Санкт-Петербургский государственный университет
информационных технологий, механики и оптики

Факультет информационных технологий и программирования
Кафедра компьютерных технологий

В. Р. Данилов

Технология генетического программирования для генерации автоматов управления системами со сложным поведением

Научные руководители: С. И. Николенко, А. А. Шалыто

Санкт-Петербург
2007

Оглавление

Оглавление 3

Введение 6

Глава 1. Эволюционные алгоритмы 8
1.1. Эволюционные вычисления . . . 8
1.2. Общая схема эволюционных алгоритмов . . . 8
1.3. Генетический алгоритм . . . 9
1.3.1. Инициализация . . . 10
1.3.2. Отбор . . . 10
1.3.3. Кроссовер . . . 10
1.3.4. Мутации . . . 11
1.4. Генетическое программирование . . . 11
1.4.1. Инициализация . . . 12
1.4.2. Отбор . . . 12
1.4.3. Кроссовер . . . 12
1.4.4. Мутация . . . 12
1.4.5. Поддержка корректности . . . 13
1.5. Программирование с экспрессией генов . . . 14
1.6. Сравнение подходов . . . 14
Выводы по главе 1 . . . 15

Глава 2. Применение эволюционных алгоритмов для генерации автоматов 16
2.1. Конечные детерминированные автоматы . . . 16
2.1.1. Определение конечного детерминированного автомата . . . 16
2.1.2. Конечный автомат Мили . . . 17
2.2. Клеточные автоматы . . . 18
2.3. Некоторые задачи генерации автоматов . . . 19
2.3.1. Распознавание регулярных языков . . . 19

2.3.2. Задача об умном муравье . . . 19
2.3.3. Обобщенная задача о муравье . . . 20
2.3.4. Задача классификации плотности . . . 21
2.4. Применение генетических алгоритмов . . . 22
2.4.1. Представление автомата в виде строк фиксированной длины 22
2.4.2. Эвристика, ускоряющая вывод автоматов . . . 24
2.5. Применение генетического программирования . . . 25
2.6. Анализ достоинств и недостатков известных подходов . . . 26
Выводы по главе 2 . . . 26

Глава 3. Технология генетического программирования для генерации автоматов на основе деревьев решений 27
3.1. Деревья решений . . . 27
3.2. Представление автомата с помощью деревьев решений . . . 28
3.3. Генетические операции . . . 30
3.4. Простой пример применения . . . 33
3.5. Задание ограничений на целевой автомат . . . 36
3.6. Реализация . . . 37
3.6.1. Общая схема фреймворка . . . 37
3.6.2. Модуль *autoant-framework* . . . 38
3.6.3. Модуль *autoant-ga* . . . 38
3.6.4. Модуль *autoant-vis* . . . 41
Выводы по главе 3 . . . 43

Глава 4. Сравнение 44
4.1. Результаты экспериментов . . . 44
4.2. Анализ . . . 47
Выводы по главе 4 . . . 48

Заключение 49

Литература 50

Приложение 1. Исходные коды 52
Tree.java . . . 52
TreeAutomata.java . . . 58

TreeConfiguration.java 61

Введение

Искусственный интеллект является одним из наиболее перспективных разделов информатики. Сегодня многие теоретические разработки в этой области успешно применяются на практике. В ряде задач решения, созданные с помощью различных технологий искусственного интеллекта, не уступают разработанным людьми, а иногда и превосходят их.

Одной из актуальных задач искусственного интеллекта является автоматическое создание программ. Часто разработка алгоритмов вручную является весьма трудоемким процессом. Поэтому целесообразно попытаться поручить эту задачу компьютеру.

Эволюционные вычисления являются одним из наиболее используемых направлений искусственного интеллекта и успешно применяются для автоматического создания программ. При этом используются такие подходы, как генетические алгоритмы [1], генетическое программирование [2] и программирование с экспрессией генов [3]. Эффективность методов напрямую зависит от способа представления программы.

Для ряда задач поведение системы удобно представлять в виде конечных автоматов. Такой подход используется в парадигме автоматного программирования [4]. Целесообразно поставить вопрос о создании метода генерации автоматов. В случае описания поведения системы, например, одним автоматом Мили, разработка программ в рамках автоматного программирования свелась бы к выделению входных и выходных воздействий автомата и заданию критерия оценки полученного решения – только к формализации задачи.

Конечные автоматы, созданные с помощью генетических алгоритмов, успешно применяются в задачах распознавания регулярных языков. Другим применением эволюционных вычислений к построению автоматов является проектирование клеточных автоматов. К сожалению, эти задачи имеют мало общего с практическими задачами автоматного программирования. Анализ достоинств и недостатков известных подходов к генерации автоматов, проведенный в данной работе, позволил предложить новый метод, основанный на представлении

автоматов с помощью деревьев решений, который может быть использован в автоматном программировании.

В заключение работы предложенный подход сравнивается с известными. Сравнение методик осуществляется на примере обобщенной задачи об умном муравье.

Глава 1.
Эволюционные алгоритмы

В этой главе кратко излагаются основные концепции эволюционных алгоритмов.

1.1. Эволюционные вычисления

Эволюционные вычисления – это метод оптимизации, базирующийся на эволюции популяции особей, в процессе которой определяется максимальное значение целевой функции. Этот метод применим для задач оптимизации плохо определенных функций. Основной идеей эволюционных вычислений является использование принципа естественного отбора, заключающегося в том, что наиболее приспособленные особи дают потомство, формирующее следующее поколение. В среднем, следующее поколение является более приспособленным к окружающей среде, чем предыдущее.

Сегодня эволюционные вычисления применяются в различных сферах науки: начиная с дизайна антенн для спутников и заканчивая расшифровкой генома человека.

1.2. Общая схема эволюционных алгоритмов

В эволюционных вычислениях [1] принято выделять подкласс эволюционных алгоритмов. При использовании эволюционных алгоритмов поиск оптимума ведется в пространстве гипотез X. Целевая функция $f : X \to R$ называется функцией приспособленности или фитнесс-функцией. Отличительной особенностью эволюционных алгоритмов является использование операции кроссовера, аналогичного скрещиванию в эволюции живых существ. Как правило, операция кроссовера принимает две особи и порождает одну. В дальнейшем рассматриваются только такие операции кроссовера. Особи эволюционного алгоритма принято называть хромосомами.

Общая схема эволюционного алгоритма может быть представлена на рис. 1.

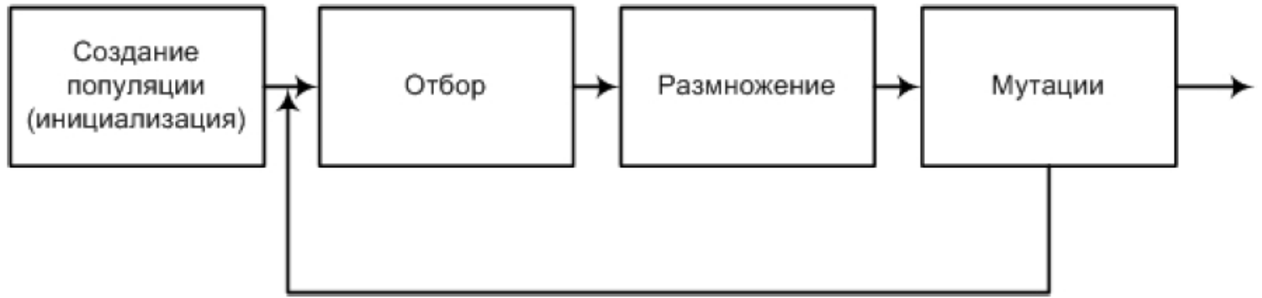

Рис. 1. Общая схема эволюционного алгоритма

Эта схема работает следуюшим образом:

- генерируется случайное множество хромосом;
- пока не выполняется критерий остановки:
    - отбираются хромосомы, в наибольшей степени удовлетворяющие условиям задачи – хромосомы, имеющие большее значение функции f;
    - отобранные хромосомы скрещиваются, порождая следующее поколение;
    - к некоторым особям потомства применяется мутация;

Операции инициализации популяции, отбора, кроссовера и мутации называются генетическими операциями. Конкретные реализации методов представления особей и генетических операций порождают различные разновидности эволюционных алгоритмов.

Математическое обоснование того, почему эволюционный алгоритм находит оптимальное решение, не известно. Для наиболее простых моделей генетических алгоритмов сходимость может быть доказана с помощью теоремы о схемах [1]. В работе [5] показано, что в при некоторых условиях использование операции кроссовера позволяет найти оптимум за конечное время, что неверно без использования такой операции.

1.3. Генетический алгоритм

Генетический алгоритм(ГА) был предложен Джоном Холландом в 1975 году в работе [1] и является первым из эволюционных алгоритмов. Особи в генети-

ческом алгоритме представляются в виде строк фиксированной длины (обычно используются битовые строки).

1.3.1. Инициализация

Создается случайная начальная популяция. Способ создания случайной строки может быть специфичным для задачи, однако обычно применяется случайная генерация строк с равномерным распределением над каждым символом строки.

1.3.2. Отбор

Наиболее распространенными методами отбора являются метод рулетки, ранговый метод, элитизм и турнирный метод. *Метод рулетки* состоит в том, что для каждой особи шанс выжить прямо пропорционален значению ее функции приспособленности. При использовании *рангового метода*, шанс выжить прямо пропорционален рангу особи – ее порядку в отсортированном по значению фитнесс-функции списку особей. *Метод элитизма* заключается в том, что отбрасываются все особи, кроме заданной доли наиболее приспособленных. Наконец, при использовании турнирного метода, выбираются две случайных особи. После этого с вероятностью p выживает более приспособленная особь, а с вероятностью $1 - p$ – менее приспособленная. Процесс повторяется до тех пор, пока не останется заданное количество особей.

1.3.3. Кроссовер

Кроссовер на строках фиксированной длины реализуют следующим образом: часть символов копируется из одной строки, остальные символы копируются из другой строки. Будем говорить, что результатом кроссовера двух строк $a_1 \dots a_n$ и $b_1 \dots b_n$ с маской $m_1 \dots m_n$, где $m_i \in [0, 1]$ является строка $c_1 \dots c_n$, для которой

$$c_i = \begin{cases} a_i, m_i = 0; \\ b_i, m_i = 1. \end{cases}$$

Таким образом, методы кроссовера отличаются методом создания маски. Наиболее распространены методы одноточечного кроссовера, двухточечного кроссовера и однородного кроссовера.

В *методе одноточечного кроссовера* используется маска

$$m_i = \begin{cases} 0, i < k; \\ 1, i \geq k, \end{cases}$$

где k – случайное число от 1 до n.

В *методе двухточечного кроссовера* используется маска

$$m_i = \begin{cases} 0, i < k \text{ или } i > l; \\ 1, k \leq i \leq l, \end{cases}$$

где k и l – случайные числа от 1 до n, $k \leqslant l$. Часто используется модификация этого способа, когда $l = k + \frac{n}{2}$.

В *методе однородного кроссовера* маска является случайной с равномерным распределением.

1.3.4. Мутации

Наиболее распространен следующий метод мутации – выбирается случайная позиция в строке и символ в этой позиции заменяется на случайный.

1.4. Генетическое программирование

Генетическое программирование является эволюционным алгоритмом, особями которого являются компьютерные программы. Первые результаты с использованием этой методики были получены в 80-х годах Крамером [6]. В последнее десятилетие в связи с резким увеличением мощности компьютеров возрос интерес к генетическому программированию как к средству автоматизации разработки ПО. В настоящее время с помощью генетического программирования получен ряд результатов, превосходящих аналогичные результаты, полученные людьми, например, созданы сортирующие сети, квантовый алгоритм для задачи Гровера и т. д.

Ключевой идеей генетического программирования является представление программы на достаточно высоком уровне абстракции, позволяющем учитывать специфику и структуру компьютерных программ. Ниже рассматривается генетическое программирование в концепции Джона Козы, описанной в работе

[2]. Этот метод использует преставление программ в виде деревьев разбора [7]. В листьях дерева находятся терминалы, а во внутренних узлах – нетерминалы. В связи со спецификой представления в генетическом программировании наиболее часто используются функциональные языки, в которых программы естественным образом представляются в виде последовательности вычислений.

Существенно ускоряет вывод техника выделения наиболее часто встречаемых в хороших решениях поддеревьев, которые могут быть вынесены в отдельные процедуры и добавлены в список нетерминалов.

1.4.1. Инициализация

Создание случайной особи является созданием случайного дерева разбора. Как правило, достаточно сгенерировать синтаксически корректные деревья разбора. Это можно сделать, например, с помощью рекурсии, которая генерирует поддерево, возвращающее результат определенного типа. С определенной долей вероятности создается терминал совместимого типа, иначе создается один из нетерминалов. После этого с учетом синтаксиса генерируется требуемое число детей, соответствующих аргументам нетерминала. Часто оказывается полезным устанавливать ограничение на высоту дерева.

1.4.2. Отбор

В генетическом программировании используются те же методы отбора, что и в генетических алгоритмах.

1.4.3. Кроссовер

Кроссовер для деревьев разбора определяется следующим образом – случайное поддерево первой особи заменяется на случайное поддерево второй особи. Пример операции кроссовера приведен на рис. 2.

1.4.4. Мутация

Мутация реализуется путем замены одного из поддеревьев на случайно сгенерированное поддерево (рис. 3).

