Цель работы

Изучить метод обратной индукции и его применение к решению конечных позиционных игр с полной информацией. Изучить свойства решений таких игр.

Постановка задачи

Найти решение конечношаговой позиционной игры с полной информацией. Для этого сгенерировать и построить дерево случайной игры согласно варианту, используя метод обратной индукции, найти решение игры и путь (все пути, если он не единственный) к этому решению. Обозначить их на дереве.

Ход работы

В ходе выполнения лабораторной работы была разработана программа для построения дерева игры с заданными параметрами (глубина дерева, количество игроков, количество стратегий для игроков, диапазон значений выигрышей) и нахождения решения игры и путей, ведущих к этим решениям. Исходный код программы представлен в приложении А.

Параметры игры в соответствии с вариантом 7:

- Глубина дерева: 5;
- Количество игроков: 3;
- Количество стратегий: 3, 3, 3;
- Диапазон выигрышей: [0, 15].

На рисунке 1 показано случайно сгенерированное дерево игры с заданными параметрами. Внутри вершины может быть представлена следующая информация:

- номер вершины;
- игрок, делающий ход — A, B или C, — если вершина не является листом;
- выигрыши, если известны.

Цифры у ребер (0, 1 или 2) соответствуют номерам альтернатив для игрока, делающего ход.

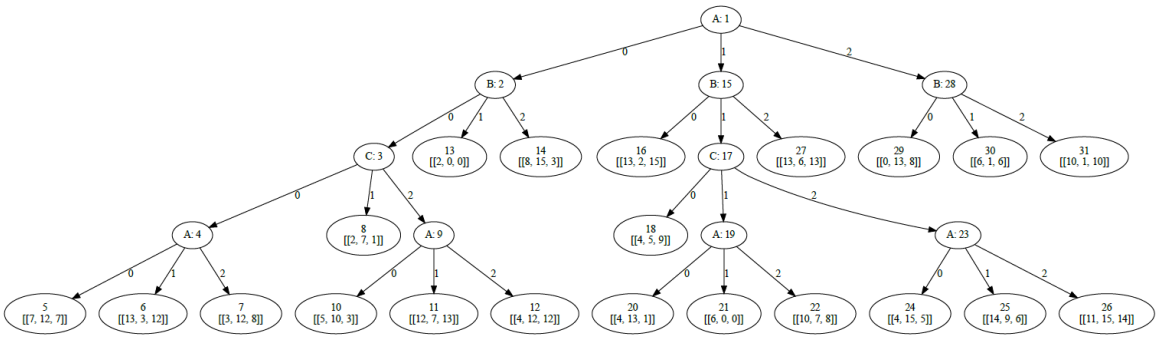

Рисунок 1 — Дерево игры

На рисунке 2 показано дерево игры после применения метода обратной индукции (у каждой вершины проставлены лучшие выигрыши). Было найдено решение с выигрышем [13, 6, 13]. Путь к этому решению, (1, 2), обозначен на дереве зеленым цветом.

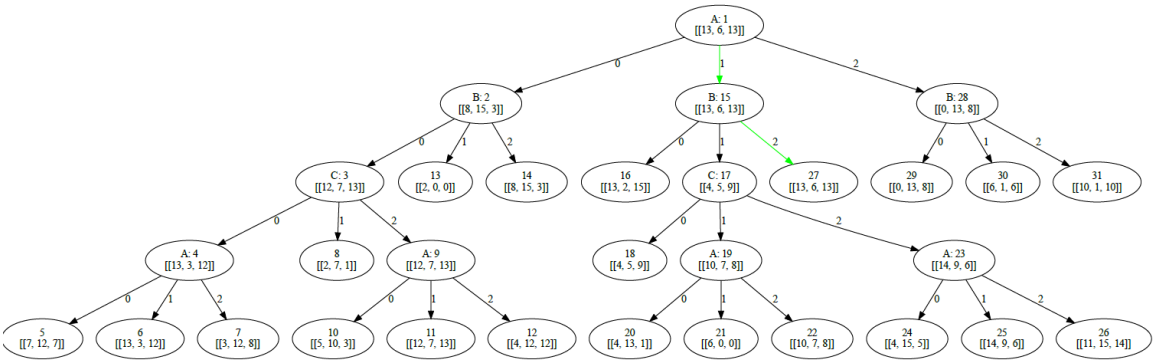

Рисунок 2 — Дерево игры после применения метода обратной индукции

Вывод

В ходе выполнения лабораторной работы была разработана программа для построения дерева игры с заданными параметрами (глубина дерева, количество игроков, количество стратегий для игроков, диапазон значений выигрышей) и нахождения решения игры и путей, ведущих к этим решениям.

