Des casinos à l'intelligence articficielle

TER M1 DS 2021

Adrien Maitammar
Maxime Le Paumier

15 mai 2021

Sommaire

1 Introduction

Le premier objectif de ce projet est l'étude de l'algorithme UCB1 (Upper Confidence Bound 1). Cet algorithme a été proposé en 2002 dans le cadre du problème du bandit à K bras.

1.1 Le problème du bandit à K bras

Le problème du bandit manchot ou, dans sa version générale, du bandit à K bras, peut se formuler ainsi : un agent est face à des machines à sous dont il ne connait pas les récompenses moyennes. Son objectif est alors de maximiser ses gains.

Ce problème est en fait un exemple d'apprentissage par renforcement dans lequel l'agent doit faire des compromis entre l'exploitation (de la machine qui empiriquement lui rapporte le plus) et l'exploration (pour espérer trouver une machine rapportant plus).

L'agent peut bien sûr adopter une approche naïve et jouer la machine maximisant la récompense moyenne sur les coups joués mais cette stratégie peut le conduire à ignorer certaines machines, qui pourraient être plus intéressantes à jouer.

Peter Auer, Nicolò Cesa-Bianchi et Paul Fischer développent dans leur article *Finite-time Analysis of the Multiarmed Bandit Problem* une stratégie permettant à l'agent de continuer à explorer les machines tout en maximisant ses gains. Plutôt que de procéder naïvement, celui-ci joue les machines selon la procédure UCB1 ci-dessous.

UCB1
Initialisation : Jouer chaque machine une fois
Boucle : Jouer la machine j qui maximise $\bar{x_j} + \sqrt{2\frac{\ln n}{n_j}}$ avec $\bar{x_j}$ la récompense moyenne de la machine j jusqu'à présent, n_j le nombre de fois où la machine j a été jouée et n le nombre de coups total.

FIGURE 1 – Algorithme UCB1

On retrouve dans cette quantité un terme d'exploitation (la moyenne des gains de la machine) ainsi qu'un terme d'exploration donnant plus d'importance aux machines ayant été visitées moins de fois pondéré par $\sqrt{2}$, que l'on appelle constante d'exploration (sur laquelle nous aurons à revenir dans le cadre du jeu de Hex).

1.2 Le jeu de Hex

Dans le cadre de ce projet, nous avons cherché à appliquer cet algorithme à un joueur artificiel du jeu de Hex. Il s'agit d'un jeu de société pour deux joueurs dont le but est de relier les deux côtés du plateau correspondant à sa couleur par une ligne continue composée de pièces hexagonales. Avec des règles pourtant simples, la complexité de ce jeu est comparable à celle des échecs, dans le sens où les nombres de coups possibles sont comparables.

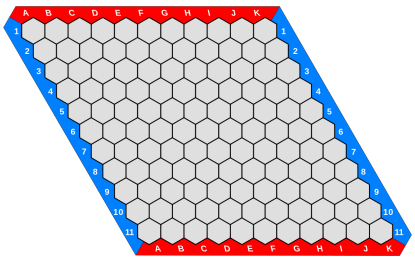

FIGURE 2 – Plateau de jeu 11x11

Nous verrons dans un premier temps une manière d'implémenter ce jeu sous python puis nous détaillerons deux méthodes de programmation d'un joueur artificiel : une méthode Monte-Carlo puis une méthode UCB appliqué à une recherche arborescente Monte-Carlo que nous appellerons UCT (Upper Confidence bound applied to Trees).

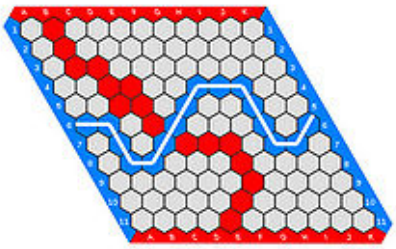

FIGURE 3 – Configuration gagnante pour le joueur bleu

2 Implémentation du jeu de Hex

Nous avons utilisé pour ce projet le langage python. Notre implémentation du jeu de Hex utilise le module pygame pour l'affichage et les interactions avec les joueurs humains. Le jeu repose sur une classe "Board" représentant le plateau.

```
class Board:

    def __init__(self, board_size):
        self.size = int(board_size)
        self.board = [ [0 for i in range(self.size)] for j in
            range(self.size)]
        self.actions = [(i,j) for i in range(self.size) for j in
            range(self.size)]

        # Composantes connexes initiales
        self.east_component = set([(i,self.size) for i in range(self.size)])
        self.west_component = set([(i,-1) for i in range(self.size)])
        self.north_component = set([(-1,i) for i in range(self.size)])
        self.south_component = set([(self.size,i) for i in
            range(self.size)])

        self.components = [[self.north_component, self.south_component],
            [self.west_component, self.east_component]]

        ...
```

2.1 Le problème de la fin de la partie

Le principal problème de cette implémentation a été de trouver un moyen assez rapide (pour ne pas ralentir les simulations) de déterminer s'il y avait un gagnant ou si la partie pouvait continuer. Après une tentative avec la théorie des graphe et l'algorithme de Dijkstra, nous avons opté pour une méthode utilisant les composantes connexes du plateau. En effet, on considère qu'un joueur gagne s'il relie par une ligne continue les deux bords opposés du plateau, c'est-à-dire si les composantes connexes artificielles Nord et Sud ou Est et Ouest sont des sous ensembles d'une même composante connexe.

```
def check_win(self, currplayer):
        """
        Checks if the current player won the game.
        Returns the winner's name if there is any or None if there is none.
        1 : red player
        2 : blue player
        """

        for component in self.board.components[currplayer.color - 1]:
            if currplayer.color == 1:
                if self.board.north_component.issubset(component) \
                    and self.board.south_component.issubset(component):
                    return currplayer
            else: # currplayer.color == 2
                if self.board.west_component.issubset(component) \
                    and self.board.east_component.issubset(component):
```

```
                return currplayer
        return None
```

3 La méthode Monte-Carlo

3.1 Bases de la méthode

La première méthode que nous avons programmé est une méthode naïve et empirique. Elle consiste simplement à simuler un certain nombre de parties à partir de chaque action restante pour déterminer le meilleur coup à jouer.

3.2 L'algorithme de sélection du meilleur coup

```
def search(self, initialState):
        self.root = treeNode(initialState, None)
        actions = self.root.state.actions

    #On copie l'etat actuel du plateau
        for action in actions:
            root_state = deepcopy(self.root.state)
            root_state.takeAction(action, root_state.currplayer)
            node = treeNode(root_state, self.root)
            self.root.children[action] = node
            self.executeRound(node)

     #On simule un certain nombre de parties sur chaque action possible
        nb_iter = int(self.searchLimit / len(actions))
        for child in self.root.children.values():
            for i in range(nb_iter):
                self.executeRound(child)

     #On selectionne l'action menant au plus haut taux de victoire
        bestChild = self.getBestChild(self.root)
        action = (action for action, node in self.root.children.items() if
             node is bestChild).__next__()

        return action
```

4 La méthode UCT

La méthode Monte-Carlo, bien que relativement efficace, présente pour principal défaut un nombre de calcul très important. Là où cette dernière simule tous les coups possibles, on va plutôt chercher à ne simuler que certains coups, choisis par le critère UCB1, dans le cadre d'une recherche arborescente Monte-Carlo.

4.1 Recherche arborescente Monte-Carlo

La recherche arborescente Monte-Carlo ou *Monte Carlo tree search* (MCTS) est un algorithme de prise de décision principalement utilisé dans les jeux [1]. Le principe de cet algorithme consiste à parcourir l'arbre des possibles dont la racine est l'état actuel du jeu. Chaque noeud est donc une configuration possible et ses enfants représentent les configurations suivantes. Ainsi, on va simuler un certain nombre de coups pour trouver le plus intéressant à jouer.

4.2 Implémentation de la méthode UCT

La première étape consiste à sélectionner un noeud à partir d'un noeud parent. Dans la méthode UCT, introduite par Levente Kocsis et Csaba Szepesvàri, on utilise le critère UCB1, c'est-à-dire que l'on va choisir le noeud qui maximise $\frac{w}{n} + \sqrt{2\frac{\ln N}{n}}$ où w est le nombre de parties gagnées après avoir joué ce coup, n le nombre de fois où l'on a joué ce coup et N le nombre de fois où l'on a joué le coup du noeud parent.

```
bestValue = float("-inf")
bestNodes = []

for child in node.children.values():
   nodeValue = child.totalReward / child.numVisits + explorationValue *
       sqrt(log(node.numVisits) / child.numVisits)

       if nodeValue > bestValue:
          bestValue = nodeValue
          bestNodes = [child]
        elif nodeValue == bestValue:
          bestNodes.append(child)

return random.choice(bestNodes)
```

Une fois le noeud sélectionné, et si ce noeud n'est pas un noeud de fin de partie, on procède à une phase d'expansion, lors de laquelle on va choisir aléatoirement un coup à jouer et créer un noeud enfant en conséquence. On va ensuite simuler une partie une partie jusqu'à la fin pour obtenir un gagnant. Enfin, on utilise ce résultat lors de la phase de rétropropagation pour mettre à jour les scores des joueurs dans l'arbre.

4.3 Recherche de la constante d'exploration optimale

Nos premiers résultats ont montré que l'algorithme UCT ne gagnait que peu de parties contre l'algorithme Monte-Carlo. On peut expliquer ce défaut par le fait que

1. On peut citer son utilisation dans certains programmes de Go, d'échecs ou de shogi, ou encore le jeu Total War : Rome II

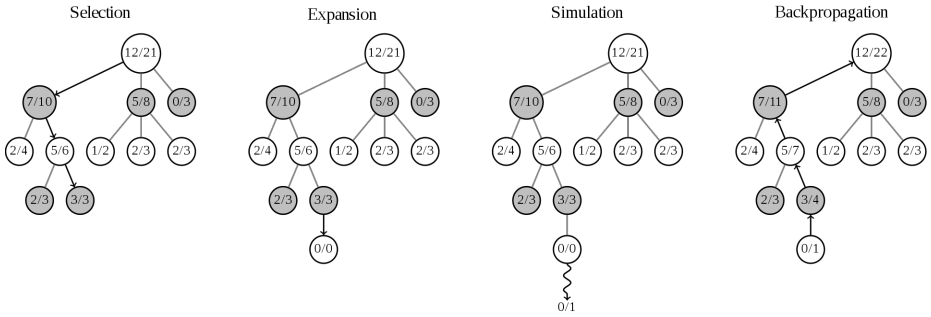

FIGURE 4 – On voit dans cet exemple d'une étape d'un algorithme MCTS que la partie simulée est perdante pour le joueur qui cherche un coup à jouer. Les noeuds blancs correspondent à un coup joué par le joueur adverse (dont le coup à la racine qui est l'état actuel du plateau). *Source : wikipedia*

la constante d'exploration $\sqrt{2}$ ne conduit pas aux meilleurs résultats. Nous avons donc simulé un grand nombre de parties grâce à des méthodes de multi-processing pour déterminer une meilleure constante.

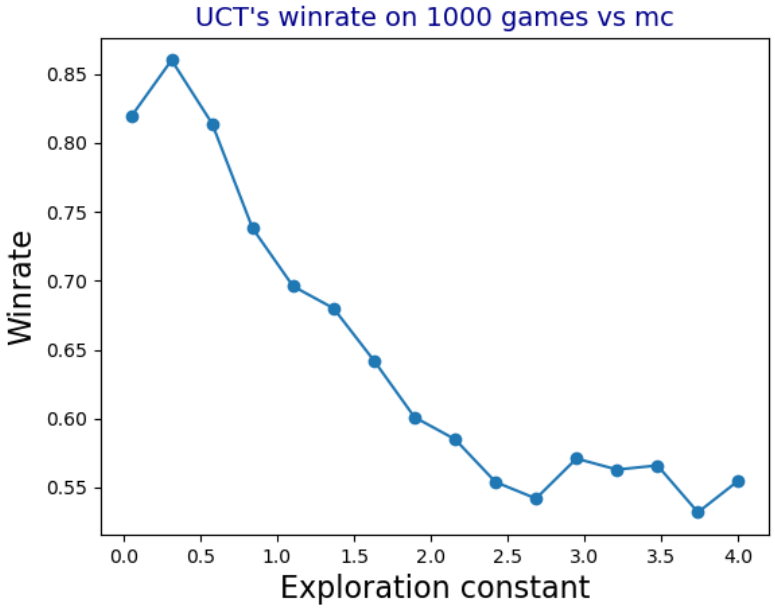

FIGURE 5

4.4 Améliorations possibles

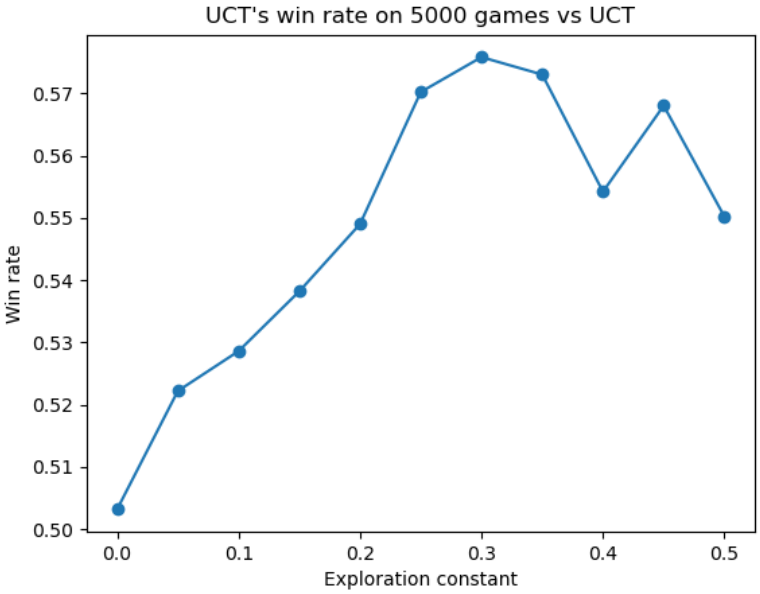

FIGURE 6

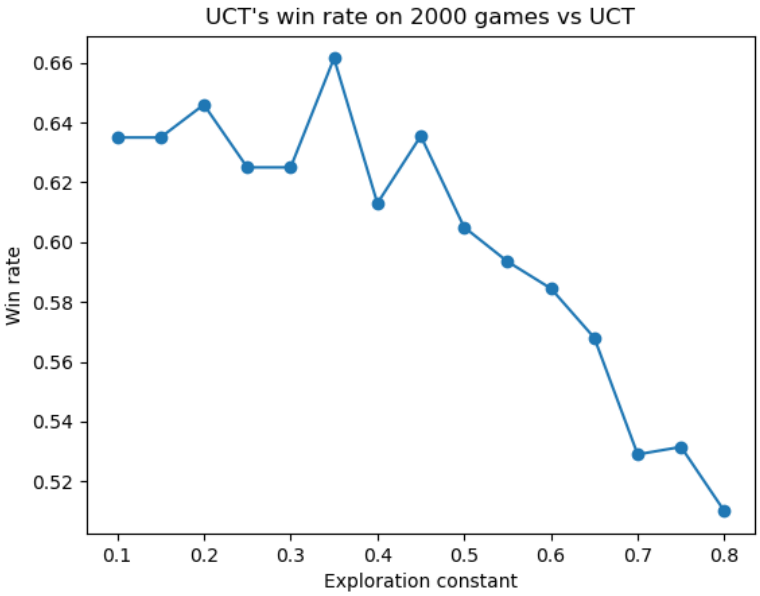

FIGURE 7

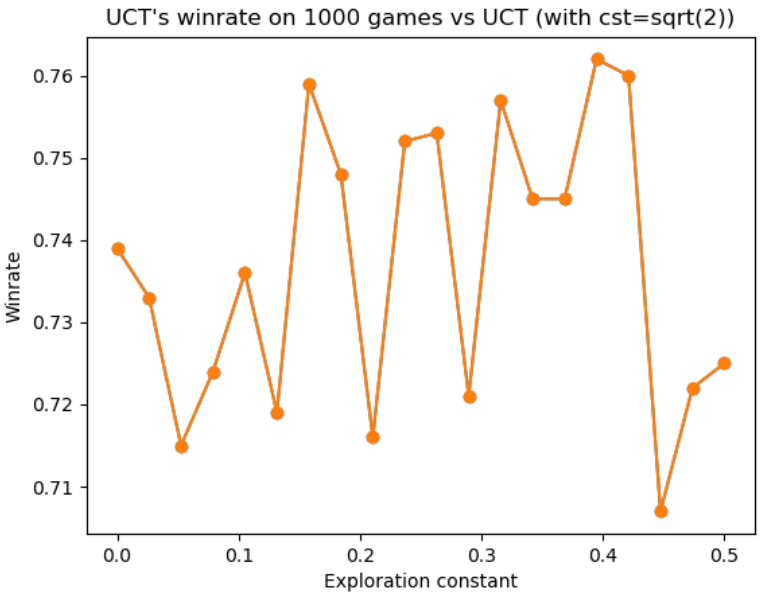
UCT's winrate on 1000 games vs UCT (with cst=sqrt(2))
Winrate
0.76
0.75
0.74
0.73
0.72
0.71
0.0
0.1
0.2
0.3
0.4
0.5
Exploration constant

FIGURE 8

Références

[1] Peter Auer, Nicolò Cesa-Bianchi & Paul Fischer, *Finite time Analysis of the Multiarmed Bandit Problem*

[2] Levente Kocsis & Csaba Szepesvàri, *Bandit Based Monte-Carlo Planning*