Mateusz Woźniak 260292 22.05.2023

Sztuczna inteligencja i inżynieria wiedzy – lista 2

1. Wstęp teoretyczny

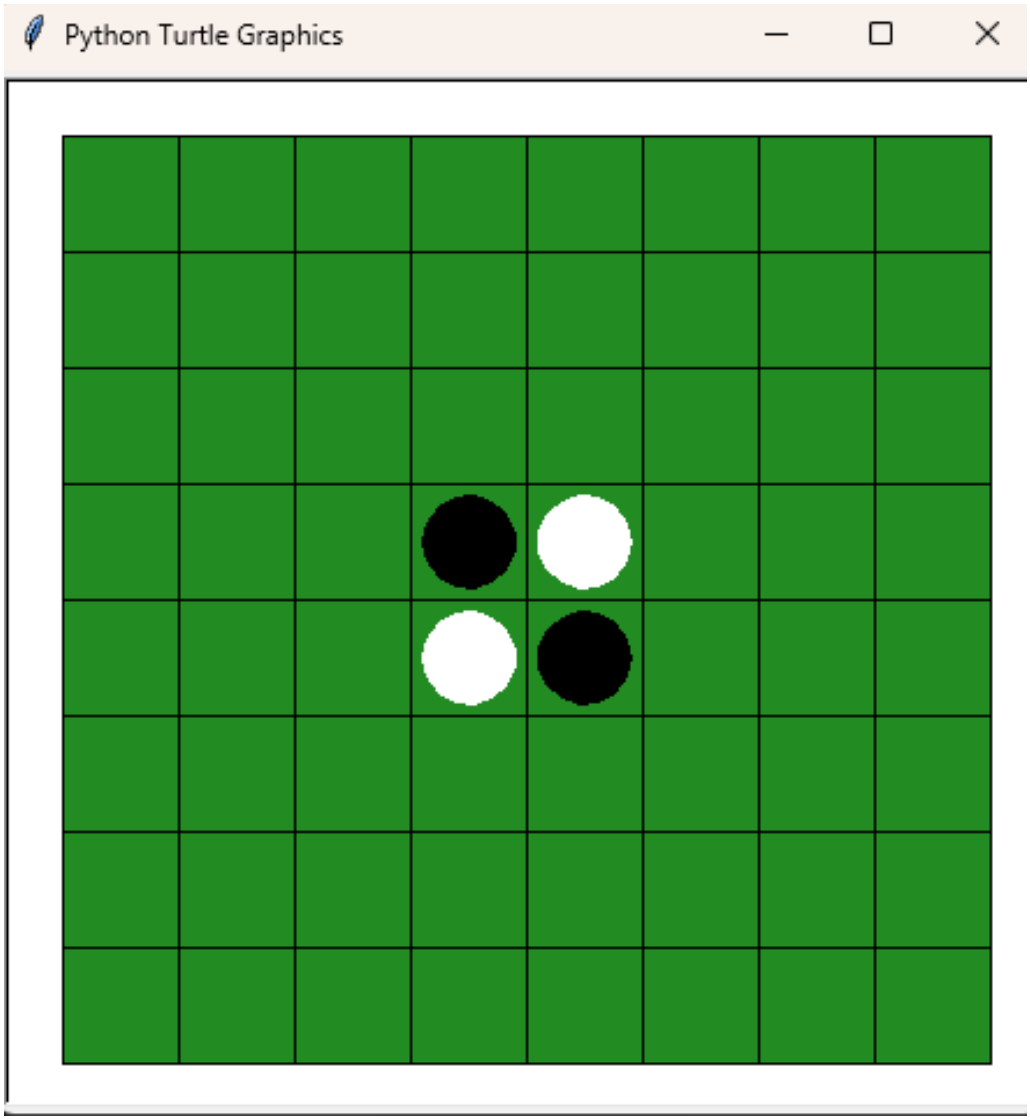
Algorytm Minimax jest algorytmem stosowanym w teorii gier do podejmowania decyzji poprzez minimalizowanie szkód oraz maksymalizowanie zysków. W przypadku Reversi/Othello opiera się on o wybieraniu najlepszego możliwego ruchu dla podmiotu uruchamiającego algorytm przy założeniu, że przeciwnik gra optymalnie. Algorytm analizuje drzewo decyzyjne – schodzi rekurencyjnie po gałęziach do czasu odnalezienia liścia, zakończenia gry lub osiągnięcia maksymalnej głębokości. Alfa-beta cięcie jest rozwinięciem Minimaxa, dzięki któremu możliwe jest pominięcie rozpatrywania niepotrzebnych węzłów.

