МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Балтийский государственный технический университет «ВОЕНМЕХ» им. Д.Ф. Устинова»
(БГТУ «ВОЕНМЕХ» им. Д.Ф. Устинова»)

Факультет И — Информационные и управляющие системы
(шифр / Наименование)
Кафедра И9 — Систем управления и компьютерных технологий
(шифр / наименование)
Дисциплина — Представление знаний в информационнных системах

Лабораторная работа №3
на тему «Поиск на игровых деревьях»
Вариант №3

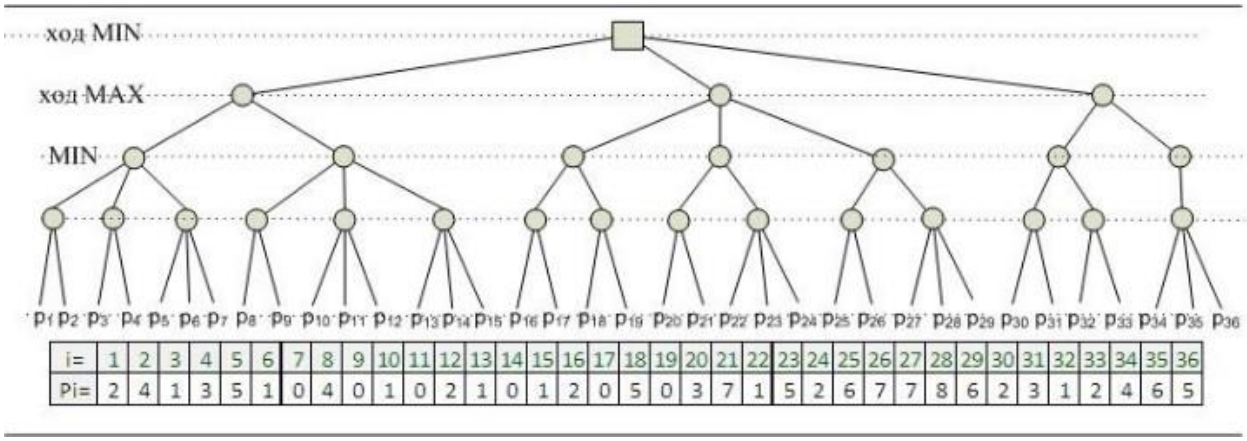
Выполнил студент группы И967
Васильев Н.А.
Фамилия И.О.

ПРЕПОДАВАТЕЛЬ
.
Фамилия И.О. Подпись

«_____» _____ 2019 г.

САНКТ-ПЕТЕРБУРГ
2019 г.

Задание: Разработать программную реализацию минимаксной процедуры с отсечениями на следующем игровом графе:



i=	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
Pi=	2	4	1	3	5	1	0	4	0	1	0	2	1	0	1	2	0	5	0	3	7	1	5	2	6	7	7	8	6	2	3	1	2	4	6	5

Необходимо:

- получить возвращенные оценки неконцевых вершин методом минимакса;
- получить возвращенные оценки неконцевых вершин методом отсечений, показать отсекаемые ветви дерева (альфа и бета).

Код программы на языке JavaScript:

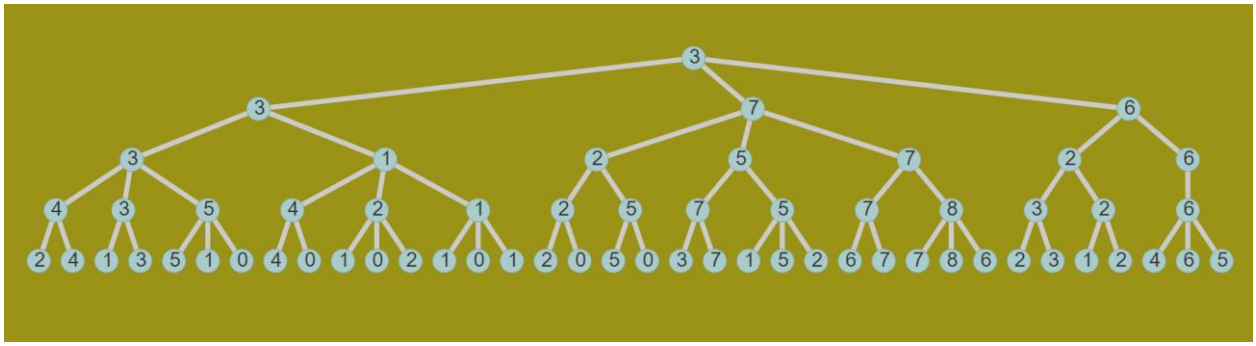
```
'use strict';

class Node {
    constructor(value=0, id=0, level=0) {
        this.parent = undefined;
        this.childrens = [];
        this.value = value;
        this.level = level;
        this.is_terminal_node;
        this.id = id;
        this.color = 0; //1 - alpha, 2 - beta, 0 - default;
    }

    add_child(value=0, id=0, level=0, is_terminal_node=false) {
        let child = new Node(value, id, level);
        child.parent = this;
        child.is_terminal_node = is_terminal_node;
        this.childrens.push(child);
        return child;
    }
}

class Tree {
    constructor() {
        this.vis_tree = null;
        this.vis_nodes = null;
        this.vis_edges = null;

        this.vis_tree_options = {
            nodes: { font: { size: 60 }, shape: 'circle'},
            edges: { width: 15 },
            layout: { hierarchical: { direction: "UD" } },
            physics: false
        };

        this.original_tree_data = [];
        this.root = new Node();
        this.nodes = {0: this.root};
        this.countID = 0;
```

```
            let json_data = [
                {"0": [0, 1, false]}, {"0": [0, 1, false]}, {"0": [0, 1, false]},
                {"1": [0, 2, false]}, {"1": [0, 2, false]},
                {"2": [0, 2, false]}, {"2": [0, 2, false]}, {"2": [0, 2, false]},
                {"3": [0, 2, false]}, {"3": [0, 2, false]},
                {"4": [0, 3, false]}, {"4": [0, 3, false]}, {"4": [0, 3, false]},
                {"5": [0, 3, false]}, {"5": [0, 3, false]}, {"5": [0, 3, false]},
                {"6": [0, 3, false]}, {"6": [0, 3, false]},
                {"7": [0, 3, false]}, {"7": [0, 3, false]},
                {"8": [0, 3, false]}, {"8": [0, 3, false]},
                {"9": [0, 3, false]}, {"9": [0, 3, false]},
                {"10": [0, 3, false]},

                {"11": [2, 4, true]}, {"11": [4, 4, true]},
                {"12": [1, 4, true]}, {"12": [3, 4, true]},
                {"13": [5, 4, true]}, {"13": [1, 4, true]}, {"13": [0, 4, true]},
                {"14": [4, 4, true]}, {"14": [0, 4, true]},
                {"15": [1, 4, true]}, {"15": [0, 4, true]}, {"15": [2, 4, true]},
                {"16": [1, 4, true]}, {"16": [0, 4, true]}, {"16": [1, 4, true]},
                {"17": [2, 4, true]}, {"17": [0, 4, true]},
                {"18": [5, 4, true]}, {"18": [0, 4, true]},
                {"19": [3, 4, true]}, {"19": [7, 4, true]},
                {"20": [1, 4, true]}, {"20": [5, 4, true]}, {"20": [2, 4, true]},
                {"21": [6, 4, true]}, {"21": [7, 4, true]},
                {"22": [7, 4, true]}, {"22": [8, 4, true]}, {"22": [6, 4, true]},
                {"23": [2, 4, true]}, {"23": [3, 4, true]},
                {"24": [1, 4, true]}, {"24": [2, 4, true]},
                {"25": [4, 4, true]}, {"25": [6, 4, true]}, {"25": [5, 4, true]},

            ];
            for (let i = 0; i < json_data.length; i++) {
                let data_chunk = Object.entries(json_data[i])[0];
                data_chunk[0] = Number(data_chunk[0]);
                this.original_tree_data.push(data_chunk);
            }
    }

    tree_reincarnated() {
        this.root.childrens = [];
        this.root.value = 0;
        this.root.level = 0;
        this.root.color = 0;
        this.countID = 0;
        this.nodes = {0: this.root};

        this.vis_tree = null;
        this.vis_nodes = new vis.DataSet();
        this.vis_edges = new vis.DataSet();

        this.vis_nodes.add({id: 0, label: ' 0', level: 0, color: '#e8e8e1', fixed: true});

        for (let node_data of this.original_tree_data) {
            this.add_node(node_data[1][0], node_data[0], node_data[1][1], node_data[1][2]);
        }

    }

    add_node(value, id, level, is_terminal_node=false) {
        this.countID++;
        this.nodes[this.countID] = this.nodes[id].add_child(value, this.countID, level,
is_terminal_node);
        this.vis_nodes.add({id: this.countID, label: ' ' + value.toString(), level: level, color:
'#acc', fixed: true});
        this.vis_edges.add({from: id, to: this.countID, color: {color: '#ccc'}, fixed: true});
    }

    draw() {
        let data = {nodes: this.vis_nodes, edges: this.vis_edges};
        this.vis_tree = new vis.Network(document.getElementById('tree_map'), data,
this.vis_tree_options);
    }

    tree_changes_applying(node) {
