МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
(МИНОБРНАУКИ)

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ «САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»
(СПБГУ ИТМО)

УДК 004.4’242
№ госрегистрации 0120.0 710295
Инв. № 370095.4

УТВЕРЖДАЮ
Ректор СПбГУ ИТМО,
докт. техн. наук, профессор
В. Н. Васильев

«____» ___________ 2008 г.

Подп. и дата	
Инв. № дубл.	
Взам. инв. №	
Подп. и дата	
Инв. № подл.	

ТЕХНОЛОГИЯ ГЕНЕТИЧЕСКОГО ПРОГРАММИРОВАНИЯ ДЛЯ ГЕНЕРАЦИИ АВТОМАТОВ УПРАВЛЕНИЯ СИСТЕМАМИ СО СЛОЖНЫМ ПОВЕДЕНИЕМ

ЗАКЛЮЧИТЕЛЬНЫЙ ОТЧЕТ ПО IV ЭТАПУ «ОБОБЩЕНИЕ И ОЦЕНКА РЕЗУЛЬТАТОВ ИССЛЕДОВАНИЙ»

ЛИСТОВ 100

Декан факультета «Информационные
технологии и программирование»
докт. техн. наук, профессор
___________________ В. Г. Парфенов

Руководитель темы
заведующий кафедрой «Технологии программирования»,
докт. техн. наук, профессор
___________________ А. А. Шалыто

Ответственный исполнитель
магистрант кафедры «Компьютерные технологии»
___________________ Ф. Н. Царев

Санкт-Петербург
2008

СПИСОК ИСПОЛНИТЕЛЕЙ

Руководитель темы докт. техн. наук, профессор	А. А. Шалыто	Отчет в целом
Ответственный исполнитель магистрант	Ф. Н. Царев	Отчет в целом
Нормоконтролер	Г. Р. Туктарова	Отчет в целом
Ведущий научный сотрудник, докт. техн. наук, профессор	В. В. Антипов	Глава 1.
Ведущий научный сотрудник, канд. техн. наук	В. В. Киселев	Глава 2.
Ведущий научный сотрудник, канд. техн. наук	Р. Н. Котляр	Глава 1.
Ведущий научный сотрудник, канд. техн. наук	Ю. П. Московцев	Глава 2.
Ведущий научный сотрудник, канд. техн. наук, доцент	В. А. Третьяков	Глава 1.
Ведущий научный сотрудник, канд. техн. наук	Г. М. Файкин	Глава 2.
Доцент, канд. техн. наук	Г. А. Корнеев	Глава 2.
Канд. техн. наук	Е. М. Кузнецова	Глава 2.
Канд. техн. наук	А. С. Пресняк	Глава 1.
Доцент, канд. техн. наук	Э. А. Опалева	Глава 1.
Руководитель временного трудового коллектива (ВТК), ассистент кафедры КТ	А. П. Мельничук	Разделы 1.1 и 1.3.
Член ВТК, профессор кафедры ИС	Е. Ю. Михайлова	Разделы 1.1 и 1.3.
Член ВТК, доцент кафедры КТ	В. Д. Наумчик	Разделы 1.1 и 1.3.
Член ВТК, доцент кафедры КТ	М. Ю. Осипов	Раздел 1.1.
Член ВТК, доцент кафедры КТ	А. Н. Воробьев	Раздел 1.3.
Член ВТК, доцент кафедры КТ	Ю. А. Щупак	Раздел 1.3.
Член ВТК, доцент кафедры КТ	С. В. Чириков	Раздел 1.1.
Член ВТК, доцент кафедры КТ	А. С. Сегаль	Раздел 1.1.
Член ВТК, доцент кафедры КТ	Д. Г. Шопырин	Раздел 1.3.
Член ВТК, доцент кафедры ИС	Д. А. Зубок	Раздел 1.1.
Член ВТК, ассистент кафедры ИС	В. В. Повышев	Раздел 1.3.
Член ВТК, ассистент кафедры ИС	В. В. Ильин	Раздел 1.1.
Член ВТК, ассистент кафедры ИС	М. Г. Холин	Раздел 1.3.
Аспирант	П. Г. Лобанов	Разделы 1.1 и 1.3.
Магистрант	Н. И. Поликарпова	Раздел 1.1.
Магистрант	В. Н. Точилин	Раздел 1.1.
Магистрант	Ю. Д. Бедный	Раздел 1.1.
Магистрант	В. Р. Данилов	Разделы 1.2. и 2.2.
Магистрант	Е. А. Мандриков	Разделы 1.3 и 2.1.
Магистрант	В. А. Кулев	Разделы 1.3 и 2.1.
Студент	А. А. Давыдов	Разделы 1.2 и 2.1.
Студент	Д. О. Соколов	Разделы 1.2 и 2.1.
Студент	В. О. Клебан	Раздел 1.3.
Студент	С. О. Попов	Раздел 2.1.

РЕФЕРАТ

Отчет 100 с., 2 гл., 75 рис., 7 табл., 34 источника.

ОБОБЩЕНИЕ И ОЦЕНКА РЕЗУЛЬТАТОВ ИССЛЕДОВАНИЙ

Объектом исследования являются методы генерации автоматов управления системами со сложным поведением.

Цель этапа – оценка и обобщение результатов исследований, проведенных в рамках работ по настоящему контракту.

В результате работ предложен набор методов генетического программирования, предназначенных для построения автоматов управления системами со сложным поведением. На основе этих методов разработаны прототипы программных средств для построения конечных автоматов, управляющих системами со сложным поведением. Проведено экспериментальное исследование этих программных средств на задаче «Умный муравей-3» и даны методические рекомендации по применению разработанных методов для построения конечных автоматов, управляющих системами со сложным поведением.

В первой главе описываются методы представления автоматов в виде хромосом генетического алгоритма и прототипы программных средств: *GAAP* (для поддержки метода сокращенных таблиц переходов), *AutoAnt* (для поддержки метода представления автоматов деревьями решений), *3Genetic* (для поддержки метода совместного применения конечных автоматов и нейронных сетей). Даны методические рекомендации по применению этих методов для генерации автоматов управления системами со сложным поведением.

Во второй главе приводятся методические рекомендации и указания по разработке систем со сложным поведением на основе разработанных методов.

Таким образом, были получены решения всех задач, поставленных в техническом задании на проведение четвертого этапа работы.

ОГЛАВЛЕНИЕ

СПИСОК ИСПОЛНИТЕЛЕЙ 2
РЕФЕРАТ 3
ОГЛАВЛЕНИЕ 4
ВВЕДЕНИЕ 6
1. ПРЕДЛОЖЕНИЯ И РЕКОМЕНДАЦИИ ПО ИСПОЛЬЗОВАНИЮ РЕЗУЛЬТАТОВ НИР ПРИ ПРОЕКТИРОВАНИИ СИСТЕМ УПРАВЛЕНИЯ ОБЪЕКТАМИ СО СЛОЖНЫМ ПОВЕДЕНИЕМ 8
1.1. Методы представления автоматов в виде хромосом генетического алгоритма 8
1.1.1. Метод сокращенных таблиц переходов 8
1.1.1.1. Постановка задачи 8
1.1.1.2. Представление состояний: полные таблицы 9
1.1.1.3. Представление состояний: сокращенные таблицы 13
1.1.1.4. Мутация, зависящая от пригодности 18
1.1.1.5. Формирование нового поколения 20
1.1.1.6. Выбор особей для скрещивания 20
1.1.2. Метод представления автоматов деревьями решений 21
1.1.2.1. Представление автомата деревьями решений 21
1.1.2.2. Генерация случайной особи 22
1.1.2.3. Алгоритм обрезки недостижимых ветвей дерева 22
1.1.2.4. Оператор мутации 22
1.1.2.5. Оператор скрещивания 24
1.1.3. Метод совместного использования конечных автоматов и нейронных сетей 25
1.1.3.1. Некоторые проблемы, возникающие при использовании генетического программирования для построения конечных автоматов 25
1.1.3.2. Искусственные нейронные сети 27
1.1.3.3. Элементы нейронных сетей 27
1.1.3.4. Структура нейронных сетей 28
1.1.3.5. Применение нейронных сетей 28
1.1.3.6. Управление системой со сложным поведением 29
1.1.3.7. Алгоритм генетического программирования для построения конечного автомата и нейронной сети для управления системой со сложным поведением 30
1.1.3.8. Структура особи в используемом алгоритме 30
1.1.3.9. Создание начального поколения 31
1.1.3.10. Операция мутации 31
1.1.3.11. Операция скрещивания 32
1.1.3.12. Формирование следующего поколения 35
1.2. Прототипы программных средств 35
1.2.1. Программное средство *GAAP* 35
1.2.2. Программное средство *AutoAnt* 39
1.2.2.1. Использование программных интерфейсов *AutoAnt* 41
1.2.3. Программное средство *3Genetic* 43
1.2.3.1. Ядро программного средства *3Genetic* 44
1.2.3.2. Работа с модулями 44

1.2.3.3. Модуль, содержащий представление особи 44
1.2.3.4. Модуль, содержащий генетический алгоритм 45
1.2.3.5. Модуль, содержащий визуализатор особи 46
1.2.3.6. Модуль, содержащий визуализируемую функцию 46
1.3. Рекомендации по применению генетических алгоритмов для построения автоматов управления системами со сложным поведением 47
1.4. Выводы 49
2. МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ И УКАЗАНИЯ ПО РАЗРАБОТКЕ СИСТЕМ СО СЛОЖНЫМ ПОВЕДЕНИЕМ С ИСПОЛЬЗОВАНИЕМ РАЗРАБОТАННЫХ МЕТОДОВ 50
2.1. Установка и настройка программных средств 50
2.1.1. Установка и настройка программного средства *GAAP* 50
2.1.1.1. Компиляция программного средства 50
2.1.1.2. Добавление представления особи 50
2.1.2. Установка и настройка программного средства *AutoAnt* 55
2.1.2.1. Описание дистрибутива программного средства 55
2.1.2.2. Установка программного средства 55
2.1.2.3. Настройка программного средства 55
2.1.3. Установка и настройка программного средства *3Genetic* 57
2.1.3.1. Описание дистрибутива программного средства *3Genetic* 57
2.1.3.2. Настройка программного средства 3Genetic 58
2.1.3.3. Настройка генетических алгоритмов, входящих в состав программного средства 3Genetic 65
2.1.3.4. Особи 66
2.1.3.5. Визуализаторы 68
2.2. Применение программных средств 69
2.2.1. Задача «Умный муравей-3» 69
2.2.2. Пример применения программного средства *GAAP* 72
2.2.2.1. Создание файла настроек 72
2.2.2.2. Вычисление функции приспособленности 74
2.2.2.3. Результаты эксперимента 76
2.2.3. Пример применения программного средства *AutoAnt* 80
2.2.3.1. Настройка программного средства для решения задачи 80
2.2.3.2. Результаты эксперимента 80
2.2.4. Пример применения программного средство *3Genetic* 84
2.2.4.1. Создание класса, реализующего представление особи 84
2.2.4.2. Написание файлов настройки и реализация дополнительных классов 85
2.2.4.3. Создание фабрики особей и загрузчика 87
2.2.4.4. Сборка *JAR*–архива и его подключение к ядру программного средства 88
2.2.4.5. Настройка параметров особи `Net automaton` 89
2.2.4.6. Результаты вычислительных экспериментов 89
2.3. Выводы 96
ЗАКЛЮЧЕНИЕ 97
СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 98

ВВЕДЕНИЕ

Технология генетического программирования для генерации автоматов управления системами со сложным поведением разрабатывается в рамках проведения научно-исследовательской работы по *лоту «Разработки в области языков программирования и моделирования программного обеспечения, технологий и инструментальных средств проектирования программ» шифр «2007-4-1.4-18-01-033» по теме «Технология генетического программирования для генерации автоматов управления системами со сложным поведением»*, выполняемой в рамках Федеральной целевой научно-технической программы «Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007−2002 годы» по *государственному контракту № 02.514.11.4044 от 18.05.2007*, заключенному между Федеральным агентством по науке и инновациям и Государственным образовательным учреждением высшего профессионального образования «Санкт-Петербургский государственный университет информационных технологий, механики и оптики» на основании решения Конкурсной комиссии Роснауки № 14 (протокол от 28.04.2007 г. № 14).

Целями настоящего этапа работы является обобщение и оценка результатов исследований. Задачами этапа являются:

1. Разработка предложений и рекомендаций по использованию результатов НИР при проектировании систем управления объектами со сложным поведением.
2. Разработка методических рекомендаций и указаний по разработке систем со сложным поведением с использованием разработанных программных средств.

В настоящее время работы по построению автоматов на основе генетического программирования проводятся в ряде зарубежных университетов, в том числе в Массачусетском технологическом институте и Университете Южной Калифорнии. Знакомство с результатами этих работ показало, что в них строятся автоматы, которые не могут быть использованы в системах со сложным поведением.

Обычно генетические алгоритмы в рамках эволюционного моделирования используются для настройки нейронных сетей. Однако если настроенная нейронная сеть функционирует в рассматриваемой среде недостаточно эффективно, то вручную ее перенастроить невозможно. Поэтому приходится вновь и вновь настраивать ее при помощи генетических алгоритмов. При этом человеку весьма трудно направить процесс в требуемом направлении, а также при необходимости понять полученный результат.

При применении генетических алгоритмов для настройки автоматов ситуация принципиально изменяется, так как построив с помощью генетических алгоритмов автоматы, их в дальнейшем обычно удается модифицировать вручную.

Поиск приемлемого по выбранным критериям управляющего автомата перебором практически невозможен из-за огромного размера пространства, в котором осуществляется поиск. Например, в такой простой задаче как задача об «Умном муравье» [1], число возможных автоматов с семью состояниями около $3{,}2 \times 10^{18}$.

Применение генетического программирования позволяет сделать перебор направленным, однако и в этом случае трудоемкость построения автоматов с требуемыми свойствами остается большой.

Указанная проблема решается за счет учета специфики автоматов, применяемых в системах управления, состоящей в том, что состояния декомпозируют входные воздействия на группы, в каждую из которых обычно входит небольшое число переменных. Это позволяет строить хромосомы только для подмножества всех автоматов, что существенно сокращает пространство возможных решений, и как следствие, время поиска. Эти идеи развиты в настоящем отчете.

Метрологическое обеспечение НИР не требуется.

Дополнительные патентные исследования, проведенные в рамках четвертого этапа работы (отчет о патентных исследованиях № 2008.09.30-1 входит в состав отчетной документации по этапу), позволяют утверждать, что в настоящее время отсутствуют патенты и иные охранные документы, которые могут препятствовать применению в Российской Федерации результатов научных исследований, проводимых по контракту.

Изложенное позволяет утверждать, что результаты выполнения научно-исследовательской работы будут соответствовать мировому уровню разработок в рассматриваемой области.

В рамках работ было подготовлено три промежуточных отчета:

- отчет по I этапу «Выбор направления исследований и базовых методов», инвентарный № 370095.1;
- отчет по II этапу «Теоретические исследования поставленных перед НИР задач», инвентарный № 370095.2;
- отчет по III этапу «Экспериментальные исследования поставленных перед НИР задач», инвентарный № 370095.3.

1. ПРЕДЛОЖЕНИЯ И РЕКОМЕНДАЦИИ ПО ИСПОЛЬЗОВАНИЮ РЕЗУЛЬТАТОВ НИР ПРИ ПРОЕКТИРОВАНИИ СИСТЕМ УПРАВЛЕНИЯ ОБЪЕКТАМИ СО СЛОЖНЫМ ПОВЕДЕНИЕМ

В настоящей главе приводятся предложения и рекомендации по использованию результатов НИР при проектировании систем управления объектами со сложным поведением.

1.1. Методы представления автоматов в виде хромосом генетического алгоритма

В настоящем разделе описаны методы представления автоматов в виде хромосом генетического алгоритма [8, 26].

1.1.1. Метод сокращенных таблиц переходов

При разработке метода сокращенных таблиц наибольшее внимание было уделено специфике управляющих автоматов – возможности построения автоматов со сложными пометками на переходах. Кроме того, для сокращения пространства поиска метод использует концепцию автоматизированного объекта управления [34]: логика сложного поведения описывается автоматом на высоком уровне абстракции, а объект управления задается в качестве входного данного и оптимизации не подвергается. Объект управления может быть произвольным (и, вообще говоря, сколь угодно сложным). Таким образом, предлагаемый метод решает задачу об использовании сложных структур данных в рамках генетического программирования, поставленную основателем генетического программирования J. Koza в работе [17].

В разд. 1.1.1.2 рассматривается упрощенная (наивная) версия метода, соответствующая по своим характеристикам большинству аналогов. Далее показаны недостатки наивного подхода, и разд. 1.1.1.3 приведено описание усовершенствованной версии метода, лишенной этих недостатков.

1.1.1.1. Постановка задачи

Сформулируем *задачу построения управляющего автомата*. Пусть задан объект управления $O = \langle V, v_0, X, Z \rangle$, где V – множество *вычислительных состояний* (или *значений*), v_0 – *начальное значение*, $X = \{x_i : V \to \{0,1\}\}_{i=1}^{n}$ – множество предикатов, $Z = \{z_i : V \to V\}_{i=1}^{m}$ – множество действий. Также задана оценочная функция $\varphi : V \to \mathbf{R}^+$ и натуральное число k.

Объект O может управляться автоматом вида $A = \langle S, s_0, \Delta \rangle$, где S – конечное множество *управляющих* состояний, s_0 – *стартовое состояние*, $\Delta : S \times \{0,1\}^n \to S \times Z^*$ – *управляющая функция*. Управляющую функцию можно разложить на две компоненты: функцию выходов $\zeta : S \times \{0,1\}^n \to Z^*$ и функцию переходов $\delta : S \times \{0,1\}^n \to S$.

Пусть перед началом работы объекту управления соответствует начальное значение v_0, а автомат находится в стартовом состоянии s_0. Назовем *шагом работы* автоматизированного объекта следующую последовательность операций.

1. Объект управления вызывает все предикаты из множества X и формирует из их значений вектор *входного воздействия* $in \in \{0,1\}^n$.
2. Автомат вычисляет значение вектора *выходного воздействия* $out = \zeta(s, in)$, где s – текущее состояние автомата, и переходит в новое управляющее состояние $s_{new} = \delta(s, in)$.

3. Объект управления по очереди вызывает действия $z \in out$, тем самым, изменяя текущее *вычислительное* состояние.

Задача построения управляющего автомата состоит в том, чтобы найти автомат заданного вида такой, что за k шагов работы под управлением этого автомата объект O перейдет в вычислительное состояние с максимальным значением функции его пригодности ($\varphi(v) \to \max$).

В связи с использованием генетического программирования для этой задачи возникают следующие подзадачи: выбор представления конечного автомата в виде особи; адаптация генетических операторов (мутации и скрещивания) для выбранного представления; настройка параметров генетической оптимизации.

В классической интерпретации генетического алгоритма особь представляется в виде набора хромосом. Управляющий автомат легко представить как набор состояний, в каждом из которых его поведение определяется сужением управляющей функции $\Delta_s : \{0,1\}^n \to S \times Z^*$, $s \in S$. Таким образом, удобно сопоставить каждому состоянию хромосому. Таким образом, от задачи представления автомата в виде особи перейдем к более конкретной постановке проблемы: представлению управляющего состояния автомата в виде хромосомы.

1.1.1.2. Представление состояний: полные таблицы

Естественный способ записи хромосомы состояния – это табличное представление функции Δ_s. Таблица содержит 2^n строк (по одной для каждой возможной комбинации значений n предикатов) и $m+1$ столбцов, в первом из которых записано значение функции переходов (номер целевого состояния), а совокупность остальных столбцов обозначает множество действий m, которые необходимо выполнить на переходе. Пример полной таблицы для одного состояния приведен на рис. 1 (контуром обведена информативная часть таблицы, именно она и заносится в хромосому).

x_0	x_1	s	z_0	z_1	z_2
0	0	0	0	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	1	2	1	1	1

Рис. 1. Хромосома состояния: полная таблица (n = 2, m = 3, $|S|$ = 3)

Отметим, что в каждой строке таблицы записано множество действий, а не их последовательность. Задание значения функции действий в виде множества значительно упрощает оператор скрещивания и повышает эффективность процесса эволюции. Выполнение на переходе последовательности действий эквивалентно осуществлению нескольких переходов, на которых выполняются множества действий. Таким образом, автомат в исходной модели всегда может быть записан в рассмотренной выше табличной форме, возможно с добавлением нескольких состояний и переходов. После получения результата оптимизации лишние элементы автомата можно устранить, преобразовав множества действий в последовательности.

Все таблицы, соответствующие состояниям одного автомата, имеют одинаковую размерность, так как число предикатов и действий объекта управления задано постановкой задачи. Что касается управляющих состояний, то их число также должно быть известно: эта информация используется при случайной генерации значений функции переходов. При этом был реализован

подход, при котором число управляющих состояний задается перед началом оптимизации и далее не изменяется. При этом число состояний можно задавать исходя из априорных представлений о сложности задачи, причем с некоторым «запасом»: в процессе оптимизации лишние состояния станут недостижимыми, и их можно будет автоматически исключить. Однако неоправданно большое число состояний негативно влияет на скорость эволюции. В этом смысле более эффективным является постепенное наращивание числа управляющих состояний в процессе оптимизации. Этот вариант также несложно реализуется в рамках предлагаемого подхода к представлению конечных автоматов в виде особей.

Опишем теперь генетические операторы над хромосомами состояний, записанными предложенным выше способом (в виде полных таблиц).

Алгоритм 1. Мутация полных таблиц. Алгоритм мутации состояния, представленного полной таблицей, описан на псевдокоде в листинге 1.

Листинг 1. Мутация полных таблиц

```
State Mutate(State state)
{ State mutant = state;
  for (для всех i: строк таблицы)
  {
    if (с вероятностью p1) {
      mutant[i].targetState = случайное число от 0 до
        nStates - 1;
    }
    if (с вероятностью p2) {
      int nActsPresent = число единиц в mutant[i].output;
      if ((nActsPresent == 0) || (nActsPresent == nActions))
        {
        Index j = случайное число от 0 до nActions - 1;
        mutant[i].output[j] = !mutant[i].output[j];
      } else {
        for (для всех j: номеров действий) {
          mutant[i].output[j] = 1 с вероятностью
                nActsPresent/nActions и 0 иначе;
        }
      }
    }
  }
  return mutant;
}
```

Этот алгоритм проиллюстрирован на рис. 2 (здесь и далее на стрелках, обозначающих изменения значений в ячейках таблицы, написаны вероятности этих изменений). При мутации состояния с некоторой вероятностью может мутировать каждый элемент таблицы. При этом номер целевого состояния изменяется на любой из допустимых, а вместо исходного набора действий генерируется новый набор, вероятность появления единиц в котором равна доле единиц в исходном наборе.

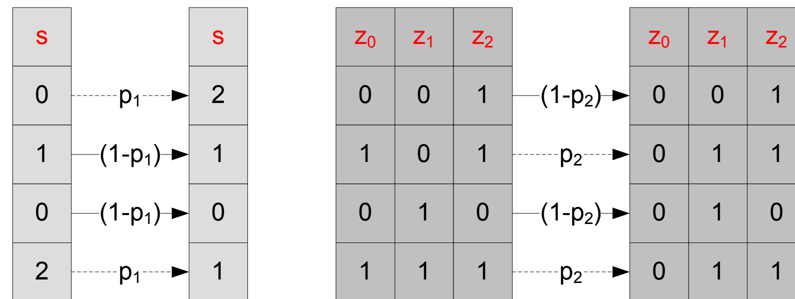


Рис. 2. Пример мутации полных таблиц

Алгоритм 2. Скрещивание полных таблиц. В настоящей работе рассматривается адаптация к предложенному представлению состояний одного способа скрещивания, известного как *одноточечное*. Руководствуясь схожими идеями, нетрудно адаптировать к табличному представлению и другие способы скрещивания.

Алгоритм одноточечного скрещивания полных таблиц представлен в листинге 2.

Листинг 2. Скрещивание полных таблиц

```
pair<State, State> Cross(State state1, State state2)
{
  State child1 = state1;
  State child2 = child1;
  int tableSize = размер таблицы;

  for (для всех j: столбцов таблицы)
  {
    int crossPoint = случайное число от 0 до tableSize;
    for (для всех i: строк таблицы от 0 до crossPoint - 1) {
      child1[i][j] = state1[i][j];
      child2[i][j] = state2[i][j];
    }
    for (для всех i: строк таблицы от crossPoint до tableSize
        - 1) {
      child1[i][j] = state2[i][j];
```

```
                    child2[i][j] = state1[i][j];
                }
            }
            return make_pair(child1, child2);
        }
```

Этот алгоритм проиллюстрирован на рис. 3. Специфика данного алгоритма обусловлена тем, что таблицы состояний двумерны, в отличие от традиционного одномерного представления хромосом в виде битовых строк. При скрещивании полных таблиц предлагается по очереди выполнять традиционное одноточечное скрещивание соответствующих столбцов этих таблиц.

Основная проблема, возникающая при использовании полных таблиц рассмотренного вида – это экспоненциальный рост размерности хромосомы с увеличением числа предикатов объекта управления (напомним, что число строк в таблице 2^n, где n – число предикатов). Опыт показывает, что в реальных задачах управляющие автоматы, построенные вручную, имеют гораздо меньше переходов, чем $|S| \cdot 2^n$. Причина этого состоит в том, что в большинстве задач предикаты имеют «локальную природу» по отношению к управляющим состояниям. В каждом состоянии *значимым* является лишь определенный, небольшой поднабор предикатов, остальные же не влияют на значение управляющей функции. Именно это свойство позволяет существенно сократить размер описания состояний. Кроме того, использование этого свойства в процессе оптимизации позволяет получить результат, более похожий на автомат, построенный вручную, а следовательно, и более понятный человеку.

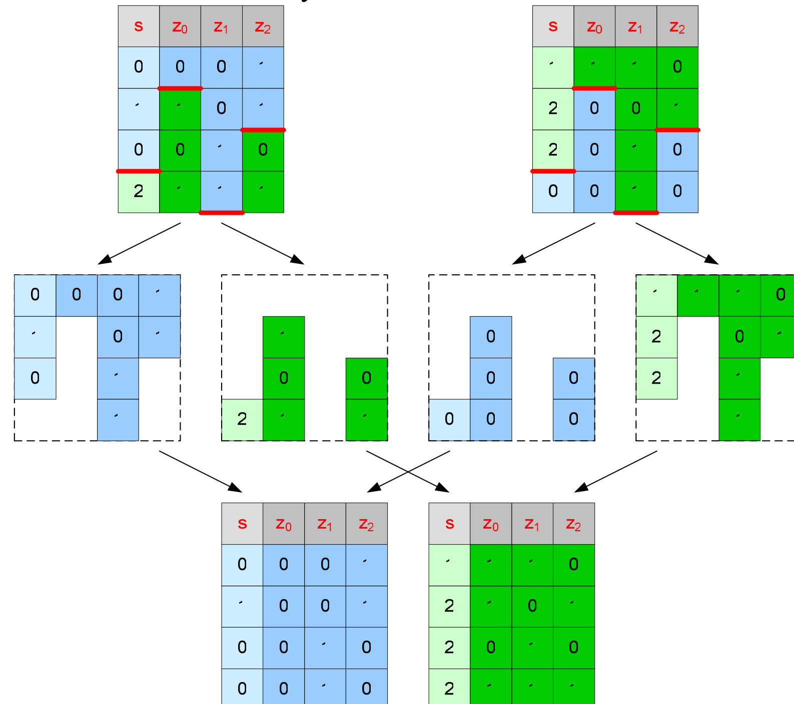


Рис. 3. Пример скрещивания полных таблиц

1.1.1.3. Представление состояний: сокращенные таблицы

Свойство локальности предикатов можно применять для сокращения описания управляющего состояния разными способами. Был выбран один из подходов, при котором число значимых в состоянии предикатов ограничивается некоторой константой r. К таблице, задающей сужение управляющей функции на данное состояние, в этом случае добавляется битовый вектор, описывающий множество значимых предикатов (рис. 4).

x_0	x_1	x_2	x_3	x_4	x_5
0	1	0	1	0	0

x_1	x_3	s	z_0	z_1	z_2
0	0	0	0	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	1	2	1	1	1

Рис. 4. Хромосома состояния: сокращенная таблица (n = 6, m = 3, r = 2, $|S|$ = 3)

Число строк таблицы в этом случае 2^r, однако, константа r обычно невелика. Ее выбор зависит от сложности задачи. Как показывает опыт, для большинства автоматизированных объектов среднее по всем состояниям значение r не больше пяти.

Опишем генетические операторы над хромосомами состояний, записанными в виде сокращенных таблиц [30, 31].

Алгоритм 3. Мутация сокращенных таблиц. По сравнению с представлением в виде полной таблицы, добавилась возможность мутации множества значимых предикатов. При этом каждый из значимых предикатов с некоторой вероятностью заменяется другим, который не принадлежит этому множеству (рис. 5).

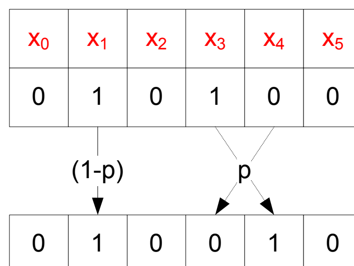


Рис. 5. Пример мутации множества значимых предикатов

Мутация сокращенной таблицы происходит так же, как мутация полной таблицы. Описание алгоритма приведено в листинге 3.

Листинг 3. Мутация сокращенных таблиц

```
State Mutate(State state) {
  State mutant = state;
  if (с вероятностью p) {
    int from, to;
    случайно выбрать from и to, так что
      (mutant.predicates[from] == 1) &&
```

```
            (mutant.predicates[to] == 0);
        mutant.predicates[from] = 0;
        mutant.predicates[to] = 1;
      }
      // Остальная часть хромосомы мутирует, как в случае полных
            //таблиц
      mutant.table = Mutate(mutant.table);
      return mutant;
    }
```

Алгоритм 4. Скрещивание сокращенных таблиц. Это наиболее сложный из предлагаемых алгоритмов. Основная последовательность его шагов отражена в листинге 4.

Листинг 4. Скрещивание сокращенных таблиц

```
    pair<State, State> Cross(State state1, State state2) {
      State child1 = state1;
      State child2 = child1;

      ChoosePreds(state1.predicates, state2.predicates,
        child1.predicates, child2.predicates);

      int crossPoint = случайное число от 0 до tableSize;
      FillChildTable(state1, state2, child1, crossPoint);
      FillChildTable(state1, state2, child2, crossPoint);
      return make_pair(child1, child2);
    }
```

Поскольку родительские хромосомы, представленные сокращенными таблицами, могут иметь разные множества значимых предикатов, сначала необходимо выбрать, какие из этих предикатов будут значимы для хромосом детей. Функция `ChoosePreds`, осуществляющая этот выбор, представлена в листинге 5.

Листинг 5. Выбор значимых предикатов детей при скрещивании сокращенных таблиц

```
    void ChoosePreds(Predicates p1, Predicates p2, Predicates
            ch1, Predicates ch2)
    {
      for (для всех i: номеров предикатов) {
        if (p1[i] && p2[i]) {      // Предикат от обоих родителей
```

```
          ch1[i] = ch2[i] = true; // достается обоим детям
          //запоминаем, что в наборах предикатов детей стало на
            //один меньше места;
        }
      }
      for (для всех i: номеров предикатов) {
        if (p1[i] != p2[i]) {
          Predicates* pCh;
          if (у обоих детей есть место) {
            pCh = равновероятно любой ребенок;
          } else {
            pCh = тот ребенок, у которого еще есть место;
          }
          (*pCh)[i] = true;
          запоминаем, у кого стало меньше места;
        }
      }
    }
```

В результате работы функции `ChoosePreds` предикаты, значимые для обоих родителей, наследуются обоими детьми, а каждый из тех предикатов, которые были значимы лишь для одной родительской особи, равновероятно достанется любому из двух детей. Пример работы функции для родительских хромосом, представленных на рис. 6, проиллюстрирован на рис. 7.

x_0	x_1	x_2	x_3	x_4	x_5
0	1	0	1	0	0

x_1	x_3	s	z_0	z_1	z_2
0	0	0	0	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	1	2	1	1	1

x_0	x_1	x_2	x_3	x_4	x_5
1	1	0	0	0	0

x_0	x_1	s	z_0	z_1	z_2
0	0	1	1	1	0
0	1	2	0	0	1
1	0	2	0	1	0
1	1	0	0	1	0

Рис. 6. Родительские хромосомы, представленные сокращенными таблицами

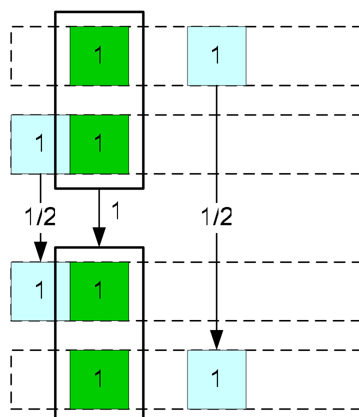


Рис. 7. Пример выбора значимых предикатов детей

После выбора значимых предикатов заполняются таблицы обоих детей. Алгоритм заполнения представлен в листинге 6.

Листинг 6. Заполнение таблиц детей при скрещивании сокращенных таблиц

```
void FillChildTable(State s1, State s2, State& child, int
        crossPoint)
{
  for (для всех i: строк таблицы child) {
    vector<int> lines1 = выбрать строки таблицы s1, в
        которых предикаты,
      значимые для child, имеют те же значения, что в
        строке i,
      причем, если предикат значим для обоих родителей и i
        >=        crossPoint, то его значение не
        учитывается;
    vector<int> lines2 = выбрать строки таблицы s2, в
        которых предикаты,
      значимые для child, имеют те же значения, что в
        строке i,
      причем, если предикат значим для обоих родителей и i
        <         crossPoint, то его значение не
        учитывается;
        vector<Probability> p1(nStates);
    vector<Probability> p2(nStates);
    for (для всех j из lines1) {
      p1[целевое состояние у s1 в строке j] += 1.0;
    }
```

```
        for (для всех j из lines2) {
          p2[целевое состояние у s2 в строке j] += 1.0;
        }
        Поделить значения p1 на число строк из lines1;
        Поделить значения p2 на число строк из lines2;
        vector<Probability> p = p1 + p2;
        child[i].targetState = выбрать случайно с
            распределением вероятностей p;
        for (для всех k: номеров действий) {
          Probability q1, q2;
          for (для всех j из lines1) {
            q1 += s1[j].output[k];
          }
          for (для всех j из lines2) {
            q2 += s2[j].output[k];
          }
          Поделить q1 на число строк из lines1;
          Поделить q2 на число строк из lines2;
          child[i].output[k] = 1 с вероятностью (q1 + q2)/2 и
            0 иначе;
        }
      }
    }
```

Иллюстрация примера заполнения первой строки таблицы одного из детей приведена на рис. 8. В этой реализации оператора скрещивания на значения каждой строки таблицы ребенка влияют значения нескольких строк родительских таблиц. При этом конкретное значение, помещаемое в ячейку таблицы ребенка, определяется «голосованием» всех влияющих на нее ячеек родительских таблиц.

В описанном выше варианте алгоритма все состояния автомата имеют равное число значимых предикатов (r – константа для всего процесса оптимизации). Однако предложенный алгоритм скрещивания легко расширяется на случай разного числа значимых предикатов у пары родителей.

1.1.1.4. Мутация, зависящая от пригодности

В классическом генетическом алгоритме мутации применяются к хромосомам с некоторой вероятностью, постоянной в процессе оптимизации. В настоящей работе принято целесообразным введение зависимости вероятности мутации особи от ее пригодности.

Применение интенсивной мутации к плохо приспособленным особям повышает эффективность эволюции, позволяет покидать локальные оптимумы и ослабляет проблему преждевременной сходимости популяции, тогда как для особей с высокой пригодностью оптимальны менее интенсивные мутации.

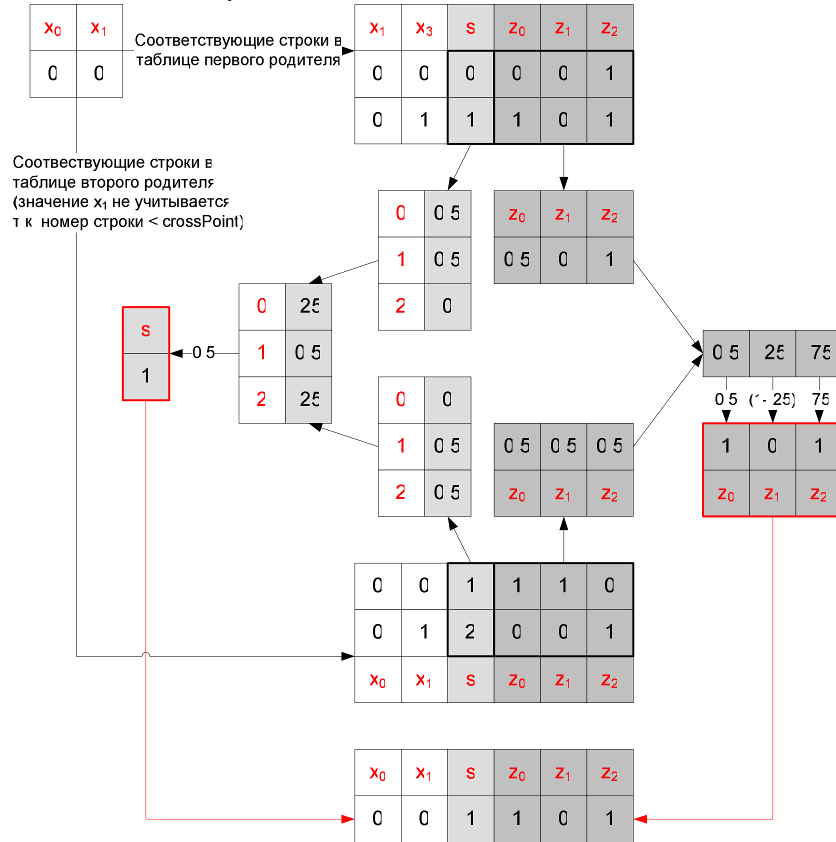


Рис. 8. Пример заполнения строки таблицы ребенка при скрещивании сокращенных таблиц

В метрическом пространстве оптимальный модуль вектора мутации можно с высокой точностью оценить сверху расстоянием между особью и глобальным оптимумом оценочной функции. Это расстояние в задаче не известно, однако можно ожидать тенденции к его сокращению с ростом пригодности особи.

Для иллюстрации рассмотрим частный случай. Пусть пространство поиска евклидово, а функция пригодности непрерывна и возрастает с приближением к оптимуму. В этом случае можно проанализировать зависимость вероятности успешной мутации от модуля вектора мутации (рис. 9).

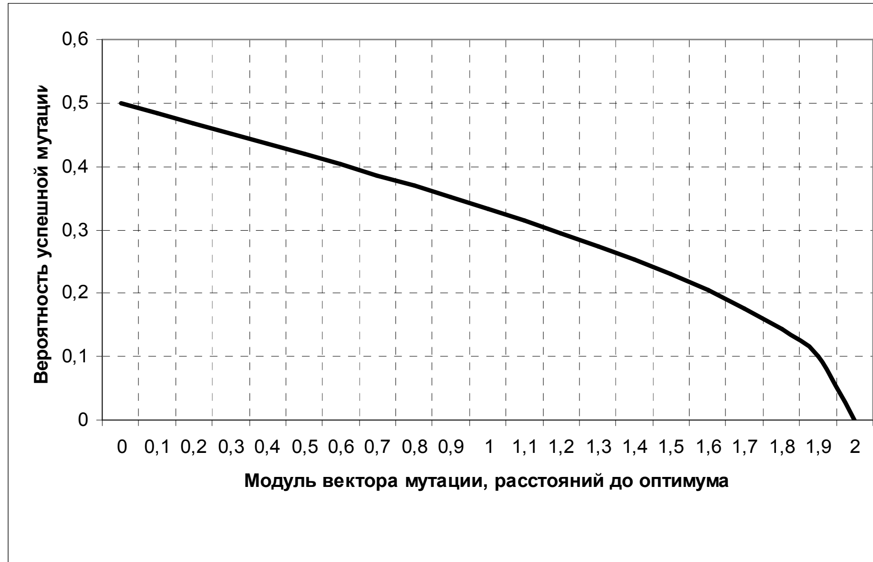


Рис. 9. Вероятность улучшения пригодности после мутации

График на рис. 9 отражает снижение вероятности улучшения пригодности особи после мутации с увеличением модуля вектора мутации. График на рис. 10 показывает характерное изменение расстояния до оптимума при успешной мутации.

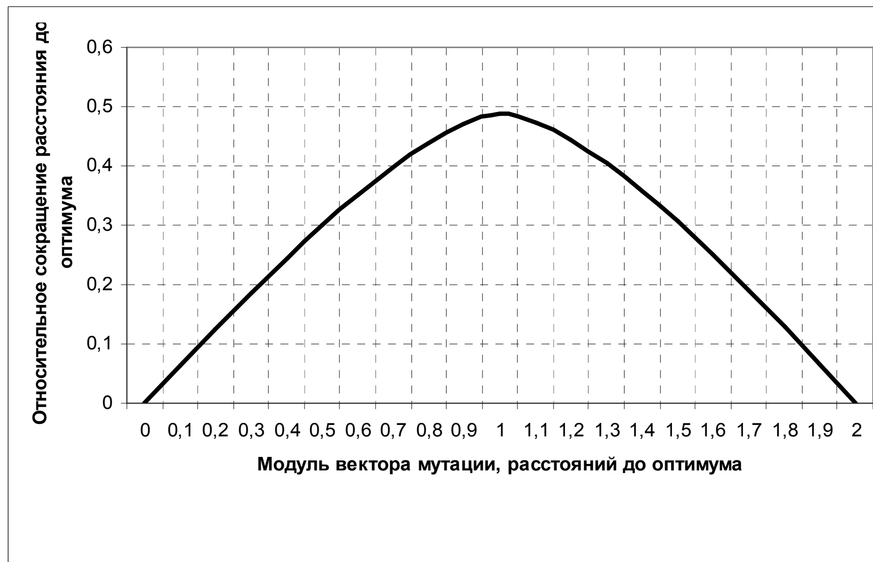


Рис. 10. Математическое ожидание относительного сокращения расстояния до оптимума в результате успешной мутации

Этот график показывает, что максимальная эффективность успешной мутации достигается в том случае, когда ее модуль равен (неизвестному) расстоянию до оптимума.

Из двух предыдущих диаграмм получена диаграмма, представленная на рис. 11. Она отражает зависимость математического ожидания относительного сокращения расстояния до оптимума в результате попытки применения мутации.

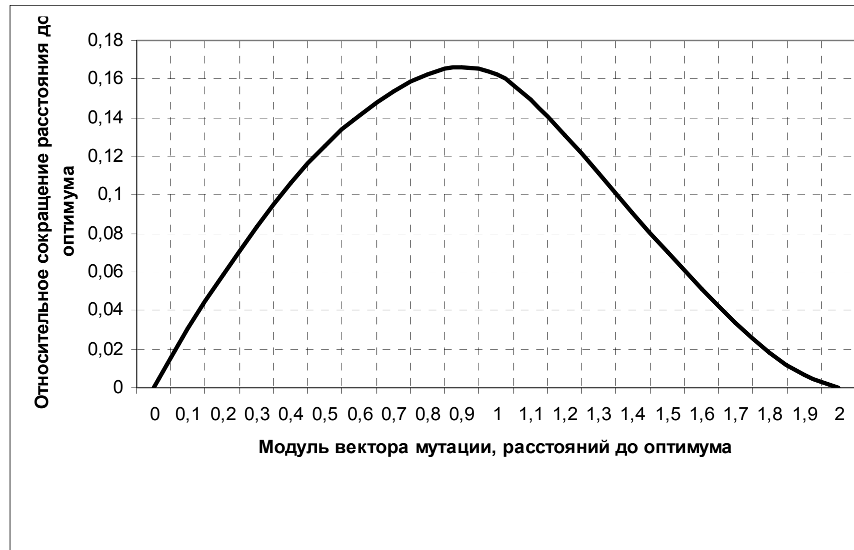


Рис. 11. Математическое ожидание сокращения расстояния до оптимума за одну операцию оценки пригодности

Оптимальная интенсивность мутации зависит от неизвестного расстояния до оптимума. С другой стороны, пригодность имеет тенденцию к возрастанию по мере приближения к оптимуму. Из этого можно сделать вывод о склонности оптимальной интенсивности мутации к обратной зависимости от значения оценочной функции. В экспериментах использовалась интенсивность мутации, обратно пропорциональную пригодности.

1.1.1.5. Формирование нового поколения

Для учета зависимости интенсивности мутации от пригодности требуется наличие оценки пригодности особи перед ее мутацией. В методах генетической оптимизации стандартный подход к формированию нового поколения предполагает следующий порядок действий: выбор пары особей из популяции родителей, их скрещивание, мутация детей, и, наконец, добавление детей, возможно мутировавших, в новую популяцию. Однако в этом случае требуется дополнительная оценка пригодности детей перед мутацией, что приводит к удвоению объема вычислений оценочной функции. Для сохранения эффективности оптимизации необходимо внести изменения в процесс формирования нового поколения. Например, можно добавить новую особь в популяцию два раза – до и после мутации – или выбрать лучшую из них.

В рамках использования метода сокращенных таблиц предлагается другой подход, предполагающий применение к каждой особи поколения не более одного генетического оператора. При этом часть особей с лучшей пригодностью переносится в новую популяцию без изменений, а каждое из оставшихся в новой популяции мест заполняется либо результатом скрещивания пары особей из родительской популяции, либо результатом мутации особи из родительской популяции.

1.1.1.6. Выбор особей для скрещивания

В процессе экспериментальной проверки был использован подход, при котором вероятность выбора особи в качестве родителя пропорциональна ее пригодности. Этот метод известен под называнием *fitness proportional selection*.

1.1.2. Метод представления автоматов деревьями решений

В настоящем разделе описывается метод представления автоматов деревьями решений [27, 30, 31].

1.1.2.1. Представление автомата деревьями решений

Дерево решений является удобным способом задания дискретной функции, зависящей от конечного числа логических 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 сформировать 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 переменной *x*.

Метод представления автомата с помощью деревьев решений заключается в следующем: функции переходов и выходов автомата выражаются с помощью деревьев решений. Более формально: зададим для каждого состояния $q \in Q$ функцию $\sigma_q : X \to Q \times Y$, такую что $\sigma_q(x) = (\delta(q, x), \lambda(q, x))$ для $\forall x \in X$. Здесь Q – множество состояний автомата, X – множество входных воздействий, Y – множество выходных воздействий, $\delta : Q \times X \to Q$ – функция переходов, $\lambda : Q \times X \to Y$ – функция выхода. Каждая из этих функций может быть определена собственным деревом решений. *Таким образом, автомат в целом может быть представлен упорядоченным набором деревьев решений и стартовым состоянием.*

Каждое состояние автомата представляется с помощью соответствующего дерева решений. Деревья решений состоят из узлов, среди которых выделяется корень. Каждый узел представляется следующим образом:

- номер переменной, соответствующей метке узла;
- указатель на дочерний узел, соответствующий нулевому значению переменной (для внутренних узлов);
- указатель на дочерний узел, соответствующий единичному значению переменной (для внутренних узлов);
- ассоциированное выходное действие (для листьев);
- номер состояния, в которое ведет переход из данной вершины (для листьев).

Следовательно, при использовании описываемого метода автомат представляется объектом следующего вида на языке *Java*:

```
class TreeAutomata {
    Tree[] trees;
    int startState;
}

class Tree {
    Node root;

    private class Node {
        Node left;
        Node right;
```

```
        int transState;
        int action;
    }
}
```

1.1.2.2. Генерация случайной особи

Для того чтобы сгенерировать внутреннюю вершину дерева применяется следующий алгоритм:

- выбрать переменную, по которой в данной вершине дерева будет проводиться расщепление;
- сгенерировать левого ребенка;
- сгенерировать правого ребенка.

Генерация листа происходит следующим образом:

- случайным образом выбрать номер состояния, в которое будет вести переход из данного листа;
- случайный образом сгенерировать вектор действий.

Для генерации случайного дерева решений применяется следующий алгоритм: с вероятностью 0.5 генерируется лист или внутренний узел.

Данный алгоритм может продолжать свою работу бесконечно долго. Для того, чтобы этого избежать введено ограничение на высоту дерева. Если при генерации высота дерева начинает превышать удвоенное число возможных входных переменных, то обязательно генерируется лист.

Для того, чтобы сгенерировать особь необходимо выполнить следующие действия:

- сгенерировать необходимое число случайных состояний (деревьев решений);
- выбрать случайным образом начальное состояние автомата.

1.1.2.3. Алгоритм обрезки недостижимых ветвей дерева

Если по пути из корня до некоторого узла переменная, по которой происходит расщепление в этом узле, встречается дважды, то ветвь, соответствующая значению противоположному тому, которое было выбрано при первом расщеплении, будет *недостижимой*. Рис. 12 поясняет это утверждение.

Светло-серым цветом на рис. 12 отмечены вершины с повторяющейся переменной расщепления, а темно-серым – недостижимая ветвь.

Для обрезки недостижимых ветвей используется модификация алгоритма *поиска в глубину* (*Depth First Search*). При рекурсивном спуске запоминаются переменные, по которым уже проводилось расщепление на пути из корня в текущий узел, а также значения этих переменных, соответствующие этому пути. Если переменная встречается второй раз, то текущий узел, заменяется на корень достижимого поддерева этого узла. Решение о том, какое из поддеревьев достижимо, принимается на основании запомненных значений переменных.

Работу данного алгоритма иллюстрирует рис. 13.

1.1.2.4. Оператор мутации

Оператор мутации выполняет:

- с вероятностью *0.5* случайное изменение стартового состояния;
- мутацию случайного состояния.

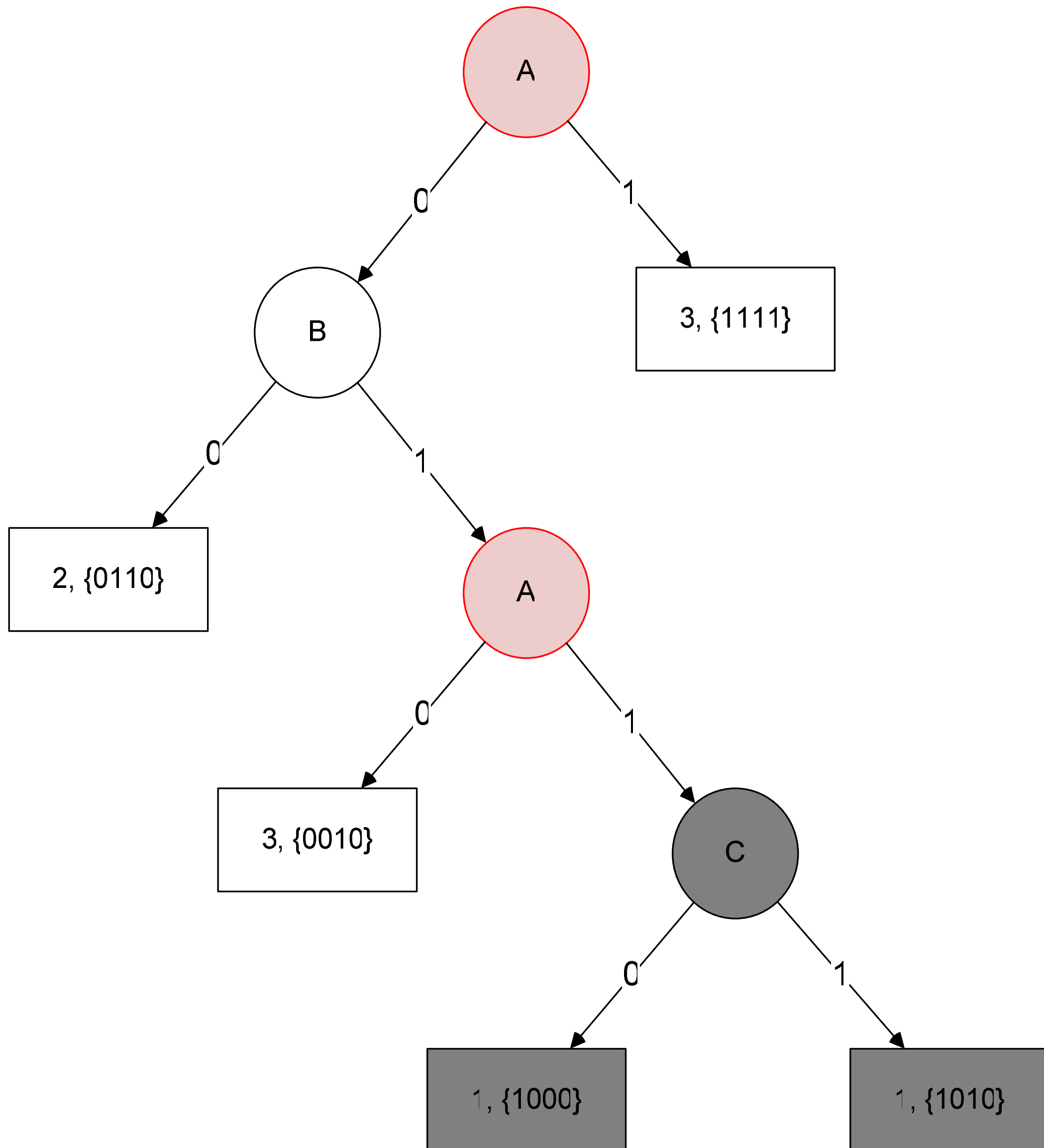


Рис. 12. Недостижимые ветви дерева решений

Оператор мутации состояния, также как и алгоритм обрезки недостижимых ветвей, представляет собой модификацию алгоритма поиска в глубину. При каждом рекурсивном вызове этого алгоритма выполняются следующие действия:

- если текущий узел является листом, то обязательно, а для внутренних узлов с вероятностью P, вернуть случайно сгенерированное дерево решений;
- иначе равновероятно пойти в одно из поддеревьев.

Запуск описанной рекурсивной процедуры производится с корня дерева.

Фактически данный алгоритм случайно выбирает некоторое поддерево и заменяет его на случайно сгенерированное. Выбор поддерева происходит не равновероятно – чем выше узел, тем больше вероятность выбора его поддерева. Создание случайного поддерева происходит по алгоритму описанному выше.

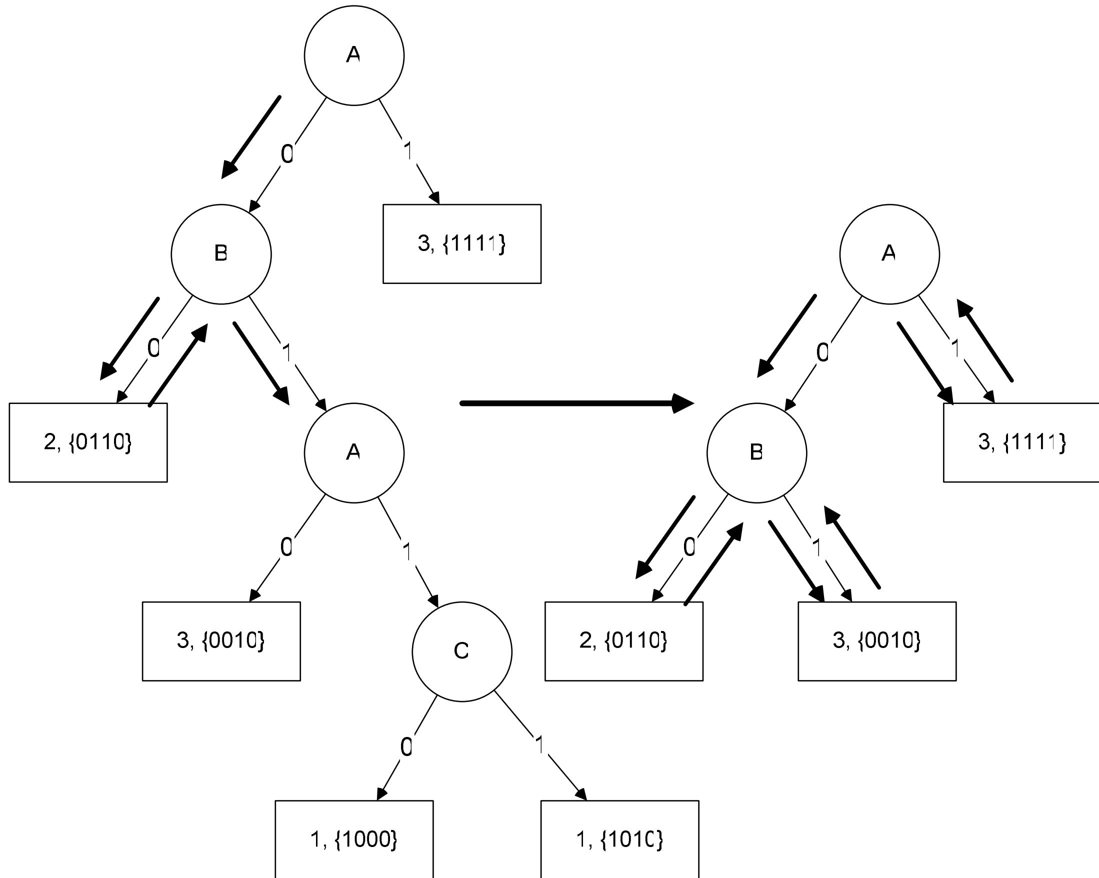


Рис. 13. Удаление недостижимых ветвей

По окончании мутации для мутированной особи необходимо запустить алгоритм обрезки недостижимых ветвей дерева. Это иллюстрирует рис. 14. На нем изображена ситуация, в которой после описанной операции мутации существует путь из корня до листа, проходящий дважды через вершину с переменной расщепления *A*.

1.1.2.5. Оператор скрещивания

Оператор скрещивания особей представляется следующим образом.

Обозначим родительские особи – *P1* и *P2*, а детей – *S1* и *S2*. Обозначим *k*-ое состояние автомата *A*, как *A.a[k]*. Обозначим *c1[k]* и *c2[k]* результат скрещивания состояний *P1.a[k]* и *P2.a[k]*. Тогда для любого *k* будет верно утверждение:

- *S1.a[k] = c1[k], S2.a[k] = c2[k]*.

Оператор скрещивания состояний фактически выбирает два поддерева: одно из первого дерева решений, второе из второго, а затем меняет их местами. Этот алгоритм также как операция мутации и алгоритм обрезки недостижимых ветвей представляет собой модификацию алгоритма поиска в глубину и имеет рекурсивную структуру. При каждом рекурсивном вызове этого алгоритма выполняются следующие действия:

- если текущий узел является листом, то обязательно, а для внутренних узлов с вероятностью *P*, вернуть данное поддерево;
- иначе равновероятно пойти в одно из поддеревьев текущего узла.

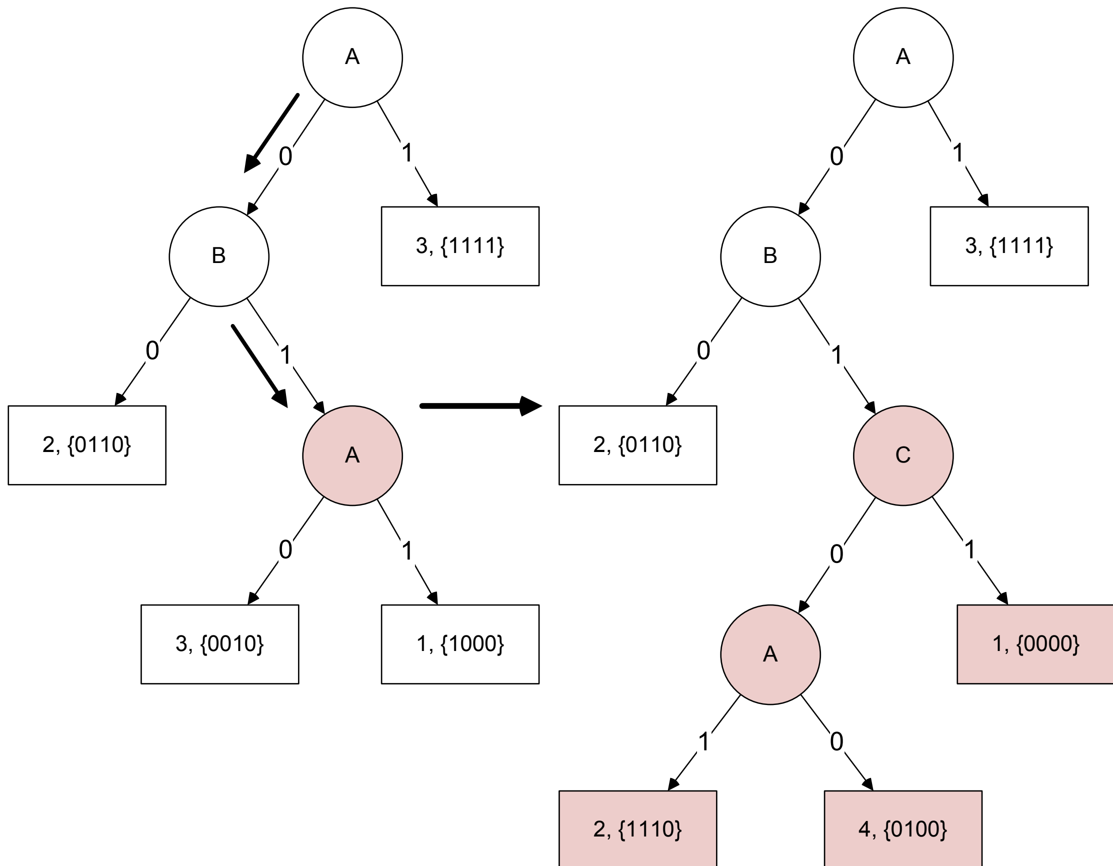


Рис. 14. Мутация деревьев решений

На рис. 15 приведен пример, в котором необходимо, как и в случае с оператором мутации, провести обрезку недостижимых ветвей. При этом для всех деревьев, соответствующих состояниям автомата, запускается алгоритм обрезки недостижимых ветвей.

1.1.3. Метод совместного использования конечных автоматов и нейронных сетей

В настоящем разделе описывается метод совместного использования конечных автоматов и нейронных сетей.

1.1.3.1. Некоторые проблемы, возникающие при использовании генетического программирования для построения конечных автоматов

Генетическое программирование [17, 18] наиболее эффективно в тех случаях, когда оптимизируемый объект (например, конечный автомат) имеет небольшой размер (небольшое число состояний). В то же время, число различных вариантов значений входных переменных может быть достаточно большим, а сами переменные могут быть не только логическими, но и числовыми. Например, в рассмотренной в отчете по третьему этапу работ задаче управления моделью беспилотного летательного аппарата ряд входных переменных были вещественными. В задаче «Умный муравей-2», на примере которой в настоящем отчете иллюстрируется применение предлагаемых методов представления автоматов и разработанных программных средств, вещественных входных переменных нет, однако логических входных переменных восемь, что так же достаточно много для представления автоматов с помощью полных таблиц переходов.

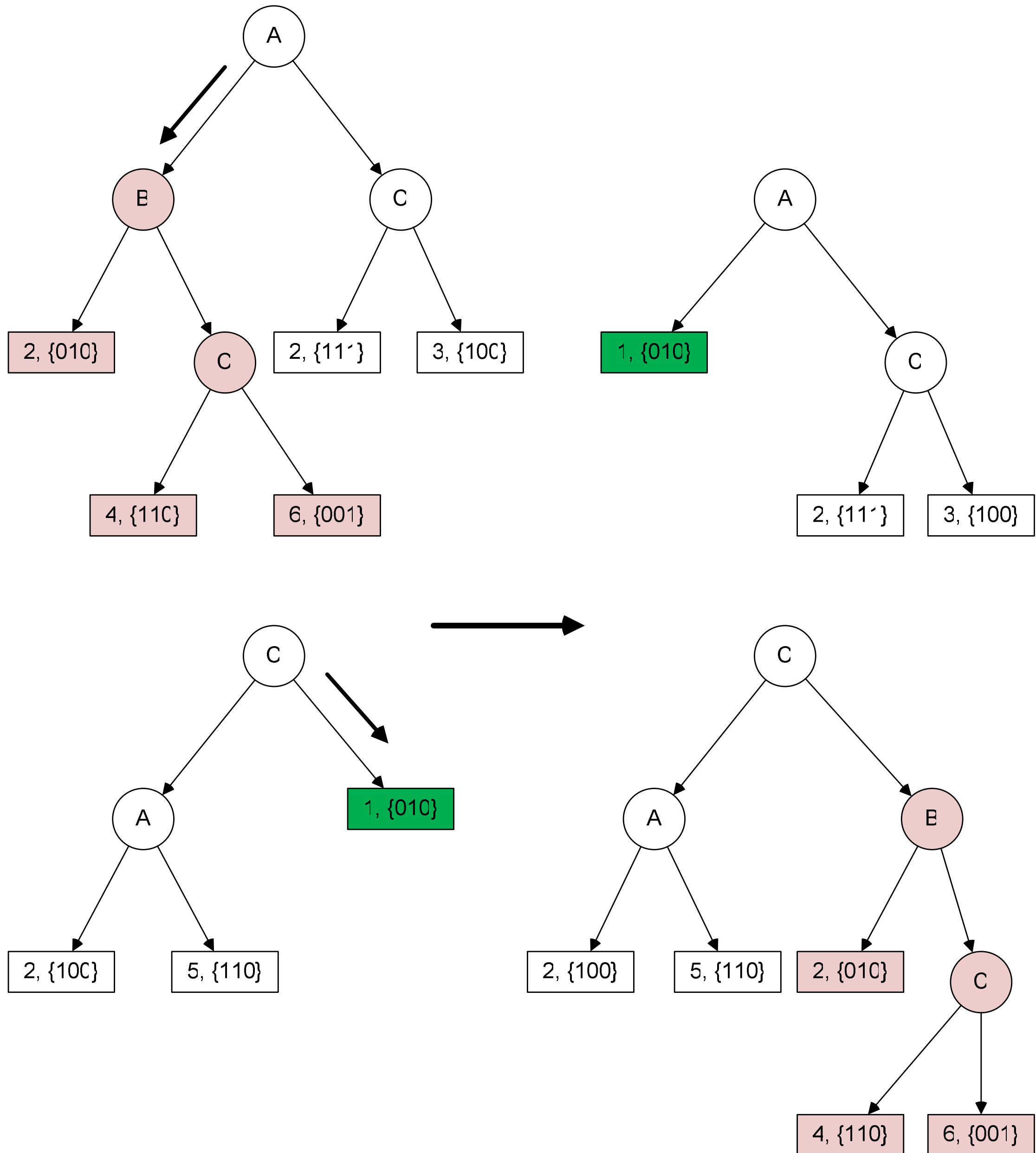


Рис. 15. Пример скрещивание деревьев решений

Обычно с помощью генетических алгоритмов строятся автоматы с входными переменными логического типа. Для того чтобы применять такой алгоритм, необходимо разработать способ перехода от произвольных входных переменных к логическим входным переменным (или, хотя бы, к переменным, множество значений которых конечно и содержит небольшое число элементов).

Одним из вариантов решения этой задачи является введение соответствующих переменных. Например, если исходно были две вещественные переменные x и y, то, например, можно ввести две новые логические переменные A равную $x > 100$ и B равную $y < 200$.

Второй вариант решения состоит в том, чтобы разбить множество значений входных переменных на несколько областей и использовать в качестве значения входной переменной номер

области, в которой лежат текущие значения входных переменных. Таким образом, перед тем, как подавать данные на вход автомата, необходимо определять, в какой из областей лежит набор текущих значений входных переменных. Это – *задача классификации*. Если для ее решения применять автоматический классификатор (нейронная сеть, дерево принятия решений, и т.д.), то возникает идея настраивать этот классификатор совместно с автоматом, с которым он связан.

В настоящем разделе описывается одна из возможных реализаций второго подхода. В качестве классификатора используется нейронная сеть. Ее настройка и построение автомата производится совместно с помощью генетического программирования.

1.1.3.2. Искусственные нейронные сети

Нейрон – это клетка головного мозга или нервной системы, основной функцией которой является сбор, обработка и распространение электрических сигналов. Считается, что способность мозга к обработке информации обусловлена функционированием сетей, состоящих из нейронов.

Одна из первых математических моделей нейрона предложена Мак-Каллоком (McCulloch) и Питтсом (Pitts) [20]. С 1943 года были разработаны более подробные и реалистичные модели, как нейрона, так и более крупных систем мозга. Это привело к созданию новой научной области – *вычислительной неврологии*. С другой стороны, исследователи в области искусственного интеллекта исследовали более абстрактные свойства нейронных сетей: способность выполнять распределенные вычисления, справляться с зашумленными входными данными, обучаться и т. д.

Со временем стало ясно, что похожими свойствами обладают и другие системы (такие, как, например, байесовские сети). Однако *искусственные нейронные сети* (в дальнейшем, нейронные сети) [32] по настоящее время остаются одним из наиболее изученных и широко применяемых методов искусственного интеллекта.

1.1.3.3. Элементы нейронных сетей

Нейронные сети состоят из *узлов* (искусственных нейронов), соединенных между собой *связями*. Связь от элемента i к элементу j предназначена для распространения *активации* a_j от j к i. Каждая связь имеет назначенный ей числовой *вес* $W_{i,j}$. Каждый элемент вычисляет взвешенную сумму своих входных данных:

$$in_i = \sum_{j=0}^{n} W_{j,i} a_j$$

и применяет к ней *функцию активации* g:

$$a_i = g(in_i) = g(\sum_{j=0}^{n} W_{j,i} a_j).$$

Обратим внимание на то, что в эту формулу входит *смещенный вес* $W_{0,i}$, относящийся к постоянному входному значению $a_0 = -1$.

Основными видами функций активации являются:

- *пороговая* функция

$$g(x) = \begin{cases} 1, x \geq 0 \\ 0, x < 0 \end{cases};$$

- *знаковая* функция

$$g(x) = \begin{cases} 1, x \geq 0 \\ -1, x < 0 \end{cases};$$

- *сигмоидальная* (*логистическая*) функция

$$\mathrm{g(x)} = \frac{1}{1 + \mathrm{e}^{-x}}.$$

Отметим, что эти функции имеют пороговое значение около нуля, а смещенный вес $W_{0,i}$ фактически задает пороговое значение для данного элемента.

Важным свойством элементов с пороговой функцией активации является то, что с их помощью можно представить логические функции *AND*, *OR* и *NOT* [32]. Таким образом, с помощью нескольких таких элементов можно выразить любую булеву функцию от входов сети.

1.1.3.4. Структура нейронных сетей

Существуют две основные категории структур нейронных сетей: ациклические сети (*сети с прямым распространением)* и циклические (*рекуррентные*) сети.

Сети с прямым распространением реализуют некоторую функцию от своих входов, в то время как в рекуррентной сети выходы ее элементов могут подаваться на вход. В связи с этим уровни активации в рекуррентной сети могут находиться в устойчивом состоянии, могут переходить в колебательный или даже в хаотический режим.

Рекуррентные сети представляют собой более сложную для понимания и исследования модель. В настоящей работе будут использоваться только сети с прямым распространением.

1.1.3.5. Применение нейронных сетей

Наиболее часто нейронные сети применяются для решения следующих задач:

- *классификация образов* – указание принадлежности входного образа, представленного вектором признаков, одному или нескольким предварительно определенным классам;
- *кластеризация* – классификация образов при отсутствии обучающей выборки с метками классов;
- *прогнозирование* – предсказание значения y_{n+1} при известной последовательности $y_1, y_2 \ldots y_n$.

В настоящей работе, как отмечалось выше, нейронные сети используются для получения по значениям большого числа входных переменных значений небольшого числа логических переменных, которые подаются на вход конечного автомата.

1.1.3.6. Управление системой со сложным поведением

Для управления системой со сложным поведением предлагается совместно применять нейронную сеть и конечный автомат.

При этом, как отмечалось выше, нейронная сеть используется для классификации значений вещественных входных переменных и выработки входных логических переменных для автомата, а автомат – для выработки выходных воздействий на беспилотный летательный аппарат (рис. 16).

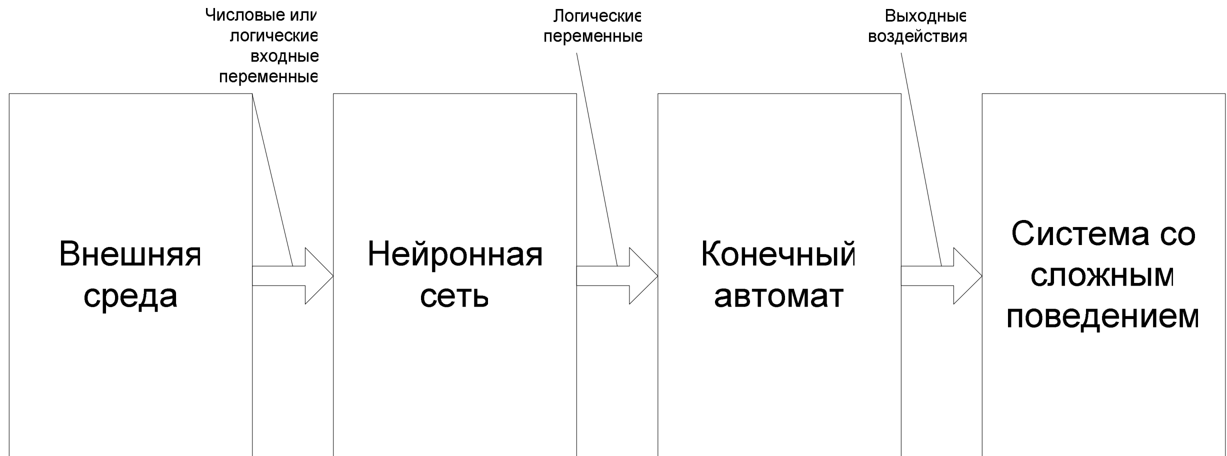


Рис. 16. Структурная схема системы управления

Одна из возможных структур нейронной сети и способ ее взаимодействия с конечным автоматом показаны на рис. 17. Отметим, что для решения других задач структура нейронной сети может быть изменена.

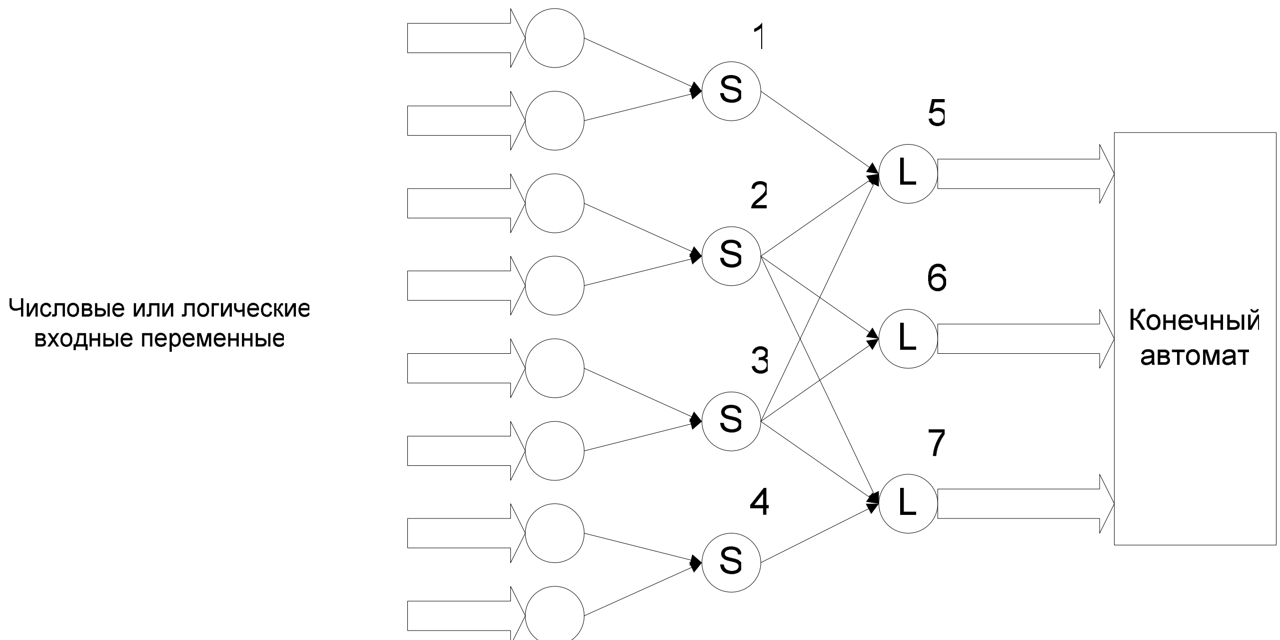


Рис. 17. Возможная структура нейронной сеть и ее взаимодействие с конечным автоматом

Символами **S** на рис. 17 обозначены нейроны с сигмоидальной функцией активации, символом **L** – нейроны с пороговой функцией активации. Рядом с нейронами указаны их

номера (они используются при описании операции скрещивания нейронных сетей). На каждый из трех выходов нейронной сети поступает число равное нулю или единице. Таким образом, существует восемь вариантов комбинаций выходных сигналов нейронной сети (000, 001, 010, 011, 100, 101, 110, 111), подаваемых на вход конечного автомата.

1.1.3.7. Алгоритм генетического программирования для построения конечного автомата и нейронной сети для управления системой со сложным поведением

В настоящем разделе описан алгоритм генетического программирования, используемый для построения совокупности конечного автомата и нейронной сети, осуществляющих управление системой со сложным поведением. *Алгоритм генетического программирования* состоит из пяти частей:

- создание начального поколения;
- мутация;
- скрещивание (кроссовер);
- отбор особей для формирования следующего поколения;
- вычисление функции приспособленности (фитнес-функции).

1.1.3.8. Структура особи в используемом алгоритме

Особь в описываемом алгоритме генетического программирования состоит из нейронной сети и конечного автомата (рис. 18).

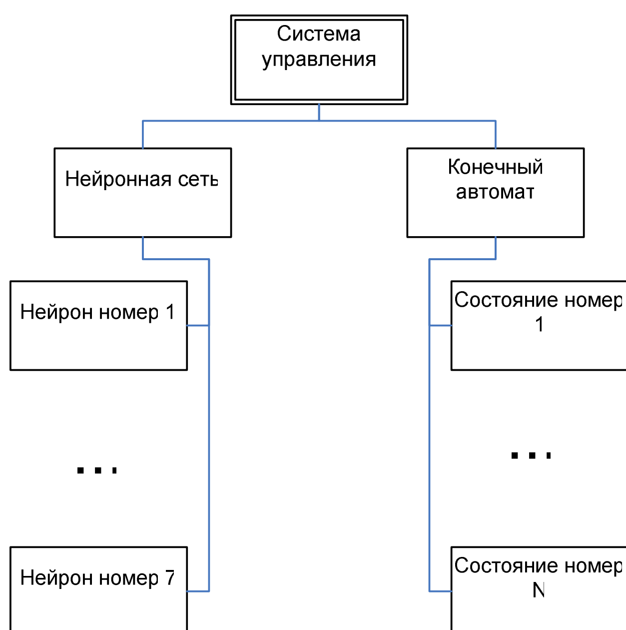


Рис. 18. Структура особи

Нейронная сеть состоит из четырех нейронов с сигмоидальной функцией активации и трех нейронов с пороговой функцией активации. Каждый из нейронов характеризуется порогом активации и весами связей, которые соединяют другие элементы сети с рассматриваемым. На языке программирования *Java* нейрон представляется следующим образом:

```
public abstract class Neuron {
     protected Neuron[] inputs;
     protected int inputsCnt;
     protected double[] w;
}
```

Описание конечного автомата состоит из номера начального состояния и описания состояний. Описание состояния состоит из описаний восьми переходов, соответствующих восьми вариантам значений входных переменных автомата. Описание каждого перехода состоит из номера состояния, в которое ведет этот переход, и множества выходных воздействий, которые вырабатываются при выборе этого перехода.

```
public class Individual {
     private NeuralNet neuralNet;
     private Automaton automaton;
}
```

1.1.3.9. Создание начального поколения

Начальное поколение заполняется случайно сгенерированными системами управления. При этом в каждой системе управления случайным образом генерируется конечный автомат и нейронная сеть – веса связей в ней инициализируются случайными числами от минус единицы до единицы.

1.1.3.10. Операция мутации

Мутация особи. При мутации особи мутирует либо нейронная сеть, либо конечный автомат.

Мутация нейронной сети. При мутации нейронной сети мутирует случайно и равновероятно выбирается один элемент (искусственный нейрон) сети и мутирует.

Мутация элемента сети. При мутации элемента сети случайно выбирается один из весов связей, и к нему прибавляется случайное число из отрезка [-0.05; 0.05]. Кроме этого, с вероятностью 0.5 аналогичная операция производится с лимитом активации нейрона.

Мутация нейронной сети проиллюстрирована на рис. 19.

Мутация конечного автомата. При мутации конечного автомата равной вероятностью производится либо изменение начального состояния, либо мутация случайно выбранного перехода.

Изменение начального состояния. Начальное состояние изменяется на случайно выбранное состояние автомата.

Мутация перехода. При мутации перехода с равной вероятностью происходит либо изменение номера состояния, в которое ведет переход, либо мутация одного из действий, связанных с переходом.

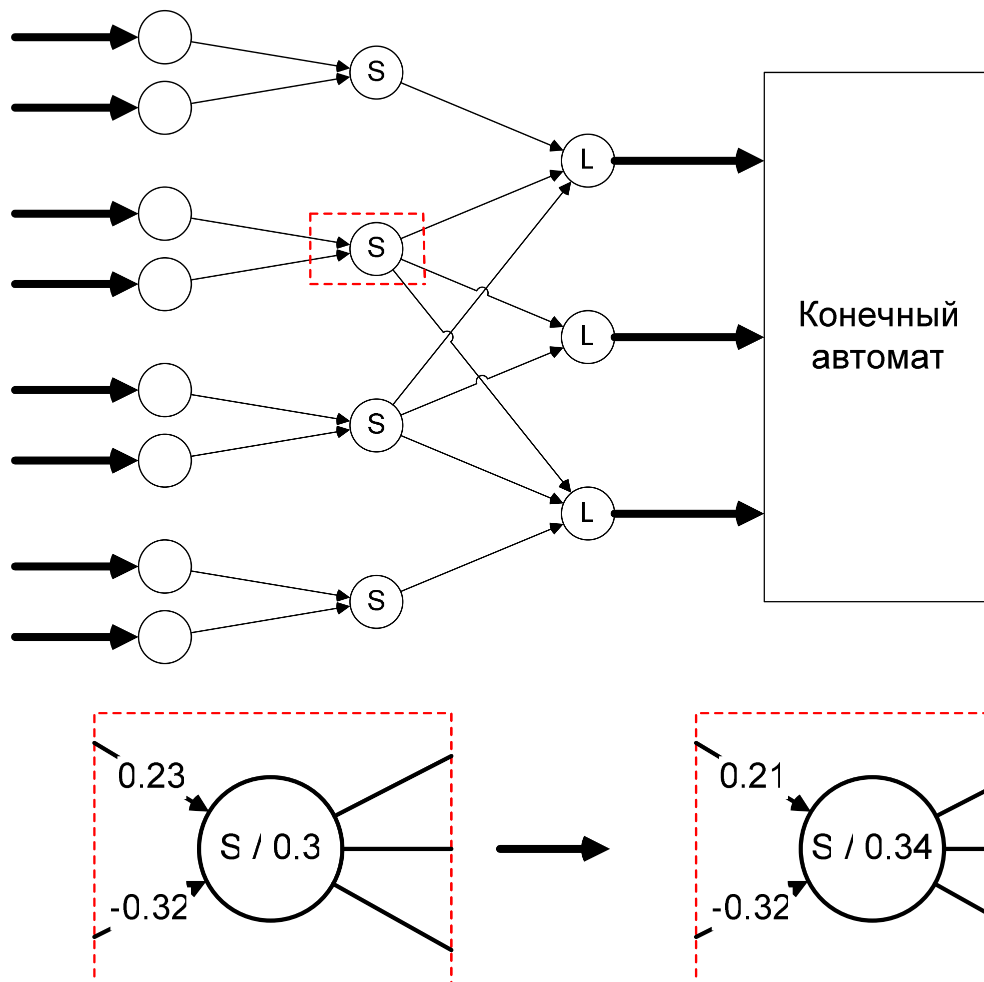


Рис. 19. Мутация нейронной сети

1.1.3.11. Операция скрещивания

Оператор скрещивания получает на вход две особи (*P1*, *P2*) и выдает две особи (*S1*, *S2*). Пусть *s* – некоторая особь. Обозначим как *s.ns* входящую в нее нейронную сеть, а как *s.a* – входящий в нее автомат.

Скрещивание систем управления. При скрещивании систем управления *s1* и *s2* происходит скрещивание автоматов *s1.a* и *s2.a* и скрещивание нейронных сетей *s1.ns* и *s2.ns*. Обозначим получающиеся в результате описанных скрещиваний автоматы *a1* и *a2*, а нейронные сети – *ns1* и *ns2*. В результате скрещивания системы управления получаются системы управления *s3* и *s4*, содержащие следующие элементы: *s3* содержит *a1* и *ns1*, *s4* – *a2* и *ns2*.

Скрещивание автоматов. Обозначим автоматы, поступающие на вход оператора скрещивания автоматов, *A1* и *A2*. Начальное состояние некоторого автомата *A* обозначим *A.is*, а переход из состояния *i* по значению входной переменной *j* как *A(i, j)*. Обозначим автоматы, получающиеся в результате скрещивания, как *A3* и *A4*. Для их начальных состояний справедливо:

- либо *A3.is* = *A1.is* и *A4.is* = *A2.is*;
- либо *A3.is* = *A2.is* и *A4.is* = *A1.is*.

Опишем переходы автоматов *A3* и *A4*. Скрещивание производится отдельно для каждого состояния *i* и для каждого значения *j* входной переменной. В каждом случае возможно два равновероятных варианта:

- *A3(i, j) = A1(i, j)* и *A4(i, j) = A2(i, j)*;
- *A3(i, j) = A2(i, j)* и *A4(i, j) = A1(i, j)*.

Проиллюстрируем скрещивание автоматов на примере случая одной входной переменной. Обозначим переход из состояния номер *i* в автомате *A1* по значению входной переменной «1» как *A1(i, 1)*, а по значению «0» как *A1(i, 0)*. Аналогичный смысл придадим обозначениям *A2(i, 0)* и *A2(i, 1)*. Тогда для переходов из состояния с номером *i* в автоматах-потомках *A3* и *A4* будет справедливо одно из четырех соотношений:

- либо *A3(i, 0) = A1(i, 0)*, *A4(i, 1) = A2(i, 1)* и *A4(i, 0) = A2(i, 0)*, *A4(i, 1) = A1(i, 1)*;
- либо *A3(i, 0) = A2(i, 0)*, *A4(i, 1) = A1(i, 1)* и *A4(i, 0) = A1(i, 0)*, *A4(i, 1) = A2(i, 1)*;
- либо *A3(i, 0) = A1(i, 0)*, *A4(i, 1) = A1(i, 1)* и *A4(i, 0) = A2(i, 0)*, *A4(i, 1) = A2(i, 1)*;
- либо *A3(i, 0) = A2(i, 0)*, *A4(i, 1) = A2(i, 1)* и *A4(i, 0) = A1(i, 0)*, *A4(i, 1) = A1(i, 1)*.

Все четыре варианта соотношений равновероятны. Возможные варианты переходов изображены на рис. 20.

В левой части этого рисунка показаны переходы из состояния номер *i* автоматов, поступающих на вход операции скрещивания, а в правой части – четыре возможных варианта переходов автоматов, которые будут получены в результате ее применения. Переходы в левой части рисунка пронумерованы числами от одного до четырех. Переходы в правой части также пронумерованы, причем нумерация переходов соответствует нумерации в левой части рисунка.

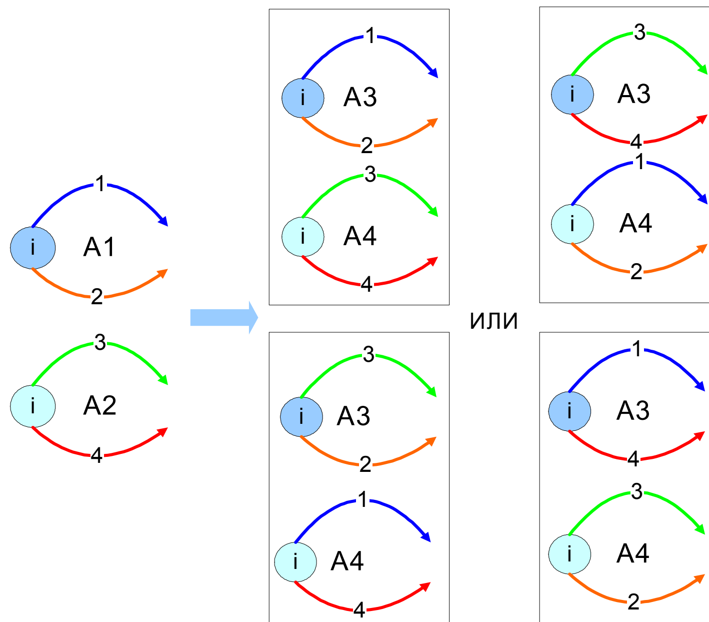


Рис. 20. Варианты переходов при скрещивании

Скрещивание нейронных сетей. Обозначим нейронные сети, поступающие на вход оператора скрещивания нейронных сетей, *NS1* и *NS2*, а получающиеся в результате его

применения – *NS3* и *NS4*. Опишем их устройство. Обозначим *NS(i)* нейрон с номером *i* сети *NS* (рис. 17). Для нейронов с номерами *NS3(i)* и *NS4(i)* возможны два варианта:

- *NS3(i) = NS1(i)* и *NS4(i) = NS2(i)*;
- *NS3(i) = NS2(i)* и *NS4(i) = NS1(i)*.

Скрещивание нейронных сетей проиллюстрировано на рис. 21.

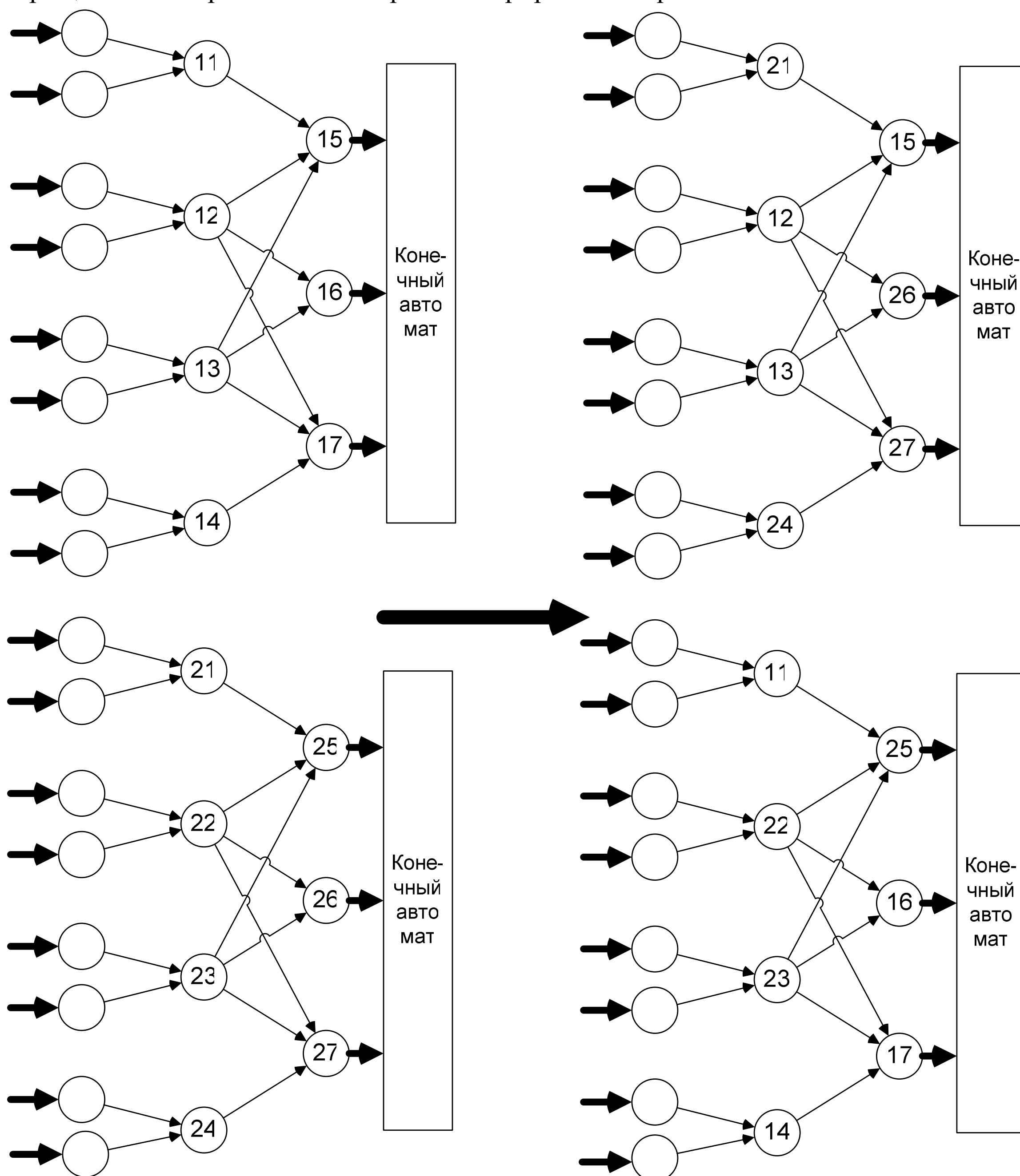


Рис. 21. Скрещивание нейронных сетей

1.1.3.12. Формирование следующего поколения

В качестве основной стратегии формирования следующего поколения используется элитизм. При обработке текущего поколения отбрасываются все особи, кроме нескольких наиболее приспособленных. Доля выживающих особей постоянна для каждого поколения и является одним из параметров алгоритма.

Эти особи переходят в следующее поколение. После этого оно дополняется до требуемого размера следующим образом: пока оно не заполнено выбираются две особи из текущего поколения, и они с некоторой вероятностью скрещиваются или мутируют. Обе особи, полученные в результате мутации или скрещивания, добавляются в новое поколение.

Кроме этого, если на протяжении достаточно большого числа поколений не происходит увеличения приспособленности, то применяются «малая» и «большая» мутации поколения. При «малой» мутации поколения ко всем особям, кроме 10% лучших, применяется оператор мутации. При «большой» мутации каждая особь либо мутирует, либо заменяется на случайно сгенерированную.

Число поколений до «малой» и «большой» мутации постоянно во время работы алгоритма, но может быть различным для разных его запусков.

1.2. Прототипы программных средств

В настоящем разделе описаны прототипы программных средств, разработанных для поддержки указанных методов представления автоматов в виде хромосом генетических алгоритмов.

Инструментальное средство *3Genetic* (разд. 1.2.3) является наиболее мощным из них. По результатом экспериментальных исследований, проведенных на третьем этапе указанной инструментальное средство было обновлено, что потребовало корректировки программной документации.

1.2.1. Программное средство *GAAP*

Разработанный метод генетического программирования для генерации автоматов, представленных сокращенными таблицами переходов, реализован в рамках проекта с открытым исходным кодом. Разработанное программное средство на английском языке имеет аббревиатуру – *GAAP* (Genetic Algorithms for Automata-based Programming).

Это программное средство позволяет генерировать управляющие автоматы. Входными данными являются: виртуальная модель среды, в которой должен работать объект управления, и функция приспособленности, оценивающая работу автомата.

Программное средство *GAAP* разработано на языке *Java* с использованием автоматизированного средства сборки *Maven* (http://maven.apache.org/) и техники *TDD* (test-driven development – разработка через тестирование). Для тестирования компонентов системы используется библиотека *TestNG* (http://testng.org/). Для реализации распределенных вычислений – *GridGain* (http://www.gridgain.com/). Используются некоторые модули из *Spring Framework* (http://www.springframework.org/) для конфигурирования компонентов системы с помощью *XML*. Это позволяет производить настройку компонентов графического интерфейса и генетического алгоритма без модификации исходного кода, конфигурация может содержать фрагменты на скриптовом языке – *Groovy* (http://groovy.codehaus.org/).

Прототип программного средства включает в себя восемь модулей, описание которых приведено в табл. 1.

Таблица 1. Описание модулей инструментального средства *GAAP*

Модуль	Описание
`ga-core`	Набор базовых интерфейсов и реализации различных эволюционных алгоритмов.
`ga-gui`	Набор компонентов, позволяющих визуализировать работу генетического алгоритма в виде графиков и таблиц.
`ga-func`	Бинарное и вещественное кодирование. Генетические операторы для задач оптимизации функций многих переменных.
`ga-grid`	Компоненты для реализации распределенных вычислений функции приспособленности и распределенной модели генетического алгоритма.
`fsm-core`	Различные представления конечных автоматов в виде хромосом и генетические операторы. Реализация модели «конечный автомат + объект управления = автоматизированный объект», позволяющая конечному автомату воздействовать на объект управления.
`fsm-gui`	Набор компонентов, позволяющих визуализировать процесс запуска автоматизированного объекта и состояние объекта управления.
`examples`	Реализации объектов управления для тестовых задач.

Программное средство *GAAP* обладает следующими достоинствами:

- программные интерфейсы просты для использования;
- допускается использование отдельных модулей инструментального средства из приложений, написанных на языке *Java*;
- инструментальное средство содержит реализацию различных способов представления автоматов в виде хромосом;
- инструментальное средство содержит набор компонентов (визуализации и анализа) для исследования работы генетического алгоритма. На рис. 22 приведен пример визуализации работы генетического алгоритма;
- инструментальное средство содержит компоненты для реализации распределенных вычислений функции приспособленности, что часто является наиболее затратной по времени частью генетического алгоритма. На рис. 23 приведена схема работы генетического алгоритма, использующего распределенные вычисления функции приспособленности.

Эволюционный алгоритм задается реализацией следующих базовых интерфейсов системы. Диаграмма классов и интерфейсов изображена на рис. 24.

Здесь:

- *Ga* – интерфейс для генетических алгоритмов;
- *Operator* – интерфейс для генетических операторов, таких как мутация, скрещивание, переупорядочение и т.д.;
- *Selector* – интерфейс для алгоритмов отбора особей из популяции;
- *Population* – интерфейс для популяций генетического алгоритма;
- *Individual* – интерфейс для особей генетического алгоритма;
- *Tester* – интерфейс для вычислителей функции приспособленности;
- *FitnessFunction* – интерфейс для функций приспособленности;
- *Breeder* – интерфейс для алгоритмов построения популяций.

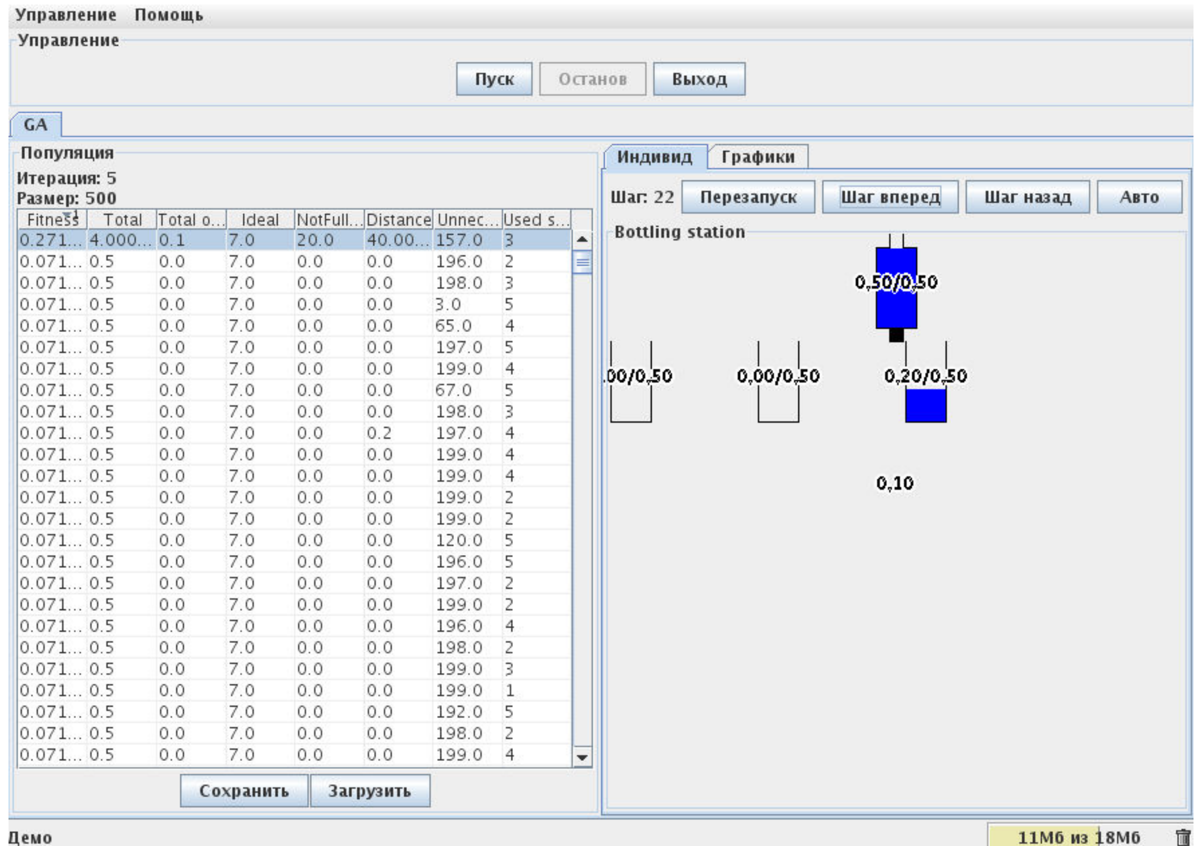


Рис. 22. Пример визуализации работы генетического алгоритма

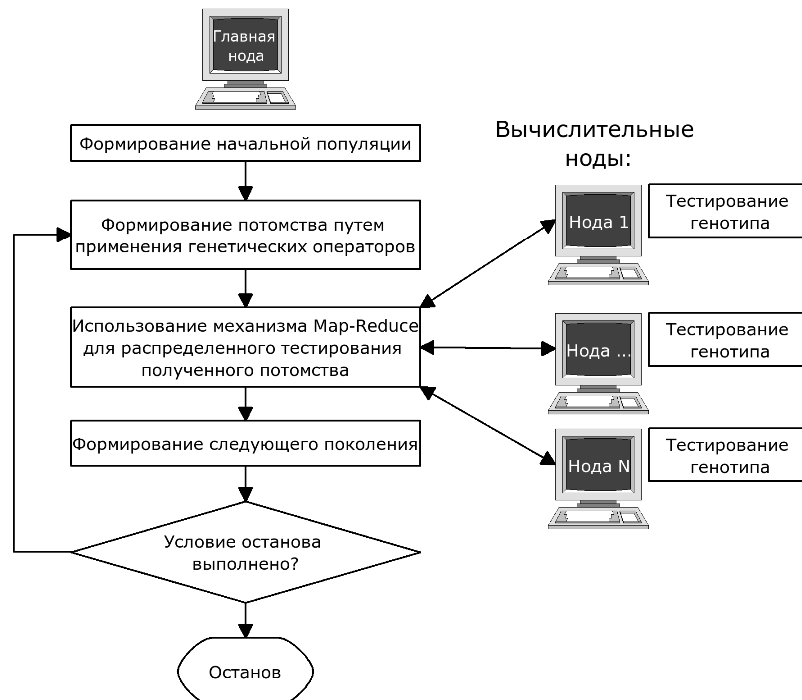


Рис. 23. Схема работы генетического алгоритма

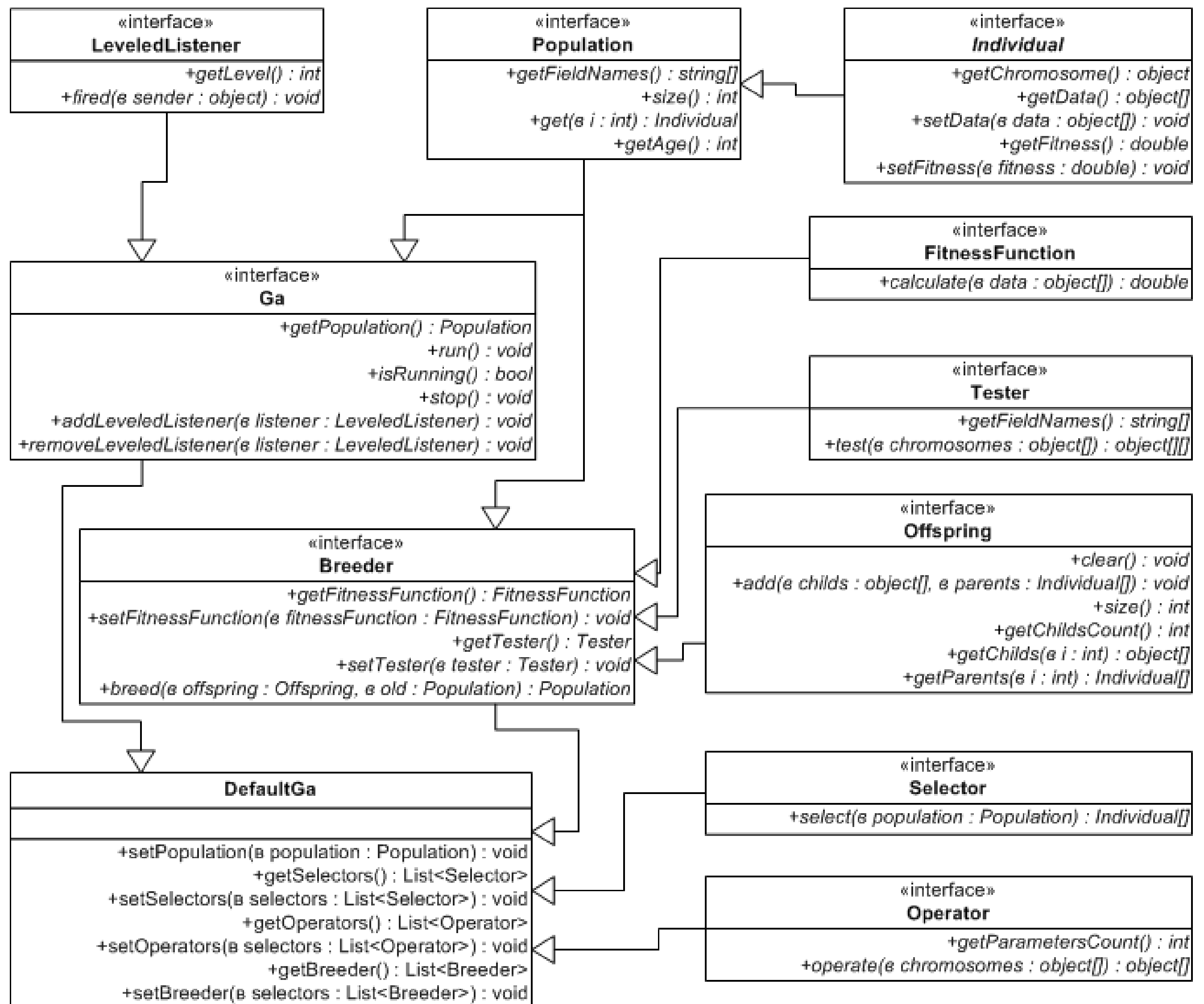


Рис. 24. Диаграмма классов и интерфейсов инструментального средства *GAAP*

Инструментальное средство содержит стандартные реализации данных интерфейсов, имена классов со стандартными реализациями обычно имеют префикс *Default*. Также инструментальное средство включает в себя набор различных алгоритмов отбора особей из популяции, например отбор по рулетке, отбор усечением и т.д.

Инструментальное средство содержит реализации следующих представлений автоматов в виде хромосом:

- полными таблицами переходов – `org.gaap.fsm.impl.simple`;
- сокращенными таблицами переходов – `org.gaap.fsm.impl.compact`;
- деревьями решений – `org.gaap.fsm.impl.tree`.

1.2.2. Программное средство *AutoAnt*

Метод генетического программирования для генерации автоматов, представленных деревьями решений, реализован в рамках проекта *AutoAnt* – программного средства для разработки генетических алгоритмов. Фреймворк обладает следующими достоинствами:

- программные интерфейсы просты для использования;
- допускается использование программного средства из приложений, написанных на языке *Java*;
- программное средство содержит реализацию различных эволюционных алгоритмов для решения исследуемых задач.

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

На рис. 25 приведена диаграмма основных классов программного средства. Приведенные классы находятся в пакете *ru.ifmo.ctddev.autoant.ga*.

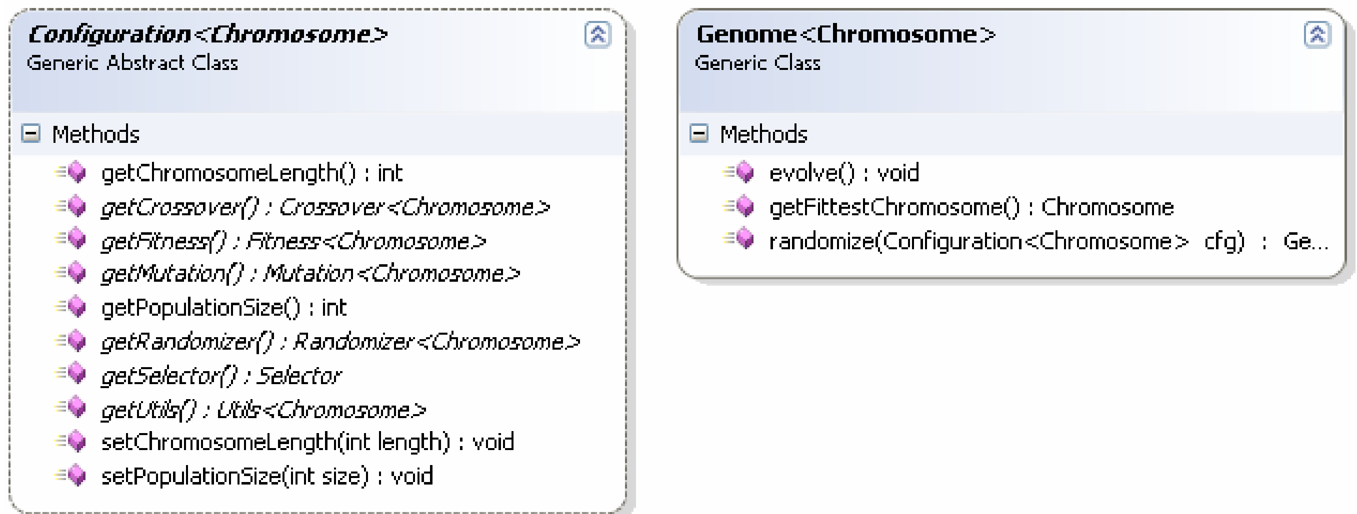


Рис. 25. Диаграмма основных классов фреймворка *AutoAnt*

Эволюционный алгоритм задается реализацией двух классов:

- *Configuration* — класс конфигурации эволюционного алгоритма. Параметром является класс, представляющий хромосому.
- *Genome* — класс, представляющий популяцию особей.

Эти классы являются *Generic*-классами, что позволяет варьировать способ представления и тем самым реализовывать различные варианты эволюционных алгоритмов.

Опишем основные методы класса *Configuration*:

- *getChromosomeLength()* – возвращает размер хромосомы;
- *getCrossover()* – возвращает функтор, реализующий генетический оператор скрещивания;
- *getFitness()* – возвращает функтор, реализующий подсчет значения функции приспособленности хромосомы;
- *getPopulationSize()* – возвращает размер популяции;
- *getRandomizer()* – возвращает функтор, реализующий генетический оператор создания случайной особи;
- *getSelector()* – возвращает функтор, реализующий генетический оператор выбора особей для кроссовера;

- *getUtils()* – возвращает вспомогательный объект, инкапсулирующий функциональность создания пустых хромосом, копирования хромосом и их сравнения.

Опишем основные методы класса *Genome*:

- *randomize(Configuration<Chromosome> configuration)* – создает популяцию случайных особей типа *Chromosome*, настроенную в соответствии с переданной конфигурацией *configuration*;
- *evolve()* – производит шаг эволюционного алгоритма, генерируя следующее поколение особей;
- *getFittestChromosome()* – возвращает хромосому особи популяции, обладающую наибольшим значением функции приспособленности.

Рассмотрим более подробно интерфейсы функторов генетических операторов. Диаграмма интерфейсов генетических операций приведена на рис. 26.

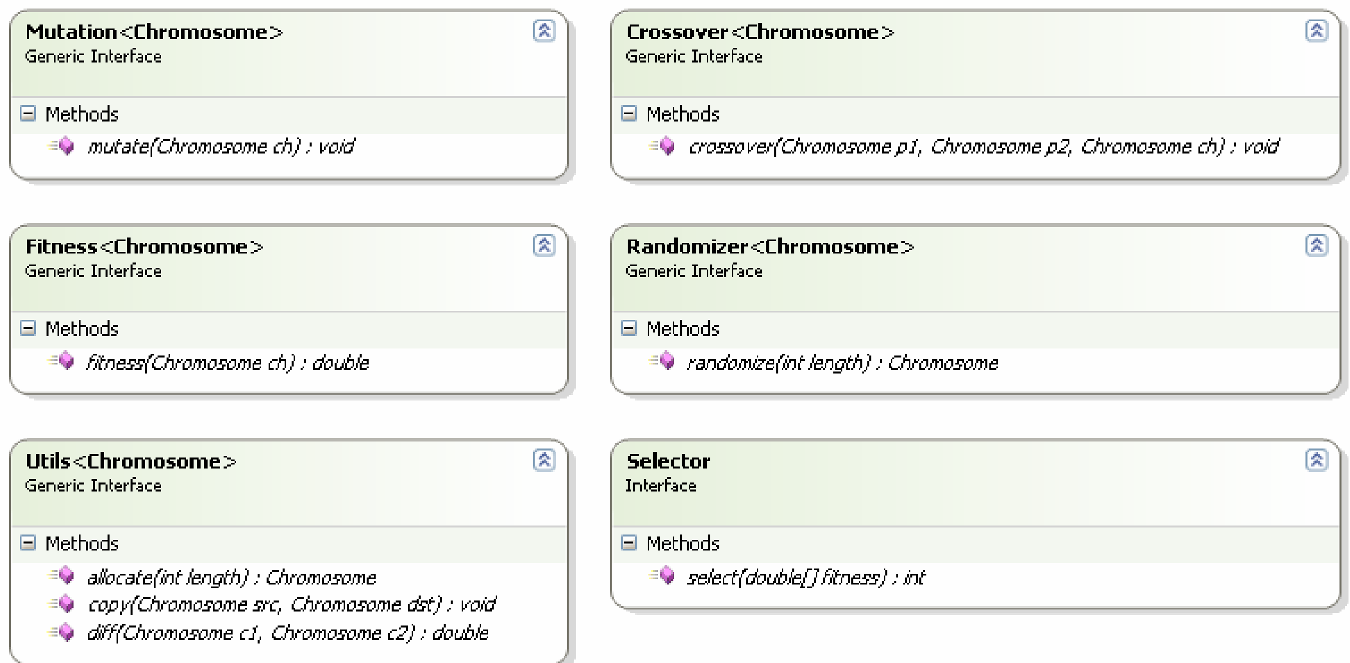


Рис. 26. Интерфейсы генетических операций

Приведем краткую характеристику этих интерфейсов:

- *Mutation* — интерфейс мутации;
- *Crossover* — интерфейс скрещивания, принимающего две особи и записывающего результат в третью;
- *Fitness* — интерфейс фитнесс-функции;
- *Randomizer* — интерфейс создания случайной особи;
- *Utils* — интерфейс утилит, позволяющих копирование особей, их сравнение и создание пустых хромосом.
- *Selector* — интерфейс отбора, выбирающего особь для размножения по набору значений фитнесс-функции.

Для реализации собственного эволюционного алгоритма необходимо выполнить следующие шаги:

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

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

Программное средство содержит готовые реализации следующих алгоритмов генерации автоматов:

- генетические алгоритмы над битовыми строками;
- генетические алгоритмы над таблицами переходов;
- метод генетического программирования, использующий представление автоматов с использованием деревьев решений.

Реализация всех перечисленных методов осуществляет шаг генетического алгоритма следующим образом:

- производится измерение функции приспособленности;
- производится отбор особей для размножения. В качестве стратегии используется элитизм: выбирается определенная доля особей, имеющая максимальное значение функции приспособленности. Конкретное значение коэффициента отбора является настраиваемым параметром;
- все отобранные особи переходят в следующее поколение;
- пока в следующем поколении недостаточно особей, две случайных особи, отобранных для размножения, скрещиваются. После этого, с определенной вероятностью к порожденной особи применяется операция мутации. Особь, полученная в результате, переходит в следующее поколение. Конкретное значение вероятности мутации является настраиваемым параметром.

1.2.2.1. Использование программных интерфейсов *AutoAnt*

Рассмотрим использование программного средства для генерации автоматов, представленных деревьями решений. Диаграмма классов, представляющих соответствующий программный интерфейс, приведена на рис. 27. Все изображенные классы находятся в пакете *ru.ifmo.ctddev.autoant.ga.dectree*.

Класс *TreeConfiguration* является реализацией конфигурации алгоритма генетического программирования для построения автоматов, представленных деревьями решений. Этот класс является *generic*-классом, параметром которого является класс выходных воздействий автомата.

Класс *TreeAutomata* соответствует автомату, представленному деревьями решений. Класс является сериализуемым (реализует интерфейс *java.lang.Serializable*), что позволяет сохранять и читать его из файла, а также передавать по сети. Его метод *step(boolean[] predicates)* осуществляет шаг автомата по значению входных переменных и возвращает выходное действие. Метод *getHeights()* возвращает массив, в котором для каждого состояния записана высота соответствующего дерева решений, метод *getLeafsCount()* – массив, в котором для каждого состояния записано число листьев в соответствующем дереве решений.

Класс *Environment* соответствует настройкам реализации генетического алгоритма. Методы класса позволяют устанавливать и получать значения параметров.

Опишем порядок применения программных интерфейсов для генерации автоматов. Для этого программисту необходимо выполнить следующие шаги:

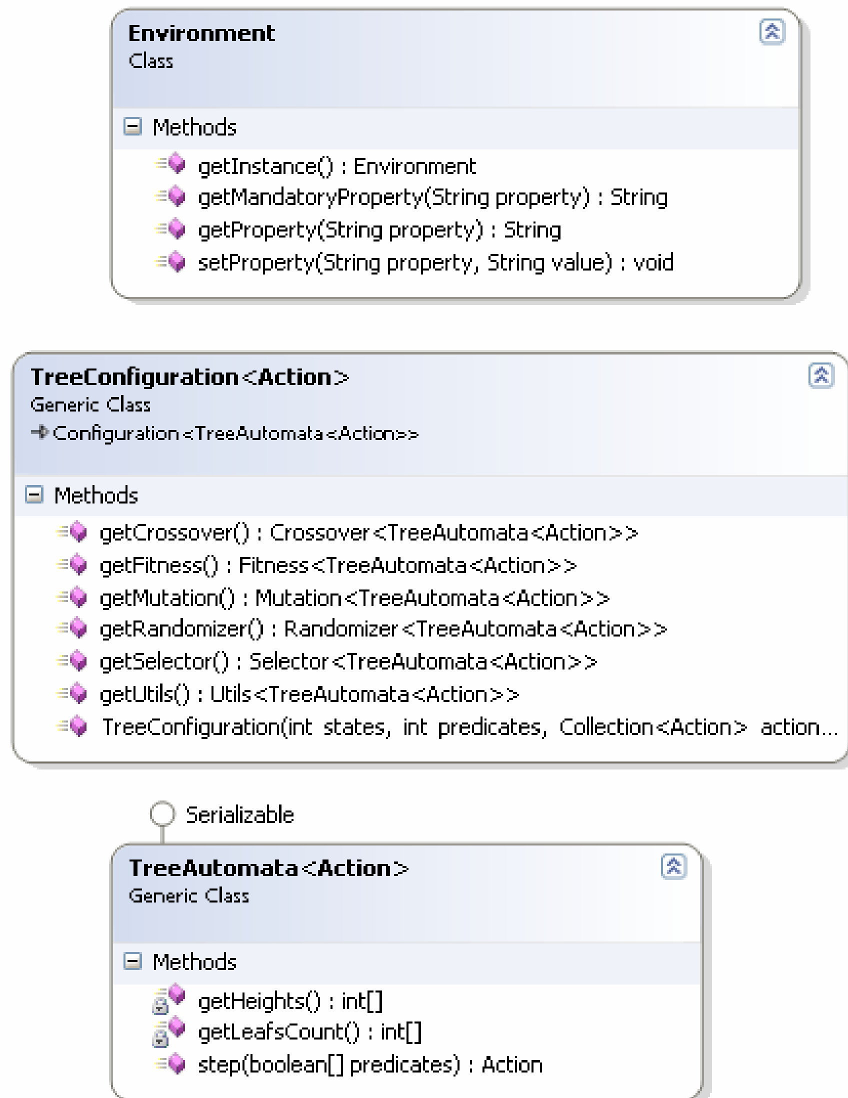


Рис. 27. Диаграмма классов API фреймворка *AutoAnt* для генерации автоматов

- реализовать класс выходных действий автомата *Action*, который требуется сгенерировать;
- реализовать функтор подсчета функции приспособленности, реализующий интерфейс *Fitness<TreeAutomata<Action>>*.

Опишем сценарий использования *AutoAnt* из программного кода. Для генерации автоматов, соответствующих поставленной задаче, требуется выполнить следующие действия:

- создать конфигурацию алгоритма генетического программирования, используя конструктор *TreeConfiguration<Action>*, указав в параметрах количество состояний, количество входных переменных, список возможных выходных действий и реализованную на предыдущем этапе функцию приспособленности;
- установить настройки с помощью метода *setProperty* класса *Environment*;
- создать популяцию автоматов, представленных деревьями решений, с помощью вызова метода *randomize* класса *Genome*, указав в параметре созданную конфигурацию;

- пока не достигнут критерий останова, выполнять шаг алгоритма генетического программирования с помощью вызова метода *evolve*;
- получить сгенерированный автомат с помощью метода *getFittestChromosome*.

Приведем шаблон описанного сценария.

```
    TreeConfiguration<Action> cfg = new
TreeConfiguration<Action>(states, predicates, actions, fitness);

    Environment.getInstance().setProperty(Environment.POPULATION_SIZE,
populationSize);

    Genome<TreeAutomata<Action>> genome = Genome.randomize(cfg);
    while (не выполняется критерий останова) {
       genome.evolve();
    }

    TreeAutomata<Action> result = genome.getFittestChromosome();
```

1.2.3. Программное средство *3Genetic*

Прототип программного средства *3Genetic* разработан с использованием языка программирования *Java*. Это программное средство имеет модульную архитектуру – оно содержит ядро, предоставляющее базовую функциональность, которая расширяется за счет подключаемых модулей (плагинов).

Ядро программы позволяет просматривать и сохранять в виде файлов графики зависимости значений некоторых функций (например, максимального, минимального и среднего значения функции приспособленности особей поколения) от номера поколения. Кроме этого поддерживается возможность визуализации особей и их сохранения в текстовом формате. Также ядро программы поддерживает подключение плагинов налету (во время работы программы).

В качестве подключаемых модулей выступают:

- функции, графики которых строятся ядром программы;
- реализации различных генетических алгоритмов;
- реализации различных представлений особей и операций скрещивания и мутации для генетических алгоритмов;
- визуализаторы особей и поведения задаваемых ими автоматов (они зависят от конкретной задачи и от конкретного представления особи).

При этом отметим, что генетические алгоритмы и особи в некотором смысле независимы – предполагается, что любой генетический алгоритм может работать с любым представлением особи.

В состав программного средства входят следующие модули:

- функции, возвращающее максимальное и среднее значение функции приспособленности особей в поколении;
- классический и островной генетические алгоритмы [26];
- представления хромосомы в виде полных таблиц переходов для одиночных автоматов Мили и Мура, а также для систем автоматов Мили;
- визуализатор, моделирующий работу автомата в задаче «Умный муравей» [1, 13, 25].

При необходимости для решения конкретной задачи этот набор подключаемых модулей может быть расширен.

На основании результатов экспериментов, проведенных на третьем этапе работ, и в соответствии с задачами четвертого этапа работы в программное средство были внесены следующие изменения:

- программное средство получило возможность генерировать автоматы управления не только для модели беспилотного летательного аппарата, рассмотренной в рамках работ по третьему этапу, но для других систем со сложным поведением. Функция приспособленности для оценки автоматов управления при этом должна задаваться пользователем программного средства;
- в программное средство добавлена реализация метода совместного применения конечных автоматов и нейронных сетей.

Кроме этого, в программное средство были внесены следующие доработки:

- оно получило название *3Genetic*;
- в программное средство был добавлен графический интерфейс для выбора и настройки генетического алгоритма;
- в программное средство добавлен графический интерфейс для просмотра графиков зависимости максимального и среднего значения функции приспособленности от номера поколения.

Указанные изменения потребовали корректировки программной документации.

1.2.3.1. Ядро программного средства *3Genetic*

Возможности и, соответственно, элементы интерфейса ядра программного средства *3Genetic* можно разделить на две группы: выбор используемого генетического алгоритма, и просмотр результатов его работы. К первой группе относятся:

- выбор алгоритма (из встроенных или реализованных пользователем);
- выбор типа особи;
- выбор способа представления особи, соответствующего выбранному типу особи.

Ко второй группе относятся:

- просмотр графиков зависимости значения функции приспособленности от номера поколения (наибольшего значения и среднего значения по поколению);
- сохранение графиков в формате `PNG`;
- просмотр особей текущего поколения и эмуляция действий системы со сложным поведением под их управлением;
- сохранение 50 лучших особей поколения;
- просмотр лучших особей (на протяжении всей работы алгоритма) и эмуляция действий системы со сложным поведением под их управлением.

1.2.3.2. Работа с модулями

В этом разделе обсуждается создание и подключение к ядру программного средства *3Genetic* модулей. Программное средство *3Genetic*, как отмечено выше, написано на языке программирования *Java*, поэтому подключаемые модули, это *JAR*-архивы (**J**ava **Ar**chive), помещенные в определенные директории и содержащие определенные *Java*-классы. Все упоминаемые ниже интерфейсы содержатся в файле `common.jar`. Некоторые необязательные, но, возможно, полезные, при написании собственного модуля классы, содержатся в файле `util.jar`.

1.2.3.3. Модуль, содержащий представление особи

Особь – это все то, с чем может работать генетический алгоритм – сущность, для которой определены операторы кроссовера и мутации и фитнесс-функция. В терминах средства *3Genetic* – это экземпляр класса реализующего интерфейс `Individual`:

```
public interface Individual extends Comparable<Individual> {

    public double fitness();
    public Individual mutate(Random r);
    public Individual[] crossover(Individual p, Random r);

}
```

Метод `toString()` используется при сохранении особи в файл.

Для подключения модуля к ядру *3Genetic* *JAR*-файл с реализацией особи должен быть помещен в директорию `individuals`. В манифесте архива должны присутствовать следующие параметры: `Main-Class`, `Extension-Name`, `Comment`. `Main-Class` – это имя класса, реализующего интерфейс `Loader<IndividualFactory>`:

```
public interface Loader<Plugin> {

    public Plugin load(Object... args);
    Properties getProperties();

}
```

`IndividualFactory` – это следующий интерфейс:

```
public interface IndividualFactory {
    public Individual randomIndividual();
}
```

Возвращаемые значения метода `load` (в данном случае он вызывается без параметров), загруженного `Loader'`ом экземпляра класса, реализующего интерфейс `IndividualFactory`, передаются системой генетическому алгоритму. Метод `getProperties()` возвращает свойства модуля, настраиваемые во время работы программного средства *3Genetic*. Класс, реализующий интерфейс `Loader`, должен иметь конструктор от одного аргумента типа `JarFile` (собственно *JAR*-файла, содержащего данный модуль).

Параметр `Extension-Name` содержит имя особи, отображаемое в диалоге выбора особи (Рис. 34). Параметр `Comment` должен иметь вид «`arg1 arg2` ||| комментарий, отображаемый в диалоге выбора особи». Здесь `arg1` и `arg2` – параметры, необходимые для рисования графиков, максимальное значение и число знаков после десятичной точки, отображаемых в значении функции приспособленности.

1.2.3.4. Модуль, содержащий генетический алгоритм

Генетический алгоритм в программном средстве *3Genetic* – это экземпляр класса *Java* реализующего интерфейс генетического алгоритма:

```
public interface GA {

    public List<Individual> getGeneration();
    public void nextGeneration();
    public void bigMutation();
    public Individual getBest();

}
```

Приведем описание методов, входящих в этот интерфейс:

- `getGenration` – возвращает текущее поколение;
- `nextGeneration` – переход к следующему поколению;
- `bigMutation` – вызов большой мутации поколения;
- `getBest` – возвращает лучшую особь на данный момент.

Для подключения модуля к ядру программного средства *3Genetic JAR*-файл с реализацией генетического алгоритма должен быть помещен в директорию `gas`. В манифесте архива должны быть следующие параметры: `Main-Class`, `Extension-Name`, `Comment`.

Параметр `Main-Class` – это имя класса, реализующего интерфейс `Loader<GA>`. В данном случае метод `load` будет принимать один параметр – экземпляр `IndividualFactory`. Авторам неизвестны реализации генетических алгоритмов, которые не используют в своей работе (хотя бы при создании начального поколения) случайно сгенерированные особи. Параметры `Extension-Name` и `Comment` используются при отображении в диалоге выбора алгоритма.

1.2.3.5. Модуль, содержащий визуализатор особи

Визуализатор особи – это экземпляр класса `java.awt.Frame`. Для подключения модуля к ядру *3Genetic JAR*-файл с реализацией визуализатора должен быть помещен в директорию *emulators*. В манифесте архива должны быть следующие параметры: `Main-Class`, `Extension-Name`, `Comment`. `Main-Class` – имя класса, реализующего интерфейс `Loader<Frame>`. В данном случае метод `load` будет принимать одни параметр – возвращаемое значение метода `getAttributes()` экземпляра `Visulaizible`.

```
        public interface Visualizable extends Individual {

    public Object[] getAttributes();

}
```

Параметр `Comment` должен иметь вид «`arg1 arg2` …||| комментарий, отображаемый в диалоге выбора визуализатора»: `arg1`, `arg2`, … – имена поддерживаемых особей.

Параметры `Extension-Name` и также используются при отображении в диалоге выбора визуализатора.

1.2.3.6. Модуль, содержащий визуализируемую функцию

Визуализируемая функция – это экземпляр класса реализующего интерфейс `Functor`:

```
public interface Functor {

    public List<Double> getValues();
    public void update(List<Individual> generation);
    public void clear();

}
```

Опишем значение методов этого интерфейса:

- `getValues` – метод, возвращающий список значений на каждое поколение;
- `update` – обновление текущего списка значений;
- `clear` – очистка списка.

Для подключения модуля к ядру программного средства *3Genetic JAR*-файл с реализацией визуализатора должен быть помещен в директорию `functors`. В манифесте архива должны быть следующие параметры: `Main-Class`, `Comment`.

Параметр `Main-Class` задает имя главного класса. Он должен иметь конструктор без параметров. Параметр `Comment` отображается как название графика.

1.3. Рекомендации по применению генетических алгоритмов для построения автоматов управления системами со сложным поведением

Выбор представления автоматов в виде хромосомы алгоритма генетического программирования должен осуществляться на основе следующих критериев:

- характера входных переменных;
- степени достоверности предположения о том, что в некоторых состояниях автомата значения некоторых из переменных не важны.

Метод представления автоматов *сокращенными таблицами переходов* целесообразно применять в том случае, когда выполняются следующие условия:

1. Предположение о том, что в каждом из состояний управляющего автомата переход может быть выбран на основе значений небольшого числа значимых входных переменных, верно.
2. Число значимых переменных в каждом состоянии невелико.

Кроме этого, метод сокращенных таблиц переходов обладает следующими свойствами:

- при несовместности некоторых входных переменных некоторые из переходов никогда не будут выполняться. При использовании сокращенных таблиц вероятность такого события значительно снижается по сравнению с полными таблицами;
- сокращение объема требуемой для работы генетического алгоритма памяти, а также некоторое ускорение его работы.

При применении сокращенных таблиц также следует учитывать, что все значимые переменные имеют равный приоритет, так как решение принимается сразу на основе всех переменных, которые используются в данном состоянии.

Функцию приспособленности особи в общем случае рекомендуется вычислять при помощи моделирования (однократного или многократного) поведения системы со сложным поведением в некоторой внешней среде. Эта функция должна оценивать то, насколько эффективно система со сложным поведением выполняет поставленные перед ней задачи.

Программное средство *GAAP* рекомендуется применять для решения задач, в которых вычисление функции приспособленности требует больших вычислительных ресурсов. Это связано с тем, что это программное средство поддерживает возможность распределенного вычисления функции приспособленности на нескольких компьютерах.

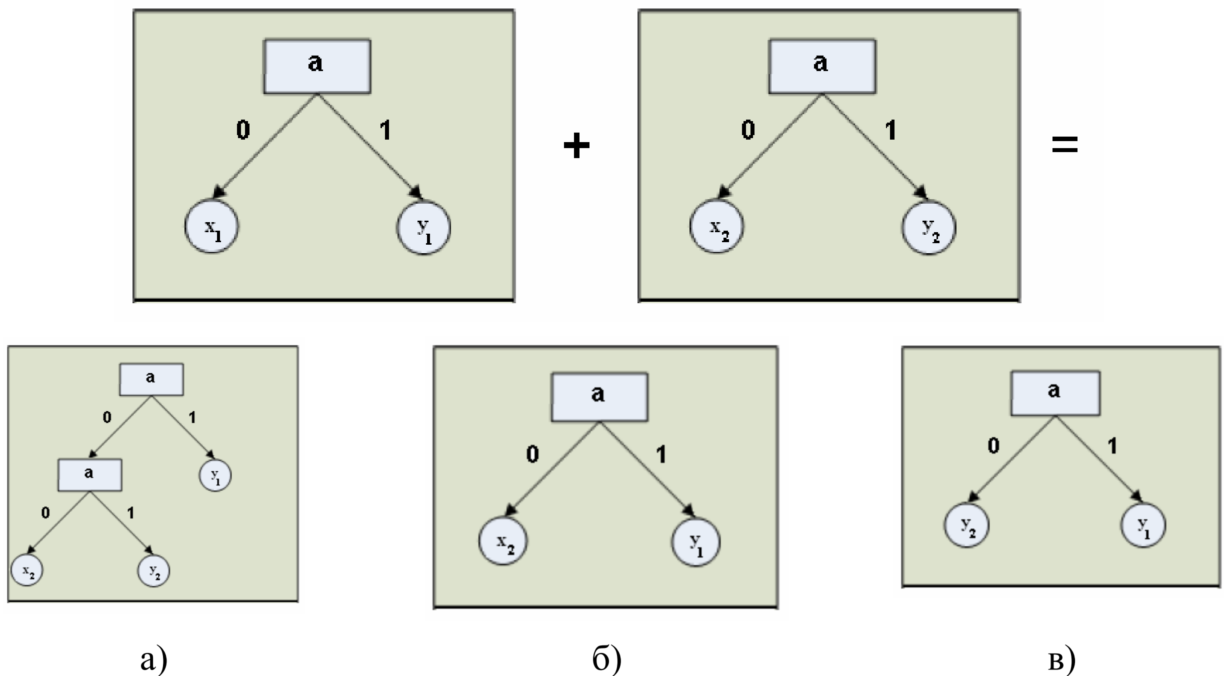
Метод представления автоматов деревьями решений является наиболее эффективным при соблюдении следующих условий:

1. Переходы искомого автомата из фиксированного состояния зависят от малого числа переменных по сравнению общим числом входных переменных автомата.
2. Высота деревьев решений достаточно велика.

При нарушении первого условия деревья решений являются недостаточно выразительными для представления функции переходов. В таком случае, хранение полной таблицы переходов предпочтительнее, так как такое представление требует меньше памяти.

Покажем, чем плохо нарушение второго условия. Для этого рассмотрим скрещивание двух деревьев решений (высота которых равна единице), помеченных одним и той же переменной *a*. Пометим для наглядности хранимые в листьях деревьев значения переменными x_1, y_1, x_2, y_2. При этом *x* — значение функции при $a = 0$, а *y* — при $a = 1$.

Пусть, например, в первом дереве выбран узел, соответствующий переменной x_1. После этого, в зависимости от выбора узла во втором дереве, равновероятно может произойти одна из трех ситуаций: замена узла x_1 вторым деревом либо одним из листьев x_2 и y_2 (рис. 28).



а) б) в)

Рис. 28. Скрещивание деревьев малой высоты

Из рис. 28 видно, что в одном случае из трех (случай в) значение, соответствующее *y*, скопировалось в *x*. При использовании же представления автоматов в виде таблицы переходов, либо битовой строки, такого бы не произошло. Такое копирование ведет к появлению плохого потомства и существенно замедляет генерацию.

Функцию приспособленности особи в общем случае рекомендуется вычислять при помощи моделирования (однократного или многократного) поведения системы со сложным поведением в

некоторой внешней среде. Эта функция должна оценивать то, насколько эффективно система со сложным поведением выполняет поставленные перед ней задачи. Кроме этого, она может содержать слагаемые, соответствующие штрафным функциям. Приведем пример такой штрафной функции.

Выше показано, что для эффективного применения метода представления автоматов деревьями решений необходимо, чтобы их высота была не слишком большой и не слишком маленькой. Покажем, как этого можно добиться с помощью штрафной функции. Эта функция может иметь, например, следующий вид:

$$f(A) = \sum_{q \in Q} g(h_q)\text{, где } g(x) = \begin{cases} -C(a-x), x < a \\ 0, a \le x \le b \\ -C(x-b) \end{cases}$$

Здесь C – некоторая константа, a и b задают рекомендуемые границы высоты деревьев решений, h_q – высота дерева решений в состоянии Q . Программный интерфейс средства *AutoAnt* предоставляет доступ к высотам деревьев решений.

Заметим, что каждый лист дерева решений задает ровно один переход из состояния. Таким образом, можно считать, что число листьев во всех деревьях равно числу переходов автомата, а число листьев в заданном дереве – число переходов из фиксированного состояния. Здесь не учитывается тот факт, что два листа могут задавать один и тот же переход. Вместе с тем, число листьев в дереве является довольно естественной оценкой числа переходов. Значения чисел листьев в деревьях решений также могут быть внесены в функцию приспособленности для сужения области поиска.

Метод совместного применения конечных автоматов и нейронных сетей наиболее эффективен в следующих случаях:

- число входные переменных велико, причем они имеют не только логический, но и вещественный или целый тип;
- формирование вручную входных переменных для системы управления на основе информации о внешней среде затруднено.

В указанных случаях метод совместного применения конечных автоматов и нейронных сетей позволяет автоматически сформировать входные переменные для системы управления.

Функцию приспособленности особи в общем случае рекомендуется вычислять на основе моделирования (однократного или многократного) поведения системы со сложным поведением во внешней среде. Эта функция должна оценивать то, насколько эффективно система со сложным поведением выполняет поставленные перед ней задачи.

Программное средство *3Genetic* рекомендуется применять для решения задач, в которых входные переменные могут иметь вещественный тип, а вычисление функции приспособленности не требует больших временных затрат. Это связано с тем, что это средство поддерживает метод совместного применения конечных автоматов и нейронных сетей, но не поддерживает распределенное вычисление функции приспособленности.

1.4. Выводы

Разработаны предложения и рекомендации по построению автоматов управления системами со сложным поведением на основе следующих методов:

- метод сокращенных таблиц переходов;
- метод представления автоматов деревьями решений;
- метод совместного применения конечных автоматов и нейронных сетей.

2. МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ И УКАЗАНИЯ ПО РАЗРАБОТКЕ СИСТЕМ СО СЛОЖНЫМ ПОВЕДЕНИЕМ С ИСПОЛЬЗОВАНИЕМ РАЗРАБОТАННЫХ МЕТОДОВ

2.1. Установка и настройка программных средств

2.1.1. Установка и настройка программного средства *GAAP*

Программное средство *GAAP* распространяется в виде исходных кодов, доступных в *Subversion* репозитории по адресу https://srv.godin.net.ru/svn/automata/.

2.1.1.1. Компиляция программного средства

Для компиляции исходных кодов программного средства *GAAP* необходимо наличие следующих программных продуктов:

- *Subversion* (http://subversion.tigris.org/);
- *Maven* (http://maven.apache.org/);
- *Java 1.6* (http://www.java.com/ru/).

Для извлечения из репозитория текущей разрабатываемой версии *GAAP* воспользуйтесь следующей командой:

```
svn co https://srv.godin.net.ru/svn/automata/branches/gaap-0.1.1/ gaap
```

Также в репозитории доступна стабильная версия 0.1.1 с помощью команды:

```
svn co https://srv.godin.net.ru/svn/automata/trunk gaap
```

Для того чтобы произвести компиляцию исходных кодов с помощью команды:

```
mvn compile
```

Для сборки *jar*-архивов, включающих все зависимые классы и библиотеки, используется команда:

```
mvn assembly:assembly
```

Для запуска скомпилированного *GAAP* требуется наличие *Java 1.6*, а для запуска распределенных алгоритмов также понадобится *GridGain 1.5*.

2.1.1.2. Добавление представления особи

Для того чтобы добавить реализацию нового представления хромосомы, необходимо добавить одну или несколько реализаций интерфейса *Operator* и один фабричный метод для создания случайных особей. Инструментальное средство содержит два абстрактных класса для упрощения процесса создания новых операторов: *CrossoverAdapter* и *MutationAdapter*.

Опишем реализацию этих классов на примере представления хромосом полными таблицами переходов автомата. В листинге 7 приведена реализация класса `SimpleStateMachine`, который представляет хромосому в виде полной таблицы переходов.

Листинг 7. Реализация класса, представляющего полную таблицу переходов

```
public class SimpleStateMachine implements Serializable {
    protected int actionsCount;
    protected SimpleState[] state;
```

```
    public SimpleStateMachine(int statesCount, int
        actionsCount) {
        this.actionsCount = actionsCount;
        state = new SimpleState[statesCount];
    }
    public SimpleStateMachine(SimpleStateMachine
        automaton) {
        actionsCount = automaton.actionsCount;
        state = new SimpleState[automaton.state.length];
        for (int i = 0; i < state.length; i++) {
            state[i] = new
        SimpleState(automaton.state[i]);
        }
    }
}
public class SimpleState implements Serializable {
    protected int[] target;
    protected int[] action;

    public SimpleState(int len) {
        target = new int[len];
        action = new int[len];
    }

    public SimpleState(SimpleState state) {
        target = Arrays.copyOf(state.target,
        state.target.length);
        action = Arrays.copyOf(state.action,
        state.action.length);
    }
}
```

В листинге 8 приведена реализация классов `SimpleMutation` и `SimpleCrossover`, которые реализуют соответственно мутацию и скрещивание хромосом в виде полных таблиц переходов.

Листинг 8. Реализация операций мутации и скрещивания для полных таблиц переходов

```
public class SimpleMutation extends
        MutationAdapter<SimpleStateMachine> {
    public SimpleStateMachine mutate(SimpleStateMachine a)
        {
        SimpleStateMachine child = new
        SimpleStateMachine(a);
        int num =
        Utils.getRandom().nextInt(a.state.length);
        for (int c = 0; c < num; c++) {
            /* State for mutation: */
            int i =
        Utils.getRandom().nextInt(child.state.length);
            SimpleState state = child.state[i];
            /* Mutation: */
            int r = Utils.getRandom().nextInt(2);
            if (r == 0) {
                int n =
        Utils.getRandom().nextInt(state.target.length);
                state.target[n] =
        Utils.getRandom().nextInt(child.state.length);
            } else {
                int n =
        Utils.getRandom().nextInt(state.target.length);
                state.action[n] =
        Utils.getRandom().nextInt(a.actionsCount);
            }
        }
        return child;
    }
}


public class SimpleCrossover extends
        CrossoverAdapter<SimpleStateMachine> {
    public SimpleStateMachine[]
        crossover(SimpleStateMachine automaton1,
```

```
SimpleStateMachine automaton2) {
SimpleStateMachine a1 = new
SimpleStateMachine(automaton1.state.length,
automaton1.actionsCount);
SimpleStateMachine a2 = new
SimpleStateMachine(automaton1.state.length,
automaton1.actionsCount);
for (int i = 0; i < automaton1.state.length; i++)
{
    /* Crossover states: */
    SimpleState s1 = automaton1.state[i];
    SimpleState s2 = automaton2.state[i];

    SimpleState child1 = new
SimpleState(s1.target.length);
    SimpleState child2 = new
SimpleState(s1.target.length);

    int crossPoint;
    crossPoint =
Utils.getRandom().nextInt(s1.target.length + 1);
    for (int j = 0; j < crossPoint; j++) {
        if (j < crossPoint) {
            child1.target[j] = s1.target[j];
            child2.target[j] = s2.target[j];
        } else {
            child1.target[j] = s2.target[j];
            child2.target[j] = s1.target[j];
        }
    }
    crossPoint =
Utils.getRandom().nextInt(s1.target.length + 1);
    for (int j = 0; j < crossPoint; j++) {
        if (j < crossPoint) {
            child1.action[j] = s1.action[j];
            child2.action[j] = s2.action[j];
```

```
                    } else {
                        child1.action[j] = s2.action[j];
                        child2.action[j] = s1.action[j];
                    }
                }

                a1.state[i] = child1;
                a2.state[i] = child2;
            }
            return new SimpleStateMachine[]{
                    a1,
                    a2
            };
        }
    }
```

В листинге 9 приведена реализация класса фабрики для представления хромосом полными таблицами переходов.

Листинг 9. Реализация класса фабрики для представления хромосом в виде полных таблиц переходов

```
    public class SimpleFactory {
        public static SimpleStateMachine randomn(int
            statesCount, int actionsCount, int
            predicatesCount) {
            SimpleStateMachine res = new
            SimpleStateMachine(statesCount, actionsCount);
            for (int i = 0; i < statesCount; i++) {
                /* Random state: */
                SimpleState state = new SimpleState((int)
            Math.pow(2, predicatesCount));
                for (int j = 0; j < state.target.length; j++)
            {
                    state.target[j] =
            Utils.getRandom().nextInt(statesCount);
                    state.action[j] =
            Utils.getRandom().nextInt(actionsCount);
                }
```

```
                res.state[i] = state;
            }
            return res;
        }

        public static List<SimpleStateMachine> random(int
            statesCount, int actionsCount, int
            predicatesCount, int count) {
            List<SimpleStateMachine> res = new
            ArrayList<SimpleStateMachine>(count);
            for (int i = 0; i < count; i++) {
                res.add(randomn(statesCount, actionsCount,
            predicatesCount));
            }
            return res;
        }
    }
```

2.1.2. Установка и настройка программного средства *AutoAnt*

В настоящем разделе описывается установка и настройка программного средства *AutoAnt*.

2.1.2.1. Описание дистрибутива программного средства

Дистрибутив программного средства *AutoAnt* включает:

- исполняемый *Java*-архив *autoant-ga.jar*;
- командный файл *autoant.cmd*;
- стандартный файл настроек *autoant.properties.default*.

Программное средство поставляется в виде *zip*-архива, содержащего перечисленные файлы.

Программное средство *AutoAnt* используется из командной строки – для запуска генерации автоматов необходимо запустить командный файл *autoant.cmd*.

2.1.2.2. Установка программного средства

Для установки программного средства *AutoAnt* требуется распаковать архив дистрибутива в папку, в которую производится установка программного средства.

2.1.2.3. Настройка программного средства

Настройки, необходимые для работы программного средства, должны быть перечислены в файле *autoant.properties* в следующем формате:

```
property_name=property_value;
```

где *property_name* – свойство, а *property_value* – его значение.

Описание настроек приведено в табл. 2.

Таблица 2. Настройки фреймворка *AutoAnt*

Свойство	Описание	Значение по умолчанию	Обязательное
`population.size`	Размер популяции	200	нет
`target.fitness`	Целевое значение функции приспособленности	–	нет
`states.count`	Число состояний генерируемого автомата	8	нет
`predicates.count`	Число входных переменных	–	да
`populations.max`	Максимальное число популяций	100	нет
`roulette.ratio`	Коэффициент отбора особей для размножения	0.25	нет
`mutation.probability`	Вероятность мутации	0.02	нет
`action.class`	Имя класса выходных состояний автомата. Должен являться перечислимым типом(*enum*) языка *Java*	–	да
`fitness.class`	Имя класса, осуществляющего подсчет функции приспособленности	–	да
`classpath`	*CLASSPATH*, используемый при запуске генерации. Должен содержать пути к классам, указанным в свойствах `action.class` и `fitness.class`.	Текущая директория	нет
`output.file`	Файл, в который должен быть выведен построенный автомат.	auto.out	нет

Генерация будет выполняться до тех пор, пока не будет построен автомат, имеющий функцию приспособленности большую или равную, чем указанная в параметре `target.fitness`, либо пока не будет сгенерировано число популяций, указанное в параметре `max.populations`.

Возможные сообщения об ошибке и методы устранения соответствующих ошибок приведены в табл. 3.

Таблица 3. Возможные сообщения об ошибках программного средства *Autoant*

Сообщение об ошибке	Возможная причина	Метод устранения
`Mandatory property <Property> not defined`	Не указано значение обязательного свойства `<Property>`.	Указать значение свойства `<Property>` в файле настроек.
`System could not find the path specified: <Path>`	Путь, указанный в свойстве `classpath`, не существует.	Проверить путь, указанный в свойстве `classpath`.
`Could not load fitness functor class.`	Путь к классу, указанному в свойстве `fitness.class`, не указан в свойстве `classpath`.	Добавить путь к классу, указанному в свойстве `fitness.class`, в свойство `classpath`.
	Неверно указан класс, подсчитывающий функцию приспособленности.	Проверить свойство `fitness.class.`
`Could not load action class.`	Путь к классу, указанному в свойстве `action.class`, не указан в свойстве `classpath`.	Добавить путь к классу, указанному в свойстве `action.class`, в свойство `classpath`.
	Неверно указан класс выходных действий автомата.	Проверить свойство `action.class.`
`Action class is not enumerable`	Класс выходных действий автомата не является перечислимым.	Сделать класс выходных действий автомата перечислимым типом языка *Java* (*enum*).

2.1.3. Установка и настройка программного средства *3Genetic*

В настоящем разделе описана установка и настройка программного средства *3Genetic*.

2.1.3.1. Описание дистрибутива программного средства *3Genetic*

Программное средство *3Genetic* написано на языке программирования *Java*, поэтому для его работы должна быть установлена *Java Runtime Environment* (*JRE*) версии не ниже 1.6.0. Для установки *3Genetic* необходимо распаковать архив *3genetic.zip* в любую директорию. Для запуска программы в *MS Windows* надо воспользоваться пакетным файлом `3genetic.bat`, для *UNIX-like* систем – файлом `3genetic.sh`. Переменная окружения `PATH` должна содержать путь к исполняемому файлу `java`.

Рассмотрим подробнее структуру проекта. Кроме указанных файлов программное средство содержит пять директорий, четыре *JAR*-архива и файл `build.properties` со служебной информацией (рис. 29). Файлы директории `conf` (за исключением `field`) отвечают за внешний вид *3Genetic* (размеры элементов интерфейса, надписи и подсказки, и т.д.) и могут быть отредактированы с соблюдением формата описанного здесь http://java.sun.com/j2se/1.4.2/docs/api/java/util/Properties.html - load(java.io.InputStream).

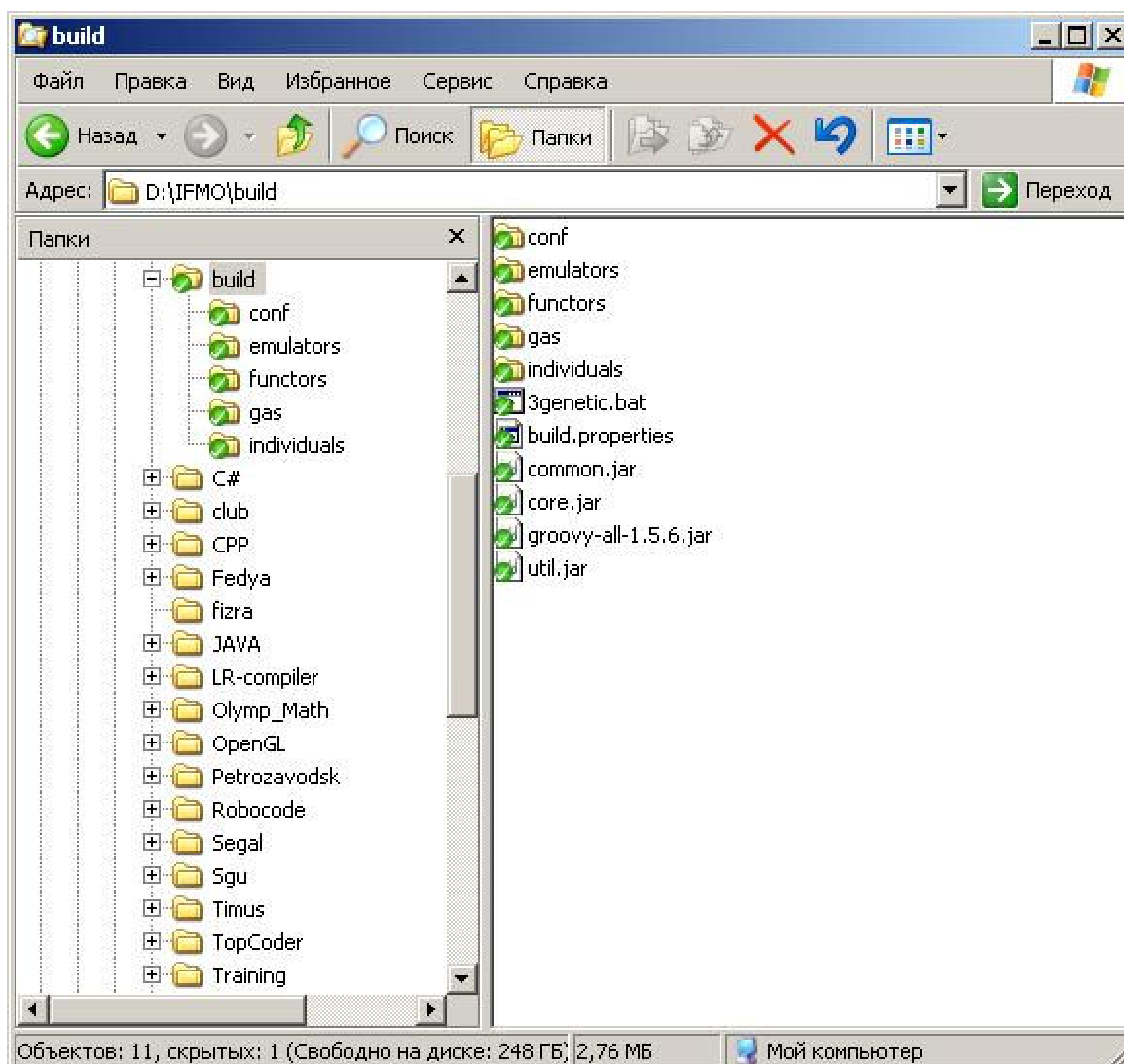


Рис. 29. Структура каталогов проекта

В директориях `emulators`, `functors`, `gas`, `individuals` находятся подключаемые модули, работа с которыми описана ниже.

Отметим, что в этих папках не должно находиться некорректных *jar*-файлов – любой *jar*-файл, находящийся в одной из этих папок, должен быть соответствующим встраиваемым модулем. В противном случае программа будет работать неверно.

JAR-архив `core.jar` содержит классы, реализующие основную функциональность *3Genetic*. Архивы `common.jar` и `util.jar` содержат соответственно интерфейсы и классы, необходимые для создания собственных модулей.

2.1.3.2. Настройка программного средства 3Genetic

После запуска программного средства *3Genetic* открывается окно (рис. 30) с вкладками, на которых изображены графики, получаемые с помощью модулей, находящихся в папке `functors`.

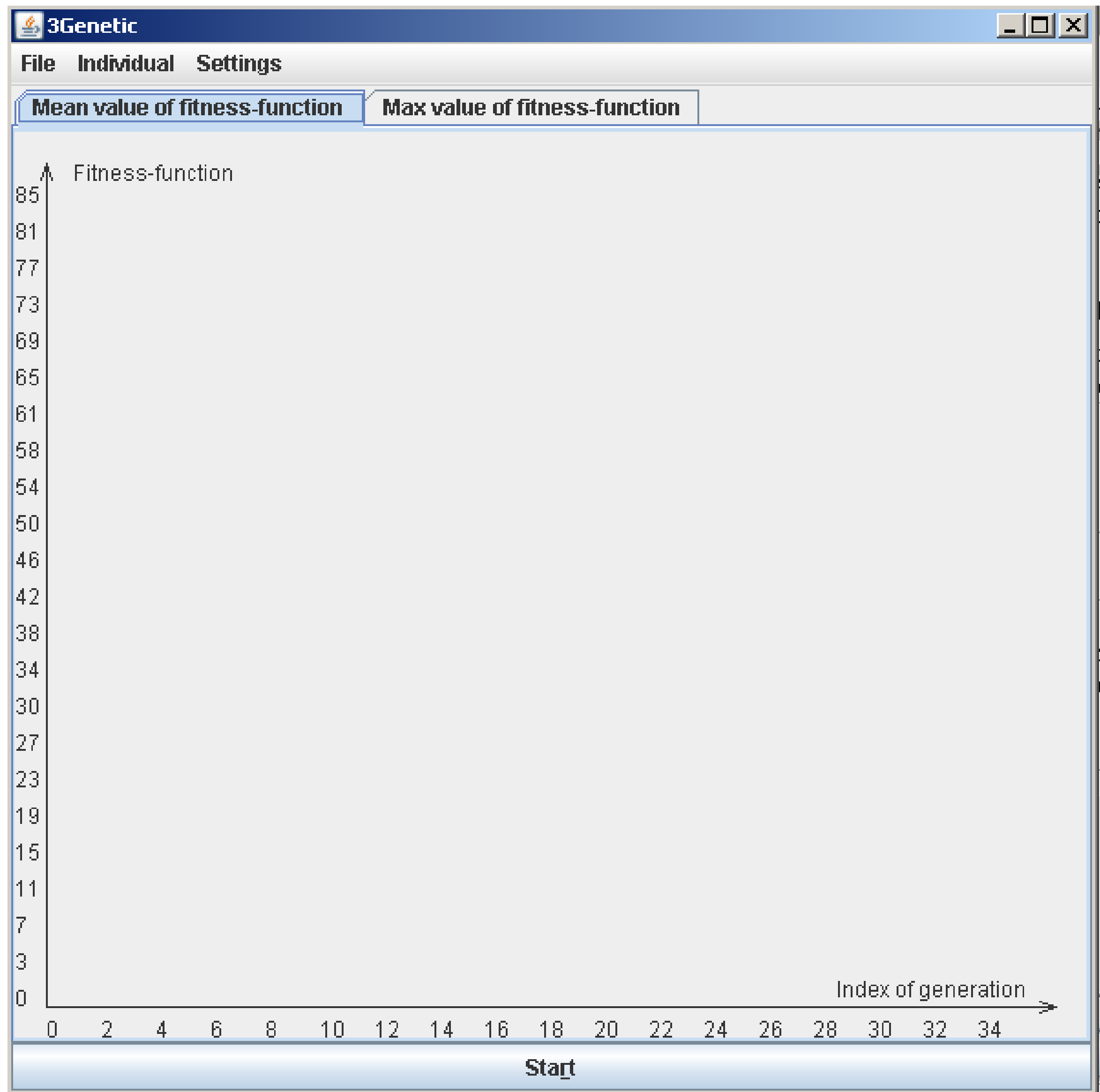


Рис. 30. Стартовое окно

Внизу окна находится кнопка *Start*, при нажатии, на которую происходит запуск алгоритма. После запуска вместо кнопки *Start*, появляется кнопка *Pause* (рис. 31). При нажатии эту кнопку программа приостанавливает свою работу. Отметим, что остановка может произойти не сразу, а лишь спустя некоторое время.

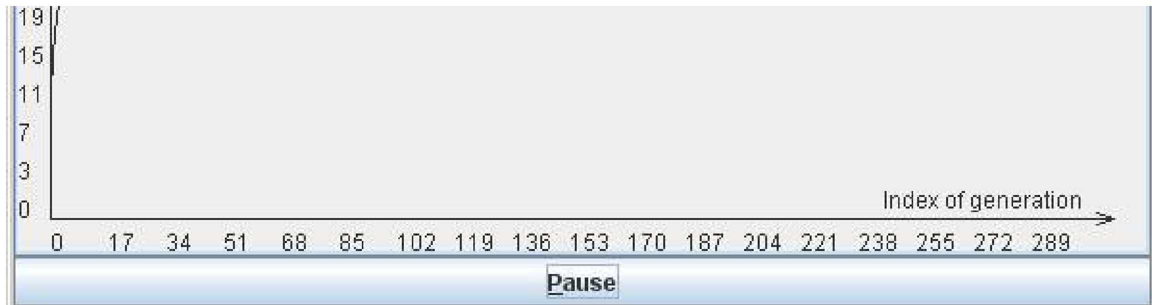


Рис. 31. Кнопка остановки программы

Рассмотрим возможности настройки программы, для этого необходимо нажать *Settings* в верхней части главного окна (рис. 32).

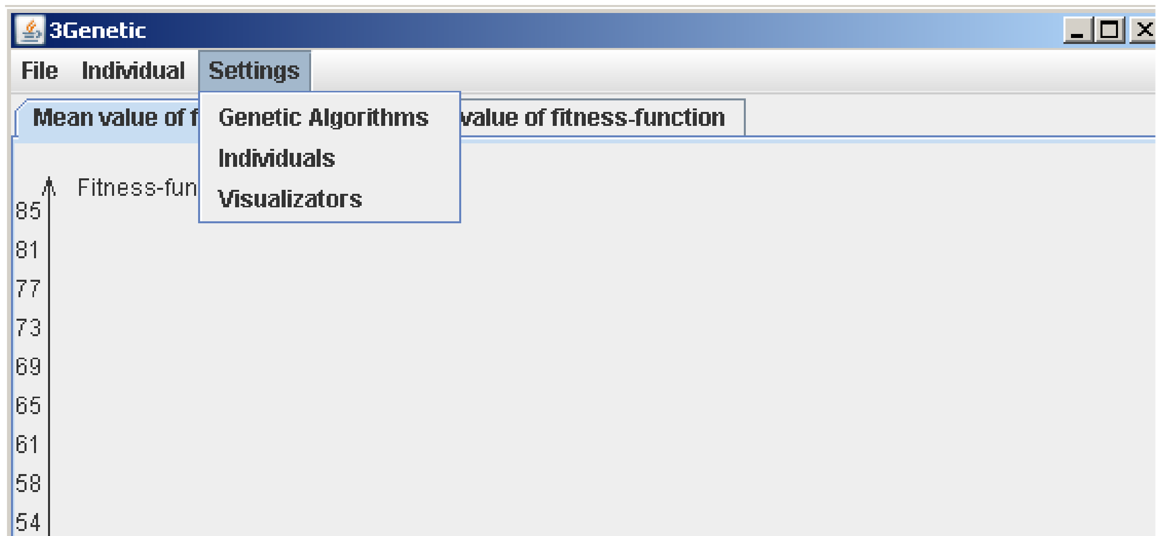


Рис. 32. Настройки программы

При выборе пункта *Genetic Algorithm*, появляется окно выбора генетического алгоритма (рис. 33). При этом в списке отображаются все алгоритмы, находящиеся в папке *gas*. Кнопка *OK* предназначена для подтверждения выбора. По кнопке *Config* открывается окно настройки параметров алгоритма.

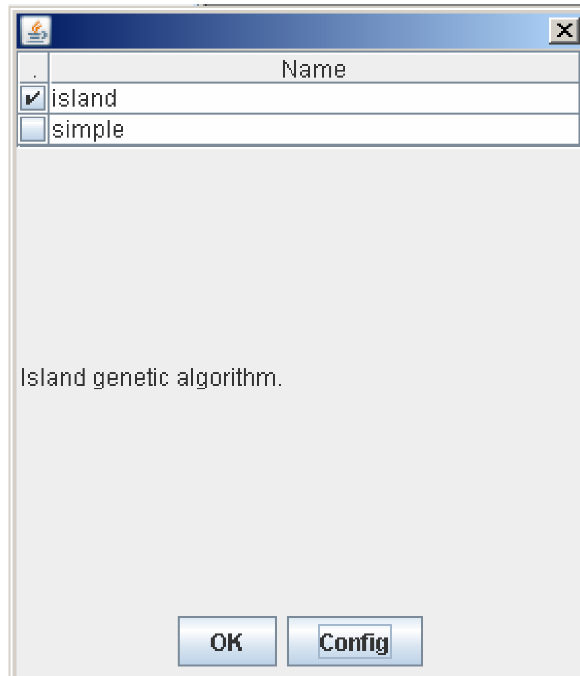


Рис. 33. Диалог выбора генетического алгоритма

Например, можно выбрать *островной генетический алгоритм* (island). После этого необходимо выбрать особь, которую будем «выращивать» при помощи островного генетического алгоритма. Сделать это можно, выбрав соответствующий пункт меню (*Settings -> Individuals)*, как показано на рис. 32.

После этого появляется диалоговое окно выбора типа особи (рис. 34).

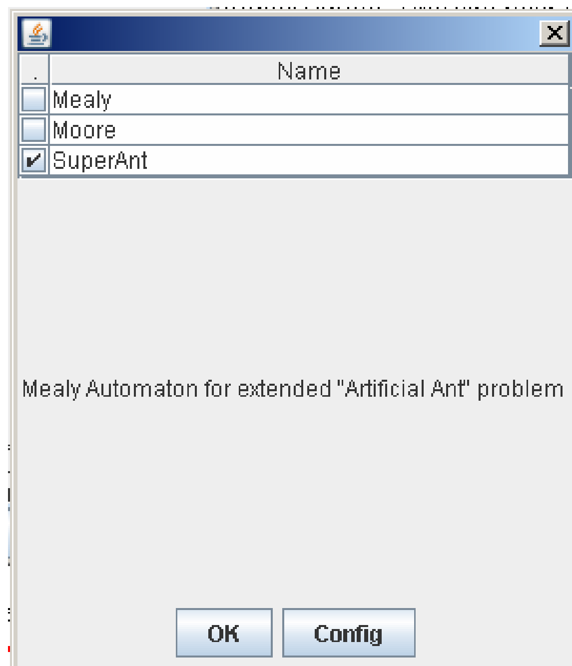


Рис. 34. Диалог выбора типа особи

Также как и в случае с диалоговым окном выбора генетического алгоритма. Кнопка *OK* предназначена для подтверждения выбора. По кнопке *Config* открывается окно настройки параметров алгоритма (рис. 35). Более подробно о настройке можно прочитать в разд. 2.1.3.3.

В качестве примера выбираем особь *SuperAnt*. Данная особь предназначена для решения расширенной задачи об «Умном муравье».

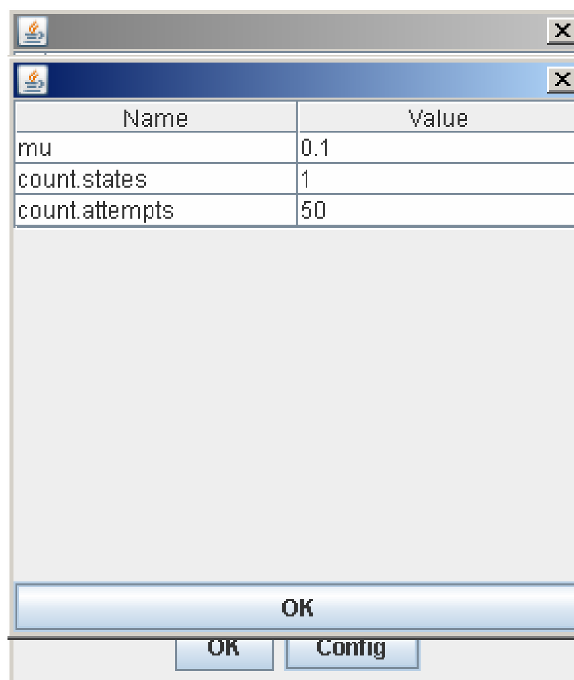


Рис. 35. Диалог настройки особи

После нажатия на кнопку *Start* (рис. 30) запускается генетический алгоритм. В соответствующих вкладках строятся графики среднего (рис. 36) и максимального значения (рис. 37) функции приспособленности в зависимости от поколения.

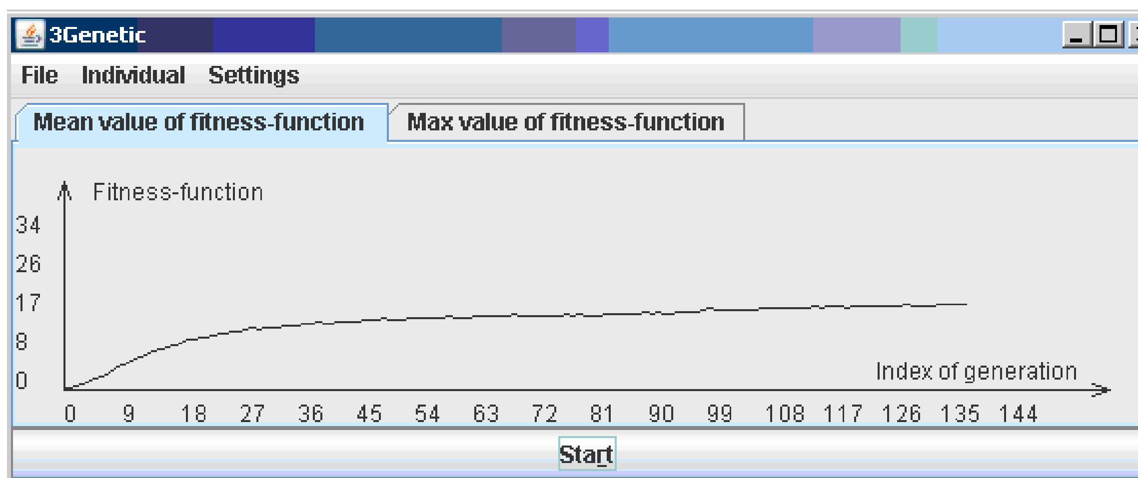


Рис. 36. Среднее значение функции приспособленности

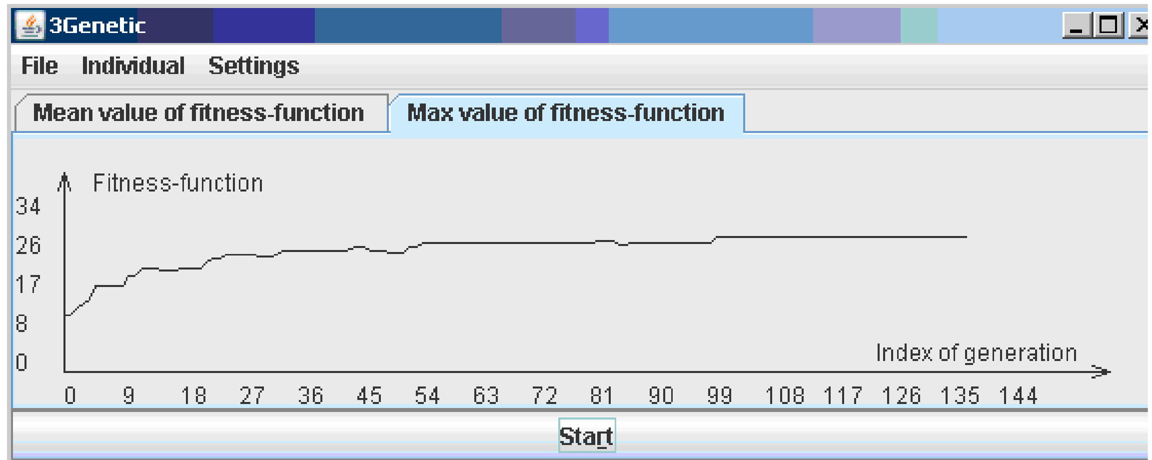


Рис. 37. Максимальное значение функции приспособленности

Для просмотра и сохранения результатов предназначены подпункты пункта меню *Individual* (рис. 38).

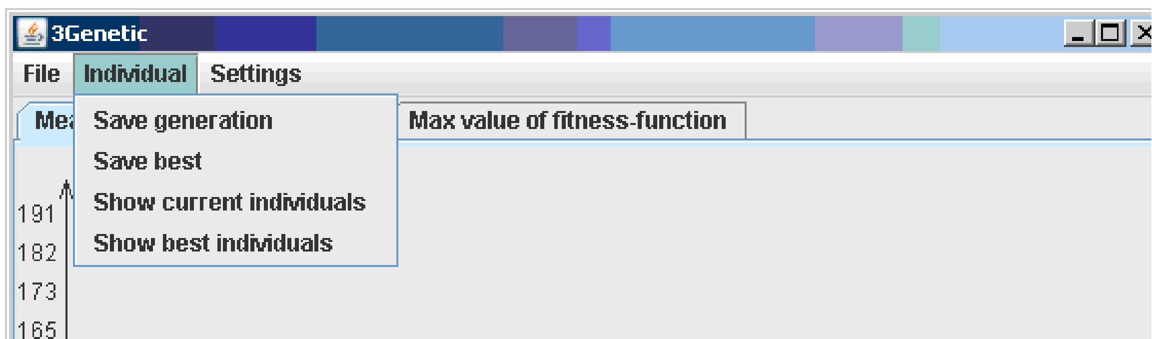


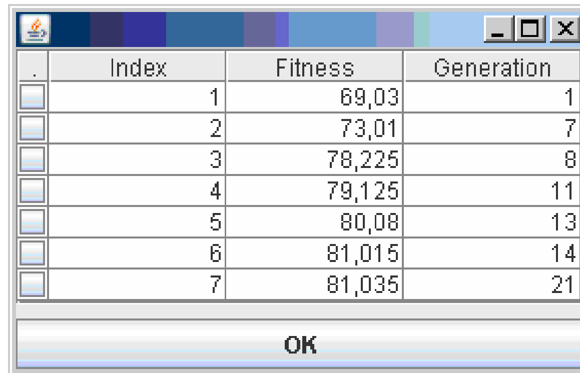
Рис. 38. Меню сохранения и просмотра особей

При выборе пункта меню *Save generation* особи текущего поколения сохраняются в текстовом формате в директории *results*. Формат текстового описания полностью определяется особью, так как невозможно создать единый язык для различных представлений. Действия при выборе пункта меню *Save best* аналогичны действиям при выборе пункта меню *Save generation* с той лишь разницей, что сохраняются лучшие особи – особи, которые имели на каком-то поколении максимальное значение фитнесс-функцииз всех предыдущих особей.

При выборе пункта меню *Show best individuals* появляется список особей для просмотра (рис. 39).

После выбора желаемой особи, открывается соответствующий решаемой задаче визуализатор (рис. 40).

Действия при выборе пункта меню *Show current generation* отличаются от действий при выборе пункта меню *Show best individuals* лишь тем, что для просмотра предлагается текущее поколение.



.	Index	Fitness	Generation
	1	69,03	1
	2	73,01	7
	3	78,225	8
	4	79,125	11
	5	80,08	13
	6	81,015	14
	7	81,035	21

Рис. 39. Диалог выбора визуализируемой особи

Рис. 40. Визуализатор для задачи «Умный муравей-3»

Если для решаемой задачи визуализаторов нет, то появится соответствующее предупреждение (рис. 41).

