МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ им. Р.Е.АЛЕКСЕЕВА

Институт радиоэлектроники и информационных технологий

Кафедра информатики и систем управления

Программа, играющая в Английские шашки 12x12

Отчет по курсовой работе

Этап №3

Вариант № 17

по дисциплине

Алгоритмы и структуры данных

РУКОВОДИТЕЛЬ:

_____ Капранов С.Н.

СТУДЕНТ:

_____ Сапожников В.О.

19-ИВТ-3

Работа защищена «___» _____

С оценкой _____

Нижний Новгород 2020

Содержание

1. Текст задания…….3
2. Правила игры…….4
3. Оценочная функция…….5
   3.1.Описание вычисления ОФ…….5
   3.2.Пример расчета ОФ…….6
   3.3.Код реализации ОФ……..8
4. Альфа-Бетта отсечение…….9
   4.1.Описание алгоритма Альфа-Бетта отсечения……9
   4.2.Пример расчета Альфа-Бетта отсечения……..10
   4.3.Код реализации алгоритма Альфа-Бетта отсечения……12
5. Список файлов проекта……15
6. Текст программы……..18
7. Результаты работы, принтринтскрины экранов…….59

1. Текст задания

Написать программу, играющую в Английские шашки на поле 12x12.

Замечание: стандартные Английские шашки играются на поле 8x8, данная модификация играется на поле 12x12 (Размер поля и расстановка шашек аналогичная Канадским шашкам).

2. Правила игры

Цель игры — уничтожить все шашки противника или лишить их возможности хода («запереть»).

Игра проводится на доске **12×12** клеток. В начальной позиции у каждого игрока по **30** шашек, расположенных на первых пяти рядах на черных клетках.

Первый ход делают белые шашки (Игрок - человек). Простая шашка ходит по диагонали вперёд на одну клетку.

При достижении любого поля последней горизонтали, простая шашка превращается в дамку. Дамка может ходить на одно поле по диагонали как вперёд, так и назад.

Взятие обязательно, если оно возможно. Шашки снимаются с доски лишь после того, как берущая шашка остановилась. При нескольких вариантах взятия игрок выбирает вариант взятия по своему усмотрению, и в выбранном варианте необходимо бить все доступные для взятия шашки. При взятии дамка побьет только через одно поле в любую сторону, а не на любое поле диагонали, как в русских или международных шашках.

3. Оценочная функция

3.1. Описание вычисления ОФ

Введем стоимость шашек: обычная шашка стоит 1000 очков, дамка стоит 4000 очков.

Поскольку в алгоритме альфа-бета отсечения компьютер будет пытаться заполучить максимальное значение от оценочной функции, а в ходе партии играть за «черную» команду, то формула примет вид:

$$f(board) = 1000(bc - wc) + 4000(bk - wk),$$

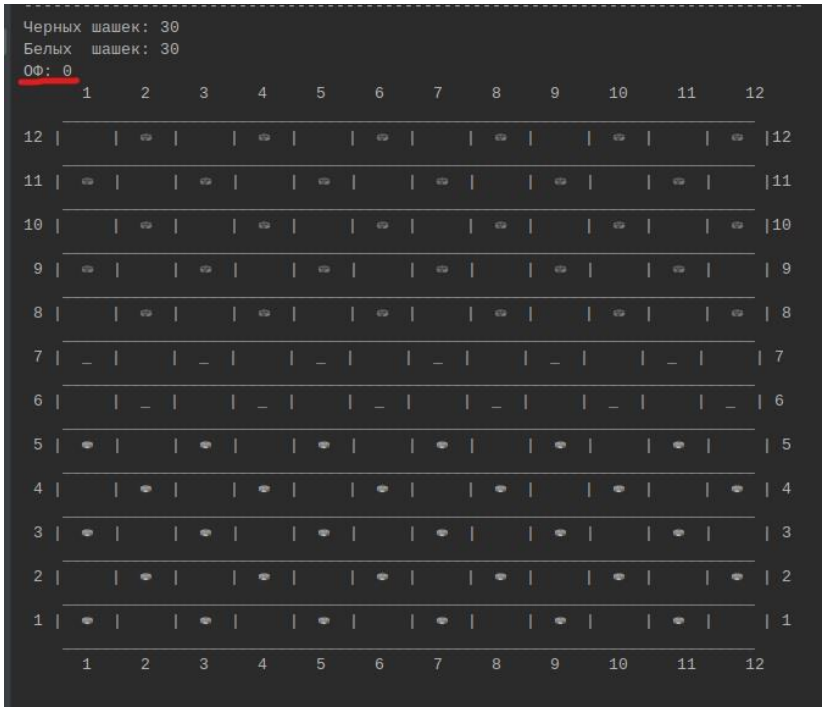
где bc – кол-во черных простых шашек, wc – кол-во белых простых шашек, bk – кол-во черных дамок, wk – кол-во белых дамок; *board* – переданное состояние игровой доски.

В программной реализации дабы не хранить отдельные значения для кол-ва белых и кол-ва черных дамок, отдельно кол-во обычных и каждый раз вычислять общее кол-во всех шашек каждой команды, было принято решение проводить перебор клеток поля (поле – массив клеток класса board), игнорируя недоступные (белые) клетки поля. Тогда оценочная функция примет вид:

$$f(board) = \sum_{i=0}^{11} \sum_{j=i\,\%2}^{11} x,$$

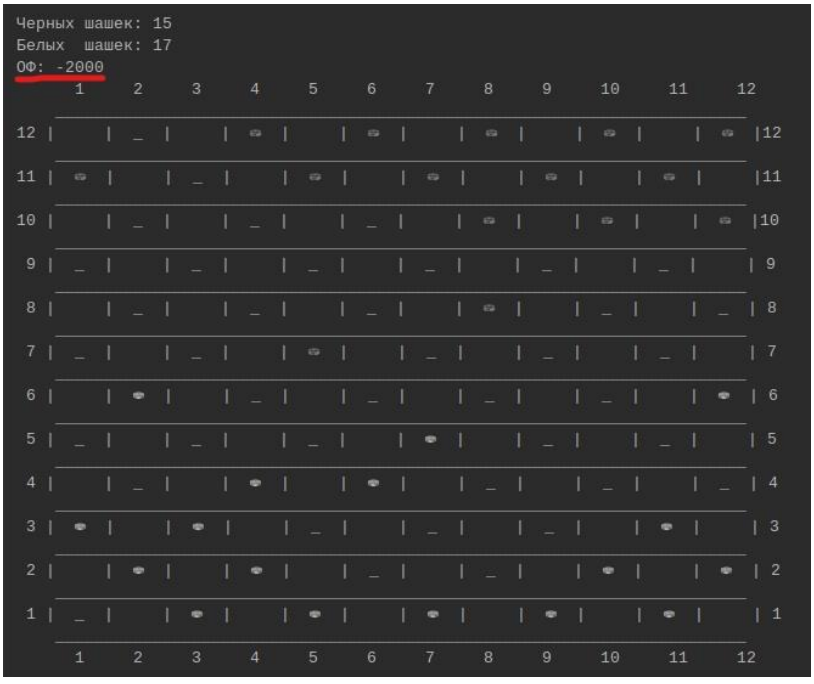
где x меняется по закону: $x = \begin{cases} 1000, \text{если в клетке черная шашка} \\ -1000, \text{если в клетке белая шашка} \\ 4000, \text{если в клетке черная дамка} \\ -4000, \text{если в клетке белая дамка} \end{cases}$

3.2. Пример расчета ОФ



Скриншот 1.

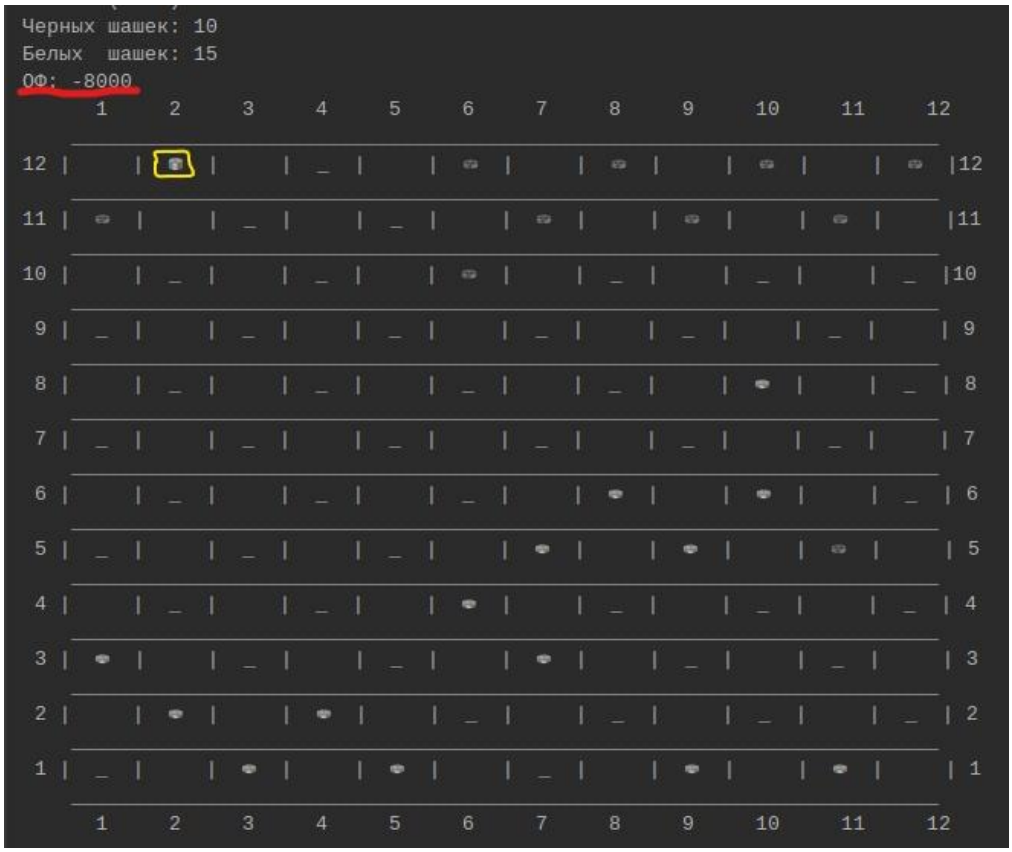
Начальное положение доски (скриншот 1), по 30 обычных шашек с каждой стороны. Условия игроков равнозначны, значение, возвращаемое оценочной функцией равно 0.



Скриншот 2.

На данном поле (скриншот 2) кол-во белых шашек(игрок) на 2 больше кол-во черных(компьютер). Значение, возвращаемое оценочной функцией:

$$1000(15 - 17) + 0 = -2 * 1000 = -2000$$



Скриншот 3.

В данном случае (скриншот 3) кол-во белых (игрок) превосходит кол-во черных (компьютер) на 5 шашек, но среди белых так же присутствует дамка (отмечена желтым цветом). Значение, возвращаемое оценочной функцией:

$$1000(10-14)+4000(0-1)=\ -4*1000+(-1)*4000=\ -8000$$

3.3.Код реализации ОФ

```
/**
 * Класс, в котором реализованы все методы
 * для оценки ходов.
 * */
public class MoveEvaluator {
    /**
     * Введем некоторую стоимость для шашек.
     */
    public final int POINT_KING   = 4000;
    public final int POINT_NORMAL = 1000;

    /**
     * Оценка поля
     */
    public int fieldAssessment(Board board) {
        int value = 0;

        // проход по доске.
        for (int r = 0; r < Board.rows; r++)
        {
            // проверка только доступных для хода столбцов
            for (int c = (r % 2 == 0) ? 0 : 1; c < Board.cols;
                 c += 2)
            {
                CellContents entry = board.cell[r][c];

                //Поскольку COMPUTER играет за черных
                //и пытается максимизировать alpha в алгоритме
                //альфа-бета отсечения, то ведется данная
                //расстановка знаков
                if (entry == CellContents.white) {
                    value -= POINT_NORMAL;
                } else if (entry == CellContents.whiteKing) {
                    value -= POINT_KING;
                } else if
                (entry == CellContents.black) {
                    value += POINT_NORMAL;
                } else if (entry == CellContents.blackKing) {
                    value += POINT_KING;
                }
            }
        }
        return value;
    }
}
```

4. Альфа-Бетта отсечение

4.1.Описание алгоритма Альфа-Бетта отсечения

Альфа-бета-отсечение — алгоритм поиска, стремящийся сократить количество узлов, оцениваемых в дереве алгоритмом **минимакса**.

Идея алгоритма заключается в том, что оценивание ветви дерева может быть досрочно прекращено, если было найдено, что для этой ветви значение **ОФ** в любом случае хуже, чем вычисленное для предыдущей ветви.

