Одеський національний університет імені І. І. Мечникова
Факультет математики, фізики та інформаційних технологій
Кафедра оптимального керування та економічної кібернетики

Дипломна робота

на здобуття ступеня вищої освіти «магістр»

на тему: «Побудова дерева рішень у багатокрокових іграх з неповною інформацією»

«Decision tree creation in multistage games with incomplete information»

Виконав: студент 2-го курсу магістратури денної форми навчання
спеціальності 113 Прикладна математика
Вдовін Кирило Юрійович

Керівник: д. фіз.-мат. наук, доц. Кічмаренко О. Д. _____
Рецензент: канд. фіз.-мат. наук, доц. Страхов Є. М.

Рекомендовано до захисту:

Протокол засідання кафедри

№ ___ від _____ 2021 р.

Завідувач кафедри

_____ _____

Захищено на засіданні ЕК № _____

Протокол № ___ від _____ 2021 р.

Оцінка _____ / _____ / _____

Голова ЕК

_____ _____

Одеса — 2021 р.

ЗМІСТ

ВСТУП

Гра з неповною інформацією - це деяка гра, в якій беруть участь гравці, які не мають точних знань про гру, у яку грають. Вони виникають в основному в економічних та політичних ситуаціях, де особливості середовища можуть не бути загальновідомими.

Актуальність обраної теми зумовлена широтою сфер застосування. Конфлікти n-сторін, які набувають вигляду безкоаліційних ігр найчастіше виникають в економіці, політиці і соціології. Прикладами таких ігор є взаємовідносини між фірмами на ринку, аукціони, торг, карточні ігри тощо.

Слід враховувати, що будь-яка дія в постійно мінливому середовищі спричиняє за собою реакцію інших гравців, залежно від їх стану та інформації, якою вони володіють. Таким чином, розглядаються багатоетапні ігрові стратегії за умови, що стани гравців можуть змінюватися з певною ймовірністю протягом гри. Якщо природа виступає одним із агентів, тобто може бути в ролі гравця, у якого зміна стану розглядається як стратегія, що реалізується з певною ймовірністю, то гра набуває нової якості - гра з повною недосконалою інформацією.

В цій роботі розглядаються методи побудови дерева станів для довільної динамічної гри в розгорнутому вигляді а також методи пошуку оптимальних стратегій у такому дереві.

Об'єктом дослідження є безкоаліційні конфлікти n-осіб. Предметом дослідження є моделі таких конфліктів у формі ігр n-гравців з ненульовою сумою з неповною інформацією та пошук оптимальних стратегій в таких іграх. Метою роботи є створення програмного комплексу для побудови дерев станів таких ігр в розгорнутій формі та пошуки оптимальних стратегій за побудованим деревом. Такий програмний комплекс дозволить абстрагувати логіку побудови дерева станів і пошуку оптимальної стратегії від деталей реалізації, що дозволить спростити процесс моделювання конфліктів n-сторін та процеси їх вирішення. Даний підхід є можливим якщо інформаційну модель гри можна задати множиною гравців, можливих ходів, множиною функцій виграшу і функцією розподілу ходів природи.

На момент написання диплому існують додатки, які довзоляють знайти хоча б одну, або декілька оптимальних стратегій для ігр в нормальній формі, але вони не підтримують введення даних гри в розгорнутій формі [?]. Розгорнута форма динамічної гри є більш зрозумілою для людини, оскільки її можна відобразити в вигляді дерева ходів. Отже науковою новизною диплому є створене програмне запезпечення, яке дозволяє задавати довільну динамічну гру в розгорнутій формі.

В дипломі використовуються наступні означення:

Фреймворк (англ. Framework) - інфрастрктурне програмне забезпечення, метою якого є спрощення реалізації та виконання деяких завдань, з якими стикається розробник в процесі створення програмного забезпечення.

Функція зворотного виклику - функція вищого порядку, або колбек(англ. callback function) є частиною виконуваного коду, що передається як аргумент до іншого коду, який має викликати цей код у відповідь, тобто виконати аргумент у певний момент часу.

Матеріали даної роботи були представлені на IX-ій міжнародній науково-практичній конференції «Results of modern scientific research and development», 14-16 листопада 2021 р., Мадрид [1]

ВИСНОВКИ

У рамках цієї роботи були проаналізовані способи побудови дерев рішень для ігр з повною та неповною інформацією.

Була створена бібліотека на мові Java, яка дозволяє будувати такі дерева для довільних ігр в розгорнутій формі та знаходити оптимальні стратегії на основі побудованого дерева.

Був створений графічний додаток з використанням технології Angular8 для візуалізації побудованого дерева.

Були розглянуті методи пошуку оптимальної стратегії для гри з неповною інформаією. Як приклад використання бібліотеки було реалізовано 2 гри з повною та неповною інформацією на основі побудованої системи абстракцій.

Були запропоновані методи оптимізації алгоритму побудови дерева станів, а саме

- Розпаралелювання алгоритму побудови станів
- Відкидання неперспективних гілок дерева

Була проаналізована продуктивність запропонованого алгоритму побудови дерева та залежність часу виконання від вхідних параметрів та реалізації гри.

Отримані результати дають можливість стверджувати що розроблене програмне забезпечення може бути використане для пошуку стретегій розв'язку конфліктів n-осіб у вигляді ігр з повною або неповною інформацією.

СПИСОК ЛІТЕРАТУРИ

1. Вдовін К. Ю. Decision tree creation in multi-stage games with incomplete information. - Madrid: Barca Academy Publishing, Мадрид, 2021. - С. 192-194
2. Циплаков А. А. Пособие по теории игр / А. А. Циплаков. [Електронний ресурс] - 2008. - С. 221. - Режим доступу: https://docplayer.com/26536386-Posobie-po-teorii-igr.html
3. P.R. Milgrom, R. J. Weber Distributional strategies for games with incomplete information, Mathematics of operations research, - 1985 - № 4 - С.619-632
4. J. C. Harsanyi, Papers in Game Theory, Springer Science+Business Media, B.V. - 1982 - С. 255
5. Rajiv Sethi, Jörgen Weibull, "WHAT IS...Nash Equilibrium" Notices of the AMS, Volume 63, №5 - 2016, С.526-528
6. Marek Mikolaj Kaminski, "Generalized Backward Induction: Justification for a Folk Algorithm" Department of Political Science and Mathematical Behavioral Sciences, University of California - 2019 - С.25
7. Rober W Rosenthal, "Games of Perfect Information, Predatory Pricing and the Chain-Store Paradox Bell Telephone Laboratories, Murray Hill, New Jersey - 1980 - С.9
8. Arnold Schwaighofer, "Tail Call Optimization in the Java HotSpot™ VM" Institut für Systemsoftware, Linz, März 2009, pp. 122
9. "sigma.js, a JavaScript library aimed at visualizing graphs of thousands of nodes and edges - 2013 - [Електронний ресурс] Режим доступу: https://github.com/jacomyal/sigma.js/blob/main/README.md
10. "graphology, robust multipurpose Graph object for JavaScript and TypeScript." - 2021 - [Електронний ресурс] Режим доступу https://graphology.github.io/
11. Israel Abramov, "JSON Serialization Libraries Performance Tests" - 2021 - [Електронний ресурс] - https://medium.com/justeattakeaway-tech/json-serialization-libraries-performance-tests-b54cbb3cccbb

12. Владислав И. Жуковский, Константин Н. Кудрявцев, "Уравновешивание конфликтов при неопределенности. I. аналог седловой точки" МТИП, 2013, С. 27–44

13. Г. Оуэн, "Теория игр" Рипол Классик - 1971 - С.232

14. Таха, Хемді, А., "Введение в исследовании операций, 6-е издание Вільямс, - 2001. - С.912.

