Jeu des pingouins à base de MCTS (*Monte Carlo Tree Search*) sur le navigateur en utilisant le format *WebAssembly*

Clément Chavanon Romain Hu Romain Hubert
Maxime Grimaud Volodia Parol-Guarino

Encadrant : Pascal Garcia

2019-2020

Résumé

Créer de toutes pièces une Intelligence Artificielle (IA) sur le jeu des pingouins. Ce jeu est un jeu de stratégie sur plateau, sa principale caractéristique vient de ses cases hexagonales. De plus ce jeu réagit très bien lorsque soumis à une IA de type *Monte-Carlo Tree Search*, que nous avons codé. Le second défi de ce projet est également sa plateforme cible : exécuter le code de l'interface et de l'IA dans un navigateur moderne. Pour cela nous utilisons `Emscripten` qui nous permet de compiler notre IA en `WebAssembly` et d'atteindre des performances proches du natif. Quant au *frontend*, c'est une application classique `Angular`.

Introduction

0.1 Le jeu des pingouins

Le jeu des pingouins est un jeu de stratégie sur plateau de 4 joueurs. Le plateau contient 60 cases hexagonales sur lesquelles se trouvent 1 à 3 poissons.

En début de partie, chaque joueur place un certain nombre de pingouins (de 2 à 4 suivant le nombre de joueurs) sur le plateau. A chaque tour, le joueur doit, si possible, bouger l'un de ses pingouins. Les mouvements de ceux se font sur en ligne droite depuis les 6 faces de la case hexagonale sur laquelle il se trouve. Il ne peut passer par des trous ou au-dessus d'autres pingouins, peu importe leur couleur. Une fois le mouvement achevé, la case de départ est retiré du plateau. Le joueur peut alors incrémenter du nombre de poisson qu'il y avait sur cette case son score.

Le jeu se termine lorsque plus aucun pingouin ne peut se déplacer. Le joueur avec le plus de points (poissons) remporte la partie.

0.2 Notre tâche

0.2.1 Au départ

Le sujet portait sur l'implémentation de ce jeu dans un environnement Web, en utilisant le nouveau standard *WebAssembly*. Les sources du projet sont compilées avec *Emscripten* qui permet de coder en `C++` pour la partie technique. L'interface devait se faire avec les bibliothèques *Simple DirectMedia Layer*.

0.2.2 Bref suivi

Afin de tester la faisabilité et les différentes technologies, nous avons décidé de procéder à la création de l'algorithme de façon abstraite et de tester avec un jeu simple et facilement implémentable : le morpion. Pour la partie graphique nous avions simplement codé en JavaScript vanilla. En parallèle nous avons testé une autre technologie pour cela : `PixiJS`. Cependant cela ne s'est pas avéré satisfaisant pour notre utilisation et avons décidé de choisir quelque chose de plus simple : `Angular`.

0.2.3 Nos prédécesseurs

Ce projet n'est pas nouveau. Une précédente équipe y a déjà passé de nombreuses heures. Cependant, afin de simplifier notre travail il a été décidé de tout refaire, y compris le MCTS dont le code leur avait été donné déjà optimisé. En effet, notre technologie étant récente, le *multithreading* par exemple pouvait s'avérer plus compliqué à porter en *WebAssembly* qu'à réécrire.

0.2.4 Notre objectif

Principalement nous nous sommes concentrés sur le fonctionnement correct de tout le projet et pas seulement de l'algorithme et du jeu. C'est pour cela que nous avons choisi de présenter un résultat plus correct qu'optimal (par exemple nous n'avons pas utilisé de représentation en *bitboards*, comme l'ont fait nos prédécesseurs, de même qu'ils n'ont pas eu l'algorithme à gérer).

1 Réalisation

1.1 Représentation du jeu

1.1.1 Représentation

1.2 MCTS

Le Monte Carlo Tree Search (ou MCTS) est un algorithme de recherche heuristique. C'est un algorithme qui explore l'arbre des possibles. Au fur et à mesure que l'algorithme se déroule, cet arbre grandit. Il essaye d'explorer toutes les parties possibles du jeu, en privilégiant les issues favorables pour lui. L'arbre est composé de noeuds répartis sur plusieurs couches. Chaque noeud représente une configuration et ses enfants, sont les configurations suivantes. Les noeuds doivent aussi stocker le nombre de parties gagnantes et le nombre total de simulation (à partir de ce noeud).

Le principe de l'algorithme est simple ; il n'y a que quatre étapes. On commence par choisir le "meilleur" noeud terminal. On détermine le meilleur noeud terminal grâce à la fonction UCT qui permet d'évaluer le meilleur compromis entre le nombre de visites et le résultat du noeud. Puis on crée ses enfants. Ensuite, on choisit un de ses enfants et on simule une partie aléatoire. Enfin, on transmet ce résultat sur tous les noeuds jusqu'à la racine.

On répète ces 4 étapes jusqu'à ce qu'on arrête l'algorithme. Ensuite, il nous retourne le meilleur coup à jouer, basé sur le nombre de visites des enfants de la racine.

1.3 Multithreading

Afin d'augmenter les performances du MCTS, nous nous sommes penchés sur le multithreading. En effet, cela nous débloque la possibilité de simuler plusieurs parties en même temps, impliquant une augmentation du nombre de parties simulées. Il y a différentes manières de multithreader le MCTS ; la *tree parallelization*, la *root parallelization* et la *leaf parallelization*. D'après cette étude [2] [1], la *root parallelization* semble la meilleure puisqu'elle permet d'explorer plus d'issues que les autres méthodes. Ainsi, cela augmente les chances de victoire du MCTS. De plus, cette méthode est facile à implémenter. En effet, il suffit d'assigner un arbre sur chaque thread. Les arbres sont donc développés indépendamment entre eux, donc il y a moins de chances que l'algorithme se bloque sur un minimum local. A la fin du temps alloué, nous mettons en commun les arbres, uniquement la première couche pour diminuer le temps de calcul. Ensuite, nous choisissons le meilleur coup à jouer.

Pour éviter de recréer l'arbre à chaque fois, nous avons mis en place un système de déplacement de la racine à un de ses enfants, gardant ainsi le sous-arbre de l'enfant.

1.4 Interface graphique

Pour offrir une expérience de jeu optimale, et afin d'exporter le jeu sur un navigateur, nous avons dû mettre en place une interface graphique pour notre jeu. Avec les contraintes de temps et les contraintes techniques, nous avons été amenés à faire des choix aux niveaux des technologies utilisées et des méthodes d'implémentation afin de pouvoir produire rapidement une interface utilisable.

1.4.1 Angular / Ionic

Afin de mettre en place, un code solide et rapidement exploitable, nous voulions impérativement utiliser `Typescript`, pour mettre en place le moteur de jeu côté graphisme. En effet, son contrôle de typage est un véritable plus, par rapport à notre *Proof Of Concept*, où le moteur du Tic-Tac-Toe était en *Javascript*. D'autre part, nous voulions mettre en place une architecture de site Web plus globale qui viendrait englober la partie véritablement jouable. Afin de mettre en place cette architecture web sur pied au plus vite, nous nous sommes tournés vers `Angular`.

Pour mettre en place la charte graphique de notre application, nous nous sommes tournés vers le framework `Ionic 4`, sorti récemment, qui offre aux développeurs des thèmes pré-conçus et des composants *responsives*. Basé sur *Angular*, il s'intègre donc parfaitement dans notre projet.

1.4.2 Organisation de l'application

Dans sa version finale notre application se compose des pages principales suivantes :

— une page d'accueil présentant le projet,
— une page avec le jeu en lui même,
— une page de présentation pour les membres de l'équipe,
— et une page pour les crédits.

Cette dernière permet de présenter le projet dans sa globalité, ainsi que les membres de l'équipe ayant participé à sa réalisation.

En utilisant `ngx rocket`, la base de l'application a pu être générée rapidement et avec une qualité de production. De cette manière notre application a pu disposer

d'un service de routage et d'un autre de traduction que nous avons agrémenté au fur et à mesure des différents ajouts de pages et de fonctionnalités.

Durant nos recherches dans les différentes possibilités que pouvait nous offrir *Ionic*, nous avons mis en place la possibilité d'accéder à une deuxième charte graphique, définissant le `Dark Theme`.

1.4.3 Développement du jeu

1.5 *Bindings* MCTS

Références

[1] H. Jaap van den Herik Guillaume M.J-B. Chaslot, Mark H.M. Winands : Parallel monte-carlo tree search.

[2] Reiji Suda Kamil Rocki : Massively parallel monte carlo tree search.