Alpha – текущее максимальное значение, меньше которого игрок максимизации (компьютер – черные шашки) никогда не выберет (изначально $-\infty$)

Betta – текущее минимальное значение, больше которого игрок минимизации (человек – белые шашки) никогда не выберет (изначально $+\infty$)

$$alpha = \max\big(alpha, f(Si)\big)$$
$$betta = \min\big(betta, f(Si)\big)$$

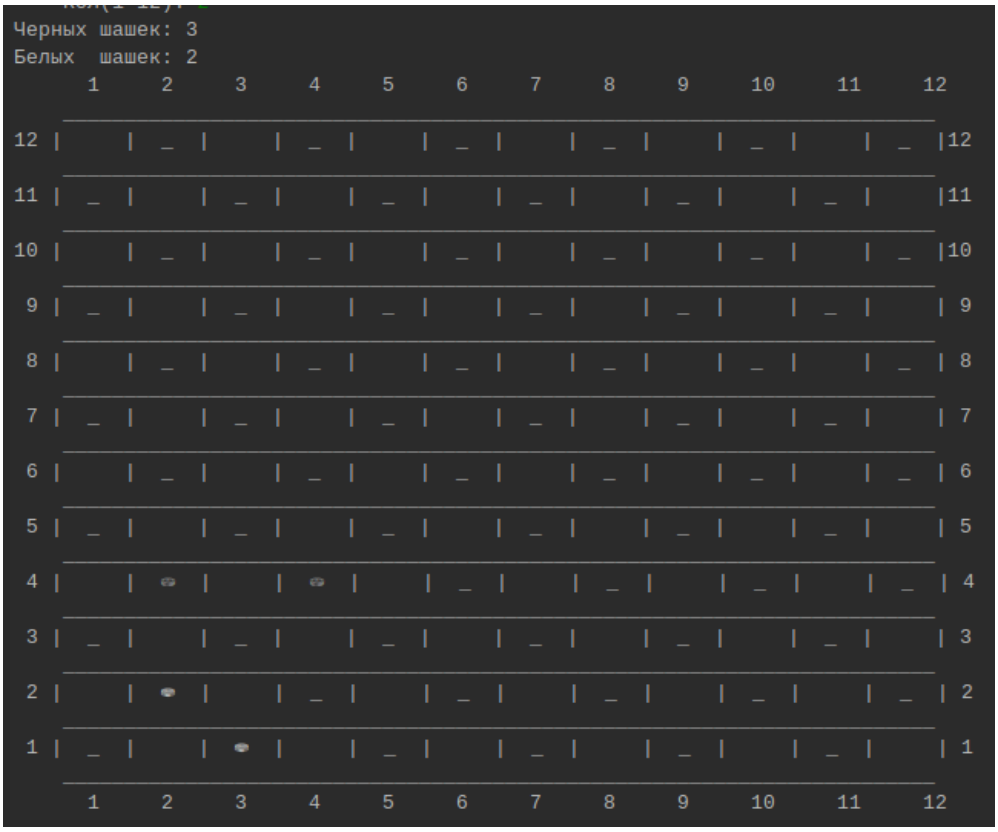
Представим alpha и betta как некий числовой промежуток $[\alpha;\ \beta]$. В ходе работы алгоритма значения alpha и beta будут изменятся, причем так, что данный промежуток будет уменьшается, т.е. betta из значения $+\infty$ стремится к наименьшему значению, а alpha из значения $-\infty$ стремится к наибольшему значению. При таких условиях может наступить ситуация, когда alpha становится больше betta, а сам промежуток $[\alpha;\ \beta]$ станет пустым множеством. Будем называть такую ситуация конфликтом интересов.

Как только наступает конфликт интересов анализ ветви данного поддерева прекращается, т.к. для этой ветви значение **ОФ** в любом случае хуже, чем вычисленное для предыдущей ветви.

4.2.Пример расчета Альфа-Бетта отсечения

Рассмотрим состояние поля, представленное на скриншоте 1. Ход делает компьютер (**MAX** – черные шашки).

Замечание: для рассмотрения алгоритма специально было выбрано такое состояние поля, при котором дерево минимакса имеет малое кол-во ветвей, что является более приемлемым для визуального восприятия.



Скриншот 4

Дерево, построенное алгоритмом минимакса для данного состояния поля, имеет вид (рис.1):

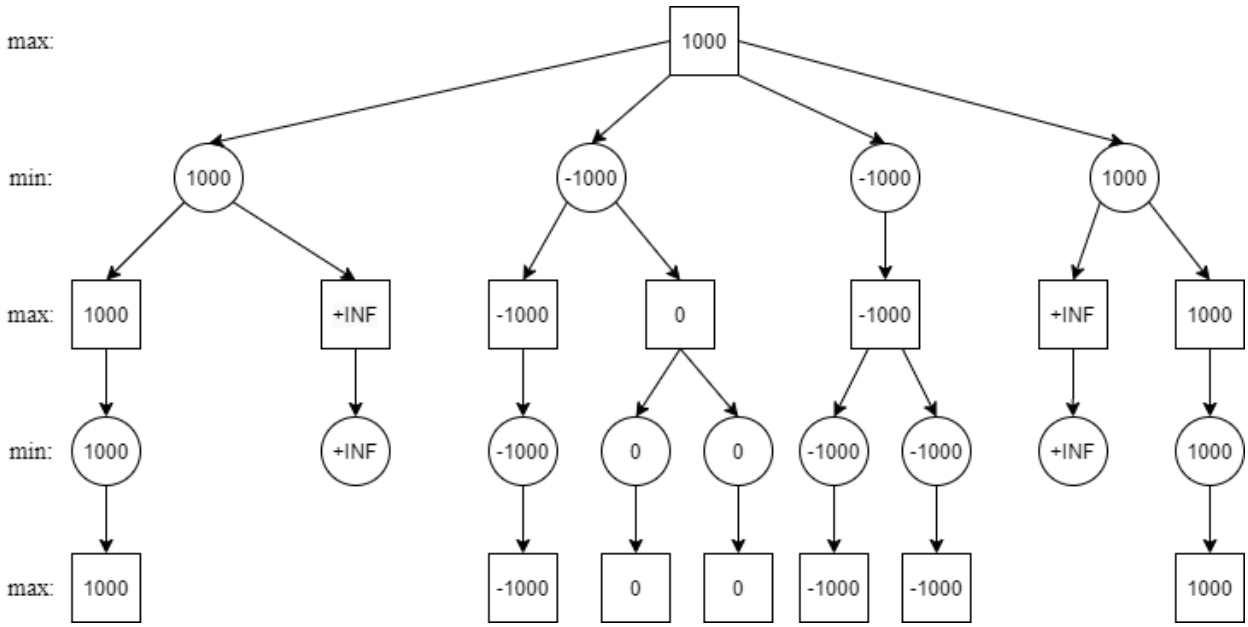


рис.1

Однако при помощи алгоритма альфа-бета отсечения данное дерево можно оптимизировать (рис.2):

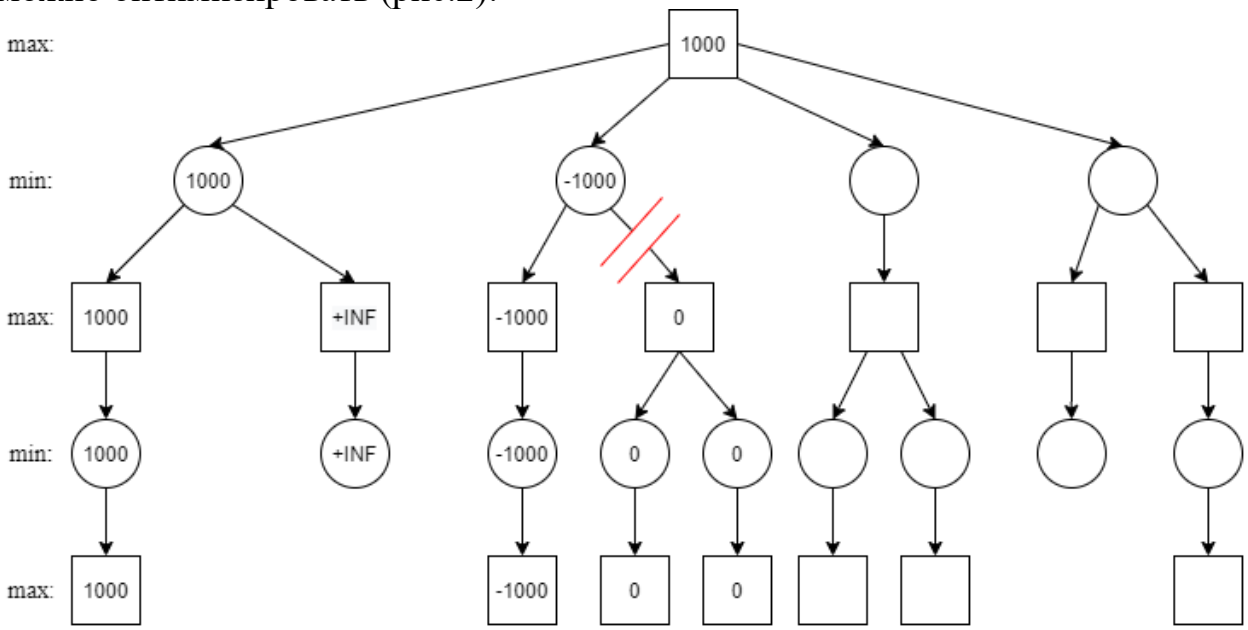


рис.2

В данной ситуации alpha для рассматриваемого поддерева равно 1000, а betta при рассматривании левой ветви равно -1000. При данных значениях возникает конфликт интересов, а значит рассмотрение правой ветви поддерева является бессмысленным и можно перейти к рассмотрению следующего поддерева.

Итоговое дерево будет иметь вид (рис.3):

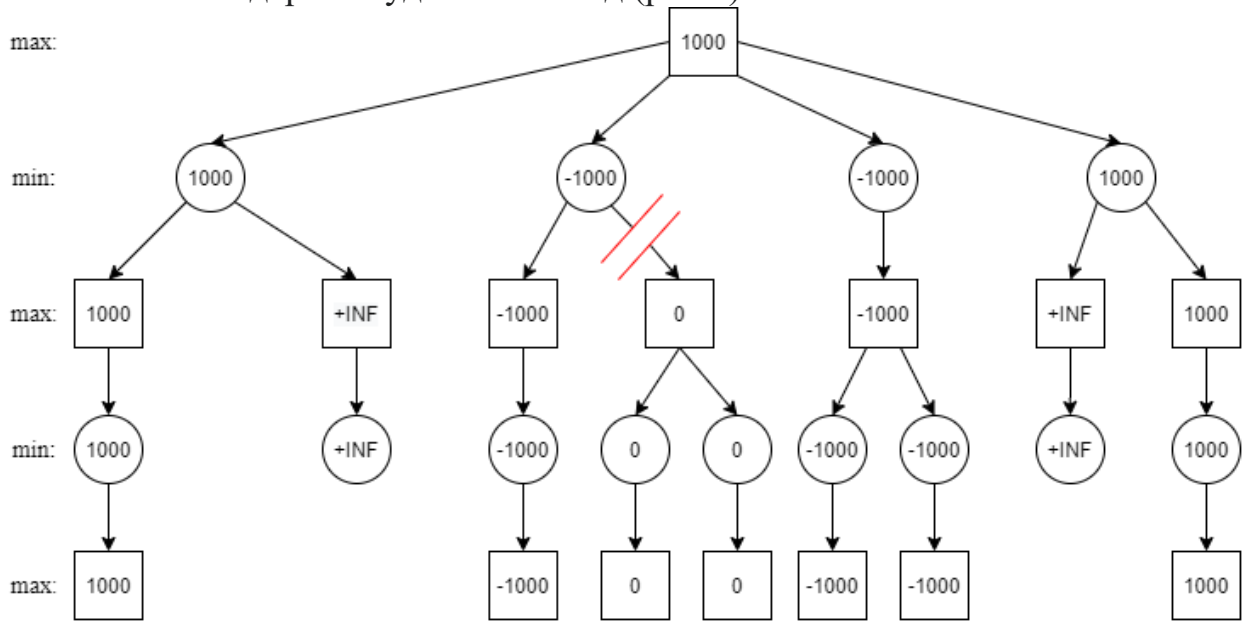


рис.3

4.3.Код реализации алгоритма Альфа-Бетта отсечения

```
/**
 * Алгоритм alphaBeta отсечений
 *
 * @param board         - текущее состояние доски
 * @param command       - 'команда' для которой обрабывается ход
 * @param depth         - текущий уровень глубины
 * @param alpha         - значение альфа
 * @param beta          - значение бета
 * @param resultMoveSeq - список результирующих ходов
 *
 * @return возвращает значение - оценка текущего,
 *         рассматриваемого положения
 * */
private static int alphaBeta(Board board, Command command,
     int depth, int alpha, int beta, Vector<Move> resultMoveSeq)
{
    //Если мы НЕ можем дальше строить дерево, то
    //оцениваем текущее положение
    if(!canExploreFurther(board, depth))
    {
        return oracle.fieldAssessment(board);
    }

    //создаем список списков возможных ходов и заполняем его
    Vector<Vector<Move>> possibleMoveSeq = expandMoves(board);

    //создаем список возможных состояний доски и заполняем его
    Vector<Board> possibleBoardConf= getPossibleBoardConf(board,
                                     possibleMoveSeq, command);

    //список лучших ходов
    Vector<Move> bestMoveSeq = null;

    if(command == Command.black)
    {
        for(int i=0; i<possibleBoardConf.size(); i++)
        {
            //получаем состояние доски из списка состояний
            Board b              = possibleBoardConf.get(i);

            //получаем список ходов из списка списков возможных
            //ходов
            Vector<Move> moveSeq = possibleMoveSeq.get(i);

            //получаем значение альфа-бета алгоритма
            int value = alphaBeta(b, Command.white, depth+1,
                                  alpha, beta, resultMoveSeq);



            //если полученное значение больше альфа,
            //то перезаписываем альфа
            //делаем данный список ходов лучшим
```

```
            if(value > alpha)
            {
                alpha = value;
                bestMoveSeq = moveSeq;
            }

            //если возникает конфликт интересов,
            //прерываем алгоритм
            if(alpha>beta)
            {
                break;
            }
        }

        //достигнута конечная глубина, то заносим ходы
        //из списка лучших ходов в список результирующих ходов
        if(depth == 0 && bestMoveSeq!=null)
        {
            resultMoveSeq.addAll(bestMoveSeq);
        }
        return alpha;
    }
    else if(command == Command.white)
        {
        for(int i=0; i<possibleBoardConf.size(); i++){

            //получаем состояние доски из списка состояний
            Board b              = possibleBoardConf.get(i);

            //получаем список ходов из списка списков возможных
            //ходов
            Vector<Move> moveSeq = possibleMoveSeq.get(i);

            //получаем значение альфа-бета алгоритма
            int value = alphaBeta(b, Command.white, depth+1,
                                  alpha, beta, resultMoveSeq);

            //если текущее значение меньше бета
            //то перезапись бета,
            //делаем данный список ходов лучшим
            if(value < beta)
            {
                bestMoveSeq = moveSeq;
                beta = value;
            }

            //если возникает конфликт интересов,
            //прерываем алгоритм
            if(alpha>beta)
            {
                break;
            }
        }
```

```
            //достигнута конечная глубина, то заносим
            //ходы из списка лучших ходов в список результирующих
            //ходов
            if(depth == 0 && bestMoveSeq!=null){
                resultMoveSeq.addAll(bestMoveSeq);
            }
            return beta;
        }
    }
```

