МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
(МИНОБРНАУКИ)

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ «САНКТ-ПЕТЕРБУРГСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ
И ОПТИКИ»
(СПбГУ ИТМО)

УТВЕРЖДАЮ
Ректор СПбГУ ИТМО,
докт. техн. наук, профессор
В. Н. Васильев

____ ___________ 200 г.

# ПРОГРАММНОЕ СРЕДСТВО 3GENETIC

# ПРОГРАММНЫЙ МОДУЛЬ PLATE
# ТЕКСТ ПРОГРАММЫ

## ЛИСТ УТВЕРЖДЕНИЯ

7.190.00001-01 12 06-ЛУ

Декан факультета «Информационные
технологии и программирование»
докт. техн. наук, профессор
_____________ В. Г. Парфенов

Руководитель темы
заведующий кафедрой «Технологии программирования»,
докт. техн. наук, профессор
______________ А. А. Шалыто

| | |
|---|---|
| Подп. и дата | |
| Инв.N дубл. | |
| Взам.инв.N | |
| Подп. и дата | |
| Инв.N подл. | |

2008

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
(МИНОБРНАУКИ)

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ «САНКТ-ПЕТЕРБУРГСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ
И ОПТИКИ»
(СПбГУ ИТМО)

УТВЕРЖДЕНО
7.190.00001-01 12 06-ЛУ

# ПРОГРАММНОЕ СРЕДСТВО 3GENETIC

# ПРОГРАММНЫЙ МОДУЛЬ PLATE
# ТЕКСТ ПРОГРАММЫ

7.190.00001-01 12 06

Листов 33

| | |
|---|---|
| Подп. и дата | |
| Инв.N дубл. | |
| Взам.инв.N | |
| Подп. и дата | |
| Инв.N подп. | |

2008

# АННОТАЦИЯ

В данном документе приводится текст модуля plate программного средства 3GENETIC, содержащего реализацию особей генетического программирования для задачи построения управляющего автомата для беспилотных летательных аппаратов.

## СОДЕРЖАНИЕ

# 1. МОДУЛЬ PLATES

Модуль plate является подключаемым к ядру программного средства 3GENETIC и реализует генетического программирования для задачи построения управляющего автомата для беспилотных летательных аппаратов. Исходный текст модуля хранится в 15-ти файлах:

1. plates/Competition.java – реализация проведения соревнования между командами беспилотных летательных аппаратов;
2. plates/Config.java – класс-контейнер конфиругационных данных;
3. plates/GameLogic.java – вспомогательный класс для Competition;
4. plates/Plate.java – модель беспилотного летательного аппарата;
5. plates/automaton/Automaton.java – расширение интерфейса Individual (модуль common);
6. plates/automaton/Fitness.java – реализация подсчета функции приспособленности особи;
7. plates/automaton/TableAutomaton.java – реализация особи геентического программирования в виде сокращенных таблиц переходов;
8. plates/automaton/TableAutomatonFactory.java – реализация генератора случайных особей в виде сокращенных таблиц переходов;
9. plates/automaton/TableAutomatonFactoryLoader.java – реализация загрузчика TableAutomatonFactory;
10. plates/automaton/TreeAutomatonFactoryLoader.java – реализация загрузчика TreeAutomatonFactory;
11. plates/automaton/TreeAutomaton.java – реализация особи геентического программирования в виде деревьев решений;
12. plates/automaton/TreeAutomatonFactory.java – реализация генератора случайных особей в виде деревьев решений;
13. plates/managers/AggressiveManager.java – реализация агрессивной стратегии управления беспилотным летательным аппаратом;
14. plates/managers/AutomatonManager.java – реализация стратегии управления беспилотным летательным аппаратом на основе автомата;
15. plates/managers/Manager.java – интерфейс управления беспилотным летательным аппаратом;

## 1.1. Пакет Plates

### 1.1.1. plates/Competition.java

```
package plates;

import plates.managers.Manager;
import plates.utils.Vector;

import java.util.List;
import java.util.ArrayList;
import java.util.Random;

public class Competition {

   private double[] result = {Double.NEGATIVE_INFINITY,
         Double.NEGATIVE_INFINITY};

   private final Manager[] manager;
   private final GameLogic logic;

   private static double[] randomXCoordinates = genRandomCoordinates(Config.getPlatesCount());

   private static double[] genRandomCoordinates(int n) {
       Random random = new Random();
       randomXCoordinates = new double[n];
       for (int i = 0; i < n; i++) {
         randomXCoordinates[i] = 5 + random.nextDouble() * 15;
       }
       return randomXCoordinates;
   }

   public double getResult(int i) {
       if (result[i] == Double.NEGATIVE_INFINITY) {
         emulate();
       }
       return result[i] == Double.NEGATIVE_INFINITY ? 0 : result[i];
   }

   private void emulate() {
       while (true) {
         for (int i = 0; i < 2; i++) {
             manager[i].doTurn();
         }
         logic.processSlowPlates();
         logic.calculateNewSpeeds();
         logic.processSlowPlates();
         logic.movePlates();
         logic.processFlyingOutPlates();
         if (logic.gameOver()) {
             for (int i = 0; i < 2; i++) {
                 List<Plate> plates = manager[i].getPlates();
                 for (Plate plate : plates) {
                     if (!plate.isCrashed()) {
                         result[i] = Math.max(result[i],
                                 plate.getPosition().x);
                     }
                 }
             }
             break;
         }
       }
   }

   public Competition(Manager man1, Manager man2) {
       manager = new Manager[]{man1, man2};
       List<Plate> plates = new ArrayList<Plate>();
       int n = Config.getPlatesCount();
       List<Plate>[] pl = new List[2];
       for (int i = 0; i < 2; i++) {
         pl[i] = new ArrayList<Plate>(n);
        }

        for (int i = 0; i < n; i++) {
            pl[0].add(new Plate(new Vector(randomXCoordinates[i],
                    Config.getFieldHeight() * i / (2 * n - 1)),
                    new Vector(Config.getInitSpeed(), 0), Config.getInitFuel()));
        }
        for (int i = 0; i < n; i++) {
            pl[1].add(new Plate(new Vector(randomXCoordinates[n - i - 1],
                    Config.getFieldHeight() * (n + i) / (2 * n - 1)),
                    new Vector(Config.getInitSpeed(), 0), Config.getInitFuel()));
        }
        for (int i = 0; i < 2; i++) {
            plates.addAll(pl[i]);
        }
        for (int i = 0; i < 2; i++) {
            manager[i].init(pl[i], plates);
        }
        logic = new GameLogic(plates);
    }
}
```

