Санкт-Петербургский государственный университет
информационный технологий, механики и оптики
Факультет информационных технологий и программирования
Кафедра компьютерных технологий

Ю. И. Попов

Отчет по лабораторной работе «Построение управляющих автоматов с помощью генетических алгоритмов»

Вариант № 2

Санкт-Петербург
2009

Оглавление

Введение

В лабораторной работе изучается метод генерации конечных автоматов с помощью генетических алгоритмов. Полученные автоматы используются для решения задачи об «Умном муравье». В рассматриваемом варианте целью работы является автомат Мура, представляющий логику муравья.

При выполнении лабораторной работы использовалась «Виртуальная лаборатория» [1], написанная студентами СПбГУ ИТМО. Лаборатория позволяет при подключении написанного плагина рассмотреть диаграмму переходов конечного автомата, а также изучить графики функции приспособленности.

1. Постановка задачи

Задачей лабораторной работы являлось построение с помощью генетических алгоритмов конечного автомата Мура, решающего задачу об «Умном муравье». При этом следовало использовать представление автоматов с помощью битовых строк, традиционный генетический алгоритм и метод «рулетки» для генерации очередного поколения [2].

1.1. Задача об «Умном муравье»

Двумерное поле 32 на 32 клетки расположено на поверхности тора [3]. На поле вдоль некоторой ломаной, но не на всех ее клетках находятся яблоки. Муравей вначале находится на стартовой клетке. Задача муравья съесть максимальное число яблок за 200 ходов. За один ход он может выполнить одно из четырех действий:

- сделать ход вперед, съедая еду, если она там находиться;
- повернуться налево;
- повернуться направо;
- остаться на месте.

Для того чтобы жить, не обязательно постоянно есть. Яблоки не восполняются, и их число равно 89. В каждый момент муравей видит только клетку перед собой.

На рис. 1 рассмотрена поверхность тора. Муравей изображен в виде стрелки, расположенной в левом верхнем углу. Яблоками являются черные прямоугольники, они расположены вдоль ломанной. Если муравей дойдет до края поля и двинется за границу, то появится с противоположной стороны (тор).

Для того чтобы визуализатор стал доступен, необходимо в конфигурационном файле эмулятора дописать в параметр `filter` название особи `BitMoore`.

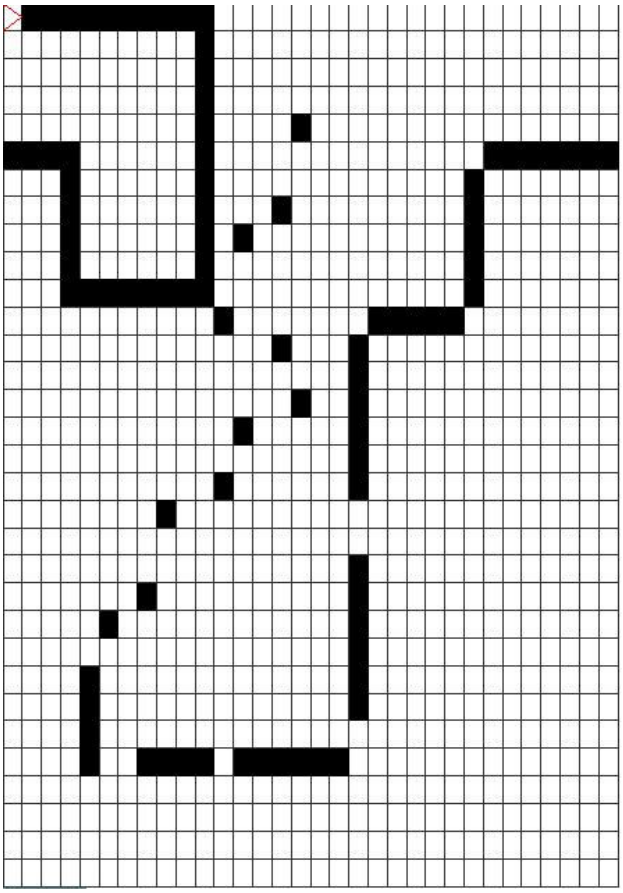

Рис. 1. Поверхность тора

2. Автомат Мура

В рассматриваемом варианте автомат для управления муравьем является автоматом Мура – совокупность пяти объектов $A = \{S, P, W, \delta, \mu\}$, где:

- S – множество вершин;
- P – множество входных воздействий;
- W – множество выходных воздействий;
- δ – отображение $S \times P \rightarrow S$;
- μ – отображение $S \rightarrow W$.

Пример диаграммы переходов автомата Мура приведен на рис. 2.

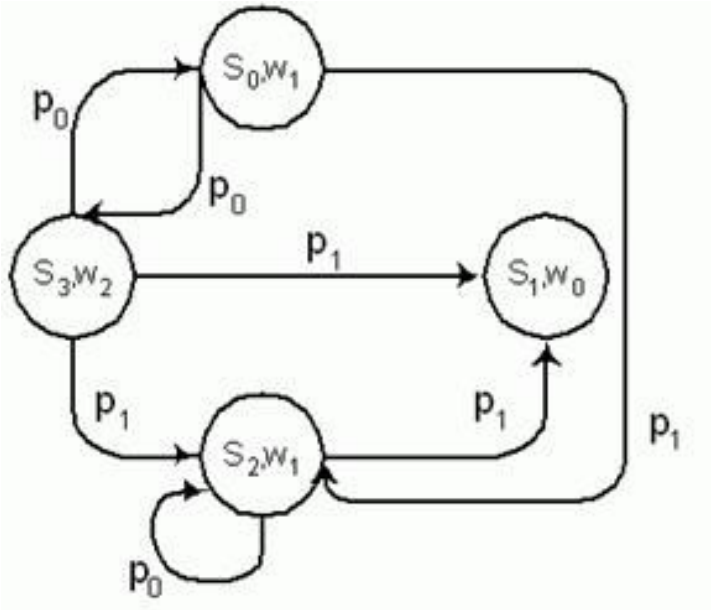

Рис. 2. Автомат Мура

Муравей получает входное воздействие в виде сигнала: перед ним есть яблоко или нет яблока. Автомат, исходя из текущего состояния и воздействия, переходит в следующую вершину (отображение δ). На следующем этапе по отображению μ выбирается действие муравья.

2.1. Способ представления автомата

Автомат, управляющий муравьем, представляется массивом битов. По числу состояний S определяется необходимое число битов для хранения номера состояния, исходя из его двоичного представления. В начале массива располагается номер стартового состояния. Далее S раз повторяется следующая конструкция: номер вершины для перехода в случае отсутствия пищи на клетке перед муравьем, номер вершины в случае наличия пищи, выходное воздействие (ход вперед, поворот налево, направо, остаться на месте). Ход кодируется двумя битами [3]:

- 00 – остаться на месте;
- 01 – повернуться налево;
- 10 – повернуться направо;
- 11 – сделать ход вперед.

3. Генетический алгоритм

Название «генетические алгоритмы» ассоциирует с биологической генетикой. Действительно, они основаны на природном механизме воспроизводства, соревновании существ за право жить. Особи могут размножаться, мутировать, а также некоторые из них «выживают» и попадают в следующее поколение. Приспособленность особи определяется по ее возможности справиться с некоторой задачей [4].

В лаборатории на первом этапе случайным образом генерируются 200 особей-муравьев, а точнее – автоматов управления. Далее происходят нижеописанные процессы, и отбираются 200 особей для следующего поколения.

3.1. Метод скрещивания

Сначала потомки получают случайным образом стартовые состояния от разных предков. Затем также случайным образом распределяются три признака родителей (переход при наличии еды, при отсутствии еды, действие муравья). При этом потомок получает два признака от одного предка и один от другого. Исходный код метода в приложении.

3.2. Оператор мутации

С некоторой вероятностью мутирует стартовое состояние особи (заменяется случайным). Далее делается проход по каждой вершине автомата. С этой же вероятностью мутирует переход по одному из входных воздействий и с той же вероятностью заменяется действие муравья случайным. Исходный код оператора в приложении.

3.3. Метод вычисления функции приспособленности

Муравей с полученным с помощью генетического алгоритма автоматом, либо со случайным автоматом (на первом этапе) передвигается по поверхности тора (32 на 32 клетки) и ест яблоки, если его путь пересечется с пищей. Запоминается ход, на котором он съел свое последнее яблоко. Функция приспособленности вычисляется как разность числа съеденных яблок и отношения последнего питательного хода к общему числу ходов. Таким образом, из двух муравьев, съевших одинаковое число яблок, более приспособленным является тот, который съест их быстрее. Исходный код метода в приложении.

3.4. Метод генерации следующего поколения

Для генерации следующего поколения используется метод «рулетки» [4]. Можно представить колесо рулетки, разбитое на секторы. Число секторов равно числу особей, секторы не равны по площади (в общем случае), а пропорциональны функции приспособленности особей. Затем генерируется случайное число и проверяется, в какой сектор оно попало. При равномерном распределении случайных чисел в промежуточное поколение попадет больше особей с хорошей функцией приспособленности. Далее случайным образом 100 раз (особей 200) выбираются две пары. Лучшая особь из первой пары скрещивается с лучшей из второй. Потомки попадают в следующее промежуточное поколение. На последнем этапе с некоторой вероятностью каждая особь мутирует. Получается новое поколение, содержащее то же число особей, что и предыдущее. Исходный код метода в приложении.

4. Построенный автомат

На рис. 3 изображен автомат Мура, полученный с помощью генетических алгоритмов в результате выполнения лабораторной. В вершинах находятся их номера и результат отображения μ (`M` – ход вперед, `L` – поворот налево, `R` – направо). Красным цветом выделена стартовая вершина. На ребрах: `T` – переход, если есть еда; `F` – переход, если нет еды.

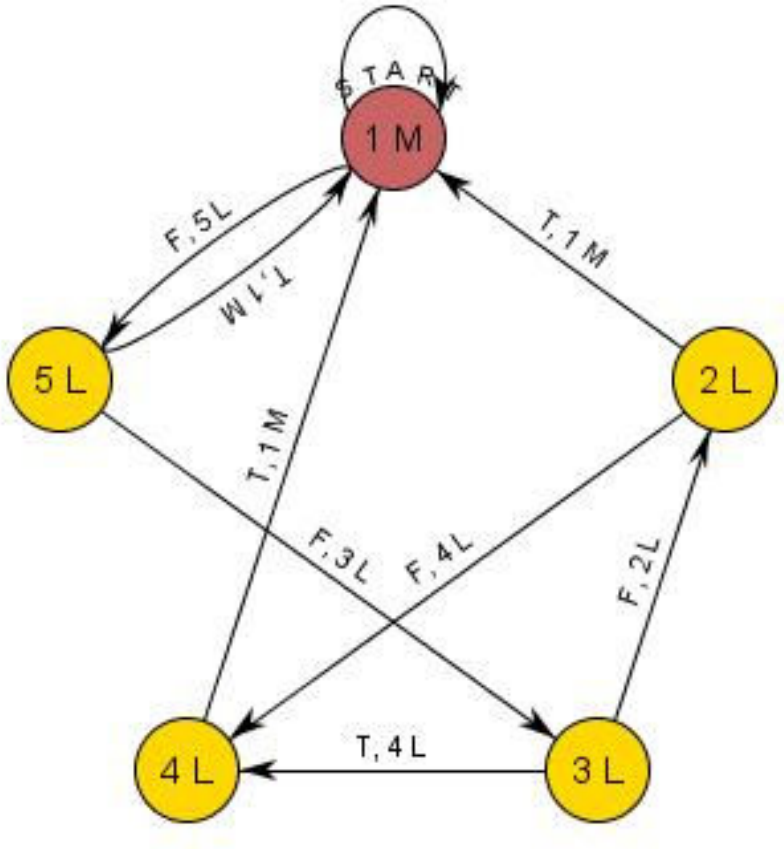

Рис. 3. Построенный автомат Мура

4.1. График максимального значения функции приспособленности

График зависимости максимального значения функции приспособленности от номера поколения изображен на рис. 4.

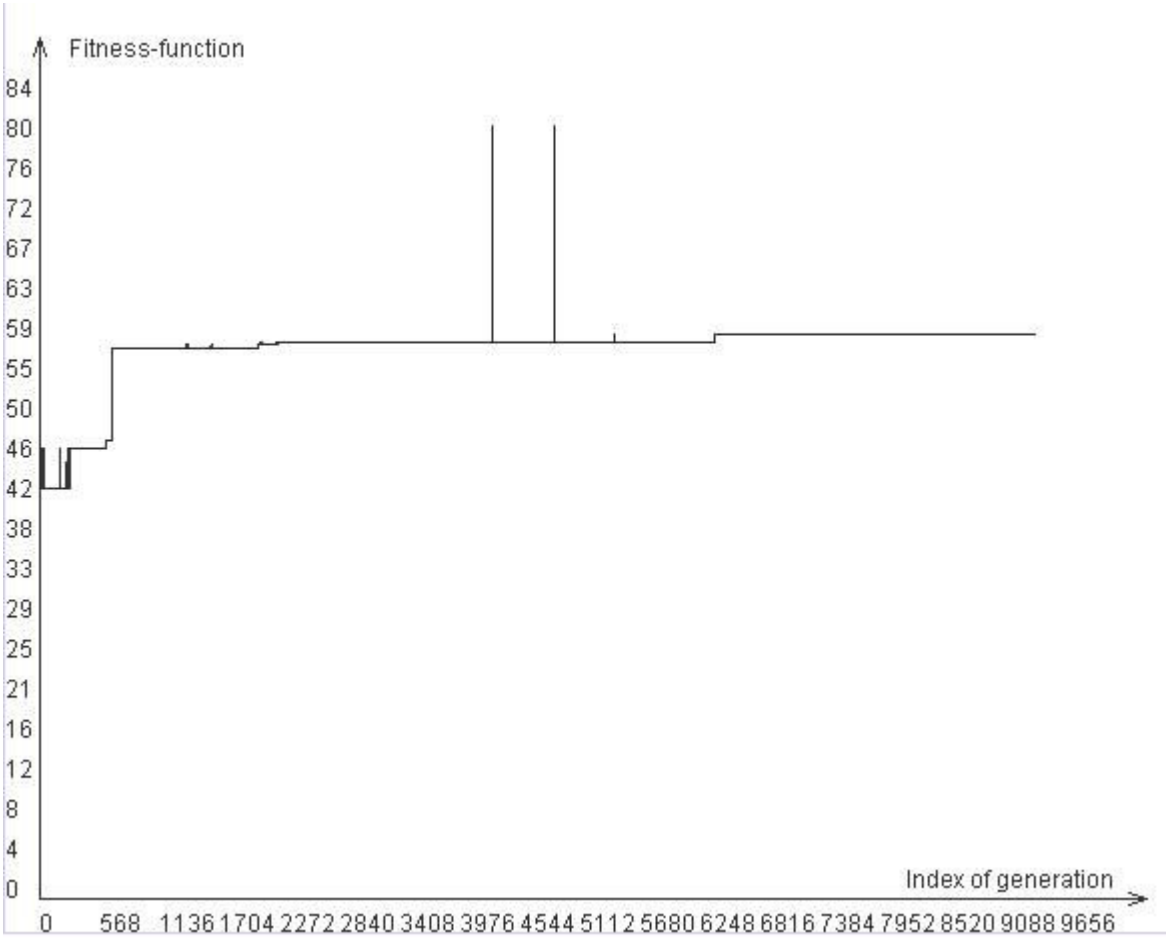

Рис. 4. График зависимости максимального значения функции приспособленности от номера поколения

4.2. График среднего значения функции приспособленности

График зависимости среднего значения функции приспособленности от номера поколения изображен на рис. 5.

