Санкт-Петербургский государственный университет
информационных технологий, механики и оптики
Факультет информационных технологий и программирования
Кафедра «Компьютерные технологии»

А. А. Шевченко

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

Вариант № 5

Санкт-Петербург
2010

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 Мили, представляющий логику муравья.

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

1. Постановка задачи об «Умном муравье»

Задача данной лабораторной работы – построить с помощью генетических алгоритмов конечный автомат Мили, решающий задачу об «Умном муравье», используя представление автоматов с помощью графов переходов, традиционный генетический алгоритм и метод «рулетки» для генерации очередного поколения.

В задаче об «Умном муравье» рассматривается поле, состоящее из клеток. Поле имеет размеры 32x32 клеток и располагается на поверхности тора. Некоторые клетки поля пусты, некоторые содержат по одному яблоку. Всего на поле 89 яблок. Муравей начинает свое движение из некоторой наперед заданной начальной клетки.

За один ход муравей может определить, есть ли в клетке перед ним яблоко, и выполнить одно из следующих действий:

– повернуть направо;
– повернуть налево;
– сделать шаг вперед, и если в новой клетке есть яблоко, то съесть его;
– ничего не делать.

Максимальное число ходов – 200.

Цель работы – создать муравья с фиксированным числом состояний, который съест как можно больше яблок.

2. Автомат Мили

Автомат Мили – это конечный автомат, генерирующий выходные воздействия в зависимости от текущего состояния и входного воздействия. Пример диаграммы переходов автомата приведен на рис. 1.

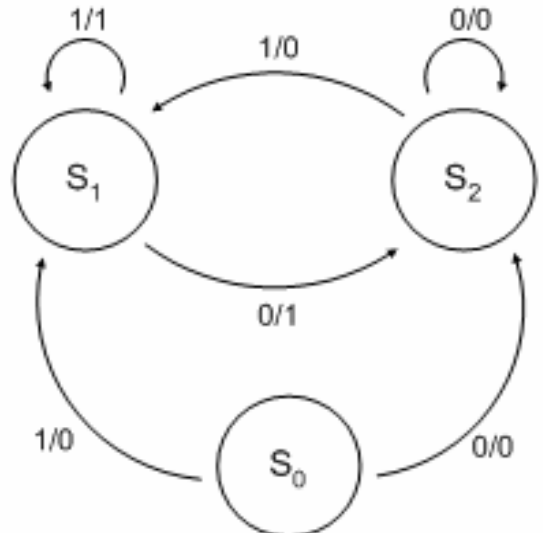

Рис. 1

Как видно из приведенной диаграммы, над каждой дугой расположена пара значений – входное и выходное воздействия. При этом выходное воздействие зависит не только от состояния, в котором находится автомат, но и от входного воздействия.

В данном генетическом алгоритме использовано представление автоматов с помощью графов переходов. Каждому состоянию соответствует одна вершина графа переходов и наоборот. Переход по входному сигналу *X* из состояния *A* в состояние *B* с подачей выходного сигнала *Y* реализуется в виде ребра в графе переходов из вершины, соответствующей состоянию *A*, в вершину, соответствующую состоянию *B*.

При этом, входное воздействие *X* представляет собой либо 1, если «перед муравьем есть еда», либо 0, когда «перед муравьем нет еды». Выходное воздействие *Y* – это одно из действий муравья:

– сделать шаг вперед, съедая еду, если она там находится;
– повернуть налево;
– повернуть направо;
– ничего не делать.

При этом выходные воздействия *Y* также кодируются целыми числами.

3. Генетический алгоритм

Название «генетические алгоритмы» ассоциирует с 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 [2].

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

3.1. Описание оператора мутации

Мутация может происходить одним из следующих способов:

– изменение начального состояния – в этом случае новое начальное состояние выбирается случайно и равновероятно;
– изменение действия на переходе – случайно и равновероятно выбирается переход, и действие на нем изменяется на случайное. При этом все возможные действия равновероятны;
– изменение состояния, в которое ведет переход – случайно и равновероятно выбирается переход. После этого состояние, в которое ведет переход, заменяется на случайно выбранное состояние;
– изменение условия на переходе – случайно и равновероятно выбирается состояние.

После этого переходы из этого состояния, соответствующие условиям «перед муравьем есть еда» и «перед муравьем нет еды», меняются местами.

3.2. Описание метода скрещивания

Обозначим переход из состояния номер *i* в автомате *A1* по значению входной переменной «Перед муравьем есть еда» как *A1(i, 1)*, а по значению «перед муравьем нет еды» как *A1(i, 0)*. Аналогичный смысл придадим обозначениям *A2(i, 0)* и *A2(i, 1)*. Тогда для переходов из состояния с номером i в автоматах-потомках *S1* и *S2* будет справедливо одно из четырех:
– либо *S1(i, 0) = A1(i, 0), S1(i, 1) = A2(i, 1)* и *S2(i, 0) = A2(i, 0), S2(i, 1) = A1(i, 1)*;
– либо *S1(i, 0) = A2(i, 0), S1(i, 1) = A1(i, 1)* и *S2(i, 0) = A1(i, 0), S2(i, 1) = A2(i, 1)*;
– либо *S1(i, 0) = A1(i, 0), S1(i, 1) = A1(i, 1)* и *S2(i, 0) = A2(i, 0), S2(i, 1) = A2(i, 1)*;
– либо *S1(i, 0) = A2(i, 0), S1(i, 1) = A2(i, 1) и S2(i, 0) = A1(i, 0), S2(i, 1) = A1(i, 1)*.

Причем, все четыре варианта равновероятны.

3.3. Описание метода генерации очередного поколения

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

Для начала создается начальное поколение. Далее на каждом шаге формируется следующее поколение.

В первую очередь, составляется промежуточное поколение: методом «рулетки» в него выбирается ровно столько же особей, сколько и в текущем поколении. Затем промежуточное поколение разбивается на пары, и каждая из них с некоторой вероятностью скрещивается.

После этого каждая особь из промежуточного поколения с некоторой вероятностью мутирует. И в следующее поколение попадают все особи из промежуточного поколения.

Однако с некоторой вероятностью вместо всего этого может произойти большая мутация: половина особей в следующее поколение будет выбрана методом «рулетки» из текущего, а половина — сгенерирована случайным образом.

3.4. Описание функции приспособленности

В данной работе для оценки «качества» особи используется следующая функция приспособленности:

$$F = E - T / 200,$$

где E – число съеденных яблок, а T – номер последнего хода, на котором было съедено яблоко.

4. Построенный автомат и графики максимального и среднего значений функции приспособленности

В табл. 1 изображена матрица смежности автомата Мили, полученного с помощью генетических алгоритмов в результате выполнения лабораторной работы. Стартовой является третья вершина. При этом переходы происходят при следующих условиях и со следующими действиями муравья:

T – переход, если есть еда; *F* – переход, если нет еды; *N* – ничего не делать; *L* – поворот налево; *R* – поворот направо; *M* – шаг вперед.

Таблица 1

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1			(T, M)	(F, R)				
2			(T, M)			(F, L)		
3						(F, M)	(T, M)	
4		(F, R)					(T, M)	
5				(T, M)	(F, R)			
6				(T, M)				(F, M)
7	(F, L)					(T, M)		
8	(F, L)			(T, M)				

На рис. 2 приведен график зависимости максимального значения функции приспособленности среди особей поколения (синий цвет), а также график зависимости среднего значения функции приспособленности в поколении (зеленый цвет).

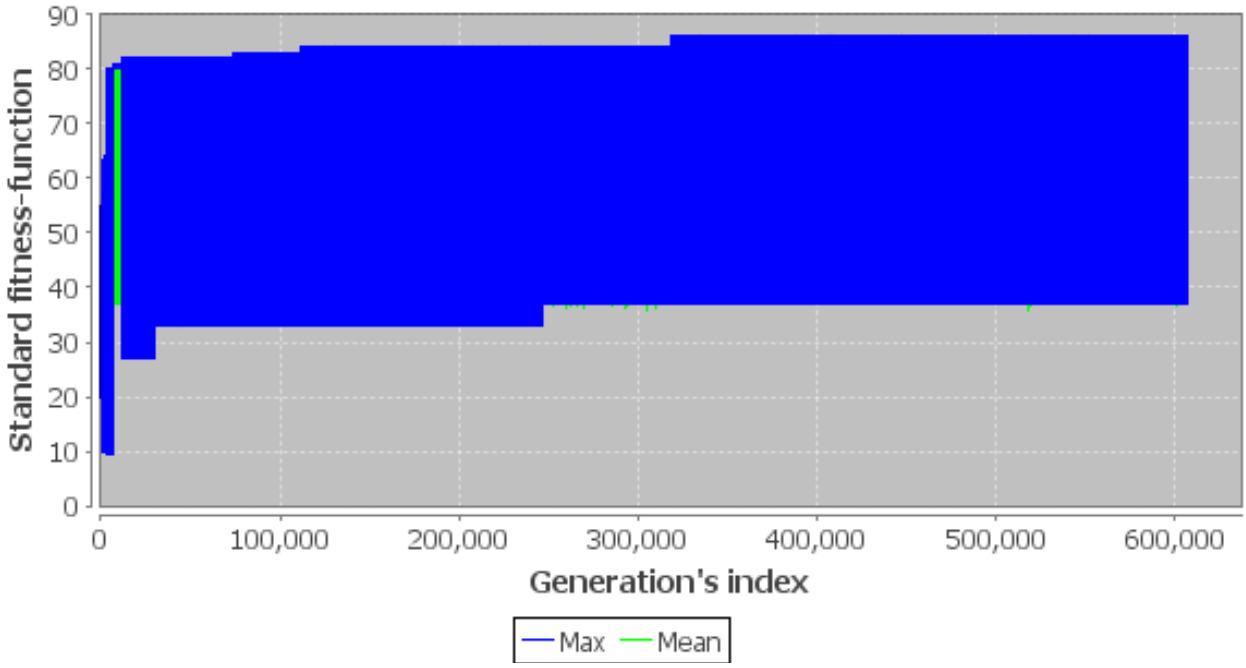

Рис. 2

Заключение

Результаты лабораторной работы показали, что используемые методы очень эффективны для решения рассматриваемой задачи. При этом предлагаемый алгоритм генерирует автомат с восемью состояниями, съедающий 87 яблок.

Заметим, что известен автомат с пятью состояниями, построенный для этой же задачи, который управляет муравьем, съедающим все 89 яблок.

Источники

1. Инструкция по созданию плагинов к виртуальной лаборатории.
http://svn2.assembla.com/svn/not_instrumental_tool/docs/pdf/interface_manual.pdf
2. Яминов Б. Генетические алгоритмы.
http://rain.ifmo.ru/cat/view.php/theory/unsorted/genetic-2005

Приложение. Исходный код

Файл StandartGA.java

```
package laboratory.plugin.algorithm.standart;

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

import laboratory.common.genetic.Algorithm;
import laboratory.common.genetic.FitIndividual;
import laboratory.common.genetic.Individual;
import laboratory.common.genetic.IndividualFactory;
import laboratory.common.genetic.operator.Mutation;
import laboratory.common.genetic.operator.Crossover;
import laboratory.common.genetic.operator.Selection;
import laboratory.common.genetic.operator.Fitness;

public class StandartGA<T extends Individual> implements Algorithm<T>{
        private FitIndividual<T>[] generation;
        private int sizeOfGeneration;

        private final IndividualFactory<T> factory;
    private final Mutation<T> mutation;
    private final Crossover<T> crossover;
    private final Selection<T> selection;
    private final Fitness<T> fitness;

    private double probabilityOfMutation;
    private double probabilityOfBigMutation;
    private double probabilityOfCrossover;

    @SuppressWarnings("unchecked")
        public StandartGA(final int sizeOfGeneration, final double probabilityOfMutation,
                final double probabilityOfBigMutation, final double probabilityOfCrossover,
                final IndividualFactory<T> factory, final Mutation<T> mutation,
                final Crossover<T> crossover, final Selection<T> selection, final Fitness<T> fitness)
{

        this.sizeOfGeneration = sizeOfGeneration;
        this.probabilityOfMutation = probabilityOfMutation;
        this.probabilityOfBigMutation = probabilityOfBigMutation;
        this.probabilityOfCrossover = probabilityOfCrossover;

        this.factory = factory;

        this.mutation = mutation;
        this.crossover = crossover;
        this.selection = selection;
        this.fitness = fitness;

        generation = new FitIndividual[sizeOfGeneration];

        for (int i = 0; i < sizeOfGeneration; i++) {
                generation[i] = makeFitIndividual(factory.getIndividual());
        }
    }

        public List<T> getGeneration() {
                List<T> list = new ArrayList<T>(this.sizeOfGeneration);
                for (int i = 0; i < this.sizeOfGeneration; i++) {
                        list.add(generation[i].ind);
                }
                return list;
        }



        public void nextGeneration() {
                Random rand = new Random();
                if (rand.nextDouble() < probabilityOfBigMutation) {
            bigMutation();
         } else {
             int fP = this.sizeOfGeneration;
                List<FitIndividual<T>>                    btwGen =
```

```
selection.apply(Arrays.asList(generation), fP);
            for (int i = 0; i < fP; i++) {
               generation[i] = btwGen.get(i);
            }

            //doing things with pairs
            boolean[] usedElements = new boolean[this.sizeOfGeneration];
            int used = 0;
            for (int i = 0; i < this.sizeOfGeneration; i++) {
               if (usedElements[i]) continue;
               used += 2;
               if (used > this.sizeOfGeneration) continue;
               usedElements[i] = true;
               int pairForMe = rand.nextInt(this.sizeOfGeneration);
               while (usedElements[pairForMe]) {
                      pairForMe = (pairForMe + 1) % sizeOfGeneration;
               }
               usedElements[pairForMe] = true;

               //now working with i & pairForMe elements

               if (rand.nextDouble() < probabilityOfCrossover) {
                      List<T> list = crossover.apply(Arrays.asList(generation[i].ind,
generation[pairForMe].ind));

                      FitIndividual<T> temp = makeFitIndividual(list.get(0));
                      if (temp.fitness > generation[i].fitness){
                             generation[i] = temp;
                      }

                      temp = makeFitIndividual(list.get(1));
                      if (temp.fitness > generation[pairForMe].fitness){
                             generation[pairForMe] = temp;
                      }
               }
            }

            for (int i = 0; i < this.sizeOfGeneration; i++) {
               if (rand.nextDouble() < probabilityOfMutation) {
                      generation[i] = makeFitIndividual(mutation.apply(generation[i].ind));
               }
            }
         }
        }

        private void bigMutation() {
               int fP = this.sizeOfGeneration / 2;
        List<FitIndividual<T>> btwGen = selection.apply(Arrays.asList(generation), fP);
         for (int i = 0; i < fP; i++) {
               generation[i] = btwGen.get(i);
         }
               for (int i = fP; i < this.sizeOfGeneration; i++) {
                      generation[i] = makeFitIndividual(factory.getIndividual());
               }
        }

        public void stop() {
        }

    private FitIndividual<T> makeFitIndividual(T i) {
        return new FitIndividual<T>(i, fitness.apply(i));
    }
}
```

Файл RouletteSelection.java

```
package laboratory.plugin.algorithm.selection.roulette;

import laboratory.common.genetic.FitIndividual;
import laboratory.common.genetic.Individual;
import laboratory.common.genetic.operator.Selection;
import laboratory.util.functional.Util;
import laboratory.util.functional.Functor0;

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

```
public class RouletteSelection<I extends Individual> implements Selection<I> {

    private final Random random = new Random();

    @Override
    public List<FitIndividual<I>> apply(final List<FitIndividual<I>> population, int m) {
        final int n = population.size();
        final double[] weight = new double[n];
        weight[0] = population.get(0).fitness;

        double max = weight[0];
        int maxI = 0;

        for (int i = 1; i < n; i++) {
            weight[i] = weight[i - 1] + population.get(i).fitness;
            if (weight[i] - weight[i - 1] > max) {
               max = weight[i] - weight[i - 1];
               maxI = i;
            }
        }

        List<FitIndividual<I>> list = new ArrayList<FitIndividual<I>>(m);

        list.add(population.get(maxI));
        m--;
        list.add(population.get(maxI));
        m--;

        double sectorLen = weight[n - 1] / m - 1e-6;
        double r = random.nextDouble() * sectorLen;
        int curInd = 0;
        for (int i = 0; i < m; i++) {
               while (weight[curInd] < r) {
                      curInd++;
               }
               list.add(population.get(curInd));
               r += sectorLen;
        }
        return list;
    }
}
```