### 1.1.2. plates/Config.java

```
package plates;

public class Config {

    public static double getInfluenceDistance() {
        return 7.0;
    }

    public static double getTimeStep() {
        return 0.3;
    }

    public static double getConstantT() {
        return 3.125;
    }

    public static double getConstantF1() {
        return 0.625;
    }

    public static double getConstantF2() {
        return 0.025;
    }

    public static int getFieldHeight() {
        return 40;
    }

    public static int getFieldWidth() {
        return 1000;
    }

    public static int getPlateDiameter() {
        return 1;
    }

    public static int getPlatesCount() {
        return 8;
    }

    public static int getMaximalRotateAngle() {
        return 25;
    }

    public static double getInitFuel() {
        return 15.0;
    }

    public static double getInitSpeed() {
        return 4.0;
    }

}
```

### 1.1.3. plates/GameLogic.java

```
package plates;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

import plates.utils.Vector;

public class GameLogic {

    private static final double EPS = 1e-9;

    private final List<Plate> plates;

    public GameLogic(List<Plate> plates) {
        this.plates = plates;
    }

    public void processSlowPlates() {
        for (Plate plate : plates) {
            if (plate.isFlying() && plate.getSpeed().getLength() < 1) {
                if (plate.getFuel() <= 0.001) {
                    plate.land();
                } else {
                    plate.crash();
                }
```

```
      }
      if ((plate.isFlying()) && (plate.getFuel() <= 0.001)) {
         plate.land();
      }
   }
}

public void calculateNewSpeeds() {
   HashMap<Plate, Double> koeff = new HashMap<Plate, Double>();
   for (Plate plate : plates) {
      if (!plate.isFlying()) continue;
      double koef = 1;
      for (Plate plate2 : plates) {
         if (!plate2.isFlying())
            continue;
         if (plate == plate2)
            continue;
         Vector speed = plate2.getSpeed();
         Vector delta = plate2.getPosition().subtract(plate.getPosition());
         if (delta.getLength() > Config.getInfluenceDistance())
            continue;
         double ca = speed.multiply(delta) / (speed.getLength() * delta.getLength());
         if (ca >= Math.cos(20 * Math.PI / 180)) {
            koef += 0.5;
         } else if (ca >= Math.cos(40 * Math.PI / 180)) {
            koef -= 0.5;
         }
      }
      if (koef < 0)
         koef = 0;
      koeff.put(plate, koef);
   }
   for (Plate plate : plates) {
      if (!plate.isFlying()) {
         continue;
      }
      Vector speed = plate.getSpeed().rotate(plate.getA() * Math.PI / 180);
      plate.decFuel(plate.getQ() * Config.getTimeStep());
      double T = Config.getConstantT() * plate.getQ();
      double F = Config.getConstantF1() + Config.getConstantF2() * speed.getLength() *
                speed.getLength();
      double inc = Config.getTimeStep() * (T - F * koeff.get(plate)) / speed.getLength();
      plate.setSpeed(speed.multiply(1 + inc));
   }
}

public void movePlates() {
   ArrayList<Plate> cr = new ArrayList<Plate>();
   double t = Config.getTimeStep();
   while (t > 0) {
      double min = Double.POSITIVE_INFINITY;
      boolean found = false;
```

```
for (int i = 0; i < plates.size(); i++) {
   Plate plate1 = plates.get(i);
   if (!plate1.isFlying())
      continue;
   for (int j = 0; j < plates.size(); j++) {
      Plate plate2 = plates.get(j);
      if (!plate2.isFlying())
         continue;
      if (j <= i) {
         continue;
      }

      Vector v1 = plate1.getSpeed();
      Vector pos1 = plate1.getPosition();
      Vector v2 = plate2.getSpeed();
      Vector pos2 = plate2.getPosition();

      Vector deltaV = v1.subtract(v2);
      Vector deltaPos = pos1.subtract(pos2);

      double a = deltaV.getLength() * deltaV.getLength();
      double b = 2 * deltaV.multiply(deltaPos);
      double c = deltaPos.getLength() * deltaPos.getLength() - Config.getPlateDiameter() *
       Config.getPlateDiameter();
      double d = b * b - 4 * a * c;

      if (d < 0)
         continue;
      if (Math.abs(a) <= EPS)
         continue;

      double t1 = (-b + Math.sqrt(d)) / (2 * a);
      double t2 = (-b - Math.sqrt(d)) / (2 * a);
      if (t1 < EPS)
         t1 = Double.POSITIVE_INFINITY;
      if (t2 < EPS)
         t2 = Double.POSITIVE_INFINITY;
      t1 = Math.min(t1, t2);
      if (t1 > t)
         continue;
      if (t1 < min) {
         min = t1;
         found = true;
      }
   }
}
if (!found)
   break;
cr.clear();
t -= min;
for (Plate plate : plates) {
   if (!plate.isFlying())
```

```
         continue;
      plate.setPosition(plate.getPosition().add(plate.getSpeed().multiply(min)));
   }

   for (int i = 0; i < plates.size(); i++) {
      Plate plate1 = plates.get(i);
      if (!plate1.isFlying())
         continue;
      for (int j = 0; j < plates.size(); j++) {
         Plate plate2 = plates.get(j);
         if (!plate2.isFlying())
            continue;
         if (i >= j)
            continue;

         if (plate1.getPosition().subtract(plate2.getPosition()).getLength() - Config.getPlateDiameter()
         <
               1e-7) {

            Vector v1 = plate1.getSpeed();
            Vector pos1 = plate1.getPosition();
            Vector v2 = plate2.getSpeed();
            Vector pos2 = plate2.getPosition();

            Vector axis = pos1.subtract(pos2);
            double d = axis.getLength();
            axis = axis.multiply(d);
            double v1a = v1.multiply(axis);
            double v2a = v2.multiply(axis);
            double vRel = Math.abs(v1a - v2a);
            if (vRel > 1) {
               cr.add(plate1);
               cr.add(plate2);
               continue;
            } else {
               double v2an = v1a;
               double v1an = v2a;
               v1 = v1.subtract(axis.multiply(v1a)).add(axis.multiply(v1an));
               v2 = v2.subtract(axis.multiply(v2a)).add(axis.multiply(v2an));
               plate1.setSpeed(v1);
               plate2.setSpeed(v2);
            }
         }
      }
   }
   for (Plate plate : cr)
      plate.crash();
}
for (Plate plate : plates) {
   if (!plate.isFlying())
      continue;
   plate.setPosition(plate.getPosition().add(plate.getSpeed().multiply(t)));
```

```
        }
    }

    public void processFlyingOutPlates() {
        for (Plate plate : plates) {
            if (!plate.isFlying())
                continue;
            Vector pos = plate.getPosition();
            if (pos.x <= 0) {
                plate.crash();
                plate.setPosition(new Vector(0, pos.y));
            } else if (pos.y <= 0) {
                plate.crash();
                plate.setPosition(new Vector(pos.x, 0));
            } else if (pos.y >= Config.getFieldHeight()) {
                plate.crash();
                plate.setPosition(new Vector(pos.x, Config.getFieldHeight()));
            } else
                continue;
            plate.setSpeed(new Vector(0, 0));
        }
    }

    public boolean gameOver() {
        for (Plate plate : plates) {
            if (plate.isFlying()) {
                return false;
            }
        }
        return true;
    }
}
```

### 1.1.4. plates/Plate.java

```
package plates;

import plates.utils.Vector;

public class Plate {

    private enum State {
        FLYING,
        LANDED,
        CRASHED
    }

    private State state;

    private Vector position;

    private Vector speed;
private double fuel;

private double q;

private double a;

public Vector getPosition() {
   return position;
}

public void setPosition(Vector v) {
   position = v;
}

public Vector getSpeed() {
   return speed;
}

public void setSpeed(Vector speed) {
   this.speed = speed;
}

public double getFuel() {
   return fuel;
}

public double getQ() {
   return q;
}

public void setQ(double q) {
   if (q > 1.0) {
      q = 1.0;
   }
   if (q < 0.0) {
      q = 0.0;
   }
   if (q > fuel) {
      q = fuel;
   }
   this.q = q;
}

public double getA() {
   return a;
}

public void setA(double a) {
   int angle = Config.getMaximalRotateAngle();
   if (a < -angle)
      a = -angle;
   if (a > angle)
```

```
        a = angle;
    this.a = a;
}

public boolean isCrashed() {
    return state == State.CRASHED;
}

public boolean isFlying() {
    return state == State.FLYING;
}

public void land() {
    state = State.LANDED;
}

public void crash() {
    state = State.CRASHED;
}

public void decFuel(double df) {
    fuel -= df;
}

public int getNumberActions() {
    return 6;
}

public void doAction(int i) {
    switch (i) {
        case 0:
            // Normal speed
            setQ(0.4);
            break;
        case 1:
            // Turn left
            setA(a + 12.5);
            break;
        case 2:
            // Turn right
            setA(a - 12.5);
            break;
        case 3:
            // Fly horizontally
            setA(a + Math.atan2(speed.y, speed.x) * 180 / Math.PI);
            break;
        case 4:
            // Full speed
            setQ(1);
            break;
        case 5:
            // Increase fuel consumption by 0.2
            setQ(q + 0.2);
            break;
         case 6:
            // Decrease fuel consumption by 0.2
            setQ(q - 0.2);
            break;
      }
   }

   public Plate(Vector position, Vector speed, double initFuel) {
      state = State.FLYING;
      this.position = position;
      this.speed = speed;
      this.fuel = initFuel;

      this.q = 0;
      this.a = 0;
   }
}
```

