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

"Pingouins" est jeu de stratégie et de plateau surlequel s'affrontent 2 à 4 joueurs. Le plateau contient 60 cases hexagonales et qui comportent 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 déplacements autorisés se font en ligne droite suivant les 6 faces de la case hexagonale sur laquelle se trouve le pingouin. Il ne peut passer par dessus des trous ou au-dessus d'autres pingouins, peu importe qu'ils appartiennent ou non au même joueur. Une fois le mouvement achevé, la case de départ est retirée du plateau. Le joueur peut alors incrémenter son score du nombre de poisson qu'il y avait sur cette case. Si un pinguoin ne peut plus se déplacer, le joueur retire ce dernier ainsi que la case sur laquelle il était. Dans ce cas, là, le joueur remporte aussi les poissons contenus dans cette case.

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).

0.2.5 Répartition des tâches du projet

Pour mener à bien notre projet, les différentes tâches ont été réparties au sein des membres du groupe. Deux équipes ont été crées :

- Volodia et Romain Hubert pour la création du moteur du jeu en C++ et optimisation du code (multithreading)
- Maxime, Romain Hu et Clément pour la création de l'interface Web et préparation du lien entre le moteur et la partie graphique

Finalement, la partie qui consistait à permettre de transporter le jeu codé en C++ vers le navigateur a été faite par les membres des deux équipes (cf Bindings MCTS).

1 Réalisation

1.1 Notre environnement de développement

Devant la variété d'OS utilisés au cours de cette année par les membres de notre équipe et le fait que nous allions développer un stack technique peu commun en `c++` nous avons décidé de "simplifier" notre développement en utilisant les dernières fonctionnalités de VSCode et en utilisant le développement dans un container Docker

(nous avons également tenté un petit laps de temps sur Vagrant [1]]} mais l'expérience n'a pas eu grand succès). Cela permet au projet d'être extrêmement portable et d'être fonctionnel chez n'importe quel développeur !

1.2 Représentation du jeu

Notre encadrant nous a indiqué au tout début du projet un guide de méthodologies complet sur les plateaux hexagonaux et leurs représentations en informatique [6]. En se basant sur ce guide et sur la forme rectangulaire de notre plateau, nous avons choisi une représentation en mémoire avec une `std::unordered_map`, d'une part afin d'obtenir une complexité en temps en $O(1)$ moyen et pas de $O(log(n))$ moyen avec les classiques `std::map`. D'autre part puisque qu'une telle représentation d'une carte rectangulaire serait lacunaire dans une structure de donnée tabulaire [6][map-storage].

1.3 Points sensibles

Utiliser un algorithme tel que le MCTS implique que la vitesse et l'efficacité de ce dernier va grandement être impacté par la puissance de la représentation. Sur le MCTS il existe 2 méthodes qui sont principalement des goulots d'étranglements :

— la méthode servant à donner tous les cas disponibles pour un joueur, qui doit en effet analyser si les quelles routes sont possible et jusqu'à quel endroit [2] ;

— la méthode servant à donner l'état du jeu (aussi utilisée pour connaître le joueur suivant [3]) qui doit vérifier s'il est encore possible pour un joueur de bouger [4].

Une passe d'optimisation a déjà été réalisée sur la deuxième méthode qui reposait à la base sur la première (dans un effort d'obtenir le plus vite une démo fonctionnelle afin de déboguer des points plus vitaux). Cependant le temps manque pour faire plus, notamment nos prédécesseurs ont eu le temps [5] de vraiment attaquer le vif de l'optimisation, notamment avec les *bitboards*, qui leur ont permis une belle différence de performance (sans compter ici le fait que nous développons pour le web).

1.4 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).

1. un ingénieur chez Sopra Steria nous l'avait conseillé, en effet Docker possède une faille majeure sur Windows : il n'est disponible sur les versions non professionnelles que en tant que *Docker Toolbox* qui ne permet pas une utilisation avec VSCode.

2. Un pingouin peut être bloqué par un trou dans le plateau ou un autre pingouin.

3. Il arrive qu'un joueur soit bloqué et que attendre son tour ne serve à rien, son adversaire peut lui continuer à récolter tous les points.

4. Un pingouin peut être bloqué par un trou dans le plateau ou un autre pingouin.

5. L'équipe précédente n'a pas eu à faire le MCTS et avaient directement une interface connectant la représentation avec cet algorithme, sur laquelle nous avons dû faire quelques ajustements après avoir développé la représentation du jeu des pingouins. Nous faisons allusions ici à une différence entre le pion et le joueur : un joueur peut posséder plusieurs pions et ceci n'était pas une contrainte sur notre première phase de tests avec un morpion...

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.

1.5 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 [5, 4], 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.

1.6 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.

1.6.1 Angular / Ionic

Afin de mettre en place, un code solide et rapidement exploitable, nous voulions impérativement utiliser `Typescript`, pour réaliser 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 construire 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 avons décidé d'utiliser `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.

1.6.2 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`.

1.6.3 Développement du jeu

1.7 *Bindings* MCTS

Il existe deux façons de faire des *bindings* sur *Emscripten*. La première consiste à indiquer tous les noms des fonctions à exporter. Nous l'avons testé lors de notre *Proof Of Concept* sur le morpion. Nous avons retenu la seconde : *Embind*[2]. Cette dernière permet de réaliser des *bindings* plus proprement sur des classes entières et leur ascendance. Nous avons mis en place une organisation un peu à la *Angular* : une fichier par dossier qui joue le rôle d'un module et qui contient tous les exports des classes locales. Ceci permet également de garder une compatibilité parfaite entre les deux plateformes sur lesquels nous développons : le Web et l'environnement Linux natif. *Embind* permet également l'export des pointeurs [6], ce qui est tout à notre bénéfice pour éviter d'avoir à ajouter des méthodes juste pour *Emscripten*.

Un second défis aura été de lier le programme une fois multithreadé avec `pthreads`[3]. En effet, le Web a introduit des threads particulier : les *Web Workers*. Cependant ils possèdent leur propre espace mémoire complètement séparé de l'application et ne permettent qu'une communication via `int` ou `strings`. Heureusement pour nous le plus gros du travail est réalisé par *Emscripten* mais nous avons néanmoins eu un problème inacceptable : le blocage du *thread* principal de notre application (à cause de la méthode d'attente de résolution des threads). Pour pallier à cela nous avons mis en place un mécanisme reposant sur *Asyncify* [1] qui permets de des `pause` et `resume` dans le code `c++` et donc la poursuite des évènements asynchrones tant appréciés de `JS`. La solution ne s'arrête pas là puisque nous avons dû utiliser un mécanisme d'attente des *threads* non-bloquant en utilisant une boucle infinie accompagné d'un temporisateur, permettant ainsi de faire reprendre la main à l'application *Angular* pendant quelques millisecondes.

6. Il existe les pointeurs intelligents en c++, seulement notre première utilisations de ces derniers a été d'utiliser la version `std::shared_pointers` à la première occasion. Devant notre ignorance nous nous sommes rabattu sur le classique des pointeurs `c`. Si nous avions continuer nous aurions certainement abusé des pointeurs `shared` et finis par perdre massivement en performance et en mémoire, surtout que nous avions déjà en tête de multithreader notre application. Nous ne parlons que des `shared_pointers` puisque nous ne connaissions pas réélement les mécanismes de *ownership* des `unique_pointers`.

2 Conclusion

conclusion

Références

[1] EMSCRIPTEN.ORG : Asyncify. https://emscripten.org/docs/porting/asyncify.html.

[2] EMSCRIPTEN.ORG : Embind. https://emscripten.org/docs/porting/connecting_cpp_and_javascript/embind.html.

[3] EMSCRIPTEN.ORG : Pthreads support. https://emscripten.org/docs/porting/asyncify.html.

[4] H. Jaap van den Herik GUILLAUME M.J-B. CHASLOT, Mark H.M. Winands : Parallel monte-carlo tree search. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.159.4373&rep=rep1&type=pdf.

[5] Reiji Suda KAMIL ROCKI : Massively parallel monte carlo tree search. https://pdfs.semanticscholar.org/0fe8/ad034ce19d7ec0faf6df988312742d327f81.pdf.

[6] Amit PATEL : Blobs in games : Improving hexagon map storage diagram. https://www.redblobgames.com/grids/hexagons/, 2019.