Des casinos à l'intelligence artificielle

Travail Encadré de Recherche M1 DS 2021

Adrien Maitammar
Maxime Le Paumier

17 mai 2021

Résumé

Hex est un jeu de plateau à information complète aux règles simples mais pouvant présenter une grande complexité. En effet, des stratégies gagnantes sont connues pour le premier joueur pour des plateaux de taille restreinte, mais la recherche de telles stratégies devient impraticable à mesure que la taille du plateau augmente. Il est alors question de trouver des stratégies efficaces à défaut de gagnantes. Nous en allons en présenter 3 reposant sur des méthodes de Monte-Carlo et Monte-Carlo avec recherche arborescente.

Sommaire

1 Introduction

1.1 Règles du jeu

Hex est 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.

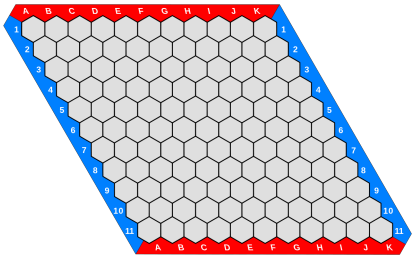


FIGURE 1 – Plateau de jeu 11x11

1.2 Spécificité du jeu

Une particularité du jeu de Hex est qu'il ne peut pas y avoir de partie aboutissant à un match nul. En effet, indépendamment des coups joués et de la taille du plateau, lorsque que le plateau est rempli, il y a obligatoirement un gagnant. Ce résultat a été établi par John Nash et repose sur une preuve par l'absurde. Ainsi, pour des plateaux de tailles inférieures ou égales à 9x9 une stratégie gagnante est connue pour le premier joueur. Mais ce n'est pas le cas pour les plateaux de taille supérieure et des résultats de la théorie de la complexité montre que la recherche d'une telle stratégie devient très rapidement impraticable quand la taille du jeu augmente. C'est pourquoi, nous allons adopter une approche statistique afin d'obtenir un comportement de jeu optimal de la part d'un joueur programmé avec des techniques d'intelligence artificielle (IA).

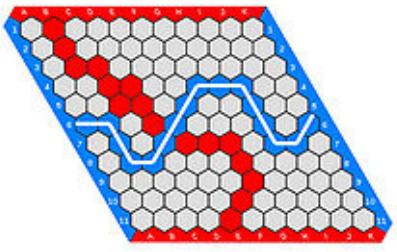


FIGURE 2 – Configuration gagnante pour le joueur bleu

2 Implémentation

2.1 Python Orienté Objet

Le jeu a été implémenté en langage Python. Le principaux modules utilisés sont le module pygame pour l'interface graphique pour les parties avec un joueur humain ainsi que le module multiprocessing pour augmenter la rapidité d'exécution des parties dans la phase de test des IA. Le projet est téléchargeable depuis GitHub.

Concernant la partie orienté objet, nous avons utilisé 3 classes. Une classe principale, `Game`, faisant interagir deux instances de la classe `Player` tour par tour en effectuant des actions sur le plateau, instance de la classe `Board`.

2.2 Condition de fin de partie

À chaque coup la fonction `check_win` est appelée pour vérifier si la partie a été remportée par un joueur. La rapidité d'exécution de cette fonction a été la principale source d'amélioration qui nous a permis d'augmenter le nombre de parties simulées par seconde. Après une tentative avec la théorie des graphe et l'algorithme de Dijkstra, nous avons opté pour une méthode utilisant les composantes connexes par couleur 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
```

Avec l'utilisation de multiprocessing, nous avons pu atteindre 23000 parties par seconde sur une machine possédant un processeur muni de 48 coeurs.

3 Joueur artificiel

Au cours du projet, nous avons implémenté des algorithmes de plus en plus performants. Chaque nouvelle version reprenait les bases de la précédente en ajoutant des amélioration, qui nous verrons, au fur et à mesure, se rapproche de la façon dont un joueur humain pourrait jouer et adopte de bons comportements liés à la structure du plateau de jeu.

3.1 Monte-Carlo

3.1.1 Principe

La première méthode que nous avons programmé est une méthode naïve et empirique. Elle consiste simplement à simuler n parties réparties équitablement entre les actions possibles pour déterminer le meilleur coup à jouer.

3.1.2 Algorithme

