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

0.1 Le jeu des pingouins

"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. Si un pingouin 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 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*.

0.2.2 Récapitulatif

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 `Proof Of Concept` - Poc). 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 le PoC 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.

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, la parallélisation de l'algorithme, par exemple, pouvait s'avérer plus compliquée à porter en `WebAssembly` qu'à réécrire.

0.2.4 Notre objectif

Nous nous sommes principalement 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 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 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]

1.1 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

1. Toutes nos sources sont disponibles [7]. Nous avons également une démonstration en ligne [8].

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*.

1.2 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 [9]. 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`, d'une part afin d'obtenir une complexité en temps en $O(1)$ moyen et pas de $O(log(n))$ moyen avec les 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 [9, voir la section *map storage*].

1.3 Points sensibles

Utiliser un algorithme tel que le MCTS implique que la vitesse et l'efficacité de ce dernier vont grandement être impactés par la représentation. Pour le MCTS, deux méthodes de la représentation du jeu ont un impact très important pour le niveau de l'intelligence artificielle que l'on obtient :

— la méthode servant à donner tous les cas disponibles pour un joueur, qui doit en effet analyser quelles routes sont possibles 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,

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...

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

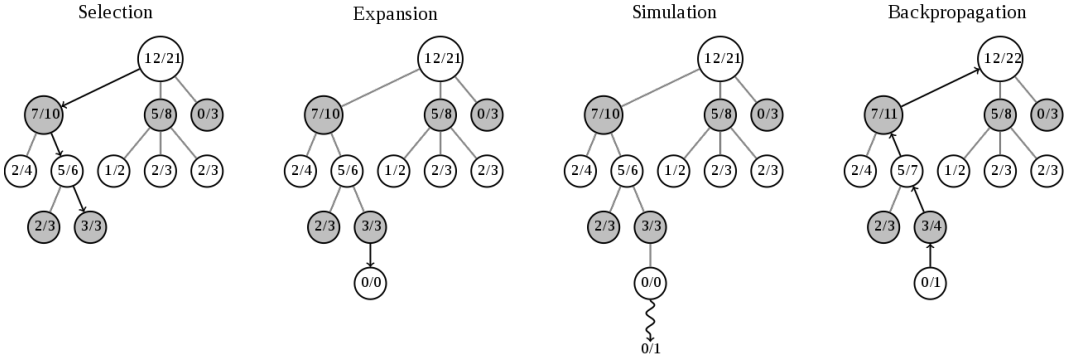

FIGURE 1 – "Les quatre étapes du MCTS"

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.

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 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.

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* [6] [6], 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 pouvaient 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 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. Elle consiste à lier 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* [2]. 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 ou 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`

6. *ngX-Rocket* est un modèle pour générer facilement une première application. Il permet de débuter avec une architecture de projet robuste, des plugins déjà présent (*I18n*), d'avoir accès à des services (authentification, api) et de choisir un style facilement.

qui reprend toutes les fonctions exportée dans le dossier courant et permet ainsi d'avoir très peu de méthode à écrire juste pour les liens. Le compilateur se charge alors de réaliser les liens automatiquement (et mêmes des pointeurs [7] !). De 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. Un exemple d'un tel code est le suivant :

```
0  ...
1      // Only target Emscripten compilation (auto-generated flag)
2  #ifdef __EMSCRIPTEN__
3  ...
4  using namespace emscripten;
5
6  // Binding code
7  EMSCRIPTEN_BINDINGS(mcts_bind)
8  {
9      // Here exporting a c style structure with a field
10     value_object<MCTSConstraints>("MCTSConstraints")
11         .field("time", &MCTSConstraints::time);
12
13     // It is a bit more complex to export a template,
14     // especially if we want to take fully advantage
15     // of why they were made in the first place :
16     // multiple types
17 #define __MCTS_BIND__(name_prefix, MCTSPlayer, AbstractGame)
18     class_<MCTSPlayer>(name_prefix "_MCTSPlayer")
19         .constructor<AbstractGame *const &, const MCTSConstraints &>
20                                             (allow_raw_pointers()) \
21         .function("bestMove", &MCTSPlayer::bestMove)
22     // We need to define the types (by clarity +
23     //                  limitation of the preprocessors)
24     typedef MCTSPlayer< ... > penguin_mcts_player_t;
25     typedef game::AbstractGame< ... > penguin_game_t;
26     // We use it for the penguin game,
27     // but it is as easy to export for the tic tac toe demo
28     __MCTS_BIND__("penguin", penguin_mcts_player_t, penguin_game_t);
29 }
30
31 #endif
```

Notre second défi a été de lier la version parallélisée de notre programme avec `pthreads`[3] et l'interface graphique. En effet, le Web a introduit sa propre version des *threads* : les *Web Workers* [8]. Cependant ils possèdent leur propre espace mémoire

7. 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 multithreader notre application. Nous ne parlons que des `shared_pointers` puisque nous ne connaissions pas réellement les mécanismes de *ownership* des `unique_pointers`.

8. 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.

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 à cela nous avons mis en place un mécanisme reposant sur *Asyncify* [1] 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 revoie 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 [^whyafterterm]. Cette notification permet alors à l'interface de savoir quand récupérée la valeur de sortie et de palier au problème initial.

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] NGX-ROCKET : ngx-rocket repository. https://github.com/ngx-rocket.

[7] Clément C. Maxime G. Romain H. Romain H. Volodia P.-G. : Penguingame github repository. https://github.com/volodiapg/penguin, 2019.

[8] Clément C. Maxime G. Romain H. Romain H. Volodia P.-G. : Penguingame github repository - demo. https://volodiapg.github.io/penguin, 2020.

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