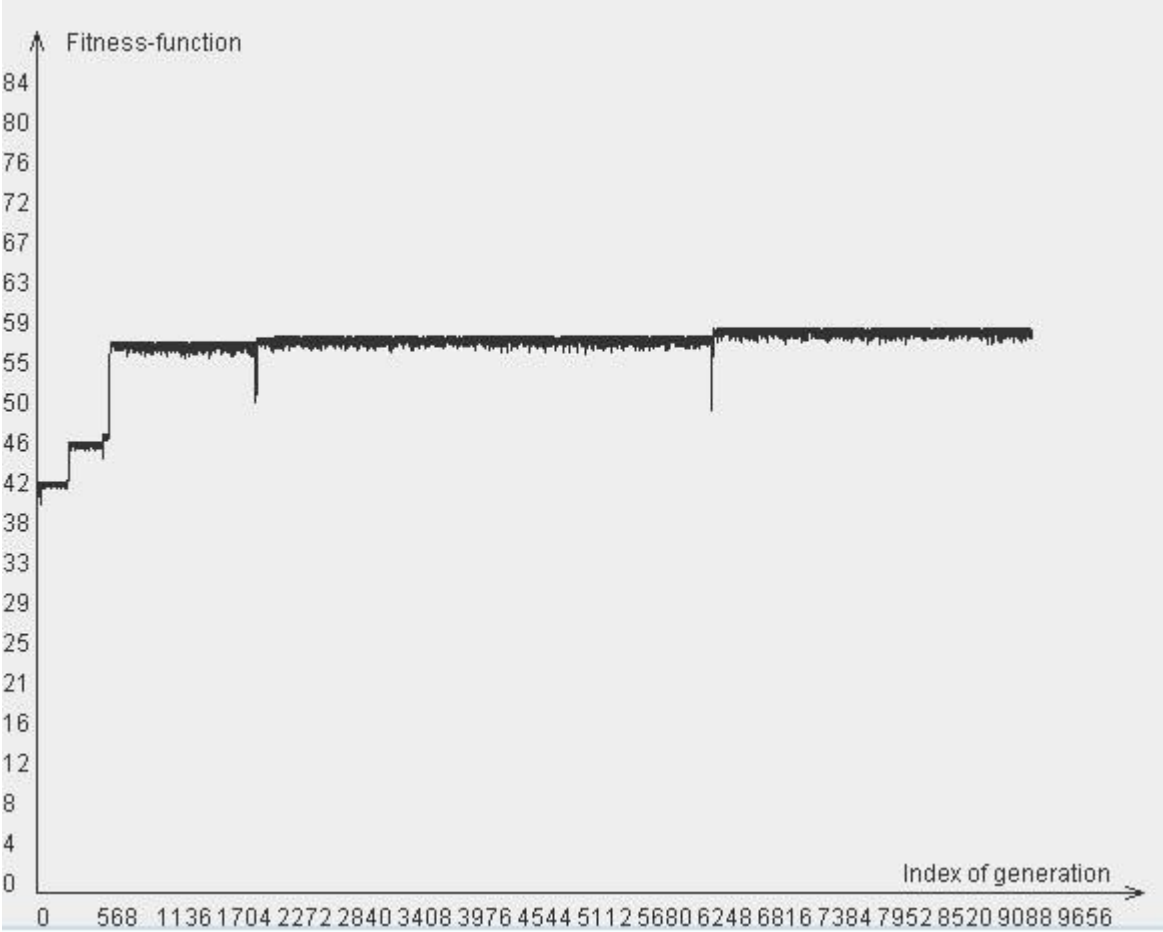

Рис. 5. Зависимость среднего значения функции приспособленности от номера поколения

Заключение

Результаты лабораторной работы показали, что с помощью генетических алгоритмов можно сгенерировать автомат Мура с пятью состояниями, который эффективно решает задачу об «Умном муравье». Полученный муравей съедает 81 яблоко. В работе [5] рассмотрен автомат Мура с 10 состояниями, съедающий всю еду за 198 шагов. Там же отмечено, что при использовании их подхода и автомата с девятью состояниями, муравей съедает за 198 ходов только 86 яблок.

Источники

1. *Инструкция* по созданию плагинов к виртуальной лаборатории. http://svn2.assembla.com/svn/not_instrumental_tool/docs/pdf/interface_manual.pdf
2. *Условие* лабораторной работы. http://svn2.assembla.com/svn/not_instrumental_tool/docs/pdf/labs.pdf
3. *Бедный Ю. Д., Шалыто А. А.* Применение генетических алгоритмов для построения автоматов в задаче «Умный муравей». http://is.ifmo.ru/works/_ant.pdf
4. *Яминов Б.* Генетические алгоритмы. http://rain.ifmo.ru/cat/view.php/theory/unsorted/genetic-2005
5. *Давыдов А. А. и др.* Применение островного генетического алгоритма для построения автоматов Мура и систем взаимодействующих автоматов Мили на примере задачи об «Умном муравье» // Сборник докладов XI международной конференции по мягким вычислениям и измерениям (SCM'2008). СПбГЭТУ. 2008, с. 266-270. http://is.ifmo.ru/genalg/_scm2008_sokolov.pdf

Приложение. Исходный код

Файл BitMooreAutomaton.java

```
package individual.bitMoore;

import laboratory.common.ga.Individual;
import task.ant.simple.individual.Automaton;
import task.ant.simple.SimpleAntTask;
import task.ant.simple.individual.SimpleMover;
import java.util.Random;

/**
 * Автомат Мура, представленный битовой строкой.
 *
 * @author Yuri Popov
 */
public class BitMooreAutomaton implements Automaton {

    /**
     * Число битов, используемых для представления действия.
     */
    private final int QUANTITY_OF_BITS_IN_ACTION = 2;

    /**
     * Число битов в номере состояния.
     */
    private final int QUANTITY_OF_BITS_IN_STATE;

    /**
     * Число состояний.
     */
    private final int QUANTITY_OF_STATES;

    /**
     * Вероятность мутации.
     */
    private final double PROBABILITY_OF_MUTATION = 0.1;

    /**
     * Битовая строка.
     */
    private boolean[] bits;

    /**
     * Первое число, имеющее в 30-м бите 1.
     */
    private int power = Integer.MAX_VALUE / 2 + 1;

    /**
     * Приспособленность особи.
     */
```

```
    private double fitness = Double.NEGATIVE_INFINITY;

    /**
     * Переход автомата.
     */
    public static class Transition implements
Automaton.Transition {

        /**
         * Номер следующего состояния.
         */
        private final int END_STATE;

        /**
         * Автомат.
         */
        private BitMooreAutomaton automaton;

        /**
         * Конструктор перехода.
         *
         * @param endState номер следующего состояния.
         * @param a автомат.
         */
        public Transition(int endState, BitMooreAutomaton a) {
            this.END_STATE = endState;
            automaton = a;
        }

        /**
         * Возвращает номер следующего состояния.
         *
         * @return следующее состояние.
         */
        public int getEndState() {
            return END_STATE;
        }

        /**
         * Возвращает действие следующего состояния.
         *
         * @return действие следующего состояния.
         */
        public char getAction() {
            return automaton.getAction(END_STATE);
        }

        /**
         * Переводит переход в строковый вид.
         *
         * @return строка, описывающая переход.
         */
        @Override
```

```
        public String toString() {
            return (END_STATE + 1) + " " + getAction();
        }
    }

    /**
     * Конструктор автомата.
     *
     * @param initialState начальное состояние автомата.
     * @param quantityOfStates число состояний в автомате.
     */
    public BitMooreAutomaton(int initialState, int
quantityOfStates) {
        if (quantityOfStates < 1 || initialState >=
quantityOfStates || initialState < 0) {
            throw new RuntimeException("Incorrect arguments!");
        }
        this.QUANTITY_OF_STATES = quantityOfStates;
        int bs = 31;
        int maxNumberOfState = quantityOfStates - 1;
        while (maxNumberOfState / power != 1) {
            power >>= 1;
            bs--;
        }
        QUANTITY_OF_BITS_IN_STATE = bs;
        bits = new boolean[QUANTITY_OF_BITS_IN_STATE + (2 *
QUANTITY_OF_BITS_IN_STATE + QUANTITY_OF_BITS_IN_ACTION) *
quantityOfStates];
        setInitialState(initialState);
    }

    /**
     * Задает номер нового начального состояния.
     *
     * @param state номер нового начального состояния.
     */
    public void setInitialState(int state) {
        if (state >= QUANTITY_OF_STATES) {
            throw new RuntimeException("Incorrect number of
state!");
        }
        int pow = power;
        for (int i = 0; i < QUANTITY_OF_BITS_IN_STATE; i++) {
            if (state / pow == 1) {
                bits[i] = true;
                state -= pow;
            } else {
                bits[i] = false;
            }
            pow >>= 1;
        }
    }
```

```
    /**
     * Возвращает номер начального состояния.
     *
     * @return номер начального состояния.
     */
    public int getInitialState() {
        int state = 0;
        int pow = power;
        for (int i = 0; i < QUANTITY_OF_BITS_IN_STATE; i++) {
            if (bits[i]) {
                state += pow;
            }
            pow >>= 1;
        }
        return state;
    }

    /**
     * Возвращает переход.
     *
     * @param numberOfState номер состояния.
     * @param condition условие перехода.
     * @return переход.
     */
    public Automaton.Transition getTransition(int numberOfState,
int condition) {
        return new Transition(getState(numberOfState,
condition), this);
    }

    /**
     * Задает переход.
     *
     * @param numberOfState номер состояния.
     * @param condition условие перехода.
     * @param t новый переход.
     */
    public void setTransition(int numberOfState, int condition,
Automaton.Transition t) {
        setState(numberOfState, t.getEndState(), condition);
    }

    /**
     * Возвращает число состояний.
     *
     * @return число состояний.
     */
    public int getNumberStates() {
        return QUANTITY_OF_STATES;
    }

    /**
     * Возвращает вложенный автомат.
```

```
     *
     * @return вложенный автомат.
     */
    public Automaton getNestedAutomaton() {
        return null;
    }

    /**
     * Переводит состояние в строковый вид.
     *
     * @param i номер состояния.
     * @return строка, описывающая состояние.
     */
    public String getStateString(int i) {
        return (i + 1) + " " + getAction(i);
    }

    /**
     * Задает новое действие.
     *
     * @param numberOfState номер состояния.
     * @param action новое действие.
     */
    public void setAction(int numberOfState, char action) {
        if (numberOfState > QUANTITY_OF_STATES) {
            throw new RuntimeException("Incorrect
numberOfState!");
        }
        int position = 3 * QUANTITY_OF_BITS_IN_STATE + (2 *
QUANTITY_OF_BITS_IN_STATE + QUANTITY_OF_BITS_IN_ACTION) *
numberOfState;
        switch (action) {
            case 'L':
                bits[position] = false;
                bits[position + 1] = true;
                break;
            case 'R':
                bits[position] = true;
                bits[position + 1] = false;
                break;
            case 'M':
                bits[position] = true;
                bits[position + 1] = true;
                break;
            default:
                throw new RuntimeException("Incorrect action!");
        }
    }

    /**
     * Возвращает действие.
     *
     * @param numberOfState номер состояния.
```

```
     * @return действие.
     */
    public char getAction(int numberOfState) {
        if (numberOfState >= QUANTITY_OF_STATES) {
            throw new RuntimeException("Incorrect
numberOfState!");
        }
        int position = 3 * QUANTITY_OF_BITS_IN_STATE + (2 *
QUANTITY_OF_BITS_IN_STATE + QUANTITY_OF_BITS_IN_ACTION) *
numberOfState;
        if (bits[position] && bits[position + 1]) {
            return 'M';
        }
        if (bits[position]) {
            return 'R';
        }
        if (bits[position + 1]) {
            return 'L';
        }
        throw new RuntimeException("Bits are corrupted!");
    }

    /**
     * Задает номер нового состояния для перехода.
     *
     * @param numberOfState номер состояния.
     * @param state новое состояние.
     * @param condition условие перехода.
     */
    public void setState(int numberOfState, int state, int
condition) {
        if (numberOfState >= QUANTITY_OF_STATES || state >=
QUANTITY_OF_STATES) {
            throw new RuntimeException("Incorrect
numberOfState!");
        }
        int position = QUANTITY_OF_BITS_IN_STATE + (2 *
QUANTITY_OF_BITS_IN_STATE + QUANTITY_OF_BITS_IN_ACTION) *
numberOfState;
        if (condition == 1) {
            position += QUANTITY_OF_BITS_IN_STATE;
        }
        int pow = power;
        for (int i = 0; i < QUANTITY_OF_BITS_IN_STATE; i++) {
            if (state / pow == 1) {
                bits[position + i] = true;
                state -= pow;
            } else {
                bits[position + i] = false;
            }
            pow >>= 1;
        }
    }
```

```
    /**
     * Возвращает номер состояния для перехода.
     *
     * @param numberOfState номер состояния.
     * @param condition условие перехода.
     * @return номер состояния для перехода.
     */
    public int getState(int numberOfState, int condition) {
        if (numberOfState >= QUANTITY_OF_STATES) {
            throw new RuntimeException("Incorrect
numberOfState!");
        }
        int state = 0;
        int pow = power;
        int position = QUANTITY_OF_BITS_IN_STATE + (2 *
QUANTITY_OF_BITS_IN_STATE + QUANTITY_OF_BITS_IN_ACTION) *
numberOfState;
        if (condition == 1) {
            position += QUANTITY_OF_BITS_IN_STATE;
        }
        for (int i = 0; i < QUANTITY_OF_BITS_IN_STATE; i++) {
            if (bits[position + i]) {
                state += pow;
            }
            pow >>= 1;
        }
        return state;
    }

    /**
     * Мутация автомата.
     *
     * @param r генератор псевдослучайных чисел.
     * @return мутированный автомат.
     */
    public Individual mutate(Random r) {
        int initState;
        if (r.nextDouble() <= PROBABILITY_OF_MUTATION) {
            initState = r.nextInt(QUANTITY_OF_STATES);
        } else {
            initState = getInitialState();
        }
        BitMooreAutomaton res = new BitMooreAutomaton(initState,
QUANTITY_OF_STATES);
        System.arraycopy(bits, QUANTITY_OF_BITS_IN_STATE,
                res.bits, QUANTITY_OF_BITS_IN_STATE, bits.length
- QUANTITY_OF_BITS_IN_STATE);
        int condition;
        for (int i = 0; i < QUANTITY_OF_STATES; i++) {
            condition = r.nextInt(2);
            if (r.nextDouble() <= PROBABILITY_OF_MUTATION) {
```

```
                res.setState(i, r.nextInt(QUANTITY_OF_STATES),
condition);
            }
            if (r.nextDouble() <= PROBABILITY_OF_MUTATION) {
                switch(r.nextInt(3)) {
                    case 0:
                        res.setAction(i, 'L');
                        break;
                    case 1:
                        res.setAction(i, 'M');
                        break;
                    case 2:
                        res.setAction(i, 'R');
                        break;
                }
            }
        }
        return res;
    }

    /**
     * Кроссовер автомата.
     *
     * @param parent второй предок.
     * @param r генератор псевдослучайных чисел.
     * @return массив из двух новых автоматов.
     */
    public Individual[] crossover(Individual parent, Random r) {
        BitMooreAutomaton p = (BitMooreAutomaton) parent;
        BitMooreAutomaton[] res = new BitMooreAutomaton[2];
        switch (r.nextInt(2)) {
            case 0:
                res[0] = new
BitMooreAutomaton(getInitialState(), QUANTITY_OF_STATES);
                res[1] = new
BitMooreAutomaton(p.getInitialState(), QUANTITY_OF_STATES);
                break;
            case 1:
                res[0] = new
BitMooreAutomaton(p.getInitialState(), QUANTITY_OF_STATES);
                res[1] = new
BitMooreAutomaton(getInitialState(), QUANTITY_OF_STATES);
                break;
        }
        for (int i = 0; i < QUANTITY_OF_STATES; i++) {
            switch (r.nextInt(3)) {
                case 0:
                    res[0].setState(i, getState(i, 1), 1);
                    res[0].setState(i, getState(i, 0), 0);
                    res[0].setAction(i, p.getAction(i));
                    res[1].setState(i, p.getState(i, 1), 1);
                    res[1].setState(i, p.getState(i, 0), 0);
                    res[1].setAction(i, getAction(i));
```

```
                    break;
                case 1:
                    res[0].setState(i, p.getState(i, 1), 1);
                    res[0].setState(i, getState(i, 0), 0);
                    res[0].setAction(i, p.getAction(i));
                    res[1].setState(i, getState(i, 1), 1);
                    res[1].setState(i, p.getState(i, 0), 0);
                    res[1].setAction(i, getAction(i));
                    break;
                case 2:
                    res[0].setState(i, getState(i, 1), 1);
                    res[0].setState(i, p.getState(i, 0), 0);
                    res[0].setAction(i, p.getAction(i));
                    res[1].setState(i, p.getState(i, 1), 1);
                    res[1].setState(i, getState(i, 0), 0);
                    res[1].setAction(i, getAction(i));
                    break;
            }
        }
        return res;
    }

    /**
     * Сравнивает два автомата.
     *
     * @param i второй автомат.
     * @return кто лучше.
     */
    public int compareTo(Individual i) {
        return Double.compare(i.fitness(), fitness());
    }

    /**
     * Функция приспособленности.
     *
     * @return значение приспособленности.
     */
    public double fitness() {
        if (fitness == Double.NEGATIVE_INFINITY) {
            fitness = new SimpleAntTask(null, null,
null).standardFitnessFunction(new SimpleMover(this));
        }
        return fitness;
    }

    /**
     * Возвращает атрибуты.
     *
     * @return атрибуты.
     */
    @Override
    public Object[] getAttributes() {
        return new Object[] {this};
```

```
    }
}
```

