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-13 Паламарчук Олександр* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Сопов Олександр* _____
(прізвище, ім'я, по батькові)

Київ 2022

Зміст

Лабораторна робота №5

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

Варіант 21

1. Мета

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	Индійські 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

```
package org.example.controller;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.FXMLLoader;
import javafx.scene.Node;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.control.TextField;
import javafx.stage.Stage;

import java.io.IOException;
public class HelloController {

    @FXML
```

```
    private TextField usernameTextField;

    @FXML
    protected void onGoButtonClick(ActionEvent event) throws IOException {
        String username = usernameTextField.getText();
        if (username.isBlank())
            username = "user";

        FXMLLoader loader = new
FXMLLoader(HelloController.class.getResource("/org.example/main-scene.fxml"));
        Parent root = loader.load();

        MainController controller = loader.getController();
        controller.setUsername(username);

        Stage stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
        Scene scene = new Scene(root);

        stage.setScene(scene);
        stage.show();
    }
}
package org.example.controller;

import javafx.fxml.FXML;
import javafx.geometry.Pos;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.image.ImageView;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.example.model.bot.BotLogic;
import org.example.model.decorator.ClickDecorator;
import org.example.model.game.Entry;
import org.example.model.game.GameBoard;
import org.example.model.game.GameSession;
import org.example.model.game.Player;
import org.example.model.game.card.Card;
import org.example.model.game.card.Rank;

import java.util.*;
import java.util.stream.Collectors;

import static org.example.model.Utils.*;

public class MainController {

    private static final Logger log =
LogManager.getLogger(MainController.class);

    @FXML
    private HBox userCardsHBox;
    @FXML
    private HBox botCardsHBox;
    @FXML
    private AnchorPane botTrumpAnchorPane;
    @FXML
    private StackPane botDeckStackPane;
    @FXML
    private AnchorPane userTrumpAnchorPane;
    @FXML
    private StackPane userDeckStackPane;
    @FXML
    private AnchorPane generalTrumpAnchorPane;
```

```
    @FXML
    private StackPane generalDeckStackPane;
    @FXML
    private Button startButton;
    @FXML
    private Button finishButton;
    @FXML
    private Button takeButton;
    @FXML
    private Button throwButton;
    @FXML
    private Button transferButton;
    @FXML
    private HBox currCardsHBox;
    @FXML
    private Label nextStepLabel;
    @FXML
    private Label winnerLabel;

    private GameSession gameSession;
    private GameBoard gameBoard;
    private String username;
    private LinkedList<Card> chosenCards;

    public void setUsername(String username) {
        this.username = username;
    }

    @FXML
    protected void onStartButtonClick() {
        setDisabled(startButton, true);
        initGameSession();
        botStep();
        renderBoard();
    }

    private void botStep() {
        if (!checkFinalState()) {
            BotLogic botLogic = new BotLogic(gameBoard);
            GameBoard gameBoard = botLogic.makeDecision();
            this.gameSession.setGameBoard(gameBoard);
            this.gameBoard = gameBoard;
        }
    }

    @FXML
    protected void onTakeButtonClick() {
        Set<Card> playingCards = getPlayingCards(gameBoard);

        Player user = gameBoard.getUser();
        user.getHand().addAll(playingCards);

        gameBoard.getPlayingEntries().clear();
        gameBoard.setUserNext(false);
        gameBoard.setUserAttacking(false);
        giveCards(gameBoard.getBot(), gameBoard);

        botStep();
        renderBoard();
    }

    @FXML
    protected void onThrowButtonClick() {
        log.trace("OnThrowButtonClick in invoked");
        if (!chosenCards.isEmpty()) {
            log.debug("ChosenCards in not empty");
```

```
            List<Entry> playingEntries = gameBoard.getPlayingEntries();

            if (gameBoard.isUserAttacking()) {

                boolean stepIsValid;
                if (playingEntries.isEmpty()) {
                    stepIsValid = checkIfChosenCardsSameRank();
                } else {
                    stepIsValid =
checkIfChosenCardRanksSameAsPlayingCardRanks();
                }

                log.debug("Step is valid {}", stepIsValid);
                if (stepIsValid) {
                    gameBoard.setUserNext(false);
                    putChosenCardsOnTable();
                    renderBoard();
                    botStep();
                }

                chosenCards.clear();
                log.debug("Render board");
                renderBoard();
            } else {
                Card hittingCard = chosenCards.get(0);
                chosenCards.clear();

                Optional<Entry> entryOptional = playingEntries.stream()
                        .filter(entry -> entry.getHitting().isEmpty())
                        .filter(entry -> canHit(hittingCard,
entry.getDefending().get(0),
                                gameBoard, gameBoard.getUser()))
                        .findFirst();

                if (entryOptional.isPresent()) {
                    Entry entry = entryOptional.get();
                    entry.getHitting().add(hittingCard);

                    log.debug("Hitting card {}", hittingCard);

                    Player user = gameBoard.getUser();
                    user.getHand().remove(hittingCard);

                    log.debug("User hand {}", gameBoard.getUser().getHand());
                    boolean isNotHitEntry =
gameBoard.getPlayingEntries().stream()
                            .anyMatch(e -> e.getHitting().isEmpty());
                    if (!isNotHitEntry) {
                        gameBoard.setUserNext(false);
                        botStep();
                    }
                    renderBoard();
                }
            }
        }
        log.trace("OnThrowButtonClick is terminated");
    }

    @FXML
    protected void onFinishButtonClick() {
        gameBoard.getPlayingEntries().clear();
        giveCards(gameBoard.getUser(), gameBoard);
        giveCards(gameBoard.getBot(), gameBoard);
        gameBoard.setUserNext(false);
        gameBoard.setUserAttacking(false);
        botStep();
```

```
        renderBoard();
    }

    @FXML
    protected void onTransferButtonClick() {
        if (transferIsAllowed(gameBoard, chosenCards, gameBoard.getBot())) {
            doTransfer(gameBoard, chosenCards, gameBoard.getUser());
            botStep();
            renderBoard();
        }
    }

    private void putChosenCardsOnTable() {
        ArrayList<Card> userCards = gameBoard.getUser().getHand();
        List<Entry> playingEntries = gameBoard.getPlayingEntries();

        userCards.removeAll(chosenCards);
        for (Card chosenCard : chosenCards) {
            log.debug("Create entry for defending card {}", chosenCard);
            Entry entry = new Entry();
            entry.getDefending().add(chosenCard);
            playingEntries.add(entry);
        }
    }

    private boolean checkIfChosenCardsSameRank() {
        Card firstCard = chosenCards.get(0);
        Rank firstCardRank = firstCard.getRank();

        long numberOfJokers = chosenCards.stream()
                .map(Card::getRank)
                .filter(rank -> rank.equals(Rank.JOKER))
                .count();

        if (numberOfJokers > 1) {
            return false;
        }

        if (numberOfJokers > 0 && chosenCards.size() > 1) {
            return false;
        }

        for (Card chosenCard : chosenCards) {
            if (!chosenCard.getRank().equals(firstCardRank)) {
                return false;
            }
        }

        return true;
    }

    private boolean checkIfChosenCardRanksSameAsPlayingCardRanks() {
        Set<Rank> playingCardRanks = getPlayingCardRanks().stream()
                .filter(rank -> !rank.equals(Rank.JOKER))
                .collect(Collectors.toSet());

        for (Card chosenCard : chosenCards) {
            if (!playingCardRanks.contains(chosenCard.getRank())) {
                return false;
            }
        }

        return true;
    }

    private boolean checkFinalState() {
```

```
        boolean allBeaten = gameBoard.getPlayingEntries().stream()
                .noneMatch(entry -> entry.getHitting().isEmpty());

        if (gameBoard.getPlayingEntries().isEmpty() || allBeaten) {
            boolean userHandIsEmpty = gameBoard.getUser().getHand().isEmpty();
            boolean botHandIsEmpty = gameBoard.getBot().getHand().isEmpty();
            if (userHandIsEmpty || botHandIsEmpty) {
                if (userHandIsEmpty && botHandIsEmpty) {
                    endGame("draw");
                } else if (userHandIsEmpty) {
                    endGame("user");
                } else {
                    endGame("bot");
                }
                return true;
            }
        }

        return false;
    }

    private void endGame(String result) {
        setDisabled(finishButton, true);
        setDisabled(takeButton, true);
        setDisabled(throwButton, true);
        setDisabled(transferButton, true);
        setDisabled(startButton, false);
        if (result.equals("user")) {
            winnerLabel.setText("Winner is " + gameBoard.getUser().getName());
        } else if (result.equals("bot")) {
            winnerLabel.setText("Winner is " + gameBoard.getBot().getName());
        } else {
            winnerLabel.setText("Drawn game");
        }
    }

    private Set<Rank> getPlayingCardRanks() {
        return getPlayingCards(gameBoard).stream()
                .map(Card::getRank)
                .collect(Collectors.toSet());
    }

    private void initGameSession() {
        winnerLabel.setText("");

        Player user = Player.newPlayer(username),
                bot = Player.newPlayer("bot");

        gameSession = GameSession.newGameSession(user, bot);
        gameSession.start();

        this.chosenCards = new LinkedList<>();
        this.gameBoard = gameSession.getGameBoard();
    }

    private void setDisabled(Button button, boolean flag) {
        button.setDisable(flag);
    }

    private void renderBoard() {
        clearBoard();
        renderBotState();
        renderUserState();
        renderGeneralState();
        renderNextPlayer();
        renderButtons();
```

```
    }

    private void renderButtons() {
        if (gameBoard.isUserNext()) {
            if (gameBoard.isUserAttacking()) {
                setDisabled(finishButton, gameBoard.getPlayingEntries().size()
== 0);
                setDisabled(takeButton, true);
                setDisabled(throwButton, false);
                setDisabled(transferButton, true);
            } else {
                setDisabled(finishButton, true);
                setDisabled(takeButton, false);
                setDisabled(throwButton, false);
                setDisabled(transferButton, false);
            }
        } else {
            setDisabled(finishButton, true);
            setDisabled(takeButton, true);
            setDisabled(throwButton, true);
            setDisabled(transferButton, true);
        }
    }

    private void clearBoard() {
        clearBotState();
        clearUserState();
        clearGeneralState();
    }

    private void clearBotState() {
        clear(botDeckStackPane);
        clear(botTrumpAnchorPane);
        clear(botCardsHBox);
    }

    private void clearUserState() {
        clear(userDeckStackPane);
        clear(userTrumpAnchorPane);
        clear(userCardsHBox);
    }

    private void clearGeneralState() {
        clear(generalDeckStackPane);
        clear(generalTrumpAnchorPane);
        clear(currCardsHBox);
    }

    private void renderNextPlayer() {
        String text = "Next: %s";
        if (gameBoard.isUserNext()) {
            nextStepLabel.setTextFill(Color.GREEN);
            nextStepLabel.setText(String.format(text,
gameBoard.getUser().getName()));
        } else {
            nextStepLabel.setTextFill(Color.RED);
            nextStepLabel.setText(String.format(text,
gameBoard.getBot().getName()));
        }
    }

    private void clear(Pane pane) {
        pane.getChildren().clear();
    }

    private void renderBotState() {
```

```
        Player bot = gameBoard.getBot();

        renderBotHand();
        renderPlayerTrump(bot, botTrumpAnchorPane);
        renderPlayerDeck(bot, botDeckStackPane);
    }

    private void renderUserState() {
        Player user = gameBoard.getUser();

        renderUserHand();
        renderPlayerTrump(user, userTrumpAnchorPane);
        renderPlayerDeck(user, userDeckStackPane);
    }

    private void renderGeneralState() {
        renderTrump(gameBoard.getGeneralTrump(), generalTrumpAnchorPane);
        renderDeck(gameBoard.getGeneralDeck(), generalDeckStackPane);
        renderPlayingCards();
    }

    private void renderBotHand() {
        Player bot = gameBoard.getBot();
        sortByRank(bot.getHand()).forEach(card -> Card.render(card,
botCardsHBox)); // todo: change card to null inside render
    }

    private void renderPlayerTrump(Player player, Pane pane) {
        renderTrump(player.getTrump(), pane);
    }

    private void renderPlayerDeck(Player player, Pane pane) {
        renderDeck(player.getDeck(), pane);
    }

    private void renderTrump(Card trump, Pane pane) {
        if (Objects.nonNull(trump)) {
            trump.render(pane);
        }
    }

    private void renderDeck(Stack<Card> deck, Pane pane) {
        if (!deck.isEmpty()) {
            Card.render(null, pane);
        }
    }

    private void renderUserHand() {
        if (gameBoard.isUserNext()) {
            renderActiveUserHand();
        } else {
            renderInactiveUserHand();
        }
    }

    private void renderActiveUserHand() {
        Player user = gameBoard.getUser();
        sortByRank(user.getHand()).forEach(card -> {
            ImageView imageView = card.toImageView();
            ClickDecorator clickDecorator = new ClickDecorator(imageView,
chosenCards, card, gameBoard);
            userCardsHBox.getChildren().add(clickDecorator.self());
        });
    }

    private void renderInactiveUserHand() {
```

```
        Player user = gameBoard.getUser();
        sortByRank(user.getHand()).forEach(card -> card.render(userCardsHBox));
    }

    private List<Card> sortByRank(ArrayList<Card> cards) {
        return cards.stream()
                .sorted(Comparator.comparing(card -> card.getRank().getValue()))
                .collect(Collectors.toList());
    }

    private void renderPlayingCards() {

        List<Entry> entries = gameBoard.getPlayingEntries();

        entries.forEach(entry -> {
            LinkedList<Card> hittingCards = entry.getHitting();
            LinkedList<Card> defendingCards = entry.getDefending();

            StackPane pane = new StackPane();
            if (hittingCards.isEmpty()) {
                setSize(pane, 150, 65);
                appendCard(defendingCards.get(0), pane, Pos.CENTER);
            } else if (hittingCards.size() == 1) {
                if (defendingCards.size() == 1) {
                    setSize(pane, 150, 65);
                    appendCard(defendingCards.get(0), pane, Pos.TOP_CENTER);
                    appendCard(hittingCards.get(0), pane, Pos.BOTTOM_CENTER);
                } else if (defendingCards.size() == 2) {
                    setSize(pane, 150, 130);
                    appendCard(defendingCards.get(0), pane, Pos.TOP_LEFT);
                    appendCard(defendingCards.get(1), pane, Pos.TOP_RIGHT);
                    appendCard(hittingCards.get(0), pane, Pos.BOTTOM_CENTER);
                } else if (defendingCards.size() == 3) {
                    setSize(pane, 150, 130);
                    appendCard(defendingCards.get(0), pane, Pos.TOP_LEFT);
                    appendCard(defendingCards.get(1), pane, Pos.TOP_CENTER);
                    appendCard(defendingCards.get(2), pane, Pos.TOP_RIGHT);
                    appendCard(hittingCards.get(0), pane, Pos.BOTTOM_CENTER);
                }
            } else if (hittingCards.size() == 2) {
                setSize(pane, 150, 130);
                appendCard(defendingCards.get(0), pane, Pos.TOP_CENTER);
                appendCard(hittingCards.get(0), pane, Pos.BOTTOM_LEFT);
                appendCard(hittingCards.get(1), pane, Pos.BOTTOM_RIGHT);
            }
            currCardsHBox.getChildren().add(pane);
        });
    }

    private void setSize(StackPane pane, double height, double width) {
        pane.setPrefWidth(width);
        pane.setPrefHeight(height);
    }

    private void appendCard(Card card, StackPane pane, Pos position) {
        ImageView cardImageView = card.toImageView();
        StackPane.setAlignment(cardImageView, position);
        pane.getChildren().add(cardImageView);
    }
}
package org.example.model.bot;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.example.model.Utils;
import org.example.model.game.Entry;
```

```java
import org.example.model.game.GameBoard;
import org.example.model.game.Player;
import org.example.model.game.card.Card;
import org.example.model.game.card.Rank;

import java.util.*;
import java.util.stream.Collectors;

import static org.example.model.Utils.*;
import static org.example.model.Utils.giveCards;

public class BotLogic {

    private static final Logger log = LogManager.getLogger(BotLogic.class);

    private GameBoard currState;
    private List<GameBoard> nextStates;
    private final Comparator<GameBoard> gameBoardComparator =
Comparator.comparing(state -> {
        PointCounter pointCounter = new PointCounter(state);
        return pointCounter.getPoints(state.getBot());
    });

    public BotLogic(GameBoard currState) {
        this.currState = Objects.requireNonNull(currState);
        this.nextStates = new LinkedList<>();
    }

    public GameBoard makeDecision() {
        log.trace("MakeDecision is invoked");
        if (isBotNext()) {
            log.debug("Bot is next");
            if (isBotAttacking()) {
                log.debug("Bot is attacking");
                attack();
            } else {
                log.debug("Bot is defending");
                defend();
            }
        }
        log.trace("MakeDecision is terminated");
        return currState;
    }

    private void attack() {
        log.trace("Attack is invoked");
        if (tableIsEmpty()) {
            log.debug("Table is empty");
            throwCards();
        } else {
            log.trace("Table is not empty");
            throwUpCards();
        }
    }

    private void defend() {
        this.nextStates = new LinkedList<>();
        List<Entry> playingEntries = currState.getPlayingEntries();
        List<Entry> entriesToHit = playingEntries.stream()
                .filter(entry -> entry.getHitting().isEmpty())
                .toList();

        Player bot = currState.getBot();
        ArrayList<Card> hand = bot.getHand();

        Map<Entry, List<Card>> map = new HashMap<>();
```

```
        for (Entry entryToHit : entriesToHit) {
            List<Card> cardsCanBeat = new ArrayList<>();
            for (Card card : hand) {
                if (canHit(card, entryToHit.getDefending().get(0), currState,
currState.getBot())) {
                    cardsCanBeat.add(card);
                }
            }

            map.put(entryToHit, cardsCanBeat);
        }

        List<Entry> entries = new LinkedList<>();
        for (Map.Entry<Entry, List<Card>> entry : map.entrySet()) {
            List<Card> value = entry.getValue();
            for (Card card : value) {
                Entry keyCopy = entry.getKey().copy();
                keyCopy.getHitting().add(card);
                entries.add(keyCopy);
            }
        }

        List<List<Entry>> allowableCombinations = new LinkedList<>();
        for (int i = entriesToHit.size(); i < entries.size(); i++) {
            List<List<Entry>> allCombinationsWithSizeI = combinations(entries,
i);

            for (List<Entry> combination : allCombinationsWithSizeI) {
                boolean isAllowed = true;
                Set<Card> cards = new HashSet<>();
                for (Entry entry : combination) {
                    LinkedList<Card> allCards = new
LinkedList<>(entry.getHitting());
                    allCards.addAll(entry.getDefending());
                    for (Card card : allCards) {
                        if (cards.contains(card)) {
                            isAllowed = false;
                            break;
                        } else {
                            cards.add(card);
                        }
                    }
                    if (!isAllowed) {
                        break;
                    }
                }
                if (isAllowed) {
                    allowableCombinations.add(combination);
                }
            }
        }

        for (List<Entry> combination : allowableCombinations) {
            GameBoard copyState = currState.copy();
            copyState.setUserNext(true);
            Player copyStateBot = copyState.getBot();
            combination.stream()
                    .flatMap(e -> e.getHitting().stream())
                    .forEach(c -> copyStateBot.getHand().remove(c));
            List<Entry> copyStatePlayingEntries = copyState.getPlayingEntries();
            for (Entry entry : combination) {
                copyStatePlayingEntries.stream()
                        .filter(e ->
e.getDefending().equals(entry.getDefending()))
                        .findFirst()
                        .ifPresent(e -> {
```

```
                            e.getHitting().addAll(entry.getHitting());
                        });
            }
            this.nextStates.add(copyState);
        }

        for (Card card : hand) {
            LinkedList<Card> chosenCards = new LinkedList<>(List.of(card));
            if (transferIsAllowed(currState, chosenCards, currState.getUser()))
{
                GameBoard stateCopy = currState.copy();
                doTransfer(stateCopy, chosenCards, stateCopy.getBot());
                nextStates.add(stateCopy);
            }
        }

        GameBoard takeCardsState = getTakeCardsState();
        if (nextStates.isEmpty()) {
            this.currState = takeCardsState;
        } else {
            setBestNextState();
        }
    }

    private GameBoard getTakeCardsState() {
        GameBoard altState = currState.copy();
        altState.setUserNext(true);
        altState.setUserAttacking(true);
        List<Card> cards = altState.getPlayingEntries().stream()
                .flatMap(entry -> entry.getHitting().stream())
                .collect(Collectors.toList());
        altState.getPlayingEntries().stream()
                .flatMap(entry -> entry.getDefending().stream())
                .forEach(cards::add);
        altState.getBot().getHand().addAll(cards);
        altState.getPlayingEntries().clear();
        giveCards(altState.getUser(), altState);
        return altState;
    }

    private boolean isBotAttacking() {
        return !currState.isUserAttacking();
    }

    private boolean isBotNext() {
        return !currState.isUserNext();
    }

    private boolean tableIsEmpty() {
        return currState.getPlayingEntries().isEmpty();
    }

    private void throwCards() {
        log.trace("throwCards is invoked");
        this.nextStates = new LinkedList<>();

        Player bot = currState.getBot();
        Map<Rank, List<Card>> cardsToThrow = groupCardsByRank(bot.getHand());
        int maxCardCountToThrowUp = getMaxCardCountToThrowUp();
        generateCombinationsAndAddToNextStates(cardsToThrow.values(),
maxCardCountToThrowUp);

        setBestNextState();
        log.trace("throwCards is terminated");
    }
```

```
    private void throwUpCards() {
        log.trace("throwUpCards is invoked");
        this.nextStates = new LinkedList<>();

        Player bot = currState.getBot();
        ArrayList<Card> hand = bot.getHand();
        Set<Card> playingCards = getPlayingCards(currState).stream()
                .filter(card -> !card.getRank().equals(Rank.JOKER))
                .collect(Collectors.toSet());
        List<Card> cardsToThrowUp = getCardsToThrowUp(hand, playingCards);
        int maxCardCountToThrowUp = getMaxCardCountToThrowUp();
        generateCombinationsAndAddToNextStates(cardsToThrowUp,
maxCardCountToThrowUp);

        GameBoard finishAttackState = getFinishAttackState();
        nextStates.add(finishAttackState);

        setBestNextState();
        log.trace("throwUpCards is terminated");
    }

    private void generateCombinationsAndAddToNextStates(List<Card> cards, int
maxCardCount) {
        for (int i = 1; i <= cards.size() && i <= maxCardCount; i++) {
            List<List<Card>> combinations = combinations(cards, i);
            for (List<Card> combination : combinations) {
                GameBoard nextState = createNextState(combination, combination);
                nextStates.add(nextState);
            }
        }
    }

    private void generateCombinationsAndAddToNextStates(Collection<List<Card>>
cards, int maxCardCount) {
        for (List<Card> cardsToCombine : cards) {
            generateCombinationsAndAddToNextStates(cardsToCombine,
maxCardCount);
        }
    }

    private GameBoard createNextState(List<Card> cardsToRemove, List<Card>
cardsToAddToEntries) {
        GameBoard nextState = currState.copy();
        ArrayList<Card> nextHand = nextState.getBot().getHand();
        nextHand.removeAll(cardsToRemove);
        for (Card card : cardsToAddToEntries) {
            Entry entry = new Entry();
            entry.getDefending().add(card);
            nextState.getPlayingEntries().add(entry);
        }
        nextState.setUserNext(true);
        return nextState;
    }

    private Map<Rank, List<Card>> groupCardsByRank(List<Card> cards) {
        return cards.stream().collect(Collectors.groupingBy(Card::getRank,
Collectors.toList()));
    }

    private List<Card> getCardsToThrowUp(List<Card> hand, Set<Card>
playingCards) {
        Set<Rank> botHandRankSet = getRankSet(hand);
        Set<Rank> playingCardRankSet = getRankSet(playingCards);
        Set<Rank> intersection = getIntersection(botHandRankSet,
playingCardRankSet);
        return hand.stream().filter(card ->
```

```
intersection.contains(card.getRank())).toList();
    }

    private Set<Rank> getRankSet(Collection<Card> cards) {
        return cards.stream().map(Card::getRank).collect(Collectors.toSet());
    }

    private Set<Rank> getIntersection(Set<Rank> set1, Set<Rank> set2) {
        Set<Rank> intersection = new HashSet<>(set1);
        intersection.retainAll(set2);
        return intersection;
    }

    private GameBoard getFinishAttackState() {
        GameBoard altState = currState.copy();
        altState.getPlayingEntries().clear();
        altState.setUserAttacking(true);
        altState.setUserNext(true);
        giveCards(altState.getUser(), altState);
        giveCards(altState.getBot(), altState);
        return altState;
    }

    private int getMaxCardCountToThrowUp() {