```
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
```

Avec cette méthode et en simulant environ 10000 parties par coup disponible, il est possible de voir que le joueur artificiel commence à jouer des coups sensés. En effet, il joue majoritairement au centre du plateau, ce qui est une bonne stratégie puisque cela offre davantage de possibilités de connexions par la suite.

3.2 Monte-Carlo et UCB1

Le problème de l'algorithme Monte-Carlo naïf est qu'il utilise autant de simulations pour chaque coup. De ce fait, il utilise autant de ressources pour évaluer la qualité des mauvais coups que pour évaluer la qualité des bons coups. Le critère de sélection UCB1 permet de pallier ce problème. Cet algorithme est nommé `mc_ucb1` dans les fichiers Python.

3.2.1 Principe

Voici le fonctionnement de l'algorithme :

UCB1
Initialisation : Jouer chaque coup une fois
Boucle : Jouer le coup j qui maximise $\bar{x_j} + c\sqrt{\frac{\ln n}{n_j}}$ avec $\bar{x_j}$ le taux de victoires du coup j jusqu'à présent, n_j le nombre de fois où le coup j a été jouée, n le nombre de coups total joués et c une constante.

FIGURE 3 – Algorithme UCB1

Cet algorithme a initialement été propos dans le cadre du problème du bandit manchot ou, dans sa version générale, du bandit à K bras. Le problème original 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 davantage.

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 cette stratégie dans un article de référence, *Finite-time Analysis of the Multiarmed Bandit Problem*, 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.

La constante théorique optimale est $c = \sqrt{2}$. Cependant, bien que des améliorations soient perceptibles comparé à l'algorithme `mc`, la constant d'exploration peut être optimisée. Nous reviendrons sur ce point dans la partie concernant l'algorithme UCT.

3.2.2 Algorithme

Une notion de nœud est intégrée à l'algorithme. Chaque nœud représente un état du jeu avec des attributs qui permettent de calculer son score avec le critère UCB1.

```
class treeNode():

    def __init__(self, state, parent):
        self.state = state
        self.parent = parent
        self.numVisits = 0
        self.totalReward = 0
        self.children = {}

        # self.root a la couleur du joueur adverse
        # Les noeuds enfat sont de la couleur du joueur utilisant
            l'algorithme
        if parent is None :
            self.player = 3 - state.player
        else:
            self.player = 3 - self.parent.player
```

Le nœud `self.root` représente l'état initial du plateau de jeu, c'est le nœud parent. Il possède ainsi des enfants représentant les coups à évaluer. Compte tenu de ce qui a été dit précédemment, la fonction retournant le meilleur coup pour l'algorithme `mc_ucb1` est la suivante :

```
def search(self, initialState):
    self.root = treeNode(initialState, None)
    actions = self.root.state.actions
    # Initialisation de chaque noeud
    ...
    # Simulation avec selection des noeuds via UCB1
    for i in range(self.iterationLimit):
        child = self.ucb1(self.root, self.explorationConstant)
        self.executeRound(child)

    bestChild = self.ucb1(self.root, 0)
    action = (action for action, node in self.root.children.items() if node
        is bestChild).__next__()
    return action

