Rapport Minimax Morpion

Minimax avec alpha-beta

L'algorithme de minimax appliqué ici est l'algorithme classique de minimax.

En plus, nous avons un profondeur maximale pour l'arbre (`max_depth`) qui permet d'obtenir un bon ratio entre précision et performance.

A chaque appel récursif à une branche fille, la variable `depth` qui est initialisé `max_depth` décroit de 1. Si on a `depth` à 0, alors on retourne l'heuristique lié au plateau actuel.

Sinon on applique la suite de minimax. L'algorithme pour `maximizingPlayer` et l'opposant sont sensiblement les mêmes. L'algorithme est l'algithme classique de minimax à la différence de 2 choses :

- On créé des branches filles à partir de coups déterminés par un énumérateur `legal_moves` qui prend en paramètre le plateau et le dernier coup joué.
- On test si il y a une victoire pour des coups, et si c'est le cas, nous ne faisons pas d'appel réccursif à minimax mais nous donnons un score très élevé (ou très faible) et on recommence la boucle (qui devrait se terminer rapidement après grâce à l'élagage alpha-beta).

```
def minimax(board, depth, alpha, beta, maximizingPlayer, curr, opp,
last_played_move, max_depth):
  # ...
  # Si la profondeur est à 0, on renvoit l'heuristique
  if depth == 0:
    return heuristic(board, symbol) * (1 if maximizingPlayer else -1), None

  if maximizingPlayer:
    maxEval = -np.inf
    best_move = None
    for col, line in legal_moves(board, last_played_move):
      board[line, col] = symbol
      # on ne vérifie la victoire que si plus de 3 symbole joués
      if np.sum(board == curr) > 3 and is_win(board, symbol):
          eval = 100000 / (max_depth - depth + 1) ** 2
      else:
        eval, _ = minimax(board, depth - 1, alpha, beta, False, curr, opp, (col,
line), max_depth)
        eval /= (max_depth - depth + 1) ** 2
      if eval > maxEval:
        best_move = (col, line)
        maxEval = eval
      alpha = max(alpha, eval)
      board[line, col] = 0
      # élagage alpha-beta
      if beta <= alpha:
```

```
          break
      return maxEval, best_move

    else:
      # ...
      # même chose pour l'adversaire (avec scores négatifs)
```

L'heuristique

Pour l'heuristique, on a compté le nombre de symbole alignés qu'on a enregistré dans des compteurs puis on donne un certain score en fonction du nombre de symboles alignés.

L'heuristique n'est pas complexe mais efficace. Elle a été optimisé pour faire le moins de boucle possible et le code n'est donc pas très lisible.

```
# Valeurs attribués au nombre de symboles alignés
# pour l'heuristique
VALUES = (
    1, #1
    10, #2
    100 #3
)

def heuristic(board, symbol):
  global BOARD_SIZE
  global BOARD_SIZE_1
  global BOARD_SIZE_4
  global BOARD_SIZE_5
  score = 0
  for i in prange(BOARD_SIZE):
    # compteurs pour chaque joeur et direction
    cOppLine, cPlayerLine, cOppCol, cPlayerCol = 0, 0, 0, 0
    cOppDiag, cPlayerDiag = [0, 0, 0, 0], [0, 0, 0, 0]
    for j in prange(BOARD_SIZE):
      # line
      tempScore, cOppLine, cPlayerLine = define_score(board[i, j], cOppLine,
cPlayerLine, symbol)
      score += tempScore
      # col
      tempScore, cOppCol, cPlayerCol = define_score(board[j, i], cOppCol,
cPlayerCol, symbol)
      score += tempScore

      # \ diag
      # from trace to left [0]
      if j + i < BOARD_SIZE_4:
        tempScore, cOppDiag[0], cPlayerDiag[0] = define_score(board[j + i, j],
cOppDiag[0], cPlayerDiag[0], symbol)
        score += tempScore
      # from trace + 1 to right [1]
```

```
      if j + i < BOARD_SIZE_5:
        tempScore, cOppDiag[1], cPlayerDiag[1] = define_score(board[j, j + i + 1],
cOppDiag[1], cPlayerDiag[1], symbol)
        score += tempScore
      # / diag
      # from / to left [2]
      if BOARD_SIZE - i - j >= 1:
        tempScore, cOppDiag[2], cPlayerDiag[2] = define_score(board[BOARD_SIZE - 1
- j - i, j], cOppDiag[2], cPlayerDiag[2], symbol)
        score += tempScore
      # from / to right [3]
      if i + j < BOARD_SIZE_1:
        tempScore, cOppDiag[3], cPlayerDiag[3] = define_score(board[BOARD_SIZE - j
- 1, j + i + 1], cOppDiag[3], cPlayerDiag[3], symbol)
        score += tempScore
  return score

# prend en paramètre des compteurs, les modifie et attribu un score ou non
def define_score(case, countOpp, countPlayer, symbol):
  global VALUES
  score = 0
  if case == 0:
    if countOpp != 0 or countPlayer != 0:
      score = VALUES[min(max(countOpp, countPlayer), 4) - 1] * (1 if countPlayer
!= 0 else -1)
      countOpp = countPlayer = 0
  elif case == symbol:
    countPlayer += 1
    if countOpp != 0:
      score = -VALUES[min(countOpp, 4) - 1]
      countOpp = 0
  else:
    countOpp += 1
    if countPlayer != 0:
      score = VALUES[min(4, countPlayer) - 1]
      countPlayer = 0
  return score, countOpp, countPlayer
```

Legal Moves

La méthode legal_moves retourne tout les coups qui peuvent être joué. Ici, on prend tout les coups qui ont été joué et on recupère les coordonnées des cases autour seulement.

Aussi on tri ces coups en fonction de leur distance au dernier coup joué. On pense que les meilleurs coups (la plupart du temps) sont ceux autour du dernier coup joué et on aura donc un élagage alpha-beta plus efficace.

```
def legal_moves(board, last_played_move):
  lines, cols = np.where(board != 0)
  col, line = last_played_move
  # on trie les coups possible en fonction
```

```
  # de leur distance au dernier coup joué
  order = np.argsort(np.floor(np.sqrt((cols - col) ** 2 + (lines - line) ** 2)))
  lines = lines[order]
  cols = cols[order]
  done = []
  for line, col in zip(lines, cols):
    for i in prange(max(0, line - 1), min(BOARD_SIZE, line + 2)):
      for j in prange(max(0, col - 1), min(BOARD_SIZE, col + 2)):
        # on retourne les coups s'il n'ont pas déjà été fait
        if (i, j) not in done and board[i, j] == 0:
          yield j, i
          done.append((i, j))
```

Méthode is_win

Cette méthode pour vérifier la victoire prend en paramètre le symbole du dernier joueur ayant joué. Cela permet de sauver du temps car le joueur qui a joué est le seul qui peut gagner lorsque la méthode est appelé.

Aussi, au lieu de vérifié toutes les case du plateau, on fait une boucle sur un énumérateur `played_move_by_symbol` qui prend en paramètre le plateau et le symbole du joueur qui renvoit tout les coups qui ont été joué uniquement par le joueur. Ainsi on à une méthode beacoup moins couteuse, surtout dans les premiers coups.

```
def is_win(board, symbol):
  global BOARD_SIZE
  global BOARD_SIZE_3
  for line, col in played_move_by_symbol(board, symbol):
      # line
      if line < BOARD_SIZE_3 and (board[line:line+4, col]==symbol).all():
          return True
      #col
      if col < BOARD_SIZE_3 and (board[line, col:col+4]==symbol).all():
          return True
      # \ diagonal
      if line < BOARD_SIZE_3 and col < BOARD_SIZE_3 and board[line, col] ==
board[line+1, col+1] == board[line+2, col+2] == board[line+3, col+3] == symbol:
          return True
      # / diagonal
      if col > 2 and line < BOARD_SIZE_3 and board[line, col] == board[line+1,
col-1] == board[line+2, col-2] == board[line+3, col-3] == symbol:
          return True

  return False


def played_move_by_symbol(board, symbol):
  global BOARD_SIZE
  for i in prange(BOARD_SIZE):
    for j in prange(BOARD_SIZE):
      if board[i, j] == symbol:
        yield i, j
```

Compilation du code python

Nous sommes astreint à utiliser Google Collab pour des soucis d'égalité des ressources et donc à coder en python. Nous avons cependant réussi à compiler le code avec la bibliothèque numba. Numba compile le code en Assembly avec des vitesses similaires à C ou FORTRAN il me semble ce qui rend le code 50 fois plus rapide.

Tout d'abord, toutes les variables global sont des constantes au moment de la compilation et donc avoir des variables globales n'est pas possible (dictionnaire par exemple) Nous devons donc passer ce genre de variables en paramètre.

Aussi, numba ne compile qu'un nombre d'opérations limités et il faut donc utiliser des fonctions basiques (certaines fonctions numpy sont prises en charge).

Nous utilisons l'énumérateur `prange` de *numba* à la place de l'énumérateur `range` pour permettre une parralélisation des opérations.

Toutes les fonctions appelé à l'intérieur des fonctions compilé doivent être elle-même des fonctions compilés.

Pour spécifier les fonctions à compiler, on ajoute le décorateur `@njit` de *numba* au début des fonctions.

On a donc une structure pour la compilation qui est la suivante :

```
from numba import njit, prange

@njit # decorator for compiling
def played_move_by_symbol(board, symbol):
  global BOARD_SIZE # compile time constant
  for i in prange(BOARD_SIZE): # prange enumerator
    # ...
    # only simple operations are allowed

@njit
def minimax(board, depth, alpha, beta, maximizingPlayer, curr, opp,
last_played_move, max_depth = 0):

  # ...
  if depth == 0:
    return heuristic(board, symbol) * (1 if maximizingPlayer else -1), None # use
of compiled function

  if maximizingPlayer:
    # ...
    for col, line in legal_moves(board, last_played_move): # compiled enumerator
      board[line, col] = symbol
      if np.sum(board == curr) > 3 and is_win(board, symbol): # compiled function
          # ...
```