Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Факультет информационных технологий и программирования

Кафедра «Компьютерные технологии»

А.Л. Новохатько

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

Вариант № 3

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 воздействия в зависимости от текущего состояния и входного воздействия. Пример представления автомата Мили в виде диаграммы переходов приведен на рисунке (рис. 1).

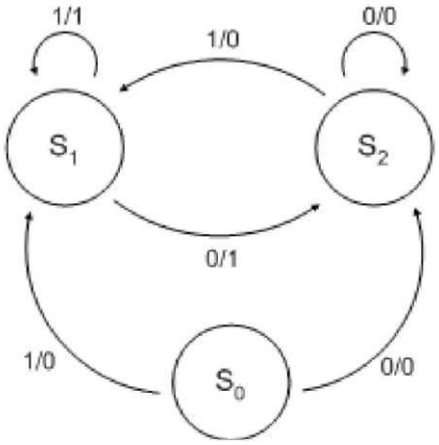

Рис. 1

Над каждой дугой расположена пара значений – входное и выходное воздействия. При этом выходное воздействие 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 ее;
- ничего не делать.

Игра длится 200 ходов. Цель игры – создать муравья, который за минимальное число ходов съест 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 островной 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 рис. 2.

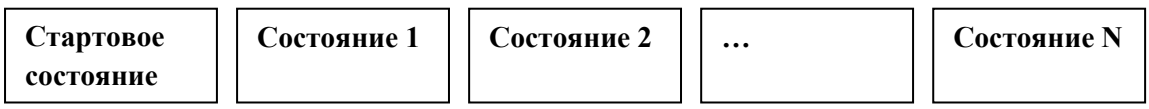

Рис. 2

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`).

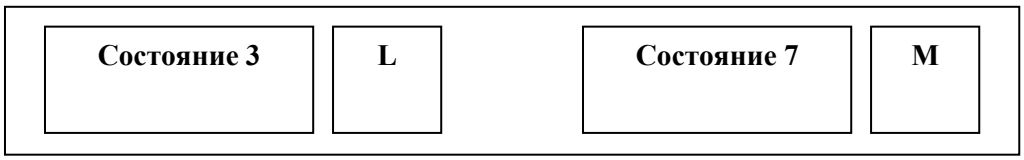

Рис. 3

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

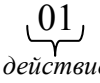

Рис. 4

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.

$$\underbrace{10100100}_{\textit{новое состояние}}$$

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. Создаем 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 четырех: **(p10, p01, p00, p11) (p00, p11, p10, p01) (p10, p11, p00, p01) (p00, p01, p10, p11)**, и присваиваем соответствующие выбранному набору **Transitions** родителей.
4. Повторяем шаг 3 для всех состояний.

2.3. Метод мутации

При мутации с некоторой вероятностью стартовым состоянием становится случайное состояние из созданного автомата. Затем изменяется либо выходное воздействие этого состояния, либо переход по какому-либо входному воздействию.

2.4. Модель генетического алгоритма

Островная модель [3] – это модель параллельного генетического алгоритма. Она заключается в следующем: пусть имеются 16 процессов и 1600 особей. Разобьем их на 16 подпопуляций по 100 особей. Каждая их них будет развиваться отдельно с помощью некого генетического алгоритма. Таким образом, можно сказать, что особи расселены по 16-ти изолированным островам.

Изредка (например, каждые пять поколений) процессы (или острова) будут обмениваться несколькими хорошими особями. Это называется миграция. Она позволяет островам обмениваться генетическим материалом (рис. 6).

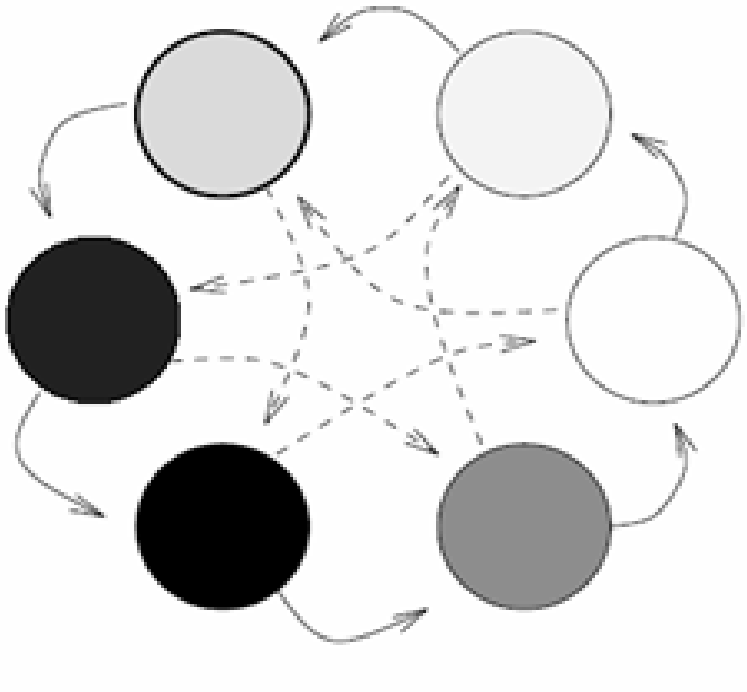

Рис. 6

Так как населенность островов обычно бывает невелика, подпопуляции будут склонны к преждевременной сходимости. Поэтому важно правильно установить частоту миграции. Чересчур частая миграция (или миграция слишком большого числа особей) приведет к смешению всех подпопуляций, и тогда островная модель будет несильно отличаться от обычного генетического

алгоритма. Если же миграция будет слишком редкой, то она не сможет предотвратить преждевременного схождения подпопуляций.

2.5. 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 островной 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. После этого 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. После этого `N` раз, запуская рулетку, выберем требуемое число особей для записи в промежуточную популяцию. Ни одна выбранная особь не удаляется с рулетки.

Рулеточный алгоритм подставляется в качестве базового для островной модели. Эта модель используется в данном модуле.

2.6. 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 82,04. Муравей, управляемый данным автоматом, съедает 83 яблока за 192 шага.

3.1. График максимального значения функции приспособленности

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

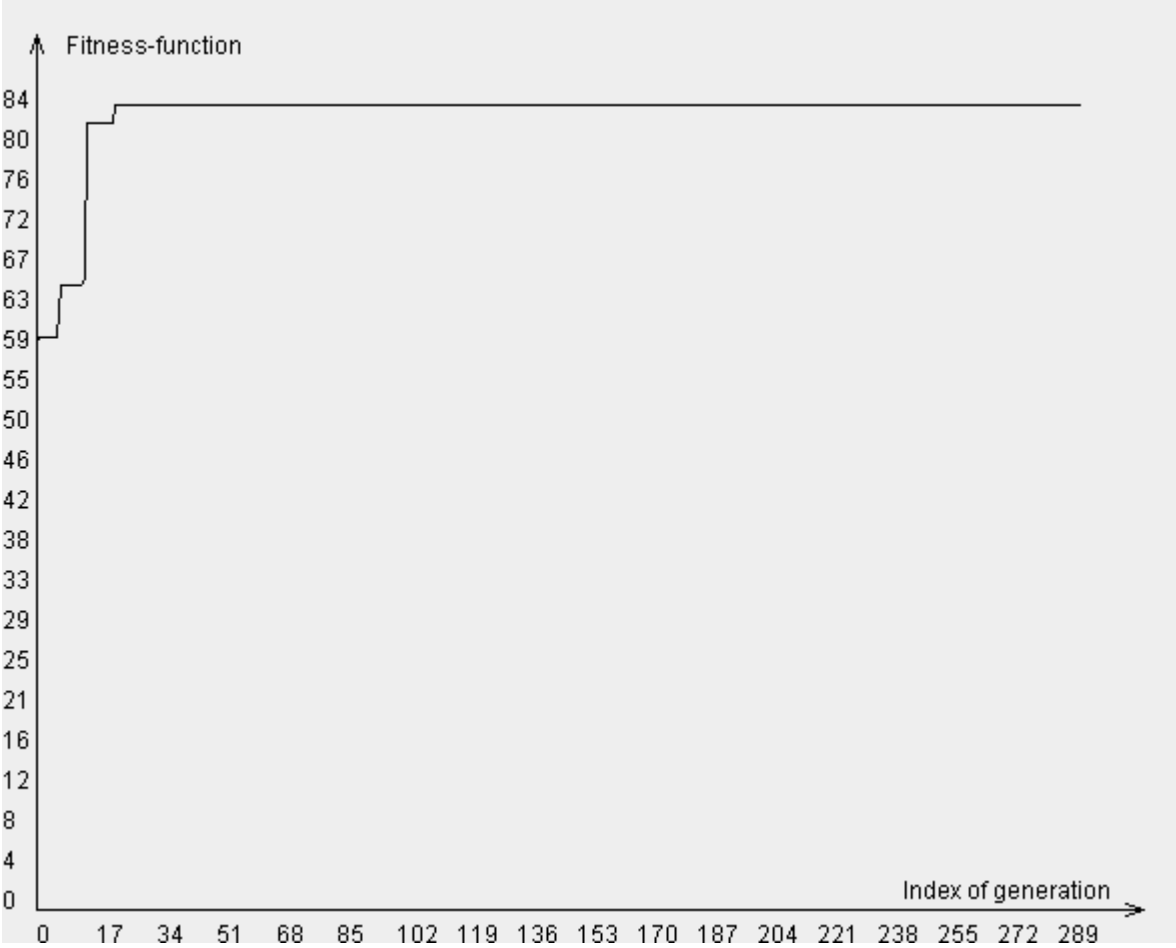

Рис. 7

3.2. График среднего значения функции приспособленности

На рис. 8 приведен график зависимости среднего значения функции приспособленности среди особей поколения.

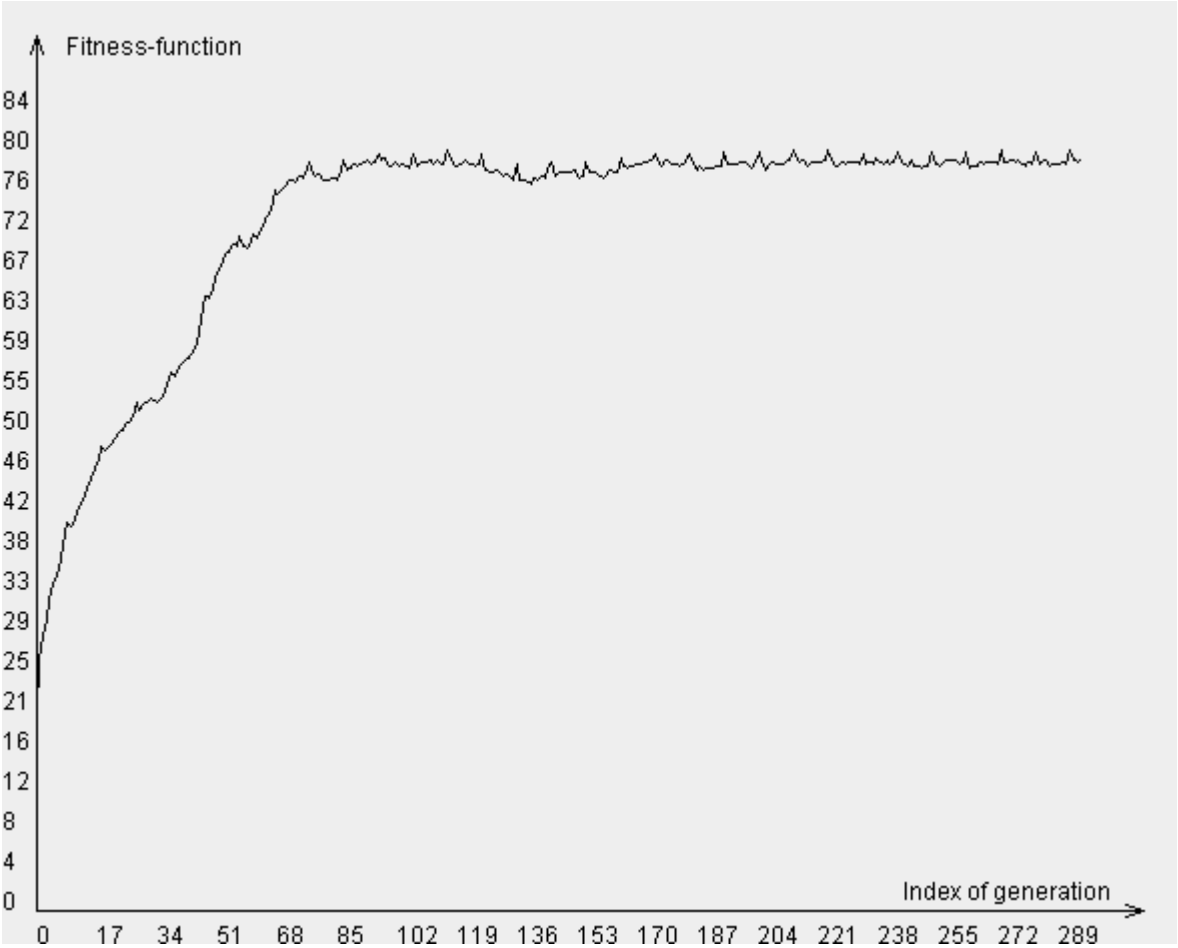

Рис. 8

3.3. Граф перехода полученного автомата

Полученный конечный автомат Мили представлен в виде графа переходов. Вершины графа – состояния автомата. Ребра графа – переходы между состояниями. На ребрах написано входное и выходное воздействия при переходе (рис. 9).

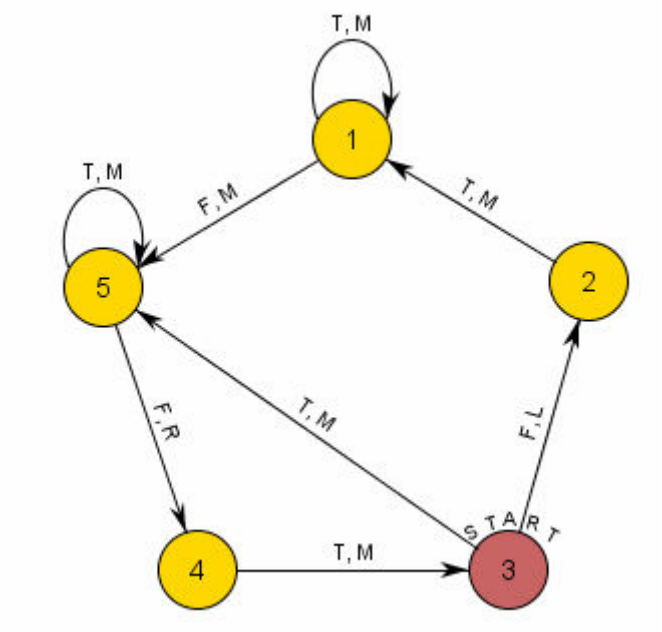

Рис. 9

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

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. Кроме того, как показывают графики функции приспособленности, достаточно неплохие результаты получены уже на первых поколениях особей, что говорит об

эффективности представления автомата и действенности островной модели [3]. При этом стоит заметить, что известен автомат, построенный для этой задачи вручную, который управляет муравьем, съедающим 81 яблоко [2]. К недостаткам можно отнести большое время создания очередного поколения, но это компенсируется тем, что в результате получен автомат, который как раз наиболее быстро и эффективно выполняет поставленную задачу.

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 использован ant-скрипт, который находится в файле `build.xml`, и пакет `NetBeans 6.5`.

Модуль «особи»:

1. `automaton.conf` – конфигурация задания:
    - `external` – число состояний автомата.
2. `plugin.properties` – конфигурация модуля.

Модуль генетического алгоритма:

1. `islandrouletteGA.conf` – конфигурация островного генетического алгоритма, с методом рулетки для генерации очередного поколения:
    - `numberIslands` – число островов.
    - `sizeIsland` – популяция острова.
    - `sizeElite` – число «элитных» особей в поколении.
    - `probabilityMutation` – вероятность мутации.
    - `numberSwap` – число миграций.
    - `periodSwap` – частота миграций.
    - `periodBigMutation` – периодичность мутаций.
2. `plugin.properties` – конфигурация модуля.

Настройки модуля генетического алгоритма совпадают с таковыми для стандартного островного ГА, реализованного в виртуальной лаборатории.

Исходные тексты программ

Модуль особи:

1. `Task3IndividualFactoryLoader.java` – класс, реализующий загрузку модуля в виртуальную лабораторию.

2. `Task3Automaton.java` – класс, реализующий работу с конечным автоматом Мили.
3. `BitArray.java` – класс, реализующий работу с представлением автомата в виде битовой строки.
4. `Task3AutomatonFactory.java` – класс, реализующий создание произвольной особи.

Модуль генетического алгоритма:

1. `RouletteGA.java` – класс, реализующий метод рулетки для генерации очередного поколения.
2. `IslandGA.java` – класс реализующий островной генетический алгоритм.
3. `IslandGALoader.java` – класс, реализующий загрузку модуля в виртуальную лабораторию.

Листинг `Task3IndividualFactoryLoader.java`:

```
package individual.task3;


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 Task3IndividualFactoryLoader implements
    Loader<IndividualFactory> {
```

```
    private final Parser properties;

    public Task3IndividualFactoryLoader(JarFile file) {
        Properties in = new Properties();
        try {
            JarEntry ent = new JarEntry("task3Automaton.conf");
            in.load(file.getInputStream(ent));
        } catch (IOException e) {
            e.printStackTrace();
        }
        properties = new Parser(in);
    }

    public IndividualFactory load(Object... args){
        return new Task3AutomatonFactory(properties.getInt("external"));
    }

    public Properties getProperties() {
        return properties.getProperties();
    }
    }
```

Листинг Task3Automaton.java:

```
package individual.task3;
```

```
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.individual.SimpleMover;

public class Task3Automaton implements Automaton {

    private final int numStates;
    private BitArray bitArr;
    private double fitness = Double.NEGATIVE_INFINITY;

    public Task3Automaton(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 Task3Automaton.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);

    Task3Automaton mutant = new Task3Automaton(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) {
    Task3Automaton[] s = new Task3Automaton[2];
    for (int i = 0; i < 2; i++) {
        s[i] = new Task3Automaton(0, getNumberStates());
    }

    Task3Automaton pp = (Task3Automaton)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;
        }

        @Override
        public String toString() {
            return "" + action;
        }

        public int getEndState() {
            return endState;
        }
    }
    }
```

Листинг BitArray.java:

```
package individual.task3;

import java.util.BitSet;

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 Task3AutomatonFactory.java:

```
package individual.task3;


import task.ant.simple.Ant;

import task.ant.simple.individual.factory.AutomatonFactory;


public class Task3AutomatonFactory extends
    AutomatonFactory<Task3Automaton>{


    public Task3AutomatonFactory(int numStates) {

        super(numStates, 0);

    }


    protected Task3Automaton fullRandomAutomaton(int numStates) {

        Task3Automaton a = new Task3Automaton(r.nextInt(numStates),
    numStates);

        for (int i = 0; i < numStates; i++) {

            a.setTransition(i, 1, new
    Task3Automaton.Transition(r.nextInt(numStates),
    Ant.ACTION_VALUES[r.nextInt(3)]));

            a.setTransition(i, 0, new
    Task3Automaton.Transition(r.nextInt(numStates),
    Ant.ACTION_VALUES[r.nextInt(3)]));

        }

        return a;

    }
```

```
    protected Task3Automaton randomAutomatonWN(int numStates,
    Task3Automaton na) {
        return fullRandomAutomaton(numStates);
    }

    }
```

Листинг RouletteGA.java:

```
package ga.islandroulette;

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 IslandGA.java:

```
package ga.islandroulette;

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 IslandGA implements GA {

    private final RouletteGA[] islands;

    private int indexGeneration;

    private final int numberSwap;

    private final int periodSwap;

    private final int sizeElite;

    private final int periodBigMutation;

    private final double percentKilledIslands;

    private Random r;

    public List<Individual> getGeneration() {
        List<Individual> gen = new ArrayList<Individual>();
        for (GA ga : islands) {
            gen.addAll(ga.getGeneration());
        }
        Collections.sort(gen);
```

```
    return gen;
}

public IslandGA(int numberIslands, int sizeIsland, double sizeElite,
double probabilityMutations,
                IndividualFactory factory, double numberSwap, int
periodSwap, int periodBigMutation,
                double percentKilledIslands) {
    indexGeneration = 0;
    islands = new RouletteGA[numberIslands];
    for (int i = 0; i < numberIslands; i++) {
        islands[i] = new RouletteGA(sizeIsland, sizeElite,
probabilityMutations, factory);
    }
    this.numberSwap = (int) (numberSwap * sizeIsland);
    this.periodSwap = periodSwap;
    this.sizeElite = (int) (sizeElite * sizeIsland);
    this.periodBigMutation = periodBigMutation;
    this.percentKilledIslands = percentKilledIslands;
    r = new Random();
}

public void nextGeneration() {
    indexGeneration++;
    for (GA ga : islands) {
```

```
            ga.nextGeneration();
        }
        if (indexGeneration % periodSwap == 0) {
            int ni = islands.length;
            List<Individual>[] generation = new List[ni];
            for (int i = 0; i < ni; i++) {
                generation[i] = islands[i].getGeneration();
            }
            int size = generation[0].size();
            for (int i = 0; i < ni; i++) {
                for (int j = 0; j < numberSwap; j++) {
                    generation[i].set(size - j - 1,
generation[r.nextInt(ni)].get(r.nextInt(sizeElite)));
                }
            }
            for (int i = 0; i < ni; i++) {
                Collections.sort(generation[i]);
            }
        } else if (indexGeneration % periodBigMutation == 0) {
            bigMutation();
        }
    }

    public void bigMutation() {
```

```
        for (GA ga : islands) {
            if (r.nextDouble() < percentKilledIslands) {
                ga.bigMutation();
            }
        }
    }

    public Individual getBest() {
        Individual best = islands[0].getBest();
        for (int i = 1; i < islands.length; i++) {
            Individual b = islands[i].getBest();
            if (b.fitness() > best.fitness()) {
                best = b;
            }
        }
    return best;
    }
    }
```

Листинг IslandGALoader.java:

```
package ga.islandroulette;

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 IslandGALoader implements Loader<GA> {

    private final Parser properties;

    public IslandGALoader(JarFile file) {
        Properties in = new Properties();
        try {
            JarEntry ent = new JarEntry("islandrouletteGA.conf");
            in.load(file.getInputStream(ent));
        } catch (IOException e) {
            e.printStackTrace();
        }
        properties = new Parser(in);
    }

    public GA load(Object... args) {
        return new IslandGA(properties.getInt("numberIslands"),
```

```
properties.getInt("sizeIsland"),
            properties.getDouble("sizeElite"),
properties.getDouble("probabilityMutation"),
            (IndividualFactory) args[0],
properties.getDouble("numberSwap"), properties.getInt("periodSwap"),
            properties.getInt("periodBigMutation"),
properties.getDouble("percentKillIsland"));
}

public Properties getProperties() {
    return properties.getProperties();
}
}
```