        int res = 7 - getNumberOfDefendingCards();
        log.debug("Max cards count to throw up {}", res);
        return res;
    }

    private int getNumberOfDefendingCards() {
        return (int) currState.getPlayingEntries().stream()
                .mapToLong(entry -> entry.getDefending().size())
                .sum();
    }

    private void setBestNextState() {
        this.currState = nextStates.stream()
                .max(gameBoardComparator)
                .orElseThrow();
    }
}
package org.example.model.bot;

import org.example.model.game.GameBoard;
import org.example.model.game.Player;
import org.example.model.game.card.Card;
import org.example.model.game.card.Rank;
import org.example.model.game.card.Suit;

import java.util.*;

import static org.example.model.game.card.Suit.DIAMONDS;
import static org.example.model.game.card.Suit.SPADES;

public class PointCounter {

    private Player player;
    private final GameBoard gameBoard;
    private int[] countsByRank;
    private int[] countsBySuit;

    public static final float[] BONUSES = new float[]{0.0f, 0.0f, 0.5f, 0.75f,
1.25f};
    public static final int RANK_MULTIPLIER = 100;
    public static final int UNBALANCED_HAND_PENALTY = 200;
    public static final int MANY_CARDS_PENALTY = 1000;
```

```
    private final Map<Rank, Integer> values = new HashMap<>();

    {
        values.put(Rank.TWO, -6);
        values.put(Rank.THREE, -5);
        values.put(Rank.FOUR, -4);
        values.put(Rank.FIVE, -3);
        values.put(Rank.SIX, -2);
        values.put(Rank.SEVEN, -1);
        values.put(Rank.EIGHT, 0);
        values.put(Rank.NINE, 1);
        values.put(Rank.TEN, 2);
        values.put(Rank.JACK, 3);
        values.put(Rank.QUEEN, 4);
        values.put(Rank.KING, 5);
        values.put(Rank.ACE, 6);
        values.put(Rank.JOKER, 7);
    }

    public PointCounter(GameBoard gameBoard) {
        this.gameBoard = gameBoard;
    }

    public float getPoints(Player player) {
        this.player = Objects.requireNonNull(player);
        this.countsByRank = new int[14];
        this.countsBySuit = new int[4];

        float points = getRankPoints();
        System.out.println("Rank points: " + points);
        float unbalancedHandPenaltyPoints = getUnbalancedHandPenaltyPoints();
        System.out.println("Unbalanced hand penalty points " +
unbalancedHandPenaltyPoints);
        points += unbalancedHandPenaltyPoints;
        float manyCardsPenaltyPoints = getManyCardsPenaltyPoints();
        System.out.println("Many cards penalty points " +
manyCardsPenaltyPoints);
        points += manyCardsPenaltyPoints;

        return points;
    }

    private float getRankPoints() {
        float points = 0.0f;

        for (Card card : player.getHand()) {
            Rank rank = card.getRank();
            Suit suit = card.getSuit();

            points += (values.get(rank) * RANK_MULTIPLIER);

            if (isJoker(rank)) {
                if (isBlack(suit)) {
                    points += 27 * RANK_MULTIPLIER;
                } else {
                    points += 28 * RANK_MULTIPLIER;
                }
            } else {
                if (isGeneralTrump(suit)) {
                    points += 26 * RANK_MULTIPLIER;
                } else if (isOwnTrump(suit)) {
                    points += 13 * RANK_MULTIPLIER;
                }
            }
```

```
            countsByRank[rank.getValue()]++;
            if (!isJoker(rank)) {
                countsBySuit[suit.getValue()]++;
            }
        }

        points += getRankBonusPoints();

        return points;
    }

    private boolean isBlack(Suit suit) {
        return suit.equals(Suit.BLACK);
    }

    private boolean isJoker(Rank rank) {
        return rank.equals(Rank.JOKER);
    }

    private float getUnbalancedHandPenaltyPoints() {
        float points = 0.0f;

        float avgSuit = 0.0f;
        for (Card card : player.getHand()) {
            if (!isGeneralTrump(card.getSuit()) && !isJoker(card.getRank())) {
                avgSuit++;
            }
        }

        avgSuit /= 3.0f;
        for (Suit suit : EnumSet.range(DIAMONDS, SPADES)) {
            if (!isGeneralTrump(suit)) {
                float dev = Math.abs((countsBySuit[suit.getValue()] - avgSuit) /
avgSuit);
                points -= (int) (UNBALANCED_HAND_PENALTY * dev);
            }
        }

        return points;
    }

    private float getManyCardsPenaltyPoints() {
        int points = 0;

        int cardsRemaining = getCardsRemaining();
        System.out.println("Cards remaining: " + cardsRemaining);
        Player bot = gameBoard.getBot();
        int size = bot.getHand().size();
        System.out.println("Bot hand size: " + size);
        float cardRatio = cardsRemaining != 0 ? (size / (float) cardsRemaining)
: 10.0f;
        System.out.println("Card radio: " + cardRatio);
        points += (int) ((0.25f - cardRatio) * MANY_CARDS_PENALTY);
        System.out.println("Points: " + points);

        return points;
    }

    private float getRankBonusPoints() {
        float points = 0.0f;
        for (Rank rank : Rank.values()) {
            points += (Math.max(values.get(rank), 1.0f) *
BONUSES[countsByRank[rank.getValue()]]);
        }
        return points;
    }
```

```
    private int getCardsRemaining() {
        Player user = gameBoard.getUser(),
                bot = gameBoard.getBot();

        int remaining = 0;
        ArrayList<Card> botHand = bot.getHand(),
                userHand = user.getHand();
        remaining += botHand.size() + userHand.size();

        Stack<Card> userDeck = user.getDeck(),
                botDeck = bot.getDeck(),
                generalDeck = gameBoard.getGeneralDeck();
        remaining += userDeck.size() + botDeck.size() + generalDeck.size();

        Card userTrump = user.getTrump(),
                botTrump = user.getTrump(),
                generalTrump = gameBoard.getGeneralTrump();

        if (Objects.nonNull(userTrump))
            remaining += 1;
        if (Objects.nonNull(botTrump))
            remaining += 1;
        if (Objects.nonNull(generalTrump))
            remaining += 1;

        return remaining;
    }

    private boolean isGeneralTrump(Suit suit) {
        Suit trumpSuit = gameBoard.getGeneralTrumpSuit();
        return suit.equals(trumpSuit);
    }

    private boolean isOwnTrump(Suit suit) {
        Suit playerTrumpSuit = player.getTrumpSuit();
        return suit.equals(playerTrumpSuit);
    }
}
package org.example.model.decorator;

import javafx.scene.image.ImageView;
import org.example.model.game.GameBoard;
import org.example.model.game.card.Card;

import java.util.List;

public class ClickDecorator extends ImageView {

    private final List<Card> src;
    private final Card card;
    private boolean isChosen;
    private ImageView imageView;
    private GameBoard state;

    public ClickDecorator(ImageView imageView, List<Card> src, Card card,
GameBoard state) {
        this.src = src;
        this.card = card;
        this.imageView = imageView;
        this.state = state;

        this.imageView.setOnMouseClicked(mouseEvent -> {
            isChosen = !isChosen;
            if (isChosen) {
                if (state.isUserAttacking()) {
```

```
                    int botHandSize = state.getBot().getHand().size();

                    if (src.size() >= botHandSize)
                        return;
                } else {
                    if (src.size() >= 1)
                        return;
                }

                src.add(card);
                imageView.setOpacity(0.8);
            } else {
                src.remove(card);
                imageView.setOpacity(1);
            }
        });
    }

    public ImageView self() {
        return this.imageView;
    }
}
package org.example.model.game.card;

import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.Pane;
import org.example.Constant;
import org.example.model.Utils;

import java.util.Map;
import java.util.Optional;

public class Card implements Comparable<Card> {

    // diamonds - бубна
    // hearts - чірва
    // clubs - хреста
    // spades - піка

    private final Suit suit;
    private final Rank rank;
    private boolean isChosen;

    private Card(Suit suit, Rank rank) {
        this.suit = suit;
        this.rank = rank;
    }

    public static Card newCard(Suit suit, Rank rank) {
        return new Card(suit, rank);
    }

    public Suit getSuit() {
        return suit;
    }

    public Rank getRank() {
        return rank;
    }

    public boolean isChosen() {
        return isChosen;
    }

    public void setChosen(boolean chosen) {
```

```
        isChosen = chosen;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Card card)) return false;

        if (suit != card.suit) return false;
        return rank == card.rank;
    }

    @Override
    public int hashCode() {
        int result = suit != null ? suit.hashCode() : 0;
        result = 31 * result + (rank != null ? rank.hashCode() : 0);
        return result;
    }

    @Override
    public String toString() {
        return "Card{" +
                "suit=" + suit +