## 1.2. Пакет plates.automaton

### 1.2.1. plates/automaton/Automaton.java

```
package plates.automaton;

import ga.Individual;

import plates.Competition;
import plates.Plate;
import plates.managers.*;

public abstract class Automaton implements Individual, Cloneable {

   private Plate plate;

   private double fitness;

   public Automaton() {
      fitness = Double.NEGATIVE_INFINITY;
   }

   public double fitness() {
      if (fitness == Double.NEGATIVE_INFINITY) {
         fitness = Fitness.fitness(this);
      }
      return fitness;
   }

   public int compareTo(Individual arg0) {
      return Double.compare(arg0.fitness(), fitness());
   }
```

```
    public abstract void doTurn(boolean[] variables);

    public abstract Automaton repairedAutomaton();

    public abstract Automaton clone();

    protected Plate getPlate() {
        return plate;
    }

    public void setPlate(Plate plate) {
        this.plate = plate;
    }

}
```

### 1.2.2. plates/automaton/Fitness.java

```
package plates.automaton;

import plates.managers.Manager;
import plates.managers.AutomatonManager;
import plates.managers.AggressiveManager;
import plates.Competition;
import static starting.Main.parse;

import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

public class Fitness {

    private static Manager opponent1 = readOpponent("conf\\automaton.def");

    private static Manager readOpponent(String name) {
        try {
            return new AutomatonManager(parse(new BufferedReader(new FileReader(name))));
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }

    private static Manager opponent2 = new AggressiveManager();

    public static double fitness(Automaton a) {
        Manager m = new AutomatonManager(a);
        double f = (new Competition(m, opponent1).getResult(0)) + (new Competition(opponent1,
                m).getResult(1));
        for (int i = 0; i < 8; i ++) {
            f += new Competition(m, opponent2).getResult(0);
        }
        return f / 10;
    }

    public static double fitness2(Automaton a) {
        Manager m = new AutomatonManager(a);
        double f = (new Competition(m, opponent1).getResult(0)) + (new Competition(opponent1,
                m).getResult(1));
        for (int i = 0; i < 28; i++) {
            f += new Competition(m, opponent2).getResult(0);
        }
        return f / 30;
    }
}
```

### 1.2.3. plates/automaton/TableAutomaton.java

```
package plates.automaton;

import ga.Individual;

import java.util.Random;
import java.util.LinkedList;

public class TableAutomaton extends Automaton {

    public static final double ENDSTATEMUTATION = 0.3;

    public static final double ACTIONSMUTATION = 0.3;

    public static final double PREDICATMUTATION = 0.3;

    private int numberVariables;

    public int getNumberVariables() {
        return numberVariables;
    }

    private int numberActions;

    private int numberStates;

    public int getNumberStates() {
        return numberStates;
    }

    private int initialState;

    public int getInitialState() {
        return initialState;
    }

    private int currentState;

    private int countAllVariables;
    private State[] state;

    private boolean[] mark;

    public TableAutomaton(int numberStates, int numberActions, int numberVariables,
                          int countAllVariables, int is) {
        this.numberStates = numberStates;
        this.numberActions = numberActions;
        this.numberVariables = numberVariables;
        this.countAllVariables = countAllVariables;
        state = new State[numberStates];
        initialState = is;
        currentState = is;
        mark = new boolean[numberStates];
    }

    public TableAutomaton mutate(Random r) {
        TableAutomaton mut = new TableAutomaton(numberStates, numberActions, numberVariables,
                countAllVariables, initialState);
        System.arraycopy(state, 0, mut.state, 0, state.length);
        int temp = r.nextInt(numberStates);
        state[temp] = state[temp].mutate(r);
        if (r.nextBoolean()) {
            mut.initialState = r.nextInt(numberStates);
        }
        return mut.repairedAutomaton();
    }

    public TableAutomaton[] crossover(Individual p, Random r) {
        TableAutomaton a = (TableAutomaton) p;
        TableAutomaton[] res = new TableAutomaton[2];
        res[0] = new TableAutomaton(numberStates, numberActions, numberVariables,
                countAllVariables, initialState);
        res[1] = new TableAutomaton(numberStates, numberActions, numberVariables,
                countAllVariables, a.initialState);
        for (int i = 0; i < numberStates; i++) {
            State[] z = state[i].crossover(a.state[i], r);
            res[0].state[i] = z[0];
            res[1].state[i] = z[1];
        }
        res[0] = res[0].repairedAutomaton();
        res[1] = res[1].repairedAutomaton();
        return res;
    }

    public void doTurn(boolean[] variable) {
        State s = state[currentState];
        int index = 0;
        int power = 1;
        for (int i = countAllVariables - 1; i >= 0; i--) {
            if (s.variable[i]) {
                if (variable[i]) {
                    index += power;
                }
                power = power << 1;
            }
        }
        for (int i = 0; i < numberActions; i++) {
            if (s.transitionTable[index][i]) {
                getPlate().doAction(i);
            }
        }
        currentState = s.endState[index];
    }

    public void setState(int i, State st) {
        state[i] = st;
    }

    public State getState(int i) {
        return state[i];
    }

    public TableAutomaton repairedAutomaton() {
        Random r = new Random();
        for (int k = 0; k < 10; k++) {
            for (int i = 0; i < numberStates; i++) {
                mark[i] = false;
            }
            mark[initialState] = true;
            LinkedList<Integer> queue = new LinkedList<Integer>();
            queue.addLast(initialState);
            while (!queue.isEmpty()) {
                State st = state[queue.removeFirst()];
                for (int i = 0; i < 1 << numberVariables; i++) {
                    int endState = 0;
                    try {
                        endState = st.endState[i];
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                    if (!mark[endState]) {
                        mark[endState] = true;
                        queue.addLast(endState);
                    }
                }
            }
            boolean flag = true;
            for (int i = initialState + 1; i < numberStates; i++) {
                if (!mark[i]) {
                    state[i - 1] = state[i - 1].setEndState(r.nextInt(1 << numberVariables), i);
                    flag = false;
                }
            }
            if (mark[0]) {
                state[numberStates - 1] = state[numberStates - 1].setEndState(r.nextInt(1 << numberVariables),
                    0);
                flag = false;
            }
            for (int i = 1; i < initialState; i++) {
                if (!mark[i]) {
                    flag = false;
                    state[i - 1] = state[i - 1].setEndState(r.nextInt(1 << numberVariables), i);
                }
            }
            if (flag) {
                break;
            }
        }
        return this;
    }

    public class State implements Cloneable {

        private int[] endState;

        private boolean[][] transitionTable;

        private boolean[] variable;

        public State(int[] endState, boolean[][] transitionTable, boolean[] variable) {
            this.transitionTable = transitionTable;
            this.endState = endState;
            this.variable = variable;
        }

        private State() {
