Yves Zakra

CSCI 440

Spring 2021

Breakthrough Game code

import pygame

import sys, os, math

from model import \*

from alpha\_beta\_agent import \*

import time

class BreakthroughGame:

    def \_\_init\_\_(self):

        pygame.init()

        self.width, self.height = 700, 560

        self.sizeofcell = int(560/8)

        self.screen = pygame.display.set\_mode((self.width, self.height))

        self.screen.fill([255, 255, 255])

        # chessboard and workers

        self.board = 0

        self.blackchess = 0

        self.whitechess = 0

        self.outline = 0

        self.reset = 0

        self.winner = 0

        self.computer = None

        # status 0: origin;  1: ready to move; 2: end

        # turn 1: black 2: white

        self.status = 0

        self.turn = 1

        # Variable for moving

        self.ori\_x = 0

        self.ori\_y = 0

        self.new\_x = 0

        self.new\_y = 0

        # matrix for position of chess, 0 - empty, 1 - black, 2 - white

        self.boardmatrix = [[1, 1, 1, 1, 1, 1, 1, 1],

                            [1, 1, 1, 1, 1, 1, 1, 1],

                            [0, 0, 0, 0, 0, 0, 0, 0],

                            [0, 0, 0, 0, 0, 0, 0, 0],

                            [0, 0, 0, 0, 0, 0, 0, 0],

                            [0, 0, 0, 0, 0, 0, 0, 0],

                            [2, 2, 2, 2, 2, 2, 2, 2],

                            [2, 2, 2, 2, 2, 2, 2, 2]]

        self.total\_nodes\_1 = 0

        self.total\_nodes\_2 = 0

        self.total\_time\_1 = 0

        self.total\_time\_2 = 0

        self.total\_step\_1 = 0

        self.total\_step\_2 = 0

        self.eat\_piece = 0

        # Caption

        pygame.display.set\_caption("Breakthrough")

        # initialize pygame clock

        self.clock = pygame.time.Clock()

        self.initgraphics()

    def run(self):

        self.clock.tick(60)

        # clear the screen

        self.screen.fill([255, 255, 255])

        if self.status == 5:

            # Black

            if self.turn == 1:

                start = time.process\_time()

                self.ai\_move(2, 2)

                self.total\_time\_1 += (time.process\_time() - start)

                self.total\_step\_1 += 1

                print('total\_step\_1 = ', self.total\_step\_1,

                      'total\_nodes\_1 = ', self.total\_nodes\_1,

                      'node\_per\_move\_1 = ', self.total\_nodes\_1 / self.total\_step\_1,

                      'time\_per\_move\_1 = ', self.total\_time\_1 / self.total\_step\_1,

                      'have\_eaten = ', self.eat\_piece)

            elif self.turn == 2:

                start = time.process\_time()

                self.ai\_move(2, 2)

                self.total\_time\_2 += (time.process\_time() - start)

                self.total\_step\_2 += 1

                print('total\_step\_2 = ', self.total\_step\_2,

                      'total\_nodes\_2 = ', self.total\_nodes\_2,

                      'node\_per\_move\_2 = ', self.total\_nodes\_2 / self.total\_step\_2,

                      'time\_per\_move\_2 = ', self.total\_time\_2 / self.total\_step\_2,

                      'have\_eaten: ', self.eat\_piece)

        # Events accepting

        for event in pygame.event.get():

            # Quit if close the windows

            if event.type == pygame.QUIT:

                exit()

            # reset button pressed

            elif event.type == pygame.MOUSEBUTTONDOWN and self.isreset(event.pos):

                self.boardmatrix = [[1, 1, 1, 1, 1, 1, 1, 1],

                            [1, 1, 1, 1, 1, 1, 1, 1],

                            [0, 0, 0, 0, 0, 0, 0, 0],

                            [0, 0, 0, 0, 0, 0, 0, 0],

                            [0, 0, 0, 0, 0, 0, 0, 0],

                            [0, 0, 0, 0, 0, 0, 0, 0],

                            [2, 2, 2, 2, 2, 2, 2, 2],

                            [2, 2, 2, 2, 2, 2, 2, 2]]

                self.turn = 1

                self.status = 0

            # computer button pressed

            elif event.type == pygame.MOUSEBUTTONDOWN and self.iscomputer(event.pos):

                self.ai\_move\_alphabeta(1)

