LUNDS UNIVERSITET
Lunds Tekniska Högskola

Assignment 1
Search
Applied Artificial Intelligence (EDAF70)

Axel Voss
elt15avo@student.lu.se
Tom Andersson
elt14tan@student.lu.se

February 22, 2019

Supervisor:
Jacek Malec

Lunds Universitet
LTH

Contents

1 How to run

The main program can be found under **"h/dk/j/elt15avo/Repos/edaf70/lab1"**

To run the program simply type **"python3 game.py"** and a graphical user interface will launch.

The graphical user interface is then controlled solely using the computer mouse. First, the player choose which color to play with. Thereafter, the game will start. When it's the player's turn, red dots on the board indicates the possible moves. At any point during the game, the player can choose to restart or exit the game using the buttons in the top corners of the graphical user interface. The current score is displayed at the bottom.

2 Structure

The program consists of three classes, OthelloBoard, OthelloGui and Ai. The class OthelloBoard handles the board itself as well as the rules of the game. OthelloGui creates the graphical user interface where the game is being played. Finally, Ai is the class that implements the players opponent, an Ai. It is using a search algorithm and is evaluating the states of the board during the game. In this report, the Ai class will be explained in more detail since that was the focus of the assignment.

3 Board representation

As said in the structure section, the class Othelloboard handles everything that has to do with the representation of the game board. The board is represented as matrix. For example, when a move is evaluated or made it checks which of the adjacent positions that will be flipped to the current players color. This is shown in the code below.

```
def _get_discs(self, xy, player):
    x, y = self._to_num(xy)
    discs_to_flip = []
    if self._gameboard[x][y] is not None:
        return discs_to_flip

    for dx, dy in [[1, 0], [1, 1], [0, 1],
                   [-1, 1], [-1, 0], [-1, -1],
                   [0, -1], [1, -1]]:
        tx = x + dx
        ty = y + dy
        op_discs = []
        while self._on_board(tx, ty)
                and self._gameboard[tx][ty]is not None:
            disc = self._gameboard[tx][ty]
            if disc == player:
                if op_discs:
                    discs_to_flip += op_discs
                break
            else:
                op_discs.append(self._to_str(tx, ty))

            tx += dx
            ty += dy

    return discs_to_flip
```

4 Search algorithm

The Ai is implemented with the minimax algorithm including alpha beta pruning. To be able to prune more of the search tree, the tree is sorted so that the nodes with a higher chance of giving a higher static evaluation gets evaluated first. First, the sorting was implemented by sorting the tree using the complete static evaluation function described in section 5. However, with this the sorting time became too long and therefore a simpler sorting method was chosen. This method prioritizes solely based on position weights described in section 5.3. With the computers used during the development, the search algorithm can reach a depth of around 5 before the search time increases significantly.

When a time limit is set the Ai uses iterative deepening depth-first search together with the minimax algorithm. As long as the time passed have not exceeded half of the time limit the minimax function is called with one extra layer to be explored.

```
def time_limit_move(self):
    depth = 3
    start_time = time()
    _, move =
    self._minimax(
    self.board, depth, -inf, inf, self.player)
    passed_time = time()-start_time
    while(passed_time < self.time_limit/2):
        depth += 1
        _, move = self._minimax(
                self.board, depth,
                -inf, inf, self.player)
        passed_time = time() - start_time
    return move
```

