Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"
Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Звіт

з лабораторної роботи № 6 з дисципліни
«Проектування алгоритмів»

„Пошук в умовах протидії, ігри з повною інформацією, ігри з елементом випадковості, ігри з неповною інформацією”

Виконав(ла) *ІП-Карпов Л В* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Головченко М.Н.* _____
(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Мета роботи - вивчити основні підходи до формалізації алгоритмів знаходження рішень задач в умовах протидії. Ознайомитися з підходами до програмування алгоритмів штучного інтелекту в іграх з повною інформацією, іграх з елементами випадковості та в іграх з неповною інформацією.

2 ЗАВДАННЯ

Для ігор з повної інформацією, згідно варіанту (таблиця 2.1) реалізувати візуальний ігровий додаток для гри користувача з комп'ютерним опонентом. Для реалізації стратегії гри комп'ютерного опонента використовувати алгоритм альфа-бета-відсікань. Реалізувати три рівні складності (легкий, середній, складний).

Для ігор з елементами випадковості, згідно варіанту (таблиця 2.1) реалізувати візуальний ігровий додаток, з користувацьким інтерфейсом, не консольним, для гри користувача з комп'ютерним опонентом. Для реалізації стратегії гри комп'ютерного опонента використовувати алгоритм мінімакс.

Для карткових ігор, згідно варіанту (таблиця 2.1), реалізувати візуальний ігровий додаток, з користувацьким інтерфейсом, не консольним, для гри користувача з комп'ютерним опонентом. Потрібно реалізувати стратегію комп'ютерного опонента, і звести гру до гри з повною інформацією (див. Лекцію), далі реалізувати стратегію гри комп'ютерного опонента за допомогою алгоритму мінімаксу або альфа-бета-відсікань.

Реалізувати анімацію процесу жеребкування (+1 бал) або реалізувати анімацію ігрових процесів (роздачі карт, анімацію ходів тощо) (+1 бал).

Реалізувати варто тільки одне з бонусних завдань.

Зробити узагальнений висновок лабораторної роботи.

Таблиця 2.1 – Варіанти

№	**Варіант**	**Тип гри**
1	Яцзи https://game-wiki.guru/published/igryi/yaczzyi.html	З елементами випадковості
2	Лудо http://www.iggamecenter.com/info/ru/ludo.html	З елементами випадковості
3	Генерал http://www.rules.net.ru/kost.php?id=7	З елементами випадковості

4	Нейтріко http://www.iggamecenter.com/info/ru/neutreeko.html	З повною інформацією
5	Тринадцять http://www.rules.net.ru/kost.php?id=16	З елементами випадковості
6	Індійські кості http://www.rules.net.ru/kost.php?id=9	З елементами випадковості
7	Dots and Boxes https://ru.wikipedia.org/wiki/Палочки_(игра)	З повною інформацією
8	Двадцять одне http://gamerules.ru/igry-v-kosti-part8#dvadtsat-odno	З елементами випадковості
9	Тіко http://www.iggamecenter.com/info/ru/teeko.html	З повною інформацією
10	Клоббер http://www.iggamecenter.com/info/ru/clobber.html	З повною інформацією
11	101 https://www.durbetsel.ru/2_101.htm	Карткові ігри
12	Hackenbush http://www.papg.com/show?1TMP	З повною інформацією
13	Табу https://www.durbetsel.ru/2_taboo.htm	Карткові ігри
14	Заєць і Вовки (за Зайця) http://www.iggamecenter.com/info/ru/foxh.html	З повною інформацією
15	Свої козирі https://www.durbetsel.ru/2_svoi-koziri.htm	Карткові ігри
16	Війна з ботами https://www.durbetsel.ru/2_voina_s_botami.htm	Карткові ігри
17	Domineering 8x8 http://www.papg.com/show?1TX6	З повною інформацією
18	Останній гравець https://www.durbetsel.ru/2_posledny_igrok.htm	Карткові ігри
19	Заєць и Вовки (за Вовків) http://www.iggamecenter.com/info/ru/foxh.html	З повною інформацією

