# Travail de maturité - Puissance4IA

Torrenté Florian

10 octobre 2021

## Table des matières

# 1 Introduction

## 1.1 Problématique et objectifs

La problématique de ce travail est : "Dans quelle mesure l'implémentation d'une intelligence artificielle pour jouer au Puissance 4 est-elle compliquée, et quelles en sont les difficultés ?"

Dans ce travail, je voulais renforcer mes connaissances sur les différents langages webs (HTML, CSS3 et JavaScript) mais surtout me rapprocher du domaine de l'intelligence artificielle qui à l'air d'être un sujet très prometteur pour le future. En clair, les objectifs de ce travail étaient :

1. Comprendre le fonctionnement d'une intelligence artificielle
2. Développer un ou plusieurs algorithmes
3. Comprendre les limites de ces algorithmes et essayer de les dépasser
4. Créer une intelligence que je ne pourrai plus battre au Puissance 4

## 1.2 Description et règles du jeu

Le Puissance 4 est un jeu avec des règles très simples. Le but du jeu est d'aligner quatre pions de même couleur (horizontalement, verticalement, ou en diagonale). Le terrain de jeu est une grille de 7x6 (sept colonnes et six rangées). Chaque joueur possède les pions d'une couleur (généralement jaune et rouge). Chacun son tour, les joueurs déposent un pion dans la colonne de leur choix, le pion descend alors le plus bas possible dans la colonne. Le premier joueur à aligner quatre pions de sa couleur gagne. S'il n'y a plus de place pour jouer, la partie est nulle.

FIGURE 1 – Un exemple de partie gagnée par le joueur rouge.

Les règles sont résolument simples, mais il y a une raison supplémentaire pour laquelle j'ai choisi ce jeu : c'est un jeu à information complète. Cela veut dire que chaque joueur connait :

— tous les coups qu'il peut jouer ;
— tous les coups que son adversaire peut jouer ;
— les gains résultants de ces actions ;
— le but de l'autre joueur.

## 1.3 L'intelligence artificielle

L'humanité s'est donné le nom scientifique **homo sapiens**—l'homme sage—parce que nos capacités mentales sont extrêmement importantes pour nous et notre sentiment d'identité. Le domaine de l'intelligence artificielle (ou IA) tente de comprendre cette intelligence. C'est pourquoi l'étudier peut nous permettre d'en apprendre davantage sur nous-même. Contrairement à la philosophie et à la psychologie, qui s'intéressent aussi à l'intelligence, l'IA essaye de *construire* des entités intelligentes et de les comprendre.

Une autre raison d'étudier l'IA est que ces entités construites sont intéressantes et utiles en elles-mêmes. En effet, ces dernières ont donné naissance à de nombreux résultats significatifs et impressionnants.

Maintenant, nous savons pourquoi l'IA est intéressant et important, nous avons toujours besoin de savoir précisément *ce que c'est*. On pourrait simplement dire : "Eh bien, ça a à voir avec les programmes intelligents", mais je pense qu'il est important de bien définir des objectifs pour pouvoir les atteindre. Une définition plus cohérente pour moi serait celle-ci : "L'intelligence artificielle a pour objectif de construire des dispositifs simulant les processus cognitifs humains" [1]

## 1.4 La structure des données

Pour implémenter cet algorithme, il faut définir une structure de donnée très légère et optimisée afin de pouvoir calculer des milliers de parties en quelques secondes. En informatique, un *long integer* est composé de huit octets, c'est-à-dire 64 bits. On numérote les bits de ce nombre de droite à gauche (sans oublier qu'en informatique on commence à compter à 0)

```
63 62 61 60 ... 48 47 46 45 ... 3 2 1 0
```

Comme on l'a vu précédemment, le Puissance 4 se joue sur un *board* vertical, avec sept colonnes et six lignes, ce qui fait 42 cases. Comme on peut le voir sur le diagramme ci-dessous, on ajoute une ligne en haut et deux colonnes à droite pour des raisons de calculs plus tard. Cette représentation est appelée *bitboard*

```
  6 13 20 27 34 41 48   55 62     Ligne supplementaire
+---------------------+
| 5 12 19 26 33 40 47 | 54 61     Ligne du haut
| 4 11 18 25 32 39 46 | 53 60
| 3 10 17 24 31 38 45 | 52 59
| 2  9 16 23 30 37 44 | 51 58
| 1  8 15 22 29 36 43 | 50 57
| 0  7 14 21 28 35 42 | 49 56 63  Ligne du bas
+---------------------+
```

FIGURE 2 – Représentation du bitboard

Les nombres indiquent la position dans la représentation binaire d'un long.
Prenons un exemple en représentant cette position :

```
. . . . . .
. . . . . .
. . . . . .
. . . O . . .
. . . X X . .
. . O X O . .
-------------
0 1 2 3 4 5 6  Colonne
```

FIGURE 3 – Partie en cours

Comme le jeu est joué par deux joueurs, on utilisera un bitboard par joueur.

```
              0 0 0 0 0 0 0  0 0 0 0 0 0 0   6 13 20 27 34 41 48
. . . . . .   0 0 0 0 0 0 0  0 0 0 0 0 0 0   5 12 19 26 33 40 47
. . . . . .   0 0 0 0 0 0 0  0 0 0 0 0 0 0   4 11 18 25 32 39 46
. . . . . .   0 0 0 0 0 0 0  0 0 0 0 0 0 0   3 10 17 24 31 38 45
. . . O . . .  0 0 0 0 0 0 0  0 0 0 1 0 0 0   2  9 16 23 30 37 44
. . . X X . .  0 0 0 1 1 0 0  0 0 0 0 0 0 0   1  8 15 22 29 36 43
. . O X O . .  0 0 0 1 0 0 0  0 0 1 0 1 0 0   0  7 14 21 28 35 42
-------------     Joueur 1       Joueur 2
0 1 2 3 4 5 6
```

FIGURE 4 – Partie en cours avec les bitboards

1. HAIECH, *Parcourir l'histoire de l'intelligence artificielle, pour mieux la définir et la comprendre.*

On peut aussi le voir "à plat", ce qui ressemble à ceci :

```
... 0000000 0000000 0000010 0000011 0000000 0000000 0000000 // Joueur 1
... 0000000 0000000 0000001 0000100 0000001 0000000 0000000 // Joueur 2
     col 6   col 5   col 4   col 3   col 2   col 1   col 0
```

FIGURE 5 – Représentation des bitboards "à plat"

Cette manière de représenter les états du jeu est la clé à la vitesse de l'algorithme. En effet, cette représentation permet en utilisant seulement des opérations de base de jouer des coups, de tester s'il y a une victoire, une nulle, etc... Enfin, cette représentation est très légère et permet donc un stockage assez simple et demande très peu de RAM pour fonctionner.

Maintenant qu'on a vu comment les boards sont représentés, on va pouvoir explorer comment on va traiter ces données, pour jouer des coups et tester s'il y a une victoire. Pour ce faire on n'a besoin que de deux types d'opérations : le *bit shifting* (décalage de bit) et la combinaison de bit (notamment XOR et AND).

### Le décalage de bits

Pour décaler des bits en informatique, on utilise 2 opérateurs : `»` et `«`. Respectivement *right shift* et *left shift*. Par exemple `0b10101110 « 3` signifie que l'on retire les 3 premiers nombres à gauche et on ajoute 3 zéros à droite. Ici, le préfixe 0b indique que les chiffres suivants représentent un nombre binaire et non un nombre décimal. Donc `0b10101110 « 3 = 0b01110000`.

De manière similaire, `0b10101110 » 3 = 0b00010101`.

### La combinaison de bits

Les deux seuls opérateurs dont nous aurons besoin sont le *XOR* et le *AND*. L'opérateur XOR, ou OR exlusif, (qu'on écrit `^`) prend deux nombres binaires, par exemple `0b10101110 ^ 0b10011111`, et compare les bits de même position. Si les deux bits sont différents, le résultat est 1, sinon, c'est 0.

Si on écrits les deux nombres l'un sur l'autre, les bits sont facile à comparer.

```
  10101110
^ 10011111
----------
  00110001
```

FIGURE 6 – Exemple de l'opérateur XOR

L'opérateur AND (qu'on écrit `&`) prend aussi 2 nombres binaires et les compare bit par bit. Si les deux bits sont un 1, le résultat est 1, sinon 0.

```
  10101110
& 10011111
----------
  10001110
```

FIGURE 7 – Exemple de l'opérateur AND

## 1.5 Application des bitboards pour le puissance 4

Pour montrer l'utilisation des bitboards je vais vous présenter les 2 fonctions principales dont nous avons besoin : `makeMove` et `isWin`. La première permet de jouer un coup tandis que la seconde permet de déterminer si un joueur a gagné.

Avant d'examiner ces fonctions, nous avons encore besoin de garder en mémoire : la position à remplir de chaque colonne qui sera stockée dans le tableau `heights` et la hauteur max de chaque colonne qui sera stockée dans le tableau `max_heights`.

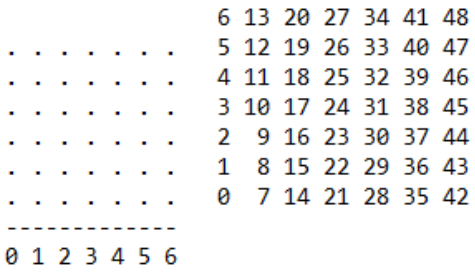


FIGURE 8 – Début de partie

Comme on le voit dans la figure ci-dessus, au début de la partie les hauteurs sont : heights = {0, 7, 14, 21, 28, 35, 42} et les hauteurs max sont : max_heights = {6,13,20,27,34,41,48}. À chaque fois qu'un coup sera joué dans une colonne, on augmentera l'indice de la colonne de 1. En gardant ceci en mémoire, cela nous évite de devoir chercher la prochaine case vide à chaque coup joué.

