

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](mailto:032290008@ogr.uludag.edu.tr)

032290008 MURAT BERK YETİŞTİRİR

[032290008@ogr.uludag.edu.tr](mailto:032290008@ogr.uludag.edu.tr)

032290037 BUĞRA ÖZGEN

[032290037@ogr.uludag.edu.tr](mailto: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)

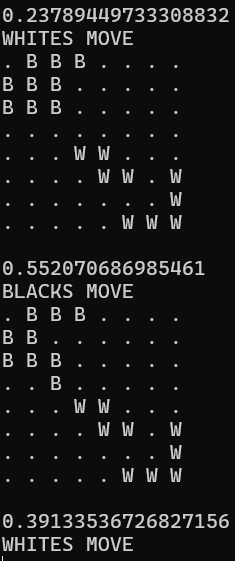
        game.board = copy.deepcopy(a)

        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:**

****