5. Список файлов проекта

- Файл **Checkers.java**
  - Класс **Checkers**
    - Метод **main**

- Файл **Enumerations.java**
  - Перечисление **CellContents**
  - Перечисление **Command**
  - Перечисление **Owner**
  - Перечисление **MoveDir**

- Файл **Board.java**
  - Класс **Board**
    - Конструктор без параметров **Board**
    - Конструктор с параметрами **Board**
    - Метод **MakeMove**
    - Метод **CaptureBlackPiece**
    - Метод **CaptureWhitePiece**
    - Метод **genericMakeWhiteMove**
    - Метод **genericMakeBlackMove**
    - Метод **Display**
    - Метод **BoardCell**
    - Метод **DrawHorizontalLine**
    - Метод **DrawVerticalLine**
    - Метод **DisplayColIndex**
    - Метод **DisplayRowIndex**
    - Метод **duplicate**
    - Метод **CheckGameComplete**

- Файл **MoveEvaluator.java**
  - Класс **MoveEvaluator**
    - Метод **fieldAssessment**

- Файл **Move.java**
  - Класс **Move**
    - Конструктор с параметрами **Move**
    - Метод **conformity**
    - Метод **existsInVector**
    - Метод **display**

- Файл **Black.java**
  - Класс **Black**
    - Метод **Move**

        - Метод **ObtainForcedMovesForBlack**
        - Метод **CalculateAllForcedMovesForBlack**
        - Метод **CalculateAllNonForcedMovesForBlack**
        - Метод **ForwardLeftForBlack**
        - Метод **ForwardLeftCaptureForBlack**
        - Метод **ForwardRightForBlack**
        - Метод **ForwardRightCaptureForBlack**
        - Метод **BackwardLeftForBlack**
        - Метод **BackwardLeftCaptureForBlack**
        - Метод **BackwardRightForBlack**
        - Метод **BackwardRightCaptureForBlack**

- Файл **White.java**
    - Класс **White**
        - Метод **Move**
        - Метод **ObtainForcedMovesForWhite**
        - Метод **CalculateAllForcedMovesForWhite**
        - Метод **CalculateAllNonForcedMovesForWhite**
        - Метод **ForwardLeftForWhite**
        - Метод **ForwardLeftCaptureForWhite**
        - Метод **ForwardRightForWhite**
        - Метод **ForwardRightCaptureForWhite**
        - Метод **BackwardLeftForWhite**
        - Метод **BackwardLeftCaptureForWhite**
        - Метод **BackwardRightForWhite**
        - Метод **BackwardRightCaptureForWhite**

- Файл **UserInteractions.java**
    - Класс **UserInteractions**
        - Метод **getNextMove**
        - Метод **takeUserInput**
        - Метод **takeInput**
        - Метод **printSeparator**
        - Метод **displayMoveSeq**

- Файл **Computer.java**
    - Класс **Computer**
        - Метод **makeNextBlackMoves**
        - Метод **alphaBeta**
        - Метод **expandMoves**
        - Метод **expandMoveRecursivelyForBlack**
        - Метод **canExploreFurther**
        - Метод **getPossibleBoardConf**

- Файл **Human.java**
    - Класс **Human**
        - Метод **makeNextWhiteMoves**
        - Метод **printWarning**
        - Метод **checkValidMoveForWhiteHuman**

- Файл **Game.java**
    - Класс **Game**
        - Конструктор без параметров **Game**
        - Метод **whoWin**
        - Метод **PlayGame**
        - Метод **printGameReference**
        - Метод **printStartError**

6. Текст программы

Checkers.java

```
/**
 * Класс, содержащий точку входа в программу - метод  main.
 * Язык: java
 *
 * Реализация курсовой работы по диспилине: Алгоритмы и структуры
 * данных. Вариант№17.
 * Программа, играющая в Английские шашки 12x12
 *
 * @release: 12.12.20
 * @last_update: 12.12.20
 *
 * @author Vladislav Sapozhnikov 19-IVT-3
 */
public class Checkers
{
    /**
     * Точка входа в программу - функция main
     *
     * @param args - аргументы запуска
     * */
    public static void main(String[] args)
    {
        //если передан 1 аргумент: -h или --help
        //то вывод справки о программе
        if (args.length == 1 && (args[0].equals("-h") ||
            args[0].equals("--help")))
        {
            Game.printGameReference();
        }
        //если передан 1 аргумент: -start
        //то запуск игры
        else if (args.length == 1 && (args[0].equals("-start")))
        {
            Game game = new Game();
            game.PlayGame();
        }
        //во всех остальных случаях
        //запуска - выводим предупреждение
        else
        {
            Game.printStartError();
        }
    }
}
```

Enumerations.java

```
/**
 * Последовательность, отвечающая за
 * содеражание клеток.
 * */
enum CellContents
{
    inaccessible,    //недоступная клетка
    empty,           //пустая доступная клетка
    white,           //белая  шашка
    whiteKing,       //белая  дамка
    black,           //черная шашка
    blackKing        //черная дамка
}

/**
 * Последовательность, отвечающая за
 * 'цвета команд'.
 * */
enum Command {
    white,           //белая  команда
    black            //черная команда
}

/**
 * Последовательность, отвечающая за
 * 'игроков' команд (человек или COMPUTER)
 * */
enum Owner
{
    HUMAN,           //игрок - человек
    COMPUTER         //игрок - компьютер
}

/**
 * Последовательность, отвечающая за
 * направление хода (ориентация снизу в верх)
 * */
enum MoveDir
{
    forwardLeft,     //вперед - влево
    forwardRight,    //вперед - вправо
    backwardLeft,    //назад  - влево
    backwardRight    //назад  - вправо
}
```

Board.java

```
/**
 * Класс отвечающий за игровое поле
 * */
public class Board
{
    private final MoveEvaluator oracle = new MoveEvaluator();
    int blackCheckers;                //кол-во черных
    int whiteCheckers;                //кол-во белых

    static final int rows = 12;       //всего строк доски
    static final int cols = 12;       //всего колон доски
    CellContents[][]    cell;         //двумерный массив клеток

    /**
     * Конструктор по умолчанию.
     * Доска в начальном положении
     * */
    Board()
    {
        this.blackCheckers = 30;    //начальное кол-во черных
        this.whiteCheckers = 30;    //начальное кол-во белых

        //Инициализация клеток поля
        //CellContents.white   - в клетке белая шашка
        //CellContents.inaccessible - неиспользуемая клетка (в нее
        //невозможно сделать ход)
        //CellContents.empty   - пустая клетка (в нее возможно
        //сделать ход)
        //CellContents.black,  - в клетке черная шашка
        this.cell = new CellContents[][]
        {
            {
                 //описание массива 12x12 сокращено
            }
        };
    }

    /**
     * Конструктор копирования.
     * Необходим для построения COMPUTER
     * возможных ходов.
     *
     * @param board - доска, которую необходимо скопировать.
     * */
    Board(CellContents[][] board)
    {
        this.blackCheckers = this.whiteCheckers = 30;

        this.cell = new CellContents[rows][cols];
        for(int i = 0; i < rows; i++)
        {
            System.arraycopy(board[i], 0, this.cell[i], 0, cols);
```

```
        }
    }

    /**
     * Ход - перемещение шашки
     *       по полю.
     *
     * @param r1, c1 - начальная координата хода
     *
     * @param r2, c2 - конечная  координата хода
     * */
    public void MakeMove(int r1, int c1, int r2, int c2)
    {
        this.cell[r2][c2] = this.cell[r1][c1];
        this.cell[r1][c1] = CellContents.empty;

        //если шашка белая и достигла верхнего конца поля
        //она становится дамкой
        if(this.cell[r2][c2].equals(CellContents.white) &&
           r2==rows-1)
        {
            this.cell[r2][c2] = CellContents.whiteKing;
        }

        //иначе если шашка черная и достигла низа поля
        //она становится дамкой
        else if(this.cell[r2][c2].equals(CellContents.black) &&
                r2==0)
        {
            this.cell[r2][c2] = CellContents.blackKing;
        }
    }

    /**
     * Поедание черной шашки.
     *
     * @param r1, c1 - начальная координата хода
     *
     * @param r2, c2 - конечная координата хода
     * */
    public void CaptureBlackPiece(int r1, int c1, int r2, int c2)
    {
        //Определяем направление захвата
        MoveDir dir =
r2>r1?(c2>c1?MoveDir.forwardRight:MoveDir.forwardLeft)
                :(c2>c1?MoveDir.backwardRight:MoveDir.backwardLeft);

        //удаление черной шашки с доски
        switch (dir)
        {
            case forwardLeft   -> this.cell[r1 + 1][c1 - 1] =
                                  CellContents.empty;
            case forwardRight  -> this.cell[r1 + 1][c1 + 1] =
                                  CellContents.empty;
```

```
            case backwardLeft  -> this.cell[r1 - 1][c1 - 1] =
                                  CellContents.empty;
            case backwardRight -> this.cell[r1 - 1][c1 + 1] =
                                  CellContents.empty;
        }

        //уменьшение кол-ва черных шашек на поле
        this.blackCheckers--;

        //перемещение шашки по полю
        this.MakeMove(r1, c1, r2, c2);
    }

    /**
     * Поедание белой шашки.
     *
     * @param r1, c1 - начальная координата хода
     *
     * @param r2, c2 - конечная  координата хода
     * */
    public void CaptureWhitePiece(int r1, int c1, int r2, int c2)
    {
        //Определяем направление захвата
        MoveDir dir =
r2<r1?(c2<c1?MoveDir.forwardRight:MoveDir.forwardLeft)
                :(c2<c1?MoveDir.backwardRight:MoveDir.backwardLeft);

        //удаление белой шашки с доски
        switch (dir)
        {
            case forwardLeft   -> this.cell[r1 - 1][c1 + 1] =
                                  CellContents.empty;
            case forwardRight  -> this.cell[r1 - 1][c1 - 1] =
                                  CellContents.empty;
            case backwardLeft  -> this.cell[r1 + 1][c1 + 1] =
                                  CellContents.empty;
            case backwardRight -> this.cell[r1 + 1][c1 - 1] =
                                  CellContents.empty;
        }

        //уменьшение кол-ва белых шашек на поле
        this.whiteCheckers--;

        //перемещение шашки по полю
        this.MakeMove(r1, c1, r2, c2);
    }

    /**
     * Перебор возможных ходов человека.
     * Метод необходим для COMPUTER в момент проверки возможных
     * ходов.
     *
     * @param move - направление хода
     * */
```

```
public void genericMakeWhiteMove(Move move)
{
    int r1 = move.initialRow;
    int c1 = move.initialCol;
    int r2 = move.finalRow;
    int c2 = move.finalCol;

    if((Math.abs(r2-r1)==2 && Math.abs(c2-c1)==2))
    {
        CaptureBlackPiece(r1, c1, r2, c2);

    }
    else
    {
        MakeMove(r1, c1, r2, c2);
    }
}

/**
 * Перебор возможных ходов робота.
 * Метод необходим для COMPUTER в момент проверки возможных
 * ходов.
 *
 * @param move - направление хода
 * */
public void genericMakeBlackMove(Move move)
{
    int r1 = move.initialRow;
    int c1 = move.initialCol;
    int r2 = move.finalRow;
    int c2 = move.finalCol;

    if(Math.abs(r2-r1)==2 && Math.abs(c2-c1)==2)
    {
        CaptureWhitePiece(r1, c1, r2, c2);
    }
    else
    {
        MakeMove(r1, c1, r2, c2);
    }

}

/**
 * Прорисовка поля.
 * */
public void Display()
{
    System.out.println("Черных шашек: " + blackCheckers);
    System.out.println("Белых  шашек: " + whiteCheckers);

    this.DisplayColIndex();
    this.DrawHorizontalLine();
```

```
    for(int i = rows-1; i >= 0; i--)
    {
        this.DisplayRowIndex(i);
        this.DrawVerticalLine();

        for(int j = 0; j< cols; j++)
        {
            System.out.print(this.BoardCell(i,j));
            this.DrawVerticalLine();
        }

        this.DisplayRowIndex(i);
        System.out.println();
        this.DrawHorizontalLine();
    }

    this.DisplayColIndex();
    System.out.println();
}

/**
 * Вспомогательный метод - прорисовка отдельной клетки.
 *                         Заполнение клетки зависит от её
 *                         содержимого в массиве cell.
 *
 * @param i, j           - элемент массива cell/
 * */
private String BoardCell(int i, int j)
{
    String str = "";

    if(this.cell[i][j] == CellContents.inaccessible)
    {
        str = "     ";
    }
    else if(this.cell[i][j] == CellContents.empty)
    {
        str = "  _  ";
    }
    else if(this.cell[i][j] == CellContents.black)
    {
        str = "  ⛃  ";
    }
    else if(this.cell[i][j] == CellContents.white)
    {
        str = "  ⛂  ";
    }
    else if(this.cell[i][j] == CellContents.blackKing)
    {
        str = "  ⛁  ";
    }
    else if(this.cell[i][j] == CellContents.whiteKing)
    {
```

```
            str = "   ⛃   ";
        }

        return str;
    }

    /**
     * Вспомогательный метод - прорисовка горизонтальной линии
     *                         на поле.
     * */
    private void DrawHorizontalLine()
    {
        System.out.println(" ____________________________________
                           ________________________________________");
    }

    /**
     * Вспомогательный метод - прорисовка вертикальной линии
     *                         на поле.
     * */
    private void DrawVerticalLine()
    {
        System.out.print("|");
    }

    /**
     * Вспомогательный метод - вывод индексов колонн
     * */
    private void DisplayColIndex()
    {
        System.out.print("    ");
        for(int colIndex = 1; colIndex <= cols; colIndex++)
        {
            System.out.print("   " + colIndex + "   " );
        }
        System.out.println();
    }

    /**
     * Вспомогательный метод - вывод индексов линий
     *
     * @param index - индекс текущей строки
     * */
    private void DisplayRowIndex(int index)
    {
        if (index < 9)
        {
            System.out.print(" " + (index + 1) + " ");
        }
        else
        {
            System.out.print("" + (index + 1) + " ");
        }
```

```
    }

    /**
     * Дублирование текущей доски.
     *
     * @return newBoard - копия текущей доски.
     * */
    public Board duplicate()
    {
        Board newBoard = new Board(this.cell);
        newBoard.blackCheckers = this.blackCheckers;
        newBoard.whiteCheckers = this.whiteCheckers;

        return newBoard;
    }

    /**
     * Проверка: кончалась ли игра или нет?
     *
     * @return true  - игра закончилась
     *         False - игра продолжается
     * */
    public boolean CheckGameComplete()
    {
        return this.blackCheckers == 0 || this.whiteCheckers == 0;
    }
}
```

MoveEvaluator.java

```
/**
 * Класс, в котором реализованы все методы
 * для оценки ходов.
 * */
public class MoveEvaluator {
    /**
     * Введем некоторую стоимость для шашек.
     */
    public final int POINT_KING   = 4000;
    public final int POINT_NORMAL = 1000;

    /**
     * Оценка поля
     */
    public int fieldAssessment(Board board) {
        int value = 0;

        // проход по доске.
        for (int r = 0; r < Board.rows; r++) {
            // проверка только доступных для хода столбцов
            for (int c = (r % 2 == 0) ? 0 : 1; c < Board.cols;
                 c += 2)
            {
```

```
                CellContents entry = board.cell[r][c];

                //Поскольку COMPUTER играет за черных
                //и пытается максимизировать alpha в алгоритме
                //альфа-бета отсечения, то ведется данная
                //расстановка знаков
                if (entry == CellContents.white) {
                    value -= POINT_NORMAL;
                } else if (entry == CellContents.whiteKing) {
                    value -= POINT_KING;
                } else if
                (entry == CellContents.black) {
                    value += POINT_NORMAL;
                } else if (entry == CellContents.blackKing) {
                    value += POINT_KING;
                }
            }
        }
        return value;
    }
}
```

Move.java

```
import java.util.Vector;

/**
 * Класс 'ходов'.
 * */
public class Move
{
    int initialRow;    //начальная строка хода
    int initialCol;    //начальная колона хода
    int finalRow;      //конечная  строка хода
    int finalCol;      //конечная  колона хода

    /**
     * Конструктор с параметрами.
     * Создает объект - ход с переданными параметрами.
     *
     * @param r1, c1 - координаты начала хода
     *
     * @param r2, c2 - координаты конца  хода
     * */
    Move(int r1, int c1, int r2, int c2)
    {
        this.initialRow = r1;
        this.initialCol = c1;
        this.finalRow = r2;
        this.finalCol = c2;
    }
```

```
    /**
     * Проверка на соответствие данного хода переданному ходу.
     *
     *
     * */
    public boolean conformity(Move move)
    {
        return this.initialRow == move.initialRow
                && this.initialCol == move.initialCol
                && this.finalRow == move.finalRow
                && this.finalCol == move.finalCol;
    }

    /**
     * Проверка: есть ли данный ход в
     * списке возможных ходов.
     *
     * @param moves - список возможных ходов.
     * */
    public boolean existsInVector(Vector<Move> moves)
    {
        for (int i = 0; i<moves.size(); i++)
        {
            if (this.conformity(moves.elementAt(i)))
            {
                return true;
            }
        }
        return false;
    }

    /**
     * Вывод информации о ходе на экран.
     * */
    public void display(){
        System.out.print("(" + (this.initialRow + 1) + ","  +
                          +(this.initialCol + 1) + ") -->"   +
                          + " ("+ (this.finalRow + 1) + ", " +
                          +(this.finalCol + 1) + ")");
    }
}
```

Black.java

```
import java.util.Vector;

/**
 * Класс, реализующий логику ходов
 * черных шашек (игрок - Computer)
 * */
public class Black
{
    static Owner owner;    //поле перечисляемого типа 'игроки'
                           //'хозяин' команды
```

```
/**
 * Метод, отвечающий за ход черных
 * (Computer)
 * */
public static void Move()
{
    UserInteractions.PrintSeparator('-');
    System.out.println("\t\t\t\t\t\t\t\t\t" + "\u001B[31m" +
                       + "Ход компьютера" + "\u001B[0m");
    UserInteractions.PrintSeparator('-');

    //Компьютер делает ход
    Computer.makeNextBlackMoves();
}

/**
 * Метод, проверяющий возможность поедания
 * за черных.
 *
 * @param r     - положение по строке
 * @param c     - положение по столбцу
 * @param board - текущее положение доски
 *
 * @return возвращает список возможных ходов
 *         (с поеданиями) для черных шашек
 * */
public static Vector<Move> ObtainForcedMovesForBlack(int r,
                                                     int c, Board board)
{
    //список последующих 'поеданий'
    Vector<Move> furtherCaptures = new Vector<>();

    //Если в текущей клетке черная шашка (любая)
    //проверка на возможность поедания обычной шашкой.
    //Проверка проходит функцией ForwardLeftCaptureForBlack
    //и ForwardRightCaptureForBlack
    //которая возвращает объект класса Move
    //если нет возможности съесть, то объект Move = null
    if (board.cell[r][c].equals(CellContents.black) ||
        board.cell[r][c].equals(CellContents.blackKing))
    {
        if (ForwardLeftCaptureForBlack(r,c,board)!=null)
        {
            furtherCaptures.add
                        (ForwardLeftCaptureForBlack(r,c,board));
        }
        if (ForwardRightCaptureForBlack(r,c,board)!=null)
        {
            furtherCaptures.add
                        (ForwardRightCaptureForBlack(r,c,board));
        }
    }
```

```
    //для дамок так же проверяем возможность на поедание
    //в 'нестандартные' для обычных шашек направления
    if(board.cell[r][c].equals(CellContents.blackKing))
    {
        if (BackwardLeftCaptureForBlack(r,c,board)!=null)
        {
            furtherCaptures.add
                        (BackwardLeftCaptureForBlack(r,c,board));
        }
        if (BackwardRightCaptureForBlack(r,c,board)!=null)
        {
            furtherCaptures.add
                      (BackwardRightCaptureForBlack(r,c,board));
        }
    }
    return furtherCaptures;
}

/**
 * Метод, аккумулирующий все возможные
 * поедания для черных шашек.
 *
 * @param board - текущее положение поля
 *
 * @return список ходов
 * */
public static Vector<Move> CalculateAllForcedMovesForBlack
                                                    (Board board)
{
    //список будующих поеданий
    Vector<Move> forcedMovesForBlack = new Vector<>();

    //проход по доске, игнорируя недоступные клетки
    for(int r = 0; r<Board.rows; r++)
    {
        for(int c = (r%2==0)?0:1; c<Board.cols; c+=2)
        {
            //возможные поедания для обычных шашек
            if(board.cell[r][c].equals(CellContents.black) ||
               board.cell[r][c].equals(CellContents.blackKing))
            {
                if (r>=2)
                {
                    //поедание по левой диагонали для черных
                    if (ForwardLeftCaptureForBlack(r,c,
                        board)!=null)
                    {
                        forcedMovesForBlack.add
                       (ForwardLeftCaptureForBlack(r,c, board));
                    }
```

```
                    //поедание по правой диагонали для черных
                    if (ForwardRightCaptureForBlack(r,c,
                        board)!=null)
                    {
                      forcedMovesForBlack.add
                      (ForwardRightCaptureForBlack(r,c, board));
                    }
                }
            }
            //возможные поедания одля дамок
            if(board.cell[r][c].equals(CellContents.blackKing))
            {
                if (r<Board.rows-2)
                {
                    //поедание по левым диагоналям
                    if (BackwardLeftCaptureForBlack(r,c,
                        board)!=null)
                    {
                        forcedMovesForBlack.add
                      (BackwardLeftCaptureForBlack(r,c, board));
                    }

                    //поедание по правым диагоналям
                    if (BackwardRightCaptureForBlack(r,c,
                        board)!=null)
                    {
                      forcedMovesForBlack.add
                      (BackwardRightCaptureForBlack(r,c,board));
                    }
                }
            }
        }
    }

    return forcedMovesForBlack;
}

/**
 * Метод, отвечающий за ход, если нет вынужденных
 * ходов (поеданий)
 *
 * @param board - текущее состояние поля
 *
 * @return возвращает список перемещений
 * */
public static Vector<Move> CalculateAllNonForcedMovesForBlack
                                                    (Board board)
{
    Vector<Move> allNonForcedMovesForBlack = new Vector<>();

    //проход по доске, игнорируя недоступные клетки
    for(int r = 0; r<Board.rows; r++)
    {
        for(int c = (r%2==0)?0:1; c<Board.cols; c+=2)
```

```
        {
            // перемещение вперед для обычной черной шашки
            if( board.cell[r][c].equals(CellContents.black))
            {
                Move move;

                move = ForwardLeftForBlack(r, c, board);
                if(move!=null)
                {
                    allNonForcedMovesForBlack.add(move);
                }

                move = ForwardRightForBlack(r, c, board);
                if(move!=null)
                {
                    allNonForcedMovesForBlack.add(move);
                }
            }

            //перемещение вперед и назад для черной дамки
            if(board.cell[r][c] == CellContents.blackKing)
            {
                Move move;

                move = ForwardLeftForBlack(r, c, board);
                if(move!=null)
                {
                    allNonForcedMovesForBlack.add(move);
                }

                move = ForwardRightForBlack(r, c, board);
                if(move!=null)
                {
                    allNonForcedMovesForBlack.add(move);
                }

                move = BackwardLeftForBlack(r, c, board);
                if(move!=null)
                {
                    allNonForcedMovesForBlack.add(move);
                }

                move = BackwardRightForBlack(r, c, board);
                if(move!=null)
                {
                    allNonForcedMovesForBlack.add(move);
                }
            }
        }
    }
    return allNonForcedMovesForBlack;
}
```

```
/**
 * Метод, выполняющий проверку на возможность хода
 * влево вперед для черных
 *
 * @param r     - положение по строке
 * @param c     - положение по столбцу
 * @param board - текущее положение доски
 *
 * @return возвращает объект типа Move(ход)
 * */
private static Move ForwardLeftForBlack(int r, int c,
                                        Board board)
{
    Move forwardLeft = null;

    if(r>=1 && c<Board.cols-1 && board.cell[r-1][c+1] ==
                                              CellContents.empty)
    {
        forwardLeft = new Move(r,c,r-1, c+1);
    }
    return forwardLeft;
}

/**
 * Метод, делающий проверку на возможность поедания
 * влево вперед для черных
 *
 * @param r     - положение по строке
 * @param c     - положение по столбцу
 * @param board - текущее положение доски
 *
 * @return возвращает объект типа Move(ход)
 * */
private static Move ForwardLeftCaptureForBlack(int r, int c,
                                                 Board board)
{
    Move forwardLeftCapture = null;

    if(r>=2 && c<Board.cols-2 && (board.cell[r-1][c+1].equals
                                  (CellContents.white) ||
                                   board.cell[r-1][c+1].equals
                                  (CellContents.whiteKing)) &&
                                   board.cell[r-2][c+2].equals
                                  (CellContents.empty))
    {
         forwardLeftCapture = new Move(r,c,r-2,c+2);
    }

    return forwardLeftCapture;
}
```

```
/**
 * Метод, делающий проверку на возможность хода
 * вправо вперед для черных
 *
 * @param r     - положение по строке
 * @param c     - положение по столбцу
 * @param board - текущее положение доски
 *
 * @return возвращает объект типа Move(ход)
 * */
private static Move ForwardRightForBlack(int r, int c,
                                         Board board)
{
    Move forwardRight = null;
    if(r>=1 && c>=1 && board.cell[r-1][c-1] ==
                           CellContents.empty)
    {
        forwardRight = new Move(r,c, r-1, c-1);
    }
    return forwardRight;
}

/**
 * Метод, делающий проверку на возможность поедания
 * вправо вперед для черных
 *
 * @param r     - положение по строке
 * @param c     - положение по столбцу
 * @param board - текущее положение доски
 *
 * @return возвращает объект типа Move(ход)
 * */
private static Move ForwardRightCaptureForBlack(int r, int c,
                                                Board board)
{

    Move forwardRightCapture = null;

    if(r>=2 && c>=2 && (board.cell[r-1][c-1].equals
                        (CellContents.white) ||
                         board.cell[r-1][c-1].equals
                        (CellContents.whiteKing)) &&
                         board.cell[r-2][c-2].equals
                        (CellContents.empty))
    {
        forwardRightCapture = new Move(r,c,r-2,c-2);
    }

    return forwardRightCapture;
}
```

```
/**
 * Метод, делающий проверку на возможность хода
 * назад влево для черной дамки
 *
 * @param r     - положение по строке
 * @param c     - положение по столбцу
 * @param board - текущее положение доски
 *
 * @return возвращает объект типа Move(ход)
 * */
private static Move BackwardLeftForBlack(int r, int c,
                                         Board board)
{
    Move backwardLeft = null;

    assert(board.cell[r][c].equals(CellContents.blackKing));
    if(r<Board.rows-1 && c<Board.cols-1 &&
       board.cell[r+1][c+1] == CellContents.empty)
    {
        backwardLeft = new Move(r,c, r+1, c+1);
    }

    return backwardLeft;
}

/**
 * Метод, делающий проверку на возможность поедания
 * назад влево для черной дамки
 *
 * @param r     - положение по строке
 * @param c     - положение по столбцу
 * @param board - текущее положение доски
 *
 * @return возвращает объект типа Move(ход)
 * */
private static Move BackwardLeftCaptureForBlack(int r, int c,
                                                Board board)
{

    Move backwardLeftCapture = null;

    if(r<Board.rows-2 && c<Board.cols-2 && (
            board.cell[r+1][c+1].equals
            (CellContents.white) || board.cell[r+1][c+1].equals
            (CellContents.whiteKing))&&
             board.cell[r+2][c+2].equals(CellContents.empty)
            )
    {
         backwardLeftCapture = new Move(r,c,r+2,c+2);
    }

    return backwardLeftCapture;
}
```

```
    /**
     * Метод, делающий проверку на возможность хода
     * назад вправо для черной дамки
     *
     * @param r     - положение по строке
     * @param c     - положение по столбцу
     * @param board - текущее положение доски
     *
     * @return возвращает объект типа Move(ход)
     * */
    private static Move BackwardRightForBlack(int r, int c,
                                              Board board)
    {
        Move backwardRight = null;

        if(r<Board.rows-1 && c>=1 && board.cell[r+1][c-1].equals
           (CellContents.empty))
        {
            backwardRight = new Move(r,c, r+1, c-1);
        }
        return backwardRight;
    }


    /**
     * Метод, делающий проверку на возможность поедания
     * назад вправо для черной дамки
     *
     * @param r     - положение по строке
     * @param c     - положение по столбцу
     * @param board - текущее положение доски
     *
     * @return возвращает объект типа Move(ход)
     * */
    private static Move BackwardRightCaptureForBlack(int r, int c,
                                                     Board board)
    {

        Move backwardRightCapture = null;

        if(r<Board.rows-2 && c>=2 && (board.cell[r+1][c-1].equals
           (CellContents.white) || board.cell[r+1][c-1].equals
           (CellContents.whiteKing)) && board.cell[r+2][c-2].equals
           (CellContents.empty))
        {
            backwardRightCapture = new Move(r,c,r+2,c-2);
        }

        return backwardRightCapture;
    }
}
```

White.java

```
import java.util.Vector;

/**
 * Класс, реализующий логику ходов
 * белыз шашек (игрок - Человек)
 * */
public class White
{
    static Owner owner;    //поле перечисляемого типа 'игроки'
                           //'хозяин' команды
    /**
     * Метод, отвечающий за ход белых
     * (Computer)
     * */
    public static void Move()
    {
        UserInteractions.PrintSeparator('-');
        System.out.println("\t\t\t\t\t\t\t\t\t" + "\u001B[34m" +
                           + "Ваш ход" + "\u001B[0m");
        UserInteractions.PrintSeparator('-');

        //Человек делает ход
        Human.makeNextWhiteMoves();
    }

    /**
     * Метод, проверяющий возможность поедания
     * за белых.
     *
     * @param r     - положение по строке
     * @param c     - положение по столбцу
     * @param board - текущее положение доски
     *
     * @return возвращает список возможных ходов
     *         (с поеданиями) для белых шашек
     * */
    public static Vector<Move> ObtainForcedMovesForWhite(int r,
                                                         int c, Board board)
    {
        //список последующих 'поеданий'
        Vector<Move> furtherCaptures = new Vector<>();

        //Если в текущей клетке белая шашка (любая)
        //проверка на возможность поедания обычной шашкой.
        //Проверка проходит функцией ForwardLeftCaptureForWhite
        //и ForwardRightCaptureForWhite
        //которая возвращает объект класса Move
        //если нет возможности съесть, то объект Move = null
        if (board.cell[r][c].equals(CellContents.white) ||
            board.cell[r][c].equals(CellContents.whiteKing))
        {
```

```
        if (ForwardLeftCaptureForWhite(r,c,board)!=null)
        {
            furtherCaptures.add
            (ForwardLeftCaptureForWhite(r,c,board));
        }
        if (ForwardRightCaptureForWhite(r,c,board)!=null)
        {
            furtherCaptures.add
            (ForwardRightCaptureForWhite(r,c,board));
        }
    }

    //для дамок так же проверяем возможность на поедание
    //в 'нестандартные' для обычных шашек направления
    if(board.cell[r][c].equals(CellContents.whiteKing))
    {
        if (BackwardLeftCaptureForWhite(r,c,board)!=null)
        {
            furtherCaptures.add
           (BackwardLeftCaptureForWhite(r,c,board));
        }
        if (BackwardRightCaptureForWhite(r,c,board)!=null)
        {
            furtherCaptures.add
           (BackwardRightCaptureForWhite(r,c,board));
        }
    }
    return furtherCaptures;
}

/**
 * Метод, аккумулирующий все возможные
 * поедания для белых шашек.
 *
 * @param board - текущее положение поля
 *
 * @return список ходов
 * */
public static Vector<Move> CalculateAllForcedMovesForWhite
                                                    (Board board)
{
    //список будующих поеданий
    Vector<Move> forcedMovesForWhite = new Vector<>();

    //проход по доске, игнорируя недоступные клетки
    for(int r = 0; r<Board.rows; r++)
    {
        for(int c = (r%2==0)?0:1; c<Board.cols; c+=2)
        {
            //возможные поедания для обычных шашек
            if(board.cell[r][c].equals(CellContents.white) ||
               board.cell[r][c].equals(CellContents.whiteKing))
            {
                if (r<Board.rows-2)
```

```
                {
                    //поедание по левой диагонали для белых
                    if (ForwardLeftCaptureForWhite(r,c,
                        board)!=null)
                    {
                        forcedMovesForWhite.add
                       (ForwardLeftCaptureForWhite(r,c, board));
                    }

                    //поедание по правой диагонали для белых
                    if (ForwardRightCaptureForWhite(r,c,
                        board)!=null)
                    {
                       forcedMovesForWhite.add
                      (ForwardRightCaptureForWhite(r,c, board));
                    }
                }
            }
            //возможные поедания одля дамок
            if(board.cell[r][c].equals(CellContents.whiteKing))
            {
                if (r>=2)
                {
                    //поедание по левым диагоналям
                    if (BackwardLeftCaptureForWhite(r,c,board)
                                                    !=null)
                    {
                       forcedMovesForWhite.add
                      (BackwardLeftCaptureForWhite(r,c, board));
                    }

                    //поедаиние по правым диагоналям
                    if (BackwardRightCaptureForWhite(r,c,board)
                        !=null)
                    {
                      forcedMovesForWhite.add
                      (BackwardRightCaptureForWhite(r,c,board));
                    }
                }
            }
        }
    }
    return forcedMovesForWhite;
}

/**
 * Метод, отвечающий за ход, если нет вынужденных
 * ходов (поеданий)
 *
 * @param board - текущее состояние поля
 *
 * @return возвращает список перемещений
 * */
```

```
    public static Vector<Move>
CalculateAllNonForcedMovesForWhite(Board board)
{
        Vector<Move> allNonForcedMovesForWhite = new Vector<>();

        //проход по доске, игнорируя недоступные клетки
        for(int r = 0; r<Board.rows; r++)
        {
            for(int c = (r%2==0)?0:1; c<Board.cols; c+=2)
            {
                // перемещение вперед для обычной белой шашки
                if( board.cell[r][c].equals(CellContents.white))
                {
                    Move move;

                    move = ForwardLeftForWhite(r, c, board);
                    if(move!=null)
                    {
                        allNonForcedMovesForWhite.add(move);
                    }

                    move = ForwardRightForWhite(r, c, board);
                    if(move!=null)
                    {
                        allNonForcedMovesForWhite.add(move);
                    }
                }

                //перемещение вперед и назад для черной дамки
                if(board.cell[r][c] == CellContents.whiteKing){
                    Move move ;

                    move = ForwardLeftForWhite(r, c, board);
                    if(move!=null)
                    {
                        allNonForcedMovesForWhite.add(move);
                    }

                    move = ForwardRightForWhite(r, c, board);
                    if(move!=null)
                    {
                        allNonForcedMovesForWhite.add(move);
                    }

                    move = BackwardLeftForWhite(r, c, board);
                    if(move!=null)
                    {
                        allNonForcedMovesForWhite.add(move);
                    }

                    move = BackwardRightForWhite(r, c, board);
                    if(move!=null)
                    {
                        allNonForcedMovesForWhite.add(move);
```

```
                }
            }
        }
    }

    return allNonForcedMovesForWhite;
}

/**
 * Метод, производящий проверку на возможность хода
 * влево вперед для белых
 *
 * @param r     - положение по строке
 * @param c     - положение по столбцу
 * @param board - текущее положение доски
 *
 * @return возвращает объект типа Move(ход)
 * */
private static Move ForwardLeftForWhite(int r, int c,
                                                    Board board)
{
    Move forwardLeft = null;
    if(r<Board.rows-1 && c>=1 && board.cell[r+1][c-1] ==
       CellContents.empty)
    {
        forwardLeft = new Move(r,c, r+1, c-1);
    }
    return forwardLeft;
}

/**
 * Метод, делающий проверку на возможность поедания
 * влево вперед для белых
 *
 * @param r     - положение по строке
 * @param c     - положение по столбцу
 * @param board - текущее положение доски
 *
 * @return возвращает объект типа Move(ход)
 * */
private static Move ForwardLeftCaptureForWhite(int r, int c,
                                                    Board board)
{
    Move forwardLeftCapture = null;

    if(r<Board.rows-2 && c>=2 && (board.cell[r+1][c-1].equals
       (CellContents.black)||board.cell[r+1][c-1].equals
       (CellContents.blackKing)) && board.cell[r+2][c-2].equals
       (CellContents.empty))
    {
         forwardLeftCapture = new Move(r,c,r+2,c-2);
    }

    return forwardLeftCapture;
```

```
}

/**
 * Метод, делающий проверку на возможность хода
 * вправо вперед для белых
 *
 * @param r     - положение по строке
 * @param c     - положение по столбцу
 * @param board - текущее положение доски
 *
 * @return возвращает объект типа Move(ход)
 * */
private static Move ForwardRightForWhite(int r, int c,
                                                    Board board)
{
    Move forwardRight = null;
    if(r<Board.rows-1 && c<Board.cols-1 &&
       board.cell[r+1][c+1] == CellContents.empty)
    {
        forwardRight = new Move(r,c, r+1, c+1);
    }
    return forwardRight;
}

/**
 * Метод, делающий проверку на возможность поедания
 * вправо вперед для белых
 *
 * @param r     - положение по строке
 * @param c     - положение по столбцу
 * @param board - текущее положение доски
 *
 * @return возвращает объект типа Move(ход)
 * */
private static Move ForwardRightCaptureForWhite(int r, int c,
                                                    Board board)
{
    Move forwardRightCapture = null;

    if(r<Board.rows-2 && c<Board.cols-2 &&
       (board.cell[r+1][c+1].equals(CellContents.black) ||
        board.cell[r+1][c+1].equals(CellContents.blackKing)) &&
        board.cell[r+2][c+2].equals(CellContents.empty))
    {
        forwardRightCapture = new Move(r,c,r+2,c+2);
    }

    return forwardRightCapture;
}

private static Move BackwardLeftForWhite(int r, int c,
                                                    Board board)
{
    Move backwardLeft = null;
```

```
    if(r>=1 && c>=1&& oard.cell[r-1][c-1] == CellContents.empty)
    {
        backwardLeft = new Move(r,c, r-1, c-1);
    }
    return backwardLeft;
}

/**
 * Метод, делающий проверку на возможность хода
 * назад влево для белой дамки
 *
 * @param r     - положение по строке
 * @param c     - положение по столбцу
 * @param board - текущее положение доски
 *
 * @return возвращает объект типа Move(ход)
 * */
private static Move BackwardLeftCaptureForWhite(int r, int c,
                                                Board board)
{
    Move backwardLeftCapture = null;

    if(r>=2 && c>=2 && (board.cell[r-1][c-1].equals
       (CellContents.black) || board.cell[r-1][c-1].equals
        (CellContents.blackKing)) && board.cell[r-2][c-2].equals
        (CellContents.empty))
    {
         backwardLeftCapture = new Move(r,c,r-2,c-2);
    }

    return backwardLeftCapture;
}

/**
 * Метод, делающий проверку на возможность хода
 * назад вправо для белой дамки
 *
 * @param r     - положение по строке
 * @param c     - положение по столбцу
 * @param board - текущее положение доски
 *
 * @return возвращает объект типа Move(ход)
 * */
private static Move BackwardRightForWhite(int r, int c,
                                                Board board)
{
    Move backwardRight = null;
    if(r>=1 && c<Board.cols-1 && board.cell[r-1][c+1] ==
                                            CellContents.empty)
    {
        backwardRight = new Move(r,c,r-1,c+1);
    }
    return backwardRight;
}
```

```
    /**
     * Метод, делающий проверку на возможность поедания
     * назад влево для белой дамки
     *
     * @param r     - положение по строке
     * @param c     - положение по столбцу
     * @param board - текущее положение доски
     *
     * @return возвращает объект типа Move(ход)
     * */
    private static Move BackwardRightCaptureForWhite(int r, int c,
                                                     Board board)
    {
        Move backwardRightCapture = null;

        if(r>=2 && c<Board.cols-2 && (board.cell[r-1][c+1].equals
           (CellContents.black)||board.cell[r-1][c+1].equals
           (CellContents.blackKing))&&board.cell[r-2][c+2].equals
           (CellContents.empty))
        {
            backwardRightCapture = new Move(r,c,r-2,c+2);
        }

        return backwardRightCapture;
    }
}
```

UserInteractions.java

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Vector;

/**
 * Класс, реализующий взаимодействие пользователя
 * с программой.
 * */
public class UserInteractions
{
    /**
     * Получение координат хода от игрока.
     * */
    public static Move getNextMove()
    {
        return TakeUserInput(-1,-1);
    }

    //Передаем r1 как -1 и c1 как -1 если мы хотим принять ход от
    //игрока.
    public static Move takeUserInput(int r1, int c1)
    {
        //прорисовка доски
        Game.board.Display();
        PrintSeparator('-');
```

```
    // Просьба о вводе
    System.out.println("Введите свой ход.");
    System.out.println("Начало:");

    System.out.print("\tСтр(1-12): ");
    if (r1==-1)
    {
        r1 = TakeInput();
    }
    else
    {
        System.out.println(r1);
    }

    System.out.print("\tКол(1-12): ");
    if (c1==-1)
    {
        c1 = TakeInput();
    }
    else
    {
        System.out.println(c1);
    }


    System.out.println("Конец:");
    System.out.print("\tСтр(1-12): ");
    int r2 = TakeInput();

    System.out.print("\tКол(1-12): ");
    int c2 = TakeInput();

    return new Move(r1,c1,r2,c2);
}

/**
 *  Ввод координат с проверкой.
 *  Если необходимо, то предлагается повторный ввод.
 * */
private static int takeInput(){

    int num;
    BufferedReader br = new BufferedReader(new
                            InputStreamReader(System.in));

    while (true)
    {
        try
        {
            num = Integer.parseInt(br.readLine());
            num -= 1;

            if (num>=0 && num < Board.rows){
```

```
                    break;
                }
            } catch (Exception ignored) {}

            System.out.print("Неправильный ввод... Повторите +
                             + попытку: ");
        }
        return num;
    }

    /**
     * Вывод 'декораций'.
     *
     * @param ch - символ для вывода.
     * */
    public static void printSeparator(char ch)
    {
        switch (ch)
        {
            case '_' -> System.out.println("_____________________ +
                                           + _____________________");
            case '-' -> System.out.println("---------------------- +
                                           + ----------------------");
            case '#' -> System.out.println("###################### +
                                           + ######################");
        }
    }

    /**
     * Вывод информации о ходе.
     *
     * @param moveSeq - список с координатами хода.
     * */
    public static void displayMoveSeq(Vector<Move> moveSeq){
        for(Move m:moveSeq)
        {
            m.display();
            System.out.print(", ");
        }

        System.out.println();
    }
}
```

Computer.java

```
import java.util.Vector;

/**
 * Класс реализующий логику игрока
 * COMPUTER.
 * */
public class Computer
{
    //поле для хранения оценок поля
```

```
static MoveEvaluator oracle = new MoveEvaluator();
static int MAX_DEPTH = 6;    //максимальная глубина дерева
                             //альфа-бета отсечений

public static void makeNextBlackMoves()
{
    //список итоговых ходов
    Vector<Move> resultantMoveSeq = new Vector<>();

    //вызов алгоритма альфа-бета отсечений
    alphaBeta(Game.board, Command.black, 0, Integer.MIN_VALUE,
              Integer.MAX_VALUE, resultantMoveSeq);

    //применение итоговых ходов к полю
    for(Move m:resultantMoveSeq)
    {
        Game.board.genericMakeBlackMove(m);
    }

    System.out.println();
    System.out.print("Ход компьютера: ");
    UserInteractions.DisplayMoveSeq(resultantMoveSeq);
    System.out.println();
}

/**
 * Алгоритм alphaBeta отсечений
 *
 * @param board         - текущее состояние доски
 * @param command       - 'команда' для которой обработается ход
 * @param depth         - текущий уровень глубины
 * @param alpha         - значение альфа
 * @param beta          - значение бета
 * @param resultMoveSeq - список результирующих ходов
 *
 * @return возвращает значение - оценка текущего,
 *         рассматриваемого положения
 * */
private static int alphaBeta(Board board, Command command,
     int depth, int alpha, int beta, Vector<Move> resultMoveSeq)
{
    //Если мы НЕ можем дальше строить дерево, то
    //оцениваем текущее положение
    if(!canExploreFurther(board, depth))
    {
        return oracle.fieldAssessment(board);
    }

    //создаем список списков возможных ходов и заполняем его
    Vector<Vector<Move>> possibleMoveSeq = expandMoves(board);

    //создаем список возможных состояний доски и заполняем его
    Vector<Board> possibleBoardConf=getPossibleBoardConf(board,
                                    possibleMoveSeq, command);
```

```
//список лучших ходов
Vector<Move> bestMoveSeq = null;

if(command == Command.black)
{
    for(int i=0; i<possibleBoardConf.size(); i++)
    {
        //получаем состояние доски из списка состояний
        Board b              = possibleBoardConf.get(i);

        //получаем список ходов из списка списков возможных
        //ходов
        Vector<Move> moveSeq = possibleMoveSeq.get(i);

        //получаем значение альфа-бета алгоритма
        int value = alphaBeta(b, Command.white, depth+1,
                              alpha, beta, resultMoveSeq);

        //если полученное значение больше альфа,
        //то перезаписываем альфа
        //делаем данный список ходов лучшим
        if(value > alpha)
        {
            alpha = value;
            bestMoveSeq = moveSeq;
        }

        //если возникает конфликт интересов,
        //прерываем алгоритм
        if(alpha>beta)
        {
            break;
        }
    }

    //достигнута конечная глубина, то заносим
    //ходы из списка лучших ходов в список результирующих
    //ходов
    if(depth == 0 && bestMoveSeq!=null)
    {
        resultMoveSeq.addAll(bestMoveSeq);
    }
    return alpha;
}
else if(command == Command.white)
    {
    for(int i=0; i<possibleBoardConf.size(); i++){

        //получаем состояние доски из списка состояний
        Board b = possibleBoardConf.get(i);

        //получаем список ходов из списка списков возможных
        //ходов
```

```
            Vector<Move> moveSeq = possibleMoveSeq.get(i);

            //получаем значение альфа-бета алгоритма
            int value = alphaBeta(b, Command.white, depth+1,
                                  alpha, beta, resultMoveSeq);

            //если текущее значение меньше бета
            //то перезапись бета,
            //делаем данный список ходов лучшим
            if(value < beta)
            {
                bestMoveSeq = moveSeq;
                beta = value;
            }

            //если возникает конфликт интересов,
            //прерываем алгоритм
            if(alpha>beta)
            {
                break;
            }
        }

        //достигнута конечная глубина, то заносим
        //ходы из списка лучших ходов в список результирующих
        //ходов
        if(depth == 0 && bestMoveSeq!=null){
            resultMoveSeq.addAll(bestMoveSeq);
        }
        return beta;
    }
}

/**
 * Метод поиска возможных ходов.
 *
 * @param board - текущее состояние доски
 *
 * @return возвращает список списков ходов.
 * */
public static Vector<Vector<Move>> expandMoves(Board board)
{
    //список списков ходов
    Vector<Vector<Move>> outerVector = new Vector<>();

    //список ходов
    Vector<Move> moves;

    //получение списка возможных ходов поедания
    moves = Black.CalculateAllForcedMovesForBlack(board);

    //если список поеданий пуст,
    //то ищем обычные ходы
    if(moves.isEmpty())
```

```
        {
            moves = Black.CalculateAllNonForcedMovesForBlack(board);

            for(Move m:moves)
            {
                Vector<Move> innerVector = new Vector<>();
                innerVector.add(m);
                outerVector.add(innerVector);
            }

        }
        //иначе ищем дальнейшие ходы
        //после поедания
        else
            {
            for(Move m:moves){

                int r = m.finalRow;
                int c = m.finalCol;
                Vector<Move> innerVector = new Vector<>();

                innerVector.add(m);

                Board boardCopy = board.duplicate();
                boardCopy.genericMakeBlackMove(m);
                expandMoveRecursivelyForBlack(boardCopy,
                outerVector, innerVector, r, c);

                innerVector.remove(m);
            }
        }

        return outerVector;
    }

    /**
     * Метод рекурсивного 'разворота' ходов
     * для черных шашек
     *
     * @param board       - текущее состояние доски
     * @param outerVector - список списков ходов - внешний список
     * @param innerVector - внутренний список, который добавляется
     *                      во внешний
     * @param c           - текущее положение по колонне
     * @param r           - текущее положение по столбцу
     * */
    private static void expandMoveRecursivelyForBlack(Board board,
                            Vector<Vector<Move>> outerVector,
                            Vector<Move> innerVector, int r, int c)
{
        //Список вынужденных ходов
        Vector<Move> forcedMoves=Black.ObtainForcedMovesForBlack(r,
                                                        c, board);
```

```
    //если список вынужденных ходов пуст, то просто
    //вносим внутренний список во внешний
    if(forcedMoves.isEmpty())
    {
        outerVector.add(innerVector);
    }
    //иначе рекурсивно по каждому ходу с поеданием
    //продолжаем 'развертывание' ходов
    else
    {
        for(Move m:forcedMoves)
        {
            Board boardCopy = board.duplicate();
            boardCopy.genericMakeBlackMove(m);

            innerVector.add(m);
            expandMoveRecursivelyForBlack(boardCopy,
            outerVector, innerVector, m.finalRow, m.finalCol);
            innerVector.remove(m);
        }
    }
}

/**
 * Вспомогательная функция проверки возможности
 * дальнейшего построения дерева MiniMax'a.
 *
 * @param board   - текущее состояние доски
 * @param depth   - текущий уровень глубины
 * */
private static boolean canExploreFurther(Board board, int depth)
{
    //Если игра кончилась или нет невозможный ходов
    //возвращаем false - дальнейшее 'построение'
    //дерева минимакса невозможно
    if(board.CheckGameComplete())
    {
        return false;
    }

    //если текущая глубина не равна максимальной,
    //то возвращает true и строим дерево минимакса дальше
    return depth != MAX_DEPTH;
}

/**
 * Вспомогательный метод,
 * 'эмулирующий' ход того или иного игрока.
 *
 * @param  board             - текущее состояние доски
 * @param  command           - 'команда' для которой
 *                             обработывается ход
 * @param  possibleMoveSeq   - список уже проведенных ходов
 *
```

```
     * @return possibleBoardConf - список возможных состояний доски
     * */
    private static Vector<Board> getPossibleBoardConf(Board board,
              Vector<Vector<Move>> possibleMoveSeq, Command command)
    {
        //список для хранения итоговых состояний доски.
        Vector<Board> possibleBoardConf= new Vector<>();

        for(Vector<Move> moveSeq: possibleMoveSeq)
        {
            //копирование текущего состояния доски
            Board boardCopy = board.duplicate();

            //перебор ходов
            for(Move move: moveSeq)
            {
                //если ход 'черных' - компьютера эмулируем ход
                //компьютера иначе эмулируем ход человека
                if(command == Command.black)
                {
                    boardCopy.genericMakeBlackMove(move);
                }
                else
                {
                    boardCopy.genericMakeWhiteMove(move);
                }
            }
            //добавление полученного состояния доски в список
            possibleBoardConf.add(boardCopy);
        }
        return possibleBoardConf;
    }
}
```

Human.java

```
import java.util.Vector;

/**
 * Класс, реализующий логику игры человека.
 * */
public class Human
{

    /**
     *  Сделать слудеющий ход для белых шашек.
     * */
    public static void makeNextWhiteMoves()
    {
        while(true)
        {
            //Получение координат хода
            Move move = UserInteractions.getNextMove();

            //Проверка правильности хода
```

```
        if (CheckValidMoveForWhiteHuman(move.initialRow,
            move.initialCol, move.finalRow, move.finalCol))
        {
            break;
        }
    }
}

/**
 * Вывод предупреждения
 * */
private static void printWarning(Vector<Move> vector)
{
    UserInteractions.PrintSeparator('-');

    System.out.println("Существует вынужденный ход!!!");
    System.out.println("Есть слудеющие варинты ходов.");
    for (int i=0; i < vector.size(); i++)
    {
        System.out.print((i+1) + ". ");
        System.out.print("(r1: "+(vector.elementAt(i).initialRow
                            + 1)       + ", ");
        System.out.print("c1: "+(vector.elementAt(i).initialCol
                           + 1)        + ")" );
        System.out.print("------> (r2: "+ (vector.elementAt(i).
                                      finalRow + 1) + ", ");
        System.out.println("c2: " + (vector.elementAt(i).
                                   finalCol + 1)        + ")");
    }

    UserInteractions.PrintSeparator('-');
}

/**
 * Проверка правильности хода для белых шашек.
 *
 * @param r1, c1 - координаты начала хода
 *
 * @param r2, c2 - координаты конца хода
 * */
private static boolean checkValidMoveForWhiteHuman(int r1,
                                          int c1, int r2, int c2)
{
    // выбор и проверка правильности хода
    if (
            Game.board.cell[r1][c1].equals
           (CellContents.inaccessible)||
            !(Game.board.cell[r1][c1].equals(CellContents.white)
            ||Game.board.cell[r1][c1].equals
            (CellContents.whiteKing))||
            !Game.board.cell[r2][c2].equals(CellContents.empty)
        )
    {
        UserInteractions.PrintSeparator('-');
```

```
    System.out.println("Невозможный ход!");
    UserInteractions.PrintSeparator('-');
    return false;
}

//Проверка на вынужденые ходы
Vector<Move> forcedMovesAtR1C1 =
          White.ObtainForcedMovesForWhite(r1,c1,Game.board);

//Если есть вынужденные ходы
if (!forcedMovesAtR1C1.isEmpty())
{
    Move move = new Move(r1,c1,r2,c2);

    // проверка: является ли ход вынужденным
    if (move.existsInVector(forcedMovesAtR1C1))
    {
        //Проверка на съедание
        while (true)
        {
            //съедание черной шашки
            Game.board.CaptureBlackPiece(r1,c1,r2,c2);

            //запись перемещения
            r1 = r2;
            c1 = c2;

            //вычисление след. позиций в которых необходимо
            //съесть
            Vector<Move> furtherCapture =
            White.ObtainForcedMovesForWhite(r1, c1,
                                            Game.board);

            //дальше некого есть
            if (furtherCapture.isEmpty()) {
                break;
            }

            //Предупреждение о том, что есть ещё
            //необходимость съесть шашки, просьба повторного
            //ввода координат
            boolean incorrectOption = true;
            while (incorrectOption)
            {
                printWarning(furtherCapture);
                UserInteractions.PrintSeparator('-');

                //запись хода
                Move furtherMove =
                      UserInteractions.TakeUserInput(r1,c1);

                //проверка на верность хода
                if (furtherMove.existsInVector
                    (furtherCapture))
```

```
                {
                    // обновление координат
                    r2 = furtherMove.finalRow;
                    c2 = furtherMove.finalCol;
                    incorrectOption = false;
                }
            }
        }

        return true;
    }
    else
    {
        printWarning(forcedMovesAtR1C1);

        UserInteractions.PrintSeparator('-');
        return false;
    }
}

// если принудительных ходов нет
// расчет всех возможных ходов для белых шашек на данной
// доске
Vector<Move> forcedMoves =
          White.CalculateAllForcedMovesForWhite(Game.board);

// если нет возможных ходов для белых шашек
if (forcedMoves.isEmpty())
{
    // ход
    if (r2 - r1 == 1 && Math.abs(c2 - c1) == 1)
    {
        Game.board.MakeMove(r1, c1, r2, c2);
        return true;
    }

    // ход для короля
    else if (Game.board.cell[r1][c1].equals
             (CellContents.whiteKing))
    {
        if (r2 - r1 == -1 && Math.abs(c2 - c1) == 1)
        {
            Game.board.MakeMove(r1, c1, r2, c2);
            return true;
        }
    }
    else{
        UserInteractions.PrintSeparator('-');
        System.out.println("Проверте правильность хода!\n");
        UserInteractions.PrintSeparator('-');
        return false;
    }
}
else
```

```
        {
            printWarning(forcedMoves);
            return false;
        }

        return false;
    }
}
```

Game.java

```
/**
 * Класс реализующий сам процесс игры.
 * */
public class Game
{

    static Board board;    //хранение текущего состояния поля

    /**
     * Конструктор по умолчанию.
     * Создание пустой доски, выбор команд игроков.
     * */
    Game()
    {
        board  = new Board();

        //Человек - белые. Ходит первым
        //Робот   - черные.
        White.owner = Owner.HUMAN;
        Black.owner = Owner.COMPUTER;
    }

    /**
     * Выводит строку с указанием победителя партии.
     * Анализирует кол-во шашек разных цветов на столе.
     * */
    private void whoWin()
    {
        UserInteractions.PrintSeparator('_');
        if (Game.board.blackCheckers == 0)
        {
            System.out.println("\t\t\t\t\t\t\t" + "\u001B[33m" +
                               + "Победил человек!" + "\u001B[0m");
        }
        else if (Game.board.whiteCheckers == 0)
        {
            System.out.println("\t\t\t\t\t\t\t" + "\u001B[33m" +
                               + "Победила машина!" + "\u001B[0m");
        }
        UserInteractions.PrintSeparator('_');
    }
```

```
/**
 * Запуск и поддержание игрового процесса.
 * */
public void playGame()
{
    //Пока игра не закончилась
    while(!Game.board.CheckGameComplete())
    {
        White.Move();      //ход человека

        if(Game.board.CheckGameComplete())
        {
            Game.board.Display();
            break;
        }

        //прорисовка поля
        Game.board.Display();
        Black.Move();      //ход COMPUTER
        if(Game.board.CheckGameComplete())
        {
            Game.board.Display();
            break;
        }
    }
    //определяем победителя
    whoWin();

}

/**
 * Вывод справки о программе при
 * запуске с флагом -h или --help
 * */
public static void printGameReference()
{
    System.out.println();
    System.out.println("\t\t\t\t\t\t\033[0;4mСправка об игре");
    System.out.println("Замечание\u001B[0m: стандартные +
                       + Английские шашки играются на поле 8x8,+
                       + данная\n модификация играется на поле +
                       + 12x12 (Размер поля и расстановка шашек+
                       +\n аналогичная Канадским шашкам).");
    System.out.println();
    System.out.println("\t\t\t\t\t\t\033[0;4mПравила +
                       + игры\u001B[0m");
    System.out.println(" Цель игры — уничтожить все шашки +
                       + противника или лишить их\n" +
                       + "возможности хода («запереть»).\n" +
                       + " Игра проводится на доске 12×12 +
                       + клеток. В начальной позиции у +
                       + каждого\n игрока по 30 шашек, +
                       + расположенных на первых пяти рядах на +
                       + черных клетках.\n Первый ход делают +
```

```
                                + белые шашки (Игрок - человек). Простая+
                                + шашка ходит\n по диагонали вперёд на +
                                + одну клетку.\n При достижении любого +
                                + поля последней горизонтали, простая +
                                + шашка\n превращается в дамку. Дамка +
                                + может ходить на одно поле по диагонали+
                                + как\n вперёд так и назад.\n Взятие +
                                + обязательно, если оно возможно. Шашки +
                                + снимаются с доски лишь\n" +
                                +"после того, как берущая шашка +
                                + остановилась. При нескольких +
                                + вариантах\n взятия игрок выбирает +
                                + вариант взятия по своему усмотрению, и+
                                + в выбранном\n варианте необходимо бить+
                                + все доступные для взятия шашки. При +
                                + взятии\n" дамка побьет только через +
                                + одно поле в любую сторону, а не на +
                                + любое поле\n диагонали, как в русских+
                                + или международных шашках.");
    }

    /**
     * Вывод предупреждение о запуске
     * с неверным(и) параметром(параметрами).
     * */
    public static void printStartError()
    {
        System.out.println();
        System.out.println("\u001B[31mНеверные аргументы +
                           + запуска!\u001B[0m");
        System.out.println("Для получения справки о программе +
                            +необходимо передать аргумент: '-h' или+
                            + '--help'");
        System.out.println("Для запуска программы необходимо +
                            +  передать аргумент:        '-start'");
    }
}
```

7. Результаты работы, принтринтскрины экранов

Неверный запуск программы

```
Неверные аргументы запуска!
Для получения справки о программе необходимо передать аргумент: '-h' или '--help'
Для запуска программы необходимо передать аргумент:             '-start'
```

Скриншот 5

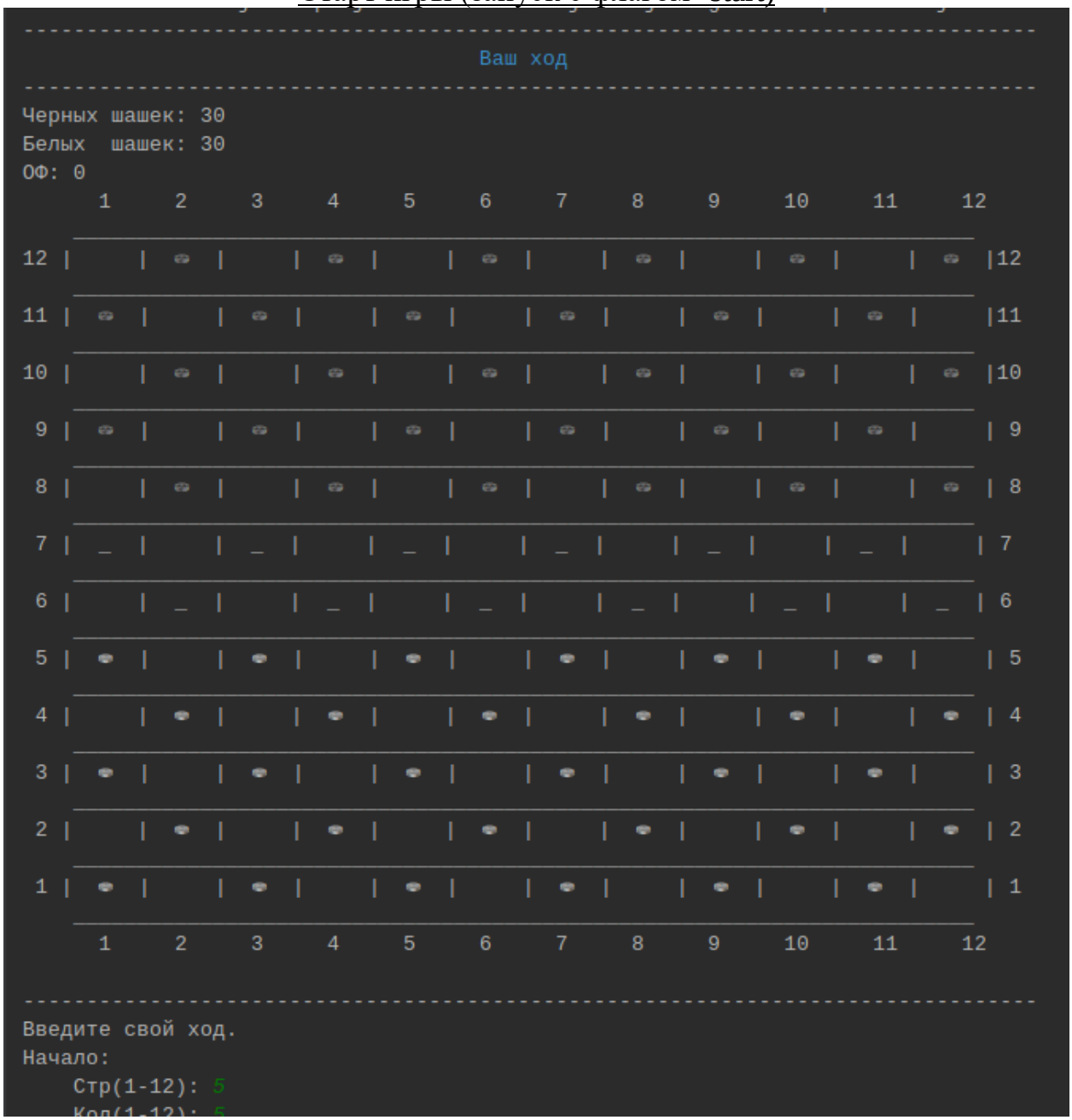
Запуск с флагом -h или – help

```
                    Справка об игре
Замечание: стандартные Английские шашки играются на поле 8x8, данная
модификация играется на поле 12x12 (Размер поля и расстановка шашек
аналогичная Канадским шашкам).

                    Правила игры
 Цель игры — уничтожить все шашки противника или лишить их
возможности хода («запереть»).
 Игра проводится на доске 12×12 клеток. В начальной позиции у каждого
игрока по 30 шашек, расположенных на первых пяти рядах на черных
клетках.
 Первый ход делают белые шашки (Игрок - человек). Простая шашка ходит
по диагонали вперёд на одну клетку.
 При достижении любого поля последней горизонтали, простая шашка
превращается в дамку. Дамка может ходить на одно поле по диагонали как
вперёд так и назад.
 Взятие обязательно, если оно возможно. Шашки снимаются с доски лишь
после того, как берущая шашка остановилась. При нескольких вариантах
взятия игрок выбирает вариант взятия по своему усмотрению, и в выбранном
варианте необходимо бить все доступные для взятия шашки. При взятии
дамка побьет только через одно поле в любую сторону, а не на любое поле
диагонали, как в русских или международных шашках.
```

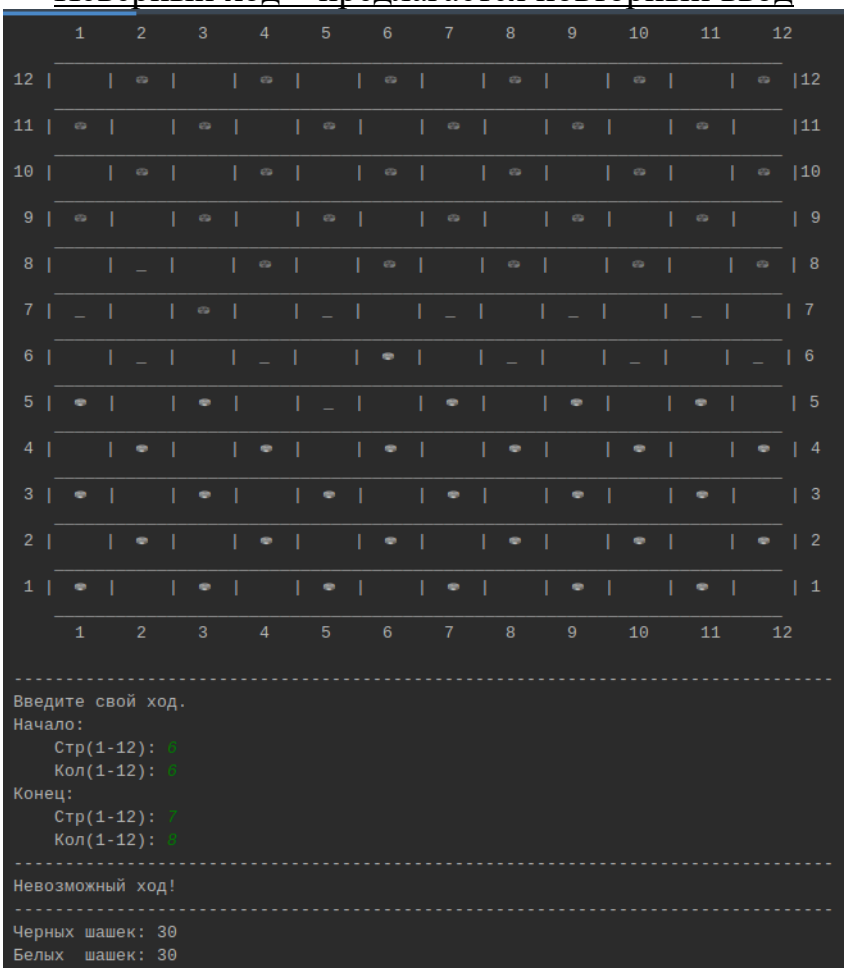
Скриншот 6

Старт игры (запуск с флагом -start)

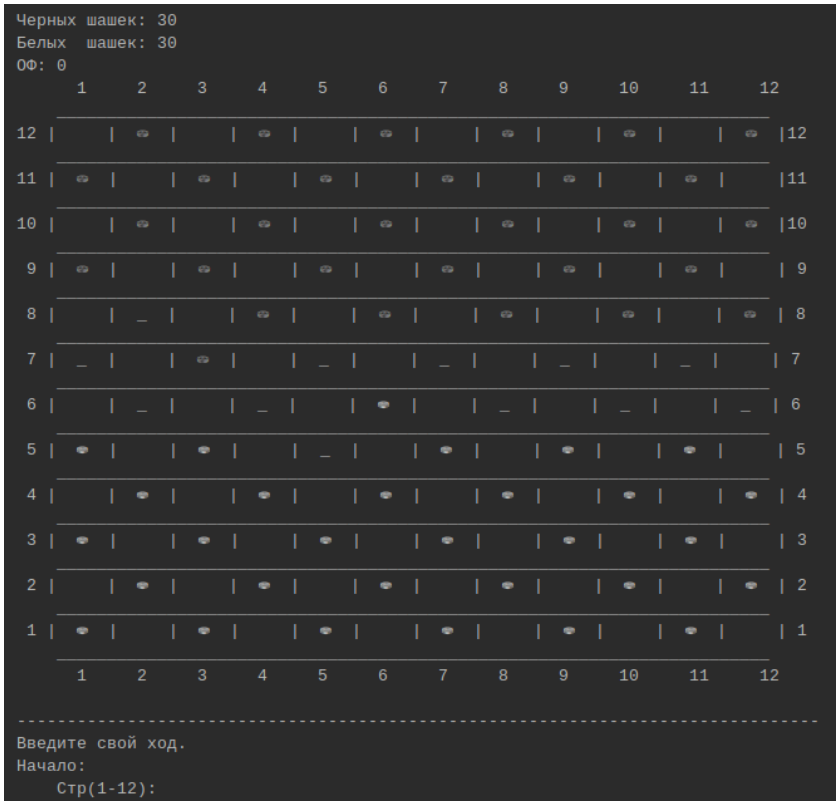


Скриншот 7

Неверный ход – предлагается повторный ввод

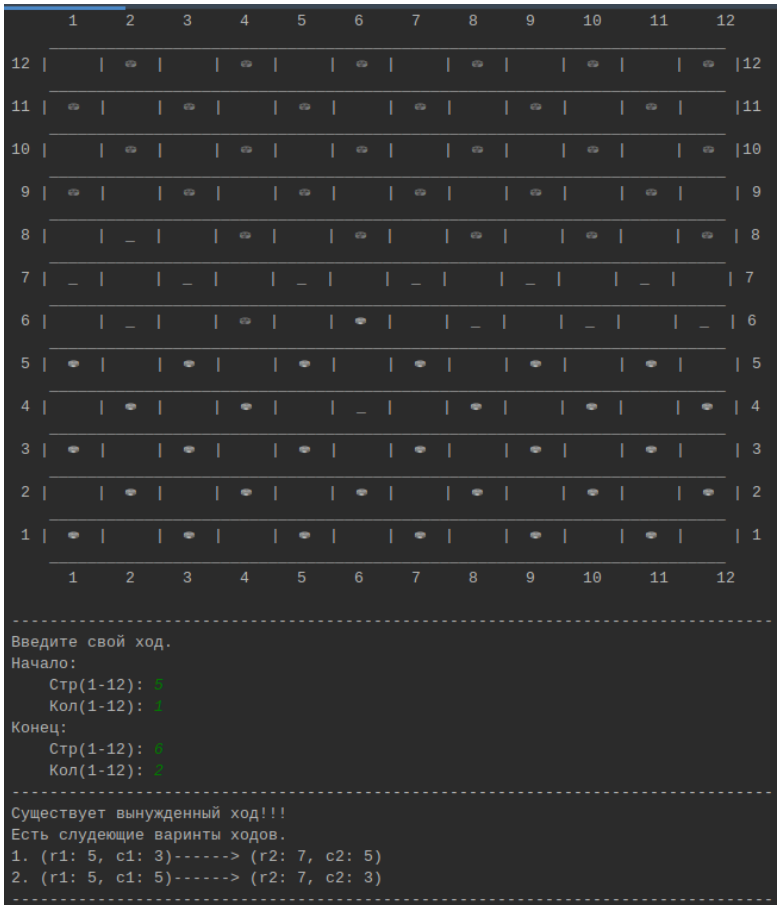


Скриншот 8

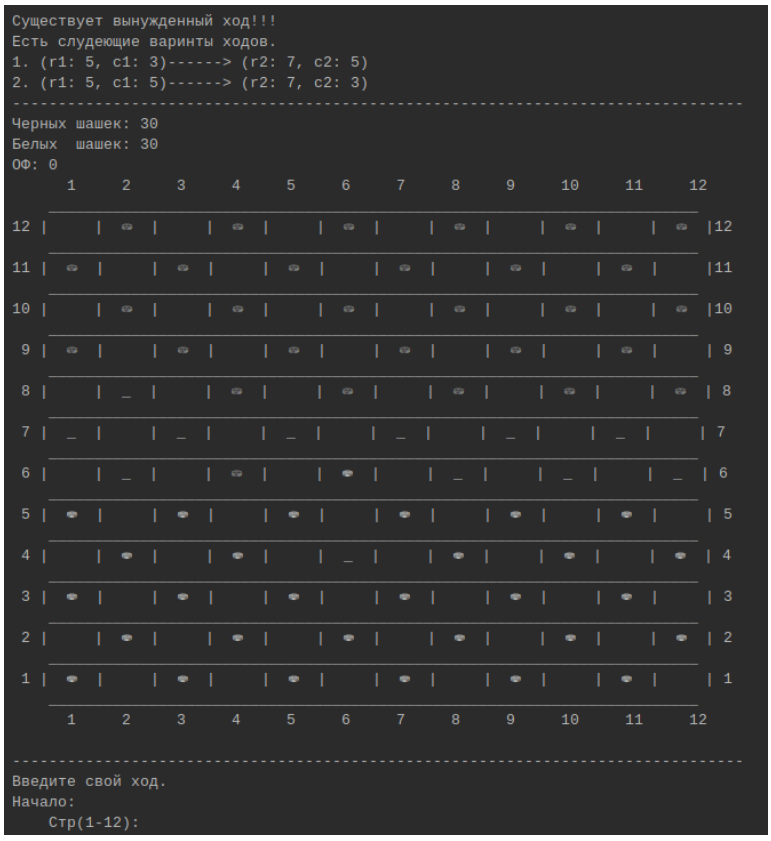


Скриншот 9

Предупреждение о необходимости поедания

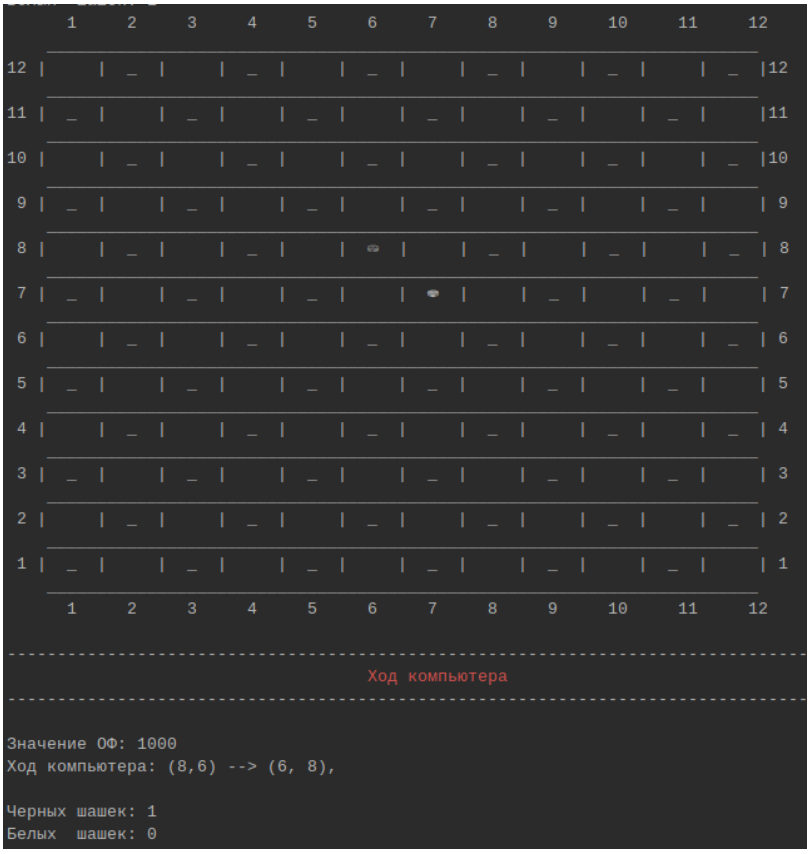


Скриншот 10

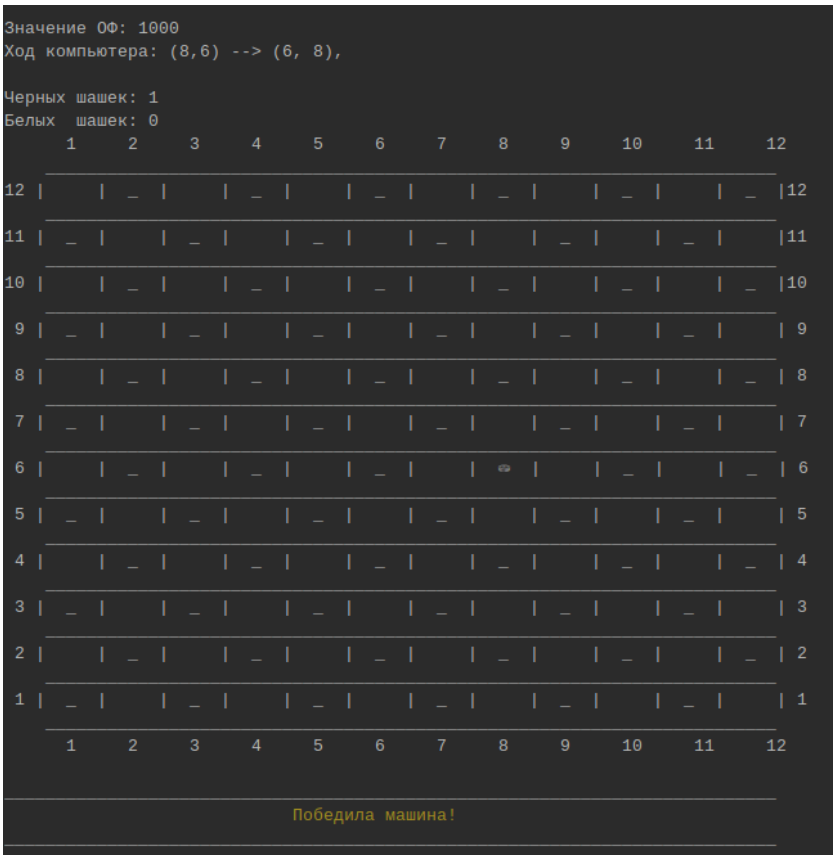


Скриншот 11

Победа компьютера

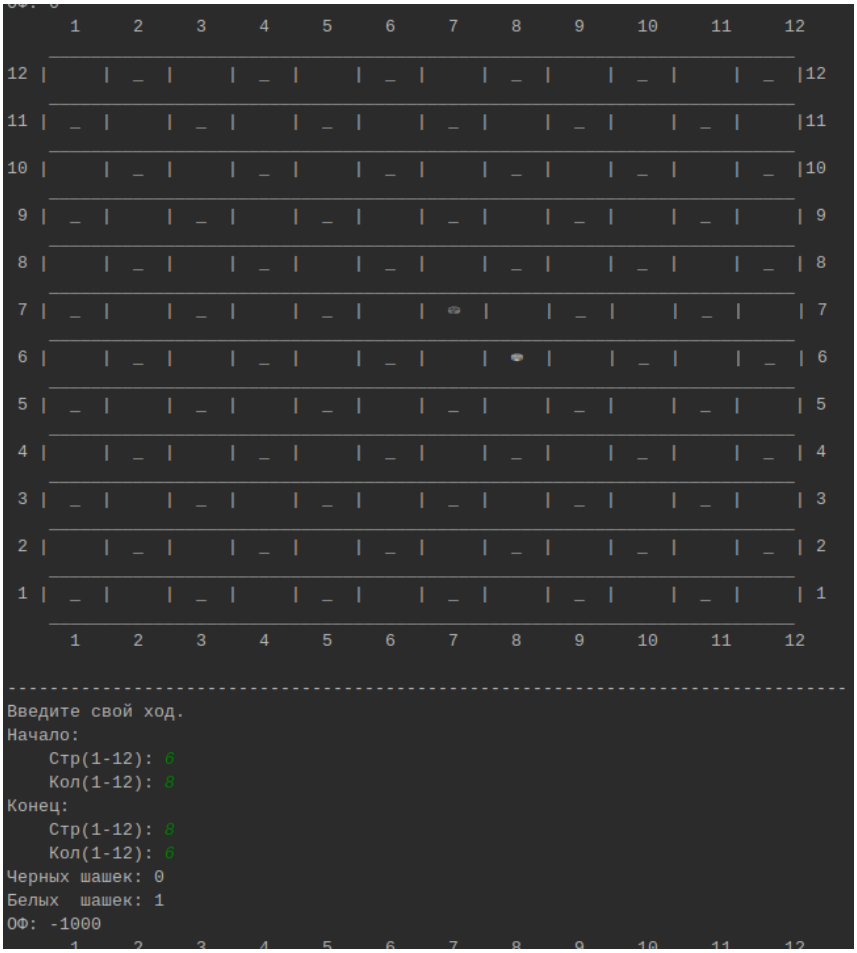


Скриншот 12

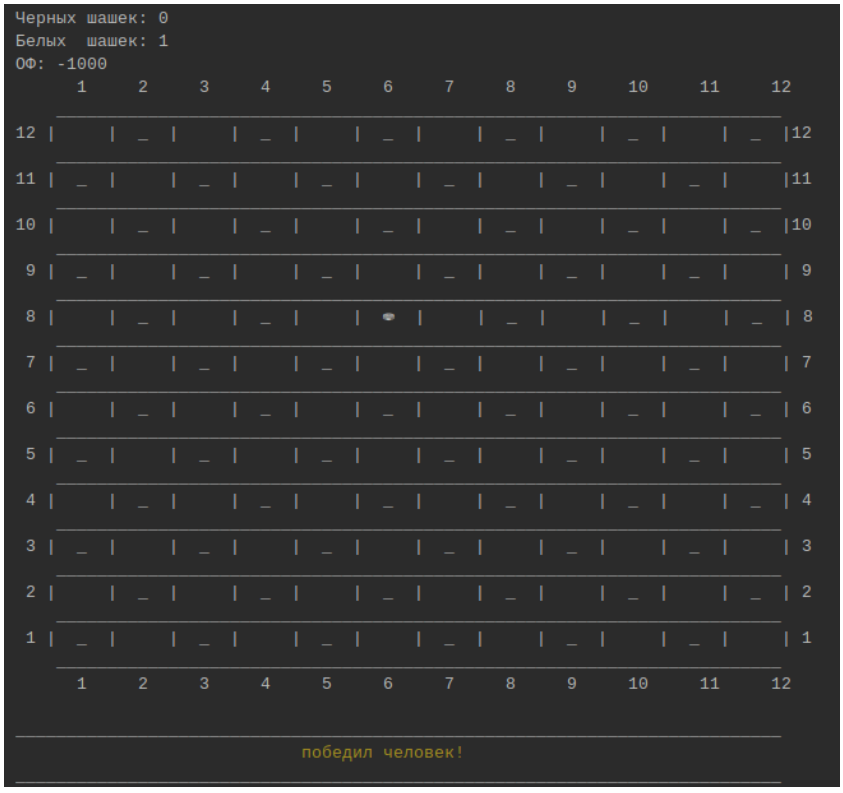


Скриншот 13

Победа игрока



Скриншот 14



Скриншот 15