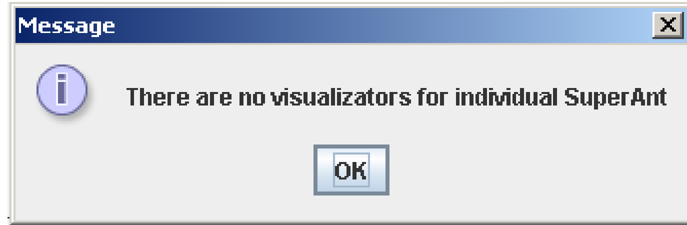


Рис. 41. Предупреждение об отсутствии визуализаторов

Выбор и настройка текущего визуализатора осуществляется при выборе в меню пункта *Settings -> Visualizators* (рис. 32) аналогично выбору и настройке типа особи и реализации генетического алгоритма. В окне выбора эмулятора, предлагаются только совместимые эмуляторы – те, при создании которых были указаны соответствующие особи. Более подробно это изложено в разд. 1.2.3.4.

2.1.3.3. Настройка генетических алгоритмов, входящих в состав программного средства *3Genetic*

В число генетических алгоритмов, входящих в состав программного средства *3Genetic* входят классический (*Standard*) и *островной* генетические алгоритмы.

Островной генетический алгоритм имеет ряд настраиваемых параметров, показанных на рис. 42.

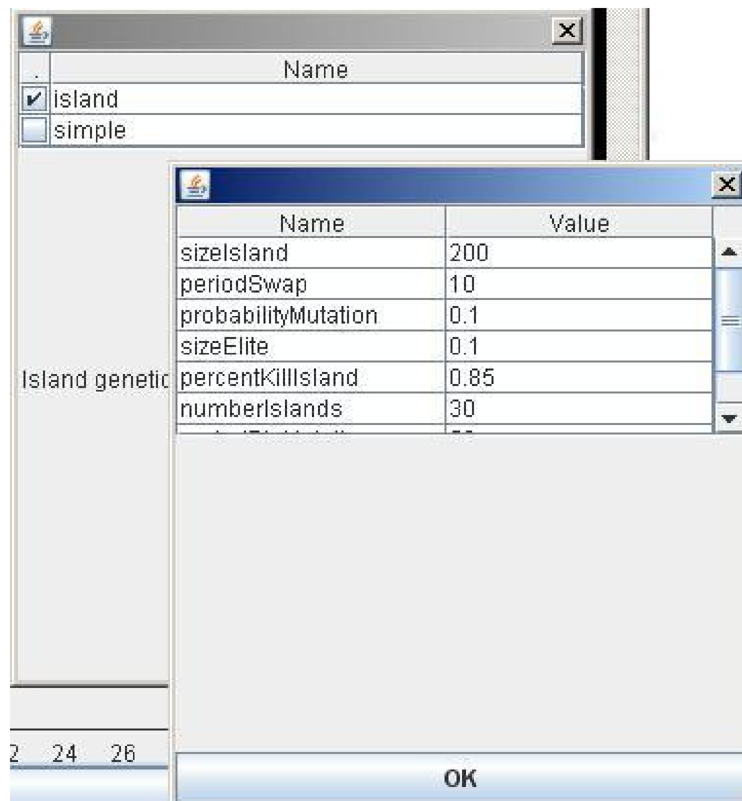


Рис. 42. Настраиваемые параметры островного генетического алгоритма

Поясним значение параметров:

- `sizeIsland` – число особей на острове;
- `periodSwap` – число поколений между миграциями;
- `probabilityMutation` – вероятность мутации особи;
- `sizeElite` – процент *элитных* особей;
- `percentKillIsland` – процент уничтожаемых островов во время большой мутации;
- `numberIslands` – число островов;
- `periodBigMutation` – число поколений между двумя большими мутациями;
- `numberSwap` – процент *мигрирующих* особей.

Укажем настраиваемые параметры классического генетического алгоритма (рис. 43):

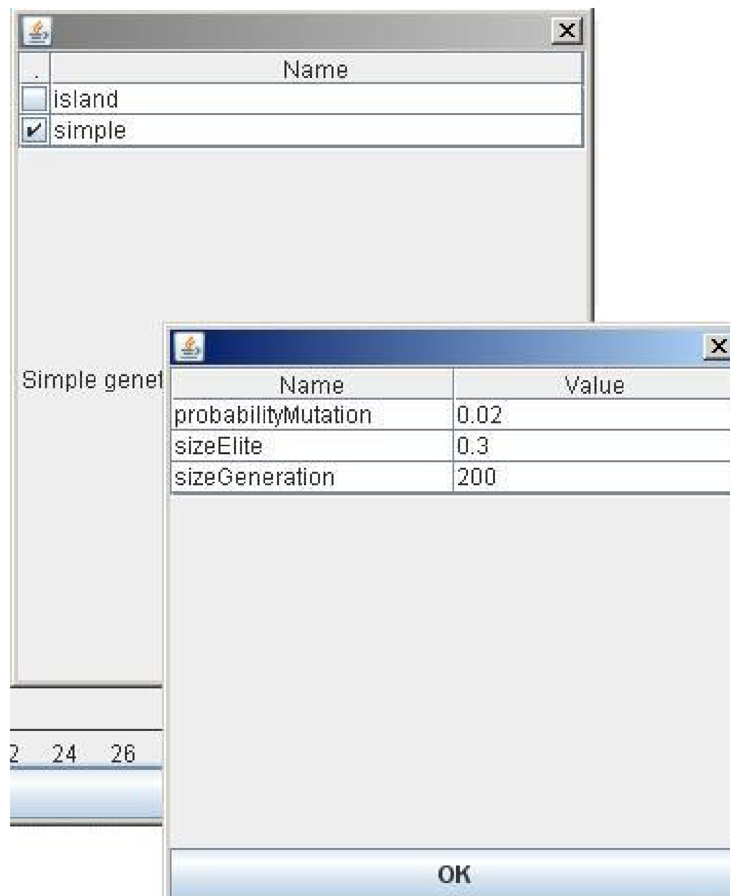


Рис. 43. Настраиваемые параметры классического генетического алгоритма

- `probabilityMutation` – вероятность мутации особи;
- `sizeElite` – процент *элитных* особей;
- `sizeGeneration` – число особей в поколении.

2.1.3.4. Особи

В состав программного средства *3Genetic* входят три представления особи: `Mealy`, `Moore`, `SuperAnt`.

Особь `Mealy` позволяет представлять как одиночный управляющий автомат Мили, так и систему из двух автоматов Мили, взаимодействующих посредством вложенности. Диалог настройки параметров особи такого типа приведен на рис. 43. Входящая в состав программного средства *3Genetic* реализация этой особи позволяет строить управляющие автоматы только для задачи «Умный муравей», однако при переопределении функции приспособленности ее можно использовать и для построения автоматов в других задачах.

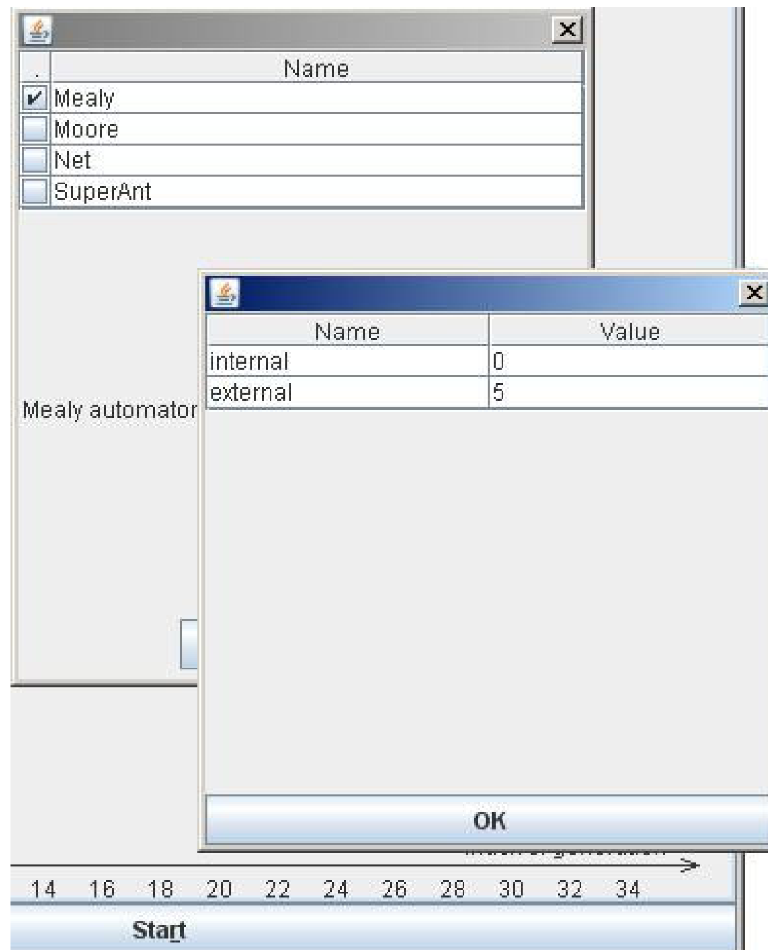


Рис. 44. Настраиваемые параметры конфигурации особи Mealy

Поясним параметры особи:

- `internal` – число состояний во внутреннем автомате;
- `external` – число состояний во внешнем автомате.

Отдельно отметим, что если внутренний автомат не содержит состояний, то в процессе работы генетического алгоритма будет строиться особь, состоящая из одиночного автомата Мили.

Особь `Moore` позволяет решать, такую же задачу, как и особь `Mealy`, но с той лишь разницей, что с ее помощью строиться система взаимодействующих автоматов Мура или одиночный автомат Мура.

Настройка особи `Moore` осуществляется так же, как и особи `Mealy`.

Третий из стандартных способов представления особи – `SuperAnt`. Эта особь предназначена для решения задачи «Умный муравей-3» при помощи одиночного автомата Мили. Настройка `SuperAnt` несколько сложнее, чем настройка предыдущих двух особей (рис. 45).

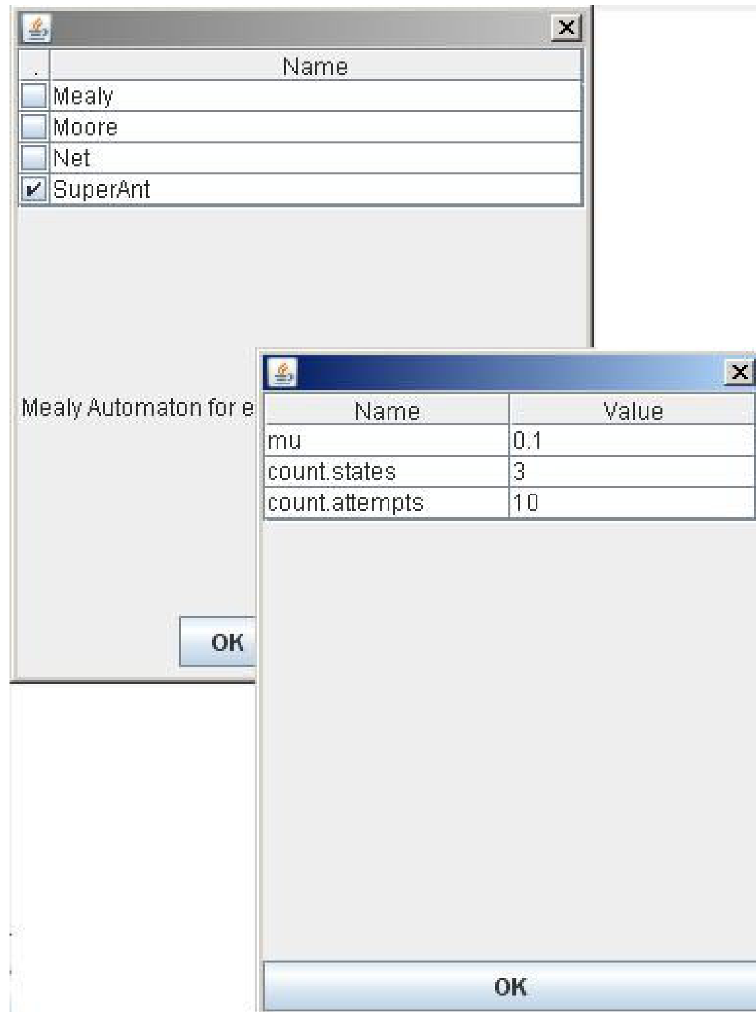


Рис. 45. Настраиваемые параметры конфигурации особи Mealy

Перечислим настраиваемые параметры:

- `mu` – вероятность появления еды в клетке поля;
- `count.states` – число состояний в автомате;
- `count.attempts` – число запусков муравья при подсчете *fitness*-функции.

2.1.3.5. Визуализаторы

Помимо указанного визуализатора для особи *SuperAnt*, в состав программного средства *3Genetic* входит визуализатор для задачи «Умный муравей» и особей `Mealy` и `Moore` (рис. 46).

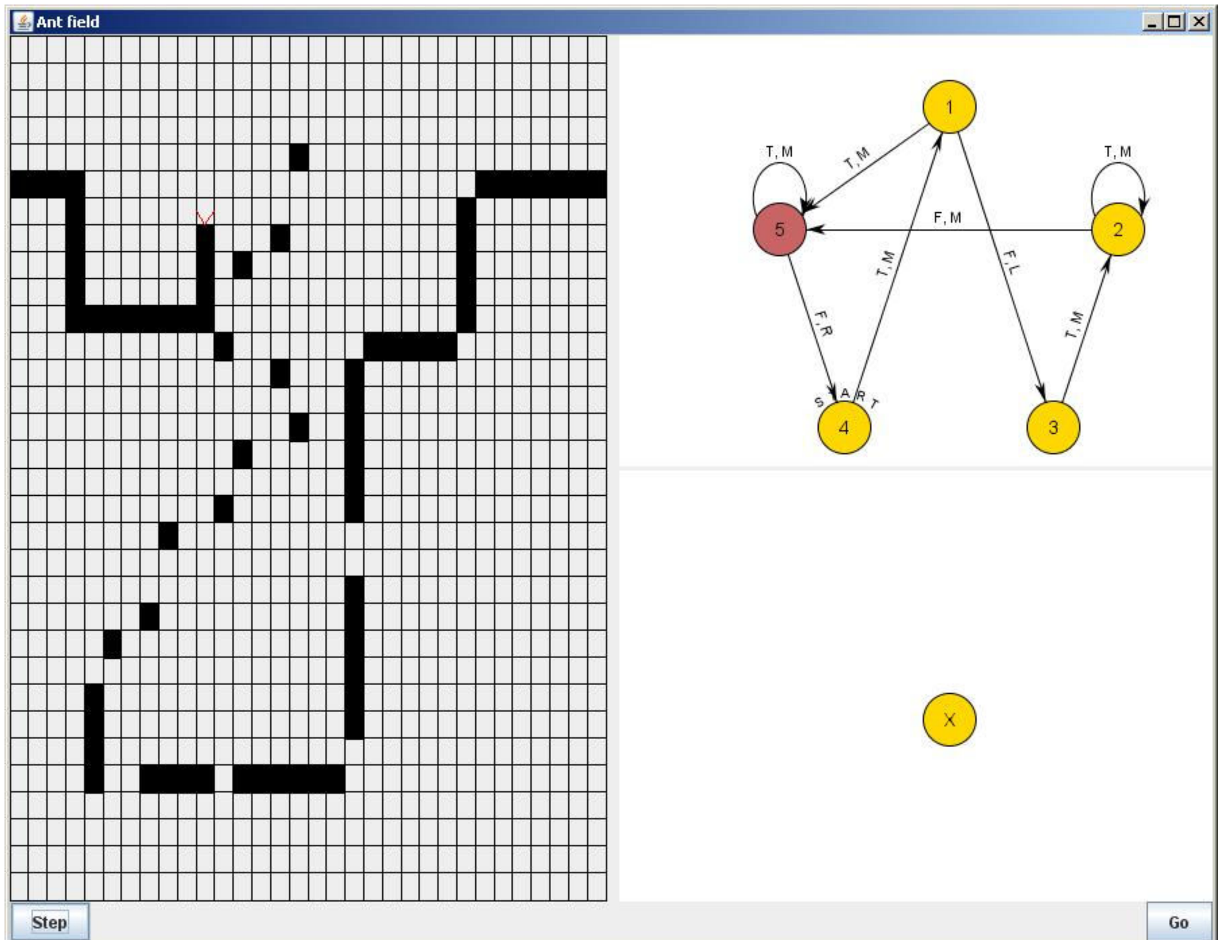


Рис. 46. Визуализатор особей Mealy и Moore

Окно визуализатора состоит из двух частей. В левой части отображается текущее состояние поля и положение муравья. В правой части отображаются диаграммы переходов автоматов, задаваемых особью, для которой проводится визуализация. При этом основной автомат отображен сверху, а вложенный в него – снизу. Если вложенный автомат не содержит состояний, то он отображается кругом, внутри которого находится крестик (рис. 46, справа внизу).

На поле черным цветом обозначены клетки, в которых находится еда, а углом обозначен муравей. При этом направление угла обозначает направление взгляда муравья. На диаграммах переходов автоматов темным цветом выделено активное в настоящий момент состояние.

Визуализация управляется двумя кнопками: по нажатию на кнопку *Step* муравей выполняет один шаг, а по нажатию на кнопку *Go* – 200 шагов.

2.2. Применение программных средств

В настоящем разделе описывается пример применения каждого программного средства для решения задачи «Умный муравей-3» [25].

2.2.1. Задача «Умный муравей-3»

Приведем описание классической постановки задачи «Умный муравей» [1, 13, 29, 33]. Используется двумерный тор размером 32 на 32 клетки (рис. 47). На некоторых клетках поля

расположены яблоки – черные клетки на рис. 47. Яблоки расположены вдоль некоторой ломаной линии, но не на всех ее клетках. Клетки ломаной, на которых яблок нет – серые. Белые клетки – не принадлежат ломаной и не содержат яблок. Всего на поле 89 яблок.

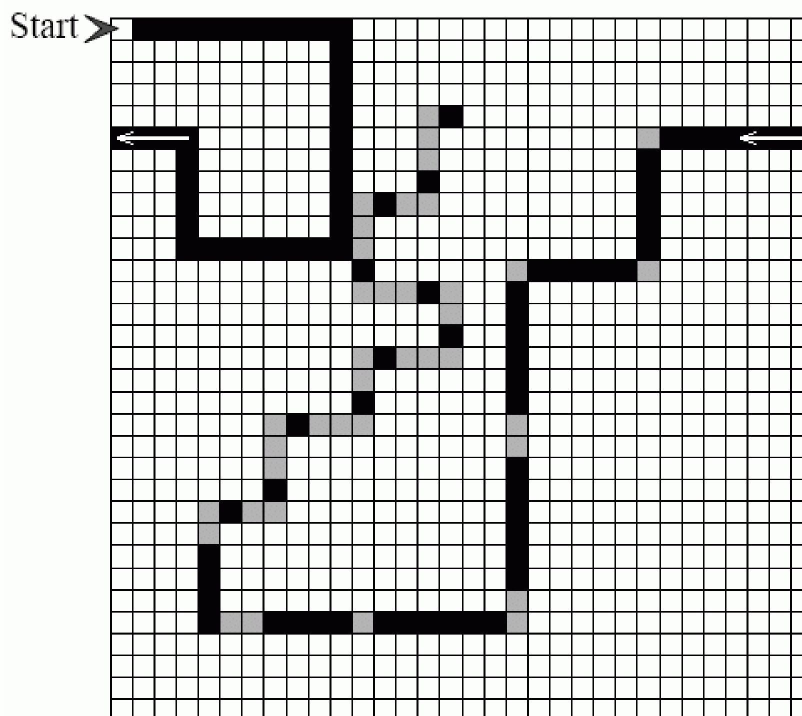


Рис. 47. Поле с яблоками

В клетке с пометкой «Start» находится муравей. Он занимает клетку поля и смотрит в одном из четырех направлений (север, запад, юг, восток). В начале игры муравей смотрит на восток. Он умеет определять находится ли яблоко непосредственно перед ним. За ход муравей совершает одно из четырех действий:

- идет вперед на одну клетку, съедая яблоко, если оно находится перед ним;
- поворачивается вправо;
- поворачивается влево;
- стоит на месте.

Съеденные муравьем яблоки не восполняются. Муравей жив на всем протяжении игры – еда не является необходимым ресурсом для его существования. Никаких других персонажей, кроме муравья, на поле нет. Ломаная *строго задана*. Муравей может ходить по любым клеткам поля. Игра длится не более 200 ходов, на каждом из которых муравей совершает одно из четырех описанных выше действий. В конце игры подсчитывается число яблок, съеденных муравьем. Это значение – результат игры.

Цель игры – создать муравья, который не более чем за 200 ходов съест как можно больше яблок. Муравьи, съевшие одинаковое число яблок, заканчивают игру с одинаковым результатом вне зависимости от числа ходов, затраченных каждым из них на процесс еды. Однако эта задача может иметь различные модификации, например, такую, в которой при одинаковом числе съеденных яблок, лучшим считается муравей, съевший яблоки за меньшее число ходов. Ниже будет показано, что поведение муравья может быть задано конечным автоматом. При этом может быть поставлена задача о построении автомата с минимальным числом состояний для муравья,

съедающего все яблоки, или автомата для муравья, съедающего максимальное число яблок при заданном числе состояний.

Конечный автомат, изображенный на рис. 48, имеет всего пять состояний.

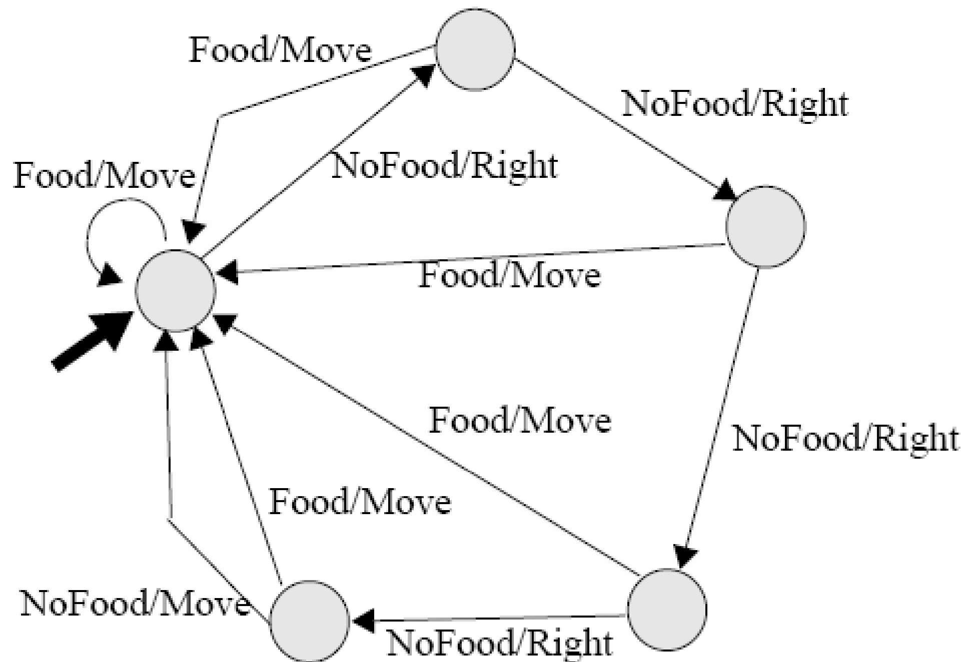


Рис. 48. Конечный автомат, задающий муравья

Этот автомат описывает поведение муравья, который не решает задачу – за 200 ходов съедает только 81 яблоко, а за 314 ходов – все 89 яблок. Муравей действует по принципу «Вижу яблоко – иду вперед. Не вижу – поворачиваю. Сделал круг, но яблок нет – иду вперед».

Постановка задачи «Умный муравей-3», предложенной в работе [25], содержит несколько существенных отличий.

Во-первых, расширена область обзора муравья – вместо одной клетки он видит восемь. Таким образом, множество значений входных переменных содержит $2^8 = 256$ элементов. На рис. 49 изображена область обзора муравья (клетка, в которой находится муравей, обозначена серым цветом).

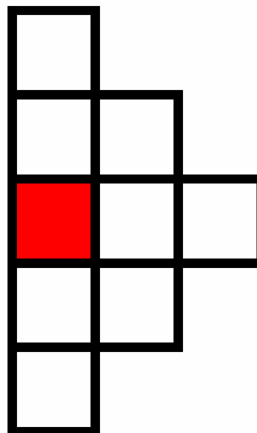


Рис. 49. Область видимости муравья

Во-вторых, расположение еды на поле не фиксировано, а генерируется случайным образом. При этом вероятность того, что яблоко окажется в некоторой клетке, одинакова для всех клеток поля и равна μ.

В этом случае число яблок, съеденных муравьем за 200 ходов, есть случайная величина ξ (определяемая муравьем) на дискретном множестве элементарных исходов Ω – множестве расположений еды – битовых матриц 32 x 32. Каждому исходу ω_i, содержащему k единиц поставим в соответствие вероятность $p(\omega_i) = \mu^k(1-\mu)^{n-k}$, где n = 32 x 32.

Для вычисления этой величины в общем случае необходимо перебрать все возможные битовые матрицы размером 32 x 32. Поэтому для оценки эффективности автомата, задающего поведение муравья, вместо точного вычисления этого математического ожидания, оно будет вычислять приближенно – с помощью моделирования поведения муравья на 10000 случайно сгенерированных полях.

2.2.2. Пример применения программного средства *GAAP*

В настоящем разделе описывается применение программного средства *GAAP* для решения задачи «Умный муравей-3».

2.2.2.1. Создание файла настроек

Программное средство *GAAP* содержит реализацию модели, соответствующей задаче «Умный муравей-3» (examples/ant). Файл настроек программного средства *spring.xml* для решения задачи «Умный муравей-3» приведен в листинге 10.

Листинг 10. Файл настроек для решения задачи «Умный муравей-3»

```
1: <beans>
2:    <description>Configuration for solving ant
     problem.</description>
3:    <import
     resource="classpath:org/gaap/examples/spring.xml"/>

04:    <bean id="tester"
     class="org.gaap.examples.ant.AntTester3">
05:        <constructor-arg value="0.05"/>
06:    </bean>

07:    <bean id="breeder"
     class="org.gaap.ga.impl.DefaultBreeder">
08:       <constructor-arg index="0" ref="tester"/>
09:       <constructor-arg index="1" value="0.9"/>
10:        <property name="fitnessFunction">
11:            <bean
     class="org.gaap.examples.ant.AntFitnessFunction3"/>
12:        </property>
13:    </bean>

14:    <bean id="ga" class="org.gaap.ga.impl.DefaultGa">
15:        <property name="selectors">
16:            <list>
```

```
17:               <bean
    class="org.gaap.ga.impl.selectors.RouletteSelector">
18:                   <constructor-arg value="0.5"/>
19:               </bean>
20:               <bean
    class="org.gaap.ga.impl.selectors.RouletteSelector">
21:                   <constructor-arg value="0.5"/>
22:               </bean>
23:           </list>
24:       </property>
25:       <property name="operators">
26:           <list>
27:               <bean
    class="org.gaap.fsm.impl.compact.CompactCrossover"/>
28:               <bean
    class="org.gaap.fsm.impl.compact.CompactMutation"/>
29:           </list>
30:       </property>
31:       <property name="breeder" ref="breeder"/>
32:       <property name="listeners">
33:           <list>
34:               <bean
    class="org.gaap.ga.impl.listeners.MaxAgeStopStrategy">
35:                   <constructor-arg value="100"/>
36:               </bean>
37:           </list>
38:       </property>
39:       <property name="population">
40:           <bean factory-bean="breeder" factory-
    method="breed">
41:               <constructor-arg
    type="org.gaap.ga.Offspring">
42:                   <bean class="org.gaap.ga.Offspring">
43:                       <constructor-arg>
44:                           <bean
45: class="org.gaap.fsm.impl.compact.CompactOperators"
46:                           factory-method="random">
47:                              <constructor-arg index="0"
    value="1"/>
48:                               <constructor-arg index="1"
    value="3"/>
49:                               <constructor-arg index="2"
    value="8"/>
50:                               <constructor-arg index="3"
    value="200"/>
51:                               <constructor-arg index="4"
    value="5"/>
52:                           </bean>
```

```
53:                         </constructor-arg>
54:                     </bean>
55:                 </constructor-arg>
56:                 <constructor-arg
     type="org.gaap.ga.Population">
57:                     <null/>
58:                 </constructor-arg>
59:             </bean>
60:         </property>
61:     </bean>

62:     <bean id="populationPanel
     parent="examplesPopulationPanel">
63:         <property name="individualGui">
64:             <bean class="org.gaap.examples.ant.AntGui2">
65:                 <constructor-arg ref="tester"/>
66:             </bean>
67:         </property>
68:     </bean>
69: </beans>
```

Опишем структуру приведенного листинга.

Строки 4–7 описывают создание виртуальной модели для задачи «Умный муравей-3» при значении параметра $\mu = 0.05$;

Строки 7–13 описывают инициализацию алгоритма формирования следующего поколения: 10% особей из предыдущей популяции переходят в следующую популяцию (использование принципа элитизма), остальные особи добираются по рулетке из промежуточной популяции (размер популяции остается неизменным). Функция приспособленности — среднее значение числа съеденных яблок на 100 случайных картах, карты внутри одной популяции совпадают, карты различных популяций различны;

Строки 14–61 описывают инициализацию генетического алгоритма.

Строки 15–30 описывают стратегию формирования промежуточной популяции (50% популяции отбирается по рулетке для скрещиваний, 50% для мутаций);

Строки 33–37 описывают выбор стратегии останова (генетический алгоритм будет остановлен после генерации 100 популяций);

Строки 45–54 описывают генерацию начальной популяции:

- используется представление автомата сокращенными таблицами (пакет *org.gaap.fsm.impl.compact* модуля *fsm-core*);
- число состояний – 1;
- число выходных воздействий автомата – 3;
- число входных переменных – 8;
- число значимых входных переменных– 5;
- размер популяции – 200 особей;

Строки 62–68 описывают создание графического интерфейса.

2.2.2.2. Вычисление функции приспособленности

В листинге 11 приводится реализации функции приспособленности, которая испльзуется для оценки особи в задаче «Умный муравей-3».

Листинг 11. Пример реализации функции приспособленности для решения задачи «Умный муравей-3»

```
public final class AntTester3 extends AbstractTester
        implements Serializable {

    public static final int FOOD_COUNT = 0;
    public static final String FOOD_COUNT_NAME = "Food
        count";
    public static final int FIELDS_COUNT = 100;

    private Ant ant = new Ant();
    Private double mu;

    public AntTester3(double mu) {
        this.mu = mu;
        fieldNames = new String[]{FOOD_COUNT_NAME};
    }

    @Override
    public Object[] test(Object chromosome) {
        AntField[] fields = new AntField[FIELDS_COUNT];
        for (int i = 0; i < fields.length; i++) {
            fields[i] = new AntField(mu);
        }

        int food = 0;
        for (AntField field : fields) {
            AutomatedObject ao =
        AutomatedObjectFactory.get(ant, (StateMachine)
        chromosome);
            ant.initialize(new AntField(field));
            while (ant.getStep() < field.maxSteps) {
                ao.step();
                if (ant.getField().getFoodCount() == 0) {
```

```
                                break;
                            }
                        }
                        food += ant.getFoodCount();
                    }
                    return new Object[]{
                            food
                    };
                }
            }


            public final class AntFitnessFunction3 implements
                    FitnessFunction {
                @Override
                public double calculate(Object[] data) {
                    double res = (Integer)
                    data[AntTester3.FOOD_COUNT];
                    return res / AntTester3.FIELDS_COUNT;
                }
            }
```

2.2.2.3. Результаты эксперимента

Результаты эксперимента приведены в табл. 4.

Таблица 4. Результаты эксперимента

Число состояний	1	2	4	8	16
Максимальное значение функции приспособленности	23.33	26.61	27.63	25.72	27.34

Ниже приводятся графики, отображающие ход эволюции для вывода автоматов с различным числом состояний (рис. 18–22).

На рис. 55 приводится пример поведения «муравья» с двумя состояниями, построенного с помощью описываемого программного средства.

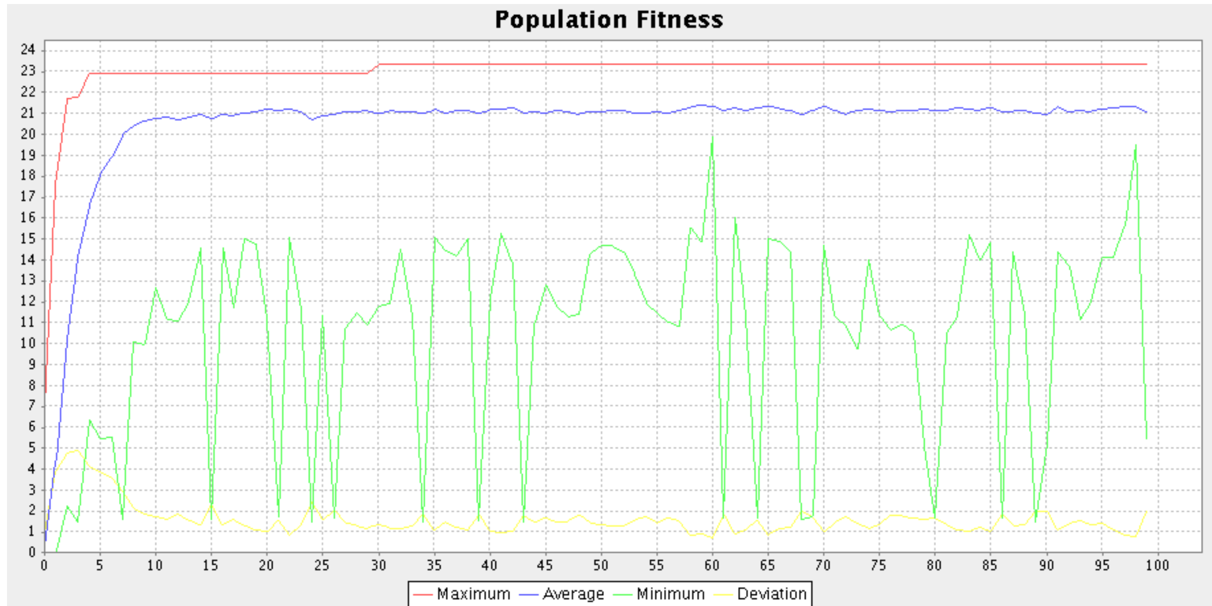


Рис. 50. Ход эволюции для автоматов с одним состоянием

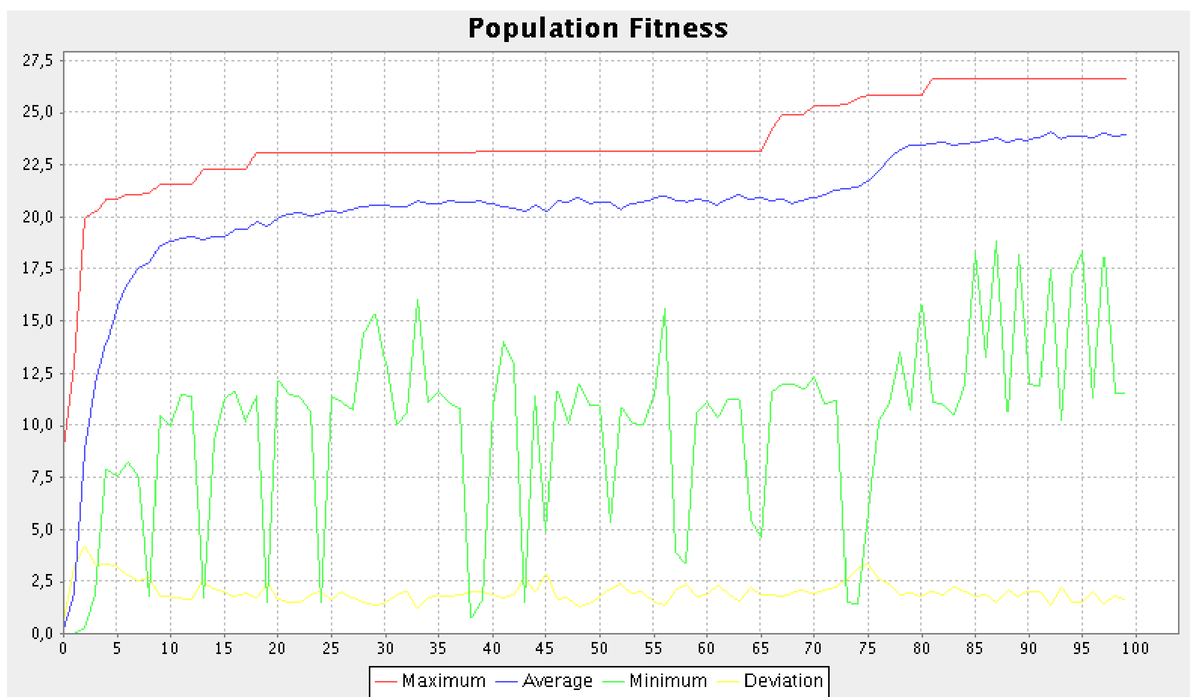


Рис. 51. Ход эволюции для автоматов с двумя состояниями

Рис. 52. Ход эволюции для автоматов с четырьмя состояниями

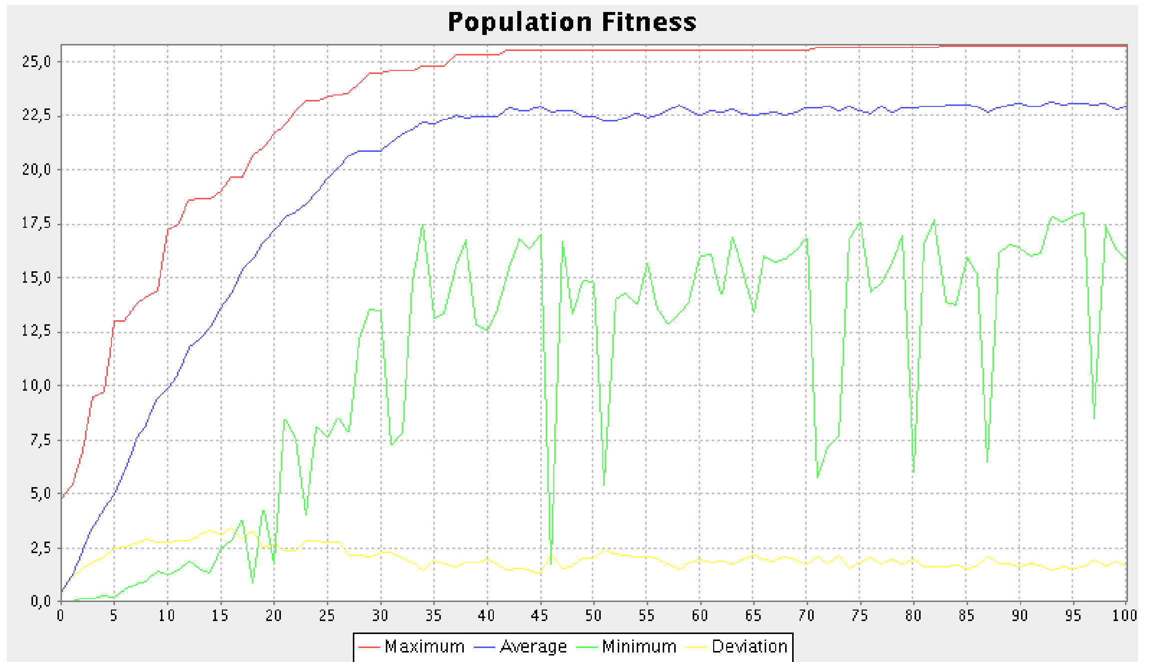


Рис. 53. Ход эволюции для автоматов с восемью состояниями

Рис. 54. Ход эволюции для автоматов с шестнадцатью состояниями

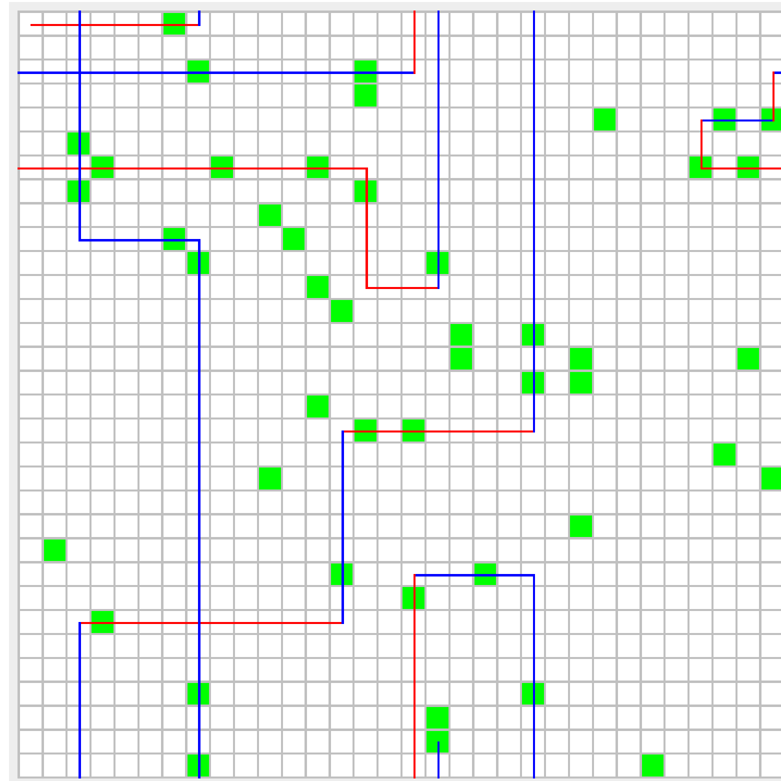


Рис. 55. Пример поведения «муравья» с двумя состояниями

2.2.3. Пример применения программного средства *AutoAnt*

В настоящем разделе описывается применение программного средства *AutoAnt* для решения задачи «Умный муравей-3».

2.2.3.1. Настройка программного средства для решения задачи

Программное средство *AutoAnt* содержит реализацию модели, соответствующей задаче «Умный муравей-3». Класс *ru.ifmo.ctddev.autoant.AntFitness* является готовым функтором, подсчитывающим функцию приспособленности. Классом выходных воздействий соответствующего автомата является перечислимый тип *ru.ifmo.ctddev.autoant.Action*.

Приведем содержимое файла настроек программного средства *autoant.properties* для решения задачи «Умный муравей-3».

```
roulette.ratio=0.25
mutation.probability=0.02
population.size=200
populations.max=100
predicates.count=8
action.class=ru.ifmo.ctddev.autoant.Action
fitness.class=ru.ifmo.genetic.autoant.AntFitness
states.count=<число состояний>
```

Для различных чисел состояний в генерируемом автомате необходимы отдельные запуски программного средства.

2.2.3.2. Результаты эксперимента

Результаты эксперимента приведены в табл. 5.

Таблица 5. Результаты эксперимента

Функция приспособленности				
Число состояний	2	4	8	16
Функция приспособленности	25.79	26.09	26.26	20.77

На рис. 31–34 приводятся графики, отображающие ход эволюции для вывода автоматов с различным числом состояний.

Приведем автомат из восьми состояний, построенный программным средством *AutoAnt* в ходе эксперимента. Дерево решений, соответствующее состоянию с номером три, приведено на рис. 60. Соответствие между входными переменными и видимыми муравьем клетками показано на рис. 61. Переходы, заданные листьями дерева, приводятся в формате *Номер состояния*/*Метка действия*. Действию «шаг вперед» соответствует метка *M*, действиям «поворот налево» и «поворот направо» – метки *L* и *R* соответственно.

Остальные состояния автомата производят переходы независимо от входных переменных. Деревья решений, соответствующие этим состояниям, приведены на рис. 62. Они имеют единственную вершину, задающую переход из состояния. Состояние автомата с номером семь является недостижимым и на рисунке не изображено. Отметим, что состояния с номерами один, четыре и восемь являются идентичными. Следовательно, число состояний построенного автомата может быть уменьшено до пяти, если объединить указанные состояния и удалить недостижимое состояние с номером семь.

Стартовым состоянием автомата является состояние с номером три.

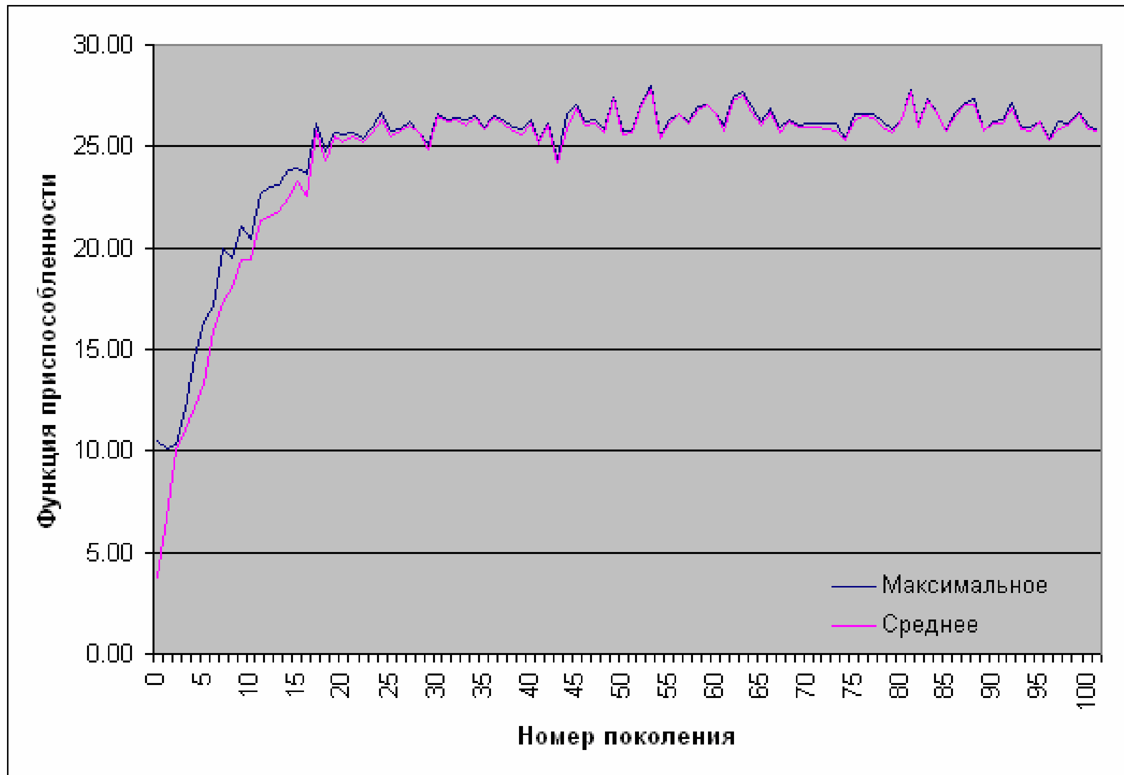


Рис. 56. Ход эволюции для автоматов с двумя состояниями

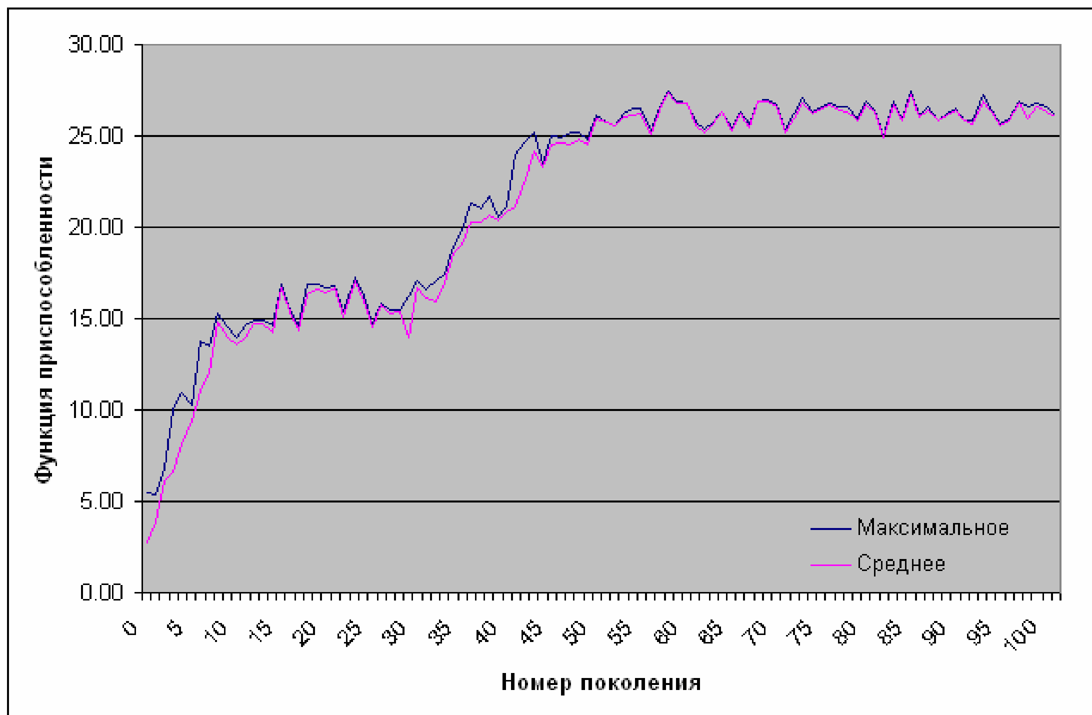


Рис. 57. Ход эволюции для автоматов с четырьмя состояниями

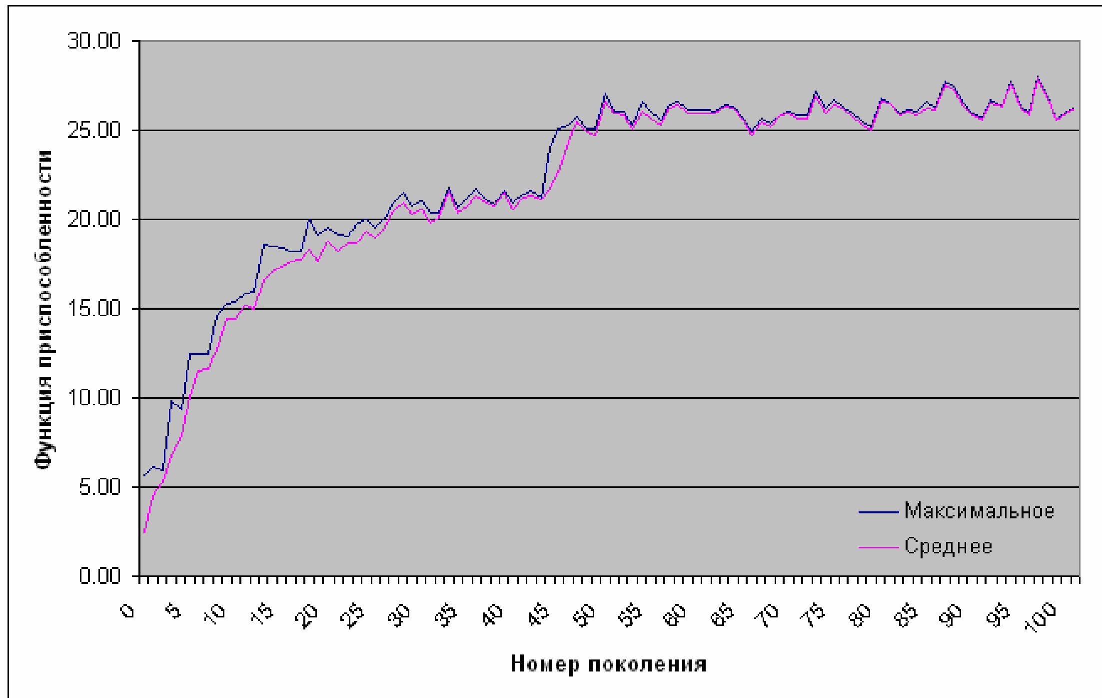


Рис. 58. Ход эволюции для автоматов с восемью состояниями

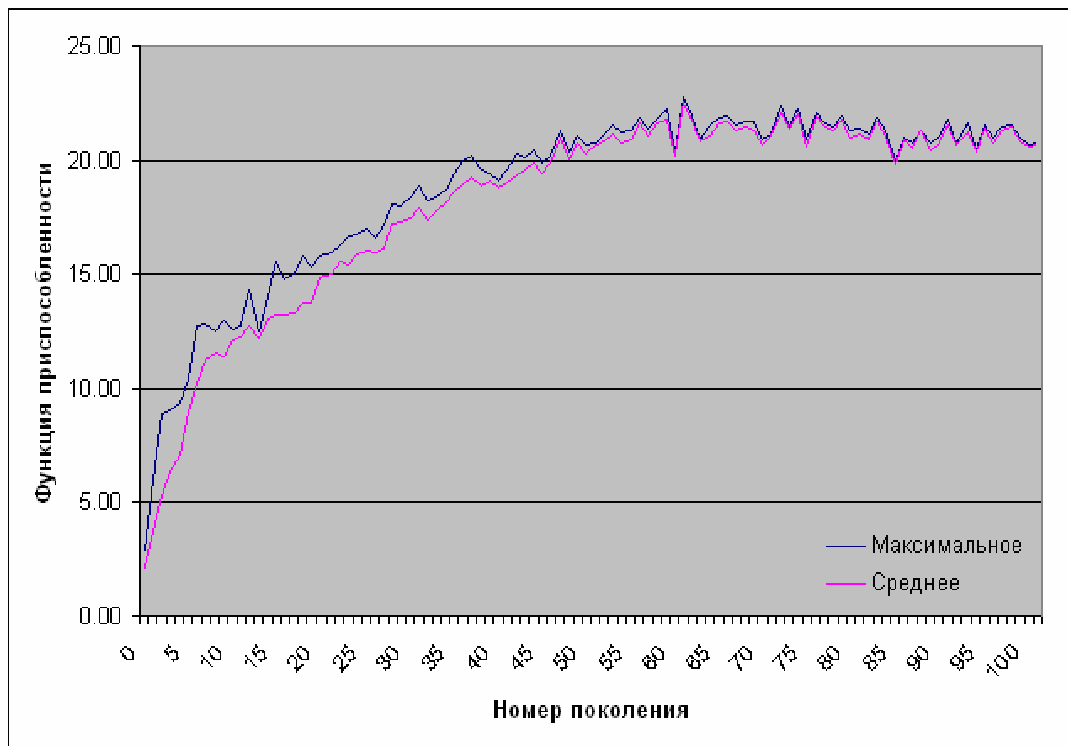


Рис. 59. Ход эволюции для автоматов с шестнадцатью состояниями

Рис. 60. Дерево решений, соответствующее состоянию с номером три

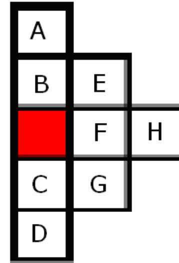


Рис. 61. Соответствие входных переменных автомата видимым клеткам

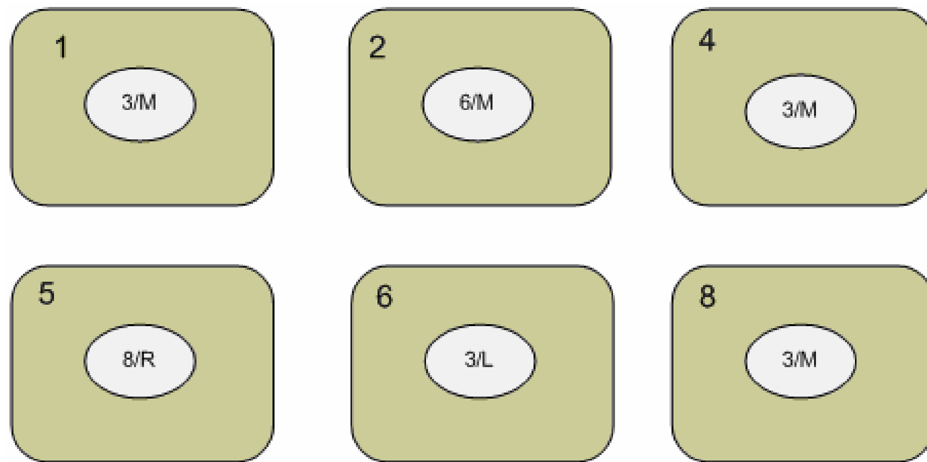


Рис. 62. Состояния построенного автомата

2.2.4. Пример применения программного средство *3Genetic*

Опишем использование применение программного средства *3Genetic* и метода совместного применения конечных автоматов и нейронных сетей для решения задачи «Умный муравей-3». Для решения этой задачи необходимо выполнить следующие шаги:

1. Создание класса, реализующего представление особи в виде совокупности нейронной сети и конечного автомата.
2. Написание файлов настройки и реализация дополнительных классов.
3. Создание фабрики, создающей экземпляры особей, и загрузчика этой фабрики.
4. Сборка *JAR*-архива и его подключение к ядру программного средства.
5. Настройка параметров представления особи.

2.2.4.1. Создание класса, реализующего представление особи

Для начала необходимо реализовать основной класс особи. Этот класс (листинг 12) должен реализовать интерфейс `Individual`. Например, допустима реализация интерфейса `Automaton`, который, в свою очередь, является потомком `Individual`.

Листинг 12. Реализация интерфейса `Individual` для представления особи, используемом в методе совместного использования конечных автоматов и нейронных сетей

```
public class NetAutomaton implements Automaton{

    public NetAutomaton (int ns, int is, Net net, int
     fitnessCount, double mu);

    public int getInitialState();
    public Transition getTransition();
    public void setTransition();
    public int getNumberStates();
    public Automaton getNestedAutomaton();
    public String getStateString();
    public double fitness();
    public NetAutomaton mutate(Random r);
    public NetAutomaton[] crossover(Individual p, Random r);
    public int compareTo(Individual o);
    public Net getNet();
    public void recalcFitness(int newFitnessCount);
    public String toString();

    public static class NetAutomatonTransition implements
     Transition{

        public NetAutomatonTransition(char action,
     int endState);
        public int getEndState();
        public char getAction();
        public String toString();
    }
```

Из приведенных в листинге методов обязательными (входящими в интерфейс `Individual`) являются только:

- `mutate` – метод должен возвращать новую особь, если метод изменяет текущую особь, то корректная работа не гарантируется;
- `crossover` – метод должен возвращать массив из двух новых особей;
- `compareTo` – метод, реализующий интерфейс `java.lang.Comparable<Individual>`;
- `fitness` – метод, возвращающий значение функции приспособленности данной особи.

Допускается использование любых вспомогательных классов. В приведенном выше случае используется класс `Net`.

2.2.4.2. Написание файлов настройки и реализация дополнительных классов

Из пары «Автомат Мили и нейронная сеть» отдельно выделим нейронную сеть. Создадим класс `Net`, и для удобства использования унаследуем его от интерфейсов `Individual` и `Cloneable`:

```
public class Net implements Individual, Cloneable{

    public Net(int variableCount, int outCount, double maxEdgeMutation, ArrayList<Integer>[] graph, double[][] w, Neuron[] neuron);

    public int getValue(int value);
    public double fitness();
    public Net mutate(Random r);
    public Net[] crossover(Individual p, Random r);
    public int compareTo(Individual o);
    public int getCountOutVariables();
    public Net clone();

}
```

Опишем методы, содержащиеся в данном классе:

- `getValue` – принимает значение входных переменных в виде битовой маски. Возвращает значение, которое выдает нейронная сеть, также в виде битовой маски;
- `fitness` – позволяет получить значение фитнесс-функции нейронной сети. В этом случае всегда возвращает ноль, так как отдельно нейронная сеть не выращивается генетическим алгоритмом;
- `mutate`, `crossover` – генетические операторы;
- `compareTo` – метод, реализующий интерфейс `java.lang.Comparable<Individual>`;
- `getCountOutVariables` – возвращает число выходных переменных;
- `clone` – метод, который переопределяет метод из класса `java.lang.Object`.

Отдельными классами реализованы и нейроны нейронной сети:

```
public interface Neuron extends Cloneable{

    public void changeValue(double val);
    public void setValue(double val);
    public double getValue();
    public Neuron mutate();
    public Neuron clone();

}
```

Используемые методы:

- `changeValue` – изменить значение на переданную величину;
- `setValue` – установить значение;
- `getValue` – получить значение;
- `mutate` – изменяет какой-нибудь из внутренних параметров нейрона;
- `clone` – метод, который переопределяет метод из класса `java.lang.Object`.

Классы, реализующие данный интерфейс:

- `AbstractNeuron`;
- `LimitNeuron`;
- `SigmoidNeuron`.

Последний дополнительный файл – `NetMover`:

```
public class NetMover implements Mover{

    public NetMover(NetAutomaton a, double mu);

    public boolean move()
    public void restart(Ant ant) ;

}
```

Данный класс предназначен для подсчета фитнесс-функции. Он реализует интерфейс `laboratory.util.individual.Mover`.

Конфигурационные файлы:

- `net.conf` – в текстовом виде задает структуру нейронной сети;
- `netAutomaton.conf` – задает структуру особи.

Более подробно структура данных файлов описана в разд. 3.4.5.

2.2.4.3. Создание фабрики особей и загрузчика

Следующий этап – создание фабрики особей. Класс фабрики, называющийся в рассматриваемом случае `NetAutomatonFactory`, должен реализовывать интерфейс `IndividualFactory`.

```
public class NetAutomatonFactory implements IndividualFactory{

    public NetAutomatonFactory(int ns, int cv, int cov, ArrayList<Integer>[] graph, int[] type, double maxMutation, int fitnessCount, double mu);


    public NetAutomaton randomIndividual();
}
```

Единственный метод интерфейса `IndividualFactory` – `randomIndividual` должен возвращать случайно сгенерированную особь.

Для того, чтобы будущий модуль смог создать фабрику особей, необходимо создать загрузчик – класс, реализующий интерфейс `Loader<IndividualFactory>`:

```
public class NetAutomatonFactoryLoader implements Loader<IndividualFactory>{

        public NetAutomatonFactoryLoader(JarFile file);
        public NetAutomatonFactory load(Object... args);
        public Properties getProperties();

}
```

Метод `getProperties` возвращает объект класса `java.util.Properties`. Именно эти свойства доступны для настройки (рис. 35). Метод `load` должен возвращать фабрику особей. Данный класс должен иметь конструктор от единственного параметра – *JAR*–файла, в котором данный модуль будет содержаться. Именно этот конструктор будет вызываться ядром программного средства *3Genetic*.

2.2.4.4. Сборка *JAR*–архива и его подключение к ядру программного средства

Со всеми дополнительными классами особь `Net automaton` имеет структуру, изображенную на рис. 63.

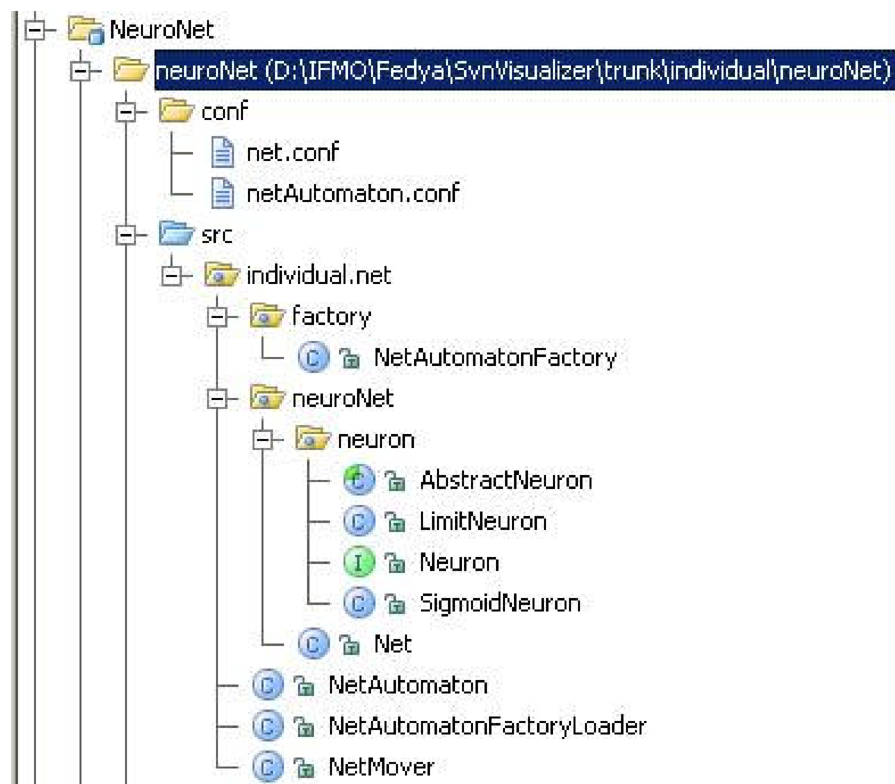


Рис. 63. Структура классов, необходимых для работы особи Net automaton

Необходимо скомпилировать файлы, находящиеся в каталоге `src`. Затем собрать из скомпилированных классов (файлов `.class`) *JAR*-архив. Это можно сделать, например, с помощью встроенных средств среды разработки. Кроме этого необходимо добавить в корень архива файлы из каталога `conf`.

Разберемся более подробно, что должен содержать *манифест JAR*-архива.

Параметр `Extension-Name` содержит имя особи, отображаемое в диалоге выбора особи (рис. 34). Параметр `Comment` должен иметь вид «`arg1 arg2` ||| комментарий, отображаемый в диалоге выбора особи». Здесь `arg1` и `arg2` – параметры, необходимые для рисования графиков, максимальное значение и число отображаемых знаков фитнесс-функции.

Параметр `Main-Class` должен указывать на класс-загрузчик, в данном примере это `individual.net.NetAutomatonFactoryLoader`.

Помещаем *JAR*–архив в каталог *individuals*. Особь готова к работе.

2.2.4.5. Настройка параметров особи `Net automaton`

Настройка особи `Net automaton` состоит из двух частей: задание структуры нейронной сети; задание параметров автомата и параметров подсчета функции приспособленности.

Структура нейронной сети задается в файле `net.conf`, который находится в корне *JAR*-архива. Зададим формат файла `net.conf`.

Пусть n – число нейронов, m – число переходов. Первая строка файла состоит из двух чисел n m. Далее идут m строк вида «a b», где a и b – это натуральные числа, причем $0 < a, b < n + 1$. Каждая такая строка описывает переход в нейронной сети из нейрона a в нейрон b. Последняя строка файла содержит n чисел, если i-ое равно нулю, то i-ый нейрон – пороговый, иначе сигмоидальный. Граф переходов должен быть *ациклическим*, в противном случае правильность работы нейронной сети не гарантируется.

Входные переменные подаются на первые нейроны, а выходные снимаются с последних.

Вторая часть настройки доступна как через файл `netAutomaton.conf`, так и через пользовательский интерфейс (рис. 35).

Поясним значение параметров:

- `numberStates` – число состояний в автомате;
- `countVariables` – число входных переменных;
- `countOutVariables` – число выходных переменных нейронной сети;
- `maxMutation` – максимальное изменение веса перехода (или порога в пороговом нейроне) в нейронной сети;
- `fitnessCount` – число эмуляций задачи об «Умном муравье» при подсчете функции приспособленности;
- `fieldPercent` – вероятность наличия в ячейки поля еды.

2.2.4.6. Результаты вычислительных экспериментов

Для решения задачи «Умный муравей-3» была выбрана структура нейронной сети, изображенная на рис. 64.

На этом рисунке символом S обозначен нейрон с сигмоидальной функцией активации, а символом L – с пороговой функцией. Нумерация нейронов – сверху вниз, слева направо.

Соответствие номеров входных воздействий и клеток поля, которые «видит» муравей, приведено на рис. 65.

Результаты экспериментов представлены в табл. 6.

Таблица 6. Результаты эксперимента

Число состояний	2	4	8	16
Максимальное значение функции приспособленности	26.48	26.65	27.80	25.82

На рис. 66–73 приведены графики эволюции для системы, состоящей из нейронной сети указанного выше вида рис. 64 и автомата Мили с различным числом состояний.

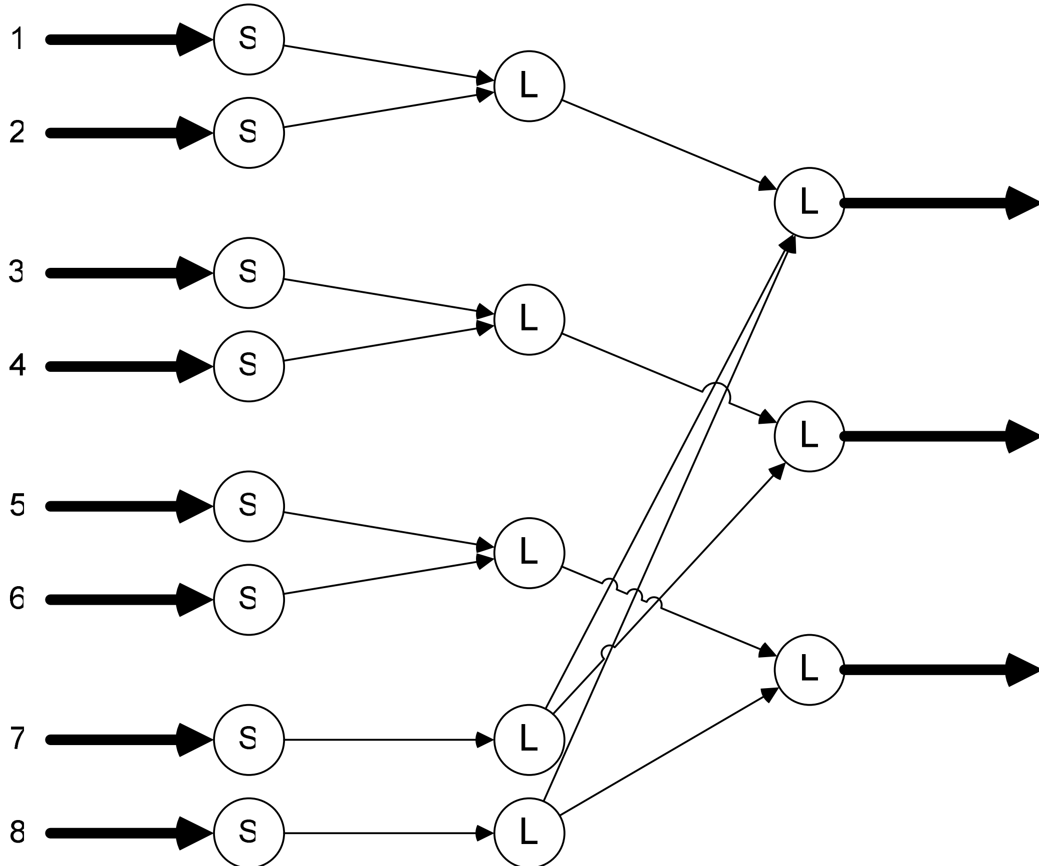


Рис. 64. Структура нейронной сети

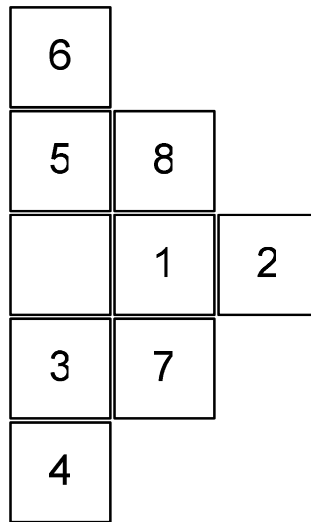


Рис. 65. Структура нейронной сети

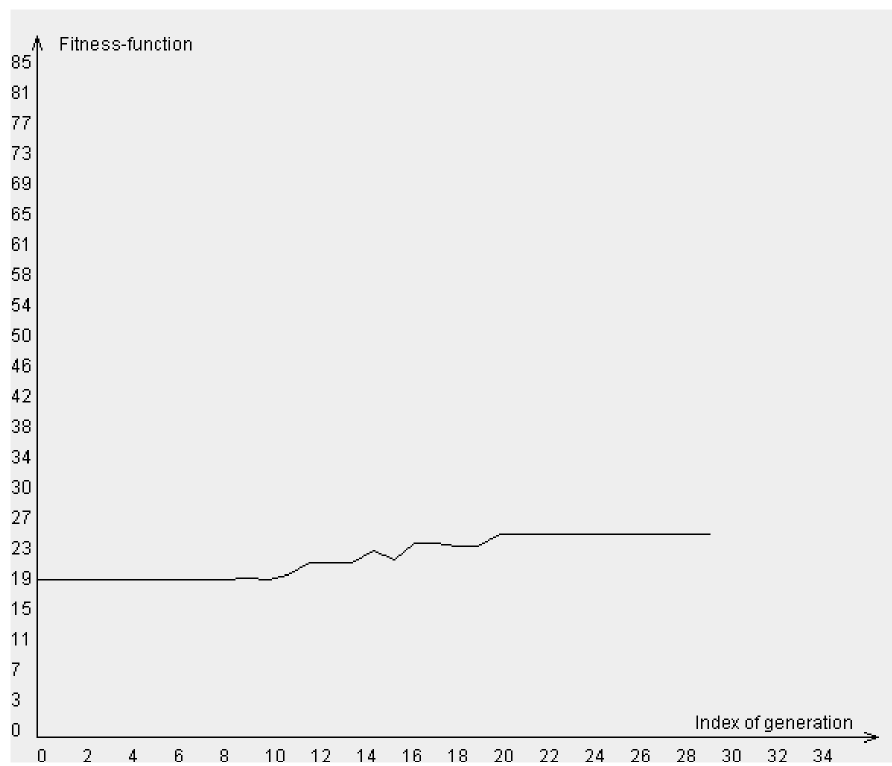


Рис. 66. Ход эволюции лучшей особи для автоматов с двумя состояниями

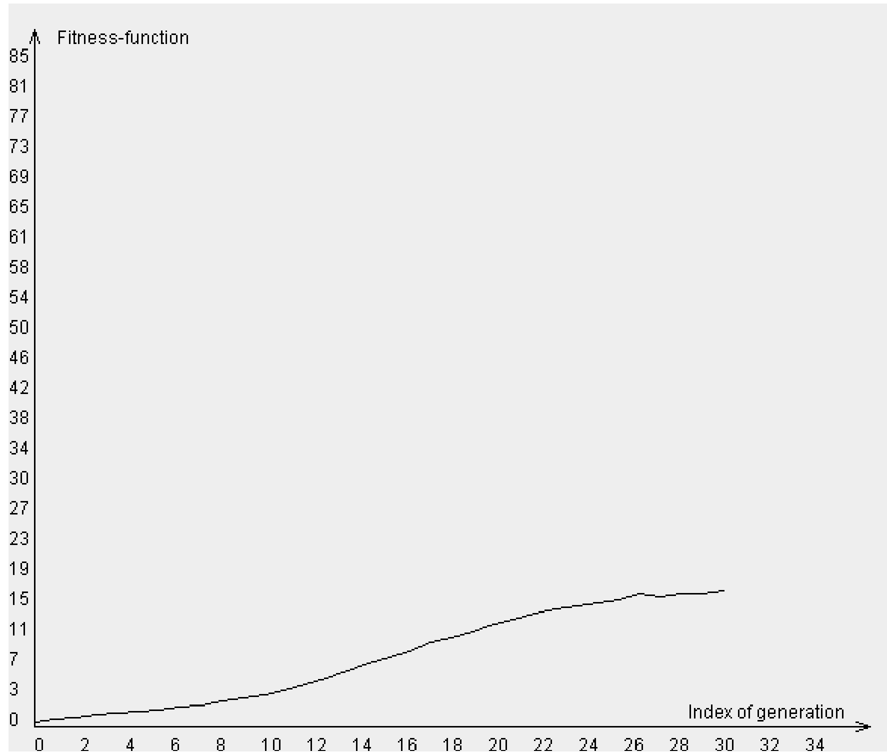


Рис. 67. Среднее значение функции приспособленности в зависимости от поколения для автоматов с двумя состояниями

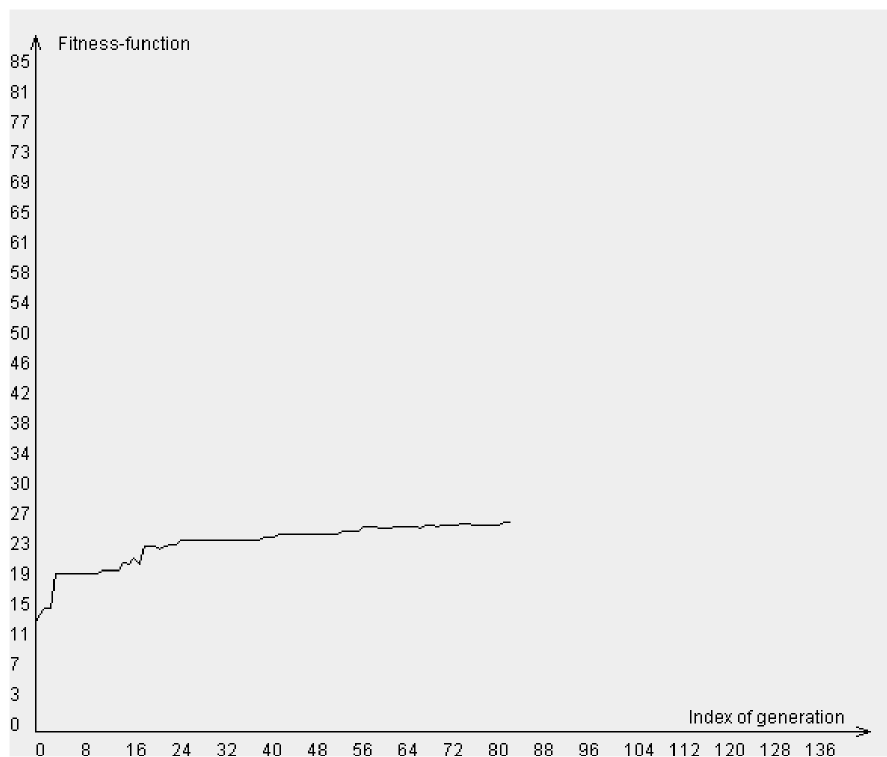


Рис. 68. Ход эволюции лучшей особи для автоматов с четырьмя состояниями

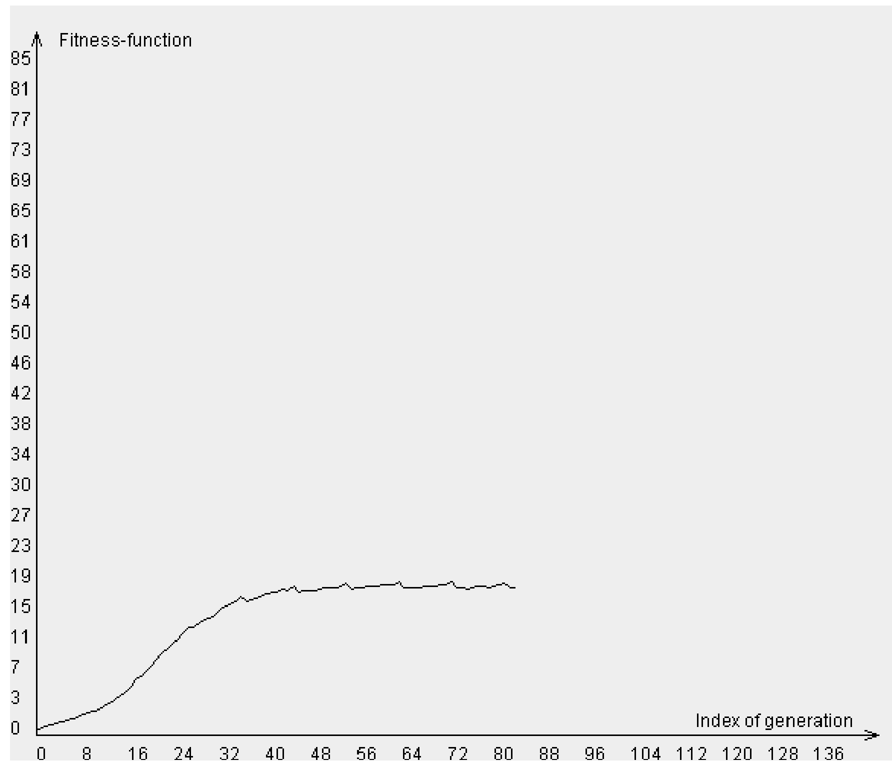


Рис. 69. Среднее значение функции приспособленности в зависимости от поколения для автоматов с четырьмя состояниями

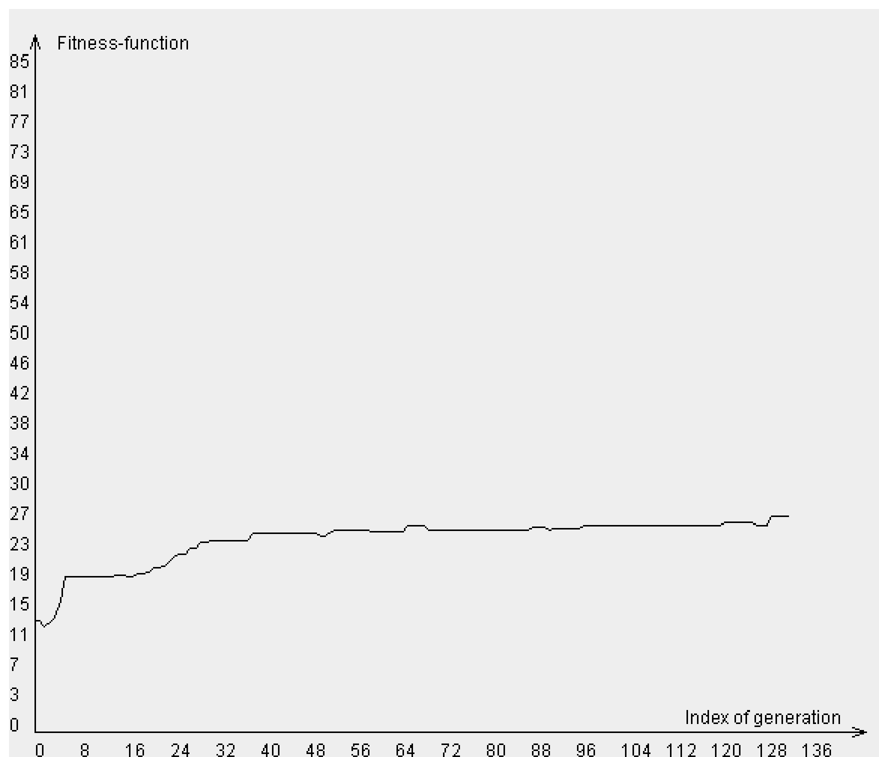


Рис. 70. Ход эволюции лучшей особи для автоматов с восемью состояниями

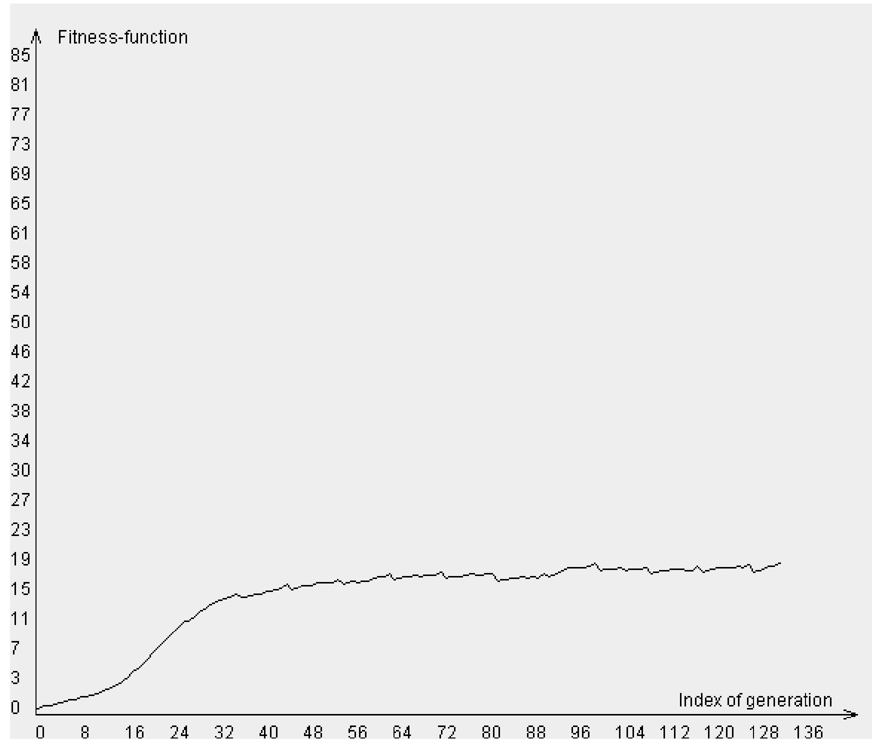


Рис. 71. Среднее значение функции приспособленности в зависимости от поколения для автоматов с восемью состояниями

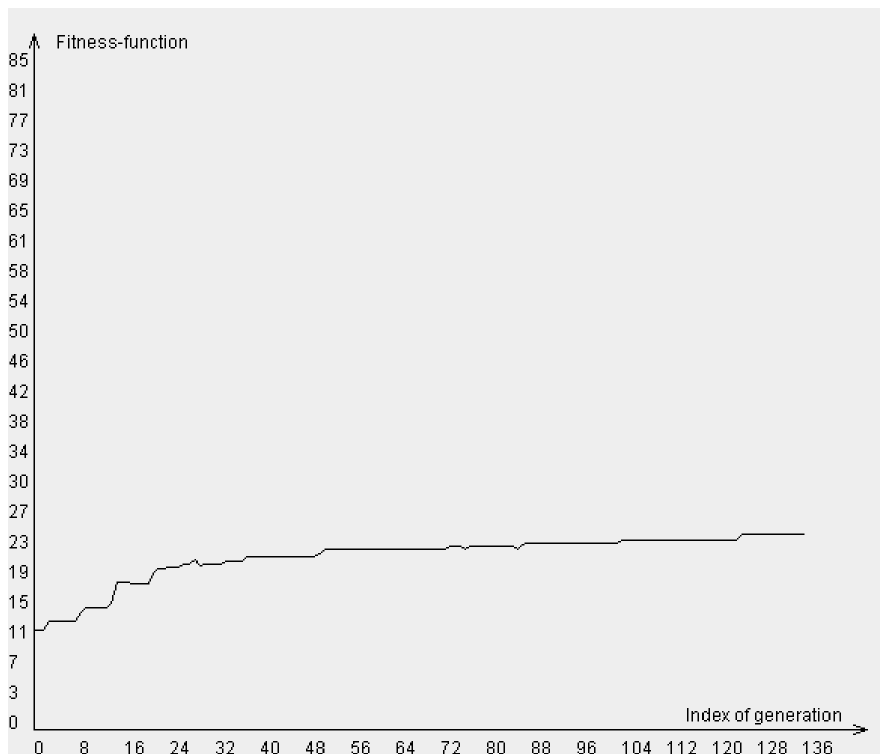


Рис. 72. Ход эволюции лучшей особи для автоматов с шестнадцатью состояниями

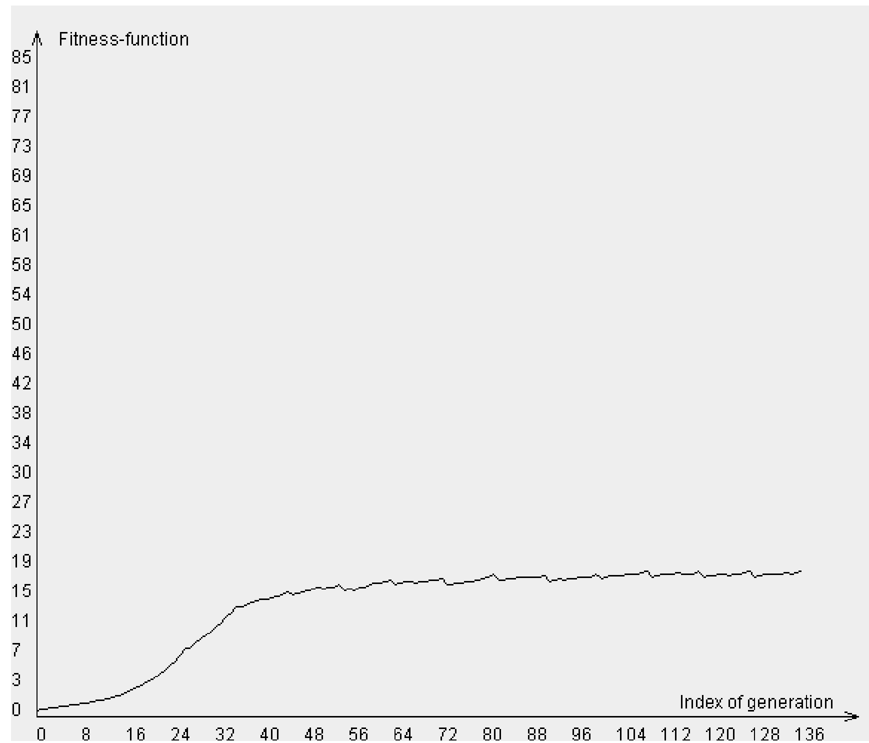


Рис. 73. Среднее значение функции приспособленности в зависимости от поколения для автоматов с шестнадцатью состояниями

Особь, изображенная на рис. 74 (нейронная сеть) и на рис. 75 (конечный автомат), позволяет муравью в среднем съесть 26 яблок.

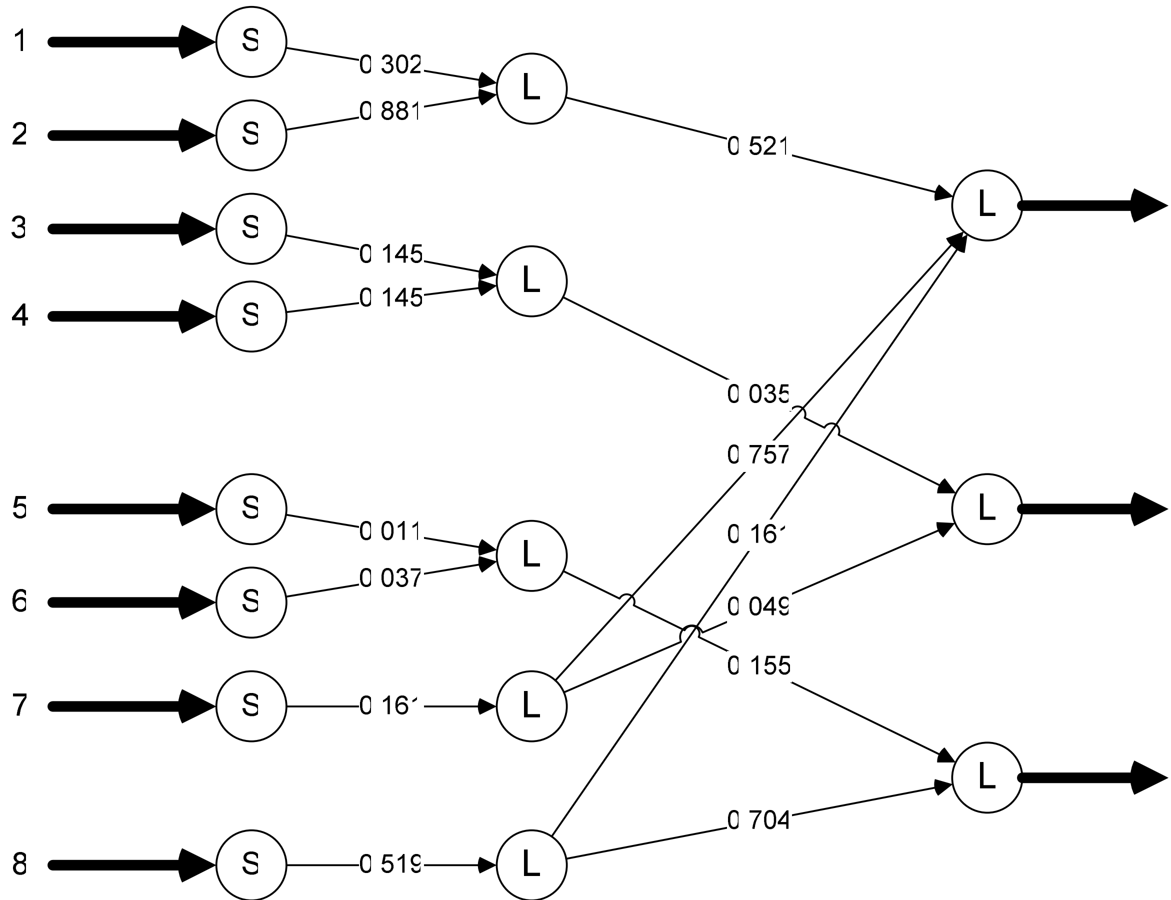


Рис. 74. Нейронная сеть особи

Значения пределов активации для нейронов с пороговой функцией активации приведены в табл. 7 (нейроны нумеруются по столбцам в соответствии с тем, как они изображены на рис. 74).

Таблица 7. Значения пределов активации

Номер порогового нейрона	1	2	3	4	5	6	7	8
Значение предела активации	0.16	0.90	0.21	0.50	0.95	0.18	0.79	0.06

Двумя окружностями на диаграмме переходов (рис. 75) обозначено начальное состояние автомата Мили.

Пометки на ребрах имеют вид *{значение переменных}, действие*. При этом переменные нумеруются таким образом, что переменной с меньшим номером соответствует выход нейронной сети, изображенный выше на рис. 74.

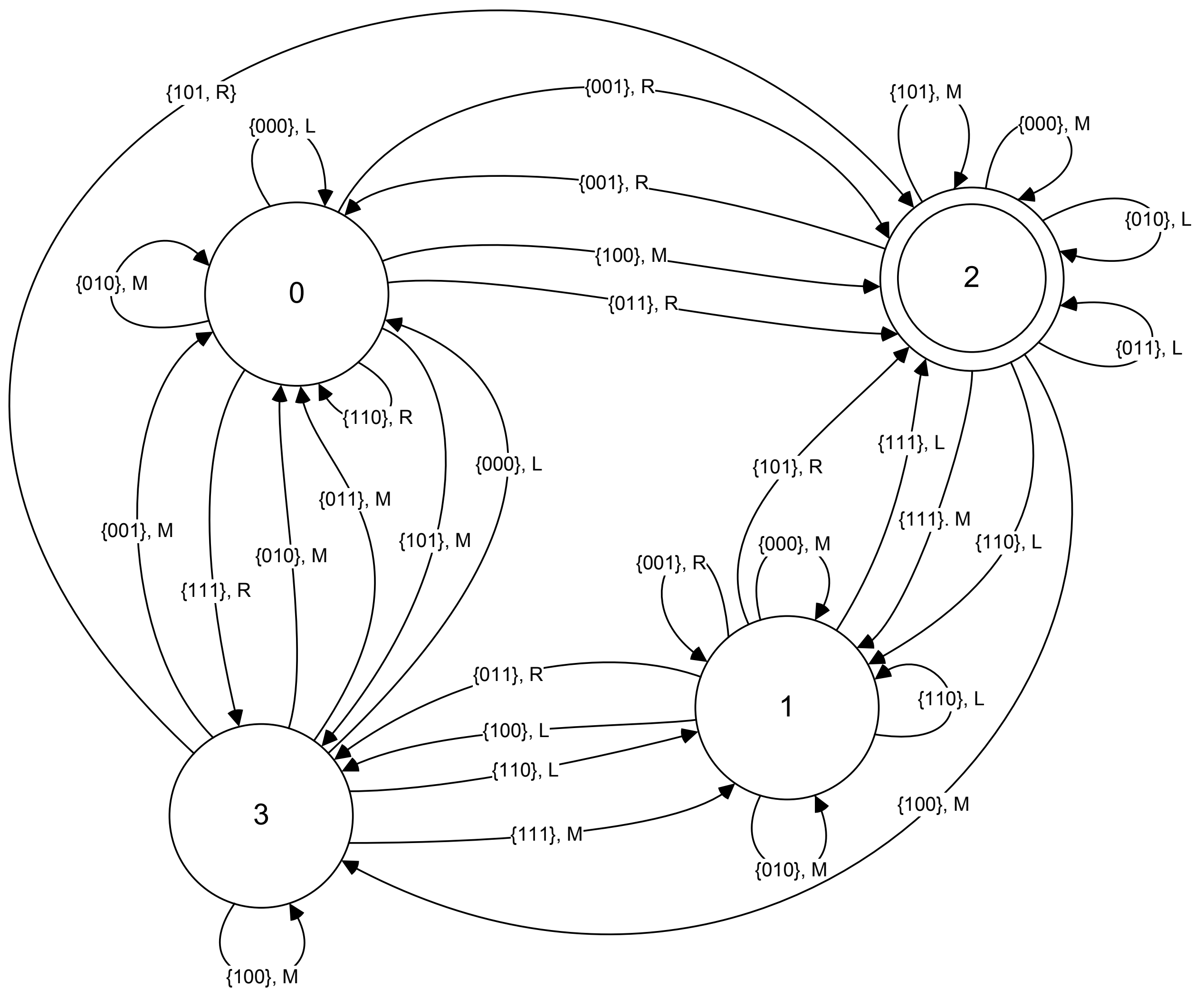


Рис. 75. Диаграмма переходов автомата Мили из четырех состояний

2.3. Выводы

Описаны прототипы трех программных средств для построения автоматов управления системами со сложным поведением. Для каждого средства даны рекомендации по его установке и настройке, а также приведен пример его применения.

На основании результатов экспериментов, проведенных на третьем этапе работ, в программное средство *3Genetic* были внесены изменения и дополнения, отраженные в скорректированной программной документации.

ЗАКЛЮЧЕНИЕ

В результате исследований, выполненных на четвертом (заключительном) этапе работ по контракту, были разработаны методы генерации автоматов управления, прототипы программных средства для их поддержки и методические рекомендации по их использованию для генерации автоматов управления системами со сложным поведением.

В первой главе разработаны три метода представления автоматов в виде хромосом генетических алгоритмов: метод сокращенных таблиц переходов, метод представления автоматов деревьями решений и метод совместного применения конечных автоматов и нейронных сетей. Кроме этого разработаны прототипы трех программных средств: *GAAP*, *AutoAnt* и *3Genetic*, и приведены методические рекомендации по их применению для генерации автоматов управления системами со сложным поведением. На основании результатов экспериментов, проведенных на третьем этапе работ, в программное средство *3Genetic* были внесены изменения и дополнения, отраженные в скорректированной программной документации.

Во второй главе приведены рекомендации по установке и настройке предложенных программных средств. Описывается применение этих программных средств для генерации управляющих автоматов в задаче «Умный муравей-3».

Реализация разработанных методов в программных средствах позволила подтвердить их эффективность. На основе экспериментов были сформулированы рекомендации по применению разработанных методов для генерации автоматов управления системами со сложным поведением.

Таким образом, были решены все задачи, поставленные в техническом задании на проведение четвертого этапа работы.

Результаты выполненных работ, а также патентных исследований, позволяют утверждать, что научно-технический уровень исследований соответствует уровню исследований в рассматриваемой области, проводимых в лучших исследовательских центрах мира.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. *Angeline P., Pollack J.* Evolutionary Module Acquisition / Proceedings of the Second Annual Conference on Evolutionary Programming. Cambridge: MIT Press. 1993, pp.154–163. http://www.demo.cs.brandeis.edu/papers/ep93.pdf
2. *Andre D., Bennet F., Koza J.* Discovery by Genetic Programming of a Cellular Automata Rule that is Better than any Known Rule for the Majority Classification Problem. 1996. http://citeseer.ist.psu.edu/andre96discovery.html
3. *Bryant K.* Genetic Algorithms and the Traveling Salesman Problem. Harvey Mudd College: Department of Mathematics, 2000. http://www.math.hmc.edu/math197/archives/2001/kbryant/kbryant-2001-thesis.pdf
4. *Chambers L.* Practical Handbook of Genetic Algorithms. Complex Coding Systems. Volumes I. II, III. CRC Press, 1999.
5. *De Jong K.* An analysis of the behavior of a class of genetic adaptive systems. PhD thesis. Univ. Michigan. Ann Arbor, 1975.
6. *Ferreira C.* Gene Expression Programming: A New Adaptive Algorithm for Solving Problems /Complex Systems. 2001. Vol. 13, issue 2, pp. 87–129. http://arxiv.org/pdf/cs.AI/0102027.pdf
7. *Fogel B., Fogel J., Atmar W.* Meta-Evolutionary Programming. In Chen, R. (ed.) / Proceedings of the 25th Asilomer Conference on Signals, Systems and Computers, CA: Maple Press. 1991. pp. 540–545. http://citeseer.ist.psu.edu/context/24073/0
8. *Frey C., Leugering G.* Evolving Strategies for Global Optimization. A Finite State Machine Approach / Genetic and Evolutionary Computation Conference (GECCO-2001). Morgan Kaufmann, 2001, pp. 27–33.
9. *Gustafson S.* An Analysis of Diversity in Genetic Programming. Ph.D. Dissertation. School of Computer Science and Information Technology. University of Nottingham. Nottingham. U.K., 2004. http://citeseer.ist.psu.edu/gustafson04analysis.html
10. *Huang D.* MS Thesis Preproposal: Adaptive Incremental Fitness Evaluation in Genetic Algorithms. 2005. NY: Rochester. http://www.cs.rit.edu/~dxh6185/downloads/MS_Thesis/Documents/Presentation.pdf
11. *Jefferson D., Collins R., Cooper C., Dyer M., Flowers M., Korf R., Taylor C., Wang A.* The Genesys System: Evolution as a Theme in Artificial Life /Proceedings of Second Conference on Artificial Life. MA: Addison-Wesley. 1992, pp.549–578. www.cs.ucla.edu/~dyer/Papers/AlifeTracker/Alife91Jefferson.html
12. *Jefferson. D., Collins. R., Cooper. C., Dyer. M., Flowers. M., Korf. R., Taylor. C., Wang. A.* Evolution as a theme in artificial life: The genesys/tracker system / Artificial Life II: Proceedings of the Workshop on Artificial Life. In Langton: Addison-Wesley. 1992, pp. 549–578.
13. *Jefferson D., Collins R., Cooper C., Dyer M., Flowers M., Korf R., Taylor C., Wang A.* The Genesys System: Evolution as a Theme in Artificial Life /Proceedings of Second Conference on Artificial Life. MA: Addison-Wesley. 1992, pp.549–578. www.cs.ucla.edu/~dyer/Papers/AlifeTracker/Alife91Jefferson.html
14. *Juillie H., Pollack J.* Coevolving the “Ideal” Trainer: Application to the Discovery of Cellular Automata Rules. 1998. http://citeseer.ist.psu.edu/16712.html
15. *Kantschik W., Dittrich P., Brameier M., Banzhaf W.* Meta-Evolution in Graph GP / Genetic Programming: Second European Workshop (EuroGP'99). 1999.

16. *Koza J.* Genetic Evolution and Co-Evolution of Computer Programs / Proceedings of Second Conference on Artificial Life. Redwood City, CA: Addison-Wesley. 1992. pp. 603-629. http://citeseer.ist.psu.edu/177879.html
17. *Koza J.* Genetic programming. On the Programming of Computers by Means of Natural Selection. MA: The MIT Press, 1998.
18. *Koza J. R.* Future Work and Practical Applications of Genetic Programming. Handbook of Evolutionary Computation. Bristol: IOP Publishing Ltd, 1997.
19. *Linton R.* Adapting binary fitness functions in genetic algorithms / Proceedings of the 42nd annual Southeast regional conference. NY: ACM Press. 2004, pp. 391–395.
20. *McCulloch W. S., Pitts W.* A logical calculus of the ideas immanent in nervous activity // Bulletin of Mathematical Biophysics, 1943, 5, pp. 115–137.
21. *Miller B., Goldberg M.* Genetic algorithms, tournament selection, and the effects of noise // Complex Systems. 1995. V. 9, I. 3, pp. 193–212.
22. *Mitchell M., Crutchfield P., Hraber T.* Evolving cellular automata to perform computations. Physica D. 75. 1993, pp. 361–391. http://web.cecs.pdx.edu/~mm/mech-imped.pdf
23. *Remote Method Invocation.* http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp
24. *Whitley D., Rana S., Heckendorn R.* The Island Model Genetic Algorithm: On separability, Population Size and Convergence. 1998. http://citeseer.ist.psu.edu/whitley98island.html
25. *Бедный Ю.Д., Шалыто А.А.* Применение генетических алгоритмов для построения автоматов в задаче «Умный муравей». СПбГУ ИТМО. 2007. http://is.ifmo.ru/works/ant/
26. *Гладков Л. А., Курейчик В. В., Курейчик В. М.* Генетические алгоритмы. М.: Физматлит. 2006.
27. *Данилов В.Р.* Технология генетического программирования для генерации автоматов управления системами со сложным поведением. Бакалаврская работа, СПбГУ ИТМО, 2007. http://is.ifmo.ru/works/danilov_bachelor
28. *Лобанов П. Г., Шалыто А. А.* Использование генетических алгоритмов для автоматического построения конечных автоматов в задаче о “Флибах” / Сборник докладов 4-й Всероссийской научной конференции «Управление и информационные технологии» (УИТ-2006). СПбГЭТУ «ЛЭТИ». 2006. с.144–149. http://is.ifmo.ru/works/flib
29. *Промежуточный отчет по этапу I* «Выбор направления исследований и базовых компонентов». http://is.ifmo.ru/genalg/_2007_01_report-genetic.pdf
30. *Промежуточный отчет по этапу II* «Теоретические исследования поставленных перед НИР задач». http://is.ifmo.ru/genalg/_2007_02_report-genetic.pdf
31. *Промежуточный отчет по этапу III* «Экспериментальные исследования поставленных перед НИР задач». http://is.ifmo.ru/genalg/_2007_03_report-genetic.pdf
32. *Рассел С., Норвиг П.* Искусственный интеллект. Современный подход. М.: Вильямс. 2006.
33. *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
34. *Шалыто А. А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.