Veštačka inteligencija - izveštaj III faza

PinkTeam

Mijajlović Anđelija 18247

Joksimović Kristina 18203

1. Odigravanje igre računar protiv korisnika

Na osnovu početnog izbora korisnika, računar igra suprotnu boju. Poziva se funkcija play, koja ukoliko je korisnik u pitanju izračunava po koordinatama odigrani potez i odigrava ga pomoću funkcija iz prethodnih faza. Ukoliko je računar na potezu, poziva se **minimax** funkcija koja vraća potez, a on se zatim odigrava.

```
def play(self, movement):

    if(self.computer != self.currentPlayer): # korisnik

        x1, y1 = self.get_field_start(movement[0], movement[1])
        x2, y2 = self.get_field_start(movement[2], movement[3])

        row1 = int(y1 / self.squareSize)
        col1 = int(x1 / self.squareSize)
        row2 = int(y2 / self.squareSize)
        col2 = int(x2 / self.squareSize)

        clicked_bit = int(((row1 + 1) * self.squareSize) - y1 ) / self.bitHeight

        positionFrom = 0
        if(clicked_bit < 0):
            return None

        if(not self.board[row1][col1][1]):
            return None

        if(clicked_bit > self.board[row1][col1][1]):
            positionFrom = self.board[row1][col1][1] - 1
        else:
            positionFrom = int(clicked_bit)

        self.move(row1, col1, row2, col2, positionFrom)

    else:
        best_move = [0, 0, 0, 0, 0]
        self.minimax(0, self.NEG_INFINITY, self.POS_INFINITY, True, 1, 1, 0, 0, 0, best_move)

        self.move(best_move[0], best_move[1], best_move[2], best_move[3], best_move[4])
```

2. Minimax algoritam

Na slici je prikazano kako je implementiran minimax algoritam sa alfa- beta odsecanjem i ograničenjem dubine na logaritam dimenzije table.

```
NEG_INFINITY = float('-inf')
POS_INFINITY = float('inf')

def minimax(self, depth, alpha, beta, is_max_player, row_from, col_from, row_to, col_to, pos_from, best_move):
    if self.terminal(row_from, col_from, row_to, col_to, pos_from):
        return self.state_value(row_from, col_from)

    if depth == math.log(self.dim, 2):
        return self.evaluate(is_max_player, row_from, col_from, row_to, col_to, pos_from)

    if is_max_player:
        max_eval = self.NEG_INFINITY
        for move in self.calculate_all_possible_moves():
            eval_score = self.minimax(depth + 1, alpha, beta, False, move[0], move[1], move[2], move[3], move[4], best_move)
            if eval_score > max_eval:
                max_eval = eval_score
                if depth == 0:
                    best_move[0], best_move[1], best_move[2], best_move[3], best_move[4] = move[0], move[1], move[2], move[3], move[4]
            alpha = max(alpha, eval_score)
            if beta <= alpha:
                break
        return max_eval
    else:
        min_eval = self.POS_INFINITY
        for move in self.calculate_all_possible_moves():
            eval_score = self.minimax(depth + 1, alpha, beta, True, move[0], move[1], move[2], move[3], move[4], best_move)
            if eval_score < min_eval:
                min_eval = eval_score
                if depth == 0:
                    best_move[0], best_move[1], best_move[2], best_move[3], best_move[4] = move[0], move[1], move[2], move[3], move[4]
            beta = min(beta, eval_score)
            if beta <= alpha:
                break
        return min_eval
```

U pomoćne funkcije spada provera da li je stanje terminalno, koja određuje da li neki od igrača ima broj stekova potrebnih za pobedu. Nakon toga, određuje se vrednost terminalnog stanja:

```
def terminal(self, row_from, col_from, row_to, col_to, pos_from):
    if(self.users[0].score + self.users[1].score < self.maxStacks - 1):
        return False
    if(self.board[row_from][col_from][1] - pos_from + self.board[row_to][col_to][1] < 8):
        return False

    return True
```

```
#ako je terminal, ova funkcija odredjuje pobednika
def stateValueTerminal(self, row_from, col_from):
    if(self.readBit(row_from, col_from, 7) == 1 and self.users[1].score > self.users[0].score):
        return 10
    elif(self.readBit(row_from, col_from, 7) == 0 and self.users[0].score > self.users[1].score):
        return -10
    else:
        return 0
```

3. Evaluacija stanja

Na datoj ograničenoj dubini određuje se vrednost stanja koje nije terminalno pomoću funkcije evaluate. Na rezultat evaluacije utiču razni faktori. Uzeto je u obzir sledeće:

1. Stanje je bolje ukoliko ima više figurica tekućeg igrača
2. Najpre se pomeraju figurice koje su same na polju
3. Prednost se daje ukoliko je na vrhu tekućeg steka boja tekućeg igrača i ukoliko je broj figura veći u rezultujućem steku
4. Prednost se daje ako je rezultujuća pozicija iznad figurice protivnika
5. Prednost se daje ako je rezultujuća pozicija u okviru steka parna
6. Kretanje (od krajeva) ka centru, u zavisnosti od popunjenosti table
7. Ukoliko se pomeraju figurice iz steka i postoji ta mogućnost, pomeraju se tako da na vrhu početnog steka ostane figura tekućeg igrača

```python
def evaluate(self, is_max_player, row_from, col_from, row_to, col_to, pos_from):
    total_score = 0

    #tezine su u zbiru 10
    piece_count_weight = 1
    piece_number_weight = 1
    sum_of_pieces_weight = 1
    top_color_weight = 1
    new_position_weight = 1
    direction_weight = 4
    stack_division_weight = 1

    piece_count_score = self.evaluate_piece_count()
    total_score += piece_count_score * piece_count_weight

    piece_number_score = self.evaluate_piece_number(row_from, col_from)
    total_score += piece_number_score * piece_number_weight

    sum_of_pieces_score = self.evaluate_sum_of_pieces(row_from, col_from, row_to, col_to, pos_from)
    total_score += sum_of_pieces_score * sum_of_pieces_weight

    top_color_score = self.evaluate_top_color(row_from, col_from)
    total_score += top_color_score * top_color_weight

    new_position_score = self.evaluate_new_position(row_from, col_from, row_to, col_to, pos_from)
    total_score += new_position_score * new_position_weight

    direction_score = self.evaluate_direction(row_from, col_from, row_to, col_to)
    total_score += direction_score * direction_weight

    stack_division_score = 0
    if pos_from != 0:
        stack_division_score = self.evaluate_stack_division(row_from, col_from, row_to, col_to, pos_from)
    total_score += stack_division_score * stack_division_weight

    total_score /= 5.6

    if(is_max_player):
        return total_score
    else:
        return -total_score
```

```
def evaluate_stack_division(self, row_from, col_from, row_to, col_to, pos_from):
    if( self.readBit(row_from, col_from, pos_from - 1) == self.computer):
        return 8
    else:
        return 0
```

```
def evaluate_direction(self, row_from, col_from, row_to, col_to):
    non_empty_elements = 0

    for row in self.board:
        for element in row:
            if(element[1]>0):
                non_empty_elements += 1

    occupancy = non_empty_elements / self.bit

    total_score = 0

    if(occupancy > 0.5):
        if(row_from < self.dim/2):
            total_score += row_to - row_from + self.dim/2 - row_from
        else:
            total_score += row_from - row_to + row_from - self.dim/2

        if(col_from < self.dim/2):
            total_score += col_to - col_from + self.dim/2 - col_from
        else:
            total_score += col_from - col_to + col_from - self.dim/2

    return total_score

def evaluate_new_position(self, row_from, col_from, row_to, col_to, pos_from):
    total_score = 0
    if(self.readBit(row_to, col_to, self.board[row_to][col_to][1] - 1) != self.computer):
        total_score += 4
    if (self.board[row_to][col_to][1] + self.board[row_from][col_from][1] - pos_from - 1) % 2 == 0:
        total_score += 4
    return total_score
```

```
def evaluate_top_color(self, row_from, col_from):
    if self.readBit(row_from, col_from, self.board[row_from][col_from][1] - 1) == self.computer:
        return 8
    return 0

def evaluate_sum_of_pieces(self, row_from, col_from, row_to, col_to, pos_from):
    return self.board[row_to][col_to][1] + self.board[row_from][col_from][1] - pos_from

def evaluate_piece_number(self, row, col):
    return 8 - self.board[row][col][1]

def evaluate_piece_count(self):
    max_count = 0
    min_count = 0

    for row in range(self.dim):
        for col in range(self.dim):
            for i in range(self.board[row][col][1]):
                if self.readBit(row, col, i):
                    max_count += 1
                else:
                    min_count += 1

    return max_count - min_count
```

U okviru play funkcije, nakon minmax algoritma,, koričćene su funkcije iz prethodne faze projekta, koje svakako kontrolisu igru sve vreme, proveravaju moguće i validne poteze.

4. Izračunavanje dozvoljenih poteza za sve figure konkretnog igrača na potezu.

Funkcija prolazi kroz sve bitove svakog bolja, i trazi svog konkretnog igrača koji je na potezu i za njega, proverava njegovu validnost a nakon toga i valjanost.
Nakon provere valjanosti, obzirom da su potezi preko dijagonala i oni dijagonalni odvojeni, postoje provere koje će vratiti tačno dozvoljene poteze, odnosno, ukoliko postoji polje koje nema prazne dijagonale oko sebe, imaće prednost nad svim ostalim potezima.

```
def calculate_all_possible_moves(self):
    possible_moves_best = []
    possible_moves_empty_diagonals = []

    for row in range(self.dim):
        for col in range(self.dim):

            # za svaki bit
            for bit_position in range(self.board[row][col][1] - 1, -1, -1):
                bit = self.readBit(row, col, bit_position)
                if bit == self.currentPlayer:  # da li odgovara trenutnom igracu
                    for dr, dc in [(-1, -1), (-1, 1), (1, -1), (1, 1)]:
                        new_row, new_col = row + dr, col + dc
                        if 0 <= new_row < self.dim and 0 <= new_col < self.dim:
                            # validnost
                            if self.valid_move(row, col, new_row, new_col, bit):
                                keys_from_stack_rules = self.stackRules(row, col, new_row, new_col, bit_position)
                                if isinstance(keys_from_stack_rules, dict):
                                    new_moves = [(row, col, k[0], k[1], v, bit_position) for k, v in keys_from_stack_rules.items()]
                                    for move in new_moves:
                                        if move not in possible_moves_empty_diagonals:
                                            possible_moves_empty_diagonals.append(move)

                                if keys_from_stack_rules is True: #ovde bi bio najbolji potez
                                    possible_moves_best.append((row, col, new_row, new_col, bit_position))

    if len(possible_moves_best) == 0:
        min_v = min(item[4] for item in possible_moves_empty_diagonals)
        possible_moves_empty_diagonals = [(elem[0], elem[1], elem[2], elem[3], elem[5]) for elem in possible_moves_empty_diagonals if elem[4] == min_v]

        return possible_moves_empty_diagonals

    return possible_moves_best
```

