|  |  |
| --- | --- |
| *voenmeh* | МИНОБРНАУКИ РОССИИ  федеральное государственное бюджетное образовательное учреждение  высшего образования  **«Балтийский государственный технический университет «ВОЕНМЕХ» им. Д.Ф. Устинова»**  **(БГТУ «ВОЕНМЕХ» им. Д.Ф. Устинова»)** |
| БГТУ.СМК-Ф-4.2-К5-02 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Факультет |  | О |  | Естественнонаучный |
|  |  | шифр |  | наименование |
| Кафедра |  | О7 |  | Информационные системы и программная инженерия |
|  |  | шифр |  | Наименование |
| Дисциплина |  | Представление знаний в ИС | | |

Лабораторная Работа №3.2

на тему

|  |
| --- |
|  |
| Реализация минимаксного алгоритма с помощью альфа бета отсечения |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Выполнил студент группы И594  Рахимбердиев Т.А. | | | | |
| Фамилия И.О. | | | | |
| **РУКОВОДИТЕЛЬ** | | | | |
| Рохлин Н.С. | |  |  | |
| Фамилия И.О. Подпись | | | | |
| Оценка |  | | |  |
| «\_\_\_\_» |  | | | 2022 г. |

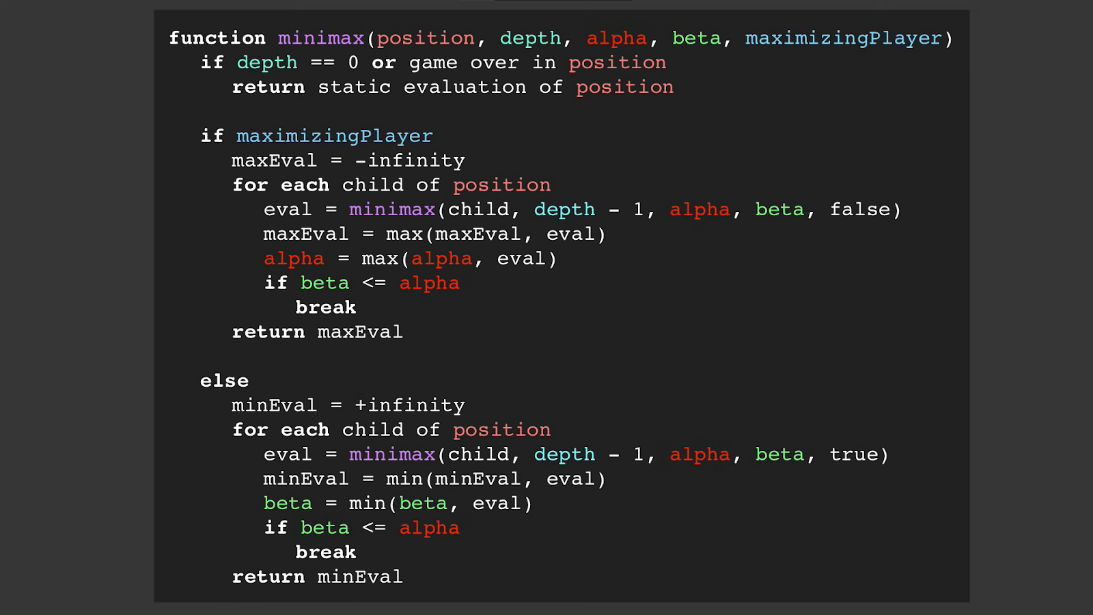
САНКТ-ПЕТЕРБУРГ

2022 г.

**Цель работы:** Разработать программу, реализующую минимаксный алгоритм.

**Краткая опись работы**

Для разработки программы крестики-нулики, с одной стороны пользователь с иной ai (adaptive intelligence) применен минимаксный алгоритм с альфы бета отсечением, реализованный на JavaScript/HTML. Для стилистической части интерфейса использован css.



Скриншот 1 -   Логика – альфа беты отсечения

**Отчет**

Реализована программа игры в “крестики нулики”. Игрок играет против ai то есть адаптивного интеллекта, используя минимакс алгоритм с альфа бета отсечением чтобы найти лучший логический ход. Игрок начинает с “X” а ai ходит “O”. Здесь невозможно выиграть игру. Можно рассчитывать только на ничью.

