Travail de maturité - Puissance4IA

Anonyme

28 mars 2022

Table des matières

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 a l'air d'être un sujet très prometteur pour le futur. 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 lignes). Chaque joueur possède des 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.

Ces différentes conclusions permettent d'utiliser des algorithmes comme *Minimax*, qui sont conçus pour minimiser les pertes. Pour ce faire on a besoin des différentes informations du jeu, en effet, si de l'aléatoire intervenait dans les règles du jeu, il serait impossible d'implémenter ce genre d'algorithmes.

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. On peut citer, par exemple, les intelligences artificielles pour la navigation (comme GoogleMaps ou Waze), la reconnaissance faciale, la correction et la traduction de texte et autre. C'est un domaine de l'informatique qui fait désormais partie de nos vies quotidiennes, mais reste assez peu connu par le grand public.

Maintenant, nous savons pourquoi le domaine de l'IA est intéressant et important, nous avons toujours besoin de savoir précisément *ce que c'est*. On pourrait simplement dire : "Eh bien, ce sont des programmes intelligents", mais je pense qu'il est important de bien définir ce qu'est l'intelligence et par extension l'intelligence artificielle avant de se lancer dans sa création. Une cohérente pour moi serait celle-ci : "L'intelligence artificielle a pour objectif de construire des dispositifs simulant les processus cognitifs humains" [1]. En clair, des programmes qui permettent de résoudre des problèmes qui étaient, avant, exclusivement résoluble avec une intervention humaine.

1.4 La structure des données

Pour implémenter un algorithme, il faut définir une structure de donnée très légère et optimisée afin de pouvoir calculer des milliers de positions de partie en quelques secondes. Pour comprendre cette structure de donnée, nous allons devoir plonger dans le fonctionnement des ordinateurs. Nous allons explorer : les *bits*, leur manipulation et enfin les *bitboards*.

Les bits Dans les ordinateurs, toutes les informations sont stockées sous forme de bits [2]. On peut voir un bit comme une boite contenant soit 0 soit 1. L'utilisation des bits est assez naturelle en informatique, car, conventionnellement, 0 est éteint et 1 allumé. Maintenant, on a cette boite, mais comment faire si on veut stocker autre chose que 0 ou 1 ? Avant de répondre à cette question, il nous faut examiner notre façon de compter. Dans la vie de tous les jours, nous comptons utilisons les nombres en base 10 [3]. Cela veut dire qu'on utilise 10 chiffres différents pour écrire tous les nombres (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). Pour compter, on incrémente la boite la plus à droite jusqu'à 9. Une fois à neuf, pour incrémenter encore, on incrémente la boite de gauche et on transforme le 9 en 0.

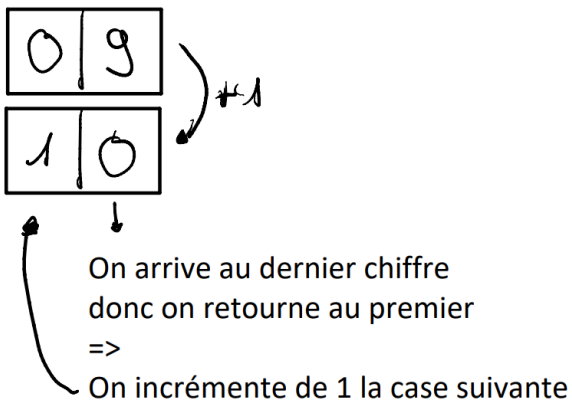


FIGURE 2 – Comptage en base 10

Revenons à notre question : pour stocker autre chose que 0 ou 1, on utilise plusieurs bits pour compter en base 2. Au lieu d'avoir 10 chiffres, on n'en a que 2 : 0 et 1. L'écriture du nombre 3 en base 2 est donc : 11. Pour passer de base 2 à base 10, il faut numéroter nos boites de droite à gauche (en partant de 0) et appliquer cette formule (on notera B_i le chiffre dans la boite i et n le nombre de boite) : $\sum_{i=0}^{n} B_i \cdot 2^i$. Par exemple 11 (en base 2) $= 1 \cdot 2^0 + 1 \cdot 2^1 = 3$.

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

1. HAIECH, *Parcourir l'histoire de l'intelligence artificielle, pour mieux la définir et la comprendre.*
2. De l'anglais *binary digit*
3. Ou système décimal

en haut et deux colonnes à droite pour éviter les erreurs de saut lors des opérations pour la vérification de victoire. Cette représentation est appelée *bitboard*

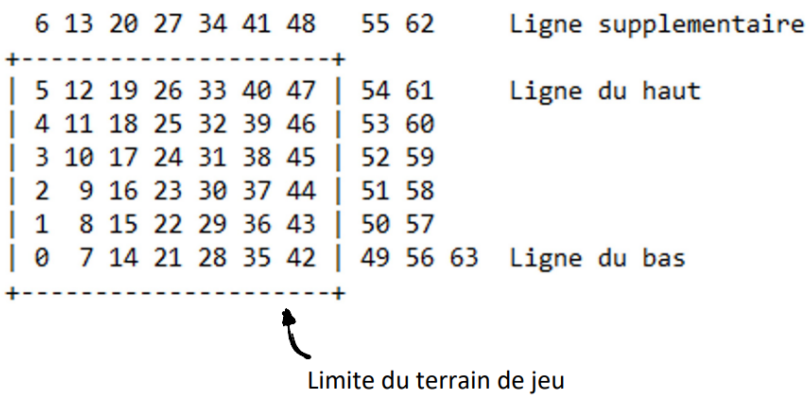


FIGURE 3 – Représentation du bitboard

Les nombres indiquent la position (l'*index*) dans la représentation binaire d'un long. Par exemple, le 25 représente le 25ème bit de ce nombre. En clair, notre bitboard est simplement un nombre, qu'on représente en base 2, et qu'on place en 2 dimensions pour représenter le terrain de jeu.

