Санкт-Петербургский национальный исследовательский университет
информационных технологий, механики и оптики
Факультет информационных технологий и программирования
Кафедра «Компьютерные Технологии»

В.Е. Аксенов

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

Вариант №1

Санкт-Петербург
2011 г.

Оглавление

Введение

В лабораторной работе требуется найти зависимость эффективности работы генетического алгоритма построения автомата, решающего задачу об умном муравье, от вероятности мутации. Для этого реализовывался плагин для виртуальной лаборатории «3genetic» на языке *Java*, который реализует конкретный принцип отбора особей и конкретный принцип мутации особей.

1. Постановка задачи

Задача данной лабораторной работы – исследовать влияние вероятности мутации на эффективность работы алгоритма, строящего автомат Мили из семи состояний, решающий задачу об «Умном муравье». Критерий оценки автомата 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. Задача об умном муравье

Дано поле размером 32×32 клетки, расположенное на поверхности тора. 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.

Всего делается 200 ходов. Требуется построить муравья с определенным 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 реализовать два новых модуля: модуль генерации очередного поколения и модуль, реализующий несколько иную, чем встроенная, операцию мутации.

2.1. 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 турнира.

Шаг алгоритма состоит из трех стадий: генерация промежуточной популяции путем отбора текущего поколения и скрещивание особей промежуточной популяции путем кроссовера. Из получившихся особей формируется новое поколение, особи которого далее подвергаются мутации. Под промежуточной популяцией подразумевается набор особей, которые получили право размножаться.

В описанном генетическом алгоритме метод отбора в промежуточную популяцию реализован в виде отбора в форме турнира. В данной работе он реализован следующим образом: имеется популяция особей – n штук, также мы знаем, какое число из них должно пройти в промежуточную популяцию – m. Для выбора каждой особи выберем случайным образом восемь особей из исходной популяции и проведем среди них турнир по «олимпийской» системе в три раунда. В каждом раунде разобьем особи на пары, из каждой пары в следующий раунд особь выходит с вероятностью, пропорциональной ее функции приспособленности.

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.

2.2. Оператор мутации

Оператор мутации реализован следующим образом. Для каждого перехода из каждого состояния действие при этом переходе и номер следующего состояния с вероятностью p изменяются на случайные. С той же вероятностью может измениться начальное состояние автомата.

3. Результаты работы

Для каждой вероятности (0.1, 0.2, 0.3, …, 1.0) были произведены 10 и 100 запусков. В каждом запуске бралась величина максимальной функции приспособленности в соответствующем поколении. После получения 10 или 100 результатов запусков, данные значения усреднялись. Результаты запусков отражены на рис. 1 (для 10 запусков) и рис. 2 (для 100 запусков).

Можно заметить, что наибольшая эффективность достигается при p от 0.05 до 0.2. Рассмотрим этот промежуток подробнее с шагом вероятности 0.01 (рис. 3 для усреднения по 10 запускам, число поколений не превышало 200, рис. 4 для усреднения по 100 запускам, число поколений не превышало 1000). По графикам, изображенным на рис. 3, можно сделать вывод, что при выборе вероятности в пределах от 0.08 до 0.12 генетический алгоритм работает наиболее эффективно. При рассмотрении графиков на рис. 4 и 5, столь однозначных выводов сделать нельзя.

Заключение

Результаты лабораторной работы показали, что эффективное построение автомата Мили с семью состояниями, который решает задачу об «Умном муравье», наблюдается при значениях p в диапазоне от 0.05 до 0.2. В качестве оптимального значения вероятности для описанного в работе оператора мутации можно порекомендовать значение 0.1.

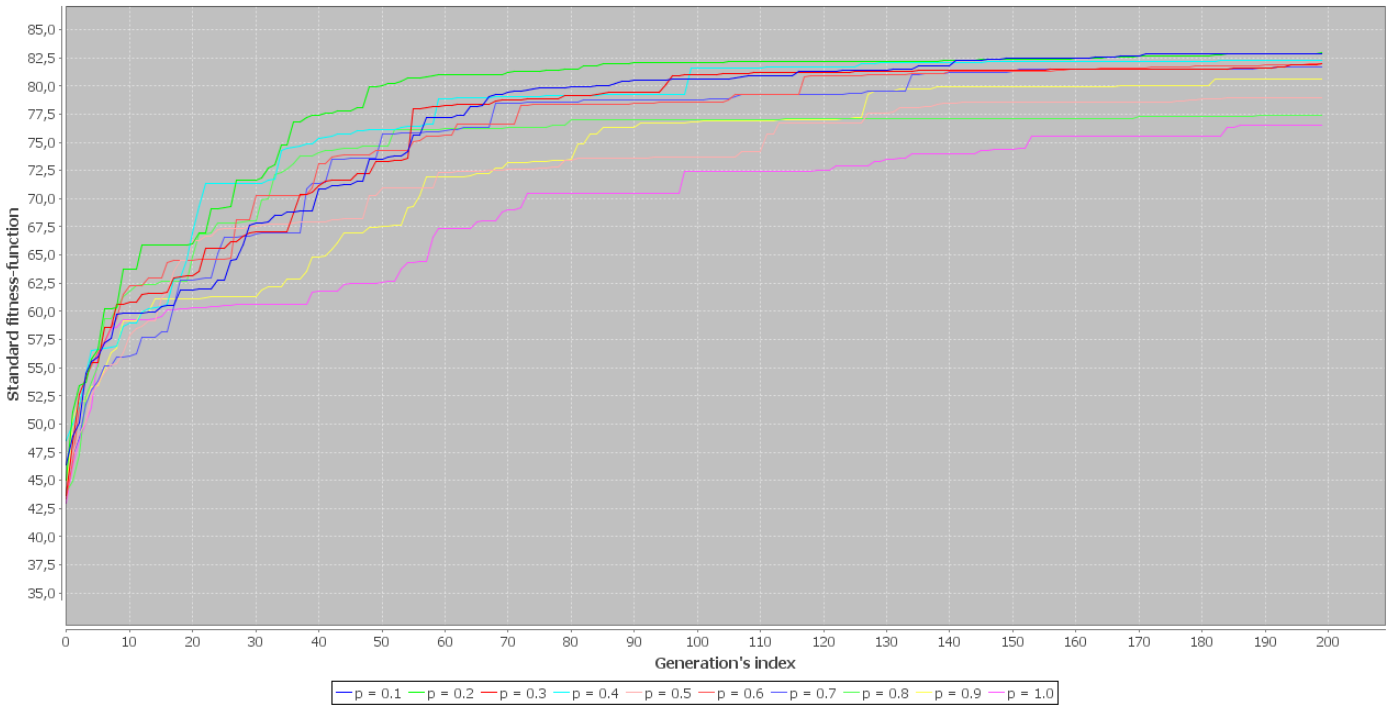

Рис. 1. Графики функции приспособленности: 10 запусков с размером поколения 200

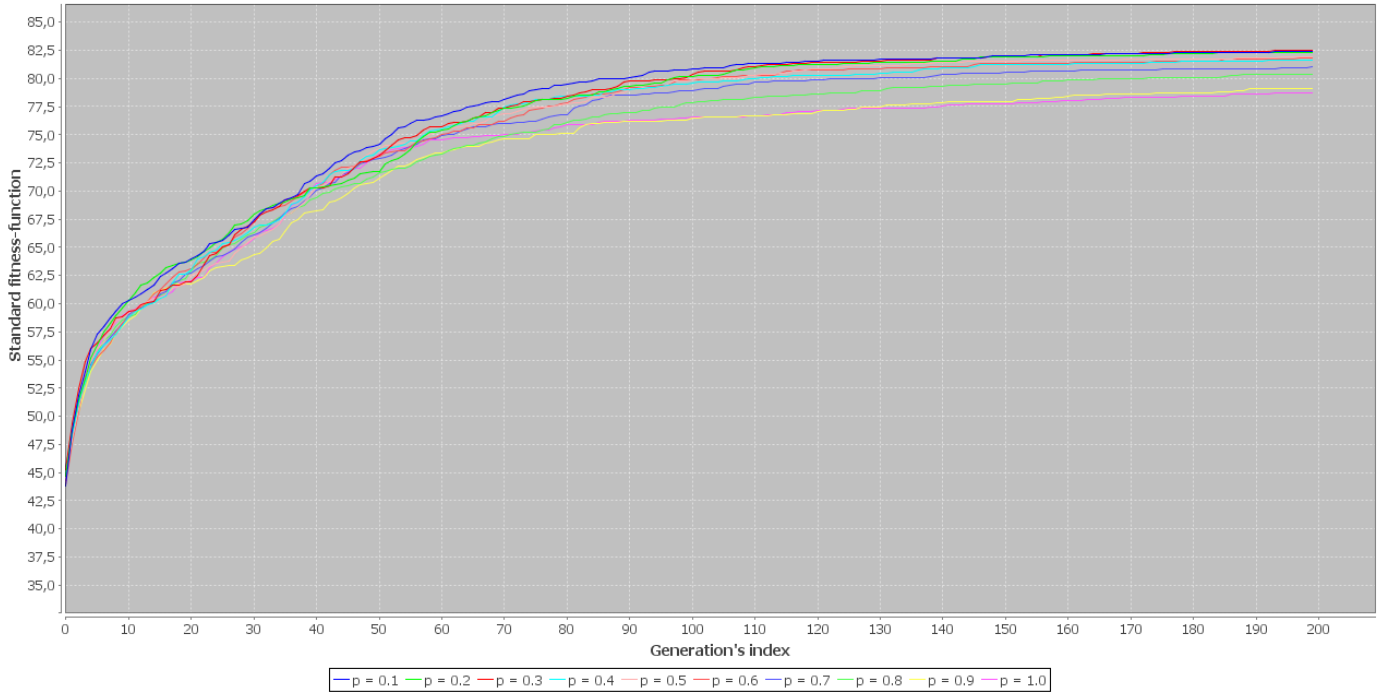

Рис. 2 Графики функции приспособленности: 100 запусков с размером поколения 200

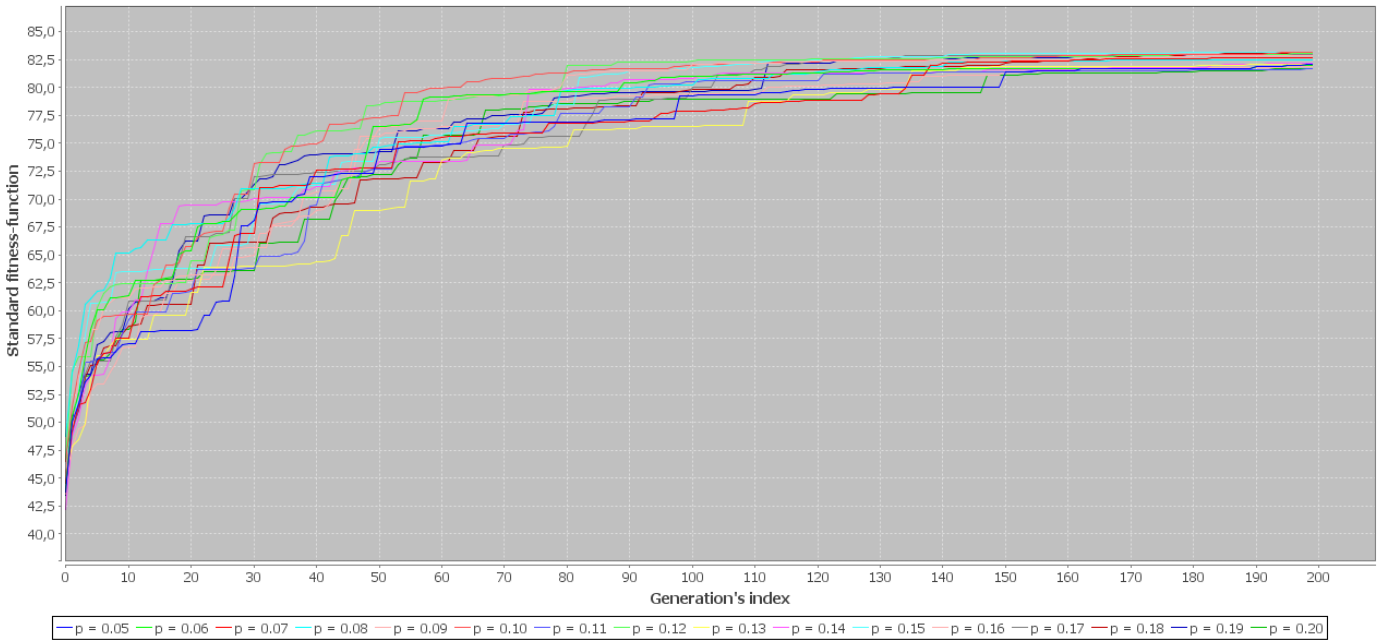

Рис. 3. Графики функции приспособленности: 10 запусков на отрезке с размером поколения 200

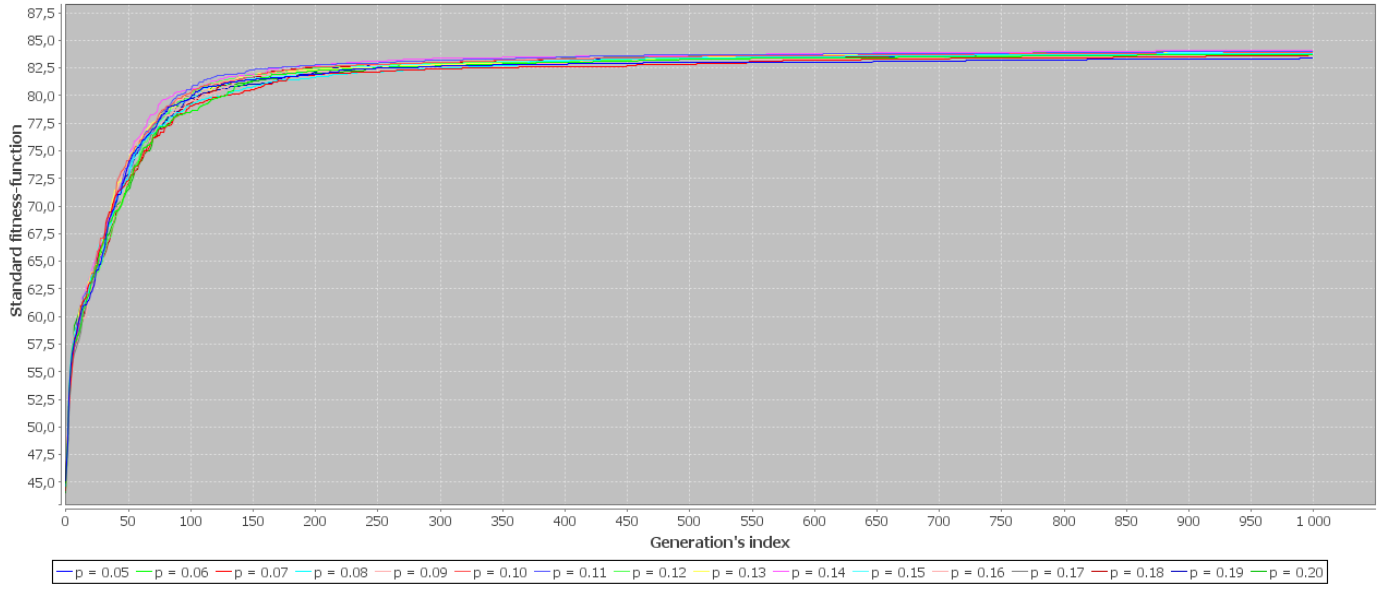

Рис. 4. Графики функции приспособленности: 100 запусков на отрезке с размером поколения 200

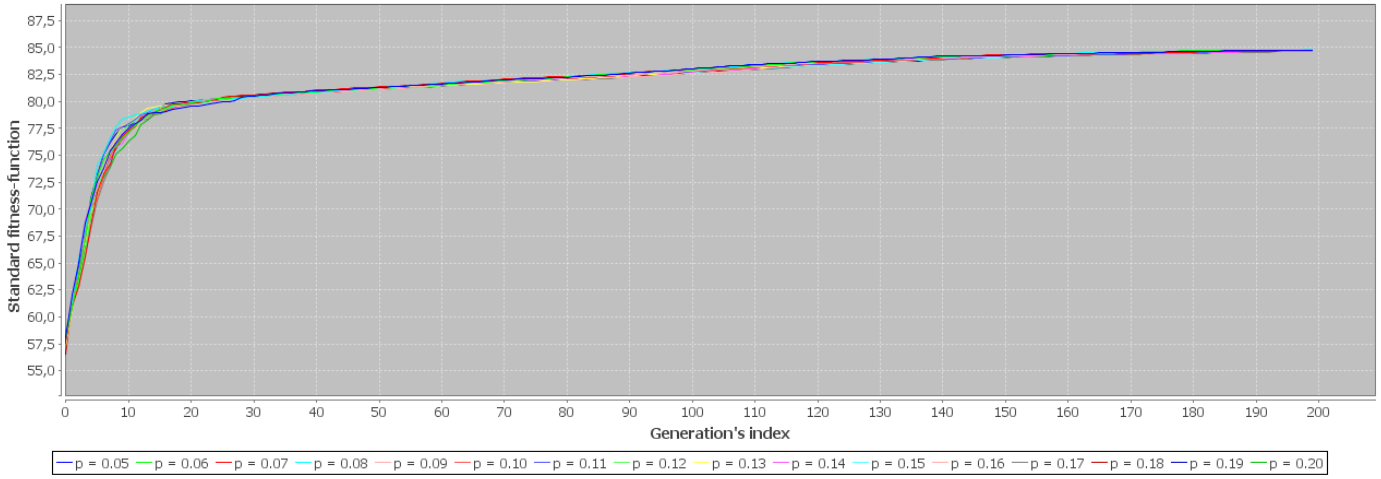

Рис. 5. Графики функции приспособленности: 100 запусков на отрезке с размером поколения 5000

Приложение. Исходный код

Файл TournamentLoader.java

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

import laboratory.common.genetic.FitIndividual;
import laboratory.common.genetic.Individual;
import laboratory.common.genetic.operator.Selection;

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

public class TournamentSelection<I extends Individual> implements Selection<I> {
    private final Random random = new Random();

    private double[] weight;

    public int getOne(int[] who) {
        if (who.length == 1)
            return who[0];
        int[] next = new int[(who.length + 1) / 2];
        int max = 0; // We take individual with the highest fitness function to
                     // the next part, if who.length % 2 != 0
        for (int i = 0; i < who.length; i++) {
            if (weight[who[i]] > weight[who[max]]) {
                max = i;
            }
        }
        int p = who[who.length - 1];
        who[who.length - 1] = who[max];
        who[max] = p;

        for (int i = 0; i < who.length; i += 2) {
            if (i + 1 > who.length) {
                next[i / 2] = who[i]; //Maximum was in the end
            } else {
                if (random.nextDouble() < 1.0 * weight[who[i]] / (weight[who[i]]
                        + weight[who[i + 1]])) {
                    next[i / 2] = who[i];
                } else {
                    next[i / 2] = who[i + 1];
                }
            }
        }
        return getOne(next);
    }

    @Override
    public List<FitIndividual<I>> apply(
            final List<FitIndividual<I>> population, int m) {
        final int n = population.size();
        weight = new double[n];
        weight[0] = population.get(0).fitness;
        for (int i = 1; i < n; i++) {
            weight[i] = population.get(i).fitness;
        }
        List<FitIndividual<I>> nextPopulation = new ArrayList<FitIndividual<I>>(m);
        int len = (n + m - 1) / m;
        for (int i = 0; i < m; i++) {
            int[] who = new int[8];
            for (int j = 0; j < who.length; j++) {
                who[j] = rand.nextInt(n);
            }
            int ind = getOne(who);
            nextPopulation.add(population.get(ind));
        }
        return nextPopulation;
    }
}
```

Файл AbstractAutomatonMutation.java

```
package laboratory.plugin.task.ant.individual.operator;

import laboratory.plugin.algorithm.simple.Config;
import laboratory.plugin.individual.mealy.MealyAutomaton;
import laboratory.plugin.task.ant.Ant;
import laboratory.plugin.task.ant.individual.AbstractAutomaton;
import laboratory.plugin.task.ant.individual.Automaton;
import laboratory.plugin.task.ant.individual.Automaton.Transition;
import laboratory.common.genetic.operator.Mutation;

import java.util.Random;

public abstract class AbstractAutomatonMutation<I extends AbstractAutomaton>
        implements Mutation<I> {

    private Random r;

    public AbstractAutomatonMutation(Random r) {
        this.r = r;
    }

    public I apply(I individual) {
        double p = Config.getInstance().getMutationProbability();

        I res = individual;
        if (r.nextDouble() < p) {
            res = (I) res.setInitialState(r.nextInt(individual.getNumberStates()));
        }

        Automaton.Transition[][] tr = res.getTransition();
        for (int temp = 0; temp < tr.length; temp++) {
            for (int tempT = 0; tempT < tr[temp].length; tempT++) {
                if (tr[temp][tempT] == null) {
                    if (r.nextDouble() < p) {
                        res.setTransition(temp, tempT,
                                new MealyAutomaton.Transition(r.nextInt(tr.length),
                                        Ant.ACTION_VALUES[r.nextInt(3)]));
                    }
                    continue;
                }

                if (r.nextDouble() < p) {
                    tr[temp][tempT].setEndState(r.nextInt(tr.length));
                }

                if (res instanceof MealyAutomaton) {
                    Automaton.Transition t = res.getTransition(temp, tempT);
                    if (r.nextDouble() < p)
                        res.setTransition(temp, tempT,
                                new MealyAutomaton.Transition(t.getEndState(),
                                        Ant.ACTION_VALUES[r.nextInt(3)]));
                }
            }
        }

        res = (I) res.setTransitions(tr);
        if ((res.getNestedAutomaton() != null) && (r.nextBoolean())) {
            res = (I) res.setNestedAutomaton(apply((I) res.getNestedAutomaton()));
        }

        return res;
    }
}
```