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-15 Шабанов Метін Шаміль огли* _____
(шифр, прізвище, ім'я, по батькові)

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

Київ 2022

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 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.

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

27	Лудо http://www.iggamecenter.com/info/ru/ludo.html	З елементами випадковості

3 ВИКОНАННЯ

3.1 Програмна реалізація алгоритму

3.1.1 Вихідний код

```
import copy
import random

from game_state import State


class MinimaxNode:
    def __init__(self, dice_roll, players, current_player_index, level,
is_main_state=False):
        self.head = State(players, current_player_index)
        self.current_player_index = current_player_index % 4
        self.is_main_state = is_main_state
        self.dice_roll = dice_roll
        self.level = level
        self.children = []
        if self.level < 3:
            self.__define_child_states()

    def __define_child_states(self):
        next_player_index = (self.current_player_index + 1) % 4
        active_pieces =
self.head.players[self.current_player_index].check_active_pieces(self.dice_roll)

        if self.is_main_state:
            for piece_index in range(len(active_pieces)):
                new_board = copy.deepcopy(self.head.players)

new_board[self.current_player_index].check_active_pieces(self.dice_roll)[piece_i
ndex].move(self.dice_roll)
                self.children.append(MinimaxNode(self.dice_roll, new_board,
next_player_index, self.level + 1))
            if self.dice_roll == 6 and
self.head.players[self.current_player_index].check_inner_pieces:
                new_board = copy.deepcopy(self.head.players)
                unmoved =
new_board[self.current_player_index].find_unmoved_piece()
                if unmoved is not None:
                    unmoved.take_out()
                    self.children.append(MinimaxNode(6, new_board,
next_player_index, self.level + 1))
        else:
            for i in range(1, 7):
                for piece_index in range(len(active_pieces)):
                    new_board = copy.deepcopy(self.head.players)

new_board[self.current_player_index].check_active_pieces(self.dice_roll)[piece_i
ndex].move(self.dice_roll)
                    self.children.append(MinimaxNode(i, new_board,
next_player_index, self.level + 1))
            if self.dice_roll == 6 and
self.head.players[self.current_player_index].check_inner_pieces:
                new_board = copy.deepcopy(self.head.players)
                unmoved =
new_board[self.current_player_index].find_unmoved_piece()
```

```
                if unmoved is not None:
                    unmoved.take_out()
                    self.children.append(MinimaxNode(6, new_board,
next_player_index, self.level + 1))

    def find_best_solution(self):
        for level_1 in self.children:
            sum1 = 0
            for level_2 in level_1.children:
                sum2 = 0
                for level_3 in level_2.children:
                    sum2 += level_3.head.state_value
                if len(level_2.children) != 0:
                    level_2.head.state_value = sum2 / len(level_2.children)
                if level_2.head.state_value is not None:
                    sum1 += level_2.head.state_value
            if len(level_1.children) != 0:
                level_1.head.state_value = sum1 / len(level_1.children)
        return self.__choose_max_state()

    def __choose_max_state(self):
        max_state = self.children[0]
        for child in self.children:
            if child.head.state_value > max_state.head.state_value:
                max_state = child
        return max_state
```