Comme le jeu est joué par deux joueurs, on utilisera un bitboard par joueur. Prenons un exemple en représentant cette position :

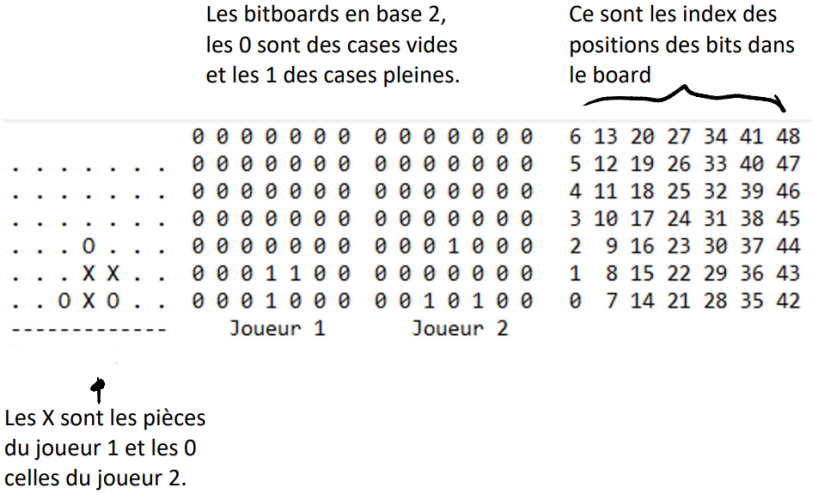


FIGURE 4 – Partie en cours avec les bitboards

On peut aussi le voir "à plat", en regroupant les bits par paquets de 7 (ce qui représente une colonne entière, c'est-à-dire avec la ligne supplémentaire) 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 représentation du board sera très pratique par la suite pour faire les opérations sur ces bits (les opérations seront plus simples à visualiser).

Cette manière de représenter les états du jeu est la clé à la vitesse de la vitesse de l'algorithme. En effet, cette représentation permet en utilisant seulement des opérations de base, comme nous allons le voir ci-dessous, 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

Toutes ces différentes opérations vont être utilisées directement dans la partie suivante pour, entre autres, jouer un coup et vérifier si un joueur a gagné.

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
```

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`).

```
   0000000 0000000 0000000 0000001 0000000 0000000 0000000 // Notre nombre
^  0000000 0000000 0000000 0000000 0000000 0000000 0000000 // Le bitboard avant le coup

-------------------------------------------------------------------------------
   0000000 0000000 0000000 0000001 0000000 0000000 0000000 // Le bitboard après le coup
    col 6   col 5   col 4   col 3   col 2   col 1   col 0