BitMooreIndividualFactoryLoader.java

```
package individual.bitMoore;

import individual.bitMoore.factory.BitMooreAutomatonFactory;
import laboratory.common.Loader;
import laboratory.common.ga.IndividualFactory;
import laboratory.util.Parser;

import java.util.jar.JarFile;
import java.util.jar.JarEntry;
import java.util.Properties;
import java.io.IOException;

/**
 * Загружает параметры автомата.
 *
 * @author Yuri Popov
 * @author Dmitry Sokolov
 */
public class BitMooreIndividualFactoryLoader implements
Loader<IndividualFactory> {

    /**
     * Парсер.
     */
    private final Parser properties;

    /**
     * Конструктор загрузчика.
     *
     * @param file
     */
    public BitMooreIndividualFactoryLoader(JarFile file) {
        Properties in = new Properties();
        try {
            JarEntry ent = new
JarEntry("bitMooreAutomaton.conf");
            in.load(file.getInputStream(ent));
        } catch (IOException e) {
            e.printStackTrace();
        }
        properties = new Parser(in);
    }

    /**
     * Создает автомат.
     *
     * @param args параметры.
     * @return автомат.
```

```
     */
    public IndividualFactory load(Object... args) {
        return new
BitMooreAutomatonFactory(properties.getInt("numberOfStates"));
    }

    /**
     * Возвращает свойства.
     *
     * @return свойства.
     */
    public Properties getProperties() {
        return properties.getProperties();
    }
}
```

Файл BitMooreAutomatonFactory.java

```
package individual.bitMoore.factory;

import individual.bitMoore.BitMooreAutomaton;
import laboratory.common.ga.IndividualFactory;
import java.util.Random;

/**
 * Создает псевдослучайный автомат.
 *
 * @author Yuri Popov
 */
public class BitMooreAutomatonFactory implements
IndividualFactory {

    /**
     * Число состояний.
     */
    private final int QUANTITY_OF_STATES;

    /**
     * Генератор псевдослучайных чисел.
     */
    private final Random r;

    /**
     * Конструктор создателя автомата.
     *
     * @param quantityOfStates
     */
    public BitMooreAutomatonFactory(int quantityOfStates) {
        this.QUANTITY_OF_STATES = quantityOfStates;
        r = new Random();
    }

    /**
```

```
     * Создает псевдослучайный автомат.
     *
     * @return псевдослучайный автомат.
     */
    public BitMooreAutomaton randomIndividual() {
        BitMooreAutomaton a = new
BitMooreAutomaton(r.nextInt(QUANTITY_OF_STATES),
QUANTITY_OF_STATES);
        for (int i = 0; i < QUANTITY_OF_STATES; i++) {
            a.setState(i, r.nextInt(QUANTITY_OF_STATES), 1);
            a.setState(i, r.nextInt(QUANTITY_OF_STATES), 0);
            switch (r.nextInt(3)) {
                case 0:
                    a.setAction(i, 'L');
                    break;
                case 1:
                    a.setAction(i, 'R');
                    break;
                case 2:
                    a.setAction(i, 'M');
                    break;
            }
        }
        return a;
    }
}
```

Файл RouletteGA.java

```
package ga.roulette;

import laboratory.common.ga.GA;
import laboratory.common.ga.Individual;
import laboratory.common.ga.IndividualFactory;

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

/**
 * Создание нового поколения методом рулетки.
 *
 * @author Yuri Popov
 */
public class RouletteGA implements GA {

    /**
     * Список особей.
     */
    private List<Individual> generation;
```

```
    /**
     * Вероятность мутации.
     */
    private final double probabilityMutation;

    /**
     * Создатель особей.
     */
    private final IndividualFactory factory;

    /**
     * Генератор псевдослучайных чисел.
     */
    private final Random r;

    /**
     * Создает следующее поколение методом рулетки.
     */
    public void nextGeneration() {
        int size = generation.size();
        List<Individual> interGeneration = new
ArrayList<Individual>(size);
        double roulette[] = new double[size + 1];
        int sumFitness = 0;
        for (int i = 0; i < size; i++) {
            sumFitness += generation.get(i).fitness();
        }
        double tmp = 0;
        for (int i = 0; i < size; i++) {
            tmp += generation.get(i).fitness();
            roulette[i + 1] = tmp / sumFitness;
        }
        double arrow;
        int left, right, middle;
        for (int i = 0; i < size; i++) {
            arrow = r.nextDouble();
            left = 0;
            right = size;
            while (right - left > 1) {
                middle = (left + right) / 2;
                if (arrow < roulette[middle]) {
                    right = middle;
                } else {
                    left = middle;
                }
            }
            interGeneration.add(generation.get(right - 1));
        }
        List<Individual> newGeneration = new
ArrayList<Individual>(size);
        for (int i = 0; i < size / 2; i++) {
            Individual a1, a2;
            a1 = interGeneration.get(r.nextInt(size));
```

```
            a2 = interGeneration.get(r.nextInt(size));
            Individual p = (a1.compareTo(a2) < 0) ? a1 : a2;
            a1 = interGeneration.get(r.nextInt(size));
            a2 = interGeneration.get(r.nextInt(size));
            Individual[] s = p.crossover((a1.compareTo(a2) < 0)
? a1 : a2, r);
            newGeneration.add(s[0]);
            newGeneration.add(s[1]);
        }
        if (newGeneration.size() < size) {

newGeneration.add(interGeneration.get(r.nextInt(size)).mutate(r)
);
        }
        for (int i = 0; i < size; i++) {
            if (r.nextDouble() < probabilityMutation) {
                newGeneration.set(i,
newGeneration.get(i).mutate(r));
            }
        }
        generation = newGeneration;
        Collections.sort(generation);
    }

    /**
     * Возвращает поколение.
     *
     * @return поколение.
     */
    public List<Individual> getGeneration() {
        return generation;
    }

    /**
     * Конструктор, создающий поколение.
     *
     * @param sizeGeneration число особей.
     * @param factory создатель особей.
     */
    public RouletteGA(int sizeGeneration, double
probabilityMutation, IndividualFactory factory) {
        this.probabilityMutation = probabilityMutation;
        generation = new ArrayList<Individual>(sizeGeneration);
        for (int i = 0; i < sizeGeneration; i++) {
            generation.add(factory.randomIndividual());
        }
        Collections.sort(generation);
        this.factory = factory;
        r = new Random();
    }

    /**
     * Создает псевдослучайное поколение.
```

```
     */
    public void bigMutation() {
        for (int i = 0; i < generation.size(); i++) {
            generation.set(i, factory.randomIndividual());
        }
        Collections.sort(generation);
    }

    /**
     * Возвращает лучшую особь.
     *
     * @return лучшая особь.
     */
    public Individual getBest() {
        return generation.get(0);
    }
}
```

Файл RouletteGALoader.java

```
package ga.roulette;

import laboratory.common.Loader;
import laboratory.common.ga.GA;
import laboratory.common.ga.IndividualFactory;
import laboratory.util.Parser;

import java.io.IOException;
import java.util.Properties;
import java.util.jar.JarEntry;
import java.util.jar.JarFile;

/**
 * Создает генератор поколений.
 *
 * @author Yuri Popov
 */
public class RouletteGALoader implements Loader<GA> {

    /**
     * Парсер.
     */
    private final Parser properties;

    /**
     * Загружает число особей, создает генератор поколений.
     *
     * @param args аргументы.
     * @return генератор поколения.
     */
    public GA load(Object... args) {
```

```
        return new
RouletteGA(properties.getInt("sizeGeneration"),
                properties.getDouble("probabilityMutation"),
(IndividualFactory) args[0]);
    }

    /**
     * Конструктор, создающий парсер.
     *
     * @param file jar file.
     */
    public RouletteGALoader(JarFile file) {
        Properties in = new Properties();
        try {
            JarEntry ent = new JarEntry("rouletteGA.conf");
            in.load(file.getInputStream(ent));
        } catch (IOException e) {
            e.printStackTrace();
        }
        properties = new Parser(in);
    }

    /**
     * Возвращает свойства.
     *
     * @return свойства.
     */
    public Properties getProperties() {
        return properties.getProperties();
    }
}
```