de l'arbre de recherche. Dans cette implémentation, la profondeur maximale est déterminée par le paramètre de profondeur passé au constructeur ComputerPlayer, donc la complexité temporelle de getBestMove dépend de la valeur de ce paramètre.
- **ComputerPlayer.generateMoves(board, player)** - $O(n^2 * b)$, où n est la longueur d'un côté du plateau et b est le facteur de branchement moyen par niveau. Dans cette implémentation de l'Othello, notre facteur de branchement maximal est de 8 (puisque le joueur peut placer une pièce dans l'une des 8 directions autour d'un espace vide), donc la complexité temporelle peut être simplifiée en $O(n^2)$.
- **ComputerPlayer.applyMove(board, player, x, y)** - $O(1)$, car elle modifie simplement un élément unique dans la matrice 8x8.
- **ComputerPlayer.evaluateBoard(board, player)** - $O(n^2)$, où n est la longueur d'un côté du plateau (dans ce cas, n=8).
- **ComputerPlayer.minimax(board, player, depth, alpha, beta, isMaximizingPlayer)** - $O(b^m)$, où b est le facteur de branchement moyen par niveau et m est la profondeur maximale de l'arbre de recherche. La complexité temporelle de minimax dépend de la valeur du paramètre de profondeur qui lui est passé, ainsi que du facteur de branchement de l'arbre de jeu (qui peut être affecté par l'état actuel du plateau). Le facteur de branchement d'un arbre de jeu pour Othello est le nombre moyen de coups possibles à chaque niveau du jeu. Au début du jeu, il y a 4 coups possibles pour chaque joueur, donc le facteur de branchement est de 4. Au fur et à mesure que le jeu progresse et que le nombre de cases vides diminue, le nombre de coups possibles diminue également, ce qui réduit le facteur de branchement.

Majoritairement, on obtient une complexité polynomiale, car elle dépend d'une puissance de n, où n est la taille du plateau de jeu (n^2 dans ce cas).

Cette complexité est considérée comme relativement efficace, et a été améliorée en utilisant des algorithmes plus sophistiqués tels que l'élagage alpha-bêta pour réduire le nombre de nœuds évalués dans l'arbre de recherche. Cela réduit considérablement le temps de calcul nécessaire pour trouver la meilleure décision à chaque tour.

4 Conclusion

Le processus de développement du projet a été divisé en plusieurs étapes, allant de la conception initiale du jeu à l'implémentation de l'algorithme IA . La mise en œuvre de l'algorithme MinMax amélioré avec l'élagage Alpha-Bêta a été un défi important, car il a fallu comprendre le fonctionnement de l'algorithme et le modifier pour qu'il s'adapte à notre jeu, mais nous avons réussi à l'implémenter.

Le projet a également été une occasion pour nous d'explorer les possibilités offertes par la bibliothèque Tree.js pour créer des graphismes interactifs pour notre jeu. Bien que nous ayons rencontré quelques difficultés avec par exemple, la rotation des jetons, l'utilisation de la 3D pour représenter le plateau de jeu a donné un aspect visuel original et attractif pour les joueurs.

Au cours de ce projet, nous aurions pu inclure certaines options dans notre jeu, comme par exemple l'ajout d'un bouton pour reset la partie, ou bien améliorer notre code en partageant les tâches avec du multiprocessing, idée que nous avions abandonné étant donné que c'était la première fois que nous codions en JavaScript et que nous devions déjà nous familiariser avec la bibliothèque Three.js.

En outre, nous avions aussi créer deux fonctions changeCellColor et ResetBoardColor fonctionnels que nous n'avons pas su intégrer à notre code dû à la complexité de Three.js. Ces deux fonctions devaient nous permettre de rendre le jeu plus user-friendly en permettant au joueur de mieux visualiser ses possibilités de jeu en colorant les cases où il pouvait placer son pion.

Dans l'ensemble, le projet a été une expérience éducative et amusante, qui nous a permis de développer des compétences en programmation js, en IA, en algorithmique et en conception de jeux. Les objectifs qu'on s'était donnés ont été atteints et le résultat final est un jeu d'Othello fonctionnel et divertissant, qui peut être joué contre une IA grâce à l'utilisation de l'algorithme MinMax amélioré avec l'élagage Alpha-Bêta.

5 Sitography

Jeux
Minimax pseudocode
Alpha-beta pseudocode

5.1 Youtube

4.Theorie des jeux — Minimax — Elagage Alpha-Beta
Algorithms Explained – minimax and alpha-beta pruning

5.2 Wikipédia

Algorithme minimax
Élagage alpha-bêta