IV – ALGORITHMES

12 – Monte-Carlo

12.1- Structure

Le Monte-Carlo Tree Search (abrégé MCTS) est un algorithme de recherche heuristique, donc un algorithme qui fournit rapidement une solution possible, sans pour autant que cette solution soit la meilleure ni la plus optimisée.

Le principe derrière le MCTS est une recherche dans un arbre contenant des nœuds qui représentent chacun une configuration possible du jeu ou de l’application dans laquelle est utilisé le MCTS, avec la racine (nœud en haut de l’arbre, n’ayant pas de nœud(s) parent(s)) comme configuration initiale, et les feuilles (nœuds tout en bas de l’arbre, n’ayant pas de nœud fils) comme configuration finale ou comme configuration n’ayant pas été explorée jusqu’au bout.

**Cet algorithme se décompose en 4 phases principales :**

* La phase de Sélection : (voir **1er** arbre sur le schéma ci-après)

Durant cette phase, l’objectif sera de sélectionner un nœud feuille prometteur. Pour cela on va descendre progressivement dans l’arbre en prenant soin de choisir à chaque fois, le « meilleur » nœud fils du nœud sur lequel on est actuellement.

Pour déterminer qui est le « meilleur » nœud fils, on applique une formule nommée **UCT** (**U**pper **C**onfidence Bound applied to **T**rees) qui va, en prenant en compte, leur chance respective d’aboutir à une victoire, et leur nombre respectif de fois où on les a visités, en sélectionner un. On répètera l’opération en appliquant à nouveau l’UCT sur ses fils, et ainsi de suite jusqu’à arriver en bas de l’arbre.

Cette formule permet de mettre en valeur les nœuds victorieux mais aussi les nœuds n’ayant pas été beaucoup visités, pour ne pas laisser de côté un nœud qui pourrait potentiellement par la suite, devenir victorieux.

* La phase d’Expansion : (voir **2eme** arbre sur le schéma ci-après)

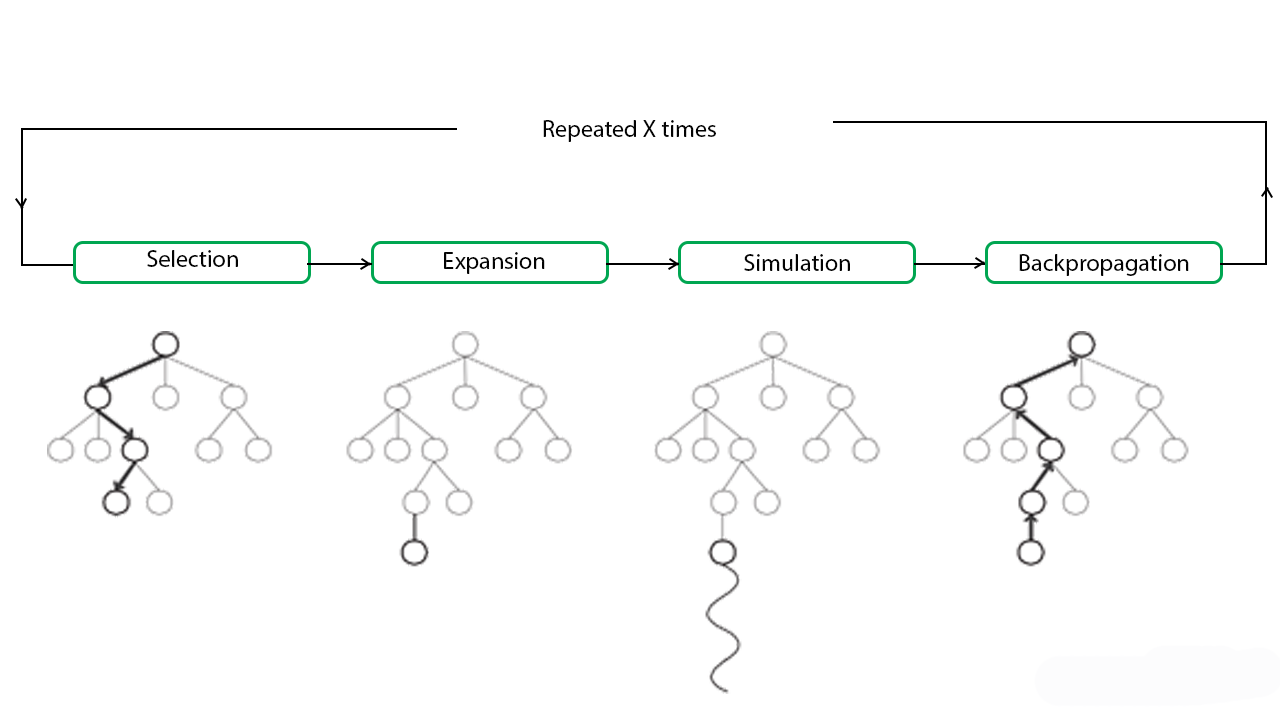
Une fois un nœud feuille choisi, nous allons dans cette phase l’expandre en construisant toutes les nœuds fils/configurations possibles à partir de celui-ci. Dans l’exemple du Meta-Morpion, on va construire, un nœud fils par coup possible à partir de la grille actuelle.

* La phase de Simulation : (voir **3eme** arbre sur le schéma ci-après)

Ensuite, on choisit un de ces nœuds fils au hasard (ou selon la méthode voulue), et on va simuler l’application de façon aléatoire jusqu’à son terme. C’est-à-dire pour un jeu, exécuter la partie jusqu’à ce qu’un des joueurs gagne ou qu’il y ait un match nul.

* La phase de BackPropagation : (voir **4eme** arbre sur le schéma ci-après)

Une fois cette simulation terminée, on remonte dans l’arbre jusqu’à la racine en actualisant les scores de chaque nœud : on incrémente le nombre de visite, et si la simulation nous indique une victoire, on incrémente également le nombre de victoire.



[Schéma récapitulatif de représentation du fonctionnement du Monte Carlo Tree Search]

Enfin, on renvoie le nœud ayant le meilleur score parmi les fils de la racine, car c’est « techniquement » la configuration qui a le plus de chance, pour le coup suivant, de nous amener vers une victoire.

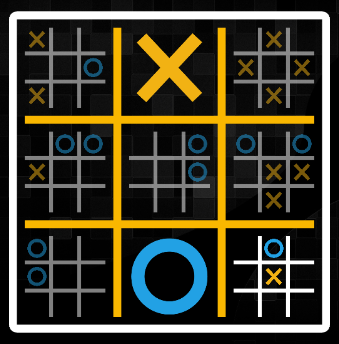
Le MCTS est un algorithme efficace, qui permet de toujours faire valoir un compromis entre les choix qui ont l’air prometteurs (grande chance de victoire) et les choix qui ont été peu explorés, desquels peu de simulations ont été effectuées.

Cependant, pour que le MCTS donne un résultat exploitable, il faut en exécuter un grand nombre, ainsi , on aura un arbre assez grand et des coefficients (nbr. victoires/ nbr. visites) plus précis sur chaque nœud.

12.2 - Déroulement

C’est donc sur la plateforme Coding Games que nous avons dû implémenter l’algorithme du MCTS, pour le jeu du Meta-Morpion.

Comme explique précédemment, le Meta-Morpion est une variante du célèbre jeu du Morpion, la différence avec celui-ci réside principalement dans la composition de la grille de 9 cases. Cette grille est en fait une « meta-grille » dont les cases contiennent chacune une grille classique de Morpion. Ce qui nous fait 9 grilles de Morpion, soit 81 cases au total. Afin d’apposer son symbole (croix ou rond) dans une des 9 cases de la « meta-grille », il faut gagner la grille contenue dans cette même case. Ainsi pour gagner la partie, il faut aligner 3 symboles dans la grille principale, et donc gagner dans 3 grilles « classiques » alignées (verticalement, horizontalement ou en diagonale).



[Grille type du jeu Meta-Morpion]

Nous devions donc créer une intelligence artificielle capable de jouer à ce jeu tout en utilisant l’algorithme Monte Carlo Tree Search.

Notre programme est décomposé comme suit :

* La fonction main :

Cette fonction s’articule autour de la boucle de jeu, boucle qui se lance à chaque fois que c’est à notre tour de jouer. En dehors de cette boucle, nous n’avons que quelques déclarations de variables, et notamment du nœud racine initial pour notre arbre. A l’intérieur de cette boucle, on commence par gérer la réception des entrées, c’est-à-dire la réception des coordonnées du dernier coup de l’adversaire. On s’occupe ensuite de vérifier si l’arbre existant **(1)** contient déjà un nœud avec comme dernier coup, le coup de l’adversaire. S’il existe **(2)**, on déclare ce dernier comme la nouvelle racine actuelle. S’il n’existe pas, on le génère et l’arbre existant est détruit car invalide.

