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

С. В. Казаков

Отчет по лабораторной работе «Применение генетических алгоритмов для построения автоматов в задаче «Умный муравей»

Вариант № 9

Санкт-Петербург
2009

Оглавление

Введение

В данной лабораторной работе изучается применение генетических алгоритмов для генерации конечных автоматов как решений выбранной задачи. В качестве примера взята *задача об «Умном муравье»*.

Результатом работы являются *автоматы Мили*, построенные с помощью генетического алгоритма, которые управляют муравьем в задаче.

При выполнении лабораторной работы использовалась *программа «Виртуальная лаборатория»* [1], написанная студентами кафедры «Компьютерные технологии» СПбГУ ИТМО. Она позволяет реализовывать генетические алгоритмы и особи для них в виде плагинов. Все исходные коды в данной работе написаны на языке программирования *Java*.

1. Постановка задачи

Задача лабораторной работы – построить близкий к оптимальному автомат Мили, решающий задачу об «Умном муравье». Оптимальность заключается в том, что муравей, управляемый данным автоматом, должен съесть всю еду. Из всех таких муравьев наиболее близким к оптимальному будет тот, который затратил на еду меньше шагов.

Для решения задачи используется клеточный генетический алгоритм. Способ представления особи – конечный автомат Мили, с его представлением в виде графа переходов.

1.1. Задача об «Умном муравье»

В задаче об «Умном муравье» [2] рассматривается игровое поле, состоящее из клеток (рис. 1).

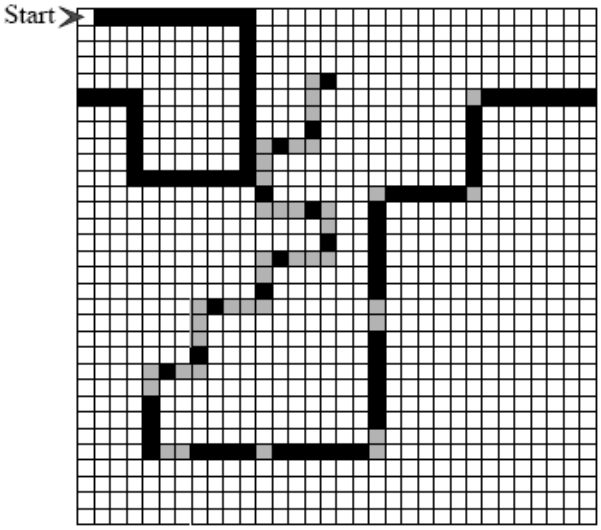


Рис. 1. Игровое поле

Поле имеет размеры 32 на 32 клетки и располагается на поверхности тора. Некоторые клетки поля пусты, некоторые – содержат по одному яблоку. Всего на поле 89 яблок.

Муравей начинает свое движение из клетки, помеченной как «Start». За один ход муравей может определить, есть ли в клетке перед ним яблоко, и выполнить одно из следующих действий:

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

Максимальное число шагов – 200. Цель – создать муравья с фиксированным числом состояний, который за минимальное число шагов съест все яблоки.

1.2. Автомат Мили

Автомат Мили – это конечный детерминированный автомат, генерирующий выходные действия в зависимости от текущего состояния и входного сигнала. Пример такого автомата приведен на рис. 2.

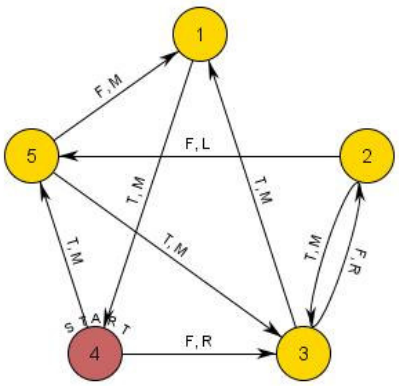


Рис. 2. Автомат Мили

Над каждой дугой расположена пара значений – входное воздействие и действие муравья, причем выходное действие зависит не только от состояния, в котором находится автомат, но и от входного воздействия.

Входные воздействия обозначаются следующим образом:

- T – перед муравьем есть еда;
- F – перед муравьем нет еды.