            transitionTable = new boolean[(1 << numberVariables)][numberActions];
            endState = new int[(1 << numberVariables)];
            variable = new boolean[countAllVariables];
        }

        public State mutate(Random r) {
            State mut = new State();
            System.arraycopy(variable, 0, mut.variable, 0, countAllVariables);
            if (r.nextDouble() < PREDICATMUTATION) {
                int f = r.nextInt(countAllVariables);
                int s = r.nextInt(countAllVariables);
                if (variable[f] && !variable[s]) {
                    mut.variable[f] = false;
                    mut.variable[s] = true;
                }
            }
            for (int i = 0; i < endState.length; i++) {
                if (r.nextDouble() < ENDSTATEMUTATION) {
```

```
            mut.endState[i] = r.nextInt(numberStates);
        } else {
            mut.endState[i] = endState[i];
        }
        if (r.nextDouble() < ACTIONSMUTATION) {
            double probability = 0;
            for (int j = 0; j < numberActions; j++) {
                if (transitionTable[i][j]) probability += 1.0 / numberActions;
            }
            for (int j = 0; j < numberActions; j++) {
                mut.transitionTable[i][j] = r.nextDouble() < probability;
            }
        }
    }
    return mut;
}

public State[] crossover(State p, Random r) {
    State[] res = new State[2];
    res[0] = new State();
    res[1] = new State();
    choosePred(p, res[0], res[1], r);
    int t = r.nextInt((1 << numberVariables));
    for (int i = 0; i < (1 << numberVariables); i++) {
        if (i <= t) {
            res[0].endState[i] = endState[i];
            res[1].endState[i] = p.endState[i];
        } else {
            res[1].endState[i] = endState[i];
            res[0].endState[i] = p.endState[i];
        }
    }
    for (int i = 0; i < numberActions; i++) {
        t = r.nextInt((1 << numberVariables));
        for (int j = 0; j < (1 << numberVariables); j++) {
            if (j <= t) {
                res[0].transitionTable[j][i] = transitionTable[j][i];
                res[1].transitionTable[j][i] = p.transitionTable[j][i];
            } else {
                res[1].transitionTable[j][i] = transitionTable[j][i];
                res[0].transitionTable[j][i] = p.transitionTable[j][i];
            }
        }
    }
    return res;
}

public void choosePred(State par, State ch1, State ch2, Random r) {
    for (int i = 0; i < countAllVariables; i++) {
        ch1.variable[i] = false;
        ch2.variable[i] = false;
    }
    int r1 = numberVariables;
    int r2 = numberVariables;
    for (int i = 0; i < countAllVariables; i++) {
        if (variable[i] && par.variable[i]) {
            ch1.variable[i] = true;
            ch2.variable[i] = true;
            r1--;
            r2--;
        }
    }
    for (int i = 0; i < countAllVariables; i++) {
        if (variable[i] != par.variable[i]) {
            if ((r1 > 0) && (r2 > 0)) {
                if (r.nextBoolean()) {
                    ch1.variable[i] = true;
                    r1--;
                } else {
                    ch2.variable[i] = true;
                    r2--;
                }
            } else {
                if (r1 > 0) {
                    ch1.variable[i] = true;
                    r1--;
                } else {
                    ch2.variable[i] = true;
                    r2--;
                }
            }
        }
    }
}

public State setEndState(int i, int en) {
    State res = clone();
    res.endState[i] = en;
    return res;
}

public int getEndState(int i) {
    return endState[i];
}

public boolean[] getActions(int i) {
    return transitionTable[i];
}

public int getVariable(int i) {
    int k = -1;
    for (int j = 0; j < countAllVariables; j++) {
        if (variable[j]) {
            k++;
```

```
                if (i == k) {
                    return j;
                }
            }
        }
        return -1;
    }

    public State clone() {
        State res = new State();
        for (int i = 0; i < transitionTable.length; i++) {
            System.arraycopy(transitionTable[i], 0, res.transitionTable[i], 0, transitionTable[0].length);
        }
        System.arraycopy(endState, 0, res.endState, 0, endState.length);
        System.arraycopy(variable, 0, res.variable, 0, variable.length);
        return res;
    }
}

@Override
public Automaton clone() {
    TableAutomaton a = new TableAutomaton(numberStates, numberActions, numberVariables,
            countAllVariables, initialState);
    System.arraycopy(state, 0, a.state, 0, numberStates);
    return a;
}

public String toString() {
    String s = "";
    s += numberStates + " " + initialState + "\n";
    s += countAllVariables + " " + numberVariables + " " + numberActions + "\n";
    for (int i = 0; i < numberStates; i++) {
        s += toIntString(state[i].endState);
        for (int j = 0; j < 1 << numberVariables; j++) {
            s += toBoolString(state[i].transitionTable[j]);
        }
        s += toBoolString(state[i].variable);
    }
    return s;
}

private String toIntString(int[] a) {
    String s = "";
    for (int anA : a) s += anA + " ";
    s += "\n";
    return s;
}

private String toBoolString(boolean[] a) {
    String s = "";
    for (boolean anA : a) {
        s += !anA ? " 0" : " 1";
        }
        s += "\n";
        return s;
    }
}
```