## Jouer un coup

```
1        function makeMove(column, playerBoard) {
2            move = 1 << height[column]; // (1)
3            playerBoard = playerBoard ^ move;  // (2)
4            height[column] += 1; // (3)
5        }
```

Voici ce que le code fait :

1. On récupère la valeur height en fonction de la colonne donnée et on l'utilise pour décaler le nombre 1 vers la gauche jusqu'à cette position. Ensuite on stock cette valeur dans la variable move.
2. On prend le board représentant le joueur et on utilise l'opérateur XOR avec la variable move. ( playerBoard ^ move ) et on met à jour la nouvelle valeur du bitboard.
3. On incrémente la valeur de la hauteur de cette colonne de 1 pour être prêt pour le prochain coup.

Je pense qu'un exemple aidera à mieux visualiser la situation. Imaginons que je veux jouer un coup dans la colonne 3. Je vais donc chercher l'index de la hauteur de cette colonne : heights[3] qui vaut 21. Je décale donc mon 1 21 fois vers la gauche ( 1 « 21 ).

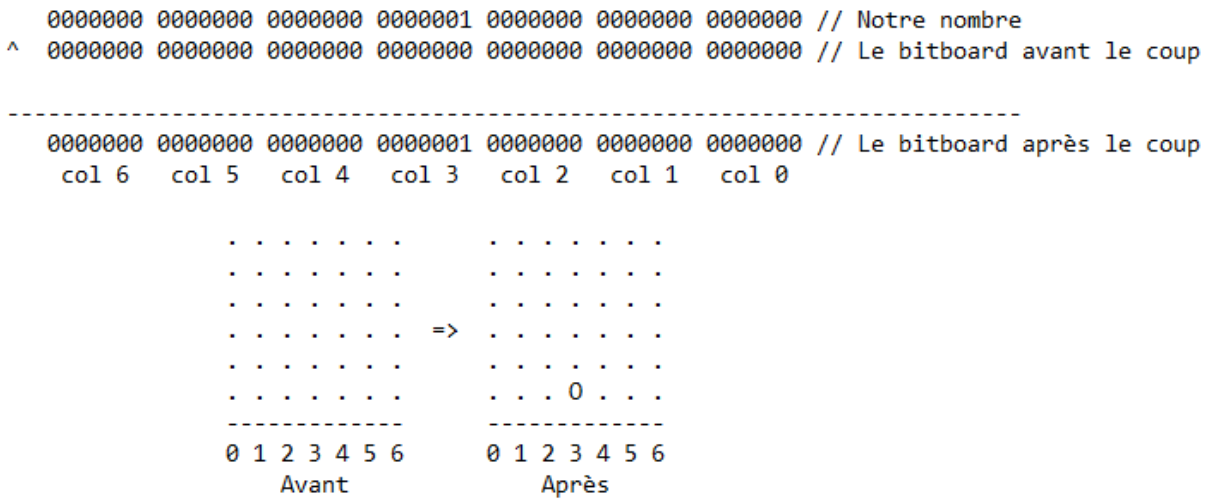


FIGURE 9 – Exemple de la fonction makeMove

## Est-ce qu'un joueur à gagné ?

Reprenons la représentation du bitboard :

```
  6 13 20 27 34 41 48   55 62     Ligne supplementaire
+---------------------+
| 5 12 19 26 33 40 47 | 54 61     Ligne du haut
| 4 11 18 25 32 39 46 | 53 60
| 3 10 17 24 31 38 45 | 52 59
| 2  9 16 23 30 37 44 | 51 58
| 1  8 15 22 29 36 43 | 50 57
| 0  7 14 21 28 35 42 | 49 56 63  Ligne du bas
+---------------------+
```

FIGURE 10 – Représentation du bitboard

Pour savoir s'il y a 4 pièces alignées horizontalement, on doit, par exemple, regarder si les positions 11, 18, 25 et 32 sont toutes des 1. On pourrait aussi regarder les positions 21, 28, 35 et 42. Le pattern là-dessous est simple : on prend la position la plus à gauche et on ajoute 7 à chaque fois.

Verticalement, ce nombre est 1 (par exemple les positions 15 et 16). En diagonale, ce nombre est soit 8 (par exemple 16 et 24) ou alors 6 (30 et 36).

Donc les nombres "magiques" sont 1, 6, 7 et 8.

Pour tester s'il y a 4 pièces alignées, on va prendre notre bitboard, faire une copie et la décaler à droite d'un de ces nombres, refaire une copie et la décaler de 2 fois plus et encore une copie, mais cette fois-ci 3 fois plus. Ensuite, on combine toutes les copies avec l'opérateur AND et le tour est joué. Si la réponse est différente de 0 alors il y a 4 pièces alignées.