Минимакс — это правило принятия решений, используемое в искусственном интеллекте, теории принятия решений, теории игр, статистике и философии для минимизации возможных потерь при наихудшем сценарии. Когда речь идет о прибыли, это называется «максимум» - максимизировать минимальную прибыль.

Лабораторная работа направлена на понимание основ минимаксного алгоритма с применением альфа-бета отсечения.

Это достигается на основе рейтинговой системы, где ИИ фактически наказывается каждый раз, когда делает неправильный ход, и вознаграждается каждый раз, когда делает правильный ход. Рекурсия используется для прохождения всех возможных ходов.

С увеличением количества возможных ходов временная сложность возрастает экспоненциально. Этого можно избежать с помощью отсечения альфа-бета.

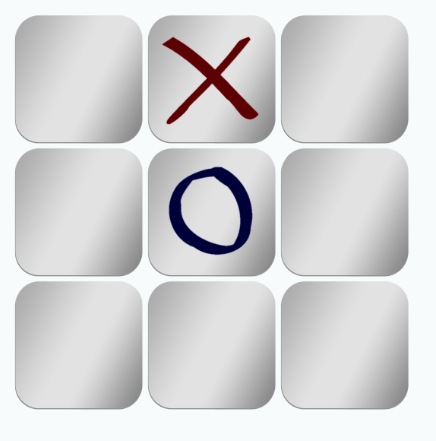


Рисунок 1.1 – Первый ход

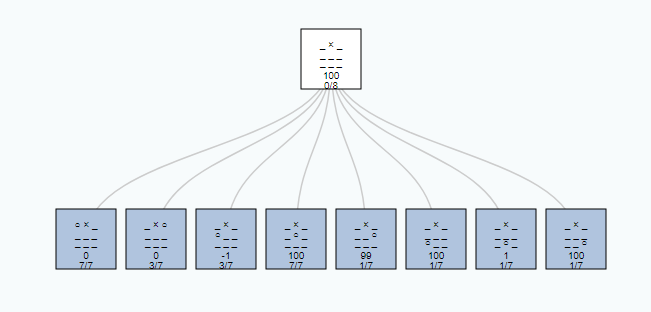


Рисунок 1.2 – Дерево состояния после первого хода

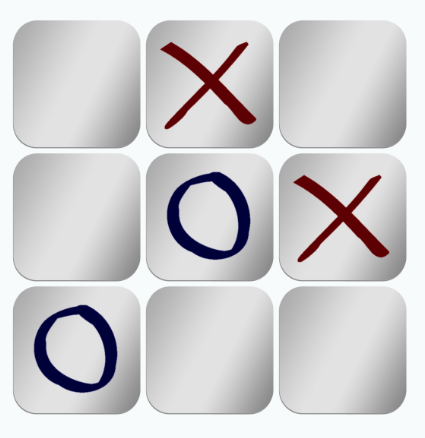


Рисунок 2.1 – Второй ход

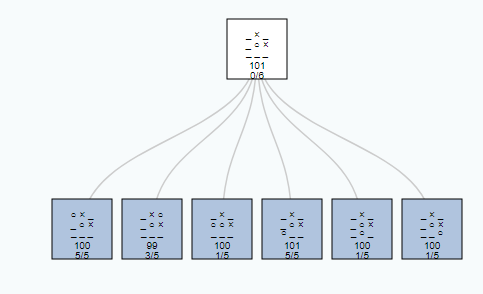


Рисунок 2.2 – Дерево состояния после второго хода

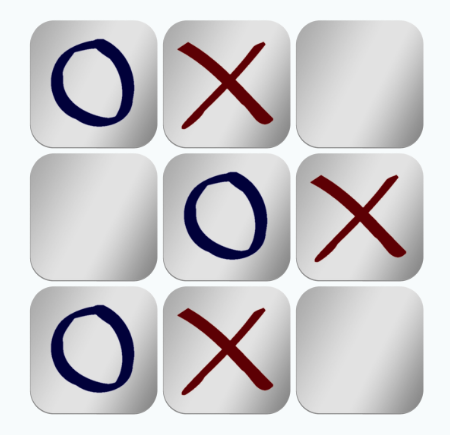


Рисунок 3.1 – Третий ход

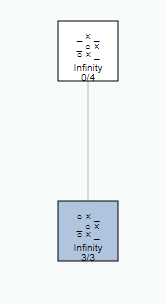


Рисунок 3.2 – Дерево состояния после третьего хода

**Альфа Бета отсечение используется следующим кодом.**

            for (let i = 0; i < curState.length; i++) {

                if (curState[i] != this.opponentSymbol && curState[i] != this.symbol) {

                    // Get next state

                    let nextState = Array.from(curState);

