INF8215 – Intelligence artificielle : Méthodes et algorithmes

Groupe 01

Projet – Agent intelligent pour le jeu Quoridor

Sanmar Simon 1938126
Ghali Harti 1953494

Nom d'équipe : GhaliSanmar

Soumis à Quentin Cappart

Jeudi 21 avril 2022
Hiver 2022

Table des matières

Introduction

Quoridor est un populaire jeu de stratégie à deux ou quatre joueurs apparu dans les années 90. Dans notre cas le jeu se joue à deux joueurs, tour par tour, est déterministe, à information parfaite et somme nulle. Cela signifie que l'environnement change après le tour de l'adversaire, qu'il n'y a pas de présence d'aléatoire, que l'environnement est parfaitement connu, et surtout qu'une action bonne action pour mon adversaire équivaut à une mauvaise action pour moi (si mon adversaire a un score de 3, alors mon score est de -3). Ces informations sont essentielles à noter car nos algorithmes se basent sur ces dernières. Maintenant que nous avons déterminer le type de jeu qu'est Quoridor, nous pouvons explique le but du jeu et ses règles. Le but du jeu est très simple, pour gagner le joueur doit faire bouger son pion jusqu'à l'autre côté du plateau avant son adversaire. Les deux pions commencent face-à-face chacun d'un côté. On dit que le joueur 1 commence à E1, et que le joueur 2 commence à E9 (E numéro de colonne suivi du numéro de ligne), en d'autres termes le but pour le joueur 1 est d'atteindre la ligne 9 avant que le joueur 2 n'atteigne la ligne 1 (Alexander Brown & Massagué Respall, 2018). Le plateau a une taille de neuf lignes et neuf colonnes pour un total de 81 cases. À chaque tour le joueur doit effectuer une opération parmi les deux types d'opération différentes :

- Déplacer son pion dans l'une des quatre directions possible (devant, gauche, droite, arrière). Dans certaines configurations si le pion de l'adversaire est devant le pion du joueur, alors ce dernier peut sauter le pion adverse. Ou encore si un mur est derrière le pion adverse de telle sorte qu'on ne peut pas le sauter, alors il est possible de bouger en diagonale.
- Placer un mur (horizontalement ou verticalement) pour ralentir la progression de son adversaire, cependant il est interdit de placer un mur qui rend la victoire impossible pour l'adversaire. Au début de la partie chaque joueur possède dix murs.

Maintenant que nous avons rappelé le type de jeu qu'est Quoridor ainsi que ses règles, nous pencher sur la complexité du jeu. On entend par là les différentes combinaisons possibles dans le jeu.

Complexité du jeu

Déterminer la complexité du jeu est essentielle pour pouvoir modéliser le problème et construire notre agent intelligent. On peut assez facilement évaluer le nombre d'états possible. Un état du graphe de recherche correspond à un état précis dans lequel se trouve l'environnement à un instant t, c'est-à-dire la position des pions et des murs. Par conséquent le nombre maximal d'états valides dans le graphe de recherche est celui du nombre de positions possible des pions multiplié par le nombre total de positions de murs. Le plateau contient exactement 81 cellules. On a donc un total de 81 positions pour le premier pion et de 80 pour le deuxième. Le nombre total de différentes combinaisons de positions de pions possible vaut donc :

$$E_p = 81 \times 80$$

$$E_p = 6480$$

Ensuite, concernant les combinaisons de mur possible on sait qu'on peut avoir un maximum de 20 murs, et qu'un mur a une longueur équivalente a deux cases. De plus, il y a huit façons de placer un mur dans une rangée. Étant donné qu'il y a huit rangées, il y a soixante-quatre endroits possibles pour placer une clôture horizontalement. Comme le plateau est carré et que nous avons le même nombre de rangées et de colonnes, un mur peut être placée à cent vingt-huit endroits. Il faut tenir compte du fait qu'un mur occupe quatre emplacements, à l'exception des cases au bord. Ainsi, le nombre total de positions possible des vingt mur E_m est obtenue par l'équations suivante (Mertens, 2006) :

$$E_m = \sum_{i=1}^{20} \prod_{j=0}^{i} (128 - 4i) = 6.1582 \times 10^{38}$$

Et le nombre total d'états E est lui :

$$E = E_p \times E_m = 6480 \times 6.1582 \times 10^{38} = 3.9905 \times 10^{42}$$

Ainsi selon Mertens, Quoridor a un nombre d'états possible similaire aux échecs et une complexité d'arbre de jeu plus élevée (Mertens 2006).