              . . . . . . .       . . . . . . .
              . . . . . . .       . . . . . . .
              . . . . . . .       . . . . . . .
              . . . . . . .   =>  . . . . . . .
              . . . . . . .       . . . . . . .
              . . . . . . .       . . . O . . .
              -------------       -------------
              0 1 2 3 4 5 6       0 1 2 3 4 5 6
                  Avant               Après
```

FIGURE 9 – Exemple de la fonction makeMove

Est-ce qu'un joueur à gagné ?

Reprenons la représentation du bitboard :

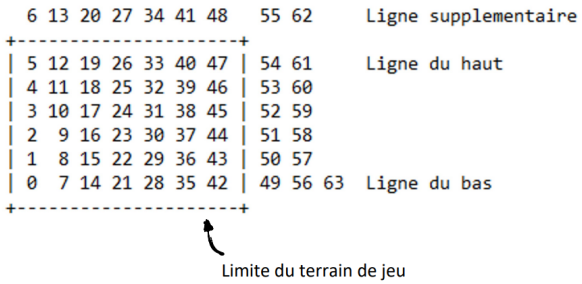


FIGURE 10 – Représentation du bitboard

Pour savoir si 4 pièces sont alignées horizontalement, on doit vérifier par exemple si les positions 11, 18, 25 et 32 sont toutes des 1. De la même manière, on pourrait vérifier les positions 21, 28, 35 et 42.

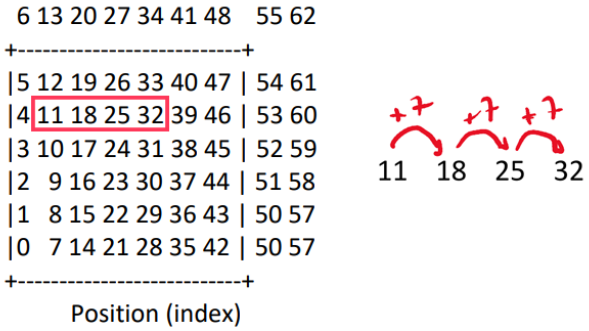


FIGURE 11 – Exemple puissance 4 horizontal

En partant de n'importe quelle position et en sautant de 7 en 7, on peut vérifier s'il y a 4 pièces alignées. De manière similaire pour les puissances 4 verticaux et diagonaux :

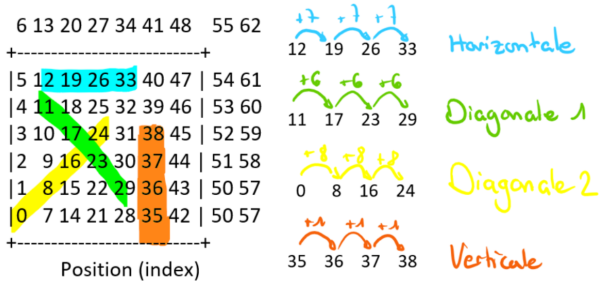


FIGURE 12 – Les chiffres "magiques"

Ces chiffres "magiques" (magiques parce que grâce à ces 4 chiffres, on peut vérifier tous les alignements possibles sur le board) sont donc : 1, 6, 7, 8. Maintenant, en utilisant 2 opérateurs vus précédemment (AND et le *right shift*) on va pouvoir vérifier toutes les possibilités très rapidement. Avec les shifts, on va aligner les positions correspondantes (en shiftant les chiffres magiques 1x, 2x et 3x). Une fois alignés, on les combine avec l'opérateur AND. Si le résultat est différent de 0, alors il y a 4 pièces alignées.

FIGURE 13 – 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 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 :

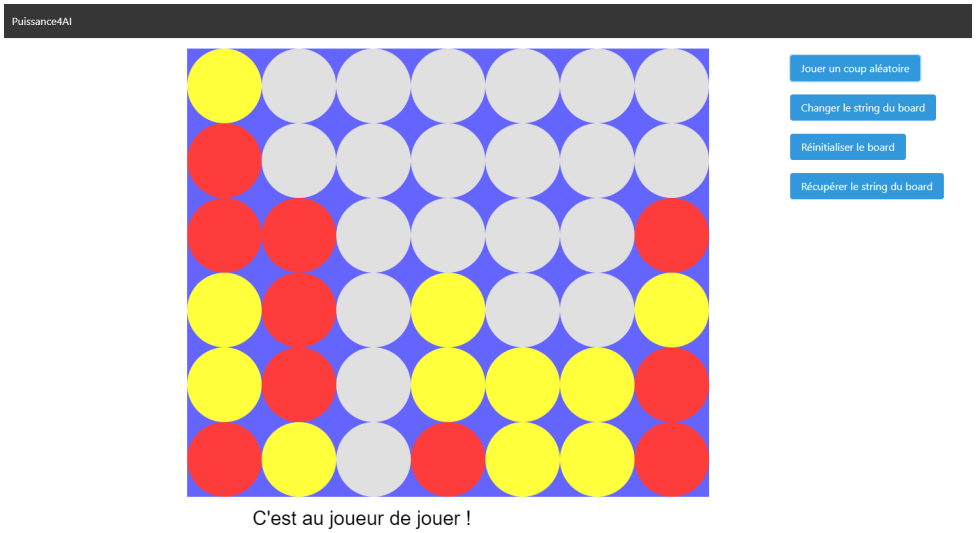


FIGURE 14 – Première version du client

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 :

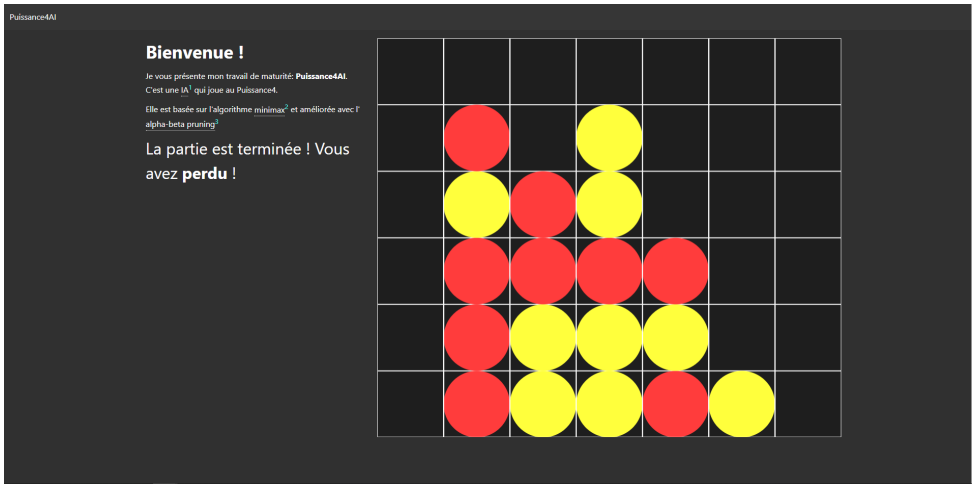


FIGURE 15 – Deuxième version du client

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 rencontré 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 interagir avec TypeScript, j'ai donc dû 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 stocker 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 $\text{max\_heights} = [5, 12, 19, 26, 33, 40, 47]$ et $\text{directions} = [1, 6, 7, 8]$ (les directions sont les chiffres "magiques" de la théorie). Je déclare aussi une variable *data* qui contiendra les bitboards des deux joueurs et une variable $\text{heights} = [0, 7, 14, 21, 28, 35, 42]$ qui gardera en mémoire la hauteur de chaque colonne. On prépare aussi une variable $\text{counter} = 0$ qui permet de compter les coups joués (qui est aussi pratique, car s'il est pair, c'est au tour du joueur 1 sinon, c'est au tour du joueur 2). 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}), mais lors de l'utilisation d'opérateurs sur les bits (comme le AND et les shifts), ces nombres sont limités à 2^{32}, 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*[4]. En clair, un jeu ou deux joueurs s'affrontent, chacun leur tour, où chaque joueur connaît toutes les informations et où la somme de la perte et des gains des deux joueurs est nulle. Le Puissance 4 remplit toutes ces conditions tout comme les échecs (trop compliqués) et le morpion (trop simple). 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*[5] 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.

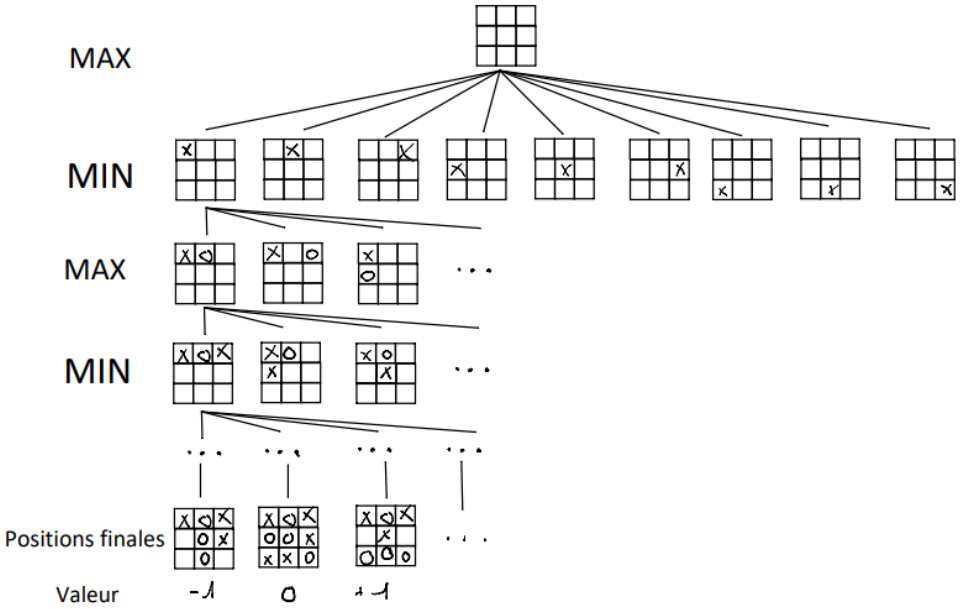


FIGURE 16 – Un arbre de recherche (partiel) du morpion

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 :

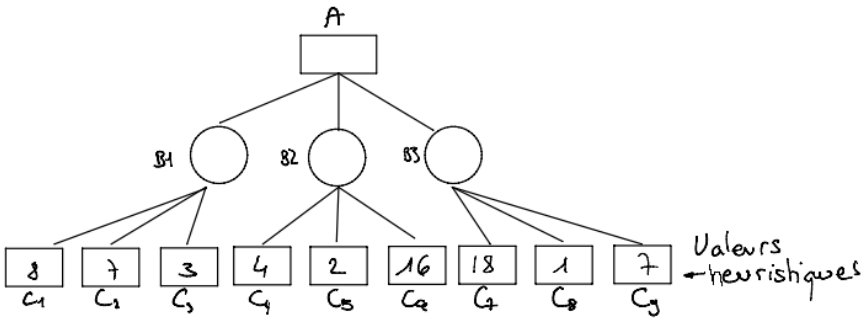


FIGURE 17 – Exemple de mininax

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

4. THÉORÈME DU MINIMAX DE VON NEUMANN, *Théorème du minimax de von Neumann — Wikipédia.*
5. Une heuristique est un nombre qui fournit, pas nécessairement de manière excacte, un score à une position.

les 3 de chaque node. Pour B1, ça sera 3, pour B2, ça sera 2 et pour B3, 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.

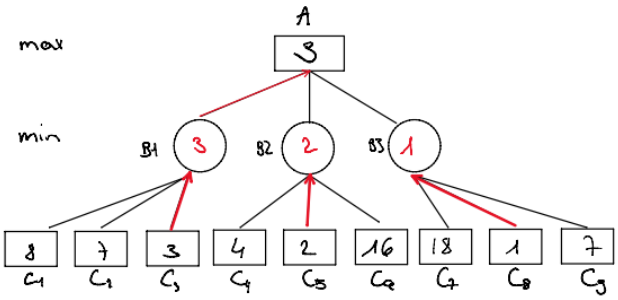


FIGURE 18 – "Remontée" des valeurs

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 [6]. Il a donc fallu trouver des moyens pour palier à ce problème.

2.5 Optimisations 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. La coupure Alpha est une coupure sur un noeud max, et la Bêta est une coupure sur un noeud min. Prenons l'exemple d'une coupure Alpha :

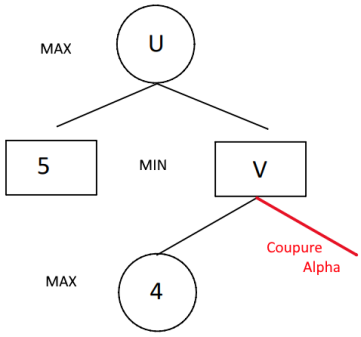


FIGURE 19 – Exemple coupure Alpha

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 :

6. 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.

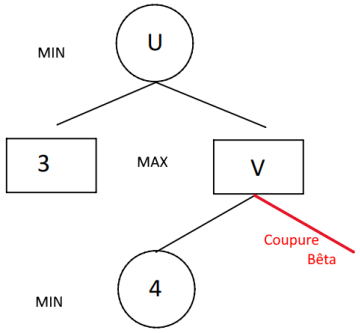


FIGURE 20 – Exemple coupure Beta

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.

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 (maximum 2 secondes) et soit relativement difficile à battre (je la bas environ une fois sur dix).

3 Conclusion

3.1 Atteinte des objectifs fixés

Mes objectifs 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

Prenons point par point. Le premier point, je pense l'avoir bien atteint, en effet, maintenant je comprends en profondeur le fonctionnement d'une intelligence artificielle, ses enjeux et son potentiel. Je suis plus à même de comprendre les situations où ces outils sont applicables et les situations où elles ne le sont pas. Je me rends aussi compte qu'une grande partie de ce domaine me reste inconnue : dans ce travail je n'ai étudié que les intelligences dites faibles, mais il existe de nos jours des intelligence dites fortes, basée sur du *deep learning*[7]

Je pense avoir aussi rempli mon second objectif qui était de développer un algorithme moi-même. Au cours de ce travail, j'ai développé l'algorithme Minimax, et au passage pu me rendre compte de ses limites.

Pour le troisième objectif, j'ai pu me rendre compte des limites de l'algorithme Minimax, comme les limites liées au temps de calcul. J'ai pu essayer d'améliorer ce problème en implémentant différentes améliorations : l'*énalage Alpha-Beta*, la limite de profondeur et une heuristique.

7. L'apprentissage profond (ou Deep Learning en anglais), est un ensemble de méthodes et de théorie permettant de créer une intelligence qui "apprend" à partir de gros paquets de données.

Le dernier point est malheureusement partiellement un échec, en effet, actuellement si je limite la profondeur à 12 (l'IA voit dont 6 coups de chaque joueurs à l'avance) j'arrive à battre l'ordinateur environ une fois sur dix. Je pense que c'est en partie du au manque de profondeur et au potentiel manque de précision de l'heuristique.

3.2 Possibilités d'améliorations

Au cours de mon travail (et de ma rédaction), j'ai pu imaginer d'autres façons d'améliorer encore la précision et la vitesse de mon IA.

Profondeur progressive Pour commencer, une profondeur fixe n'est pas très optimisée selon moi : le nombre de positions à évaluer dépend fortement des découpes (alpha et bêta) qui peuvent arriver pendant le calcul et si, par chance on, peut couper une grande partie de l'arbre, il serait peut-être intelligent d'explorer une profondeur en plus. L'idée serait de fixer un temps maximum pour le calcul d'un coup et d'explorer de plus en plus profond pendant le temps imparti. En se basant sur les résultats de la profondeur précédente ou pourrait calculer la suivante sans devoir recalculer tout l'arbre.

Du *multi-threading* ?

Un *thread* en informatique est un processus qui effectue des calculs dans le processeur. Classiquement, un programme utilise un thread, mais avec les nouvelles technologies et les besoins en puissance de calcul, le multi-threading devient plus commun. Au lieu d'utiliser seulement un thread pour un programme, on en utilise plusieurs, multipliant la puissance de calcul. Cette méthode qui paraît être une méthode miracle pose tout de même quelques problèmes comme, par exemple, la synchronisation de ces différents threads. Je n'ai pas implémenté cette solution parce que JavaScript ne gère pas le multi-threading et que les gains de cette méthode ne me paraissaient pas suffisants.

Les positions similaires

La dernière idée que j'ai eue serait d'éliminer les positions similaires, c'est-à-dire : s'il existe deux manières d'arriver à la même position, il est inutile d'évaluer les deux. Cette idée est par exemple utilisée aux échecs pour gagner du temps de calcul et je pense qu'elle aurait pu être implémentée pour le Puissance 4. Mais le manque de temps m'a empêché de creuser plus cette idée et donc de l'implémenter.

3.3 Mon avis

Au final, je suis un peu déçu par mon travail. En effet, je pensais réussir à créer une intelligence artificielle plus forte que cela, mais je n'y suis pas parvenu. Ce travail m'a énormément appris de choses sur moi-même, notamment sur ma gestion de ma motivation. Malgré tout ça, je reste très fier du travail que j'ai créé, c'est le plus long et le plus important que j'ai pu faire et cela me donne plus confiance en moi pour des projets futurs.

Je tiens à remercier mon professeur accompagnateur Monsieur Mettral pour ses précieux conseils ainsi que sa patience. Je voudrais aussi remercier mes relecteurs qui m'ont aider à augmenter la clarté de mon travail.

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

5.1 Comment utiliser le code ?

Afin d'utiliser mon projet, il vous faudra installer NodeJS sur votre machine. Pour ce faire, il suffit de se rendre ici : `https://nodejs.org/en/download/` et de télécharger la version correspondant à votre machine. Une fois installé, vous pouvez passer à l'étape suivante.

Une fois NodeJS installé, vous pouvez vous rendre sur le repo du projet : `https://github.com/Lukyrouge3/travail_maturite` et le cloner. Pour cloner un repo, vous pouvez utiliser git avec la commande : "git clone `https://github.com/Lukyrouge3/travail_maturite`" ou alors cliquer sur le bouton "Code" pui "Download ZIP" et ensuite l'extraire dans le dossier de votre choix.

Une fois le repo cloné, rendez-vous dans le dossier où vous l'avez extrait et ouvrez un terminal. Dans ce terminal, écrivez : "npm i && npm run dev". Cette commande va mettre à jour les dépendances du projet et le lancer.

Une fois l'application lancée il vous suffit de vous rendre sur `http://localhost:1234` et vous pourrez jouer au puissance 4 contre l'IA.

5.2 Site

Le site en ligne avec le projet est disponible ici : `https://bit.ly/3GjgNaJ`

5.3 Code

Tout le code de ce travail, ainsi que tous les documents de la rédaction (images, bibliographie et fichiers LaTeX), sont trouvable sur mon github ici : `https://bit.ly/3pAwdBE`

5.4 Journal de bord

Journal de bord 2021- 2022

Recherche

Mercredi 17.03.2021

Je commence mes recherches sur les IA et plus précisément les IA liées aux jeux. Je tombe sur des tas d'articles décrivant la démarches d'autres développeurs ayant créé leur propre IA pour le puissance 4 mais aussi pour les échecs.

Je commence à avoir une image plus claire de quel algorithme je veux utiliser : un algorithme **minimax**. Je tombe sur un livre qui à l'air d'être une référence en la matière : Artificial Intelligence A Modern Approach [1].

(Temps passé ~2h)

[1] Consulté le 17.03.2021

Samedi 28.03.2021 et dimanche 29.03.2021

Je me rends compte que je n'ai pas eu beaucoup le temps de me consacrer à mon TM et donc je me relance dedans. Je commence par réorganiser un peu mon git, j'ajoute un schéma conceptuel de l'interaction client-serveur, j'ajoute un readme que je compte tenir un peu à jour et je complète un peu mon plan. Je me dis qu'il serait aussi temps de me lancer dans mon journal de bord - *du coup, me voilà…* - Je continue mes recherches et je décide de toutes les technologies que je voudrais utiliser.

Je travaille un peu tard (jusqu'à 3-4h du matin) mais je suis content. Je pense que ma phase de recherche (certes un peu courte) est plus ou moins terminée. Je pense y revenir de temps à autre quand j'ai besoin de plus de renseignements mais je compte me lancer bientôt dans l'implémentation du jeu en lui-même dans un canvas.

Je me demande aussi si ce format de livre de bord convient…

(Temps passé ~6h)

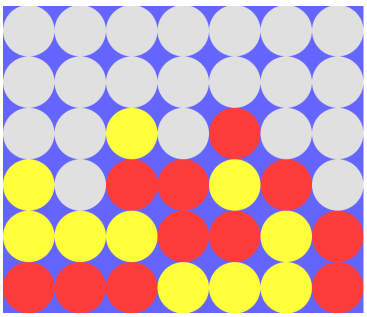
Implémentation du jeu

Lundi 29.03.2021

Aujourd'hui j'ai décidé de me lancer dans l'implémentation du puissance4 dans le navigateur. J'ai décidé d'utiliser p5js pour la gestion du canvas (ce qui me fait gagner énormément de temps) et j'utilise quand même typescript pour du js typé et un code plus lisible. J'ai décidé de stocker l'état de mon board comme un string (un peu comme aux échecs) pour que la communication entre serveur et client soit simple (un simple envoi de texte ducoup). Il sera sous cette forme : "pPpppP1/7/7/7/7/7/7" où "p" représente une pièce du joueur rouge, "P" une pièce du joueur jaune et les chiffres représentent des cases vides.

J'ai donc implémenté la fonction pour dessiner le board, une fonction pour générer le texte du board, une pour passer de string à array2d et la gestion des coups des joueurs.

Il manque une fonction pour vérifier si personne n'a gagné et une communication avec un serveur. Je compte utiliser Socket.IO pour cette communication.

 [2]

(Temps passé ~3h)

[2] Preview du board

Jeudi 01.04.2021 et vendredi 02.04.2021

Sur ces 2 jours j'ai terminé l'implémentation du jeu en soit, en ajoutant quelques fonctionnalités pratiques. J'ai créé les fonctions pour vérifier si un joueur à gagné ainsi que celle pour vérifier un match nul (board plein). Je me suis créé des boutons pour : jouer des coups aléatoires, réinitialiser le board, récupérer et modifier le string du board. Je me suis motivé à bien documenter et commenter mon code afin de mieux m'y retrouver et faciliter le maintien.
(Temps passé ~4h)

Samedi 10.04.2021 et dimanche 11.04.2021

Pendant ces deux jours j'ai rajouté un peu de style à ma page puis j'ai essayé de la servir de différentes manières. J'ai d'abord essayé d'utiliser expressJs pour implémenter un petit serveur HTTP et délivrer ma page, mais j'ai rencontré pas mal de problèmes car le typescript est mal supporté par express. Après plus de réflexions, j'ai décidé de ne pas utiliser express, car dans tous les cas, je compte ne délivrer qu'une seule page et donc un gros serveur http me parait un peu "overkill". J'ai temporairement laissé tombé et me suis concentré sur le socket. J'ai implémenté le côté client de mon socket et me suis arrêté là pour ce week-end.
(Temps passé ~15h (sur les 2 jours))

Dimanche 18.04.2021

Aujourd'hui, j'ai implémenté le côté serveur de mon socket et je me suis remis à la gestion de ces problèmes pour servir mon application. J'ai donc implémenté un tout petit serveur http qui envoie juste la page, son style et ses scripts. Le serveur tourne sur node-ts (une implémentation de nodejs pour se lancer directement en typescript sans autre compilation). Pour le client j'avais des problèmes parce que les navigateurs ne gèrent pas du tout typescript et j'ai donc dû trouver un moyen de le compiler pour l'utiliser sans problèmes. J'ai trouvé une librairie appelée Browserify qui faisait exactement ce dont j'avais besoin, c'est donc ce que j'ai utilisé. Donc maintenant pour utiliser le client, il faut le compiler avec typescript puis avec Browserify. Je ne sais pas s'il y a des moyens plus rapides, mais pour le moment cela me suffit.

Maintenant le serveur envoie des coups aléatoires quand c'est son tour et il faut donc que je me lance dans l'implémentation de l'IA.
(Temps passé ~4h30)

Implémentation de l'IA

Lundi 19.04.2021

J'ai implémenté un bitboard pour permettre des opérations beaucoup plus rapides afin de pouvoir en faire plusieurs millions pour trouver les meilleurs coups.

J'ai aussi commencé à tester des implémentations de l'IA, mais sans succès pour le moment.

(Temps passé ~6h)

Jeudi 22.04 et vendredi 23.04.2021

J'ai essayé encore beaucoup de choses pour faire marcher mon IA, j'ai créer des "Leafs" qui contiennent les feuilles du niveau en dessous. Cette méthode reste rapide bien que pas encore optimisée. Je rencontre tout de même beaucoup de problèmes dans le calcul du score qui ne reflète jamais la réalité et je rencontre aussi des OutOfMemoryError (ce qui veut dire que j'utilise trop de RAM et suis donc limité à une profondeur d'environ 7).

Je commence à me rendre compte que mon implémentation est trop lourde et je n'arrive toujours pas à la debug comme je voudrai. Je commence à penser à repartir de zéro pour mon implémentation du minimax,
(Temps passé sur les 2 jours ~22h)

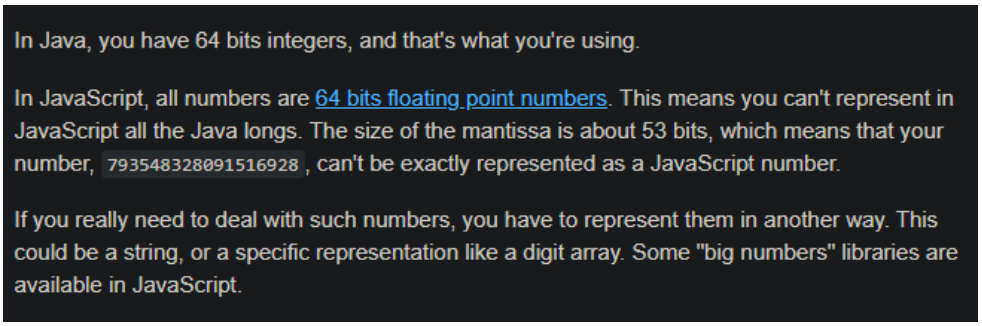
Mardi 27.04.2021

Aujourd'hui j'ai encore essayé de debug mon code mais je n'y arrive pas, je bloque toujours sur des scores qui ne font aucun sens. Je pense vraiment bientôt essayer de recommencer de 0 l'implémentation de l'IA, au lieu de continuer à perdre mon temps avec celle-ci. Je me replonge dans des documentation et dans le livre sur les intelligences artificielles pour essayer de comprendre où cela coince.

Je finis par décider d'abandonner mon implémentation et d'en recommencer une nouvelle, plus propre. Au bout de quelques heures j'obtiens déjà de meilleurs résultats qu'avec la précédente implémentation.
(Temps passé ~8h)

Samedi 08.06.2021

Pendant que je faisais des tests pour voir d'où venaient les problèmes de mon board, je me suis rendu compte que quand j'essayais de faire des coups sur la droite du board (colonne 6 ou 7) le coup était mal joué. J'ai pensé que le problème venait de la taille de mes variables (parce que j'ai besoin de *long* mais de mémoire je me rappelais qu'il y avait des subtilités avec javascript.) J'ai donc posé la question sur un discord avec des amis développeurs et voici ce qu'ils m'ont répondu:


In Java, you have 64 bits integers, and that's what you're using.

In JavaScript, all numbers are 64 bits floating point numbers. This means you can't represent in JavaScript all the Java longs. The size of the mantissa is about 53 bits, which means that your number, `793548328091516928`, can't be exactly represented as a JavaScript number.

If you really need to deal with such numbers, you have to represent them in another way. This could be a string, or a specific representation like a digit array. Some "big numbers" libraries are available in JavaScript.

Je vais donc essayer de fix ce problème avec cette librairie: long même si cela signifie abandonner un peu de performances.
(Temps passé ~6h)

Dimanche 09.06.2021

Aujourd'hui j'ai pu terminer la première implémentation du minimax grâce à la librairie "*long"*, et déjà maintenant, je n'arrive plus à battre l'IA. Elle joue en moyenne en 300 ms mais je veux maintenant implémenter un Alpha-Beta Pruning pour couper les branches inutiles de l'arbre et donc gagner du temps (et potentiellement pouvoir aller à une plus grande profondeur). Pour l'instant, l'implémentation ne marche pas mais je vais continuer à travailler dessus jusqu'à fin mai.
(Temps passé 4h)

Lundi 12.07.2021

Après environ un mois de pause, je me replonge dans mon code et décide de réécrire toute la fonction d'*alpha-beta pruning*. En effet, plus j'y pensais, moins elle faisait de sens. En me plongeant dans mes recherches, je tombe sur une vidéo : https://www.youtube.com/watch?v=l-hh51ncgDI qui explique très clairement le fonctionnement de l'algorithme. Je le ré implémente donc complètement, ajoute un nouvel heuristique, et le teste : les résultats sont là, l'IA est bien plus rapide qu'avant et bien plus forte. Je profite de mon regain de motivation pour refaire entièrement le côté client, le rendant plus léger, plus épuré et plus joli.

A la fin de ma session de développement, je décide d'uploader la version actuelle ici
(Temps passé ~12h)

Rédaction

Jeudi 12.08.2021

Après une autre pause d'environ un mois, je décide de me lancer dans la rédaction. Trouvant ce travail long et rébarbatif, je décide d'apprendre une nouvelle compétence pendant mon écriture : le LaTeX. Après quelques heures de galère et d'installations, j'arrive enfin à faire fonctionner mon compilateur et peux commencer à écrire. Je met en place la structure de mon travail et la bibliographie et décide de m'arrêter là.
(Temps passé ~4h)

Jeudi 16.08.2021

Je ne sais pas pourquoi, mais les jeudis je suis beaucoup plus motivé à travailler sur ma rédaction mais c'est comme ça. Aujourd'hui je commence la partie théorique, qui est la partie la plus compliquée car elle demande beaucoup d'illustrations et de réflection pour expliquer au mieux ce sujet à des néophytes. A la fin de la journée, j'ai écrit toutes les parties 1.1 (Description et règles du jeu) et 1.2 (L'intelligence artificielle) et j'ai commencé la partie suivante sur la structure des données.
(Temps passé ~6h)

Samedi 26.08.2021

Aujourd'hui, je continue la rédaction de la théorie et j'arrive enfin à finir la partie sur la structure des données.
(Temps passé environ 8h)

Mardi 07.09.2021

Aujourd'hui, grosse motivation, j'avance énormément dans la rédaction et fini entièrement l'introduction (qui était la partie la plus dure) et comment un peu le développement.
(Temps passé ~10h)

Mardi 14.09.2021

Après un rendez-vous avec mon professeur accompagnateur, je réarrange un peu mon intro et continue le développement.
(Temps passé 4h)

Samedi 02.10.2021 et Dimanche 03.10.2021

Sur tout le week-end je termine le développement. C'était long, mais j'en suis fier. Il ne manque que la conclusion, avant d'envoyer ma version provisoire à M. Mettral.
(Temps passé ~15h)

Lundi 11.10.2021

Aujourd'hui, j'ai fini la rédaction et j'envoie ma version provisoire à mon professeur accompagnateur.
(Temps passé ~3h)

Lundi 18.10.2021

Dernier rendez-vous avec M.Mettral, on discute de tous les points à améliorer dans mon travail. Il y en a beaucoup, mais je ne me démotive pas, c'est le dernier rush. Je commence à réécrire la partie théorique sur la structure des données, car elle manque de clareté.
(Temps passé ~2h)

Jeudi 21.10.2021 et Vendredi 22.10.2021

Sur les 2 jours, j'ai 2 trajets de train de 2h. J'en profite pour continuer les modifications de mon travail et continue en arrivant chez moi. Je pense avoir fini les modifications, je vais utiliser le week-end pour corriger l'orthographe et la syntaxe dans l'optique d'aller l'imprimer et le relier lundi.
(Temps passé ~6h)

Lundi 25.10.2021

Ce week-end je n'ai pas trouvé la motivation de travailler, donc je me suis levé tôt aujourd'hui et ai totalement corrigé l'orthographe de mon travail et suis allé l'imprimer l'après-midi. Mon travail est officiellement terminé, il ne me manque plus que la soutenance.
(Temps passé ~5h)

Temps total: environ 145h.