Dauphine | PSL
UNIVERSITÉ PARIS

Projet Monte Carlo :
Appliqué aux jeux d'échecs

Louis Fontaine
Kenza Hammou
Olivier Randavel

July 31, 2020

1 Introduction

Ce projet a pour but d'implémenter les différentes méthodes de Monte Carlo vues en cours. Le choix du jeu s'est porté sur le jeu d'échecs. Ce rapport sera constitué de trois parties. Une première partie présentera l'interface et le code utilisés pour effectuer des parties. Dans un second temps, l'implémentation des méthodes de Monte Carlo sera détaillée. Enfin, la dernière partie évoquera les résultats obtenus ainsi que les difficultés rencontrées.

2 Préparation du code

Pour réaliser notre projet, nous avons exploité le code fourni par le git suivant: [3]. Ce code regroupe une interface graphique permettant à un utilisateur de jouer contre une intelligence artificielle. Le fichier python nommé "python_easy_chess_gui.py" rassemble les fonctionnalités de l'interface et la génération des coups par l'IA et l'utilisateur. Ce fichier a été renommé "MChess.py". Il est accessible sur notre git [2]. Ce code utilise la librairie "python-chess" [4], elle définit les classes *board* et *move*. Notre travail a été dans un premier temps de comprendre et d'explorer les 4000 lignes de code. Ensuite nous avons modifié le code pour l'adapter à notre projet. Les détailles sur l'implémentation de l'IA ne sont pas communiqués, seul le fichier binaire est fourni. Notre objectif fut dans un premier temps de remplacer l'action de l'utilisateur par celle d'un robot suivant une loi uniforme sur les *legal_moves*.

```
elif condition == "UNIFORM":
    moves = [i for i in board.legal_moves]
    n = random.randint(0, len(moves) - 1)
    best_move = moves[n]
```

Ensuite, nous avons souhaité optimiser le code afin que l'interface ne soit pas modifiée au cours de la partie. L'algorithme renvoie donc uniquement le score lorsque la partie se termine. Le score peut prendre plusieurs valeurs (1-0 blanc gagne, 0-1 noir gagne, 1/2-1/2 égalité, * indéterminé) suivant les règles du jeu d'échecs. En effet, ces dernières mettent en avant un nombre de coups maximum traduit par * lorsque celui-ci est dépassé. A l'aide du paramètre "SHOW_GUI = True" il est possible de montrer les coups de chaque joueur au cours de la partie sur l'interface graphique.
Puis, avant d'implémenter les méthodes de Monte Carlo, nous avons retiré la partie concernant l'intelligence artificielle. Nous nous sommes retrouvés avec un code permettant de jouer avec deux joueurs suivant une loi uniforme sur les *legal_moves*. En outre, dans le but de préparer la phase expérimentale, la fonction "main" a été modifiée. Celle-ci prend en argument la méthode de jeu de chaque joueur (UCT, Uniform, UCB), les paramètres *condition1* et *condition2* permettent de spécifier le rôle des deux joueurs. De plus, elle permet de lancer de façon itérative un nombre de parties spécifiées en paramètre. Le score moyen permet alors d'évaluer le taux de réussite des méthodes de Monte Carlo. De plus, nous avons renvoyé le score à chaque fin de partie et relancé automatiquement une nouvelle partie sans qu'un utilisateur n'ait à interagir avec l'interface.

```
def main(condition2, condition1):
    ...
    score = []
    score = pecg.main_loop(score, 30, condition2, condition1)
    print(score)
```

```
    sum_white, sum_black, others = 0, 0, 0
    for i in score:
        if i == "1-0":
            sum_white += 1
        elif i == "0-1":
            sum_black += 1
        else:
            others += 1
    return sum_white, sum_black, others
```

Enfin, il a fallu implémenter le point d'entrée du script. Celui-ci comporte les paramètres pour spécifier la méthode de jeu de chaque joueur, ainsi que l'actualisation de l'interface à chaque partie SHOW_GUI. Il est possible d'exécuter plusieurs partie à la suite avec le paramètre *X*. Enfin le *NB_PLAYOUT* est spécifique aux méthodes UCB et UCT, il définit le nombre de parties à faire avant d'évaluer le meilleur coup.

```
if __name__ == "__main__":

    SHOW_GUI = True
    NB_PLAYOUT = 100
    X = 30

    """
    Choix des IA (UCT, UCB, UNIFORM).
    condition2 : black
    condition1 : white
    """
    condition2, condition1 = "UCT", "UNIFORM"

    main(condition2, condition1)
```

3 Implémentation des méthodes de Monte Carlo

Cette partie détaille les méthodes de Monte Carlo implémentées. L'objectif est de présenter les fonctions créées et d'évoquer les difficultés rencontrées.
Dans un premier temps, UCB (Upper Confidence Bound) a été implémenté suivant l'exemple effectué sur le jeu Breakthrough vu en cours. Notre implémentation est présente dans le fichier nommé "UCB.py". Ce fichier est constitué de plusieurs fonctions :

- **score** : renvoie 1 uniquement lorsque le joueur blanc gagne c'est à dire lorsque score égale 1-0
- **playout** : pour un board donné, il effectue une partie jusqu'à l'obtention de l'état terminal
- **UCB** : cette fonction est très similaire à celle du cours. Il a fallu adapter les actions à notre classe "board".

Dans un second temps, il a été question d'implémenter la méthode de Monte Carlo UCT (Upper Confidence Trees). Notre implémentation est présente dans le fichier nommé "UCT_IA.py". Ce fichier est constitué de plusieurs fonctions

- **add** : Si un board n'a jamais été exploré alors il est ajouté à la table de transposition

- **look** : Permet de récupérer la table de transposition suivant le h.
- **score** : Renvoie 1 uniquement lorsque le joueur noir gagne c'est à dire lorsque score égale 0-1 et 0 sinon.
- **playout** : Pour un board donné, il effectue une partie jusqu'à l'obtention de l'état terminal.
- **get_color_code** : Renvoie un code associé à une couleur : 1 pour une case contenant un pion blanc, 2 pour une case contenant un pion noir et 0 si la case est vide.
- **update_hashcode** : Met à jour le hashcode du board. Cette fonction est appelé avant de push un move. Cette fonction contient le code XOR. Il a été adapté au jeu d'échec suivant le hachage de Zobrist détaillé dans cet article : [1]. Cette implémentation est plus compliqué que celle de breakthrough. En effet, le jeux d'échecs comporte 6 pions différents et 2 couleurs. Il a donc fallu récupérer à l'aide de la classe board la pièce sur la case où l'on va et celle où l'on est. L'implémentation est présentée dans l'annexe.
- **play** : Retourne le hashcode du board mis à jour à partir de **update_hashcode** et push le *best_move*.
- **UCT** : cette fonction est très similaire à celle du cours. Il a fallu adapter les actions à notre classe "board" en utilisant les fonctions précédemment détaillées.
- **BestMoveUCT** : de même.

Enfin, nous avons envisagé de coder les méthodes de Monte Carlo RAVE et GRAVE. Ces méthodes requièrent l'utilisation de la métrique AMAF. En effet, la table de transposition possède alors 2 paramètres de plus, soit 5 au total. Mais nous ne sommes pas parvenu à déterminer comment coder le code permettant d'attribuer un index pour la liste nplayoutsAMAF et nwinsAMAF pour chaque board. De plus, lors du cours, ce code avait été ajouté à la classe *move*, or pour rappel, le git utilisé ne donne pas accès à la librairie spécifiant les classes *board* ou *move*. Le code ci-dessous est celui correspondant à Breakthrough et n'a pas pu être adapté au jeu d'échecs.

```
1   def code (self):
2         direction = 1
3         if self.y2 > self.y1:
4           direction = 0
5         if self.y2 < self.y1:
6           direction = 2
7         if self.color == White:
8           return 3 * (Dy * self.x1 + self.y1) + direction
9         else:
10          return 3 * Dx * Dy + 3 * (Dy * self.x1 + self.y1) + direction
```

4 Résultats

Dans cette partie, nous exposons les résultats obtenus lors des différentes expérimentations. L'objectif a été d'évaluer les performances de notre implémentation de UCT, UCB et Uniform. A notre disposition nous avions plusieurs hyperparamètres pour effectuer une moyenne du score sur un nombre large d'essais. De plus, il est possible de spécifier le nombre de parties a effectuer avant que UCT ou UCB ne déterminent le

best_move. Cependant, les performances de nos méthodes nous ont contraints à utiliser un nombre de partie raisonnable pour limiter le temps de réponse fournit par UCT et UCB. Il est important d'avoir en tête que le nombre de *legal_moves* aux échecs est de plusieurs dizaines pour un board donné en moyenne. L'utilisateur doit donc choisir un *n* au moins supérieur à 100 pour obtenir de bons résultats. En effet, nous avions essayé un *n=10* et le match UCT contre Uniform se terminait rarement avec une partie gagnée pour UCT. De plus en moyenne une partie d'échec demande 40 coups au total et plus *n* est grand plus les méthodes de Monte Carlo mettent du temps à répondre. Il a donc fallu faire peser le pour et le contre pour obtenir des temps d'attente raisonnables. Nos expérimentations ont été effectuées sur 30 parties avec différentes valeurs de *n*. De plus, une moyenne du temps de réponse des méthodes est spécifiée.