Действия обозначаются так:

- M – сделать шаг вперед и съесть еду в новой клетке, если она была там;
- L – повернуть налево;
- R – повернуть направо.

2. Генетический алгоритм

Для задачи поиска оптимального автомата, управляющего муравьем, требуется построить генетический алгоритм [3]. Работа генетического алгоритма в общем случае состоит из нескольких фаз.

В начале происходит генерация первого поколения особей. Далее алгоритм начинает выполнение итеративного процесса – на каждой итерации алгоритм строит следующее поколение из предыдущего.

При этом применяются операции:

- *Отбор* – из предыдущего поколения выбирается часть особей. Для сравнения особей между собой алгоритм использует *функцию приспособленности*, которая сопоставляет каждой особи число, определяющее ее приспособленность;
- *Скрещивание* – по двум особям-родителям создаются две новые особи;
- *Обычная мутация* – случайным образом изменяется строение некоторых особей;
- *Большая мутация* – большая часть всех особей заменяется новыми.

2.1. Представление особи

Особями в данном алгоритме являются автоматы Мили. Автомат Мили представляется в виде графа переходов, который реализуется в виде двумерного массива `Transition[][]transitions`. Этот массив для каждой пары, состоящей из текущего состояния автомата и входного воздействия, хранит переход (выходное действие и новое состояние).

2.2. Клеточный генетический алгоритм

Используемый генетический алгоритм является «клеточным». Поколение представляется в виде квадратного тора, в каждой ячейке которого находится особь. Генетический алгоритм работает параллельно с каждой особью. Для каждой особи применяется метод отбора, в котором также участвуют четыре соседа особи.

2.3. Генерация нового поколения

Начальное поколение генерируется из особей, созданных случайным образом. Рассмотрим процесс генерации нового поколения из текущего. Возможны два пути генерации.

Первый путь состоит в замене большей части особей новыми, случайно сгенерированными. В этом состоит операция большой мутации. Она должна выполняться редко для того, чтобы дать время новому поколению на развитие. В данной реализации указанная операция выполняется один раз в заданное число поколений.

При выборе второго пути, во-первых, строится промежуточное поколение при помощи заранее выбранного метода отбора. Во-вторых, к построенному промежуточному поколению применяются методы скрещивания и мутации, и получившиеся особи добавляются в новое поколение.

2.3.1. Методы отбора

В данной реализации возможно использование *отбора методом рулетки* или *методом турнира*.

В методе рулетки выполняется следующее:

1. Для каждой особи рассчитывается вероятность ее выбора по формуле

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j},$$

где f_i – значение функции приспособленности `i`-той особи, `N` – число особей в поколении.

2. В соответствии с полученным распределением алгоритм случайным образом выбирает особь из текущего поколения и добавляет ее в промежуточное поколение.

В методе турнира выбор новой особи, которую требуется поместить в промежуточное поколение, выполняется следующим образом. Случайным образом выбирается `k` особей из текущего поколения. Между ними проводится турнир, и определяется лучшая особь. В данной реализации `k = 2`.

2.3.2. Метод скрещивания особей

При скрещивании алгоритм порождает две новые особи из двух особей-родителей. Состояния автоматов нумеруются числами от `1` до `S` (у всех автоматов одинаковое число состояний). Рассмотрим процедуру скрещивания. Для всех номеров состояний автомата `i` (от `1` до `S`) выполняются шаги:

1. Рассматривается состояние с номером `i` первого родителя и состояние с тем же номером второго. Запоминаются переходы из этих состояний.
2. Рассматривается состояние с одинаковым номером `i` в автоматах-потомках. Случайным образом переходы особей-родителей распределяются среди состояний особей-потомков.

2.3.3. Мутация особей

Мутация особи-автомата заключается в случайном выборе состояния и случайном изменении одного из переходов из данного состояния. Также с некоторой вероятностью изменяется начальное состояние автомата на случайно выбранное.

2.4. Функция приспособленности

Функция приспособленности вычисляется по формуле

```
Fitness = Apples – Steps/200,
```

где `Apples` – число яблок, съедаемых муравьем за 200 шагов, `Steps` – номер шага, на котором муравей съедает последнее яблоко.

Таким образом, муравей, съедающий большее число яблок за 200 шагов, чем другие, всегда будет иметь большее значение функции приспособленности, а из муравьев, съедающих одинаковое число яблок, большая функция приспособленности будет у того, кто потратил на поедание меньше шагов.

3. Построенные автоматы

В результате работы генетического алгоритма были получены автоматы Мили, которые решают задачу об «Умном муравье». Были проведены исследования для автоматов из шести, семи и восьми состояний.

Результаты показаны в табл. 1.

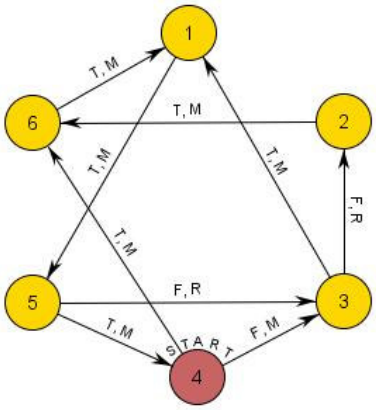
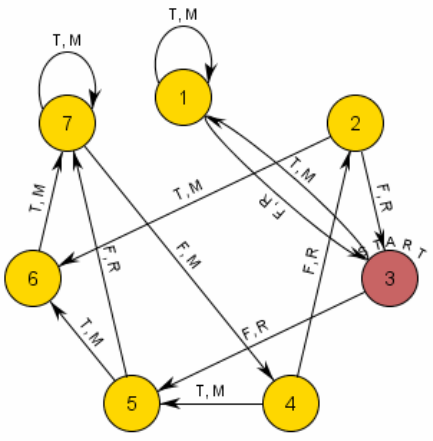
Таблица 1. Результаты

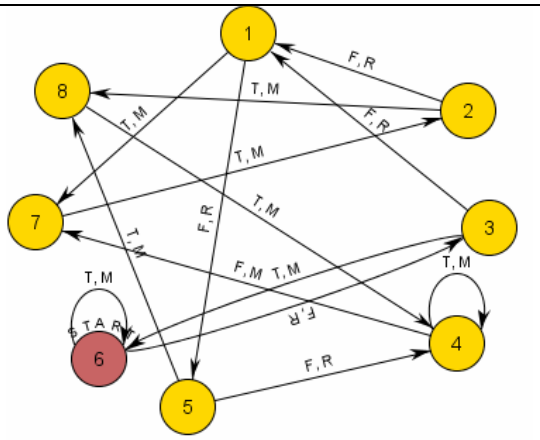
Число состояний	Значение функции приспособленности	Комментарий
6	84.055	Съедает 85 яблок за 189 шагов
7	87.035	Съедает 88 яблок за 193 шага
8	88.01	Съедает все 89 яблок за 198 шагов

3.1. Графы переходов

В табл. 2 представлены графы переходов полученных автоматов.

Таблица 2. Построенные автоматы

Число состояний	Граф переходов
6	
7	

8	

Для удобства рассмотрения автоматов повторим семантику используемых обозначений. Входные воздействия обозначаются следующим образом:

- T – перед муравьем есть еда;
- F – перед муравьем нет еды.

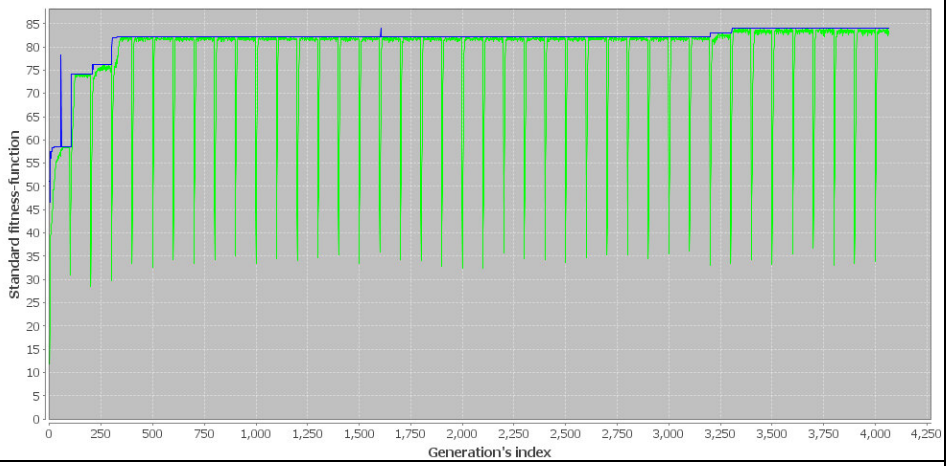
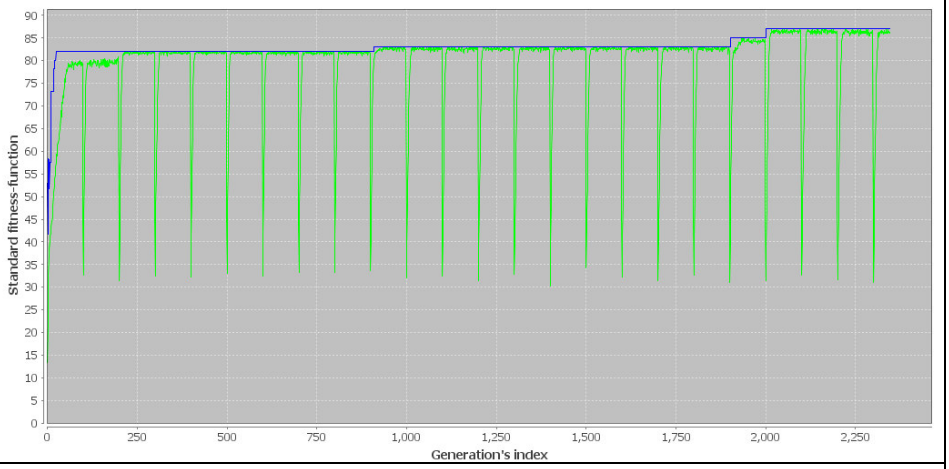
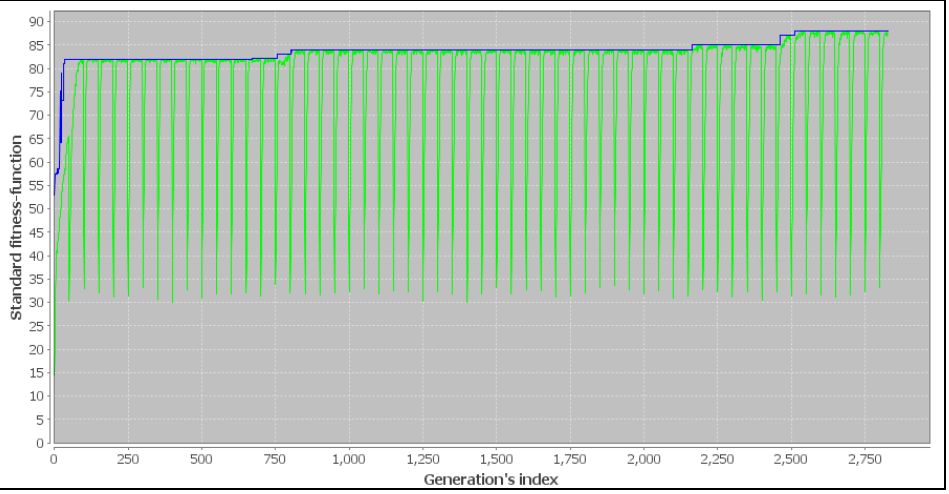
Действия обозначаются так:

- M – сделать шаг вперед и съесть еду в новой клетке, если она была там;
- L – повернуть налево;
- R – повернуть направо.

3.2. Графики максимального и среднего значений функции приспособленности

Графики максимального и среднего значений функции приспособленности среди особей поколения приведены в табл. 3.

Таблица 3. Графики максимального (синяя линия) и среднего (зеленая линия) значений функции приспособленности

Число состояний	График
6	
7	
8	

Заметим, что график среднего значения функции приспособленности очень часто скачет. Связано это с большой мутацией. Именно после большой мутации новые, случайно сгенерированные особи имеют близкое к нулю значение функции приспособленности, что делает среднее значение небольшим по сравнению с максимальным.

Также можно заметить, что в большинстве случаев именно после большой мутации максимальное значение функции приспособленности увеличивается. Этот факт свидетельствует о значительной пользе большой мутации в используемом генетическом алгоритме.

Заключение

Результаты лабораторной работы показали, что генетические алгоритмы очень эффективны для поиска оптимального решения задачи об «Умном муравье».

При помощи клеточного генетического алгоритма были получены автоматы Мили, решающие задачу об «Умном муравье».

Результаты научных исследований, опубликованные в работе [4], показывают, что полученные автоматы очень близки к оптимальным решениям. Наиболее оптимальный автомат Мили из шести состояний съедает 85 яблок, что было доказано в работе [4] алгоритмом полного перебора. Автомат из шести состояний, построенный в данной работе, тоже съедает 85 яблок. Автомат же из восьми состояний, построенный в данной лабораторной работе, полностью решает задачу об «Умном муравье», съедая все 89 яблок.

Проведенные исследования показали значение большой мутации в генетическом алгоритме, благодаря которой были получены столь «хорошие» автоматы.

Источники

1. *Инструкция по созданию plugin'ов к виртуальной лаборатории* http://svn2.assembla.com/svn/not_instrumental_tool/docs/pdf/interface_manual.pdf
2. *Бедный Ю. Д., Шалыто А. А*. Применение генетических алгоритмов для построения автоматов в задаче «Умный муравей». http://is.ifmo.ru/works/_ant.pdf
3. *Яминов Б.* Генетические алгоритмы. http://rain.ifmo.ru/cat/view.php/theory/unsorted/genetic-2005
4. *Царев Ф. Н., Шалыто А. А.* Применение генетических алгоритмов для построения автоматов с минимальным числом состояний для задачи об «Умном муравье» / Тезисы научно-технической конференции «Научно-программное обеспечение в образовании и научных исследованиях». СПбГУ ПУ. 2008, с. 209–215. http://is.ifmo.ru/download/2008-02-25_tsarev_shalyto.pdf