1. car on conserve l’arbre généré par le MCTS entre chaque tour, pour gagner du temps et éviter de tout re-générer à chaque fois
2. ce qui est quasiment toujours le cas car on a déjà généré l’entièreté de la partie de l’arbre qui nous intéresse lors des tours précédents

On lance ensuite, dans une boucle, l’algorithme MCTS sur la nouvelle racine, et on boucle tant qu’on est encore en dessous de la limite de temps par tour imposée par la plateforme. Après cette boucle, on sélectionne le nœud fils ayant le meilleur score parmi ceux de la racine, et on renvoi les coordonnées de son dernier coup. Celui-ci correspondra au coup censé être le meilleur coup possible calculé par notre algorithme à ce moment-là du jeu.

* Les différentes classes : Pour intégrer un modèle commun et permettre une réutilisation de notre code pour le développement d’autres algorithmes (précisé plus haut), nous avons dû implémenter un système de classes. Ainsi notre algorithme s’articule autour de plusieurs classes et structures de donnees nécessaires à son fonctionnement et dont il est totalement dépendant. L’UML vu précédemment décris l’interaction et le lien entre ces classes. Elles possèdent chacune leurs attributs et méthodes respectifs. Certaines sont des classes abstraites, et/ou des classes template.
* Ensembles de fonctions pour le MCTS : Là est le cœur de l’algorithme, nous avons une fonction principale MonteCarloTreeSearch, orchestrant le déroulé précis et appelant les différentes fonctions associées à chaque phase de l’algorithme. Ces dernières effectuent ce pour quoi elles existent, remplissent leur rôle avant de renvoyer des valeurs

// … //

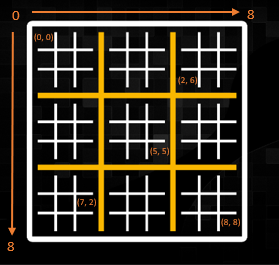
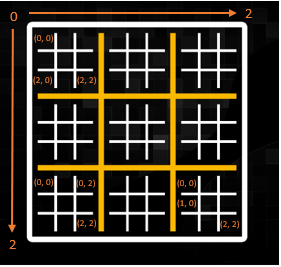
Partie de Mihai :

* 12.3 : performance et complexité
* 12.4 : progression et comparaison

// … //

12.5 - Débogage et difficultés rencontrées

Coding Games impose un format pour les entrées et sorties, en effet, la plateforme voit la grille du Meta-Morpion comme une seule et même grille de 81 cases, et note les coordonnées des cases comme suit : (x = [0, 8], y = [0, 8]) **(voir illustration ci-dessous, à gauche)**. Or, notre structure de classes étant basée sur une « Meta » grille de 3 par 3, contenant dans chacune de ses cases, une grille de Morpion classique, elle-même de 3 par 3 **(voir illustration ci-dessous, à droite)**, il y avait donc un problème de concordance des coordonnées. Ceci nous enjoint à convertir à plusieurs reprises dans le code, les coordonnées, que ce soit dans un sens ou dans l’autre.



[Structuration des coordonnées de la grille et des sous-grilles par notre algorithme (grille principale 3x3 contenant des grilles de 3x3)]

[Structuration des coordonnées de la grille par Coding Games]

**conversion**

Une image contenant texte

Description générée automatiquement

Ce projet a premièrement nécessité de notre part que nous approfondissions nos connaissances dans le langage C++, notamment pour la Programmation Orientée Objet Avancée ou encore le concept de pointeurs intelligents. Etant habitué à utiliser le langage de programmation Java, passer au C++ qui est un langage de plus bas niveau et donc par définition moins facile à appréhender, n’a pas été une mince affaire.

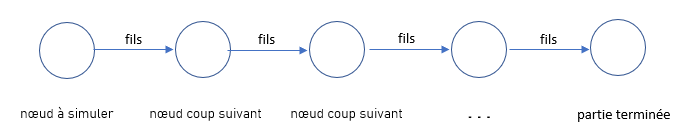
Aussi, la plateforme Coding Games impose des contraintes de temps pour les tours de chaque joueur : 1000 millisecondes (= 1 sec) pour le premier tour et 100ms pour les tours suivants. Le but, comme dit précédemment, est de faire le plus de MCTS possible par tour, ceci nous mena donc à essayer de faire le code le plus rapide et efficace possible. Nos efforts ont permis de passer de quelques dizaines de MCTS par tour, à plusieurs centaines et même plusieurs milliers en fin de partie. Grâce notamment à une meilleure gestion de la boucle while MCTS, conditionnée par un chrono vérifiant de ne pas dépasser le temps limite.