20	Богач https://www.durbetsel.ru/2_bogach.htm	Карткові ігри
21	Редуду https://www.durbetsel.ru/2_redudu.htm	Карткові ігри
22	Эльферн https://www.durbetsel.ru/2_elfern.htm	Карткові ігри
23	Ремінь https://www.durbetsel.ru/2_remen.htm	Карткові ігри
24	Реверсі https://ru.wikipedia.org/wiki/Реверси	З повною інформацією
25	Вари http://www.iggamecenter.com/info/ru/oware.html	З повною інформацією
26	Яцзи https://game-wiki.guru/published/igryi/yaczzyi.html	З елементами випадковості
27	Лудо http://www.iggamecenter.com/info/ru/ludo.html	З елементами випадковості
28	Генерал http://www.rules.net.ru/kost.php?id=7	З елементами випадковості
29	Сим https://ru.wikipedia.org/wiki/Сим_(игра)	З повною інформацією
30	Col http://www.papg.com/show?2XLY	З повною інформацією
31	Snort http://www.papg.com/show?2XM1	З повною інформацією
32	Chomp http://www.papg.com/show?3AEA	З повною інформацією
33	Gale http://www.papg.com/show?1TPI	З повною інформацією
34	3D Noughts and Crosses 4 x 4 x 4 http://www.papg.com/show?1TND	З повною інформацією
35	Snakes http://www.papg.com/show?3AE4	З повною інформацією

3 ВИКОНАННЯ

3.1 Програмна реалізація алгоритму

3.1.1 Вихідний код

Game

```
import numpy as np


class boardGame:
    def __init__(self, board, player, n):
        self.board = np.array(board)
        self.playerPlaying = player
        self.remainingPawns = n

    def print(self):
        print("\n")
        print(self.board)
        if self.remainingPawns != 8:
            print("\nPlayer " + str(-self.playerPlaying) + " juste played. Turn
for " + str(self.playerPlaying) + ".\n")

    def initialize(self):
        self.board = np.zeros((5, 5), dtype=np.int16)
        self.playerPlaying = 1
        self.remainingPawns = 8

    def switchPlayer(self):
        self.playerPlaying *= -1

    def place(self, x, y, cflag):
        if self.board[x][y] != 0:
            if cflag:
                print("This position is occupied. \n")
            return False
        else:
            self.board[x][y] = self.playerPlaying
            self.remainingPawns -= 1
            self.switchPlayer()
            return True

    def move(self, x, y, a, b, cflag):
        if a != x or b != y:
            if self.board[a][b] != 0:
                if cflag:
                    print("This destination is occupied. \n")
                return False
            else:
                if self.board[x][y] == 0:
                    if cflag:
                        print("There is no pawn to move at this position. \n")
                    return False
                else:
                    if self.board[x][y] == self.playerPlaying:
                        adjacentSlots = self.getAdjacent(x, y)
                        if [a, b] in adjacentSlots:
                            self.board[a][b] = self.board[x][y]
                            self.board[x][y] = 0
```

```
                            self.switchPlayer()
                            return True
                        else:
                            if cflag:
                                print("Destination is not adjacent to the
selected pawn. \n")
                            return False
                    else:
                        if cflag:
                            print("The pawn selected is not one of yours")
                        return False
        else:
            if cflag:
                print("Initial position and destination must be different. \n")
            return False

    def getAdjacent(self, a, b):
        adjacentSlots = []
        directions = [
            [-1, -1], [-1, 0], [-1, +1],
            [0, -1], [0, +1],
            [+1, -1], [+1, 0], [+1, +1],
        ]

