Travail de maturité - Puissance4IA

Torrenté Florian

28 septembre 2021

Table des matières

1 Introduction

1.1 Problématique et objectifs

Dans quelle mesure l'imnplémentation d'un 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 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 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 optimisisée 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 puissance4 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

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.*

```
... 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, ect... 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 resultat 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 à gagné.

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

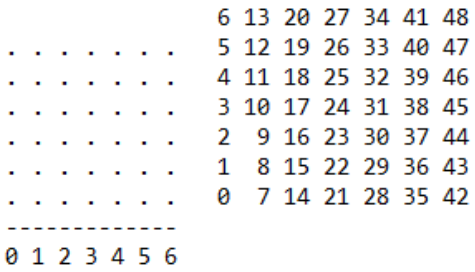

FIGURE 8 – Début de partie

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}. A 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 stoque 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).

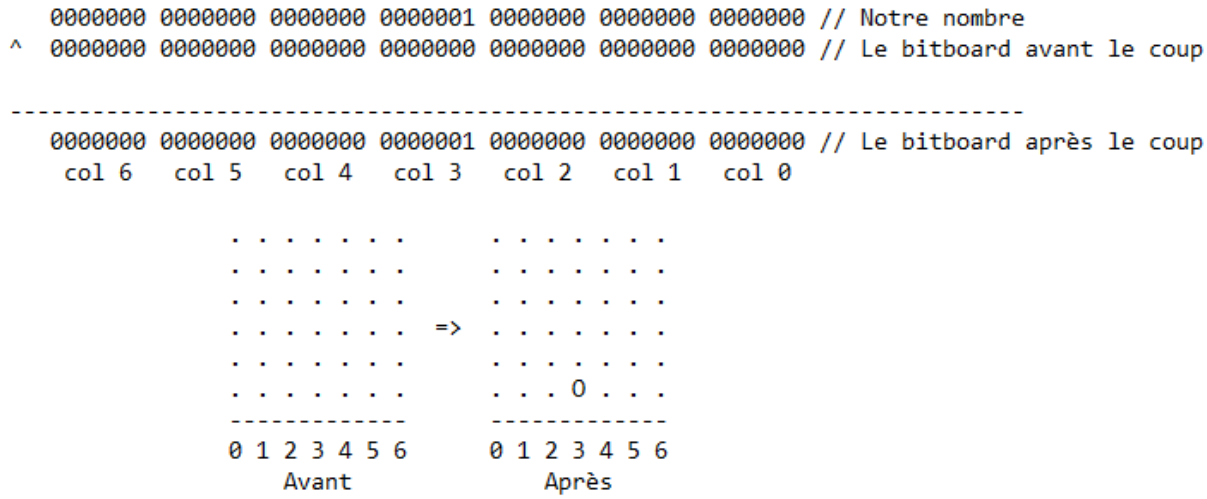

FIGURE 9 – Exemple de la fonction makeMove

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 patterne là-dessous est simple : on prend la position la plus à gauche et on ajoute 7 à chaque fois.

Verticallement 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 si 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 copier 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 Mise en place

Pour commencer, il a fallu choisir la structure et les différentes technologies que je voulais utiliser pour ce projet.

La structure du projet

Les technologies

2.2 Implémentation des bitboards

2.3 Implémentation du puissance 4

2.4 Implémentation de m inimax

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 stoquer des données.

AND: L'opérateur *AND* (ou opérateur ET en français) prend deux nombre 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.

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`.

[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