Для построенной игры по методу обратной индукции было найдено решение [13, 6, 13] и путь (1, 2), ведущий к этому решению.

ПРИЛОЖЕНИЕ А

Листинг программы на языке Python

```
import numpy as np
import random
import os
os.environ["PATH"] += os.pathsep + 'C:/Program Files (x86)/Graphviz/bin/'

from graphviz import Digraph


def generate_color():
    """Генератор цветов"""

    colors = ['green', 'yellow', 'red', 'blue', 'pink', 'orange', 'magenta', 'cyan']
    for i in range(len(colors)):
        yield colors[i]


class Path:
    """
    Класс, реализующий путь.

    Атрибуты:
    gain - один из выигрышей корневой вершины дерева.  Путь строится от корня до листа,
                от которого "поднялся" этот выигрыш

    nodes - вершина пути
    color — цвет пути
    """

    def __init__(self, path_gain, initial_node, color):
        self.gain = path_gain
        self.nodes = [initial_node]
        self.color = color


class Player:
    """
    Класс, реализующий игрока.

    Атрибуты:
    name - имя игрока
    number_of_strategies - число стратегий игрока
    """

    def __init__(self, name, number_of_strategies):
        self.name = name
        self.number_of_strategies = number_of_strategies
```

```
class Node:
    """
    Класс, реализующий вершину дерева.

    Атрибуты:
    number — номер вершины в дереве
    player_number — номер игрока, который ходит в данной вершине
    parent — родительская вершина
    children — потомки данной вершины
    paths_to — список вершин-потомков, принадлежащих найденным путям
    depth — глубина, на которой находится вершина
    gains — выигрыши в данной вершине
    terminal — является ли вершина листом
    """

    def __init__(self, Tree, player_number, node_depth, prunning_depth, parent = None):
        Tree.number_of_nodes += 1
        self.number = Tree.number_of_nodes
        self.player_number = player_number
        self.parent = parent
        self.paths_to = []
        self.depth = node_depth

        # случайным образом определяем, делать ли вершину листом, если глубина, на которо
й она находится,
        # не меньше заданной глубины prunning_depth.
        self.terminal = random.randint(False, True) if node_depth >= prunning_depth else
False

        # вершина в любом случае является листом, если достигнута макс. глубина дерева
        if node_depth == Tree.max_depth:
            self.terminal = True

        # если вершина явл. листом, задаем ей случайный выигрыш
        if self.terminal:
            self.children = []
            self.gains = []
            random_gain = [random.randint(Tree.lowest_gain, Tree.highest_gain)
                    for i in range(Tree.players[player_number].number_of_strategies)]
            self.gains.append(random_gain)

        else:
        # если вершина не явл. листом
            next_player_number = player_number + 1 if player_number < len(Tree.players) -
 1 else 0
            self.gains = []
            self.children = []

            # создаем потомков
```

```
            for i in range(Tree.players[player_number].number_of_strategies):
                self.children.append(Node(Tree, next_player_number,
                                          node_depth + 1, prunning_depth, self))


    def print(self, Tree, with_paths=False):
        """Печать поддерева с корнем в текущей вершине"""

        if len(self.gains) > 0:
            if self.terminal:
                label = '{0}\n{1}'.format(self.number, self.gains)
            else:
                label = '{0}: {1}\n{2}'.format(Tree.players[self.player_number].name,
                                               self.number, self.gains)
        else:
            label = '{0}: {1}'.format(Tree.players[self.player_number].name, self.number)

        if with_paths:
            Tree.digraph_with_paths.node(str(self.number), label,
                                         _attributes={'color': 'black'})

            # связываем вершину с ее потомками
            for alternative_number, child in enumerate(self.children):
                child.print(Tree, with_paths)
                child_has_path = False

                # если вершина лежит на каком-то пути, то соединяем цветным ребром
                for path in Tree.paths:
                    if child in path.nodes:
                        child_has_path = True
                        color = path.color
                        Tree.digraph_with_paths.edge(str(self.number),
                                                     str(child.number),
                                                     label=str(alternative_number),
                                                     _attributes={'color': color})

                # если вершина не лежит ни на одном пути, то соединяем простым чер-
ным ребром
                if not child_has_path:
                    Tree.digraph_with_paths.edge(str(self.number),
                                                 str(child.number),