Une image contenant texte, capture d’écran, écran

Description générée automatiquement[Boucle MCTS principale]

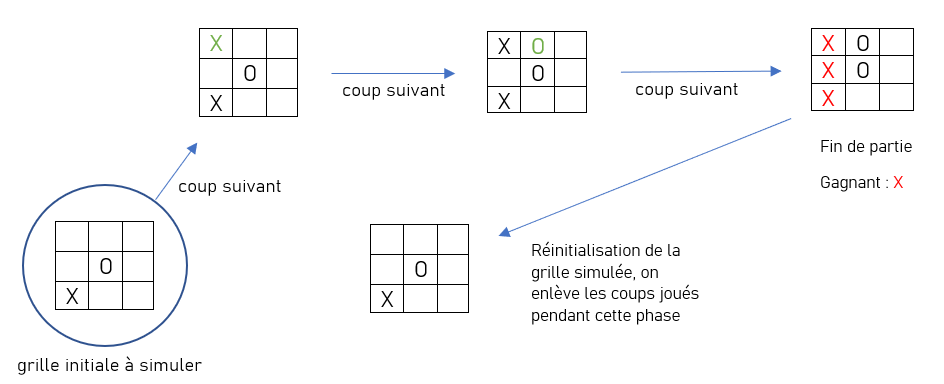
Nous avons aussi dû au fur et à mesure simplifier certaines structures de donnees, et certaines fonctions. Quelques-unes de nos tactiques ont ainsi aussi été revues, en voici un exemple :

Pour la fonction gérant la phase de simulation *(rappel : l’objectif de cette phase est de simuler à partir d’un nœud choisi, le reste de la partie aléatoirement jusqu’à son terme, et ensuite de faire disparaitre ces coups car ils ne sont que temporaires)* ; Notre première tactique a été de générer un nœud pour chaque coup (fils du nœud du coup précédent) jusqu’à la fin de la partie. Et une fois le statut de fin récupéré, nous détruisions ces nœuds.



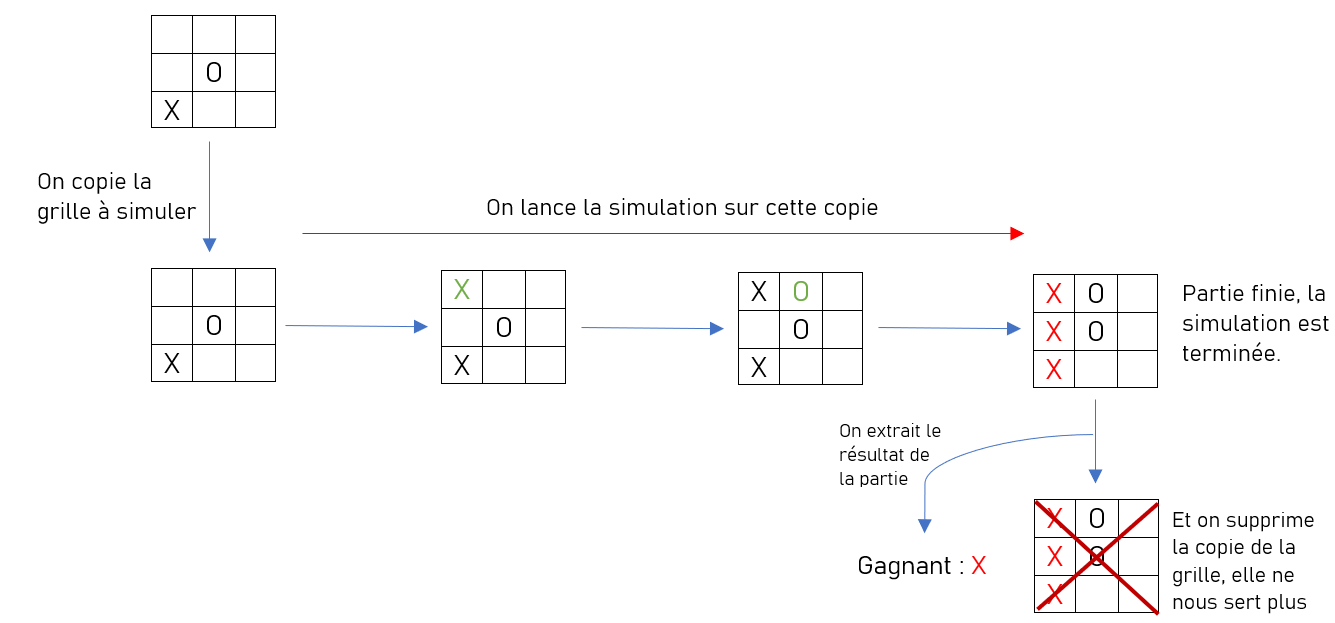
[Schéma explicatif de notre première approche du problème]

Cependant cette méthode prenait beaucoup trop de temps, nous sommes donc passé à une autre façon de faire : dans le nœud ciblé par la simulation, nous génèrerions aléatoirement le reste de la partie tout en mémorisant les coups simulés dans une liste, pour pouvoir ensuite les « annuler » (undo) et ainsi revenir à l’état antérieur du nœud avant simulation.



[Schéma explicatif de notre deuxième approche du problème]

Mais encore une fois, nous devions trouver une technique plus rapide, nous avons opté finalement, pour la méthode la plus simple mais qui nous avait échappée, celle de prendre une copie de la grille à simuler, de faire nos modifications dessus, et une fois le résultat acquis, détruire entièrement cette copie. Ceci évite de gaspiller des ressources dans la gestion des coups à annuler.



[Schéma explicatif de notre troisième approche du problème]

* Les fonctions servant à renvoyer les coups possibles et valides ont été revues, en effet, nous les avons raccourcies, le cout des opérations sur celles-ci a donc été réduit.

Pour annexes : Liste fonctions/méthodes

(les constructeurs et accesseurs/mutateurs classiques sont volontairement omis pour éviter une surcharge)

|  |  |  |  |
| --- | --- | --- | --- |
| **Nom fonction/méthode** | **Liste paramètres** | **Type de retour** | **Brève description** |
| getValue | entier i | Type template | Renvoi l’élément du conteneur à l’indice ‘**i**’ |
| getValueXY | entier i, entier j | Type template | Renvoi l’élément du conteneur à l’indice (‘**i**’, ‘**j**’) |
| setValue | entier i, type\_template value |  | Remplace l’élément du conteneur à l’indice ‘**i**’ par la valeur ‘**value’** |
| setValueXY | entier i, entier j, type\_template value |  | Remplace l’élément du conteneur à l’indice (‘**i**’, ‘**j**’) par la valeur ‘**value’** |
| getEmptyPositions |  | vector<Position> | Renvoi une liste de Positions vide correspondant aux cases vides de la grille visée |
| checkStatus |  | int | Renvoi le statut actuel de la grille visée  (1 ou 2 = le joueur 1/2 est gagnant ; 0 = égalité ;  -1 = partie en cours) |
| getValidActions |  | vector<Position> | Renvoi la liste des positions valides (que l’on a le droit de jouer) de la grille cible |
| getRandomChildNode |  | shared\_ptr<Node> | Choisi aléatoirement un nœud fils parmi la liste des fils du nœud cible |
| getChildWithMaxScore |  | shared\_ptr<Node> | Renvoi le nœud fils ayant le plus haut score « victoires/visites » parmi les fils du nœud cible |
| getAllPossibleStatesEnhanced |  | vector<State> | Renvoi une liste de State ( grille) possibles à partir du nœud cible (nécessaire à la phase d’expansion) |
| MonteCarloTreeSearch | shared\_ptr<Node> root, int playerNo |  | Lance l’algorithme du MCTS sur le nœud ‘**root**’ avec ‘**playerNo**’ indiquant si on joue en premier ou pas |
| selectPromissingNode | shared\_ptr<Node> root | shared\_ptr<Node> | Applique la phase de sélection sur ‘**root**’ |
| findBestNodeWithUCT | shared\_ptr<Node> node | shared\_ptr<Node> | S’occupe de la partie calcul de l’UCT |
| expandNode | shared\_ptr<Node> node |  | Applique la phase d’expansion sur ‘**node**’ |
| simulateRandomPlayout | shared\_ptr<Node> selectedNode | int | Applique la phase de simulation sur le nœud ‘**selectedNode’** |
| backPropagation | shared\_ptr<Node> node, int playerNo, int randomPlayout |  | Applique la phase de BackPropagation en remontant l’arbre à partir de ‘**node**’ |

