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

Д.С.Чивилихин

Отчет по курсовой работе

«Метод построения управляющих автоматов на основе муравьиных алгоритмов»

Санкт-Петербург
2011

Оглавление

Введение

Муравьиные алгоритмы (англ. *Ant colony optimization algorithms*) [1] давно и успешно используются в решении задач оптимизации на графах, таких как, например, задача о коммивояжере. Пространством поиска в этой задаче является граф, а результатом работы муравьиного алгоритма – гамильтонов (проходящий через все вершины ровно один раз) путь в этом графе.

Рассмотрим задачу о построении управляющего автомата при фиксированном множестве входных и выходных воздействий. Пусть задано число состояний *N*, множество входных воздействий *Events* и множество выходных воздействий *Actions* детерминированного конечного автомата. Пусть на множестве всех автоматов с заданными выше параметрами (*N*, *Events*, *Actions*) задана функция, отражающая степень “приспособленности” автомата – функция приспособленности. Задача состоит в отыскании автомата, максимизирующего данную функцию.

Существует множество методов автоматического построения конечных автоматов при заданной функции приспособленности. Так, для решения этой задачи успешно применяются генетические алгоритмы и генетическое программирование, эволюционные алгоритмы, построение автоматов на основе тестов и сценариев работы. Метод построения автоматов на основе тестов также применяется в комбинации с методами верификации конечных автоматов.

Целью настоящей работы является разработка метода построения управляющих автоматов на основе муравьиных алгоритмов. В развиваемом подходе применяется представление автоматов в виде графов переходов.

1. Постановка задачи

Рассмотрим представление конечных автоматов в виде графов переходов. При этом подходе состояния автомата соответствуют вершинам графа, а переходы – его дугам.

Заметим, что любой автомат из выбранного множества автоматов как граф является подграфом *полного недетерминированного конечного автомата* с параметрами (*N*, *Events*, *Actions*). Под полным недетерминированным автоматом понимается автомат, в котором из каждого состояния по каждому входному воздействию существует переход в каждое другое состояние с каждым выходным воздействием. Теперь сформулируем задачу для муравьиного алгоритма оптимизации.

Задано множество входных воздействий *Events*, множество выходных воздействий *Actions* и число состояний *N* ДКА. Задана функция приспособленности на множестве всех ДКА с указанными параметрами, отображающая автомат в действительное число. На полном НКА с теми же параметрами требуется найти такой путь, что ДКА, составленный из ребер и вершин найденного пути, имеет наибольшую функцию приспособленности.

2. Метод построения управляющих автоматов на основе муравьиных алгоритмов

В данном разделе приводится описание разработанного алгоритма, приведена общая схема и подробно описаны все этапы его работы.

Идея алгоритма заключается в следующем. Всем ребрам полного НКА приписывается некоторый одинаковый ненулевой вес – значение «феромона» в терминах муравьиных алгоритмов. В каждую вершину полного НКА помещается по муравью. Муравей выбирает наиболее «выгодные» переходы из своего состояния по каждому входному символу исходя из значений феромона на переходах (ребрах), выходящих из данного состояния (вершины). Каждый муравей строит переходы только из своего состояния.

По выбранным муравьями состояниям и переходам строится ДКА. Далее, для ДКА вычисляется функция приспособленности. Выделяются переходы, посещенные автоматом в ходе вычисления значения функции приспособленности, и к их весам добавляются дополнительные веса, пропорциональные этому значению. На следующем этапе со всех ребер НКА «испаряется» феромон – веса всех ребер уменьшаются в одинаковое число раз.

Наконец, проверяются условия сходимости (достижения требуемого значения функции приспособленности) и стагнации. Под стагнацией понимается ситуация, при которой значение функции приспособленности лучших автоматов не увеличивается в течение некоего фиксированного промежутка времени (числа шагов алгоритма). В таком случае алгоритм перезапускается. При достижении требуемого значения функции приспособленности алгоритм прекращает свою работу, выдавая лучший из построенных автоматов.

2.1. Схема работы алгоритма

Можно выделить следующие основные этапы работы алгоритма:

1) чтение входных данных;
2) формирование полного НКА;
3) инициализация муравьиного алгоритма
4) запуск муравьев на недетерминированном конечном автомате – получение списка переходов, которые сделали муравьи;
5) формирование детерминированного конечного автомата по полученному списку переходов;
6) вычисление функции приспособленности построенного детерминированного конечного автомата;
7) откладывание феромона построенным автоматом на ребрах НКА;
8) откладывание феромона лучшими из построенных автоматов;
9) испарение феромона;
10) проверка выходных условий.

2.2. Чтение входных данных

Входные данные алгоритма подразделяются на:

- параметры искомого автомата:
  - число состояний – *N;*
  - множество входных воздействий – *Events;*
  - множество выходных воздействий – *Actions.*
- параметры муравьиного алгоритма:
  - скорость испарения феромона – *evaporationRate;*
  - максимальное число шагов алгоритма – *maxNumberOfSteps;*
  - параметр стагнации – *stagnationNumber;*
  - значение функции приспособленности, при котором алгоритм завершает свою работу – *goalFitness.*
- параметры задачи.

Под параметрами задачи понимается информация о решаемой автоматом задаче, которая используется при вычислении функции приспособленности. Вот некоторые примеры параметров задачи:

- поле в задаче об умном муравье [2, 3];
- набор сценариев в задаче о построении управляющих автоматов на основе сценариев.

2.3. Формирование полного НКА

На этом этапе формируется полный недетерминированный конечный автомат с заданным числом состояний и множествами входных и выходных воздействий.

2.4. Инициализация муравьиного алгоритма

Инициализация муравьиного алгоритма состоит в том, что всем ребрам (переходам) НКА ставится в соответствие некоторый вес, равный минимальному значению феромона.

2.5. *Запуск муравьев на недетерминированном конечном автомате*

Этот основной этап алгоритма состоит из следующих подпунктов:

- хранится список переходов, которые муравьи делают в НКА – *visitedTransitions;*
- в каждую вершину НКА помещается муравей;
- каждый муравей с помощью метода “рулетки” определяет наиболее выгодный переход по каждому из входных воздействий. Затем, выбранные переходы добавляются в список *visitedTransitions;*
- по полученному списку переходов *visitedTransitions* строится детерминированный конечный автомат.

Заметим, что автомат, полученный на этом шаге алгоритма, содержит все состояния НКА и переходы по каждому входному воздействию из каждого состояния.

Данная особенность алгоритма может стать помехой для задач, в которых требуется помимо максимизации функции приспособленности минимизировать число переходов в автомате.

Для решения этой проблемы можно использовать предлагаемый метод в комбинации с эволюционными стратегиями. Использованная стратегия заключается в мутировании получаемых автоматов для минимизации количества переходов; мутации заключаются в удалении некоторых переходов из автомата. Если при удалении перехода функция приспособленности увеличилась, то он удаляется, иначе – возвращается в список переходов.

Другой способ решения этой проблемы – добавлять в список посещенных переходов *visitedTransitions* только те переходы, которые делал автомат при его запуске на конкретной задаче, то есть при вычислении значения функции приспособленности.

2.6. Вычисление функции приспособленности построенного ДКА

На этом шаге построенному ДКА сопоставляется действительное число с помощью вычисления функции приспособленности. Здесь используются параметры задачи, о которых упоминалось в разд. 2.2.

2.7. Откладывание феромона построенным ДКА на ребрах НКА

На данном шаге алгоритма обновляются значения феромона на ребрах, которые посетили муравьи при построении последнего ДКА. К значению феромона на этих ребрах прибавляется величина, пропорциональная функции приспособленности ДКА.

2.8. Откладывание феромона лучшим из построенных ДКА

В данной работе в качестве модификации муравьиного алгоритма используется система “элитарный муравей” (*Elitist ant system)* Модификация заключается в том, что лучшая особь (ДКА) на каждом шаге алгоритма

откладывает феромон наряду с текущей особью. Такая модификация существенно повышает устойчивость алгоритма.

Также существует возможность использования не одного лучшего ДКА, а нескольких. В таком случае, феромон на ребрах НКА откладывает каждая из списка лучших особей.

2.9. Испарение феромона

На этом шаге количество феромона на всех ребрах НКА уменьшается в $(1 - evaporationRate)$ раз. Значение параметра *evaporationRate* подбирается индивидуально для каждой задачи.

2.10. Проверка выходных условий

Если значение функции приспособленности лучшей особи не меньше, чем минимальное требуемое значение *goalFitness*, алгоритм прекращает свою работу, выдавая на выход лучший построенный автомат. Если значение приспособленности меньше, чем *goalFitness*, муравьи запускаются на НКА еще раз.

3. Применение предложенного метода для построения управляющих автоматов на основе тестов

В данном разделе приводятся результаты тестирования предложенного метода построения автоматов на задаче о построении автоматов на основе тестовых примеров. Вначале дается описание метода генерации автоматов по тестовым примерам, затем приводятся результаты численного эксперимента.

3.1. Описание метода построения автоматов на основе тестов

Описанный метод построения автоматов в данной формулировке позволяет применять его к весьма широкому классу задач. Интересным примером служит использование метода для построения автоматов с заданным поведением, основываясь на тестовых примерах.

Тестовый пример представляет собой последовательность входных воздействий *Events*[*i*] и ожидаемую последовательность выходных воздействий *Actions*[*i*], которую должен сгенерировать автомат.

Параметрами задачи в данном случае является набор сценариев, а вычисление функции приспособленности автомата основано на редакционном расстоянии (расстоянии Левенштейна). Для вычисления функции приспособленности автомату подается на вход каждая из входных последовательностей *Events*[*i*]. Получив на вход данную последовательность, автомат генерирует последовательность выходных воздействия *Output*[*i*]. Далее, согласно работе [4], вычисляется величина:

$$FF_1 = \frac{\sum_{i=1}^{n}(1 - \frac{ED(Output[i], Answer[i])}{\max(|Output[i]|, |Answer[i]|)})}{n}.$$

Значения этой величины лежат между *0* и *1* и она тем больше, чем «лучше» автомат соответствует сценариям. Сама функция приспособленности, которая учитывает прохождение сценариев и число переходов в автомате, вычисляется по формуле:

$$FF_2 = \begin{cases} 0.5 \cdot T \cdot FF_1 + 0.01 \cdot (100 - numberOfTransitions), FF_1 < 1 \\ T + 0.01 \cdot (100 - numberOfTransitions), FF_1 = 1 \end{cases} \quad (1)$$

Здесь за T обозначена «стоимость» прохождения всех сценариев, а за *numberOfTransitions* – число переходов в автомате.

3.2. Построение автомата управления часами с будильником с помощью муравьиных алгоритмов на основе тестов

Метод построения управляющих автоматов с помощью муравьиных алгоритмов был протестирован на задаче о построении автомата управления часами с будильником на основе тестовых примеров, описанной в книге [2]. Для этого был выбран набор из 38 тестов, которые были использованы в работах [4, 5] для построения автомата управления часами с будильником различными методами.

После построения автомата, удовлетворяющего всем тестам, для уменьшения числа в автомате была применена эволюционная стратегия, описанная в разд. 2.5. Для вычислений использовался компьютер с процессором *Intel Core i3 380M* с тактовой частотой 2,53 ГГц.

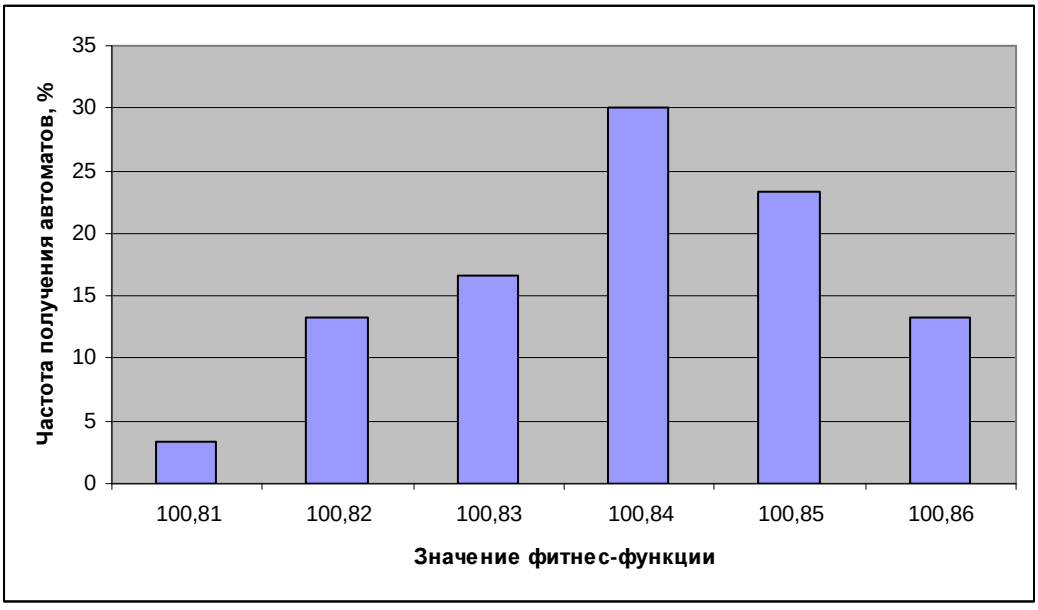

Рис. 1. Частота получения автоматов, удовлетворяющих всем тестам

Эксперимент по построению автомата был повторен 30 раз. На рис. 1. приведены частоты получения автоматов с различными значениями

функций приспособленности. Полученные автоматы удовлетворяют всем тестам, имеют три состояния и отличаются лишь числом переходов.

Минимальное время построения автомата, удовлетворяющего всем тестам, составило три секунды, среднее время – 19,8 секунд, максимальное – 595 секунд.

Наилучший из построенных автоматов удовлетворяет всем тестам, имеет три состояния и 14 переходов, а значение функции приспособленности равно 100,86. Для построения этого автомата потребовалось 7905 вычислений функции приспособленности и три секунды. Диаграмма переходов полученного автомата приведена на рис. 1. Худший из автоматов также удовлетворяет всем сценариям, однако имеет 19 переходов.

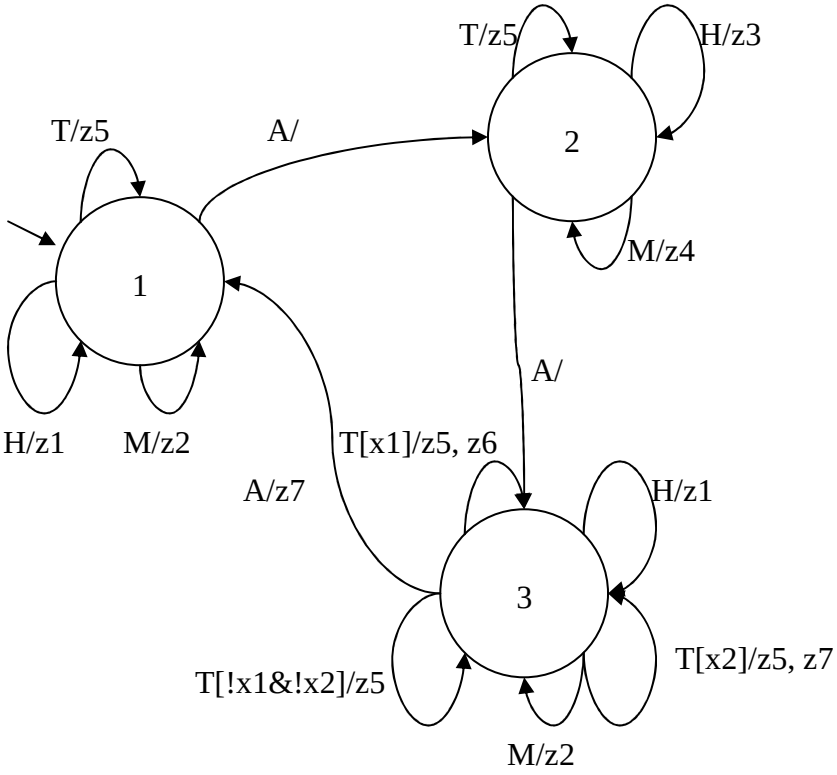

Рис. 2. Автомат управления часами с будильником, построенный с помощью муравьиного алгоритма и эволюционной стратегии

Можно заметить, что полученный автомат изоморфен автомату, полученному в работах [4, 5].

4. Применение предложенного метода для решения задачи об «Умном муравье»

Еще одной задачей, для решения которой был применен предложенный метод, является задача об «Умном муравье» [2, 3]. Задача состоит в следующем. Задано прямоугольное поле, представляющее собой тор, в некоторых клетках которого расположены «яблоки». Муравей может совершать следующие действия: перейти на одну клетку вперед, повернуть направо, повернуть налево. Если при переходе на клетку в ней находится яблоко, муравей его съедает.

Целью игры является съесть все 89 яблок за 200 ходов. В работе [3] предлагается решение данной задачи путем построения конечного автомата, управляющего муравьем, с помощью генетического программирования. Так, авторами был найден автомат, съедающий всю еду и имеющий семь состояний. Число вычислений функции приспособленности, которое потребовалось для этого – порядка сотен миллионов (160 и 230 млн. автоматов соответственно для двух различных экспериментов).

При попытке решения задачи с помощью предложенного метода был получен автомат, имеющий семь состояний, который, однако, съедает лишь 86 яблок за 200 ходов. Автомат был получен после перебора 2,6 млн. автоматов за 267 секунд. Диаграмма переходов полученного автомата приведена на рис. 3.

Кроме того, после перебора 170000 автоматов, был получен автомат, содержащий восемь состояний и съедающий всю еду. Для вычислений потребовалось 29 секунд. Диаграмма переходов автомата приведена на рис. 4.

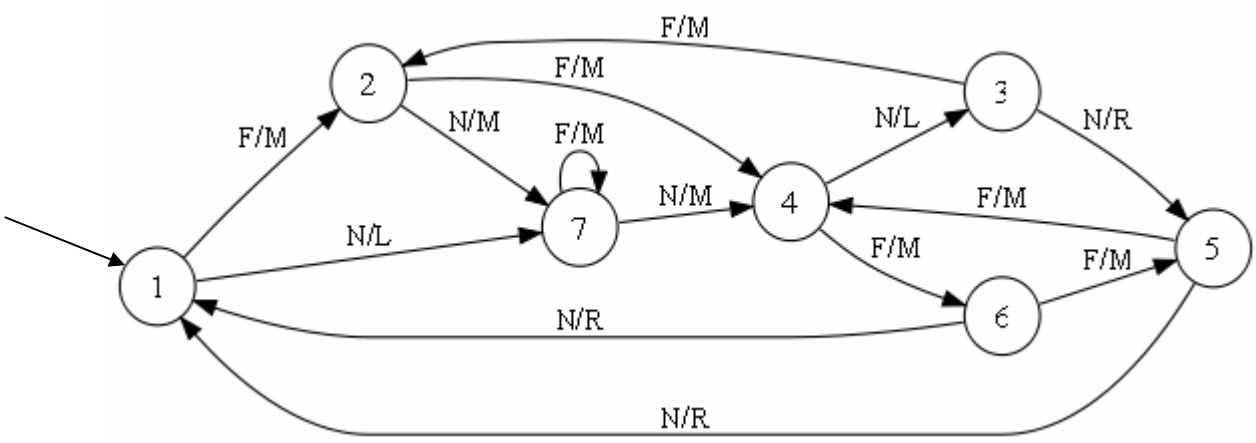

Рис.3. Автомат из семи состояний, съедающий 86 яблок в задаче об «Умном муравье»

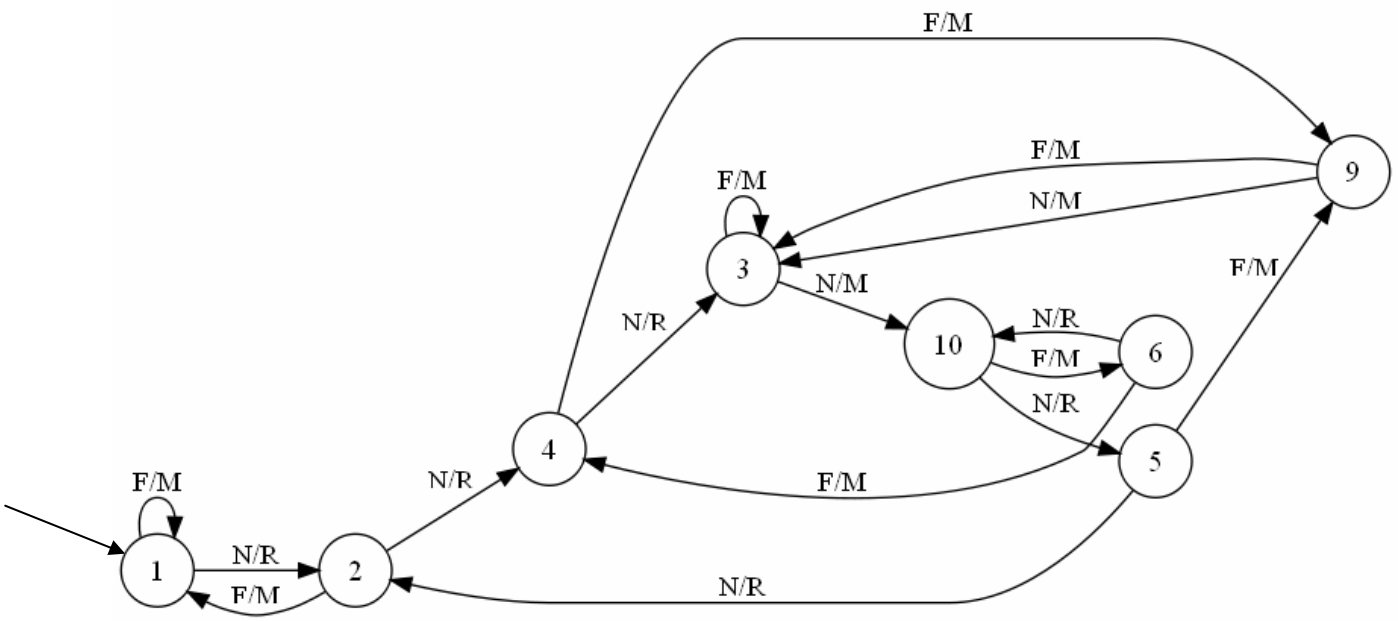

Рис.4. Автомат из восьми состояний, съедающий все яблоки в задаче об «Умном муравье»

Предполагается, что для улучшения этих результатов требуется подобрать более оптимальные параметры алгоритма.

Заключение

Разработан метод построения управляющих автоматов с помощью муравьиных алгоритмов и написана программа на языке *Java*, реализующая этот метод. Метод был проверен на задаче о построении управляющих автоматов на основе тестовых примеров и показал результаты в производительности, сравнимые с результатами, приведенными в работах [4, 5]. Был получен автомат управления часами с будильником, изоморфный автоматам, построенным в указанных работах.

Метод был также проверен на задаче об «Умном муравье». При этом построить оптимальный автомат не удалось, что свидетельствует о необходимости дальнейшего подбора параметров алгоритма, что является основной трудностью при применении предложенного метода.

Предложенный метод построения автоматов является достаточно общим и может быть применен в построении управляющих автоматов для любых задач, для которых можно ввести функцию приспособленности автомата, определенную на множестве всех автоматов с заданными параметрами.

Источники

1. *Dorigo M.* Optimization, Learning and Natural Algorithms. PhD thesis, Politecnico di Milano, Italy, 1992.
2. *Поликарпова, Н.И., Шалыто А.А.* Автоматное программирование. СПб: Питер, 2009.
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. *Царев Ф.Н.* Метод построения автоматов управления системами со сложным поведением на основе тестов с помощью генетического программирования / Материалы Международной научной конференции «Компьютерные науки и информационные технологии». Саратов: СГУ, 2009, с.216-219.
5. *К.В. Егоров, Ф.Н. Царев.* Совместное применение генетического программирования и верификации моделей для построения автоматов управления системами со сложным поведением / Сборник докладов конференции молодых ученых и специалистов «Информационные технологии и системы». М.: ИППИ РАН, 2009, с.77-82.

Приложение. Исходные тексты программы

Модули метода построения автоматов:

1. AntColonyOptimizationAlgorithmRunner.java – класс, реализующий предложенный алгоритм построения автоматов.
2. ProblemRunner.java – класс, реализующий запуск алгоритма построения автоматов с конкретной функцией приспособленности.

Модули, реализующие вычисление функции приспособленности:

1. AbstractAutomatonTask.java – базовый абстрактный класс для классов, реализующих вычисление функции приспособленности.
2. ScenariosTask.java – класс, реализующий вычисление функции приспособленности в задаче о построении автоматов на основе тестовых сценариев.

Листинг *AntColonyOptimizationAlgorithmRunner.java*

```
package aco;

import java.io.File;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Random;
import task.AbstractAutomatonTask;
import automaton.DFA;
import automaton.FullNfaFactory;
import automaton.NFA;
import common.Pair;

public class AntColonyOptimizationAlgorithmRunner {
    private Random random = new Random();
    private AbstractAutomatonTask task;
    private NFA nfa;
    public ArrayList<Pair<DFA, Double>> bestAutomatas;

    public AntColonyOptimizationAlgorithmRunner(AbstractAutomatonTask task) {
        this.task = task;
        bestAutomatas = new ArrayList<Pair<DFA, Double>>();
    }

    public NFA.Transition rouletteSelection(
            ArrayList<NFA.Transition> transitions) {
        Collections.sort(transitions);
        int size = transitions.size();
```

```
    double weight[] = new double[size];
    weight[0] = transitions.get(0).getPheromone();

    for (int i = 1; i < size; i++) {
        weight[i] = weight[i - 1] + transitions.get(i).getPheromone();
    }

    double p = weight[size - 1] * random.nextDouble();
    int j = 0;
    while (p > weight[j]) {
        j++;
    }

    return transitions.get(j);
}

public DFA runAntThroughNFA(NFA nfa,
        ArrayList<Pair<Integer, NFA.Transition>> visitedTransitions) {
    // переходы ДКА
    DFA.Transition[][] dfaTransitions = new DFA.Transition[nfa
            .getNumberOfStates()][nfa.getNumberOfExternalInfluences()];

    for (int i = 0; i < nfa.getNumberOfStates(); i++) {
        int currentState = i;

        NFA.Transition transitions[] = new NFA.Transition[nfa
                .getNumberOfExternalInfluences()];
        for (int j = 0; j < transitions.length; j++) {
                        transitions[j] = rouletteSelection(nfa.transitions[currentState][j]);
            if (transitions[j].getEndState() == -1) {
                dfaTransitions[currentState][j] = new DFA.Transition(
                        transitions[j]);
                continue;
            }
            visitedTransitions.add(new Pair<Integer, NFA.Transition>(
                    currentState, transitions[j]));
            dfaTransitions[currentState][j] = new DFA.Transition(
                    transitions[j]);
        }
    }
    return new DFA(nfa.getNumberOfStates(),
            nfa.getNumberOfExternalInfluences(), dfaTransitions);
}



private void imprintPheromone(
        ArrayList<Pair<Integer, NFA.Transition>> transitions,
        double fitness, double goalValue) {
    for (Pair<Integer, NFA.Transition> p : transitions) {
        for (int k = 0; k < nfa.transitions[p.first][p.second.getSymbol()]
                .size(); k++) {
            NFA.Transition tmp = nfa.transitions[p.first][p.second
```

```
                .getSymbol()].get(k);
            if (tmp.getEndState() == p.second.getEndState()
                && tmp.getAction() == p.second.getAction()) {
                double curP = tmp.getPheromone();
                curP += fitness / goalValue;

                if (curP < NFA.MIN_PHEROMONE_VALUE) {
                    curP = NFA.MIN_PHEROMONE_VALUE;
                }
                if (curP > NFA.MAX_PHEROMONE_VALUE) {
                    curP = NFA.MAX_PHEROMONE_VALUE;
                }

                tmp.setPheromone(curP);
                nfa.transitions[p.first][p.second.getSymbol()].set(k, tmp);
            }
        }
    }
}

private double evaporatePheromone(double evaporationRate) {
    double maxPheromoneValue = 0;
    for (int i = 0; i < nfa.getNumberOfStates(); i++) {
        for (int j = 0; j < nfa.getNumberOfExternalInfluences(); j++) {
            for (int k = 0; k < nfa.transitions[i][j].size(); k++) {
                NFA.Transition transition = nfa.transitions[i][j].get(k);
                double currentPheromone = transition.getPheromone();
                currentPheromone = (1.0 - evaporationRate)
                        * currentPheromone;

                if (currentPheromone < NFA.MIN_PHEROMONE_VALUE) {
                    currentPheromone = NFA.MIN_PHEROMONE_VALUE;
                }
                if (currentPheromone > maxPheromoneValue) {
                    maxPheromoneValue = currentPheromone;
                }
                transition.setPheromone(currentPheromone);
                nfa.transitions[i][j].set(k, transition);
            }
        }
    }
    return maxPheromoneValue;
}

public Pair<DFA, Integer> runACO(double evaporationRate,
        int numberOfStates, double goalValue, double printBound,
        int numberOfBestAutomatas, double minPheromoneValueCoefficient,
        int stagnationNumber) {
    bestAutomatas.clear();
    long start = System.currentTimeMillis();
    nfa = FullNfaFactory.getFullAutomata(numberOfStates, task.getEvents()
            .size(), task.getActions());
```

```
double maxPheromoneValue = NFA.MIN_PHEROMONE_VALUE;
String resultDirName = "er=" + evaporationRate;
File resultDir = new File(resultDirName);
resultDir.mkdir();

        ArrayList<Pair<Integer, NFA.Transition>> visitedTransitions = new
        ArrayList<Pair<Integer, NFA.Transition>>();
        ArrayList<Pair<Integer, NFA.Transition>> bestTransitions = new
        ArrayList<Pair<Integer, NFA.Transition>>();

int stepCounter = 0;
double fitness = 0;
double bestFitness = -10000;
DFA dfa;
DFA bestAutomata = null;
ArrayList<Double> bestFitnessHistory = new ArrayList<Double>();
int lastBestFitnessOccurence = 0;
while (true) {
    stepCounter++;

    if (stepCounter - lastBestFitnessOccurence > stagnationNumber) {
        System.out.println("Stagnation. Restarting....");
        lastBestFitnessOccurence = stepCounter;
        nfa = FullNfaFactory.getFullAutomata(numberOfStates, task
                .getEvents().size(), task.getActions());
        bestFitness = 0.0;
        bestAutomatas.clear();
        bestTransitions.clear();
    }

    // запускаем муравья гулять по НКА. строим ДКА.
    dfa = runAntThroughNFA(nfa, visitedTransitions);

    // построили ДКА. запускаем на нем
    // тестовую задачу
    fitness = task.getFitness(dfa);
    visitedTransitions = dfa.visitedTransitions;

    bestFitnessHistory.add(bestFitness);
    if (fitness > bestFitness) {
        lastBestFitnessOccurence = stepCounter;
        bestAutomatas.add(new Pair<DFA, Double>(dfa, fitness));

        if (bestAutomatas.size() > numberOfBestAutomatas) {
            bestAutomatas.remove(0);
        }

        bestAutomata = dfa;
        bestFitness = fitness;
        bestTransitions.clear();
        bestTransitions.addAll(dfa.visitedTransitions);

        if (bestFitness > printBound) {
```

```
                                double time = (int) ((System.currentTimeMillis() - start) /
                                1000.0);
                        bestAutomata.printTransitionDiagram(resultDirName + "/"
                                + String.valueOf(fitness) + "_" + stepCounter
                                + "_time=" + time);
                        bestAutomata.printToGV(
                                resultDirName + "/" + String.valueOf(fitness)
                                        + "_.gv", task.getEvents());
                    }
                }
                maxPheromoneValue = evaporatePheromone(evaporationRate);
                imprintPheromone(visitedTransitions, fitness, goalValue);
                NFA.MIN_PHEROMONE_VALUE = minPheromoneValueCoefficient
                        * maxPheromoneValue;

                for (int i = 0; i < bestAutomatas.size(); i++) {
                    DFA a = bestAutomatas.get(i).first;
                    double f = bestAutomatas.get(i).second;
                    imprintPheromone(a.getListOfTransitions(), f
                            * (double) (i + 1.0) / (double) bestAutomatas.size(),
                            goalValue);
                }

                visitedTransitions.clear();

                if (stepCounter % 100 == 0) {
                    System.out.println(stepCounter + ": current = " + fitness
                            + "; best = " + bestFitness + "; max = "
                            + maxPheromoneValue);
                }

                if (bestFitness >= goalValue) {
                    return new Pair<DFA, Integer>(bestAutomata, stepCounter);
                }
            }
        }
    }
```

Листинг *ScenariosTask.java*

```
package task.scenarios;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Properties;
import java.util.Scanner;
```

```java
import task.AbstractAutomatonTask;
import automaton.DFA;
import common.StringUtils;

public class ScenariosTask extends AbstractAutomatonTask {
    private AutomatonScenario[] scenarios;
    private ArrayList<String> events;
    private String[] actions;
    private String[][] outputs;

    public ScenariosTask(String configFileName) {
        configure(configFileName);
        outputs = new String[scenarios.length][];
    }

    private void readScenarios(String filename) {
        Scanner in = null;
        try {
            in = new Scanner(new File(filename));
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }

        int numberOfScenarios = in.nextInt();
        scenarios = new AutomatonScenario[numberOfScenarios];

        for (int i = 0; i < numberOfScenarios; i++) {
            String[] inputAndOutput = in.next().split("->");
                        scenarios[i] = new AutomatonScenario(inputAndOutput[0].split("_"),
                        inputAndOutput[1].split("_"));
        }

    }

    private void configure(String propertiesFileName) {
        Properties config = new Properties();

        try {
            config.load(new FileInputStream(new File(propertiesFileName)));
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        events = new ArrayList<String>();
        String eventArray[] = config.getProperty("events").split(" ");
        for (int i = 0; i < eventArray.length; i++) {
            events.add(eventArray[i]);
        }
        actions = config.getProperty("actions").split(" ");

        readScenarios(config.getProperty("tests"));
    }
```

```
private double runDFAOnScenario(DFA dfa, int scenarioIndex) {
    int currentState = dfa.getInitialState();
    int numberOfSteps = 0;
    String[] answers = new String[scenarios[scenarioIndex].input.length];
    for (int i = 0; i < scenarios[scenarioIndex].input.length; i++) {
                int influenceIndex =
                events.indexOf(scenarios[scenarioIndex].getInput()[numberOfSteps++]);
        String answer = dfa.transitions[currentState][influenceIndex].getAction();
                int nextState = dfa.transitions[currentState][influenceIndex].
                                getEndState();
        if (nextState == -1) {
            continue;
        }
        currentState = nextState;
        answers[i] = answer;
    }
    outputs[scenarioIndex] = new String[answers.length];
    outputs[scenarioIndex] = answers.clone();
            return (double)scenarios[scenarioIndex].getLevensteinDistance(answers) /
            (double)answers.length;
}

public String[][] getAutomatonOutputs() {
    return outputs;
}

@Override
public String[] getActions() {
    return actions;
}

@Override
public ArrayList<String> getEvents() {
    return events;
}

@Override
public double getFitness(DFA dfa) {
    double f1 = 0;
    for (int i = 0; i < scenarios.length; i++) {
        f1 += 1.0 - runDFAOnScenario(dfa, i);
    }
    f1 /= (double)scenarios.length;

    double T = 100;

    int numberOfTransitions = dfa.getNumberOfTransitions();

    if (f1 < 0) {
        return 0;
    }
```

```
        if (f1 < 1.0) {
            return 0.5 * T * f1 + 0.01 * (100 - numberOfTransitions);
        }
        return T + 0.01 * (100 - numberOfTransitions);
    }

    /**
     * Класс реализующий сценарий работы конечного автомата.
     * Хранит входную и выходную последовательности в виде массивов строк
     */
    private class AutomatonScenario {
        private String[] input;
        private String[] output;

        public AutomatonScenario(String[] input, String[] output) {
            this.input = input.clone();
            this.output = output.clone();
        }

        public String[] getInput() {
            return input;
        }

        public int getLevensteinDistance(String[] answer) {
            String ans = "";
            String out = "";
            for (int i = 0; i < answer.length; i++) {
                ans += answer[i];
                out += output[i];
            }
            return StringUtils.levensteinDistance(ans, out);
        }
    }

    public void printAutomataOutputs(String filename) {
        PrintWriter out = null;
        try {
            out = new PrintWriter(new File(filename));
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        for (int i = 0; i < outputs.length; i++) {
            for (int j = 0; j < outputs[i].length; j++) {
                out.print(outputs[i][j] + " ");
            }
            out.println();
        }
        out.close();
    }
}
```

Листинг *ProblemRunner.java*

```
package taskrunner;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import task.AbstractAutomatonTask;
import task.scenarios.ScenariosTask;
import aco.AntColonyOptimizationAlgorithmRunner;
import automaton.DFA;

import common.Pair;

public class ProblemRunner {
    AbstractAutomatonTask task;

    public ProblemRunner() {
        task = new ScenariosTask("scenarios.properties");
    }

    public DFA findBestMutatedDFA(DFA dfa) {
        double fitness = task.getFitness(dfa);
        Pair<Double, DFA> mutatedAutomatas[] = new Pair[dfa.getNumberOfTransitions()];
        int transitionCounter = 0;
        for (int i = 0; i < dfa.getNumberOfStates(); i++) {
            for (int j = 0; j < dfa.getNumberOfExternalInfluences(); j++) {
                if (dfa.transitions[i][j].getEndState() == -1) {
                    continue;
                }
                DFA mutatedDFA = mutateDFA(dfa, i, j);
                double testFitness = task.getFitness(mutatedDFA);
                                mutatedAutomatas[transitionCounter++] = new Pair<Double,
                                DFA>(testFitness, mutatedDFA);
            }
        }

        Arrays.sort(mutatedAutomatas, new Comparator<Pair<Double, DFA>>() {
            @Override
            public int compare(Pair<Double, DFA> arg0, Pair<Double, DFA> arg1) {
                if (arg0.first < arg1.first) {
                    return 1;
                }
                return -1;
            }
        });

        if (mutatedAutomatas[0].first > fitness) {
```

```
            return mutatedAutomatas[0].second;
        }
        return dfa;
    }

    public DFA mutateDFA(DFA dfa, int state, int event) {
        DFA result = new DFA(dfa);
        result.transitions[state][event] = new DFA.Transition(-1, "");
        return result;
    }

    public void run() {
        AntColonyOptimizationAlgorithmRunner acoAlgorithm =
            new AntColonyOptimizationAlgorithmRunner(task);

        File attempts = new File("attempts");
        attempts.mkdir();

        ArrayList<Double> stepHistory = new ArrayList<Double>();
        ArrayList<Double> timeHistory = new ArrayList<Double>();
        int j = 0;
        while (stepHistory.size() < 30) {
            ScenariosTask task2 = new ScenariosTask("scenarios.properties");
            task = new ScenariosTask("scenarios.properties");
            String dirName = "attempts/attempt" + j + "/";
            File dir = new File(dirName);
            dir.mkdir();
            long start = System.currentTimeMillis();
            double evaporationRate = 0.7;
            double goalValue = 920;
            double printBound = 850;
            int numberOfStates = 20;
            int numberOfBestAutomatas = 2;
            double minPheromoneValueCoefficient = 0.01;
            int stagnationNumber = 5000000;

                        Pair<DFA, Integer> bestAutomataAndNumberOfSteps =
                        acoAlgorithm.runACO(evaporationRate, numberOfStates, goalValue, printBound,
                    numberOfBestAutomatas, minPheromoneValueCoefficient, stagnationNumber);

            DFA best = bestAutomataAndNumberOfSteps.first;
            int numberOfSteps = bestAutomataAndNumberOfSteps.second;
            for (int i = 0; i < 100; i++)
                best = findBestMutatedDFA(best);
            double fitness = task.getFitness(best);
            int time = (int)((System.currentTimeMillis() - start) / 1000.0);
            System.out.println(task2.getFitness(best));
                        best.printTransitionDiagram(dirName + fitness + "_time=" + time+"_stepcnt=" +
                        numberOfSteps);
            best.printToGV(dirName + task.getFitness(best) + ".gv", task.getEvents());
            stepHistory.add((double)numberOfSteps);
            timeHistory.add((double)time);
            j++;
```

```
        }
        PrintWriter out = null;
        try {
            out = new PrintWriter(new File("runData"));
        } catch(FileNotFoundException e) {
            e.printStackTrace();
        }

        out.println("minTime = " + Collections.min(timeHistory));
        out.println("minStepCount = " + Collections.min(stepHistory));
        out.println("maxTime = " + Collections.max(timeHistory));
        out.println("maxStepCount = " + Collections.max(stepHistory));
        out.close();
    }
}
```