5. Vraćanje svih dozvoljenih poteza sa datog polja.

Pomoćna funkcija funkciji stackRules koja ce za svako obrađeno polje tražiti najbolji sledeći skok. Funkcija predstavlja modifikovani BFS algoritam, kreće se od traženog polja i obilaziće u širinu sve dok ne naiđe na polje koje sadrži neki stek. Tada je kao pomoćna, iskorišćena funkcija return_position koja koristi ideju algoritma traženja A* i vraća putanju nazad do skoka koji će zapravo predstavljati najbolji izbor za to polje.

```
def find_nearest_nonzero(self, start_row, start_col):
    directions = [(-1, -1), (-1, 1), (1, -1), (1, 1)]
    visited = [[False for _ in range(self.dim)] for _ in range(self.dim)]
    queue = deque([(start_row + dr, start_col + dc)
        for dr, dc in directions
        if 0 < start_row + dr < len(self.board) and 0 < start_col + dc < len(self.board[0])
        ])

    lista = list(queue)
    lista2 = list()
    allowed_elements = dict()
    my_dict = {(start_row, start_col): lista}
    counter = 0
    while queue:

        if len(lista) == 0:
            if len(my_dict) >= 1:
                allowed_elements = {k: v for k, v in allowed_elements.items() if k is not None}
                if len(allowed_elements) > 0:
                    return allowed_elements
                else:
                    elements_to_add = list(queue)[:counter]
                    lista.extend(elements_to_add)
            else: #naredna distanca, da znam zbog vracanja unazad
                elements_to_add = list(queue)[:counter]
                lista.extend(elements_to_add)

        else:
            current_row, current_col= queue.popleft()
            lista2.clear()
            lista.remove((current_row,current_col))
            visited[current_row][current_col] = True



           for dr, dc in directions:
               new_row, new_col = current_row + dr, current_col + dc
               distance = self.calculate_distance(start_row, new_row, start_col, new_col)
               #value
               lista2.append((new_row,new_col))
               my_dict[(current_row, current_col)] = lista2

               if(new_row >= 0 and new_row < self.dim and new_col >= 0 and new_col < self.dim): #dodata provera
                   if self.board[new_row][new_col][1] > 0:
                       if(distance == 2):
                           allowed_elements[(current_row,current_col)] = distance
                       else:
                           allowed_elements[self.return_position((new_row, new_col), (start_row, start_col), my_dict)] = distance

               if (0 <= new_row < self.dim and 0 <= new_col < self.dim and not visited[new_row][new_col]
                   and (new_row != start_row or new_col != start_col)):
                   queue.append((new_row, new_col))

               counter += 1

    allowed_elements = {k: v for k, v in allowed_elements.items() if k is not None}
    return allowed_elements
```

```
def return_position(self, start, target, my_dict):
    visited = set()
    queue = [(start, [start])]

    while queue:
        current, path = queue.pop(0)
        if current == target:
            #da li je startna pozicija u putanji
            if start in path:
                # prva pre ciljne
                for i in range(len(path) - 1, 0, -1):
                    if path[i] == target and i > 1:  # element pre ciljne
                        return path[i - 1]
            else:

                return path

        if current not in visited:
            visited.add(current)
            for key, value in my_dict.items():
                neighbors = value
                if current in neighbors:
                    queue.append((key, path + [key]))
    return None
```

6. Provera valjanosti poteza

Funkcija stackRules će nakon što se utvrdi validnost poteza, utvrđivati valjanost isti i ukoliko su osnovni uslovi (pomeranje je u rangu, ukupan proj u steku je 8, pozicija sa koje se pomeramo je manja od one na koju skačemo) ispunjeni, poziva se find_nearest_nonzero.
Ukoliko nisu prazne dijagonale, funkcija će vratiti true/false vrednosti koje će u funkciji calculate_all_possible_moves biti pomoć da vrate samo te pozicije kao moguće opcije.

```
def stackRules(self, row1, col1, row2, col2, positionFrom):
    nearest_nonzero = dict()

    #pozicija sa koje se pomera je u rangu
    if(positionFrom < 0 or positionFrom >= self.board[row1][col1][1]):
        return None

    #broj ukupnih bitova na novom steku je manji od 8
    if(self.board[row2][col2][1] + self.board[row1][col1][1] - positionFrom > 8):
        return False

    if(positionFrom > self.board[row2][col2][1]):
        return False

    if(positionFrom == self.board[row2][col2][1]):
        if(self.areDiagonalEmpty(row1, col1)):
            #naci najblizi stek i proveriti da li je u pravcu
            nearest_nonzero = self.find_nearest_nonzero(row1, col1)
            if nearest_nonzero:
                return nearest_nonzero
            else:
                return False

        return False

    return True
```