Devoir Maison IA Solve :
Utilisation de MCTS sur un jeu Othello 6 × 6

Chaimae EL HOUJJAJI - Kexin LI - Pauline TURK

10 janvier 2022

1 Jeu Othello

1.1 Présentation du jeu

Othello est un jeu qui se joue à 2 joueurs, un joueur avec des pions blancs et un joueur avec des pions noirs. Dans le cadre de notre projet on considèrera deux joueurs : `X` et `O`. C'est un jeu qui se joue habituellement sur un plateau de taille 8 x 8 que l'on nomme un othellier. Nous décidons de l'adapter à un plateau de taille 6 x 6.

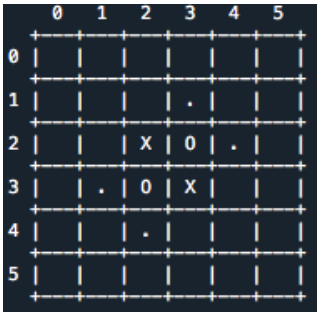
1.2 Ojectifs du jeu

Le gagnant est le joueur qui dispose du plus de pions sur le plateau à la fin de la partie. La partie se finit lorsque plus aucun des deux joueurs ne peut jouer, ce qui correspond dans la majorité des parties à un plateau rempli de pions.

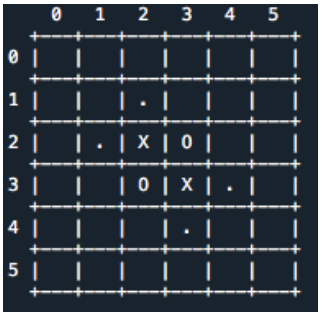
1.3 Règle du jeu

Initialisation du jeu
À l'état initial, 4 pions sont posés sur le plateau. Aux positions [2,2] et [3,3] pour le joueur `X` et [2,3] et [3,2] pour le joueur `O` comme illustré sur la Figure 1a. Les joueurs jouent de manière alterné. À chaque tour, ils ont un certain nombre de positions qui leurs sont permises. Par exemple pour le premier tour on voit sur la Figure 1b les positions autorisées pour le joueur `X` qui sont marquées par un `"."` si c'est lui qui joue en premier. Sur la figure 1c, on peut voir les positions autorisées pour le joueur `O` s'il joue en premier .

(a) Position initiale



(b) Positions légales X



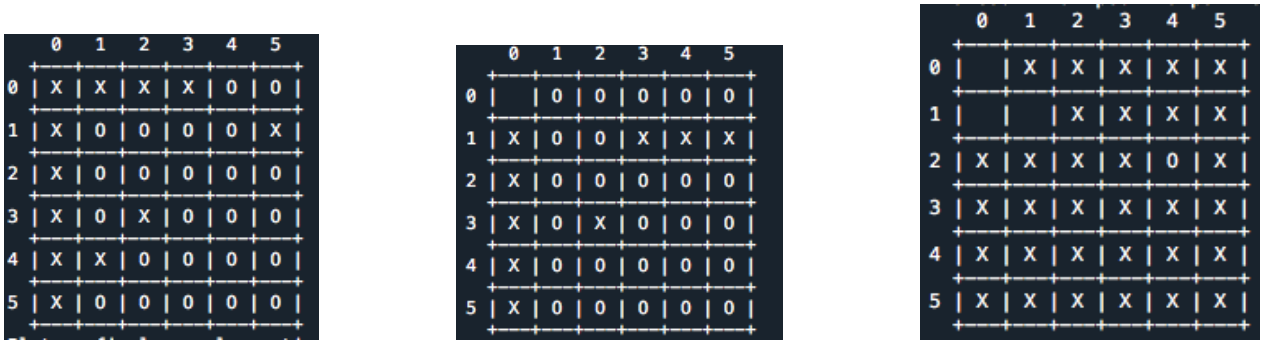
(c) Positions légales O

FIGURE 1 – Départ du Jeu

Les positions autorisées correspondent aux cases vides qui sont adjacentes aux pions du joueur adverse et qui permettent d'encadrer un ou plusieurs pions adverses entre un pion déjà sur le plateau et le pion qui vient d'être déposé. Ces pions encadrés deviennent donc de la même couleur que celle du joueur qui vient de jouer. On peut encadrer dans plusieurs directions en un coup, soit horizontalement, verticalement et/ou en diagonale.

Fin de la partie

Une partie se termine lorsque aucun des deux joueurs ne peut plus jouer ce qui correspond à aucune position légale qui permettent un retournement de pions. On représente sur la Figure 2 trois situations finales de jeu possibles.



(a) Toutes les cases occupées (b) 1 case non occupée (c) 3 cases non occupées

FIGURE 2 – Exemples de situations de fin de jeu

Pour avoir une description plus complète des règles du jeu, cliquez ici.

2 Matériel, ressources et emplacement des codes

Pour coder ce jeu nous avons utilisé deux sources :

- Une partie de l'algorithme décrit sur cette page que l'on a complété avec certaines fonctions pour coder l'algorithme du MCTS.
- Une partie du code décrit à l'url à cette page pour définir les règles du jeu Othello.

L'ensemble de nos codes sont disponibles sur ce répertoire Github qu'il faudra importer. Celui-ci comprend les fichiers suivants, dont la structure est représentée dans la Figure 3 :

- Un fichier `fonction00.py` qui définit les règles du jeu Othello à l'aide de fonctions.
- Un fichier `class00.py` qui décrit l'algorithme du MCTS et qui fait appel au fichier `fonction00.py`.
- Un fichier `main00_2ordi.py` qui définit les fonctions permettant de simuler une partie entre deux ordinateurs. Avant de l'exécuter, on peut modifier dans le code le nombre de parties et choisir les paramètres *no_simulation* et *c_param* des deux joueurs. À la fin, un graphique de répartition des gains est généré. Ce fichier importe les fonctions du fichier `fonction00.py` et la classe MonteCarloTreeSearchNode du fichier `class00.py`.
- Un fichier `main00_humain_ordi.py` qui définit la fonction permettant de simuler un jeu entre un ordinateur et un humain. De même, avant de l'exécuter, on peut modifier dans le code le nombre de parties et choisir les paramètres *no_simulation* et *c_param* de l'ordinateur pour déterminer la difficulté de la partie. Ce fichier importe les fonctions du fichier `fonction00.py` et la classe MonteCarloTreeSearchNode du fichier `class00.py`.
- Un fichier `test_parametres.py` qui permet d'analyser le jeu entre deux ordinateurs en modifiant les deux paramètres : le nombre de simulation ainsi que le paramètre d'exploration que nous définirons plus bas. En exécutant ce fichier, celui-ci tourne pendant 9 heures et donne en sortie un graphique dans un fichier .png que nous avons inséré dans la partie 4. Ce fichier importe les fonctions du fichier `main00_2ordi.py`.

Une fois le répertoire importé, en fonction de l'utilisation souhaitée, différents lancements doivent être opérés :

- Pour affronter un ordinateur, il faudra exécuter le fichier `main00_humain_ordi.py`.
- Pour se faire affronter 2 ordinateurs, il faudra exécuter le fichier `main00_2ordi.py`.

