Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики

Д. С. Шевченко

Отчет по лабораторной работе

«Построение управляющих автоматов с помощью генетических алгоритмов»

Вариант №14

Санкт-Петербург

2011

Содержание

1. Постановка задачи

Задача лабораторной работы — исследовать эффективность применения различных вероятностей мутации для генов различного типа при генерации автоматов, решающих задачу об «Умном муравье-3». Эффективность применения мутации определяется значением функции приспособленности, усредненным по нескольким опытам с одинаковыми условиями.

Для решения задачи используется (1+1)-эволюционная стратегия. Способ хранения особи — автомат Мура, представленный сокращенными таблицами.

1.1. Задача об «Умном муравье-3»

В задаче используется квадратное поле размером 32 на 32 клетки, представляющее из себя поверхность тора. В каждой из клеток поля с вероятностью 5% перед запуском муравья располагается еда. Задача муравья — передвигаясь по соседним клеткам, собрать за 200 ходов как можно большую долю еды. За один ход муравей может совершить одно из следующих действий:

- перейти на одну клетку вперёд и, если там находится еда, забрать ее;
- повернуться на 90 градусов по часовой стрелке;
- повернуться на 90 градусов против часовой стрелки.

Перед каждым ходом муравей видит перед собой восемь клеток, из которых четыре могут влиять на его действие (рис. 1). При меньшем количестве значимых полей эффективность алгоритма недостаточна из-за меньшего числа «степеней свободы», при большем — из-за их избытка и, как следствие, увеличения времени поиска при аналогичных результатах.

Рис. 1 — Видимые поля и предикат. Фрагмент визуализатора

2. Автомат Мура

Автомат Мура — детерминированный конечный автомат, выходные воздействия в котором зависят только от номера состояния. В задаче применяется автомат Мура первого рода (рис. 2): при переходе из одного состояния в другое выходное воздействие определяется старым состоянием. Переходы задаются сокращенными таблицами: среди восьми видимых клеток выделяются четыре, входящие в предикат, и для каждого возможного значения предиката из любого состояния определяется ровно один переход.

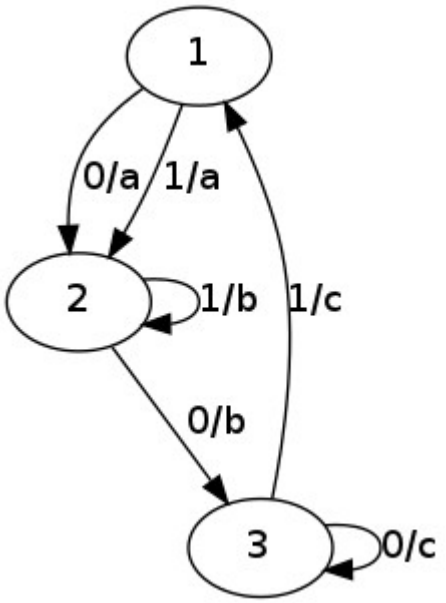

Рис. 2 — Пример автомата Мура со входными воздействиями {0, 1} и выходными воздействиями {a, b, c}

3. Эволюционная стратегия

Алгоритм использует (1+1)-эволюционную стратегию. Первая особь-автомат генерируется случайно. В каждом последующем поколении существующая особь дублируется, и к дублю с некоторыми вероятностями последовательно применяются четыре мутации. После этого автоматы тестируются на 50 одинаковых для всех поколений случайных полях с вероятностью еды 5% в каждой клетке. Лучшая по функции приспособленности особь переходит в следующее поколение.

3.1. Функция приспособленности

Функция приспособленности, или fitness-функция, — мера успешности особи при прохождении испытаний. В данной задаче функция бралась равной усредненной по 50 полям доле еды, собираемой автоматом за 200 ходов.

3.2. Мутации

В работе исследуется зависимость эффективности алгоритма от вероятностей четырех видов мутации.

1. Мутация номера следующего состояния: переход из каждого состояния по каждой возможной маске предиката с некоторой вероятностью сменяется случайным состоянием.
2. Мутация выходного воздействия: действие при переходе из каждого состояния с некоторой вероятностью заменяется на случайное.
3. Мутация предиката: с некоторой вероятностью множество значимых полей обновляется случайным образом.
4. Мутация номера стартового состояния: стартовое состояние с некоторой вероятностью заменяется на случайное.

4. Результаты

В первую очередь, было найдено такое значение *p=8%*, что применение всех мутаций с вероятностями *p* дает наилучший результат (рис. 3). Наиболее эффективными оказались вероятности *L={5%, 8%, 10%, 12%, 15%}*. Было обнаружено, что после 2500 итерации эволюционный процесс практически останавливается.

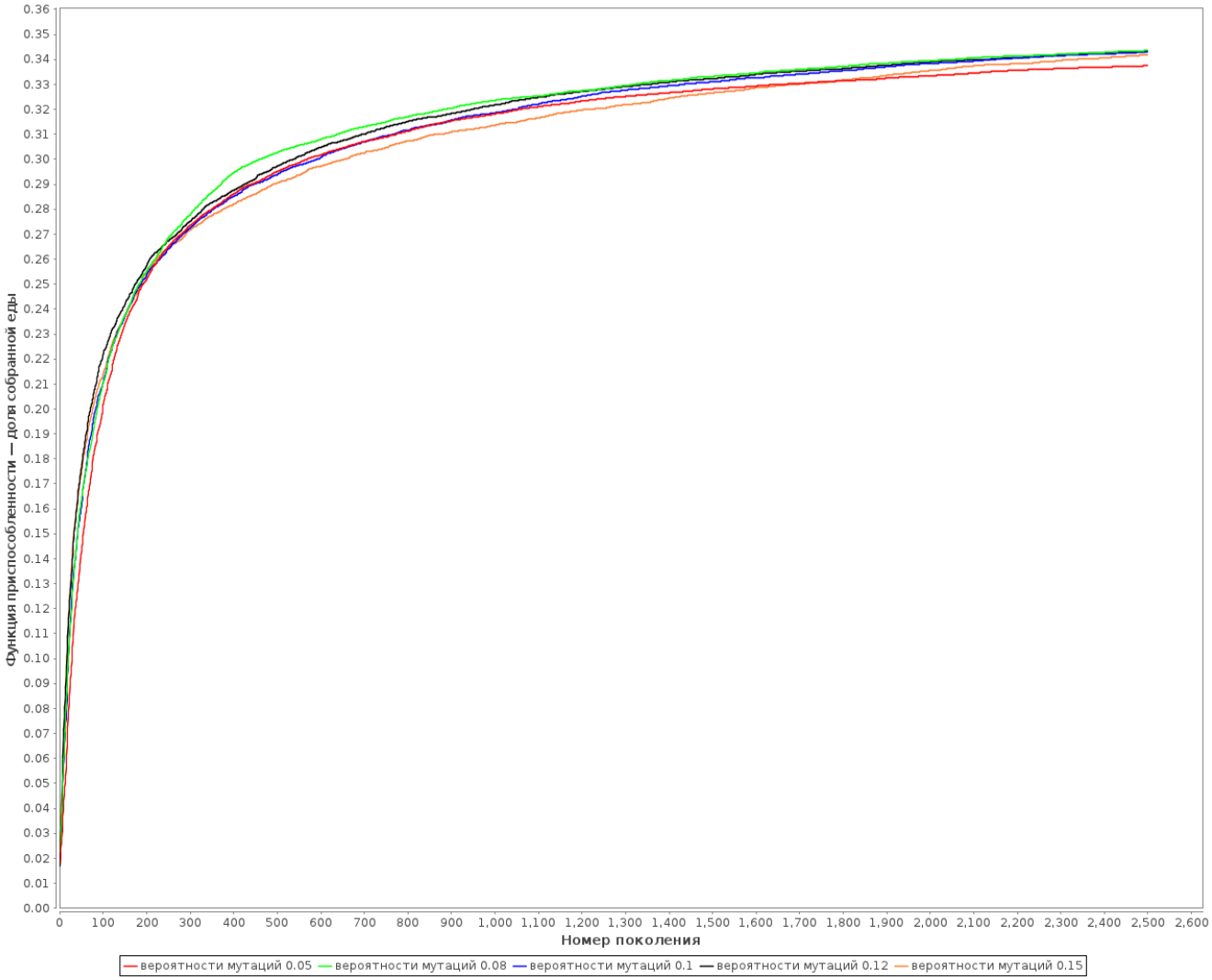

Рис. 3 — Зависимость функции приспособленности, усредненной по 200 запускам, от общей вероятности мутации

Для определения зависимости эффективности алгоритма от вероятностей отдельных видов мутации проводилась серия опытов. В каждом из опытов значение вероятности тестируемой мутации являлось равным одному из значений *L*, а для оставшихся видов мутации — *p*.

4.1. Мутация номера следующего состояния

Применение мутации номера следующего состояния оказывает наибольшее влияние на функцию приспособленности, так как количество генов велико. Лучший результат дает 8% мутация (рис. 4).

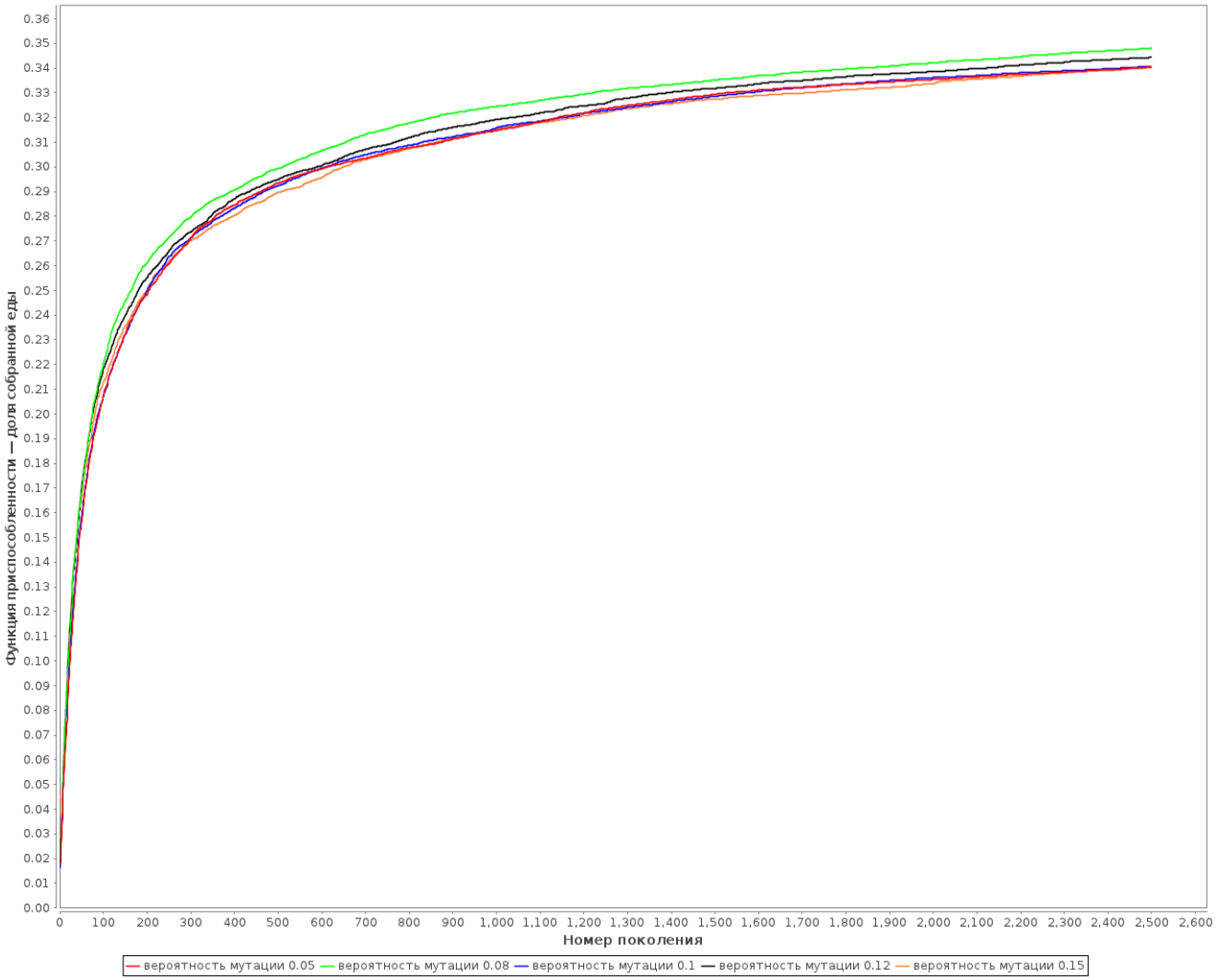

Рис. 4 — Зависимость функции приспособленности, усредненной по 200 запускам, от вероятности мутации номера следующего состояния

4.2. Мутация выходного воздействия

Здесь пятипроцентная мутация наиболее эффективна (рис. 5).

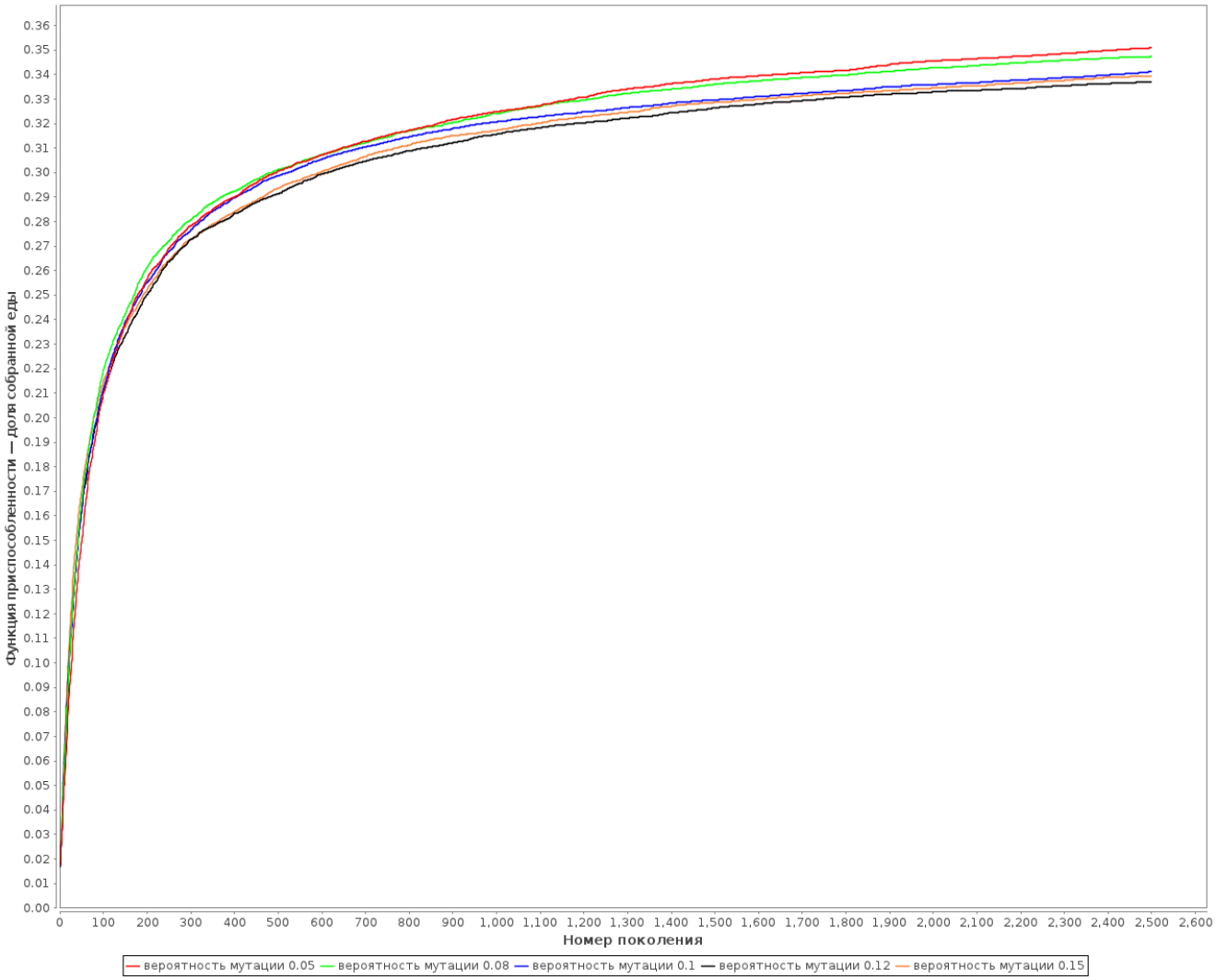

Рис. 5 — Зависимость функции приспособленности, усредненной по 200 запускам, от вероятности мутации выходного воздействия

4.3. Мутация предиката

Лучший прогресс отмечается при вероятности 12% (рис. 6).

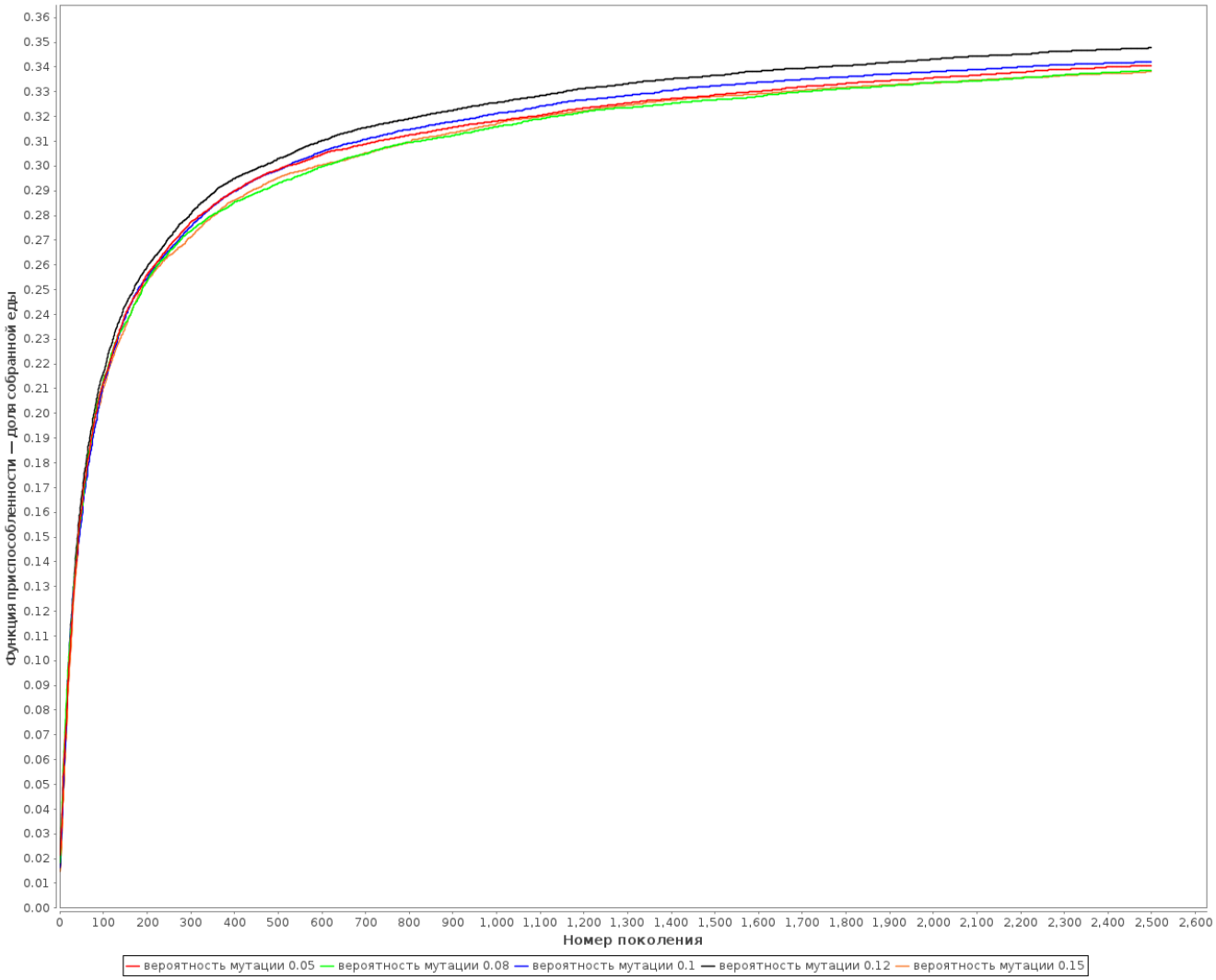

Рис. 6 — Зависимость функции приспособленности, усредненной по 200 запускам, от вероятности мутации предиката

4.4. Мутация номера стартового состояния

Здесь результаты крайне плотны (рис. 7). Выбор стартового состояния менее всего влияет на эффективность алгоритма.

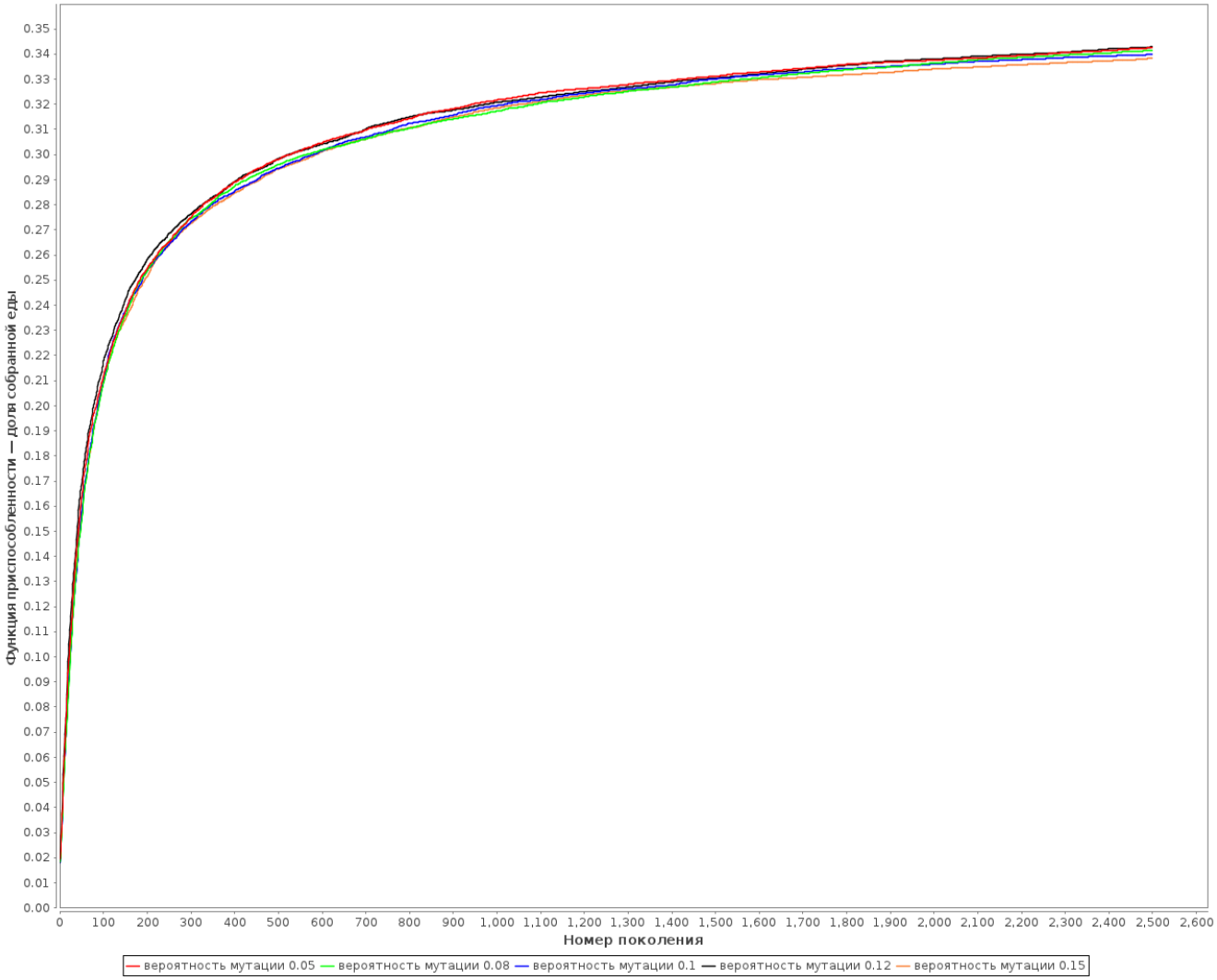

Рис. 7 — Зависимость функции приспособленности, усредненной по 200 запускам, от вероятности мутации номера стартового состояния

5. Проверка результатов

После получения результатов алгоритм был запущен с лучшими вероятностями мутаций. Был получен автомат (рис. 8), в среднем за 200 ходов успевающий собрать около 46,9% еды.

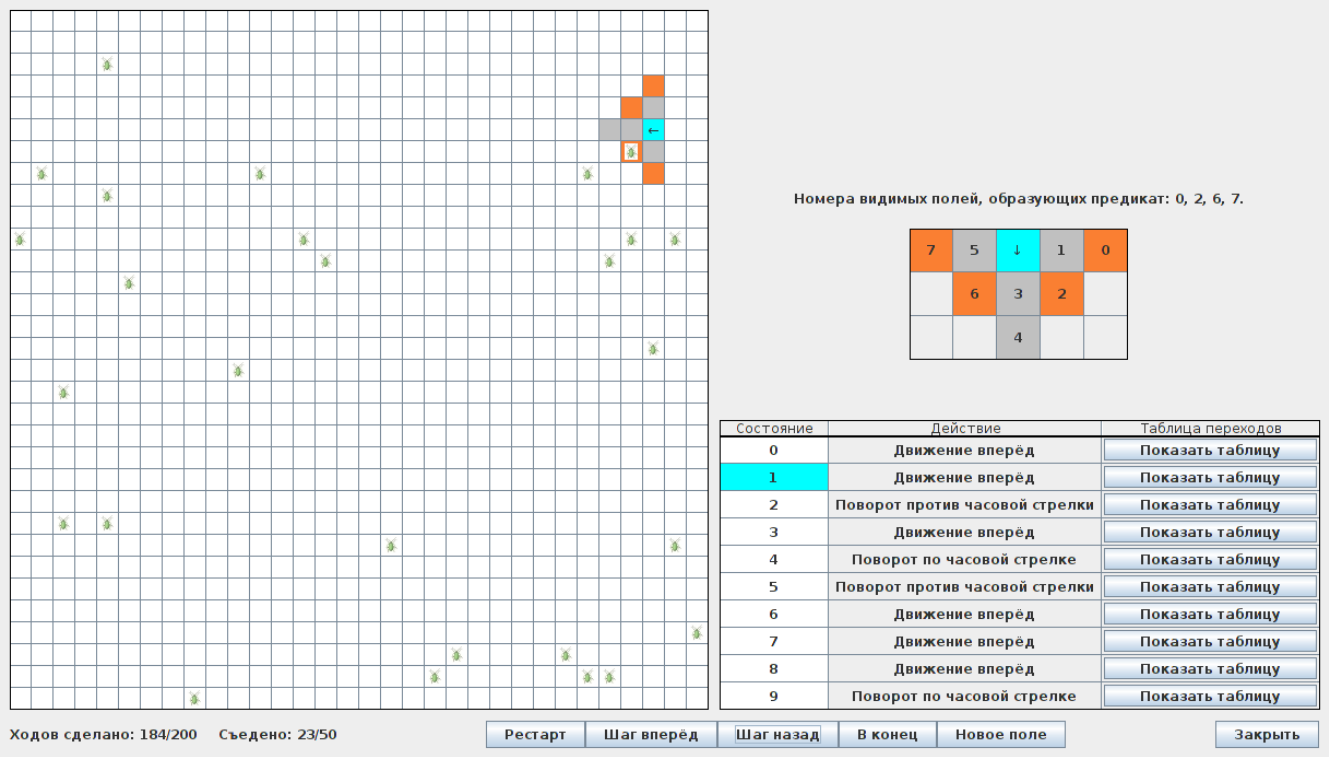

Рис. 8 — Визуализатор поля и лучшего из сгенерированных автоматов

Применение четырех мутаций с оптимальными вероятностями оказывается эффективнее использования везде одинаковой вероятности *p* (рис. 9).

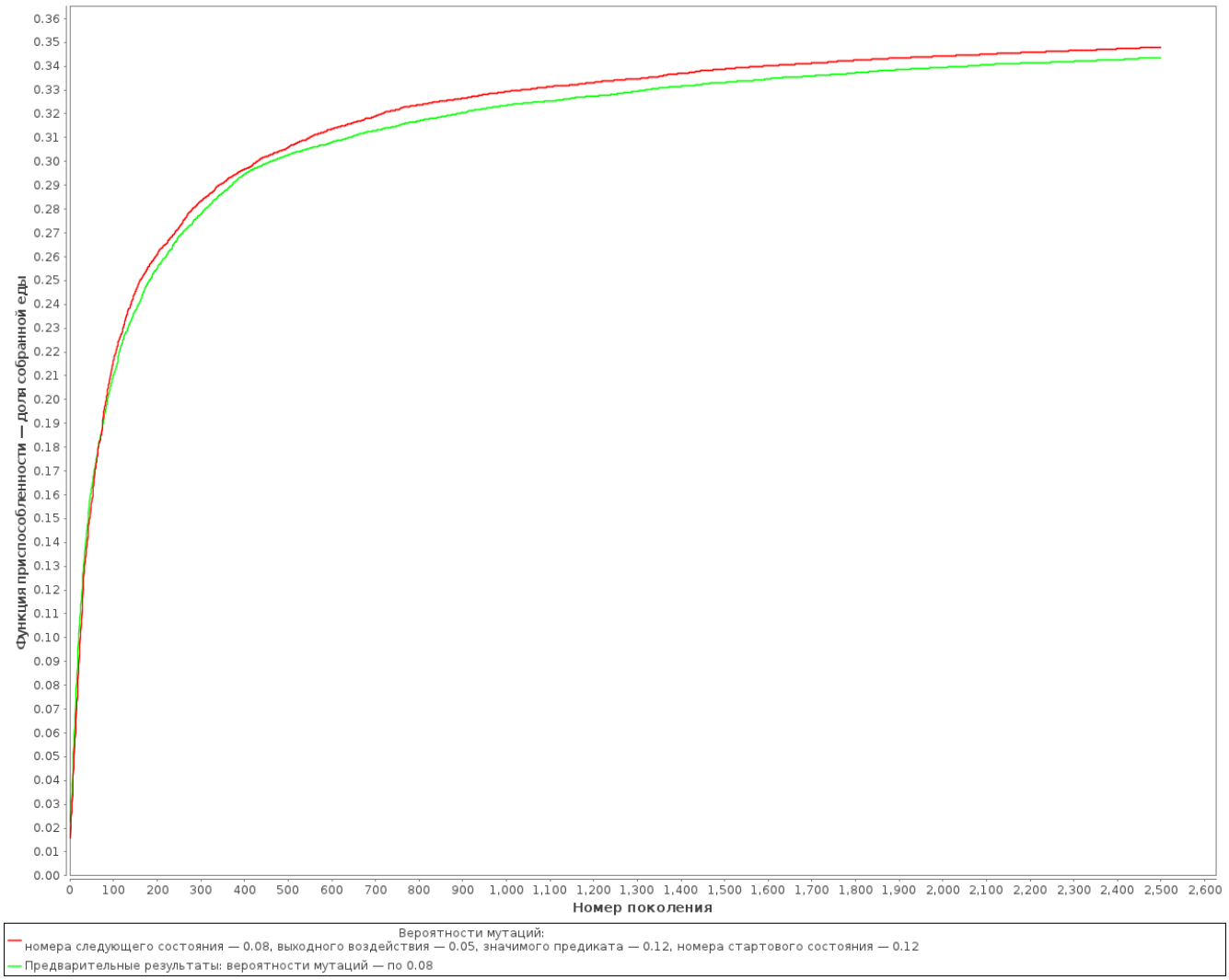

Рис. 9 — Сравнение эффективности мутаций с лучшими найденными вероятностями и с общей оптимальной вероятностью

6. Заключение

Применение разных видов мутации с разными вероятностями положительно влияет на эффективность алгоритма. В качестве лучших значений вероятности можно порекомендовать:

- для мутации номера следующего состояния — 8%;
- для мутации выходного воздействия — 5%;
- для мутации предиката — 12%;
- для мутации номера стартового состояния — 12%.

Исходный код эволюционной стратегии, а также визуализатора графиков, поля и автомата можно найти в репозитории: https://github.com/shevchen/CleverAnt3.

7. Источники

1. *Царев Ф. Н.* Методы представления конечных автоматов в генетических алгоритмах. http://rain.ifmo.ru/~buzdalov/lab-2011/presentations/automata-representation.pdf
2. Sean Luke. The Mersenne Twister in Java. http://www.cs.gmu.edu/~sean/research/
3. David Gilbert, Thomas Morgner. JfreeChart — a free Java chart library. http://www.jfree.org/jfreechart/

8. Приложение. Основные классы реализации

8.1. MooreMachine.java

```
package core;

import java.io.PrintWriter;

import ec.util.MersenneTwister;

public class MooreMachine implements Cloneable {
	private int startState;
	private int significantMask;
	private int[][] nextState;
	private Turn[] moves;

	public MooreMachine(int startState, int significantMask, int[][] nextState,
			Turn[] moves) {
		this.startState = startState;
		this.significantMask = significantMask;
		this.nextState = nextState;
		this.moves = moves;
	}

	public MooreMachine() {
		final MersenneTwister rand = Constants.rand;
		final int states = Constants.STATES_NUMBER;
		startState = rand.nextInt(states);
		significantMask = 0;
		final int vis = Constants.VISIBLE_CELLS;
		for (int i = 0; i < Constants.SIGNIFICANT_INPUTS; ++i) {
			int currentBit = rand.nextInt(vis);
			while (((significantMask >> currentBit) & 1) == 1) {
				currentBit = rand.nextInt(vis);
			}
			significantMask |= (1 << currentBit);
		}
		final int sign = Constants.SIGNIFICANT_INPUTS;
		nextState = new int[states][1 << sign];
		for (int i = 0; i < states; ++i) {
			for (int j = 0; j < (1 << sign); ++j) {
				nextState[i][j] = rand.nextInt(states);
			}
		}
		moves = new Turn[states];
		Turn[] values = Turn.values();
		for (int i = 0; i < states; ++i) {
			moves[i] = values[rand.nextInt(values.length)];
		}
	}

	@Override
```

```
public MooreMachine clone() {
	int[][] nextSt = new int[nextState.length][];
	for (int i = 0; i < nextSt.length; ++i) {
		nextSt[i] = new int[nextState[i].length];
		for (int j = 0; j < nextSt[i].length; ++j) {
			nextSt[i][j] = nextState[i][j];
		}
	}
	Turn[] mv = new Turn[moves.length];
	for (int i = 0; i < moves.length; ++i) {
		mv[i] = moves[i];
	}
	return new MooreMachine(startState, significantMask, nextSt, mv);
}

public int getStartState() {
	return startState;
}

public int getSignificantMask() {
	return significantMask;
}

public Turn getMove(int state) {
	return moves[state];
}

public int getNextState(int state, int mask) {
	return nextState[state][mask];
}

private void nextStateMutation(double p) {
	final MersenneTwister rand = Constants.rand;
	final int states = Constants.STATES_NUMBER;
	for (int i = 0; i < states; ++i) {
		for (int j = 0; j < 1 << Constants.SIGNIFICANT_INPUTS; ++j) {
			if (rand.nextDouble() < p) {
				nextState[i][j] = rand.nextInt(states);
			}
		}
	}
}

private void actionMutation(double p) {
	final MersenneTwister rand = Constants.rand;
	final Turn[] values = Turn.values();
	for (int i = 0; i < Constants.STATES_NUMBER; ++i) {
		if (rand.nextDouble() < p) {
			moves[i] = values[rand.nextInt(values.length)];
		}
	}
}

private void significantPredicateMutation(double p) {
	final MersenneTwister rand = Constants.rand;
	if (rand.nextDouble() < p) {
```

```
                significantMask = 0;
                while (Integer.bitCount(significantMask) < Constants.SIGNIFICANT_INPUTS) {
                        significantMask |= 1 << rand.nextInt(Constants.STATES_NUMBER);
                }
        }
}

private void startStateMutation(double p) {
        final MersenneTwister rand = Constants.rand;
        if (rand.nextDouble() < p) {
                startState = rand.nextInt(Constants.STATES_NUMBER);
        }
}

public void mutate(Mutation m, double p) {
        switch (m) {
        case NEXT_STATE_MUTATION:
                nextStateMutation(p);
                return;
        case ACTION_MUTATION:
                actionMutation(p);
                return;
        case SIGNIFICANT_PREDICATE_MUTATION:
                significantPredicateMutation(p);
                return;
        case START_STATE_MUTATION:
                startStateMutation(p);
                return;
        }
}

public static String getBitString(int mask, int length) {
        char[] ans = new char[length];
        for (int i = 0; i < length; ++i) {
                ans[length - 1 - i] = (char) ('0' + ((mask >> i) & 1));
        }
        return new String(ans);
}

public void print(PrintWriter out) {
        out.println("Start state = " + startState);
        out.print("Significant inputs:");
        final int vis = Constants.VISIBLE_CELLS;
        for (int i = 0; i < vis; ++i) {
                if (((significantMask >> i) & 1) == 1) {
                        out.print(" " + i);
                }
        }
        out.println();
        final int states = Constants.STATES_NUMBER;
        out.print("Moves:");
        for (int i = 0; i < states; ++i) {
                out.print(" " + moves[i]);
        }
        out.println();
        out.printf("%15s%15s%15s", "state", "inputs", "next state");
```

```
                out.println();
                final int sign = Constants.SIGNIFICANT_INPUTS;
                for (int i = 0; i < states; ++i) {
                        for (int j = 0; j < (1 << sign); ++j) {
                                out.printf("%15d%15s%15d", i, getBitString(j,
                                                Constants.SIGNIFICANT_INPUTS), nextState[i][j]);
                                out.println();
                        }
                }
        }
}
```

8.2. Field.java

```
package core;

import java.io.PrintWriter;

import ec.util.MersenneTwister;

public class Field {
        private boolean[][] field;
        private int totalFood;

        public boolean hasFood(int row, int column) {
                return field[row][column];
        }

        private static int modSize(int n) {
                final int size = Constants.FIELD_SIZE;
                if (n < 0) {
                        n += size;
                }
                if (n >= size) {
                        n -= size;
                }
                return n;
        }

        public static Cell[] getVisible(int row, int column, Direction dir) {
                Cell[] result = new Cell[Constants.VISIBLE_CELLS];
                switch (dir) {
                case LEFT:
                        // __7
                        // _65
                        // 43x
                        // _21
                        // __0
                        result[0] = new Cell(modSize(row + 2), column);
                        result[1] = new Cell(modSize(row + 1), column);
                        result[2] = new Cell(modSize(row + 1), modSize(column - 1));
                        result[3] = new Cell(row, modSize(column - 1));
```

```
                result[4] = new Cell(row, modSize(column - 2));
                result[5] = new Cell(modSize(row - 1), column);
                result[6] = new Cell(modSize(row - 1), modSize(column - 1));
                result[7] = new Cell(modSize(row - 2), column);
                return result;
            case RIGHT:
                // 0__
                // 12_
                // x34
                // 56_
                // 7__
                result[0] = new Cell(modSize(row - 2), column);
                result[1] = new Cell(modSize(row - 1), column);
                result[2] = new Cell(modSize(row - 1), modSize(column + 1));
                result[3] = new Cell(row, modSize(column + 1));
                result[4] = new Cell(row, modSize(column + 2));
                result[5] = new Cell(modSize(row + 1), column);
                result[6] = new Cell(modSize(row + 1), modSize(column + 1));
                result[7] = new Cell(modSize(row + 2), column);
                return result;
            case UP:
                // __4__
                // _236_
                // 01x57
                result[0] = new Cell(row, modSize(column - 2));
                result[1] = new Cell(row, modSize(column - 1));
                result[2] = new Cell(modSize(row - 1), modSize(column - 1));
                result[3] = new Cell(modSize(row - 1), column);
                result[4] = new Cell(modSize(row - 2), column);
                result[5] = new Cell(row, modSize(column + 1));
                result[6] = new Cell(modSize(row - 1), modSize(column + 1));
                result[7] = new Cell(row, modSize(column + 2));
                return result;
            case DOWN:
                // 75x10
                // _632_
                // __4__
                result[0] = new Cell(row, modSize(column + 2));
                result[1] = new Cell(row, modSize(column + 1));
                result[2] = new Cell(modSize(row + 1), modSize(column + 1));
                result[3] = new Cell(modSize(row + 1), column);
                result[4] = new Cell(modSize(row + 2), column);
                result[5] = new Cell(row, modSize(column - 1));
                result[6] = new Cell(modSize(row + 1), modSize(column - 1));
                result[7] = new Cell(row, modSize(column - 2));
                return result;
            }
            throw new RuntimeException();
    }

    private static int getNextRow(int row, Direction dir) {
            switch (dir) {
            case LEFT:
                return row;
            case RIGHT:
                return row;
```

```
            case UP:
                return modSize(row - 1);
            case DOWN:
                return modSize(row + 1);
            }
            throw new RuntimeException();
    }

    private static int getNextColumn(int column, Direction dir) {
            switch (dir) {
            case LEFT:
                return modSize(column - 1);
            case RIGHT:
                return modSize(column + 1);
            case UP:
                return column;
            case DOWN:
                return column;
            }
            throw new RuntimeException();
    }

    private static int getVisibleMask(int row, int column, Direction dir,
                boolean[][] left) {
            Cell[] cells = getVisible(row, column, dir);
            int result = 0;
            for (int i = 0; i < cells.length; ++i) {
                result |= ((left[cells[i].row][cells[i].column] ? 1 : 0) << i);
            }
            return result;
    }

    private static int getActualMask(int fullMask, int signMask) {
            final int vis = Constants.VISIBLE_CELLS;
            int currentBit = 0;
            int result = 0;
            for (int i = 0; i < vis; ++i) {
                if (((signMask >> i) & 1) == 1) {
                    result |= ((fullMask >> i) & 1) << (currentBit++);
                }
            }
            return result;
    }

    public Field() {
            final int size = Constants.FIELD_SIZE;
            field = new boolean[size][size];
            final MersenneTwister rand = Constants.rand;
            totalFood = 0;
            for (int i = 0; i < size; ++i) {
                for (int j = 0; j < size; ++j) {
                    if (i == Constants.START_ROW && j == Constants.START_COLUMN) {
                        continue;
                    }
                    field[i][j] = rand.nextDouble() < Constants.FOOD_PROBABILITY;
                    if (field[i][j]) {
```

```
                                        ++totalFood;
                                }
                        }
                }
        }

        public int getTotalFood() {
                return totalFood;
        }

        public static boolean makeStep(MooreMachine auto, AntState as,
                        boolean[][] curField) {
                int visibleMask = getVisibleMask(as.row, as.column, as.dir, curField);
                Turn action = auto.getMove(as.autoState);
                as.autoState = auto.getNextState(as.autoState, getActualMask(
                                visibleMask, auto.getSignificantMask()));
                switch (action) {
                case MOVE:
                        as.row = getNextRow(as.row, as.dir);
                        as.column = getNextColumn(as.column, as.dir);
                        if (curField[as.row][as.column]) {
                                curField[as.row][as.column] = false;
                                ++as.eaten;
                                return true;
                        }
                        return false;
                case ROTATELEFT:
                        as.dir = as.dir.rotateLeft();
                        return false;
                case ROTATERIGHT:
                        as.dir = as.dir.rotateRight();
                        return false;
                }
                throw new RuntimeException();
        }

        public static int simulate(MooreMachine auto, Field f) {
                final int size = Constants.FIELD_SIZE;
                boolean[][] curField = new boolean[size][size];
                for (int i = 0; i < Constants.FIELD_SIZE; ++i) {
                        System.arraycopy(f.field[i], 0, curField[i], 0,
                                        Constants.FIELD_SIZE);
                }
                AntState antState = new AntState(auto.getStartState(),
                                Constants.START_ROW, Constants.START_COLUMN,
                                Constants.START_DIRECTION, 0);
                for (int i = 0; i < Constants.TURNS_NUMBER; ++i) {
                        makeStep(auto, antState, curField);
                }
                return antState.eaten;
        }

        public void print(PrintWriter out) {
                final int size = Constants.FIELD_SIZE;
                for (int i = 0; i < size; ++i) {
                        for (int j = 0; j < size; ++j) {
```

```
                        out.print(field[i][j] ? 'x' : '.');
                    }
                    out.println();
                }
            }
}
```

8.3. Processor.java

```
package main;

import java.util.Arrays;

import core.Constants;
import core.Field;
import core.MooreMachine;
import core.Mutation;
import core.SimulationResult;

public class Processor {
    private Field[] fields;
    private SimulationResult best;
    private ResultSaver rs;
    private double[] prob;

    public Processor(String dirName) {
        this.rs = new ResultSaver(prob, dirName);
    }

    private void updateGeneration(int genNumber) {
        MooreMachine current = best.getAuto().clone();
        for (Mutation m : Mutation.values()) {
            current.mutate(m, prob[m.ordinal()]);
        }
        SimulationResult sr = new SimulationResult(current, 0., 0);
        for (Field f : fields) {
            FitnessCounter.updateFitness(sr, f);
        }
        if (sr.compareTo(best) < 0) {
            best = sr;
        }
        double part = best.getMeanEatenPart();
        rs.saveGeneration(genNumber, part);
    }

    public void run(double[] prob, final int iterations) {
        this.prob = prob;
        final int fieldsN = Constants.FIELDS_IN_GENERATION;
        fields = new Field[fieldsN];
        for (int i = 0; i < fieldsN; ++i) {
            fields[i] = new Field();
        }
```

```
                best = new SimulationResult(new MooreMachine(), 0., 0);
                for (Field f : fields) {
                        FitnessCounter.updateFitness(best, f);
                }
                for (int j = 0; j < iterations; ++j) {
                        updateGeneration(j + 1);
                }
                rs.saveAutomaton(best);
                rs.close();
                System.out.println("Done " + Arrays.toString(prob));
        }
}
```

8.4. PreliminaryAnalyzer.java

```
package main;

import java.util.Arrays;

import core.Constants;
import core.Mutation;

public class PreliminaryAnalyzer {
        public static void main(String[] args) {
                final int thr = Constants.THREADS;
                Thread[] searcher = new Thread[thr];
                for (int i = 0; i < thr; ++i) {
                        searcher[i] = new Thread(new Runnable() {
                                @Override
                                public void run() {
                                        for (int i = 0; i < Constants.SEARCHES_PER_THREAD; ++i) {
                                                for (double p : Constants.MUTATION_PROBABILITIES) {
                                                        double[] prob = new double[Mutation.values().length];
                                                        Arrays.fill(prob, p);
                                                        final String dirName =
Constants.PRELIMINARY_RESULTS_DIR
                                                                        + "/" +
ResultSaver.getDirectoryName(prob);
                                                        new Processor(dirName).run(prob,
                                                                        Constants.SEARCHER_ITERATIONS);
                                                }
                                        }
                                }
                        });
                }
                for (int i = 0; i < thr; ++i) {
                        searcher[i].start();
                }
        }
}
```

8.5. Runner.java

```
package main;

import java.util.Arrays;

import core.Constants;
import core.Mutation;

public class Runner {
	public static void main(String[] args) throws InterruptedException {
		final int thr = Constants.THREADS;
		Thread[] proc = new Thread[thr];
		for (int i = 0; i < thr; ++i) {
			proc[i] = new Thread(new Runnable() {
				@Override
				public void run() {
					for (int i = 0; i < Constants.RUNNINGS_PER_THREAD; ++i) {
						for (Mutation m : Mutation.values()) {
							for (double p : Constants.MUTATION_PROBABILITIES) {
								double[] prob = new
double[Mutation.values().length];
								Arrays.fill(prob,

	Constants.COMMON_MUTATION_PROBABILITY);
								prob[m.ordinal()] = p;
								final String dirName = Constants.RESULTS_DIR
										+ "/" + m + "/"
										+
ResultSaver.getDirectoryName(prob);
								new Processor(dirName).run(prob,
										Constants.ITERATIONS);
							}
						}
					}
				}
			});
		}
		for (int i = 0; i < thr; ++i) {
			proc[i].start();
		}
	}
}
```

8.6. BestAutomataSearcher.java

```
package main;

import core.Constants;

public class BestAutomataSearcher {
```

```
	public static void main(String[] args) {
		final int thr = Constants.THREADS;
		Thread[] searcher = new Thread[thr];
		for (int i = 0; i < thr; ++i) {
			searcher[i] = new Thread(new Runnable() {
				@Override
				public void run() {
					for (int i = 0; i < Constants.SEARCHES_PER_THREAD; ++i) {
						new Processor(Constants.BEST_AUTO_DIR).run(
					Constants.BEST_MUTATION_PROBABILITIES,
							Constants.SEARCHER_ITERATIONS);
					}
				}
			});
		}
		for (int i = 0; i < thr; ++i) {
			searcher[i].start();
		}
	}
}
```