лабораторная работа 1  
"изучение алгоритмОВ Minimax, negmax и Alpha-beta отсечения"

**Цель работы**: знакомство с задачей поиска пути на дереве игры, реализация и ислледование алгоритмов Minimax, Negmax и Alpha-Beta отсечения.

**Задачи**:

– выбрать детерменированную игру для двух игроков с открытой информацией (шашки, крестики-нолики, "точки" и т.д.);

– описать состояние игры, ход игры в терминах структур данных выбранного языка реализации алгоритмов;

– придумать и реализовать следующие алгоритмы:

1) инициализация начального состояния игры;

2) получение списка всех возможных ходов для заданного состояния;

3) выполнение хода;

4) отмена хода;

5) проверка, что игрок выиграл/проиграл;

– придумать и реализовать как минимум одну оценочную функцию;

– изучить учебный пример – реализацию игры в крестики-нолики;

– реализовать алгоритмы Minimax, Negmax и Alpha-Beta (или использовать их реализацию из учебного примера);

– сравнить количество вершин дерева игры для трех алгоритмов и время поиска хода в зависимости от заданной грубины дерева;

– реализовать игру компьютера с самим собой или с человеком (или использовать реализацию из учебного примера).

Отчет по лабораторной работе должен содержать:

– описание игры и ее правил;

– описание состояние игры, ход игры в терминах структур данных выбранного языка реализации алгоритмов;

– описание алгоритма расчета оценочной функции;

– исходный код программы (код модулей учебного примера, которые использовались, но не изменялись, в отчет не включать).

# Описание учебного примера для игры в крестики-нолики 3х3

Учебный пример реализован в виде следующего набора модулей:

– base.py – модуль базового класса для описания состояния игры;

– xo.py – модуль класса для описания состояния игры в крестики-нолики;

– minimax.py – модуль для поиска лучшего хода по алгоритму minimax;

– negmax.py – модуль для поиска лучшего хода по алгоритму negmax;

– alpha\_beta.py – модуль для поиска лучшего хода по алгоритму alpha-beta отсечения.

Модуль базового класса *base.py* определяет набор методов, которые необходимо реализовать в классе-наследнике (для реализации конкретной игры), чтобы работали методы поиска в модулях *minimax*, *negmax*, *alpha\_beta*.

Пример реализации класса-наследника – в модуле *xo.py*.

В модулях minimax\_test.py, negmax\_test.py, alpha\_beta\_test.py реализованы примеры тестирования и запуска алгоритмов поиска для игры в крестики-нолики:

– в модуле minimax\_test.py производится поиск лучшего хода для первого игрока при глубине дерева равной 6;

– в модуле negmax\_test.py производится поиск лучшего хода, а также расчет времени поиска и строится график количества вершин дерева в зависимости от глубины поиска;

– в модуле alpha\_beta\_test.py производится расчет график количества вершин дерева в зависимости от глубины поиска и тестируется игра крестика против нолика.

При реализации своей игры можно использовать тестовые модули из учебного примера (заменив имена модуля и класса *state\_xo* на имена своих модуля и класса).

## Модуль base.py

# Базовый класс для описания состояния -

#   игровой ситуации.

# Класс должен уметь:

# - создавать список ходов

# - выполнять ход, изменяя текущее состояние

# - отменять ход

# - проверять, является ли состояние выигрышным

# - рассчитывать оценочную функцию

class state:

    def \_\_init\_\_(self, value):

        '''Конструктор класса, инициализация полей'''

        raise NotImplementedError

    def get\_moves(self, player):

        '''Получение списка ходов'''

        raise NotImplementedError

    def do\_move(self, move):

        '''Выполнение хода'''

        raise NotImplementedError

    def undo\_move(self, move):

        '''Отмена хода'''

        raise NotImplementedError

    def is\_win(self, player):

        '''Проверка, что игрок player выиграл'''

        raise NotImplementedError

    def score(self, player):

        '''Расчет оценочной функции'''

        raise NotImplementedError

## Модуль xo.py

from base import state

# Класс для описания состояния -

#   игровой ситуации при игре в крестики-нолики 3х3.

class state\_xo(state):

    '''Значение бесконечности для оценочной функции'''

    infinity = 100

    '''Возможный набор координат столбцов и строк'''

    lines = [

        [(0,0), (0,1), (0,2)], [(1,0), (1,1), (1,2)],

        [(2,0), (2,1), (2,2)], [(0,0), (1,0), (2,0)],

        [(0,1), (1,1), (2,1)], [(0,2), (1,2), (2,2)],

        [(0,0), (1,1), (2,2)], [(0,2), (1,1), (2,0)]]

    '''список игроков - крестик и нолик'''

    players = ["X", "0"]

    '''противники'''

    opponent = {"X":"0", "0":"X"}

    '''инициализация игрового состояния'''

    def \_\_init\_\_(self, value=None):

        # если value==None, то генерируем

        # пустое поле

        if value:

            self.value = value

        else:

            # иначе создаем список

            # [[None, None, None]

            #  [None, None, None]

            #  [None, None, None]]

            self.value = [[None for \_ in range(3)] for \_ in range(3)]

    '''представления состояния в виде строки'''

    def \_\_str\_\_(self):

        s = ""

        for row in self.value:

            for item in row:

                if item == None:

                    s += "[ ]"

                else:

                    s += f"[{item}]"

            s += '\n'

        return s

        # return str(self.value)

    '''выполнение хода'''

    def do\_move(self, move):

        # ход представляет собой кортеж

        # (row, col, player)

        # где

        # - row, col - координаты клетки,

        #              в которую ставится символ

        # - player - игрок, который делает ход

        row, col, player = move

        self.value[row][col] = player

    '''отмена хода'''

    def undo\_move(self, move):

        # ход представляет собой кортеж

        # (row, col, player)

        # где

        # - row, col - координаты клетки,

        #              в которую ставится символ

        # - player - игрок, который делает ход

        row, col, \_ = move

        # при отмене хода мы ставим в ячейку

        # пустое значение

        self.value[row][col] = None

    '''проверка на то, что ситуация выигрышная'''

    '''player - игрок, которого проверяем'''

    def is\_win(self, player):

        # по всем строкам, столбцам и диагоналям

        for line in state\_xo.lines:

            is\_win = True

            # проверяем, может ли игрок

            # заполнить линию

            for i, j in line:

                is\_win = is\_win and (self.value[i][j]==player)

            if is\_win:

                return True

        return False

    '''получить список ходов'''

    def get\_moves(self, player):

        # если ситуация выигрышная или проигрышная

        # то ходов нет

        if self.is\_win(player) or self.is\_win(self.opponent[player]):

            return []

        moves = []

        for row in range(3):

            for col in range(3):

                # добавляем в список ход

                # для каждой пустой клетки

                if self.value[row][col] == None:

                    moves.append((row, col, player))

        return moves

    '''вспомогательная функция для расчета оценочной функции'''

    '''подсчет количества строк, столбцов и диагоналей'''

    '''которые может заполнить игрок'''

    def nc(self, player):

        count = 0

        for (r1,c1),(r2,c2),(r3,c3) in state\_xo.lines:

            if self.value[r1][c1] != state\_xo.opponent[player] \

                and self.value[r2][c2] != state\_xo.opponent[player] \

                and self.value[r3][c3] != state\_xo.opponent[player]:

                count += 1

        return count

    '''оценочная функция'''

    def score(self, player):

        oppenent = state\_xo.opponent[player]

        # если выиграл игрок, то +бесконечность

        if self.is\_win(player):

            return state\_xo.infinity

        # если игрок проиграл, то -бесконечность

        elif self.is\_win(oppenent):

            return (-1)\*state\_xo.infinity

        else:

            # инача расчет разности функций nc

            return self.nc(player) - self.nc(oppenent)

## Модуль MiniMax

def minimax (state, level, original, player, opponent):

    ''' Алгоритм поиска лучше хода MiniMax

        - state - начальное состояние

        - level - максимальная глубина рекрсии (количество полуходов)

        - original - исходный игрок, для которого считается

                     дерево

        - player - текущий игрок

        - opponent - оппонент

    '''

    # инициализируем лучший ход и оценку

    best\_move, best\_score = None, None

    # получаем список возможных ходов

    moves = state.get\_moves(player)

    # если достигнута максимальная глубина дерева

    # или ходов нет, то рассчитываем оценку

    # при помощи оценочной функции

    if level == 0 or moves == []:

        return None, state.score(player)

    # перебираем последовательно все возможные ходы

    for m in moves:

        state.do\_move(m) # выполняем ход

        # вызываем рекурсивно MiniMax,

        # уменьшая уровень на 1

        # и меняя местами игрока и оппонента

        \_, score = minimax(state, level-1, original, opponent, player)

        state.undo\_move(m) # отменяем ход

        # для уровня Max выбираем узел с максимальной оценкой

        if player == original:

            if best\_score == None or score > best\_score:

                best\_move, best\_score =  m, score

        else:

        # для уровня Min выбираем узел с минимальной оценкой

            if best\_score == None or score < best\_score:

                best\_move, best\_score =  m, score

    return best\_move, best\_score

def bestmove(state, level, player, opponent):

    ''' Вызов функции MiniMax с начальными значениями

        - state - начальное состояние

        - level - максимальная глубина рекрсии (количество полуходов)

        - player - игрок

        - opponent - оппонент

    '''

    return minimax(state, level, player, player, opponent)

## Модуль NegMax

nodes = 0 # переменная для расчета статистики

def negmax(state, level, player, opponent):

    ''' Алгоритм поиска лучше хода NegMax

        - state - начальное состояние

        - level - максимальная глубина рекрсии (количество полуходов)

        - original - исходный игрок, для которого считается

                     дерево

        - player - текущий игрок

        - opponent - оппонент

    '''

    # инициализируем лучший ход и оценку

    best\_move, best\_score = None, None

    # получаем список возможных ходов

    moves = state.get\_moves(player)

    # накапливаем количество сгенерированных ходов

    global nodes

    nodes += len(moves)

    # если достигнута максимальная глубина дерева

    # или ходов нет, то рассчитываем оценку

    # при помощи оценочной функции

    if level == 0 or moves == []:

        return None, state.score(player)

    # перебираем последовательно все возможные ходы

    for m in moves:

        state.do\_move(m) # выполняем ход

        # вызываем рекурсивно NegMax,

        # уменьшая уровень на 1

        # и меняя местами игрока и оппонента

        \_, score = negmax(state, level-1, opponent, player)

        state.undo\_move(m) # отменяем ход

        # меняем знак оценочной функции на противоположный

        if best\_score == None or (-1)\*score > best\_score:

            best\_move, best\_score =  m, (-1)\*score

    return best\_move, best\_score

def bestmove(state, level, player, opponent):

    ''' Вызов функции MiniMax с начальными значениями

        - state - начальное состояние

        - level - максимальная глубина рекрсии (количество полуходов)

        - player - игрок

        - opponent - оппонент

    '''

    return negmax(state, level, player, opponent)

## Модуль Alpha\_Beta

from xo import state\_xo

# количество узлов для расчета статистики

nodes = 0

def alpha\_beta(state, level, player, opponent, low, high):

    ''' Алгоритм AlphaBeta-отсечения (на основе negmax)

        - state - начальное состояние

        - level - максимальная глубина рекрсии (количество полуходов)

        - player - текущий игрок

        - opponent - оппонент

        - low - нижняя граница для отсечения

        - high - верхняя граница для отсечения

    '''

    # инициализируем лучший ход и оценку

    best\_move, best\_score = None, None

    # получаем список возможных ходов

    moves = state.get\_moves(player)

    # накапливаем количество сгенерированных ходов

    global nodes

    nodes += len(moves)

    # если достигнута максимальная глубина дерева

    # или ходов нет, то рассчитываем оценку

    # при помощи оценочной функции

    if level == 0 or moves == []:

        return None, state.score(player)

    # перебираем последовательно все возможные ходы

    for m in moves:

        # выполняем ход

        state.do\_move(m)

        # вызываем рекурсивно NegMax,

        # уменьшая уровень на 1

        # и меняя местами игрока и оппонента

        \_, score = alpha\_beta(state, level-1, opponent, player, -high, -low)

        # отменяем ход

        state.undo\_move(m)

        # если текущий ход лучше best\_score:

        if best\_score == None or -1\*score > best\_score:

            # устанавливаем новое значение нижней границы

            low = -1\*score

            # лучшим ходом считаем текущий

            best\_move, best\_score =  m, -1\*score

        # если выполняется условие для AlphaBeta-отсечения

        if low >= high:

            return best\_move, best\_score

    return best\_move, best\_score

def bestmove(state, level, player, opponent):

    ''' Вызов функции AlphaBet с начальными значениями

        - state - начальное состояние

        - level - максимальная глубина рекрсии (количество полуходов)

        - player - игрок

        - opponent - оппонент

        - low = -infinity

        - high = +infinity

    '''

    return alpha\_beta(state, level, player, opponent, \

        -state\_xo.infinity, state\_xo.infinity)