        for i in directions:
            x = a + i[0]
            y = b + i[1]

            if 0 <= x <= 4 and 0 <= y <= 4:
                adjacentSlots.append([x, y])
        return adjacentSlots

    def winner(self):
        x = 0
        y = 0
        posX = 0
        posY = 0

        # LINES
        for i in range(5):
            if np.sum(self.board[i] == 1) == 4:
                if np.all(self.board[i][0:4] == 1) or np.all(self.board[i][1:5]
== 1):
                    return 1
            elif np.sum(self.board[i] == -1) == 4:  #
                if np.all(self.board[i][0:4] == -1) or np.all(self.board[i][1:5]
== -1):
                    return -1

        # COLUMNS
        for i in range(5):
            column = self.board[:, i]
            if np.sum(column == 1) == 4:
                if np.all(column[0:4] == 1) or np.all(column[1:5] == 1):
                    return 1
            elif np.sum(column == -1) == 4:
                if np.all(column[0:4] == -1) or np.all(column[1:5] == -1):
                    return -1

        # DIAGONALS
        for i in range(-1, 2):
            diag = self.board.diagonal(i)
            oppdiag = np.fliplr(self.board).diagonal(i)
```

```
            if np.sum(diag == 1) == 4 or np.sum(oppdiag == 1) == 4:
                if np.all(diag[0:4] == 1) or np.all(oppdiag[0:4] == 1):
                    return 1
                if len(diag) > 4:
                    if np.all(diag[1:5] == 1) or np.all(oppdiag[1:5] == 1):
                        return 1
            elif np.sum(diag == -1) == 4 or np.sum(oppdiag == -1) == 4:
                if np.all(diag[0:4] == -1) or np.all(oppdiag[0:4] == -1):
                    return -1
                if len(diag) > 4:
                    if np.all(diag[1:5] == -1) or np.all(oppdiag[1:5] == -1):
                        return -1

        # CUBES
        for i in range(4):
            for j in range(4):
                c = [self.board[i][j:j + 2], self.board[i + 1][j:j + 2]]
                cube = np.array(c)

                if np.all(cube == 1):
                    return 1
                elif np.all(cube == -1):
                    return -1

        return 0

    def playPlayer(self):

        if self.remainingPawns != 0:  # If game in placing phase
            while True:
                try:
                    x_pos = int(input("Choose x position: "))
                    y_pos = int(input("Choose y position: "))
                except:
                    print("Sorry, I didn't understand that.")
                    continue

                if not (0 <= x_pos <= 4 and 0 <= y_pos <= 4):
                    print("Please select values from 0 to 4.")
                    continue

                if not self.place(x_pos, y_pos, True):
                    continue
                else:
                    break

        else:  # if game in moving phase
            while True:
                try:
                    x_pos = int(input("Choose x position: "))
                    y_pos = int(input("Choose y position: "))

                    a_pos = int(input("Choose destination x position: "))
                    b_pos = int(input("Choose destination y position: "))

                except:
                    print("Sorry, I didn't understand that.")
                    continue

                if not (0 <= x_pos <= 4 and 0 <= y_pos <= 4 and 0 <= a_pos <= 4
and 0 <= b_pos <= 4):
                    print("Please select values from 0 to 4.")
                    continue
```

```
                if not self.move(x_pos, y_pos, a_pos, b_pos, True):
                    continue
                else:
                    break
```

ui

```
from tkinter import *
import ai
import game

boardWidth = 400
boardHeight = 400
ncols = 5
nrows = 5
cellWidth = boardWidth / ncols
cellHeight = boardHeight / nrows

COLOR = 'grey'
COLOR1 = 'blue'
COLOR2 = 'red'
BGCOLOR = 'dark grey'

depth = 3

state = game.boardGame([
    [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]
], 1, 8)

ai = ai.TeekoAI(state, -1)


class Interface():
    def __init__(self):

        # Game parameters
        self.mode = 0
        self.difficulty = 1

        # Main UI
        self.window = Tk()
        self.window.title("Teeko Game")

        self.frame = Frame(self.window)
        self.frame.config(background=BGCOLOR)

        self.playButton = Button(self.frame, text="Play", font=("Courrier", 25),
bg=BGCOLOR,
                                 command=lambda: [self.window.withdraw(),
self.openPlayConfig()])
        self.playButton.pack(pady=20, fill=X)

        self.quitButton = Button(self.frame, text="Quit", font=("Courrier", 25),
bg=BGCOLOR, command=self.window.quit)
        self.quitButton.pack(pady=20, fill=X)

        self.frame.pack(expand=YES)

        self.window.geometry('600x400')
        self.window.minsize(600, 400)
```

```
        self.window.config(background=BGCOLOR)
        self.center_window(self.window, 600, 400)

        self.window.mainloop()

    def center_window(self, wind, w, h):
        ws = wind.winfo_screenwidth()
        hs = wind.winfo_screenheight()
        x = (ws / 2) - (w / 2)
        y = (hs / 2) - (h / 2)
        wind.geometry('%dx%d+%d+%d' % (w, h, x, y))

    def changeMode(self, i):
        self.mode = 1 if i == 1 else 0
        modeText = "PvP" if self.mode == 0 else "PvE"
        self.modeLabel.config(text="Mode selected : " + modeText)

    def changeDifficulty(self, i):
        self.difficulty = i

    def drawCell(self, canvas, x, y, s):
        x1 = cellWidth * x
        y1 = cellHeight * y
        x2 = x1 + cellWidth
        y2 = y1 + cellHeight

        if (s == 1):
            self.canvas.create_rectangle(x1, y1, x2, y2, fill=COLOR1,
activefill=COLOR)
        elif (s == -1):
            self.canvas.create_rectangle(x1, y1, x2, y2, fill=COLOR2,
activefill=COLOR)
        else:
            self.canvas.create_rectangle(x1, y1, x2, y2, fill='white',
activefill=COLOR)

        self.canvas.bind('<Button-1>', self.onClick)

    def drawBoard(self, canvas, state):

        for i in range(5):
            for j in range(5):
                s = state.board[i][j]
                self.drawCell(self.canvas, j, i, s)

    def openPlayConfig(self):
        self.playConfig = Tk()
        self.playConfig.title("Game configuration")

        frame = Frame(self.playConfig)
        frame.config(background=BGCOLOR)

        # Difficulty slider
        difficultyLabel = Label(frame, text="Choose your difficulty level:",
font=("Courrier", 15), bg=BGCOLOR,
                                fg="White")
        difficultyLabel.pack()
        self.selector = Scale(frame, from_=1, to=3, orient=HORIZONTAL,
bg=BGCOLOR, fg="White")
        self.selector.pack(expand=YES, pady=10, fill=X)

        # Mode buttons
        pvpButton = Button(frame, text="PvP", bg=BGCOLOR, command=lambda:
self.changeMode(0))
```

```
        pvaiButton = Button(frame, text="PvE", bg=BGCOLOR, command=lambda:
self.changeMode(1))
        pvpButton.pack(expand=YES, side='left', pady=20, fill=X)
        pvaiButton.pack(expand=YES, side='left', pady=20, fill=X)

        frame.pack(expand=YES)

        self.modeLabel = Label(self.playConfig, text="Mode selected : PvP",
font=("Courrier, 20"), fg="White",
                               bg=BGCOLOR)
        self.modeLabel.pack(expand=YES)

        # Back button
        popupButton = Button(self.playConfig, text="Back", font=("Courrier,
20"),
                             command=lambda: [self.playConfig.withdraw(),
self.window.deiconify()], bg=BGCOLOR)
        popupButton.pack(side="bottom", pady=20)

        # Launch game button
        launchGameButton = Button(self.playConfig, text='Launch Game',