                ", value=" + rank +
                '}';
    }

    @Override
    public int compareTo(Card another) {
        return Integer.compare(rank.getValue(), another.rank.getValue());
    }

    public boolean isGreaterThan(Card another) {
        return this.compareTo(another) > 0;
    }

    public ImageView renderActive(Pane pane) {
        ImageView imageView = null;
        imageView.setOnMouseEntered(mouseEvent -> {
            if (!isChosen)
                imageView.setOpacity(0.8);
        });
        imageView.setOnMouseExited(mouseEvent -> {
            if (!isChosen)
                imageView.setOpacity(1);
        });
        return imageView;
    }

    public void render(Pane pane) {
        render(this, pane);
    }

    public ImageView toImageView() {
        return toImageView(this);
    }

    public static void render(Card card, Pane pane) {
        ImageView imageView = toImageView(card);
        pane.getChildren().add(imageView);
    }

    private static ImageView toImageView(Card card) {
        Image image = toImage(card);
        ImageView imageView = new ImageView(image);
        setSize(imageView);
```

```
        return imageView;
    }

    private static Image toImage(Card card) {
        Map<Card, Image> cardImageMap = Utils.getCardImageMap();
        return Optional.ofNullable(card)
                .map(cardImageMap::get)
                .orElse(new Image(Constant.CARD_BACK_SIDE));
    }

    private static void setSize(ImageView imageView) {
        imageView.setFitHeight(Constant.CARD_HEIGHT);
        imageView.setFitWidth(Constant.CARD_WIDTH);
    }
}
```

```
package org.example.model.game.card;

public enum Rank {

    TWO(0),
    THREE(1),
    FOUR(2),
    FIVE(3),
    SIX(4),
    SEVEN(5),
    EIGHT(6),
    NINE(7),
    TEN(8),
    JACK(9), // валет
    QUEEN(10), // дама
    KING(11),
    ACE(12),
    JOKER(13);

    private final int value;

    Rank(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}
```

```
package org.example.model.game.card;

public enum Suit {

    DIAMONDS(0),
    HEARTS(1),
    CLUBS(2),
    SPADES(3),
    BLACK(4),
    RED(5);

    private final int value;

    Suit(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}
```

```java
package org.example.model.game;

import org.example.model.game.card.Card;

import java.util.LinkedList;
import java.util.Objects;

public class Entry {

    private final LinkedList<Card> defending;
    private final LinkedList<Card> hitting;
    private boolean isBeaten;

    public Entry() {
        this.defending = new LinkedList<>();
        this.hitting = new LinkedList<>();
    }

    private Entry(LinkedList<Card> defending, LinkedList<Card> hitting) {
        this.defending = defending;
        this.hitting = hitting;
    }

    public static Entry newEntry() {
        return new Entry();
    }

    public LinkedList<Card> getDefending() {
        return defending;
    }

    public LinkedList<Card> getHitting() {
        return hitting;
    }


    public boolean isBeaten() {
        return isBeaten;
    }

    public void setBeaten(boolean beaten) {
        isBeaten = beaten;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Entry entry)) return false;

        if (isBeaten != entry.isBeaten) return false;
        if (!Objects.equals(defending, entry.defending)) return false;
        return Objects.equals(hitting, entry.hitting);
    }

    @Override
    public int hashCode() {
        int result = defending.hashCode();
        result = 31 * result + hitting.hashCode();
        result = 31 * result + (isBeaten ? 1 : 0);
        return result;
    }

    @Override
    public String toString() {
        return "Entry{" +
                "defending=" + defending +
```

```
                ", hitting=" + hitting +
                ", isBeaten=" + isBeaten +
                '}';
    }

    public Entry copy() {
        return new Entry(
                new LinkedList<>(this.defending),
                new LinkedList<>(this.hitting)
        );
    }
}
package org.example.model.game;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.example.model.Utils;
import org.example.model.game.card.Card;
import org.example.model.game.card.Rank;
import org.example.model.game.card.Suit;

import java.util.*;
import java.util.stream.Collectors;

/**
 * Редуду
 * Количество колод: 1
 * Количество карт в колоде: 52, 2 черных джокера и 1 красный джокер
 * Количество игроков: 2 - 3
 * Старшинство карт: 2, 3, 4, 5, 6, 7, 8, 9, 10, В, Д, К, Т.
 * Цель игры: первым избавиться от всех своих карт.
 * Правила игры. Первый сдатчик определяется жребием, в следующей игре карты
сдает игрок,
 * который проиграл в предыдущей игре. Колода тщательно тасуется, снимается и
сдается по 7 карт
 * каждому игроку. Далее снимается по одной карте в закрытом виде и кладется
возле каждого игрока,
 * начиная с игрока слева от сдатчика. Эти карты являются козырями для каждого
игрока. После этого с
 * оставшейся колоды снимается верхняя карта, открывается и кладется в центр
стола. Эта карта общий козырь.
 * Оставшаяся колода делится на равные части, если игроков трое, то делится на 4
части. 3 из них кладутся на
 * козырь каждого из игроков, а четвертая на общий козырь. Если колода не
делится поровну, то остаток карт
 * кладется на общий козырь. После этого начинается розыгрыш.
 * Козырями считаются следующие карты
 * Красная редуда - красный джокер, самая старшая карта, которая бьет 3 любые
карты или одну черную редуду
 * и любую карту, также переводит любую карту. Черная редуда - черный джокер,
который бьет две любые карты,
 * кроме редуды. Черная редуда и общий козырь бьют вместе красную редуду. Черная
редуда переводит любую карту.
 * Общий козырь - бьет любую карту, кроме редуды. Свой козырь - бьет любую
карту, кроме редуды и общего козыря.
 * Имеет значение только для игрока, которому он предназначен.
 * Розыгрыш
 * Розыгрыш происходит следующим образом. Игрок может зайти любой картой.
Следующий игрок должен по желанию
 * покрыть эту карту, перевести или взять. Правила перевода следующие:
 * под игрока с одной картой - не более трех простых карт;
 * под игрока с двумя картами - не более четырех;
 * под игрока с тремя картами - не более пяти, при условии, что у отбивающегося
есть красная редуда.
 * При переводе редудой, редуда играет самостоятельно, а не заменяет карту,
которую переводит.
```

```java
 * Если у игрока становится менее семи карт, то он добирает карты из общей
колоды. Как только общая колода
 * заканчивается, то игрок добирает из своей колоды. Игрок, который первым
остался без карт, становится
 * победителем. Проигравшего называют "редудак".
 */

public class GameBoard {
    private static final Logger log = LogManager.getLogger(GameBoard.class);
    private Stack<Card> generalDeck;
    private Card generalTrump;
    private Suit generalTrumpSuit;
    private List<Entry> playingEntries;
    private Player user;
    private Player bot;
    private boolean isUserNext;
    private boolean isUserAttacking;
    private static final int PLAYERS_DECK_SIZE = 12;
    private static final int PLAYER_CARDS_SIZE = 7;

    private GameBoard() {
        playingEntries = new LinkedList<>();
    }

    private GameBoard(
            Stack<Card> generalDeck,
            Card generalTrump,
            Suit generalTrumpSuit,
            List<Entry> playingEntries,
            Player user, Player bot,
            boolean isUserNext, boolean isUserAttacking) {
        this.generalDeck = generalDeck;
        this.generalTrump = generalTrump;
        this.playingEntries = playingEntries;
        this.generalTrumpSuit = generalTrumpSuit;
        this.user = user;
        this.bot = bot;
        this.isUserNext = isUserNext;
        this.isUserAttacking = isUserAttacking;
    }

    public static GameBoard newGameBoard() {
        return new GameBoard();
    }

    public Stack<Card> getGeneralDeck() {
        return generalDeck;
    }

    public Card getGeneralTrump() {
        return generalTrump;
    }

    public Player getUser() {
        return user;
    }

    public void setUser(Player user) {
        this.user = user;
    }

    public Player getBot() {
        return bot;
    }

    public void setBot(Player bot) {
```

```
        this.bot = bot;
    }

    public boolean isUserNext() {
        return isUserNext;
    }

    public void setUserNext(boolean userNext) {
        this.isUserNext = userNext;
    }

    public boolean isUserAttacking() {
        return isUserAttacking;
    }

    public void setUserAttacking(boolean userAttacking) {
        isUserAttacking = userAttacking;
    }

    public List<Entry> getPlayingEntries() {
        return playingEntries;
    }

    public Suit getGeneralTrumpSuit() {
        return generalTrumpSuit;
    }

    public void initDeck() {
        log.trace("Init deck is invoked");
        ArrayList<Card> cards = Utils.getCards();
        Stack<Card> deck = new Stack<>();
        deck.addAll(cards);
        log.debug("Deck state {}", deck);
        this.generalDeck = deck;
        log.trace("Init deck is terminated");
    }

    public void shuffleDeck() {
        log.trace("Shuffle deck is invoked");