                    nextState[i] = this.symbol;

                    let result = this.minimax(nextState, -player, depth - 1, α, ß);

                    // Push state tree node

                    if (result[2] != undefined)

                        curNode.children.push(result[2]);

                    else

                        alert('PUSH filed');

                    // Update player level

                    if (result[0] > value) {

                        value = result[0];

                        move = nextState;

                        position = i;

                    }

                    α = Math.max(α, value);

                    if (α >= ß)

                        break;

                }

            }

            // Update tree node heuristic value

            curNode.heuristic = value;

            // Return value, move, tree branch, position  of move

            return [value, move, curNode,position];

        } else { // if MIN plays

            let value = Number.POSITIVE\_INFINITY;

            let move = curState;

            // For tree drawing

            let curNode = this.getJsonNode(curState, value);

            let position = 0;

            // Go through al possible moves for this player

            for (let i = 0; i < curState.length; i++) {

                if (curState[i] != this.opponentSymbol && curState[i] != this.symbol) {

                    // Get next state

                    let nextState = Array.from(curState);

                    nextState[i] = this.opponentSymbol;

                    let result = this.minimax(nextState, -player, depth - 1, α, ß);

                    // Push state tree node

                    if (result[2] != undefined)

                        curNode.children.push(result[2]);

                    else

                        alert('PUSH filed');

                    // Update player level

                    if (result[0] < value) {

                        value = result[0];

                        move = nextState;

                        position = i;

                    }

                    ß = Math.min(ß, value);

                    if (α >= ß)

                        break;

                }

            }

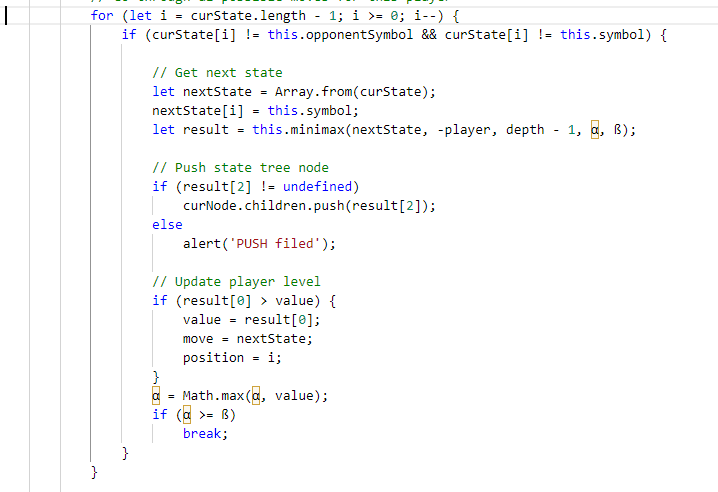
****

Рисунок 4.1 – Использование альфы беты отсечения (справа налево)

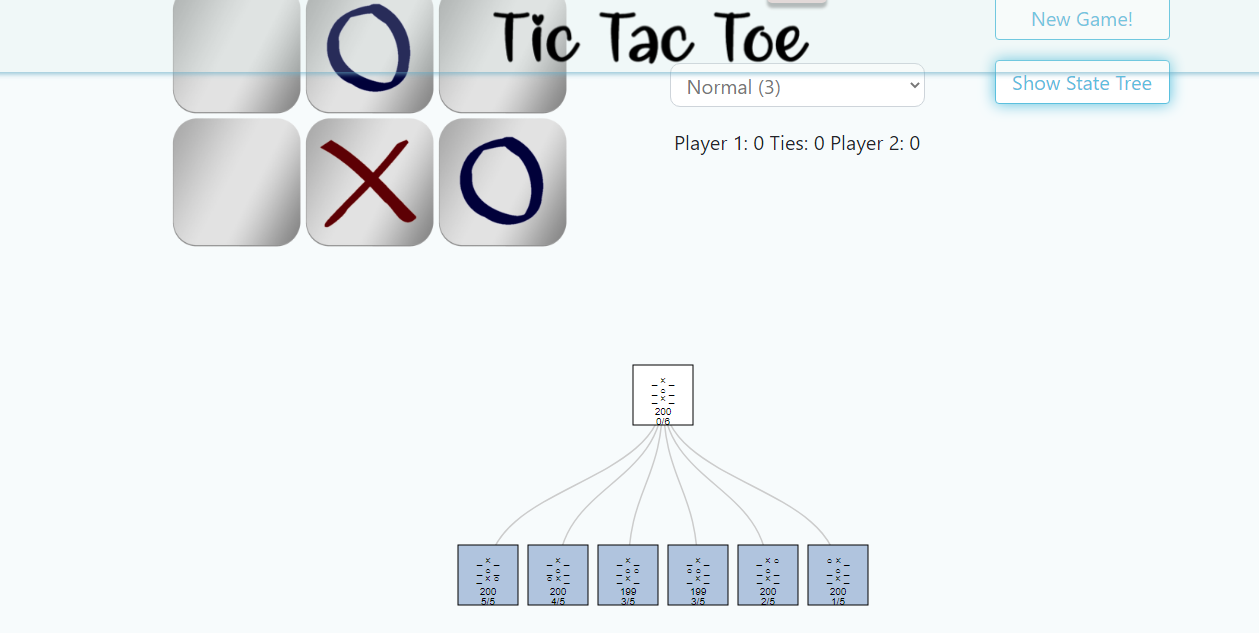


Рисунок 4.2 – Ходы игры с использованием альфы беты отсечения (справа налево)

**Приложение А**

**Bot.js**

/\*

\* Auto player implementation: minimax algorithm  with alfa-beta pruning

\*/

//

// Player

//

class Player{

    constructor(symbol, imgSrc){

        this.symbol = symbol;

        this.symbol\_ImgSrc = imgSrc;

    }

}

//

// Autoplayer

//

class GameBot extends Player{

    constructor(symbol, imgSrc, opponentSymbol, depth){

        super(symbol, imgSrc);

        this.opponentSymbol = opponentSymbol;

        this.depth = depth;

    }

    //

    // Get line heuristic estimation

    //

    estimateLine(firstCell, secondCell, thirdCell) {

        let score = 0;

        let line = [firstCell, secondCell, thirdCell];

        let countBotS = line.filter(item => item == this.symbol).length;

        let countHuS = line.filter(item => item == this.opponentSymbol).length;

        if (countHuS == 0 && countBotS != 0) {

            score = 1;

            if (countBotS > 1) {

                score = 100;

            }

        } else if (countHuS != 0 && countBotS == 0) {

            score = -1;

            if (countHuS > 1) {

                score = -100;

            }

        }

        if (countBotS > 2)

            score = Number.POSITIVE\_INFINITY;

        else if (countHuS > 2) {

            score = Number.NEGATIVE\_INFINITY;

            //alert('1');

        }

        return score;

    }

    //

    // Get state heuristic estination

    //

    getHeuriticEstimation(state) {

        let heuristic = 0;

        heuristic += this.estimateLine(state[0], state[1], state[2]); // row1 |

        heuristic += this.estimateLine(state[3], state[4], state[5]); // row2  |

        heuristic += this.estimateLine(state[6], state[7], state[8]); // row3   |

        heuristic += this.estimateLine(state[0], state[3], state[6]); // col1 —

        heuristic += this.estimateLine(state[1], state[4], state[7]); // col2 —

        heuristic += this.estimateLine(state[2], state[5], state[8]); // col3 —

        heuristic += this.estimateLine(state[0], state[4], state[8]); // diag1 \

        heuristic += this.estimateLine(state[2], state[4], state[6]); // diag2 /

        return heuristic;

    };

    //

    // Check if the state has winner or is a tie

    //

    isLeaf(curState, heuristic) {

        return (heuristic >= Number.MAX\_SAFE\_INTEGER || heuristic <= Number.MIN\_SAFE\_INTEGER) || (curState.every((element) => {

            return (element == this.opponentSymbol || element == this.symbol);

