

# Travail de maturité - Puissance4IA

Torrenté Florian

30 septembre 2021

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Problématique et objectifs . . . . .	2
1.2	Description et règles du jeu . . . . .	2
1.3	L'intelligence artificielle . . . . .	2
1.4	La structure des données . . . . .	3
1.5	Application des bitboards pour le puissance 4 . . . . .	4
<b>2</b>	<b>Déroulement du travail</b>	<b>7</b>
2.1	Structure et technologies . . . . .	7
2.2	Implémentation des bitboards . . . . .	7
2.3	Implémentation du puissance 4 . . . . .	8
2.4	Implémentation de minimax . . . . .	8
2.5	Améliorations de l'algorithme . . . . .	8
<b>3</b>	<b>Conclusion</b>	<b>8</b>
3.1	Atteinte des objectifs fixés . . . . .	8
3.2	Possibilités d'améliorations . . . . .	8
3.3	Remerciements . . . . .	8
<b>4</b>	<b>Bibliographie</b>	<b>9</b>
4.1	Lexique . . . . .	9
4.2	Livres . . . . .	9
4.3	Sites web . . . . .	9
<b>5</b>	<b>Annexes</b>	<b>9</b>

# 1 Introduction

## 1.1 Problématique et objectifs

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.

## 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 connaît :

- 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" <sup>1</sup>

## 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)

1 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 supplémentaire
+-----+		
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 :

.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	0	.	.	.
.	.	.	X	X	.	.
.	.	0	X	0	.	.
-----						
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
. . . 0 . . .	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
. . 0 X 0 . .	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

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

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

```

... 00000000 00000000 0000010 0000011 0000000 0000000 0000000 // Joueur 1
... 00000000 00000000 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 exclusif, (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 écrit 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`.

```

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

```

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.

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

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.

```
00000000 00000000 00000000 00000001 00000000 00000000 00000000 // Notre nombre
^ 00000000 00000000 00000000 00000000 00000000 00000000 00000000 // Le bitboard avant le coup
```

Diagram illustrating the transformation of a 6x6 grid of points (dots) into a 6x6 grid of points (dots) after a transformation. The transformation is indicated by an arrow pointing from the left grid to the right grid. The right grid shows a point labeled '0' at the intersection of the 3rd row and 4th column. Below the grids, the labels 'Avant' and 'Après' are present, corresponding to the left and right grids respectively. The x-axis is labeled with indices 0, 1, 2, 3, 4, 5, 6.

## Est-ce qu'un joueur à gagné ?

Reprenons la représentation du bitboard :

6	13	20	27	34	41	48	55	62	Ligne supplémentaire
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 :

```

      . . . . .
      . . . . .
      . . . 0 . .
      . . . 0 . .
      . . . 0 X .
      . . . 0 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 0

^ 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  $4 \times 7 = 28$  opérations. Les ordinateurs actuels effectuent environ 3 milliards d'opérations par seconde, on peut donc évaluer environ **100 millions de positions par seconde**.

## 2 D roulement du travail

### 2.1 Structure et technologies

Le projet est divis  en 2 parties principales : le *client* et le *serveur*. Le premier contient tout ce que l'utilisateur verra (la page internet, le jeu) et g rera toutes les interactions avec ce dernier. Toutes les informations du client sont ensuite trait es par le serveur   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 tr s l g re et donc utilisable sur tous les appareils. Le probl me est que la charge de calcul est enti rement sur le serveur, ce qui peut poser probl me si beaucoup d'utilisateurs voulaient affronter l'IA.

Pour tout le projet, j'utilise NPM (pour "*Node Package Manager*") qui permet de g rer les d pendances de mon projet simplement. Toutes les informations pour NPM sont contenues dans un fichier *package.json*. On y trouve : la description du projet (nom, description, version, auteur, etc...), les diff rents scripts (raccourcis pour lancer, g n rer le projet) et les d pendances.

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 d cid  de ne pas utiliser le langage CSS, mais de passer par SASS (Syntactically Awesome Stylesheets). Ce langage permet d' crire un code plus visuel et pratique qui sera ensuite compil  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 g re de mani re  l gante et l g re toutes les actions de dessins et les interactions avec l'utilisateur. Pour ce projet, j'ai aussi d cid  d'utiliser *TypeScript* qui est un langage (qui se compile en JavaScript) permettant d'avoir un code plus pr visible et d'attraper les erreurs plus t t.

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

Le dossier serveur quant   lui contient :

1. Le dossier src
2. package.json

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

- a. Bitboard.ts
- b. AI.ts
- c. index.ts

(a) Le fichier Bitboard.ts (*ts* est l'extension des fichiers contenant du code TypeScript) qui contient l'impl mentation 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 pr c demment, toutes les informations pour le gestionnaire de paquets NPM.

### 2.2 Impl mentation des bitboards

L'impl mentation des bitboards est r solument simple, gr ce   une th orie claire et des op rations extr mement simples. J'ai tout de m me rencontr  certains probl mes li s   des sp cificit s de JavaScript. Mais avant d'exposer ces probl mes, voyons l'impl mentation.

Comme vu dans la th orie, 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 th orie). Je d clare 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 m moire la hauteur de chaque colonne. On pr pare aussi une variable `counter = 0` qui permet de compter les coups jou s (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 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
3     dans heights avec la colonne et on shift le 1 ce nombre de fois a gauche
4     this.heights[column]++; // On incremente la hauteur de la colonne
5     this.data[this.counter & 1] = this.data[this.counter & 1].xor(move); // On joue le
6     coup sur le board
7     this.moves[this.counter++] = column; // On enregistre le coup
8 }
```

(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.

## 2.3 Implémentation du puissance 4

## 2.4 Implémentation de minimax

## 2.5 Améliorations de l'algorithme

# 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 stocker 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'exécution 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*. 2021. URL : <https://socket.io/docs/v4>.
- [3] Jacques HAIECH. *Parcourir l'histoire de l'intelligence artificielle, pour mieux la définir et la comprendre*. Oct. 2020. URL : [https://www.medecinesciences.org/en/articles/medsci/full\\_html/2020/08/msc200112/msc200112.html](https://www.medecinesciences.org/en/articles/medsci/full_html/2020/08/msc200112/msc200112.html).
- [4] Ye QIANQIAN et Evelyn MASSO. *p5.js Reference*. 2021. URL : <https://p5js.org/reference/>.
- [5] Node.js's TEAM. *Node.js v16.6.1 documentation*. 2021. URL : <https://nodejs.org/api/>.

## 5 Annexes