5 Static Evaluation

The static evaluation function is used by the Ai to evaluate how advantageous different board states are. This is what the search tree is using for comparison. It consists of several helper functions and behaves differently depending on how many discs that have already been placed on the board, since the strategies in the game changes depending on how close the game is to be finished. The helper functions are also weighted differently depending on the importance of each function.

```
  def _static_evaluation(self, board, player):
    discs = board.score[player] +
            board.score[player*-1]
    score = 0
    if(board.current_player == 0):
        score +=
        100000*self._true_score(board, player)
        return score
    elif(discs < 19):
        score +=
        5*self._mobiliy(board, player)
        score +=
        20*self._position_weights(board,player)
        score +=
        10000*self._corners(board, player)
        score +=
        10000*self._corner_bonus(board, player)
        return score
    elif(discs < 57):
        score +=
        10*self._disc_difference(board, player)
        score +=
        2*self._mobility(board, player)
        score +=
        10*self._position_weights(board,player)
        score +=
        100*self._last_disc(board, player)
        score +=
        10000*self._corners(board, player)
        score +=
        10000*self._corner_bonus(board, player)
        return score
    else:
        score +=
        500*self._disc_difference(board,player)
        score +=
        500*self._last_disc(board, player)
        score +=
        10000*self._corners(board, player)
        score +=
```

```
        10000*self._corner_bonus(board, player)
        return score
```

5.1 True Score

True score returns the actual true score of the current board - ie, the difference between each players discs.

```
def _true_score(self, board, player):
    return board.score[player] - board.score[player * -1]
```

5.2 Mobility

The mobility function returns the ratio of the player advantage in mobility where mobility is calculated as the amount of legal moves a player can make during a turn.

```
def _mobiliy(self, board, player):
    own_moves =
    sum([len(x) for x in board.moves[player].values()])
    op_moves =
    sum([len(x) for x in board.moves[player*-1].values()])
    return 100 * (own_moves - op_moves)/
                 (own_moves + op_moves + 1)
```

5.3 Position Weights

This function returns the score of a players current disc positions. As some positions gives a slight advantage, e.g, corners and edges, the Ai should be rewarded for protecting those strategic positions. Here, corners are seen as the best position to have so the Ai will prioritize corners first. It will also prioritize to not take the positions that are surrounding a corner since that could give its opponent the possibility to take the corner. The Ai will also prioritize positions that gives its opponent the possibility to take a position that is surrounding the corner and thereby open up the corner for the Ai. Next in the prioritization line are the edges, the same logic as for the corners are applied to them, only with a lower prioritization compared to the corners.

5.4 Last Disc

Being the player placing the last disc is also an advantage since the game is ending after the last disc is placed.

```
def _last_disc(self, board, player):
    remaining_discs =64 - board.score[player]
                        - board.score[player * -1]
    if (remaining_discs % 2):
        return -1
    else:
        return 1
```

5.5 Corners

Getting corners is one of the key moves in Othello and having more corners than your opponent is a major advantage.

```
def _corners(self, board, player):
    corners = ['A1', 'H1', 'A8', 'H8']
    player = 0
    op = 0
    for corner in corners:
        disc = board.get_disc_value(corner)
        if disc == player:
            player += 1
        if disc == player * -1:
            op += 1
    return 100 * (player - op)/(player + op + 1)
```

5.6 Corner Bonus

When a corner is taken discs along the edges are starting to get stable -e.i, they can not be converted to the opponents color. Getting corners and generating stable disc for oneself increases the chance of victory.

```
def _corner_bonus(self, board, player):
    def _stable_discs(player):
        directions = {'A1': [1, 'H', 1, '8'],
                      'H1': [-1, 'A', 1, '8'],
                      'A8': [1, 'H', -1, '1'],
                      'H8': [-1, 'A', -1, '1']}
        stables = 0
        for key, value in directions.items():
            dx, xstop, dy, ystop = value
            for x in range(ord(key[0]),
            ord(xstop)+dx, dx):
                xy = chr(x)+key[1]
                if board.get_disc_value(xy) ==
                player:
                    for y in
                    range(ord(key[1])+dy,
                    ord(ystop)+dy, dy):
                        xy = chr(x)+chr(y)
                        if board.get_disc_value(xy)
                        == player:
                            stables += 1
                        else:
                            break
                else:
                    break
        return stables
    own_bonus = _stable_discs(player)
    op_bonus = _stable_discs(player * -1)
    return own_bonus - op_bonus
```