**פרויקט סיום באלגוריתמים מתקדמים למערכות נבונות**

**לינקים**

סרט הדגמה של המשחק - <https://youtu.be/lvfdve0tTfU>

סרט הדגמה של המצגת - <https://youtu.be/pPr-KWPTbwo>

GitHub - <https://github.com/yuvch98/K.L.Y.R>

**סיכום הפרויקט**

תכננו ופתחנו משחק אסטרטגיה המתבסס על המשחק המוכר – אבן, נייר ומספריים. לכל אחד מהשחקנים יש מלך אחד שמטרת כל שחקן היא "לאכול" את הדגל של היריב. המחשב (היריב) בוחר בכל פעם את הצעד הבא למשחק בעזרת אלגוריתמים שונים שמימשנו, ובכך הוא מקבל את ההחלטה הטובה ביותר בהתאם לאלגוריתם הרלוונטי.

**הבעיה**

במשחק המקורי שהיווה לנו השראה, לא הייתה אופציה לשחק מול המחשב. בעת תכנון ויישום המשחק נתקלנו בקושי רב בלימוד המחשב לקבל את ההחלטה הטובה ביותר, כלומר כמובן שהקו המנחה לניצחון הוא אכילת הדגל של היריב, אך לא תמיד זהו הצעד החכם והנכון ביותר, ואף צעד בכיוון זה גם יכול להוות את הצעד הגרוע ביותר בהסתכלות כללית.

**הפתרון**

הגענו להבנה שהיצמדות למתודולוגיה אחת ויחידה היא בעייתית, לא גמישה ולא דינאמית. לכן הוספנו עוד היבטים של קבלת ההחלטות :

1) בדיקת כל שחקן בתור המחשב, ובעזרת ניקוד מקסימלי המחשב מנבא טוב יותר באיזה שחקן לבחור וגם לאיזו משבצת לזוז.

2) שיטת הניקוד שהוזכרה, מתחשבת גם במרחק החייל מהדגל, וגם בודקת האם לחייל הנבדק יש יכולת ניצחון בתזוזה לאחד התאים הסמוכים לו.

3) יכולת ניצחון נבדקת ע"פ 3 פרמטרים:

א. ניצחון

ב. תיקו

ג. הפסד

שחקן תמיד יעדיף לבחור בניצחון, במידה וישנו מצב של תיקו השחקן ללא מצב ניצחון השחקן יבחר להיכנס למצב תיקו. וכמובן נמנע מהפסדים ככל האפשר.

**הקוד**

**Computer.py**

from player import Player

class Computer(*Player*):

    def \_\_init\_\_(self, positions, items):

        super().\_\_init\_\_(positions, items)

        self.flag\_compromised = False

    def make\_best\_move(self, game\_logic):

        self.shuffle\_items\_check(game\_logic=game\_logic)

        best\_move = None

        best\_score = -float('inf')

        for position in self.positions:

            for move in game\_logic.get\_valid\_moves(position, self.positions, game\_logic.wall\_position):

                score = self.\_evaluate\_move(old\_pos=position, new\_pos=move, game\_logic=game\_logic)

                if score > best\_score:

                    best\_score = score

                    best\_move = (position, move)

        if best\_move:

            game\_logic.move\_and\_battle(self, game\_logic.player, best\_move[0], best\_move[1], is\_computer=True)

            game\_logic.check\_victory()

    def \_evaluate\_move(self, old\_pos, new\_pos, game\_logic):

        # Heuristic evaluation function for a move

        score = 0

        score += self.is\_winner(old\_pos, new\_pos, game\_logic)

        score += self.distance\_flag(new\_pos, game\_logic.player.flag\_pos, game\_logic)  # Add distance evaluation

        return score

    def distance\_flag(self, start\_pos, flag\_pos, game\_logic):  # uses IDA\*

        def heuristic(pos1, pos2):

            return abs(pos1[0] - pos2[0]) + abs(pos1[1] - pos2[1])

        def search(path, g, bound):  # Helper function for IDA\*

            node = path[-1]  # node = (0, 1)

            f = g + heuristic(node, flag\_pos)  # 0 + number

            if f > bound:  # f == bound

                return f

            if node == flag\_pos:

                return 'FOUND'

            min\_bound = float('inf')

            for neighbor in get\_neighbors(node):

                if neighbor not in path:  # valid moves = neighbor

                    path.append(neighbor)

                    t = search(path, g + 1, bound)

                    if t == 'FOUND':

                        return 'FOUND'

                    if t < min\_bound:

                        min\_bound = t

                    path.pop()

            return min\_bound

        def ida\_star(root):

            if root == self.flag\_pos:

                # do not move with it towards the flag

                return -10

            bound = heuristic(root, flag\_pos)

            path = [root]

            while True:

                t = search(path, 0, bound)

                if t == 'FOUND':

                    return len(path) - 1

                if t == float('inf'):

                    return float('inf')

                bound = t

        def get\_neighbors(pos):

            row, col = pos

            neighbors = [(row-1, col), (row+1, col), (row, col-1), (row, col+1)]

            valid\_neighbors = [

                move for move in neighbors

                if 0 <= move[0] < 6 and 0 <= move[1] < 7

                and move not in game\_logic.computer.positions

                and move != game\_logic.wall\_position

            ]

            return valid\_neighbors

        distance = ida\_star(start\_pos)

        return -distance

    def is\_winner(self, old\_pos, new\_pos, game\_logic):

        score = 0

        if new\_pos in game\_logic.player.positions:

            player\_item = game\_logic.player.items[new\_pos]

            computer\_item = game\_logic.computer.items[old\_pos]

            if player\_item == computer\_item:

                score += 0  # worth taking the chance...

                return score

            if self.win(computer\_item, player\_item):

                score += 20

            #if it's a loss for the computer

            else:

                score -= 20

        return score

    def win(self, computer\_item, opponent\_item):

        if (computer\_item == 'Rock' and opponent\_item == 'Scissors') or \

                (computer\_item == 'Paper' and opponent\_item == 'Rock') or \

                (computer\_item == 'Scissors' and opponent\_item == 'Paper'):

            return True

        if opponent\_item == 'Flag':

            return True

        else:

            return False

    def shuffle\_items\_check(self, game\_logic):

        if self.flag\_compromised:  # if the computer already did shuffle

            return

        for position in self.positions:

            if position == self.flag\_pos:

                for move in game\_logic.get\_valid\_moves(position, self.positions, game\_logic.wall\_position):

                    if move in game\_logic.player.positions:

                        self.flag\_compromised = True

        if self.flag\_compromised:

            self.shuffle\_items()

**constants.py**

# constants.py

SCREEN\_WIDTH, SCREEN\_HEIGHT = 1000, 600  # Increased width for the side column

CELL\_SIZE = SCREEN\_WIDTH // 10  # Adjust cell size if needed

RADIUS = CELL\_SIZE // 3

rock\_png = 'rock.png'

flag\_png = 'red-flag.png'

scissors\_png = 'scissors.png'

paper\_png = 'paper.png'

COLORS = {'Player': (0, 0, 255), 'Computer': (186, 0, 0), 'Empty': (255, 255, 255), 'Wall': (128, 128, 128), 'Aura': (255, 255, 255)}

BUTTON\_WIDTH = 100

BUTTON\_HEIGHT = 50

BUTTON\_COLOR = (255, 222, 173)

BUTTON\_HOVER\_COLOR = (235, 202, 153)

BUTTON\_TEXT\_COLOR = (0, 0, 0)

SIDE\_COLUMN\_WIDTH = 300

BUTTON\_COLOR\_AFTER\_CLICK = (128, 128, 128)

SIDE\_COLUMN\_COLOR = (255, 255, 240)

CELL\_EVEN\_COLOR = (179, 225, 172)

CELL\_ODD\_COLOR = (115, 195, 108)

**Game.py**

import pygame

import constants

pygame.init()

# Constants

SCREEN\_WIDTH, SCREEN\_HEIGHT = constants.SCREEN\_WIDTH, constants.SCREEN\_HEIGHT

CELL\_SIZE = constants.CELL\_SIZE

# Load images

rock\_img = pygame.image.load(constants.rock\_png)

paper\_img = pygame.image.load(constants.paper\_png)

scissors\_img = pygame.image.load(constants.scissors\_png)

flag\_img = pygame.image.load(constants.flag\_png)

images = {'Rock': rock\_img, 'Paper': paper\_img, 'Scissors': scissors\_img, 'Flag': flag\_img}

# Scale images to fit the cell size

rock\_img = pygame.transform.scale(rock\_img, (CELL\_SIZE, CELL\_SIZE))

paper\_img = pygame.transform.scale(paper\_img, (CELL\_SIZE, CELL\_SIZE))

scissors\_img = pygame.transform.scale(scissors\_img, (CELL\_SIZE, CELL\_SIZE))

flag\_img = pygame.transform.scale(flag\_img, (CELL\_SIZE, CELL\_SIZE))

# Create screen

screen = pygame.display.set\_mode((SCREEN\_WIDTH, SCREEN\_HEIGHT))

pygame.display.set\_caption('K.L.Y.R')

FONT = pygame.font.Font(None, 24)

class Game:

    def \_\_init\_\_(self, logic):

        self.logic = logic

        self.COLORS = constants.COLORS

        self.shuffle\_used = False

        self.player\_turn = True

    def draw\_board(self):

        screen.fill((255, 255, 255))

        for row in range(6):

            for col in range(7):

                rect = pygame.Rect(col \* CELL\_SIZE, row \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE)

                if col % 2 == 0 and row % 2 == 0:

                    pygame.draw.rect(screen, constants.CELL\_EVEN\_COLOR, rect)

                    pygame.draw.rect(screen, (0, 0, 0), rect, 1)

                elif col % 2 == 1 and row % 2 == 0:

                    pygame.draw.rect(screen, constants.CELL\_ODD\_COLOR, rect)

                    pygame.draw.rect(screen, (0, 0, 0), rect, 1)

                elif col % 2 == 0 and row % 2 == 1:

                    pygame.draw.rect(screen, constants.CELL\_ODD\_COLOR, rect)

                    pygame.draw.rect(screen, (0, 0, 0), rect, 1)

                else:

                    pygame.draw.rect(screen, constants.CELL\_EVEN\_COLOR, rect)

                    pygame.draw.rect(screen, (0, 0, 0), rect, 1)

        self.draw\_pieces()

        self.draw\_side\_column()

    def draw\_pieces(self):

        for pos in self.logic.player.positions:

            self.draw\_piece(pos, self.COLORS['Player'], self.logic.player.items[pos])

        for pos in self.logic.computer.positions:

            self.draw\_piece(pos, self.COLORS['Computer'], self.logic.computer.items[pos])

        self.draw\_piece(self.logic.wall\_position, self.COLORS['Wall'], None)

    def draw\_piece(self, position, color, item, selected=False):

        row, col = position

        center = (col \* CELL\_SIZE + CELL\_SIZE // 2, row \* CELL\_SIZE + CELL\_SIZE // 2)

        if selected:

            pygame.draw.circle(screen, self.COLORS['Aura'], center, constants.RADIUS + 7)

        pygame.draw.circle(screen, color, center, constants.RADIUS)

        if item:

            screen.blit(images[item], (col \* CELL\_SIZE, row \* CELL\_SIZE))

    def draw\_side\_column(self):

        # Draw the background for the side column

        side\_column\_rect = pygame.Rect(7 \* CELL\_SIZE, 0, constants.SIDE\_COLUMN\_WIDTH, SCREEN\_HEIGHT)

        pygame.draw.rect(screen, constants.SIDE\_COLUMN\_COLOR, side\_column\_rect)

        # Display whose turn it is

        turn\_text = FONT.render("Player's Turn" if self.player\_turn else "Computer's Turn", True, (0, 0, 0))

        screen.blit(turn\_text, (7 \* CELL\_SIZE + 10, 10))

        # Display the number of soldiers

        player\_soldiers\_text = FONT.render(f"Player Soldiers: {len(self.logic.player.positions)}", True, (0, 0, 0))

        screen.blit(player\_soldiers\_text, (7 \* CELL\_SIZE + 10, 50))

        computer\_soldiers\_text = FONT.render(f"Computer Soldiers: {len(self.logic.computer.positions)}", True, (0, 0, 0))

        screen.blit(computer\_soldiers\_text, (7 \* CELL\_SIZE + 10, 90))

        # Draw the shuffle button

        button\_rect = pygame.Rect(7 \* CELL\_SIZE + 10, 150, constants.BUTTON\_WIDTH, constants.BUTTON\_HEIGHT)

        self.draw\_button(button\_rect)

    def draw\_button(self, button\_rect):

        mouse\_pos = pygame.mouse.get\_pos()

        if button\_rect.collidepoint(mouse\_pos) and not self.shuffle\_used:

            pygame.draw.rect(screen, constants.BUTTON\_HOVER\_COLOR, button\_rect)

        elif button\_rect.collidepoint(mouse\_pos) and not self.shuffle\_used:

            pygame.draw.rect(screen, constants.BUTTON\_COLOR, button\_rect)

        elif self.shuffle\_used:

            pygame.draw.rect(screen, constants.BUTTON\_COLOR\_AFTER\_CLICK, button\_rect)

        else:

            pygame.draw.rect(screen, constants.BUTTON\_COLOR, button\_rect)

        text = FONT.render('Shuffle', True, constants.BUTTON\_TEXT\_COLOR)

        text\_rect = text.get\_rect(center=button\_rect.center)

        screen.blit(text, text\_rect)

    def handle\_button\_click(self, event, player):

        button\_rect = pygame.Rect(7 \* CELL\_SIZE + 10, 150, constants.BUTTON\_WIDTH, constants.BUTTON\_HEIGHT)

        if button\_rect.collidepoint(event.pos) and not self.shuffle\_used:

            player.shuffle\_items()

            self.shuffle\_used = True  # Set the flag to True after shuffle

            return True

        return False

**player.py**

import random

import tkinter as tk

from tkinter import messagebox

def show\_message\_box(message):

    root = tk.Tk()

    root.withdraw()  # Hide the main window

    messagebox.showinfo("Shuffle", message)

    root.destroy()

class Player:

    def \_\_init\_\_(self, positions, items):

        self.positions = positions

        self.items = items

        self.flag\_pos = random.choice(positions[:6])

        self.items[self.flag\_pos] = 'Flag'

    def move(self, old\_pos, new\_pos):

        self.positions.remove(old\_pos)

        self.positions.append(new\_pos)

        self.items[new\_pos] = self.items.pop(old\_pos)

    def shuffle\_items(self):

        for pos in self.positions:

            self.items[pos] = random.choice(['Rock', 'Paper', 'Scissors'])

        self.flag\_pos = random.choice(self.positions)

        self.items[self.flag\_pos] = 'Flag'

        show\_message\_box("Items have been shuffled!")

**main.py**

import time

import pygame

import constants

from player import Player

from computer import Computer

from game import Game

from game\_logic import GameLogic

import random

# Initialize Pygame

pygame.init()

# Game loop

player = Player([(5, col) for col in range(7)] + [(4, col) for col in range(7)], {pos: random.choice(['Rock', 'Paper', 'Scissors']) for pos in [(5, col) for col in range(7)] + [(4, col) for col in range(7)]})

computer = Computer([(0, col) for col in range(7)] + [(1, col) for col in range(7)], {pos: random.choice(['Rock', 'Paper', 'Scissors']) for pos in [(0, col) for col in range(7)] + [(1, col) for col in range(7)]})

game\_logic = GameLogic(player, computer)

game = Game(game\_logic)

running = True

selected\_position = None

while running:

    for event in pygame.event.get():

        if event.type == pygame.QUIT:

            running = False

        elif event.type == pygame.MOUSEBUTTONDOWN:

            mouse\_x, mouse\_y = event.pos

            if mouse\_x >= 7 \* constants.CELL\_SIZE:  # Click is in the side column

                if game.handle\_button\_click(event, player):

                    print("Item's has been shuffled !")

            elif game.player\_turn:

                row, col = mouse\_y // constants.CELL\_SIZE, mouse\_x // constants.CELL\_SIZE

                if selected\_position:

                    if (row, col) in game\_logic.get\_valid\_moves(selected\_position, player.positions, game\_logic.wall\_position):

                        game\_logic.move\_and\_battle(player, computer, selected\_position, (row, col))

                        selected\_position = None

                        game.player\_turn = False

                    else:

                        selected\_position = None

                else:

                    for pos in player.positions:

                        if pos == (row, col):

                            selected\_position = pos

                            break

                game.draw\_board()

                if selected\_position:

                    game.draw\_piece(selected\_position, constants.COLORS['Player'], player.items[selected\_position], selected=True)

                pygame.display.flip()

    if not game.player\_turn:

        computer.make\_best\_move(game\_logic)

        time.sleep(0.2)

        game.player\_turn = True

    game.draw\_board()

    if selected\_position:

        game.draw\_piece(selected\_position, constants.COLORS['Player'], player.items[selected\_position], selected=True)

    pygame.display.flip()

pygame.quit()

**game\_logic.py**

import random

import sys

import tkinter as tk

from tkinter import messagebox

import pygame

import time

def show\_message\_box(message):

    root = tk.Tk()

    root.withdraw()

    messagebox.showinfo("Game Over", message)

    root.destroy()

def show\_message\_box\_same\_weapon(message):

    root = tk.Tk()

    root.withdraw()

    messagebox.showinfo("Same Weapon", message)

    root.destroy()

class GameLogic:

    def \_\_init\_\_(self, player, computer):

        self.player = player

        self.computer = computer

        self.wall\_position = (random.randint(2, 3), random.randint(0, 6))

    def move\_and\_battle(self, player, opponent, position, new\_position, is\_computer=False):

        player.move(position, new\_position)

        if new\_position in opponent.positions:

            self.battle(player, opponent, new\_position, is\_computer)

        self.check\_victory()

    def check\_victory(self):

        if 'Flag' not in self.player.items.values():

            show\_message\_box("Computer wins!")

            pygame.quit()

            sys.exit()

        if 'Flag' not in self.computer.items.values():

            show\_message\_box("Player wins!")

            pygame.quit()

            sys.exit()

    def determine\_winner(self, player\_item, opponent\_item, position, is\_computer):

        if player\_item == opponent\_item and player\_item != 'Flag':

            show\_message\_box\_same\_weapon("Same Weapon, Beginning the random choice")

            while player\_item == opponent\_item:

                player\_item = random.choice(['Rock', 'Paper', 'Scissors'])

                opponent\_item = random.choice(['Rock', 'Paper', 'Scissors'])

            if is\_computer:

                show\_message\_box\_same\_weapon(f"Computer's new weapon is :{player\_item}\nPlayer's new weapon is :{opponent\_item}")

            else:

                show\_message\_box\_same\_weapon(f"Player's new weapon is :{player\_item}\nComputer's new weapon is :{opponent\_item}")

        if is\_computer:

            self.computer.items[position] = player\_item

            self.player.items[position] = opponent\_item

        else:

            self.player.items[position] = player\_item

            self.computer.items[position] = opponent\_item

        if (player\_item == 'Rock' and opponent\_item == 'Scissors') or \

           (player\_item == 'Paper' and opponent\_item == 'Rock') or \

           (player\_item == 'Scissors' and opponent\_item == 'Paper'):

            return "Attacker"

        if opponent\_item == 'Flag':

            return "Attacker"

        else:

            return "Defender"

    def battle(self, attacker, defender, position, is\_computer=False):

        attacker\_item = attacker.items[position]

        defender\_item = defender.items[position]

        result = self.determine\_winner(attacker\_item, defender\_item, position, is\_computer)

        time.sleep(0.1)

        print("Beginning battle")

        if result == "Attacker":

            if is\_computer:

                print("Computer Won")

            else:

                print("Player Won")

            defender.positions.remove(position)

            defender.items.pop(position)

        else:

            if is\_computer:

                print("Player Won")

            else:

                print("Computer Won")

            attacker.positions.remove(position)

            attacker.items.pop(position)

    def get\_valid\_moves(self, position, other\_positions, wall\_position):

        row, col = position

        potential\_moves = [(row-1, col), (row+1, col), (row, col-1), (row, col+1)]

        valid\_moves = [move for move in potential\_moves if 0 <= move[0] < 6 and 0 <= move[1] < 7 and move not in other\_positions + [wall\_position]]

        return valid\_moves