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 l'interface graphique, c'est une application classique *Ionic* (un *framework* basé sur *Angular*).

Introduction

"Pingouins" est un jeu de stratégie et de plateau sur lequel s'affrontent 2 à 4 joueurs. Le plateau contient 60 cases hexagonales 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 poissons qu'il y avait sur 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.

1 Sujet

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*.

1.1 Précédemment

Ce projet n'est pas nouveau. Une précédente équipe a déjà réalisé une application bureau, en *Java*. Cependant, notre cachier des charges était différent, notamment nous devions déployer le jeu sur le web. Ainsi pour préparer ce passage sur le web, et faciliter notre compréhension du MCTS, nous avons décidé de reprendre la logique du début. De plus le fait que _WebAssembly_soit aussi récent, certaines fonctionnnalités, comme la parallélisation de l'algorithme, semblaient plus simple à réécrire qu'à porter telles quelles.

1.2 Preuve de Concept

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 (servant alors de *Preuve de Concept* - PdC). Pour la partie graphique, nous avions simplement codé en JavaScript pur. Pour la suite du projet, pour faciliter le développement de la partie *front-end*, nous avons décidé de choisir : *Angular*. Sur la PdC, nous avions testé une autre technologie pour gérer le graphisme du jeu : *PixiJS*. Cependant, plus tard, cela ne s'est pas avéré satisfaisant pour notre utilisation. En effet *PixiJS* nécessite une gestion asynchrone de son canvas, son intégration dans une application *Angular* doit donc se faire dans une zone indépendante, le lien avec le *WebAssembly* devenait alors trop complexe.

1.3 Répartition

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

- 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 du jeu et la partie graphique.

Finalement, la tâche 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).

2 Réalisation [1]

2.1 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. Cela permet au projet d'être extrêmement portable et d'être fonctionnel chez n'importe quel développeur !

Et en bonus nous avons réalisé ce rapport en *Markdown* afin qu'il soit facilement visible sur notre *repository*.

1. Toutes nos sources sont disponibles [2]. Nous avons également une démonstration en ligne [3].

2.2 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 [7]. En se basant sur ce guide et sur la forme rectangulaire de notre plateau, nous avons choisi une représentation en mémoire avec un conteneur associatif sous forme de table de hachage : `std::unordered_map`. Cela permet d'obtenir une complexité moyenne en temps de $O(1)$ et pas de $O(log(n))$ avec les représentations classiques, soit avec un conteneur associatif basé sur des arbres équilibrés : `std::map`. La représentation de la grille hexagonale sous forme rectangulaire crée des parties non utilisées dans le tableau [7, voir la section *map storage*].

2.3 Points sensibles

Les performances du MCTS sont liées à son efficacité et à sa vitesse d'execution, ces derniers sont grandement impactés par la représentation du jeu. Pour le MCTS, deux méthodes de représentation sont alors possible :

— basée sur l'énumération de tous les cas disponibles pour un joueur, et d'une analyse des mouvements possibles. [2] ;

— basée sur l'état du jeu, afin de connaître à tout instant le joueur qui doit jouer [3], et s'il est encore possible pour lui de bouger [4].

Pour obtenir au plus vite une démonstration fonctionnelle pour le debuguage, nous avons optimisé la deuxième représentation avec une énumération efficace des cas possibles. Cependant par manque de temps, contrairement à nos prédécesseurs [5], nous n'avons pas pu pousser l'optimisation aussi loin. En particulier avec la mise en place de *bitboards*, qui augmentent de manière significative les performances, qui nécessitent pas mal de temps pour le déployer sur le web.

3 MCTS

3.1 Principe

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 simulations (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

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 qu'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 avait 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 allusion 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...

ses enfants et on simule une partie aléatoire. Enfin, on transmet ce résultat sur tous les noeuds jusqu'à la racine.

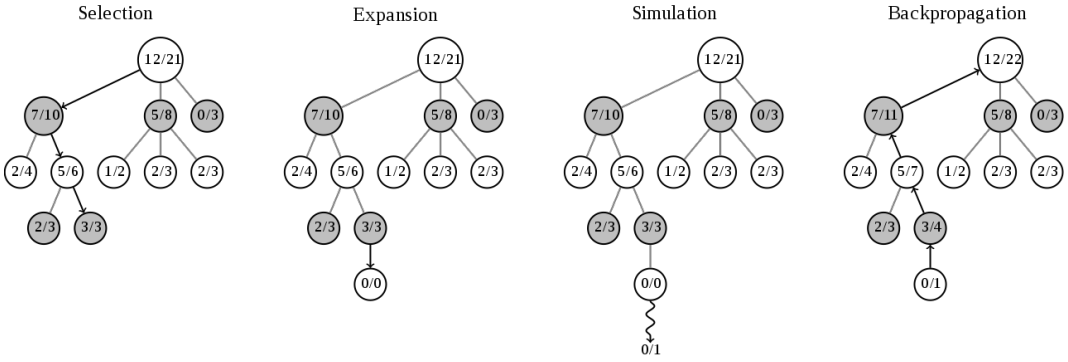

FIGURE 1 – Les quatre étapes du MCTS

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.

3.2 Parallélisation

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 [8, 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.

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.

4.1 *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 preuve de concept, où le moteur du morpion 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 adaptatifs. Basé sur *Angular*, il s'intègre donc parfaitement dans notre projet.

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.

FIGURE 2 – Aperçu interface graphique

Cette dernière permet, en plus de mettre à disposition le jeu des pingouins dans un navigateur web, de présenter le projet dans sa globalité, ainsi que les membres de l'équipe ayant participé à sa réalisation. L'ensemble du rendu graphique est défini par un ensemble de composants venant s'incruster dans des *pages Ionic*. La gestion et la levée d'évènements se fait conformément au standard *Angular*, et par un jeu de double lien dans la hiérarchie des composants.

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

4.3 Automates à états finis

Que ce soit pour l'application entière, ou le jeu en particulier il a fallu mettre en place des automates finis (*Finite-State Machine*), afin de gérer le flot de contrôle, et contenir les actions possibles en fonction de l'état d'avancement.

Le flot de contrôle est modélisé par 2 machines à états :

— une pour l'application globale (apparition des différents composants en fonction des interactions avec l'utilisateur) ;

— une deuxième pour gérer exclusivement le jeu.

Pour mettre en place, ces automates finis, nous avons utilisé la librairie *Typescript* `+xstate`, permettant de mettre en place rapidement des automates sous le format *JSON*. Cette dernière offre aussi un système de visualisation des machines.

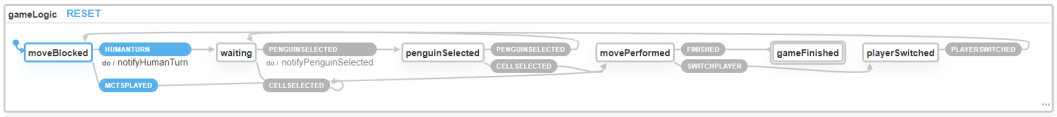

FIGURE 3 – Aperçu Automate fini du jeu

L'Automate du jeu permet de dérouler la logique du jeu des pingouins, en limitant les interactions en fonction du joueur qui doit jouer. Le passage d'un état à un autre se fait par le déclenchement d'une action pré-enregistrée, souvent cette dernière est associée à un événement sur un composant *Ionic*. La progression dans le jeu se fait donc complètement indépendamment de l'application dans laquelle il est intégré.

Cette manipulation d'états et d'événements permet d'offrir à l'utilisateur une interaction agréable et visuelle avec le plateau de jeu.

5 Liens entres toutes les parties

5.1 Lien entre interface graphique, représentation et MCTS

Il faut maintenant faire le lien entre l'interface graphique et le cœur du jeu. Il existe plusieurs niveaux de difficulté pour réaliser ces liens. Le plus simple nous l'avons utilisé lors de notre preuve de concept avec le morpion. Il consiste à marquer les fonctions à exporter directement dans la commande de compilation et est adaptée pour une petite quantité de fonctions. Cependant, le passage à l'échelle ne se fait pas bien, c'est pour cela que nous avons utilisé la seconde méthode : *Embind* [5].

Elle se traduit pour l'utilisateur en de simples lignes d'export de méthodes dans un préprocesseur. Les seules difficultés peuvent venir des *templates* en *C++* qui peuvent faire grossir le code, mais un préprocesseur adapté suffit à limiter cela et de l'organisation générale du projet. C'est-à-dire que suivant où l'on situe ces lignes de lien, on peut avoir du mal à savoir quels classes sont concernées, c'est pour cela qu'en nous inspirant de *Angular* nous avons un ficher avec l'extension `*.bind.cpp` qui reprend toutes les fonctions exportées dans le dossier courant et permet ainsi d'avoir très peu de méthodes à écrire spécifiques aux les liens. Le compilateur se charge alors de réaliser ces liens automatiquement (et mêmes des pointeurs [6] !). De

6. Il existe les pointeurs intelligents en C++, seulement notre première utilisation de ces derniers a été d'utiliser la version `std::shared_pointers` à la première occasion. Devant notre ignorance nous nous sommes rabattus sur le classique des pointeurs *C*. Si nous avions continué nous aurions certainement abusé des pointeurs `shared` et fini par perdre massivement en performance et en mémoire, surtout que nous avions déjà en tête de paralléliser notre application. Nous ne parlons

plus la clarté gagnée par cette structure permet aussi de continuer à garder deux plateformes pour développer : le Web et Linux pour avoir accès à l'éventail d'outils de débogage existants.

5.2 Parallélisation

Notre second défi a été de lier la version parallélisée de notre programme avec `pthreads`[6] et l'interface graphique. En effet, le Web a introduit sa propre version des *threads* : les *Web Workers*[7]. Cependant ils possèdent leur propre espace mémoire complètement séparé de l'application et ne permettent qu'une communication via des types primitifs : les `int` ou les `strings`. Il n'est donc pas aisé de communiquer des valeurs d'instances entres ces *Web Workers*. Heureusement pour nous, le plus gros du travail est réalisé par *Emscripten*. Néanmoins, nous avons eu un problème inacceptable : le blocage du *thread* principal de notre application lors du développement des arbres du MCTS, l'interface ne répondait alors plus. Pour pallier cela nous avons mis en place un mécanisme reposant sur *Asyncify* [4] qui permet de faire des `pause` et `resume` dans le code *C++* exporté. Plus largement ce module permet de rendre le code asynchrone et donc de poursuivre le traitement des évènements tant appréciés de *JavaScript* lors de l'exécution de notre algorithme qui n'est alors plus bloquant. Le résultat n'est pourtant pas ce que nous espérions, puisque la fonction exécutant le MCTS ne renvoie alors plus de valeur au final. Nous avons alors défini une fonction *JavaScript* dans le code *C++*, de façon à ce que ce dernier puisse l'appeler. Cette fonction permet alors d'émettre un événement après que la fonction *C++* ait terminé[8]. Cette notification permet alors à l'interface de savoir quand récupérer la valeur de sortie et de pallier le problème initial.

Conclusion

La mise en place de ce projet a permis de mettre en évidence les difficultés liées à la gestion de ce type de travail, notamment au niveau de l'organisation et les échéances temporelles. Notablement, au début nous n'avions pas les mêmes quantités de travail pour l'équipe graphique que pour la première version du MCTS.

Les technologies utilisées étaient le second point important de ce projet, certaines étaient déjà connues – voire maîtrisées – par des membres du groupe, néanmoins la plupart se sont avérées être une totale découverte. Il fallait donc être capable d'acquérir des connaissances technologiques (*WebAssembly*, MCTS, *Multithreading*, *Angular* ...) mais également dans les outils nécessaires pour travailler dans une position peu commune (*VSCode*, *Docker*, *Doxygen*, *Compodoc*) tout en développant – pour permettre au projet d'avancer. Finalement, le résultat attendu par le cahier des charges a été plus qu'atteint : en effet, nous sommes en mesure de proposer un jeu des pingouins, implémentant une intelligence artificielle, et jouable à partir d'un navigateur Web. Nous avons démontré la viabilité et la maturité du *WebAssembly*[9], tout en se heurtant à des obstacles – pas impossibles à passer – mais néanmoins

que des `shared_pointers` puisque nous ne connaissions pas réellement les mécanismes de propriété des `unique_pointers`.

7. Tout comme le *WebAssembly* les *Web Workers* ont un support encore limité aux versions récentes des navigateurs, pour ceux ne l'ayant pas désactivé pour des raisons de sécurité.

8. Cela garantie que le traitement de l'événement se produit après la sortie de la fonction qui crée cet événement.

9. un point étonnant est la possibilité d'allier deux géants dans leurs domaines : la versatilité du *JavaScript* et la puissance crue du *C++*.

gênants pour un environnement de production. Cette expérience nous a permis, en plus d'approfondir nos connaissances acquises au cours de l'année, de mieux connaître le fonctionnement de chacun et d'apprendre, en demandant conseil à notre encadrant lorsque cela devenait ardu, mais aussi à présenter notre travail [10].

Pour finir :

> *Laissons l'avenir dire la vérité, et évaluer chacun en fonction de son travail et de ses accomplissements. Le présent est à eux ; le futur, pour lequel j'ai réellement travaillé, est mien.*
>
> – Nikola Tesla

Références

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

[2] Clément Chavanon, Romain Hu, Romain Hubert, Maxime Grimaud et Volodia Parol-Guarino : Penguingame github repository. https://github.com/volodiapg/penguin, 2019.

[3] Clément Chavanon, Romain Hu, Romain Hubert, Maxime Grimaud et Volodia Parol-Guarino : Penguingame github repository - demo. https://volodiapg.github.io/penguin, 2020.

[4] emscripten.org : Asyncify. https://emscripten.org/docs/porting/asyncify.html.

[5] emscripten.org : Embind. https://emscripten.org/docs/porting/connecting_cpp_and_javascript/embind.html.

[6] emscripten.org : Pthreads support. https://emscripten.org/docs/porting/asyncify.html.

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

[8] Kamil Rocki et Reiji Suda : Massively parallel monte carlo tree search. https://pdfs.semanticscholar.org/0fe8/ad034ce19d7ec0faf6df988312742d327f81.pdf.

10. L'exercice s'est avéré étrange mais encourageant.