**Chapitre I : Minimax**

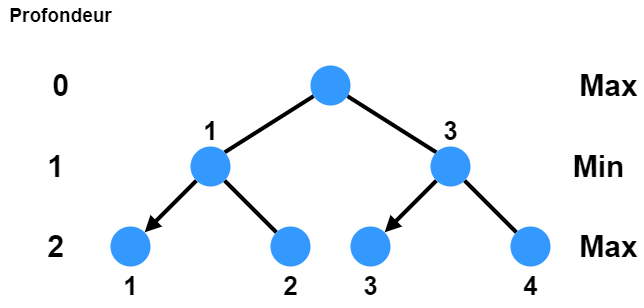
Disons que l’on joue à un jeu avec une autre personne. Et la métrique de "*votre performance dans ce jeu*" est mesurée par un score numérique qui augmente lorsque vous vous rapprochez de votre "gagner le jeu" et diminue lorsque votre antagoniste se rapproche de son "*gagner le jeu*". Dans ce cadre, vous pouvez penser que vous essayez de maximiser le score et que votre adversaire essaie de le minimiser.

Nous pouvons trouver un algorithme capable de prendre de bonnes décisions et de gagner un tel jeu en modélisant la situation ci-dessus de la manière suivante : nous aurons 2 entités (fonctions) qui s'appellent ; l'un essaie de maximiser le score, l'autre de le minimiser. En gros, ces 2 fonctions imiteront les deux joueurs.

Cet algorithme est également un bon exemple d'intelligence artificielle. L'algorithme minimax est un tel algorithme qui fait que les ordinateurs se comportent intelligemment mais ils n'apprennent rien. Et malgré cela, cela fonctionne assez bien dans de nombreux jeux.

**Algorithme**

Si nous pensons à un jeu en termes de ces 2 joueurs, Max & Min, changeant de tour l'un avec l'autre, alors nous pouvons représenter le jeu comme un arbre de décisions. Voici un exemple assez simple :



Chaque nœud de cet arbre (à l'exception des nœuds terminaux) représente une décision qui doit être prise à ce moment du jeu. Nous décidons quel mouvement prendre. Dans cet exemple, nous devons choisir parmi seulement 2 mouvements, mais en général, nous pouvons avoir n'importe quel nombre de mouvements, et ce nombre peut varier d'un nœud à l'autre, en fonction de l'état du jeu.

Le nœud supérieur (celui à la profondeur 0) est l'état actuel du jeu. C'est ici que nous devons décider du prochain mouvement du jeu. Ici, nous commençons avec Max, car c'est le joueur qui veut ce que nous faisons : maximiser le score. Au lieu de décider en se basant uniquement sur les prochains états possibles du jeu que Max peut atteindre à partir de ce point, il pense : « Après avoir fait l'un de ces mouvements, que fera mon ennemi Min ?»

Alors, il appelle Min et dit : "Quel coup ferez-vous si je choisis à gauche ? ", Et après cela : "Quel coup ferez-vous si je choisis bien ? ".

Une fois que Max a trouvé ce que Min ferait, il choisit la branche qui lui donnera le score maximum.

Mais attendez. Comment Min décidera-t-il quoi faire pour minimiser le score ? Il appliquera la même stratégie que Max. Min à son tour appellera Max pour lui demander ce qu'il fera pour chaque choix possible de Min. Mais après cela, au lieu de faire le choix qui maximise le score, Min fera le contraire : fera le choix qui minimisera le score.

Et ainsi de suite… Chacun appellera l'autre, construisant un grand arbre de façon récursive comme ça jusqu'à ce qu'ils atteignent un état terminal.

Un état terminal doit être, de préférence, un état dans lequel le jeu est terminé. Mais c'est généralement trop coûteux en calcul ; laisser l'algorithme explorer tous les mouvements possibles jusqu'à la fin du jeu peut prendre un temps extrêmement long. Donc, nous définissons une profondeur maximale. Un état de jeu sera considéré comme terminal lorsque le jeu est terminé ou que la profondeur maximale est atteinte.

Dans l'exemple ci-dessus, les états terminaux sont ceux du bas (profondeur 2).

Que se passera-t-il sur les nœuds terminaux ?

Le joueur qui atteindra les états terminaux, que ce soit Max ou Min, ne pourra pas appliquer la même stratégie utilisée jusqu'à présent ; la stratégie d'appeler l'autre joueur à l'aide.

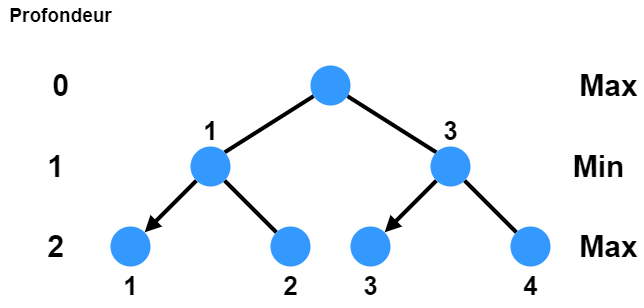
Maintenant, dans les nœuds terminaux, nous devons calculer le score du jeu dans chacun de ces états terminaux.

Cela n'est peut-être pas si évident dans certains jeux, mais nous devons au moins estimer le score en fonction de cet état du jeu.

Dans notre exemple ci-dessus, les scores des états terminaux sont les nombres du bas : 1, 2, 3 et 4.

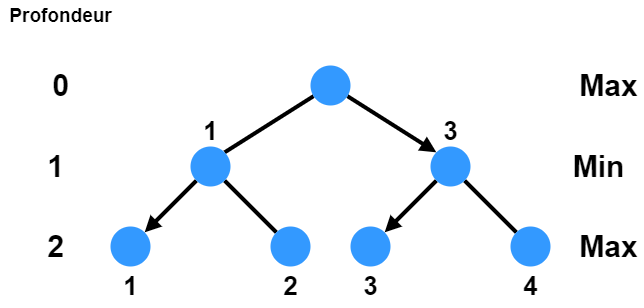
La seule chose que le joueur Max fera à ce dernier niveau est de renvoyer ces scores au Min appelant au niveau précédent.

Après cela, notre arbre ressemblera à :



Les choix que Min fera sont indiqués par une flèche. Les scores sont propagés du niveau 2 au niveau 1 (niveau Min).

Maintenant, une fois que nous savons ce qui va se passer à la profondeur 1, nous laissons Max faire son mouvement au niveau 0 :

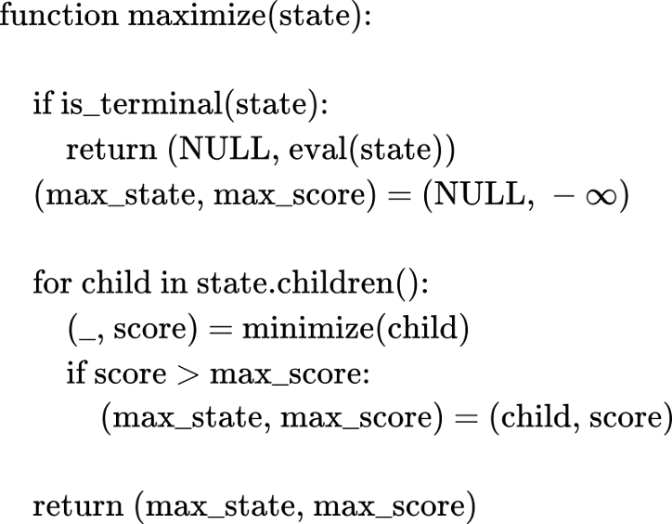


Cela signifie que le mouvement optimal à faire à l'état actuel du jeu est celui représenté par le bord droit de ce nœud supérieur, et le score que nous obtiendrons est d'au moins 3.

