Санкт-Петербургский государственный университет
информационных технологий, механики и оптики

Факультет информационных технологий и программирования

Кафедра «Компьютерные технологии»

А.А. Терескин

Отчет по лабораторной работе «Использование генетических алгоритмов для построения управляющих автоматов»

Вариант №1

Санкт-Петербург
2009

Оглавление

Введение

Цель лабораторной работы – применение генетических алгоритмов для построения конечных автоматов. В качестве примера рассматривается построение конечного автомата Мили, решающего задачу об «Умном муравье».

При выполнении лабораторной работы использовалась программа «Виртуальная лаборатория» [1], написанная студентами кафедры «Компьютерные технологии» СПбГУ ИТМО и позволяющая реализовывать генетические алгоритмы и особи для них в виде модулей.

1. Постановка задачи

Задача данной лабораторной работы – построить автомат Мили, решающий задачу об «Умном муравье». При построении следует стремиться к тому, чтобы автомат был близок к оптимальному. Критерий оценки решения заключается в том, что автомат, имея фиксированное число состояний, должен приводить к тому, что муравей, управляемый автоматом, съедает всю еду на поле за ограниченное число шагов.

1.1. Автомат Мили

Автомат типа Мили – конечный автомат, который генерирует выходные действия в зависимости от текущего состояния и входного сигнала. Пример представления автомата Мили в виде диаграммы переходов приведен на рисунке (рис. 1).

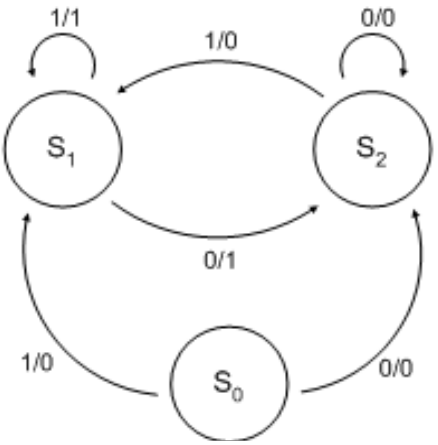

Рис. 1

Над каждой дугой расположена пара значений – входное действие и выходное действие. При этом выходное действие зависит от состояния автомата и от входного воздействия.

1.2. Задача об «Умном муравье»

Игра происходит на поверхности тора размером 32x32 клетки. В некоторых клетках находится еда. Муравей начинает движение из клетки, помеченной «Start».

За ход муравей может выполнить следующие действия:

- повернуть налево;
- повернуть направо;
- сделать шаг вперед и, если в новой клетке есть еда, съесть её;
- ничего не делать.

Игра длится 200 ходов. Цель игры – создать муравья «с минимальным числом состояний», который за минимальное число ходов ест как можно больше яблок.

2. Реализация

Виртуальная лаборатория состоит из ядра и подключаемых модулей. Для решения поставленной задачи требуется создать два модуля. Первый модуль – модуль генетического алгоритма, использующий традиционный генетический алгоритм и метод рулетки для генерации очередного поколения. Второй модуль должен реализовывать «особь» – конечный автомат Мили, решающий задачу об «Умном муравье». Для «особи» необходимо реализовать операции мутации и скрещивания.

2.1. Представление автоматов

Каждый автомат представляется в виде битовой строки, реализованной на основе класса `BitSet` из стандартной библиотеки *Java*. Представление автомата из N состояний изображено на рисунке (рис. 2).

Рис. 2

В описании каждого состояния для каждого из двух входных воздействий («есть еда», «нет еды») закодировано сначала состояние, куда будет переход, затем действие при переходе. На рис. 3 представлен пример кодирования переходов из состояния. В данном случае при входном воздействии «есть еда» автомат переходит в третье состояние, совершая при этом поворот налево (действие L), при входном воздействии «нет еды» – переход в седьмое состояние с шагом вперед (действие M).

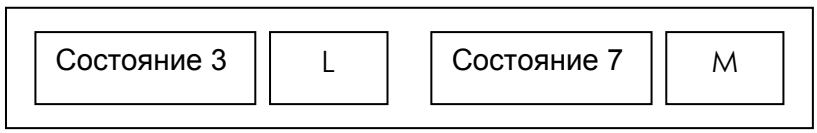

Рис. 3

Выходное действие, совершаемое автоматом при переходе, кодируется двумя битами (рис. 4).

Рис. 4

Эти два бита представляют действие следующим образом: 01 – поворот налево (L), 10 – шаг вперед (M), 11 – поворот направо (R).

Входное воздействие – это состояние поля, которое видит муравей. Муравей видит одну клетку впереди себя. На рис. 5 изображен пример кодирования нового состояния после перехода.

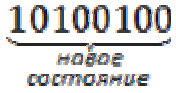

Рис. 5

Эти восемь бит являются двоичным представлением номера нового состояния (от 0 до 255). Состояния конечного автомата нумеруются, начиная с нуля. Число бит для представления номера состояния вычисляется как наименьшая степень двойки большая или равная числу состояний в автомате (число бит в представлении состояния с наибольшим номером).

2.2. Метод скрещивания

Оператор скрещивания получает на вход две особи и выдает также две особи. Число автоматов фиксировано и одинаково для всех особей. Процесс скрещивания происходит следующим образом:

1. Создаём два новых автомата с тем же число состояний в каждом.
2. Случайным образом либо первый сын получает номер начального состояния от первого родителя, второй – от второго, либо первый – от второго родителя, второй – от первого.
3. Рассмотрим состояние с номером **i** в каждом родителе. Для каждого состояния есть два элемента **Transition** (индексы **[i][0]** и **[i][1]**). Таким образом, от двух родителей получаем четыре элемента **Transition**: **p10** – нулевой элемент первого родителя, **p11** – первый элемент первого родителя, **p20** – нулевой элемент второго родителя, **p21** – первый элемент второго родителя. Аналогично для того же **i** для детей имеем: **s10, s11, s20, s21**. Теперь выбираем случайно один набор из четырёх: **(p10, p01, p00, p11) (p00, p11, p10, p01) (p10, p11, p00, p01) (p00, p01, p10, p11)**, и присваиваем соответствующие **Transitions** родителей соответственно **s00, s01, s10, s11**.
4. Повторяем шаг 3 для всех состояний.

2.3. Метод мутации

С заданной вероятностью для какого-то состояния меняем действие при переходе в другое состояние, определяемое случайным образом. При мутации с вероятностью 20% может измениться начальное состояние автомата.

2.4. Метод генерации очередного поколения

Начальное поколение состоит из фиксированного числа случайно сгенерированных автоматов. Все автоматы в поколении имеют одинаковое наперед заданное число состояний.

Для генерации очередного поколения используется традиционный генетический алгоритм и метод рулетки.

Шаг алгоритма состоит из трех стадий: генерация промежуточной популяции путем отбора текущего поколения, скрещивание особей промежуточной популяции путем кроссовера. Это приводит к формированию нового поколения, далее происходит мутация нового поколения.

Промежуточная популяция — это набор особей, которые получили право размножаться. Приспособленные особи могут быть записаны туда несколько раз. «Плохие» особи с большой вероятностью туда вообще не попадут.

В классическом генетическом алгоритме вероятность каждой особи попасть в промежуточную популяцию пропорциональна ее приспособленности – работает пропорциональный отбор. В данной работе он реализован следующим образом: пусть особи располагаются на колесе рулетки, так что размер сектора каждой особи пропорционален ее приспособленности. В начале промежуточная популяция пуста. N раз запуская рулетку, выберем требуемое число особей для записи в промежуточную популяцию. Ни одна выбранная особь не удаляется с рулетки.

На этапе скрещивания особей между собой используется элитизм. Определенная часть всей популяции особей не скрещивается, а сразу отправляется в следующее поколение. Таким образом, удается добиться более стабильной работы генетического алгоритма.

2.5. Способ вычисления функции приспособленности

Эмулируем поведение муравья до тех пор, пока он, либо не съест всю еду, либо не превысит допустимое число шагов. Далее значение функции приспособленности (***Fitness***) считается по формуле:

$$Fitness = Apples - Steps / 200,$$

Apples – число съеденных муравьем яблок за 200 шагов, ***Steps*** – ход, на котором муравей съел последний раз яблоко.

3. Результаты работы модуля

В результате работы генетического алгоритма был получен автомат Мили, который решает задачу об «Умном муравье». Автомат состоит из пяти состояний, значение функции приспособленности равно 82,05. Муравей, управляемый данным автоматом, съедает 83 яблока за 195 шагов.

3.1. Графики максимального значения функции приспособленности

На рис. 6 приведен график зависимости максимального значения функции приспособленности среди особей поколения.

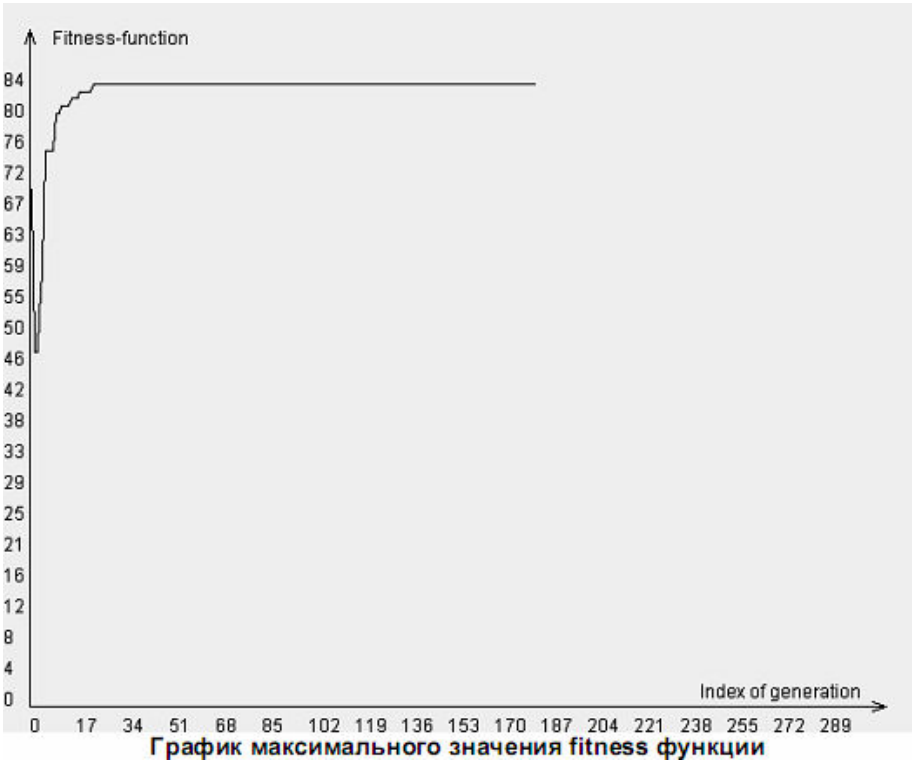

Рис. 6

3.2. Графики среднего значения функции приспособленности

На рис. 7 приведен график зависимости среднего значения функции приспособленности среди особей поколения.

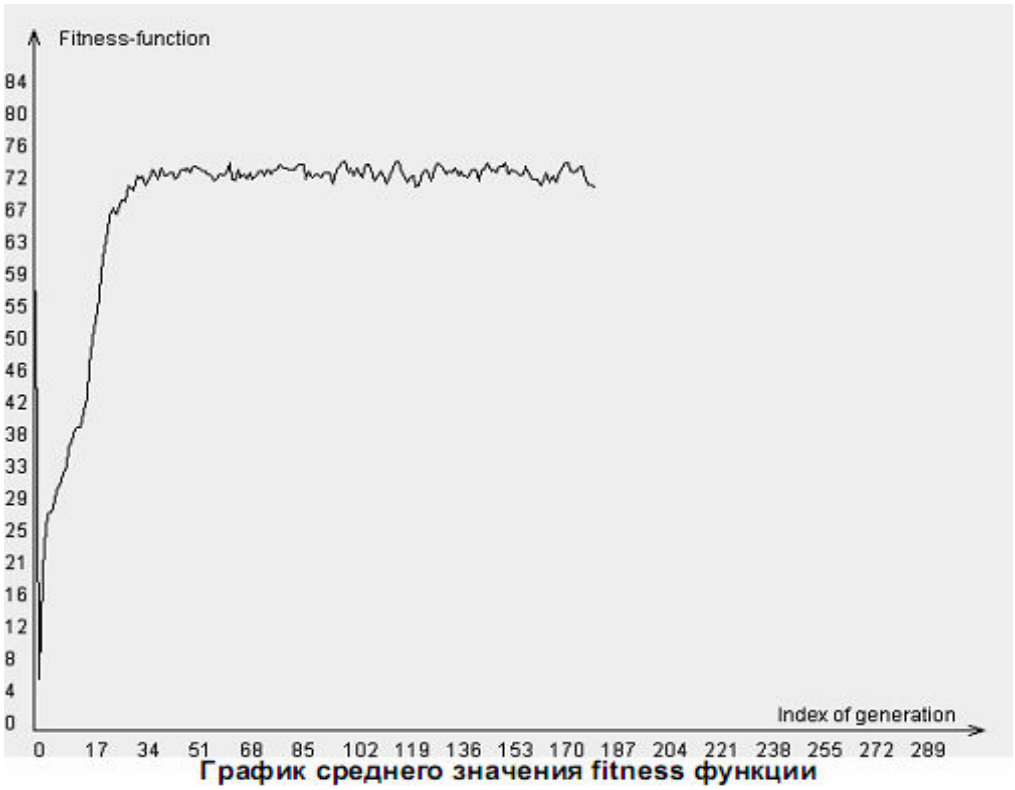

Рис. 7

3.3. Граф переходов полученного автомата

Полученный конечный автомат Мили представлен в виде графа переходов. Вершины графа – состояния автомата. Ребра графа – переходы между состояниями. На ребрах написано входное воздействие и действие при переходе.

Входное воздействие может быть:

1. F – «впереди пусто»;
2. T – «впереди есть еда».

Действие при переходе может иметь одно из следующих значений:

1. L — поворот налево;
2. M — шаг вперед;
3. R — поворот направо.

Число съедаемых яблок – 83

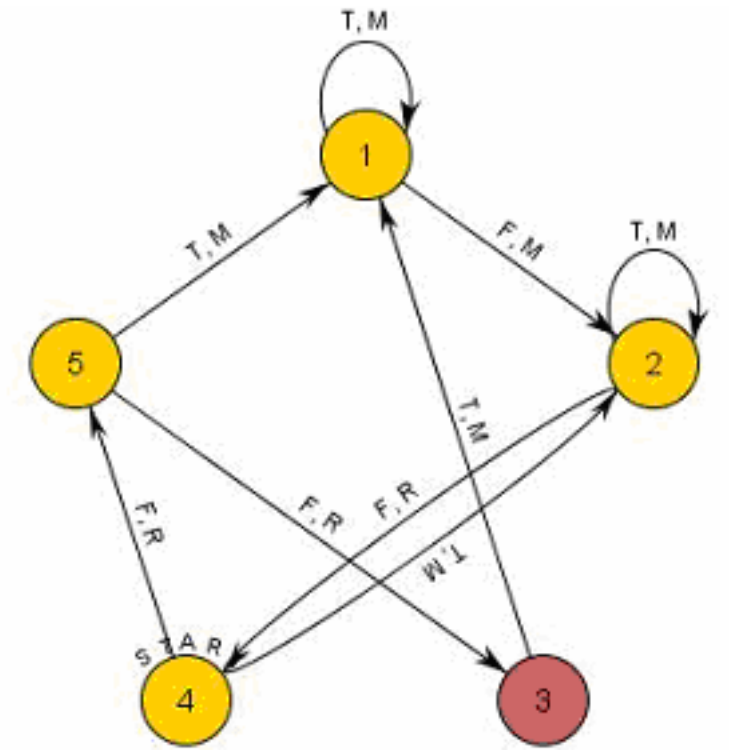

Рис. 8

Заключение

Результаты лабораторной работы показали, что используемые методы достаточно эффективны для построения автомата Мили с пятью состояниями, который решает задачу об «Умном муравье», съедая 83 яблока, так как полный перебор позволяет строить автомат с таким же результатом. При этом стоит заметить, что известен автомат, построенный для этой задачи вручную, который управляет муравьём, съедающим 81 яблоко [2].

Источники

1. Инструкция по созданию plugin’ов к виртуальной лаборатории. http://svn2.assembla.com/svn/not_instrumental_tool/docs/pdf/interface_manual.pdf

2. *Бедный Ю.Д., Шалыто А.А.* Применение генетических алгоритмов для построения автоматов в задаче «Умный муравей». http://is.ifmo.ru/works/_ant.pdf

3. *Яминов Б.* Генетические алгоритмы. http://rain.ifmo.ru/cat/view.php/theory/unsorted/genetic-2005

Приложение

Конфигурационные файлы

Для сборки модулей виртуальной лаборатории был создан ant-скрипт, который находится в файле `build.xml`. Файл `build.properties` содержит параметры для этого скрипта.

Модуль «особи»:

1. `automaton.conf` — конфигурация задания.
    - `external` — количество состояний автомата.
2. `plugin.properties` — конфигурация модуля.

Модуль генетического алгоритма:

1. `rouletteGA.conf` — конфигурация клеточного генетического алгоритма.
    - `sizeGeneration` — количество особей в поколении.
    - `sizeElite` — количество «элитных» особей в поколении.
    - `probabilityMutation` — вероятность мутации.
2. `plugin.properties` — конфигурация модуля.

Исходные тексты программ

Модуль особи:

1. `Task1IndividualFactoryLoader.java` — класс, реализующий загрузку модуля в виртуальную лабораторию.
2. `Task1Automaton.java` — класс, реализующий работу с конечным автоматом Мили.
3. `BitArray.java` — класс, реализующий работу с представлением автомата в виде битовой строки.
4. `Task1AutomatonFactory.java` — класс, реализующий создание произвольной особи.

Модуль генетического алгоритма:

1. `RouletteGA.java` – класс, реализующий клеточный генетический алгоритм.
2. `RouletteGALoader.java` – класс, реализующий загрузку модуля в виртуальную лабораторию.

Листинг Task1IndividualFactoryLoader.java

```
package individual.task1;

import individual.task1.factory.Task1AutomatonFactory;

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;

public class Task1IndividualFactoryLoader implements
Loader<IndividualFactory> {

    private final Parser properties;
```

```
    public Task1IndividualFactoryLoader(JarFile file) {
        Properties in = new Properties();
        try {
            JarEntry ent = new JarEntry("task1Automaton.conf");
            in.load(file.getInputStream(ent));
        } catch (IOException e) {
            e.printStackTrace();
        }
        properties = new Parser(in);
    }
    public IndividualFactory load(Object... args){
        return new
Task1AutomatonFactory(properties.getInt("external"));
    }
    public Properties getProperties() {
        return properties.getProperties();
    }
}
```

Листинг Task1Automaton.java

```
package individual.task1;
import laboratory.common.ga.Individual;
import task.ant.simple.individual.Automaton;
import task.ant.simple.Ant;
import java.util.Random;
import task.ant.simple.SimpleAnt;
import task.ant.simple.SimpleAntTask;
import task.ant.simple.individual.SimpleMover;
public class Task1Automaton implements Automaton {
```

```
    private final int numStates;
    private BitArray bitArr;
    private double fitness = Double.NEGATIVE_INFINITY;
    public Task1Automaton(int initState, int numStates)   {
        this.numStates = numStates;
        this.bitArr = new BitArray(numStates, 2);
        setInitialState(initState);
    }
    private void setInitialState(int is) {
        bitArr.setInitialState(is);
    }
    public int getInitialState() {
        return bitArr.getInitialState();
    }
    public Transition getTransition(int index, int condition) {
        char act = bitArr.getAction(index, condition);
        int nextState = bitArr.getNextState(index,
condition)%numStates;
        return new Task1Automaton.Transition(nextState, act);
    }
    public void setTransition(int index, int condition,
Automaton.Transition t) {
        bitArr.setAction(index, condition, t.getAction());
        bitArr.setNextState(index, condition, t.getEndState());
    }
    public int getNumberStates() {
        return numStates;
    }
    public Automaton getNestedAutomaton() {
        return null;
```

```
    }
    public String getStateString(int i) {
        return (i+1)+"";
    }
    public Object[] getAttributes() {
        return new Object[] {this};
    }
    public double fitness() {
        //if (fitness == Double.NEGATIVE_INFINITY)
        //    fitness = new
SimpleAntTask(null,null,null).standardFitnessFunction(new
SimpleMover(this));
        //return fitness;
        if (fitness == Double.NEGATIVE_INFINITY) {
            SimpleMover mover = new SimpleMover(this);
            mover.restart(new SimpleAnt());
            int count = 0;
            int lem = 0;
            for (int i = 0; i < Ant.NUMBER_STEPS; i++) {
                if (mover.move()) {
                    count++;
                    lem = i;
                }
                if (count == Ant.NUMBER_FOOD) {
                    break;
                }
            }
            fitness = (count - lem * 1.0 / Ant.NUMBER_STEPS);
        }
        return fitness;
```

```
    }
    public Individual mutate(Random r) {
        int state = r.nextInt(numStates);
        Task1Automaton mutant = new
Task1Automaton(getInitialState(), numStates);
        mutant.bitArr = this.bitArr.clone();
        int c = r.nextBoolean() ? 1 : 0;
        int es = r.nextInt(numStates);
        mutant.setTransition(state, c, new Transition(es,
Ant.ACTION_VALUES[r.nextInt(Ant.ACTION_VALUES.length)]));
        if (r.nextDouble() < 0.2)
            mutant.setInitialState(r.nextInt(numStates));
        return mutant;
    }
    public Individual[] crossover(Individual p, Random r) {
        Task1Automaton[] s = new Task1Automaton[2];
        for (int i = 0; i < 2; i++) {
            s[i] = new Task1Automaton(0, getNumberStates());
        }

        Task1Automaton pp = (Task1Automaton)p;
        if (r.nextBoolean()) {
            s[0].setInitialState(pp.getInitialState());
            s[1].setInitialState(getInitialState());
        } else {
            s[1].setInitialState(pp.getInitialState());
            s[0].setInitialState(getInitialState());
        }
        for (int i = 0; i < getNumberStates(); i++) {
            int flag = r.nextInt(4);
```

```
            switch (flag) {
                case 0:
                    s[0].bitArr.setNextState(i, 0,
pp.bitArr.getNextState(i, 0));
                    s[0].bitArr.setAction(i, 0,
pp.bitArr.getAction(i, 0));
                    s[0].bitArr.setNextState(i, 1,
pp.bitArr.getNextState(i, 1));
                    s[0].bitArr.setAction(i, 1,
pp.bitArr.getAction(i, 1));
                    s[1].bitArr.setNextState(i, 0,
bitArr.getNextState(i, 0));
                    s[1].bitArr.setAction(i, 0,
bitArr.getAction(i, 0));
                    s[1].bitArr.setNextState(i, 1,
bitArr.getNextState(i, 1));
                    s[1].bitArr.setAction(i, 1,
bitArr.getAction(i, 1));
                    break;
                case 1:
                    s[0].bitArr.setNextState(i, 0,
bitArr.getNextState(i, 0));
                    s[0].bitArr.setAction(i, 0,
bitArr.getAction(i, 0));
                    s[0].bitArr.setNextState(i, 1,
bitArr.getNextState(i, 1));
                    s[0].bitArr.setAction(i, 1,
bitArr.getAction(i, 1));
                    s[1].bitArr.setNextState(i, 0,
pp.bitArr.getNextState(i, 0));
                    s[1].bitArr.setAction(i, 0,
pp.bitArr.getAction(i, 0));
                    s[1].bitArr.setNextState(i, 1,
pp.bitArr.getNextState(i, 1));
                    s[1].bitArr.setAction(i, 1,
pp.bitArr.getAction(i, 1));
```

```
                    break;

                case 2:

                    s[0].bitArr.setNextState(i, 0,
pp.bitArr.getNextState(i, 0));

                    s[0].bitArr.setAction(i, 0,
pp.bitArr.getAction(i, 0));

                    s[0].bitArr.setNextState(i, 1,
bitArr.getNextState(i, 1));

                    s[0].bitArr.setAction(i, 1,
bitArr.getAction(i, 1));

                    s[1].bitArr.setNextState(i, 0,
pp.bitArr.getNextState(i, 0));

                    s[1].bitArr.setAction(i, 0,
pp.bitArr.getAction(i, 0));

                    s[1].bitArr.setNextState(i, 1,
bitArr.getNextState(i, 1));

                    s[1].bitArr.setAction(i, 1,
bitArr.getAction(i, 1));

                    break;

                case 3:

                    s[0].bitArr.setNextState(i, 0,
bitArr.getNextState(i, 0));

                    s[0].bitArr.setAction(i, 0,
bitArr.getAction(i, 0));

                    s[0].bitArr.setNextState(i, 1,
pp.bitArr.getNextState(i, 1));

                    s[0].bitArr.setAction(i, 1,
pp.bitArr.getAction(i, 1));

                    s[1].bitArr.setNextState(i, 0,
bitArr.getNextState(i, 0));

                    s[1].bitArr.setAction(i, 0,
bitArr.getAction(i, 0));

                    s[1].bitArr.setNextState(i, 1,
pp.bitArr.getNextState(i, 1));

                    s[1].bitArr.setAction(i, 1,
pp.bitArr.getAction(i, 1));
```

```
                    break;
                }
            }
            return s;
        }
        public int compareTo(Individual o) {
            return Double.compare(o.fitness(), fitness());
        }
        public static class Transition implements
    Automaton.Transition {
            private final char action;
            private final int endState;
            public Transition(int endState, char action) {
                this.endState = endState;
                this.action = action;
            }
            public char getAction() {
                return action;
            }
            public String toString() {
                return "" + action;
            }
            public int getEndState() {
                return endState;
            }
        }
    }
```

Листинг BitArray.java

```
package individual.task1;

import java.util.BitSet;

import task.ant.simple.Ant;

public class BitArray {

    public static int NUMBER_LENGTH = 8;

    public static int ACTION_LENGTH = 2;

    private final int countInput;

    private final int countStates;

    public BitSet bitarray;

    public BitArray(int countStates, int countInput) {

        this.countInput = countInput;

        this.countStates = countStates;

        int pow = 0;

        while(countStates > 0) {

            countStates = countStates >> 1;

            pow++;

        }

        NUMBER_LENGTH = pow;

        bitarray = new BitSet(countStates * (NUMBER_LENGTH +
ACTION_LENGTH) * countInput + NUMBER_LENGTH);

    }

    @Override

    public BitArray clone() {

        BitArray copy = new BitArray(this.countStates,
this.countInput);

        copy.bitarray = (BitSet)this.bitarray.clone();

        return copy;

    }
```

```
    public void setAction(int stateNumber, int inputNumber, char
action) {
        int begin = stateNumber * (NUMBER_LENGTH +
ACTION_LENGTH) * countInput;
        begin += NUMBER_LENGTH;
        begin += NUMBER_LENGTH;
        begin += (NUMBER_LENGTH + ACTION_LENGTH) * inputNumber;
        if (action == 'L') {
            bitarray.set(begin, false);
            bitarray.set(begin + 1, true);
        } else if (action == 'M') {
            bitarray.set(begin, true);
            bitarray.set(begin + 1, false);
        } else if (action == 'R') {
            bitarray.set(begin, true);
            bitarray.set(begin + 1, true);
        } else {
            bitarray.set(begin, false);
            bitarray.set(begin + 1, false);
        }
    }
    public void setInitialState(int initState) {
        int begin = 0;
        for (int i = 0; i < NUMBER_LENGTH; i++) {
            if ((initState & 1) == 1) {
                bitarray.set(begin + i, true);
            } else {
                bitarray.set(begin + i, false);
            }
            initState = initState >> 1;
```

```
        }
    }
    public int getInitialState() {
        int power = 1;
        int initSt = 0;
        int begin = 0;
        for (int i = 0; i < NUMBER_LENGTH; i++) {
            if (bitarray.get(begin + i)) {
                initSt += power;
            }
            power *= 2;
        }
        return initSt;
    }
    public void setNextState(int stateNumber, int inputNumber,
int nextState) {
        int begin = stateNumber * (NUMBER_LENGTH +
ACTION_LENGTH) * countInput;
        begin += NUMBER_LENGTH;
        begin += (NUMBER_LENGTH + ACTION_LENGTH) * inputNumber;
        for (int i = 0; i < NUMBER_LENGTH; i++) {
            if ((nextState & 1) == 1) {
                bitarray.set(begin + i, true);
            } else {
                bitarray.set(begin + i, false);
            }
            nextState = nextState >> 1;
        }
    }
```

```
    public char getAction(int stateNumber, int inputNumber) {
        int begin = stateNumber * (NUMBER_LENGTH +
ACTION_LENGTH) * countInput;
        begin += NUMBER_LENGTH;
        begin += NUMBER_LENGTH;
        begin += (NUMBER_LENGTH + ACTION_LENGTH) * inputNumber;
        if (bitarray.get(begin)) {
            if (bitarray.get(begin + 1)) {
                return 'R';
            } else {
                return 'M';
            }
        } else {
            if (bitarray.get(begin + 1)) {
                return 'L';
            } else {
                return ' ';
            }
        }
    }
    public int getNextState(int stateNumber, int inputNumber) {
        int nextState = 0;
        int power = 1;
        int begin = stateNumber * (NUMBER_LENGTH +
ACTION_LENGTH) * countInput;
        begin += NUMBER_LENGTH;
        begin += (NUMBER_LENGTH + ACTION_LENGTH) * inputNumber;
        for (int i = 0; i < NUMBER_LENGTH; i++) {
            if (bitarray.get(begin + i)) {
                nextState += power;
```

```
            }
            power *= 2;
        }
        return nextState;
    }
}
```

Листинг Task1AutomatonFactory.java

```
package individual.task1.factory;
import individual.task1.Task1Automaton;
import task.ant.simple.Ant;
import task.ant.simple.individual.factory.AutomatonFactory;
public class Task1AutomatonFactory extends
AutomatonFactory<Task1Automaton>{
    public Task1AutomatonFactory(int numStates) {
        super(numStates, 0);
    }

    protected Task1Automaton fullRandomAutomaton(int numStates)
{
        Task1Automaton a = new
Task1Automaton(r.nextInt(numStates), numStates);
        for (int i = 0; i < numStates; i++) {
            a.setTransition(i, 1, new
Task1Automaton.Transition(r.nextInt(numStates),
Ant.ACTION_VALUES[r.nextInt(3)]));
            a.setTransition(i, 0, new
Task1Automaton.Transition(r.nextInt(numStates),
Ant.ACTION_VALUES[r.nextInt(3)]));
        }
        return a;
    }
    protected Task1Automaton randomAutomatonWN(int numStates,
Task1Automaton na) {
```

```
        return fullRandomAutomaton(numStates);
    }
}
```

Листинг 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;
public class RouletteGA implements GA {
    private List<Individual> generation;
    private final double probabilityMutation;
    private final int sizeElite;
    private final IndividualFactory factory;
    private final Random r;
    public void nextGeneration() {
        int size = generation.size();
        List<Individual> intGeneration = new
ArrayList<Individual>(size);
        double roulette[] = new double[size];
        double totalFitness = 0.0;
        for (Individual individual : this.generation) {
          totalFitness += individual.fitness();
        }
        roulette[0] = generation.get(0).fitness() /
totalFitness;
        for (int i = 1; i < size; i++) {
```

```
            roulette[i] = roulette[i - 1] +
generation.get(i).fitness() / totalFitness;
        }
        double arrow;
        for (int i = 0; i < size; i++) {
            arrow = r.nextDouble();
            for (int j = 0; j < size; j++) {
                if (arrow <= roulette[j]) {
                    intGeneration.add(generation.get(j));
                    break;
                }
            }
        }
        List<Individual> newGeneration = new
ArrayList<Individual>(size);
        for (int i = 0; i < size; i += 2) {
          Individual father = intGeneration.get(i);
          Individual mother = intGeneration.get(i + 1);
            Individual[] s = father.crossover(mother, this.r);
            newGeneration.add(s[0]);
            newGeneration.add(s[1]);
        }
        if (newGeneration.size() < size) {

newGeneration.add(intGeneration.get(r.nextInt(size)).mutate(r));
        }
        for (int i = 0; i < size; i++) {
            if (this.r.nextDouble() < this.probabilityMutation)
{
                newGeneration.set(i,
newGeneration.get(i).mutate(this.r));
            }
```

```
        }
        Collections.sort(intGeneration);
        for (int i = 0; i < sizeElite; i++) {
            newGeneration.set(i, intGeneration.get(i));
        }
        this.generation = newGeneration;
        Collections.sort(generation);
    }
    public List<Individual> getGeneration() {
        return this.generation;
    }
    public RouletteGA(int sizeGeneration, double sizeElite,
double probabilityMutation, IndividualFactory factory) {
        this.probabilityMutation = probabilityMutation;
        this.sizeElite = (int) (sizeElite * sizeGeneration);
        // Creating new generation
        this.generation = new
ArrayList<Individual>(sizeGeneration);
        for (int i = 0; i < sizeGeneration; i++) {
            this.generation.add(factory.randomIndividual());
        }
        Collections.sort(this.generation);
        this.factory = factory;
        this.r = new Random();
    }
    public void bigMutation() {
        for (int i = 0; i < this.generation.size(); i++) {
            this.generation.set(i,
this.factory.randomIndividual());
        }
        Collections.sort(this.generation);
```

```
    }
    public Individual getBest() {
        return this.generation.get(0);
    }
}
```

Листинг 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;
public class RouletteGALoader implements Loader<GA> {
    private final Parser properties;
    public GA load(Object... args) {
        return new
RouletteGA(this.properties.getInt("sizeGeneration"),
                properties.getDouble("sizeElite"),
properties.getDouble("probabilityMutation"),
                (IndividualFactory) args[0]);
    }
    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();
        }
        this.properties = new Parser(in);
    }
    public Properties getProperties() {
        return this.properties.getProperties();
    }
}
```