        for (let child of node.childrens) {
            if (child.color == 0) {
                this.vis_nodes.update({id: node.id, label: ' ' + node.value.toString(), level:
node.level, color: '#acc', fixed: true});
```

```
                this.vis_nodes.update({id: child.id, label: ' ' + child.value.toString(), level:
child.level, color: '#acc', fixed: true});
            }
            else if (child.color == 1) {
                this.vis_nodes.update({id: child.id, label: ' ' + child.value.toString(), level:
child.level, color: 'orange', fixed: true});
                this.vis_edges.update({from: node.id, to: child.id, color: {color: 'orange'},
fixed: true});
            }
            else if (child.color == 2) {
                this.vis_nodes.update({id: child.id, label: ' ' + child.value.toString(), level:
child.level, color: '#06b5cc', fixed: true});
                this.vis_edges.update({from: node.id, to: child.id, color: {color: '#06b5cc'},
fixed: true});
            }
            this.tree_changes_applying(child);
        }
    }

    minimax(from_node=undefined) {
        function min_move(from_node) {
            if (from_node.is_terminal_node) return from_node.value;
            let min = +Infinity;
            for (let child of from_node.childrens) {
                if (child.value < min && child.color == 0) {min = child.value;}
            }
            if (min == +Infinity) {return from_node.value;}
            else {return min;}
        }

        function max_move(from_node) {
            if (from_node.is_terminal_node) return from_node.value;
            let max = -Infinity;
            for (let child of from_node.childrens) {
                if (child.value > max && child.color == 0) {max = child.value;}
            }
            if (max == -Infinity) {return from_node.value;}
            else {return max;}
        }

        if (from_node == undefined) {from_node = this.root;}
        for (let child of from_node.childrens) {
            this.minimax(child)
        }
        //max-min
        if (from_node.level % 2) {from_node.value = max_move(from_node);}
        else {from_node.value = min_move(from_node);}
    }

    minimax_optimization(from_node, alpha=-Infinity, beta=+Infinity) {
        function prune(from_node, child=undefined, is_alpha_pruning) {
            let start;
            child == undefined ? start = -1 : start = from_node.childrens.indexOf(child);
            for (let node of from_node.childrens.slice(start+1)) {
                prune(node, undefined, is_alpha_pruning);
                if (is_alpha_pruning) node.color = 1;
                else node.color = 2;
            }
        }
        if (from_node == undefined) from_node = this.root;
        //max-min
        if (!(from_node.level % 2)) {
            if (from_node.is_terminal_node) return from_node.value;
            let bestValue = +Infinity;
            for (let child of from_node.childrens) {
                let value = this.minimax_optimization(child, alpha, beta);
                bestValue = Math.min(bestValue, value);
                beta = Math.min(beta, bestValue);
                from_node.value = bestValue;
                if (beta < alpha) {
                    prune(from_node, child, true);
                    break;
                }
            }
            return bestValue;
        } else {
            if (from_node.is_terminal_node) return from_node.value;
            let bestValue = -Infinity;
            for (let child of from_node.childrens) {
```

```
                let value = this.minimax_optimization(child, alpha, beta);
                bestValue = Math.max(bestValue, value);
                alpha = Math.max(alpha, bestValue);
                from_node.value = bestValue;
                if (beta < alpha) {
                    prune(from_node, child, false);
                    break;
                }
            }
            return bestValue;
        }
    }
}


var tree = new Tree();
function button_minimax() {
    tree.tree_reincarnated();
    tree.minimax();
    tree.tree_changes_applying(tree.root);
    tree.draw();
}
function button_prunings() {
    tree.tree_reincarnated();
    tree.minimax_optimization();
    tree.tree_changes_applying(tree.root);
    tree.draw();
}
button_minimax();
```

Результат работы метода Minimax:



Результат работы метода Альфа-Бета отсечений: