Санкт-Петербургский Политехнический Университет Петра Великого

Институт компьютерных наук и технологий

Высшая школа интеллектуальных систем и суперкомпьютерных технологий

**КУРСОВОЙ ПРОЕКТ**

**Дисциплина:** Алгоритмы и структуры данных

**Тема:** Разработка GUI приложения «Автоматический решатель тетриса»

Выполнил

студент гр. 3530903/80003 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Никифоров Т. А.

(подпись)

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Глухих М. И.

(подпись)

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2019 г.

Санкт-Петербург   
2019

**Содержание**

1. [**Техническое задание** 3](#_Toc26959442)
2. [**Метод решения** 4](#_Toc26959443)
3. [**Листинг программы** 6](#_Toc26959444)
4. [**Tetrominoe.java** 6](#_Toc26959445)
5. [**Shape.java** 6](#_Toc26959446)
6. [**Board.java** 8](#_Toc26959447)
7. [**Tetris.java** 16](#_Toc26959448)
8. [**Solver.java** 19](#_Toc26959449)
9. [**GeneticWeights.java** 26](#_Toc26959450)
10. [**Тесты** 33](#_Toc26959451)
11. [**Скриншоты программы** 37](#_Toc26959452)

# **Техническое задание**

Необходимо создать автоматический решатель тетриса. Считать, что известно только одна следующая фигура.

Правила игры Тетрис:

Случайные фигурки тетрамино падают сверху в прямоугольный стакан шириной 10 и высотой 20 клеток. В полёте игрок может поворачивать фигурку на 90° и двигать её по горизонтали. Также можно «сбрасывать» фигурку, то есть ускорять её падение, когда уже решено, куда фигурка должна упасть. Фигурка летит до тех пор, пока не наткнётся на другую фигурку либо на дно стакана. Если при этом заполнился горизонтальный ряд из 10 клеток, он пропадает и всё, что выше него, опускается на одну клетку. Дополнительно показывается фигурка, которая будет следовать после текущей. Игра заканчивается, когда новая фигурка не может поместиться в стакан. Очки начисляются за каждый заполненный ряд, поэтому задача игрока (решателя) — заполнять ряды, не заполняя сам стакан (по вертикали) как можно дольше.

GitHub репозиторий проекта: <https://github.com/nikiforovta/TetrisSolver>

# **Метод решения**

Код разделен на два пакета классов: game и solver.

Содержимое пакета game:

* Tetrominoe.java – класс- перечисление, в котором приведены характеристики фигур на игровом поле.
* Shape.java – класс, отвечающий за отображение и изменение координат фигуры на игровом поле.
* Board.java – класс, отвечающий за графическое представление игрового поля.
* Tetris.java – класс, отвечающий за графическое представление всего приложения.

Cодержимое пакета solver:

* Solver.java – класс, в котором происходит оценка данного хода.
* GeneticWeights.java – класс, в котором генетически создаются штрафные (весовые) параметры для оценки хода.

Описание работы приложения:

При запуске приложения начинается новая игра. Пользователь может видеть счёт, количество очищенных линий, время от начала игры и следующую фигуру.

Помимо этого, пользователю доступно выпадающее меню, содержащее следующие кнопки:

* Game
  + New – начать новую игру
  + Settings – настройки игры
    - Turn on/off solver – включить/выключить решателя
    - Reset genesis – сброс штрафных параметров и новая генерация
    - Game speed – выбор скорости игры
  + Exit – покинуть игру
* Help
  + About – короткое описание приложения
  + Controls – описание управления игрой
  + Factors – список весовых коэффициентов, использованных на момент вызова этого диалогового окна

Игра ведётся по стандартным правилам, приведённым в техническом задании. При включении соответствующей кнопки меню или сочетания клавиш, управление игровым процессом переходит решателю (метод solver из класса Solver), который получает игровое поле, нынешнюю и следующую фигуру, производит оценку вариантов размещения нынешней (метод gradeCurrent) и следующей фигуры (метод gradeNext) на игровом поле и выбирает лучшую тактику (метод makeMove).

Для оценки ходов используются весовые коэффициенты, которые изначально заданы вручную, но далее меняются в процессе работы генетического алгоритма (метод startGeneration из класса GeneticWeights). Генетический алгоритм работает параллельно основному процессу игры, проводя соревнование геномов (метод playGame), состоящих их 4 штрафных коэффициентов: «штраф» за высоту колонок игрового поля, «штраф» за очищенные линии, «штраф» за дырки на игровом поле и «штраф» за перепад высот между колонками. В результате соревнования 30% худших геномов заменяются новыми геномами (метод createNewGeneration), полученными в результате скрещивания (метод crossover) с мутацией (метод mutation) , а лучший геном передаётся решателю для дальнейшего использования при оценке возможных ходов.

**Описание фигур из класса Tetrominoe.java**

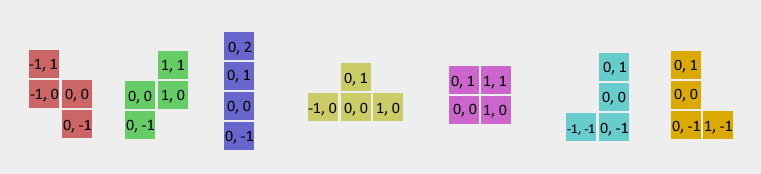
Ниже приведены изображения фигур, появляющихся на игровом поле. Внутри каждой клетки фигуры подписаны координаты, отображённые в классе Tetrominoe.java. Поворот, перемещение фигуры и проверка возможности этого самого перемещения проходят относительно «центральной точки» фигуры, которая имеет координаты (0, 0).

Рис. 1. Изображение фигур с подробными координатами

# **Листинг программы**

# **Tetrominoe.java**

package game;  
  
import java.awt.\*;  
  
public enum Tetrominoe {  
 *NoShape*(new int[][]{{0, 0}, {0, 0}, {0, 0}, {0, 0}}, new Color(0, 0, 0)), *S*(new int[][]{{0, -1}, {0, 0}, {-1, 0}, {-1, 1}}, new Color(204, 102, 102)),  
 *Z*(new int[][]{{0, -1}, {0, 0}, {1, 0}, {1, 1}}, new Color(102, 204, 102)),  
 *I*(new int[][]{{0, -1}, {0, 0}, {0, 1}, {0, 2}}, new Color(102, 102, 204)),  
 *T*(new int[][]{{-1, 0}, {0, 0}, {1, 0}, {0, 1}}, new Color(204, 204, 102)),  
 *O*(new int[][]{{0, 0}, {1, 0}, {0, 1}, {1, 1}}, new Color(204, 102, 204)),  
 *J*(new int[][]{{-1, -1}, {0, -1}, {0, 0}, {0, 1}}, new Color(102, 204, 204)),  
 *L*(new int[][]{{1, -1}, {0, -1}, {0, 0}, {0, 1}}, new Color(218, 170, 0));  
  
 public int[][] coords;  
 public Color color;  
  
 Tetrominoe(int[][] coords, Color c) {  
 this.coords = coords;  
 color = c;  
 }  
}

# **Shape.java**

package game;  
  
import java.util.Arrays;  
import java.util.Collections;  
import java.util.LinkedList;  
  
public class Shape {  
 private Tetrominoe pieceShape;  
 private int[][] coords;  
 private LinkedList<Tetrominoe> bag = new LinkedList<>(); //Список фигур, необходимый для улучшенного случайного подбора следующей фигуры  
  
  
 public Shape() {  
 coords = new int[4][2];  
 setShape(Tetrominoe.*NoShape*);  
 }  
  
 private void setX(int index, int x) {  
 coords[index][0] = x;  
 }  
  
 private void setY(int index, int y) {  
 coords[index][1] = y;  
 }  
  
 public int x(int index) {  
 return coords[index][0];  
 }  
  
 public int y(int index) {  
 return coords[index][1];  
 }  
  
 public Tetrominoe getShape() {  
 return pieceShape;  
 }  
  
 public void setShape(Tetrominoe shape) {  
 for (int i = 0; i < 4; i++) {  
 System.*arraycopy*(shape.coords[i], 0, coords[i], 0, 2);  
 }  
 pieceShape = shape;  
 }  
  
 */\*\*  
 \* Метод, выбирающий случайную фигуру из "мешка" фигур  
 \*/* void setRandomShape() {  
 if (bag.size() == 0) {  
 bag = new LinkedList<>(Arrays.*asList*(Tetrominoe.*values*()));  
 bag.remove(0);  
 }  
 Collections.*shuffle*(bag);  
 setShape(bag.removeFirst());  
 }  
  
 int minY() {  
 int m = coords[0][1];  
 for (int i = 0; i < 4; i++) {  
 m = Math.*min*(m, coords[i][1]);  
 }  
 return m;  
 }  
  
 */\*\*  
 \* Метод поворота фигуры на 90 градусов против часовой стрелки  
 \*/* public Shape rotateLeft() {  
 if (pieceShape == Tetrominoe.*O*)  
 return this;  
 Shape result = new Shape();  
 result.pieceShape = pieceShape;  
 for (int i = 0; i < 4; i++) {  
 result.setX(i, -y(i));  
 result.setY(i, x(i));  
 }  
 return result;  
 }  
}

# **Board.java**

package game;  
  
import javax.swing.\*;  
import java.awt.\*;  
import java.awt.event.ActionEvent;  
import java.awt.event.ActionListener;  
import java.awt.event.KeyAdapter;  
import java.awt.event.KeyEvent;  
import java.time.Duration;  
import java.time.LocalDateTime;  
  
public class Board extends JPanel implements ActionListener {  
 public static final int *BOARD\_WIDTH* = 10; //Ширина игрового поля  
 public static final int *BOARD\_HEIGHT* = 22; //Высота игрового поля  
 private static final int *INITIAL\_DELAY* = 1000; //Задержка срабатывания таймеров при инициализации, а также коэффициент, участвующий в увеличении начисляемых очков при увеличении скорости игры  
 static boolean *isSolverOn* = false;  
 static Shape *nextPiece*;  
 private static Next *next*;  
 private static boolean *isFallingFinished* = false;  
 public Timer timer; //Таймер, отвечающий за падение фигур на игровом поле  
 private Timer timerStats; //Таймер, отвечающий за отображение времени игры  
 public boolean isStarted = false;  
 public boolean isPaused = false;  
 public int numLinesRemoved = 0;  
 private int curX = 0;  
 private int curY = 0;  
 public int scoreInt = 0;  
 private JLabel score;  
 private JLabel lines;  
 private JLabel time;  
 private LocalDateTime startTime;  
 private Shape curPiece;  
 private Tetrominoe[][] board;  
  
 public Tetrominoe[][] getBoard() {  
 return board;  
 }  
  
 public Shape getCurPiece() {  
 return curPiece;  
 }  
  
 public Shape getNextPiece() {  
 return *nextPiece*;  
 }  
  
 public Board(Tetris parent) {  
 setFocusable(true);  
 setPreferredSize(new Dimension(260, 460));  
 *next* = new Next();  
  
 curPiece = new Shape();  
 *nextPiece* = new Shape();  
 nextPiece();  
  
 score = parent.getScore();  
 lines = parent.getLines();  
 time = parent.getTime();  
  
 timerStats = new Timer(*INITIAL\_DELAY*, e -> {  
 LocalDateTime now = LocalDateTime.*now*();  
 Duration duration = Duration.*between*(startTime, now);  
 time.setText(format(duration));  
 });  
  
 timer = new Timer(*INITIAL\_DELAY*, this);  
 board = new Tetrominoe[*BOARD\_WIDTH*][*BOARD\_HEIGHT*];  
 clearBoard();  
 addKeyListener(new TetrisAdapter());  
 }  
  
 static Next getNext() {  
 return *next*;  
 }  
  
 private String format(Duration duration) {  
 long hours = duration.toHours();  
 long mins = duration.minusHours(hours).toMinutes();  
 long seconds = duration.minusMinutes(mins).toMillis() / *INITIAL\_DELAY*;  
 return String.*format*("%02dh %02dm %02ds", hours, mins, seconds);  
 }  
  
 private int squareWidth() {  
 return (int) getSize().getWidth() / *BOARD\_WIDTH*;  
 }  
  
 private int squareHeight() {  
 return (int) getSize().getHeight() / *BOARD\_HEIGHT*;  
 }  
  
 public Tetrominoe shapeAt(int x, int y) {  
 return board[x][y];  
 }  
  
 private void clearBoard() {  
 for (int x = 0; x < *BOARD\_WIDTH*; x++) {  
 for (int y = 0; y < *BOARD\_HEIGHT*; y++) {  
 board[x][y] = Tetrominoe.*NoShape*;  
 }  
 }  
 }  
  
 private void pieceDropped() {  
 for (int i = 0; i < 4; i++) {  
 int x = curX + curPiece.x(i);  
 int y = curY + curPiece.y(i);  
 board[x][y] = curPiece.getShape();  
 }  
 removeFullLines();  
 if (!*isFallingFinished*) {  
 newPiece();  
 }  
 }  
  
 private void nextPiece() {  
 *nextPiece*.setRandomShape();  
 }  
  
 public void newPiece() {  
 curPiece = *nextPiece*;  
 *nextPiece* = new Shape();  
 nextPiece();  
 curX = *BOARD\_WIDTH* / 2 - 1;  
 curY = *BOARD\_HEIGHT* - 1 + curPiece.minY();  
 if (!tryMove(curPiece, curX, curY - 1)) {  
 curPiece.setShape(Tetrominoe.*NoShape*);  
 *nextPiece*.setShape(Tetrominoe.*NoShape*);  
 timer.stop();  
 timerStats.stop();  
 isStarted = false;  
 }  
 }  
  
 private void oneLineDown() {  
 if (!tryMove(curPiece, curX, curY - 1))  
 pieceDropped();  
 }  
  
 @Override  
 public void actionPerformed(ActionEvent e) {  
 if (*isFallingFinished*) {  
 *isFallingFinished* = false;  
 newPiece();  
 } else {  
 oneLineDown();  
 }  
 }  
  
 private void drawSquare(Graphics g, int x, int y, Tetrominoe shape) {  
 g.setColor(shape.color);  
 g.fillRect(x + 1, y + 1, squareWidth() - 2, squareHeight() - 2);  
 }  
  
 @Override  
 public void paint(Graphics g) {  
 super.paint(g);  
 Dimension size = getSize();  
 if (!isPaused && isStarted && curPiece.getShape() != Tetrominoe.*NoShape*)  
 *next*.repaint();  
 int boardTop = (int) size.getHeight() - *BOARD\_HEIGHT* \* squareHeight();  
 for (int i = 0; i < *BOARD\_HEIGHT*; i++) {  
 for (int j = 0; j < *BOARD\_WIDTH*; ++j) {  
 Tetrominoe shape = shapeAt(j, *BOARD\_HEIGHT* - i - 1);  
 if (shape != Tetrominoe.*NoShape*) {  
 drawSquare(g, j \* squareWidth(), boardTop + i \* squareHeight(), shape);  
 }  
 }  
 }  
  
 if (curPiece.getShape() != Tetrominoe.*NoShape*) {  
 for (int i = 0; i < 4; ++i) {  
 int x = curX + curPiece.x(i);  
 int y = curY + curPiece.y(i);  
 drawSquare(g, x \* squareWidth(), boardTop + (*BOARD\_HEIGHT* - y - 1) \* squareHeight(), curPiece.getShape());  
 }  
 }  
 }  
  
 void start() {  
 if (isPaused)  
 return;  
 isStarted = true;  
 *isFallingFinished* = false;  
 numLinesRemoved = 0;  
 scoreInt = 0;  
 clearBoard();  
 newPiece();  
 timer.start();  
 startTime = LocalDateTime.*now*();  
 timerStats.start();  
 score.setText("");  
 lines.setText("");  
 }  
  
 private void pause() {  
 if (!isStarted)  
 return;  
 isPaused = !isPaused;  
 if (isPaused) {  
 timer.stop();  
 timerStats.stop();  
 } else {  
 timer.start();  
 timerStats.start();  
 score.setText(String.*valueOf*(scoreInt));  
 }  
 repaint();  
 }  
  
 public boolean tryMove(Shape newPiece, int newX, int newY) {  
 for (int i = 0; i < 4; ++i) {  
 int x = newX + newPiece.x(i);  
 int y = newY + newPiece.y(i);  
 if (x < 0 || x >= *BOARD\_WIDTH* || y < 0 || y >= *BOARD\_HEIGHT*)  
 return false;  
 if (shapeAt(x, y) != Tetrominoe.*NoShape*)  
 return false;  
 }  
 curPiece = newPiece;  
 curX = newX;  
 curY = newY;  
 repaint();  
 return true;  
 }  
  
 public void removeFullLines() {  
 int numFullLines = 0;  
 final int LINES\_1 = 100;  
 final int LINES\_2 = 300;  
 final int LINES\_3 = 700;  
 final int LINES\_4 = 1500;  
 for (int i = *BOARD\_HEIGHT* - 1; i >= 0; --i) {  
 boolean lineIsFull = true;  
 for (int j = 0; j < *BOARD\_WIDTH*; ++j) {  
 if (shapeAt(j, i) == Tetrominoe.*NoShape*) {  
 lineIsFull = false;  
 break;  
 }  
 }  
 if (lineIsFull) {  
 numFullLines++;  
 for (int k = i; k < *BOARD\_HEIGHT* - 1; ++k) {  
 for (int j = 0; j < *BOARD\_WIDTH*; ++j) {  
 board[j][k] = shapeAt(j, k + 1);  
 }  
 }  
 }  
 }  
 if (numFullLines > 0) {  
 numLinesRemoved += numFullLines;  
 switch (numFullLines) {  
 case 4:  
 scoreInt += LINES\_4 \* (*INITIAL\_DELAY* / timer.getDelay());  
 break;  
 case 3:  
 scoreInt += LINES\_3 \* (*INITIAL\_DELAY* / timer.getDelay());  
 break;  
 case 2:  
 scoreInt += LINES\_2 \* (*INITIAL\_DELAY* / timer.getDelay());  
 break;  
 case 1:  
 scoreInt += LINES\_1 \* (*INITIAL\_DELAY* / timer.getDelay());  
 break;  
 default:  
 throw new IllegalStateException("Unexpected value: " + numFullLines);  
 }  
 lines.setText(String.*valueOf*(numLinesRemoved));  
 score.setText(String.*valueOf*(scoreInt));  
 *isFallingFinished* = true;  
 curPiece.setShape(Tetrominoe.*NoShape*);  
 repaint();  
 }  
 }  
  
 public void dropDown() {  
 int newY = curY;  
 while (newY > 0) {  
 if (!tryMove(curPiece, curX, newY - 1))  
 break;  
 --newY;  
 }  
 pieceDropped();  
 }  
  
 class TetrisAdapter extends KeyAdapter {  
 @Override  
 public void keyPressed(KeyEvent k) {  
 int keyCode = k.getKeyCode();  
 if ((!isStarted || curPiece.getShape() == Tetrominoe.*NoShape*) && (keyCode == 'r' || keyCode == 'R')) {  
 start();  
 dropDown();  
 }  
 if (keyCode == 'p' || keyCode == 'P') {  
 pause();  
 }  
 if (isPaused || !isStarted || curPiece.getShape() == Tetrominoe.*NoShape* || Board.*isSolverOn*) {  
 return;  
 }  
 if (!(!isStarted || curPiece.getShape() == Tetrominoe.*NoShape*))  
 switch (keyCode) {  
 case KeyEvent.*VK\_LEFT*:  
 tryMove(curPiece, curX - 1, curY);  
 break;  
 case KeyEvent.*VK\_RIGHT*:  
 tryMove(curPiece, curX + 1, curY);  
 break;  
 case KeyEvent.*VK\_DOWN*:  
 oneLineDown();  
 break;  
 case KeyEvent.*VK\_UP*:  
 tryMove(curPiece.rotateLeft(), curX, curY);  
 break;  
 case KeyEvent.*VK\_SPACE*:  
 dropDown();  
 break;  
 }  
 }  
 }  
}

# **Tetris.java**

package game;  
  
import solver.GeneticWeights;  
import solver.Solver;  
  
import javax.swing.\*;  
import java.awt.\*;  
import java.awt.event.ActionListener;  
import java.awt.event.InputEvent;  
import java.awt.event.KeyEvent;  
  
import static solver.Solver.*solver*;  
  
public class Tetris extends JFrame {  
 private JLabel score;  
 private JLabel lines;  
 private JLabel time;  
 private static final int *TIME\_TO\_FALL* = 1200; //Время падения фигуры на игровом поле. Чем меньше значение, тем быстрее идёт игра  
 private static final int *TIME\_DECREASE* = 200; //Величина, на которую уменьшается время падения фигуры при увеличении скорости игры  
  
 public Tetris() {  
 score = new JLabel("0");  
 lines = new JLabel("0");  
 time = new JLabel("0h 00mm 00ss");  
  
 Board board = new Board(this);  
 add(board, BorderLayout.*WEST*);  
 board.start();  
 board.setBorder(BorderFactory.*createLineBorder*(Color.*black*));  
  
 JMenuBar menuBar = new JMenuBar();  
 setJMenuBar(menuBar);  
 JMenu menu = new JMenu("Game");  
 JMenuItem newGame = new JMenuItem("New");  
 JMenu settings = new JMenu("Settings");  
 JMenuItem exit = new JMenuItem("Exit");  
 JCheckBoxMenuItem solver = new JCheckBoxMenuItem("Turn on/off solver");  
 JMenuItem resetGenetic = new JMenuItem("Reset genesis");  
 JMenu speed = new JMenu("Game Speed");  
 JRadioButton speed1 = new JRadioButton("1", true);  
 JRadioButton speed2 = new JRadioButton("2", false);  
 JRadioButton speed3 = new JRadioButton("3", false);  
 JRadioButton speed4 = new JRadioButton("4", false);  
 JRadioButton speed5 = new JRadioButton("5", false);  
 ActionListener speedIncrease = actionEvent -> {  
 AbstractButton aButton = (AbstractButton) actionEvent.getSource();  
 board.timer.setDelay(*TIME\_TO\_FALL* - *TIME\_DECREASE* \* Integer.*parseInt*(aButton.getText()));  
 };  
 speed1.addActionListener(speedIncrease);  
 speed2.addActionListener(speedIncrease);  
 speed3.addActionListener(speedIncrease);  
 speed4.addActionListener(speedIncrease);  
 speed5.addActionListener(speedIncrease);  
 speed.add(speed1);  
 speed.add(speed2);  
 speed.add(speed3);  
 speed.add(speed4);  
 speed.add(speed5);  
 ButtonGroup gameSpeed = new ButtonGroup();  
 gameSpeed.add(speed1);  
 gameSpeed.add(speed2);  
 gameSpeed.add(speed3);  
 gameSpeed.add(speed4);  
 gameSpeed.add(speed5);  
 solver.setAccelerator(KeyStroke.*getKeyStroke*(KeyEvent.*VK\_S*, InputEvent.*SHIFT\_MASK*));  
 settings.add(solver).addActionListener(e -> {  
 Board.*isSolverOn* = !Board.*isSolverOn*;  
 *solver*(board, Board.*isSolverOn*);  
 });  
 resetGenetic.setAccelerator(KeyStroke.*getKeyStroke*(KeyEvent.*VK\_G*, InputEvent.*SHIFT\_MASK*));  
 settings.add(resetGenetic).addActionListener(e -> {  
 Solver.*setPenHeight*(-0.5);  
 Solver.*setPenClear*(1.0);  
 Solver.*setPenHole*(-1.5);  
 Solver.*setPenBump*(-1.0);  
 GeneticWeights.*weights*.clear();  
 GeneticWeights.*createFirstGeneration*();  
 });  
 settings.add(speed);  
 menu.add(newGame).addActionListener(e -> board.start());  
 menu.addSeparator();  
 menu.add(settings);  
 menu.addSeparator();  
 menu.add(exit).addActionListener(e -> System.*exit*(0));  
 JMenu help = new JMenu("Help");  
 JMenuItem about = new JMenuItem("About");  
 JMenuItem control = new JMenuItem("Controls");  
 JMenuItem factors = new JMenuItem("Factors");  
 help.add(about).addActionListener(e -> JOptionPane.*showMessageDialog*(Tetris.this, "Это Тетрис. С решателем. " +  
 "Не ругайте его, он теперь учится.",  
 "About", JOptionPane.*INFORMATION\_MESSAGE*));  
 help.add(control).addActionListener(e -> JOptionPane.*showMessageDialog*(Tetris.this,  
 new String[]{"Вверх - поворот влево", "Влево - переместить влево", "Вправо - переместить вправо",  
 "Вниз - опустить на одну линию", "Пробел - опустить вниз", "P - пауза", "R - рестарт",  
 "Shift + S - включить/выключить решателя", "Shift + G - сброс штрафных параметров"},  
 "Controls",  
 JOptionPane.*WARNING\_MESSAGE*));  
 help.add(factors).addActionListener(e -> JOptionPane.*showMessageDialog*(Tetris.this,  
 new String[]{"\"Штраф\" за высоту " + Solver.*getPenHeight*(),  
 "\"Штраф\" за очищенные линии " + Solver.*getPenClear*(),  
 "\"Штраф\" за наличие дырок " + Solver.*getPenHole*(),  
 "\"Штраф\" за перепад высот " + Solver.*getPenBump*()},  
 "Factors",  
 JOptionPane.*ERROR\_MESSAGE*));  
 menuBar.add(menu);  
 menuBar.add(help);  
 Next next = Board.*getNext*();  
  
 JPanel stats = new JPanel();  
 stats.setLayout(new BoxLayout(stats, BoxLayout.*Y\_AXIS*));  
 stats.add(Box.*createVerticalStrut*(50));  
 stats.add(new JLabel("Score :"));  
 stats.add(Box.*createVerticalStrut*(20));  
 stats.add(score);  
 stats.add(Box.*createVerticalStrut*(30));  
 stats.add(new JLabel("Lines :"));  
 stats.add(Box.*createVerticalStrut*(20));  
 stats.add(lines);  
 stats.add(Box.*createVerticalStrut*(30));  
 stats.add(new JLabel("Time :"));  
 stats.add(Box.*createVerticalStrut*(20));  
 stats.add(time);  
 stats.add(Box.*createVerticalStrut*(80));  
 stats.add(new JLabel("Next :"));  
 stats.add(Box.*createVerticalStrut*(40));  
 stats.add(next);  
 add(stats, BorderLayout.*EAST*);  
  
 setSize(500, 600);  
 setTitle("Tetris solver");  
 setDefaultCloseOperation(WindowConstants.*EXIT\_ON\_CLOSE*);  
 }  
  
 public static void main(String[] args) {  
 Tetris tetris = new Tetris();  
 tetris.setLocationRelativeTo(null);  
 tetris.setVisible(true);  
 tetris.setResizable(false);  
 }  
  
 JLabel getScore() {  
 return score;  
 }  
  
 JLabel getLines() {  
 return lines;  
 }  
  
 JLabel getTime() {  
 return time;  
 }  
}

# **Solver.java**

package solver;  
  
import game.Board;  
import game.Shape;  
import game.Tetrominoe;  
  
import javax.swing.\*;  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.NavigableMap;  
import java.util.TreeMap;  
  
import static game.Board.*BOARD\_HEIGHT*;  
import static game.Board.*BOARD\_WIDTH*;  
  
public class Solver {  
 private static Shape *cur*;  
 private static Shape *next*;  
 private static NavigableMap<Double, ArrayList<Integer>> *grades* = new TreeMap<>(); // Ключ - оценка данного хода, значение - список параметров данного хода (x, y, rotations)  
  
 private static double *PenHeight* = -0.5; //Штрафные (весовые) коэффициенты, используемые  
 private static double *PenClear* = 1.0; //в функции оценки возможного хода. Изначально заданы примерные значения.  
 private static double *PenHole* = -1.5; //Далее значения будут меняться в ходе выполнения метода startGeneration  
 private static double *PenBump* = -1.0; //из класса GeneticWeights.java  
 private static Timer *solve*; //Таймер, необходимый для задержки срабатывания решателя  
  
  
 public static double getPenHeight() {  
 return *PenHeight*;  
 }  
  
 public static double getPenClear() {  
 return *PenClear*;  
 }  
  
 public static double getPenHole() {  
 return *PenHole*;  
 }  
  
 public static double getPenBump() {  
 return *PenBump*;  
 }  
  
 public static void setPenHeight(double penHeight) {  
 *PenHeight* = penHeight;  
 }  
  
 public static void setPenClear(double penClear) {  
 *PenClear* = penClear;  
 }  
  
 public static void setPenHole(double penHole) {  
 *PenHole* = penHole;  
 }  
  
 public static void setPenBump(double penBump) {  
 *PenBump* = penBump;  
 }  
  
 */\*\*  
 \* Главный метод решателя, запускаемый по таймеру, который вызывает основные методы класса (см. ниже)  
 \*/* public static void solver(Board gameBoard, boolean start) {  
 if (*solve* == null) {  
 Thread gen = new generationThread();  
 gen.start();  
 *solve* = new Timer(gameBoard.timer.getDelay(), e -> {  
 *solve*.setDelay(gameBoard.timer.getDelay()); //Задержка срабатывания решателя подстраивается под скорость игры  
 Tetrominoe[][] board = gameBoard.getBoard();  
 *cur* = gameBoard.getCurPiece();  
 *next* = gameBoard.getNextPiece();  
 *gradeCurrent*(board);  
 *makeMove*(gameBoard);  
 *grades*.clear();  
 });  
 }  
 *solve*.start();  
 if (start && !gameBoard.isPaused && gameBoard.isStarted && !(gameBoard.getCurPiece().getShape() == Tetrominoe.*NoShape*)) {  
 *solve*.start();  
 } else {  
 *solve*.stop();  
 }  
 }  
  
 */\*\*  
 \* Метод, который делает окончательный ход фигуры на игровом поле  
 \*/* private static void makeMove(Board gameBoard) {  
 try {  
 ArrayList<Integer> bestMove = *grades*.lastEntry().getValue(); // Список, содержащий параметры данного хода  
 for (int i = 0; i < bestMove.get(2); i++) { // 2 - количество поворотов фигуры  
 *cur* = *cur*.rotateLeft();  
 }  
 gameBoard.tryMove(*cur*, bestMove.get(0), bestMove.get(1)); // 0 - координата x, 1 - координата y  
 gameBoard.dropDown();  
 } catch (NullPointerException ignored) {  
 }  
 }  
  
 */\*\*  
 \* Метод, который проводит оценку данного возможного хода  
 \*/* private static double makeGrade(Tetrominoe[][] tetrominoes, int clearedLines) {  
 int aggregateHeight = 0; //Суммарная высота колонок данного экспериментального поля  
 int holes = 0; //Количество пустых клеток под непустыми в данном экспериментальном поле  
 int bump = 0; //"Перепад" высот между колонками данного экспериментального игрового поля  
 boolean[] heightCount = new boolean[*BOARD\_WIDTH*]; //Показывает, найдена ли высшая непустая клетка для данной колонки  
 int[] heights = new int[*BOARD\_WIDTH*]; //Массив высот  
 for (int y = *BOARD\_HEIGHT* - 1; y >= 0; y--) {  
 for (int x = *BOARD\_WIDTH* - 1; x >= 0; x--) {  
 if (tetrominoes[x][y] != Tetrominoe.*NoShape*) {  
 if (!heightCount[x]) {  
 heightCount[x] = true;  
 aggregateHeight += y;  
 heights[x] = y;  
 }  
 }  
 if (heightCount[x] && tetrominoes[x][y] == Tetrominoe.*NoShape*) {  
 holes++;  
 }  
 }  
 }  
 for (int i = 0; i < *BOARD\_WIDTH* - 1; i++) {  
 bump += Math.*abs*(heights[i] - heights[i + 1]);  
 }  
 return *PenHeight* \* aggregateHeight + *PenClear* \* clearedLines + *PenHole* \* holes + *PenBump* \* bump;  
 }  
  
 */\*\*  
 \* Метод, который рассматривает все возможные ходы следующей фигуры после возможного хода нынешней фигуры  
 \*/* private static void gradeNext(Tetrominoe[][] tetrominoes, double gradeCur, ArrayList<Integer> currentParams) {  
 int[] heights = *gradeHeights*(tetrominoes); //Массив высот данного экспериментального игрового поля  
 for (int i = 0; i < 4; i++) {  
 for (int x = *BOARD\_WIDTH* - 1; x >= 0; x--) {  
 for (int y = *BOARD\_HEIGHT* - 2; y >= heights[x]; y--) {  
 Tetrominoe[][] expNextBoard = new Tetrominoe[*BOARD\_WIDTH*][*BOARD\_HEIGHT*];  
 for (int j = 0; j < tetrominoes.length; j++) {  
 System.*arraycopy*(tetrominoes[j], 0, expNextBoard[j], 0, tetrominoes[j].length);  
 }  
 if (!*canPlace*(expNextBoard, x, y - 1, *next*) && *canPlace*(expNextBoard, x, y, *next*)  
 && *canPlace*(expNextBoard, x, y + 1, *next*)) {  
 *placeOnBoard*(x, y, expNextBoard, *next*);  
 double gradeNext = *makeGrade*(expNextBoard, *countClearedLines*(expNextBoard));  
 *grades*.put(gradeCur + gradeNext, currentParams);  
 }  
 }  
 }  
 *next* = *next*.rotateLeft();  
 }  
 }  
  
 */\*\*  
 \* Метод, который рассматривает все возможные ходы нынешней фигуры  
 \*/* private static void gradeCurrent(Tetrominoe[][] tetrominoes) {  
 int[] heights = *gradeHeights*(tetrominoes); //Массив высот данного экспериментального игрового поля  
 for (int i = 0; i < 4; i++) {  
 for (int x = *BOARD\_WIDTH* - 1; x >= 0; x--) {  
 for (int y = *BOARD\_HEIGHT* - 2; y >= heights[x]; y--) {  
 Tetrominoe[][] expCurBoard = new Tetrominoe[*BOARD\_WIDTH*][*BOARD\_HEIGHT*];  
 for (int j = 0; j < tetrominoes.length; j++) {  
 System.*arraycopy*(tetrominoes[j], 0, expCurBoard[j], 0, tetrominoes[j].length);  
 }  
 if (!*canPlace*(expCurBoard, x, y - 1, *cur*) && *canPlace*(expCurBoard, x, y, *cur*)  
 && *canPlace*(expCurBoard, x, y + 1, *cur*)) {  
 *placeOnBoard*(x, y, expCurBoard, *cur*);  
 int clearedLines = *countClearedLines*(expCurBoard);  
 *gradeNext*(expCurBoard, *makeGrade*(expCurBoard, clearedLines),  
 new ArrayList<>(Arrays.*asList*(x, y, i)));  
 }  
 }  
 }  
 *cur* = *cur*.rotateLeft();  
 }  
 }  
  
 */\*\*  
 \* Метод, который рассчитывает высоту каждой колонки экспериментального игрового поля на момент вызова и позволяет уменьшить  
 \* количество переборов для размещения фигур на экспериментальном игровом поле  
 \*/* public static int[] gradeHeights(Tetrominoe[][] tetrominoes) {  
 boolean[] heightCount = new boolean[*BOARD\_WIDTH*];  
 int[] heights = new int[*BOARD\_WIDTH*];  
 for (int j = *BOARD\_HEIGHT* - 1; j >= 0; j--) {  
 for (int k = *BOARD\_WIDTH* - 1; k >= 0; k--) {  
 if (tetrominoes[k][j] != Tetrominoe.*NoShape*) {  
 if (!heightCount[k]) {  
 heightCount[k] = true;  
 heights[k] = j;  
 }  
 }  
 }  
 }  
 return heights;  
 }  
  
 */\*\*  
 \* Метод, который размещает данную фигуру с данными координатами на экспериментальное игровое поле  
 \*/* public static void placeOnBoard(int x, int y, Tetrominoe[][] tetrominoes, Shape cur) {  
 for (int i = 0; i < 4; ++i) {  
 int xx = x + cur.x(i);  
 int yy = y + cur.y(i);  
 tetrominoes[xx][yy] = cur.getShape();  
 }  
 }  
  
 */\*\*  
 \* Метод подсчёта очищенных линий на экспериментальном игровом поле в результате некоторого размещения фигуры  
 \*/* public static int countClearedLines(Tetrominoe[][] tetrominoes) {  
 int clearedLines = 0;  
 for (int f = *BOARD\_HEIGHT* - 1; f >= 0; --f) {  
 boolean lineIsFull = true;  
 for (int j = 0; j < *BOARD\_WIDTH*; ++j) {  
 if (tetrominoes[j][f] == Tetrominoe.*NoShape*) {  
 lineIsFull = false;  
 break;  
 }  
 }  
 if (lineIsFull) {  
 clearedLines++;  
 for (int k = f; k < *BOARD\_HEIGHT* - 1; ++k) {  
 for (int j = 0; j < *BOARD\_WIDTH*; ++j) {  
 tetrominoes[j][k] = tetrominoes[j][k + 1];  
 }  
 }  
 }  
 }  
 return clearedLines;  
 }  
  
 */\*\*  
 \* Метод, проверяющий возможность размещения данной фигуры на данных координатах на данном экспериментальном игровом поле  
 \*/* public static boolean canPlace(Tetrominoe[][] tetrominoes, int newX, int newY, Shape shape) {  
 for (int i = 0; i < 4; i++) {  
 int x = newX + shape.x(i);  
 int y = newY + shape.y(i);  
 if (x < 0 || x >= *BOARD\_WIDTH* || y < 0 || y >= *BOARD\_HEIGHT* - 2)  
 return false;  
 if (tetrominoes[x][y] != Tetrominoe.*NoShape*)  
 return false;  
 }  
 return true;  
 }  
  
 private static class generationThread extends Thread {  
 @Override  
 public void run() {  
 GeneticWeights.*startGeneration*();  
 }  
 }  
}

# **GeneticWeights.java**

package solver;  
  
  
import game.Shape;  
import game.Tetrominoe;  
import javafx.util.Pair;  
  
import java.util.\*;  
  
import static game.Board.*BOARD\_HEIGHT*;  
import static game.Board.*BOARD\_WIDTH*;  
import static solver.Solver.\*;  
  
public class GeneticWeights {  
 public static List<double[]> *weights*; //Список геномов  
 private static List<Pair<Integer, double[]>> *gradeWeights* = new ArrayList<>(); //Список пар геномов и их оценки в результате соревнования  
 private static Shape *cur* = new Shape(); //Некоторые переменные и методы позаимствованы из класса решателя  
 private static Shape *next* = new Shape();  
 private static NavigableMap<Double, ArrayList<Integer>> *grades* = new TreeMap<>();  
 private static List<Pair<Double, ArrayList<double[]>>> *genomsForCrossover* = new ArrayList<>(); //Список пар геномов, которым предстоит скрещивание и отношение их оценок в результате соревнования  
 private static final int *GENOMES\_IN\_GENERATION* = 10; //Количество геномов в поколении  
 private static final int *TETROMINOES* = 1000; //Количество фигур за одно соревнование для одного генома  
 private static final int *GENOMES\_NS* = 300; //Количество геномов, заменяемых в результате естественного отбора (30% от GENOMES\_IN\_GENERATION)  
 private static Tetrominoe[] *gameSet* = new Tetrominoe[*TETROMINOES*]; //Массив фигур, которые будут во время соревнования у геномов  
  
 */\*\*  
 \* Основной метод генетического алгоритма, в котором происходит смена поколений и в результате в конфигурационный  
 \* файл выводятся коэффициенты для решателя.  
 \*/* static void startGeneration() {  
 *createFirstGeneration*();  
 while (true) { //Запускаем цикл для постепенного улучшения изначальных коэффициентов  
 *gradeWeights*.clear();  
 *playGame*();  
 *selection*();  
 *createNewGeneration*();  
  
 double[] bestWeights = *gradeWeights*.get(*gradeWeights*.size() - 1).getValue(); //Обновляем весовые коэффициенты  
 *setPenHeight*(bestWeights[0]);  
 *setPenClear*(bestWeights[1]);  
 *setPenHole*(bestWeights[2]);  
 *setPenBump*(bestWeights[3]);  
 }  
 }  
  
 */\*\*  
 \* Метод создания первого поколения  
 \*/* public static void createFirstGeneration() {  
 *weights* = new ArrayList<>();  
 for (int i = 0; i < *GENOMES\_IN\_GENERATION*; i++) {  
 double[] penalties = new double[4];  
 for (int j = 0; j < 4; j++) {  
 penalties[j] = 10 \* Math.*random*() - 5; //Создание случайных чисел в диапазоне от -5.0 до 5.0  
 }  
 *weights*.add(penalties);  
 }  
 }  
  
 */\*\*  
 \* Метод, в котором проходит соревнование  
 \*/* private static void playGame() {  
 Random generate = new Random();  
 for (int j = 0; j < *TETROMINOES*; j++) {  
 *gameSet*[j] = Tetrominoe.*values*()[generate.nextInt(Tetrominoe.*values*().length)]; //Создание массива случайных фигур заданного размера, одинакового для всех участников данного соревнования  
 }  
 for (int i = 0; i < *GENOMES\_IN\_GENERATION*; i++) { //Проведение одинаковой игры для каждого генома  
 Tetrominoe[][] board = new Tetrominoe[*BOARD\_WIDTH*][*BOARD\_HEIGHT*];  
 for (int j = 0; j < *BOARD\_WIDTH*; j++) {  
 for (int k = 0; k < *BOARD\_HEIGHT*; k++) {  
 board[j][k] = Tetrominoe.*NoShape*;  
 }  
 }  
 int cleared = 0; //Количество очищенных линий - показатель эффективности данного генома  
 for (int k = 0; k < *TETROMINOES* - 1; k++) {  
 *cur*.setShape(*gameSet*[k]);  
 *next*.setShape(*gameSet*[k + 1]);  
 *gradeCurrent*(board, *weights*.get(i));  
 cleared += *makeMove*(board); //Метод возвращает количество очищенных данным ходом линий  
 *grades*.clear();  
 }  
 Pair<Integer, double[]> weightGraded = new Pair<>(cleared, *weights*.get(i));  
 *gradeWeights*.add(weightGraded);  
 }  
 }  
  
 */\*\*  
 \* Метод, который отбирает два случайных генома для скрещивания  
 \*/* private static void selection() {  
 Random chooser = new Random();  
 for (int i = 0; i < *GENOMES\_NS*; i++) { //Выбор нужного количества случайных пар, потомки которых заменят худшие геномы  
 Pair rand1 = *gradeWeights*.get(chooser.nextInt(*gradeWeights*.size()));  
 int par1Grade = (int) rand1.getKey();  
 double[] par1 = (double[]) rand1.getValue();  
 Pair rand2 = *gradeWeights*.get(chooser.nextInt(*gradeWeights*.size()));  
 while (rand1 == rand2) {  
 rand2 = *gradeWeights*.get(chooser.nextInt(*gradeWeights*.size()));  
 }  
 int par2Grade = (int) rand2.getKey();  
 double[] par2 = (double[]) rand2.getValue();  
 Double proportion = par2Grade != 0 ? ((double) par1Grade) / par2Grade : 1.0; //Отношение оценок эффективности родителей  
 ArrayList<double[]> parents = new ArrayList<>(Arrays.*asList*(par1, par2)); //Список родителей  
 *genomsForCrossover*.add(new Pair(proportion, parents)); //Добавление пары случайных геномов, которые будут участвовать в скрещивании  
 }  
 }  
  
 */\*\*  
 \* Метод, который возвращает новый геном на основе геномов родителей и отношения их эффективности  
 \*/* private static double[] crossover(Pair<Double, ArrayList<double[]>> parents) {  
 double[] child = new double[4];  
 for (int i = 0; i < 4; i++) {  
 child[i] = parents.getValue().get(0)[i] \* parents.getKey() + parents.getValue().get(1)[i]; //Линейная комбинация родительских параметров  
 }  
 *mutation*(child); //Мутация ребёнка  
 return child;  
 }  
  
 */\*\*  
 \* Метод, добавляющий мутацию в новый геном (5% шанс на изменение случайной характеристики на 0.3)  
 \*/* private static void mutation(double[] child) {  
 double chance = Math.*random*();  
 if (chance < 0.05) {  
 Random mutate = new Random();  
 child[mutate.nextInt(child.length)] += chance < 0.025 ? 0.3 : -0.3;  
 }  
 }  
  
 */\*\*  
 \* Метод создания нового поколения, удаляющий наименее приспособленные геномы и добавляющий новые геномы  
 \*/* private static void createNewGeneration() {  
 *gradeWeights*.sort(Comparator.*comparing*(Pair::getKey));  
 for (int i = 0; i < *GENOMES\_NS*; i++) {  
 *weights*.remove(*gradeWeights*.get(0).getValue()); //Удаление генома с худшим результатом  
 *weights*.add(*crossover*(*genomsForCrossover*.get(0))); //Добавление нового ребёнка  
 *genomsForCrossover*.remove(0); //Удаление пары геномов из списка для скрещивания  
 }  
 }  
  
 */\*\*  
 \* Здесь и далее приведены методы из класса решателя с незначительными изменениями (в некоторых методах передаются  
 \* штрафные (весовые) коэффициенты из геномов, проходящих соревнование в данный момент)  
 \*/* private static int makeMove(Tetrominoe[][] board) {  
 try {  
 ArrayList<Integer> bestMove = *grades*.lastEntry().getValue();  
 for (int i = 0; i < bestMove.get(2); i++) {  
 *cur* = *cur*.rotateLeft();  
 }  
 for (int i = 0; i < 4; ++i) {  
 int xx = bestMove.get(0) + *cur*.x(i);  
 int yy = bestMove.get(1) + *cur*.y(i);  
 board[xx][yy] = *cur*.getShape();  
 }  
 return bestMove.get(3);  
 } catch (NullPointerException ignored) {  
 }  
 return 0;  
 }  
  
 private static double makeGrade(Tetrominoe[][] tetrominoes, int clearedLines, double[] penalties) {  
 int aggregateHeight = 0;  
 int holes = 0;  
 int bump = 0;  
 boolean[] heightCount = new boolean[*BOARD\_WIDTH*];  
 int[] heights = new int[*BOARD\_WIDTH*];  
 for (int y = *BOARD\_HEIGHT* - 1; y >= 0; y--) {  
 for (int x = *BOARD\_WIDTH* - 1; x >= 0; x--) {  
 if (tetrominoes[x][y] != Tetrominoe.*NoShape*) {  
 if (!heightCount[x]) {  
 heightCount[x] = true;  
 aggregateHeight += y;  
 heights[x] = y;  
 }  
 }  
 if (heightCount[x] && tetrominoes[x][y] == Tetrominoe.*NoShape*) {  
 holes++;  
 }  
 }  
 }  
 for (int i = 0; i < *BOARD\_WIDTH* - 1; i++) {  
 bump += Math.*abs*(heights[i] - heights[i + 1]);  
 }  
 return penalties[0] \* aggregateHeight + penalties[1] \* clearedLines + penalties[2] \* holes + penalties[3] \* bump;  
 }  
  
 private static void gradeNext(Tetrominoe[][] tetrominoes, double gradeCur, ArrayList<Integer> currentParams, double[] penalties) {  
 int[] heights = *gradeHeights*(tetrominoes);  
 for (int i = 0; i < 4; i++) {  
 for (int x = *BOARD\_WIDTH* - 1; x >= 0; x--) {  
 for (int y = *BOARD\_HEIGHT* - 2; y >= heights[x]; y--) {  
 Tetrominoe[][] expNextBoard = new Tetrominoe[*BOARD\_WIDTH*][*BOARD\_HEIGHT*];  
 for (int j = 0; j < tetrominoes.length; j++) {  
 System.*arraycopy*(tetrominoes[j], 0, expNextBoard[j], 0, tetrominoes[j].length);  
 }  
 if (!*canPlace*(expNextBoard, x, y - 1, *next*) && *canPlace*(expNextBoard, x, y, *next*)  
 && *canPlace*(expNextBoard, x, y + 1, *next*)) {  
 *placeOnBoard*(x, y, expNextBoard, *next*);  
 double gradeNext = *makeGrade*(expNextBoard, *countClearedLines*(expNextBoard), penalties);  
 *grades*.put(gradeCur + gradeNext, currentParams);  
 }  
 }  
 }  
 *next* = *next*.rotateLeft();  
 }  
 }  
  
 private static void gradeCurrent(Tetrominoe[][] tetrominoes, double[] penalties) {  
 int[] heights = *gradeHeights*(tetrominoes);  
 for (int i = 0; i < 4; i++) {  
 for (int x = *BOARD\_WIDTH* - 1; x >= 0; x--) {  
 for (int y = *BOARD\_HEIGHT* - 2; y >= heights[x]; y--) {  
 Tetrominoe[][] expCurBoard = new Tetrominoe[*BOARD\_WIDTH*][*BOARD\_HEIGHT*];  
 for (int j = 0; j < tetrominoes.length; j++) {  
 System.*arraycopy*(tetrominoes[j], 0, expCurBoard[j], 0, tetrominoes[j].length);  
 }  
 if (!*canPlace*(expCurBoard, x, y - 1, *cur*) && *canPlace*(expCurBoard, x, y, *cur*)  
 && *canPlace*(expCurBoard, x, y + 1, *cur*)) {  
 *placeOnBoard*(x, y, expCurBoard, *cur*);  
 int clearedLines = *countClearedLines*(expCurBoard);  
 *gradeNext*(expCurBoard, *makeGrade*(expCurBoard, clearedLines, penalties),  
 new ArrayList<>(Arrays.*asList*(x, y, i, clearedLines)), penalties);  
 }  
 }  
 }  
 *cur* = *cur*.rotateLeft();  
 }  
 }  
}

# **Тесты**

В классе TetrisTest.java тестируются методы из классов Board и Solver, отвечающие за логику игрового процесса (расположение фигур, очищение линий).

import game.Board;  
import game.Shape;  
import game.Tetris;  
import game.Tetrominoe;  
import org.junit.jupiter.api.Test;  
import solver.Solver;  
  
import static game.Board.*BOARD\_HEIGHT*;  
import static game.Board.*BOARD\_WIDTH*;  
import static org.junit.jupiter.api.Assertions.\*;  
  
class TestTetris {  
  
 private Tetris testTetris = new Tetris();  
 private Board testBoard = new Board(testTetris);  
  
 //Проверка определения фигуры на игровом поле по координате  
 @Test  
 void TestShapeAt() {  
 for (int i = 0; i < *BOARD\_WIDTH*; i++) {  
 for (int j = 0; j < *BOARD\_HEIGHT*; j++) {  
 *assertEquals*(Tetrominoe.*NoShape*, testBoard.shapeAt(i, j));  
 }  
 }  
 }  
  
 //Проверка соответствия следующей фигуры  
 @Test  
 void newPieceTest() {  
 for (int i = 0; i < 150; i++) {  
 Shape next = testBoard.getNextPiece();  
 testBoard.newPiece();  
 *assertEquals*(next, testBoard.getCurPiece());  
 }  
 }  
  
 //Проверка возможности перемещения фигуры на игровом поле  
 @Test  
 void testTryMove() {  
 Shape shape = new Shape();  
 shape.setShape(Tetrominoe.*I*);  
 for (int i = 0; i < *BOARD\_WIDTH*; i++) {  
 for (int j = 1; j < *BOARD\_HEIGHT* - 2; j++) { //Значения связаны с координатой центра фигуры  
 *assertTrue*(testBoard.tryMove(shape, i, j)); //На пустом игровом поле можно разместить везде, где возможно размещение центра фигуры  
 }  
 }  
 *assertFalse*(testBoard.tryMove(shape, 0, 0)); //Не можем разместить центр фигуры в левом нижнем углу игрового поля  
 }  
  
 //Проверка очистки заполненных линий  
 @Test  
 void testRemoveFullLines() {  
 Tetrominoe[][] board = testBoard.getBoard();  
 for (int i = 0; i < *BOARD\_WIDTH*; i++) {  
 for (int j = 0; j < 3; j++) {  
 board[i][j] = Tetrominoe.*O*;  
 }  
 }  
 testBoard.removeFullLines();  
 *assertEquals*(3, testBoard.numLinesRemoved);  
 *assertEquals*(700, testBoard.scoreInt);  
 for (int i = 0; i < *BOARD\_WIDTH*; i++) {  
 for (int j = 0; j < *BOARD\_HEIGHT*; j++) {  
 board[i][j] = Tetrominoe.*O*;  
 }  
 }  
 *assertThrows*(IllegalStateException.class, () -> testBoard.removeFullLines()); //Не может быть очищено больше 4 линий одновременно  
 }  
  
 //Проверка возможности размещения фигуры на экспериментальном поле  
 @Test  
 void testCanPlace() {  
 Tetrominoe[][] board = testBoard.getBoard();  
 for (int i = 0; i < *BOARD\_WIDTH*; i++) {  
 for (int j = 0; j < *BOARD\_HEIGHT*; j++) {  
 board[i][j] = Tetrominoe.*O*; //Заполняем всё игровое поле  
 }  
 }  
 Shape = new Shape();  
 shape.setShape(Tetrominoe.*I*);  
 for (int i = 0; i < *BOARD\_WIDTH*; i++) {  
 for (int j = 1; j < *BOARD\_HEIGHT* - 2; j++) {  
 *assertFalse*(Solver.*canPlace*(board, i, j, shape)); //Проверяем невозможность размещения фигуры на поле  
 }  
 }  
 }  
  
 //Проверка числа очищенных линий  
 @Test  
 void testCountClearedLines() {  
 Tetrominoe[][] board = testBoard.getBoard();  
 *assertEquals*(0, Solver.*countClearedLines*(board));  
 for (int i = 0; i < *BOARD\_WIDTH*; i++) {  
 for (int j = 0; j < *BOARD\_HEIGHT*; j++) {  
 board[i][j] = Tetrominoe.*O*; //Заполняем экспериментальное поле  
 }  
 }  
 *assertEquals*(22, Solver.*countClearedLines*(board)); //Проверяем, что очистилось всё поле  
 }  
  
 //Проверка размещения фигуры на экспериментальном поле  
 @Test  
 void testPlaceOnBoard() {  
 Tetrominoe[][] board = testBoard.getBoard();  
 Shape shape = new Shape();  
 shape.setShape(Tetrominoe.*O*);  
 Solver.*placeOnBoard*(0, 0, board, shape); //Размещаем фигуру на экспериментальном поле  
 for (int i = 0; i < 1; i++) {  
 for (int j = 0; j < 1; j++) {  
 *assertEquals*(Tetrominoe.*O*, board[i][j]); //Проверяем нахождение размещённой фигуры на экспериментальном поле  
 }  
 }  
 }  
  
 //Проверка подсчёта высоты колонок экспериментального поля  
 @Test  
 void testGradeHeights() {  
 Tetrominoe[][] board = testBoard.getBoard();  
 for (int i = 0; i < *BOARD\_WIDTH*; i++) {  
 board[i][i + 5] = Tetrominoe.*T*; //Размещаем фигуру на поле  
 }  
 int[] heights = Solver.*gradeHeights*(board); //Подсчитываем высоты колонок  
 for (int i = 0; i < *BOARD\_WIDTH*; i++) {  
 *assertEquals*(i + 5, heights[i]); //Проверяем на равенство высот  
 }  
 }  
}

# **Скриншоты программы**

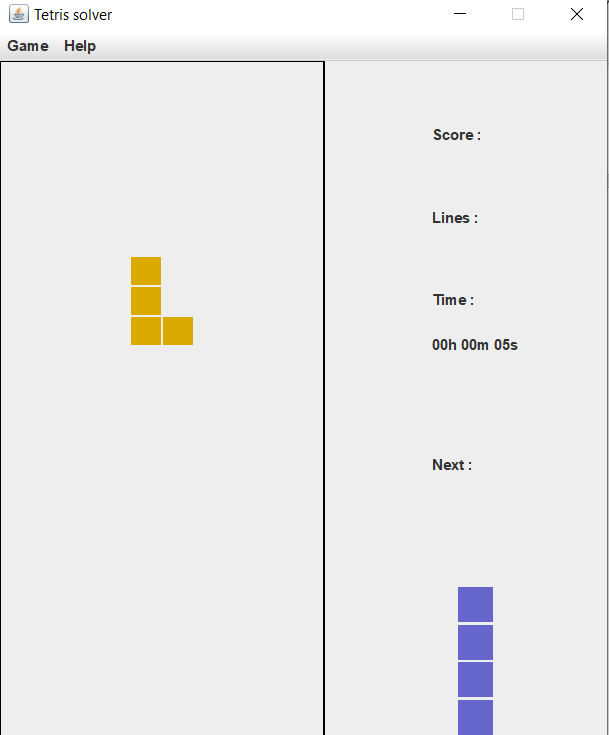


Рис. 2. Приложение при запуске



Рис. 3. Выпадающее меню Help

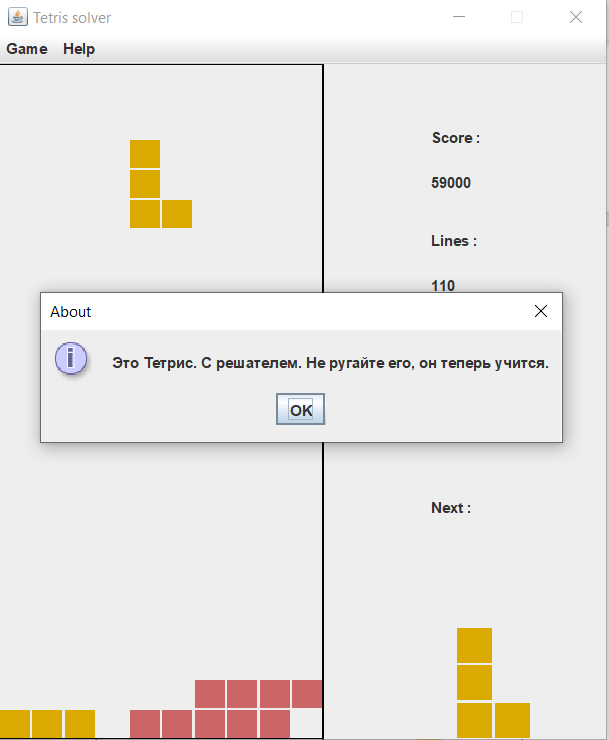


Рис. 4. Диалоговое окно About

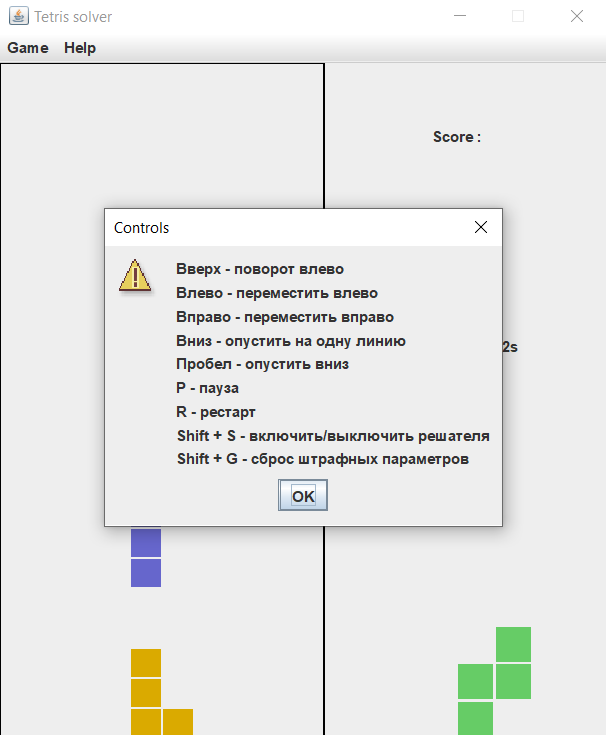


Рис. 5. Диалоговое окно Controls

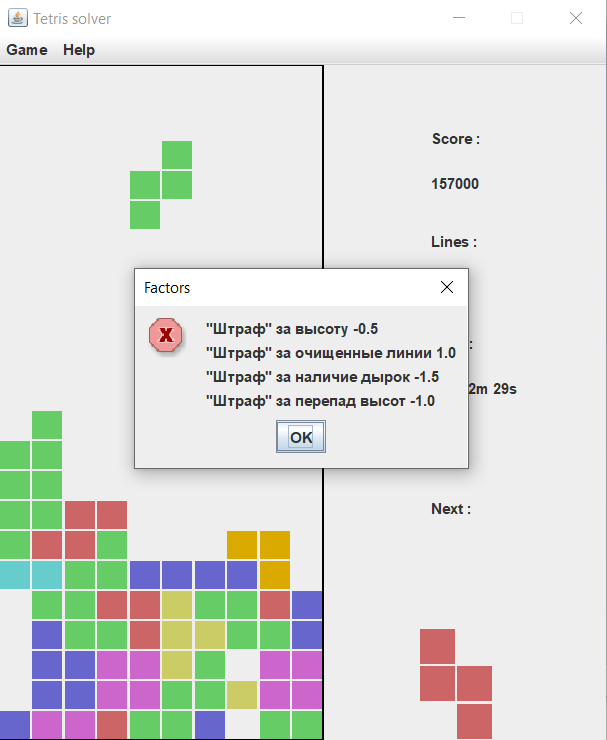


Рис. 6. Диалоговое окно Factors при запуске игры или сбросе коэффициентов

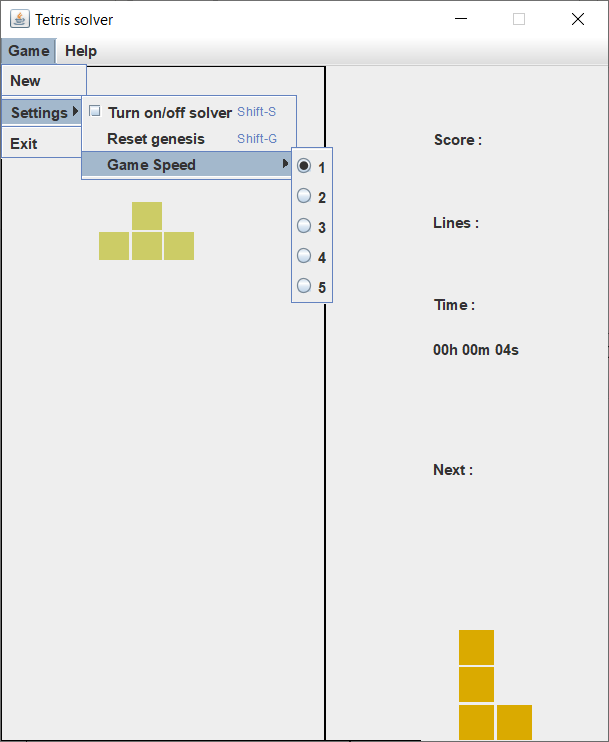


Рис. 7. Меню Settings

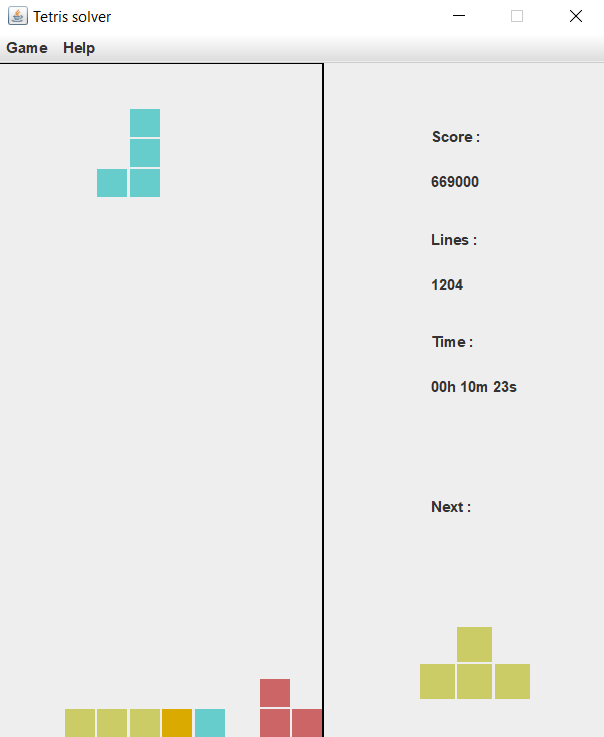


Рис. 8. Решатель после некоторого времени работы

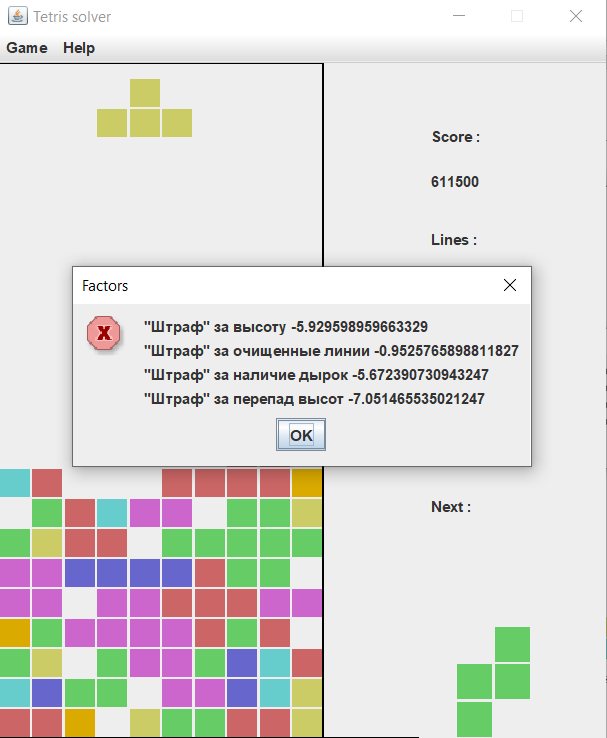


Рис. 9. Диалоговое окон Factors после некоторого времени работы решателя, показывающее, что коэффициенты изменились в процессе работы генетического алгоритма