yМинистерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра информационных технологий**

**ОТЧЁТ №7**

**Дисциплина: Многоагентное моделирование**

Работу выполнил: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_М. В. Сидоренко

Направление подготовки: 02.03.03 Математическое обеспечение и администрирование информационных систем

Преподаватель: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А. А. Миков

Краснодар

2024

**Цель работы**

Запрограммировать карточную игру в «Дурака» 2 на 2, где агенты соревнуются друг против друга разными алгоритмами игры.

**Описание задачи**

Есть 4 агента, поделённые на соперничающий пары и взаимодействующие со средой – карточным столом.

Среда содержит 52 карты, делённые на 7 подмножеств, которые определяют её состояние:

1. У каждого из 4 агентов своё подмножество карт;

2. Открытые карты раунда на столе;

3. Колода «Бито»;

4. Общая колода, из которой добираются карты, с нижней картой-козырем.

Каждый агент видит своё подмножество карт, открытые карты на столе и нижнюю (козырную) карту общей колоды.

Процесс игры выглядит так:

1. Инициализация (перемешивание колоды, выдача карт, определение порядка ходов – всё это выполняется средой);

2. Игра по правилам «подкидного и переводного Дурака» по раундам;

3. Учёт результата.

В каждом раунде у пары определяется ведущий игрок. Ведущие игроки играют друг против друга, напарники же помогают тем, что подкидывают карты.

Ведущим и заходящим игроком первого раунда становится игрок, у которого наименьший номинал карты козырной масти. В соперничающей паре ведущий игрок выбирается случайно.

В ходе игры заходящий игрок кладёт на стол любую из имеющихся у него карт, а отбивающийся игрок (игрок, под которого сделан заход) должен либо побить её, либо взять. Чтобы побить (синоним – покрыть) карту, нужно из имеющихся на руках карт положить на неё более старшую карту такой же масти, либо козыря, если битая карта – не козырь. Если битая карта – козырь, то побить её можно только более старшим козырем. После этого, если у игрока остаётся одна карта, то никто не имеет права атаковать его.

Заходящий игрок или его напарник могут положить (подкинуть, подбросить) ещё одну или несколько карт любой масти, достоинство которых совпадает с достоинством любой из карт, уже участвовавших в данном заходе.

Каждый раунд выглядит так:

1. Заходящий игрок выбирает карту;

2. Если итерация первая, то отбивающийся игрок может перевести ход, если у него есть карта того же достоинства, но другой масти, и добавить её на стол. В таком случае игра продолжается со следующего пункта, но роли изменены;

3. Отбивающий игрок выбирает карту, которой побьёт карту заходящего. Если такой карты нет, то он обязан взять карту;

4. Если отбивающий находит подходящую карту, он бьёт карту заходящего, и тот может подкинуть карту. Если заходящий решает не подкидывать, то он предлагает подкинуть карту своему напарнику;

5. Пункт аналогичен пункту 3, однако если отбивающему нечем крыть новую карту, он забирает все, что лежат на столе (в открытом множестве карт), и раунд заканчивается;

6. Если подкидывать нечего и все карты биты, то раунд заканчивается;

7. Следующий раунд начинает пара, выигравшая предыдущий раунд.

Ротация ведущих игроков в каждой паре остаётся на усмотрение разработчика.

Если в паре у одного игрока закончились карты, то он выходит. Победой считается ситуация, когда у обоих игроков пары закончились карты.

**Задача**

Разработать и описать два алгоритма игры (для каждой пары свой). Провести 1000 экспериментов и сравнить количество побед для разных алгоритмов.

**Описание решения**

В процессе разработки программы для симуляции карточной игры на языке Java была создана структура проекта, включающая несколько классов и интерфейсов. Вот описание решения:

**Структура проекта**

1. Пакет org.example:

- Agent.java: Этот класс представляет игрового агента (игрока). Каждый агент имеет уникальный идентификатор, стратегию и список карт (руку). Агент может выполнять ход в зависимости от своей стратегии.

- Card.java: Класс, реализующий объект "Карта". Каждая карта имеет масть и ранг. Предоставляются методы сравнения карт по масти и рангу, а также метод для представления карты в виде строки.

- GameEnvironment.java: Основной класс, управляющий игрой. Он отвечает за раздачу карт, определение текущего атакующего и защитника, а также за управление очередью игроков и переходом хода.

- StrategyComparisonChart.java: Класс для визуализации результатов экспериментов с использованием библиотеки JFreeChart. Он создает график, отображающий процент побед для различных стратегий.

2. Пакет org.example.strategy:

- Strategy.java: Интерфейс, определяющий метод execute, который каждая стратегия должна реализовать. Стратегия определяет, какую карту агент должен сыграть в зависимости от роли (атакующий или защитник) и текущей игровой ситуации.

- AggressiveStrategy.java, DefensiveStrategy.java, RandomStrategy.java, BalancedStrategy.java: Реализации различных стратегий игры. Каждая стратегия имеет свою логику выбора карт для атаки и защиты.

**Основной класс**

- App.java: Основной класс приложения, который запускает симуляцию игр. Он создает экземпляры агентов с различными стратегиями и проводит серию игр для сравнения их эффективности. Результаты игр выводятся в консоль.

**Логика игры**

- Игра состоит из нескольких раундов, где агенты поочередно атакуют и защищаются, используя свои стратегии.

- Игра продолжается до тех пор, пока у одной из пар игроков не закончатся карты.

- Победитель определяется на основании количества выигранных раундов.

**Визуализация результатов**

- Результаты симуляции отображаются с помощью графика, который строится с использованием JFreeChart. График помогает визуализировать, какие стратегии более успешны в данной игровой среде.

**Код программы**

public class Agent {  
 private int id;  
 private Strategy strategy;  
 private List<Card> hand;  
  
 public Agent(int id, Strategy strategy) {  
 this.id = id;  
 this.strategy = strategy;  
 this.hand = new ArrayList<>();  
 }  
  
 public Card makeMove(GameEnvironment env, String role, List<Card> partnerHand) {  
 if (hand.isEmpty()) {  
 return null;  
 }  
 return strategy.execute(this, env, role, partnerHand);  
 }  
  
 public List<Card> getHand() {  
 return hand;  
 }  
  
}

public class Card {  
 private String suit;  
 private int rank;  
  
 public Card(String suit, int rank) {  
 this.suit = suit;  
 this.rank = rank;  
 }  
  
 @Override  
 public String toString() {  
 return rank + suit.toString();  
 }  
  
 public boolean isLessThan(Card other) {  
 if (this.suit == other.suit) {  
 return this.rank < other.rank;  
 }  
 return false;  
 }  
  
 public String getSuit() {  
 return suit;  
 }  
  
 public int getRank() {  
 return rank;  
 }  
}

public interface Strategy {  
 Card execute(Agent agent, GameEnvironment env, String role, List<Card> partnerHand);  
}

public class RandomStrategy implements Strategy {  
 private Random random = new Random();  
  
 @Override  
 public Card execute(Agent agent, GameEnvironment env, String role, List<Card> partnerHand) {  
 List<Card> hand = agent.getHand();  
 return hand.isEmpty() ? null : hand.get(random.nextInt(hand.size()));  
 }  
}

public class DefensiveStrategy implements Strategy {  
 @Override  
 public Card execute(Agent agent, GameEnvironment env, String role, List<Card> partnerHand) {  
 List<Card> hand = agent.getHand();  
 if (role.equals("attacker")) {  
 return hand.stream().min((a, b) -> Integer.*compare*(a.getRank(), b.getRank())).orElse(null);  
 } else if (role.equals("defender")) {  
 Card attackingCard = env.getTable().get(env.getTable().size() - 1);  
 return hand.stream()  
 .filter(card -> card.getSuit().equals(attackingCard.getSuit()) && card.getRank() > attackingCard.getRank())  
 .max((a, b) -> Integer.*compare*(a.getRank(), b.getRank()))  
 .orElse(null);  
 }  
 return null;  
 }  
}

public class BalancedStrategy implements Strategy {  
 @Override  
 public Card execute(Agent agent, GameEnvironment env, String role, List<Card> partnerHand) {  
 List<Card> hand = agent.getHand();  
 if (role.equals("attacker")) {  
 return hand.stream()  
 .sorted((a, b) -> Integer.*compare*(a.getRank(), b.getRank()))  
 .skip(hand.size() / 2)  
 .findFirst()  
 .orElse(null);  
 } else if (role.equals("defender")) {  
 Card attackingCard = env.getTable().get(env.getTable().size() - 1);  
 List<Card> validMoves = hand.stream()  
 .filter(card -> card.getSuit().equals(attackingCard.getSuit()) && card.getRank() > attackingCard.getRank())  
 .sorted((a, b) -> Integer.*compare*(a.getRank(), b.getRank()))  
 .collect(Collectors.*toList*());  
 return validMoves.isEmpty() ? null : validMoves.get(validMoves.size() / 2);  
 }  
 return null;  
 }  
}

public class AggressiveStrategy implements Strategy {  
 @Override  
 public Card execute(Agent agent, GameEnvironment env, String role, List<Card> partnerHand) {  
 List<Card> hand = agent.getHand();  
 if (role.equals("attacker")) {  
 return hand.stream().max((a, b) -> Integer.*compare*(a.getRank(), b.getRank())).orElse(null);  
 } else if (role.equals("defender")) {  
 Card attackingCard = env.getTable().get(env.getTable().size() - 1);  
 return hand.stream()  
 .filter(card -> card.getSuit().equals(attackingCard.getSuit()) && card.getRank() > attackingCard.getRank())  
 .min((a, b) -> Integer.*compare*(a.getRank(), b.getRank()))  
 .orElse(null);  
 }  
 return null;  
 }  
}

package org.example;  
  
import java.util.\*;  
  
public class GameEnvironment {  
 private Deque<Card> deck;  
 private List<Card> discardPile;  
 private Map<Integer, List<Card>> hands;  
 private List<Card> table;  
 private Card trump;  
  
 public GameEnvironment() {  
 String[] suits = {"♠️", "♣️", "♥️", "♦️"};  
 int[] ranks = {6, 7, 8, 9, 10, 11, 12, 13, 14};  
 deck = new ArrayDeque<>();  
 discardPile = new ArrayList<>();  
 hands = new HashMap<>();  
 table = new ArrayList<>();  
  
 for (String suit : suits) {  
 for (int rank : ranks) {  
 deck.add(new Card(suit, rank));  
 }  
 }  
  
 // Преобразуем ArrayDeque в List для перемешивания  
 List<Card> deckList = new ArrayList<>(deck);  
 Collections.*shuffle*(deckList); // Перемешиваем список  
  
 // Если нужно сохранить перемешанную колоду в ArrayDeque, добавляем обратно  
 deck.clear(); // Очищаем оригинальный deque  
 deck.addAll(deckList); // Добавляем все перемешанные карты обратно в deque  
 }  
  
 public void deal() {  
 trump = ((ArrayDeque<Card>) deck).peekLast();  
 for (int i = 0; i < 6; i++) {  
 for (int j = 0; j < 4; j++) {  
 if (!deck.isEmpty()) {  
 hands.computeIfAbsent(j, k -> new ArrayList<>()).add(deck.pollFirst());  
 }  
 }  
 }  
 }  
  
 public void resetTable() {  
 table.clear();  
 }  
  
 public void moveToDiscardPile() {  
 discardPile.addAll(table);  
 resetTable();  
 }  
  
 public List<Card> getTable() {  
 return table;  
 }  
  
 public Map<Integer, List<Card>> getHands() {  
 return hands;  
 }  
  
 public Card getTrump() {  
 return trump;  
 }  
}

package org.example;  
  
import org.jfree.chart.ChartFactory;  
import org.jfree.chart.ChartPanel;  
import org.jfree.chart.JFreeChart;  
import org.jfree.chart.plot.CategoryPlot;  
import org.jfree.chart.plot.PlotOrientation;  
import org.jfree.chart.axis.CategoryLabelPositions;  
import org.jfree.data.category.CategoryDataset;  
import org.jfree.data.category.DefaultCategoryDataset;  
  
import javax.swing.\*;  
import java.awt.\*;  
import java.util.Map;  
import java.util.HashMap;  
  
public class StrategyComparisonChart extends JFrame {  
  
 private Map<String, Map<String, double[]>> results;  
  
 public StrategyComparisonChart(String title, Map<String, Map<String, double[]>> results) {  
 super(title);  
 this.results = results;  
  
 // Формирование набора данных  
 CategoryDataset dataset = createDataset();  
  
 // Создание графика  
 JFreeChart chart = ChartFactory.*createStackedBarChart*(  
 "Сравнение победных стратегий",  
 "Сравнение стратегий",  
 "Процент побед",  
 dataset,  
 PlotOrientation.*VERTICAL*,  
 true,  
 true,  
 false  
 );  
  
 // Настройка отображения графика  
 CategoryPlot plot = (CategoryPlot) chart.getPlot();  
 plot.getDomainAxis().setCategoryLabelPositions(CategoryLabelPositions.*UP\_45*);  
  
 ChartPanel chartPanel = new ChartPanel(chart);  
 chartPanel.setPreferredSize(new Dimension(800, 600));  
 setContentPane(chartPanel);  
 }  
  
 private CategoryDataset createDataset() {  
  
  
// String[] strategies = {  
// "агрессивная\_стратегия", "улучшенная\_защитная\_стратегия", "случайная\_стратегия", "сбалансированная\_ловушка"  
// };  
//  
// Map<String, Map<String, double[]>> results = new HashMap<>();  
// results.put("агрессивная\_стратегия", Map.of(  
// "агрессивная\_стратегия", new double[]{39.8, 60.2},  
// "улучшенная\_защитная\_стратегия", new double[]{38.7, 61.3},  
// "случайная\_стратегия", new double[]{40.9, 59.1},  
// "сбалансированная\_ловушка", new double[]{41.2, 58.8}  
// ));  
// results.put("улучшенная\_защитная\_стратегия", Map.of(  
// "агрессивная\_стратегия", new double[]{44.4, 55.6},  
// "улучшенная\_защитная\_стратегия", new double[]{39.4, 60.6},  
// "случайная\_стратегия", new double[]{42.6, 57.4},  
// "сбалансированная\_ловушка", new double[]{42.2, 57.8}  
// ));  
// results.put("случайная\_стратегия", Map.of(  
// "агрессивная\_стратегия", new double[]{43.4, 56.6},  
// "улучшенная\_защитная\_стратегия", new double[]{38.4, 61.6},  
// "случайная\_стратегия", new double[]{41.5, 58.5},  
// "сбалансированная\_ловушка", new double[]{40.7, 59.3}  
// ));  
// results.put("сбалансированная\_ловушка", Map.of(  
// "агрессивная\_стратегия", new double[]{42.3, 57.7},  
// "улучшенная\_защитная\_стратегия", new double[]{39.7, 60.3},  
// "случайная\_стратегия", new double[]{40.2, 59.8},  
// "сбалансированная\_ловушка", new double[]{39.9, 60.1}  
// ));  
  
  
 DefaultCategoryDataset dataset = new DefaultCategoryDataset();  
  
// for (String strategy : strategies) {  
// Map<String, double[]> comparisonResults = results.get(strategy);  
// for (Map.Entry<String, double[]> entry : comparisonResults.entrySet()) {  
// String comparedStrategy = entry.getKey();  
// double[] percentages = entry.getValue();  
// dataset.addValue(percentages[0], strategy, comparedStrategy);  
// }  
// }  
  
 for (String strategy : results.keySet()) {  
 Map<String, double[]> comparisonResults = results.get(strategy);  
 for (Map.Entry<String, double[]> entry : comparisonResults.entrySet()) {  
 String comparedStrategy = entry.getKey();  
 double[] percentages = entry.getValue();  
 dataset.addValue(percentages[0], strategy, comparedStrategy);  
 }  
 }  
  
 return dataset;  
 }  
  
 public static void init(String[] args, Map<String, Map<String, double[]>> results) {  
 SwingUtilities.*invokeLater*(() -> {  
 StrategyComparisonChart chart = new StrategyComparisonChart("Сравнение стратегий", results);  
 chart.setSize(800, 600);  
 chart.setLocationRelativeTo(null);  
 chart.setDefaultCloseOperation(WindowConstants.*EXIT\_ON\_CLOSE*);  
 chart.setVisible(true);  
 });  
 }  
}

package org.example;  
import org.example.stratygy.\*;  
  
import java.util.ArrayList;  
import java.util.HashMap;  
import java.util.List;  
import java.util.Map;  
  
public class App {  
 public static void main(String[] args) {  
 List<Strategy> strategies = List.*of*(  
 new AggressiveStrategy(),  
 new DefensiveStrategy(),  
 new RandomStrategy(),  
 new BalancedStrategy()  
 );  
  