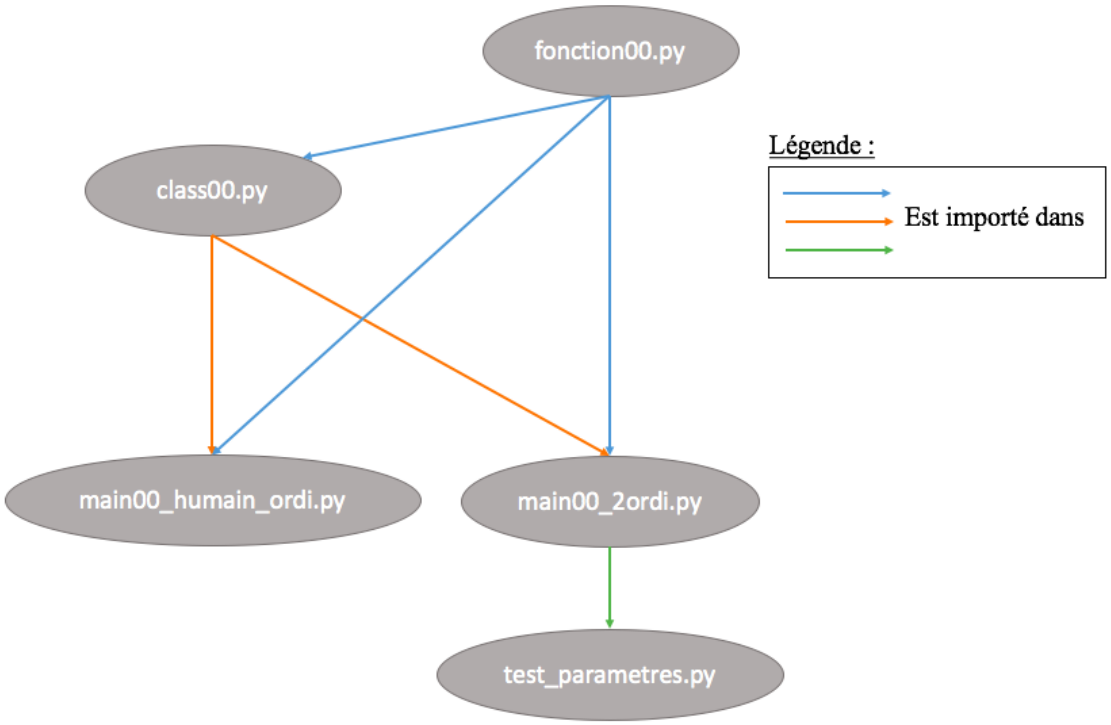


FIGURE 3 – Structure des fichiers python entre eux

3 Algorithme MCTS et explication du fichier class00.py

Nous souhaitons programmer ce jeu en utilisant l'algorithme MCTS. Nous avons codé deux versions du jeu : une première version où l'ordinateur joue contre un humain en mode console. Ce code est disponible dans le fichier main00_humain_ordi.py. Nous avons également codé une deuxième version où deux ordinateurs jouent l'un contre l'autre et où le code est disponible dans le fichier main00_2ordi.py . C'est cette dernière version qui nous servira par la suite pour analyser les performances des joueurs lorsque l'on change certains paramètres.

3.1 Algorithme de Recherche arborescente Monte-Carlo ou Monte Carlo tree search (MCTS) tel qu'implémenté dans ce projet

L'algorithme MCTS permet d'explorer l'arbre des possibles. Avant chaque coup joué, il va évaluer l'état du jeu en simulant des parties jouées aléatoirement jusqu'à leur fin. En se basant sur les résultats (gain/ défaite/nul) de ces parties simulées, l'algorithme choisit le coup qui lui permettra les meilleurs résultats globaux connus. Le MCTS utilise la méthode de Monte Carlo puisqu'il choisi au hasard des coups parmi l'espace de décision. En terme de vocabulaire, la racine est la configuration initiale du jeu. Chaque nœud est une configuration de jeu et ses enfants sont les configurations suivantes. Une feuille est dite terminale si elle correspond à une fin de partie i.e soit c'est un nul soit on sait quel joueur a gagné. Sinon la feuille correspond à un noeud de l'arbre de recherche qui n'a pas encore été développé. Chaque noeud comporte le nombre de parties simulées auxquelles il a participé, ce qui correspond au nombre de visites. En effet, à chaque simulation de partie depuis une feuille, on lui incrémente de 1 le nombre de parties simulées ainsi qu'à tous ses ascendants c'est à dire les noeuds parents de sa branche. Chaque noeud comporte aussi le nombre de ses simulations gagnantes. Chaque itération de l'algorithme MCTS suit les quatres étapes suivantes :

- **Sélection**
- **Expansion**
- **Simulation**
- **Rétropropagation**

L'algorithme du MCTS a été implémenté en choisissant une programmation orientée objet et en définissant une classe `MonteCarloTreeSearchNode` dans le fichier `class00.py` qui se base sur les règles du jeu définies dans le fichier `fonctions00.py`.

Expansion et sélection

Le noeud de départ sur lequel s'applique l'algorithme de MCTS est celui de la configuration de partie actuelle. S'il n'est pas terminal et pas complètement développé, alors l'algorithme choisit aléatoirement une action parmi celles qui lui sont encore permises et crée le noeud enfant correspondant. Ce noeud enfant sera alors sélectionné pour les étapes suivantes de simulation et de rétropropagation. Si ce noeud de départ est totalement développé i.e tous les noeuds enfants ont été explorés au moins une fois, l'algorithme doit choisir parmi l'ensemble des noeuds qu'il a sa disposition. Pour cela il rencontre un dilemme entre exploitation des noeuds qui ont l'air plus prometteurs et exploration d'autres noeuds où peu de simulations ont été réalisées. Pour tenir compte de ce dilemme, le noeud sélectionné pour les étapes suivantes et donc considéré comme le meilleur sera le noeud enfant d'UCB max (formule ci-dessous).

$$UCB = \frac{w_i}{n_i} + c\sqrt{\frac{ln(N)}{n_i}}$$

Où l'on a :

- w_i, le nombre de parties gagnées par le ième noeud dans les noeuds enfants.
- n_i, le nombre de fois où le successeur i a été visité.
- N, le nombre de fois où le noeud, père de i a été visité.
- c, le paramètre d'exploration, en théorie égal à $\sqrt{2}$, en pratique, choisi expérimentalement.

Le premier terme correspond au terme d'exploitation et le deuxième terme à celui d'exploration.

Méthode ._tree_policy(c_param) pour les étapes d'expansion et de sélection :

```
    def _tree_policy(self,c_param):
        current_node = self                              # état actuel de la partie
        while not current_node.is_terminal_node():       # si le noeud n'est pas terminal
            if not current_node.is_fully_expanded(): # s'il reste des enfants non visités
                return current_node.expand()         # développer un nouvel enfant
                                                     # et le sélectionner
            else:                                    # sinon
                current_node = current_node.best_child(c_param) # sélection du meilleur
                                                                # enfant connu d'UCB max
                return current_node
```

Méthode .expand() pour le détail de l'étape d'expansion :

```
    def expand(self):
        action = self._untried_actions.pop()               # sélection d'un nouveau coup
        next_state = self.getChildState(action, self.tile)
        if self.tile == 'X':
            child_tile = 'O'
        else:
            child_tile = 'X'
        child_node = MonteCarloTreeSearchNode(next_state, tile=child_tile, parent=self, \
        parent_action=action)            # générer le noeud enfant associé
        self.children.append(child_node) # l'ajouter à la liste des noeuds enfants connus
        return child_node
```

Simulation

À partir de ce noeud enfant sélectionné, l'algorithme va simuler une exécution de partie au hasard en choisissant aléatoirement parmi les coups possibles jusqu'à atteindre une fin de partie. Puis le résultat de cette partie est récupéré (victoire, perte ou nul).

Méthode .rollout() pour l'étape de simulation :

```
    def rollout(self):
        current_rollout_state = getBoardCopy(self.state)
        current_tile = self.tile

        # tant que la partie simulée avec le noeud enfant sélectionné n'est pas terminée
        while not is_game_over(current_rollout_state):
            # récupérer tous les coups possibles
            possible_moves = get_legal_actions(current_rollout_state, current_tile)
            if possible_moves !=[] :
                # choisir un coup possible au hasard
                action = possible_moves[np.random.randint(len(possible_moves))]
                # jouer ce coup dans la partie simulée
                flipTiles(current_rollout_state, current_tile, action[0], action[1])

            if current_tile == 'X':
                 current_tile = 'O'
            else:
                 current_tile = 'X'
        # récupérer les résultats de la partie simulée
        resultats = getScoreOfBoard(current_rollout_state)
        return self.game_result(resultats)
```

Rétropropagation

Une fois la partie simulée terminée, les statistiques du noeud enfant sélectionné et de tous ses ascendants sont mises à jour, c'est à dire le nombre de parties simulées et le nombre de ses parties gagnantes.

Méthode .backpropagate(result) pour l'étape de rétropropagation :

```
    def backpropagate(self, result):
        self._number_of_visits += 1.  # nombre de visites du noeud
        self._results[result] += 1.   # nombre de victoires
                                      # Rq. dans cet implémentation de MCTS
                                      # on compte aussi le nombre de défaites
        if self.parent:               # si le nœud a un parent i.e ce n'est pas la racine
            self.parent.backpropagate(result) # récursivité sur le self.parent
                                              # pour remonter la branche
```

Sélection du meilleur coup à jouer

Une fois le nombre de simulations atteint. On retourne le meilleur noeud enfant, c'est à dire celui qui possède le rapport $\frac{w}{n}$ le plus élevée. Pour sélectionner ce meilleur noeud on prend `c_param = 0` puisqu'on veut exploiter l'information de meilleur coup à jouer connu, on ne cherche plus à explorer l'arbre.

Méthode .best_action(simulation_no,c_param) pour sélectionner le meilleur coup connu

```
    def best_action(self, simulation_no, c_param):
        for i in range(simulation_no):     # effectuer le nombre de simulations fixé
            v = self._tree_policy(c_param) # selection/expansion
            reward = v.rollout()           # simulation
            v.backpropagate(reward)        # rétropropagation

        return self.best_child(c_param=0.) # meilleur coup connu
```

3.2 Conclusion du MCTS

Le MCTS, est un algorithme qui tend vers le MinMax à mesure que l'on augmente le nombre de simulations. Toutefois, il n'explore pas autant de noeuds que le MinMax.

4 Analyse de la modification des paramètres

Dans l'implémentation de l'algorithme MCTS, on peut jouer sur deux paramètres, le nombre de simulations `no_simulation`, ainsi que le paramètre d'exploration `c_param`.

`no_simulation`
Le nombre de simulations correspond au nombre d'itérations de l'algorithme MCTS avant de décider du prochain coup à jouer et donc du nombre de parties qui vont être simulées pour évaluer les coups possibles. On s'attend à avoir une performance (capacité à gagner une partie) d'autant meilleure que ce nombre de simulation est élevé. Ce lien p.27 paragraphe 1.4.1 précise une limite de ce comportement. En effet, à partir d'un certain nombre de simulations, une augmentation exponentielle de celui-ci est attendu pour continuer d'observer des améliorations de performance.

`c_param`
En modifiant ce paramètre, on accorde plus ou moins d'importance à l'exploration ou à l'exploitation lors de l'étape de sélection.

- Lorsque `c_param = 0` : on n'a que de l'exploitation, c'est à dire qu'on sélectionne à chaque fois le noeud le plus prometteur sans explorer les autres noeuds.
- Plus `c_param` augmente, plus on accorde de l'importance à l'exploration.

La Figure 4 permet d'estimer l'influence de chacun de ces paramètres. Elle a été générée par l'exécution du fichier `test_parametres.py`.

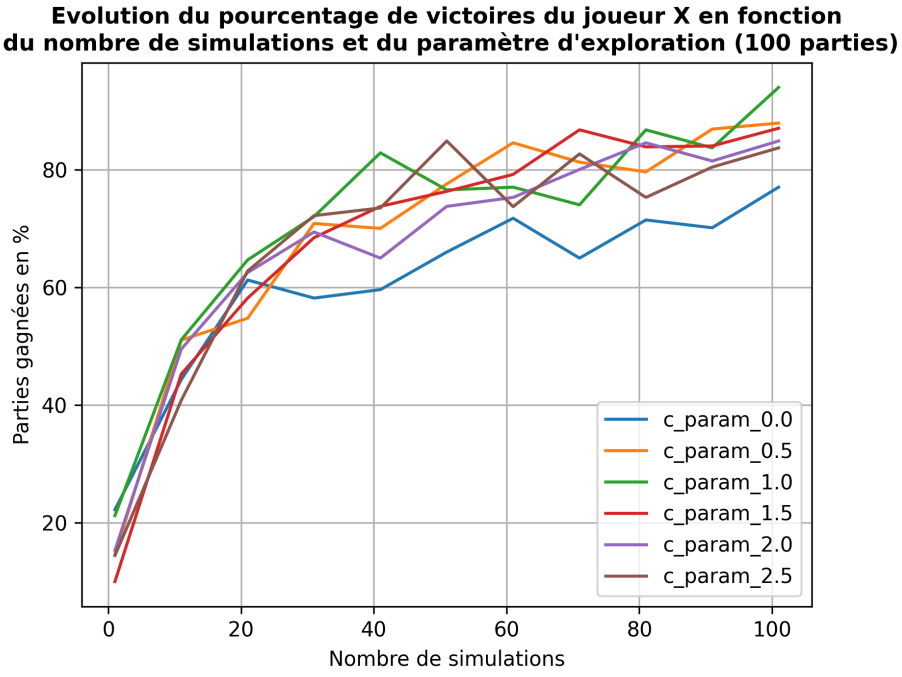


FIGURE 4 – Variation du paramètre d'exploration et du nombre de simulations pour une centaine de parties jouées

Un grand nombre de parties ont été jouées entre deux ordinateurs. Un qui joue toujours avec 'X' et l'autre toujours avec 'O'. Pour évaluer les performances d'un joueur, nous avons décidé de faire varier les paramètres de l'ordinateur 'X' que l'on fait jouer contre l'ordinateur 'O' dont les paramètres ont été fixés. Ainsi :

Ordinateur 'X' :
- `no_simulation` : varie de 1 (valeur minimale possible) à 101 avec un pas de 10.
- `c_param` : varie de 0 (cas sans exploration) à 2,5 par pas de 0,5.

Ordinateur 'O' :
— `no_simulation` : fixé à 11.
— `c_param` : fixé à 1.5 (proche de la valeur par défaut de $\sqrt{2}$).
— **Rq.** l'ordinateur 'O' a été choisi pas trop mauvais afin de bien visualiser la progression de 'X' lorsqu'on augmente la valeur de ses paramètres.

Parties gagnées en % (par 'X') :
— Ce pourcentage est utilisé pour quantifier la qualité de jeu de l'ordinateur 'X'. Il est défini par la formule suivante :

$$PartiesGagnées\% = \frac{PartiesGagnées}{PartiesJouées - PartiesNulles} * 100$$

— **Rq.** Le nombre de parties nulle a été soustrait du nombre de parties jouées pour qu'une baisse du pourcentage de parties gagnées par 'X' ne soit possible qu'en cas d'une augmentation du nombre de défaites de 'X' et non potentiellement de plus de parties nulles.
— Le pourcentage de parties gagnées a été estimé par 100 parties jouées entre l'ordinateur 'X' pour chaque couple de valeurs que ces paramètres peuvent prendre et l'ordinateur 'O' (qui a le même niveau à chaque partie).

Principales observations de la Figure 4 :
— On retrouve comme attendu une grande amélioration des performances de jeu avec l'augmentation du nombre de simulations. En effet, un plus grand nombre de simulations permet à l'agent d'apprendre une plus grande quantité d'informations et donc d'améliorer sa capacité de jeu. Il semble même qu'un pallier de performance soit atteint vers 80% de gain à partir d'un nombre de simulations d'environ 60.
— Ces performances sont aussi globalement améliorées avec l'augmentation du paramètre d'exploration.
— La courbe bleue avec un paramètre d'exploration nul, se distingue des autres par une performance moindre qui n'est toujours pas compensée par le nombre de simulations maximal testé de 101.
— Sur la courbe rouge avec un paramètre d'exploration de 1,5, pour un nombre de simulations de 11, les ordinateurs 'X' et 'O' ont les même valeurs de paramètres. On observe en ce point un pourcentage proche de 50%, ainsi, à leur niveau de jeu, il semblerait que commencer la partie ne donne pas particulièrement d'avantage.
— L'intervalle de valeurs testés pour chaque paramètre est assez restreint. Il aurait été pertinent d'élargir cet intervalle afin de mieux évaluer l'influence de ces paramètres sur les performances de l'algorithme. Toutefois, générer ce graphe a nécessité environ 9h de temps d'exécution, donc pour tester de plus larges intervalles de valeurs en un temps de calcul raisonnable, il serait pertinent de paralléliser l'exécution des parties d'Othello.