                                                 label=str(alternative_number))
        else:
            Tree.digraph.node(str(self.number), label, _attributes={'color': 'black'})
            # связываем вершину с ее потомками
            for alternative_number, child in enumerate(self.children):
                child.print(Tree)
                Tree.digraph.edge(str(self.number),
                                  str(child.number),
                                  label=str(alternative_number))
```

```
    def find_gain(self, Tree):
        """Найти выигрыш(и) для вершины"""

        # определим макс. выигрыш тек. игрока среди выигрышей всех потомков
        max_gain = 0
        for child in self.children:
            if len(child.gains) == 0: # если выигрыши потомка неизвестны, ищем их
                child.find_gain(Tree)

            # для каждого потомка определим макс. выигрыш тек. игрока
            # среди всех выигрышей тек. игрока данного потомка
            max_child_gain = 0
            for gain in child.gains:
                if gain[self.player_number] > max_child_gain:
                    max_child_gain = gain[self.player_num-
ber] # выигрыш потомка тек. игрока

            if max_child_gain > max_gain:
                max_gain = max_child_gain

        # добавляем все выигрыши потомков, у которых есть макс. выигрыш тек. игрока
        for child in self.children:
            for gain in child.gains:
                if gain[self.player_number] == max_gain:
                    for gain in child.gains:
                        self.gains.append(gain)
                    break


    def find_path(self, Tree):
        """Найти путь к листьям, выигрыши которых являются выигрышем корня дерева"""

        # если тек. вершина явл. листом, то путь проложен;
        if self.depth == Tree.max_depth:
            return

        # для каждого потомка проверяем, нет ли среди его выигрышей выигрыша пути
        for child in self.children:
            for child_gain in child.gains:
                if child_gain == Tree.paths[-
1].gain: # если есть - добавляем потомка в путь
                    Tree.paths[-1].nodes.append(child)
                    child.find_path(Tree) # продолжаем строить путь из потомка


class Tree:
    """
    Класс, реализующий дерево игры.
```

```
    Атрибуты:
    players - игроки
    max_depth — максимальная глубина дерева
    lowest_gain — минимально возможный выигрыш
    highest_gain — максимально возможный выигрыш
    number of nodes — кол-во вершин в дереве
    digraph — экземпляр класса Digraph из graphviz для отрисовки изначального дерева
    digraph_with_paths — экземпляр класса Digraph из graphviz для отрисовки дерева с найденным путями
    root — корневая вершина
    color_generator — генератор цветов для найденных путей
    paths — список всех найденных путей
    """

    def __init__(self, number_of_players, players_strategies_numbers,
                 lowest_gain, highest_gain, max_depth, prunning_depth):
        self.players = [Player(chr(ord('A') + i), players_strategies_numbers[i]) for i in range(number_of_players)]
        self.max_depth = max_depth
        self.lowest_gain = lowest_gain
        self.highest_gain = highest_gain
        self.number_of_nodes = 0
        self.digraph = Digraph(comment='Tree')
        self.digraph_with_paths = Digraph(comment='Tree with paths')
        self.root = Node(Tree=self, player_number=0, node_depth=1, prunning_depth=prunning_depth)
        self.color_generator = generate_color()
        self.paths = []


    def print(self, with_paths=False):
        """Печать дерева"""

        if with_paths:
            self.root.find_gain(self)
            self.find_paths()
            self.root.print(self, with_paths)
            self.digraph_with_paths.render('test-output/tree_with_paths.gv', view=True)
        else:
            self.root.print(self)
            self.digraph.render('test-output/tree.gv', view=True)


    def find_paths(self):
        """Найти пути"""

        for child in self.root.children:
            for child_gain in child.gains:
                if child_gain in self.root.gains:
```

```
                    # если корневой потомок имеет выигрыш, который есть в корне, то созда
ем новый путь
                    # каждый путь представляется в виде тройки: (выигрыш, цепь вер-
шин, цвет пути)
                    self.paths.append(Path(child_gain, child, next(self.color_generator))
)
                    child.find_path(self) # продолжаем выстраивать путь от потомка


def main():
    # случайное дерево с параметрами для 7 варианта
    random.seed()
    tree = Tree(number_of_players = 3,
                players_strategies_numbers = [3,3,3],
                lowest_gain = 0,
                highest_gain = 15,
                max_depth=5,
                prunning_depth=3
                )

    tree.print(with_paths=False)
    tree.print(with_paths=True)

main()
```