

ATOLL PROJEKAT

TREĆA FAZA

STEFAN CVETKOVIC 19461 | LAZAR STANKOVIĆ 19375

Contents

FUNKCIJE IGRE	2
options.py	2
board.py	2
startPage()	2
isGameFinished(board, curPlayer)	3
confirmClick()	4
on_tile_click(r, c)	5
botMove()	5
botController.py	6
Globalne promenljive	6
performBotMove(board, botPlayer, islands, isEndFunc)	6
minimax(board, xPlayer, oPlayer, islands, isEndFunc, alpha, beta)	6
maxValue(state, depth, alpha, beta, xPlayer, oPlayer, islands, isEndFunc, turn)	7
minValue(state, depth, alpha, beta, xPlayer, oPlayer, islands, isEndFunc, turn)	8
availableMoves(board)	9
eval(player, state, islands, isEndFunc)	9

FUNKCIJE IGRE

options.py

```
if(notNum or n < 3 or n > 9 or n % 2 == 0):
    infoBox = customtkinter.CTkToplevel()
    infoBox.geometry('300x100')
    infoBox.grid_rowconfigure(0, weight=1)
    infoBox.grid_columnconfigure(0, weight=1)
    infoBox.title('Notification')

    label = customtkinter.CTkLabel(infoBox, text='N mora biti neparan broj
izmedju 3 i 9')

    label.grid(row = 0, column = 0, sticky='nsew')
    infoBox.grab_set()
    return
```

Izbačena zabrana igranja protiv računara.

board.py

```
from botController import performBotMove
```

Kreiran novi fajl koji sadrži funkcije vezane za igranje poteza robota. Na početku fajla učitava funkciju koju koristi za određivanje sledećeg poteza.

startPage()

```
if getCurrentPlayerName() == 'Bot':
    botMove()
```

Na kraju funkcije **startPage** dodata provera prvog igrača, ukoliko je prvi igrač računar poziva funkciju za izvršenje njegovog poteza.

isGameFinished(board, curPlayer)

```
def isGameFinished(board, curPlayer = None):
    processedGroups = set()

    if curPlayer is None:
        curPlayer = players[currentPlayer]

    for group in groups:
        ni, nj = groups[group][0][0]
        player = board[ni][nj]
        if player != curPlayer or group in processedGroups:
            continue

        boardCopy = [row[:] for row in board]
        dfs(boardCopy, ni, nj, player, eTile.Connected.value[0])
        processedGroups.add(group)

        connectedGroups = getConnectedGroups(group, boardCopy)
        lowest = iterateThroughGroups(connectedGroups, boardCopy)

        for conGroup in connectedGroups:
            processedGroups.add(conGroup)

        if lowest >= len(groups) / 2 + 1:
            return (True, boardCopy)

    return (False, [])
```

Dodatak parametara kako bi se ova funkcija ponovo koristila prilikom provere kraja igre kod MiniMax algoritma. Dodati parametri **board** i **curPlayer** koji će se koristiti umesto globalnih parametara koji su bili korišćeni pre njih. Ukoliko se ne pošalje parametar curPlayer program automatski postavlja trenutnog igrača.

confirmClick()

```

def confirmClick():
    global currentPlayerLabel, lastMove, currentPlayer, n
    print(f"{getCurrentPlayerName()} je odigrao: {chr(ord('A') - 1 +
lastMove[1])} {(lastMove[0] + lastMove[1] - n + 1)//2}\n")
    status, board = isGameFinished(matrix)

    confirmButton.configure(state = 'disabled')
    undoButton.configure(state = 'disabled')

    if status == True or isGameDraw():
        for i in range(len(board)):
            for j in range(len(board[0])):
                if board[i][j] == eTile.Connected.value[0]:
                    buttons[i][j].configure(fg_color = 'blue')
                if buttons[i][j] is not None:
                    buttons[i][j].configure(state = 'disabled')
                    if status != True:
                        buttons[i][j].configure(fg_color = 'yellow')

        print(f'Pobeda igraca: {getCurrentPlayerName()}' if status == True
else 'Igra je neresena')
        if status == True:
            currentPlayerLabel.configure(text = f'Pobeda igraca:
{getCurrentPlayerName()}', text_color= 'blue')
        else:
            currentPlayerLabel.configure(text = 'Igra je neresena',
text_color= 'yellow')
        else:
            currentPlayer = (currentPlayer + 1) % 2
            lastMove = None
            currentPlayerLabel.configure(text = getCurrentPlayer(), text_color=
playerColors[currentPlayer])
            if getCurrentPlayerName() == 'Bot':
                botMove()

```

Dodat je parametar **matrix** pri pozivu funkcije **isGameFinished** i na kraju funkcije proverava da li je sledeći igrač računar, ukoliko jeste poziva funkciju koja izvršava njegov potez.

on_tile_click(r, c)

```
if hideButtons or getCurrentPlayerName() == 'Bot':  
    confirmClick()
```

Na kraju funkcije se izvršava provera da li je igrač koji je odigrao računar, ukoliko jeste automatski potvrđuje njegov potez.

botMove()

```
def botMove():  
    for i in range(len(buttons)):  
        for j in range(len(buttons[i])):  
            if matrix[i][j] == eTile.Playable.value[0]:  
                buttons[i][j].configure(state = 'disabled')  
  
    row, col = performBotMove(matrix, players[currentPlayer], groups,  
isGameFinished)  
    on_tile_click(row, col)  
  
    for i in range(len(buttons)):  
        for j in range(len(buttons[i])):  
            if matrix[i][j] == eTile.Playable.value[0]:  
                buttons[i][j].configure(state = 'normal')
```

Funkcija koja se poziva za izvršenje poteza računara. Pre samog igranja računara postavlja svako polje na tabli na disabled. Nakon toga poziva funkciju iz novog fajla koja vraća odigrano polje. Nakon dobijta odgovara vraća polja na enabled (normal) stanje.

botController.py

Globalne promenljive

```
gameEndingMoves = {}
hexDirections = [(-2, 0), (-1, 1), (1, 1), (2, 0), (1, -1), (-1, -1)]
queue = deque()
visited = set()
```

gameEndingMoves se koristi u funkcijama max i min i on pamti kod kojih čvorova je došlo do kraja igre, ovo se radi kako bi pri sledećem prolazu prvo prošao kod tog polja kako bi se proverilo da li je rešena pobeda protivnika ili ne. hexDirections pamti načine za kretanje kroz matricu polja i queue i visited se koriste za evaluaciju.

performBotMove(board, botPlayer, islands, isEndFunc)

```
def performBotMove(board, botPlayer, islands, isEndFunc):
    boardCopy = [row[:] for row in board]
    humanPlayer = eTile.Player1.value[0] if botPlayer !=
eTile.Player1.value[0] else eTile.Player2.value[0]
    turn, _ = minimax(boardCopy, botPlayer, humanPlayer, islands, isEndFunc)
    return turn
```

Funkcija koja se poziva iz fajla **board.py**. Kao argumente prihvata izgled table, igrača koji je računar, početna ostva i funkciju kojom se proverava kraj igre. Ova funkcija se šalje kao parametar kako bi se izbeglo kružno importovanje. Pre samog određivanja poteza pravi kopiju table i odredi koja vrednost u enumu je čovek. Nakon toga poziva funkciju **minimax**.

minimax(board, xPlayer, oPlayer, islands, isEndFunc, alpha, beta)

```
def minimax(board, xPlayer, oPlayer, islands, isEndFunc, alpha=(None, -10000),
beta=(None, 10000)):
    gameEndingMoves.clear()
    return maxValue(board, 3, alpha, beta, xPlayer, oPlayer, islands,
isEndFunc)
```

Ova funkcija nam služi kao *shell* koji će pozivati samu funkciju za max. Nema potrebe da se u jednoj funkciji gleda da li je trenutni igrač računar zato što se ova funkcija koristi samo za potez robota. Pre poziva funkcije **maxValue** briše poteze iz prethodnih izvršenja. Kao parametre prima izgled table, igrača x (robota), igrača o (čoveka), listu ostva, funkciju koja se poziva za proveru kraja i alpha i beta.

maxValue(state, depth, alpha, beta, xPlayer, oPlayer, islands, isEndFunc, turn)

```
def maxValue(state, depth, alpha, beta, xPlayer, oPlayer, islands, isEndFunc,
turn = None):
    iE, _ = isEndFunc(state, xPlayer)
    if iE:
        return (turn, 10001)

    moves = list(availableMoves(state))

    if depth in gameEndingMoves:
        gameEndingMove = gameEndingMoves[depth]
        if gameEndingMove in moves:
            moves.remove(gameEndingMove)
            moves.insert(0, gameEndingMove)

    if depth == 0:
        return(turn, eval(xPlayer, state, islands, isEndFunc))

    for m in moves:
        row, col = m
        state[row][col] = xPlayer
        alpha = max(alpha, minValue(state, depth - 1, alpha, beta, xPlayer,
oPlayer, islands, isEndFunc, m if turn is None else turn), key = lambda x:
x[1])
        state[row][col] = eTile.Playable.value[0]
        if alpha[1] >= beta[1]:
            gameEndingMoves[depth] = m

            if depth == 3 and alpha[0] == None:
                alpha = (m, alpha[1])

            return alpha

    if depth == 3 and alpha[0] == None:
        alpha = (moves[0], alpha[1])

    return alpha
```

Funkcija max na samom početku proverava da li je došlo do kraja igre, ukoliko jeste vraća se automatski prethodnom pozivu iz steka, to može biti ili minValue ili minimax algoritam. Ukoliko nije došlo do kraja dobija listu mogućih poteza pomoću funkcije **availableMoves**. Ukoliko je neki prolazak na ovaj dubini došao do kraja igre uzimamo polje kod kojeg je došlo do kraja kako bismo prvo prošli kroz njega. Ovo radimo kako bismo izbedli prolazak kroz celu tablu ako je dođe do kraja na kraju liste. Ukoliko dođemo

do dubine 0 vraćamo samo evaluaciju table. Ukoliko nismo došli do kraja dubine onda prolazimo kroz sva moguća polja. Upišemo na tabli na određenom polju trenutnog igrača i poziva funkciju min. Tablu vraćamo na početno stanje nakon završetka funkcije min. Nakon konačnog odgovora vršimo sigurnosnu proveru da li je na kraju poteza alpha ostala None, ukoliko jeste postavljamo trenutni ili prvi potez kao odgovor, zavisno od toga da li izlazimo iz funkcije u toku petlje ili na kraju same funkcije.

`minValue(state, depth, alpha, beta, xPlayer, oPlayer, islands, isEndFunc, turn)`

```
def minValue(state, depth, alpha, beta, xPlayer, oPlayer, islands, isEndFunc,
turn = None):
    iE, _ = isEndFunc(state, oPlayer)
    if iE:
        return (turn, -10001)

    moves = list(availableMoves(state))

    if depth in gameEndingMoves:
        gameEndingMove = gameEndingMoves[depth]
        if gameEndingMove in moves:
            moves.remove(gameEndingMove)
            moves.insert(0, gameEndingMove)

    if depth == 0:
        return(turn, eval(oPlayer, state, islands, isEndFunc))

    for m in moves:
        row, col = m
        state[row][col] = oPlayer
        beta = min(beta, maxValue(state, depth - 1, alpha, beta, xPlayer,
oPlayer, islands, isEndFunc, m if turn is None else turn), key = lambda x:
x[1])
        state[row][col] = eTile.Playable.value[0]
        if beta[1] <= alpha[1]:
            gameEndingMoves[depth] = m
            return beta

    return beta
```

Funkcija min ima skoro pa istu logiku kao i funkcija max. Jedina razlika je što ne vršimo proveru da li je alpha None jer nema potrebe za tim. Min funkcija, naravno, poziva max funkciju i tako se međusobno pozivaju dok se ne dođe do pobede ili kraja dubine.

availableMoves(board)

```
def availableMoves(board):
    for row in range(len(board)):
        for col in range(len(board[row])):
            if board[row][col] == eTile.Playable.value[0]:
                yield (row, col)
```

Funkcija koja nam vraća sva Playable polja na tabli.

eval(player, state, islands, isEndFunc)

```
def eval(player, state, islands, isEndFunc):
    score = 0
    boardN = len(state)
    boardM = len(state[0])

    for island in islands:
        islandR, islandC = islands[island][0][0]
        owner = islands[island][1]
        targetR, targetC = boardN - islandR, boardM - islandC

        visited.clear()
        queue.clear()
        queue.append((islandR, islandC))
        visited.add((islandR, islandC))

        while queue:
            r, c = queue.popleft()
            for dr, dc in hexDirections:
                nr, nc = r + dr, c + dc
                if 0 <= nr < boardN and 0 <= nc < boardM:
                    if (nr, nc) not in visited and state[nr][nc] == owner:
                        visited.add((nr, nc))
                        queue.append((nr, nc))

        bestDistance = float("inf")
        for r, c in visited:
            dist = abs(r - targetR) + abs(c - targetC)
            if dist < bestDistance:
                bestDistance = dist

        if owner == player:
            score += bestDistance
        else:
            score -= bestDistance

    return score
```

Funkcija koja vrši evaluaciju table trenutnog stanja. Prolazimo kroz sva ostvra na tabli. Krećemo sa prvog kamenčića prvog igrača i idemo dok ne naiđemo na poslednji. Kada naiđemo na poslednji prolazimo kroz listu posećenih ostvra i gledamo kolika nam je distanca do krajnjeg. Nakon određivanja distance nju dodajemo ili oduzimamo zavisno od toga da li je ostvro igrača ili računara.