Reversi/Othello to gra planszowa dla dwóch graczy, zazwyczaj na planszy o rozmiarze 8x8. Każde pole posiada trzy stany – puste pole, zajęte przez gracza 1, zajęte przez gracza 2. Po otoczeniu łańcucha pól przeciwnika swoimi polami, odwracają się one, zmieniając przynależność.

Drzewo decyzyjne reprezentuje wszystkie możliwe stany gry – możliwe konfiguracje plansz po różnych sekwencjach ruchów. Każdemu liściowi można przypisać wartość, dzięki której można zdeterminować najlepsze możliwe pole do wyboru.

2. Kod

Interfejs graficzny został zaprojektowany przy użyciu biblioteki *turtle,* będącej Pythonową implementacją znanego programu Logomocja. Autorem GUI jest użytkownik GitHuba – SiyanH. Repozytorium jest dostępne pod linkiem https://github.com/SiyanH/othello-game



Aby uniknąć wyświetlania badanych sekwencji ruchów na planszy użyłem biblioteki *copy* do wykonywania głębokich kopii z użyciem *deepcopy()*. Po dodaniu argumentu odpowiedzialnego za rysowanie do kilku istniejących już funkcji, można było przeszukiwać drzewa:

```
def make_move(self, draw=True):
    ''' Method: make_move
        Parameters: self
        Returns: nothing
        Does: Draws a tile for the player's next legal move on the
              board and flips the adversary's tiles. Also, updates the
              state of the board (1 for black tiles and 2 for white
              tiles), and increases the number of tiles of the current
              player by 1.
    '''
    if self.is_legal_move(self.move):
        self.board[self.move[0]][self.move[1]] = self.current_player + 1
        self.num_tiles[self.current_player] += 1
        if draw:
            self.draw_tile(self.move, self.current_player)
        self.flip_tiles(draw)

def flip_tiles(self, draw=True):
    ''' Method: flip_tiles
        Parameters: self
        Returns: nothing
        Does: Flips the adversary's tiles for current move. Also,
              updates the state of the board (1 for black tiles and
              2 for white tiles), increases the number of tiles of
              the current player by 1, and decreases the number of
              tiles of the adversary by 1.
    '''
    curr_tile = self.current_player + 1
    for direction in MOVE_DIRS:
        if self.has_tile_to_flip(self.move, direction):
            i = 1
            while True:
                row = self.move[0] + direction[0] * i
                col = self.move[1] + direction[1] * i
                if self.board[row][col] == curr_tile:
                    break
                else:
                    self.board[row][col] = curr_tile
                    self.num_tiles[self.current_player] += 1
                    self.num_tiles[(self.current_player + 1) % 2] -= 1
                    if draw:
                        self.draw_tile((row, col), self.current_player)
                    i += 1
```

Aby nadać wartości poszczególnym sekwencjom ruchów, zaproponowałem 3 heurystyki:

```python
def _taken(board):
    taken = 0
    for row in range(board.n):
        for col in range(board.n):
            if board.board[row][col] in [1,2]:
                taken += 1
    return taken / board.n ** 2


def count_tiles(board, player):
    score = 0
    for row in range(board.n):
        for col in range(board.n):
            if board.board[row][col] == (player + 1):
                score += 1
            elif board.board[row][col] != 0:
                score -= 1
    return score

def tile_value(board, player):
    weight = [
        [ 10,  -5,   5,   3,   3,   5,  -5,  10],
        [ -5,  -5,  -3,  -1,  -1,  -3,  -5,  -5],
        [  5,  -3,   3,   3,   3,   3,  -5,   5],
        [  3,  -1,   2,   1,   1,   2,  -1,   3],
        [  3,  -1,   2,   1,   1,   2,  -1,   3],
        [  5,  -5,   3,   3,   3,   3,  -5,   5],
        [ -5,  -5,  -3,  -1,  -1,  -3,  -5,  -5],
        [ 10,  -5,   5,   3,   3,   5,  -5,  10]
    ]

    score = 0
    for row in range(board.n):
        for col in range(board.n):
            if board.board[row][col] == (player + 1):
                score += weight[row][col]
            elif board.board[row][col] != 0:
                score -= weight[row][col]
    return score

def progressive_aggressiveness(board, player):
    coeff = _taken(board)
    prim_coeff = 1 - coeff
    return tile_value(board, player) * prim_coeff + count_tiles(board, player) * coeff
```

_taken – jest funkcją pomocniczą, obliczającą ilość zajętych pól (określa jak długo trwa gra)

count_tiles – nadaje sekwencji wartość równą ilości pól gracza po ruchach

tile_value – określa ważoną wartość pól gracza po ruchach (ujemne wartości określają niebezpieczne pola, czyli takie, które sa strategicznie trudne do utrzymania)

progressive_aggressiveness – jest połączeniem dwóch wyższych, ale im bliżej końca (_taken), tym bardziej zachowuje się jak tile_value

Po zaimplementowaniu heurystyk należało napisać algorytm Minimax:

```python
def minimax(board, player, depth, rating):
    bestMove = None

    if depth == 0 or board.is_over():
        return bestMove, rating(board, player)

    if player:
        maxEval = float('-inf')
        for move in board.get_legal_moves():
            board.move = move
            board.make_move(False)
            board.current_player = int(not player)
            _, eval = minimax(board, board.current_player, depth - 1, rating)
            if maxEval < eval:
                maxEval = eval
                bestMove = move
                print(move, maxEval, eval, bestMove)
        return bestMove, maxEval

    minEval = float('inf')
    for move in board.get_legal_moves():
        board.move = move
        board.make_move(False)
        board.current_player = int(not player)
        _, eval = minimax(board, board.current_player, depth - 1, rating)
        if minEval > eval:
            minEval = eval
            bestMove = move
    return bestMove, minEval
```

Czas trwania przeszukań na głębokości 2 (każdy z graczy dokonuje dwóch ruchów) trwał pomiędzy 0,02s a 0,05s:

```
Computer's turn.
Minimax move time: 0.03895759582519531
Your turn.
Computer's turn.
Minimax move time: 0.04849076271057129
Your turn.
Computer's turn.
Minimax move time: 0.02457118034362793
Your turn.
```

Zaś jego rozszerzenie – Alfa-beta cięcie – jest kwestią dodania kilku linijek kodu. I pomimo tak małych zmian, był w stanie skrócić czasy przeszukiwania średnio trzykrotnie – do ~0,01s.

```python
def abpruning(board, player, depth, rating):
    alpha = float('-inf')
    beta = float('inf')
    bestMove = None

    if depth == 0 or board.is_over():
        return bestMove, rating(board, player)

    if player:
        maxEval = float('-inf')
        for move in board.get_legal_moves():
            board.move = move
            board.make_move(False)
            board.current_player = int(not player)
            _, eval = abpruning(board, board.current_player, depth - 1, rating)
            if maxEval < eval:
                maxEval = eval
                bestMove = move
            alpha = max(alpha, maxEval)

            if beta <= alpha:
                return bestMove, alpha
        return bestMove, maxEval

    minEval = float('inf')
    for move in board.get_legal_moves():
        board.move = move
        board.make_move(False)
        board.current_player = int(not player)
        _, eval = abpruning(board, board.current_player, depth - 1, rating)
        if minEval > eval:
            minEval = eval
            bestMove = move
        beta = min(beta, minEval)

        if alpha <= beta:
            return bestMove, beta
    return bestMove, minEval
```

```
Your turn.
Computer's turn.
Alpha-beta pruning move time: 0.009756088256835938
Your turn.
Computer's turn.
Alpha-beta pruning move time: 0.008324623107910156
Your turn.
Computer's turn.
Alpha-beta pruning move time: 0.010531902313232422
Your turn.
```