### 1.2.4. plates/automaton/TableAutomatonFactory.java

```
package plates.automaton;

import java.util.Random;

import ga.IndividualFactory;

public class TableAutomatonFactory implements IndividualFactory {

    private final int numberStates;
    private final int countAllVariables;
    private final int numberVariables;
    private final int numberActions;

    private final double pAction;

    private static final Random RANDOM = new Random();

    public TableAutomaton randomIndividual() {
        TableAutomaton a = new TableAutomaton(numberStates, numberActions, numberVariables,
                countAllVariables,
            RANDOM.nextInt(numberStates));
        for (int i = 0; i < numberStates; i++) {
            int numberTransitions = 1 << numberVariables;
            int[] endState = new int[numberTransitions];
            boolean[][] transitionTable = new boolean[numberTransitions][numberActions];
            for (int j = 0; j < numberTransitions; j++) {
                endState[j] = RANDOM.nextInt(numberStates);
                for (int k = 0; k < numberActions; k++) {
                    transitionTable[j][k] = RANDOM.nextDouble() < pAction;
                }
            }
            boolean[] variable = new boolean[countAllVariables];
            for (int j = 0; j < numberVariables; j++) {
                while (true) {
                    int index = RANDOM.nextInt(countAllVariables);
                    if (!variable[index]) {
                        variable[index] = true;
                        break;
                    }
                }
            }
            a.setState(i, a.new State(endState, transitionTable, variable));
        }
        return a.repairedAutomaton();
    }
    public TableAutomatonFactory(int numberStates, int countAllVariables, int numberVariables,
                                  int numberActions, double pAction) {
        this.countAllVariables = countAllVariables;
        this.numberActions = numberActions;
        this.numberStates = numberStates;
        this.numberVariables = numberVariables;
        this.pAction = pAction;
    }
}
```

### 1.2.5. plates/automaton/TableAutomatonFactoryLoader.java

```
package plates.automaton;

import laboratory.util.AbstractLoader;

import java.util.jar.JarFile;

public class TableAutomatonFactoryLoader extends AbstractLoader<TableAutomaton>{

    public TableAutomatonFactory load(Object... args){
        return new TableAutomatonFactory(properties.getInt("count.states"), 8, 3,
                properties.getDouble("mu"));
    }

    public FactoryLoader(JarFile file) {
        super(file, "automaton.conf");
    }
}
```

### 1.2.6. plates/automaton/TreeAutomatonFactoryLoader.java

```
package plates.automaton;

import laboratory.util.AbstractLoader;

import java.util.jar.JarFile;

public class TreeAutomatonFactoryLoader extends AbstractLoader<TreeAutomaton>{

    public TreeAutomatonFactory load(Object... args){
        return new TreeAutomatonFactory(properties.getInt("count.states"), 8, 3,
                properties.getDouble("mu"));
    }

    public FactoryLoader(JarFile file) {
        super(file, "automaton.conf");
    }
}
```