	UCT (100)	UNIFORM	UCB (300)	Temps (secondes)
UNIFORM	27-0			<1
UCB (100)	14-14	0-27		<30
UCT (300)			18-12	<70

Ces résultats permettent de montrer que nos méthodes sont correctement implémentées. Les méthodes UCT et UCB sont bien meilleures que la méthode Uniforme. Cependant, il est compliqué de conclure entre les performances de UCT et UCB. En augmentant le nombre de parties, on obtient une meilleure performance pour UCT. Ces observations corroborent celles vues lors du cours avec le jeu Breakthrough. Néanmoins, les temps d'exécutions restent très élevés, le jeu d'échecs étant plus complexe que celui développé en cours, l'IA met donc plus de temps à répondre. Il serait intéressant d'améliorer les performances d'UCB et d'UCT pour permettre à un utilisateur de jouer contre l'IA sans trop patienter.

References

[1] ADAM BERENT. *Transposition Table and Zobrist Hashing*. URL: https://adamberent.com/2019/03/02/transposition-table-and-zobrist-hashing/. (accessed: 16.07.2020).

[2] Daugit. *Python Easy Chess GUI*. URL: https://github.com/Daugit/MChess. (accessed: 16.07.2020).

[3] fsmosca. *A Chess GUI based from Python using PySimpleGUI and Python-Chess.* URL: https://github.com/fsmosca/Python-Easy-Chess-GUI. (accessed: 16.07.2020).

[4] niklasf. *python-chess: a pure Python chess library*. URL: https://python-chess.readthedocs.io/en/latest/. (accessed: 16.07.2020).

5 Annexe

```
###UCB###
import copy
import random
import math
import time

```

```
def score(board):
    return 1 if board.result(claim_draw=True)=="1-0" else 0


def playout(b):
    while (True):
        moves = b.legal_moves
        moves = [i for i in moves]
        if b.is_game_over():
            return score(b)
        n = random.randint(0, len(moves) - 1)
        b.push(moves[n])


def UCB(board, n):
    moves = [i for i in board.legal_moves]
    print(len(moves))
    sumScores = [0.0 for x in range(len(moves))]
    nbVisits = [0 for x in range(len(moves))]
    for i in range(n):  # on fait les n tirages et on apprend au fur et à mesure
        bestScore = 0
        bestMove = moves[0]
        place = 0
        for m in range(len(moves)):  # on calcule le score de chaque coût
            if nbVisits[m] > 0:
                score = sumScores[m] / nbVisits[m] + 0.4 * math.sqrt(math.log(i) /
    nbVisits[m])
            else:
                score = 10000000  # on explore tout !! du dernier au premier
            if score > bestScore:
                bestScore = score
                bestMove = moves[m]
                place = m
        b = copy.deepcopy(board)
        b.push(bestMove)  # on joue le meilleur score
        r = playout(b)

        sumScores[place] += r  # on met à jour les poids
        nbVisits[place] += 1
    bestScore = 0
    bestMove = moves[0]
    for m in range(1, len(moves)):  # on renvoie le meilleur move
        score = sumScores[m]
        if score > bestScore:
            bestScore = score
            bestMove = moves[m]
    return bestMove
```

```
###UCT\_IA###
import copy
import random
from math import *
import time

import chess