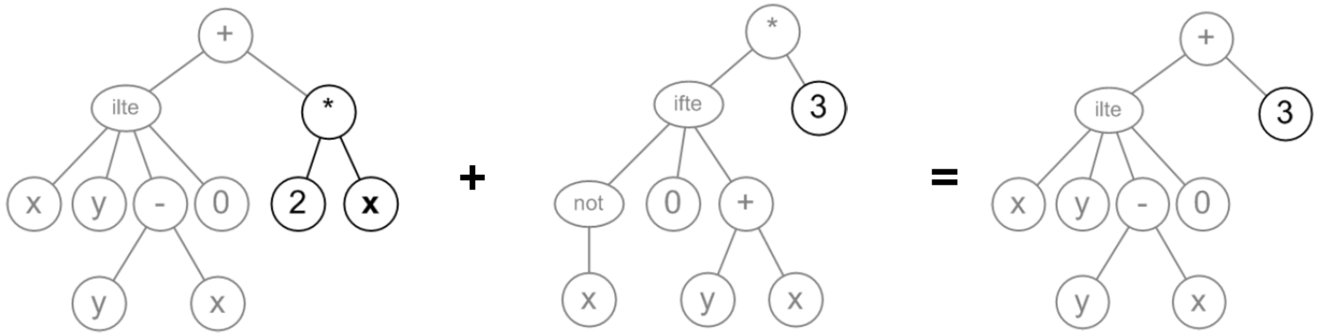

Рис. 2. Пример операции кроссовера

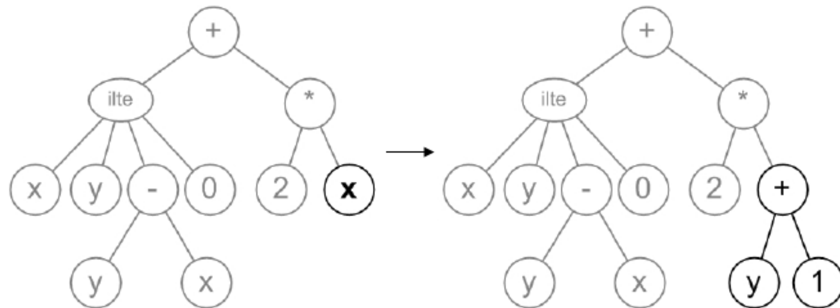

Рис. 3. Пример операции мутации

1.4.5. Поддержка корректности

При использовании определенных таким образом операций кроссовера и мутации возможна ситуация, при которой созданная особь будет соответствовать некорректному выражению. Эта проблема может решаться следующими способами:

- повторять кроссовер, пока не будет получено корректное дерево разбора;
- всегда добавлять особь в популяцию, но приспособленность некорректных выражений считать нулевой;
- выполнять операции таким образом, что результирующие деревья вычислений всегда будут корректны – например, заменяя поддерево только на поддерево, возвращающее совместимый тип.

1.5. Программирование с экспрессией генов

Программирование с экспрессией генов [3] является интересным компромиссом между генетическим программированием и генетическими алгоритмами. Особями в этом эволюционном алгоритме являются префиксные записи выражений, хранящиеся как строки фиксированной длины. Для того, чтобы по строке всегда можно было построить выражение, вводится требование корректности, заключающееся в том, чтобы с определенного места в строке встречались только терминалы. Заметим, что возможна ситуация, когда выражение строится по некоторому префиксу строки, оставшаяся часть которой ничему не соответствует. Кроме того, это требование не помогает в случае, когда поддерживаются разные типы.

Генетические операции при применении данного подхода реализуются аналогично операциям над строками фиксированной длины, используемым в генетических алгоритмах. Единственное отличие заключается в поддержке требования корректности строк, которое сравнительно легко соблюдать, видоизменив операции инициализации и мутации.

1.6. Сравнение подходов

К достоинствам генетических алгоритмов по сравнению с генетическим программированием относится простота реализации. С другой стороны, представление в виде строк слабо структурировано и не применимо к генерации программ.

Генетическое программирование, как правило, быстро находит хорошие простые приближения и после этого постепенно усложняет их структуру. Таким образом, эволюция сама определяет эффективный способ представления решений. Недостатком является сложная структура деревьев вычислений, накладывающая множество ограничений. Возникает необходимость поддержки корректности выражений.

Программирование с экспрессией генов сочетает гибкость генетического программирования с простотой реализации генетических алгоритмов. Однако, в случае, когда необходима поддержка типов, этот метод не применим.

Выводы по главе 1

1. Рассмотрены различные эволюционные алгоритмы.

2. Проведен сравнительный анализ различных подходов.

Глава 2.
Применение эволюционных алгоритмов для генерации автоматов

В этой главе проводится краткий обзор существующих техник использования эволюционных алгоритмов для автоматического создания автоматов, рассматриваются некоторые классические задачи.

2.1. Конечные детерминированные автоматы

2.1.1. Определение конечного детерминированного автомата

Конечный детерминированный автомат – это математическая абстракция, позволяющая описывать пути изменения состояний объекта в зависимости от текущего состояния и входных данных. Ограничением является то, что множество возможных состояний автомата конечно. Конечный детерминированный автомат может быть описан пятеркой $\{Q, S, F, \Sigma, \delta\}$, где:

- Q – конечное множество состояний;
- $S \in Q$ – начальное состояние;
- $F \subset Q$ – множество допускающих состояний;
- Σ – входной алфавит;
- $\delta : Q \times \Sigma \to Q$ – функция переходов.

Работа конечного автомата может рассматриваться как последовательность тактов. За каждый такт происходит считывание очередного символа $a \in \Sigma$ из входной строки. После этого автомат переходит из состояния q в состояние $\delta(q, a)$. Первоначально автомат находится в состоянии S. Автомат допускает цепочку $w \in \Sigma^*$ тогда и только тогда, когда после считывания w он оказывается в одном из допускающих состояний.

Будем говорить, что автомат A допускает язык L над алфавитом Σ, если $\forall w \in \Sigma^*$, $w \in L$ тогда и только тогда, когда A допускает w.

2.1.2. Конечный автомат Мили

В ряде задач удобно описывать поведение объекта в виде конечного автомата Мили [4]. Такой конечный автомат может быть представлен шестеркой $\{Q, S, X, Y, \delta, \gamma\}$, где:

- Q–конечное множество состояний;
- $S \in Q$–начальное состояние;
- X – конечное множество входных воздействий;
- Y – конечное множество выходных воздействий;
- $\delta : Q \times X \to Q$ – функция переходов;
- $\lambda : Q \times X \to Y$ – функция действий.

При этом элементы множеств Q, X и Y связаны в абстрактном времени следующими отношениями:

$$\begin{cases} Q(t+1) = \delta(Q(t), X(t)); \\ Y(t) = \lambda(Q(t), X(t)). \end{cases}$$

В рамках парадигмы автоматного программирования [8] конечный автомат Мили может быть обобщен за счет понятия предикатов – отображений вычислительных состояний в логические переменные. Заметим, что они не могут быть вынесены в состояния автомата, так как зависят от текущей ситуации. Переход будет осуществляться в зависимости не только от входных воздействий и текущего состояния, но и от значений предикатов. Предикаты также называются входными переменными. Таким образом, если множество предикатов обозначить за P, то указанные функции примут следующий вид:

$$\begin{cases} \delta : Q \times X \times 2^P \to Q; \\ \lambda : Q \times X \times 2^P \to Q. \end{cases}$$

Заметим, что от предикатов можно избавиться, увеличив набор входных воздействий. Однако такой способ усложняет логику системы для восприятия человеком. Кроме того, при этом множество входных воздействий растет экспоненциально от числа входных переменных.

2.2. Клеточные автоматы

Клеточные автоматы были предложены в работе [9]. Клеточный автомат представляет собой совокупность одинаковых клеток, соединенных между собой определенным способом, образуя однородную решетку. Каждая клетка представляет собой конечный детерминированный автомат, изменяющий состояние в зависимости от своего состояния и состояний соседних клеток. Изменение состояний всех клеток происходит одновременно. Большой интерес к клеточным автоматам вызван тем, что они являются универсальной моделью вычислений, аналогично машине Тьюринга [10]. Более того, они способны осуществлять параллельные вычисления.

Одномерный клеточный автомат – клеточный автомат, решетка которого одномерна. Конечный автомат, соответстствующий клетке изменяет состояние в зависимости от состояний клеток, удаленных не более, чем на R – радиус автомата. Пример одномерного клеточного автомата приведен на рис. 4.

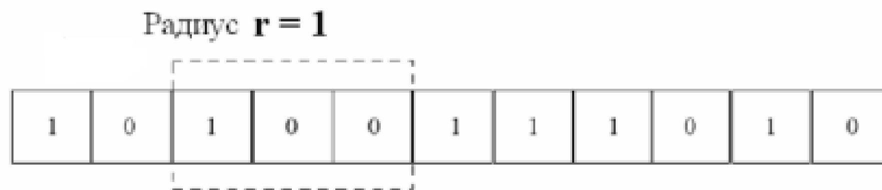

Рис. 4. Пример одномерного клеточного автомата

Генерация клеточных автоматов сводится к генерации конечного автомата, описывающего поведение отдельной клетки (разд. 2.5).

2.3. Некоторые задачи генерации автоматов

2.3.1. Распознавание регулярных языков

Множество регулярных языков совпадает с множеством языков, распознаваемых конечными автоматами [10]. В связи с этим возникает задача построения автомата, распознающего регулярный язык, соответствующий множеству примеров. Примерами такой задачи могут быть построение автомата, распознающего рукописный текст, или построение автомата, распознающего двухсложные слова [12].

2.3.2. Задача об умном муравье

Задача о муравье является классической задачей генерации автоматов и подробно рассмотрена в работе [2]. Многие предлагаемые модификации генетических алгоритмов применяют ее для сравнения эффективности. Задача состоит в нахождении оптимальной стратегии муравья, передвигающегося по игровому полю. Игровое поле представляет собой двумерный тор размера 32 на 32 клетки, в определенных клетках которого расположена еда (рис. 5). Еда расположена вдоль некоторой ломаной, но не во всех ее клетках.

Муравей видит, расположена ли еда на клетке непосредственно перед ним. За один ход может быть сделано одно из следующих действий:

- шаг вперед;
- поворот направо;
- поворот налево.

Если муравей после хода попадает на клетку, где находится еда, то он ее съедает. Съеденная муравьем еда не восполняется. Муравей жив на протяжении всей игры – еда не является необходимым ресурсом для его существования. Никаких других персонажей на поле нет. Ломаная и расположение еды на ней фиксированы.

Игра длится не более 200 ходов, на каждом из которых муравей совершает одно из разрешенных действий. Результатом игры является количество съеденной за это время еды, а положительным результатом – съедание всей еды.

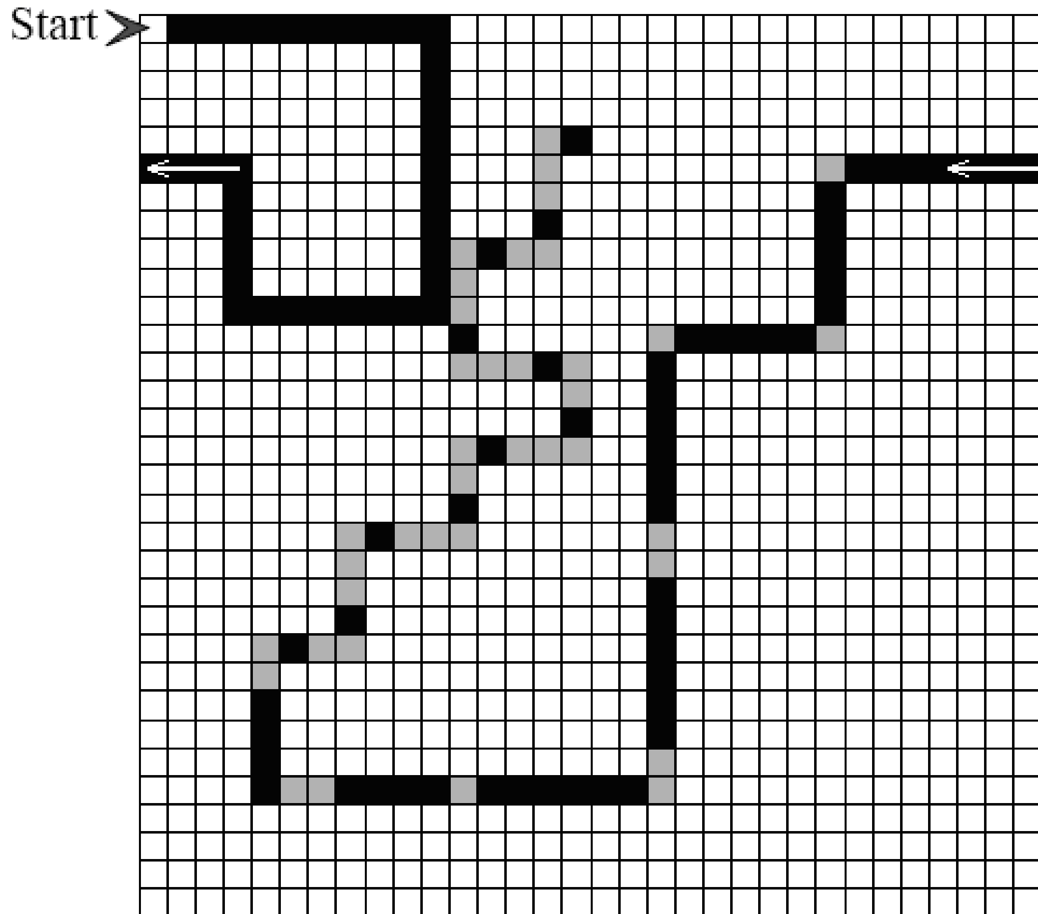

Рис. 5. Игровое поле

Поиск стратегий поведения муравья производится среди конечных детерминированных автоматов. Искомый автомат является автоматом Мили, имеющим одну входную и три выходных переменных. Оптимизация может проводится по числу ходов и/или состояний автомата.

2.3.3. Обобщенная задача о муравье

Задачу о муравье можно модифицировать следующим образом. Пусть муравей видит перед собой не одну клетку, а определенную область (рис. 6). В случае, когда используется фиксированная карта, эта модификация упрощает задачу.

Возможно обобщение задачи на случайные карты, когда еда в каждой клетке существует с заданным параметром μ. При этом необходимо найти такую стратегию поведения муравья, при которой математическое ожидание ко-

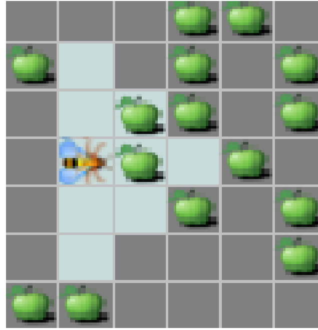

Рис. 6. Видимые муравью клетки

личества съеденной еды максимально. В дальнейшем будем называть эту задачу обобщенной задачей о муравье.

Поиск стратегий, как и в исходной задаче, производится среди автоматов Мили, но число входных воздействий увеличивается до восьми.

Оптимальная стратегия в данной задаче сильно зависит от значения параметра μ. Например, при $\mu = 1$ оптимальной является стратегия, при которой муравей посещает наибольшее число клеток, а в случае $\mu \approx 0$ – стратегия, когда муравей увидит наибольшее число клеток и обязательно посетит клетку, в которой увидит еду.

2.3.4. Задача классификации плотности

Задача классификации плотности является классической задачей генерации клеточных автоматов. Рассмотрим одномерный клеточный автомат, действующий на отрезке фиксированной длины, замкнутом в кольцо. Пусть число возможных состояний автомата равно двум. Назовем их q_0 и q_1. Плотностью конфигурации ρ назовем отношение число клеток, исходно находящихся в состоянии q_0, к общему числу клеток. Будем говорить, что клеточный автомат распознает плотность конфигурации, если он приходит в одно из состояний:

- все клетки находятся в состоянии q_0, если $\rho > 0.5$;
- все клетки находятся в состоянии q_1, если $\rho < 0.5$.

Задача состоит в построении клеточного автомата, распознающего плотность наибольшего числа конфигураций. Число входных переменных определяется радиусом автомата, выходных воздействий нет.

2.4. Применение генетических алгоритмов

Генетические алгоритмы успешно используются для решения задач генерации автоматов. Напомним, что для использования генетических алгоритмов необходимо задать способ представления автомата в виде строки.

2.4.1. Представление автомата в виде строк фиксированной длины

В работах [13, 14] используется следующий алгоритм кодирования автомата в строку:

1) фиксируется число состояний автомата;

2) логика работы представляется в виде таблицы, где для каждого состояния и каждого входного воздействия записывается пара (Новое Состояние, Действие);

3) таблица записывается в виде строки.

Поясним, как таблица преобразуется в строку. На рис. 7 изображен автомат управления муравьем. Здесь ситуации, когда на клетке перед муравьем есть еда, соответствует входное воздействие F, когда ее нет – входное воздействие N. Выходные воздействия L, R, M обозначают такие действия муравья, как поворот налево, поворот направо и движение вперед соответственно. Поведение муравья может быть представлена в виде таблицы переходов автомата табл. 1.

Теперь, занумеровав некоторым образом входные и выходные воздействия, можно записать элементы этой таблицы в строку, выписав двойки (Новое Состояние, Действие), соответствующие переходам. Для этого переходы упорядочиваются в порядке возрастания начального состояния, а в случае равенства – по номеру входного воздействия.

Возможно применение одного из двух подходов:

- Использование битовых строк. Такая техника налагает следующее ограничение – для того, чтобы строке всегда соответствовал корректный автомат необходимо, чтобы число его состояний и выходных воздействий было степенью двух. Как правило, это ведет к добавлению фиктивных выходных воздействий автомата. Например, в задаче о муравье можно

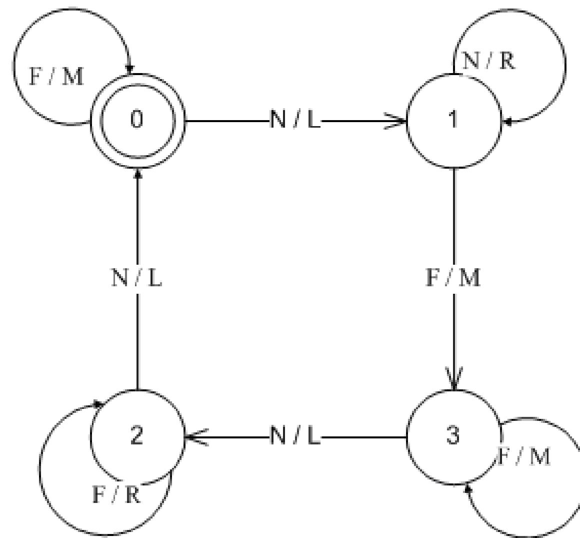

Рис. 7. Пример автомата управления муравьем

Таблица 1. Таблица переходов автомата

Состояние	Вход	Новое Состояние	Действие
0	N	1	L
0	F	0	M
1	N	1	R
1	F	3	M
2	N	0	L
2	F	2	R
3	N	2	L
3	F	3	M

добавить действие „Стоять на месте“ (NO). Заметим, что фактически оно не является необходимым, и может быть устранено алгоритмом, аналогичным ϵ-замыканию [7].

- Использование строк над числами. При таком способе записи проблем с корректностью не возникнет. Заметим, что алфавит для состояний и выходных воздействий может различаться. Этот метод эквивалентен методу, использующему представление автомата в виде таблицы переходов [12].

Кроме того, возможно хранение начального состояния в строке. Однако, анализ, проведенный в работе [14], показывает, что такой вариант не дает выигрыша по сравнению с зафиксированным начальным состоянием.

Для задачи построения автомата, распознающего регулярный язык, кроме того необходимо вывести для всех состояний бит, указывающий на то, является ли это состояние допустимым.

Ниже в качестве примера приведем строку, соответствующую таблице переходов автомата (табл. 1).

1L 0M 1R 3M 0L 2R 2L 3M

2.4.2. Эвристика, ускоряющая вывод автоматов

Вывод автоматов с помощью генетических алгоритмов может иметь низкую эффективность в силу следующих причин:

1. Автомат, соответствующий сгенерированной строке, может иметь недостижимые состояния. Информация, хранящаяся для этих состояний, является генетическим мусором.

2. Представление не является естественным – двум разным строкам может соответствовать один и тот же автомат. Это приводит к тому, что изоморфные автоматы соревнуются друг с другом, замедляя работу генетического алгоритма.

Метод *MTF(Move To Front)*, описанный в работе [15], помогает устранить эти недостатки. Он заключается в перенумеровывании вершин автомата для приведения всех изоморфных автоматов к каноническому виду. Для этого из стартовой вершины запускается обход в ширину. После этого вершины нумеруются в порядке обхода. Заметим, что недостижимые вершины автоматически помещаются в конец строки, соответствующей автомату. Пример преобразования приведен на рис. 8.

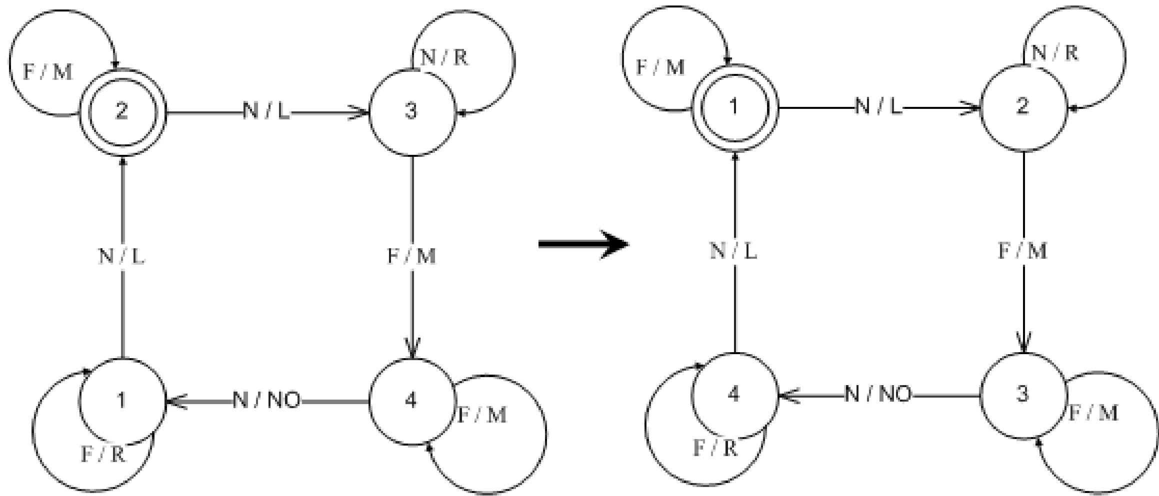

Рис. 8. Пример преобразования *MTF*

Анализ, проведенный в [15], подтверждает предположение о том, что эвристика *MTF* ускоряет вывод автоматов.

2.5. Применение генетического программирования

Генетическое программирование применялось для генерации клеточных автоматов в задаче классификации плотности [2]. Так как выходных воздействий в этой задаче нет, необходимо определить только состояние клетки на следующем шаге. Существенным является то, что возможных состояний всего два, поэтому функция переходов может быть представлена как булева. Она была представлена в виде дерева, вершинами которого являлись различные булева функции и входные переменные.

Аналогичный подход был использован для решения задачи классификации плотности с помощью программирования с экспрессией генов [3].

Заметим, что идея оказывается неприменимой при добавлении выходных воздействий или увеличении числа состояний.

2.6. Анализ достоинств и недостатков известных подходов

Достоинством генетических алгоритмов над строками фиксированной длины является простота реализации. Однако этот метод плохо подходит для задач со многими входными переменными. В силу того, что приходится задавать переход автомата для всех комбинаций значений переменных, длина строки, соответствующей автомату, растет экспоненциально.

Генетическое программирование позволяет добиться лучших, чем традиционные генетические алгоритмы, результатов в задаче классификации плотности. Недостатком является то, что примененные подходы существенно используют специфику задачи и не могут быть естественным образом обобщены для решения других задач проектирования автоматов.

Выводы по главе 2

1. Рассмотрены некоторые задачи генерации автоматов.
2. Выполнен обзор существующих реализаций эволюционных алгоритмов для генерации автоматов на примере приведенных задач.
3. Отмечены достоинства и недостатки известных подходов.

Глава 3. Технология генетического программирования для генерации автоматов на основе деревьев решений

В этой главе предлагается новый метод применения эволюционных алгоритмов для генерации автоматов, основанный на их представлении с помощью деревьев решений. Демонстрируется возможность задания ограничений на целевой автомат. Это придает методу дополнительную гибкость.

3.1. Деревья решений

Дерево решений является удобным способом задания дискретной функции, зависящей от конечного числа дискретных переменных. Оно представляет собой помеченное дерево, метки в котором расставлены по следующему правилу:

- внутренние узлы помечены символами переменных;
- ребра – значениями переменных;
- листья – значениями искомой функции.

Для определения значения функции по значениям переменных необходимо спуститься от корня до листа, и выдать значение, которым помечен полученный лист. При этом из вершины, помеченной переменной x, переход производится по тому ребру, которое помечено тем же значением, что и значение x.

Пример дерева решений изображен на рис. 9. Здесь для простоты показано дерево, реализующее булеву функцию от переменных a, b и c.

Как правило, в большинстве практических задач дерево решений требует значительно меньше памяти по сравнению с заданием функции на всех наборах значений входных переменных (с помощью таблиц истинности).

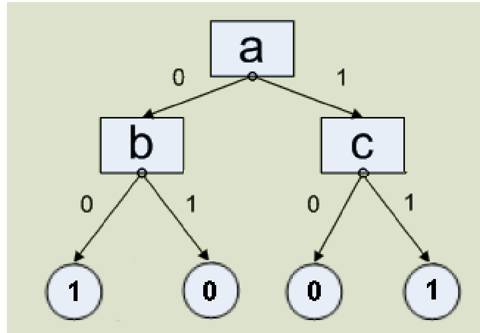

Рис. 9. Пример дерева решений

3.2. Представление автомата с помощью деревьев решений

Опишем предлагаемый метод представления автомата с помощью деревьев решений. Для задания автомата необходимо выразить его функции переходов и выходов с помощью деревьев решений. Возможно осуществить следующее преобразование автомата: вместо задания функций переходов и выходов для автомата в целом, представим эти функции для каждого состояния. Более формально: зададим для каждого состояния $q \in Q$ функцию $\sigma_q : X \to Q \times Y$, такую что $\sigma_q(x) = (\delta(q, x), \lambda(q, x)), \forall x \in X$.

Функции σ_q соответствуют функциям переходов и действий из состояния q. Каждая из этих функций может быть выражена с помощью дерева решений. В этих деревьях переменными являются входные переменные автомата, а множеством значений – все возможные пары *(Новое Состояние, Действие)*. Таким образом, автомат в целом задается упорядоченным набором деревьев решений.

На рис. 10 приведен пример автомата Мили. При этом a, b, c – входные переменные булевого типа, а L, R, F – выходные воздействия. Этот автомат, представленный с использованием деревьев решений, приведен на рис. 11.

При таком представлении функции переходов из каждого состояния можно ожидать значительного уменьшения длин хромосом по сравнению с хранением ее в виде таблицы. Как правило, в автоматном программировании [4] в каждом состоянии важен лишь сравнительно небольшой набор из всего множества входных переменных автомата. Таким образом, представление автомата с

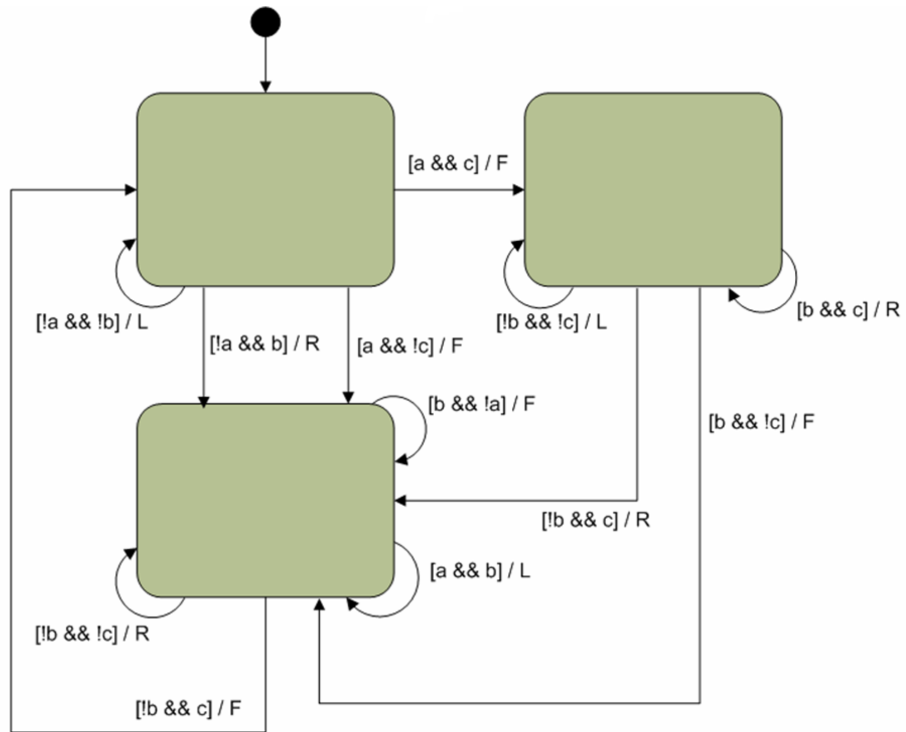

Рис. 10. Пример автомата Мили

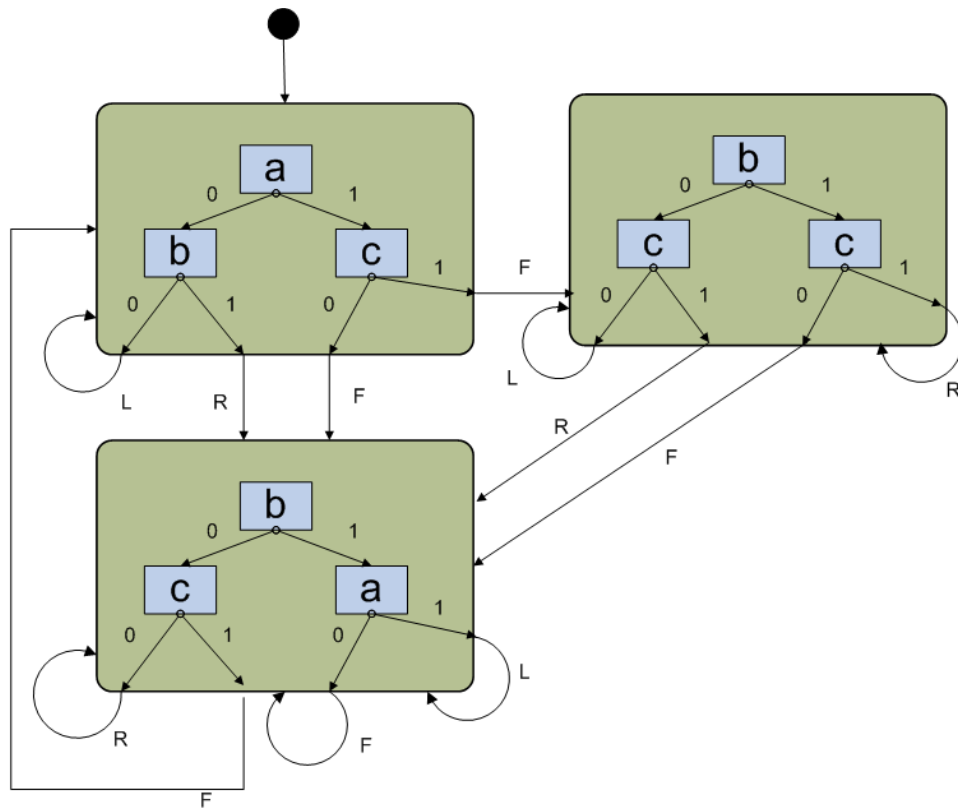

Рис. 11. Представление автомата Мили с помощью деревьев решений

помощью набора деревьев решений должно значительно сократить длину хромосомы, а, следовательно, и ускорить генерацию.

3.3. Генетические операции

Для такого представления автоматов в эволюционных алгоритмах необходимо задать генетические операции. Они могут быть следующими:

- случайное порождение автомата — в каждом состоянии создается случайное дерево решений;
- скрещивание автоматов — скрещиваются деревья решений в соотвествующих состояниях;
- мутация автомата — в случайном дереве решений выполняется мутация;
- отбор — можно использовать любой из методов, используемых в генетических алгоритмах.

Здесь считается, что число состояний в автомате фиксировано, и поэтому противоречий при выполнении определенных таким образом операций не возникнет.

После определения операций над набором деревьев решений, определим генетические операции над собственно деревьями решений. Это можно сделать с помощью операций, аналогичных операциям, используемым в генетическом программировании над деревьями разбора [2]. В деревьях решений переменные являются нетерминалами, а значения функции — терминалами. При этом арность нетерминала, соответствующего переменной x, равна мощности множества возможных значений переменной. Приведем описание генетических операций над деревьями решений:

- Случайное порождение дерева решений — случайным образом выбирается метка: либо одно из возможных значений функции переходов, либо одна из переменных. После этого создается вершина, помеченная выбранной меткой. При этом если была выбрана переменная, то рекурсивно генерируются дети вершины, иначе вершина становится листом дерева.

- Мутация — выбирается случайный узел в поддереве. После этого поддерево, соответствующее выбранному узлу, заменяется на случайно сгенерированное (рис. 12).

- Скрещивание — в скрещиваемых деревьях выбираются два случайных узла. После этого поддерево, соответствующее выбранному узлу в первом дереве, заменяется поддеревом, соответствующим узлу второго дерева (рис. 13).

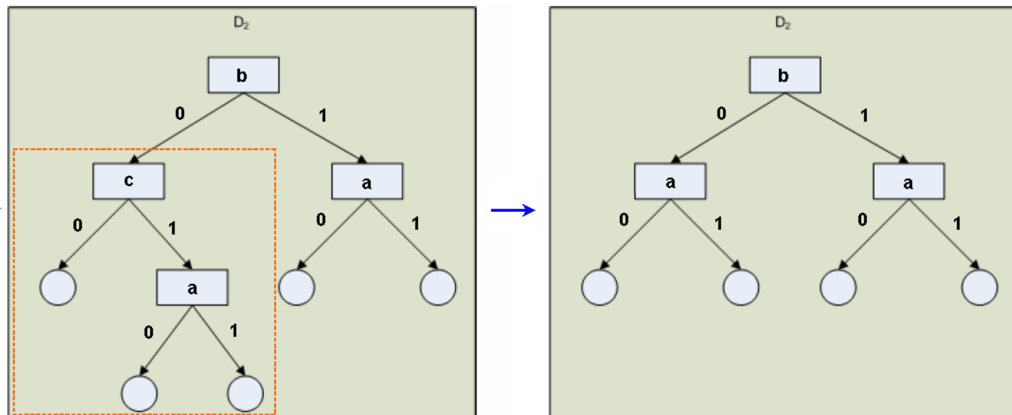

Рис. 12. Мутация деревьев решений

Отметим, что заданные таким образом операции могут порождать деревья, в которых некий атрибут встречается на пути от корня до листа дважды (например, дерево, полученное в результате скрещивания на рис. 13). Такие деревья являются корректными - определяют функцию на любом наборе значений атрибутов единственным образом, однако имеют недостижимые вершины. Действительно, если атрибут встречается на пути дважды, то его значение уже известно, следовательно, одна из ветвей недостижима ни при каких значениях предикатов. Появление недостижимых ветвей может влиять на эволюцию отрицательным образом по следующим причинам:

- Появление недостижимых ветвей ведет к увеличению высоты деревьев, экспоненциально увеличивая память, необходимую на хранение популяции. В условиях ограниченной памяти это ведет к уменьшению популяции.

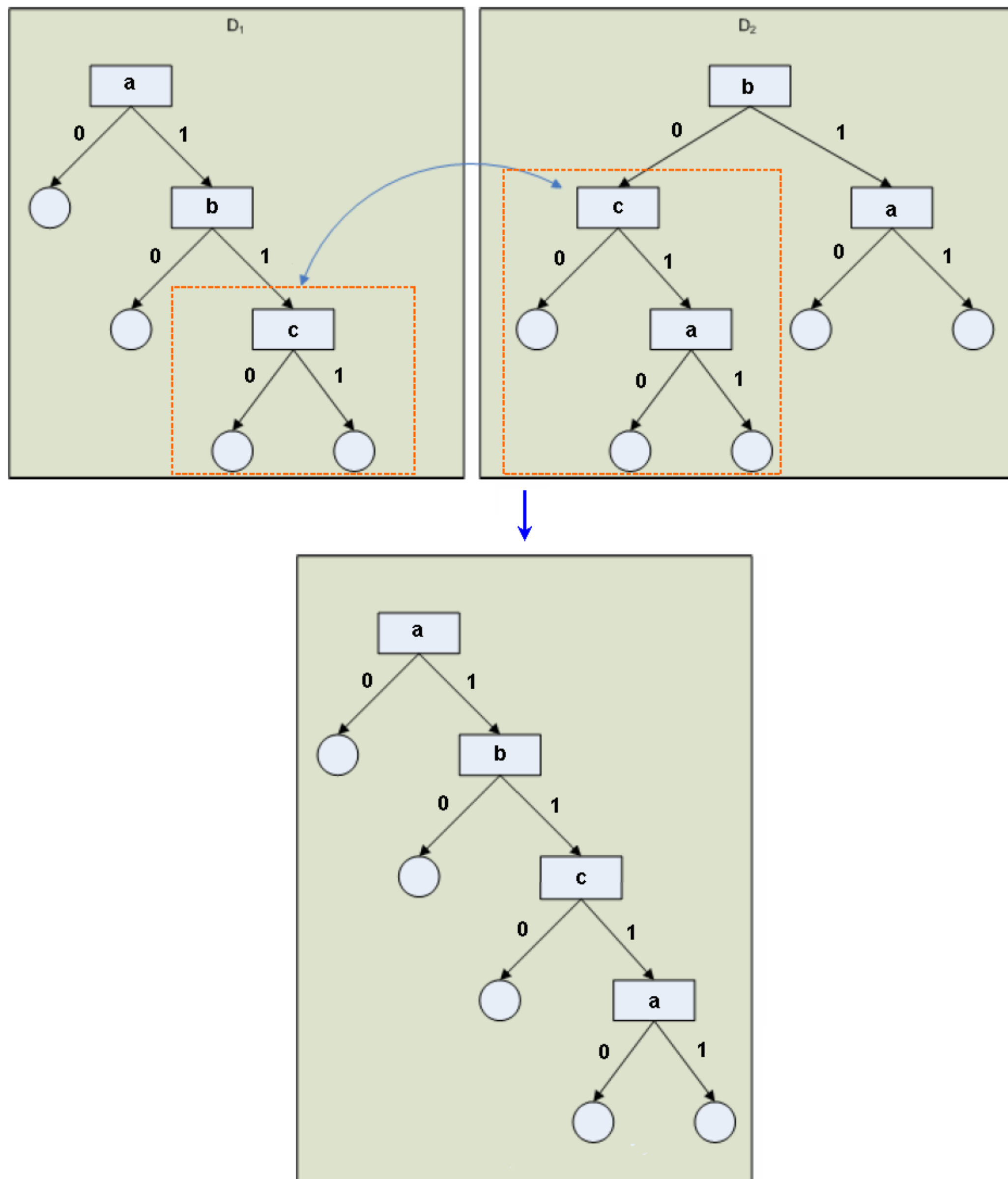

Рис. 13. Скрещивание деревьев решений

- Недостижимые ветви дерева могут являться плохими приближениями искомой функции (так как не учитываются при подсчете функции приспособленности), но при этом, однако, копироваться в потомков, замедляя генерацию.

Таким образом, необходимо ввести операцию обрезки – удаление недостижимых ветвей. Операция может быть выполнена следующим образом: узел, одна из дочерних вершин которого недостижима, заменяется на свою достижимую дочернюю вершину. На рис. 14 приведен пример обрезки недостижимых

ветвей. Операция обрезки должна выполняться после генетических операций скрещивания и мутации.

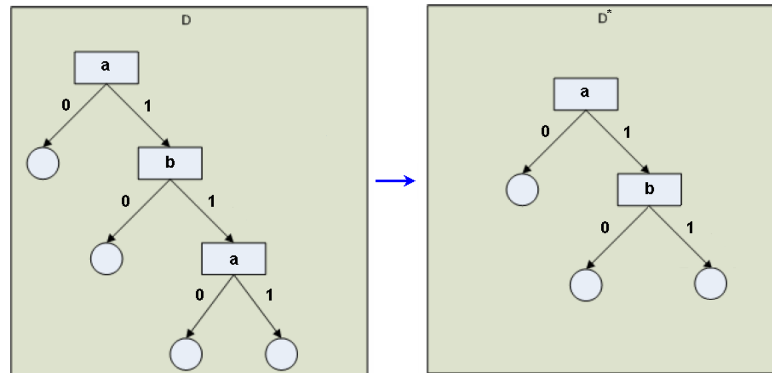

Рис. 14. Пример обрезки недостижимых ветвей

3.4. Простой пример применения

Ниже демонстрируется простой пример применения разработанного метода к решению задачи „Умный Муравей“. В данной задаче используется одна входная переменная булевого типа. Обозначим его X. Выходные воздействия поворота налево, направо и шага вперед обозначим L, R и F соответственно. В случае, если дерево решений, подвешенное в состоянии, не использует никаких входных переменных будем рисовать один переход.

Для простоты покажем работу метода на примере популяции, размер которой равен четырем. Число состояний в каждом из автоматов положим равным двум.

Допустим, на определенном шаге генерации популяция имеет вид, приведенный на рис. 15. Здесь под каждым из автоматов популяции приведено значение фитнесс-функции – количество съеденной за отведенное время еды.

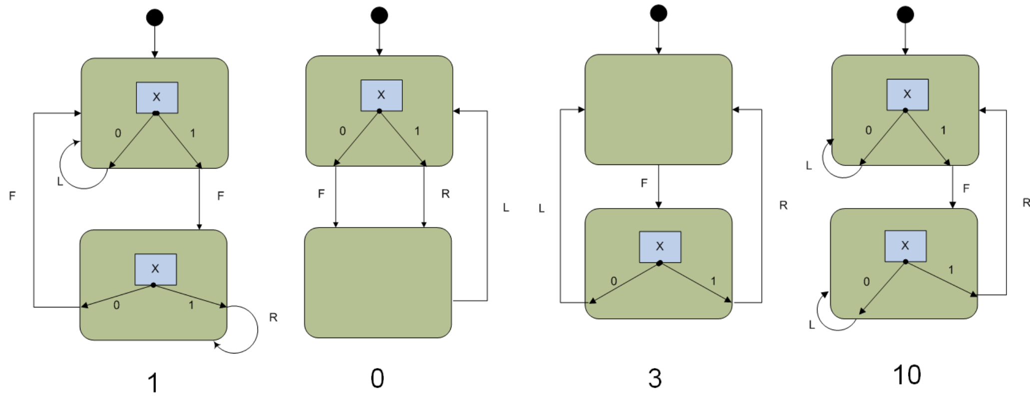

Рис. 15. Популяция автоматов

В случае, когда в качестве метода отбора применяется метод рулетки, для производства потомства отбирается несколько лучших представителей. Пусть в рассматриваемом примере отбираются два лучших автомата популяции. Тогда, два оставшихся автомата покидают популяцию, а их место занимают потомки отобранных для размножения представителей. Порождение потомства путем скрещивания показано на рис. 16.

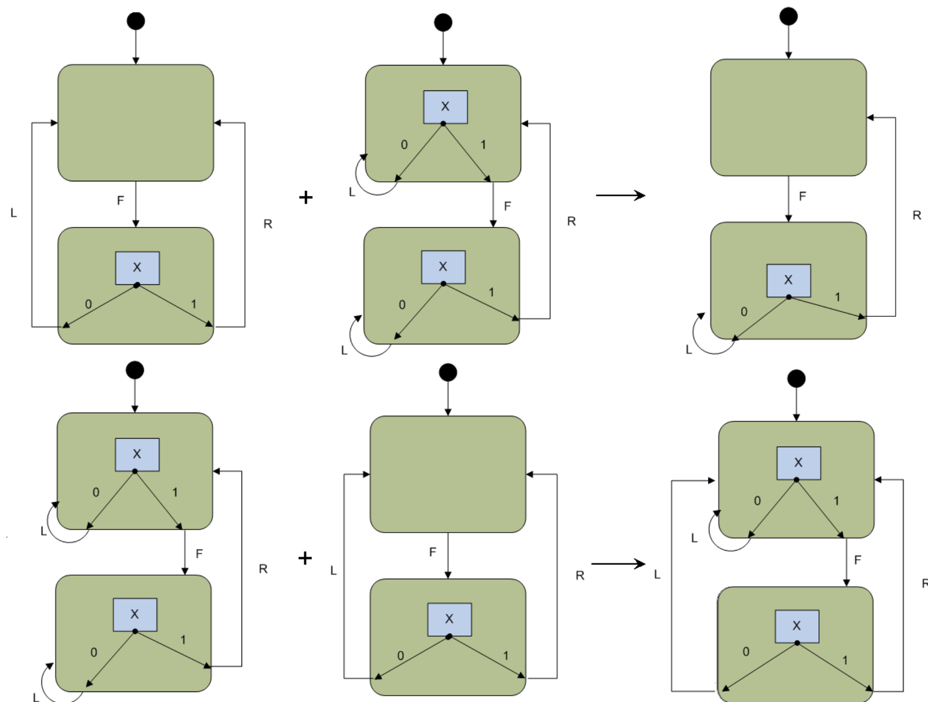

Рис. 16. Порождение потомства

После этого порожденное скрещиванием потомство подвергается действию мутаций. Операция совершается над каждым из порожденных автоматов. С определенной долей вероятности происходит мутация в одном из состояний автомата — над соответствующим деревом решений. На рис. 17 приведен пример действия мутаций на порожденное потомство.

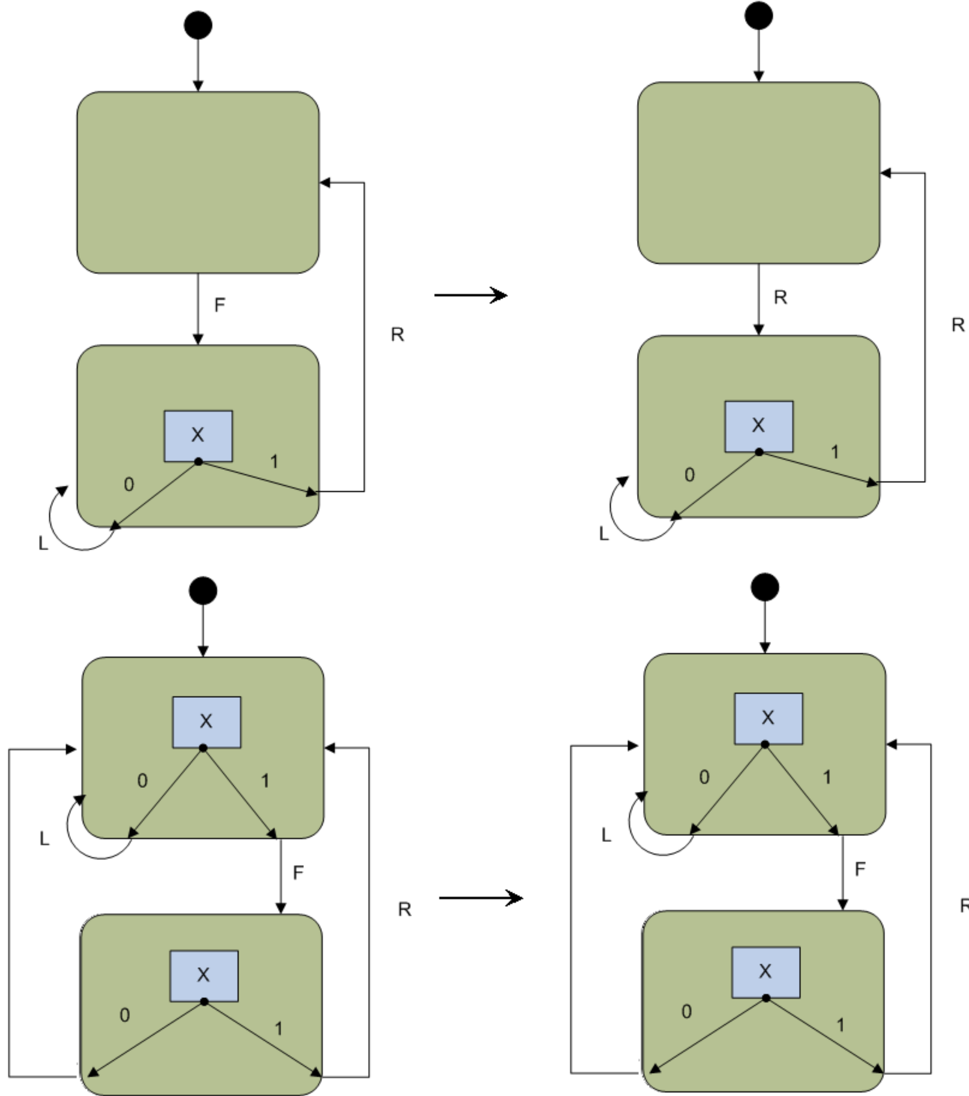

Рис. 17. Мутация потомства

Таким образом, популяция преобразовалась, как показано на рис. 18:

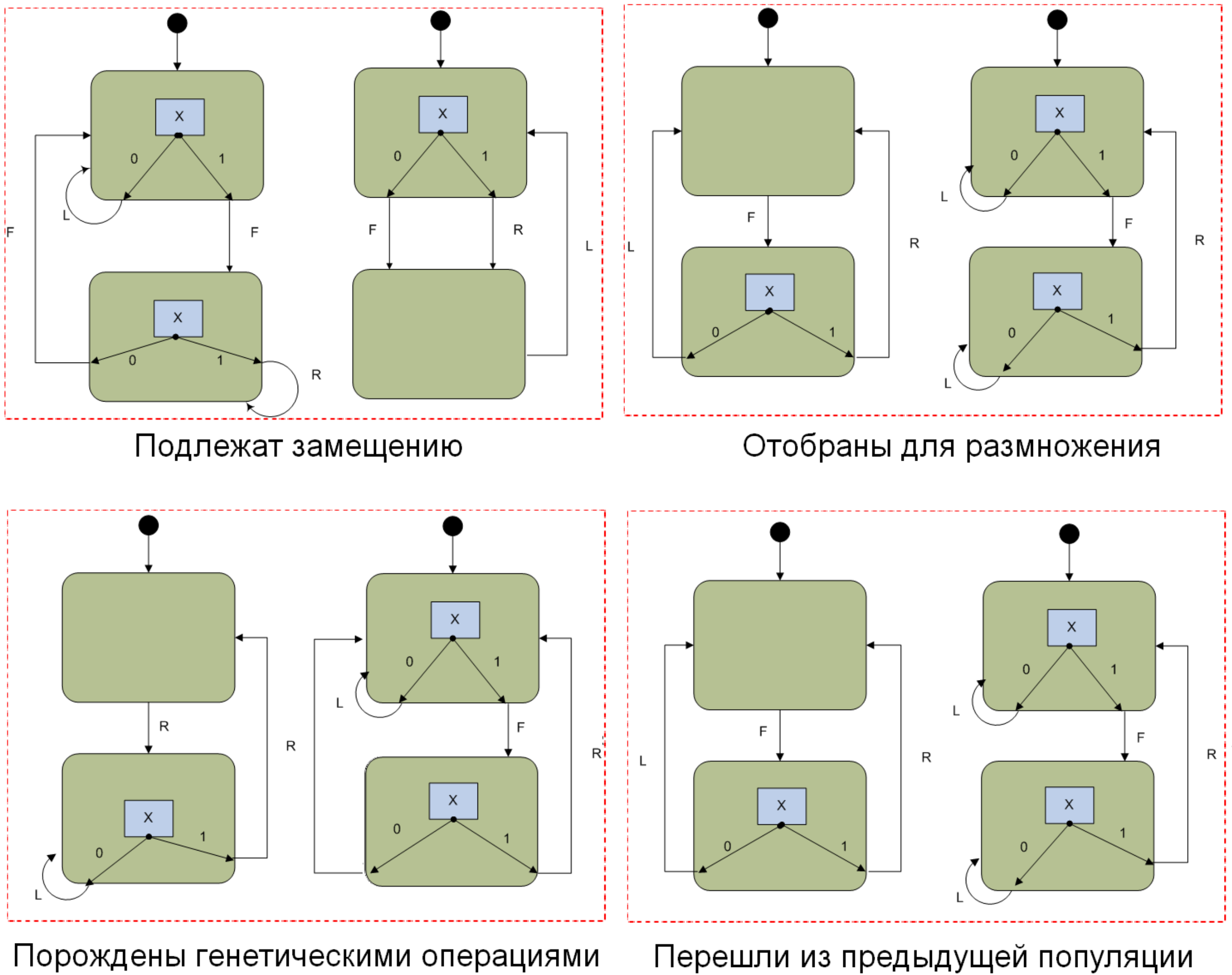

Рис. 18. Преобразование популяции

Многократное повторение подобных шагов генерирует исходный автомат.

3.5. Задание ограничений на целевой автомат

Иногда при применении эволюционных алгоритмов оказывается удобным задавать ограничения на целевую особь. Заметим, что для автоматов было бы полезно уметь ограничивать сложность логики и число переходов. При использовании деревьев решений в качестве представления автомата, сложность логики естественным образом соотвествует глубине деревьев решений, а число переходов – числу листьев. Ограничения можно задать следующим способом: ввести функцию штрафа за нарушение ограничений и прибавлять ее к фитнесс-функции. Таким образом, неподходящие под ограничения хромосомы будут от-

сеяны самой эволюцией. Возможно применение и других способов, например повторение генетических операций кроссовера и мутации до получения особи, удовлетворяющей ограничениям.

3.6. Реализация

Предложенный метод генетического программирования для автоматных моделей был реализован в рамках проекта *Autoant* – разработка фреймворка для написания генетических алгоритмов и исследования задачи об умном муравье и ее модификаций, рассмотренных в работе [16]. Данный фреймворк был разработан автором совместно с Ю. Д. Бедным. Фреймворк обладает следующей функциональностью:

- архитектура проекта позволяет проводить исследования задачи о умном муравье и ее модификациях;
- программные интерфейсы просты для использования;
- допускается использование фреймворка из приложений, написанных на языках *C++* и *Java*;
- поведение муравья и соответствующего ему автомата, визуализируются.
- в рамках фреймворка реализованы различные эволюционные алгоритмы для решения исследуемых задач.
- реализован модуль, упрощающий написание генетических алгоритмов.

Фреймворк разработан на языка *Java* и доступен по адресу `http://neerc.ifmo.ru/svn/automata/autoant`.

3.6.1. Общая схема фреймворка

Фреймворк состоит из трех модулей:

- Модуль *autoant-framework*. Модуль позволяет пользователю измерять количество съеденной муравьем еды при различных параметрах запуска.

- Модуль *autoant-ga*. Простой фреймворк для реализации эволюционных алгоритмов. Пользователь имеет возможность переопределять представление автомата и генетические операции.

- Модуль *autoant-vis*. Модуль позволяет визуализировать поведение муравья на поле. Это упрощает понятие логики работы сгенерированного автомата.

3.6.2. Модуль *autoant-framework*

Приведем краткое описание архитектуры модуля *autoant-framework*:

- *Ant* – интерфейс муравья, с помощью которого пользователь задает поведение;

- *Game* – класс, инкапсулирующий игру, который позволяет запускать муравья на конкретной задаче о муравье, возвращая количество съеденной еды;

- *Visible* – перечисление, описывающее видимые муравьем клетки;

- *Info* – интерфейс, используемый муравьем для получения информации о наличии еды в видимых им клетках;

- *Action* – перечисление, описывающее возможные действия муравья;

- *Information* – аннотация, которая может быть добавлена к муравью. Она описывает имя муравья и его разработчика.

Диаграмма классов модуля приведена на рис. 19.

Заметим, что дизайн позволяет использовать, в том числе, и не автоматные стратегии муравья.

3.6.3. Модуль *autoant-ga*

Этот модуль является фреймворком для разработки различных эволюционных алгоритмов. Его важным достоинством является то, что он применим для широкого круга задач, так как не ориентирован на специфику задачи об

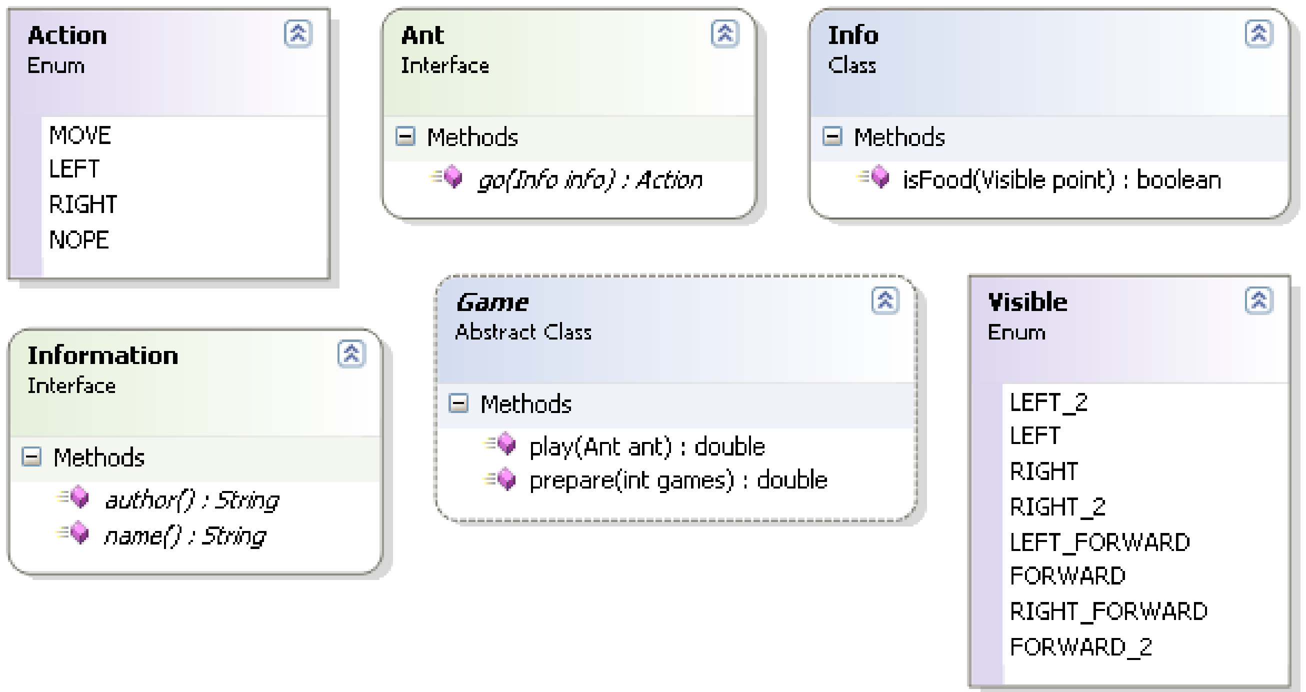

Рис. 19. Диаграмма классов модуля *autoant-framework*

умном муравье и автоматного программирования. На рис. 20 приведена диаграмма классов модуля.

Эволюционный алгоритм задается реализацией двух классов:

- *Configuration* – класс конфигурации эволюционного алгоритма. Параметром является класс, представляющий хромосому, который и будет генерироваться.

- *Genome* – класс, выражающий популяцию особей.

Все классы являются *generic*-классами. Это позволяет варьировать способ представления и тем самым реализовывать различные варианты эволюционных алгоритмов.

Класс *Configuration* предоставляет доступ к реализациям генетических операций. Диаграмма интерфейсов генетических операций приведена на рис. 21.

Приведем краткую характеристику этих интерфейсов:

- *Crossover* – интерфейс кроссовера, принимающего две особи и записывающего результат в третью;

- *Mutation* – интерфейс мутации;

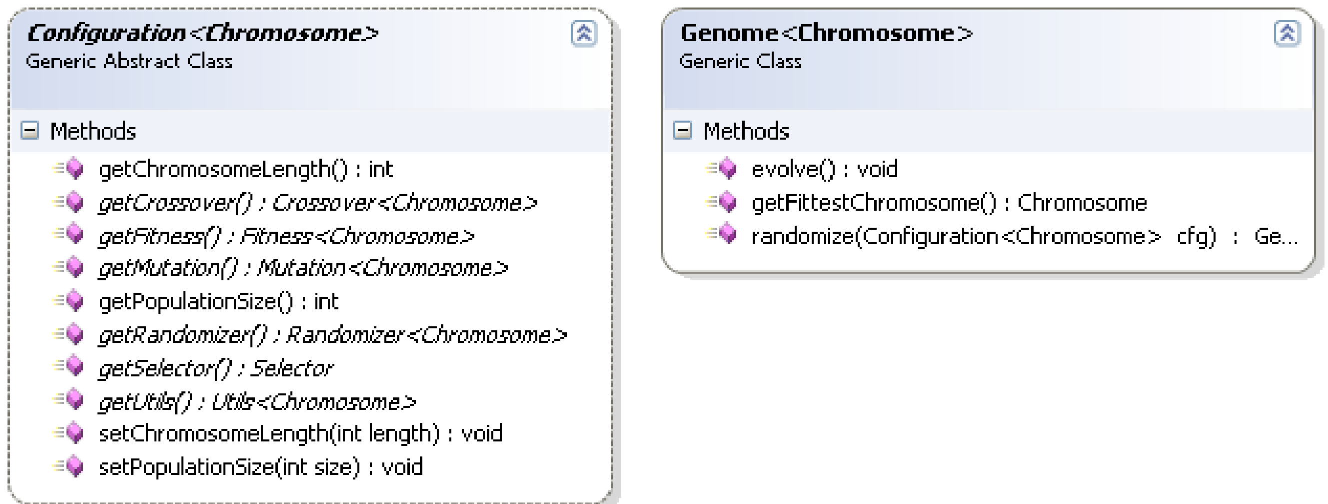

Рис. 20. Диаграмма классов модуля *autoant-ga*

- *Randomizer* – интерфейс создания случайной особи;
- *Selector* – интерфейс отбора, выбирающего особь для размножения по набору значений фитнесс-функции;
- *Utils* – интерфейс утилит, позволяющих копирование особей, их сравнение и создание пустых хромосом.

Для реализации конкретного эволюционного алгоритма необходимо:

1. Реализовать класс-хромосому – представление сущности в виде особи эволюционного алгоритма.
2. Реализовать классы, наследующие интерфейсы генетических операций над классом-хромосомой.
3. Реализовать класс, наследующий *Configuration* и предоставляющий доступ к настройкам и генетическим операциям.

После этого класс конфигурации может быть использован для создания генома особей, с которым может выполняться работа.

В рамках проекта были реализованы следующие эволюционные алгоритмы:

- генетический алгоритм с разделенным алфавитом, описанный в работе [12];

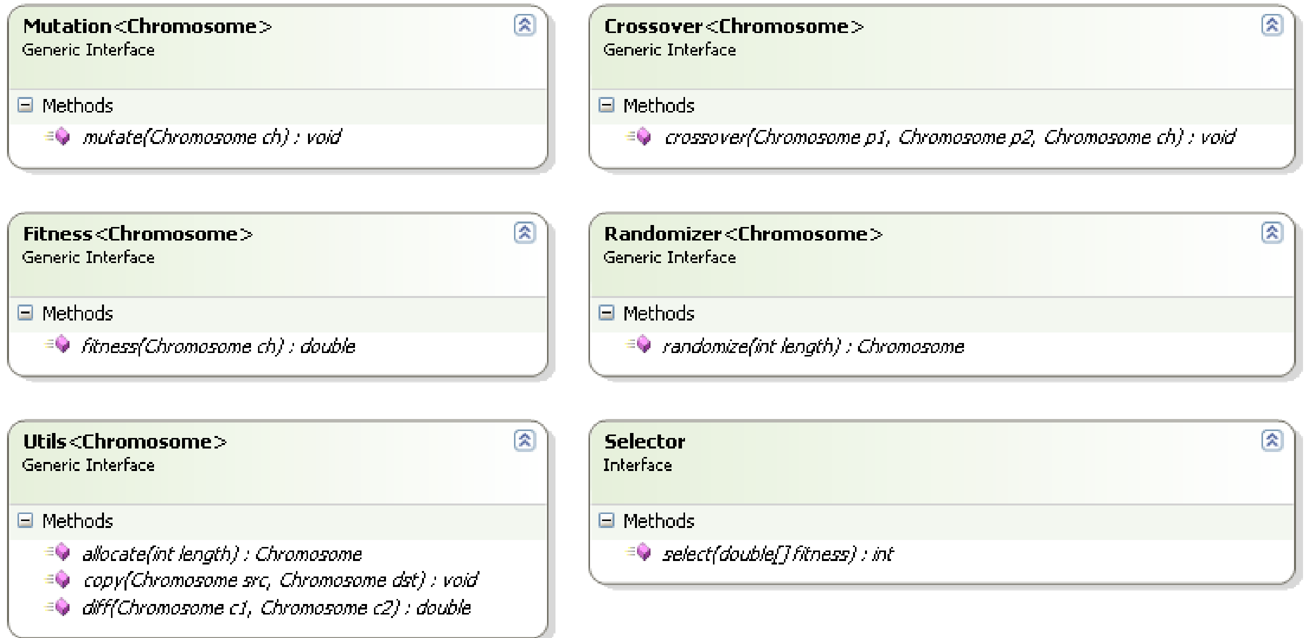

Рис. 21. Диаграмма интерфейсов генетических операций

- генетический алгоритм над битовыми строками, приведенный в работе [13];
- метод генетического программирования, основанный на представлении автоматов с помощью деревьев решений, предложенный в данной работе.

Исходные коды реализации разработанного метода приведены в Приложении. Классом-хромосомой является *TreeAutomata*, классом-конфигурацией, который обеспечивает доступ к интерфейсам генетических операций – *TreeConfiguration*. Класс *Tree* представляет собой вспомогательный класс, реализуюший дерево решений.

3.6.4. Модуль *autoant-vis*

Одной из целей проекта является подробное исследование задачи о муравье и ее модификациях. Однако, выведенные стратегии достаточно сложны для восприятия человеком. Весьма полезным в данном случае оказывается возможность визуального наблюдения за поведением муравья, реализующего порожденную стратегию. Помочь понять логику работы может и визуальное отображения автомата управления муравьем в удобном для человека виде.

Модуль *autoant-vis* имеет следующую функциональность:

- Визуальное отображение поведения муравья. Возможность автоматического проигрывания и осуществления промотки на шаг вперед и назад.
- Визуальное отображение автомата управления муравьем.
- Возможность визуализации динамически и статически созданных муравьев, загрузки классов управления муравьем из *JAR*-файла.

Внешний вид визуализатора после десяти шагов показан на рис. 22. Интерфейс пользователя позволяет выполнять шаги вперед и назад, а также работу в автоматическом режиме. Посередине отображается количество съеденной еды и число сделанных шагов.

Пример графического отображения генетически построенного автомата приведен на рис. 23. Здесь изображен автомат управления в задаче об умном муравье. Каждое из ребер помечено условием перехода(F - если перед муравьем есть еда, N - если нет) и ассоциированным действием(L - поворот налево, R - поворот направо, F - сделать шаг вперед). Для визуализации использована библиотека с открытым кодом *JUNG* – `http://jung.sourceforge.net`.

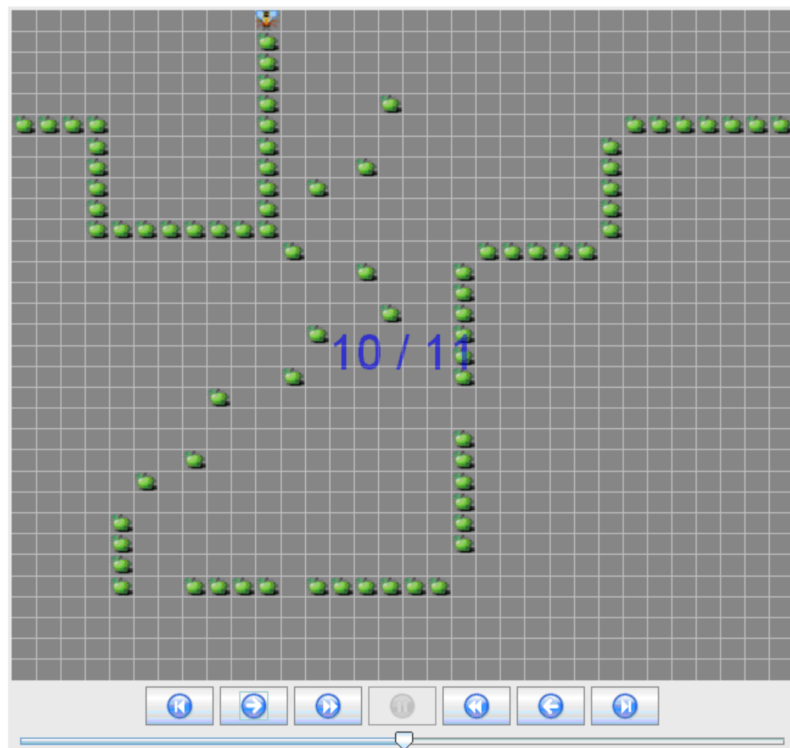

Рис. 22. Внешний вид визуализатора для классической задачи об умном муравье

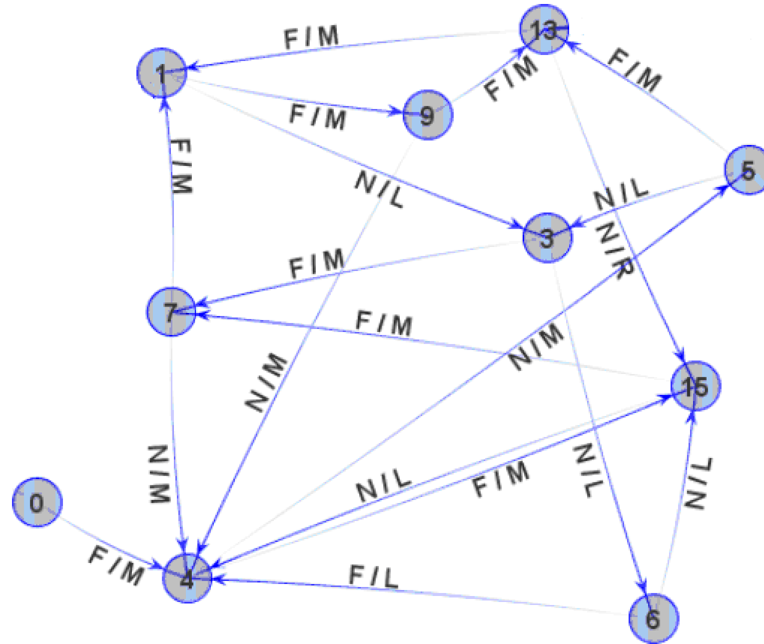

Рис. 23. Визуализация автомата

Выводы по главе 3

1. Разработан новый метод генетического программирования для генерации автоматов.
2. Рассмотрен вопрос задания ограничений на целевой автомат.
3. Описан проект *Autoant*, в рамках которого реализован разработанный метод.

Глава 4.
Сравнение

В этой главе выполняется сравнение разработанного подхода с традиционными генетическими алгоритмами на обобщенной задаче о муравье. Выделяются ситуации, когда предпочтительнее использовать предложенный метод.

4.1. Результаты экспериментов

Предложенный подход сравнивался с генетическими алгоритмами, оперирующими над битовыми строками и генетическими алгоритмами, оперирующими над строками с разделенным алфавитом – таблицей переходов.

Первый эксперимент заключался в сравнении полученных значений фитнесс-функции (объема съеденной еды) за фиксированное число шагов с одинаковыми настройками. Запуск алгоритмов производился со следующими настройками:

- стратегия отбора – элитизм, для размножения отбираются 25% популяции, имеющих наибольшее значение фитнесс-функции;
- частота мутации – 2%;
- размер популяции – 200 особей;
- количество популяций – 100;
- фитнесс-функция – среднее значение съеденной игры на 200 случайных картах, карты внутри одной популяции совпадают, карты различных популяций различны;
- последнее измерение фитнесс-функции осуществлялось на случайном наборе из 2000 карт.

Результаты экспериментов приведены в табл. 2.

Таблица 2. Результаты экспериментов

μ	0.01			
Количество состояний	2	4	8	16
Битовые строки	2.74	2.93	2.83	2.89
Таблица переходов	2.68	3.49	3.74	3.78
Предложенный метод	2.71	2.916	2.88	3.69
μ	0.02			
Количество состояний	2	4	8	16
Битовые строки	7.66	8.38	7.95	6.98
Таблица переходов	6.12	7.32	7.24	7.28
Предложенный метод	7.68	8.04	7.32	8.25
μ	0.03			
Количество состояний	2	4	8	16
Битовые строки	14.46	13.81	13.23	11.93
Таблица переходов	12.48	12.17	11.72	11.15
Предложенный метод	14.14	13.86	13.77	14.18
μ	0.04			
Количество состояний	2	4	8	16
Битовые строки	19.11	18.68	17.47	15.10
Таблица переходов	17.18	15.94	15.03	13.68
Предложенный метод	18.28	20.28	18.60	20.18

Второй эксперимент проверял ход эволюции в известных методах и в предложенном. Для этого осуществлялся запуск при тех же настройках, но на 400 популяций. Сравнить ход эволюции битовых строк и предложенного метода можно на графике (рис. 24).

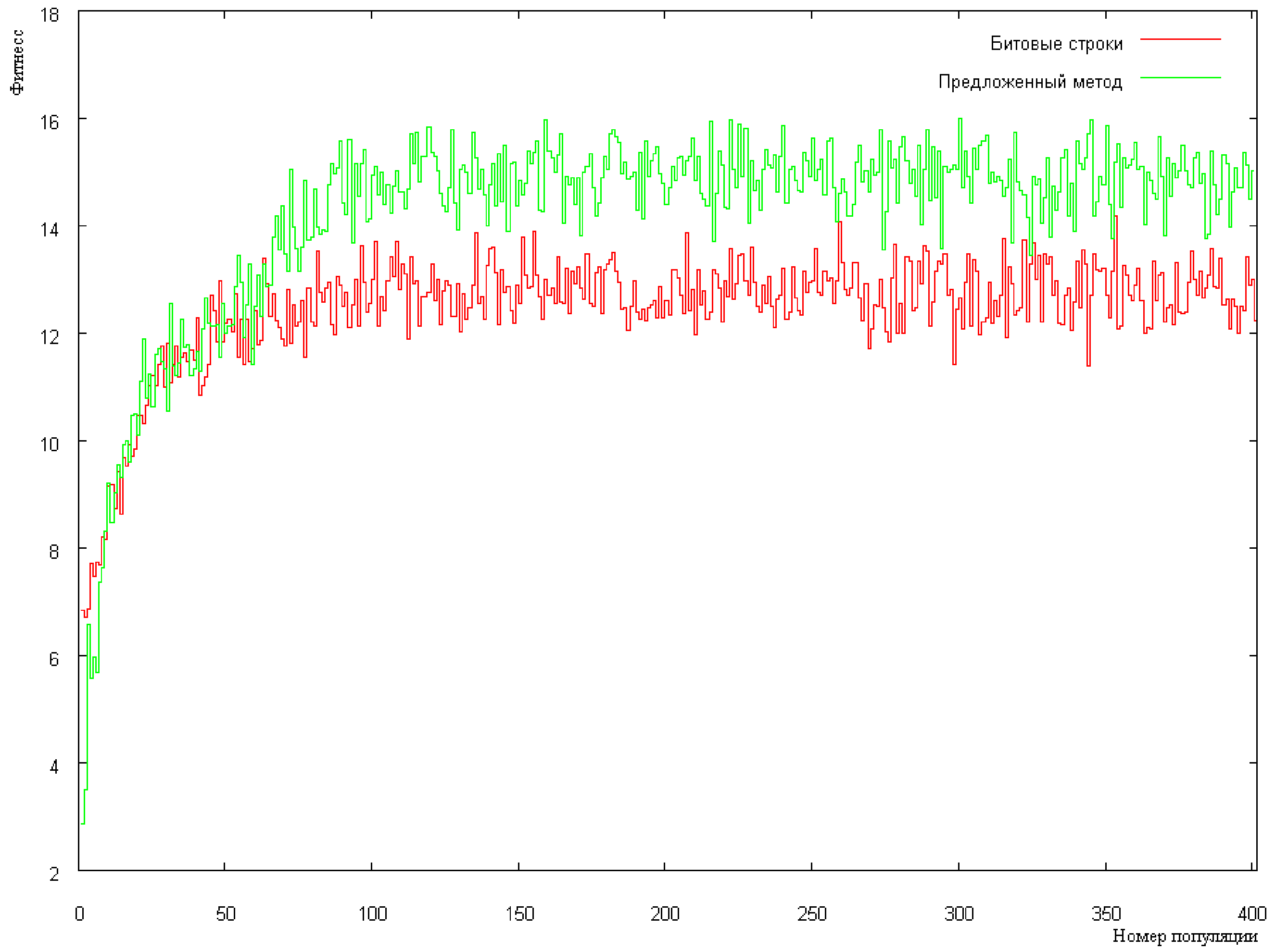

Рис. 24. Ход эволюции при μ=0.03 для автоматов с 16 состояниями

4.2. Анализ

Анализ показывает, что в случаях, когда важны значения почти всех предикатов (малые значения μ), новый метод работает хуже. Это можно обьяснить тем, что искомые автоматы плохо описываются деревьями решений. Однако, при бо́льших значениях μ предложенный метод работает лучше, особенно при большом числе состояний. Таким образом можно сделать вывод – метод работает в большинстве случаев, за исключением ситуаций, когда функция переходов не может быть эффективно выражена деревом решений.

Интересна зависимость высоты дерева от параметров, приведенная в табл. 3.

Таблица 3. Средняя высота деревьев в выведенных автоматах

μ	0.01			
Количество состояний	2	4	8	16
Высота	5.5	3.25	2.375	3.625
μ	0.02			
Количество состояний	2	4	8	16
Высота	5.0	7.0	2.875	2.0
μ	0.03			
Количество состояний	2	4	8	16
Высота	4.5	2.5	1.875	1.5625
μ	0.04			
Количество состояний	2	4	8	16
Высота	4.0	3.0	1.75	2.625

Можно заметить, что средняя высота деревьев падает с увеличением количества состояний. Это можно объяснить тем, что логика переходов из определенного состояния достаточно проста. Учитывая то, что в автоматном программировании состояния играют роль неких характеристик уже совершенных действий, можно сказать, что предложенный метод хорошо подходит для генерации автоматов. Отметим, что метод превосходит генерацию с помощью

битовых строк или таблиц переходов в случае малой высоты, и работает хуже в случае большой высоты деревьев.

Таким образом, установлено, что метод работает эффективнее, чем традиционные генетические алгоритмы, при соблюдении следующих условий:

1) решение задачи имеет естественное представление в виде автомата;

2) функция переходов может быть эффективно выражена в виде дерева решений – количество предикатов, важных в конкретном состоянии невелико по сравнению с суммарным числом предикатов автомата.

Выводы по главе 4

1. Проведено сравнение разработанного метода с известными на примере обобщенной задачи о муравье.

2. Определены условия применимости предложенного подхода.

Заключение

Основные результаты работы состоят в следующем:

1. Предложен метод генетического программирования для генерации автоматов, основанный на представлении автоматов с помощью деревьев решений.

2. Показано, что метод позволяет естественным образом задавать ограничения на целевой автомат, вводя их в фитнесс-функцию.

3. Рассмотрена обобщенная задача об умном муравье. На ней выполнено сравнение предложенного метода с известными.

4. Определены условия большей эффективности разработанного метода по сравнению с традиционными генетическими алгоритмами.

5. Совместно с Ю. Д. Бедным реализован проект *Autoant*, включающий разработку среды для реализации генетических алгоритмов на примере задачи о муравье.

Дальнейшие исследования:

1) апробация метода на практических задачах;

2) разработка эвристик, ускоряющих генерацию автоматов, представленных с помощью деревьев решений;

3) детальное исследование задачи о муравье и объяснение оптимальности полученных стратегий;

4) обобщение метода на задачи, использующие небулевые входные переменные;

5) применение разрешающих диаграмм для представления автоматов.

Литература

1. *Holland J. H.* Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence. The University of Michigan Press, 1975.

2. *Koza J. R.* Genetic programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems). The MIT Press. MA: Cambridge, 1992.

3. *Ferreira C.* Gene Expression Programming: A New Adaptive Algorithm for Solving Problems //Complex Systems. 2001. Vol. 13, issue 2, pp. 87–129. http://www.gene-expression-programming.com/webpapers/GEP.pdf

4. *Шалыто А. А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1

5. *MacKay D. J. C.* Information Theory, Inference, and Learning Algorithms. Cambridge University Press, 2003. http://www.inference.phy.cam.ac.uk/itprnn/book.pdf

6. *Cramer N. C.* A representation for the Adaptive Generation of Simple Sequential Programs / Proceedings of an International Conference on Genetic Algorithms and the Applications. Carnegie Mellon University, 1985. http://www.sover.net/~nichael/nlc-publications/icga85/index.html

7. *Aho A. V., Sethi R., Ullman J.* Compiler Design: Principles, Tools, and Techniques. Reading. MA: Addison Wesley, 1986. http://lib.mexmat.ru/books/2500

8. *Шалыто А. А.* Алгоритмизация и программирование для систем логического управления и „реактивных“систем. Обзор //Автоматика и телемеханика. 2001. №1, с.3–39. http://is.ifmo.ru/download/arew.pdf

9. *Von Neumann J., Burks A. W.* Theory of Self-Reproducing Automata. University of Illinois Press, 1966.

10. *Ullman J. D., Hopcroft J. E.* Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, 1979. http://lib.mexmat.ru/books/9019

11. *Parekh B., Honavar V.* Learning DFA from simple examples. Berlin, 1997. http://www.cs.iastate.edu/~honavar/Papers/parekh-dfa.pdf

12. *Belz A., Eskikaya B. A.* Genetic Algorithm for Finite State Automata Induction with an Application to Phonotactics /Proceedings of the ESSLLI-98 Workshop on Automated Acquisition of Syntax and Parsing, Saarbruecken, 1998. http://www.pcs.usp.br/~lta/artigos/icannga_2005_pistori_martins_castro.pdf

13. *Jefferson D., Collins R., Cooper C., Dyer M., Flowers M., Korf R., Taylor C., Wang A.* The Genesys System. UCLA, 1992. http://www.demo.cs.brandeis.edu/papers/ep93.pdf

14. *Angeline P. J., Pollack J.* Evolutionary Module Acquisition / Proceedings of the Second Annual Conference on Evolutionary Programming. 1993. http://citeseer.ist.psu.edu/angeline93evolutionary.html

15. *Chambers D. L.* Handbook of Genetic Algorithms: Complex Coding Systems(Volume III). CRC, 1999. http://lib.mexmat.ru/books/8894

16. *Бедный Ю. Д., Шалыто А. А.* Применение генетических алгоритмов для построения автоматов в задаче „Умный муравей“. http://is.ifmo.ru/works/_ant.pdf

Приложение 1.
Исходные коды

Tree.java

```
package ru.ifmo.ctddev.autoant.dectree;

import ru.ifmo.ctddev.autoant.engine.Util;

import java.util.ArrayList;
import java.util.Arrays;

/**
 * This class implements a decision tree that expresses a discrete function.
 * This class is a generic with a parameter of function result.
 *
 * @author Vladimir Danilov
 */
public class Tree <Action> {
    /**
     * This internal class implements a node of a decision tree.
     *
     * @author Vladimir Danilov
     */
    private class Node {
        /**
         * Creates a node with a given parent.
         *
         * @param parent - parent node
         */
        public Node(Node parent) {
            this.parent = parent;
            this.action = null;
        }

        /**
         * Creates a deep copy of a node and attaches it to given parent.
         *
         * @param par - parent node
         * @return deep copy of node
         */
        public Node copy(Node par) {
            Node cp = new Node(par);
            cp.predicate = this.predicate;
            cp.action = this.action;
            if (this.left != null) {
                cp.left = this.left.copy(cp);
            }
            if (this.right != null) {
                cp.right = this.right.copy(cp);
            }
            return cp;
        }

        /**
         * Returns list of all nodes in subtree.
         *
         * @return List of all nodes in subtree.
         */
        public ArrayList<Node> subtree() {
```

```
        ArrayList<Node> sub = new ArrayList<Node>();
        sub.add(this);
        if (this.left != null) {
            sub.addAll(this.left.subtree());
        }
        if (this.right != null) {
            sub.addAll(this.right.subtree());
        }
        return sub;
    }

    /**
     * Replaces one of the childs.
     *
     * @param src  Child to replace.
     * @param dest New child.
     */
    public void replace(Node src, Node dest) {
        if (this.left == src) {
            this.left = dest;
        } else if (this.right == src) {
            this.right = dest;
        } else {
            assert false;
        }
        dest.parent = this;
    }

    /**
     * Returns height of subtree, corresponding to the node.
     *
     * @return Height of subtree
     */
    public int height() {
        if (this.predicate == -1) {
            return 1;
        }
        return 1 + Math.max(this.left.height(), this.right.height());
    }

    /**
     * Return number of leafes in subtree.
     *
     * @return Numver of leafes in subree.
     */
    public int leafes() {
        if (this.predicate == -1) {
            return 1;
        }
        return (this.left.leafes() + this.right.leafes());
    }

    /**
     * Cuts off all unreachable nodes.
     * Naive recursive implementation.
     *
     * @param used  Array representing if a predicate was met.
     * @param value Array representing values of met predicates.
     * @return Root of cutted-off subtree.
     */
    public Node cutoff(boolean[] used, boolean[] value) {
        if (this.predicate != -1) {
            if (used[this.predicate]) {
                if (value[this.predicate]) {
                    this.left.parent = null;
                    this.right.parent = this.parent;
                    return this.right.cutoff(used, value);
                } else {
                    this.right.parent = null;
                    this.left.parent = this.parent;
                    return this.left.cutoff(used, value);
                }
            } else {
```

```
                used[this.predicate] = true;
                value[this.predicate] = false;
                this.left = this.left.cutoff(used, value);
                value[this.predicate] = true;
                this.right = this.right.cutoff(used, value);
                used[this.predicate] = false;
                return this;
            }
        } else {
            this.left = null;
            this.right = null;
            return this;
        }
    }

    /**
     * Generates a random subtree.
     *
     * @param used    Array representing if a predicate was met.
     * @param allowed Allowed values of represented function.
     */
    public void generate(boolean[] used, Action[] allowed) {
        ArrayList<Integer> free = new ArrayList<Integer>();
        for (int i = 0; i < used.length; i++) {
            if (!used[i]) {
                free.add(i);
            }
        }
        int pred = Util.rnd(free.size() + 1);
        if (pred != free.size()) {
            this.predicate = free.get(pred);
            this.left = new Node(this);
            this.right = new Node(this);
            used[predicate] = true;
            this.left.generate(used, allowed);
            this.right.generate(used, allowed);
            used[predicate] = false;
        } else {
            this.action = allowed[Util.rnd(allowed.length)];
            this.predicate = -1;
            this.left = null;
            this.right = null;
        }
    }

    /**
     * Left son.
     */
    private Node left;

    /**
     * Right son.
     */
    Node right;

    /**
     * Index of a predicate, the node is marked with.
     */
    int predicate;

    /**
     * Action assigned with a node. Is valid only if a node is leaf.
     */
    Action action;

    /**
     * Parent node.
     */
    Node parent;
}

/**
 * Creates a random tree with given predicate count and given possible values of function.
```

```
 *
 * @param predicates Predicate count.
 * @param allowed    Allowed values of represented function.
 * @return Random tree.
 */
public static <Action> Tree<Action> randomTree(int predicates, Action[] allowed) {
    Tree<Action> tree = new Tree<Action>();
    tree.predicateCount = predicates;
    tree.actions = allowed;
    boolean[] used = new boolean[predicates];
    tree.root.generate(used, allowed);
    Arrays.fill(used, false);
    boolean[] values = new boolean[predicates];
    tree.root = tree.root.cutoff(used, values);
    assert (tree.height() <= predicates + 1);
    return tree;
}

/**
 * Default constructor.
 */
private Tree() {
    this.root = new Node(null);
}

/**
 * Returns function value by predicate values.
 *
 * @param predicateValues Array representings values of predicates.
 * @return Value of represented function with given arguments.
 */
public Action getAction(boolean[] predicateValues) {
    assert predicateValues.length == predicateCount;

    Node node = root;
    while (node.predicate != -1) {
        boolean look = predicateValues[node.predicate];
        node = look ? node.right : node.left;
    }
    return node.action;
}


/**
 * Creates a deep copy of tree.
 *
 * @return Deep copy of tree.
 */
public Tree<Action> copy() {
    Tree<Action> cp = new Tree<Action>();
    cp.root = root.copy(null);
    cp.predicateCount = this.predicateCount;
    cp.actions = this.actions;
    return cp;
}

/**
 * Applies a mutation to the tree.
 * Replaces subtree of random node with a randomly created subtree.
 */
public void mutate() {
    Node node = randomNode();

    boolean[] used = new boolean[predicateCount];
    boolean[] values = new boolean[predicateCount];
    node.generate(used, actions);

    Arrays.fill(used, false);
    this.root = this.root.cutoff(used, values);

    assert height() <= this.predicateCount + 1;
}
```

```
/**
 * Crosses over two trees.
 * Replaces random subtree of one parent with a random subtree from other parent.
 *
 * @param first  One of parents
 * @param second One of parents
 * @return Offspring.
 */
public static <Action> Tree<Action> cross(Tree<Action> first, Tree<Action> second) {
    assert first.predicateCount == second.predicateCount;

    Tree<Action> son = first.copy();
    Tree<Action>.Node toRep = son.randomNode();
    Tree<Action>.Node newNode = second.randomNode().copy(null);
    son.replace(toRep, newNode);

    boolean[] used = new boolean[first.predicateCount];
    boolean[] value = new boolean[first.predicateCount];

    son.root = son.root.cutoff(used, value);

    return son;
}

/**
 * Returns number of leafes in the tree.
 *
 * @return Number of leafes.
 */
public int leafes() {
    return root.leafes();
}

/**
 * Returns height of the tree.
 *
 * @return
 */
public int height() {
    return root.height();
}


/**
 * Return a random node.
 *
 * @return Random node.
 */
private Node randomNode() {
    ArrayList<Node> all = root.subtree();
    return all.get(Util.rnd(all.size()));
}


/**
 * Replaces subtree corresponding to one node with a subtree corresponding to other node.
 *
 * @param src  Node to replace.
 * @param dest New node.
 */
private void replace(Node src, Node dest) {
    assert src != null;
    Node par = src.parent;
    if (par == null) {
        this.root = dest;
    } else {
        par.replace(src, dest);
    }
}

/**
 * Root of the tree.
 */
```

```
    private Node root;

    /**
     * Allowed actions.
     */
    private Action[] actions;

    /**
     * Number of predicates.
     */
    private int predicateCount;
}
```

TreeAutomata.java

```java
package ru.ifmo.ctddev.autoant.dectree;

/**
 * This class implements automaton represented with decision trees.
 * This class is a generic class with parameter of output action class.
 *
 * @author Vladimir Danilov
 */
public class TreeAutomata <Action> {
    /**
     * Internal class representing transition.
     */
    private class Transition {
        /**
         * State the transition leads to.
         */
        int state;

        /**
         * Assigned action on transition.
         */
        Action action;
    }

    /**
     * Creates a new automaton with given number of states, number of predicates and number of
     * allowed output actions.
     *
     * @param states     Number of states.
     * @param predicates Number of input predicates.
     * @param actions    Allowed output actions.
     */
    public TreeAutomata(int states, int predicates, Action[] actions) {
        this.states = states;
        this.predicates = predicates;

        Transition[] all = (Transition[]) (new Object[actions.length * states]);
        for (int i = 0; i < all.length; i++) {
            all[i] = new Transition();
            all[i].action = actions[i / states];
            all[i].state = i % states;
        }

        trees = new Tree[states];
        for (int i = 0; i < states; i++) {
            trees[i] = Tree.randomTree(predicates, all);
        }
    }

    /**
     * Default private constructor.
     */
    private TreeAutomata() {
    }

    /**
     * Makes a transition corresponding to input predicates.
     *
     * @param predicates Values of input predicates.
     * @return Action done by automaton.
     */
    public Action step(boolean[] predicates) {
        Tree<Transition> dectr = trees[currentState];
        Transition dec = dectr.getAction(predicates);
        currentState = dec.state;
        return dec.action;
    }

    /**
```

```
 * Applies a mutation operation to the automaton.
 * A tree corresponding to random state is mutated.
 */
public void mutate() {
    Tree<Transition> tr = trees[((int) (trees.length * Math.random()))];
    tr.mutate();
}

/**
 * Creates a copy of automaton.
 *
 * @return Deep copy of automaton.
 */
public TreeAutomata<Action> copy() {
    TreeAutomata<Action> cp = new TreeAutomata<Action>();
    cp.copyFrom(this);
    return cp;
}

/**
 * Crosses over two automatons and returns offspring.
 *
 * @param first  First parent.
 * @param second Second parent.
 * @return Offsprint.
 */
public static <AutomataAction> TreeAutomata<AutomataAction> cross(
        TreeAutomata<AutomataAction> first,
        TreeAutomata<AutomataAction> second) {
    assert (first.states == second.states);
    assert (first.predicates == second.predicates);

    TreeAutomata<AutomataAction> ant = first.copy();
    for (int i = 0; i < ant.trees.length; i++) {
        ant.trees[i] = Tree.cross(first.trees[i], second.trees[i]);
    }
    return ant;
}

/**
 * Returns an average height of expressing decision trees.
 *
 * @return Average height of decision trees.
 */
public double height() {
    double avg = 0;
    for (Tree<Transition> tree : this.trees) {
        avg += tree.height();
    }
    return avg / this.trees.length;
}

/**
 * Returns an average leafes count in decision trees.
 *
 * @return Average leafes count in decision trees.
 */
public double leafes() {
    double avg = 0;
    for (Tree<Transition> tree : this.trees) {
        avg += tree.leafes();
    }
    return avg / this.trees.length;
}

/**
 * Copies one tree automaton from other.
 *
 * @param other Automaton to copy from.
 */
public void copyFrom(TreeAutomata<Action> other) {
    this.states = other.states;
    this.predicates = other.predicates;
```

```
        this.trees = new Tree[other.states];
        for (int i = 0; i < this.trees.length; i++) {
            this.trees[i] = other.trees[i].copy();
        }
    }

    /**
     * State count.
     */
    private int states;

    /**
     * Predicate count.
     */
    private int predicates;

    /**
     * Current state of automaton.
     */
    private int currentState;

    /**
     * Expressing decision trees.
     */
    Tree<Transition>[] trees;
}
```

TreeConfiguration.java

```java
package ru.ifmo.ctddev.autoant.dectree;

import ru.ifmo.ctddev.autoant.fx.*;

/**
 * Configuration of evolutionary algorithm over automatons, represented by decision trees.
 * Wraps genetic operations over decision trees automatons.
 * This class is generic class over automaton output action.
 *
 * @author Vladimir Danilov
 */
public class TreeConfiguration <Action> extends Configuration<TreeAutomata<Action>> {
    /**
     * Creates a configuration for automatons evolving.
     *
     * @param predicates Number of predicates of automaton.
     * @param actions Allowed output actions.
     */
    public TreeConfiguration(int predicates, Action[] actions) {
        this.predicates = predicates;
        this.actions = actions;
    }


    /**
     * Creates selector. We use a roulette selector in this implementation.
     *
     * @return Genetic operator of selector.
     */
    protected Selector createSelector() {
        return new RouletteSelector();
    }

    /**
     * Creates a genetic operator of mutation.
     *
     * @return Genetic operator of mutation.
     */
    protected Mutation<TreeAutomata<Action>> createMutation() {
        return new Mutation<TreeAutomata<Action>>() {
            public void mutate(TreeAutomata<Action> c) {
                if (Math.random() < MUTATION_RATE) {
                    c.mutate();
                }
            }
        };
    }

    /**
     * Creates a genetic operator of crossover.
     *
     * @return Genetic operator of crossover.
     */
    protected Crossover<TreeAutomata<Action>> createCrossover() {
        return new Crossover<TreeAutomata<Action>>() {
            public void crossover(TreeAutomata<Action> p1, TreeAutomata<Action> p2, TreeAutomata<
                Action> c) {
                TreeAutomata<Action> ant = TreeAutomata.cross(p1, p2);
                c.copyFrom(ant);
            }
        };
    }

    /**
     * Creates a genetic operator of random creation.
     *
     * @return Genetic operator of random creation.
     */
    protected Randomizer<TreeAutomata<Action>> createRandomizer() {
```

```
        return new Randomizer<TreeAutomata<Action>>() {
            public TreeAutomata<Action> randomize(int length) {
                return new TreeAutomata<Action>(length, predicates, actions);
            }
        };
    }

    /**
     * Creates utils for statictics and control of evolution.
     *
     * @return Utils.
     */
    protected Utils<TreeAutomata<Action>> createUtils() {
        return new Utils<TreeAutomata<Action>>() {
            public void copy(TreeAutomata<Action> src, TreeAutomata<Action> dst) {
                dst.copyFrom(src);
            }

            public double diff(TreeAutomata<Action> c1, TreeAutomata<Action> c2) {
                return 0;
            }

            public TreeAutomata<Action> allocate(int length) {
                return new TreeAutomata<Action>(length, predicates, actions);
            }
        };
    }

    /**
     * Mutation rate.
     */
    private static final double MUTATION_RATE = 0.02;

    /**
     * Automatons' predicates count.
     */
    private int predicates;

    /**
     * Allowed automatons' output actions.
     */
    private Action[] actions;
}
```