### 1.2.7. plates/automaton/TreeAutomaton.java

```
package plates.automaton;

import plates.automaton.tree.Tree;
import plates.automaton.tree.Tree.TreeNode;

import java.util.Random;

import ga.Individual;

public class TreeAutomaton extends Automaton {

   public static final double VERTEX_MUTATION_PROBABILITY = 0.5;

   public static final double VERTEX_CROSSOVER_PROBABILITY = 0.5;

   public static final int PENALTY = 3;

   private Tree[] state;

   private int initialState;

   private int currentState;

   private int height;

   private double fitness;

   private TreeAutomatonFactory fact;

   public TreeAutomaton(int numberStates, int is, int height, TreeAutomatonFactory fact) {
      state = new Tree[numberStates];
      initialState = is;
      currentState = is;
      this.height = height;
      fitness = Double.NEGATIVE_INFINITY;
      this.fact = fact;
   }

   public void doTurn(boolean[] variable) {
      TreeNode s = state[currentState].getNode(variable);
      for (int i = 0; i < s.getActions().length; i ++) {
         if (s.getActions()[i]) {
            getPlate().doAction(i);
         }
      }
      currentState = s.getEndState();
   }

   public TreeAutomaton repairedAutomaton() {
      return this;
   }
```

```
public void setState(int i, Tree a) {
   state[i] = a;
}

public Tree getState(int i) {
   return state[i];
}

public int getInitialState() {
   return initialState;
}

public int getNumberStates() {
   return state.length;
}

public int getCountAllVariables() {
   return fact.getCountAllVariables();
}

public int getNumberActions() {
   return fact.getNumberActions();
}

public int getHeight() {
   return height;
}

public TreeAutomaton clone() {
   TreeAutomaton res = new TreeAutomaton(state.length, initialState, height, fact);
   System.arraycopy(state, 0, res.state, 0, state.length);
   return res;
}

public TreeAutomaton mutate(Random r) {
   TreeAutomaton mut = clone();
   if (r.nextBoolean()) {
      mut.initialState = r.nextInt(state.length);
   }
   mut.state[r.nextInt(state.length)] = mut.state[r.nextInt(state.length)].mutate(fact.randomTree(0), r);
   return mut.repairedAutomaton();
}

public TreeAutomaton[] crossover(Individual p, Random r) {
   TreeAutomaton[] res = new TreeAutomaton[2];
   res[0] = new TreeAutomaton(state.length, initialState, height, fact);
   res[1] = new TreeAutomaton(state.length, initialState, height, fact);
   TreeAutomaton a = (TreeAutomaton) p;
   for (int i = 0; i < state.length; i ++) {
      Tree[] tree = state[i].crossover(a.state[i], r);
      res[0].setState(i, tree[0]);
```

```
            res[1].setState(i, tree[1]);
        }
        return res;
    }

    public double fitness() {
        if (fitness == Double.NEGATIVE_INFINITY) {
            int max = height;
            for (int i = 0; i < state.length; i ++) {
                max = Math.max(max, state[i].getHeight());
            }
            fitness = super.fitness() - (max - height) * PENALTY;
        }
        return fitness;
    }

    public String toString() {
        String s = "";
        s += state.length + " " + initialState + "\n";
        s += fact.getNumberActions() + "\n";
        for (int i = 0; i < state.length; i ++) {
            s += state[i].toString() + "\n";
        }
        return s;
    }
}
```

### 1.2.8. plates/automaton/TreeAutomatonFactory.java

```
package plates.automaton;

import ga.IndividualFactory;
import plates.automaton.tree.Tree;
import plates.automaton.tree.Tree.TreeNode;

import java.util.Random;

public class TreeAutomatonFactory implements IndividualFactory {

    private static final double VERTEXPROBABILITY = 0.75;

    private int numberStates;

    private int countAllVariables;

    private int numberActions;

    private double pAct;

    private int height;

    private static final Random RANDOM = new Random();
    public TreeAutomatonFactory(int numberStates, int countAllVariables, int numberActions,
                  double pAct, int height) {
        this.countAllVariables = countAllVariables;
        this.numberActions = numberActions;
        this.numberStates = numberStates;
        this.pAct = pAct;
        this.height = height;
    }

    public int getCountAllVariables() {
        return countAllVariables;
    }

    public int getNumberActions() {
        return numberActions;
    }

    public TreeAutomaton randomIndividual() {
        TreeAutomaton res = new TreeAutomaton(numberStates, RANDOM.nextInt(numberStates), height,
              this);
        for (int i = 0; i < numberStates; i ++) {
            res.setState(i, new Tree(randomTree(0)));
        }
        return res;
    }

    public TreeNode randomTree(int level) {
        if((RANDOM.nextDouble() > VERTEXPROBABILITY) || (level >= 2 * countAllVariables)){
            int endState = RANDOM.nextInt(numberStates);
            boolean[] var = new boolean[numberActions];
            for (int i = 0; i < var.length; i ++) {
                var[i] = RANDOM.nextDouble() < pAct;
            }
            return new TreeNode(endState, var);
        }
        return new TreeNode(randomTree(level + 1), randomTree(level + 1),
              RANDOM.nextInt(countAllVariables));
    }

}
```

## 1.3. Пакет plates.managers

### 1.3.1. plates/managers/AgressiveManager.java

```
package plates.managers;

import java.util.List;
import java.util.Random;

import plates.Plate;
import plates.Config;
```

```
public class AggressiveManager implements Manager {

    private List<Plate> plates;
    private Random random = new Random();

    public void doTurn() {
        int cnt = 0;
        for (Plate plate : plates) {
            cnt++;
            if (cnt < Config.getPlatesCount()) {
                if ((Math.abs(plate.getSpeed().x) < 1e-6) || (plate.getSpeed().y / plate.getSpeed().x > -0.3)) {
                    plate.setA(random.nextDouble() * Config.getMaximalRotateAngle());
                } else {
                    plate.setA(0);
                }
                plate.setQ(random.nextDouble() + 0.4);
            } else {
                if (plate.getPosition().y > Config.getFieldHeight() - 2) {
                    plate.setA(5);
                } else {
                    plate.setA(Math.atan2(plate.getSpeed().y, plate.getSpeed().x) * 180 / Math.PI);
                }
                plate.setQ(0.4);
            }
        }

    }

    public List<Plate> getPlates() {
        return plates;
    }

    public void init(List<Plate> plates, List<Plate> allPlates) {
        this.plates = plates;
    }
}
```