3.1.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

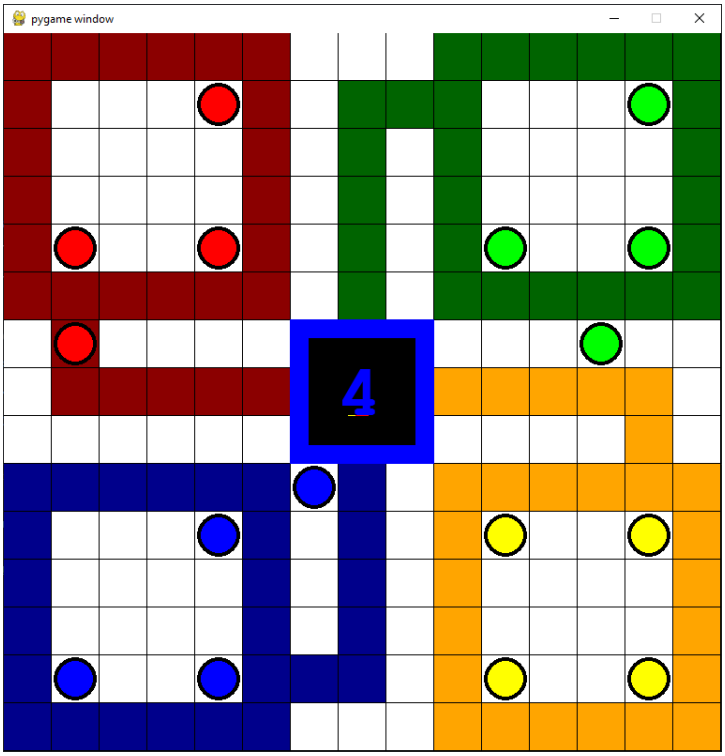

Рисунок 3.1 – приклад роботи програми

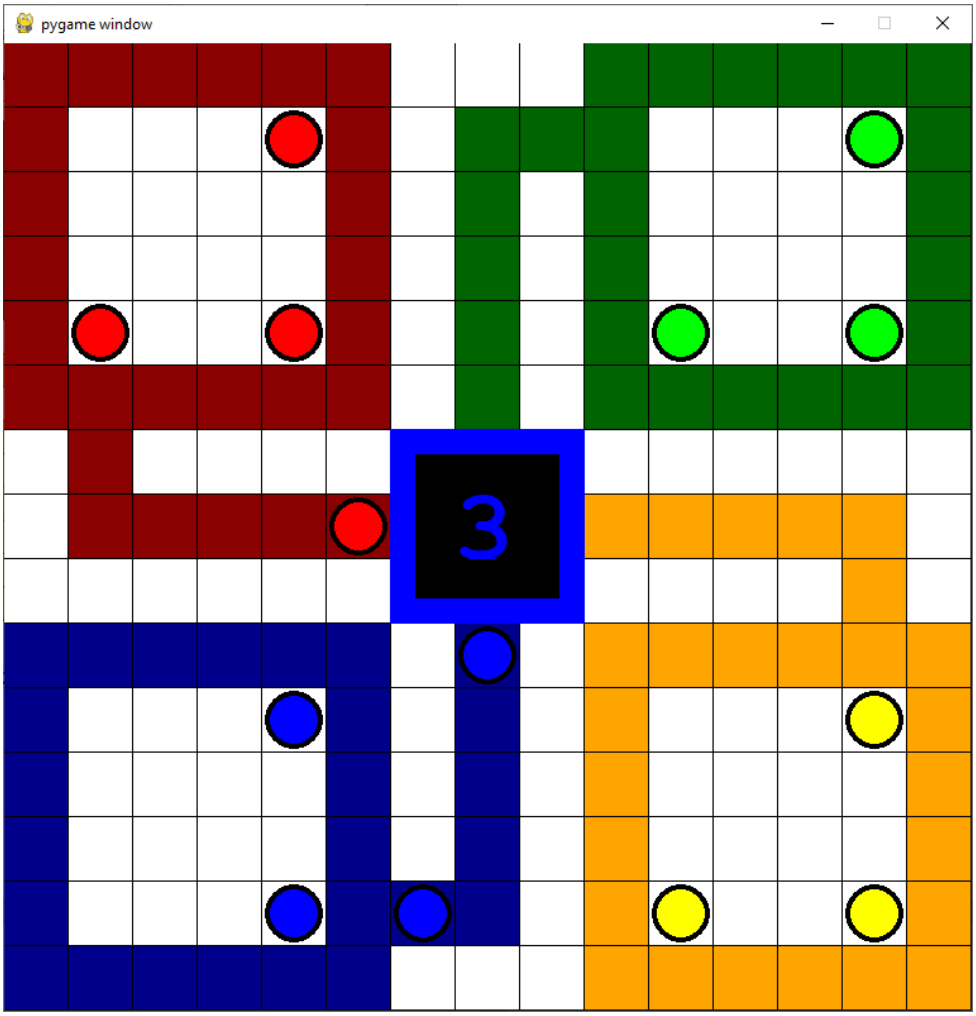

Рисунок 3.2 – приклад роботи програми

ВИСНОВОК

В рамках даної лабораторної роботи було розглянуто алгоритм мінімакс, у вигляді формалізаціїї під візуальний ігровий додаток з елементами випадковості, який називається “Лудо”. Для досягнення мети було внесено деякі модифікації в алгоритм, що стосуються випадкового визначення наступного ходу противника алгоритму – тобто користувача. Це дозволило підвищити ефективність визначення оптимального рішення, і покращило швидкість роботи програми.

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).