 Map<String, Map<String, double[]>> results = new HashMap<>();  
  
 for (Strategy agent1Strategy : strategies) {  
 for (Strategy agent2Strategy : strategies) {  
 String agent1Name = agent1Strategy.getClass().getSimpleName();  
 String agent2Name = agent2Strategy.getClass().getSimpleName();  
 int pair1Wins = 0;  
 int pair2Wins = 0;  
 int totalGames = 1000; // 1000 игр  
  
 for (int i = 0; i < totalGames; i++) {  
 int result = *game*(agent1Strategy, agent2Strategy);  
 if (result == 0) {  
 pair1Wins++;  
 } else {  
 pair2Wins++;  
 }  
 }  
  
 // Вычисляем процент побед  
 double pair1WinPercentage = (pair1Wins / (double) totalGames) \* 100;  
 double pair2WinPercentage = (pair2Wins / (double) totalGames) \* 100;  
  
 System.*out*.printf("Стратегия Агентов 1: %s, Стратегия Агентов 2: %s%n",  
 agent1Strategy.getClass().getSimpleName(),  
 agent2Strategy.getClass().getSimpleName());  
 System.*out*.printf("Пара 1 выиграла %.2f%%%n", pair1WinPercentage);  
 System.*out*.printf("Пара 2 выиграла %.2f%%%n", pair2WinPercentage);  
 System.*out*.println("-".repeat(50));  
  
 results  
 .computeIfAbsent(agent1Name, k -> new HashMap<>())  
 .put(agent2Name, new double[]{pair1WinPercentage, pair2WinPercentage});  
 }  
 }  
 StrategyComparisonChart.*init*(args, results);  
 }  
  
 public static int game(Strategy agent1Strategy, Strategy agent2Strategy) {  
 GameEnvironment env = new GameEnvironment();  
 List<Agent> agents = List.*of*(  
 new Agent(0, agent1Strategy), // Пара 1  
 new Agent(1, agent1Strategy),  
 new Agent(2, agent2Strategy), // Пара 2  
 new Agent(3, agent2Strategy)  
 );  
  
 env.deal();  
 int[] scores = {0, 0}; // Победные очки для пары 1 и пары 2  
  
 int currentAttacker = 0;  
 int minRank = Integer.*MAX\_VALUE*;  
 for (int i = 0; i < 4; i++) {  
 List<Card> hand = env.getHands().get(i);  
 for (Card card : hand) {  
 if (card.getSuit().equals(env.getTrump().getSuit()) && card.getRank() < minRank) {  
 minRank = card.getRank();  
 currentAttacker = i;  
 }  
 }  
 }  
  
 int defendingTeam = (currentAttacker + 2) % 4;  
  
 while (true) {  
 if (env.getHands().entrySet().stream()  
 .filter(entry -> entry.getKey() < 2)  
 .map(Map.Entry::getValue)  
 .allMatch(List::isEmpty)) {  
 return 0; // Победила пара 1  
 }  
 if (env.getHands().entrySet().stream()  
 .filter(entry -> entry.getKey() >= 2 && entry.getKey() < 4)  
 .map(Map.Entry::getValue)  
 .allMatch(List::isEmpty)) {  
 return 1; // Победила пара 2  
 }  
  
 Agent attacker = agents.get(currentAttacker);  
 Agent defender = agents.get(defendingTeam);  
 Card attackCard = attacker.makeMove(env, "attacker", null);  
  
 if (attackCard == null) {  
 return (currentAttacker < 2) ? 1 : 0;  
 }  
  
 Card defenderCard = defender.makeMove(env, "defender", null);  
 if (defenderCard == null) { // Не смог отбить  
 scores[defendingTeam / 2]++;  
 break;  
 } else {  
 env.moveToDiscardPile(); // Карты биты  
 }  
 }  
 return scores[0] > scores[1] ? 0 : 1;  
 }  
}

**Примеры вывода**