def ucb1(self, node, explorationValue):
    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)
```

La fonction `ucb1` calcule le score de chaque nœud via le critère UCB1. On note qu'une constante d'exploration nulle correspond au taux de parties gagnées pour un nœud.

3.3 Upper Confidence bounds applied to Trees (UCT)

L'algorithme `mc_ucb1` permet d'optimiser les simulations sur un coup mais ne permet pas d'anticiper les états du jeu une fois ce coup joué. Il est alors intéressant d'effectuer une recherche arborescente du meilleur coup. Cette méthode est appelée Upper Confidence bounds applied to Trees, nommée `uct` dans les programmes Python. Cet algorithme de prise de décision est largement développé dans les jeux [1].

3.3.1 Principe

On dispose de n simulations par tour.

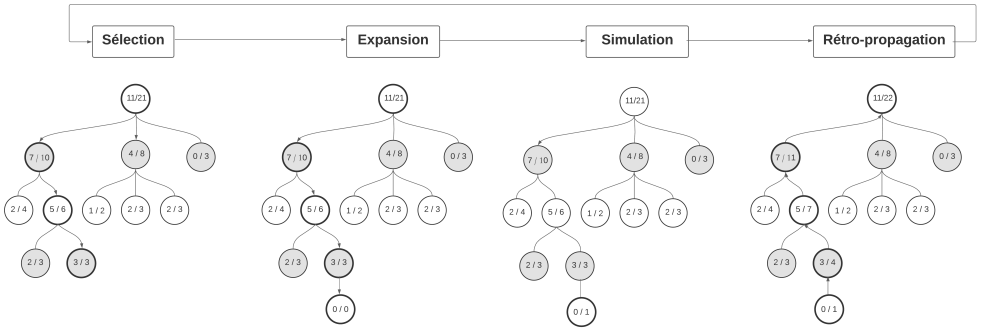


FIGURE 4 – Déroulement d'une simulation avec l'algorithme UCT

Tout d'abord, il faut noter que le coup à la racine de l'arborescence est l'état actuel du plateau. De plus, les nœuds blancs correspondent à un coup joué par le joueur adverse.
On initialise tous les nœuds de profondeur 1 en effectuant une simulation de parties aléatoires sur ces derniers avant de répéter les quatre étapes décrites ci-dessous. La figure 4 est prise comme exemple pour illustrer le propos.

1. Sélection

Premièrement, on effectue la phase de descente en sélectionnant les nœuds via le critère UCB1 jusqu'à atteindre un nœud sans enfant.
Exemple : Les nœuds gris sur la ligne de profondeur 1 ont pour score avec la constance théorique optimale $c = \sqrt{2}$ respectivement de la gauche à la droite $\frac{8}{10} + \sqrt{\frac{2\log 21}{10}} \simeq 1,480$, 1.372 et 1.425. On choisit donc le nœud de gauche. Puis, on calcule de nouveau les scores des nœuds enfant, ici de profondeur 2. On obtient 1.572 pour le nœud de gauche et 1.709 pour le nœud de droite. On sélectionne donc le nœud de droite puis avec le même critère encore le nœud de droite. On arrive à un nœud sans enfant aussi appelé feuille de l'arbre. C'est la fin de la première étape.

2. Expansion

Deuxièmement, lorsque on atteint une feuille de l'arbre, on crée un nouveau nœud

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

représentant un nouveau coup joué. Ce coup est choisi de manière aléatoire.

3. Simulation

Troisièmement, on finit la partie en sélectionnant les coups de manière aléatoire. Dans l'exemple la partie simulée a été perdue.

4. Rétro-propagation

Quatrièmement, on rétro-propage ce résultat jusqu'à la racine.
Exemple : On ajoute 1 au dénominateur (nombre de visites) pour chaque nœuds correspondant aux coups joués et on ajoute 1 au numérateur si la partie simulée a été gagnée et si le nœud est gris, c'est-à-dire si il correspond à un coup joué par le joueur artificiel "utilisant" l'algorithme.

Une fois les n simulations effectuées, on sélectionne le nœud de profondeur 1 ayant le meilleur taux de victoire.

3.3.2 Algorithme

L'algorithme `uct` reprend en grande partie le code de l'algorithme `mc_ucb1` mais avec une structure d'arbre.

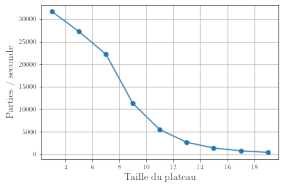
```
def randomPolicy(state):
    while not state.isTerminal():
        action = random.choice(state.actions)
        state.takeAction(action, state.currplayer)
    return state.getReward()

