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-12Бобрик Максим Геннадійович*
(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

```
import numpy as np
from game2dboard import Board
import easygui as eg
import math


def init(board):
    odd_middle = math.floor(size / 2)
    board[0, odd_middle - 1] = 1
    board[0, odd_middle + 1] = 1
    board[1, odd_middle] = 2

    board[size - 1, odd_middle - 1] = 2
    board[size - 1, odd_middle + 1] = 2
    board[size - 2, odd_middle] = 1


def array_to_gui(array_board, GUI_board):
    GUI_board.clear()
    for i in range(0, size):
        for j in range(0, size):
                GUI_board[i][j] = int(array_board[i, j])


def move_piece_gui(board, init_coord, final_coord):
    if not np.array_equal(final_coord, init_coord):
        board[final_coord[0]][final_coord[1]] =
board[init_coord[0]][init_coord[1]]
        board[init_coord[0]][init_coord[1]] = 0


def location_of_pieces(board):
    pieces_loc = np.array([[0, 0]])
    for i in range(size):
        for j in range(size):
            pos = [i, j]
            if board[i, j] == 1:
                pieces_loc = np.insert(pieces_loc, 0, pos, axis=0)
            elif board[i, j] == 2:
                pieces_loc = np.insert(pieces_loc, pieces_loc.shape[0] - 1, pos,
axis=0)

    pieces_loc = np.delete(pieces_loc, -1, 0)
    return pieces_loc


def legal(board, coord, direction):
    init_coord = np.array(coord)
    empty = True
    new_pos = np.array(coord)

    if direction == 1:  # UP
        while empty:
            if new_pos[0] != 0 and board[new_pos[0] - 1, new_pos[1]] == 0:
                new_pos[0] = new_pos[0] - 1
```

```
            else:
                empty = False

    elif direction == 2:  # Down
        while empty:
            if new_pos[0] != size - 1 and board[new_pos[0] + 1, new_pos[1]] ==
0:
                new_pos[0] = new_pos[0] + 1
            else:
                empty = False

    elif direction == 3:  # Right
        while empty:
            if new_pos[1] != size - 1 and board[new_pos[0], new_pos[1] + 1] ==
0:
                new_pos[1] = new_pos[1] + 1
            else:
                empty = False

    elif direction == 4:  # Left
        while empty:
            if new_pos[1] != 0 and board[new_pos[0], new_pos[1] - 1] == 0:
                new_pos[1] = new_pos[1] - 1
            else:
                empty = False

    elif direction == 5:  # Up right
        while empty:
            if (new_pos[0] != 0 and new_pos[1] != size - 1) and board[new_pos[0]
- 1, new_pos[1] + 1] == 0:
                new_pos[0] = new_pos[0] - 1
                new_pos[1] = new_pos[1] + 1
            else:
                empty = False

    elif direction == 6:  # Up left
        while empty:
            if (new_pos[0] != 0 and new_pos[1] != 0) and board[new_pos[0] - 1,
new_pos[1] - 1] == 0:
                new_pos[0] = new_pos[0] - 1
                new_pos[1] = new_pos[1] - 1
            else:
                empty = False

    elif direction == 7:  # Down right
        while empty:
            if (new_pos[0] != size - 1 and new_pos[1] != size - 1) and
board[new_pos[0] + 1, new_pos[1] + 1] == 0:
                new_pos[0] = new_pos[0] + 1
                new_pos[1] = new_pos[1] + 1
            else:
                empty = False

    elif direction == 8:  # Down left
        while empty:
            if (new_pos[0] != size - 1 and new_pos[1] != 0) and board[new_pos[0]
+ 1, new_pos[1] - 1] == 0:
                new_pos[0] = new_pos[0] + 1
                new_pos[1] = new_pos[1] - 1
            else:
                empty = False

    if np.array_equal(new_pos, init_coord):
        return init_coord
```

```
    else:
        return new_pos


def move_piece(board, init_coord, final_coord):
    if not np.array_equal(final_coord, init_coord):
        board[final_coord[0], final_coord[1]] = board[init_coord[0],
init_coord[1]]
        board[init_coord[0], init_coord[1]] = 0


def find_piece(board, coord, player, piece_next_to):
    new_pos = coord[:]
    direction = 0
    empty = True

    while empty:
        if board[new_pos[0], new_pos[1]] == player and not
np.array_equal(new_pos, piece_next_to):
            direction = 1
            return new_pos, direction
        elif new_pos[0] != 0 and board[new_pos[0], new_pos[1]] == 0:
            new_pos[0] = new_pos[0] - 1
        else:
            empty = False   # up

    empty = True
    new_pos = coord[:]
    while empty:
        if board[new_pos[0], new_pos[1]] == player and not
np.array_equal(new_pos, piece_next_to):
            direction = 2
            return new_pos, direction
        elif new_pos[0] != size - 1 and board[new_pos[0], new_pos[1]] == 0:
            new_pos[0] = new_pos[0] + 1
        else:
            empty = False   # down

    empty = True
    new_pos = coord[:]
    while empty:
        if board[new_pos[0], new_pos[1]] == player and not
np.array_equal(new_pos, piece_next_to):
            direction = 3
            return new_pos, direction
        elif new_pos[1] != size - 1 and board[new_pos[0], new_pos[1]] == 0:
            new_pos[1] = new_pos[1] + 1
        else:
            empty = False   # right

    empty = True
    new_pos = coord[:]
    while empty:
        if board[new_pos[0], new_pos[1]] == player and not
np.array_equal(new_pos, piece_next_to):
            direction = 4
            return new_pos, direction
        elif new_pos[1] != 0 and board[new_pos[0], new_pos[1]] == 0:
            new_pos[1] = new_pos[1] - 1
        else:
            empty = False   # left

    empty = True
    new_pos = coord[:]
```

```
    while empty:
        if board[new_pos[0], new_pos[1]] == player and not
np.array_equal(new_pos, piece_next_to):
            direction = 5
            return new_pos, direction
        elif (new_pos[0] != 0 and new_pos[1] != size - 1) and board[new_pos[0],
new_pos[1]] == 0:
            new_pos[0] = new_pos[0] - 1
            new_pos[1] = new_pos[1] + 1
        else:
            empty = False   # up right

    empty = True
    new_pos = coord[:]
    while empty:
        if board[new_pos[0], new_pos[1]] == player and not
np.array_equal(new_pos, piece_next_to):
            direction = 6
            return new_pos, direction
        elif (new_pos[0] != 0 and new_pos[1] != 0) and board[new_pos[0],
new_pos[1]] == 0:
            new_pos[0] = new_pos[0] - 1
            new_pos[1] = new_pos[1] - 1
        else:
            empty = False   # up left

    empty = True
    new_pos = coord[:]
    while empty:
        if board[new_pos[0], new_pos[1]] == player and not
np.array_equal(new_pos, piece_next_to):
            direction = 7
            return new_pos, direction
        elif (new_pos[0] != size - 1 and new_pos[1] != size - 1) and
board[new_pos[0], new_pos[1]] == 0:
            new_pos[0] = new_pos[0] + 1
            new_pos[1] = new_pos[1] + 1
        else:
            empty = False   # down right

    empty = True
    new_pos = coord[:]
    while empty:
        if board[new_pos[0], new_pos[1]] == player and not
np.array_equal(new_pos, piece_next_to):
            direction = 8
            return new_pos, direction
        elif (new_pos[0] != size - 1 and new_pos[1] != 0) and board[new_pos[0],
new_pos[1]] == 0:
            new_pos[0] = new_pos[0] + 1
            new_pos[1] = new_pos[1] - 1
        else:
            empty = False   # down left

    new_pos = piece_next_to[:]
    return new_pos, direction


def boundary(board, direction, coord):
    pieces_loc = location_of_pieces(board)
    bounded = False
    for pos in pieces_loc:
        if direction == 1:
            if coord[0] + 1 == size or (pos[0] == coord[0] + 1 and pos[1] ==
```

```
coord[1]):
                bounded = True  # up
        elif direction == 2:
            if coord[0] - 1 == -1 or (pos[0] == coord[0] - 1 and pos[1] ==
coord[1]):
                bounded = True  # down
        elif direction == 3:
            if coord[1] - 1 == -1 or (pos[0] == coord[0] and pos[1] == coord[1]
- 1):
                bounded = True  # left
        elif direction == 4:
            if coord[1] + 1 == size or (pos[0] == coord[0] and pos[1] ==
coord[1] + 1):
                bounded = True  # right
        elif direction == 5:
            if coord[0] + 1 == size or coord[1] - 1 == -1 or (pos[0] == coord[0]
+ 1 and pos[1] == coord[1] - 1):
                bounded = True  # up right
        elif direction == 6:
            if coord[0] + 1 == size or coord[1] + 1 == size or (pos[0] ==
coord[0] + 1 and pos[1] == coord[1] + 1):
                bounded = True  # up left
        elif direction == 7:
            if coord[0] - 1 == -1 or coord[1] - 1 == -1 or (pos[0] == coord[0] -
1 and pos[1] == coord[1] - 1):
                bounded = True  # down right
        elif direction == 8:
            if coord[0] - 1 == -1 or coord[1] + 1 == size or (pos[0] == coord[0]
- 1 and pos[1] == coord[1] + 1):
                bounded = True  # down left
        else:
            bounded = False
    return bounded


def evaluate(board, player_piece):
    pieces_loc = location_of_pieces(board)
    piece_count = 0

    # sort pieces
    if player_piece == 1:
        play = pieces_loc[0:3]
        ind1 = np.lexsort((play[:, 1], play[:, 0]))
        play = play[ind1]
    else:
        play = pieces_loc[3:6]
        ind2 = np.lexsort((play[:, 1], play[:, 0]))
        play = play[ind2]

    points = 0
    for ind in range(0, 2):
        for p in range(-1, 2):
            for q in range(-1, 2):

                if play[ind][0] + p == play[ind + 1][0] and play[ind][1] + q ==
play[ind + 1][1]:
                    points = 3  # 2 pieces together with coords like 0 and 1

                    if piece_count == 0 and ind == 0 and (
                            (play[ind][0] + p + p == play[ind + 2][0] and
play[ind][1] + q + q == play[ind + 2][1]) or (
                            play[ind][0] - p == play[ind + 2][0] and
play[ind][1] - q == play[ind + 2][1])):
                        points = 100    # victory
```

```
                        return points

                    if piece_count == 0 and play[ind][0] + p + p != -1 and
play[ind][1] + q + q != -1 and play[ind][0]\
                            + p + p != size and play[ind][1] + q + q != size:
# checks all direction to try and find the third missing piece
                        blank_coord = [play[ind][0] + p + p, play[ind][1] + q +
q]
                        next_to_piece = [play[ind][0] + p, play[ind][1] + q]
                        player = board[play[ind][0], play[ind][1]]
                        pos_piece3, direction = find_piece(board, blank_coord,
player, next_to_piece)

                        if boundary(board, direction, blank_coord):
                            # last move to win
                            points = 10
                            return points

                        elif not np.array_equal(pos_piece3, next_to_piece) \
                                and ((pos_piece3[0] - next_to_piece[0]) == 0 or
(pos_piece3[0] - next_to_piece[0]) ** 2 == 1) \
                                and ((pos_piece3[1] - next_to_piece[1]) == 0 or
(pos_piece3[1] - next_to_piece[1]) ** 2 == 1):
                            points = 5   # 3 pieces in a cluster in different
directions
                            return points

                    if piece_count == 0 and play[ind][0] - p != -1 and
play[ind][1] - q != -1 and play[ind][0] - p != size and play[ind][1] - q !=
size:
                        blank_coord = [play[ind][0] - p, play[ind][1] - q]  #
checks all direction to try and find the third missing piece for the other
extremity
                        next_to_piece = [play[ind][0], play[ind][1]]
                        player = board[play[ind][0], play[ind][1]]
                        pos_piece3, direction = find_piece(board, blank_coord,
player, next_to_piece)
                        if boundary(board, direction, blank_coord):
                            points = 10    # last move to win
                            return points

                        elif not np.array_equal(pos_piece3, next_to_piece) and (
                                (pos_piece3[0] - next_to_piece[0]) == 0 or (
                                pos_piece3[0] - next_to_piece[0]) ** 2 == 1) and
(
                                (pos_piece3[1] - next_to_piece[1]) == 0 or (
                                pos_piece3[1] - next_to_piece[1]) ** 2 == 1):
                            points = 5  # 3 pieces in a cluster in different
directions
                            return points

                    piece_count = piece_count + 1
                    if piece_count == 2:
                        points = 5
                        return points

                elif ind == 0 and play[ind][0] + p == play[ind + 2][0] and
play[ind][1] + q == play[ind + 2][1]:
                    if points == 0:  # 2 pieces together with coords like 0 and
2
                        points = 3
    return points
```

```
def children(board, player):
    pieces_loc = location_of_pieces(board)
    board_children = np.array([np.zeros((size, size))])

    if player == 1:
        play = pieces_loc[0:3]
        ind1 = np.lexsort((play[:, 1], play[:, 0]))
        play = play[ind1]
    else:
        play = pieces_loc[3:6]
        ind2 = np.lexsort((play[:, 1], play[:, 0]))
        play = play[ind2]

    test = board.copy()
    for pos in play:
        for direction in range(1, 9):
            move_piece(test, pos, legal(board, pos, direction))
            if not np.array_equal(test, board):
                test_3d = np.array([test])
                board_children = np.concatenate((board_children, test_3d))
                test = board.copy()
    board_children = np.delete(board_children, 0, 0)

    eva = []
    for child in board_children:
        ev = evaluate(child, 1) - evaluate(child, 2)
        eva.append(ev)

    eva = np.array(eva)
    new_index = eva.argsort()[::-1]
    board_children = board_children[new_index]

    return board_children


def gameover(board):
    result = False

    if evaluate(board, 1) >= 90:
        result = True
    elif evaluate(board, 2) >= 90:
        result = True

    return result


def find_move(init_board, final_board):
    for i in range(0, size):
        for j in range(0, size):
            if init_board[i, j] != 0 and final_board[i, j] == 0:
                init_coord = [i, j]
            elif init_board[i, j] == 0 and final_board[i, j] != 0:
                final_coord = [i, j]
    return init_coord, final_coord


def minimax_ab(board, depth, alpha, beta, maximizingPlayer):
    if depth == 0 or gameover(board):
        ev = evaluate(board, 1) - evaluate(board, 2)
        return board, ev

    best_play = np.zeros((size, size))
    if maximizingPlayer:
        maxEval = -1000000
```

```
        for child in children(board, 1):
            _, eval = minimax_ab(child, depth - 1, alpha, beta, False)

            if eval > maxEval:
                maxEval = eval
                best_play = child

            alpha = max(eval, alpha)
            if beta < alpha:
                break

        return best_play, maxEval

    else:
        minEval = 1000000
        for child in children(board, 2):
            _, eval = minimax_ab(child, depth - 1, alpha, beta, True)

            if eval < minEval:
                minEval = eval
                best_play = child

            beta = min(beta, eval)
            if beta < alpha:
                break

        return best_play, minEval


def human_computer(btn, row, col):
    global click
    global origin
    global selected
    global turn
    global winner
    global level
    global b
    if not gameover(b):

        if turn % 2 == 0:
            jog = 1

        pieces_loc = location_of_pieces(b)

        if jog == 1:
            play = pieces_loc[0:3]
            ind1 = np.lexsort((play[:, 1], play[:, 0]))
            play = play[ind1]

            for pos in play:
                if click == 1 and np.array_equal(pos, [row, col]) and selected
is False:

                    if jog == 1:
                        bo[row][col] = 4

                    for dir in range(1, 9):
                        legal_coord = legal(b, [row, col], dir)
                        if not np.array_equal(legal_coord, [row, col]):
                            bo[legal_coord[0]][legal_coord[1]] = 3
                    click = 2
                    selected = True
                    origin = [row, col]
                    break
```

```
                elif click == 2 and np.array_equal(origin, [row, col]) and
selected is True:
                    if jog == 1:
                        bo[row][col] = 1

                    for dir in range(1, 9):
                        legal_coord = legal(b, [row, col], dir)
                        if not np.array_equal(legal_coord, [row, col]):
                            bo[legal_coord[0]][legal_coord[1]] = 0
                    click = 1
                    selected = False
                    origin = [-1, -1]
                    break

            if selected and bo[row][col] == 3:

                if jog == 1:
                    bo[origin[0]][origin[1]] = 1

                move_piece_gui(bo, origin, [row, col])
                move_piece(b, origin, [row, col])
                turn = turn + 1
                bo.print('A.I. Engine to move!', 'Heuristic Evaluation - Player
1:', evaluate(b, 1), '\nPlayer 2:',
                         -evaluate(b, 2), ' Total Evaluation: ', evaluate(b, 1)
- evaluate(b, 2))
                selected = False
                click = 1
                for i in range(0, size):
                    for j in range(0, size):
                        if bo[i][j] == 3:
                            bo[i][j] = 0

                if not gameover(b):
                    if turn % 2 == 0:
                        jog = 1
                    else:
                        jog = 2
                        if level == "Easy":
                            best_p, eval = minimax_ab(b, 1, -100000, 100000,
False)

                        elif level == "Medium":
                            best_p, eval = minimax_ab(b, 2, -100000, 100000,
False)

                        else:
                            best_p, eval = minimax_ab(b, 3, -100000, 100000,
False)

                        coord_init, coord_final = find_move(b, best_p)
                        move_piece_gui(bo, coord_init, coord_final)
                        move_piece(b, coord_init, coord_final)
                        bo.print('Player1 to move!  Predicted «', level, '»
evaluation:', eval, '\nCurrent Board Eval:',
                                 evaluate(b, 1) - evaluate(b, 2))

                        turn = turn + 1

        if gameover(b):
            if jog == 1:
                text = ' ' * 27 + "Congratulations, Black wins!"
                title = "Congratulations"
            else:
```

```
                text = ' ' * 27 + "You will beat it next time!"
                title = "Retry?"

            menu = eg.buttonbox(msg=text, title=title, choices=("Play again",
"Choose level", "Quit"))
            if menu == "Play again":
                b = np.zeros((size, size))
                init(b)
                array_to_gui(b, bo)
                turn = 0
                bo.print('          Black (Player 1) Moves First')
            elif menu == "Choose level":
                level = eg.buttonbox(
                    msg=f"\n{' ' * 27}Human goes First \n\n{' ' * 27}What is the
A.I. Level?",
                    title="Engine Level", choices=("Easy", "Medium", "Hard"))
                b = np.zeros((size, size))
                init(b)
                array_to_gui(b, bo)
                turn = 0
                bo.print('          Black (Player 1) Moves First')
            else:
                bo.close()

            '''if eg.ccbox(msg=text, title=title, choices=("Play again",
"Quit!")):
                b = np.zeros((size, size))
                init(b)
                array_to_gui(b, bo)
                turn = 0
                bo.print('          Black (Player 1) Moves First')
            else:
                bo.close()'''


def start_game(size):
    global bo
    bo = Board(size, size)
    array_to_gui(b, bo)
    bo.title = "Neutreeko"
    bo.cell_size = 69
    bo.cell_color = "LightSeaGreen"
    bo.margin = 15
    bo._margin_color = "#001f35"
    bo.create_output(background_color="#001f35", color="#ffff70", font_size=12)
    bo.print('          Black (Player 1) Moves First')


if __name__ == "__main__":
    size = 5
    welcome = "hello"
    rules = True
    mode = "let's see"
    while not (welcome == "PLAY!" or not rules):
        welcome = eg.buttonbox(msg=f"\n\n\n\n{' ' * 31}Welcome to Neutreeko",
title="Neutreeko", choices=("PLAY!", "Game Rules"))
        if welcome == "PLAY!":
            b = np.zeros((size, size))
            init(b)
            click = 1
            origin = [-1, -1]
            selected = False
            turn = 0
            level = eg.buttonbox(
```

```
                msg=f"\n{' ' * 27}Human goes First \n\n{' ' * 27}What is the
A.I. Level?",
                title="Engine Level", choices=("Easy", "Medium", "Hard"))
            start_game(size)
            bo.on_mouse_click = human_computer
            bo.show()
        elif welcome == "Game Rules":
            rules = eg.ccbox(
                msg="-Movement: A piece slides orthogonally or diagonally until
stopped by \nan occupied square or the border of the board. Black always moves
first.\n\n-Objective: To get three in a row, orthogonally or diagonally. The row
must be connected.",
                title="Neutreeko Game Rules", choices=("Go back to Main Menu",
"Exit"))
        else:
            break
```

3.1.2 Приклади роботи

На рисунках 3.1 - 3.4 показані приклади роботи програми.

Рисунок 3.1 – Початкове меню програми

Рисунок 3.2 – Вибір важкості

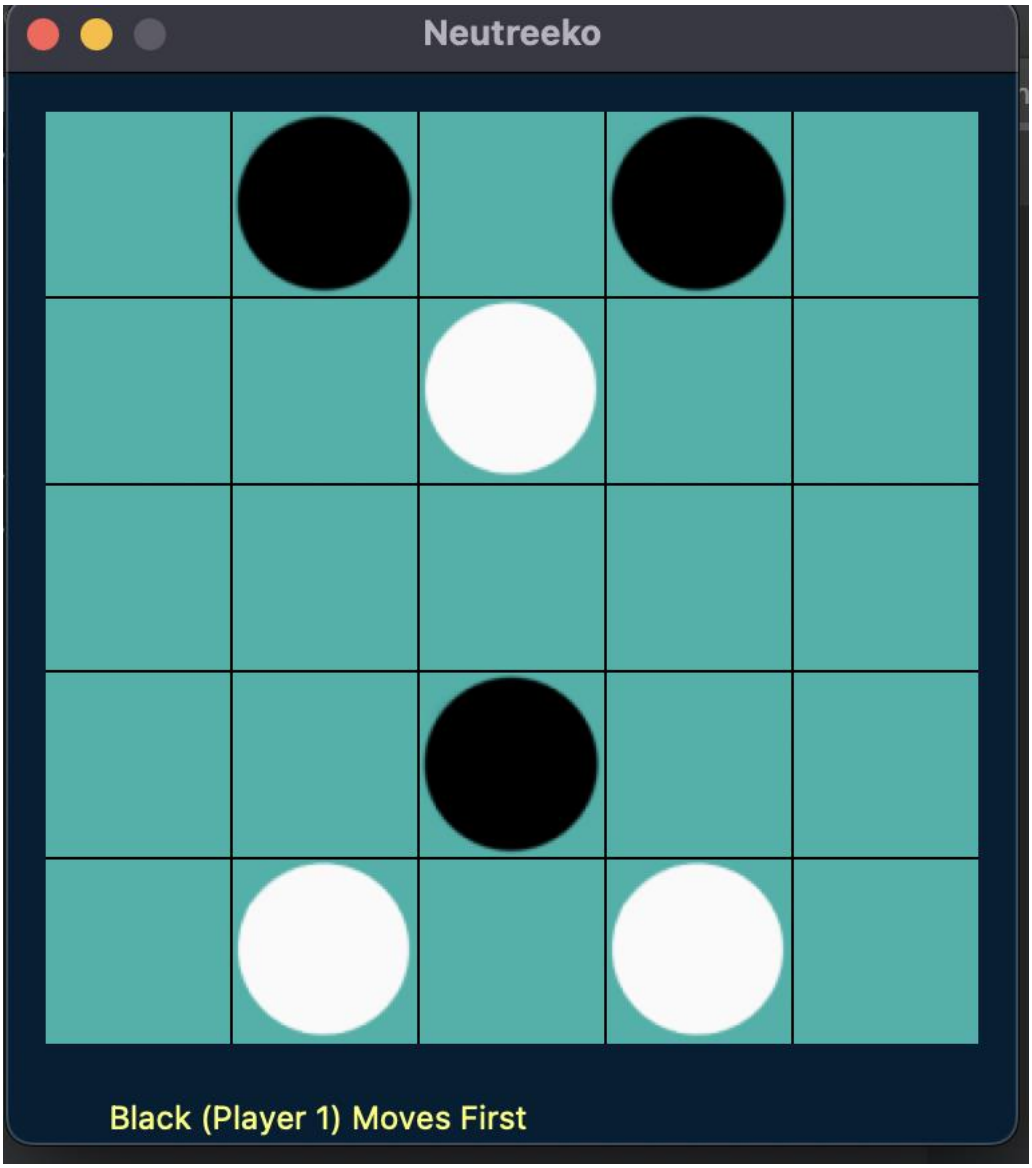

Рисунок 3.3 – Ігра (за чорних)

Рисунок 3.4 – Меню після гри

ВИСНОВОК

В рамках даної лабораторної роботи я реалізував алгоритм альфа-бета відсічення для гри Neutreeko. Для цього були створено алгоритм що обраховувad цінність кожного вузла розстановки шашок на дошці. Для гри було створено GUI за допомогою бібліотеки easygui.

КРИТЕРІЇ ОЦІНЮВАННЯ

При здачі лабораторної роботи до 25.12.2022 включно максимальний бал дорівнює – 5. Після 25.12.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

- програмна реалізація – 95%;
- висновок – 5%.

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