Encore une fois, un exemple sera probablement plus clair :

```
. . . . . .
. . . . . .
. . . O . . .
. . . O . . .
. . . O X . .
. . . O X X .
-------------
0 1 2 3 4 5 6

   col 6   col 5   col 4   col 3   col 2   col 1   col 0
  0000000 0000000 0000000 0001111 0000000 0000000 0000000 // Le bitboard du joueur O

^ 0000000 0000000 0000000 0000111 1000000 0000000 0000000 // Première copie >> 1
^ 0000000 0000000 0000000 0000011 1100000 0000000 0000000 // Deuxième copie >> 2
^ 0000000 0000000 0000000 0000001 1110000 0000000 0000000 // Troisième copie >> 3
-------------------------------------------------------------------------------
  0000000 0000000 0000000 0000001 0000000 0000000 0000000
```

FIGURE 11 – Exemple de la fonction isWin

Maintenant que nous savons comment ça marche, il suffit de l'implémenter pour toutes les directions. Le symbole `!=` signifie "différent de".

```
1    isWin(bitboard) {
2        if (bitboard & (bitboard >> 6) & (bitboard >> 12) & (bitboard >> 18) != 0) return
             true; // diagonal \
3        if (bitboard & (bitboard >> 8) & (bitboard >> 16) & (bitboard >> 24) != 0) return
             true; // diagonal /
4        if (bitboard & (bitboard >> 7) & (bitboard >> 14) & (bitboard >> 21) != 0) return
             true; // horizontal
5        if (bitboard & (bitboard >> 1) & (bitboard >>  2) & (bitboard >>  3) != 0) return
             true; // vertical
6        return false;
7    }
```

Par direction, la fonction a besoin d'environ 7 opérations : 3 shifts, 3 AND et 1 comparaison. Une évaluation complète du board requiert donc 4x7=28 opérations. Les ordinateurs actuels effectuent environ 3 milliards d'opérations par seconde, on peut donc évaluer environ **100 millions de positions par seconde**.

# 2 Déroulement du travail

## 2.1 Structure et technologies

Le projet est divisé en 2 parties principales : le *client* et le *serveur*. Le premier contient tout ce que l'utilisateur verra (la page internet, le jeu) et gérera toutes les interactions avec ce dernier. Toutes les informations du client sont ensuite traitées par le serveur à distance. C'est donc le serveur qui sera en charge de calculer les coups de l'IA et de les envoyer au client. Cette structure permet entre autres d'avoir une interface client très légère et donc utilisable sur tous les appareils. Le problème est que la charge de calcul est entièrement sur le serveur, ce qui peut poser problème si beaucoup d'utilisateurs voulaient affronter l'IA.

Pour tout le projet, j'utilise NPM (pour "*Node Package Manager*") qui permet de gérer les dépendances de mon projet simplement. Toutes les informations pour NPM sont contenues dansun fichier *package.json*. On y trouve : la description du projet (nom, description, version, auteur, etc...), les différents scripts (raccourcis pour lancer, générer le projet) et les dépendances.

### Le client

Dans le dossier client se trouve :

1. Le dossier css
2. Le dossier js
3. index.html
4. package.json

(1) Le dossier CSS (pour *"Cascading StyleSheet"*) contient les feuilles de style qui permettent de modifier l'apparence de ma page web. Pour ce projet, j'ai décidé de ne pas utiliser le langage CSS, mais de passer par SASS (Syntactically Awesome Stylesheets). Ce langage permet d'écrire un code plus visuel et pratique qui sera ensuite compilé en CSS.

(2) Le dossier JS (pour *JavaScript*) contient toute la logique de la page. En effet, le JavaScript est un langage qui permet de rendre une page web dynamique et d'y dessiner ce qu'on veut (comme par exemple un jeu du puissance 4). Pour dessiner le plateau de jeu, j'utilise la librairie *p5js* qui gère de manière élégante et légère toutes les actions de dessins et les interactions avec l'utilisateur. Pour ce projet, j'ai aussi décidé d'utiliser *TypeScript* qui est un langage (qui se compile en JavaScript) permettant d'avoir un code plus prévisible et d'attraper les erreurs plus tôt.

(3) Le fichier *index.html* contient simplement la structure de la page et met charge le css et le js.

(4) Le *package.json* contient les informations pour NPM, permettant notamment d'automatiser la compilation du TypeScript et du SASS, ainsi que l'installation de p5js.

En résumé, pour le client j'utilise 3 langages (HTML, SASS et TypeScript), la librairie p5js et le gestionnaire de paquet NPM.

### Le serveur

Le dossier serveur quant à lui contient :

1. Le dossier src
   (a) Bitboard.ts
   (b) AI.ts
   (c) index.ts
2. package.json

(1) Le dossier *src* (abréviation de *source*, comme "code source") contient toute la logique du serveur. Tout le code qu'il contient est écrit en TypeScript. Il contient 3 fichiers :

(a) Le fichier Bitboard.ts (*ts* est l'extension des fichiers contenant du code TypeScript) qui contient l'implémentation des bitboards.

(b) Le fichier AI.ts qui contient toute la logique de l'intelligence artificielle

(c) Enfin, l'index.ts contient la logique pour recevoir et envoyer des informations au client.

(2) Le package.json contient, comme vu précédemment, toutes les informations pour le gestionnaire de paquets NPM.

## 2.2 Implémentation du client

Dans cette section, je vais survoler l'implémentation du client, son évolution et les différents problèmes que j'ai pu rencontrer.

Pour commencer, le client étant une page web, j'ai créé le `index.html`. Ce fichier appelé "index", par convention, contient la structure de notre page. C'est aussi ce fichier qui va regrouper la structure, le code

Javascript et le style. En réalité, sa structure est extremement simple, elle contient simplement quelques informations, et un endroit préparé pour afficher le jeu.

Les deux fichiers les plus importants du client sont le fichier `sketch.ts` et `connect4.ts`. A eux deux, ils gèrent tout le jeu et les intéractions avec l'utilisateur.

A l'origine, la page ressemblait à ceci :

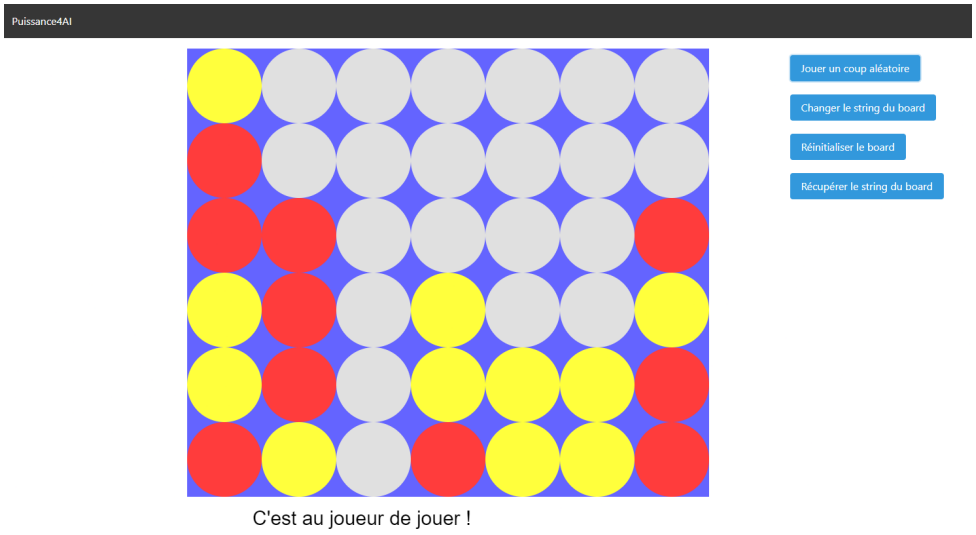


FIGURE 12 – Première version du client

Ce n'était pas très beau, pas très ergonomique et pas dynamique du tout. Mais cela suffisait pour mes tests. Après quelques temps j'ai décidé de changer tout cela, et de donner une look plus moderne à mon travail. Voici ce à quoi la page ressemble maintenant :

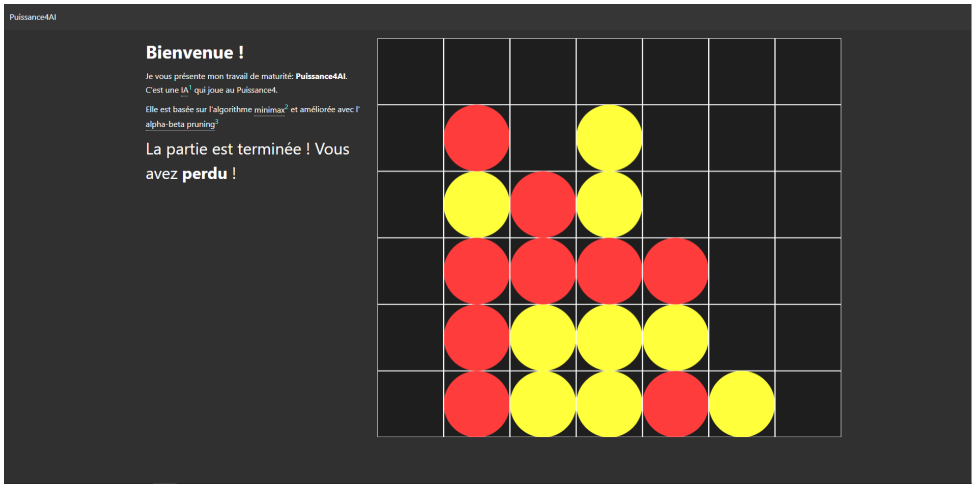


FIGURE 13 – Deuxième version du client

En plus d'être plus joli, il y a une animation quand les pièces tombent pour que cela soit plus agréable.

En soit, dans l'implémentation je n'ai renontré presque aucun problème, en effet, j'étais habitué à développer ce genre de jeu, la seule nouveauté était le langage, car les navigateurs ne peuvent intéragir avec TypeScript, j'ai donc du trouver comment le compiler correctement afin que l'application marche sur tous les navigateurs. Pour ce faire, j'utilise simplement une libraire appelée *Browserify* qui permet de compiler tout le projet et de le stoquer dans un fichier unique.

## 2.3 Implémentation des bitboards

L'implémentation des bitboards est résolument simple, grâce à une théorie claire et des opérations extrêmement simples. J'ai tout de même rencontré certains problèmes liés à des spécificités de JavaScript. Mais avant d'exposer ces problèmes, voyons l'implémentation.

Comme vu dans la théorie, je commence par initialiser les constantes comme `max_heights = [5, 12, 19, 26, 33, 40, 47]` et `directions = [1, 6, 7, 8]` (les directions sont les nombres "magiques" de la théorie). Je déclare aussi une variable *data* qui contiendra les bitboards des deux joueurs et une variable `heights = [0, 7, 14, 21, 28, 35, 42]` qui gardera en mémoire la hauteur de chaque colonne. On prépare aussi une variable `counter = 0` qui permet de compter les coups joués (qui est aussi pratique, car s'il est pair, c'est au tour de l'IA sinon, c'est au tour du joueur). Enfin, on crée un tableau *moves* qui gardera tous les coups en mémoire.