class treeNode():
    def __init__(self, state, parent):
        self.state = state
        self.parent = parent
        self.numVisits = 0
        self.totalReward = 0
        self.children = {}
        if parent is None :
            self.player = 3 - state.player
        else:
            self.player = 3 - self.parent.player

def isFullyExpanded(self):
    return len(self.state.actions)==len(self.children)

class UCT():
    def __init__(self, explorationConstant,iterationLimit=None):
        self.iterationLimit = iterationLimit
        self.explorationConstant = explorationConstant
        self.rollout = randomPolicy

    def search(self, initialState):
        self.root = treeNode(initialState, None) if root==None else root
        for i in range(self.iterationLimit):
            self.executeRound()
        bestChild = self.getBestChild(self.root, 0)
        action = (action for action, node in self.root.children.items() if
            node is bestChild).__next__()
        return action
```

```
def executeRound(self):
   node = self.selectNode(self.root)
   state = deepcopy(node.state)
   reward = self.rollout(state)
   self.backpropogate(node, reward)

def selectNode(self, node):
   while not node.state.isTerminal():
      if node.isFullyExpanded():
         node = self.getBestChild(node, self.explorationConstant)
      else:
         return self.expand(node)
   return node

def expand(self, node):
   actions = node.state.getPossibleActions()
   while actions!=[]:
      action = random.choice(actions)
      if action not in node.children.keys():
         node_state = deepcopy(node.state)
         node_state.takeAction(action, node_state.currplayer)
         newNode = treeNode(node_state, node)
         node.children[action] = newNode
         return newNode

def backpropogate(self, node, reward):
   while node is not None:
      node.numVisits += 1
      node.totalReward += (reward == 1) * (node.player !=
          self.root.player)
      node = node.parent

def getBestChild(self, node, explorationValue):
   return self.ucb1(node,explorationValue)

def ucb1(self,node,explorationValue):
   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)
```

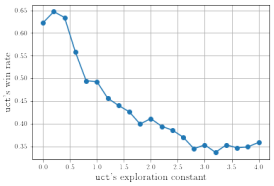
3.3.3 Constante d'exploration optimale

La constante d'exploration optimale ne correspond pas à la valeur théorique $c = \sqrt{2}$. En pratique cette dernière est plus faible pour le jeu de Hex que pour le

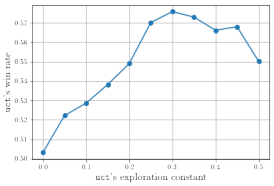
problème des bandits manchots et les meilleurs résultats sont obtenus pour $c \simeq 0.3$. Cette observation semble être partagée pour les jeux ayant des récompenses entre 0 et 1, ce qui est le cas ici : une partie gagnée vaut une récompense de 1 et une partie perdue donne une récompense nulle.



(a) $n = 100$, parties/constante : 2000, contre `mc`



(b) $n = 100$, parties/constante : 2000, contre `uct` avec $c = \sqrt{2}$



(c) $n = 100$, parties/constante : 5000, contre `uct` avec $c = \sqrt{2}$

FIGURE 5 – Constante d'exploration optimale pour UCT

4 Évaluation des algorithmes

5 Axes d'amélioration

5.1 UCT avec mémoire

L'algorithme `uct` pourrait garder en mémoire une partie des simulations effectuées pour ses coups suivants. En effet, après avoir joué une première fois, le joueur artificiel `uct` pourrait tronquer l'arbre et considérer comme racine le nœud correspond à l'état actuel du plateau.

Illustration d'arbre tronqué ?

Cependant cette méthode ne produit pas d'amélioration significative si le nombre de simulations par coup n'est pas assez grand. Avec une telle méthode, nous n'avons obtenu qu'un gain d'information inférieur à 1%.

5.2 Algorithme RAVE et GRAVE

Un autre axe d'amélioration se trouve en début de partie. Plus il y a de coups possibles, moins les estimations de la qualité de ces derniers sont justes. C'est pourquoi compléter l'algorithme `uct` avec une autre approche serait intéressant.

A développer

5.3 Connections virtuelles et cases mortes

A développer

6 Conclusion

A développer

Table des figures

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*