font=("Courrier, 20"),
                                  command=lambda: [self.playConfig.withdraw(),
self.openGameWindow(),

self.changeDifficulty(self.selector.get())], bg=BGCOLOR)
        launchGameButton.pack(side="bottom", pady=20)

        # Window parameters
        self.playConfig.minsize(1200, 400)
        self.playConfig.configure(bg=BGCOLOR)
        self.center_window(self.playConfig, 1200, 300)

    def openGameWindow(self):
        self.gameWindow = Tk()
        self.gameWindow.title("Playing game...")

        self.gameFinished = False

        board = Frame(self.gameWindow)
        board.config(background=BGCOLOR)

        # Previously selected pawn
        self.selectedPawnX = None
        self.selectedPawnY = None

        # Game Label
        playerValue = 1 if state.playerPlaying == 1 else 2
        self.gameLabel = Label(self.gameWindow, text="Player " +
str(playerValue) + "'s turn:", font=("Courrier", 40),
                               bg=BGCOLOR, fg="White")
        self.gameLabel.pack(pady=(100, 20))

        self.canvas = Canvas(board, width=boardWidth, height=boardHeight, bd=1,
highlightthickness=0, relief='ridge')
        self.canvas.pack()
        board.pack(expand=YES)

        # initialize the game board
        self.drawBoard(self.canvas, state)

        # Quit game button
        quitButton = Button(self.gameWindow, text="Quit game", bg=BGCOLOR,
                            command=lambda: [self.gameWindow.withdraw(),
```

```
self.window.deiconify(), state.initialize()])
        quitButton.pack(side="bottom", pady=20)

        # Window parameters
        self.gameWindow.minsize(800, 800)
        self.gameWindow.config(background=BGCOLOR)
        self.center_window(self.gameWindow, 800, 800)

    def onClick(self, event):

        y = int(event.x // cellWidth)
        x = int(event.y // cellHeight)

        global state

        if state.board[x][y] != 0 and state.remainingPawns == 0:
            self.selectedPawnX = x
            self.selectedPawnY = y

        if not self.gameFinished:

            if state.remainingPawns != 0:

                if state.place(x, y, True):

                    if self.mode == 1:
                        if self.difficulty == 1:
                            ai.playEasy()  # Easy
                        elif self.difficulty == 2:
                            state = ai.playMediumOrHard(depth, 0)  # Medium
                        elif self.difficulty == 3:
                            state = ai.playMediumOrHard(depth, 1)  # Hard
                        else:
                            print("Error\n")

            else:
                try:
                    if self.selectedPawnX is not None and self.selectedPawnY is
not None:
                        if state.move(self.selectedPawnX, self.selectedPawnY, x,
y, True):

                            self.selectedPawnX = None
                            self.selectedPawnY = None

                            if self.mode == 1:

                                if self.difficulty == 1:
                                    ai.playEasy()  # Easy
                                elif self.difficulty == 2:
                                    state = ai.playMediumOrHard(depth, 0)  #
Medium
                                elif self.difficulty == 3:
                                    state = ai.playMediumOrHard(depth, 1)  #
Hard
                                else:
                                    print("Error\n")
                except:
                    print("Fail")

        self.drawBoard(self.canvas, state)
        playerValue = 1 if state.playerPlaying == 1 else 2
        self.gameLabel.config(text="Player " + str(playerValue) + "'s turn:")
```

```
        if state.winner() != 0:
            if (state.winner() == 1):
                self.gameLabel.config(text="you've won!")
            else:
                self.gameLabel.config(text="you lost")
            self.gameFinished = True



app = Interface()
```

ai

```
import random
import copy

maxSize = float("inf")


class TeekoAI:
    def __init__(self, board, IAplayer):
        self.board = board
        self.IAplayer = IAplayer

    def playEasy(self):
        if self.board.remainingPawns != 0:
            while 1:
                x = random.randint(0, 4)
                y = random.randint(0, 4)
                if self.board.place(int(x), int(y), False):
                    break
        else:
            while 1:
                x = random.randint(0, 4)
                y = random.randint(0, 4)
                a = random.randint(0, 4)
                b = random.randint(0, 4)
                if self.board.move(int(x), int(y), int(a), int(b), False):
                    break

    def playMediumOrHard(self, depth, mode):

        player = self.board.playerPlaying
        bestScore = maxSize * -player
        tempScore = 0
        alpha = -maxSize
        beta = maxSize

        tempState = copy.deepcopy(self.board)
        bestState = None

        if self.board.remainingPawns != 0:
            for x in range(5):
                for y in range(5):
                    if tempState.place(x, y, False):
                        if player == -1:
                            tempScore = self.minimax(mode, tempState, depth - 1,
alpha, beta, False)
                            if tempScore <= bestScore:
                                bestScore = tempScore
                                bestState = tempState
                        elif player == 1:
                            tempScore = self.minimax(mode, tempState, depth - 1,
alpha, beta, True)
```

```
                            if tempScore >= bestScore:
                                bestScore = tempScore
                                bestState = tempState
                        tempState = copy.deepcopy(self.board)
        else:
            for x in range(5):
                for y in range(5):
                    adjacents = tempState.getAdjacent(x, y)

                    for adjacent in adjacents:
                        if tempState.move(x, y, adjacent[0], adjacent[1],
False):
                            if player == -1:
                                tempScore = self.minimax(mode, tempState, depth
- 1, alpha, beta, False)
                                if tempScore <= bestScore:
                                    bestScore = tempScore
                                    bestState = tempState
                            elif player == 1:
                                tempScore = self.minimax(mode, tempState, depth
- 1, alpha, beta, True)
                                if tempScore >= bestScore:
                                    bestScore = tempScore
                                    bestState = tempState
                            tempState = copy.deepcopy(self.board)

        self.board = bestState
        return bestState

    def minimax(self, mode, childState, depth, alpha, beta, isMaximizing):

        if childState.winner() != 0 or depth == 0:
            return self.eval(childState, depth, mode)

        if isMaximizing:
            bestScore = -maxSize
            tempScore = 0

            tempState = copy.deepcopy(childState)

            if tempState.remainingPawns != 0:
                for x in range(5):
                    for y in range(5):
                        if tempState.place(x, y, False):
                            tempScore = self.minimax(mode, tempState, depth - 1,
alpha, beta, False)
                            alpha = tempScore

                            if tempScore >= bestScore:
                                bestScore = tempScore
                            if alpha >= beta:
                                return alpha
                            tempState = tempState = copy.deepcopy(childState)

            else:
                for x in range(5):
                    for y in range(5):
                        adjacents = tempState.getAdjacent(x, y)

                        for adjacent in adjacents:
                            if tempState.move(x, y, adjacent[0], adjacent[1],
False):

                                tempScore = self.minimax(mode, tempState, depth
```

```
- 1, alpha, beta, False)
                                alpha = tempScore

                                if tempScore >= bestScore:
                                    bestScore = tempScore
                                if alpha >= beta:
                                    return alpha
                                tempState = tempState =
copy.deepcopy(childState)
            return bestScore
        else:
            bestScore = maxSize
            tempScore = 0

            tempState = tempState = copy.deepcopy(childState)

            if tempState.remainingPawns != 0:
                for x in range(5):
                    for y in range(5):
                        if tempState.place(x, y, False):
                            tempScore = self.minimax(mode, tempState, depth - 1,
alpha, beta, True)
                            beta = tempScore

                            if tempScore <= bestScore:
                                bestScore = tempScore
                            if alpha >= beta:
                                return beta
                            tempState = tempState = copy.deepcopy(childState)

            else:
                for x in range(5):
                    for y in range(5):

                        adjacents = tempState.getAdjacent(x, y)
                        for adjacent in adjacents:
                            if tempState.move(x, y, adjacent[0], adjacent[1],
False):

                                tempScore = self.minimax(mode, tempState, depth
- 1, alpha, beta, True)
                                beta = tempScore
                                if tempScore <= bestScore:
                                    bestScore = tempScore
                                if alpha >= beta:
                                    return beta
                                tempState = tempState =
copy.deepcopy(childState)
            return bestScore

    def eval(self, state, depth, mode):
        if state.winner() != 0:
            return maxSize * state.winner()
        else:
            value = 0
            if mode == 0:  ##MEDIUM LEVEL
                for x in range(5):
                    for y in range(5):
                        if state.board[x][y] != 0:
                            pawnsWeight = self.stateWeightForPawn(state, x, y)
                            for a in range(5):
                                for b in range(5):
                                    if state.board[a][b] != 0:
                                        value = value + state.board[a][b] *
```

```
pawnsWeight[a][b]
                return value
            elif mode == 1:  ##HARD LEVEL
                boardWeights = [
                    [0, 1, 0, 1, 0],
                    [1, 2, 2, 2, 1],
                    [0, 2, 3, 2, 0],
                    [1, 2, 2, 2, 1],
                    [0, 1, 0, 1, 0]
                ]
                for x in range(5):
                    for y in range(5):
                        if state.board[x][y] != 0:
                            value = value + state.board[x][y] *
boardWeights[x][y]

                return value

    def stateWeightForPawn(self, state, x, y):
        stateWeight = [
            [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]
        ]

        nearPawns = state.getAdjacent(x, y)
        for nearPawn in nearPawns:
            stateWeight[nearPawn[0]][nearPawn[1]] = 2

        remotePawnsCoordinates = [[x - 1, y - 2], [x + 1, y - 2], [x - 1, y +
2], [x + 1, y + 2], [x - 2, y - 1],
                                  [x - 2, y + 1], [x + 2, y - 1], [x + 2, y +
1]]
        for remotePawn in remotePawnsCoordinates:
            if 0 <= remotePawn[0] <= 4 and 0 <= remotePawn[1] <= 4:
                stateWeight[remotePawn[0]][remotePawn[1]] = 1

        return stateWeight
```

3.1.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

Рисунок 3.1 –

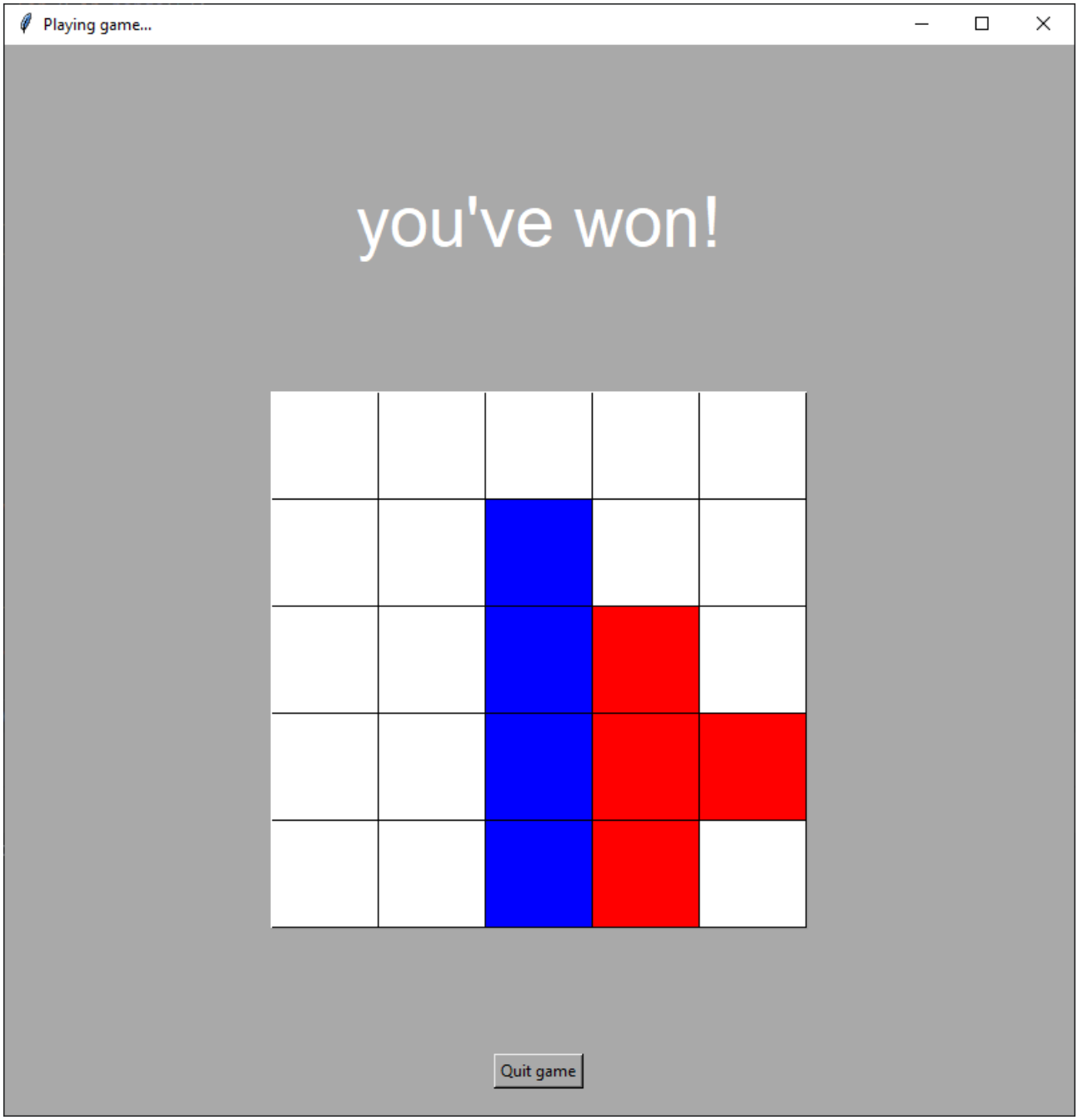
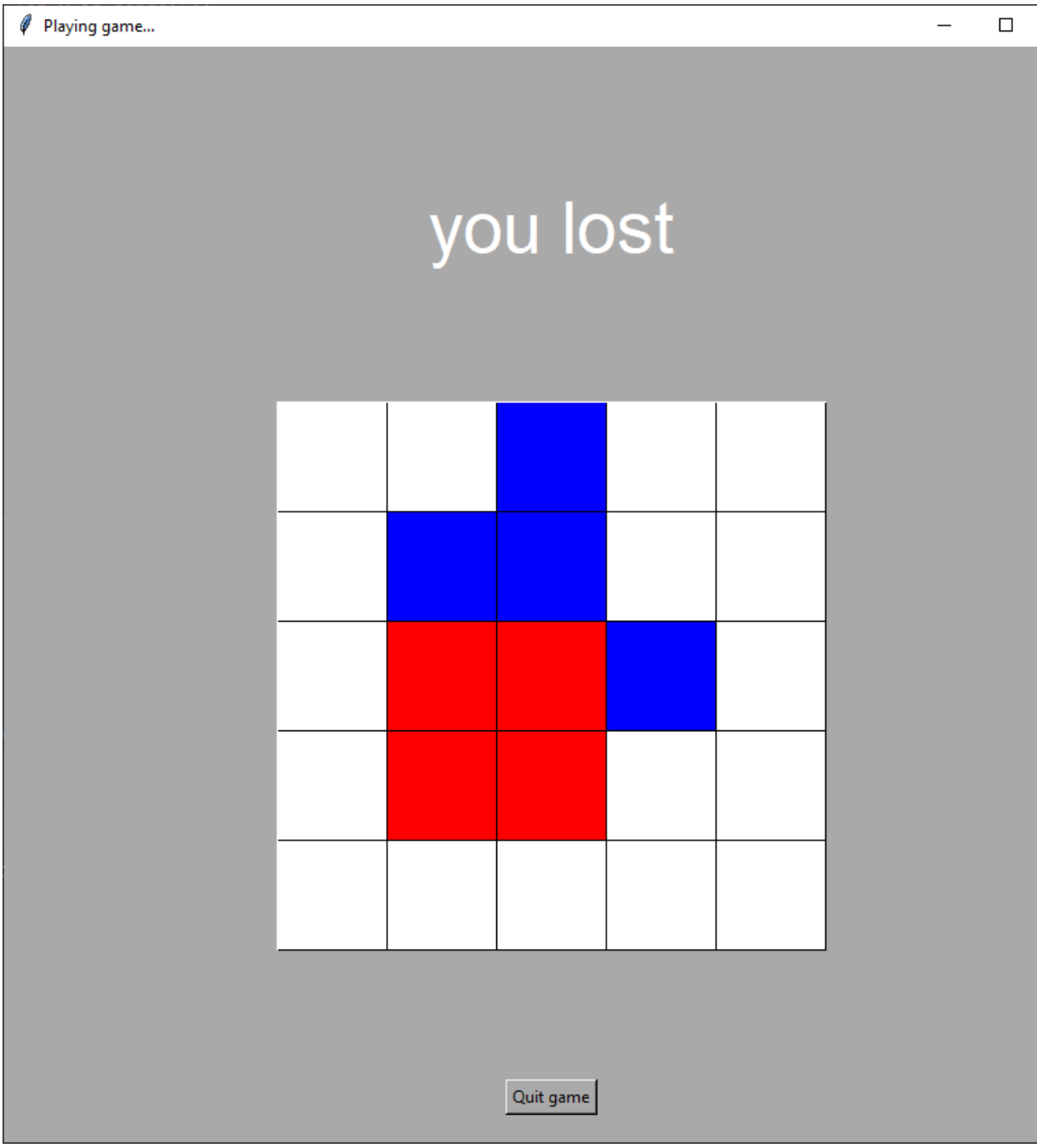


Рисунок 3.2 –

3.2



ВИСНОВОК

В рамках даної лабораторної роботи був розроблений алгоритм для гри з повною інформацією тіко

КРИТЕРІЇ ОЦІНЮВАННЯ

При здачі лабораторної роботи до 31.12.2023 включно максимальний бал дорівнює – 5. Після 31.12.2023 максимальний бал дорівнює – 4,5.

Критерії оцінювання у відсотках від максимального балу:

- програмна реалізація – 75%;
- робота з гіт – 20%;
- висновок – 5%.

+1 додатковий бал можна отримати за реалізацію анімації ігрових процесів (жеребкування, роздачі карт, анімацію ходів тощо).

+1 додатковий бал можна отримати за виконання та захист роботи до 24.12.2023.