```

```
BLACK = False
WHITE = True

d = {
    "a": 0,
    "b": 1,
    "c": 2,
    "d": 3,
    "e": 4,
    "f": 5,
    "g": 6,
    "h": 7,
    "8": 0,
    "7": 1,
    "6": 2,
    "5": 3,
    "4": 4,
    "3": 5,
    "2": 6,
    "1": 7
}


def add(board, h, Table):
    """
    Ajoute un board et son hash dans la table de transposition.
    :param board:
    :param h:
    :param Table:
    :return:
    """
    nplayouts = [0.0 for x in range(len([i for i in board.legal_moves]))]  # propre au board
    nwins = [0.0 for x in range(len([i for i in board.legal_moves]))]

    Table[h] = [1, nplayouts, nwins]


def look(h, Table):
    """
    Cherche un board dans la table de transposition.
    :param h:
    :param Table:
    :return:
    """
    try:
        t = Table[h]
    except:
        t = None
    return t


def score(board):
    """
    Favoriser le noir
    :param board:
    :return:
    """
```

```
    return 1 if board.result(claim_draw=True) == "0-1" else 0


def playout(b, h, piece_hash):
    """
    Joue une partie aléatoire
    :param b:
    :param h:
    :param piece_hash:
    :return:
    """
    while (True):
        moves = b.legal_moves
        moves = [i for i in moves]
        if b.is_game_over():
            return score(b), h
        n = random.randint(0, len(moves) - 1)
        h = play(b, h, moves[n], piece_hash)


def get_color_code(col):
    """
    Détermine l'indice de la couleur (pour la table de hashage)
    :param col:
    :return:
    """
    if (col == None):
        code = 0
    elif (col):
        code = 1
    else:
        code = 2
    return code


def update_hashcode(piece, board, h, hashTable, hashTurn, move):
    """
    Update hashcode of a board.
    Need to call this function before using board.push(move).
    :param board:
    :param h:
    :param move:
    :return:
    """
    col = board.color_at(move.to_square)
    col = get_color_code(col)

    from_uci = chess.square_name(move.from_square)
    x1 = d[from_uci[0]]
    y1 = d[from_uci[1]]
    to_uci = chess.square_name(move.to_square)
    x2 = d[to_uci[0]]
    y2 = d[to_uci[1]]

    move_color = get_color_code(board.turn)

    if col != None:
        h = h ^ hashTable[col][x2][y2][piece-1]
```

```
124
125     h = h ^ hashTable[move_color][x2][y2][piece-1]
126     h = h ^ hashTable[move_color][x1][y1][piece-1]
127     h = h ^ hashTurn
128
129     return h
130
131 def update_hashcode_zobriest(piece, board, h, piece_hash, move):
132     """
133         Update hashcode of a board with Zobriest Hashing
134         Need to call this function before using board.push(move).
135         :param board:
136         :param h:
137         :param move:
138         :return:
139         """
140
141     to_col = board.color_at(move.to_square)
142     to_col = get_color_code(to_col)
143     to_piece = board.piece_type_at(move.to_square)
144
145     from_uci = chess.square_name(move.from_square)
146     x1 = d[from_uci[0]]
147     y1 = d[from_uci[1]]
148     to_uci = chess.square_name(move.to_square)
149     x2 = d[to_uci[0]]
150     y2 = d[to_uci[1]]
151
152     indice_color = 0 if board.turn else 1 #True = White
153
154     h = h ^ piece_hash[(piece - 1) + 6*indice_color][x1][y1]
155     h = h ^ piece_hash[(piece - 1) + 6*indice_color][x2][y2]
156
157     if(to_col == 1):
158         h = h ^ piece_hash[(to_piece - 1)][x2][y2]
159     elif(to_col == 2):
160         h = h ^ piece_hash[(to_piece - 1) + 6][x2][y2]
161
162     return h
163
164 def play(board, h, best_move, piece_hash):
165     """
166     Joue un move et update le hashcode du board.
167     :param board:
168     :param h:
169     :param best_move:
170     :param piece_hash:
171     :return:
172     """
173     piece = board.piece_type_at(best_move.from_square)
174     h = update_hashcode_zobriest(piece, board, h, piece_hash, best_move)
175     board.push(best_move)
176     return h
177
178
179 def UCT(board, h, piece_hash, Table):
180     """
181     IA de l'UCT
```

```
    :param board:
    :param h:
    :param piece_hash:
    :param Table:
    :return:
    """
    if board.is_game_over():
        return score(board), h

    t = look(h, Table)
    if t != None:  # Selection and expansion step
        bestValue = -1000000.0
        best = 0

        moves = [i for i in board.legal_moves]
        if len(moves) != len(t[1]):
            print("Error : ", len(moves))
            print("Error : ", len(t[1]))
        for i in range(0, len(moves)):
            val = 1000000.0
            if t[1][i] > 0:
                Q = t[2][i] / t[1][i]
                if board.turn == WHITE:
                    Q = 1 - Q
                val = Q + 0.4 * sqrt(log(t[0]) / t[1][i])
            if val > bestValue:
                bestValue = val
                best = i

        res = 0.0
        if len(moves) > 0:
            h = play(board, h, moves[best], piece_hash)
            res, h = UCT(board, h, piece_hash, Table)
            t[0] += 1
            t[1][best] += 1  # mise à jour à l'indice best, qui est propre au board
            t[2][best] += res
        return res, h
    else:  # Sampling step
        add(board, h, Table)
        score_playout, h = playout(board, h, piece_hash)
        return score_playout, h


def BestMoveUCT(board, h, piece_hash, nb_playout):
    """
    Détermine le best move selon UCT.
    :param board:
    :param h:
    :param piece_hash:
    :param nb_playout:
    :return:
    """
    Table = {}

    for i in range(nb_playout):
        b1 = copy.deepcopy(board)
        h1 = h
        UCT(b1, h1, piece_hash, Table)
```

```
240         t = look(h, Table)
241 
242         moves = [i for i in board.legal_moves]
243         best = moves[0]
244         bestValue = t[1][0]
245         for i in range(0, len(moves)):
246             if (t[1][i] > bestValue):
247                 bestValue = t[1][i]
248                 best = moves[i]
249         return best
```