Voyons ensemble, par exemple, la fonction *move* en TypeScript :

```
1 move(column: number): void {
2     let move = new Long(1).shl(this.heights[column]); // On recupere la hauteur stockee
          dans heights avec la colonne et on shift le 1 ce nombre de fois a gauche
3     this.heights[column]++; // On incremente la hauteur de la colonne
4     this.data[this.counter & 1] = this.data[this.counter & 1].xor(move); // On joue le
          coup sur le board
5     this.moves[this.counter++] = column; // On enregistre le coup
6 }
```

(1) On voit `column : number`, cela permet d'indiquer que la variable *column* est un nombre. On voit aussi *: void*, ce qui indique que cette fonction ne retourne rien.

(2) *shl* est l'abréviation de *shift left*.

Un problème que je n'avais néanmoins pas anticipé, est que JavaScript stock ses nombres dans un espace qui fait $2^{52}$ bytes (ce qui limite leur taille à maximum $2^{52}$) alors que dans certains cas, notre représentation du bitboard peut dépasser ce nombre. Il a donc fallu que j'utilise une librairie appelée *long* qui permet elle de gérer les nombres jusqu'à $2^{64}$, ce qui est suffisant.

## 2.4 Implémentation de minimax

Avant d'implémenter un algorithme connu, j'ai d'abord essayé beaucoup de choses, certaines marchaient mais étaient beaucoup trop lentes, certaines ne marchaient simplement pas. Après avoir essayé toutes ces choses, je me suis renseigné sur les algorithmes existants et ai dessidé d'utiliser celui-ci car il me semblait être le plus adapté. Mais avant de l'implémenter, voyons comment il fonctionne.

Pour utiliser l'algorithme minimax sur un jeu, il faut que ce jeu soit *un jeu non-coopératif synchrone à information complète opposant deux joueurs, [...] à somme nulle*[2]. En clair, un jeu ou deux joueurs s'affrontent, chacun leur tour, où chaque joueur connait toutes les informations et où la somme de la perte et des gains des deux joueurs est nulle. Hors le Puissance 4 remplit toutes ces conditions. En théorie, l'algorithme fonctionne en 5 étapes :

1. On génère tout l'arbre du jeu (voir ci-dessous), jusqu'en tout en bas aux positions finales (victoire ou nulle)
2. On applique une *heuristique* à chaque position finale
3. On détermine la valeur des nodes au dessus grâce à la valeur des nodes d'en dessous.
4. On fait remonter ces valeurs d'étage en étage jusqu'en haut
5. On choisi le chemin avec la plus grande valeur

Pour les représenter, j'utiliserai des versions simplifiées (comme par exemple des plateaux de morpion), pour que les shémas restent lisibles.

2. Théorème du minimax de von Neumann, *Théorème du minimax de von Neumann — Wikipédia.*

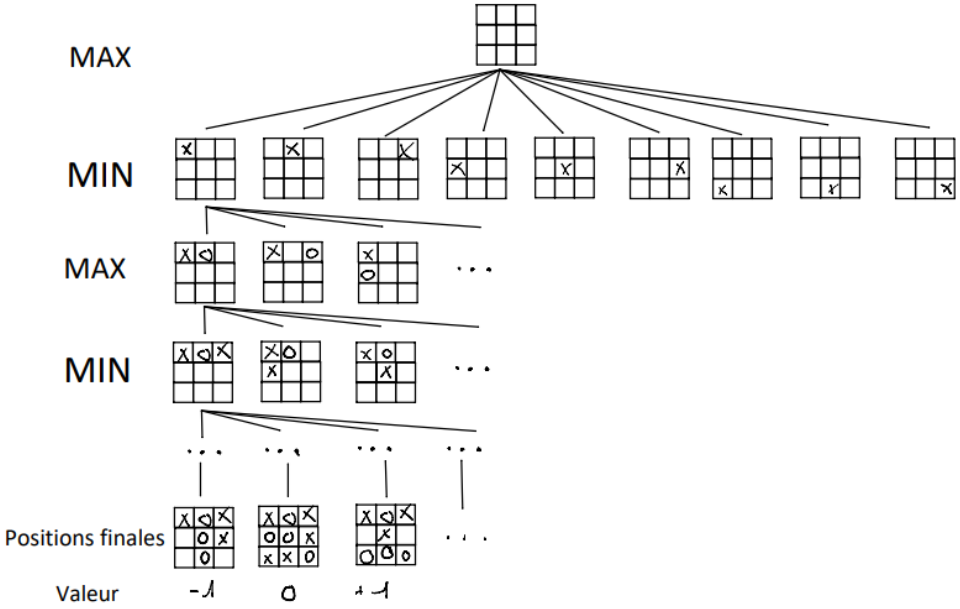


FIGURE 14 – Un arbre de recherche (partiel) du morpion

On voit que sur les coups du joueur $X$, on cherche la valeur maximale (max) et sur les coups de l'autre joueur, on cherche la valeur minimale (min). D'où le nom de l'algorithme : **minimax**. Une application sur un jeu encore plus trivial pourrait ressembler à ceci :

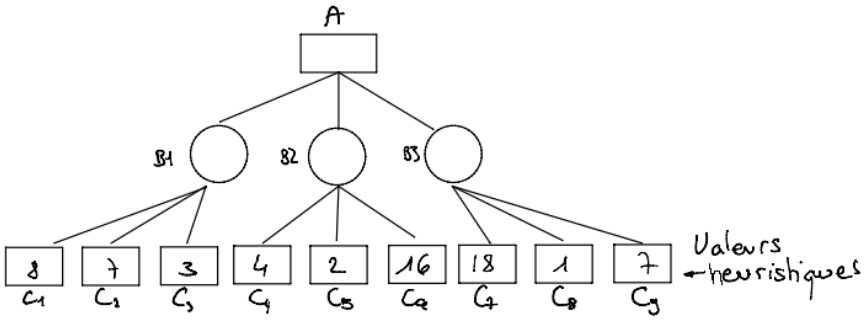


FIGURE 15 – Exemple de mininax

On voit que ce jeu se termine après le deuxième coup. Maintenant, pour déterminer le meilleur coup pour le premier joueur, il faut "remonter" les valeurs jusqu'en haut pour voir quel chemin est le plus rentable. La dernière ligne est une ligne impaire (donc min), on cherche donc la plus petite valeur entre les 3 de chaque node. Pour B1, ça sera 3, pour B2, ça sera 2 et pour B1, ça sera 1. Maintenant pour A on cherche le maximum entre les 3 en dessous, c'est-à-dire 3. Donc le meilleur chemin est A -> B1 -> C3.

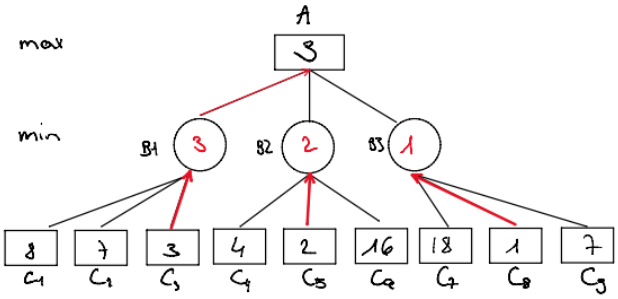


FIGURE 16 – "Remontée" des valeurs

Cet algorithme est en théorie infaillible. En effet, si l'on atteint la fin de l'arbre il suffit de jouer le meilleur coups à chaque fois pour gagner. Mais calculer toutes ces positions à chaque coups est beaucoup trop long. En effet, une estimation du nombre de positions possibles au Puissance 4 est d'environ $1.6 \cdot 10^{13}$

positions. Même si on a estimé qu'avec les bitboards on pouvait calculer environ 100 millions de positions par seconde (en réalité probablement beaucoup moins), il faudrait au minimum 44h et 25 minutes pour calculer 1 coup [3]. Il a donc fallu trouver des moyens pour palier à ce problème.

## 2.5 Améliorations de l'algorithme

Pour améliorer et rendre plus viable l'algorithme, j'y ai appliqué 3 modifications. Les deux premières pour gagner du temps, et la dernière pour le rendre plus "précis".

### La profondeur

La première chose que j'ai fait, c'est d'imposer à l'IA une limite de profondeur. C'est à dire qu'au lieu d'aller jusqu'au positions finales, elle s'arrêtera, au plus bas, à la profondeur donnée. Cela permet de raccourcir énormément le temps de recherche mais pose un problème fondamentale par rapport au fonctionnement de l'algorithme. En effet, pour fonctioner, il a besoin de valeurs sur les nodes les plus basses pour pouvoir les remonter. Pour palier à ce problème j'ai, arbitrairement, décidé d'attribuer une valeur de 0 au nodes qui n'étaient pas une victoire.

Avec cette seule modification, l'IA peut jouer en quelques secondes mais est plutôt mauvaise. Elle n'est très bonne que sur les fins de parties (car elle "voit" les positions finales).

### L'élagage alpha-bêta

Pour gagner encore plus de temps, j'ai utilisé une autre méthode, qui permet de "couper" des branches qu'on sait inutiles de l'arbre.

Il y a 2 types de coupures : Alpha et Bêta. Prenons l'exemple d'une coupure Alpha :

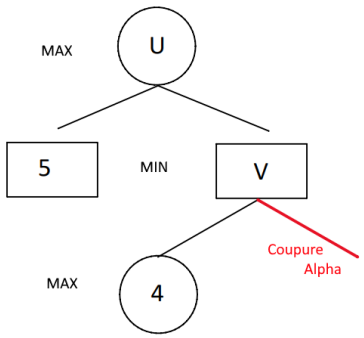


FIGURE 17 – Exemple coupure Alpha

Le premier enfant de la deuxième ligne vaut 5, donc logiquement U vaut au minimum 5. Hors le premier enfant de V vaut 4 donc V vaudra au maximum 4 et peut donc être ignoré.

La coupure bêta fonctionne de manière homologue sur un noeud min :

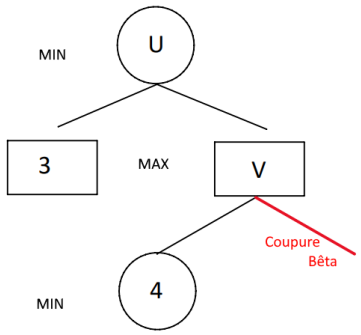


FIGURE 18 – Exemple coupure Beta

L'efficacité de ces coupures est un peu aléatoire : si par chance les valeurs sont rangée dans un ordre tel qu'on peut couper 99% de l'arbre, on est très content. Mais d'un autre côté, si toutes les valeurs sont rangées de sorte qu'on ne peut rien couper, elle ne sert à rien. C'est pourquoi il peut être rentable, avec cette méthode, de mélanger les positions de manière aléatoire avant d'appliquer l'algorithme afin de minimiser ces cas.

3. Le calcul est le suivant : $1.6 \cdot 10^{13} \div 10^{8} = 1.6 \cdot 10^{5}$ secondes. Soit 44h 26min et 40sec. A noter que cette valeur serait pour un ordinateur entièrement consacré à ce calcul, avec un language plus optimisé que le JavaScript.

### L'heuristique

La dernière amélioration que j'ai apporté à cet algorithme est une fonction d'évaluation des positions. En effet, au lieu de simplement donner une valeur de 0 aux positions non terminées, on va donner une valeur à chaque coups. Pour déterminer la valeur à donner j'ai décidé d'une formule (relativement simple mais efficace) qui est la suivante :

```
1     function evaluate(position) {
2         if (position.isWin(IA)) return Infinity;
3         else if (position.isWin(Player)) return -Infinity;
4         else return 100 * (position.has3InARow(IA) - position.has3InARow(Player)) + 3 * (
              position.has2InARow(IA) - position.has2InARow(Player));
5     }
```

Ce que fait cette fonction est assez simple. Si la position est une victoire de l'IA elle attribue à la node une très grande valeur (l'infini). Si c'est le joueur qui gagne cette position, elle attribue l'infini négatif. Enfin, si ni l'un ni l'autre ne gagne cette position, elle compte le nombre de fois où 3 (et 2) pièces sont alignée chez l'IA et multiplie ce nombre par un poids. Elle prend ce même nombre pour le joueur et la différence est la valeur de la node. En clair une valeur négative signifie un avantage pour le joueur et une valeur positive une avantage pour l'IA. Plus la valeur est grande, plus l'avantage est grand.

Toutes ces améliorations font en sorte que l'IA réponde en quelques secondes et soit relativement difficile à battre.

# 3 Conclusion

## 3.1 Atteinte des objectifs fixés

## 3.2 Possibilités d'améliorations

## 3.3 Remerciements

# 4 Bibliographie

## 4.1 Lexique

**Implémentation**: Réalisation (d'un produit informatique) à partir de documents, mise en œuvre, développement.

**Entier long**: En programmation informatique, un entier long (en anglais long integer) est un type de données qui représente un nombre entier pouvant prendre plus de place sur une même machine qu'un entier normal.

**RAM**: En anglais *Random Access Memory* et en français **mémoire vive**, est la mémoire utilisée pendant le traitement de donnée. En opposition à la mémoire dite morte qui ne sert qu'à stoquer des données.

**AND**: L'opérateur *AND* (ou opérateur ET en français) prend deux nombres en entrée et retourne VRAI (ou dans notre cas 1) si les deux nombres sont les mêmes. Sinon il retourne FAUX (0 pour nous).

**XOR**: L'opérateur *XOR* (appelé OU exclusif en français) prend deux nombres en entrée et retourne VRAI si les deux nombres sont différents. Sinon il retourne FAUX.

**Dépendance**: Autres logiciels/programmes nécessaires à l'éxcécution d'un autre programme.

## 4.2 Livres

[1] Stuart J. Russell, Peter Norvig et Ming-wei Chang. *Artificial intelligence a modern approach*. Pearson Education Limited, 2021.

## 4.3 Sites web

[2] Socket.IO's devs. *Socket.IO documentation*. En ligne, consulté le 06.04.2021. 2021. url : `https://socket.io/docs/v4`.

[3] Jacques Haiech. *Parcourir l'histoire de l'intelligence artificielle, pour mieux la définir et la comprendre*. En ligne, consulté le 06.05.2021. Oct. 2020. url : `https://www.medecinesciences.org/en/articles/medsci/full_html/2020/08/msc200112/msc200112.html`.

[4] Ye Qianqian et Evelyn Masso. *p5.js Reference*. En ligne, consulté le 09.04.2021. 2021. url : `https://p5js.org/reference/`.

[5] Node.js's team. *Node.js v16.6.1 documentation*. En ligne, consulté le 02.04.2021. 2021. url : `https://nodejs.org/api/`.

[6] Théorème du minimax de von Neumann. *Théorème du minimax de von Neumann — Wikipédia*. En ligne, consulté le 06.10.2021. 2021. url : `https://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_du_minimax_de_von_Neumann`.

# 5 Annexes