        Collections.shuffle(generalDeck);
        log.debug("Deck state {}", generalDeck);
        log.trace("Shuffle deck is terminated");
    }

    public void giveCards(Player player) {
        log.trace("Give cards is invoked");
        ArrayList<Card> cards = new ArrayList<>();
        Card card;
        while (cards.size() < PLAYER_CARDS_SIZE && Objects.nonNull(card =
generalDeck.pop()))
            cards.add(card);
        log.debug("Player cards number is {}", cards.size());
        log.debug("Cards state {}", cards);
        log.debug("Game board deck size id {}", generalDeck.size());
        player.setHand(cards);
        log.trace("Give cards is terminated");
    }

    public void giveTrump(Player player) {
        log.trace("Give trump is invoked");
        Card trump = getDeckCardsListWithoutJokers().get(0);
        player.setTrump(trump);
        log.debug("Player {} got {} trump", player.getName(), trump);
        player.setTrumpSuit(trump.getSuit());
        log.debug("Player {} got {} trump suit", player.getName(),
trump.getSuit());
```

```
        generalDeck.remove(trump);
        log.trace("Give trump is terminated");
    }

    public void initGeneralTrump() {
        log.trace("Set general trump is invoked");
        Card trump = getDeckCardsListWithoutJokers().get(0);
        this.generalTrump = trump;
        this.generalTrumpSuit = trump.getSuit();
        generalDeck.remove(trump);
        log.trace("Set general trump is terminated");
    }

    private List<Card> getDeckCardsListWithoutJokers() {
        return generalDeck.stream()
                .filter(card -> !card.getRank().equals(Rank.JOKER))
                .toList();
    }

    public void giveDeck(Player player) {
        log.trace("Give deck is invoked");
        Stack<Card> deck = new Stack<>();
        for (int i = 0; i < PLAYERS_DECK_SIZE; i++)
            deck.push(this.generalDeck.pop());
        log.debug("Player deck size is {}", deck.size());
        log.debug("Player deck state {}", deck);
        log.debug("Game board deck size is {}", this.generalDeck.size());
        player.setDeck(deck);
        log.trace("Give deck is terminated");
    }

    public void setGeneralTrump(Card generalTrump) {
        this.generalTrump = generalTrump;
    }

    @SuppressWarnings("unchecked")
    public GameBoard copy() {
        Stack<Card> generalDeckCopy = (Stack<Card>) this.generalDeck.clone();
        List<Entry> playingEntriesCopy = this.playingEntries.stream()
                .map(entry -> entry.copy())
                .collect(Collectors.toList());
        Player userCopy = this.user.copy();
        Player botCopy = this.bot.copy();
        return new GameBoard(
                generalDeckCopy,
                this.generalTrump,
                this.generalTrumpSuit,
                playingEntriesCopy,
                userCopy,
                botCopy,
                this.isUserNext,
                this.isUserAttacking
        );
    }
}
package org.example.model.game;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import java.util.Objects;
import java.util.concurrent.ThreadLocalRandom;

public class GameSession {

    private static final Logger log = LogManager.getLogger(GameSession.class);
```

```
    private final Player user;
    private final Player bot;
    private GameBoard gameBoard;
    private Player looser;

    private GameSession(Player user, Player bot) {
        this.user = user;
        this.bot = bot;
        this.gameBoard = GameBoard.newGameBoard();
        gameBoard.setUser(this.user);
        gameBoard.setBot(this.bot);
    }

    public static GameSession newGameSession(Player user, Player bot) {
        return new GameSession(user, bot);
    }

    public void start() {
        log.trace("Start is invoked");
        this.gameBoard.initDeck();
        this.gameBoard.shuffleDeck();
        this.gameBoard.giveCards(user);
        this.gameBoard.giveCards(bot);
        this.gameBoard.giveTrump(user);
        this.gameBoard.giveTrump(bot);
        this.gameBoard.initGeneralTrump();
        this.gameBoard.giveDeck(user);
        this.gameBoard.giveDeck(bot);
        this.chooseNextPlayer();
        log.trace("Start is terminated");
    }

    private void chooseNextPlayer() {
        log.trace("Choose next player is invoked");
        if (Objects.nonNull(looser)) {
            gameBoard.setUserNext(looser == user);
            gameBoard.setUserAttacking(looser == user);
            log.debug("Next player is looser");
        } else {
            int random = ThreadLocalRandom.current().nextInt(0, 2);
            if (random == 0) {
                gameBoard.setUserNext(true);
                gameBoard.setUserAttacking(true);
                log.debug("Next player is user");
            } else {
                gameBoard.setUserNext(false);
                gameBoard.setUserAttacking(false);
                log.debug("Next player is bot");
            }
        }
        log.trace("Choose next player is terminated");
    }

    public Player getUser() {
        return user;
    }

    public Player getBot() {
        return bot;
    }

    public GameBoard getGameBoard() {
        return gameBoard;
    }

    public void setGameBoard(GameBoard gameBoard) {
```

```
        this.gameBoard = gameBoard;
    }
}
package org.example.model.game;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.example.model.game.card.Card;
import org.example.model.game.card.Suit;

import java.util.ArrayList;
import java.util.Objects;
import java.util.Stack;

public class Player {
    private static final Logger log = LogManager.getLogger(Player.class);
    private String name;
    private ArrayList<Card> hand;
    private Card trump;
    private Suit trumpSuit;
    private Stack<Card> deck;
    private static final int MAX_NUMBER_OF_CARDS = 7;

    private Player(String name) {
        this.name = name;
    }

    private Player(String name, ArrayList<Card> hand, Card trump, Suit
trumpSuit, Stack<Card> deck) {
        this.name = name;
        this.hand = hand;
        this.trump = trump;
        this.trumpSuit = trumpSuit;
        this.deck = deck;
    }

    public static Player newPlayer(String name) {
        return new Player(name);
    }

    public String getName() {
        return name;
    }

    public ArrayList<Card> getHand() {
        return hand;
    }

    public Card getTrump() {
        return trump;
    }

    public Stack<Card> getDeck() {
        return deck;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setHand(ArrayList<Card> hand) {
        this.hand = hand;
    }

    public void setTrump(Card trump) {
        this.trump = trump;
```

```
    }

    public void setDeck(Stack<Card> deck) {
        this.deck = deck;
    }


    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Player player)) return false;

        return Objects.equals(name, player.name);
    }

    @Override
    public int hashCode() {
        return name != null ? name.hashCode() : 0;
    }

    @Override
    public String toString() {
        return "Player{" +
                "name='" + name + '\'' +
                '}';
    }

    @SuppressWarnings("unchecked")
    public Player copy() {
        ArrayList<Card> handCopy = new ArrayList<>(this.getHand());
        Stack<Card> deckCopy = (Stack<Card>) this.deck.clone();
        return new Player(this.name, handCopy, this.trump, this.trumpSuit,
deckCopy);
    }

    public void setTrumpSuit(Suit suit) {
        this.trumpSuit = suit;
    }

    public Suit getTrumpSuit() {
        return trumpSuit;
    }
}
package org.example.model;

import javafx.scene.image.Image;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.example.Constant;
import org.example.model.game.Entry;
import org.example.model.game.GameBoard;
import org.example.model.game.Player;
import org.example.model.game.card.Card;
import org.example.model.game.card.Rank;
import org.example.model.game.card.Suit;

import java.io.File;
import java.util.*;
import java.util.stream.Collectors;

import static java.util.stream.Collectors.toSet;
import static org.example.model.game.card.Suit.*;

public class Utils {

    private static final Logger log = LogManager.getLogger(Utils.class);
```

```
    private static final ArrayList<Card> cards;
    private static final Map<Card, Image> cardImageMap;

    static {
        cards = initCards();
        cardImageMap = initCardImageMap();
    }

    private static ArrayList<Card> initCards() {
        log.trace("Init cards invoked");
        ArrayList<Card> cards = EnumSet.range(Rank.TWO, Rank.ACE)
                .stream()
                .map(Utils::buildCardsOfAllSuitsFromValue)
                .flatMap(Collection::stream)
                .collect(Collectors.toCollection(ArrayList::new));

        List<Card> jokers = List.of(
                Card.newCard(BLACK, Rank.JOKER),
                Card.newCard(BLACK, Rank.JOKER),
                Card.newCard(RED, Rank.JOKER));

        log.trace("Add jokers");
        cards.addAll(jokers);

        log.debug("Cards {}", cards);
        log.debug("Number of cards {}", cards.size());

        return cards;
    }

    private static Map<Card, Image> initCardImageMap() {
        log.trace("Init card image map is invoked");
        Map<Card, Image> cardImageMap = new HashMap<>();
        File cardImages = new File(Constant.CARD_IMAGES);
        log.debug("Card imaged file path {}", cardImages.getPath());
        for (File cardImage : Objects.requireNonNull(cardImages.listFiles())) {
            String cardImageName = cardImage.getName();
            log.debug("Card image name {}", cardImageName);
            cardImageName = cardImageName.replace(".png", "");
            log.debug("Card image name without .png {}", cardImageName);
            String[] rankSuitMapping = cardImageName.split("_of_");
            log.debug("Rank and suit mapping {}",
Arrays.toString(rankSuitMapping));
            Rank rank =
Rank.valueOf(rankSuitMapping[0].toUpperCase(Locale.ROOT));
            Suit suit =
Suit.valueOf(rankSuitMapping[1].toUpperCase(Locale.ROOT));
            Card card = Card.newCard(suit, rank);
            String cardImagePath = cardImage.getAbsolutePath();
            log.debug("Card image path {}", cardImagePath);
            cardImageMap.put(card, new Image(cardImagePath));
        }
        log.debug("Card image map {}", cardImageMap);
        log.trace("Init card image map is terminated");
        return cardImageMap;
    }

    private static LinkedList<Card> buildCardsOfAllSuitsFromValue(Rank rank) {
        log.debug("Cards for {} value", rank.name().toLowerCase());
        return EnumSet.range(DIAMONDS, SPADES)
                .stream()
                .map(suit -> Card.newCard(suit, rank))
                .collect(Collectors.toCollection(LinkedList::new));
    }

    public static ArrayList<Card> getCards() {
```

```
        return new ArrayList<>(cards);
    }
    public static Map<Card, Image> getCardImageMap() {return cardImageMap;}

    public static void shuffleCards(ArrayList<Card> cards) {
        Collections.shuffle(cards);
    }

    public static Set<Card> getPlayingCards(GameBoard gameBoard) {
        Set<Card> defendingCardSet = gameBoard.getPlayingEntries().stream()
                .flatMap(entry -> entry.getDefending().stream())
                .collect(toSet());

        Set<Card> hittingCardSet = gameBoard.getPlayingEntries().stream()
                .flatMap(entry -> entry.getHitting().stream())
                .collect(toSet());

        defendingCardSet.addAll(hittingCardSet);

        return defendingCardSet;
    }

    public static <T> List<List<T>> combinations(List<T> values, int size) {

        if (0 == size) {
            return Collections.singletonList(Collections.<T> emptyList());
        }

        if (values.isEmpty()) {
            return Collections.emptyList();
        }

        List<List<T>> combination = new LinkedList<List<T>>();

        T actual = values.iterator().next();

        List<T> subSet = new LinkedList<T>(values);
        subSet.remove(actual);

        List<List<T>> subSetCombination = combinations(subSet, size - 1);

        for (List<T> set : subSetCombination) {
            List<T> newSet = new LinkedList<T>(set);
            newSet.add(0, actual);
            combination.add(newSet);
        }

        combination.addAll(combinations(subSet, size));

        return combination;
    }

    public static boolean canHit(Card hitting, Card defending, GameBoard state,
Player player) {
        Suit hittingSuit = hitting.getSuit(),defendingSuit =
defending.getSuit();
        Rank hittingRank = hitting.getRank(), defendingRank =
defending.getRank();

        log.debug("Player trump suit {}", player.getTrumpSuit());

        if (hittingSuit.equals(Suit.RED))
            return true;

        if (hittingSuit.equals(Suit.BLACK))
            return !defendingRank.equals(Rank.JOKER);
```

```
        Suit generalTrumpSuit = state.getGeneralTrumpSuit();
        if (hittingSuit.equals(generalTrumpSuit)) {
            if (defendingRank.equals(Rank.JOKER))
                return false;

            if (defendingSuit.equals(generalTrumpSuit))
                return hittingRank.getValue() > defendingRank.getValue();

            return true;
        }
        if (hittingSuit.equals(player.getTrumpSuit())) {
            log.debug("Hitting card's suit equals to player trump suit");
            if (defendingRank.equals(Rank.JOKER)
                    || defendingSuit.equals(generalTrumpSuit))
                return false;
            log.debug("Defending card is not joker or general trump");

            if (defendingSuit.equals(hittingSuit))
                return hittingRank.getValue() > defendingRank.getValue();
            log.debug("Defending suit is not equals to hitting suit");

            return true;
        }
        if (!defendingRank.equals(Rank.JOKER) &&
!defendingSuit.equals(generalTrumpSuit)) {
            if (defendingSuit.equals(hittingSuit)) {
                return hittingRank.getValue() > defendingRank.getValue();
            }
        }

        return false;
    }

    public static void giveCards(Player player, GameBoard gameBoard) {
        Stack<Card> generalDeck = gameBoard.getGeneralDeck();
        ArrayList<Card> playerHand = player.getHand();
        while (playerHand.size() < 7 && !generalDeck.empty()) {
            playerHand.add(generalDeck.pop());
        }

        Card generalTrump = gameBoard.getGeneralTrump();
        if (playerHand.size() < 7 && generalTrump != null) {
            playerHand.add(generalTrump);
            gameBoard.setGeneralTrump(null);
        }

        Stack<Card> playerDeck = player.getDeck();
        while (playerHand.size() < 7 && !playerDeck.empty()) {
            playerHand.add(playerDeck.pop());
        }

        Card playerTrump = player.getTrump();
        if (playerHand.size() < 7 && playerTrump != null) {
            playerHand.add(playerTrump);
            player.setTrump(null);
        }
    }

    public static void doTransfer(GameBoard gameBoard, List<Card> chosenCards,
Player defender) {
        Card transferCard = chosenCards.get(0);
        chosenCards.clear();

        defender.getHand().remove(transferCard);
```

```
        List<Entry> playingEntries = gameBoard.getPlayingEntries();
        Entry entry = new Entry();
        entry.getDefending().add(transferCard);
        playingEntries.add(entry);

        gameBoard.setUserAttacking(!gameBoard.isUserAttacking());
        gameBoard.setUserNext(!gameBoard.isUserNext());
    }

    public static boolean transferIsAllowed(GameBoard gameBoard,
LinkedList<Card> chosenCards, Player opponent) {
        if (defenderStartedHitting(gameBoard)) {
            return false;
        }

        if (!isTransferCardCountAllowed(gameBoard, opponent)) {
            return false;
        }

        if (transferredCardIsJoker(gameBoard)) {
            return false;
        }

        if (transferCardIsJoker(chosenCards)) {
            return true;
        }

        return isTransferAndTransferredCardsSameRank(chosenCards, gameBoard);
    }

    public static boolean transferredCardIsJoker(GameBoard gameBoard) {
        return gameBoard.getPlayingEntries().stream()
                .flatMap(entry -> entry.getDefending().stream())
                .findFirst()
                .orElseThrow()
                .getRank().equals(Rank.JOKER);
    }

    public static boolean transferCardIsJoker(LinkedList<Card> chosenCards) {
        Card transferCard = chosenCards.get(0);
        return transferCard.getRank().equals(Rank.JOKER);
    }

    public static boolean isTransferAndTransferredCardsSameRank(LinkedList<Card>
chosenCards, GameBoard gameBoard) {
        Card transferCard = chosenCards.get(0);
        Card transferredCard = gameBoard.getPlayingEntries().stream()
                .flatMap(entry -> entry.getDefending().stream())
                .findFirst()
                .orElseThrow();
        return transferCard.getRank().equals(transferredCard.getRank());
    }

    public static boolean isTransferCardCountAllowed(GameBoard gameBoard, Player
opponent) {
        int opponentHandSize = opponent.getHand().size();

        return opponentHandSize >= (gameBoard.getPlayingEntries().stream()
                .mapToLong(entry -> entry.getDefending().size())
                .count() + 1);
    }

    public static boolean defenderStartedHitting(GameBoard gameBoard) {
        return gameBoard.getPlayingEntries().stream()
                .mapToLong(entry -> entry.getHitting().size())
                .sum() > 0;
```

```
    }
}
package org.example;

public class Constant {

    public static final String CARD_BACK_SIDE = "D:\\KPI\\Algorithms\\Labs\\lab-
6\\src\\main\\resources\\back-side.png";
    public static final String CARD_IMAGES = "src/main/resources/cards";
    public static final int CARD_HEIGHT = 100;
    public static final int CARD_WIDTH = 65;

}
package org.example;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Main extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage stage) throws Exception {
        Parent root =
FXMLLoader.load(Main.class.getResource("/org.example/start-scene.fxml"));
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.show();
    }
}
```

3.1.2. Приклади роботи.

На рисунках 3.1, 3.2, 3.3, 3.4, 3.5 показані приклади роботи програми.

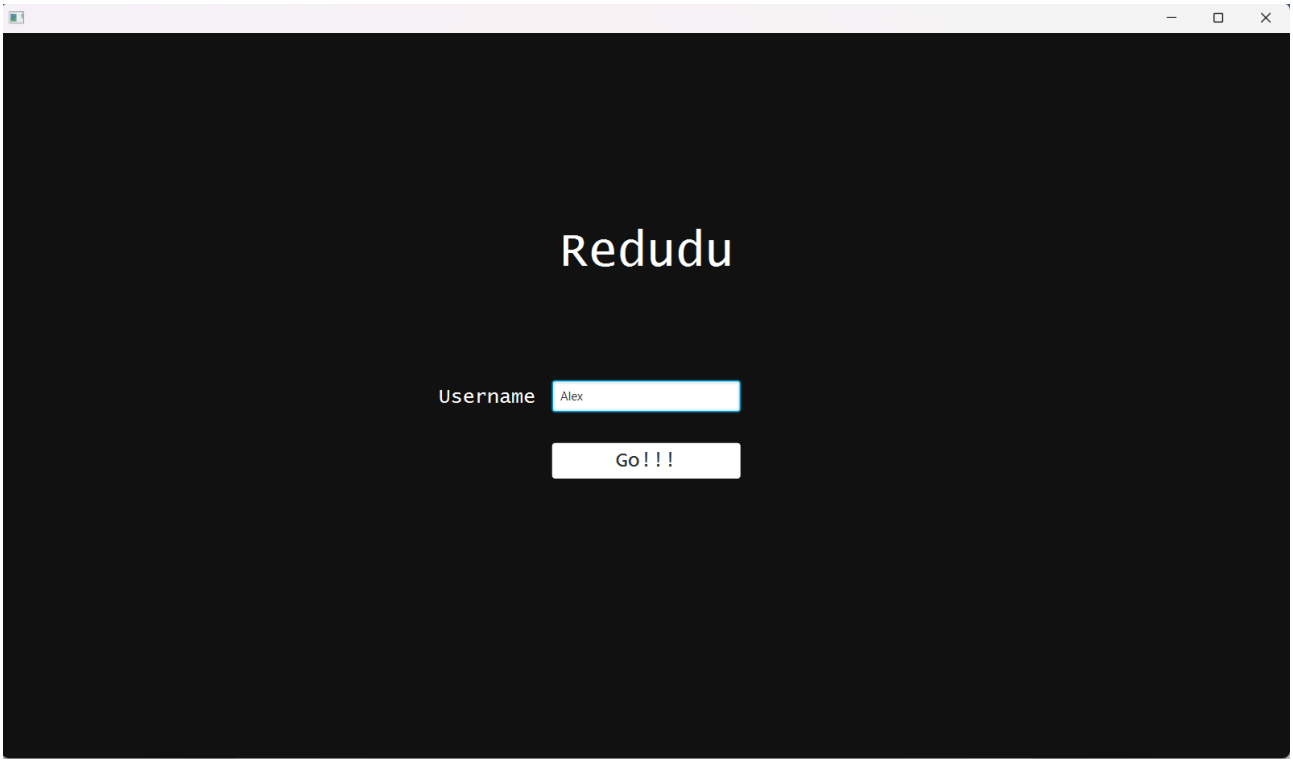

Рисунок 3.1 – Початкова сторінка

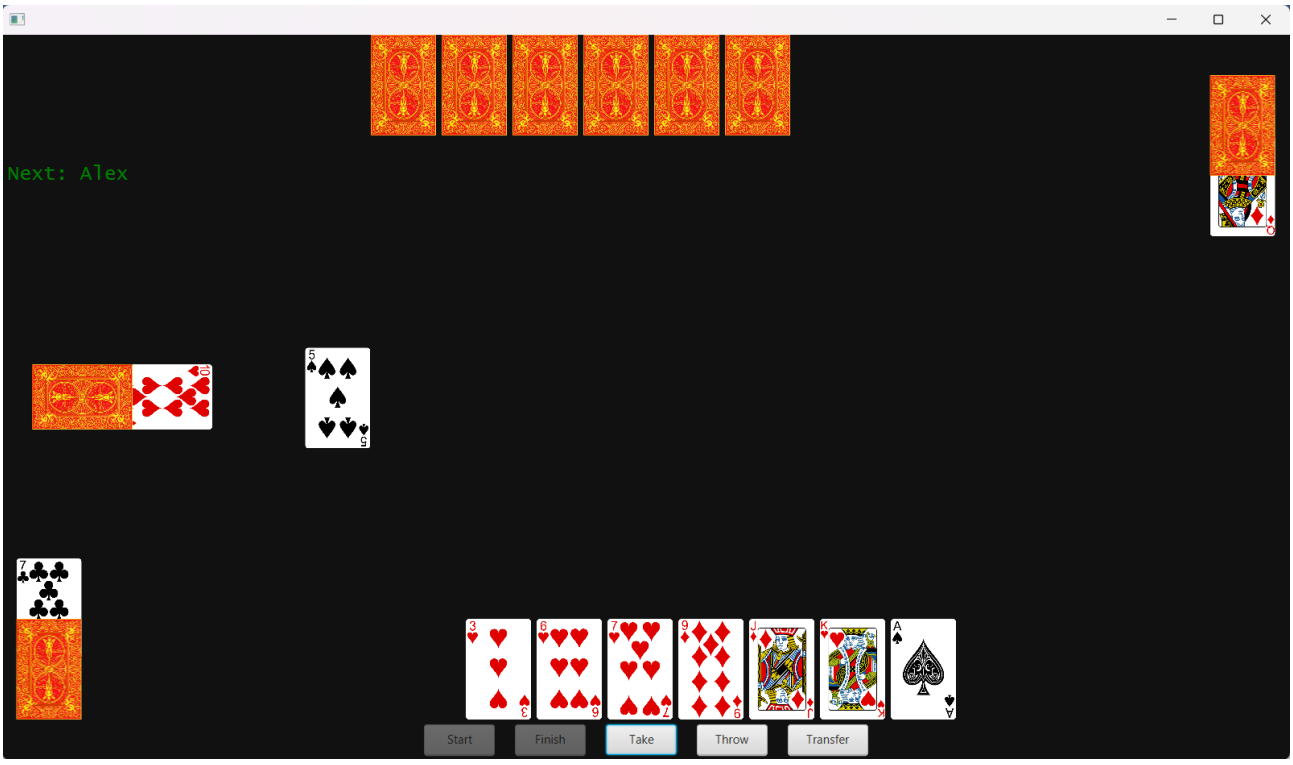

Рисунок 3.2 – Роздача карт

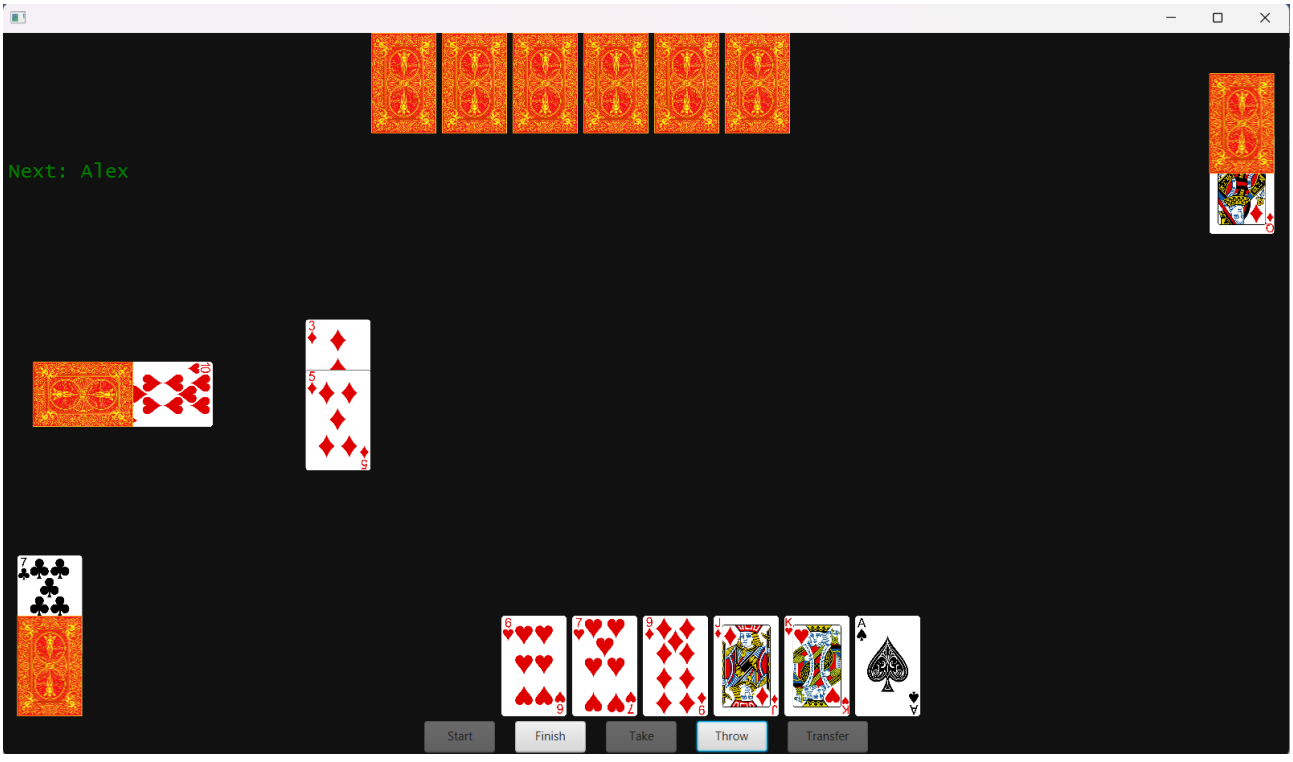

Рисунок 3.3 – Хід гри.

Рисунок 3.4 – Хід гри.

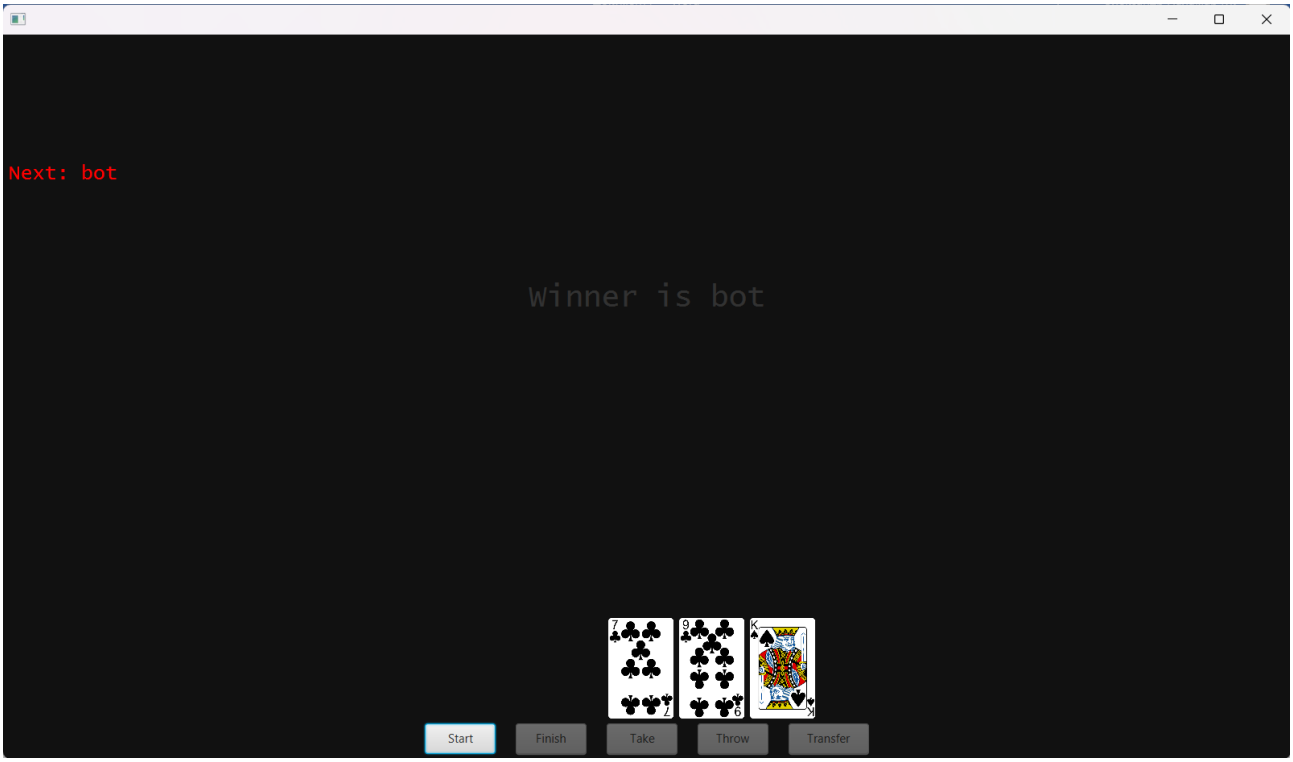

Рисунок 3.5 – Завершення гри.

4. Висновок

На лабораторній роботі було вивчено основні підходи до формалізації алгоритмів знаходження рішень задач в умовах протидії. Було ознайомлено з підходами до програмування алгоритмів штучного інтелекту в іграх з повною інформацією, іграх з елементами випадковості та в іграх з неповною

інформацією. Було реалізовано візуальний ігровий додаток для гри користувача з комп'ютерним опонентом.