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

Бужинский И. П.

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

Вариант №3

Санкт-Петербург
2011

Содержание

Введение

В данной лабораторной работе предлагалось сравнить эффективность работы генетического алгоритма (ГА) при использовании различных видов кроссовера на примере построения автомата Мура [1] (способ задания автомата — с помощью полных или сокращенных таблиц). Необходимо было исследовать двухточечный кроссовер и двухточечный кроссовер со смещением (особи обмениваются подстроками одной длины с различными смещениями). С этой целью была написана программа на языке программирования *Java*, производящая серии запусков генетического алгоритма с возможностью задания его параметров.

1. Постановка задачи

Задача, предложенная в данном варианте лабораторной — задача «Умный муравей – 3», которая заключается в построении конечного автомата для управления муравьем на тороидальном поле, в каждой клетке которого может находиться единица пищи.

Цель — с помощью генетического алгоритма получить автомат, при использовании которого за 200 тактов автомата муравей съедает большее количество пищи, случайно расположенной на поле.

1.1. Задача «Умный муравей – 3»

Автомат управляет муравьем на поле размером 32×32 клетки. Поле представляет из себя тор, то есть выход за его край означает появление муравья с другой стороны поля. В каждый момент времени муравей находится в конкретной клетке поля и смотрит в одну из четырех сторон: влево, вправо, вниз или вверх. В некоторых клетках может находиться пища (ее расположение задается случайно). Муравей видит, есть ли пища на восьми клетках вокруг него (рис. 1).

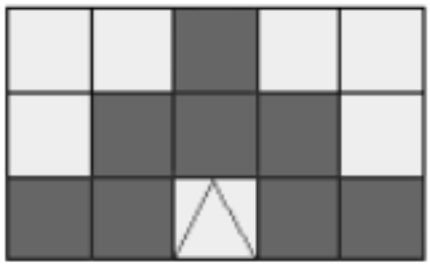

Рис. 1: Клетки, которые видит муравей (обозначены темно-серым цветом)

У муравья есть 200 ходов, чтобы съесть как можно больше пищи. Муравей съедает одну единицу пищи, когда попадает на клетку, где она есть, при этом пища исчезает с этой клетки. Каждый ход муравей может (действия для краткости обозначим буквами):

- пойти вперед (*F*);
- повернуться на 90° против часовой стрелки (*L*);
- повернуться на 90° по часовой стрелке (*R*);
- ничего не делать (*N*).

Это и есть выходные воздействия автомата, управляющего муравьем. В данной работе рассматривались автоматы только с воздействиями типов *F*, *L*, *R*. Выходные воздействия типа *N* бесполезны, поскольку они могут быть устранены из автоматов, их содержащих, алгоритмом, подобным алгоритму ε-замыкания для автоматов, распознающих регулярные языки.

Уточним формулировку задачи. Будем считать, что в каждой клетке поля еда находится с некоторой вероятностью μ. В работе [2] проводились исследования решения той же задачи с па-

раметром μ = 0,05, при этом получалась особь, съедавшая в среднем 51.6% единиц пищи. В данной работе было взято значение μ = 0,15.

С одной стороны, такое значение взято, так как интерес представляет предельная эффективность автомата на поле с большим количеством пищи. С другой стороны, при таком μ переходы автомата при прохождении поля муравьем будут использоваться более равномерно. Поясним это. Рассмотрим случайную величину X, равную количеству единиц пищи, которое муравей видит в начале прохождения поля. Поскольку в каждой видимой клетке нахождение единицы пищи независимо, X имеет биномиальное распределение (см. таблицу). Распределение количества видимой муравьем пищи в процессе прохождения поля не так просто вычислить, но можно ожидать смещение в сторону меньших значений.

Таблица: Распределение и математического ожидание X при различных μ

	μ=0,05	μ=0,10	μ=0,15
$P(X = 0)$	0,6634	0,4305	0,2725
$P(X = 1)$	0,2793	0,3826	0,3847
$P(X = 2)$	0,0515	0,1488	0,2376
$P(X = 3)$	0,0054	0,0331	0,0839
$P(X > 3)$	0,0004	0,0050	0,0213
$E[X]$	0,4000	0,8000	1,2000

Из таблицы видно, что при μ = 0,05 муравей редко видит больше двух единиц пищи. Это означает, что переходы управляющего автомата по наборам входных переменных, соответствующим большому количеству пищи, почти не используются. При μ = 0,05 предложенные в задании способы представления автомата оказалось бы малоэффективным: можно было бы не хранить редко используемые переходы, а при необходимости их использования получать их некоторым детерминированным способом.

При μ = 0,10 и μ = 0,15 среднее количество видимых единиц пищи увеличивается. Можно было бы взять еще большее значение μ, но при его увеличении оптимальная стратегия муравья становится примитивной: он преимущественно движется вперед и иногда совершает поворот, если слева или справа от него находится пища. Так как это сделало бы задачу менее интересной, использовалось значение μ = 0,15.

2. Реализация

В этом разделе описывается реализация ГА, используемая в данной работе, а также выбранная функция приспособленности и способ задания автоматов. Указаны значения параметров (вероятность мутации и т. п.) алгоритма, используемых при исследованиях. Для некоторых параметров приводятся результаты экспериментов, обосновывающие выбор их значения.

2.1. Вычисление функции приспособленности

Функцию приспособленности (ФП) автомата для поля определим как $f_0 = 100\ N / M$, где N — число единиц пищи, съеденное муравьем за 200 ходов, M — число единиц пищи на поле. То есть f_0 является процентом съедаемой муравьем пищи.

Определим истинную ФП автомата f' как математическое ожидание f_0, где поле берется случайным способом, описанным в разделе 1.1.

Разумеется, при таком определении ФП ее непосредственный подсчет представляет трудность, поэтому при работе алгоритма будем считать ее приближение f, делая усреднение f_0 по 100 заранее случайно сгенерированным полям. Для оценки качества приближения будем считать, что истинная ФП равна усреднению f_0 по 2000 случайным полям.

После работы ГА найденное максимальное значение f в подавляющем большинстве случаев не превосходило истинное значение более чем на 5% (при f, достигающей значения 40), что служит подтверждением, что усреднение по 100 полям является приемлемым приближением.

2.2. Способ задания автомата

В настоящей работе для задания автомата рассматривалось два способа.

- Полные таблицы входных воздействий. В каждом состоянии хранятся переходы по каждому двоичному набору из восьми входных воздействий.
- Сокращенные таблицы входных воздействий. Для каждого состояния есть набор из k ($k \leq 8$ и постоянно для каждого автомата) входных воздействий, который считается значимым. В каждом состоянии хранятся переходы только по двоичным наборам из значимых состояний.

2.3. Метод генерации очередного поколения

Считаем, что в поколении N особей. Во всех приведенных далее статистических данных N бралось равным 200.

Для генерации следующего поколения используются метод «рулетки» и элитизм. Сначала сформируем промежуточное поколение следующим образом. Разобьем отрезок [0, 1] на области, при этом сопоставим каждой области особь из старого поколения и возьмем длину области, пропорциональную функции приспособленности данной особи (аналогия с колесом рулетки). Таким образом, при случайном (с равномерным распределением) выборе точки на отрезке [0, 1] с большей вероятностью мы попадем в области, соответствующие более приспособленным особям.

Далее отбираем особи в новое поколение по одной. С помощью метода «рулетки» выбираются две случайные особи. С некоторой вероятностью p_{cr} они скрещиваются. Из двух особей выбирается случайная, мутирует и добавляется в новое поколение.

Так будем поступать, пока не заполним 90% нового поколения. Оставшиеся 10% поколения заполняются при помощи элитизма (берутся 10% лучших особей старого поколения и проходят в новое поколение без мутации).

Для подбора оптимальной вероятности кроссовера проводились запуски ГА до 300-ого поколения при p_{cr} = 0,3; 0,35; 0,4; 0,5; 0,6; 0,7. Более быстрая сходимость наблюдалась при

p_{cr} = 0,35 и 0,6, при этом однозначный выбор среди этих двух значений сделать было нельзя. Для определенности далее в работе использовалось значение p_{cr} = 0,6.

2.4. Оператор мутации

Используемый оператор мутации имеет следующий вид. С вероятностью p происходит изменение на случайное стартового состояния. С этой же вероятностью в каждом состоянии случайным образом меняется выходное воздействие и переход по каждому из восьми входных воздействий.

Если автомат представлен в виде сокращенных таблиц, с той же вероятностью p происходит мутация маски значимых входных воздействий: случайным образом выбираются два неравных бита маски и меняются местами.

Часто в ГА применение самого оператора мутации происходит с некоторой вероятностью q (порядка 0,01 — 0,1), чтобы особи не мутировали слишком сильно. Но существует другой подход: положить q = 1 и использовать меньшие p. Результаты применения этих двух подходов могут не совпадать, поэтому были рассмотрены оба. Запуски ГА (с представлением автомата в виде сокращенных таблиц) при q = 0,1 и q = 1 с различными p показали, что q = 1 и p порядка 0,001 — 0,0015 дают наилучшие результаты. Было принято решение использовать p = 0,001.

2.5. «Малые» и «большие» мутации

В данной работе была проведена попытка использовать «малые» и «большие» мутации, применение которых, например, упоминается в статье [3]. Если в течение k поколений нет роста ФП, происходит «малая» мутация: ко всем особям, кроме 10% лучших, десять раз применяется оператор мутации. Если после «малой» мутации за k поколений ФП снова не возрастает, происходит «большая» мутация: каждая особь или мутирует десять раз, или заменяется на случайную (выбор одного из этих двух действий происходит равновероятно). Далее, если снова не происходит улучшений ФП, применяется «малая» мутация и т. д.

Исследовались значения k = 5, k = 7 и $k = \infty$ (отсутствие «малых» и «больших» мутаций), при этом проводилось усреднение по 20 запускам ГА, где ФП считалась на 100 полях (как и во всех дальнейших исследованиях), а автоматы представлялись в виде сокращенных таблиц. Отдельно рассматривались случаи с двухточечным кроссовером и двухточечным кроссовером со смещением. Результаты, представленные на рис. 2, говорят о том, что «малые» и «большие» мутации ухудшают работу ГА при двухточечном кроссовере и улучшают при двухточечном кроссовере со смещением. Из-за отсутствия однозначности результатов, а также из-за увеличения стохастичности в работе ГА было принято решение не использовать «малые» и «большие» мутации.

На рис. 3 представлены графики зависимости максимальной и средней ФП от номера поколения при использовании «малых» и «больших» мутаций и без них для одного конкретного запуска ГА. Эти графики не позволяют делать по себе статистических выводов, но являются более наглядными. Например, на графике максимальной ФП при k = 5 после 200-ого поколения становятся заметными последствия «больших» мутаций — спад максимального значения ФП.

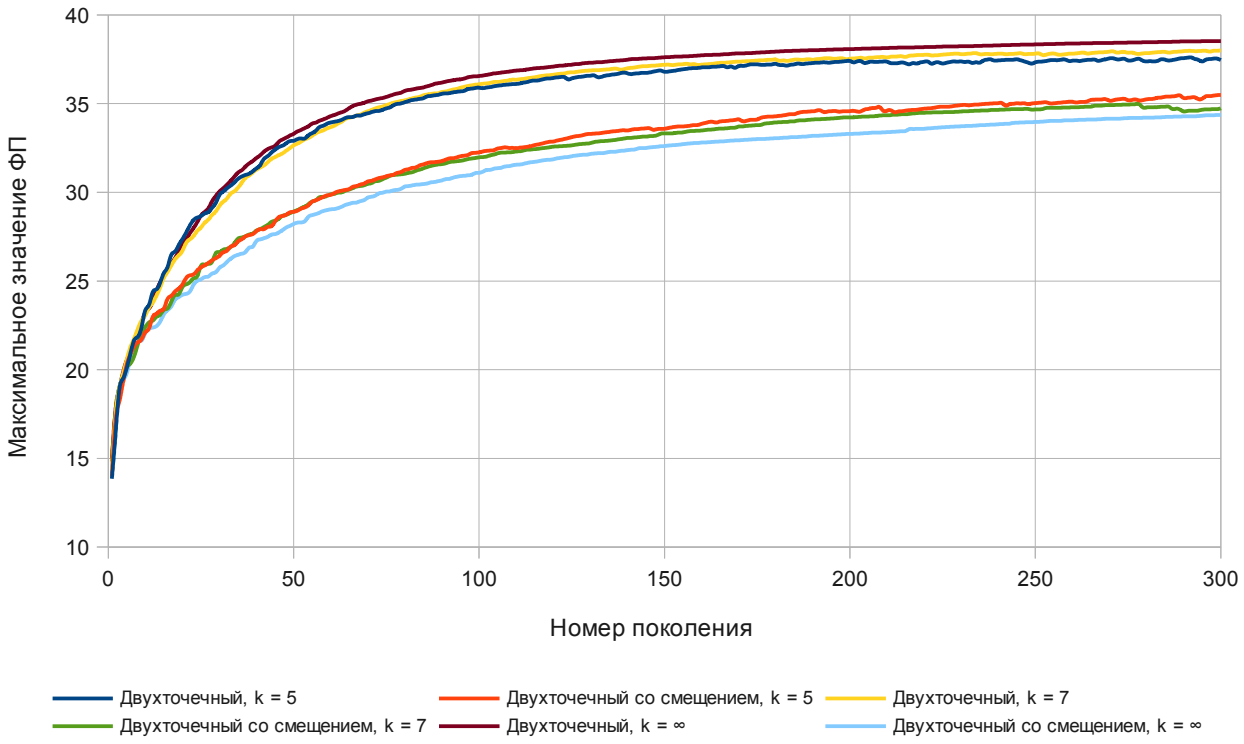

Рис. 2: Зависимость максимального значения ФП от номера поколения при двух видах кроссовера и различных значениях параметра k

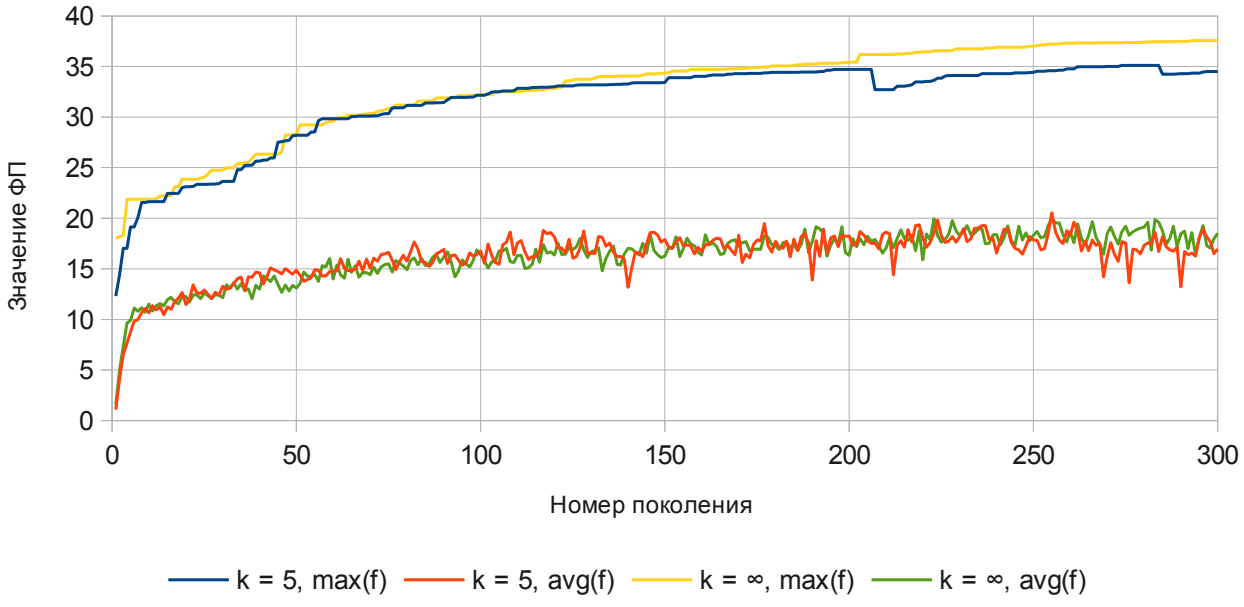

Рис. 3: Зависимость максимального и среднего значения ФП от номера поколения при двухточечном кроссовере со смещением, один запуск

2.6. Оператор кроссовера

На уровне массивов использовались следующие виды кроссовера:

- двухточечный кроссовер;
- двухточечный кроссовер со смещением (особи обмениваются подстроками одной длины с различными смещениями);
- одноточечный кроссовер;
- однородный кроссовер.

Описание приведенных операторов содержится ниже.

Две особи скрещиваются следующим образом. С вероятностью 0,5 они меняются стартовыми состояниями (было проверено, что эта вероятность не имеет большого значения). Далее происходит кроссовер массивов выходных воздействий и массивов переходов для каждого состояния.

Если автомат представлен с помощью сокращенных таблиц, дополнительно производится скрещивание масок значимых состояний. При их скрещивании будем поддерживать постоянным количество единиц в каждой маске.

Используется следующий стандартный алгоритм. Осуществляется параллельный проход по двум маскам. Пары равных битов пропускаются. С каждыми двумя новыми парами, в которых биты различаются, если при этом еще различаются и сами пары, с вероятностью 0,5 происходит инверсия.

3. Результаты работы

Для получения статистики, представленной далее в виде графиков, на исследуемых конфигурациях производилось по 40 запусков ГА с одинаковыми параметрами, после чего данные усреднялись, при этом алгоритм работал до 200-ого поколения. Были проведены несколько запусков ГА до 700-ого поколения на тех же конфигурациях, при этом после 200-ого поколения графики вели себя вполне предсказуемо, поэтому было принято решение в основных исследованиях не изучать работу алгоритма дальше 200-ого поколения.

Все генерируемые автоматы имели семь состояний. Рассматривались автоматы, заданные полными таблицами и сокращенными таблицами. При задании автоматов сокращенными таблицами было экспериментально установлено, что ГА работает лучше, когда число значимых состояний равно пяти.

Было изучено несколько типов кроссовера, которые применялись по отдельности к массиву выходных воздействий (для всех состояний вместе) и к массивам переходов для каждого состояния. Будем называть массивы, к которым применялся кроссовер, строками.

3.1. Двухточечный кроссовер

При двухточечном кроссовере случайным образом выбираются две позиции i и j, после чего строки обмениваются подстроками от i до j, при этом строки считаются зацикленными.

На рис. 4 представлена зависимость максимального значения ФП от номера поколения

для двухточечного кроссовера. Как можно видеть из графика, при представлении автомата сокращенными таблицами за 200 поколений в среднем достигались значения ФП порядка 38. На отдельных запусках достигалось значение 40 (то есть, особь съедает примерно 40% еды на случайном поле).

Представление автомата в виде полных таблиц не оказалось таким удачным, но оно также исследовалось, поскольку в нем вид кроссовера для массивов играет большую роль, чем при представлении в виде сокращенных таблиц (не происходит кроссовер масок значимых состояний, который осуществляется по-другому). При нем достигалось значение ФП около 35.

Для обоих способов представления автомата по истечении 200 поколений все еще наблюдается рост ФП (примерно с одинаковой скоростью).

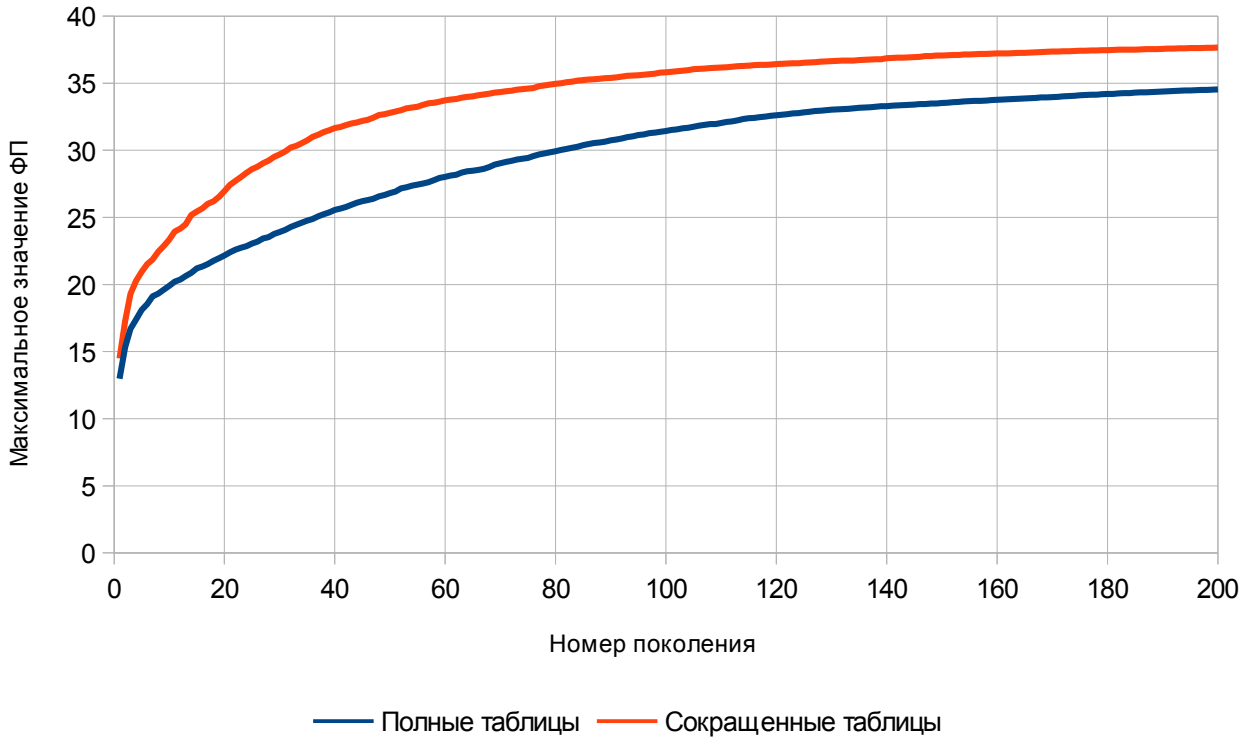

Рис. 4: Зависимость максимальной ФП от номера поколения при двухточечным кроссовере

3.2. Двухточечный кроссовер со смещением

В двухточечном кроссовере со смещением случайно выбирается длина строки и два смещения (для первой и для второй строки), после чего происходит обмен получившимися подстроками.

При обоих способах представления автомата двухточечный кроссовер со смещением оказался заметно хуже обычного двухточечного кроссовера. При использовании сокращенных таблиц за 200 поколений достигалось значение ФП около 34, а при использовании полных — около 28 (см. рис. 5)

Более медленную сходимость алгоритма при использовании данного вида кроссовера можно объяснить следующим образом. У особей отдельные комбинации входных воздействий имеют разную семантику и разную частоту использования. Например, переходы по воздействиям с одной видимой единицей пищи разумно делать в состояние, выходное воздействие которого приблизит муравья к этой единице пищи. При использовании двухточечного кроссовера со смещением для скрещивания массивов переходов местами меняются элементы, соответствующие переходам по разным входным переменным. Для сравнения, у двухточечного кроссовера, очевидно, нет такого недостатка.

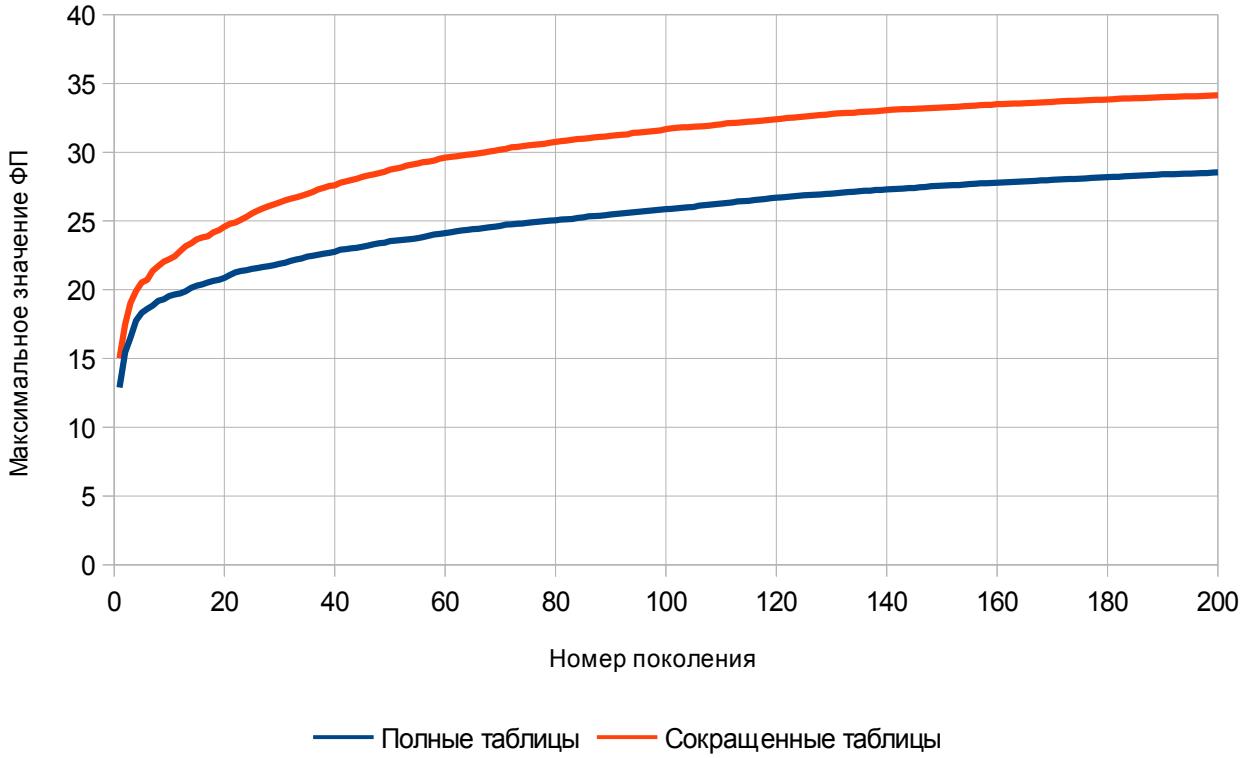

Рис. 5: Зависимость максимальной ФП от номера поколения при двухточечным кроссовере со смещением

3.3. Другие виды кроссовера

Кроме двух видов кроссовера, которые необходимо было исследовать в работе, были рассмотрены одноточечный кроссовер (выбирается случайная позиция, и массивы обмениваются частями от начала до этой позиции) и однородный кроссовер (массивы обмениваются каждым элементом с вероятностью 0,5).

Результаты применения одноточечного и однородного кроссовера оказались почти идентичными результатам применения двухточечного. На рис. 6 графики запусков на полных и сокращенных таблицах практически точно повторяют графики на рис. 4. Лишь для одноточечного кроссовера при полных таблицах видна чуть более медленная сходимость алгоритма.

Заметим, что одноточечный и однородный кроссовер, так же как и двухточечный, не обладают недостатком, упомянутым в разделе 3.2, поэтому подобные результаты не вызывают удивления.

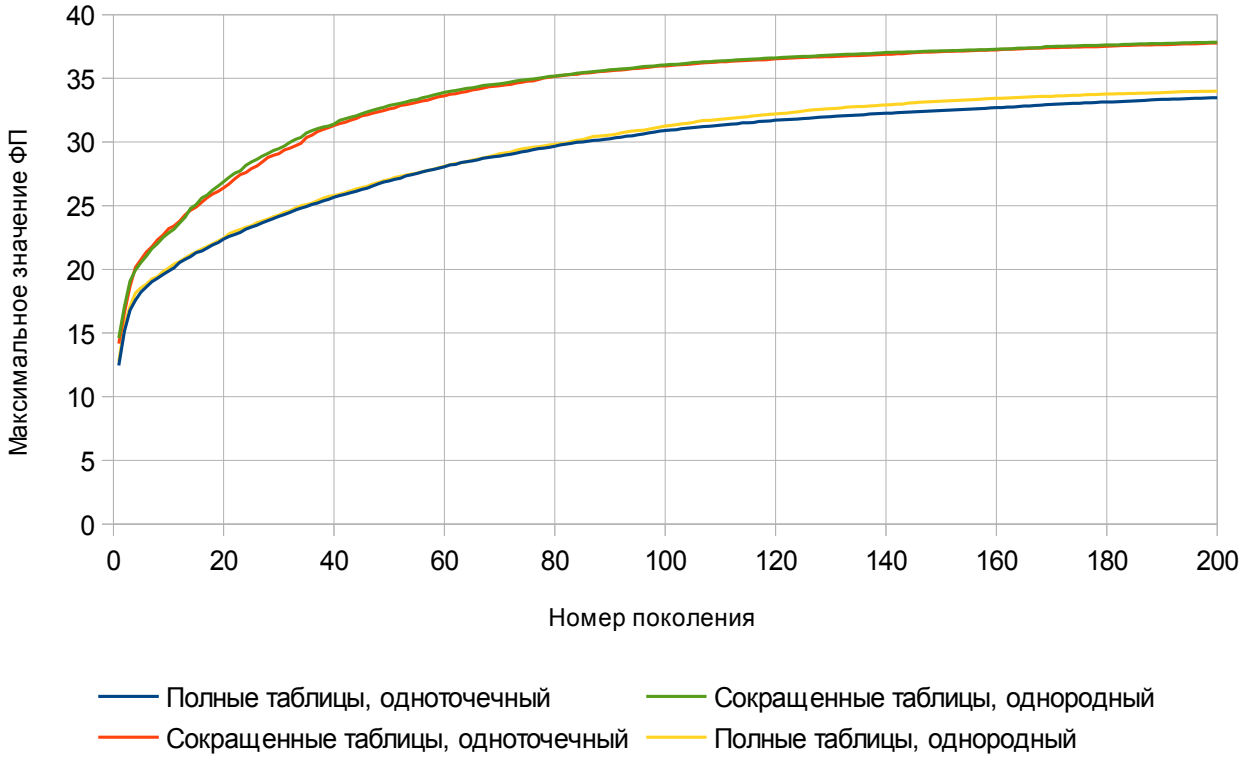

Рис. 6: Зависимость максимальной ФП от номера поколения при одноточечном и однородном кроссовере

Заключение

Как показали результаты исследований, более подходящими (по скорости сходимости ГА) для решения задачи «Умный муравей — 3» оказались двухточечный, одноточечный и однородный кроссовер, при этом среди них не было определено однозначно лучшего. Напротив, при их использовании сходимость ГА была практически одинаковой.

Двухточечный кроссовер со смещением оказался заметно хуже других. Объяснение этому факту приведено в разделе 3.2.

В дополнение к основной цели работы была подтверждена эффективность использования сокращенных таблиц для представления автоматов с большим количеством входных воздействий, о которой говорится, например, в работе [4].

Источники

1. *Н.И. Поликарпова, А.А. Шалыто.* Автоматное программирование. СПб.: Питер, 2009.

2. *Ф.Н. Царев.* Применение метода представления функции переходов с помощью абстрактных конечных автоматов в генетическом программировании.
   http://fppo.ifmo.ru/kmu/kmu6/ВЫПУСК_6/Ready_инф_техн/26_TSAREV_F_N.pdf

3. *Царев Ф.Н., Шалыто А.А.* Применение генетического программирования для генерации автомата в задаче об «Умном муравье» / Сборник трудов IV-ой Международной научно-практической конференции «Интегрированные модели и мягкие вычисления в искусственном интеллекте». Том 2. М.: Физматлит. 2007, с. 590–597.
   http://is.ifmo.ru/genalg/_ant_ga.pdf

4. *Поликарпова Н.И., Точилин В.Н., Шалыто А.А.* Применение генетического программирования для генерации автоматов с большим числом входных переменных //Научно-технический вестник СПбГУ ИТМО. Выпуск 53. Автоматное программирование, с. 24–42 .

Приложение. Исходный код

Исходный код файлов ValueLog.java, Log.java и Config.java не приведен, так как не представляет большого интереса.

1. Файл MooreMachine.java

```
import java.util.Arrays;

public class MooreMachine implements Comparable<MooreMachine>, Cloneable {
    static final int n = Config.getInt("num_states");
    static final int signif = Config.getInt("significant_inputs");
    static final int size = 1 << signif;
    static final double pMutOut = Config.getDouble("p_mut_out");
    static final double pMutStart = Config.getDouble("p_mut_start");
    static final double pMutTrans = Config.getDouble("p_mut_trans");
    static final double pMutMask = Config.getDouble("p_mut_mask");
    static final double pMutIndividual = Config.getDouble("p_mut_individual");
    static final boolean useTotalCrossover =
            Config.getInt("total_crossover") == 1;
    static final CrossoverStrategy crossoverStrategy;

    static {
        switch (Config.getInt("crossover_strategy")) {
        case 1:
            crossoverStrategy = new Crossover1Point();
            break;
        case 2:
            crossoverStrategy = new Crossover2Point();
            break;
        case 3:
            crossoverStrategy = new Crossover2PointFixedLength();
            break;
        case 4:
            crossoverStrategy = new Crossover2PointWithOffset();
            break;
        case 5:
            crossoverStrategy = new CrossoverUniform();
            break;
        default:
            throw new RuntimeException("Invalid crossover strategy!");
        }
    }

    private int start;
    private int[] out;
    private int[][] trans;
    private boolean[][] signifMask;
    private double fitness;

    public int getStart() {
        return start;
```

```
    }

    public double getFitness() {
        return fitness;
    }

    public Object clone() {
        MooreMachine m = new MooreMachine();
        m.start = start;
        m.out = Arrays.copyOf(out, n);
        m.trans = Arrays.copyOf(trans, n);
        m.signifMask = Arrays.copyOf(signifMask, n);
        for (int i = 0; i < n; i++) {
            m.trans[i] = Arrays.copyOf(m.trans[i], size);
            m.signifMask[i] = Arrays.copyOf(m.signifMask[i], Ant.INPUTS);
        }
        return m;
    }

    public boolean[] getMask(int state) {
        return signifMask[state];
    }

    public int transition(int from, int to) {
        return trans[from][to];
    }

    public int action(int state) {
        return out[state];
    }

    private MooreMachine() {
    }

    public static void swap(boolean[] a, boolean[] b, int index) {
        boolean t = a[index];
        a[index] = b[index];
        b[index] = t;
    }

    public void crossover(MooreMachine other) {
        if (Math.random() < 0.5) {
            int t = start;
            start = other.start;
            other.start = t;
        }

        crossoverStrategy.arrayCrossover(out, other.out);
        if (useTotalCrossover) {
            fillTotal();
            other.fillTotal();
            crossoverStrategy.arrayCrossover(total, other.total);
            loadTotal();
            other.loadTotal();
        } else {
```

```
            for (int i = 0; i < n; i++) {
                crossoverStrategy.arrayCrossover(trans[i], other.trans[i]);
            }
        }
        // mask
        if (signif < Ant.INPUTS) {
            for (int i = 0; i < n; i++) {
                maskCrossover(signifMask[i], other.signifMask[i]);
            }
        }
        fitness = other.fitness = -1;
    }

    private static void maskCrossover(boolean[] m1, boolean[] m2) {
        int j = -1;
        boolean firstVal = false;
        for (int i = 0; i < Ant.INPUTS; i++) {
            if (m1[i] == m2[i]) {
                continue;
            }
            if (j == -1) {
                j = i;
                firstVal = m1[i];
            } else {
                if (m1[i] != firstVal && Math.random() < 0.5) {
                    swap(m1, m2, j);
                    swap(m1, m2, i);
                }
                j = -1;
            }
        }
    }

    private int[] total = new int[size * n];

    private void fillTotal() {
        for (int i = 0; i < n; i++) {
            System.arraycopy(trans[i], 0, total, i * size, size);
        }
    }

    private void loadTotal() {
        for (int i = 0; i < n; i++) {
            System.arraycopy(total, i * size, trans[i], 0, size);
        }
    }


    public void mutate() {
        if (Math.random() < pMutStart) {
            start = Main.random().nextInt(n);
        }
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < size; j++) {
                if (Math.random() < pMutTrans) {
```

```
                    trans[i][j] = Main.random().nextInt(n);
                }
            }
            if (Math.random() < pMutOut) {
                out[i] = Main.random().nextInt(3);
            }
        }
        // mutate mask
        if (signif < Ant.INPUTS) {
            for (int i = 0; i < n; i++) {
                mutateMask(signifMask[i]);
            }
        }
        fitness = -1;
    }

    public void repeatedMutate() {
        for (int i = 0; i < 10; i++) {
            mutate();
        }
    }

    private static void mutateMask(boolean[] mask) {
        if (Math.random() < pMutMask) {
            int i, j;
            do {
                i = Main.random().nextInt(Ant.INPUTS);
                j = Main.random().nextInt(Ant.INPUTS);
            } while (mask[i] == mask[j]);
            boolean t = mask[i];
            mask[i] = mask[j];
            mask[j] = t;
        }
    }

    public static MooreMachine getRandom() {
        MooreMachine m = new MooreMachine();
        m.trans = new int[n][];
        m.out = new int[n];
        for (int i = 0; i < n; i++) {
            m.trans[i] = new int[size + 1];
            for (int j = 0; j < size; j++) {
                m.trans[i][j] = Main.random().nextInt(n);
            }
            m.out[i] = Main.random().nextInt(3);
        }
        m.signifMask = new boolean[n][Ant.INPUTS];
        for (int k = 0; k < n; k++) {
            for (int i = 0; i < signif; i++) {
                int j = Main.random().nextInt(Ant.INPUTS);
                if (!m.signifMask[k][j]) {
                    m.signifMask[k][j] = true;
                } else {
                    i--;
                }
```

```
            }
        }
        m.start = Main.random().nextInt(n);
        m.recalcFitness();
        return m;
    }

    public void recalcFitness() {
        fitness = MachineTester.fitness(this);
    }

    @Override
    public int compareTo(MooreMachine other) {
        return Double.compare(fitness, other.fitness);
    }

    @Override
    public String toString() {
        StringBuffer s = new StringBuffer("MooreMachine, f="
                + (fitness == -1 ? "undefined" : (float)fitness)
                + "\nstart=" + start + "\n");
        for (int i = 0; i < n; i++) {
            s.append("#" + i + ": out=" + Ant.ACTION_VALUES[out[i]]
                    + ", trans=" + Arrays.toString(trans[i]) +
                    ", signif=" + Arrays.toString(signifMask[i]) + "\n");
        }
        return s.toString();
    }
}
```

2. Файл Generation.java

```
import java.util.*;

public class Generation {
    static final int size; // even
    static final int threadNum = Config.getInt("threads") - 1;
    static final double elitism = Config.getDouble("elitism");
    static final double pCrossover = Config.getDouble("p_crossover");
    static final int eliteNum;
    static final int smallMutAfter = Config.getInt("small_mut_after");

    static private Thread[] fitnessThreads;

    static {
        int t = Config.getInt("generation_size");
        if (t % 2 == 1) {
            t++;
        }
        size = t;

        t = (int) (size * elitism);
        if (t % 2 == 1) {
            t++;
```

```
        }
        eliteNum = t;
    }

    private int num;
    private MooreMachine[] solutions;
    private double maxFitness;
    private double avgFitness;
    private int stagnation = 0;

    public int getNum() {
        return num;
    }

    private class FitnessThread implements Runnable {
        int from;
        int to;

        FitnessThread(int from, int to) {
            this.from = from;
            this.to = to;
        }

        @Override
        public void run() {
            for (int i = from; i < to; i++) {
                solutions[i].recalcFitness();
            }
        }
    }

    public MooreMachine best() {
        return solutions[size - 1];
    }

    public MooreMachine[] best(int amount) {
        MooreMachine[] arr = new MooreMachine[amount];
        System.arraycopy(solutions, size - amount, arr, 0, amount);
        return arr;
    }

    public double getMaxFitness() {
        return maxFitness;
    }

    public double getAvgFitness() {
        return avgFitness;
    }

    private Generation() {
    }

    public static Generation getRandom() {
        Generation g = new Generation();
        g.solutions = new MooreMachine[size];
```

```
        for (int i = 0; i < size; i++) {
            g.solutions[i] = MooreMachine.getRandom();
        }
        g.recalcStatistics();
        Main.generationCreated(g);
        Main.addFitnessCalls(size);
        return g;
    }

    public void toNextGeneration() {
        Arrays.sort(solutions);
        if (stagnation == smallMutAfter * 2) {
            bigMutation();
            stagnation = 0;
        } else if (stagnation == smallMutAfter) {
            smallMutation();
        } else {
            selection();
        }
        recalcFitness();

        double oldMax = maxFitness;
        recalcStatistics();
        if (maxFitness <= oldMax) {
            stagnation++;
        } else {
            stagnation = 0;
        }

        num++;
        Main.generationCreated(this);
    }

    private void selection() {
        // roulette method
        long time = System.currentTimeMillis();
        double[] right = new double[size];
        double sum = avgFitness * size;
        right[0] = solutions[0].getFitness() / sum;
        for (int i = 1; i < size; i++) {
            right[i] = right[i - 1] + solutions[i].getFitness() / sum;
        }
        right[size - 1] = 1;

        MooreMachine[] next = new MooreMachine[size];

        // crossover and mutation
        for (int i = 0; i < size - eliteNum; i++) {
            double px = Main.random().nextDouble();
            double py = Main.random().nextDouble();

            int x = Arrays.binarySearch(right, px);
            if (x < 0) {
                x = -x - 1;
            }
```

```
            int y = Arrays.binarySearch(right, py);
            if (y < 0) {
                y = -y - 1;
            }

            MooreMachine mx = (MooreMachine) solutions[x].clone();
            MooreMachine my = (MooreMachine) solutions[y].clone();

            if (Math.random() < pCrossover) {
                mx.crossover(my);
            }

            MooreMachine mz = Math.random() < 0.5 ? mx : my;
            mz.mutate();
            next[i] = mz;
        }
        for (int i = size - eliteNum; i < size; i++) {
            next[i] = solutions[i];
        }
        solutions = next;

        Main.addTransformationTime(System.currentTimeMillis() - time);
    }

    private void recalcFitness() {
        long time = System.currentTimeMillis();

        fitnessThreads = new Thread[threadNum];
        int effectiveSize = size - eliteNum;
        for (int i = 0; i < threadNum; i++) {
            FitnessThread th = new FitnessThread(effectiveSize * i /
                    (threadNum + 1),
                    effectiveSize * (i + 1) / (threadNum + 1));
            fitnessThreads[i] = new Thread(th);
        }
        for (int i = 0; i < threadNum; i++) {
            fitnessThreads[i].start();
        }
        new FitnessThread(effectiveSize * threadNum
                / (threadNum + 1), effectiveSize).run();
        for (int i = 0; i < threadNum; i++) {
            try {
                fitnessThreads[i].join();
            } catch (InterruptedException e) {
                fitnessThreads[i].interrupt();
                throw new RuntimeException("Calculation interrupted!");
            }
        }

        Main.addFitnessTime(System.currentTimeMillis() - time);
        Main.addFitnessCalls(effectiveSize);
    }

    private void smallMutation() {
        for (int i = 0; i < size - eliteNum; i++) {
```

```
            solutions[i].repeatedMutate();
        }
    }

    private void bigMutation() {
        for (int i = 0; i < size; i++) {
            if (Main.random().nextDouble() < 0.5) {
                solutions[i].repeatedMutate();
            } else {
                solutions[i] = MooreMachine.getRandom();
            }
        }
    }

    private void recalcStatistics() {
        double max = 0;
        double sum = 0;
        for (int i = 0; i < size; i++) {
            sum += solutions[i].getFitness();
            max = Math.max(max, solutions[i].getFitness());
        }
        avgFitness = sum / size;
        maxFitness = max;
    }

    @Override
    public String toString() {
        return "Generation " + num + ": " + "max(f)="
                + (float)maxFitness + "; avg(f)=" + (float)avgFitness;
    }
}
```

3. Файл CrossoverStrategy.java

```
public abstract class CrossoverStrategy {
    public static void swap(int[] a, int[] b, int index) {
        int t = a[index];
        a[index] = b[index];
        b[index] = t;
    }

    public abstract void arrayCrossover(int[] a, int[] b);
}

class Crossover1Point extends CrossoverStrategy {
    public void arrayCrossover(int[] a, int[] b) {
        int s = a.length;
        int j = Main.random().nextInt(s);
        for (int i = 0; i < j; i++) {
            swap(a, b, i);
        }
```

```
    }
}
class Crossover2Point extends CrossoverStrategy {
    public void arrayCrossover(int[] a, int[] b) {
        int s = a.length;
        int j = Main.random().nextInt(s);
        int k = Main.random().nextInt(s);
        for (int i = j; i != k; i = (i + 1) % s) {
            swap(a, b, i);
        }
    }
}

class Crossover2PointFixedLength extends CrossoverStrategy {
    public void arrayCrossover(int[] a, int[] b) {
        int s = a.length;
        int j = Main.random().nextInt(s);
        int l = s / 2;
        int end = (j + l) % s;
        for (int i = j; i != end; i = (i + 1) % s) {
            swap(a, b, i);
        }
    }
}

class Crossover2PointWithOffset extends CrossoverStrategy {
    public void arrayCrossover(int[] a, int[] b) {
        int s = a.length;
        int o1 = Main.random().nextInt(s);
        int o2 = Main.random().nextInt(s);
        int l = Main.random().nextInt(s);
        for (int i = 0; i < l; i++) {
            int i1 = (o1 + i) % s;
            int i2 = (o2 + i) % s;
            int t = a[i1];
            a[i1] = b[i2];
            b[i2] = t;
        }
    }
}

class CrossoverUniform extends CrossoverStrategy {
    public void arrayCrossover(int[] a, int[] b) {
        int s = a.length;
        for (int i = 0; i < s; i++) {
            if (Math.random() < 0.5) {
                swap(a, b, i);
            }
        }
    }
}
```

4. Файл Ant.java

```
public class Ant {
    public static final int NUMBER_STEPS = 200;
    public static final int FIELD_DIM = 32;
    public static final int INPUTS = 8;
    public static final char[] ACTION_VALUES = { 'L', 'R', 'M' };

    private Direction direction;
    private Cell current;

    private boolean[] ret;
    private Cell[] cells;
    private boolean[][] field;
    private int eaten;
    private int maxFood;


    public enum Direction {
        LEFT,
        RIGHT,
        TOP,
        BOTTOM;

        @Override
        public String toString() {
            switch (this) {
                case LEFT:
                    return "<";
                case BOTTOM:
                    return "v";
                case RIGHT:
                    return ">";
                case TOP:
                    return "^";
            }
            return null;
        }

        public Direction left() {
            switch (this) {
                case LEFT:
                    return BOTTOM;
                case BOTTOM:
                    return RIGHT;
                case RIGHT:
                    return TOP;
                case TOP:
                    return LEFT;
            }
            return null;
        }

        public Direction right() {
```

```
            switch (this) {
                case LEFT:
                    return TOP;
                case BOTTOM:
                    return LEFT;
                case RIGHT:
                    return BOTTOM;
                case TOP:
                    return RIGHT;
            }
            return null;
        }
    }

    public void left() {
        direction = direction.left();
    }

    public void right() {
        direction = direction.right();
    }

    public static class Cell {
        public int x;
        public int y;

        public Cell(int x, int y) {
            this.x = x;
            this.y = y;
        }

        public Cell next(Direction d) {
            switch (d) {
                case LEFT:
                    return new Cell(x, (y + FIELD_DIM - 1) % FIELD_DIM);
                case RIGHT:
                    return new Cell(x, (y + 1) % FIELD_DIM);
                case TOP:
                    return new Cell((x + FIELD_DIM - 1) % FIELD_DIM, y);
                case BOTTOM:
                    return new Cell((x + 1) % FIELD_DIM, y);
            }
            return null;
        }
    }

    public Cell getCurrent() {
        return current;
    }

    public double currentFitness() {
        return eaten * 100.0 / maxFood;
    }

    public Ant(boolean[][] field, int maxFood) {
```

```
        direction = Direction.TOP;
        current = new Cell(0, 0);

        ret = new boolean[8];
        cells = new Cell[8];
        this.field = field;
        this.maxFood = maxFood;
    }

    public boolean[] getInputs() {
        Cell cur = getCurrent();
        Direction dir = direction;
        cells[2] = cur.next(dir);
        cells[0] = cells[2].next(dir);
        cells[1] = cells[2].next(dir.left());
        cells[3] = cells[2].next(dir.right());
        cells[5] = cur.next(dir.left());
        cells[6] = cur.next(dir.right());
        cells[4] = cells[5].next(dir.left());
        cells[7] = cells[6].next(dir.right());
        for (int i = 0; i < 8; i++) {
            ret[i] = field[cells[i].x][cells[i].y];
        }
        return ret;
    }

    public void forward() {
        Cell current = getCurrent().next(direction);
        this.current = current;
        if (field[current.x][current.y]) {
            field[current.x][current.y] = false;
            eaten++;
        }
    }

    @Override
    public String toString() {
        StringBuffer buffer = new StringBuffer();
        Cell current = getCurrent();
        for (int i = 0; i < FIELD_DIM; i++) {
            for (int j = 0; j < FIELD_DIM; j++) {
                if (current.x == i && current.y == j) {
                    buffer.append(direction.toString());
                } else {
                    buffer.append(field[i][j] ? '*' : '.');
                }
            }
            buffer.append('\n');
        }
        return buffer.toString();
    }
}
```

5. Файл MachineTester.java

```
import java.util.*;

public class MachineTester {
    private static int NUM_FIELDS = Config.getInt("num_fields");
    public static double FOOD_PROBABILITY = Config.getDouble("food_probability");
    private static boolean[][][] fields = new boolean[NUM_FIELDS]
            [Ant.FIELD_DIM][Ant.FIELD_DIM];
    private static int[] fieldMaxFood = new int[NUM_FIELDS];
    private static boolean[][][] fieldsCopy;
    private static Random rnd = new Random(678);

    static {
        for (int i = 0; i < NUM_FIELDS; i++) {
            fillField(i, FOOD_PROBABILITY);
        }
    }

    // arr.length = 8
    private static int arrayToInt(boolean[] arr, boolean[] mask) {
        int k = 0;
        int shift = 0;
        for (int i = 0; i < 8; i++) {
            if (mask[i]) {
                k |= (arr[i] ? 1 : 0) << shift++;
            }
         }
        return k;
    }

    private static void fillField(int index, double p) {
        boolean[][] matr = fields[index];
        fieldMaxFood[index] = 0;
        for (int i = 0; i < Ant.FIELD_DIM; i++) {
            for (int j = 0; j < Ant.FIELD_DIM; j++) {
                matr[i][j] = rnd.nextDouble() < p;
                if (matr[i][j]) {
                    fieldMaxFood[index]++;
                }
            }
        }
    }

    public static void restoreFields() {
        fields = fieldsCopy;
        NUM_FIELDS = Config.getInt("num_fields");
    }

    public static void randomizeFields(int num) {
        fieldsCopy = fields;
        if (num != NUM_FIELDS) {
            NUM_FIELDS = num;
            fields = new boolean[NUM_FIELDS][Ant.FIELD_DIM][Ant.FIELD_DIM];
```

```
            fieldMaxFood = new int[NUM_FIELDS];
        }
        for (int i = 0; i < NUM_FIELDS; i++) {
            fillField(i, FOOD_PROBABILITY);
        }
    }

    public static double fitness(MooreMachine m) {
        double sum = 0;
        for (int i = 0; i < NUM_FIELDS; i++) {
            boolean[][] field = Arrays.copyOf(fields[i], Ant.FIELD_DIM);
            for (int j = 0; j < Ant.FIELD_DIM; j++) {
                field[j] = Arrays.copyOf(field[j], Ant.FIELD_DIM);
            }

            Ant ant = new Ant(field, fieldMaxFood[i]);
            int curState = m.getStart();
            for (int j = 0; j < Ant.NUMBER_STEPS; j++) {
                int inputs = arrayToInt(ant.getInputs(), m.getMask(curState));
                curState = m.transition(curState, inputs);
                int action = m.action(curState);
                char out = Ant.ACTION_VALUES[action];
                switch (out) {
                case 'L':
                    ant.left();
                    break;
                case 'R':
                    ant.right();
                    break;
                case 'M':
                    ant.forward();
                    break;
                }
            }
            sum += ant.currentFitness();
        }
        return sum / NUM_FIELDS;
    }
}
```

6. Файл Main.java

```
import java.util.*;

public class Main {

    static class Result {
        double realFitness;
        int fitnessCalls;
        int generations;
        MooreMachine bestIndividual;
        int transTime;
        int fitnessTime;
```

```
    @Override
    public String toString() {
        StringBuffer buffer = new StringBuffer();
        buffer.append("Mut/cross time: " + transTime / 1000.0 + " sec.");
        buffer.append("\nFitness time: " + fitnessTime / 1000.0 + " sec.");
        buffer.append("\nFitness calls: " + fitnessCalls);
        buffer.append("\nGenerations: " + generations);
        buffer.append("\nMax fitness: " +
                (float) bestIndividual.getFitness());
        buffer.append("\nReal fitness: " + (float) realFitness);
        buffer.append("\n");
        return buffer.toString();
    }
}

private static Random rnd = new Random();
private static int fitnessCalls = 0;
private static Log log;

private static int transformationTime;
private static int fitnessTime;

public static void addFitnessCalls(int num) {
    fitnessCalls += num;
}

public static void addTransformationTime(long time) {
    transformationTime += (int)time;
}

public static void addFitnessTime(long time) {
    fitnessTime += (int)time;
}

public static void generationCreated(Generation g) {
    log.log(g.toString());
}

public static Random random() {
    return rnd;
}

public static void main(String[] args) {
    String suffix = args.length == 0 ? "" : args[0];

    int maxI = Config.getInt("stop_at_generation");
    int stopAfterIfNoImprovements =
            Config.getInt("stop_after_if_no_improvements");

    int runs = Config.getInt("runs");
    Result[] results = new Result[runs];
    ValueLog vLog = new ValueLog(runs, suffix);
    log = new Log(suffix);
    for (int r = 0; r < runs; r++) {
```

```
        log.log("Starting run " + r);
        fitnessCalls = 0;
        transformationTime = 0;
        fitnessTime = 0;

        long time = System.currentTimeMillis();
        Generation g = Generation.getRandom();
        double maxF = 0;
        int badGenerations = 0;
        Scanner sc = new Scanner(System.in);
        vLog.add(r, g.getMaxFitness(), g.getAvgFitness());
        for (int i = 1; i < maxI; i++) {
            g.toNextGeneration();
            vLog.add(r, g.getMaxFitness(), g.getAvgFitness());
            if (g.best().getFitness() > maxF) {
                maxF = g.best().getFitness();
                badGenerations = 0;
            } else {
                badGenerations++;
                if (badGenerations == stopAfterIfNoImprovements) {
                    String s = null;
                    do {
                        System.out.println("Finish (y/n)?");
                        s = sc.next().toLowerCase();
                    } while (!s.equals("y") && !s.equals("n"));
                    if (s.equals("y")) {
                        break;
                    } else {
                        badGenerations = 0;
                    }
                }
            }
        }
        MooreMachine best = g.best();
        MachineTester.randomizeFields(2000);
        double realFitness = MachineTester.fitness(best);

        MachineTester.restoreFields();

        results[r] = new Result();
        results[r].transTime = transformationTime;
        results[r].fitnessCalls = fitnessCalls;
        results[r].fitnessTime = fitnessTime;
        results[r].generations = g.getNum() + 1;
        results[r].realFitness = realFitness;
        results[r].bestIndividual = best;
        log.log((System.currentTimeMillis() - time) / 1000.0 + " sec.");
        log.log("\n" + results[r].toString());
        log.log("Best individual:\n" + best.toString());
    }
    double sumF = 0;
    double sumRealF = 0;
    log.log("======================");
    for (int r = 0; r < runs; r++) {
        log.log("\n" + results[r].toString());
```

```
            sumF += results[r].bestIndividual.getFitness();
            sumRealF += results[r].realFitness;
        }
        log.log("Total runs: " + runs);
        log.log("Average fitness: " + (float) (sumF / runs));
        log.log("Average real fitness: " + (float) (sumRealF / runs));
        log.write();
        vLog.write();
    }
}
```