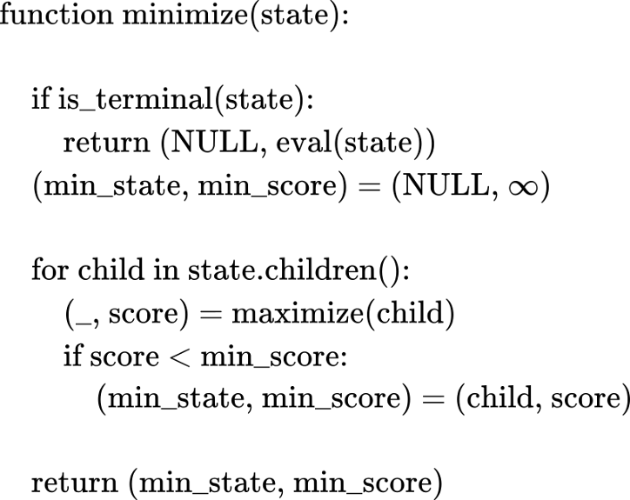
On dit «au moins » parce que cet algorithme fonctionne comme si Max et Min choisissaient toujours leur meilleur coup. Mais si, en réalité, notre ennemi est pire que Min, nous pouvons obtenir un meilleur score.

Alors maintenant, comment pouvons-nous exprimer clairement cet algorithme dans le code ? L'arbre ci-dessus était juste pour le décrire intuitivement. Un ordinateur n'a pas besoin de construire explicitement un tel arbre. Tout ce dont nous avons besoin, ce sont 2 fonctions : maximiser et minimiser qui s'appelleront.

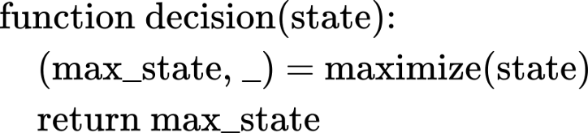
Voici un aperçu de l'algorithme décrit ci-dessus :



La fonction maximize()retourne un **tuple** qui contient à sa première position l'état enfant du jeu qui maximise le score, et à la seconde position une estimation du score qui sera atteint en suivant cet état. Une quantité analogique est renvoyée par la minimize() fonction.



La decision()fonction ci-dessous prend en entrée l'état actuel du jeu et renvoie l'état à suivre si nous voulons maximiser notre score.

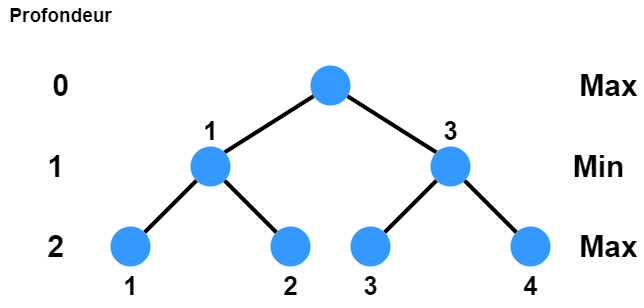


Ce qui eval() précède est une fonction qui estime le score de son état d'entrée du jeu.

**Minimax avec taille α – β**

Plus la profondeur est grande, meilleur sera notre joueur. Mais cet algorithme peut prendre un certain temps si nous lui définissons une grande profondeur. Nous devons donc choisir la profondeur maximale possible qui correspond toujours à nos exigences de temps. Pouvons-nous rendre cet algorithme un peu plus rapide, de sorte que nous puissions utiliser une valeur plus grande pour la profondeur, tout en obéissant à la contrainte de temps ?

Voici un exemple :



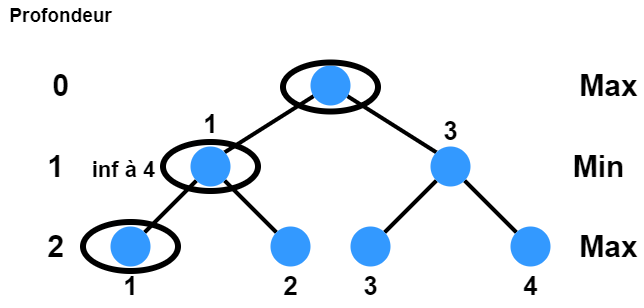
Pour décider du déplacement du nœud supérieur, devons-nous évaluer les nœuds de l'arborescence ?

L'élagage **α - β** est une stratégie que nous pouvons utiliser pour améliorer l'algorithme Minimax en ignorant certaines branches de l'arbre dont nous savons à l'avance qu'elles ne nous aideront pas à prendre la décision optimale. Le nom élagage **α - β** vient des 2 paramètres utilisés dans cet algorithme qui sont α et **β**.

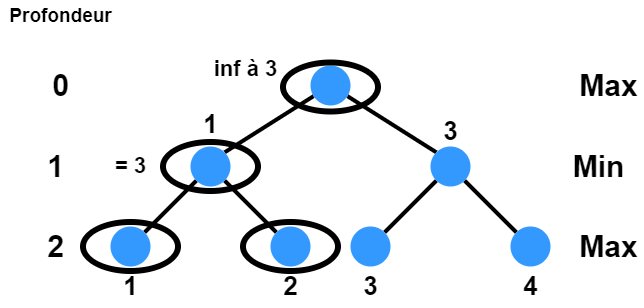
Maintenant, comment fonctionne cette méthode ?

Chaque fois que nous établissons le score d'un nœud, nous le propageons également vers le haut dans l'arborescence jusqu'à son parent et l'utilisons pour définir une limite inférieure ou supérieure du résultat du nœud parent.

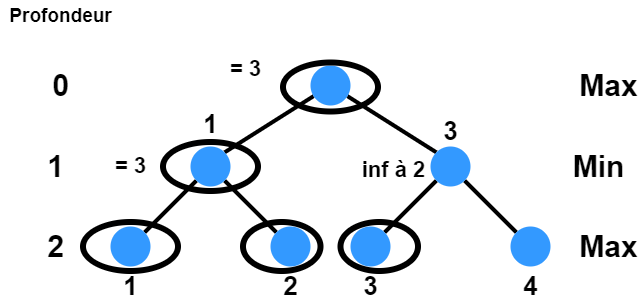
Par exemple, dans notre arbre, si nous commençons à évaluer les nœuds de gauche à droite après avoir établi le score du nœud terminal le plus à gauche (qui est 4), nous savons que le résultat pour le nœud parent ne peut pas être supérieur à 4. C'est parce que le parent est un nœud Min, et si nous en avons déjà 4, le nœud parent ne peut pas choisir quelque chose de plus grand que cela.



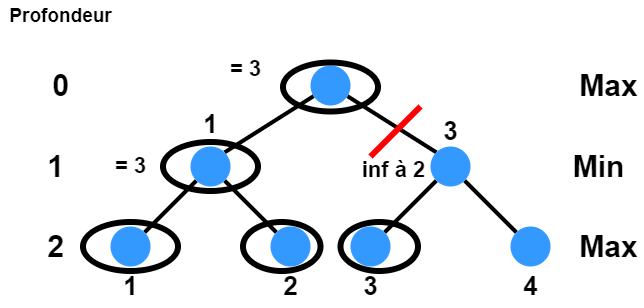
Mais, à ce stade, cette information n'aide pas trop. Continuons et voyons ce qui se passe. Nous évaluons le nœud terminal suivant et constatons que la valeur du parent est 3 et que le nœud supérieur ne peut pas être inférieur à 3.



Si nous continuons à évaluer les nœuds, nous découvrons le terminal avec un score de 2, ce qui signifie que son parent doit être ≤ 2.



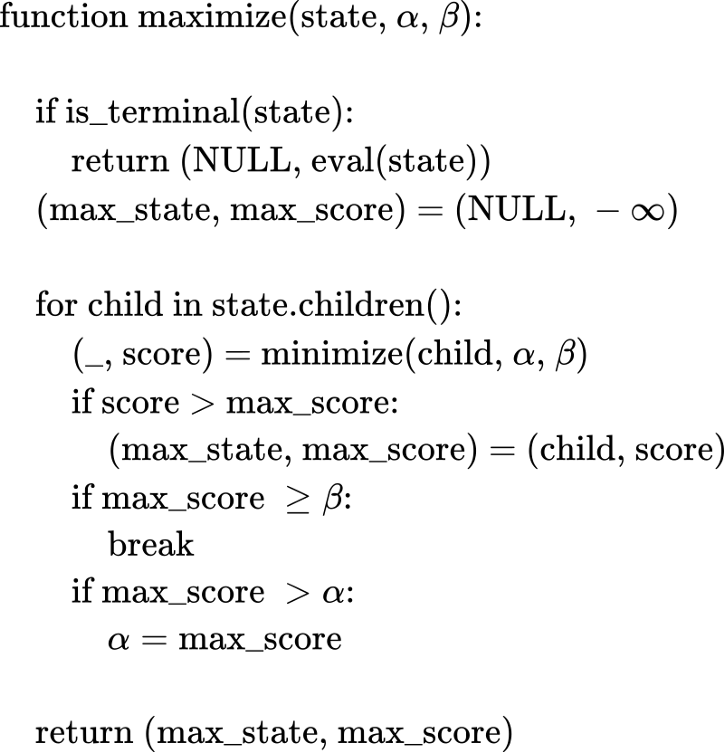
Maintenant, devons-nous également évaluer le dernier nœud ? Non. Nous savons déjà que le nœud supérieur peut obtenir au moins un 3 s'il suit la branche gauche. Et nous savons qu'à partir de la bonne branche, nous ne pouvons pas obtenir plus de 2. Ainsi, nous pouvons simplement ignorer la bonne branche à partir de maintenant.



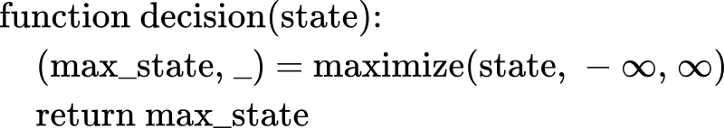
Dans ce petit exemple, cela peut ne pas sembler une grande amélioration car nous avons sauté un seul nœud terminal. Mais dans un arbre plus grand, un tel nœud peut contenir un sous-arbre assez grand. Et notez qu'ici nous avons ignoré un seul nœud frère car c'était le seul qui restait dans cette branche. S'il y avait plus de nœuds restants, nous les ignorerions tous. Par exemple, si la branche de droite avait 2, 1 et 0 comme nœuds terminaux, alors après avoir évalué le 2, nous pourrons sauter à la fois 1 et 0.

De plus, le nombre de nœuds ignorés dépendra de l'ordre des nœuds terminaux. Vous avez probablement remarqué que pour cet exemple, j'ai inversé les scores de 1, 2, 3, 4 à 4, 3, 2, 1. Ce n'était pas au hasard. Lorsque les scores sont dans l'ordre croissant, c'est le pire des cas dans lequel aucun élagage ne se produit. Mais il y a peu de chances que ce mauvais ordre se produise dans la pratique. En moyenne, la taille α - β permet à l'algorithme minimax d'aller presque deux fois plus profondément dans le même laps de temps par rapport à l'absence d'élagage.

Vous trouverez ci-dessous le pseudocode de l'algorithme décrit ci-dessus.



L'infini - / + dans la fonction de décision (premier appel à maximiser) signifie que nous commençons l'algorithme sans restriction sur ce que le score résultant peut-être.



Ainsi, l'algorithme minimax est un algorithme relativement simple qui fonctionne bien sur des jeux simples (faible facteur de branchement

**Chapitre II : Morpion**

Pour développer un jeu comme le morpion, on pourrait être tenté d’adopter une approche itérative. En effet, on peut partir du principe que les règles sont assez simples à coder à première vue, dans la mesure où l’espace est limité en taille, et parce que la seule règle est d’aligner trois croix ou ronds soit en diagonale, soit à la verticale ou horizontale. Par ailleurs, il s’agit d’un jeu « **fini** » où les combinaisons pour gagner sont déjà connues. Pourtant et en dépit de sa relative simplicité, ce jeu peut offrir plusieurs dizaines voire centaines de configurations différentes. Le fait de coder ce jeu de façon assez traditionnelle (soit avec des règles explicites) va imposer d’énumérer les nombreux cas de figure possibles.  On va devoir par exemple signaler les cas où deux éléments sont alignés, si deux éléments sont séparés uniquement par un espace vide à combler, prévoir des stratégies pour obtenir un match nul ou pour gagner etc… Cela peut être extrêmement fastidieux à coder.

L’approche qui consiste à utiliser une intelligence artificielle va nous permettre au contraire de développer un programme beaucoup plus simple en fournissant à notre programme une certaine autonomie. L’objectif ici n’est donc plus de lui indiquer l’ensemble des règles et des exceptions à suivre, mais de disposer d’exemples (ou d’un semblant de logique) à partir desquels notre programme va pouvoir prendre des décisions avec une relative autonomie. En nous basant sur ce que nous savons du machine learning, nous pourrions alors opter dans un premier temps pour une approche basée sur l’apprentissage (ou sur l’analyse) d’exemples et de contre exemples. On pourrait décider ainsi de fournir de nombreux exemples de situation à notre programme pour lui permettre de savoir quand jouer tel ou tel coup. On pourrait également combiner avec cette approche des méthodes issues de l’apprentissage par renforcement.

Cette méthode présente pourtant un inconvénient : c’est qu’il va falloir fournir un très grand nombre d’exemples à notre programme pour qu’il apprenne à reconnaître les meilleurs coups à jouer. Cette méthode n’est donc pas très économique. Par ailleurs elle ne garantit pas une prise de décision optimale. Notre algorithme va simplement chercher des similitudes sans pour autant comprendre ce qu’il fait, si l’on peut s’exprimer ainsi. Nous pouvons alors nous tourner vers d’autres méthodes qui consistent non pas à fournir des dizaines ou des centaines d’exemples à un programme, mais à lui permettre au contraire de se projeter dans les états futurs d’une configuration donnée pour prendre la meilleure décision possible. C’est notamment ce que permet l’algorithme [**Minimax**](https://en.wikipedia.org/wiki/Minimax) qui est très largement appliqué dans l’univers des jeux comme le morpion. Si on reprend l’exemple du morpion, l’algorithme va en quelque sorte déplier l’ensemble des coups possibles à la suite d’un mouvement jusqu’à atteindre un stade dit « terminal » (victoire, défaite ou match nul). Ces différents états sont évalués de façon négative ou positive par l’algorithme, un peu comme sous la forme d’une récompense, d’une punition ou encore d’une action neutre. L’algorithme sera alors amené à chaque fois à prendre la décision qui implique la plus forte récompense. Prenons un exemple simple avec cette première position au morpion :

X - -

X - O

O O X

Imaginons maintenant que ce soit à l’ordinateur de jouer pour les croix. La première étape de l’algorithme va alors consister à simuler les différents coups possibles. Ce qui nous donne l’ensemble des possibilités suivantes :

X X - X - X X - -

X - O X - O X X O

O O X O O X O O X

Les deux premières situations à gauche sont neutres, mais la troisième situation est par contre positive. On va alors stocker en mémoire le score des différentes situations. Notre algorithme va alors jouer à coup sûr le troisième coup. Le but de l’algorithme n’est pas seulement d’observer ce qui va se passer après un coup, mais de penser l’ensemble des coups possibles qui peuvent alors l’amener à une victoire ou une défaite par exemple. Ce travail va de pair avec ce que l’on appelle d’une part la maximisation et minimisation. Ce principe veut tout simplement dire qu’un joueur chercher à maximiser ses gains (donc jouer les coups qui lui permettent de se rapprocher d’une victoire), tandis que l’autre va chercher à minorer au maximum les gains possibles pour l’autre joueur (donc jouer des coups qui amèneront vers un match nul ou une victoire). Si vous avez par exemple un joueur A et joueur B, A cherchera à maximiser ses gains tandis que B sera à la recherche des coups pouvant limiter les gains du joueur A. Maintenant que nous savons comment fonctionne l’algorithme du Minimax, nous allons pouvoir passer à l’étape de programmation.

**Mise en place Minimax**

Nous allons donc maintenant mettre en place l’algorithme du Minimax pour développer un jeu de morpion doté d’une certaine capacité à prendre des décisions. Pour commencer, nous allons tout d’abord déclarer une variable « Grid » qui sera un tableau représentant notre grille de morpion :

let Board = [

0, 1, 2,

3, 4, 5,

6, 7, 8

]

On déclare ensuite une variable « AIMove » dans laquelle nous allons stocker les mouvements de l’ordinateur :

let AIMove

Le jeu de morpion possède l’avantage d’être assez simple pour qu’on considère qu’il s’agit d’une victoire : il faut simplement que trois croix ou ronds soit alignés à la verticale, horizontale ou en diagonale. On va donc commencer par coder cette fonction qui a pour but de vérifier si nous avons à faire ou non à cette configuration. Commençons par déclarer une fonction « Win » :

function Win (Grid) {

On peut alors créer à l’intérieur de cette fonction une variable qui va contenir l’ensemble des alignements possibles. Nous allons déclarer une variable « goal » qui va alors contenir le tableau suivant :

let goal = [

[0,1,2],

[3,4,5],

[6,7,8],

[0,3,6],

[1,4,7],

[2,5,8],

[0,4,8],

[6,4,2]

]

Il suffit ensuite de créer une boucle pour itérer à travers l’ensemble de ces propositions pour vérifier le contenu de ces cases. Si ces cases contiennent à chaque fois une croix ou un rond, on sait qu’il s’agit d’une victoire. On commence donc d’abord par créer une boucle pour itérer à travers l’ensemble des situations possibles :

for (let i = 0; i < WinSituation.length; i++) {

On récupère ensuite un ensemble à tester que l’on va conserver dans une variable « TestRow ». Il faut également stocker les valeurs contenues dans cet ensemble. On va alors créer une valeur « RowValues »

let TestRow = WinSituation[i]

let RowValues = []

On met alors en place une deuxième boucle pour itérer à travers cet ensemble. On va alors pousser l’ensemble de ces éléments dans la variable « RowValues ». Une fois que cela est fait, nous allons mettre en place les deux fonctions pour savoir si toutes les cases sont occupées par des croix ou par des ronds. En fonction du résultat nous pouvons alors retourner la valeur correspondante (« 1 » si les croix gagnent, « -1 » pour les ronds) :

for (let j = 0; j < TestRow.length; j++) {

RowValues.push(Grid[TestRow[j]])

}

let CrossesWin = function (Value) {

return Value === 'X'

}

let NoughtsWin = function (Value) {

return Value === 'O'

}

if ( RowValues.every(CrossesWin) ) {

return 1

} else if ( RowValues.every(NoughtsWin) ) {

return -1

}

}

}

Maintenant que nous disposons de notre fonction pour vérifier qu’il y a une victoire ou non, nous allons pouvoir commencer à coder notre fonction qui va nous servir à mettre en place l’algorithme Minimax. Pour ce faire, commençons par déclarer la fonction « Minimax » :

function Minimax (NewGrid, Player) {

La première chose à faire est de vérifier que des cases sont encore libres pour jouer. On déclare donc une variable « Available » qui nous retourner toutes les valeurs du jeu qui sont égales à des nombres :

let Available = NewGrid.filter( function(a) {

return typeof a === 'number'

})

Nous allons ensuite avoir besoin de savoir quelle est la situation du jeu. Pour ce faire nous allons vouloir faire appel à notre fonction « Win » sous la forme de conditions, et nous retournons la valeur correspondante. Si toutes les cases sont occupées (soit si la variable « Available » ne contient aucun élément) nous retournons une valeur de 0 :

if ( Win(NewGrid) === 1 ) {

return 1

} else if ( Win(NewGrid) === -1 ) {

return -1

} else if (Available.length === 0) {

return 0

}

Une fois que cela est fait, nous allons pouvons passer à l’étape qui consiste à simuler les mouvements possibles. Nous allons devoir consigner ces mouvements ainsi que les scores correspondants. Nous déclarons les variables « Moves » et « Scores » :

let Moves = []

let Scores = []

Nous allons ensuite pouvoir itérer à travers les cases disponibles. Nous mettons donc en place une boucle pour itérer à travers la variable « Available » :

for (var i = 0; i < Available.length; i++) {

Nous déclarons à la suite une variable « Move » dans laquelle nous allons stocker le coup correspondant car nous aurons besoin plus tard :

let Move

On va donc ensuite passer dans cette variable « Move » la position du coup disponible de cette façon :

Move = NewGrid[Available[i]]

Il faut ensuite modifier le contenu de la grille en y inscrivant ensuite soit une croix ou un rond de la façon suivante :

NewGrid[Available[i]] = Player

On fait alors appel à la fonction « Minimax » avec le joueur adverse. Si ce sont les croix qui sont en train de jouer, nous faisons appel aux ronds. Et vice versa. Nous poussons ensuite les résultats obtenus dans la variable « Scores » :

if (Player === 'X') {

Scores.push(Minimax(NewGrid, 'O'))

} else {

Scores.push(Minimax(NewGrid, 'X'))

}

Il faut ensuite rétablir l’état de la grille. On remet donc en état la grille en remplaçant le contenu par la variable « Move » (qui contient le numéro du coup). Puis on passe ensuite ce coup dans la variable « Moves », puis on ferme la boucle :

NewGrid[Available[i]] = Move

Moves.push(Move)

}

Nous avons ensuite besoin de faire remonter le meilleur coup (ou le coup qui permet de minimiser les gains de l’adverse). Nous allons donc ici partir du principe que ceux sont les croix qui souhaitent maximiser leur score. Nous allons donc faire remonter le meilleur score en recherchant le maximum dans la variable « Scores » et en le stockant dans la variable « HighScore ». Nous recherchons ensuite la position de ce meilleur score et nous la stockons dans la variable « HighScoreIndex ». Nous assignons à la variable « AIMove » le numéro du mouvement qui correspond au score le plus élevé. Et enfin nous retournons le score le plus élevé :

if (Player === 'X') {

var HighScore = Math.max(...Scores)

let HighScoreIndex = Scores.indexOf(HighScore)

AIMove = Moves[HighScoreIndex]

return Scores[HighScoreIndex]

Puis on fait la même chose si c’est l’autre joueur qui joue. Il s’agit ici au contraire de réaliser une minimisation. Nous récupérons le score le plus bas que nous stockons dans la variable « LowScore ». Nous récupérons la position de ce score et nous la stockons dans une variable « LowScoreIndex ». Puis nous assignons à la variable « AIMove » le mouvement correspondant à la position du score le plus bas. Et enfin nous retournons ce score :

} else {

let LowScore = Math.min(...Scores)

let LowScoreIndex = Scores.indexOf(LowScore)

AIMove = Moves[LowScoreIndex]

return Scores[LowScoreIndex]

}

}

Maintenant que notre fonction « Minimax » et « Win » sont codées nous allons pouvoir faire appel à ce programme. L’ordinateur va jouer contre lui-même. L’initiative appartiendra aux croix. Nous ajoutons alors le code suivant :

let Play = 1

let Turn = 1

let AI

while (Play !== 0) {

let Available = Grid.filter(function (a) {

return typeof a === 'number'

})

if ( (Win(Grid) === 1) ||

(Win(Grid) === -1) ||

(Available.length === 0) ) {

Play = 0

} else {

if (Turn === 1) {

AI = 'X'

Turn = 0

} else {

AI = 'O'

Turn = 1

}

let Choice = Minimax(Grid, AI)

Grid[AIMove] = AI

console.log('\n console.log(Grid[0],Grid[1],Grid[2])

console.log(Grid[3],Grid[4],Grid[5])

console.log(Grid[6],Grid[7],Grid[8])

}

}

Si vous lancez ensuite le programme, vous pourrez alors obtenir la partie suivante. Les mouvements de l’ordinateur sont mis en valeur en rouge :

**X 1 2**

**3 4 5 Tour des croix**

**6 7 8**

**X 1 2**

**3 O 5 Tour des ronds**

**6 7 8**

**X X 2**

**3 O 5 Tour des croix**

**6 7 8**

**X X O**

**3 O 5 Tour des ronds**

**6 7 8**

**X X O**

**3 O 5 Tour des croix**

**X 7 8**

**X X O**

**O O 5 Tour des ronds**

**X 7 8**

**X X O**

**O O X Tour des croix**

**X 7 8**

**X X O**

**O O X Tour des ronds**

**X O 8**

**X X O**

**O O X Tour des croix | Match nul**

**X O X**

Vous venez donc d’apprendre à créer un programme capable de jouer au morpion en utilisant une technique d’optimisation par maximisation et minimisation qui consiste à rechercher de façon exhaustive l’ensemble des positions possibles pour prendre la meilleure décision. Vous noterez que pour toutes configuration, les matchs machine contre machine aboutissent continuellement à un match nul.

**Chapitre III : Puissance 4**

Le jeu du puissance 4 est extrêmement connu et très simple. Deux joueurs jouent à tour de rôle en faisant tomber des pions dans une grille. Leur but est d’aligner verticalement, horizontalement ou en diagonale quatre pions de leur couleur, avant que l’adversaire ne fasse de même. Le but de ce projet a été de programmer en **JavaScript** une intelligence artificielle jouant au puissance 4. Dans une première partie, une IA jouant aléatoirement a permis d’implémenter les fonctions de jeu "basiques". Dans un second temps, une fonction de score a été créée pour évaluer la valeur d’un coup. En utilisant cette fonction et l’algorithme **minimax**, une IA a pu être programmée. Enfin, quelques modifications ont pu la rendre meilleure.

**1. Partie aléatoire**

Dans cette version du programme, l’IA se contente de jouer aléatoirement dans une des colonnes. Pour permettre ça, il a fallu implémenter les fonctions de décodage/encodage de la grille et de "jeu dans une colonne". Une structure ***game*** contient le jeu ainsi que le tour du joueur en cours (c’est donc la représentation du fichier jeu en mémoire).

**1.1 decode(), to\_file()**

Ces deux fonctions ont comme rôle de charger la grille d’un fichier texte (decode()) et de la réécrire à partir de ce qui est en mémoire du programme (to\_file()). Pour la lecture, on récupère le tour du joueur ayant joué en dernier, en première ligne (et on l’inverse). On parcourt ensuite la grille et on convertit tout sous forme d’entiers (0 étant une case vide, 1 un X et 2 un O). Pour l’écriture, c’est exactement le même principe, on écrit d’abord le joueur ayant joué, puis on reconvertit toute la grille en \_, X et O qu’on écrit dans le fichier.

**1.2 play()**

La fonction play() joue un coup dans la colonne donnée. Elle retourne 0 si tout s’est bien passé, -2 si le coup est hors de la grille et -1 si la colonne est pleine. Pour jouer ce coup, elle regarde juste quelle est la première ligne vide dans la colonne donnée, elle modifie ensuite le jeu directement.

**2 Fonction de score**

Dans cette seconde partie, l’IA choisit la colonne maximisant la fonction eval(), donnant le score d’un coup.

**2.1 Fonction eval()**

Dans cette première version de la fonction, elle évalue le coup même, et pas la valeur de la grille. Le but de faire une fonction aussi simple est qu’elle ne nécessite pas de parcourir toute la grille et est plus rapide à exécuter (notamment pour la construction de l’arbre ensuite). 2 Elle prend en argument le jeu, et la case précise ou le jeton serait potentiellement posé. De là, elle compte le nombre de pions alignés autour de cette position donnée. Et c’est grâce à ça qu’elle ne parcourt pas tout le jeu : elle ne regarde que les cases autour de la position donnée. Elle retourne le nombre de pions alignés par le coup ajouté au nombre qu’aurait le joueur adverse jouant au même endroit (c’est une sorte de valeur de "blocage"). Afin de rendre plus important un grand nombre de jetons alignés, on renvoie la somme des puissances pour induire un non-linéarité.

**3 Algorithme minimax**

On ajoute maintenant l’algorithme minimax. Cet algorithme consiste, à chaque étape, à regarder tous les coups possibles de chaque joueur, et à considérer soit celui qui maximise le score (si c’est au joueur de jouer) soit qui le minimise (si c’est à l’adversaire). On crée donc une structure d’arbre, qui contient la valeur d’un coup ainsi que la colonne à jouer pour obtenir cette valeur. En construisant récursivement l’arbre à l’aide de l’algorithme minimax, le coup à jouer sur la racine sera le meilleur coup à jouer pour l’IA. Pour construire ces arbres, il faut pouvoir les évaluer. Avec la première fonction d’évaluation, on ajoutait les valeurs des coups successifs pour avoir une idée de l’évaluation d’une suite de coup. On verra plus bas que suite au changement de fonction d’évaluation, l’algorithme a également un peu changé.

**4 Raffinements**

Suite à la première version, de nombreuses petites choses ont été ajoutées et modifiées pour améliorer l’intelligence de cette IA.

**4.1 Fonction de score**

Le problème de l’ancienne fonction d’évaluation est qu’elle ne prenait en compte que les pions autour du coup joué, pas la globalité de la grille. Elle pouvait donc ne pas voir un "ancien" alignement de trois pions un peu plus loin que le coup considéré à ce moment. La fonction d’évaluation a donc été modifiée, afin d’évaluer non pas la valeur d’un coup donné, mais celle de la grille en entier. Pour cela, on compte tous les alignements de chaque joueur, et on les additionne, toujours avec une fonction puissance pour renforcer l’importance des nombreux pions alignés. Les pions alignés par je joueur comptent positivement, et ceux par l’adversaire négativement. Pour la construction de l’arbre **minimax**, on ne calcule donc plus que la fonction sur les feuilles, et on choisit pour les feuilles le max ou le min de ses fils en fonction du joueur considéré. Cela ressemble donc plus à l’algorithme **minimax** "normal". A été également ajouté un cas particulier pour forcer l’IA à jouer en cas d’alignement potentiel de 4 pions.

**4.2 Construction de l’arbre**

Un problème se posait en cas d’égalité de score entre deux coups, l’IA choisissait juste la première colonne. Cela posait problème notamment lorsque l’IA était certaine d’avoir gagné, et que tous ses coups pouvaient l’amener à la victoire, mais parfois dans de plus nombreux coups que d’autres, elle attendait le joueur pouvait la contrer. Il a donc fallu ajouter un test, et en cas d’égalité de score entre plusieurs configurations, on teste la valeur du coup (ancienne fonction d’évaluation), et on prend le coup ayant la plus haute. Une autre amélioration a été faite : si jamais un nœud possède un fils détectant la victoire ou défaite assurée (4 pions alignés pour un des joueurs), il n’est pas nécessaire de calculer les autres fils, on a déjà la valeur de ce nœud.

**5.Explication du code**

### **Constantes & attributs**

Les constantes suivantes permettent de contrôler les paramètres globaux du jeu.

| **Constantes** | **Définitions** |
| --- | --- |
| n | Représente le nombre de lignes du plateau |
| m | Représente le nombre de colonnes du plateau |

**n: 6,**

**// Nombre de colonnes**

**m: 7,**

Les attributs suivants sont utilisés pour modéliser le jeu.

| **Attributs** | **Définitions** |
| --- | --- |
| dom\_plateau : [] | Représente le plateau sous la forme d'un tableau |
| coups : int | Représente le nombre de coups joués au total |
| turn : int | Permet de savoir à quel joueur c'est le tour |
| game\_status : int | Permet de savoir si une partie est terminée |

**// ce tableau contient des pointeurs directs vers les cases**

**// (noeuds <td> du DOM) du plateau dans la page html**

**dom\_plateau: [],**

**// un entier: 1 ou 2 (le numéro du prochain joueur)**

**turn: 1,**

**/\* un entier:**

**0: la partie continue**

**-1: la partie est nulle**

**1: joueur 1 a gagné**

**2: joueur 2 a gagné**

**\*/**

**game\_status: 0,**

**// Nombre de coups joués**

**coups: 0,**

### **Méthodes**

#### init(parent, lignes, colonnes)

Méthode permettant d'initialiser l'objet p4 et de créer le tableau HTML contenant un certain nombre de lignes et de colonnes et d'ajouter le ***handler*** gérant le click pour jouer.

| **Paramètres** | **Définitions** |
| --- | --- |
| parents | Le nom de l'élément HTML auquel ajouté l'objet p4 |
| lignes | Le nombre de lignes |
| colonnes | Le nombre de colonnes |

**if (lignes) this.n = lignes;**

**if (colonnes) this.m = colonnes;**

**t = document.createElement('table');**

**t.id = 'plateau';**

**for (var i = this.n - 1; i >= 0; i--) {**

**var tr = document.createElement('tr');**

**this.dom\_plateau[i] = [];**

**for (var j = 0; j < this.m; j++) {**

**var td = document.createElement('td');**

**td.dataset.column = j;**

**tr.appendChild(td);**

**this.dom\_plateau[i][j] = td;**

**}**

**t.appendChild(tr);**

**}**

**parent.innerHTML = '';**

**parent.appendChild(t);**

**t.addEventListener('click', function(e) { p4.handler(e); });**

**}**

#### **set(row, column, player)**

Méthode permettant de remplir la case du tableau après un coup, incrémente le nombre de coup et change le tour actuel.

| **Paramètres** | **Définitions** |
| --- | --- |
| row | Ligne du coup à jouer |
| column | Colonne du coup à jouer |
| player | Numéro du joueur qui à jouer le coup |

**// On colore la case**

**this.dom\_plateau[row][column].className = 'joueur' + player;**

**// On compte le coup**

**this.coups++;**

**// On passe le tour : 3 - 2 = 1, 3 - 1 = 2**

**this.turn = 3 - this.turn;**

#### **play(column)**

Méthode permettant de jouer un coup à partir de la colonne choisie (récupérée lors du clic), afin de pouvoir cliquer n'importe où sur la colonne et non pas sur un point précis puis vérifie si la partie est finie et affiche les résultats si c'est le cas.

| **Paramètres** | **Définitions** |
| --- | --- |
| column | La colonne dans laquelle jouer le coup |

**// Vérifier si la partie est encore en cours**

**if (this.game\_status != 0) {**

**/\*if (window.confirm("La partie est finie!\n\nSouhaitez-vous recommencer?")) {**

**this.reset();**

**}\*/**

**return;**

**}**

**// Trouver la première case libre dans la colonne**

**var row;**

**for (var i = 0; i < this.n; i++) {**

**if (!this.dom\_plateau[i][column].className) {**

**row = i;**

**break;**

**}**

**}**

**if (row === undefined) {**

**window.alert("La colonne est pleine!");**

**return;**

**}**

**// Effectuer le coup**

**this.set(row, column, this.turn);**

**// Vérifier s'il y a un gagnant, ou si la partie est finie**

**if (this.win(row, column, 'joueur' + (3 - this.turn))) {**

**this.game\_status = 3 - this.turn;**

**} else if (this.coups >= this.n \* this.m) {**

**this.game\_status = -1;**

**}**

**//Au cours de l'affichage, pensez eventuellement, à afficher un**

**//message si la partie est finie...**

**switch (this.game\_status) {**

**case -1:**

**// window.alert("Partie Nulle!!");**

**var txtElem = document.createElement('h2');**

**txtElem.id='resultat';**

**var txt = document.createTextNode("Partie nulle !");**

**txtElem.appendChild(txt);**

**document.getElementById("banniere").appendChild(txtElem);**

**var buttonElem= document.createElement('button');**

**buttonElem.id='bouton';**

**var txtButton = document.createTextNode("Rejouer !");**

**buttonElem.appendChild(txtButton);**

**buttonElem.onclick = p4.resetButton;**

**document.getElementById("wrapper").appendChild(buttonElem);**

**break;**

**case 1:**

**//window.alert("Victoire du joueur 1");**

**var txtElem = document.createElement('h2');**

**txtElem.id='resultat';**

**var txt = document.createTextNode("Victoire du joueur !");**

**txtElem.appendChild(txt);**

**document.getElementById("banniere").appendChild(txtElem);**

**var buttonElem= document.createElement('button');**

**buttonElem.id='bouton';**

**var txtButton = document.createTextNode("Rejouer !");**

**buttonElem.appendChild(txtButton);**

**buttonElem.onclick = p4.resetButton;**

**document.getElementById("wrapper").appendChild(buttonElem);**

**break;**

**case 2:**

**// window.alert("Victoire du joueur 2");**

**var txtElem = document.createElement('h2');**

**txtElem.id='resultat';**

**var txt = document.createTextNode("Victoire de l'IA !");**

**txtElem.appendChild(txt);**

**document.getElementById("banniere").appendChild(txtElem);**

**var buttonElem= document.createElement('button');**

**buttonElem.id='bouton';**

**var txtButton = document.createTextNode("Rejouer !");**

**buttonElem.appendChild(txtButton);**

**buttonElem.onclick = p4.resetButton;**

**document.getElementById("wrapper").appendChild(buttonElem);**

**break;**

**}**

#### **handler(event)**

Méthode permettant d'appeler la fonction play lors du clic de l'utilisateur ainsi que d'appeller la fonction permettant à l'IA de jouer après le coup du joueur.

| **Paramètres** | **Définitions** |
| --- | --- |
| event | Le clic sur la colonne dans laquelle le joueur veut jouer |

**var column = event.target.dataset.column;**

**//attention, les variables dans les datasets sont TOUJOURS**

**//des chaînes de caractère. Si on veut être sûr de ne pas faire de bêtise,**

**//il vaut mieux la convertir en entier avec parseInt**

**if (column){**

**this.play(parseInt(column));**

**}**

**if(this.getTurn()==2){**

**bestMoove = this.minmax(this.getTab(), 7, Number.NEGATIVE\_INFINITY,+Number.POSITIVE\_INFINITY,true);**

**console.log(bestMoove);**

**this.play(parseInt(bestMoove["Moove"]));**

#### **getTab()**

Méthode permettant de récupérer le plateau de jeu sous forme de tableau **JavaScript**

| **Paramètres** | **Définitions** |
| --- | --- |
| RETURN | Le plateau de jeu sous forme de tableau javascript |

**var resTab = []**

**for(var i = 0 ; i < this.n ; i++){**

**resTab.push([]);**

**}**

**for (var i = this.n - 1; i >= 0; i--){**

**var string = "";**

**for (var j = 0 ; j < this.m ; j++){**

**if(this.dom\_plateau[i][j].className == "joueur1"){**

**string+="1 ";**

**resTab[i].push(1);**

**}else if(this.dom\_plateau[i][j].className == "joueur2") {**

**string+="2 ";**

**resTab[i].push(2);**

**}else{**

**string+="0 ";**

**resTab[i].push(0);**

**}**

**}**

**}**

return resTab;

#### **is\_won(board, player)**

Méthode permettant de savoir si le joueur à gagné.

| **Paramètres** | **Définitions** |
| --- | --- |
| board | Le plateau de jeu à tester |
| player | Le numéro du joueur à tester |
| RETURN | True si player à gagner, false sinon |

**// On vérifie les ligne**

**for(var i = 0 ; i<this.m-3 ; i++){**

**for(var j = 0 ; j<this.n ; j++){**

**if(board[j][i]==player && board[j][i+1]==player && board[j][i+2]==player && board[j][i+3]==player){**

**return true;**

**}**

**}**

**}**

**//On vérifie les colonnes**

**for(var i = 0 ; i<this.m ; i++){**

**for(var j = 0 ; j<this.n-3 ; j++){**

**if(board[j][i]==player && board[j+1][i] == player && board[j+2][i]==player && board[j+3][i]==player){**

**return true;**

**}**

**}**

**}**

**//on vérifie les diagonales qui montent**

**for(var i = 0 ; i<this.m-3 ; i++){**

**for(var j = 0 ; j<this.n-3 ; j++){**

**if(board[j][i]==player && board[j+1][i+1]==player && board[j+2][i+2]==player && board[j+3][i+3]==player){**

**return true;**

**}**

**}**

**}**

**//on vérifie les diagonales qui descendent**

**for(var i = 0 ; i<this.m-3 ; i++){**

**for(var j = 3 ; j<this.n ; j++){**

**if(board[j][i]==player && board[j-1][i+1]==player && board[j-2][i+2]==player && board[j-3][i+3]==player){**

**return true;**

**}**

**}**

**}**

**return false;**

#### **getMovePlayable(board)**

Méthode permettant de récupérer les numéros des colonnes où l'on peut jouer sous forme de tableau **JavaScript**.

| **Paramètres** | **Définitions** |
| --- | --- |
| board | Le plateau de jeu à tester |
| RETURN | Un tableau contenant les numéros des colonnes où l'on peut jouer |

**var resTab = [];**

**for (var j = 0 ; j < this.m ; j++){**

**if(board[this.n-1][j] == 0) resTab.push(j)**

**}**

**return resTab;**

#### **is\_terminal\_node(board)**

Méthode permettant de savoir si la partie est finie

| **Paramètres** | **Définitions** |
| --- | --- |
| board | Le plateau de jeu à tester |
| RETURN | True si un joueur à gagner ou qu'aucun coup n'est possible, false sinon |

**return (is\_won(board,1) || is\_won(board,2) || getMovePlayable(board).length == 0)**

#### **evaluate\_window(window, player)**

Méthode permettant de retourner une valeur représentant les possibilités offertes au joueur dans une "fenêtre" de 4 pions.

| **Paramètres** | **Définitions** |
| --- | --- |
| window | Tableau représentant 4 pions du plateau |
| RETURN | Le score représentant les possibilités offerte par la fenêtre au joueur |

**countOwn = 0;**

**countAdv = 0;**

**for(var i = 0 ; i<4 ; i++){**

**if(window[i] == player) countOwn++;**

**else if(window[i] !=0) countAdv++;**

**}**

**//console.log("countOwn : "+countOwn);**

**//console.log("Count adv : "+countAdv)**

**if(countOwn==4) return 100;**

**else if(countOwn == 3 && countAdv == 0) return 5;**

**else if(countOwn == 2 && countAdv == 0) return 2;**

**else if(countAdv == 3 && countOwn == 0) return -4;**

**return 0;**

**},**

**score\_position: function(board, player){**

**var score = 0;**

**//on test toutes les lignes possibles à réaliser**

**for(i = 0 ; i< this.n; i++){**

**for(j = 0 ; j<this.m-3 ; j++){**

**window\_test = [];**

**window\_test.push(board[i][j]);**

**window\_test.push(board[i][j+1]);**

**window\_test.push(board[i][j+2]);**

**window\_test.push(board[i][j+3]);**

**score+= this.evaluate\_window(window\_test, player);**

**//console.log(window\_test);**

**}**

**}**

**for(i = 0 ; i < this.m ; i++){**

**for(j = 0; j< this.n-3 ; j++){**

**window\_test = [];**

**window\_test.push(board[j][i]);**

**window\_test.push(board[j+1][i]);**

**window\_test.push(board[j+2][i]);**

**window\_test.push(board[j+3][i])**

**score+= this.evaluate\_window(window\_test, player);**

**//console.log(window\_test);**

**//console.log(score)**

**}**

**}**

**//diagonale qui monte**

**for(i = 0 ; i< this.n - 3 ; i++){**

**for(j = 0 ; j < this.m - 3 ; j++){**

**window\_test = []**

**window\_test.push(board[i][j]);**

**window\_test.push(board[i+1][j+1]);**

**window\_test.push(board[i+2][j+2]);**

**window\_test.push(board[i+3][j+3]);**

**score+= this.evaluate\_window(window\_test, player);**

**//console.log(window\_test);**

**//console.log(score);**

**}**

**}**

**for(i=0 ; i<this.n-3 ; i++){**

**for(j = 0 ; j< this.m - 3 ; j++){**

**window\_test = [];**

**window\_test.push(board[i+3][j]);**

**window\_test.push(board[i+3-1][j+1]);**

**window\_test.push(board[i+3-2][j+2]);**

**window\_test.push(board[i][j+3]);**

**//console.log(window\_test);**

**//console.log(score);**

**}**

**}**

**return score;**

#### **getRandomInt(max)**

Méthode permettant de choisir un entier au hasard compris entre 0 et max-1

| **Paramètres** | **Définitions** |
| --- | --- |
| max | La borne supérieur des choix possibles |
| RETURN | Un entier aléatoire compris entre 0 et max-1 |

**return Math.floor(Math.random() \* Math.floor(max));**

#### **minmax(board, depth, alpha, beta, maximizingPlayer)**

Méthode récursive de l'algorithme minmax

| **Paramètres** | **Définitions** |
| --- | --- |
| board | Plateau sur lequel on applique le minmax |
| depth | Entier permettant de limiter la profondeur de la recherche dans l'arbre |
| alpha | Valeur du paramètre alpha pour la coupe alpha-bêta |
| beta | Valeur du paramètre beta pour la coupe alpha-bêta |
| maximizingPlayer | Boolean permettant de savoir si on est le joueur qui maximise ou celui qui minimise |
| RETURN | Un couple (Valeur, Moove) représentant le score et la colonne du meilleur coup possible |

**var valid\_moove = this.getMovePlayable(board);**

**var res = new Object();**

**//var res = []**

**is\_terminal\_node = this.is\_won(board, 1) || this.is\_won(board, 2);**

**if(depth == 0 || is\_terminal\_node){**

**if(is\_terminal\_node){**

**if(this.is\_won(board, 2)){**

**res["Moove"]= 'None';**

**res["Score"]= 9999999999;**

**//console.log(depth);**

**//console.log(res);**

**return res;**

**}else if(this.is\_won(board, 1)){**

**res["Moove"]= 'None';**

**res["Score"]= -99999999;**

**//console.log(res);**

**//console.log(depth);**

**//console.log(res);**

**return res;**

**}else{**

**res["Moove"]= 'None';**

**res["Score"]= 0;**

**//console.log(depth);**

**//console.log(res);**

**}**

**}else{**

**res["Moove"]= 'None';**

**res["Score"] = this.score\_position(board, 2);**

**//console.log(res);**

**//console.log(depth);**

**//console.log(res);**

**return res;**

**}**

**}else{**

**if(maximizingPlayer){**

**value = Number.NEGATIVE\_INFINITY;**

**var choices = this.getMovePlayable(board);**

**var number\_of\_choice = choices.length;**

**var column = this.getRandomInt(number\_of\_choice);**

**for(var i = 0 ; i < number\_of\_choice ; i++){**

**var tempBoard = []**

**var col = choices[i];**

**row = this.get\_open\_row(board, col);**

**for(z = 0 ; z<this.n ; z++){**

**tempBoard.push(Array.from(board[z]))**

**}**

**tempBoard[row][col] = 2;**

**var new\_score = this.minmax(tempBoard, depth-1, alpha, beta, false)**

**if(new\_score["Score"] > value){**

**value = new\_score["Score"];**

**column = col;**

**}**

**var alpha = Math.max(alpha, value);**

**if(alpha >= beta){**

**break;**

**}**

**}**

**res["Moove"] = column;**

**res["Score"] = value;**

**}else{**

**var value = Number.POSITIVE\_INFINITY;**

**var choices = this.getMovePlayable(board);**

**var number\_of\_choice = choices.length;**

**var column = this.getRandomInt(number\_of\_choice);**

**for(i=0 ; i<number\_of\_choice ; i++){**

**var tempBoard = [];**

**var col = choices[i];**

**row = this.get\_open\_row(board,col);**

**for(j=0;j<this.n;j++){**

**tempBoard.push(Array.from(board[j]));**

**}**

**tempBoard[row][col] = 1;**

**var new\_score = this.minmax(tempBoard, depth-1, alpha, beta, true)**

**if(new\_score["Score"] < value){**

**value = new\_score["Score"];**

**column = col;**

**}**

**var beta = Math.min(beta, value)**

**if(alpha >= beta){**

**break;**

**}**

**}**

**res["Moove"] = column;**

**res["Score"] = value;**

**}**

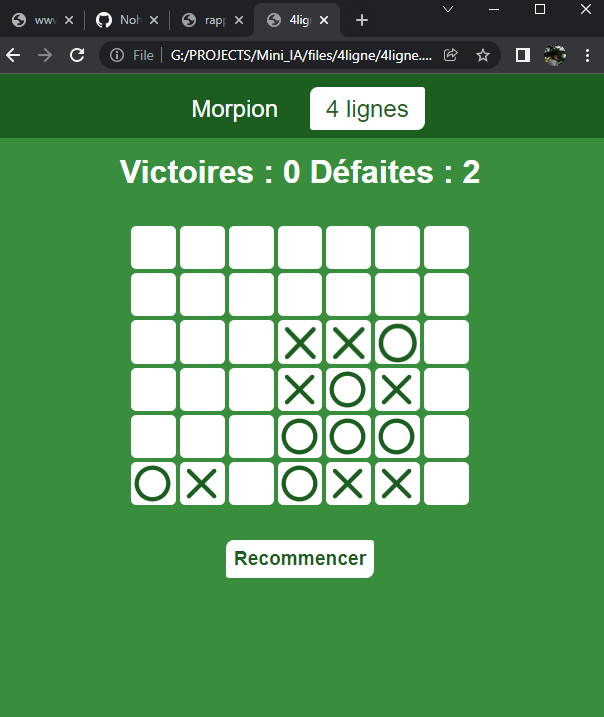
**}**

**//console.log(depth);**

**//console.log(res);**

**return res;**

**Chapitre IV : Captures d’écran**



**Conclusion**

En guise de conclusion, on vous a présenté notre mini-projet d’intelligence artificielle avec l’utilisation de l’algorithme de **Minimax**. La réalisation du projet nous a permis d’améliorer notre connaissance de l’IA, en programmation et aussi en gestion de travail en équipe. Nous espérons que cette réalisation répond aux norme requis pour être considéré comme un bon travail car nous y avons mis les efforts nécessaires. Les améliorations possibles pour ce projet sont multiples, comme l’ajout des jeux N-puzzle, le juste prix et tant d’autre, mais aussi la possibilité de joueur entre deux humains même si cela sort du cadre de l’intelligence artificielle. Pouvoir voir deux IA s’affronter est aussi une amélioration possible.

**Bibliographie**

1. **Les cours dispensés à l’Institut Supérieur Polytechnique de Madagascar**
2. **Cours d’Intelligence Artificielle** du Pr RABOANARY Julien Amedée
3. **Cours d’Algorithme avancé** de mr RABOANARY Andry
4. **Cours de Technologie JavaScript** de mr FIDIMALALA Ismael
5. **L’intelligence Artificielle pour les développeurs** de Virginie MATHIVET