### 1.3.2. plates/managers/AutomatonManager.java

```
package plates.managers;

import java.util.List;
import java.util.ArrayList;

import plates.Plate;
import plates.Config;
import plates.automaton.Automaton;
import plates.utils.Vector;

public class AutomatonManager implements Manager {

    private List<Automaton> automata;
    private List<Plate> plates;
private List<Plate> allPlates;

public void doTurn() {
   for (int i = 0; i < plates.size(); i++) {
      Plate plate = plates.get(i);
      if (plate.isFlying()) {
         boolean leftBorderIsNear = plate.getPosition().y < 2;
         boolean rightBorderIsNear = Config.getFieldHeight() - plate.getPosition().y < 2;
         boolean otherPlateOnTheLeft = false;
         boolean otherPlateOnTheRight = false;
         boolean otherPlateInFront = false;
         boolean otherPlateBehind = false;
         for (Plate otherPlate : allPlates) {

            if (plate == otherPlate) continue;
            if (!otherPlate.isFlying())
               continue;

            if (!canHit(plate, otherPlate))
               continue;

            if (plate.getPosition().subtract(otherPlate.getPosition()).getLength() <= 10) {
               if (((plate.getPosition().y - otherPlate.getPosition().y) > 0) &&
                     ((plate.getPosition().y - otherPlate.getPosition().y) <= 5) &&
                     (Math.abs(plate.getPosition().x - otherPlate.getPosition().x) < 2)) {
                  // Somebody on the left
                  otherPlateOnTheLeft = true;
               }
               if (((plate.getPosition().y - otherPlate.getPosition().y) < 0) &&
                     ((plate.getPosition().y - otherPlate.getPosition().y) >= -5) &&
                     (Math.abs(plate.getPosition().x - otherPlate.getPosition().x) < 2)) {
                  // Somebody on the right
                  otherPlateOnTheRight = true;
               }

               if ((Math.abs(plate.getPosition().y - otherPlate.getPosition().y) < 3) &&
                     ((plate.getPosition().x - otherPlate.getPosition().x) >= -5) &&
                     ((plate.getPosition().x - otherPlate.getPosition().x) < 0)) {
                  // Somebody in front of us
                  otherPlateInFront = true;

               }

               if ((Math.abs(plate.getPosition().y - otherPlate.getPosition().y) < 3) &&
                     ((plate.getPosition().x - otherPlate.getPosition().x) <= 5) &&
                     ((plate.getPosition().x - otherPlate.getPosition().x) > 0)) {
                  // Somebody behind us
                  otherPlateBehind = true;
               }

            }
         }
```

```
            boolean[] x = new boolean[]{leftBorderIsNear, rightBorderIsNear,
                    otherPlateOnTheLeft, otherPlateOnTheRight, otherPlateInFront, otherPlateBehind};
            automata.get(i).doTurn(x);
        }
    }
}

private boolean canHit(Plate thisPlate, Plate other) {
    Vector v1 = thisPlate.getSpeed();
    Vector pos1 = thisPlate.getPosition();
    Vector v2 = other.getSpeed();
    Vector pos2 = other.getPosition();

    Vector deltaV = v1.subtract(v2);
    Vector deltaPos = pos1.subtract(pos2);

    double a = deltaV.getLength() * deltaV.getLength();
    double b = 2 * deltaV.multiply(deltaPos);
    double c = deltaPos.getLength() * deltaPos.getLength() - 4 * Config.getPlateDiameter() *
            Config.getPlateDiameter();
    double d = b * b - 4 * a * c;

    if (d < 0)
        return false;
    if (Math.abs(a) <= 1e-9)
        return false;

    double t1 = (-b + Math.sqrt(d)) / (2 * a);
    double t2 = (-b - Math.sqrt(d)) / (2 * a);

    return (t1 >= 0) || (t2 >= 0);
}

public List<Plate> getPlates() {
    return plates;
}

public AutomatonManager(Automaton a) {
    int n = Config.getPlatesCount();
    automata = new ArrayList<Automaton>(n);
    automata.add(a);
    for (int i = 1; i < n; i++) {
        automata.add(a.clone());
    }
}

public void init(List<Plate> plates, List<Plate> allPlates) {
    this.plates = plates;
    this.allPlates = allPlates;
    for (int i = 0; i < automata.size(); i++) {
        automata.get(i).setPlate(plates.get(i));
    }
    }

}
```

### 1.3.3. plates/managers/Manager.java

```
package plates.managers;

import java.util.List;

import plates.Plate;

public interface Manager {

    public void doTurn();

    public List<Plate> getPlates();

    public void init(List<Plate> plates, List<Plate> allPlates);
}
```

# ЛИСТ РЕГИСТРАЦИИ ИЗМЕНЕНИЙ

| *Номера листов (страниц)* | | | | | *Всего листов (страниц) в докум.* | *№ документа* | *Входящий № сопроводительного докум. и дата* | *Подп.* | *Дата* |
|---|---|---|---|---|---|---|---|---|---|
| *Изм.* | *Измененных* | *Замененных* | *Новых* | *Аннулированных* | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |