BURSA ULUDAĞ ÜNİVERSİTESİ

BİLGİSAYAR MÜHENDİSLİĞİ

2024-2025 EĞİTİM ÖĞRETİM YILI GÜZ DÖNEMİ

YAPAY ZEKA RAPORU

KLASIK ÇİN DAMASI OYUNUNU MINI-MAX ALGORİTMASI İLE OYNAYAN BIR AJAN

032290004 BARIŞ IŞIK

032290004@ogr.uludag.edu.tr

032290008 MURAT BERK YETİŞTİRİR

032290008@ogr.uludag.edu.tr

032290037 BUĞRA ÖZGEN

032290037@ogr.uludag.edu.tr

SORU 1: Çin daması oynayan ajan için mini-max algoritması ve alfa-beta budaması tekniklerini kullanarak ajanı nasıl tasarladığınızdan kısaca bahsediniz.

CEVAP 1: Ajan, Min-Max algoritmasını kullanarak her iki oyuncunun da en iyi hamleyi yapmasını sağlar. Min-Max, her hamleyi değerlendirip en iyi hamleyi seçerken, Alfa-Beta budaması gereksiz hesaplamaları keserek hız kazandırır. Beyaz oyuncu (W) maksimum, siyah oyuncu (B) ise minimum değerleri hedefler. Algoritma, oyun derinliği boyunca her iki oyuncunun hamlelerini değerlendirir, bu süreç, ajan'ların oyun sırasında optimal kararlar almasını sağlar.

SORU 2: Problemin çözümü için kodladığınız durum, hareket, algı, hedef fonksiyonu, ardıl fonksiyonu ve sezgi fonksiyonu tanımlamalarını yapınız.

CEVAP 2:

- **Algı fonksiyonu** evaluate_board fonksiyonudur. Bu fonksiyon, MinMax algoritmasında kullanılacak bir **algı fonksiyonu** sağlar, çünkü her bir oyun durumu için (yani her bir board durumu) bu fonksiyon bir değer döndürür. Bu değer, oyunun ne kadar "iyi" veya "kötü" olduğunu, hangi oyuncunun avantajda olduğunu belirtir.
- **Ardıl fonksiyonu** possible_boards fonksiyonudur. Bu fonksiyon, her iki oyuncu için geçerli tüm taş hareketlerini hesaplar ve her bir hareket sonucu oluşan yeni tahtaları döndürür. Bu fonksiyon, özellikle Minimax gibi oyun oynama algoritmalarında, her bir hamle için olasılıkları belirlemek için kullanılır.
- **Hakeret Fonksiyonu** get_movable fonksiyonudur. Bu fonksiyonu, taşın o anki pozisyonuna göre hangi hücrelere hareket edebileceğini belirler ve bu hareketlerin her birini birer yeni tahta (kopya tahtalar) olarak döndürür. Bu sayede ajanlar, her olası hamleyi görebilir ve en uygun olanını seçebilir.
- **Hedef Fonksiyonu** check_win fonksiyonudur. Bu fonksiyon, oyunun bitip bitmediğini kontrol etmek için kullanılır. Beyaz veya siyah ajan belirli bir bölgedeki tüm taşlarını yerleştirmesi durumunda (yani 9 taş) oyun sona erer ve kazanan ajan döndürülür. Eğer hiçbiri 9 taş yerleştirmemişse, oyun devam eder.
- **Sezgi fonksiyonu** euclidian_distance fonksiyonudur. Bu fonksiyon, her iki tarafın (beyaz ve siyah) hedeflerinden (yani oyunun sonunda ulaşması gereken karelerden) ne kadar uzak olduğunu hesaplar.
- **Durum fonksiyonu** get_average_distances fonksiyonudur. Bu fonksiyon bir dama tahtasında beyaz ve siyah taşlarının mevcut konumlarının ortalamalarını hesaplar ve her iki ajanın taşlarının hedef konumlarına olan mesafelerini döndürür. Fonksiyon, tahtadaki her bir taşın konumunu toplayarak ortalama konumları bulur, ardından her ajanın taşlarının hedefe olan uzaklıklarını hesaplayarak bu değerleri döndürür. Bu mesafeler, oyunun ilerleyişine göre ajanların hedeflerine ne kadar yaklaştığını ölçmek için kullanılabilir.

SORU 3: Mini-max algoritması ile farklı katlarda (2-kat, 3-kat, …) ileriye bakabilen iki ajan tasarlayınız. Ajanlar birbirlerine karşı 5 kez oynatıldığında hangi ajanın hangi skorla kazanmış olduğunu belirleyiniz. Bir oyunun tamamlanması için geçen ortalama süreyi belirleyiniz. Her ajanın bir hamle gerçekleştirilmeden evvel harcadığı ortalama süreyi belirleyiniz.

CEVAP 3: Ajanlar nasıl tasarlanırlarsa tasarlansın maçlar berabere bitiyor. Bu yüzden hiçbir şekilde bir ajan diğerine üstünlük sağlayamıyor. Genellikle maçlar sonsuza kadar sürüyor hame tekrarından berabere kalıyor.

SORU 4: Mini-max algoritmasına alfa-beta budaması dahil edildiğinde belirlenen ortalama sürelerde bir değişme olup olmadığını açıklayınız.

CEVAP 4: Alfa-beta budaması uygulamamıza dahil edilince uygulamanın çalışma hızında gözle görünür bir azalma söz konusu oldu. Budama hem kodun çalışma hızını hem de ajanların davranışları olumlu yönde etkiledi.

SORU 5: Yazdığınız kodu IDE'de göründüğü kalitede kopyalayınız.

CEVAP 5:

```
import math

import copy

import sys

NO_PAWN_SELECTED = (-1, -1)

# FLOAT

MIN = -1024

MAX = 1024

NAN = float('nan')


# DIRECTIONS

UP = (-1,0)

DOWN = (1,0)

LEFT = (0,-1)

RIGHT = (0,1)

W_DIR = (UP, LEFT)

B_DIR = (DOWN, RIGHT)

ALL = (UP,DOWN,LEFT,RIGHT)

NONE = ()


# PAWNS

EMPTY = '.'

TRACE = '#' # to disable promoteds from moving same place all the time

WHITE = 'W'

WHITE_CURRENT = 'V'

WHITE_PROMOTED = 'w'

WHITE_CURRENT_PROMOTED = 'v'

BLACK = 'B'
```

```
BLACK_PROMOTED = 'b'
BLACK_CURRENT = 'P'
BLACK_CURRENT_PROMOTED = 'p'
WHITES = (WHITE,WHITE_CURRENT,WHITE_PROMOTED,WHITE_CURRENT_PROMOTED)
BLACKS = (BLACK,BLACK_CURRENT,BLACK_PROMOTED,BLACK_CURRENT_PROMOTED)

#PAWN DIRS
PAWNS = (EMPTY,) + WHITES + BLACKS
DIRS = (NONE,) + 2 * (W_DIR,) + 2 * (ALL,) + 2 * (B_DIR,) + 2 * (ALL,)
DIRS_TEST = ("NONE",) + 2 * ("W_DIR",) + 2 * ("ALL",) + 2 * ("B_DIR",) + 2 * ("ALL",)

class Game:
    def __init__(self):
        self.board = self.create_board()
        self.current_ai = 'W'
        self.done = False
        self.player = []

    def create_board(self):
            board = [['.' for _ in range(8)] for _ in range(8)]

            for i in range(3):
                for j in range(3):
                    board[i][j] = 'B'
                    board[i + 5][j + 5] = 'W'
            return board

    def print_board(self):
        for row in self.board:
            print(" ".join(row))
        print()

    def add_player(self, player):
        if len(self.player) < 2:
```

```
            self.player.append(player)

    def check_win(self):
        white = 0
        black = 0
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 'W':
                    white += 1
                if self.board[i + 5][j + 5] == 'B':
                    black += 1

        if black == 9:
            return (True, 'B')
        if white == 9:
            return (True, 'W')
        return (False, '.')




    def play(self):
        while not self.check_win()[0]:
            pass

        print(self.check_win()[1] + " won the game")



class MinMaxAI:
    def __init__(self, side, ply, game):
        self.side = side
        self.ply = ply
        self.game = game
```

```
# returns "tuple (board, eval)" for now, or returns (board or eval) ?? not sure yet
# at the end we need to get the board that we want to play from it
# while iteration we need the eval values not the boards
# side is 'W' or 'B'
# it will get the first move not all moves with same values
def minmax( board, side, ply, alpha, beta, isMaxNode):
    boards = possible_boards(board, side)

    # handle ply 0 here and if its not ply 0 just continue down
    if ply == 0:
        # we need to assign a value to the board sent end pass that up right ??
        return (copy.deepcopy(board), evaluate_board(board))

    #print("MINMAX STARTING ON SIDE", side, "PLY", ply, isMaxNode ,"WITH BOARD")
    #print_b(board)
    #print("ALPHA", alpha, "BETA", beta)

    # if side is 'W' max is +
    # if side is 'B' max is -
    # then it is max node if 'W' else min
    # print() if isMaxNode is not side is 'W' else print()
    isMaxNode = True if side == 'W' else False
    next_moves = [] # we can put return values from minmaxes here

    isRoot = True if alpha is NAN and beta is NAN else False

    alpha = MIN if alpha is NAN else alpha
    beta = MAX if beta is NAN else beta
    #print("ALPHA", alpha, "BETA", beta)

    # do minmax for all boards
    i = 0
    for this_board in boards:
```

```
next_moves.append(minmax(this_board,
                         'B' if side == 'W' else 'W',  # switch sides
                         ply - 1,
                         NAN if not isMaxNode else alpha,
                         NAN if isMaxNode else beta,
                         not isMaxNode))

just_added_value = next_moves[i][1]
# assuming we are getting (board, float) tuple

# to cut off we need the pop the board we just added ?
if isMaxNode:
    alpha = just_added_value if just_added_value > alpha else alpha
    #print(alpha)
    #print_b(next_moves[i][0])
    if beta <= alpha:
        #print("BETA CUTOFF")
        break
        next_moves.pop()
        i -= 1
        # beta cutoff
        pass
else:
    beta = just_added_value if just_added_value < beta else beta
    #print(alpha)
    if beta <= alpha:
        #print("ALPHA CUTOFF")
        #print(alpha,beta,just_added_value)
        break
        next_moves.pop()
        i -= 1
        # alpha cutoff
        pass
i += 1
```

```
    # all moves added to next moves now we can send the best one above
    # send the first max/min value from next_moves
    if isRoot:
        #print("IS ROOT")
        a = max(next_moves, key=lambda x: x[1])
        #print_b(boards[next_moves.index(a)])
        return(boards[next_moves.index(a)], a[1])
        return a
    if isMaxNode:
        return max(next_moves, key=lambda x: x[1])
    else:
        return min(next_moves, key=lambda x: x[1])

#   eval = b - w
#   + means white is winning
#   - means black is winning
def evaluate_board(board):
    (w_distance, b_distance) = get_average_distances(board)

    return b_distance - w_distance

# returns list of boards
# W V can move up and left
# B P can move down and right
# w v and b p can move to all
def possible_boards(board, side):
    boards = [] # possible moves
    # a tuple of a board and current piece that moved (board, (i, j))
    board_queue = [(board, NO_PAWN_SELECTED, False)] # init iterative moves = [current board]
    # iterative moves : queue to iterate to find all possible jump moves
    counter = 0
    while len(board_queue) != 0: # while iterative moves not empty
```

```
#print("in while")
#print_b(board_queue[len(board_queue) - 1][0])
(current_board, current_pos, _) = board_queue.pop() # pop first board
#print("popped")
#print_b(current_board)
pawn_selected = False

if current_pos is  NO_PAWN_SELECTED:
   pass
else:
   pawn_selected = True
   (i_cur,j_cur) = current_pos
   current_pawn = current_board[i_cur][j_cur]

# if there is a selected pawn iterate through that
(range_i,range_j) = ((i_cur,), (j_cur,)) if pawn_selected else (range(8),range(8))
# if ther is not a selected pawn iterate through all
#print(range_i,range_j)
movables = []
# iterate through the ranges
for i in range_i:
   for j in range_j:
      current_cell = current_board[i][j]
      if current_cell is EMPTY:
         continue
      elif current_cell in WHITES and side in WHITES or current_cell in BLACKS and side in BLACKS:
         # current_cell and side is same so we can get moves
         movables_append = get_movable(current_board,(i,j))
         for movable in movables_append:
            if counter == 1000:
               #print("noluo")
               pass
            if counter > 1000:
               pass
```

```
                    break
                movables.append(movable)
                #print_b(movable[0])
                #print(movable[1])
                counter = counter + 1
                #print("COUNTER",counter)

            pass

    # we got movables now we cana add them to board_queue and boards
    for movable in movables:
        if movable[2]: # if jumped append
            #print("appended")
            #print_b(movable[0])
            #print("appended")
            #print(len(board_queue))
            board_queue.append(movable)
            #print(len(board_queue))

        copied_board = copy.deepcopy(movable[0])
        copied_board[movable[1][0]][movable[1][1]] =
selected_to_pawn(copied_board[movable[1][0]][movable[1][1]])
        promote_if_possible(copied_board,movable[1])
        remove_trace(copied_board)
        boards.append(copied_board)
        #print_b(copied_board)
        pass
  return boards


# a movable is
# (board, (i,j), isJumped)
def get_movable(board, position):
  boards = []
```

```
    (i,j) = position
    current_pawn = board[i][j]
    #print(current_pawn)
    #print("IS SELECTED ",is_selected(current_pawn))
    #if is_selected(current_pawn):
        #print("############################")
    current_pawn_selected = pawn_to_selected(current_pawn)
    #print(current_pawn, current_pawn_selected)
    dirs_to_check = pawn_to_movable_dirs(current_pawn)

    for dir in dirs_to_check:
        # first dir in bounds
        i_first = i + dir[0]
        j_first = j + dir[1]
        i_second = i + dir[0] * 2
        j_second = j + dir[1] * 2

        if in_bounds(i_first, j_first):
            if board[i_first][j_first] is EMPTY and not is_selected(current_pawn):
                copied_board = copy.deepcopy(board)
                copied_board[i][j] = EMPTY
                copied_board[i_first][j_first] = current_pawn_selected
                #promote_if_possible(copied_board,(i_first,j_first))
                #print_b(copied_board)
                #print("MOVED",current_pawn,i,j, "TO", i_first, j_first,"\n")
                boards.append((copied_board,(i_first, j_first), False))
                pass
            elif in_bounds(i_second, j_second) and board[i_first][j_first] in PAWNS[1:] and
board[i_second][j_second] is EMPTY:
                #print(i,j,i_first,j_first,i_second, j_second)
                #print(board[i_first][j_first])
                copied_board = copy.deepcopy(board)
                if current_pawn in
(WHITE_PROMOTED,WHITE_CURRENT_PROMOTED,BLACK_PROMOTED,BLACK_CURRENT_PROMOTED):
```

```
                copied_board[i][j] = TRACE
            else:
                copied_board[i][j] = EMPTY
            copied_board[i_second][j_second] = current_pawn_selected
            #promote_if_possible(copied_board,(i_second,j_second))
            #print_b(copied_board)
            #print("JUMPED",i,j, "TO", i_second, j_second,"\n")
            boards.append((copied_board,(i_second, j_second), True))
            pass
        pass


    return boards

def get_average_distances(board):
    w_target_average = [1,1]
    b_target_average = [6,6]

    w_current_average = [0,0]
    b_current_average = [0,0]

    for i in range(8):
        for j in range(8):
            if board[i][j] == '.':
                pass
            else:
                if board[i][j] == 'W':
                    w_current_average[0] += i
                    w_current_average[1] += j
                else:
                    b_current_average[0] += i
                    b_current_average[1] += j
```

```
    w_current_average[0] = w_current_average[0] / 9
    w_current_average[1] = w_current_average[1] / 9
    b_current_average[0] = b_current_average[0] / 9
    b_current_average[1] = b_current_average[1] / 9

    w_goal = euclidian_distance(w_target_average,w_current_average)
    b_goal = euclidian_distance(b_target_average,b_current_average)

    return (w_goal, b_goal)

def euclidian_distance(start,end):
    squared_diff = (start[0] - end[0]) ** 2 + (start[1] - end[1]) ** 2

    return math.sqrt(squared_diff)

def pawn_to_selected(pawn):
    index = PAWNS.index(pawn)
    return PAWNS[index + 1 if index % 2 == 1 else index]

def selected_to_pawn(pawn):
    index = PAWNS.index(pawn)
    return PAWNS[index - 1 if index % 2 == 0 else index]

def pawn_to_promoted(pawn):
    index = PAWNS.index(pawn)

def is_selected(pawn):
    index = PAWNS.index(pawn)
    return index % 2 == 0

def pawn_to_movable_dirs(pawn):
    return DIRS[PAWNS.index(pawn)]
def in_bounds(i,j):
    if i > -1 and i < 8:
```

```
        if j > -1 and j < 8:
            return True
    return False
def print_b(board):
        for row in board:
            print(" ".join(row))
        print()
def promote_if_possible(board, pos):
    current_pawn = board[pos[0]][pos[1]]
    if current_pawn in WHITES and pos[0] < 3 and pos[1] < 3:
        next_index = WHITES.index(current_pawn) + 2
        board[pos[0]][pos[1]] = WHITES[ next_index if next_index < 4 else next_index - 2]
    if current_pawn in BLACKS and pos[0] > 4 and pos[1] > 4:
        next_index = BLACKS.index(current_pawn) + 2
        #print(next_index)
        board[pos[0]][pos[1]] = BLACKS[next_index if next_index < 4 else next_index - 2]
def remove_trace(board):
    for i in range(8):
        for j in range(8):
            board[i][j] = EMPTY if board[i][j] == TRACE else board[i][j]

if __name__ == "__main__":
    game = Game()
    while game.check_win()[1]:
    #for i in range(20):
        print("WHITES MOVE")
        (a,b) = minmax(game.board,WHITE,3,NAN,NAN,True)
        game.board = copy.deepcopy(a)
        print_b(a)
        print(evaluate_board(a))
        print("BLACKS MOVE")
        (a,b) = minmax(game.board,BLACK,3,NAN,NAN,True)
        game.board = copy.deepcopy(a)
        print_b(a)
```

```
print(evaluate_board(a))
```

SORU 6: Ajanların birbirlerine karşı oynadığı oyunlardaki farklı tahta durumlarında yaptığı hamlelere dair konsol veya grafik arayüz ekran çıktısını ekleyiniz.

CEVAP 6: