Vestačka inteligencija

Izveštaj I faze projekta

Byte

Formulacija problema i interfejs

Stefan Stanković 18395,

Ksenija Seizović 18367

Predstavljanje stanja problema

Za predstavljanje stanja igre koristi se klasa Game koja sadrzi 2 instance klase Player (X i O), ko je sledeći na potezu i instancu klase Board koja se kreira i inicijalizuje na osnovu prosledjenog parametra o dimenzijama.

Očekuju se 2 već kreirana igrača sa podešenim modovima da li je igrač čovek ili mašina i ko igra prvi potez. Usvojeno je da igrač X uvek prvi započinje igru, pa se odabirom X ili O figura, kao parametar igrača reguliše prvenstvo na početku.

```
class Game:
    def __init__(self, player1: Player, player2: Player, dimensions: int):
        if player1.byte_color == "X":
            self.playerX = player1
            self.playerO = player2
        else:
            self.playerX = player2
            self.playerO = player1

        self.play_turn = "X"
        self.board = Board()
        self.board.initialize_board(dimensions)
```

```
class Player:
    def __init__(self, isHuman: bool, byte_color: str):
        self.score = 0
        self.isHuman = isHuman
        self.byte_color = byte_color
```

```
class Board:
    def __init__(self):
        self.dim = -1
        self.board = []
```

Funkcija za postavljanje početnog stanja

Na osnovu prosledjene velične table, ako je u opsegu 8-16, kreira se tabla zadatih dimenzija. Na osnovu veličine se takodje računa broj figura koje treba postaviti. Njihova inicijalna pozicija je definisana pravilima igre - figure jednog igrača se nalaze u parnim, a drugog u neparnim redovima, pri čemu su prvi i poslednji red prazni.

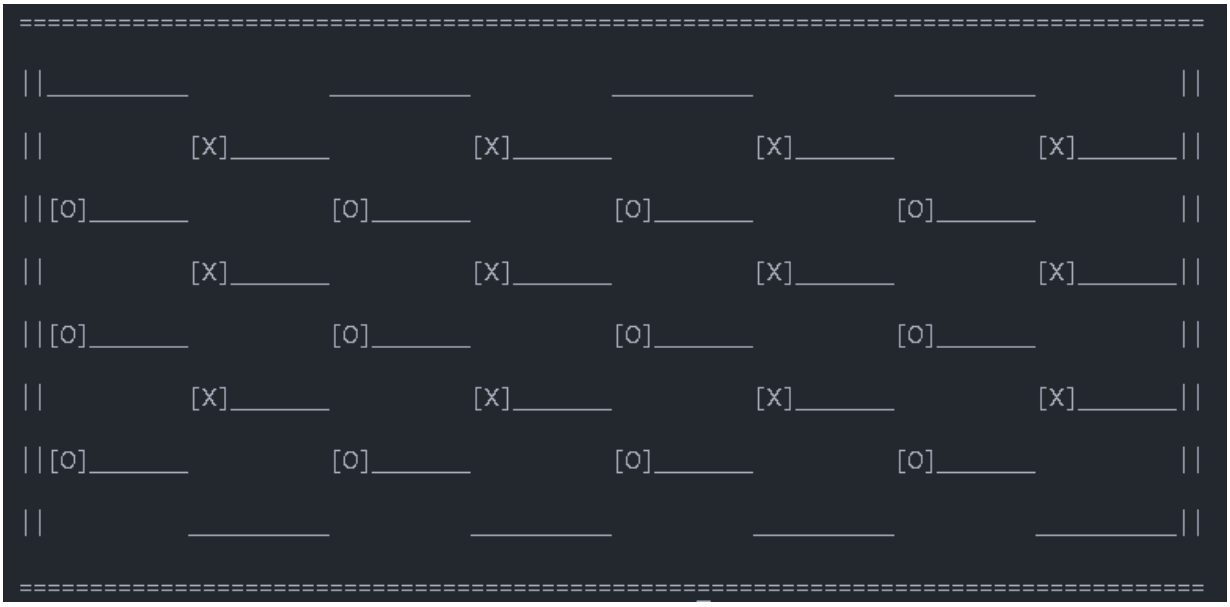
```
class Board:
    def __init__(self):
        self.dim = -1
        self.board = []

    def initialize_board(self, dim):
        """Initialize the board dimension: dim x dim, and set the figures. Note: dim in range 8 - 16"""
        if(dim < 8 or dim > 16):
            return False
        if (dim - 2) * dim / 2 % 8 != 0:
            return False

        self.dim = dim
        for i in range(0,dim):
            self.board.append([])
            for j in range(0,dim):
                if((i+j)%2 == 0):
                    self.board[i].append(None)
                else:
                    self.board[i].append(False)
        row = 0
        black = False
        while row < dim:
            column = 1 if row % 2 == 1 else 0
            while column < dim:
                self.board[row][column] = (
                    Byte("", (row, column))
                    if row < 1 or row >= dim - 1
                    else Byte("X" if black else "O", (row, column))
                )
                column += 2
            row+=1
            black = not black
```

Funkcija za prikaz stanja

Funkcija članica klase Board koja prikazuje aktuelno stanje table, tj. štampa njenu grafičku reprezentaciju u konzoli. Sva validna polja na kojima se može stati (crna polja) su označena _____. Polja na kojima se nalaze naredjane figure prikazuju taj redosled u uglastim zagradama sleva nadesno, primer [XOOXX].

```
def print_board(self):
    for i in range(0,self.dim*10+4):
        print("=", end="")
    print()
    print()
    for i in range(0,self.dim):
        print('||', end="")
        for j in range(0, self.dim):
            if self.board[i][j] == False:
                print("          ", end='')
            elif len(self.board[i][j].colors) == 0:
                print("__________", end='')
            else:
                n = 8 - len(self.board[i][j].colors)
                print(self.board[i][j].to_string(), end='')
                for k in range(0,n):
                    print('_', end="")
        print('||',end="")
        print()
        print()
    for i in range(0,self.dim*10+4):
        print("=", end="")
    print()
```

Prikaz početnog stanja table u konzoli:



Funkcija za proveru kraja igre

Funkcije članice klase Game. Proverava se da li je došlo do kraja igre - da je tabla prazna tj. nema figura, ili da je neki igrač pobedio jer u vlasništvu ima više od polovine mogućih stekova.

```
def is_board_empty(self):
    return self.board.is_empty()

def has_player_won(self):
    if self.playerX.score > 3 * self.board.dim // 8 // 2:
        return True
    if self.playerO.score > 3 * self.board.dim // 8 // 2:
        return True
    return False

def is_game_over(self):
    return self.is_board_empty() or self.has_player_won()
```

Funkcija za proveru ispravnosti poteza

Deo o ispravnosti poteza koji se odnosi na odabir pozicije na kojoj će se postaviti figura/stek figura se nalazi u klasi Player. Pre nego što se odigra željeni potez, potrebno je potvrditi njegovu ispravnost. Ispituje se da li je uneto polje postoji na tabli i da li je u okviru polja table (ne može se ići van granica) – funkcija *is_move_out_of_bound*. Takodje ispituje se da li postoje figure na zadatom polju – *is_byte_empty* i da li je indeks od koga se pomeraju figure u steku validan – *is_index_in_byte_correct*.

```
def is_move_out_of_bound(self, board: Board, src_byte, move) -> bool:
    if src_byte[0] == "A" and move[0] == "G":
        return True
    if src_byte[0] == letters_to_numbers[board.dim] and move[0] == "D":
        return True
    if src_byte[1] == "1" and move[1] == "L":
        return True
    if src_byte[1] == str(board.dim) and move[1] == "D":
        return True
    return False

def is_byte_empty(self, board: Board, src_byte) -> bool:
    return len(board.board[letters_to_numbers[src_byte[0]]][int(src_byte[1])-1]) > 0

def is_index_in_byte_correct(self, board: Board, src_byte, index_in_byte) -> bool:
    if(board.board[letters_to_numbers[src_byte[0]]][int(src_byte[1]) - 1].get_color(index_in_byte) != -1):
        return True
    return False

def test_move(self, board: Board, src_byte, move, index_in_byte):
    return (
        not self.is_move_out_of_bound(board, src_byte, move)
        and not self.is_byte_empty(board, src_byte)
        and not self.is_index_in_byte_correct(board, src_byte, index_in_byte)
    )
```

U klasi Board se ispituje da li je stanje na tabli validno nakon odigranog poteza tj. da li ne postoji stek veličine veće od 8, u slučaju da te figure nisu uklonjene.

```
def is_tile_black(self, i, j):
    return (i+j)%2==0

def is_tile_white(self,i,j):
    return (i+j)%2==1

def is_state_valid(self) -> bool:
    for i in range(0, self.dim):
        for j in range(0, self.dim):
            if(self.is_tile_white(i,j) and self.board[i][j] != False):
                return False
            if(self.is_tile_black(i,j) and len(self.board[i][j].colors) > 8):
                return False
    return True
```

Funkcija za igranje poteza

Funkcija članice klase Player. Za sada funkcije samo sa standardnog ulaza očekuje 3 unosa i proveru da li može izvršiti željeni potez. Najpre treba uneti poziciju figure koju želimo pomeriti. Zatim se očekuje smer kretanja navodjenjem jedne od opcija: GL (gore levo), GD (gore desno), DL (dole levo), DD (dole desno). Na kraju se unosi broj figura koje se prenose sa steka navodjenjem indeksa od kog počinje.

```
def play_move(self, board: Board):
    print("It is your turn to play!")
    src_byte = ""
    move = ""
    index_in_byte = ""

    print("Source byte: ", end="")
    src_byte = input()

    print("Move: ", end="")
    move = input()

    print("Index of src byte: ", end="")
    index_in_byte = input()
    if not self.test_move(board, src_byte, move, index_in_byte):
        print("Bad move. Please try again... Press anything to continue.")
        return False

    return True
```

Funkciju odigravanja poteza pozivaju igrači u funkciji *start_game* klase Game sve dok se igra ne završi. Igrači treba da izgraju naizmenično jedan pa drugi.

```
def start_game(self):
    while True:
        if(self.is_board_state_valid()):
            self.show_state()
        if(self.is_game_over()):
            break

        if self.play_turn == "X":
            if self.playerX.isHuman == True:
                if self.playerX.play_move(self.board):
                    self.play_turn = "O"
                    continue
                input()
            else:
                #self.playerX.play_best_move()
                print('Computer turn')
        else:
            if self.playerO.isHuman == True:
                if self.playerO.play_move():
                    self.play_turn = "X"
                    continue
                input()
            else:
                #self.playerO.play_best_move()
                print('Computer turn')

    print("The game is over.")
```

Klasa Byte

```
class Byte:
    def __init__(self, col: str, coords: Tuple[int,int]):
        self.colors = []
        self.colors.extend(list(col))
        self.coords = coords

    def get_color(self, ind):
        try:
            if(len(self.colors)==0):
                return -1
            return self.colors[ind]
        except:
            print("Indexing error: ", ind)
            return -1

    def to_string(self):
        returningString = '['
        for color in self.colors:
            returningString += color
        returningString += ']'
        return returningString

    def move_to(self, byte, startingIndex):
        if(not self.is_movable(byte, startingIndex)):
            return False
        self.colors.append(byte.colors)

    def is_movable(self, byte, startingIndex):
        #To be implemented in the next phase
        i = 0
```

Faza 2

Funkcije za proveru valjanosti poteza

```
def find_all_possible_moves(self):
    """A function that returns the list of all possible moves of the current player"""
    i = 0
    j = 0
    possibleMoves = []

    while i < self.board.dim:
        j = 0 if i % 2 == 0 else 1
        while j < self.board.dim:
            for k in range(0, len(self.board.board[i][j].colors)):
                currentIndexColor = self.board.board[i][j].get_color(k)
                if currentIndexColor == self.play_turn:
                    areNeighboursEmpty = self.board.are_neighbours_empty(i, j)
                    if areNeighboursEmpty:
                        # DFS za GL, GD, DL, DD i da se nadje broj poteza do steka
                        GL = self.board.find_nearest_stack_iterative(i, j, i - 1, j - 1)
                        GD = self.board.find_nearest_stack_iterative(i, j, i - 1, j + 1)
                        DL = self.board.find_nearest_stack_iterative(i, j, i + 1, j - 1)
                        DD = self.board.find_nearest_stack_iterative(i, j, i + 1, j + 1)

                        minimum = min([GL, GD, DL, DD])
                        if GL == minimum:
                            srcByte = mappings.numbers_to_letters[i] + str(j + 1)
                            possibleMoves.append((srcByte, "GL", k))
                        if GD == minimum:
                            srcByte = mappings.numbers_to_letters[i] + str(j + 1)
                            possibleMoves.append((srcByte, "GD", k))
                        if DL == minimum:
                            srcByte = mappings.numbers_to_letters[i] + str(j + 1)
                            possibleMoves.append((srcByte, "DL", k))
                        if DD == minimum:
                            srcByte = mappings.numbers_to_letters[i] + str(j + 1)
                            possibleMoves.append((srcByte, "DD", k))
                    else:
```

```
                    else:
                        GL = self.board.is_movable_from_to(i - 1, j - 1, i, j, k)
                        GD = self.board.is_movable_from_to(i - 1, j + 1, i, j, k)
                        DL = self.board.is_movable_from_to(i + 1, j - 1, i, j, k)
                        DD = self.board.is_movable_from_to(i + 1, j + 1, i, j, k)
                        if GL:
                            if not self.board.board[i-1][j-1].is_empty():
                                srcByte = mappings.numbers_to_letters[i] + str(j + 1)
                                possibleMoves.append
                        if GD:
                            if not self.board.board[i-1][j+1].is_empty():
                                srcByte = mappings.numbers_to_letters[i] + str(j + 1)
                                possibleMoves.append((srcByte, "GD", k))
                        if DL:
                            if not self.board.board[i+1][j-1].is_empty():
                                srcByte = mappings.numbers_to_letters[i] + str(j + 1)
                                possibleMoves.append((srcByte, "DL", k))
                        if DD:
                            if not self.board.board[i+1][j+1].is_empty():
                                srcByte = mappings.numbers_to_letters[i] + str(j + 1)
                                possibleMoves.append((srcByte, "DD", k))
            j += 2
        i += 1
    return possibleMoves
```

(variable) i: int

*Funkcija **find_all_possible_moves** pronalazi listu svih mogucih poteza

U sebi, ova funkcija koristi pomocne funkcije:

-find_nearest_stack_iterative

-is_movable_from_to

```
def find_nearest_stack_iterative(self, iStart, jStart, iCurrent, jCurrent):
    if iCurrent < 0 or iCurrent >= self.dim or jCurrent < 0 or jCurrent >= self.dim:
        return 100000
    nodesToVisit = queue.Queue(self.dim*self.dim/2)
    nodesToVisit.put((iCurrent, jCurrent, 1))
    visitedNodes = set()

    while not nodesToVisit.empty():
        currentNode = nodesToVisit.get()
        iCur = currentNode[0]
        jCur = currentNode[1]
        roadLen = currentNode[2]

        if(not (iCur, jCur) in visitedNodes):
            visitedNodes.add((iCur, jCur))
            if (not self.board[iCur][jCur].is_empty()) and (iCur != iStart or jCur != jStart):
                return roadLen
            if iCur - 1 >= 0 and jCur - 1 >= 0 and not (iCur-1,jCur-1) in visitedNodes:
                nodesToVisit.put((iCur - 1, jCur - 1, roadLen + 1))
            if iCur - 1 >= 0 and jCur + 1 < self.dim and not (iCur-1,jCur+1) in visitedNodes:
                nodesToVisit.put((iCur - 1, jCur + 1, roadLen + 1))
            if iCur + 1 < self.dim and jCur - 1 >= 0 and not (iCur+1,jCur-1) in visitedNodes:
                nodesToVisit.put((iCur + 1, jCur - 1, roadLen + 1))
            if iCur + 1 < self.dim and jCur + 1 < self.dim and not (iCur+1,jCur+1) in visitedNodes:
                nodesToVisit.put((iCur + 1, jCur + 1, roadLen + 1))
```

```
def is_movable_from_to(self, iFrom, jFrom, iTo, jTo, startingIndex):
    '''
    Returns True if its possible to move the stack on position (iFrom, jFrom), starting from the startingIndex, to (iTo, jTo)\n
    Returns False if the index is out of bound, if its not possible
    '''
    if(iFrom < 0 or iFrom >= self.dim or jFrom < 0 or jFrom >= self.dim):
        return False

    return self.board[iTo][jTo].is_movable(self.board[iFrom][jFrom], startingIndex)
```

```
def is_movable(self, byte, startingIndex):
    # self.colors[startingIndex : ] -> byte
    lenByte = len(byte.colors)
    lenSelf = len(self.colors)

    # premestanje na prazno polje
    if lenByte == 0 and startingIndex == 0:
        return True

    # ukoliko bi duzina bila veca od 8
    if lenByte + lenSelf - startingIndex > 8:
        return False

    # ukoliko startingIndex dolazi na manju ili jednaku poziciju
    if startingIndex >= lenByte:
        return False

    # u suprotnom (ne premesta se prazno polje, duzina je manja ili jednaka 8, dize se na polje vece pozicije)
    return True
```

Operator promene stanja

Nakon pronalazenja liste poteza, sami potezi se prikazuju igracu, i igrac iz CMD prompta unosi zeljen potez. Igranje poteza se desava u funkciji **play_move**

```
def play_move(self, board: Board, playTurn, possibleMoves):
    print("It is your turn to play! Color: " + self.byte_color)
    src_byte = ""
    move = ""
    index_in_byte = ""

    print("Source byte: ", end="")
    src_byte = input()
    possibleMovesFromSrcByte = list(filter(lambda current: src_byte == current[0] , possibleMoves))

    if len(possibleMovesFromSrcByte) == 0:
        print("Bad move. Please try again... Press anything to continue.")
        return (False, False, False, False)

    print(possibleMovesFromSrcByte)
    # izbacuje listu mogucih poteza za src_byte

    print("Index of src byte: ", end="")
    index_in_byte = input()
    possibleMovesFromIndex = list(filter(lambda current: int(index_in_byte) == current[2] , possibleMovesFromSrcByte))

    if len(possibleMovesFromIndex) == 0:
        print("Bad move. Please try again... Press anything to continue.")
        return (False, False, False, False)
    print(possibleMovesFromIndex)

    print("Move: ", end="")
    move = input()
    finalMove = list(filter(lambda current: move == current[1], possibleMovesFromIndex))

    if(len(finalMove) == 0):
        print("Bad move. Please try again... Press anything to continue.")
        return (False, False, False, False)

    if not self.test_move(board, src_byte, move, index_in_byte, playTurn):
        print("Bad move. Please try again... Press anything to continue.")
        return (False, False, False, False)

    iFrom = letters_to_numbers[src_byte[0]]
    jFrom = int(src_byte[1]) - 1

    iTo = iFrom + (1 if move[0] == 'D' else -1)   # D - dole
    jTo = jFrom + (1 if move[1] == 'D' else -1)   # D - desno

    lenOfByte = board.board[iFrom][jFrom].move_to_byte(board.board[iTo][jTo], int(index_in_byte))
    return (True, lenOfByte, iTo, jTo)
```

Menjanje stanja, odnosno operator promene stanja predstavlja funkcija **move_to_byte**

```
def move_to_byte(self, byte, startingIndex):
    """Move from self[startingIndes] to byte[top]."""
    if not self.is_movable(byte, startingIndex):
        return False

    byte.colors = byte.colors + self.colors[startingIndex:]
    self.colors = self.colors[:startingIndex]
    return len(self.colors)
```

Faza 3 – MIN MAX (+ alpha-beta prunning)

```python
def get_best_move(self):
    best_move = None
    max_eval = float('-inf')
    min_eval = float('inf')
    alpha = float('-inf')
    beta = float('inf')
    current_depth = 2

    possible_moves = self.find_all_possible_moves()

    for move in possible_moves:
        current_state = deepcopy(self)
        current_state.make_move(move)
        eval = current_state.minimax(current_depth, alpha, beta, self.play_turn == "X")

        if self.play_turn == 'X':
            if eval > max_eval:
                max_eval = eval
                best_move = move
                alpha = max(alpha, eval)
        else:
            if eval < min_eval:
                min_eval = eval
                best_move = move
                beta = min(beta, eval)


    current_depth += 1
    return best_move
```

```python
def minimax(self, depth, alpha, beta, maximizing_player):
    if depth == 0 or self.is_game_over():
        return self.utility(maximizing_player)

    possible_moves = self.find_all_possible_moves()

    if maximizing_player:
        max_eval = float('-inf')
        for move in possible_moves:
            current_state = deepcopy(self)
            current_state.make_move(move)
            eval = current_state.minimax(depth - 1, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)

            if beta <= alpha:
                break

        return max_eval
    else:
        min_eval = float('inf')
        for move in possible_moves:
            current_state = deepcopy(self)
            current_state.make_move(move)
            eval = current_state.minimax(depth - 1, alpha, beta, True)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)

            if beta <= alpha:
                break

        return min_eval
```

Utility je pomocna funkcija koja sluzi za evaluaciju table:

```
def utility(self, maximize):
    if maximize:
        return self.utility_maximize_player()
    return self.utility_minimize_player()
```

```
def utility_minimize_player(self):
    util = 0

    for i in range(self.board.dim):
        for j in range(self.board.dim):
            if not self.board.is_tile_white(i, j):
                stack_height = len(self.board.board[i][j].colors)

                for k in range(0, stack_height):
                    if k == 0:
                        if self.board.board[i][j].get_color(0) == "O":
                            util -= 7
                        else:
                            util += 7
                    elif k == 7:
                        if self.board.board[i][j].get_color(k) == "O":
                            util -= 50
                        else:
                            util += 50
                    elif k == stack_height - 1:
                        if k < 5:
                            if self.board.board[i][j].get_color(k) == "O":
                                util -= 6
                            else:
                                util += 6
                        else:
                            if self.board.board[i][j].get_color(k) == "O":
                                util -= 10
                            else:
                                util += 10
                    else:
                        if self.board.board[i][j].get_color(k) == "O":
                            util -= 4
                        else:
                            util += 4
    possibleMovesO = self.find_all_possible_moves()
    self.play_turn = "X"
    possibleMovesX = self.find_all_possible_moves()
    util += (len(possibleMovesX) - len(possibleMovesO)) * 6
    self.play_turn = "O"

    if self.playerX.score == 1:
        util += 50
    if self.playerX.score == 2:
        util += 1000

    if self.playerO.score == 1:
        util -= 50
    if self.playerO.score == 2:
        util -= 1000

    return util
```

```
def utility_maximize_player(self):
    util = 0

    for i in range(self.board.dim):
        for j in range(self.board.dim):
            if not self.board.is_tile_white(i, j):
                stack_height = len(self.board.board[i][j].colors)

                for k in range(0, stack_height):
                    if k == 0:
                        if self.board.board[i][j].get_color(0) == "X":
                            util += 5
                        else:
                            util -= 5
                    elif k == 7:
                        if self.board.board[i][j].get_color(k) == "X":
                            util += 100
                        else:
                            util -= 100
                    elif k == stack_height - 1:
                        if k < 4:
                            if self.board.board[i][j].get_color(k) == "X":
                                util += 4
                            else:
                                util -= 4
                        else:
                            if self.board.board[i][j].get_color(k) == "X":
                                util += 6
                            else:
                                util -= 6
                    else:
                        if self.board.board[i][j].get_color(k) == "X":
                            util += 4
                        else:
                            util -= 4
    possibleMovesX = self.find_all_possible_moves()
    self.play_turn = "O"
    possibleMovesO = self.find_all_possible_moves()
    util += (len(possibleMovesX) - len(possibleMovesO))//2
    self.play_turn = "X"

    if self.playerX.score == 1:
        util += 120
    if self.playerX.score == 2:
        util += 1000

    if self.playerO.score == 1:
        util -= 120
    if self.playerO.score == 2:
        util -= 1000

    return util
```