                #self.ai\_move\_minimax(1)

            elif event.type == pygame.MOUSEBUTTONDOWN and self.isauto(event.pos):

                self.status = 5

            # ====================================================================================

            # select chess

            elif event.type == pygame.MOUSEBUTTONDOWN and self.status == 0:

                x, y = event.pos

                coor\_y = math.floor(x / self.sizeofcell)

                coor\_x = math.floor(y / self.sizeofcell)

                if self.boardmatrix[coor\_x][coor\_y] == self.turn:

                    self.status = 1

                    self.ori\_y = math.floor(x / self.sizeofcell)

                    self.ori\_x = math.floor(y / self.sizeofcell)

            # check whether the selected chess can move, otherwise select other chess

            elif event.type == pygame.MOUSEBUTTONDOWN and self.status == 1:

                x, y = event.pos

                self.new\_y = math.floor(x / self.sizeofcell)

                self.new\_x = math.floor(y / self.sizeofcell)

                if self.isabletomove():

                    self.movechess()

                    if (self.new\_x == 7 and self.boardmatrix[self.new\_x][self.new\_y] == 1) \

                        or (self.new\_x == 0 and self.boardmatrix[self.new\_x][self.new\_y] == 2):

                        self.status = 3

                elif self.boardmatrix[self.new\_x][self.new\_y] == self.boardmatrix[self.ori\_x][self.ori\_y]:

                    self.ori\_x = self.new\_x

                    self.ori\_y = self.new\_y

                    # display the board and chess

        self.display()

        # update the screen

        pygame.display.flip()

    # load the graphics and rescale them

    def initgraphics(self):

        self.board = pygame.image.load\_extended(os.path.join('src', 'chessboard.jpg'))

        self.board = pygame.transform.scale(self.board, (560, 560))

        self.blackchess = pygame.image.load\_extended(os.path.join('src', 'blackpiece.png'))

        self.blackchess = pygame.transform.scale(self.blackchess, (self.sizeofcell- 15, self.sizeofcell - 15))

        self.whitechess = pygame.image.load\_extended(os.path.join('src', 'redpiece.png'))

        self.whitechess = pygame.transform.scale(self.whitechess, (self.sizeofcell - 20, self.sizeofcell - 20))

        self.outline = pygame.image.load\_extended(os.path.join('src', 'square-outline.png'))

        self.outline = pygame.transform.scale(self.outline, (self.sizeofcell, self.sizeofcell))

        self.reset = pygame.image.load\_extended(os.path.join('src', 'reset.jpg'))

        self.reset = pygame.transform.scale(self.reset, (80, 80))

        self.winner = pygame.image.load\_extended(os.path.join('src', 'winner.png'))

        self.winner = pygame.transform.scale(self.winner, (250, 250))

        self.computer = pygame.image.load\_extended(os.path.join('src', 'computer.png'))

        self.computer = pygame.transform.scale(self.computer, (80, 80))

        self.auto = pygame.image.load\_extended(os.path.join('src', 'auto.png'))

        self.auto = pygame.transform.scale(self.auto, (80, 80))

    # display the graphics in the window

    def display(self):

        self.screen.blit(self.board, (0, 0))

        self.screen.blit(self.reset, (590, 50))

        self.screen.blit(self.computer, (590, 200))

        self.screen.blit(self.auto, (590, 340))

        for i in range(8):

            for j in range(8):

                if self.boardmatrix[i][j] == 1:

                    self.screen.blit(self.blackchess, (self.sizeofcell \* j + 10, self.sizeofcell \* i + 10))

                elif self.boardmatrix[i][j] == 2:

                    self.screen.blit(self.whitechess, (self.sizeofcell \* j + 10, self.sizeofcell \* i + 10))

        if self.status == 1:

            # only downward is acceptable

            if self.boardmatrix[self.ori\_x][self.ori\_y] == 1:

                x1 = self.ori\_x + 1

                y1 = self.ori\_y - 1

                x2 = self.ori\_x + 1

                y2 = self.ori\_y + 1

                x3 = self.ori\_x + 1

                y3 = self.ori\_y

                # left down

                if y1 >= 0 and self.boardmatrix[x1][y1] != 1:

                    self.screen.blit(self.outline,

                                     (self.sizeofcell \* y1, self.sizeofcell \* x1))

                # right down

                if y2 <= 7 and self.boardmatrix[x2][y2] != 1:

                    self.screen.blit(self.outline,

                                     (self.sizeofcell \* y2, self.sizeofcell \* x2))

                # down

                if x3 <= 7 and self.boardmatrix[x3][y3] == 0:

                    self.screen.blit(self.outline,

                                     (self.sizeofcell \* y3, self.sizeofcell \* x3))

            if self.boardmatrix[self.ori\_x][self.ori\_y] == 2:

                x1 = self.ori\_x - 1

                y1 = self.ori\_y - 1

                x2 = self.ori\_x - 1

                y2 = self.ori\_y + 1

                x3 = self.ori\_x - 1

                y3 = self.ori\_y

                # left up

                if y1 >= 0 and self.boardmatrix[x1][y1] != 2:

                    self.screen.blit(self.outline,

                                     (self.sizeofcell \* y1, self.sizeofcell \* x1))

                # right up

                if y2 <= 7 and self.boardmatrix[x2][y2] != 2:

                    self.screen.blit(self.outline,

                                     (self.sizeofcell \* y2, self.sizeofcell \* x2))

                # up

                if x3 >= 0 and self.boardmatrix[x3][y3] == 0:

                    self.screen.blit(self.outline,

                                     (self.sizeofcell \* y3, self.sizeofcell \* x3))

        if self.status == 3:

            self.screen.blit(self.winner, (100, 100))

    def movechess(self):

        self.boardmatrix[self.new\_x][self.new\_y] = self.boardmatrix[self.ori\_x][self.ori\_y]

        self.boardmatrix[self.ori\_x][self.ori\_y] = 0

        if self.turn == 1:

            self.turn = 2

            self.ai\_move\_alphabeta(1)

        # elif self.turn == 2:

        #     self.turn = 1

        self.status = 0

    def isreset(self, pos):

        x, y = pos

        if 670 >= x >= 590 and 50 <= y <= 130:

            return True

        return False

    def iscomputer(self, pos):

        x, y = pos

        if 590 <= x <= 670 and 200 <= y <= 280:

            return True

        return False

    def isauto(self, pos):

        x, y = pos

        if 590 <= x <= 670 and 340 <= y <= 420:

            return True

        return False

    def isabletomove(self):

        if (self.boardmatrix[self.ori\_x][self.ori\_y] == 1

            and self.boardmatrix[self.new\_x][self.new\_y] != 1

            and self.new\_x - self.ori\_x == 1

            and self.ori\_y - 1 <= self.new\_y <= self.ori\_y + 1

            and not (self.ori\_y == self.new\_y and self.boardmatrix[self.new\_x][self.new\_y] == 2)) \

            or (self.boardmatrix[self.ori\_x][self.ori\_y] == 2

                and self.boardmatrix[self.new\_x][self.new\_y] != 2

                and self.ori\_x - self.new\_x == 1

                and self.ori\_y - 1 <= self.new\_y <= self.ori\_y + 1

                and not (self.ori\_y == self.new\_y and self.boardmatrix[self.new\_x][self.new\_y] == 1)):

            return 1

        return 0

    def ai\_move(self, searchtype, evaluation):

        if searchtype == 1:

            return self.ai\_move\_minimax(evaluation)

        elif searchtype == 2:

            return self.ai\_move\_alphabeta(evaluation)

    def ai\_move\_minimax(self, function\_type):

        board, nodes, piece = MinimaxAgent(self.boardmatrix, self.turn, 3, function\_type).minimax\_decision()

        self.boardmatrix = board.getMatrix()

        if self.turn == 1:

            self.total\_nodes\_1 += nodes

            self.turn = 2

        elif self.turn == 2:

            self.total\_nodes\_2 += nodes

            self.turn = 1

        self.eat\_piece = 16 - piece

        if self.isgoalstate():

            self.status = 3

            #print(self.boardmatrix)

    def ai\_move\_alphabeta(self, function\_type):

        board, nodes, piece = AlphaBetaAgent(self.boardmatrix, self.turn, 5, function\_type).alpha\_beta\_decision()

        self.boardmatrix = board.getMatrix()

        if self.turn == 1:

            self.total\_nodes\_1 += nodes

            self.turn = 2

        elif self.turn == 2:

            self.total\_nodes\_2 += nodes

            self.turn = 1

        self.eat\_piece = 16 - piece

        if self.isgoalstate():

            self.status = 3

    def isgoalstate(self, base=0):

        if base == 0:

            if 2 in self.boardmatrix[0] or 1 in self.boardmatrix[7]:

                return True

            else:

                for line in self.boardmatrix:

                    if 1 in line or 2 in line:

                        return False

            return True

        else:

            count = 0

            for i in self.boardmatrix[0]:

                if i == 2:

                    count += 1

            if count == 3:

                return True

            count = 0

            for i in self.boardmatrix[7]:

                if i == 1:

                    count += 1

            if count == 3:

                return True

            count1 = 0

            count2 = 0

            for line in self.boardmatrix:

                for i in line:

                    if i == 1:

                        count1 += 1

                    elif i == 2:

                        count2 += 1

            if count1 <= 2 or count2 <= 2:

                return True

        return False

def main():

    game = BreakthroughGame()

    while 1:

        game.run()

if \_\_name\_\_ == '\_\_main\_\_':

    main()