Приложение. Исходные коды

Клеточный генетический алгоритм

Файл *CellularGA.java*

```
package laboratory.plugin.algorithm.cellular;

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

import java.util.*;

/**
 * Клеточный генетический алгоритм
 *
 * @author Сергей Казаков, группа 3539, СПбГУ ИТМО.
 */
public class CellularGA<I extends Individual> implements Algorithm<I> {
      private FitIndividual<I>[][] generation;       // Текущее поколение

      private final int tableLength;                 // Размер (длина и высота)
таблицы (тора)
      private final double mutationProbability;      //   Вероятность   обычной
мутации для особи
      private final int bigMutationPeriod;     //    Период    между    большими
мутациями всего поколения
      private final double newIndividualsPartBig;    // Часть новых особей при
большой мутации

      private final IndividualFactory<I> factory;    //   Выбранный   метод   для
создания новых особей

      private final Mutation<I> mutation;      // Выбранный метод мутации
      private final Crossover<I> crossover;          //     Выбранный     метод
скрещивания
      private final Selection<I> selection;          // Выбранный метод отбора
      private final Fitness<I> fitness;  //  Выбранный  метод  подсчета  функции
приспособленности

      private final Random random;  // Генератор случайных чисел

      private final int[] dx = {0,  0, 1, -1}; //  Возможные  смещения  координат
клетки для просмотра соседних клеток
      private final int[] dy = {1, -1, 0,  0};

      private int indexGeneration;       // Номер текущего поколения
      private double maxF;               //   Максимальное   значение   функции
приспособленности

      public CellularGA(final int tableLength, final double mutationProbability,
final int bigMutationPeriod,
                  final double newIndividualsPartBig,
                  final IndividualFactory<I> factory,
                  final Mutation<I> mutation, final Crossover<I> crossover,
                  final Selection<I> selection, final Fitness<I> fitness) {
```

```
		// Копируем выбранные параметры и методы
		this.tableLength = tableLength;
		this.mutationProbability = mutationProbability;
		this.bigMutationPeriod = bigMutationPeriod;
		this.newIndividualsPartBig = newIndividualsPartBig;

		this.factory = factory;

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

		// Создаем первое поколение
		generation = new FitIndividual[tableLength][tableLength];

		for (int i = 0; i < tableLength; i++) {
			for (int j = 0; j < tableLength; j++) {
				generation[i][j] = construct(factory.getIndividual());
			}
		}

		// Инициализация переменных
		indexGeneration = 1;
		maxF = 0;
		random = new Random();
	}

	/**
	 * Возвращает лучшую особь из двух переданных
	 */
	private FitIndividual<I> winner(FitIndividual<I> a1, FitIndividual<I> a2)
{
		return (a1.compareTo(a2) < 0) ? a1 : a2;
	}

	/**
	 * Конструирует особь с вычисленной функцией приспособленности по особи
	 */
	private FitIndividual<I> construct(I i) {
		return new FitIndividual<I>(i, fitness.apply(i));
	}

	/**
	 * Приводит индекс к необходимому диапазону (0 .. tableLength - 1)
	 */
	private int norm(int i) {
		return (i + tableLength) % tableLength;
	}

	/**
	* Находит позицию лучшей особи в торе
	*/
	int[] bestIndividualPosition() {
		int wi = 0, wj = 0;
		for (int i = 0; i < tableLength; i++) {
			for (int j = 0; j < tableLength; j++) {
				if (generation[i][j].fitness >
			generation[wi][wj].fitness) {
					wi = i;
					wj = j;
				}
			}
		}
```

```
      return new int[]{wi, wj};
}

/*
* Создает новое поколение
*
* @see laboratory.common.genetic.Algorithm#nextGeneration()
*/
public void nextGeneration() {
      FitIndividual<I>[][]          newGeneration          =          new
FitIndividual[tableLength][tableLength];

      if (indexGeneration % bigMutationPeriod == 0) {
            bigMutation(newGeneration);   // Применяем большую мутацию
      } else {
            for (int i = 0; i < tableLength; i++) {
                  for (int j = 0; j < tableLength; j++) {
                        // Выбираем особь для скрещивания
                        List<FitIndividual<I>>       list       =       new
                  ArrayList<FitIndividual<I>>(4);
                        for (int k = 0; k < 4; k++) {
                              list.add(generation[norm(i  +  dx[k])][norm(j
                        + dy[k])]);
                        }
                        FitIndividual<I>    w    =    selection.apply(list,
                  1).get(0);

                        // Скрещиваем выбранную особь с текущей
                        List<I>                     s                     =
                  crossover.apply(Arrays.asList(generation[i][j].ind,
                  w.ind));

                        // Выбираем лучшую из полученных особей
                        w = construct(s.get(0));
                        for (int k = 1; k < s.size(); k++) {
                              w = winner(w, construct(s.get(k)));
                        }

                        //     Применяем     мутацию     с     вероятностью
                  mutationProbability
                        if (random.nextDouble() < mutationProbability) {
                              w = construct(mutation.apply(w.ind));
                              if (random.nextDouble() < 0.2) {
                                    w = construct(mutation.apply(w.ind));
                              }
                        }

                        newGeneration[i][j] = w;
                  }
            }
      }

      // Новое поколение готово
      generation = newGeneration;
      indexGeneration++;
}

/*
 * Возвращает текущее поколение
 *
 * @see laboratory.common.genetic.Algorithm#getGeneration()
 */
public List<I> getGeneration() {
      List<I> ans = new ArrayList<I>(tableLength * tableLength);
```

```
            for (int i = 0; i < tableLength; i++) {
                  for (int j = 0; j < tableLength; j++) {
                        ans.add(generation[i][j].ind);
                  }
            }

            int[] wp = bestIndividualPosition();
            FitIndividual<I> fi = generation[wp[0]][wp[1]];
            if (maxF < fi.fitness) {
                  maxF = fi.fitness;
                  System.out.println("Max F = " + maxF);
            }

            I ci = ans.get(0);
            ans.set(0, fi.ind);
            ans.set(wp[0] * tableLength + wp[1], ci);

            return ans;
      }

      /*
       * Останавливает процесс генерации нового поколения
       *
       * @see laboratory.common.genetic.Algorithm#stop()
       */
      public void stop() {
      }

      /**
       * Применяем большую мутацию
       */
      public void bigMutation(FitIndividual<I>[][] newGeneration) {
            double index = 1;
            double every = 1 / newIndividualsPartBig;

            for (int i = 0; i < tableLength; i++) {
                  for (int j = 0; j < tableLength; j++) {
                        FitIndividual<I> w;

                        if (index >= every) {
                              // Создаем новую особь
                              w = construct(factory.getIndividual());
                              index -= every;
                        } else {
                              // Выбираем лучшую из старых
                              w = generation[i][j];
                              for (int k = 0; k < 4; k++) {
                                    w    =    winner(w,    generation[norm(i    +
dx[k])][norm(j + dy[k])]);
                              }
                        }

                        newGeneration[i][j] = w;
                        index++;
                  }
            }
      }
}
```

Файл *CellularGALoader.java*

```
package laboratory.plugin.algorithm.cellular;

import laboratory.util.loader.AbstractAlgorithmLoader;
import laboratory.util.gui.config.Util;
import laboratory.common.genetic.Individual;
```

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

import javax.swing.*;
import java.util.jar.JarFile;
import java.util.List;
import java.io.File;
import java.awt.*;

/**
 * Загрузчик клеточного генетического алгоритма
 *
 * @author Сергей Казаков, группа 3539, СПбГУ ИТМО.
 */
public class CellularGALoader<I extends Individual> extends
AbstractAlgorithmLoader<I> {
      public CellularGALoader(JarFile file, File dir) {
            super(file, dir);
            Config.getInstance().setJar(file);
      }

      @Override
      public String getMessage() {
            if (getSelections().isEmpty()) {
                  return "Please, select only one selection strategy!";
            } else {
                  return "OK";
            }
      }

      public Algorithm<I> loadAlgorithm(List<IndividualFactory<I>>
      individualFactories,
            List<Crossover<I>> crossovers, List<Mutation<I>> mutations,
            List<Selection<I>> sel, List<Fitness<I>> functions) {
            Config c = Config.getInstance();
            return new CellularGA<I>(c.getTableLength(),
c.getMutationProbability(),
                        c.getBigMutationPeriod(), c.getNewIndividualsPartBig(),
                        individualFactories.get(0), mutations.get(0),
                        crossovers.get(0), getSelections().get(0),
                        functions.get(0));
      }

      @Override
      public JDialog getConfigDialog(JFrame owner) {
            final JDialog dialog = getSelectionChooser(owner, "Cellular Genetic
Algortihm. Choose selection straregies");
            Util.setSize(dialog, new Dimension(300, 250));
            //Util.showModal(dialog);
            return dialog;
      }

}
```

Файл *Config.java*

```
package laboratory.plugin.algorithm.cellular;

import laboratory.util.loader.JarReader;
import laboratory.util.Parser;

import java.util.jar.JarFile;
```

```
/**
 * Конфигурация клеточного алгоритма
 *
 * @author Сергей Казаков, группа 3539, СПбГУ ИТМО.
 */
public class Config {
	private Config() {
	}

	private static Config ourInstance = new Config();

	public static Config getInstance() {
		return ourInstance;
	}

	// Размер (длина и высота) таблицы (тора)
	private int tableLength;

	public int getTableLength() {
		return tableLength;
	}

	public void setTableLength(int tableLength) {
		this.tableLength = tableLength;
	}

	// Вероятность обычной мутации для особи
	private double mutationProbability;

	public double getMutationProbability() {
		return mutationProbability;
	}

	public void setMutationProbability(double mutationProbability) {
		this.mutationProbability = mutationProbability;
	}

	// Период между большими мутациями всего поколения
	private int bigMutationPeriod;

	public int getBigMutationPeriod() {
		return bigMutationPeriod;
	}

	public void setBigMutationPeriod(int bigMutationPeriod) {
		this.bigMutationPeriod = bigMutationPeriod;
	}

	// Часть новых особей при большой мутации
	private double newIndividualsPartBig;

	public double getNewIndividualsPartBig() {
		return newIndividualsPartBig;
	}

	public void setNewIndividualsPartBig(double newIndividualsPartBig) {
		this.newIndividualsPartBig = newIndividualsPartBig;
	}

	public void setJar(JarFile jar) {
		Parser p = new Parser(JarReader.getProperties(jar,
"algorithm.properties"));
		setTableLength(p.getInt("length.table"));
```

```
            setMutationProbability(p.getDouble("probability.mutation"));
            setBigMutationPeriod(p.getInt("period.mutation.big"));
            setNewIndividualsPartBig(p.getDouble("part.newIndividuals.big"));
      }

}
```

Файл *algorithm.properties*

```
name=Cellular

length.table=19

probability.mutation=0.03

period.mutation.big=100
part.newIndividuals.big=0.5

description.file=description.html
```

Отбор методом рулетки

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.List;
import java.util.Random;

/**
 * Отбор методом рулетки (roulette selection)
 *
 * @author Сергей Казаков, группа 3539, СПбГУ ИТМО.
 */
public class RouletteSelection<I extends Individual> implements Selection<I> {

	private final Random random = new Random();

	/*
	 * Применить отбор для списка population и выбрать m особей.
	 *
	 * @see laboratory.common.genetic.operator.Selection#apply(java.util.List,
int)
	 */
	@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;
		for (int i = 1; i < n; i++) {
			weight[i] = weight[i - 1] + population.get(i).fitness;
		}

		return Util.listFromFunctor(new Functor0<FitIndividual<I>>() {
			public FitIndividual<I> apply() {
				double p = weight[n - 1] * random.nextDouble();
				int i = Arrays.binarySearch(weight, p);
				if (i >= 0) {
					i++;
				} else {
					i = - i - 1;
				}
				return population.get(i);
			}
		}, m);
	}
}
```

Файл *RouletteLoader.java*

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

import laboratory.util.loader.OperatorLoader;
import laboratory.common.genetic.operator.Selection;
import laboratory.common.genetic.Individual;
```

```
import javax.swing.*;
import java.util.jar.JarFile;

public class RouletteLoader<I extends Individual> implements
OperatorLoader<Selection<I>> {
	@Override
	public Selection<I> loadOperator() {
		return new RouletteSelection<I>();
	}

	@Override
	public JDialog getConfigDialog(JDialog owner) {
		return null;
	}

	@Override
	public String getName() {
		return "Roulette";
	}

	public RouletteLoader(JarFile jar) {}
}
```

Отбор методом турнира

Файл *TournamentSelection.java*

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

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.List;
import java.util.Random;

/**
 * Турнирный отбор (tournament selection)
 *
 * @author Сергей Казаков, группа 3539, СПбГУ ИТМО.
 */
public class TournamentSelection<I extends Individual> implements Selection<I> {

	private final Random random = new Random();

	/*
	 * Применить отбор для списка population и выбрать m особей.
	 *
	 * @see laboratory.common.genetic.operator.Selection#apply(java.util.List,
int)
	 */
	@Override
	public List<FitIndividual<I>> apply(final List<FitIndividual<I>>
population, int m) {
		final int n = population.size();
		final int t = Math.max(2, (int) Math.round(n / (double) m));

		return Util.listFromFunctor(new Functor0<FitIndividual<I>>() {
			public FitIndividual<I> apply() {
				FitIndividual<I> w = null;
```

```
                        for (int i = 0; i < t; i++) {
                              FitIndividual<I> nw =
population.get(random.nextInt(n));
                              if ((w == null) || (nw.fitness > w.fitness)) {
                                    w = nw;
                              }
                        }
                        return w;
                  }
            }, m);
      }
}
```

Файл TournamentLoader.java

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

import laboratory.util.loader.OperatorLoader;
import laboratory.common.genetic.operator.Selection;
import laboratory.common.genetic.Individual;

import javax.swing.*;
import java.util.jar.JarFile;

public class TournamentLoader<I extends Individual> implements
OperatorLoader<Selection<I>> {
      @Override
      public Selection<I> loadOperator() {
            return new TournamentSelection<I>();
      }

      @Override
      public JDialog getConfigDialog(JDialog owner) {
            return null;
      }

      @Override
      public String getName() {
            return "Tournament";
      }

      public TournamentLoader(JarFile jar) {}
}
```