Méthodologie

Compte tenu de la complexité du jeu Quoridor, et du budget temps de 20 minutes alloués à notre agent, notre algorithme se doit d'être efficace pour permettre à notre agent de jouer le meilleur coup à chaque tour et dans un temps minimum. Dans un premier temps nous nous sommes tournés vers l'algorithme minimax avec élagage alpha-beta, puis nous l'avons remplacé par un négamax avec élagage avant de finalement opter pour un Monte Carlo Tree Search (MCTS). Nous verrons dans la partie des résultats les performances de nos anciens algorithmes et la démarche qui nous as mené à opter pour un MCTS, mais avant ça nous allons nous pencher sur l'algorithme retenu.

Monte Carlo Tree Search

L'algorithme de recherche arborescente de Monte Carlo est celui que nous avons retenu pour construire notre agent intelligent. Il s'agit d'un algorithme de recherche probabiliste doté d'une capacité de prise de décision unique en raison de son efficacité dans les environnements ouverts offrant une quantité énorme de possibilités. Comme elle est basée sur l'échantillonnage aléatoire des états de jeu, elle n'a pas besoin d'utiliser la force brute. Nous avons construit un arbre de jeu avec un nœud racine, puis il est élargi avec des simulations aléatoires. L'algorithme se divise quatre phases :

La sélection

Cette partie correspond à la phase initiale durant laquelle l'algorithme commence par un nœud racine et sélectionne un nœud enfant de manière à choisir le nœud ayant le taux de victoire maximum. Cette étape est répétée jusqu'à temps qu'on arrive à un nœud feuille. Afin de s'assurer que chaque nœud a une chance équitable d'être sélectionné et d'équilibrer la situation entre l'exploration et l'exploitation, nous utilisons UCT :

$$UCB(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\ln(N(Parent(n)))}{N(n)}}$$

UCB(n) représente le résultat du upper bound confidence pour un nœud n.

U(n) : Le nombre de victoires de toutes les simulations pour un nœud n.

N(n) : Le nombre total de simulation ayant impliqué le nœud n.

N(Parent(n)) : Le nombre total de simulation ayant impliqué le nœud parent de n.

C : Paramètre d'exploration, théoriquement $\sqrt{2}$ selon Kocsis et Szepesvári (2006).

Cette formule garantit que l'agent jouera les branches prometteuses plus souvent que leurs homologues, mais qu'il explorera aussi parfois de nouvelles options pour trouver un meilleur nœud, s'il existe.

L'expansion

Cette partie est relativement simple, elle s'effectue après la sélection. En effet une fois un nœud feuille sélectionné on expand l'arbre on génère tous les prochains états possibles à partir du nœud actuel. Autrement dit, on génère tous les nœuds enfants du nœud actuel.

La simulation

Pour donner suite à l'expansion on a probablement l'étape la plus importante qui est la simulation. C'est au cours de celle-ci que l'on va simuler à partir du nœuds actuel le reste de la partie pour ainsi déterminer si on termine par une victoire ou une défaite. Dans notre cas, compte tenu des contraintes de temps nous n'allons pas jouer des coups aléatoirement avec une certaine probabilité à chaque tour. Pour simuler le reste de la partie nous utiliserons une heuristique qui calcule la longueur du chemin le plus court séparant chaque joueur de son objectif. On considère à partir de ce résultat que le joueur étant le plus proche de son objectif gagne la partie. Pour calculer le chemin le plus cour on utilise l'algorithme A*.

La rétropropagation

Cette phase est la dernière, elle consiste à remonter les résultats de la simulation du nœud feuille actuel jusqu'à la racine de l'arbre en mettant ainsi à jour pour chaque nœud rencontrés le nombre de victoires et le nombre total de simulations.

Implémentation et gestion des ressources

Dans notre implémentation de MCTS nous avons créé 3 classes :

- Tree Cette classe représente l'arbre des différents états. Chaque nœud représente un état. C'est cette classe qui s'occupe des différentes phases de MCTS (sélection, expansion, simulation et rétropropagation).
- Node Cette classe représente un Nœud de l'arbre donc un état du jeu.
- CustomBoard Cette classe reprends les mêmes principes que la classe Board, mais avec des fonctions en plus et des modifications à certaines fonctions.

Une des contraintes importantes que notre agent a à respecter est une limite de temps de 20 minutes. Ainsi différentes façons de gérer la ressource temps se sont offerts à nous. Nous avons cependant d'allouer notre temps de manière progressive. Étant donné qu'on estime que les premiers coups joués présentent que peu de conséquence pour le reste de la partie, nous avons décidé d'allouer que quelques secondes pour ces derniers (15 secondes pour les cinq premiers tours cumulés). Pour le reste nous sommes partis du principe qu'une partie dure généralement 40 tours, alors on divise le temps restant par le nombre de tours, ainsi à partir du 6^{e} tour tous les tours possèdent la même durée.

Résultats et évolution de l'agent

Comme mentionné précédemment, MCTS ne fut pas notre premier choix. Dans un premier temps nous avions implémenté Minimax avec élagage alpha-beta et avec une profondeur maximale. Les résultats étaient convenables puisque notre agent battait le joueur aléatoire et le joueur gourmand. Cependant nous avions le pressentiment qu'en raison du nombres d'heuristiques que nous utilisions pour évaluer la qualité de l'état les performances de notre agent pouvaient être améliorés. En effet on utilisait 5 heuristique différentes (Djikstra pour chaque joueur, distance entre pions, nombre de murs restant pour chacun) que l'on additionnait ensemble avec des coefficients choisis. Choisir les coefficients fut également une difficulté puisqu'il n'y avait pas de réelles méthodes autre que de les choisir arbitrairement et tester. Compte tenu de ces difficultés nous nous sommes dirigées vers l'algorithme negamax qui est une variante de minimax. La seule différence est que l'on inverse la valeur obtenue à la suite de la fonction d'évaluation. Cela a pour conséquence que la valeur de la fonction d'évaluation est la valeur vue du point de vue du nœud actuel et donc pas toujours du point de vue de MAX comme dans minimax (Glendenning L, 2005). Cela a pour but de simplifier la compréhension, mais les performances demeurent les mêmes. C'est alors que nous nous sommes orientés vers l'algorithme MCTS. Ce dernier remporte tous ses matchs contre nos anciens agents, et il en va de même pour les matchs contre les agents gourmands et aléatoires.

Discussion

Pour conclure le tout, nous tenons à dire que les performances de notre agent sont convenables et encourageantes. En effet au fur et à mesure nous avons constaté de nombreuses pistes d'améliorations possibles pour notre agent intelligent. Premièrement avec du recul sur la situation on considère que notre gestion du temps aurait pu être plus intéressante. En effet pour notre gestion du temps on estime qu'une partie dure 40 tours, or on a constaté que cela varie beaucoup, de nombreuses parties dure moins jusqu'à voir 30 tours, tandis que d'autres durent plus jusqu'à 50 tours. Lorsqu'une partie dure moins de 40 tours cela veut dire que l'on n'a pas pu exploiter notre budget temps dans son intégralité ce qui est regrettable.

Un autre point qui peut être sujet à amélioration concerne le calcul du chemin le plus court. En effet à chaque tour on re effectue l'algorithme A*, cela est redondant car entre un nœud parent et un nœud fils seul 1 seule action diffère, ainsi l'environnement est plus ou moins pareil. Il aurait donc été intéressent de pouvoir sauvegarder ces informations-là même si cela utiliserait plus de mémoire.

Finalement, on pense que notre phase de simulation pourrait être améliorer pour pouvoir avoir des résultats de simulations les plus proches de la réalité possible. Pour cela on pourrait notamment utiliser les heuristiques utilisées avec nos anciens agents, à condition de trouver une bonne combinaison des coefficients de sorte que le résultat de la simulation reflète dans la plupart des cas le résultat le plus probable.

Malgré les nombreuses pistes d'améliorations soulevées on constate que notre agent performe bien face aux autre agents, ainsi il serait donc intéressant de le revoir performer une fois les améliorations appliqués.

Liste de référence

[1] Alexander Brown J., Massagué Respall V. (2018). *Monte Carlo Tree Search for Quoridor* [Rapport technique]. ResearchGate. https://www.researchgate.net/publication/327679826_Monte_Carlo_Tree_Search_for_Quoridor

[2] Glendenning L. (2005). *Masterizing Quoridor* [Thèse de Doctorat, The university of New Mexico]. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.5204&rep=rep1&type=pdf

[3] Kocsis L., Szepesvári C. (2006). *Bandit Based Monte-Carlo Planning* [Rapport technique]. http://ggp.stanford.edu/readings/uct.pdf

[4] Mertens P.J. (2006). *A Quoridor-playing agent [Rapport technique]*. file:///Users/sanmarsimon/Downloads/Mertens_BSc-paper.pdf