ДОДАТОК А

Код алгоритму побудови дерева станів

```
    public Tree<DataNode>
    computeGame(AbstractGame game,
    int turns,
    Consumer<Tree<DataNode>>dataNodeConsumer) {
    String nodeId = UUID.randomUUID().toString();
    Tree<DataNode> gameTree = runGame(
    game, turns, dataNodeConsumer, nodeId);
    return gameTree;
}

private Tree<DataNode>
runGame(AbstractGame game,
  int turns,
  Consumer<Tree<DataNode>> dataNodeConsumer,
  String nodeId) {
    Tree<DataNode> gameTree = new Tree<>();
    gamesResults.put(nodeId, gameTree);
    DataNode initialGameState = DataNode.builder()
            .id(nodeId)
            .parentId(null)
            .turn("Initial game state")
            .player(null)
            .round(-1)
            .gameIsFinished(false)
            .nature(
            game.getPlayers().get(0)
            instanceof AbstractNature ?(AbstractNature)
            game.getPlayers().get(0)
                    : null)
```

```
                .game(game)
                .build();
        gameTree.addNode(initialGameState);
        dataNodeConsumer.accept(gameTree);
        computeGameRecursive(game, dataNodeConsumer, turns,
        nodeId, gameTree);
        return gameTree;
    }

    private void computeGameRecursive(
    AbstractGame game,
    Consumer<Tree<DataNode>> dataNodeConsumer,
    int turns, String parentId,
    Tree<DataNode> gameTree) {
        AbstractPlayer currentPlayer =
        game.getPlayers().get(game.getRound()
                % game.getPlayers().size());
        Set<Turn> possibleTurns =
        currentPlayer.getPossibleTurns(game);
        for (Turn possibleTurn : possibleTurns) {
            String nodeId = UUID.randomUUID().toString();
            Set<Income> incomes = null;
            AbstractGame copy = game.copy();
            possibleTurn.apply(copy);
            copy.nextRound();
            if (copy.gameIsFinished()) {
                incomes = copy.getPlayers().stream()
                        .filter(player -> !player.getTag()
                        .equals(AbstractNature.NAME))
                        .map(player -> new Income(
                        player.getTag(),
                        player.getIncome(copy)))
                        .collect(Collectors.toSet());
            }
```

```
            DataNode dataNode = DataNode.builder()
                    .id(nodeId)
                    .parentId(parentId)
                    .player(currentPlayer.getTag())
                    .turn(possibleTurn.getName())
                    .round(game.getRound())
                    .gameIsFinished(copy.gameIsFinished())
                    .incomes(incomes)
                    .nature(currentPlayer instanceof
                    AbstractNature ? (AbstractNature)
                    currentPlayer : null)
                    .game(game)
                    .build();
            gameTree.addNode(dataNode);
            dataNodeConsumer.accept(gameTree);

            if (!copy.gameIsFinished()
            && copy.getRound() < turns) {
                computeGameRecursive(copy,
                dataNodeConsumer,
                turns, nodeId, gameTree);
            }
        }
    }
```

ДОДАТОК Б

Код алгоритму пошуку гарантуючою стратегії

```
private Solution findMaxMin(TreeNode<DataNode> node) {
Solution bestSolution = null;
boolean nextTurnIsNature = false;
if (!node.getChildren().isEmpty()) {
    nextTurnIsNature = node.getChildren()
    .stream()
            .findFirst()
            .map(child -> child.getValue().getPlayer()
                    .equals(AbstractNature.NAME))
            .orElse(nextTurnIsNature);
}
if (nextTurnIsNature) {
    Map<String, Double> mathExpectedOutcomes
            = new HashMap<>();
    for (TreeNode<DataNode> child :
    node.getChildren()) {
        Solution solution;
        Map<String, Double> turnsProbabilities;
        turnsProbabilities = (node.getValue()
        .getNature())
          .getTurnsProbabilities(
          child.getValue().getGame());
        if (child.getValue().isGameIsFinished()) {
            solution = new Solution();
            child.getValue().getIncomes()
                    .forEach(income -> solution
                    .getOutcomes()
                            .put(income.getPlayer(),
```

```
                                    income.getValue()));
                } else {
                    solution = findMaxMin(child);
                }
                solution.getOutcomes()
                .forEach((player, outcome) -> {
                    if (!player.equals(AbstractNature.NAME)) {
                        if (mathExpectedOutcomes
                        .containsKey(player)) {
                            mathExpectedOutcomes.put(
                            player,
                            mathExpectedOutcomes.get(player)
                            + outcome * turnsProbabilities
                            .get(child
                            .getValue()
                            .getTurn()));
                        } else {
                            mathExpectedOutcomes
                            .put(player,
                            outcome * turnsProbabilities.get(
                            child.getValue().getTurn()));
                        }
                    }
                });
            }
            bestSolution = new Solution();
            bestSolution.setOutcomes(mathExpectedOutcomes);
        } else {
            for (TreeNode<DataNode> child :
            node.getChildren()) {
                Solution solution;
                if (child.getValue().isGameIsFinished()) {
                    solution = new Solution();
                    child.getValue()
```

```
                    .getIncomes()
                    .forEach(income ->
                    solution.getOutcomes()
                    .put(
                    income.getPlayer(),
                    income.getValue()));
                } else {
                    solution = findMaxMin(child);
                }
                if (bestSolution == null ||
                        solution.getOutcomes()
                        .get(child.getValue().getPlayer())
                        > bestSolution.getOutcomes().get(
                        child.getValue().getPlayer())) {
                    bestSolution = solution;
                    bestSolution.getTurns().addFirst(
                    new PlayerTurnPair(
                    child.getValue().getPlayer(),
                    child.getValue().getTurn()));
                }
            }
        }
        return bestSolution;
    }
```