        }));

    }

    //

    // Get empty sells count

    //

    getFreeCells(state) {

        let counter = 0;

        state.forEach((element) => {

            if (element != this.opponentSymbol && element != this.symbol) {

                counter++;

            }

        });

        return counter;

    }

    //

    // Create tree node

    //

    getJsonNode(State, heuristic) {

        let state = Array.from(State);

        for (let i = 0; i < state.length; i++) {

            if (state[i] != this.opponentSymbol && state[i] != this.symbol) {

                state[i] = "\_";

            }

        }

        let row1 = "" + state[0] + " " + state[1] + " " + state[2];

        let row2 = "" + state[3] + " " + state[4] + " " + state[5];

        let row3 = "" + state[6] + " " + state[7] + " " + state[8];

        return {

            "row1": row1,

            "row2": row2,

            "row3": row3,

            "heuristic": heuristic,

            "childrenMax": this.getFreeCells(state),

            "children": []

        }

    }

    //

    // Minimax algorithm

    //

    minimax(curState, player, depth, α, ß) {

        // Get heuristic of the current state

        let h = this.getHeuriticEstimation(curState);

        // Return if it is a leaf

        if (this.isLeaf(curState, h) || depth == 0) {

            console.log([h, curState]);

            let curNode = this.getJsonNode(curState, h);

            return [h, curState, curNode];

        }

        // Check a player

        if (player > 0) {  // if MAX plays

            let value = Number.NEGATIVE\_INFINITY;

            let move = curState;

            // For tree drawing

            let curNode = this.getJsonNode(curState, value);

            let position = 0;

            // Go through al possible moves for this player

            for (let i = 0; i < curState.length; i++) {

                if (curState[i] != this.opponentSymbol && curState[i] != this.symbol) {

                    // Get next state

                    let nextState = Array.from(curState);

                    nextState[i] = this.symbol;

                    let result = this.minimax(nextState, -player, depth - 1, α, ß);

                    // Push state tree node

                    if (result[2] != undefined)

                        curNode.children.push(result[2]);

                    else

                        alert('PUSH filed');

                    // Update player level

                    if (result[0] > value) {

                        value = result[0];

                        move = nextState;

                        position = i;

                    }

                    α = Math.max(α, value);

                    if (α >= ß)

                        break;

                }

            }

            // Update tree node heuristic value

            curNode.heuristic = value;

            // Return value, move, tree branch, position  of move

            return [value, move, curNode,position];

        } else { // if MIN plays

            let value = Number.POSITIVE\_INFINITY;

            let move = curState;

            // For tree drawing

            let curNode = this.getJsonNode(curState, value);

            let position = 0;

            // Go through al possible moves for this player

            for (let i = 0; i < curState.length; i++) {

                if (curState[i] != this.opponentSymbol && curState[i] != this.symbol) {

                    // Get next state

                    let nextState = Array.from(curState);

                    nextState[i] = this.opponentSymbol;

                    let result = this.minimax(nextState, -player, depth - 1, α, ß);

                    // Push state tree node

                    if (result[2] != undefined)

                        curNode.children.push(result[2]);

                    else

                        alert('PUSH filed');

                    // Update player level

                    if (result[0] < value) {

                        value = result[0];

                        move = nextState;

                        position = i;

                    }

                    ß = Math.min(ß, value);

                    if (α >= ß)

                        break;

                }

            }

            // Update tree node heuristic value

            curNode.heuristic = value;

            // Return value, move, tree branch, position  of move

            return [value, move, curNode, position];

        }

    }

    //

    // Perform a bot move

    //

    botMove(curState) {

        let result = this.minimax(curState, Number.POSITIVE\_INFINITY, this.depth, Number.NEGATIVE\_INFINITY, Number.POSITIVE\_INFINITY);

        updateTree(result[2]);

        return result[3];

    }

}

**Список литературы**

1. Краткое описание алгоритма А\*
2. Краткое описание алгоритма «муравьиной колонии».
3. Учебное пособие: Толмачев С.Г. «Алгоритмы поиска в системах искусственного интеллекта». БГТУ 2012. см. глава 4 – Поиск на графе (параграф 7.4)