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

Курсовая работа

Отчет

по курсовой работе

по дисциплине

Технологии программирования

РУКОВОДИТЕЛЬ:

_____ Капранов С.Н.
(подпись) (фамилия, и.,о.)

СТУДЕНТ:

_____ Гореев А. Д.
(подпись) (фамилия, и.,о.)

19-ИСТ-2
(шифр группы)

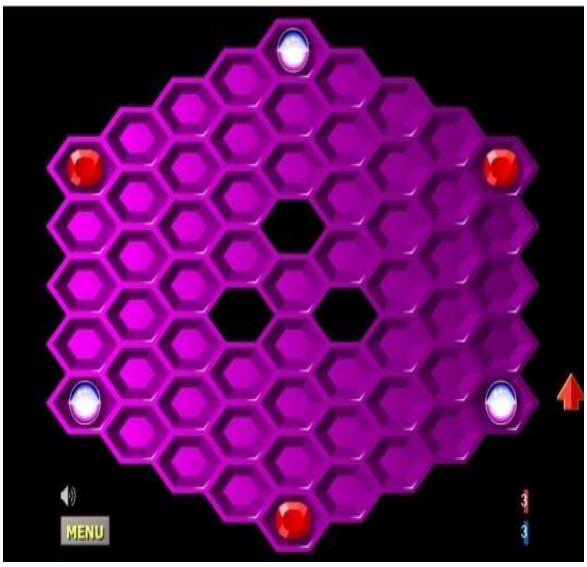
Работа защищена «___» _____

С оценкой _____

Нижний Новгород 2021

Текст задания

Вариант 7. Написать программу, играющую в игру Гексагон.



Правила игры

Для игры используются фишки двух цветов — красные и белые. Один игрок ходит красными, другой белыми. Изначально в угловые поля выкладываются по три фишки каждого цвета, цвета фишек в углах чередуются. Ход игрока заключается в следующем. Игрок выбирает на доске фишку своего цвета и пустое поле, в которое он ходит (оба поля указываются с помощью клика мыши). Это поле должно быть расположено не далее чем через одно поле от выбранной фишки. Доступные поля, в которые можно ходить, выделены желтым и зеленым цветами. Игрок может либо "удвоить" фишку — положить новую фишку на обозначенное желтым пустое поле, которое граничит с выбранной фишкой), либо переместить выбранную фишку на выделенное зеленым пустое поле, которое расположено через одно поле. Фишки противника, которые граничат с полем, в которое был сделан ход, захватываются — меняют свой цвет на цвет фишек игрока .Игра заканчивается, когда очередной игрок не может сделать ход. Побеждает игрок, фишек которого на доске больше.

Описание вычисления Оф

Для того чтобы определить насколько выигрышным (проигрышным) является тот или иной ход, была разработана оценочная функция. Эта функция позволяет ИИ выставить приоритет для хода при текущем состоянии игрового поля. В зависимости от уровня сложности игры, выбирается из возможных вариантов хода подходящий. Чем выше уровень сложности, тем выгоднее ход у компьютера.

```
int evalFun(char** field, int current_player)
{
    return Score(field, current_player) - Score(field, (current_player * 2) % 3);
}
```

Данная ОФ была реализованна в виде функции evalFun, которая осуществляет выбор наилучшего исхода для текущего игрока. Данная ОФ находит разность между количество наших фишек и количеством фишек противника.

Результат работы ОФ

Реализация минимаксного алгоритма искусственного интеллекта в игре Гексагон

Для решения игр с использованием ИИ используем понятие игрового дерева с последующим минимаксным алгоритмом. В игровом дереве узлы расположены на уровнях, соответствующих поворотам каждого игрока в игре, так что “корневой” узел дерева (обычно изображенный в верхней части диаграммы) является начальной позицией в игре. Под корнем, на втором уровне, есть возможные состояния, которые могут возникнуть в результате ходов первого игрока. Мы называем эти узлы” дочерними " от корневого узла.Каждый узел на втором уровне будет иметь в качестве своих дочерних узлов состояния, которые могут быть достигнуты из него ходами противоположного игрока. Это

продолжается, уровень за уровнем, до достижения состояний, в которых игра закончена. В крестики-нолики это означает, что либо один из игроков выигрывает, либо доска заполнена и игра заканчивается вничью

Алгоритм поиска, рекурсивно, лучший ход, который приводит Максимальный игрок, чтобы выиграть или не проиграть (ничья). Он рассматривает текущее состояние игры и доступные ходы в этом состоянии, а затем для каждого действительного хода он играет (чередуя min и max), пока не найдет конечное состояние (выигрыш, ничья или проигрыш)

Была реализованна функция

int minMax(char **field, int depth, int player, int current_player)

- **field: состояние игры в гексагон
- depth: индекс узла в игровом дереве. Может увеличиваться, так как мы задаем уровень игрока в начале игры.
- player: игрок, для которого мы и строим дерево
- current_player: текущий игрок. в начале игры player и current_player равны, изменяются только тогда, когда мы расчитываем возможные ходы для игрока и противника, возволяют не запутаться в вычислениях.

Если глубина равна нулю, то на доске нет новых пустых ячеек для игры. Или, если игрок выигрывает, то игра заканчивается на МАКС или мин. Таким образом, оценка для этого состояния будет возвращена.

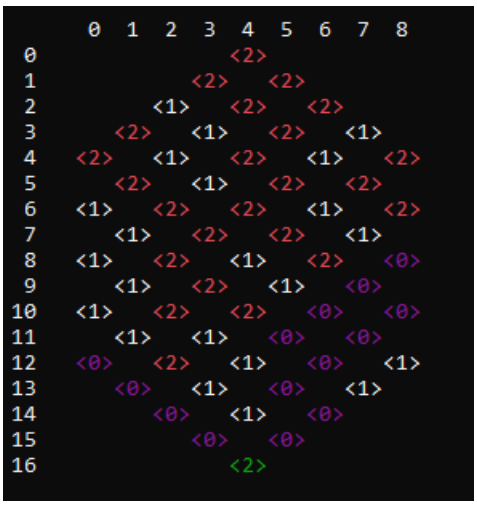
Для максимального игрока будет получен больший счет. Для минимального игрока будет получен более низкий балл. И в конце концов лучший ход возвращается.

С помощью рекурсии, получаем для каждого хода оценку, каждую позицию записываем откуда и куда ходим в массив. На каждом уровне итеративно, проводя поиск в ширину находим минимум или максимум.

Посте того, как прошли все дерево ищем максимальное значение среди минимальных. И находим в массиве позиции откуда и куда пойдет наша фишка.

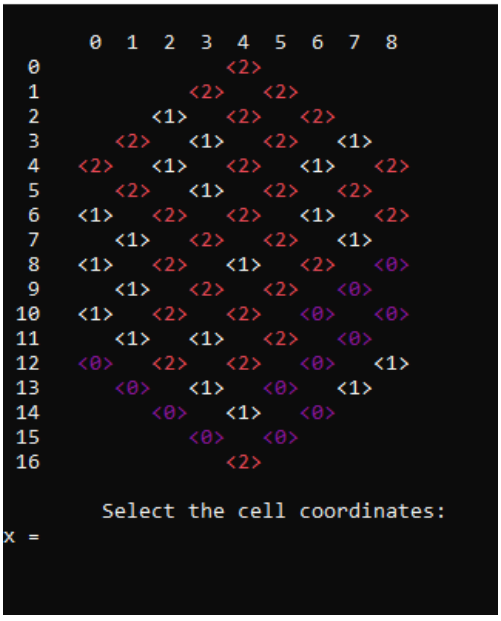
Так как у нас поле состоит из 61 поля, то достаточно сложно построить дерево даже для одной позиции, так как максимально возможное количестов ходов для одной фишки – это 18 состояний, если же фишек много, то и дерево разростается в геометрической прогрессии.

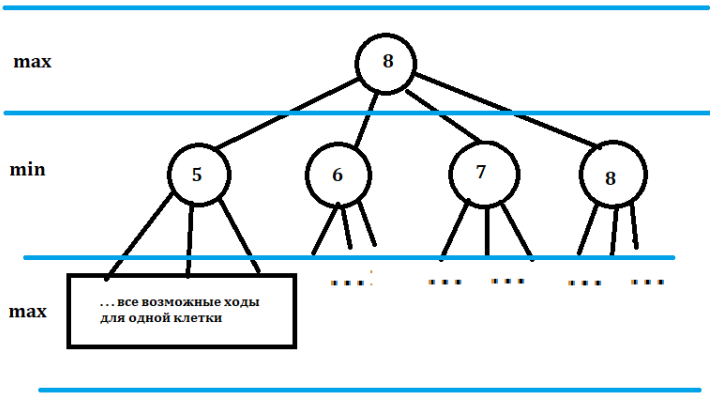
Примерное дерево



Рассматриваем все возможные ходы, как свои, так и ответные от противника. В результате получаем значения на следующем уровне относительно вершины полученные значения оценочной функции 5,6,7,8

Проходясь по массиву находим максимальное, у нас это 8. Из возможных ходов находим позицию откуда и куда нам ходить. В данном примере из клетки (4,10) в клетку(5,11)





продолжаем проходить по полю, смотря возможные ходы

Реализация альфа-бета отсечения

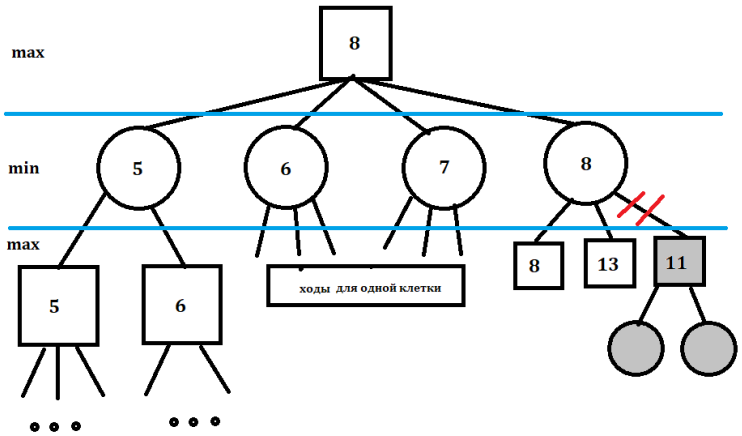
Альфа-бета отсечение основано на той идее, что анализ некоторых ходов можно прекратить досрочно, игнорируя результат их показаний. Если было найдено, что для этой ветви значение оценивающей функции в любом случае хуже, чем вычисленное для предыдущей ветви. Альфа-бета-отсечение является оптимизацией, так как не влияет на корректность работы алгоритма. Немного изменим нашу функцию минимакса, добавляем два дополнительных параметра alpha и beta.

```
int minMax(char **field, int level, int depth, int player, int current_player, int alpha, int beta);
```

- alpha — текущее максимальное значение, меньше которого игрок максимизации (компьютер) никогда не выберет (изначально -100)
- beta — текущее минимальное значение, больше которого игрок минимизации (человек) никогда не выберет (изначально + 100)

Те ветки, где alpha> beta отсекаются алгоритмом.

Рекурсивно проходясь по всем полям находим такие положения фишки, когда alpha> beta, тогда функция минимакса сразу возвращает оптимальный результат, не совершая лишних действий, которые только нагружают компьютер дополнительными вычислениями. Также, для уровня max запоминаем в alpha больший из результатов оценки, а на уровне минимум наименьшее значение, между beta и оптимальным значением. Возьмем в качестве примера предыдущий этап с минимаксом, в результате оценок мы пришли к выводу, что некоторые не нужные ветви лучше сразу отсечь.

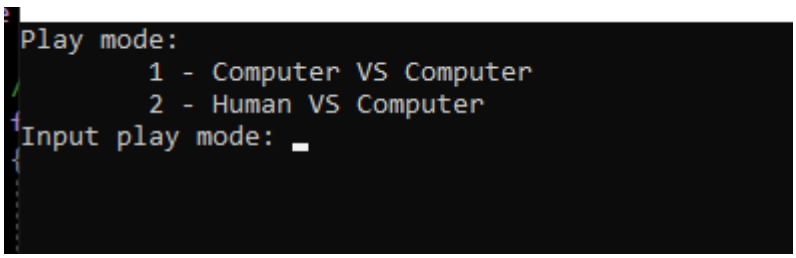


Список файлов проекта

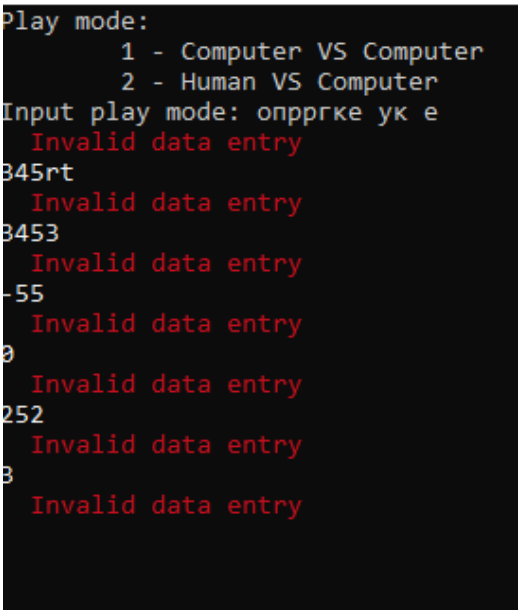
- Считывание поля из файла
- Проверка на выигрыш
- Функция, которая отвечает за игру
- Вывод поля на экран
- Отрисаовка новых фишек
- Проверка правильного выбрат координат
- Функция проверки перемещения, можем ли мы переместиться из клетки в клетку
- Функция, осуществляющая перемещение игрока
- Перемещение пользователя
- Создание ходов
- Игровой счет
- Функция Альфа-Беты
- Правильный ввод данных

Инструкция пользователя

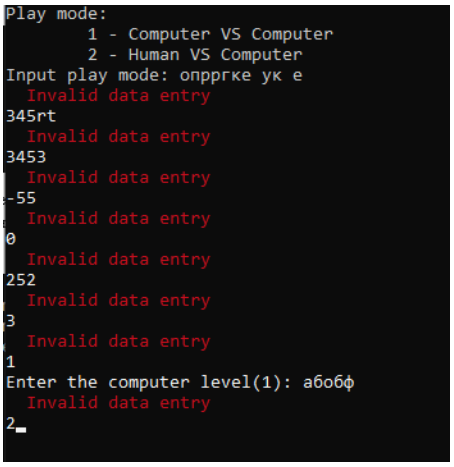
1) Пользователь взаимодействует с программой в текстовом режиме, через консоль. Для этого ему нужна клавиатура и мышь. При запуске игры пользователю дается выбор, играть с самому с AI или же AI с AI.



2) Дальше, если пользователь введет не те параметры, которые от него требуют, то программа выведет информацию и потребует ввести параметры снова.



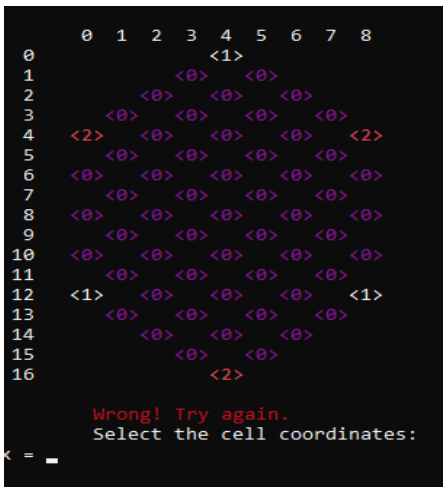
3) После того, как ввели необходимый параметр, программа потребует ввод уровня для AI.



4) Если ввод верный, то экран очищается и появляется поле, на котором может играть пользователь.

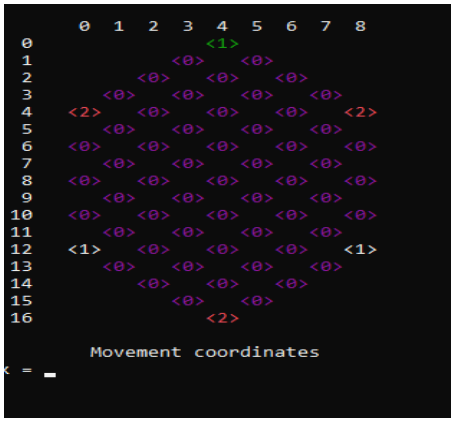
Дальше, пользователю нужно ввести координаты клетку откуда он собирается ходить. Горизонтальные значения – это значения X, а вертикальные – значения Y.

При неверном вводе программа будет требовать правильный ввод.



Если пользователь ввел координаты, но там не оказалось фишки, то его попросят ввести правильные значения.

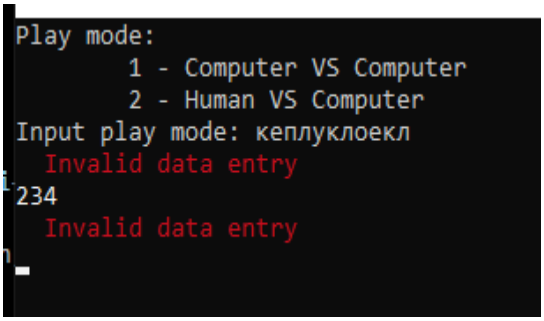
5) После того, как ввод начальных координат оказался верным, клетка, откуда мы ходим, окрашиватеся в зеленый цвет. Дальше пользокателя просят ввести конечные координаты.

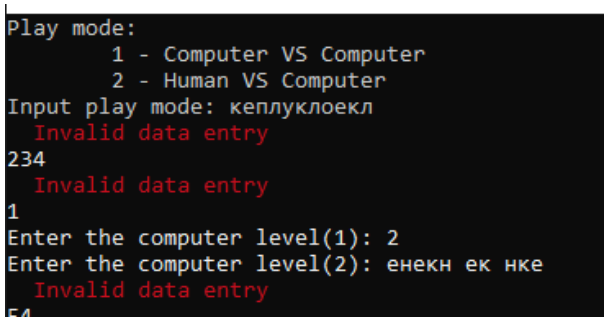


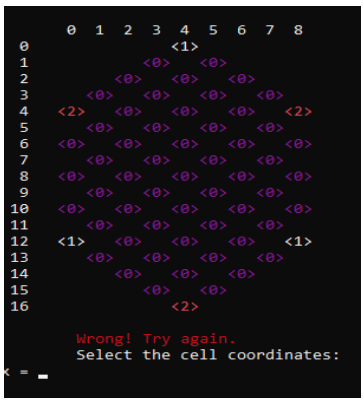
Мы сходили своей фишкой, а враг своей

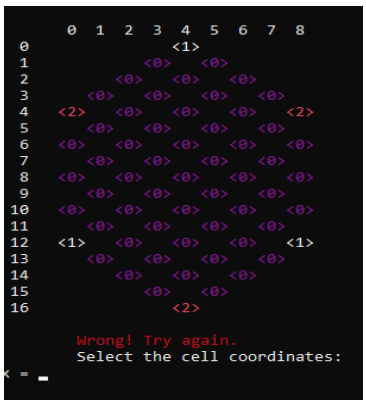
Результаты работы программы

- Защита от дурака

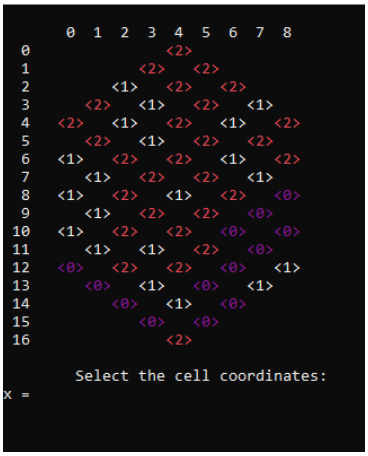


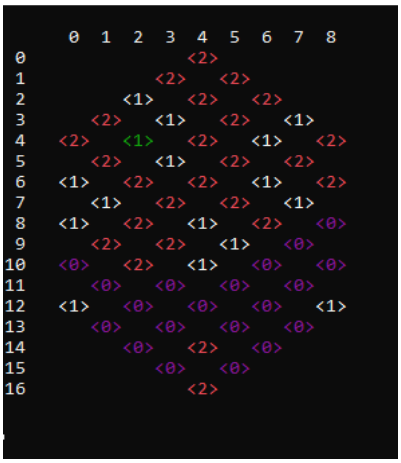


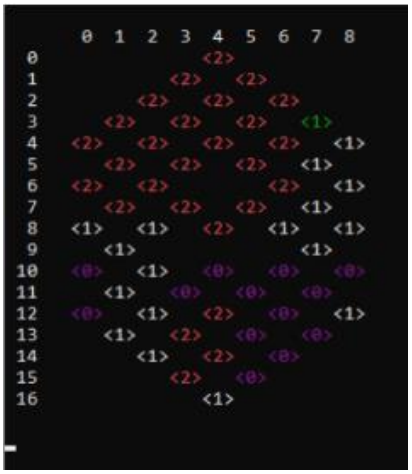




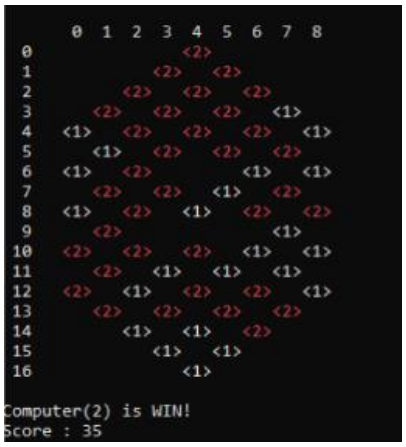
- Ход игры

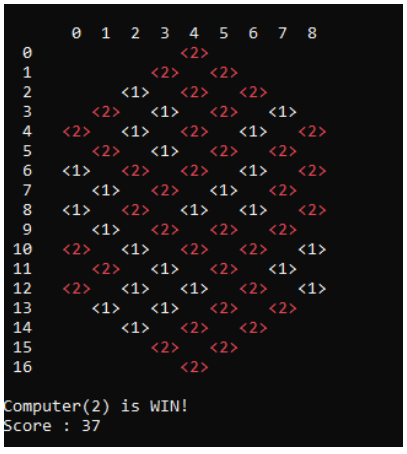


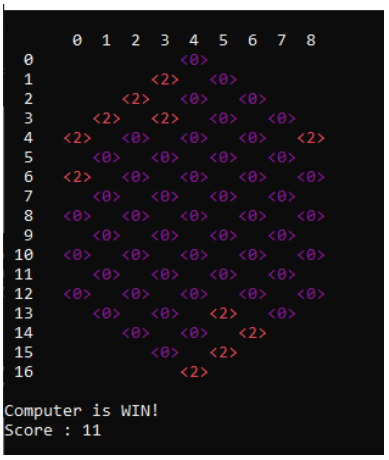




- Вывод







Листинг программы

```
//Заголовочный файл hexagon.h
#ifndef H_HEXAGON
#define H_HEXAGON

#include <windows.h>
#include <iostream>
#include <fstream>
using namespace std;

const int mapWidth = 9;
```

```
const int mapHeight = 17;
const int infinity = 100;
//Считывание поля из файла
void scanField(char** field, ifstream& file);
//Проверка на выигрыш
void isWin(char** field, int mode);
//Осноная функция, которая отвечает за игру
void game(char** field, int mode);
//Вывод поля на экран
void printField(char** field);
//Отрисаовка новых фишек
void paintCell(char** field, int x, int y);
//Проверка правильного выбрат координат
bool checkChoice(char** field, int x, int y, int player);
//Проверка перемещения,можем ли мы переместиться из клетки А в Б
bool checkMove(char** field, int x1, int y1, int x2, int y2, int
player);
//Перемещение игрока
void movePlayer(char** field, int x1, int y1, int x2, int y2,
int player);
//Перемещение пользователя
void moveHuman(char** field);
//Создание ходов
bool generateMoves(char** field, int player);
//Игровой счет
int Score(char** field, int player);
//Функция Альфа-Беты
int minMax(char ** field, int level, int depth, int player, int
current_player, int alpha, int beta);
//Правильный ввод данных
int validInput(int& input);
void validText();
int validInpParam(int& input, int param1, int param2);
#endif

//Файл hexagon.cpp
#include "hexagon.h"
#include <fstream>
#include <Windows.h>
#include <cmath>
using namespace std;

HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);

const int x[18] = { -1, 1, 2, 1, -1, -2, -4, -3, -2, 0, 2, 3, 4,
3, 2, 0, -2, -3 };
const int y[18] = { 1, 1, 0, -1, -1, 0, 0, 1, 2, 2, 2, 1, 0, -1,
-2, -2, -2, -1 };

const int player1 = 1;
const int player2 = 2;
```

```
//Считывание файла, генерация игрового поля
void scanField(char** field, ifstream& file)
{
     for (int i = 0; i < mapHeight; i++)
     {
          field[i] = new char[mapHeight];
     }

     for (int i = 0; i < mapHeight; i++)
     {
          for (int j = 0; j < mapWidth; j++)
          {
               field[i][j] = file.get();
          }
          file.get();
     }
}

void isWin(char** field, int mode)
{
     if (Score(field, 1) > Score(field, 2))
     {
          if (mode == 2)
               cout << "You are WIN!\n" << "Score : " <<
               Score(field, 1) << endl;
          if (mode == 1)
               cout << "Computer(1) is WIN!\n" << "Score : " <<
               Score(field, 1) << endl;
     }
     else if (Score(field, 1) < Score(field, 2))
     {
          if (mode == 2)
               cout << "Computer is WIN!\n" << "Score : " <<
               Score(field, 2) << endl;
          if (mode == 1)
               cout << "Computer(2) is WIN!\n" << "Score : " <<
               Score(field, 2) << endl;
     }

     else
     {
          cout << "Standoff\n" << Score(field, 1) << " : " <<
          Score(field, 1) << endl;
     }
}

void validText()
{
     SetConsoleTextAttribute(hStdOut, FOREGROUND_RED);
     cout << "  Invalid data entry" << endl;
     SetConsoleTextAttribute(hStdOut, FOREGROUND_RED |
     FOREGROUND_BLUE | FOREGROUND_GREEN | FOREGROUND_INTENSITY);
}
```

```
int validInput(int& input)
{
     while (!(cin >> input) || (cin.peek() != '\n'))
     {
          cin >> input;
          cin.clear();
          cin.ignore(1000, '\n');
          validText();
     }
     return input;
}

int validInpParam(int& input, int param1, int param2)
{
     do
     {
          validInput(input);
          if (input < param1 || input>param2)
          {
               validText();
          }
     } while (input < param1 || input>param2);
     return input;
}

int evalFun(char** field, int current_player)
{
     return Score(field, current_player) -
            Score(field,      (current_player * 2) % 3);
}

void game(char** field, int mode)
{
     int compLevel_1 = 1;
     int compLevel_2 = 1;

     if (mode == 2)
     {
          cout << "Enter the computer level: ";
          validInpParam(compLevel_2,1,10);
     }

     else
     {
          cout << "Enter the computer level(1): ";
          validInpParam(compLevel_1,1,10);
          cout << "Enter the computer level(2): ";
          validInpParam(compLevel_2,1,10);
     }

     system("cls");
```

```
     printField(field);

     while (true)
     {
          if (mode == 2)
          {
               if (generateMoves(field, player1))
               {
                    moveHuman(field);
               }
               else
               {
                    break;
               }
          }

          else if (generateMoves(field, player1) && mode == 1)
          {
               cout << "Evaluation function for the first player"
                    << evalFun(field, player1) << endl;
               //   system("pause");
               minMax(field, 1, compLevel_1, 1, player1, -
                    infinity, infinity);
          }

          else
          {
               break;
          }

          if (generateMoves(field, player2))
          {
              cout << "evaluation function for the second player"
                   << evalFun(field, player2) << endl;
               //   system("pause");
               minMax(field, 1, compLevel_2, 2, player2, -
               infinity, infinity);
          }

          else
          {
               break;
          }

     }

     isWin(field, mode);
}

void colorField(char** field, int& i, int& j)
{
     if (field[i][j] == '1')
     {
```

```
            SetConsoleTextAttribute(hStdOut, FOREGROUND_RED |
            FOREGROUND_BLUE | FOREGROUND_GREEN |
            FOREGROUND_INTENSITY);
            cout << '<' << field[i][j] << '>';
        }
        if (field[i][j] == '2')
        {
            SetConsoleTextAttribute(hStdOut, FOREGROUND_RED |
            FOREGROUND_INTENSITY);
            cout << '<' << field[i][j] << '>';
        }

        SetConsoleTextAttribute(hStdOut, FOREGROUND_RED |
        FOREGROUND_BLUE | FOREGROUND_GREEN | FOREGROUND_INTENSITY);

        if (field[i][j] == '0')
        {
            SetConsoleTextAttribute(hStdOut, FOREGROUND_RED |
            FOREGROUND_BLUE);
            cout << '<' << field[i][j] << '>';
        }

        if (field[i][j] == ' ')
        {
            cout << "   ";
        }
    }

    void printField(char** field)
    {
        cout << "\n       0  1  2  3  4  5  6  7  8 " << endl;
        for (int i = 0; i < mapHeight; i++)
        {
            SetConsoleTextAttribute(hStdOut, FOREGROUND_RED |
            FOREGROUND_BLUE | FOREGROUND_GREEN |
            FOREGROUND_INTENSITY);

            if (i < 10)
            {
                cout << "  " << i << "   ";
            }
            else
            {
                cout << " " << i << "   ";
            }
            for (int j = 0; j < mapWidth; j++)
            {
                colorField(field, i, j);
            }

            cout << endl;
        }
        cout << endl;
```

```
}

void paintCell(char** field, int x, int y)
{

     cout << "\n       0  1  2  3  4  5  6  7  8 " << endl;
     for (int i = 0; i < mapHeight; i++)
     {
          SetConsoleTextAttribute(hStdOut, FOREGROUND_RED |
          FOREGROUND_BLUE | FOREGROUND_GREEN |
          FOREGROUND_INTENSITY);
          if (i < 10)
          {
               cout << "  " << i << "   ";
          }
          else
          {
               cout << " " << i << "   ";
          }
          for (int j = 0; j < mapWidth; j++)
          {
               if (i == x && j == y)
               {
                    SetConsoleTextAttribute(hStdOut,
                    FOREGROUND_GREEN);
                    cout << '<' << field[i][j] << '>';

                    SetConsoleTextAttribute(hStdOut,
                    FOREGROUND_RED | FOREGROUND_BLUE |
                    FOREGROUND_GREEN | FOREGROUND_INTENSITY);
               }
               else
               {
                    colorField(field, i, j);
               }
          }
          cout << endl;
     }
     cout << endl;
}

bool checkChoice(char** field, int x, int y, int player)
{
     if (x >= 0 && y >= 0 && x < mapHeight && y < mapWidth &&
field[x][y] == 48 + player)
     {
          system("cls");
          paintCell(field, x, y);
          return true;
     }
     return false;
}
```

```
bool checkMove(char** field, int x1, int y1, int x2, int y2, int
player)
{
     if (x2 >= 0 && y2 >= 0 && x2 < mapHeight && y2 < mapWidth &&
     field[x2][y2] == '0' && abs(x1 - x2) <= 4 && abs(y1 - y2)
                     <= 2 && abs(x1 - x2) + abs(y1 - y2) <= 4)
     {
          return true;
     }
     return false;
}

void movePlayer(char** field, int x1, int y1, int x2, int y2,
int player)
{
     field[x2][y2] = 48 + player;

     if (abs(x2 - x1) + abs(y2 - y1) > 2 || abs(x2 - x1) +
          abs(y2 - y1) == 2 && abs(y2 - y1) == 2)
     {
          field[x1][y1] = 48;
     }
     //перекрашивание соседних клеток
     if (x2 - 2 >= 0)
     {
          if (field[x2 - 2][y2] != '0'&&field[x2 - 2][y2] !=' ')
               field[x2 - 2][y2] = 48 + player;
     }

     if (x2 - 1 >= 0 && y2 + 1 < mapWidth)
     {
          if (field[x2 - 1][y2 + 1] != '0' &&
               field[x2 - 1][y2 + 1] != ' ')
          {
               field[x2 - 1][y2 + 1] = 48 + player;
          }
     }

     if (x2 + 1 < mapHeight && y2 + 1 < mapWidth)
     {
          if (field[x2 + 1][y2 + 1] != '0' &&
                field[x2 + 1][y2 + 1] != ' ')
          {
               field[x2 + 1][y2 + 1] = 48 + player;
          }
     }

     if (x2 + 2 < mapHeight)
     {
          if (field[x2 + 2][y2] != '0' &&
                    field[x2 + 2][y2] != ' ')
          {
               field[x2 + 2][y2] = 48 + player;
```

```
            }
        }

        if (y2 - 1 >= 0 && x2 + 1 < mapHeight)
        {
            if (field[x2 + 1][y2 - 1] != '0' &&
                    field[x2 + 1][y2 - 1] != ' ')
            {
                field[x2 + 1][y2 - 1] = 48 + player;
            }
        }

        if (x2 - 1 >= 0 && y2 - 1 >= 0)
        {
            if (field[x2 - 1][y2 - 1] != '0' &&
                     field[x2 - 1][y2 - 1] != ' ')
            {
                field[x2 - 1][y2 - 1] = 48 + player;
            }
        }
    }

    void moveHuman(char** field) {
        //координаы текущей ячейки
        int x1 = 0, y1 = 0;
        //координаты перехода
        int x2 = 0, y2 = 0;
        while (true)
        {
            puts("\tSelect the cell coordinates: ");
            cout << "x = ";
            validInput(x1);
            cout << "y = ";
            validInput(y1);
            if (checkChoice(field, x1, y1, 1))
            {
                puts("\tMovement coordinates");
                cout << "x = ";
                validInput(x2);
                cout << "y = ";
                validInput(y2);
                if (checkMove(field, x1, y1, x2, y2, 1))
                {
                    movePlayer(field, x1, y1, x2, y2, 1);
                    system("cls");
                    printField(field);
                    break;
                }
                else
                {
                    system("cls");
                    printField(field);
```

```
                    SetConsoleTextAttribute(hStdOut,
                    FOREGROUND_RED);
                    puts("\tWrong! Try again.");

                    SetConsoleTextAttribute(hStdOut,
                    FOREGROUND_RED | FOREGROUND_BLUE |
                    FOREGROUND_GREEN | FOREGROUND_INTENSITY);
               }
          }
          else
          {
               system("cls");
               printField(field);

               SetConsoleTextAttribute(hStdOut, FOREGROUND_RED);
               puts("\tWrong! Try again.");

               SetConsoleTextAttribute(hStdOut, FOREGROUND_RED |
               FOREGROUND_BLUE | FOREGROUND_GREEN |
               FOREGROUND_INTENSITY);
          }
     }
}

//Проходимся по клеткам и находим возможные ходы
bool generateMoves(char** field, int player)
{

     //   int x[18] = { -1, 1, 2, 1, -1, -2, -4, -3, -2, 0, 2, 3,
4, 3, 2, 0, -2, -3 };
     //   int y[18] = { 1, 1, 0, -1, -1, 0, 0, 1, 2, 2,  2, 1, 0,
-1, -2, -2, -2, -1 };

     //   const int amount_move = 18;

     for (int i = 0; i < mapHeight; i++)
     {
          for (int j = 0; j < mapWidth; j++)
          {
               if (field[i][j] == 48 + player)
               {
                    for (int k = 0; k < 18; k++)
                    {
                         if (checkMove(field, i, j, i +
                              x[k], j + y[k], player))
                         {
                              return true;
                         }
                    }
               }
          }
     }
     return false;
```

```
}

int Score(char** field, int player)
{
     int score = 0;

     for (int i = 0; i < mapHeight; i++)
     {
          for (int j = 0; j < mapWidth; j++)
          {
               if (field[i][j] == player + 48)
                    score++;
          }
     }

     return score;
}

int minMax(char** field, int level, int depth, int player, int
current_player, int alpha, int beta)
{

     char copy[mapHeight][mapWidth];

     const int all_move = 1100;

     const int amount_move = 18;

     int begin_x[all_move];

     int begin_y[all_move];

     int end_x[all_move];

     int end_y[all_move];

     int value[all_move];

     int counter_move = 0, i, j, optimal;
     //Изначальные параметры в начале рассчетов
     if (level % 2 == 0)
     {
          optimal = infinity;
     }
     else
     {
          optimal = -infinity;
     }
     //Копируем начальное состояние доски
     for (i = 0; i < mapHeight; i++)
     {
          for (j = 0; j < mapWidth; j++)
```

```
            {
                copy[i][j] = field[i][j];
            }
        }
        //Если текущий уровень больше глубины дерева
        if (level > depth)
        {
            evalFun(field, current_player);
        }

        else
        {
            //проходимся по каждой клетке
            for (i = 0; i < mapHeight; i++)
            {
                for (j = 0; j < mapWidth; j++)
                {
                    //смотрим, возможно ли сходить на текущую
                    //клетку в матрице
                    if (checkChoice(field, i, j, player))
                    {
                        //Проходимся по всем возможным ходам для
                        //одной позиции
                        for (int k = 0; k < amount_move; k++)
                        {
                            //Досрочно прекращаем минимакс
                            if (alpha > beta)
                            {
                                return optimal;
                            }
                            //Далее проверяем, возможно ли
                            //сходить в текущую клетку
                            if (checkMove(field, i, j, i +
                                x[k], j + y[k], player))
                            {
                                //начальные координаты
                                begin_x[counter_move] = i;

                                begin_y[counter_move] = j;
                                //координаты, куда сходит
                                //фишка
                                end_x[counter_move] = i +
                                                      x[k];

                                end_y[counter_move] = j +
                                                      y[k];
                                //Смотрим, возможно ли ходить
                                //в клетку
movePlayer(field, begin_x[counter_move], begin_y[counter_move],
end_x[counter_move], end_y[counter_move], player);
//Если уровень в дереве четный
if (level % 2 == 0)
{
```

```
     //Рекурсивно проходимся по функции, оценивая наши шансы с
     //противником
     value[counter_move] = minMax(field, level + 1, depth,
               (player * 2) % 3, current_player, alpha, beta);
     //Смотрим, минимум на уровне
     optimal = ((optimal < value[counter_move]) ? optimal :
                    value[counter_move]);
     //Далее записываем значение минимума в массив
     value[counter_move] = optimal;
     beta = ((beta < optimal) ? beta : optimal);
}
//Если не четный
else
{
     //Переход на новый уровень в дереве, смотрим возможные ходы
     value[counter_move] = minMax(field, level + 1, depth,
               (player * 2) % 3,current_player, alpha, beta);
     //Находим максимум на уровне
     optimal = ((optimal > value[counter_move]) ? optimal :
                                        value[counter_move]);
     //Запоминаем значение
     value[counter_move] = optimal;
     alpha = ((alpha > optimal) ? alpha : optimal);
}
//Возвращаем прежнее соостояние на доске
for (int t1 = 0; t1 < mapHeight; t1++)
{
     for (int t2 = 0; t2 < mapWidth; t2++)
     {
          field[t1][t2] = copy[t1][t2];
     }
}
counter_move++;
}}}
}
}
//Остановка рекурсии
//Если уровень в дереве не единица, то возвращаем оптимальное
//значение функции
//нужно для того, чтобы понять какая оценка, будет для данной
//позиции
if (level != 1)
{
     return optimal;
}
//Если мы дошли до первого уровня в рекурсии и произвели поиск в
//ширину в дереве
if (level == 1 && i == mapHeight)
{
     //Находим максимальное значение на первом уровне
     //и перемещаем фишку в клетку, с координатами
     //рассчитанными с помощью минимакса
     int max_value = value[0], max_index = 0;
```

```
	for (int k = 1; k < counter_move; k++)
	{
		if (max_value < value[k])
		{
			max_value = value[k];
			max_index = k;
		}
	}
	movePlayer(field, begin_x[max_index], begin_y[max_index],
			end_x[max_index], end_y[max_index], player);
	system("cls");
	printField(field);
}
}
}
//Файл Game.cpp
#include "hexagon.h"
#include <algorithm>
#include <iostream>
#include<fstream>

using namespace std;


int main(int argc, char* argv[]) {

	if (argc < 2)
	{
		cout << "Wrong format" << endl;
		return -1;
	}

	ifstream input_file(argv[1]);

	if (!input_file) {
		cout << "Error writing file" << endl;
		return -3;
	}

	char* map[mapHeight];
	int mode;

	cout << "Play mode:\n    1 - Computer VS Computer\n    2 -
Human VS Computer\n";
	cout << "Input play mode: ";
	validInpParam(mode,1,2);

	scanField(map, input_file);
	game(map, mode);

	input_file.close();

	for (int i = 0; i < mapHeight; i++)
```

```
            delete[] map[i];


        return 0;
    }
```