М. Тим Джонс

Программирование искусственного интеллекта в приложениях

- Методы и технологии искусственного интеллекта
- Системы, основанные на правилах
- Генетические алгоритмы
- Нейронные сети
- Нечеткая логика

ДМК Пресс
Издательство
CHARLES RIVER MEDIA

М. Тим Джонс

Программирование искусственного интеллекта в приложениях

Второе издание

AI Application Programming

M. Tim Jones

CHARLES RIVER MEDIA, INC.
Hingham, Massachusetts

Программирование искусственного интеллекта в приложениях

М. Тим Джонс

Второе издание

ДМК Пресс
издательство

Москва, 2011

УДК 004.8
ББК 32.813
Д42

Джонс М. Т.
Д42 Программирование искусственного интеллекта в приложениях / М. Тим Джонс ; Пер. с англ. Осипов А. И. – М. : ДМК Пресс, 2011. – 312 с.: ил.

ISBN 978-5-94074-746-8

Данная книга посвящена вопросам искусственного интеллекта (ИИ), то есть методам и технологиям, призванным сделать ПО более умным и полезным. Рассмотренные алгоритмы в основном предназначены для встраивания в другое программное обеспечение, что позволяет создавать программы, гибко подстраивающиеся под требования и привычки пользователя.

Здесь описан ряд алгоритмов ИИ – нейронные сети, генетические алгоритмы, системы, основанные на правилах, нечеткая логика, алгоритмы муравья и умные агенты. Для каждого алгоритма приведены примеры реализации. Некоторые из этих приложений применяются на практике, другие относятся скорее к теоретическим изысканиям. Так или иначе, автор раскрывает секреты наиболее интересных алгоритмов ИИ, что делает их доступными для более широкой аудитории. Предполагается, что благодаря подробному описанию алгоритмов методики и технологии ИИ займут свое место в списке традиционных программ.

Книга призвана помочь разработчикам использовать технологии ИИ при создании более умного программного обеспечения.

ISBN 1-58450-278-9 (англ.)
ISBN 978-5-94074-746-8 (рус.)

Содержание

Эта книга посвящается моей жене Джилл (Jill) и детям Меган (Megan), Элизе (Elise) и Марку (Marc). Их терпение и поддержка сделали возможным написание данной книги.

Благодарности

Над созданием данной книги работало очень много людей. Когда я писал примеры для книги, алгоритмы разрабатывались и развивались большой группой исследователей и практиков. Хочется особо отметить работу Алана Туринга (Alan Turing), Джона МакКарти (John McCarthy), Артура Самуэля (Arthur Samuel), Н. Метрополис (N. Metropolis), Гейла Карпентера (Gail Carpenter), Стефена Гроссберга (Stephen Grossberg), Марко Дориго (Marco Dorigo), Дэвида Румельхарта (David Rumelhart), Джоффри Хинтона (Geoffrey Hinton), Джона Ван Ньюмена (John van Neumann), Дональда Хеббса (Donald Hebbs), Тейво Конена (Teuvo Kohonen), Джона Хопфилда (John Hopfield), Уоррена МакКуллоча (Warren McCulloch), Вальтера Питтса (Walter Pitts), Марвина Мински (Marvin Minski), Сеймура Паперта (Seymour Papert), Джона Холланда (John Holland), Джона Коза (John Koza), Томаса Бэка (Thomas Back), Брюса МакЛеннана (Bruce MacLennan), Патрика Уинстона (Patrick Winston), Чарльза Форги (Charles Forgy), Лотфи Заде (Lotfi Zadeh), Родни Брукса (Rodney Brooks), Андрея Маркова (Andrey Markov), Джеймса Бэйкера (James Baker), Дуга Лената (Doug Lenat), Клода Шэннона (Claud Shannon) и Алана Кэй (Alan Kay). Также хочу поблагодарить Дэна Клэйна (Dan Klein) за полезные обзоры ранних изданий данной книги.

Вступление

Эта книга посвящена вопросам искусственного интеллекта (ИИ), который также называют «слабым ИИ», или методам и технологиям, призванным сделать программное обеспечение более умным и полезным. Ранние разработки ИИ были ориентированы на создание умных машин, которые копировали поведение человека (по-другому эти системы назывались «сильным ИИ»), однако в настоящее время большинство исследователей и разработчиков ИИ преследуют более практичные цели. Эти задачи включают встраивание алгоритмов и технологий ИИ в программное обеспечение, что позволяет создавать программы, гибко подстраивающиеся под требования и привычки пользователя.

В данной книге описывается ряд алгоритмов ИИ, а также подробно рассматривается их работа. В число этих алгоритмов входят нейронные сети, генетические алгоритмы, системы, основанные на продукционных правилах, нечеткая логика, алгоритмы муравья и умные агенты. Для каждого алгоритма приведены примеры приложений. Некоторые из этих приложений полезны для практического применения, другие относятся скорее к теоретическим изысканиям. Так или иначе, эта книга раскрывает секреты наиболее интересных алгоритмов ИИ, что

делает их доступными для более широкой аудитории. Я надеюсь, что благодаря подробному описанию алгоритмов в данной книге методики и технологии ИИ займут свое место в списке более традиционных программ. ИИ будет развиваться по-настоящему только при условии, что ему найдется практическое применение. Поэтому я рассчитываю, что эта книга поможет разработчикам использовать технологии ИИ при создании более умного программного обеспечения. Со мной вы можете связаться по адресу mtj@mtjones.com.

Обозначения, используемые в книге

В книге используются следующие шрифтовые выделения:

- *курсивом* помечены смысловые выделения в тексте;
- названия переменных, команд и других элементов языка программирования набраны `моноширинным шрифтом`;
- специальным стилем оформлена дополнительная информация;
- все Internet-адреса подчеркнуты;
- каждый листинг имеет порядковый номер (например, листинг 1.1). Тот же код без номера можно загрузить с сайта издательства «ДМК Пресс» www.dmk.ru.

Глава 1. История искусственного интеллекта

В этой вводной главе представлено краткое обсуждение искусственного интеллекта и вкратце описана история создания современного ИИ и научные изыскания ряда разработчиков, которые внесли свой вклад в формирование концепции ИИ. В конце главы рассмотрена структура книги, включая изложенные здесь методики и технологии.

Что такое искусственный интеллект

Искусственным интеллектом, или ИИ (Artificial Intelligence – AI), называют процесс создания машин, которые способны действовать таким образом, что будут восприниматься человеком как разумные. Это может быть повторение поведения человека или выполнение более простых задач, например, выживание в динамически меняющейся обстановке.

Для некоторых исследователей результат данного процесса состоит в том, чтобы научиться лучше понимать нас самих. Для других это база, на основе которой можно научить искусственные системы вести себя разумно. В любом случае ИИ обладает таким потенциалом для изменения мира, которого нет ни у одной другой технологии.

В период становления ИИ его разработчики обещали достичь очень многого, а добились несравненно меньшего. В то время создание разумных систем казалось очень простой задачей, которая так и не была решена. В наше время цели создания ИИ стали намного практичнее. ИИ был разделен на несколько частей, имеющих различные цели и средства их достижения.

Проблема ИИ заключается в том, что технологии, которые исследуются в его рамках, становятся обычными сразу после их внедрения. Например, построение машины, которая смогла бы различать человеческую речь, когда-то считалось частью разработки ИИ. Теперь такие технологии, как нейронные сети и скрытые модели Маркова, уже никого не удивляют и не рассматриваются как разработка ИИ. Родни Брукс (Rodney Brooks) описывает этот феномен как «эффект ИИ». После того как технология ИИ находит применение, она перестает быть технологией ИИ. Поэтому сочетание букв ИИ также получило расшифровку как «Почти применено» (Almost Implemented), поскольку после своего создания эта технология перестает быть чудом и используется повсеместно.

Сильный и слабый ИИ

Так как искусственный интеллект по-разному понимается разными людьми, было принято решение использовать другую классификацию. *Сильный ИИ* (Strong AI) представляет собой программное обеспечение, благодаря которому компьютеры смогут думать так же, как люди. Помимо возможности думать, компьютер обретет и сознание разумного существа.

Слабый ИИ (Weak AI) представляет собой широкий диапазон технологий ИИ. Эти функции могут добавляться в существующие системы и придавать им различные «разумные» свойства. Данная книга фокусируется на слабом ИИ и методиках, используемых для встраивания в другие системы.

Результат внедрения ИИ

Исследование ИИ привело к появлению многих технологий, которые мы сейчас принимаем как должное. Вспомните, что в ранние 1960-е разработки в области миниатюризации при создании космической системы «Аполлон» способствовали изобретению и внедрению интегрированных схем, которые играют такую важную роль в современных технологиях. Системы распознавания голоса и письма также обязаны своим возникновением ИИ.

В настоящее время многие коммерческие продукты включают технологии ИИ. Видеокамеры используют нечеткую логику, которая позволяет зафиксировать изображение при перемещении камеры. Нечеткая логика также нашла применение в посудомоечных машинах и других устройствах. Суть этих технологий не интересует массового потребителя, потому что люди желают получать устройства, которые работают, но не хотят знать, как именно они работают. Кроме того, здесь присутствует определенный фактор страха, который может повлиять на некоторых покупателей. Узнав, что прибор использует технологию ИИ, они просто могут отказаться от его покупки.

Примечание *Нечеткая логика подробно описывается в главе 9.*

История развития ИИ

Об истории и развитии ИИ можно написать очень много. В этом разделе представлены важнейшие вехи в развитии ИИ, а также кратко рассказано о людях, которые внесли в этот процесс свой вклад. Рассмотрение истории ИИ включает в себя современную точку зрения о том, как развивалась данная технология с 1940-х гг. (Stottler Henke, 2002).

Рождение компьютера, 1940-е

Эра разумных машин наступила вскоре после появления первых компьютеров. Большинство компьютеров были созданы для того, чтобы взломать немецкие шифры во время Второй мировой войны. В 1940 г. был построен первый

рабочий компьютер на электромагнитных реле – Робинсон (Robinson). Он предназначался для расшифровки военных переговоров немцев, которые были зашифрованы с помощью машины Энигма (Enigma). Робинсон назван в честь создателя мультипликационных трюков, Хита Робинсона (Heath Robinson). Три года спустя вакуумные трубки заменили электромагнитные реле, что позволило построить Колосс (Colossus). Этот более быстрый компьютер был создан для взлома новых усовершенствованных кодов. В 1945 г. в Университете Пенсильвании доктор Джон В. Мохли (Dr. John W. Mauchly) и Д. П. Экерт (J. P. Eckert, Jr.) разработали более известный компьютер, ENIAC. Его задачей было рассчитать баллистические таблицы времен Второй мировой войны.

Нейронные сети с обратной связью построены Вальтером Питтсом (Walter Pitts) и Уорреном МакКуллочем (Warren McCulloch) в 1945 г., чтобы показать возможности их применения при расчетах. Эти ранние сети были электронными и весьма поспособствовали росту энтузиазма у создателей технологии. Примерно в то же время Норберт Винер (Norbert Wiener) создал область кибернетики, которая включала математическую теорию обратной связи для биологических и инженерных систем. Важным аспектом данного открытия стала концепция о том, что разум – это процесс получения и обработки информации для достижения определенной цели.

Наконец в 1949 г. Дональд Хеббс (Donald Hebbs) открыл способ создания самообучающихся искусственных нейронных сетей. Этот процесс (получивший название «Обучение по Хеббсу») позволяет изменять весовые коэффициенты в нейронной сети так, что данные на выходе отражают связь с информацией на входе. Хотя использование этого метода не избавляло от проблем, почти все свободные процедуры обучения основаны на обучении по Хеббсу.

Рождение ИИ, 1950-е

1950-е отмечены в истории как годы рождения ИИ. Алан Тьюринг (Alan Turing) предложил специальный тест (который впоследствии получил название «Тест Тьюринга») в качестве способа распознать разумность машины. В этом тесте один или несколько людей должны задавать вопросы двум тайным собеседникам и на основании ответов определять, кто из них машина, а кто человек. Если не удавалось раскрыть машину, которая маскировалась под человека, предполагалось, что машина разумна. Существует и дополнение к тесту Тьюринга (так называемый «Приз Лебнера»), которое представляет собой соревнование по выявлению лучшего имитатора человеческого разговора.

В 1950-е гг. ИИ по своей природе был в первую очередь символичным. Именно в то время сделано открытие, что компьютеры могут управлять символами так же, как и числовыми данными. Это привело к разработке таких программ, как предназначавшаяся для доказательства теорем Logic Theorist (авторы – Ньюэлл (Newell), Симон (Simon) и Шоу (Shaw)) и General Problem Solver (создатели – Ньюэлл и Симон), использовавшаяся для анализа нерешаемых проблем. Наверное, самым крупным открытием в программной области в 1950-е было создание

Артуром Самуэлем программы для игры в шашки, которая постепенно научилась обыгрывать своего создателя.

В 1950-е гг. были также разработаны два языка ИИ. Первый, язык IPL, был создан Ньюэллом, Симоном и Шоу для программы Logic Theorist. IPL являлся языком обработки списка данных и привел к созданию более известного языка LISP. LISP появился в конце 1950-х и вскоре заменил IPL, став основным языком приложений ИИ. Язык LISP был разработан в лабораториях Массачусетсткого технологического института (MIT). Его автором был Джон МакКарти, один из первых разработчиков ИИ.

Джон МакКарти представил концепцию ИИ как часть своего предложения для Дормутской конференции по проблемам ИИ. В 1956 г. разработчики ИИ встретились в Дормутском колледже, чтобы обсудить дальнейшее развитие разумных машин. В своем предложении Джон МакКарти написал: «Задача заключается в том, чтобы работать на основе предположения, что любой аспект обучения или другой функции разума может быть описан так точно, чтобы машина смогла его симулировать. Мы попытаемся определить, как сделать так, чтобы машины смогли пользоваться языком, формулировать абстракции и концепции, решать задачи, которыми сейчас занимаются только люди, а также заниматься самообучением».

Дормутская конференция позволила впервые встретиться всем разработчикам ИИ, однако общее решение по ИИ принято не было.

В конце 1950-х Джон МакКарти и Марвин Мински основали в MIT лабораторию искусственного интеллекта, которая работает и по сей день.

Подъем ИИ, 1960-е

В 1960-е гг. произошел скачок в развитии ИИ, вызванный прогрессом в компьютерных технологиях, а также увеличением количества разработок в данной области. Наверное, самым важным показателем того, что ИИ достиг приемлемого уровня, стало появление критиков. К этому времени относится написание двух книг: «Компьютеры и здравый смысл: миф о мыслящих машинах» Мортимера Тауба (Mortimer Taub) и «Алхимия и ИИ» Хуберта и Стюарта Дрейфус (Hubert and Stuart Dreyfus).

В 1960-е наиболее важным было представление знаний, так как сильный ИИ продолжал оставаться главной темой в разработках ИИ. Были построены игрушечные миры, например, «Blocks Microworld Project» Мински и Паперта в MIT, а также SHRDLU Терри Винограда (Terry Winograd). С помощью этих миров создавалась окружающая среда, в которой тестировались идеи по компьютерному зрению, роботехнике и обработке человеческого языка.

В начале 1960-х Джон МакКарти основал лабораторию ИИ в Стэндфордском университете. Сотрудники лаборатории, помимо прочего, создали робота Шейки (Shakey), который мог перемещаться по искусственному миру и выполнять простые приказания.

Разработки нейронных сетей процветали до конца 1960-х, ровно до тех пор, пока не была издана книга Мински и Паперта «Перцептроны: введение в вычислительную геометрию». Авторы описали ограничения по использованию простых

одноуровневых перцептронов, что привело к серьезному снижению инвестиций в исследования нейронных сетей более чем на декаду.

Пожалуй, самым интересным аспектом ИИ в 1960-е стало изображение ИИ в романе Артура Кларка и фильме Стенли Кубрика «Космическая Одиссея 2001 года». HAL, разумный компьютер, движимый безумием и желанием выжить, уничтожил большую часть команды космического корабля, находящегося на орбите Юпитера.

Спад исследований ИИ, 1970-е

1970-е гг. показали резкий спад интереса к ИИ после того, как исследователям не удалось выполнить нереальные обещания его успеха. Практическое применение ИИ было по-прежнему минимальным. Ситуация осложнялась тем, что в MIT, Стэнфорде и Карнеги Меллон возникли проблемы с финансированием исследований в области ИИ. Такое же препятствие ожидало и разработчиков в Великобритании. К счастью, исследования продолжались и не без успеха.

Дуг Ленет в Стэндфордском университете создал программу «Автоматический математик» (AM) и позднее – EURISKO, что позволило открыть новые теории в математике. AM успешно повторил открытие теории чисел, но по причине ограниченного количества кодировок в эвристике быстро достиг потолка своих возможностей. EURISKO, следующая разработка Ленета, был построен с учетом ограничений AM и мог определять свою эвристику, а также решать, что в ней является полезным, а что нет.

Впервые нечеткая логика была применена на практике в начале 1970-х гг. Следует отметить, что Лотфи Заде (Lotfi Zadeh) создал саму концепцию еще в 1960-х. Нечеткая логика была использована в работе над паровым двигателем в колледже Королевы Марии (Queen Mary), что стало первым из многочисленных примеров применения нечеткой логики для управления процессами.

В 1970-х продолжалось создание языков для ИИ. Был разработан язык ПРОЛОГ (Prolog – Программирование логики). Язык ПРОЛОГ предназначался для разработки программ, которые управляли символами (а не выполняли числовые расчеты) и работал с правилами и фактами. В то время как ПРОЛОГ распространялся за пределами США, язык LISP сохранял свой статус основного языка для приложений ИИ.

Развитие ИИ для игр продолжилось в 1970-х. В Карнеги Меллон была создана программа для игры в триктрак. Она играла настолько хорошо, что смогла победить чемпиона мира по триктраку, Луиджи Вилла (Luigi Villa) из Италии. Это был первый случай победы компьютера над человеком в сложной игре.

Подъем и спад ИИ, 1980-е

1980-е гг. казались многообещающими для ИИ после того, как продажи аппаратных средств и программного обеспечения, связанного с ИИ, превысили 400 млн. долларов в 1986 г. Большую часть этого дохода принесли продажи экспертных систем на LISP и, соответственно, LISP-машин, которые развивались, становясь лучше и дешевле.

Экспертные системы использовались многими компаниями для разработки ископаемых, прогнозирования инвестиций, а также весьма специфических

задач, например, диагностики электровоза. Также были идентифицированы ограничения в работе экспертных систем, поскольку их знания становились все больше и сложнее. Например, системный конфигуратор XCON компании Digital Equipment Corporation достиг предела в 10000 правил, и поддерживать его работу стало очень сложно.

Нейронные сети в 1980-е гг. также переживали возрождение. Они нашли применение при решении ряда различных задач, таких как распознавание речи и возможность самообучения машин.

К сожалению, 1980-е продемонстрировали как рост, так и спад интереса к ИИ. Основной причиной этого были сбои экспертных систем. Однако многие другие приложения ИИ были серьезно улучшены именно в 1980-е гг. Например, системы распознавания речи смогли наконец работать независимо от говорящего (то есть распознавать речь любого человека, а не только специально обученного диктора), а также выполнять распознавание в режиме реального времени (позволяя человеку говорить обычно, а не делать остановки и паузы). Кроме того, значительно увеличился их словарный запас.

Постепенный прогресс ИИ, 1990-е и настоящее время

1990-е гг. стали новой эпохой в развитии приложений слабого ИИ. Было обнаружено, что создание продукта, включающего элементы ИИ, является интересной задачей, поскольку позволяет добиться решения многих проблем быстрее и более эффективно, чем при использовании традиционных методов. Поэтому элементы ИИ были интегрированы в ряд приложений (по материалам журнала Stottler Henke, 2002):

- системы распознавания фальшивых кредитных карт;
- системы распознавания лиц;
- системы автоматического планирования;
- системы предсказания прибыли и потребности в персонале;
- конфигурируемые системы «добычи данных» из баз данных;
- системы персонализации.

Важным событием в развитии компьютерных игр с использованием ИИ стало создание в 1997 г. суперкомпьютера для игры в шахматы Deep Blue (он был разработан в Карнеги Меллон). Эта машина смогла победить Гарри Каспарова, чемпиона мира по шахматам.

Другое интересное событие для развития ИИ в 1990-е гг. произошло в 60 млн. миль от Земли. Была создана система Deep Space 1 (DS1), которая могла тестировать технологии 12-ой степени риска, включая полет кометы и тестирование для будущих космических полетов. DS1 включала систему искусственного интеллекта под названием Remote Agent, которой на небольшое время предоставлялось управление космическим кораблем. Обычно такая работа исполнялась командой ученых посредством терминалов. Remote Agent продемонстрировала, что искусственная система способна управлять сложным космическим кораблем, позволяя ученым и экипажам кораблей сконцентрироваться на решении других задач.

Направления ИИ

Выделение уникальных направлений в технологиях и методиках ИИ представляется достаточно сложной задачей, поэтому в табл. 1.1 приведен стандартный подход. Слева представлены возникшие перед исследователями проблемы, справа – пути их решения, а сам список является хорошей отправной точкой для понимания особенностей ИИ.

Таблица 1.1. Направления ИИ (по материалам сайта «AI FAQ»)

Проблемы	Пути решения
Автоматическое программирование	Определение поведения с тем, чтобы позволить системе ИИ написать программу
Сети Байезана (Bayesian)	Построение сетей на основании вероятностей
Решение проблемы ограничений	Решение переборных задач с помощью различных методик оптимизации поиска
Построение структуры знания	Модификация человеческих знаний в форму, которую сможет понять компьютер
Обучение машин	Создание программ, которые учатся на своем опыте
Нейронные сети	Моделирование программ, которые имеют структуру, схожую с человеческим мозгом
Планирование	Системы, которые способны идентифицировать наилучшую последовательность действий для достижения заданной цели
Поиск	Поиск пути от начальной точки к заданной цели

Все эти темы рассмотрены в данной книге. Описание каждой технологии сопровождается примерами на языке C, которые помогут лучше понять принцип работы того или иного алгоритма.

Основоположники

Хотя многие разработчики внесли свой вклад в развитие ИИ, в данном разделе мы попытаемся рассказать о пионерах этого направления и их достижениях.

Алан Тьюринг

Английский математик Алан Тьюринг (Alan Turing) впервые высказал идею о том, что все проблемы, решаемые людьми, могут быть сведены к набору алгоритмов. Это позволило родиться другой идее – сама мысль может быть сведена к алгоритму, а поэтому машины могут имитировать мышление и, возможно, даже сознание людей. Придя к этому заключению, Алан Тьюринг создал обучающуюся систему «Машина Тьюринга», которая могла имитировать работу любой другой компьютерной системы. Позднее исследователь предложил специальный тест для распознавания разумности машины.

Джон МакКарти

Джон МакКарти (John McCarthy) продолжает свою работу и сейчас в Стэндфордском университете. Он был одним из основателей лаборатории ИИ в MIT,

основал лабораторию ИИ в Стэнфорде, а также организовал первую конференцию по ИИ в 1956 году (Дормутская конференция). Его исследования привели к созданию языка LISP, который сегодня считается основным языком для разработки программного обеспечения ИИ. Ранним открытием Джона МакКарти были компьютерные системы, которые могли математически доказывать корректность компьютерных программ.

Марвин Мински

Марвин Мински (Marvin Minsky) был одним из самых успешных разработчиков в сфере ИИ, а также во многих других сферах. Сейчас он профессор в MIT, где вместе с Джоном МакКарти в 1958 году основал лабораторию ИИ. Марвин Мински является автором ряда работ различной тематики, включая нейронные сети, представление знаний и психологию познания. Он создал концепцию кадров, которая моделировала феномен познавательной способности, языкового восприятия и визуального предчувствия. Профессор Мински также построил первый компьютер на базе нейронных сетей.

Артур Самуэль

Артур Самуэль (Arthur Samuel, 1901–1990) стал одним из первых в сфере обучения машин и искусственного интеллекта. Его карьера была долгой и интересной. Учитель и инженер, он известен своими достижениями в разных областях. Более всего Самуэль известен как автор программы для игры в шашки, разработанной в 1957 г. Это был один из первых примеров разумной программы, которая играла в сложную игру. Программа смогла обыграть не только самого Самуэля, но и четвертого призера чемпионата страны (США) по шашкам. Работы Самуэля по обучению машин не потеряли актуальности и сегодня.

Философские, моральные и социальные аспекты

За идеей создания искусственного интеллекта последовало множество философских вопросов. Например, возможно ли создать мыслящую машину, в то время как мы и сами до конца не понимаем, что есть процесс мышления? Как мы поймем, что машина разумна? Если она поступает разумно, значит ли это, что у нее есть сознание? Другими словами, если будет создана разумная машина, будет ли она действительно разумной или просто будет имитировать действия, которые нам кажутся разумными?

Сейчас многие верят, что эмоции играют определенную роль при формировании разума, поэтому невозможно создать мыслящую машину, которая не будет ничего чувствовать. Так как мы считаем эмоции причиной многих наших ошибок, можем ли сделать машину чувствующей, заранее зная результат? Будем ли мы вкладывать в машину все эмоции или только выборочные? В книге «Космическая Одиссея 2001 года» Артур Кларк рассматривает интересный аспект этой проблемы.

Помимо страха перед созданием искусственного разума, который превратит нас в своих слуг, существует множество других моральных вопросов, на которые

придется отвечать в том случае, если все же удастся создать разумную машину. Например, если ученые построят разумную машину, которая будет думать как человек и обладать сознанием, сможем ли мы выключить ее?

Структура данной книги

В данной книге приводится описание различных технологий и методик ИИ, а также примеры кода, которые демонстрируют их применение. Каждый метод сначала рассматривается с общих позиций, затем освещаются конкретные детали, свойства и поток выполнения алгоритмов. Во многих случаях для демонстрации алгоритмов выбираются практические и полезные приложения. Однако представлено достаточно много и теоретических примеров.

В главе 2 рассказывается о симуляции восстановления, то есть моделировании решения проблемы на основе физического процесса восстановления (сведения разрозненной субстанции к единому целому). Для иллюстрации возможностей алгоритма была выбрана задача N-ферзей.

В главе 3 описывается теория адаптивного резонанса (или ART) и ее работа с кластерами. Для демонстрации ее возможностей выбрана проблема персонализации, существующая в настоящее время.

В главе 4 рассказывается о новой технологии алгоритмов муравья и ее возможностей по поиску пути решения. Эту технологию иллюстрирует задача коммивояжера.

В главе 5 обсуждаются нейронные сети, использующие алгоритм обучения с помощью обратной связи. Возможности обучающихся сетей демонстрируются на примере создания нейроконтроллера для компьютерных игр.

В главе 6 вводится понятие генетических алгоритмов и их среда – генетическое программирование. Возможности генетического алгоритма к оптимизации демонстрируются при разработке последовательности инструкций, использующихся для решения специфических числовых и символьных проблем.

В главе 7 рассказывается о понятии искусственной жизни путем описания нейронных сетей, построенных по принципу «победитель получает все». В закрытой среде создаются простые организмы для демонстрации пищевых цепочек и развития навыков выживания в нейроконтроллере организма.

В главе 8 описывается старая методика построения ИИ на основе систем с правилами. Создается простая система с правилами и кодируется подсистема с правилами и фактами, которые используются для управления сенсорами.

В главе 9 вводится понятие нечеткой логики и описываются ее преимущества при построении систем управления. Приводится ряд примеров, описывающих простую систему управления процессом зарядки батареи с использованием нечеткой логики.

В главе 10 обсуждаются скрытые модели Маркова наряду с другими методами вероятностных графов. Технология применяется для анализа существующих литературных работ, что позволяет использовать их в качестве модели при создании нового текста.

В главе 11 вводится понятие умных агентов, а также описываются агенты различных типов и их возможности. Создается простой агент фильтрации, который автономно собирает новые данные на основании предварительно заданного критерия поиска.

Наконец, в главе 12 обсуждаются новые методики и будущее ИИ.

Литература и ресурсы

1. Stottler Henke. История ИИ. (History of AI, 2002). Доступно по адресу http://www.shai.com/ai_general/history.htm.
2. Вагман М. Процессы научных открытий для людей и компьютеров: теория и исследование по психологии и искусственному разуму (Wagman M. Scientific Discovery Processes in Humans and Computers: Theory and Research in Psychology and Artificial Intelligence. – Westport, Conn.: Praeger Publishers, 2000).
3. Кантровиц М. Часто задаваемые вопросы по ИИ (Kantrowitz M. The AI Frequently Asked Questions, 2002). При поддержке Рика Крэбба (Ric Crabbe) и Амита Дюби (Amit Dubey). Доступно по адресу http://www.faqs.org/faqs/ai-faq/general.
4. Кревье Д. ИИ: история поиска искусственного разума (Crevier D. AI: The Tumultuous History of the Search for Artificial Intelligence. – New York: Basic Books, 1993).
5. Курзвэйл Р. Век спиритических машин: когда компьютеры превосходят людей по разуму (Kurzweil R. The Age of Spiritual Machines: When Computers Exceed Human Intelligence. – New York: Viking, 1999).
6. МакКарти Дж. (McCarthy J.). Web-сайт Стэндфордского университета, http://www-formal.stanford.edu/jmc.
7. МакКарти Дж. Артур Самуэль: пионер в обучении машин (McCarthy J. Arthur Samuel, Pioneer in Machine Learning). Доступно по адресу http://www-db.stanford.edu/pub/voy/museum/samuel.html.
8. МакКарти Дж., Мински М., Рочестер Н., Шэннон С. Е. Предложение для Дортмуртского исследовательского проекта по проблеме искусственного разума (McCarty J., Minsky M., Rochester N., Shannon C. E. A Proposal for the Darmouth Research Project on Artificial Intelligence, 1955). Доступно по адресу http://www.formal.stanford.edu/jmc/history/dartmouth/dartmouth.html.
9. Мински М. (Minsky M). Web-сайт MIT, http://web.media.mit.edu/~minsky.
10. Мински М., Паперт С. Перцептроны: введение в вычислительную геометрию (Minsky M., Papert S. Perceptrons: An Introduction to Computational Geometry. – Cambridge, Mass.: MIT Press, 1969).
11. Тауб М. Компьютеры и здравый смысл: миф о мыслящих машинах (Taube M. Computers and Common Sense: The Myth of Thinking Machines, Columbia University Press, 1961).

Глава 2. Алгоритм отжига

В этой главе мы изучим метод оптимизации, который называется *отжигом,* или *симуляцией восстановления* (Simulated annealing). Как ясно из названия, метод поиска моделирует процесс восстановления. Восстановление – это физический процесс, который заключается в нагреве и последующем контролируемом охлаждении субстанции. В результате получается прочная кристаллическая структура, которая отличается от структуры с дефектами, образующейся при быстром беспорядочном охлаждении. Структура здесь представляет собой кодированное решение, а температура используется для того, чтобы указать, как и когда будут приниматься новые решения.

Естественная мотивация

Свойства структуры зависят от коэффициента охлаждения после того, как субстанция была нагрета до точки плавления. Если структура охлаждалась медленно, будут сформированы крупные кристаллы, что очень полезно для строения субстанции. Если субстанция охлаждается скачкообразно, образуется слабая структура.

Чтобы расплавить материал, требуется большое количество энергии. При понижении температуры уменьшается и количество энергии. Чтобы яснее представить процесс восстановления, рассмотрим следующий пример. «Взбалтывание» при высокой температуре сопровождается высокой молекулярной активностью в физической системе. Представьте себе, что вы взбалтываете емкость, в которой находится какая-то поверхность сложной формы. Внутри емкости также имеется шарик, который пытается найти точку равновесия. При высокой температуре шарик может свободно перемещаться по поверхности, а при низкой температуре «взбалтывание» становится менее интенсивным и передвижения шарика сокращаются. Задача заключается в том, чтобы найти точку минимального перемещения при сильном «взбалтывании». При снижении температуры уменьшается вероятность того, что шарик выйдет из точки равновесия. Именно в таком виде процесс поиска заимствуется из восстановления.

Алгоритм отжига

Давайте рассмотрим, как метафора охлаждения растаявшей субстанции используется для решения проблемы. Алгоритм отжига очень прост и может быть разделен на пять этапов (рис. 2.1).

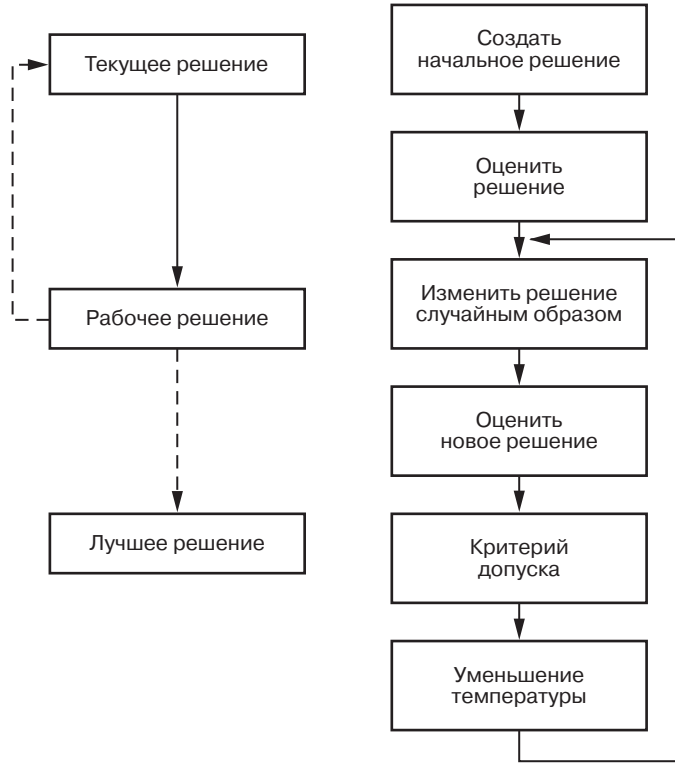


Рис. 2.1. Алгоритм отжига

Начальное решение

Для большинства проблем начальное решение будет случайным. На самом первом шаге оно помещается в *текущее решение* (Current solution). Другая возможность заключается в том, чтобы загрузить в качестве начального решения уже существующее, возможно, то самое, которое было найдено во время предыдущего поиска. Это предоставляет алгоритму базу, на основании которой выполняется поиск оптимального решения проблемы.

Оценка решения

Оценка решения состоит из декодировки текущего решения и выполнения нужного действия, позволяющего понять его целесообразность для решения данной проблемы. Обратите внимание, что закодированное решение может просто состоять из набора переменных. Они будут декодированы из существующего решения, а затем эффективность решения будет оценена на основании того, насколько успешно удалось решить данную задачу.

Случайный поиск решения

Поиск решения начинается с копирования текущего решения в *рабочее решение* (Working solution). Затем мы произвольно модифицируем рабочее решение. Как

именно модифицируется рабочее решение, зависит от того, каким образом оно представляется (кодируется). Представьте себе кодировку задачи коммивояжера, в которой каждый элемент представляет собой город. Чтобы выполнить поиск по рабочему решению, мы берем два элемента и переставляем их. Это позволяет сохранить целостность решения, так как при этом не происходит повторения или пропуска города.

После выполнения поиска рабочего решения мы оцениваем решение, как было описано ранее. Поиск нового решения основан на методе Монте-Карло (то есть случайным образом).

Критерий допуска

На этом этапе алгоритма у нас имеется два решения. Первое – это наше оригинальное решение, которое называется текущим решением, а второе – найденное решение, которое именуется рабочим решением. С каждым решением связана определенная энергия, представляющая собой его эффективность (допустим, что чем ниже энергия, тем более эффективно решение).

Затем рабочее решение сравнивается с текущим решением. Если рабочее решение имеет меньшую энергию, чем текущее решение (то есть является более предпочтительным), то мы копируем рабочее решение в текущее решение и переходим к этапу снижения температуры.

Однако если рабочее решение хуже, чем текущее решение, мы определяем критерий допуска, чтобы выяснить, что следует сделать с текущим рабочим решением. Вероятность допуска основывается на уравнении 2.1 (которое, в свою очередь, базируется на законе термодинамики):

$$P(\delta E) = \exp(-\delta E/T) \qquad (2.1)$$

Значение этой формулы визуально показано на рис. 2.2. При высокой температуре (свыше 60 °C) плохие решения принимаются чаще, чем отбрасываются. Если энергия меньше, вероятность принятия решения выше. При снижении температуры

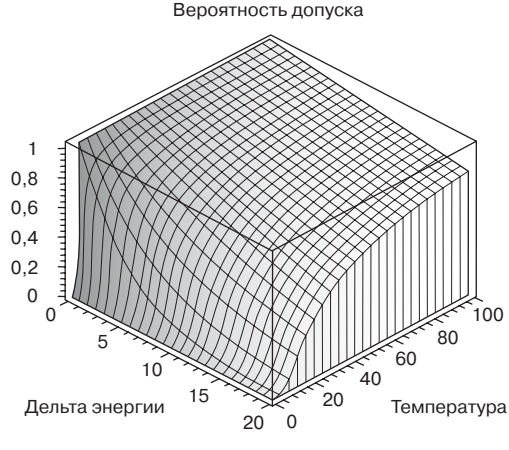


Рис. 2.2. Визуализация вероятности принятия решения

вероятность принятия худшего решения также снижается. При этом более высокий уровень энергии также способствует уменьшению вероятности принятия худшего решения.

При высоких температурах симулированное восстановление позволяет принимать худшие решения для того, чтобы произвести более полный поиск решений. При снижении температуры диапазон поиска также уменьшается, пока не достигается равенство при температуре 0°.

Снижение температуры

После ряда итераций по алгоритму при данной температуре мы ненамного снижаем ее. Существует множество вариантов снижения температуры. В данном примере используется простая геометрическая функция (см. уравнение 2.2):

$$T_{i+1} = \alpha T_i \qquad (2.2)$$

Константа α меньше единицы. Возможны и другие стратегии снижения температуры, включая линейные и нелинейные функции.

Повтор

При одной температуре выполняется несколько итераций. После завершения итераций температура будет понижена. Процесс продолжится, пока температура не достигнет нуля.

Пример итерации

Чтобы проиллюстрировать алгоритм, проследим несколько итераций. Обратите внимание, что если рабочее решение имеет меньший уровень энергии (то есть является лучшим решением) по сравнению с текущим решением, то всегда используется именно оно. Критерий допуска вступает в силу только при условии, что рабочее решение хуже, чем текущее.

Предположим, что температура окружающей среды равна 50°, а энергия текущего решения составляет 10. Мы копируем текущее решение в рабочее решение и выполняем поиск. После оценки энергии устанавливаем, что энергия нового рабочего решения равна 20. В этом случае энергия рабочего решения выше, чем энергия начального решения. Поэтому мы используем критерий допуска:

Энергия текущего решения равна 10.
Энергия рабочего решения равна 20.

Дельта энергии для этого примера (энергия рабочего решения минус энергия текущего решения) равна 10. Подставив это значение и температуру 50 в уравнение 2.1, получаем вероятность:

$$P = \exp(-10/50) = 0{,}818731.$$

Таким образом, на этом примере мы видим, что вероятность принятия худшего решения достаточно велика. Теперь рассмотрим пример с более низкой температурой. Предположим, что температура равна 2, а энергия имеет следующие показатели:

Энергия текущего решения равна 3.
Энергия рабочего решения равна 7.

Дельта энергии в этом примере равна 4. Подставив это значение и температуру в уравнение 2.1, получаем вероятность:

$P = \exp(-4/2) = 0{,}135335$.

Данный пример показывает, что вероятность выбора рабочего решения для последующих итераций очень невелика. Это базовая форма алгоритма. Далее мы применим этот метод к реальной задаче и посмотрим, как работает данный алгоритм.

Пример задачи

Для демонстрации этого алгоритма используется широко известная задача, решить которую пытались с помощью множества алгоритмов поиска. Задача N шахматных ферзей (или NQP) – это задача размещения N ферзей на шахматной доске размером N×N таким образом, чтобы ни один ферзь не угрожал другому (рис. 2.3).

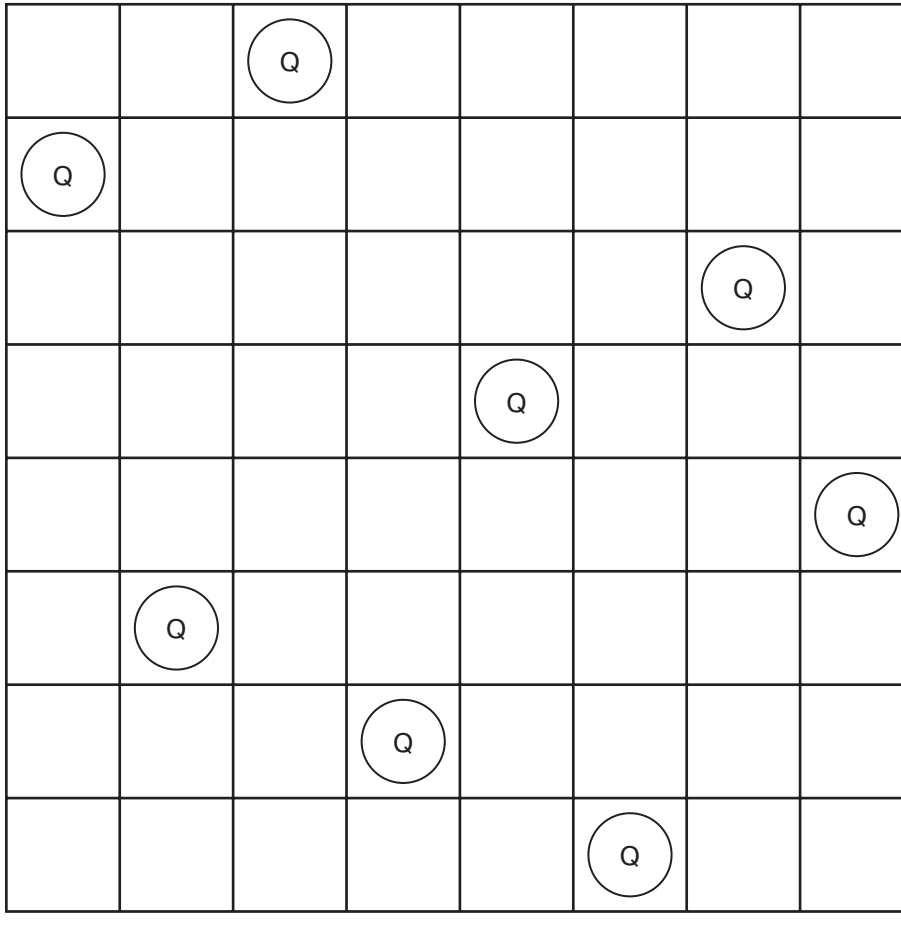


Рис. 2.3. Одно из 92 решений задачи 8 ферзей

Задача 8 ферзей впервые была решена в 1850 г. Карлом Фридрихом Гаубом (Carl Friedrich Gaub). Алгоритм поиска (как видно из даты решения) представлял собой метод проб и ошибок. Затем задача ферзей решалась с помощью метода поиска в глубину (1987), метода «разделяй и властвуй» (1989), генетических алгоритмов (1992) и многими другими способами. В 1990 г. Рок Сосик (Rok Sosic) и Цзюн Гу (Jun Gu) решили проблему для 3000000 ферзей с использованием метода локального поиска и минимизации конфликтов.

Представление решения

Представление решения (кодировка) задачи о N ферзях стандартна: для сужения области поиска используется конечное решение. Обратите внимание (рис. 2.3), что в каждой строке и каждом столбце может располагаться только один ферзь. Это ограничение намного упрощает создание кодировки, которая управляется алгоритмом отжига.

Так как каждый столбец содержит только одного ферзя, для отображения решения будет использоваться массив из N элементов (рис. 2.4). В элементах этого массива хранятся строчные индексы положения ферзя. Например, на рис. 2.4 столбец 1 содержит значение 5, соответствующее строке, в которую будет помещен ферзь.

Создать произвольное решение очень просто. Сначала нужно инициализировать решение, позволив каждому ферзю занять строку, соответствующую столбцу. Затем необходимо пройти по каждому столбцу и выбрать произвольное число от 1 до N для каждого столбца. Затем два элемента перемещаются (текущий столбец и произвольно выбранный столбец). Когда алгоритм достигает конца, автоматически формируется решение.

Наконец, получив кодировку, мы видим, что на доске нет конфликтов по горизонтали или диагонали. При оценке решения следует учитывать только конфликты по диагонали.

Энергия

Энергия решения определяется как количество конфликтов, которые возникают в кодировке. Задача заключается в том, чтобы найти кодировку, при которой энергия равна нулю (то есть на доске нет конфликтов).

Температура

Для данной проблемы мы начнем поиск решения с температуры 30° и постепенно будем снижать ее до нуля, используя геометрическую формулу (уравнение 2.2). При этом значение α будет равно 0,98. Как будет видно далее, график температуры показывает сначала быстрое снижение, а потом медленное схождение к конечной температуре – нулю.

При каждом изменении температуры мы выполним 100 итераций. Это позволит алгоритму осуществить несколько операций поиска на каждом уровне.

Совет *Исходный код алгоритма отжига, который используется для решения задачи N ферзей, можно загрузить с сайта издательства «ДМК Пресс» www.dmk.ru. В папке /software/ch2/emsapor также содержится экспериментальный алгоритм отжига, который использовался для решения задачи N ферзей (при N не больше 80).*

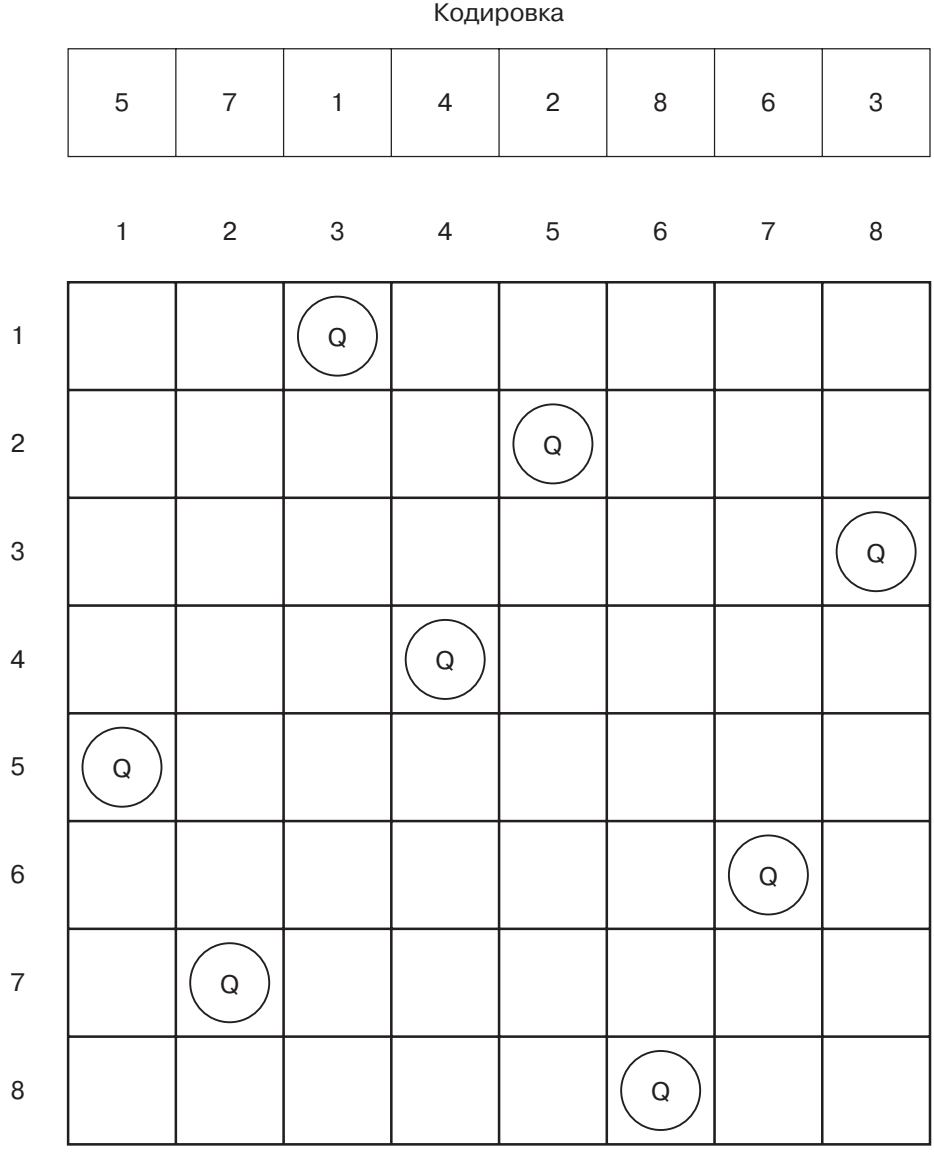


Рис. 2.4. Кодировка решения задачи N ферзей

Исходный код

Рассмотрим исходный код, который реализует алгоритм отжига для решения задачи N ферзей.

Сначала мы взглянем на константы и типы данных, которые использует алгоритм (см. листинг 2.1).

Листинг 2.1. Типы данных и константы

```
#define MAX_LENGTH 30

typedef int solutionType[MAX_LENGTH];

typedef struct {
  solutionType solution;
  float energy;
} memberType;

/* Параметры алгоритма */
#define INITIAL_TEMPERATURE 30.0
#define FINAL_TEMPERATURE   0.5
#define ALPHA               0.98
#define STEPS_PER_CHANGE    100
```

Массив `solutionType` – это наша кодировка задачи о ферзях. Символьная константа `MAX_LENGTH` определяет размер доски (в данном случае решается задача для 30 ферзей). Допускается изменять значение константы `MAX_LENGTH` (до 50 или более), однако при использовании большего значения могут понадобиться изменения в графике охлаждения.

Решение хранится в структуре `memberType`, которая также включает энергию, рассчитываемую для решения.

Остальная часть листинга 2.1 определяет график охлаждения. Константы `INITIAL_TEMPERATURE` и `FINAL_TEMPERATURE` задают границы графика, а константа `ALPHA` используется для определения геометрического охлаждения. Константа `STEPS_PER_CHANGE` устанавливает количество итераций, которые будут выполнены после каждого изменения температуры.

Далее мы рассмотрим вспомогательные функции алгоритма. В листинге 2.2 содержатся функции инициализации кодировки и поиска нового решения. Они используются для создания начального решения и его произвольного изменения.

Листинг 2.2. Инициализация и функции поиска

```
void tweakSolution( memberType *member )
{
  int temp, x, y;

  x = getRand(MAX_LENGTH);
  do {
    y = getRand(MAX_LENGTH);
  } while (x == y);
```

```
    temp = member->solution[x];
    member->solution[x] = member->solution[y];
    member->solution[y] = temp;
}

void initializeSolution( memberType *member )
{
  int i;
  /* Начальная инициализация решения */
  for (i = 0 ; i < MAX_LENGTH ; i++) {
    member->solution[i] = i;
  }

 /* Изменение решения случайным образом */
  for (i = 0 ; i < MAX_LENGTH ; i++) {
    tweakSolution( member );
  }


}
```

Функция `initializeSolution` создает решение, при котором все ферзи помещаются на доску. Для каждого ферзя задаются идентичные индексы строки и столбца. Это обозначает отсутствие конфликтов по горизонтали и вертикали. Затем решение изменяется при помощи функции `tweakSolution`. Позднее функция `tweakSolution` используется в алгоритме, чтобы изменить рабочее решение, выведенное из текущего решения.

Оценка решения выполняется с помощью функции `computeEnergy`. Функция идентифицирует все конфликты, которые существуют для текущего решения (листинг 2.3).

Листинг 2.3. Оценка решения

```
void computeEnergy( memberType *member )
{
  int  i, j, x, y, tempx, tempy;
  char board[MAX_LENGTH][MAX_LENGTH];
  int conflicts;
  const int dx[4] = {-1,  1, -1,  1};
  const int dy[4] = {-1,  1,  1, -1};

   /* Стандартная функция очистки памяти */
   bzero( (void *)board, MAX_LENGTH * MAX_LENGTH );

   for (i = 0 ; i < MAX_LENGTH ; i++) {
     board[i][member->solution[i]] = 'Q';
   }

   /* Считает количество конфликтов для каждого ферзя */
```

```
conflicts = 0;

 for (i = 0 ; i < MAX_LENGTH ; i++) {
    x = i; y = member->solution[i];
    /* Замечания: по условию кодировки конфликты по вертикали
     * и горизонтали исключены
     */
    /* Проверяем диагонали */
    for (j = 0 ; j < 4 ; j++) {

      tempx = x ; tempy = y;
      while(1) {
        tempx += dx[j]; tempy += dy[j];
        if ((tempx < 0) || (tempx >= MAX_LENGTH) ||
               (tempy < 0) || (tempy >= MAX_LENGTH)) break;
        if (board[tempx][tempy] == 'Q') conflicts++;
      }
    }
  }
  member->energy = (float)conflicts;
}
```

Чтобы продемонстрировать результат, построим шахматную доску. Это не является обязательным, но упрощает зрительное восприятие решения проблемы. Обратите внимание на диапазоны `dx` и `dy`. Эти диапазоны используются для расчета следующего положения на доске для каждого найденного ферзя. В первом случае `dx = -1`, а `dy = -1`, что соответствует перемещению на северо-запад. Конечный результат, `dx = 1`, `dy = -1`, отвечает перемещению на северо-восток.

Мы выбираем по очереди каждого ферзя на доске (ферзь определяется значениями `x` и `y`), а затем перемещаемся по все четырем диагоналям в поиске конфликтов, то есть другого ферзя. Если ферзь найден, значение переменной конфликта увеличивается. После завершения поиска конфликты загружаются в структуру с решением в качестве значения энергии.

Следующая функция используется для копирования одного решения в другое. Вспомните (листинг 2.1), что решение кодируется и сохраняется в структуре `memberType`. Функция `copySolution` копирует содержимое одной структуры `memberType` в другую (листинг 2.4).

Листинг 2.4. Копирование одного решения в другое

```
void copySolution( memberType *dest, memberType *src )
{
  int i;

  for (i = 0 ; i < MAX_LENGTH ; i++) {
    dest->solution[i] = src->solution[i];
  }
  dest->energy = src->energy;
}
```

Последняя вспомогательная функция, которую мы рассмотрим, – это функция `emitSolution`. Она просто распечатывает представление доски из закодированного решения и выдает его. Печатаемое решение передается как аргумент функции (листинг 2.5).

Листинг 2.5. Отображение решения в виде шахматной доски

```
void emitSolution( memberType *member )
{
  char board[MAX_LENGTH][MAX_LENGTH];
  int x, y;

  bzero( (void *)board, MAX_LENGTH * MAX_LENGTH );

  for (x = 0 ; x < MAX_LENGTH ; x++) {
    board[x][member->solution[x]] = 'Q';
  }

  printf("board:\n");
  for (y = 0 ; y < MAX_LENGTH ; y++) {
    for (x = 0 ; x < MAX_LENGTH ; x++) {
      if (board[x][y] == 'Q') printf("Q");
      else printf(". ");
    }
    printf("\n");
  }
  printf("\n\n");
}
```

При помощи функции `emitSolution` шахматная доска печатается на основе кодировки (строка – это индекс, а содержимое – это столбец). Затем доска отображается («Q» соответствует ферзю, а «.» – пустой клетке доски).

После изучения всех вспомогательных функций можно приступать к рассмотрению самого алгоритма отжига, представленного с помощью функции `main()` (листинг 2.6).

Листинг 2.6. Алгоритм отжига

```
int main()
{
  int  timer=0, step, solution=0, useNew, accepted;
  float temperature = INITIAL_TEMPERATURE;
  memberType current, working, best;
  FILE *fp;

  fp = fopen("stats.txt", "w");

  srand(time(NULL));

  initializeSolution( &current );
  computeEnergy( &current );
  best.energy = 100.0;
```

```
copySolution( &working, &current);

while (temperature > FINAL_TEMPERATURE) {

  printf("Temperature : %f\n", temperature);

  accepted = 0;

  /* Изменены решения случайным образом */
  for (step = 0 ; step < STEPS_PER_CHANGE ; step++) {

    useNew = 0;

    tweakSolution( &working );
    computeEnergy( &working );

    if (working.energy <= current.energy) {

      useNew = 1;

    } else {

      float test = getSRand();
      float delta = working.energy - current.energy;
      float calc = exp(-delta/temperature);

      if (calc > test) {
        accepted++;
        useNew = 1;
      }

    }

    if (useNew) {
      useNew = 0;
      copySolution( &current, &working );

      if (current.energy < best.energy) {
        copySolution( &best, &current );
        solution = 1;
      }

    } else {

      copySolution( &working, &current);

    }

  }

  fprintf(fp, "%d %f %f %d\n",
          timer++, temperature, best.energy, accepted);
```

```
      printf("Best energy = %f\n", best.energy);

      temperature *= ALPHA;
    }

    fclose(fp);
    if (solution) {
      emitSolution( &best );
    }

    return 0;
  }
```

Алгоритм отжига, представленный листингом 2.6, практически повторяет обобщенный алгоритм, показанный на рис. 2.1. После двух вводных действий (открытия файла для вывода лога и инициализации генератора случайных чисел) мы инициализируем текущее решение в переменной `current` и оцениваем энергию решения при помощи функции `computeEnergy`. Текущее решение копируется в рабочее, и запускается алгоритм.

Внешний цикл алгоритма выполняется до тех пор, пока текущая температура не станет меньше конечной температуры или не сравняется с ней. Это позволяет избежать использования нулевой температуры в функции расчета вероятности.

Внутренний цикл алгоритма работает по методу Монте-Карло. Он выполняет ряд итераций при текущей температуре с целью полного изучения возможностей поиска при данной температуре.

Первый шаг – изменение рабочего решения с помощью функции `tweakSolution`. Затем рассчитывается энергия рабочего решения, которая сравнивается с текущим решением. Если энергия нового рабочего решения меньше или равна энергии текущего решения, рабочее решение принимается по умолчанию. В противном случае выполняется уравнение 2.1 (оценка вероятности допуска). Таким образом определяется, будет ли выбрано худшее решение. Дельта энергии рассчитывается как разница между рабочей энергией и текущей. Это означает, что энергия рабочего решения больше, чем энергия текущего решения. В нашем случае просто генерируется случайное число в интервале от 0 до 1, которое затем сравнивается с результатом уравнения 2.1. Если условие допуска выполнено (результат уравнения 2.1 больше случайного значения), то рабочее решение принимается. Затем рабочее решение необходимо скопировать в текущее, так как переменная `working`, в которой на данный момент хранится рабочее решение, будет повторно изменена при следующей итерации внутреннего цикла.

Если рабочее решение не было принято, текущее решение копируется поверх рабочего. При следующей итерации старое рабочее решение удаляется, программа изменяет текущее решение и пробует снова.

После вывода статистической информации в лог-файл температуру необходимо снизить (выполнив требуемое количество итераций внутреннего цикла). Вспомните (уравнение 2.2), что график температуры представляет собой простую геометрическую функцию. Следует умножить текущую температуру на константу `ALPHA` и повторить внешний цикл.

В конце алгоритма выводится решение, хранящееся в переменной `best`, которое было найдено (если оно вообще было найдено, об этом сигнализирует переменная `solution`). Эта переменная устанавливается во внутреннем цикле после того, как было определено, что обнаружено решение, энергия которого меньше энергии текущего решения `best`.

Пример выполнения

Рассмотрим результат запуска алгоритма. В этом примере мы начнем с температуры 100, хотя для решения проблемы достаточно температуры 30. Такая высокая температура задана для иллюстрации работы алгоритма (рис. 2.5).

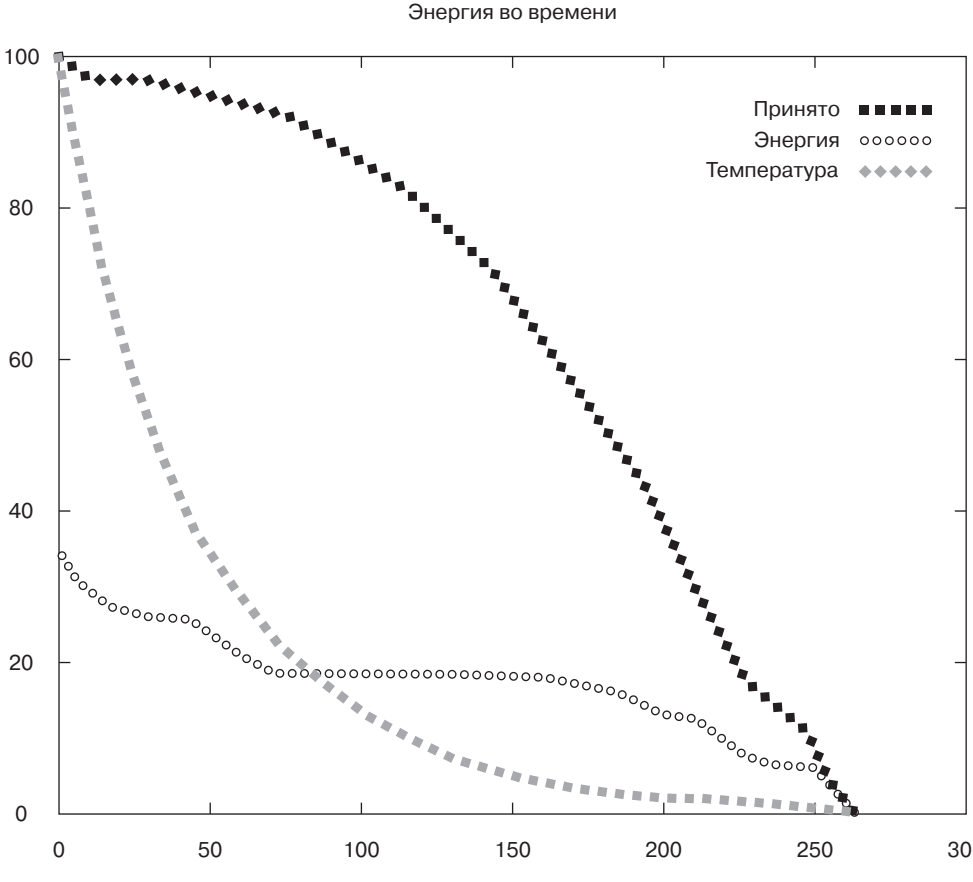


Рис. 2.5. Пример выполнения алгоритма отжига для задачи с 40 ферзями

Элементом, значение которого резко уменьшается от 100 до 0, является температура. График охлаждения использует уравнение 2.2. Элемент, значение

которого уменьшается не так резко, – это количество принимаемых худших решений (основанное на вероятностном уравнении допуска 2.1). Так как вероятность допуска представляет собой функцию температуры, легко заметить их взаимосвязь. Наконец, третий график иллюстрирует энергию лучшего решения. Как показывает график, идеальное решение находится только в самом конце поиска. Это справедливо для всех запусков алгоритма (причина – уравнение критерия допуска), что демонстрируется резким спадом кривой «допустимых» худших решений в конце графика.

Пример решения задачи о 40 ферзях (представленной на рис. 2.5) показан на рис. 2.6.

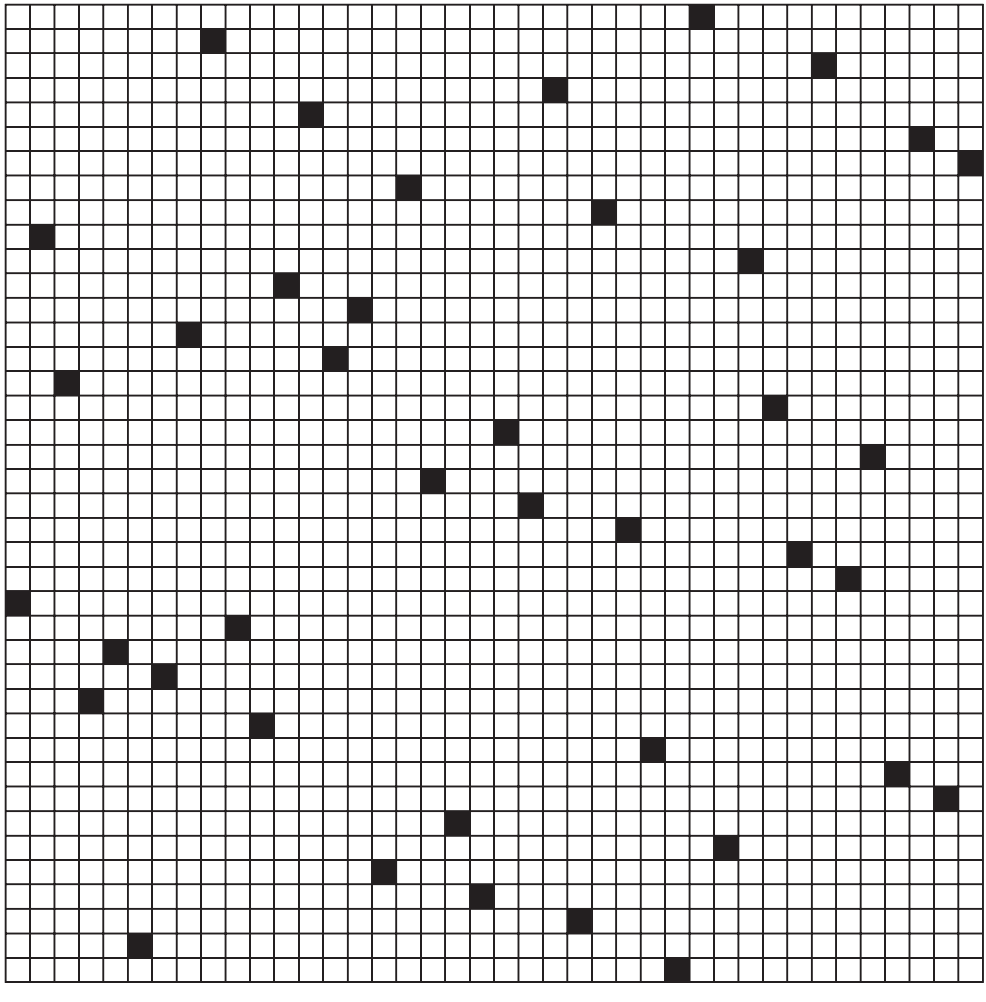


Рис. 2.6. Пример решения задачи 40 ферзях

Оптимизация алгоритма

Вы можете изменять параметры алгоритма в зависимости от сложности проблемы, которую нужно решить. В этом разделе описаны параметры, которые допускается переопределять, а также результаты возможных изменений.

Начальная температура

Начальная температура должна быть достаточно высокой, чтобы сделать возможным выбор из других областей диапазона решений. По утверждению Грэхема Кендалла (Graham Kendall), если известно максимальное расстояние между соседними решениями, то легко рассчитать начальную температуру.

Начальную температуру также можно изменять динамически. Если задать статистику по коэффициенту допуска худших решений и нахождению новых лучших решений, можно повышать температуру до тех пор, пока не будет достигнуто нужное количество допусков (открытий новых решений). Этот процесс аналогичен нагреву субстанции до перехода ее в жидкую форму, после чего уже нет смысла повышать температуру.

Конечная температура

Хотя ноль является удобной конечной температурой, геометрическая функция, которая используется в примере, показывает, что алгоритм будет работать намного дольше, чем это действительно необходимо. Поэтому конечная температура в листинге 2.1 задана как 0,5°. Это значение может изменяться в зависимости от того, какая функция изменения температуры используется.

Функция изменения температуры

Используемую функцию изменения температуры можно модифицировать в зависимости от решаемой задачи. На рис. 2.5 показано изменение температуры во времени при применении геометрической функции. Снижение температуры допускается определять и с помощью многих других функций. Результатом использования этих функций может быть постепенное снижение температуры в первой половине графика или медленное снижение, за которыми следует резкий спад. Очень интересный пример графиков температуры вы найдете на Web-сайте Брайана Люка (Brian Luke) в статье «Графики охлаждения при симулированном восстановлении».

Количество итераций при одном значении температуры

При высоких температурах алгоритм отжига выполняет поиск оптимального решения во всем диапазоне решений. При снижении температуры движение уменьшается, и алгоритм ищет локальный оптимум, чтобы улучшить решение. Поэтому количество итераций, заданное для каждой температуры, имеет большое значение. При решении данной задачи (листинг 2.1) указано 100 итераций. Чтобы правильно определить количество итераций, которое оптимально подходит для решения проблемы, необходимо поэкспериментировать.

Другие области применения

Метод отжига может быть эффективным при решении задач различных классов, требующих оптимизации. Ниже приводится их краткий список:

- создание пути;
- реконструкция изображения;
- назначение задач и планирование;
- размещение сети;
- глобальная маршрутизация;
- обнаружение и распознавание визуальных объектов;
- разработка специальных цифровых фильтров.

Поскольку метод отжига представляет собой процесс генерации случайных чисел, поиск решения с использованием данного алгоритма может занять значительное время. В некоторых случаях алгоритм вообще не находит решение или выбирает не самое оптимальное.

Итоги

В этой главе рассматривался алгоритм отжига как способ выполнения процедур поиска и оптимизации. Данный метод является аналогом процесса нагревания тела до состояния плавления с последующим постепенным охлаждением. При высоких температурах поиск ведется по всему диапазону. При снижении температуры диапазон поиска уменьшается до небольшой области вокруг текущего решения. Алгоритм проиллюстрирован с помощью классической задачи размещения N ферзей на шахматной доске. Наконец, вы изучили параметры алгоритма, а также узнали о возможностях их изменения для решения более сложных или аналогичных задач, но с более высокой скоростью поиска.

Литература и ресурсы

1. Галлант С. Обучение в нейронных сетях (Gallant S. Neural Network Learning. – Cambridge, Mass.: MIT Press, 1994).
2. Даусленд К. Симулированное восстановление (Dowsland K. Simulated Annealing // Modern Heuristic Techniques for Combinatorial Problems, Colin Reeves (ed.). – New York: McGraw-Hill, 1995).
3. Карпентер Д., Гроссберг С. Массивная параллельная архитектура для самостоятельной машины, распознающей нейронные модели (Carpenter G., Grossberg S. A Massively Parallel Architecture for a Self-Organizing Neural Pattern Recognition Machine // Computer Vision, Graphics and Image Processing, 1987).
4. Кендалл Г. Симулированное восстановление (Kendall G. Simulated Annealing, 2002). Доступно по адресу http://www.cs.nott.ac.uk/~gxk/aim/notes/simulatedannealing.doc.

5. Люк Б. Графики охлаждения при симулированном восстановлении (Luke B. T. Simulated Annealing Cooling Schedule, 2001). Доступно по адресу http://members.aol.com/btluke/simanf1.htm.
6. Метрополис Н. Уравнение расчетов состояния для быстрых компьютеров (Metropolis N. Equation of State Calculation by Fast Computing Machines // Journal of Chem. Phys. 21:1087–1091).
7. Шаллер Н. Проблема N ферзей (NQP) (Schaller N. The N-Queens Problem (NQP), 2001). Доступно по адресу http://www.dsitri.de/projects/NQP/.

Глава 3. Введение в теорию адаптивного резонанса

Данная глава посвящена знаменитому алгоритму Гроссберга и Карпентера, ART1, который был первым в семье алгоритмов *теории адаптивного резонанса* (Adaptive Resonance Theory). Это очень простой алгоритм с обучением, основанный на биологической мотивации. После подробного рассмотрения алгоритма ART1 его работа будет продемонстрирована на примере персонализации (данная проблема также известна как система выдачи рекомендаций).

Алгоритмы кластеризации

Алгоритм кластеризации (Clustering algorithm) – это метод, благодаря которому данные разделяются и объединяются в небольшие группы (кластеры) по принципу аналогии. По тому же принципу осуществляется отделение несхожих данных, поэтому главной задачей при разбивке данных на кластеры является классификация. Хотя классификация используется во многих случаях, ее основное предназначение – изучение данных в кластерах для выявления различий между ними. Более специфическое использование кластеров будет рассмотрено далее.

Биологическая мотивация

Алгоритмы кластеризации имеют биологическое происхождение, поскольку предоставляют возможность обучения посредством классификации. Человеческий мозг изучает новые понятия, сравнивая их с уже существующими знаниями. Мы классифицируем новое, пытаясь объединить его в одном кластере с чем-то, что нам уже известно (это является основой для понимания нового). Если новое понятие нельзя связать с тем, что мы уже знаем, нам приходится создавать новую структуру, чтобы понять явление, которое выходит за рамки существующей структуры. Впоследствии эта новая модель может стать основой для усвоения другой информации.

Объединяя новые понятия в кластеры с уже существующими знаниями, а также создавая новые кластеры для усвоения абсолютно новой информации мы решаем проблему, которую Гроссберг назвал «дилеммой стабильности-гибкости». Вопрос состоит в том, как классифицировать новые данные и при этом не уничтожать уже изученные. Алгоритм ART1 включает все необходимые элементы,

позволяющие не только создавать новые кластеры при обнаружении новой информации, но и реорганизовывать с ее учетом уже существующие кластеры.

Алгоритм ART1

Алгоритм ART1 работает с объектами, которые называются *векторами признаков* (Feature vector). Вектор признаков является группой значений в двоичном коде, которые представляют определенный тип информации. Примером вектора признаков может служить выбор покупок (рис. 3.1). Каждый объект вектора признаков показывает, приобрел ли покупатель товар (если да, то значение равно 1, если нет – 0). Покупатель на рис. 3.1 купил молоток и гаечный ключ.

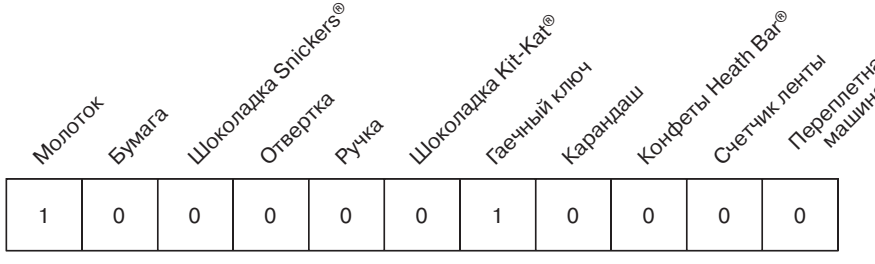


Рис. 3.1. Пример вектора признаков, включающего информацию о покупках

Этот вектор признаков описывает покупательную способность путем идентификации приобретенных покупателем предметов (о которых мы имеем информацию). Собираются векторы признаков покупателя, к которым затем применяется алгоритм ART1, чтобы разделить данные на кластеры. Идея состоит в том, что группа схожих данных о покупателе (содержащаяся в кластере) будет сообщать интересную информацию о схожих параметрах для группы покупателей.

ART1 в деталях

Мы начнем с группы векторов признаков (назовем эти примеры $E_{1...K}$) и группы инициализированных *векторов-прототипов* (Prototype vector, $P_{1...N}$). Вектор-прототип является центром кластера. Количество векторов-прототипов, равное N, является максимальным количеством кластеров, которое может поддерживаться. Параметр d показывает длину вектора. Мы инициализируем параметр *внимательности* (ρ или rho), равный небольшому значению между 0 и 1,0, а также бета-параметр (B), равный небольшому положительному целому числу. Эти параметры будут рассмотрены более подробно. Список рабочих параметров представлен в табл. 3.1.

Таблица 3.1. Параметры алгоритма ART1

Параметр	Описание
$\nu \cap \omega$	Побитовый И-вектор
$\|\nu\|$	Значимость ν (количество значимых элементов вектора)
N	Количество векторов-прототипов

Таблица 3.1. Параметры алгоритма ART1 (окончание)

Параметр	Описание
ρ	Параметр внимательности (0 < ρ <= 1)
P	Вектор-прототип
E	Вектор признаков
d	Размер векторов (длина)
β	Бета-параметр

Некоторые действия, показанные в табл. 3.1, могут показаться вам непонятными. Например, побитовый И-вектор представляет собой просто результат побитового И для двух векторов. В итоге получается новый вектор. При этом если в каждом из родительских векторов в одном и том же разряде бит установлен в 1, то в результирующем векторе в этом разряде также ставится 1. В противном случае, когда хотя бы в одном родительском векторе в разряде бит определен как 0, будет установлен 0. Значимость вектора – это количество разрядов в векторе, которые не равны нулю.

Блок-схема алгоритма ART1 показана на рис. 3.2. В алгоритме используются уравнения 3.1–3.4.

Изначально не существует ни одного вектора-прототипа, поэтому при выполнении алгоритма создается первый вектор-прототип из первого вектора-признаков (уравнение 3.1). Затем проверяются на схожесть все последующие векторы признаков с вектором-прототипом. Цель проверки – выяснить, насколько схож вектор признаков и текущий вектор-прототип.

$$P_o = E_o \tag{3.1}$$

Бета-параметр (β), который используется в уравнении проверки на схожесть (уравнение 3.2), – это параметр «разрушения связи». Он выбирает прототипы, в которых больше значений 1, при условии, что все значения 1 в векторе-прототипе также присутствуют в тестируемом векторе признаков.

$$\|P_i \cap E\| / (\beta + \|P_i\|) > \|E\| / (\beta + d) \tag{3.2}$$

Если тест на схожесть прошел успешно, выполняется следующий тест, чтобы проверить вектор признаков и вектор-прототип против параметра внимательности (уравнение 3.3). Задачей данного параметра является определение размера класса. Если значение параметра велико, образуются более крупные классы (кластеры с большим количеством данных). При уменьшении значения создаются кластеры с меньшим количеством данных. Если параметр внимательности задан достаточно низким (< 0,1), для допуска векторы признаков должны соответствовать вектору-прототипу.

$$\|P_i \cap E\| / \|E\| < \rho \tag{3.3}$$

Наконец, если пройден тест на внимательность, алгоритм добавляет текущий вектор признаков в текущий вектор-прототип (уравнение 3.4). Этот процесс представляет собой простое слияние вектора признаков и вектора-прототипа с помощью операции И. Если тест на внимательность (или тест на схожесть) не

был пройден, проверяется следующий вектор-прототип. Если все векторы-прототипы были проверены и при этом вектор признаков не был помещен в кластер, создается новый вектор-прототип из вектора признаков. Это приводит к формированию нового кластера, так как рассматриваемый вектор признаков не соответствует ни одному существующему кластеру.

$$P_i = P_i \cap E \tag{3.4}$$

Теперь алгоритм проходит через все векторы признаков и сравнивает их со всеми векторами-прототипами (в соответствии с диаграммой на рис. 3.2). Хотя

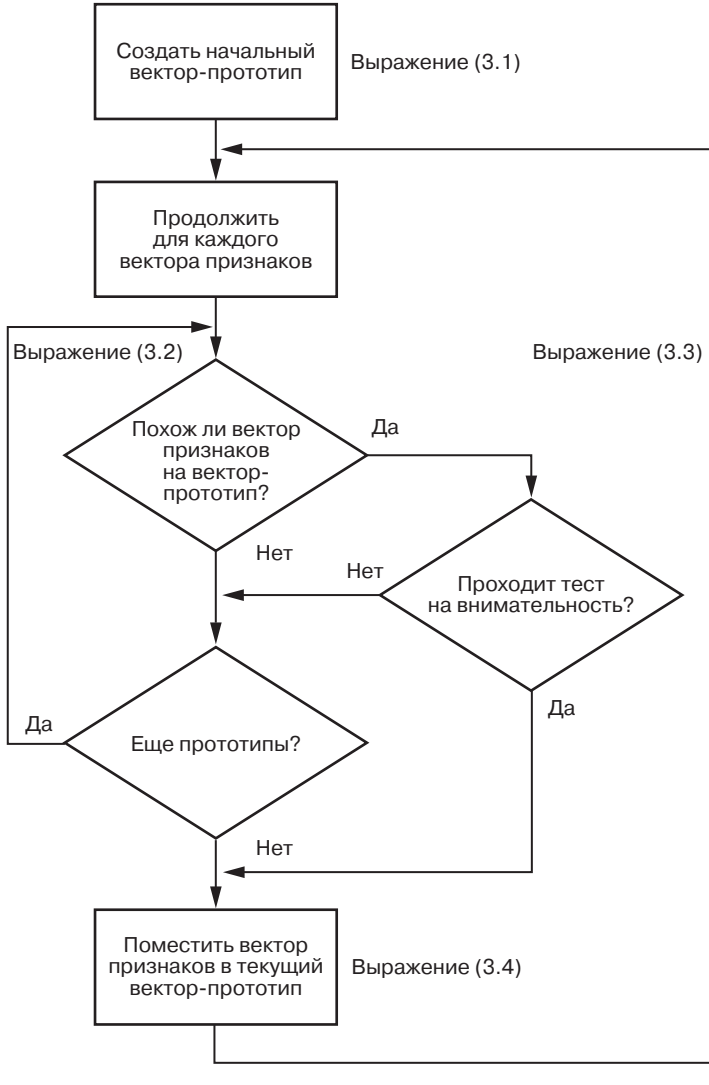


Рис. 3.2. Выполнение алгоритма ART1

все векторы уже размещены по кластерам, проверка необходима. Она позволяет убедиться в том, что векторы расположены в нужных кластерах. Дело в том, что последующие тесты векторов признаков могли создать новые кластеры, поэтому необходимо выполнить дополнительную проверку и удостовериться, что векторы не нужно перемещать в другие кластеры.

После проверки всех векторов признаков, которая не потребовала дополнительных изменений, процесс формирования кластеров можно считать завершенным. Чтобы избежать перемещения вектора признаков между двумя векторами-прототипами, алгоритм выполняет несколько итераций, чтобы объединить кластеры. Количество итераций должно быть достаточно большим, чтобы избежать преждевременного слияния.

Разбор выполнения алгоритма

Рассмотрим работу алгоритма на простом примере. Предположим, что мы имеем два кластера, представленных векторами-прототипами P_0 и P_1:

$P_0 = \{1,0,0,1,1,0,1\}$
$P_1 = \{1,1,0,0,0,1,0\}$

Вектор признаков имеет следующий вид:

$E_0 = \{1,1,1,0,0,1,0\}$

Кроме того, параметры алгоритма имеют следующие значения:

$\delta = 7$
$\beta = 1{,}0$
$\rho = 0{,}9$

Совет *Параметры β и ρ были выбраны после ряда экспериментов. При решении любой другой задачи рекомендуется использовать несколько комбинаций, чтобы найти параметры, при которых можно добиться наилучшего результата.*

Теперь выполним тесты алгоритма ART1, чтобы определить, в какой кластер будет помещен вектор признаков:

Тест на сходство	P_0/E_x	(3.2)	$1/5 > 4/8$	(Нет)
Тест на сходство	P_1/E_x	(3.2)	$3/5 > 4/8$	(Да)
Тест на внимательность	P_1/E_x	(3.3)	$3/4 < 0{,}9$	(Нет)
P_1 И E_x		(3.4)	$\{1,1,0,0,0,1,0\}$ И $\{1,1,1,0,0,1,0\} = \{1,1,0,0,0,1,0\}$	

В первом тесте выполнена проверка на схожесть для вектора-прототипа P_0. Используя уравнение 3.2, мы определили, что тест прошел неудачно (0,2 не больше, чем 0,5). Затем вектор признаков был проверен против вектора P_1. В этом случае тест прошел успешно, поэтому алгоритм выполняет тест на внимательность.

Данный тест также прошел успешно, поэтому вектор признаков ассоциируется с кластером, который представлен вектором P_1. Затем вектор-прототип изменяется путем выполнения операции побитового И между вектором-прототипом P_1 и вектором признаков (уравнение 3.4). После обновления вектора-прототипа все векторы признаков проверяются против всех доступных векторов-прототипов: следует убедиться, что они находятся в нужных кластерах. После завершения изменений процесс заканчивается.

Обучение в ART1

В то время как векторы признаков сверяются с векторами-прототипами, создаются новые кластеры или модифицируются уже существующие. Это действие известно как «резонанс» и отображает процесс обучения в алгоритме. Когда алгоритм достигает равновесия (то есть векторы-прототипы больше не подвергаются изменениям), обучение завершается, и в результате мы получаем классифицированные исходные данные.

Преимущества ART1 по сравнению с другими алгоритмами кластеризации

Алгоритм ART1 концептуально прост и легок в реализации. Более ранние алгоритмы, такие как алгоритм кластеризации McQueen, хотя и были проще, но имели ряд существенных недостатков. Например, они не позволяли создавать новые кластеры (кластеры задавались в начале работы алгоритма). Кроме того, в ранних алгоритмах не было параметра, позволяющего изменять размеры класса для кластеров. Недостаток всех алгоритмов (ранних и ART1) заключается в том, что конечный набор кластеров (и векторов-прототипов) может изменяться в зависимости от порядка, в котором проводилось обучение.

Семейство алгоритмов ART

Было создано множество версий алгоритма ART1 как с целью усовершенствования, так и для решения различных проблем. Алгоритм ART1 работает с дискретными данными, а алгоритм ART2 позволяет классифицировать непрерывный поток данных (например, временные диаграммы). ARTMAP – это измененный алгоритм ART, который может изучать изменяющиеся двоичные схемы. Он представляет собой синтез ART и нечеткой логики.

Существуют и другие алгоритмы из семейства ART. Дополнительную информацию по данному вопросу вы найдете в разделе «Литература и ресурсы» в конце этой главы.

Использование ART1 для персонализации

Рассмотрим применение алгоритма ART1 для решения задачи персонализации. Сначала необходимо определить, что такое персонализация.

Определение персонализации

Идея персонализации не нова, многие компании занимаются ей в течение определенного времени. Повышением своей популярности персонализация обязана, в первую очередь, преимуществами ее использования в Глобальной сети. Internet-магазины позволяют осуществлять персонализацию практически в режиме реального времени. Перед тем как покупатель делает заказ, сайт может рекомендовать другие товары, которые могут больше подойти посетителю. Фактор времени для продавца очень важен, поскольку до покупки он может воздействовать на посетителя таким образом, чтобы изменить его решение в сторону увеличения своей прибыли.

Применение персонализации

Персонализация включает ввод определенной информации и вывод рекомендаций, предназначенных для пользователя, на основании некоторых расчетов. Метод осуществления персонализации различается в зависимости от типа данных на входе, требований к рекомендациям на выходе и в некоторых случаях от скорости и точности выполнения расчета.

Многие компании пользуются разнообразными способами персонализации. Большинство алгоритмов (или методов настройки существующих алгоритмов) хранятся в секрете, так как являются стратегически важными для владельцев. Amazon.com использует метод, который называется «Фильтрация на основе сотрудничества». Этот метод выдает рекомендации на основании сходства между покупками данного покупателя и других покупателей. Для выделения подгруппы покупателей из общей массы используется мера схожести. Получив подгруппу покупателей, система выдает рекомендации в зависимости от различий между членами подгруппы.

Персонализация с использованием ART1

Персонализация с использованием алгоритма ART1 состоит из двух этапов. Сначала выполняется стандартный алгоритм ART1 для векторов признаков (данных о покупателях). Далее, чтобы получить рекомендацию, анализируется вектор признаков (отображающий покупателя, которому нужно дать рекомендацию), а также новый элемент, так называемый *вектор суммирования* (Sum vector). Вектор суммирования, который не входит собственно в алгоритм ART1, представляет собой сумму столбцов векторов признаков в кластере (рис. 3.3).

Рассмотрим процесс выдачи рекомендации на примере. Предположим, что покупатель, которому мы должны дать рекомендацию, представлен вектором признаков u, входящим в кластер A. Содержимое вектора признаков соответствует примеру на рис. 3.1 (истории покупок клиента). Сначала по вектору суммирования необходимо определить, какие товары (столбцы) представлены в кластере (то есть не равны 0). Затем алгоритм находит самое большое значение в векторе суммирования, которое соответствует объекту в векторе признаков покупателя со значением 0. Оно представляет товар, не приобретенный покупателем, но популярный

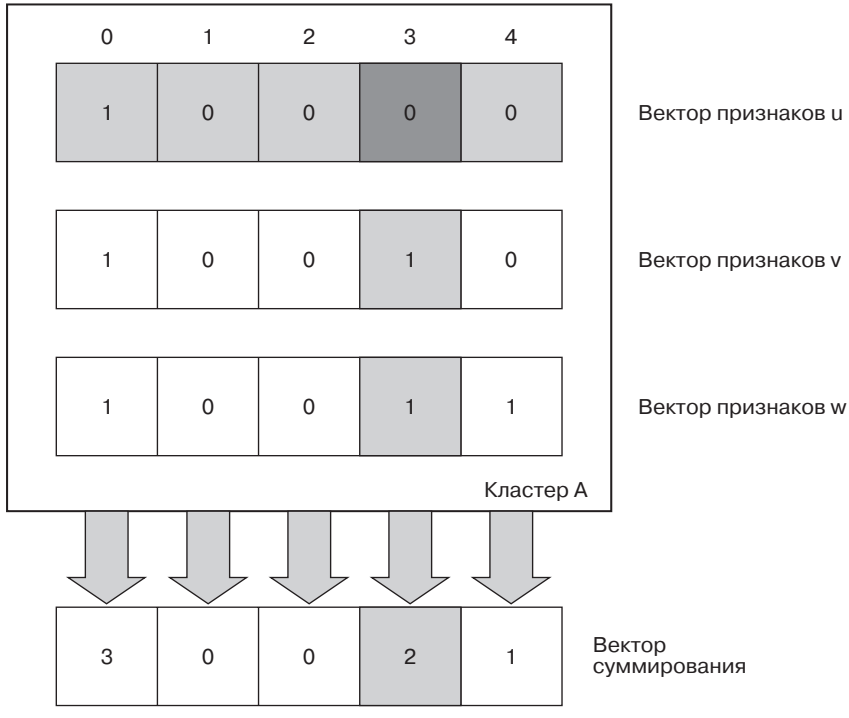


Рис. 3.3. Рекомендация товара с использованием вектора признаков и вектора суммирования

для кластера. Такая информация является основой для рекомендации. Предположение (или статистическое предвидение) заключается в следующем: 66% покупателей в кластере уже приобрели этот товар (в примере товар с номером три), значит, высока вероятность того, что данный клиент тоже пожелает его купить.

Совет *Вы можете найти исходный код алгоритма ART в папке/software/ch3 архива, который можно загрузить с сайта «ДМК Пресс» по адресу www.dmk.ru.*

Исходный код

Исходный код алгоритма ART1 включает процедуру выдачи рекомендаций, о которой рассказывалось выше. Полностью код содержится в архиве с примерами на сайте издательства «ДМК Пресс» и может быть скомпилирован в Linux или Windows с использованием библиотеки Cygwin.

Исходный код начинается с примера данных. Данные – это набор векторов признаков, которые отображают записи о покупках клиента. Как уже обсуждалось ранее, значение 1 соответствует приобретению товара покупателем, а 0 показывает,

что покупки не было. Векторы признаков (база данных о покупателях) и другие связанные структуры представлены в листинге 3.1.

Листинг 3.1. Структура данных ART1 для персонализации

```
#define MAX_ITEMS                (11)
#define MAX_CUSTOMERS            (10)
#define TOTAL_PROTOTYPE_VECTORS   (5)

const float beta = 1.0;     /* Небольшое положительное целое */
const float vigilance = 0.9;/* 0 <= внимательность < 1 */

int numPrototypeVectors = 0;/* Количество векторов-прототипов */

int prototypeVector[TOTAL_PROTOTYPE_VECTORS][MAX_ITEMS];

/* Вектор суммирования для выдачи рекомендаций */
int sumVector[TOTAL_PROTOTYPE_VECTORS][MAX_ITEMS];

/* Количество членов в кластерах */
int members[TOTAL_PROTOTYPE_VECTORS];

/* Номер кластера, к которому принадлежит покупатель */
int membership[MAX_CUSTOMERS];

/* Строковые названия элементов векторов */
char *itemName[MAX_ITEMS] = {
     "Hammer", "Paper", "Snickers", "Screwdriver",
     "Pen", "Kit-Kat", "Wrench", "Pencil",
     "Heath-Bar", "Tape-Measure", "Binder" };

 /*
  * Массив векторов-признаков. Поля представляют товар, который
  * приобретет покупатель. Нуль - товар покупателем еще не
  * приобретен
  */

/*       Hmr   Ppr   Snk   Scr   Pen   Kkt   Wrn   Pcl   Hth   Tpm   Bdr */
int database[MAX_CUSTOMERS][MAX_ITEMS] = {
        { 0,   0,   0,   0,   0,   1,   0,   0,   1,   0,   0},
        { 0,   1,   0,   0,   0,   0,   0,   1,   0,   0,   1},
        { 0,   0,   0,   1,   0,   0,   1,   0,   0,   1,   0},
        { 0,   0,   0,   0,   1,   0,   0,   1,   0,   0,   1},
        { 1,   0,   0,   1,   0,   0,   0,   0,   0,   1,   0},
        { 0,   0,   0,   0,   1,   0,   0,   0,   0,   0,   1},
        { 1,   0,   0,   1,   0,   0,   0,   0,   0,   0,   0},
        { 0,   0,   1,   0,   0,   0,   0,   0,   1,   0,   0},
```

```
        { 0,   0,   0,   0,   1,   0,   0,   1,   0,   0,   0},
        { 0,   0,   1,   0,   0,   1,   0,   0,   1,   0,   0}
};
```

Массив `prototypeVector` отображает векторы-прототипы для каждого кластера. Вектор `sumVector` используется только для выдачи рекомендаций и не входит в стандартный алгоритм ART1. Массив `members` показывает количество членов в определенном кластере, а массив `membership` устанавливает, к какому кластеру принадлежит вектор признаков покупателя. Наконец, массив `database` определяет уникальные векторы признаков для покупателей.

Функция `main` производит обработку данных с помощью алгоритма ART1, а затем выдает рекомендации на основе результатов выполнения алгоритма. Функция `main` представлена в листинге 3.2.

Листинг 3.2. Функция `main` при персонализации с использованием алгоритма ART1

```
int main()
{
  int customer;

  srand( time( NULL ) );

  initialize();

  performART1();

  displayCustomerDatabase();

  for (customer = 0 ; customer < MAX_CUSTOMERS ; customer++) {
    makeRecommendation( customer );

  }
  return 0;
}
```

После инициализации генератора случайных чисел функцией `srand` вызывается функция `initialize`, которая предназначена для очистки и инициализации структур, используемых алгоритмом ART1, а также алгоритмом выдачи рекомендаций. Алгоритм ART1 представлен функцией `performART1`. Рекомендации выдаются с помощью функции `makeRecommendation`. Функция `displayCustomerDatabase` отображает векторы-прототипы для каждого создаваемого кластера, а также векторы признаков в кластерах.

Листинг 3.3. Инициализация структур данных алгоритма

```
void initialize( void )
{
  int i, j;
```

```
  /* Очистка векторов-прототипов */
  for (i = 0 ; i < TOTAL_PROTOTYPE_VECTORS ; i++) {
    for (j = 0 ; j < MAX_ITEMS ; j++) {
      prototypeVector[i][j] = 0;
      sumVector[i][j] = 0;
    }
    members[i] = 0;
  }

  /* Сброс значения принадлежности векторов к кластерам */
  for (j = 0 ; j < MAX_CUSTOMERS ; j++) {
    membership[j] = -1;
  }

}
```

Функция `initialize` в листинге 3.3 очищает векторы-прототипы, вектор суммирования, а также массивы `members` и `membership`.

В листинге 3.4 представлены две вспомогательные функции – `vectorMagnitude` и `vectorBitwiseAnd`.

Листинг 3.4. Вспомогательные функции для алгоритма ART1

```
int vectorMagnitude( int *vector )
{
  int j, total = 0;

  for (j = 0 ; j < MAX_ITEMS ; j++) {
    if (vector[j] == 1) total++;
  }

  return total;
}

void vectorBitwiseAnd( int *result, int *v, int *w )
{
  int i;
  for (i = 0 ; i < MAX_ITEMS ; i++) {
    result[i] = (v[i] && w[i]);
  }

  return;
}
```

Функция `vectorMagnitude` просто считает количество объектов в векторе (со значением 1) и выдает сумму. Функция `vectorBitwiseAnd` выполняет побитовую операцию И с двумя векторами, в результате чего образуется новый вектор.

Для персонализации также потребуются две функции управления векторами-прототипами, которые используются, чтобы создать новый кластер и обновить его на основе произошедших в нем изменений (данные удалены из кластера или помещены в него). Эти функции представлены в листинге 3.5.

Листинг 3.5. Функции управления векторами-прототипами

```
int createNewPrototypeVector( int *example )
{
  int i, cluster;

  for (cluster = 0; cluster < TOTAL_PROTOTYPE_VECTORS; cluster++)
  {
    if (members[cluster] == 0) break;
  }

  if (cluster == TOTAL_PROTOTYPE_VECTORS) assert(0);

#ifdef DEBUG
  printf("Creating new cluster %d\n", cluster);
#endif

  numPrototypeVectors++;

  for (i = 0 ; i < MAX_ITEMS ; i++) {

    prototypeVector[cluster][i] = example[i];

#ifdef DEBUG
    printf("%1d ", example[i]);
#endif

  }

  members[cluster] = 1;
#ifdef DEBUG
  printf("\n");
#endif

  return cluster;
}

void updatePrototypeVectors( int cluster )
{
  int item, customer, first = 1;

  assert( cluster >= 0);

#ifdef DEBUG
  printf("Recomputing prototypeVector %d (%d)\n",
```

```
cluster, members[cluster]);
#endif
  for (item = 0 ; item < MAX_ITEMS ; item++) {
    prototypeVector[cluster][item] = 0;
    sumVector[cluster][item] = 0;
  }

  for (customer = 0 ; customer < MAX_CUSTOMERS ; customer++) {
    if (membership[customer] == cluster) {
      if (first) {
        for (item = 0 ; item < MAX_ITEMS ; item++) {
          prototypeVector[cluster][item] =
                   database[customer][item];
          sumVector[cluster][item] = database[customer][item];
        }
        first = 0;
      } else {
        for (item = 0 ; item < MAX_ITEMS ; item++) {
          prototypeVector[cluster][item] =
            prototypeVector[cluster][item] &&
               database[customer][item];
         sumVector[cluster][item] += database[customer][item];
        }
      }
    }
  }
  return;
}
```

Первая функция, `createNewPrototypeVector`, берет вектор признаков и создает для него новый кластер. Вектор признаков просто копируется в вектор-прототип кластера. Количество членов в кластере автоматически становится равным 1. Функция `updatePrototypeVectors` пересчитывает вектор-прототип на основании тех данных, которые в нем содержатся. Вспомните (уравнение 3.4), что вектор-прототип – это всего лишь результат применения операции побитового И для всех векторов признаков. Функция из листинга 3.5 загружает первый вектор признаков в вектор-прототип, а затем выполняет операцию И для последующих векторов признаков в кластере. Здесь также рассчитывается вектор `sumVector`, который используется только для выдачи рекомендаций и не входит в алгоритм ART1.

Реализация алгоритма ART1 представлена в листинге 3.6. Код включает отладочные директивы. Чтобы задействовать функцию отладки и отслеживать процесс выполнения алгоритма, измените строку `#undef` на `#define` в начале файла для константы `DEBUG`.

Листинг 3.6. Алгоритм ART1

```
int performART1( void )
{
  int andresult[MAX_ITEMS];
```

```
int pvec, magPE, magP, magE;
float result, test;
int index, done = 0;
int count = 50;

while (!done) {

  done = 1;

/* По всем покупателям */
for (index = 0 ; index < MAX_CUSTOMERS ; index++) {

    /* Шаг 3 */
    for (pvec = 0 ; pvec < TOTAL_PROTOTYPE_VECTORS ; pvec++) {
      /* Есть ли в этом кластере элементы? */
      if (members[pvec]) {

        vectorBitwiseAnd( andresult, &database[index][0],
                                &prototypeVector[pvec][0] );

        magPE = vectorMagnitude( andresult );
        magP  = vectorMagnitude( &prototypeVector[pvec][0] );
        magE  = vectorMagnitude( &database[index][0] );

        result = (float)magPE / (beta + (float)magP);

        test = (float)magE / (beta + (float)MAX_ITEMS);

        /* Выражение 3.2 */
       if (result > test) {

          /* Тест на внимательность / (Выражение 3.3) */
          if (((float)magPE/(float)magE) < vigilance) {

            int old;

            /* Убедиться, что это другой кластер */
            if (membership[index] != pvec) {
              /* Переместить покупателя в другой кластер */

              old = membership[index];
              membership[index] = pvec;

              if (old >= 0) {
                members[old]--;
                if (members[old] == 0) numPrototypeVectors--;
              }
              members[pvec]++;
```

```
                    /* Пересчитать векторы-прототипы для всех
                     * кластеров
                     */
                    if ((old >= 0) && (old < TOTAL_PROTOTYPE_VECTORS))
{
                      updatePrototypeVectors( old );
                    }

                    updatePrototypeVectors( pvec );

                    done = 0;
                    break;

                  } else {
                    /* Уже в этом кластере */
                  }

                } /* Тест на внимательность */
              }
            }
          } /* Цикл по векторам */

          /* Проверяем, обработан ли вектор */
          if (membership[index] == -1) {

           /* Не был найден подходящий кластер - создаем новый
            * кластер для этого вектора признаков
            */
            membership[index] =  createNewPrototypeVector
                              ( &database[index][0] );
            done = 0;

          }

} /* Цикл по покупателям */

    if (!count--) break;

  } /* Закончили */
  return 0;
}
```

Алгоритм ART1 очень прост: он сравнивает все векторы признаков со всеми векторами-прототипами кластера. Если в кластерах больше не происходит изменений, алгоритм считается выполненным, и мы возвращаемся к функции `main`. Представленный исходный код является реализацией уравнений 3.1–3.4. Переменные `mag*` представляют собой значимости векторов, которые рассчитываются один раз с целью повышения эффективности. Алгоритм также включает ряд оптимизаций, позволяющих игнорировать векторы-прототипы, которые не имеют членов (пустые кластеры не рассматриваются). Если в процессе

классификации вектор признаков не был помещен в кластер, будет автоматически создан новый кластер. Таким образом, строго выполняется уравнение 3.1.

Последняя функция, `makeRecommendation`, выдает рекомендацию для заданного покупателя (представленного вектором признаков) – см. листинг 3.7.

Листинг 3.7. Алгоритм рекомендации

```
void makeRecommendation ( int customer )
{
  int bestItem = -1;
  int val = 0;
  int item;

  for (item = 0 ; item < MAX_ITEMS ; item++) {

    if ((database[customer][item] == 0) &&
        (sumVector[membership[customer]][item] > val)) {
      bestItem = item;
      val = sumVector[membership[customer]][item];
    }

  }

  printf("For Customer %d, ", customer);

  if (bestItem >= 0) {
    printf("The best recommendation is %d (%s)\n",
            bestItem, itemName[bestItem]);
    printf("Owned by %d out of %d members of this cluster\n",
            sumVector[membership[customer]][bestItem],
            members[membership[customer]]);
  } else {
    printf("No recommendation can be made.\n");
  }

  printf("Already owns: ");
  for (item = 0 ; item < MAX_ITEMS ; item++) {

    if (database[customer][item]) printf("%s ", itemName[item]);
  }
  printf("\n\n");
}
```

Согласно рис. 3.3 алгоритм выполняет поиск самого популярного товара в кластере, который еще не был приобретен покупателем. Начальный цикл функции `makeRecommendation` находит соответствие, определяя самый популярный товар и записывая его номер (столбец функции `sumVector`). Оставшийся код выдает текстовую информацию о рекомендации (если ее возможно сделать). Функция также возвращает список всех товаров, которые приобрел покупатель (в качестве проверки результата).

Оптимизация алгоритма

Вы можете изменить три важных параметра алгоритма. Максимально допустимое количество кластеров (задается как константа TOTAL_PROTOTYPE_VECTORS) должно быть достаточно большим, чтобы алгоритм при необходимости мог создать новый кластер. Поскольку способность создавать новые кластеры является одним из основных преимуществ ART1, следует допустить формирование достаточного количества новых кластеров.

Бета-параметр и параметр внимательности очень важны для правильной ориентации алгоритма ART1. Большое значение имеет параметр внимательности, поскольку он определяет количество рекомендаций, которые могут быть сделаны. Если кластеры слишком велики, рекомендации могут быть неправильными, потому что векторы признаков в одном кластере будут слишком сильно различаться. Тот же результат будет получен, если кластеры слишком малы, так как в таких кластерах недостаточно векторов признаков, чтобы найти правильные соответствия. Бета-параметр (определенный Стефеном Галлантом) – это «разрушитель связей», который отдает предпочтение векторам-прототипам с большей степенью схожести.

Пример запуска

Рассмотрим лог выполнения алгоритма, полученный из программы выдачи рекомендаций, исходный код которой был описан выше. Исходный код был откомпилирован без использования константы DEBUG, поэтому внутренний механизм алгоритма ART1 не показан.

Первый элемент, выведенный в результате, представляет собой классификацию векторов признаков по кластерам. При этом показывается каждый из векторов-прототипов (отображающих кластеры) вместе с векторами признаков (см. листинг 3.8). На основе исходных данных были созданы всего четыре кластера несмотря на то, что были доступны пять.

Листинг 3.8. Отображение кластеров с использованием программы рекомендации

```
ProtoVector 0 : 0 0 0 0 0 0 0 0 0 1 0 0

Customer 0    : 0 0 0 0 0 0 1 0 0 1 0 0 : 0 :
Customer 7    : 0 0 1 0 0 0 0 0 0 1 0 0 : 0 :
Customer 9    : 0 0 1 0 0 1 0 0 1 0 0 : 0 :

ProtoVector 1 : 0 0 0 0 0 1 0 0 1 0 0 0

Customer 3    : 0 0 0 0 0 1 0 0 1 0 0 1 : 1 :
Customer 8    : 0 0 0 0 0 1 0 0 1 0 0 0 : 1 :

ProtoVector 2 : 0 0 0 1 0 0 0 0 0 0 0 0

Customer 2    : 0 0 0 1 0 0 1 0 0 1 0 : 2 :
```

```
Customer 4    : 1 0 0 1 0 0 0 0 0 1 0 : 2 :
Customer 6    : 1 0 0 1 0 0 0 0 1 0 0 : 2 :

ProtoVector 3 : 0 0 0 0 0 0 0 0 0 0 1

Customer 1    : 0 1 0 0 0 0 0 1 0 0 1 : 3 :
Customer 5    : 0 0 0 0 1 0 0 0 0 0 1 : 3 :

ProtoVector 4 : 0 0 0 0 0 0 0 0 0 0 0
```

В листинге 3.9 показана часть информации, которую выводит программа выдачи рекомендаций. Эта программа применяет текстовые имена к объектам, представленным вектором признаков, а также к рекомендации.

Листинг 3.9. Результат выполнения алгоритма выдачи рекомендаций

```
For Customer 0, The best recommendation is 2 (Snickers)
Owned by 2 out of 3 members of this cluster
Already owns: Kit-Kat Heath Bar

For Customer 1, The best recommendation is 4 (Pen)
Owned by 1 out of 2 members of this cluster
Already owns: Paper Pencil Binder

For Customer 2, The best recommendation is 0 (Hammer)
Owned by 2 out of 3 members of this cluster
Already owns: Screwdriver Wrench Tape-Measure

For Customer 3, No recommendation can be made.
Already owns: Pen Pencil Binder
```

В каждом случае алгоритм ART1 правильно сегментирует покупателей в группы на основании товаров, которые они приобрели. Покупатель 0 был помещен в кластер `Конфеты`, Покупатель 1 – в кластер `Офисные принадлежности`, а Покупатель 2 – в кластер `Инструменты`. Рекомендации определяются группами, которые должны соответствовать привычкам покупателя. Обратите внимание, что для Покупателя 3 не могут быть сделаны рекомендации. Причина в том, что вектор признаков данного покупателя такой же, как и вектор-прототип (нет отличий), а это значит, что покупатель приобрел все товары, которые были представлены кластером.

Аспекты соблюдения конфиденциальности

Другое интересное применение этого алгоритма – рекомендации книг для чтения. Представьте себе систему выдачи рекомендаций для пользователей библиотеки. Каждый пользователь закодирован как вектор признаков. Вектор признаков представляет книги, которые читал пользователь. Каждый элемент вектора признаков отображает небольшой диапазон книг в пределах системы

цифровой классификации (например, 100–103). Указанный диапазон включает книги по философии. При применении алгоритма ART1 кластеры векторов признаков будут усовершенствованы таким образом, чтобы идентифицировать процент людей в группе, которые читали определенную книгу. Эти данные могут использоваться для последующей рекомендации книги пользователю, который еще ее не читал.

Программа хорошо работает с данными тестирования, однако в реальных условиях ее применение неосуществимо по причине сохранения конфиденциальности. Большинство библиотек сообщают, что не хранят записи о том, какие книги читает каждый пользователь. Без этих данных выполнение алгоритма ART1 становится невозможным.

Существует также определенный страх, связанный с применением алгоритмов персонализации. При корректной работе алгоритмы могут предсказывать действия пользователя. Это может создать трудности для некоторых людей, которые полагают, что их поведение предсказывается исключительно на основании внешних факторов. Хотя при таком применении результат может быть полезен для пользователя, существуют и другие реализации, вмешивающиеся в частную жизнь людей. Для борьбы со страхом, связанным с подобным использованием алгоритма, большинство компаний информируют клиентов о политике соблюдения конфиденциальности. Это позволяет покупателям узнать, какая именно информация собирается, что с ней делают, а также кто ее видит или использует.

Другие области применения

Алгоритм ART1 предоставляет возможность классификации данных в отдельные сегменты (кластеры). Классификация может быть полезна как средство исследования классов (типов) кластеров. Кроме того, как видно по алгоритму персонализации, изучение членов отдельного кластера позволяет получить интересную информацию. Данный алгоритм можно использовать в следующих областях:

- статистике;
- распознавании образов;
- уменьшении диапазона поиска;
- биологии;
- поиске в сети Internet;
- добыче данных (data mining).

Итоги

В данной главе рассматривался простой алгоритм, который группирует данные в кластеры для системы выдачи рекомендаций. Изначально он создавался как инструмент, который может использоваться для обработки данных. Высокая эффективность алгоритма проявляется при обработке данных в сети Internet в коммерческих целях.

Пример алгоритма, представленный в этой главе, очень прост и работает с небольшим объемом данных. При персонализации в Internet данные могут включать не только отображение содержания Web-страницы, но и время, которое было потрачено на ее просмотр. Тип и отображение данных зависят от алгоритма, который выполняет персонализацию. При правильной кодировке в векторах признаков алгоритм ART1 может работать с широким диапазоном данных, отображающим многие аспекты поведения покупателя в сети Internet.

Несмотря на то что существует и опасная сторона применения алгоритмов персонализации, они могут быть очень эффективными инструментами при сборе самой разнообразной информации.

Литература и ресурсы

1. Wolfram Research. Мир математики Эрика Вайштайна (Eric Weisstein's World of Mathematics). Доступно по адресу http://mathworld.wolfram.com/.
2. Галлант С. Обучение в нейронных сетях (Gallant S. Neural Network Learning. – Cambridge, Mass.: MIT Press, 1994).
3. Карпентер Д., Гроссберг С. Массивная параллельная архитектура для самостоятельной машины, распознающей нейронные модели (Carpenter G., Grossberg S. A Massively Parallel Architecture for a Self-Organizing Neural Pattern Recognition Machine // Computer Vision, Graphics and Image Processing, 37: 54–115, 1987).

Глава 4. Алгоритмы муравья

В этой главе рассматривается интересный алгоритм, основанный на применении нескольких агентов, с помощью которого можно решать самые разнообразные задачи. *Алгоритмы муравья* (Ant algorithms), или оптимизация по принципу муравьиной колонии (это название было придумано изобретателем алгоритма, Марко Дориго (Marco Dorigo)), обладают специфическими свойствами, присущими муравьям, и используют их для ориентации в физическом пространстве. Природа предлагает различные методики для оптимизации некоторых процессов (как показано в других главах этой книги, например, «Метод отжига» и «Введение в генетические алгоритмы»). Алгоритмы муравья особенно интересны потому, что их можно использовать для решения не только статичных, но и динамических проблем, например, проблем маршрутизации в меняющихся сетях.

Естественная мотивация

Хотя муравьи и слепы, они умеют перемещаться по сложной местности, находить пищу на большом расстоянии от муравейника и успешно возвращаться домой. Выделяя ферменты во время перемещения, муравьи изменяют окружающую среду, обеспечивают коммуникацию, а также отыскивают обратный путь в муравейник.

Самое удивительное в данном процессе – это то, что муравьи умеют находить самый оптимальный путь между муравейником и внешними точками. Чем больше муравьев используют один и тот же путь, тем выше концентрация ферментов на этом пути. Чем ближе внешняя точка к муравейнику, тем больше раз к ней перемещались муравьи. Что касается более удаленной точки, то ее муравьи достигают реже, поэтому по дороге к ней они применяют более сильные ферменты. Чем выше концентрация ферментов на пути, тем предпочтительнее он для муравьев по сравнению с другими доступными. Так муравьиная «логика» позволяет выбирать более короткий путь между конечными точками.

Алгоритмы муравья интересны, поскольку отражают ряд специфических свойств, присущих самим муравьям. Муравьи легко вступают в сотрудничество и работают вместе для достижения общей цели. Алгоритмы муравья работают так же, как муравьи. Это выражается в том, что смоделированные муравьи совместно решают проблему и помогают другим муравьям в дальнейшей оптимизации решения.

Рассмотрим пример, представленный на рис. 4.1. Два муравья из муравейника должны добраться до пищи, которая находится за препятствием. Во время перемещения каждый муравей выделяет немного фермента, используя его в качестве маркера.

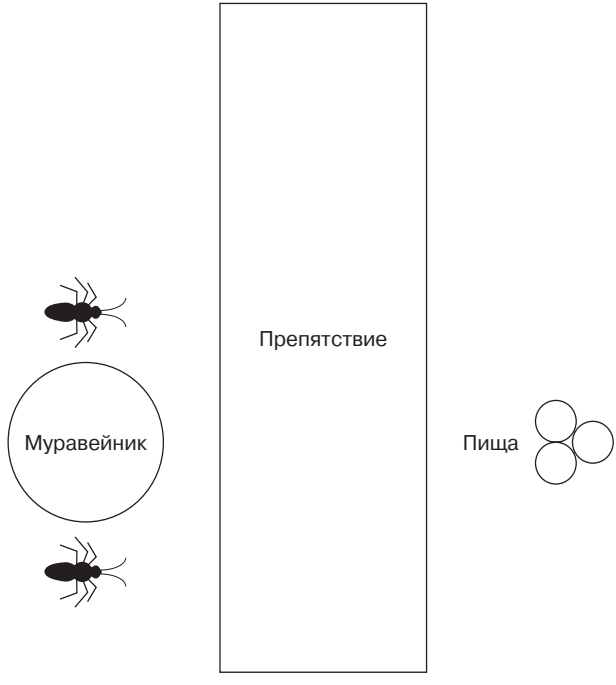


Рис. 4.1. Начальная конфигурация (T_0)

При прочих равных каждый муравей выберет свой путь. Первый муравей выбирает верхний путь, а второй – нижний. Так как нижний путь в два раза короче верхнего, второй муравей достигнет цели за время T_1. Первый муравей в этот момент пройдет только половину пути (рис. 4.2).

Когда один муравей достигает пищи, он берет один из объектов и возвращается к муравейнику по тому же пути. За время T_2 второй муравей вернулся в муравейник с пищей, а первый муравей достиг пищи (рис. 4.3).

Вспомните, что при перемещении каждого муравья на пути остается немного фермента. Для первого муравья за время T_0–T_2 путь был покрыт ферментом только один раз. В то же самое время второй муравей покрыл путь ферментом дважды. За время T_4 первый муравей вернулся в муравейник, а второй муравей уже успел еще раз сходить к еде и вернуться. При этом концентрация фермента на нижнем пути будет в два раза выше, чем на верхнем. Поэтому первый муравей в следующий раз выберет нижний путь, поскольку там концентрация фермента выше.

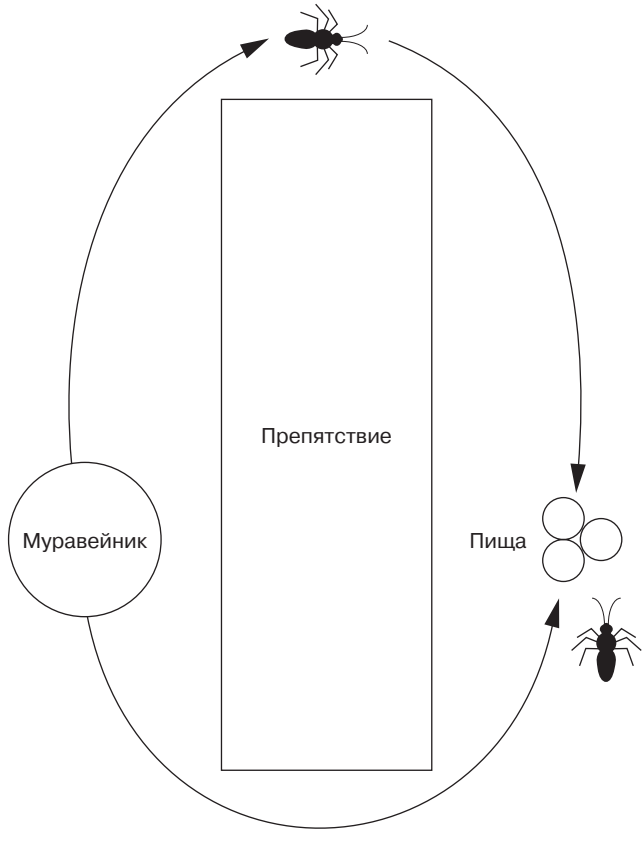


Рис. 4.2. Прошел один период времени (T_1)

В этом и состоит базовая идея алгоритма муравья – оптимизация путем непрямой связи между автономными агентами.

Алгоритм муравья

Подробно рассмотрим алгоритм муравья, чтобы понять, как он работает при решении конкретной проблемы.

Граф

Предположим, что окружающая среда для муравьев представляет собой закрытую двумерную сеть. Вспомните, что сеть – это группа узлов, соединенных посредством граней. Каждая грань имеет вес, который мы обозначим как расстояние между двумя узлами, соединенными ею. Граф двунаправленный, поэтому муравей может путешествовать по грани в любом направлении (рис. 4.4).

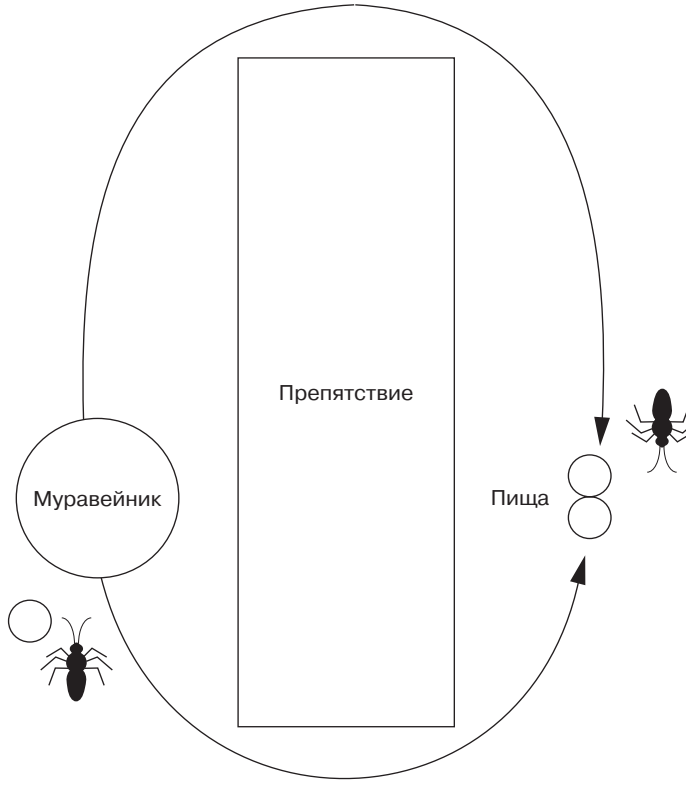
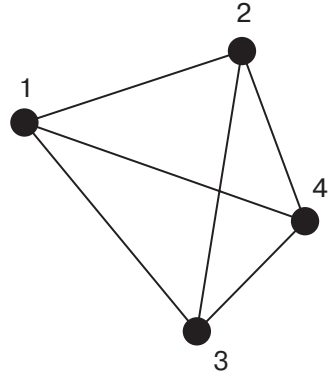


Рис. 4.3. Прошло два периода времени (T_2)



Граф с вершинами V = {1,2,3,4}
Грани E = {{1,2},{1,4},{1,3},{2,3},{2.4},{3.4}}

Рис. 4.4. Пример полностью замкнутой двумерной сети V с набором граней E

Муравей

Муравей – это программный агент, который является членом большой колонии и используется для решения какой-либо проблемы. Муравей снабжается набором простых правил, которые позволяют ему выбирать путь в графе. Он поддерживает *список табу* (tabu list), то есть список узлов, которые он уже посетил. Таким образом, муравей должен проходить через каждый узел только один раз. Путь между двумя узлами графа, по которому муравей посетил каждый узел только один раз, называется *путем Гамильтона* (Hamiltonian path), по имени математика сэра Уильяма Гамильтона (Sir William Hamilton).

Узлы в списке «текущего путешествия» располагаются в том порядке, в котором муравей посещал их. Позже список используется для определения протяженности пути между узлами.

Настоящий муравей во время перемещения по пути будет оставлять за собой фермент. В алгоритме муравья агент оставляет фермент на гранях сети после завершения путешествия. О том, как это происходит, рассказывается в разделе «Путешествие муравья».

Начальная популяция

После создания популяция муравьев поровну распределяется по узлам сети. Необходимо равное разделение муравьев между узлами, чтобы все узлы имели одинаковые шансы стать отправной точкой. Если все муравьи начнут движение из одной точки, это будет означать, что данная точка является оптимальной для старта, а на самом деле мы этого не знаем.

Движение муравья

Движение муравья основывается на одном и очень простом вероятностном уравнении. Если муравей еще не закончил *путь* (path), то есть не посетил все узлы сети, для определения следующей грани пути используется уравнение 4.1:

$$P = \frac{\tau(r,u)^a \times \eta(r,u)^b}{\sum_k \tau(r,u)^a \times \eta(r,u)^b} \qquad (4.1)$$

Здесь $\tau(r,u)$ – интенсивность фермента на грани между узлами r и u, $\tau(r,u)$ – функция, которая представляет измерение обратного расстояния для грани, α – вес фермента, а β – коэффициент эвристики. Параметры α и β определяют относительную значимость двух параметров, а также их влияние на уравнение. Вспомните, что муравей путешествует только по узлам, которые еще не были посещены (как указано списком табу). Поэтому вероятность рассчитывается только для граней, которые ведут к еще не посещенным узлам. Переменная k представляет грани, которые еще не были посещены.

Путешествие муравья

Пройденный муравьем путь отображается, когда муравей посетит все узлы диаграммы. Обратите внимание, что циклы запрещены, поскольку в алгоритм включен список табу. После завершения длина пути может быть подсчитана – она равна

сумме всех граней, по которым путешествовал муравей. Уравнение 4.2 показывает количество фермента, который был оставлен на каждой грани пути для муравья k. Переменная Q является константой.

$$\Delta\tau_{ij}^{k}(t) = \frac{Q}{L^{k}(t)} \tag{4.2}$$

Результат уравнения является средством измерения пути, – короткий путь характеризуется высокой концентрацией фермента, а более длинный путь – более низкой. Затем полученный результат используется в уравнении 4.3, чтобы увеличить количество фермента вдоль каждой грани пройденного муравьем пути.

$$\tau_{ij}(t) = \Delta\tau_{ij}(t) + (\tau_{ij}^{k}(t) \times \rho) \tag{4.3}$$

Обратите внимание, что данное уравнение применяется ко всему пути, при этом каждая грань помечается ферментом пропорционально длине пути. Поэтому следует дождаться, пока муравей закончит путешествие и только потом обновить уровни фермента, в противном случае истинная длина пути останется неизвестной. Константа ρ – значение между 0 и 1.

Испарение фермента

В начале пути у каждой грани есть шанс быть выбранной. Чтобы постепенно удалить грани, которые входят в худшие пути в сети, ко всем граням применяется процедура *испарения фермента* (Pheromone evaporation). Используя константу ρ из уравнения 4.3, мы получаем уравнение 4.4.

$$\tau_{ij}(t) = \tau_{ij}(t) \times (1-\rho) \tag{4.4}$$

Поэтому для испарения фермента используется обратный коэффициент обновления пути.

Повторный запуск

После того как путь муравья завершен, грани обновлены в соответствии с длиной пути и произошло испарение фермента на всех гранях, алгоритм запускается повторно. Список табу очищается, и длина пути обнуляется. Муравьям разрешается перемещаться по сети, основывая выбор грани на уравнении 4.1

Этот процесс может выполняться для постоянного количества путей или до момента, когда на протяжении нескольких запусков не было отмечено повторных изменений. Затем определяется лучший путь, который и является решением.

Пример итерации

Давайте разберем функционирование алгоритма на простом примере, чтобы увидеть, как работают уравнения. Вспомните (рис. 4.1) простой сценарий с двумя муравьями, которые выбирают два разных пути для достижения одной цели. На рис. 4.5 показан этот пример с двумя гранями между двумя узлами (V_0 и V_1). Каждая грань инициализируется и имеет одинаковые шансы на то, чтобы быть выбранной.

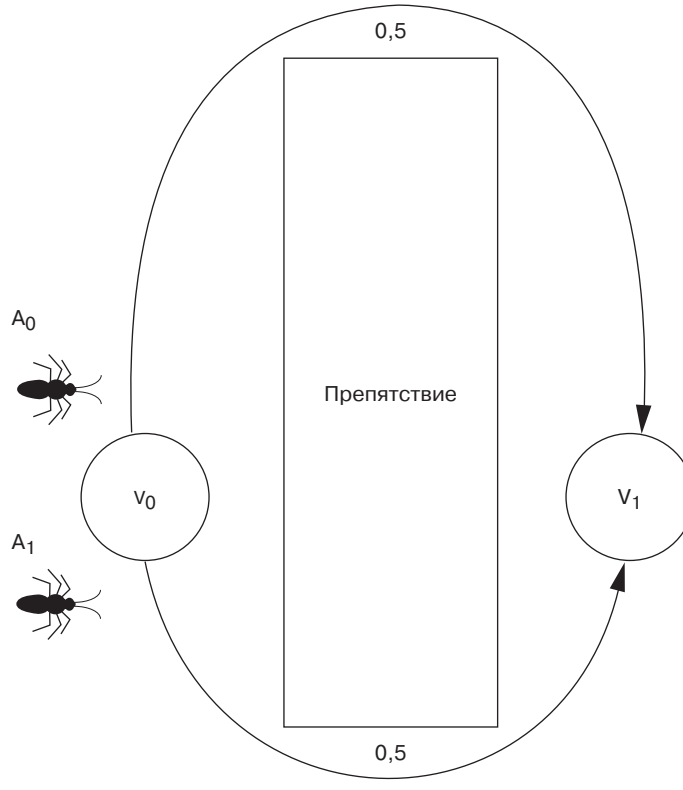


Рис. 4.5. Начальная конфигурация проблемы

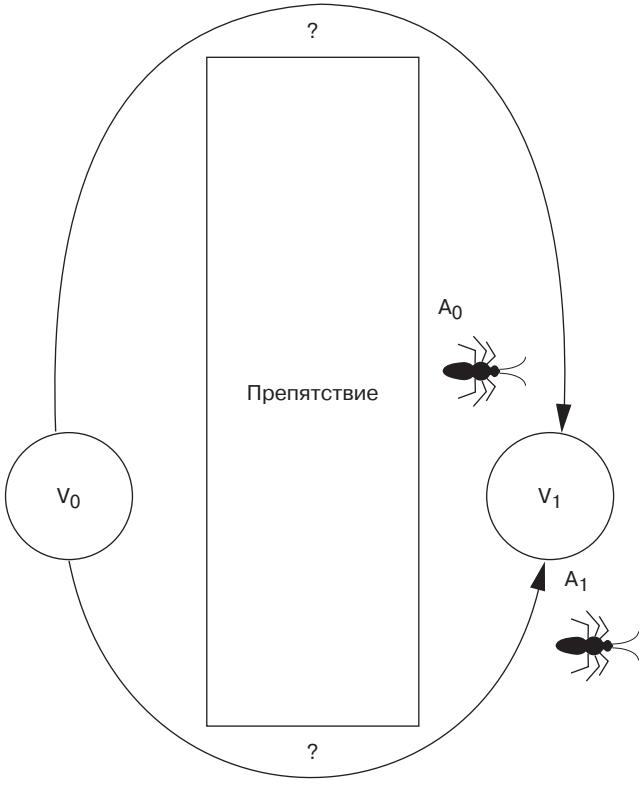
Два муравья находятся в узле V_0 и помечаются как A_0 и A_1. Так как вероятность выбора любого пути одинакова, в этом цикле мы проигнорируем уравнение выбора пути. На рис. 4.6 каждый муравей выбирает свой путь (муравей A_0 идет по верхнему пути, а муравей A_1 – по нижнему).

В таблице на рис. 4.6 показано, что муравей A_0 сделал 20 шагов, а муравей A_1 – только 10. По уравнению 4.2 мы рассчитываем количество фермента, которое должно быть «нанесено».

Примечание *Работу алгоритма можно изменить, переопределив его параметры (например, ρ, α или β), например придав больший вес ферменту или расстоянию между узлами. Подробнее об этом рассказывается после обсуждения исходного кода.*

Далее по уравнению 4.3 рассчитывается количество фермента, которое будет применено. Для муравья A_0 результат составляет:

$= 0,1 + (0,5 \times 0,6) = 0,4.$



$\rho = 0,6$
$\alpha = 3,0$
$\beta = 1,0$

	A_0	A_1
Пройденное расстояние	20	10
Уровень фермента Q/пройденное расстояние	0,5	1,0

Рис. 4.6. Путь муравья завершен

Для муравья A_1 результат составляет:

$= 0,1 + (1,0 \times 0,6) = 0,7.$

Далее с помощью уравнения 4.4 определяется, какая часть фермента испарится и, соответственно, сколько останется. Результаты (для каждого пути) составляют:

$= 0,4 \times (1,0 - 0,6) = 0,16$
$= 0,7 \times (1,0 - 0,6) = 0,28.$

Эти значения представляют новое количество фермента для каждого пути (верхнего и нижнего, соответственно). Теперь переместим муравьев обратно в узел V_0 и воспользуемся вероятностным уравнением выбора пути 4.1, чтобы определить, какой путь должны выбрать муравьи.

Вероятность того, что муравей выберет верхний путь (представленный количеством фермента 0,16), составляет:

$(0,16)^{3,0} \times (0,5)^{1,0} / ((0,16)^{3,0} \times (0,5)^{1,0}) + ((0,28)^{3,0} \times (1,0)^{1,0}) = 0,002048 / 0,024 =$
$= P(0,085).$

Вероятность того, что муравей выберет нижний путь (представленный количеством фермента 0,28), составляет:

$(0,28)^{3,0} \times (1,0)^{1,0} / ((0,16)^{3,0} \times (0,5)^{1,0}) + ((0,28)^{3,0} \times (1,0)^{1,0}) = 0,021952 / 0,024 = = P(0,915)$.

При сопоставлении двух вероятностей оба муравья выберут нижний путь, который является наиболее оптимальным.

Пример задачи

В качестве примера рассмотрим *задачу коммивояжера* (Traveling Salesman Problem – TSP). Она заключается в том, чтобы найти кратчайший путь между городами, при котором каждый город будет посещен всего один раз. То есть надо найти кратчайший Гамильтонов путь в графе, где в качестве узлов выступают города, а в качестве граней – соединяющие их дороги. Математики впервые изучили задачу TSP в 1930-е гг., в частности, ею занимался Карл Менгер (Karl Menger) в Вене. Следует отметить, что похожие задачи исследовались еще в 19 в. ирландским математиком сэром Уильямом Роуэном Гамильтоном (Sir William Rowan Hamilton).

В следующем разделе рассмотрен исходный код программы, которая используется для решения задачи TSP, и представлены варианты решения.

Совет *Исходный код алгоритма муравья для решения задачи коммивояжера вы сможете найти в папке ./software/ch4, содержащейся в архиве с примерами на сайте издательства «ДМК Пресс» www.dmk.ru.*

Исходный код

Следующие листинги иллюстрируют алгоритм муравья, который используется для поиска оптимальных решений задачи коммивояжера.

Сначала изучим структуру данных как для городов, так и для агентов (муравьев), которые будут по ним путешествовать. Листинг 4.1 включает типы данных и символьные константы, которые используются для представления городов и муравьев.

Листинг 4.1. Типы данных и символьные константы для представления городов и муравьев

```
#define MAX_CITIES    30
#define MAX_DISTANCE  100

#define MAX_TOUR   (MAX_CITIES * MAX_DISTANCE)

typedef struct {
   int x;
   int y;
} cityType;
```

```
#define MAX_ANTS   30

typedef struct {
   int curCity;
   int nextCity;
   unsigned char tabu[MAX_CITIES];
   int pathIndex;
   unsigned char path[MAX_CITIES];
   double tourLength;
} antType;
```

Структура `cityType` используется для определения города в матрице `MAX_DISTANCE` на `MAX_DISTANCE`. Структура `antType` представляет одного муравья. Кроме отслеживания текущего и следующего города на пути (поля `curCity` и `nextCity`) каждый муравей также должен учитывать города, которые уже были посещены (массив `tabu`) и длину пройденного пути. Наконец, общая длина пути сохраняется в поле `tourLength`.

В зависимости от размера задачи TSP иногда полезно изменить параметры в уравнениях. Ниже приводятся значения параметров по умолчанию для задачи TSP с 30 городами (листинг 4.2). Изменения параметров будут описаны в следующих разделах.

Листинг 4.2. Параметры задачи, для решения которой используется алгоритм муравья

```
#define ALPHA      1.0
#define BETA       5.0   /* Приоритет расстояния над количеством
                          * фермента
                          */
#define RHO        0.5   /* Интенсивность / Испарение */
#define QVAL       100

#define MAX_TOURS  20

#define MAX_TIME       (MAX_TOURS * MAX_CITIES)

#define INIT_PHEROMONE   (1.0 / MAX_CITIES)
```

Параметр `ALPHA` (α) определяет относительную значимость пути (количество фермента на пути). Параметр `BETA` (β) устанавливает относительную значимость видимости (обратной расстоянию). Параметр `RHO` (ρ) используется как коэффициент количества фермента, которое муравей оставляет на пути (этот параметр также известен как *продолжительность пути*), где (`1.0`-ρ) показывает коэффициент испарения на пути после его завершения. Параметр `QVAL` (или просто Q в уравнении 4.2) – это константа, относящаяся к количеству фермента, которое было оставлено на пути. Остальные константы будут рассмотрены при описании исходного кода алгоритма.

Структуры данных в алгоритме включают массивы `cities` и `ants`. Другой специальный двумерный массив, `distance`, содержит предварительно рассчитанные расстояния от одного города до всех других. Уровни фермента сохраняются в массиве `pheromone`. Каждый двумерный массив в алгоритме использует первый индекс в качестве начала грани, то есть исходного города, а второй – в качестве конечного. Все глобальные структуры данных представлены в листинге 4.3, а функция инициализации – в листинге 4.4.

Листинг 4.3. Глобальные структуры данных

```
cityType cities[MAX_CITIES];


antType ants[MAX_ANTS];

              /*   Из          В     */
double   distance[MAX_CITIES][MAX_CITIES];

              /*    Из         В      */
double pheromone[MAX_CITIES][MAX_CITIES];

double  best=(double)MAX_TOUR;
int     bestIndex;
```

Сначала мы рассмотрим функцию инициализации `init` (см. листинг 4.4).

Листинг 4.4. Функция инициализации

```
void init( void )
{
  int from, to, ant;

  /* Создание городов */
  for (from = 0 ; from < MAX_CITIES ; from++) {

    /* Случайным образом располагаем города */
    cities[from].x = getRand( MAX_DISTANCE );
    cities[from].y = getRand( MAX_DISTANCE );

    for (to = 0 ; to < MAX_CITIES ; to++) {
      distance[from][to] = 0.0;
      pheromone[from][to] = INIT_PHEROMONE;
    }

  }

  /* Вычисляем расстояние между городами */
  for ( from = 0 ; from < MAX_CITIES ; from++) {

    for ( to = 0 ; to < MAX_CITIES ; to++) {
```

```
        if ((to != from) && (distance[from][to] == 0.0)) {
          int xd = abs(cities[from].x - cities[to].x);
          int yd = abs(cities[from].y - cities[to].y);

          distance[from][to] = sqrt( (xd * xd) + (yd * yd) );
          distance[to][from] = distance[from][to];
        }

      }

    }

    /* Инициализация муравьев */
    to = 0;
    for ( ant = 0 ; ant < MAX_ANTS ; ant++ ) {

      /* Распределяем муравьев по городам равномерно */
      if (to == MAX_CITIES) to = 0;
      ants[ant].curCity = to++;

      for ( from = 0 ; from < MAX_CITIES ; from++ ) {
        ants[ant].tabu[from] = 0;
        ants[ant].path[from] = -1;
      }

      ants[ant].pathIndex = 1;
      ants[ant].path[0] = ants[ant].curCity;
      ants[ant].nextCity = -1;
      ants[ant].tourLength = 0.0;

     /* Помещаем исходный город, в котором находится муравей,
      * в список табу
      */
      ants[ant].tabu[ants[ant].curCity] = 1;

    }
  }
```

Функция `init` выполняет три базовых действия, которые требуются для подготовки алгоритма муравья. Первое действие – это создание городов. Для каждого города, который требуется создать (количество задано константой `MAX_CITIES`) генерируются произвольные числа для координат по осям x и y, которые затем сохраняются в запись текущего города. Кроме того, во время создания городов одновременно очищаются массивы `distance` и `pheromone`.

Далее, немного оптимизируя алгоритм, мы выполняем предварительный расчет расстояний между всеми городами, которые были созданы. Расстояние вычисляется с помощью обычной двумерной геометрии координат (теорема Пифагора).

Наконец, инициализируется массив ant. Вспомните, что муравьи должны быть равномерно распределены по всем городам. Для этого используется переменная to в качестве счетчика превдоцикла. Когда значение переменной to становится равным номеру последнего города, она возвращается к первому городу (городу 0), и процесс повторяется. Поле curCity в структуре ant устанавливается в значение текущего города (первого города для муравья). Затем очищаются списки tabu и path. Ноль в массиве tabu обозначает, что город еще не был посещен; единица свидетельствует, что город уже включен в путь муравья. В массив path помещается номер текущего города для муравья, и очищается поле tourLength (длина пути).

После завершения пути каждый муравей должен быть повторно инициализирован, а затем распределен по графу. Это выполняет функция restartAnts (листинг 4.5).

Листинг 4.5. Функции restartAnts предназначена для повторной инициализации всех муравьев

```
void restartAnts( void )
{
  int ant, i, to=0;

  for ( ant = 0 ; ant < MAX_ANTS ; ant++ ) {

    if (ants[ant].tourLength < best) {
      best = ants[ant].tourLength;
      bestIndex = ant;
    }

    ants[ant].nextCity = -1;
    ants[ant].tourLength = 0.0;

    for (i = 0 ; i < MAX_CITIES ; i++) {
      ants[ant].tabu[i] = 0;
      ants[ant].path[i] = -1;
    }

    if (to == MAX_CITIES) to = 0;
    ants[ant].curCity = to++;
    ants[ant].pathIndex = 1;
    ants[ant].path[0] = ants[ant].curCity;

    ants[ant].tabu[ants[ant].curCity] = 1;
  }

}
```

Во время повторной инициализации самая оптимальная длина пути сохраняется, чтобы можно было оценить прогресс муравьев. Затем структура алгоритма очищается и повторно инициализируется – начинается следующее путешествие.

Функция `selectNextCity` позволяет выбрать следующий город для составления пути. Она вызывает функцию `antProduct`, которая используется для расчета уравнения 4.1 (см. листинг 4.6). Функция `pow` (возведение числа x в степень y) является частью стандартной математической библиотеки.

Листинг 4.6. Функции `antProduct` и `selectNextCity`

```
double antProduct( int from, int to )
{
  return (( pow( pheromone[from][to], ALPHA ) *
            pow( (1.0 / distance[from][to]), BETA ) ));
}

int selectNextCity( int ant )
{
  int from, to;
  double denom=0.0;

  /* Выбрать следующий город */
  from = ants[ant].curCity;

  /* Расчет знаменателя */
  for (to = 0 ; to < MAX_CITIES ; to++) {
    if (ants[ant].tabu[to] == 0) {
      denom += antProduct( from, to );
    }
  }

  assert(denom != 0.0);
  do {

    double p;

    to++;
    if (to >= MAX_CITIES) to = 0;

    if ( ants[ant].tabu[to] == 0 ) {

      p = antProduct(from, to)/denom;

      if (getSRand() < p ) break;

    }

  } while (1);

  return to;

}
```

Функция `selectNextCity` вызывается для заданного муравья и определяет, которую из граней выбрать в соответствии со списком `tabu`. Выбор грани основывается на вероятностном уравнении 4.1, которое вычисляет коэффициент заданного пути от суммы других путей. Первой задачей функции `selectNextCity` является расчет знаменателя выражения. После этого все грани, которые еще не были выбраны, проверяются с помощью уравнения 4.1, чтобы муравей мог выбрать путь. Как только грань найдена, функция определяет город, к которому перейдет муравей.

Следующая функция, `simulateAnts`, рассчитывает движения муравьев по графу (листинг 4.7).

Листинг 4.7. Функция `siulateAnts`

```
int simulateAnts( void )
{
  int k;
  int moving = 0;

  for (k = 0 ; k < MAX_ANTS ; k++) {

    /* Убедиться, что у муравья есть куда идти */
    if (ants[k].pathIndex < MAX_CITIES) {

      ants[k].nextCity = selectNextCity( k );

      ants[k].tabu[ants[k].nextCity] = 1;

      ants[k].path[ants[k].pathIndex++] = ants[k].nextCity;

      ants[k].tourLength +=
            distance[ants[k].curCity][ants[k].nextCity];

        /* Обработка окончания путешествия (из последнего города
         * в первый)
         */
      if (ants[k].pathIndex == MAX_CITIES) {
        ants[k].tourLength +=
          distance[ants[k].path[MAX_CITIES-1]][ants[k].path[0]];
      }

      ants[k].curCity = ants[k].nextCity;

      moving++;

     }
  }

  return moving;
}
```

Функция simulateAnts проходит по массиву ant и перемещает муравьев из текущего города в новый город, предлагаемый вероятностным уравнением 4.1. Программа отмечает поле pathIndex, чтобы убедиться, что муравей еще не закончил путь. После этого вызывается функция selectNextCity, которая определяет следующую грань. Затем выбранный город загружается в структуру ant в поле nextCity, а также в списки path и tabu. Рассчитывается значение tourLength, чтобы сохранить длину пути. Наконец, если достигнут конец пути, в значение tourLength добавляется расстояние до начального города. Это позволяет построить полный путь через все остальные города и вернуться в начальный город.

Один вызов функции simulateAnts позволяет каждому муравью перейти от одного города к другому. Функция simulateAnts возвращает нуль, если путь уже завершен, а в противном случае – значение, отличное от нуля.

После завершения пути выполняется процесс обновления путей. Он включает не только обновление путей по количеству фермента, оставленного муравьями, но и существующего фермента, часть которого испарилась. Функция updateTrails выполняет эту часть алгоритма (листинг 4.8).

Листинг 4.8. Функция updateTrails – испарение и размещение нового фермента

```
void updateTrails( void )
{
  int from, to, i, ant;

  /* Испарение фермента */
  for (from = 0 ; from < MAX_CITIES ; from++) {

    for (to = 0 ; to < MAX_CITIES ; to++) {

      if (from != to) {

         pheromone[from][to] *= (1.0 - RHO);

        if (pheromone[from][to] < 0.0)
         pheromone[from][to] = INIT_PHEROMONE;

      }

    }

  }

  /* Нанесение нового фермента */

  /* Для пути каждого муравья */
  for (ant = 0 ; ant < MAX_ANTS ; ant++) {

    /* Обновляем каждый шаг пути */
    for (i = 0 ; i < MAX_CITIES ; i++) {
```

```
        if (i < MAX_CITIES-1) {
          from = ants[ant].path[i];
          to = ants[ant].path[i+1];
        } else {
          from = ants[ant].path[i];
          to = ants[ant].path[0];
        }

        pheromone[from][to] += (QVAL / ants[ant].tourLength);
        pheromone[to][from] = pheromone[from][to];

      }

    }

    for (from = 0 ; from < MAX_CITIES ; from++) {
      for (to = 0 ; to < MAX_CITIES ; to++) {
        pheromone[from][to] *= RHO;
      }
    }

  }
```

Первая задача функции `updateTrails` заключается в том, чтобы «испарить» часть фермента, который уже находится на пути. Она выполняется на всех гранях с помощью уравнения 4.4. Следующая задача – получение нового фермента на путях, пройденных муравьями во время последних перемещений. Для этого необходимо пройти по элементам массива `ant` и, руководствуясь значением поля `path`, «распылить» новый фермент на посещенной муравьем грани в соответствии с уравнением 4.2. Затем следует применить параметр `RHO`, чтобы снизить интенсивность фермента, который выделяют муравьи.

Следующая функция `main` очень проста (см. листинг 4.9). После инициализации генератора случайных чисел и установки начальных значений параметров алгоритма запускается симуляция для количества шагов, заданного константой `MAX_TIME`. Когда муравей завершает путь (который он проходит с помощью функции `simulateAnts`), вызывается функция `updateTrails` для того, чтобы изменить окружающую среду.

Листинг 4.9. Функция `main` для симуляции алгоритма муравья

```
int main()
{
  int curTime = 0;

  srand( time(NULL) );

  init();
```

```
    while (curTime++ < MAX_TIME) {
      if ( simulateAnts() == 0 ) {
        updateTrails();

        if (curTime != MAX_TIME)
          restartAnts();

        printf("Time is %d (%g)\n", curTime, best);

      }

    }

    printf("best tour %g\n", best);
    printf("\n\n");

    emitDataFile( bestIndex );

    return 0;
  }
```

Обратите внимание, что каждый раз при завершении пути вызывается функция `restartAnts`. Это делается для того, чтобы подготовиться к следующему путешествию по диаграмме. Функция `restartAnts` не вызывается только в том случае, если симуляция уже завершена. Дело в том, что функция уничтожает информацию о пути муравья, которую необходимо сохранить в самом конце, чтобы определить наилучший путь.

Совет *Здесь не представлена функция* `emitDataFile`*, которая используется для создания графиков, показанных в разделе «Примеры запуска». Ее исходный код вы можете загрузить с сайта издательства «ДМК Пресс» www.dmk.ru.*

Примеры запуска

Теперь рассмотрим несколько примеров запуска алгоритма муравья для задачи коммивояжера.

При первом запуске выполняется решение задачи для 30 городов (рис. 4.7). Были заданы следующие параметры: $\alpha = 1{,}0$, $\beta = 5{,}0$, $\rho = 0{,}5$, $Q = 100$.

При втором запуске выполняется решение задачи для 50 городов (рис. 4.8). Для нее были заданы те же параметры, что и для предыдущей.

Решение задачи в первом и втором случае было найдено быстрее, чем за пять путешествий муравьев. При каждом запуске количество муравьев равнялось количеству городов.

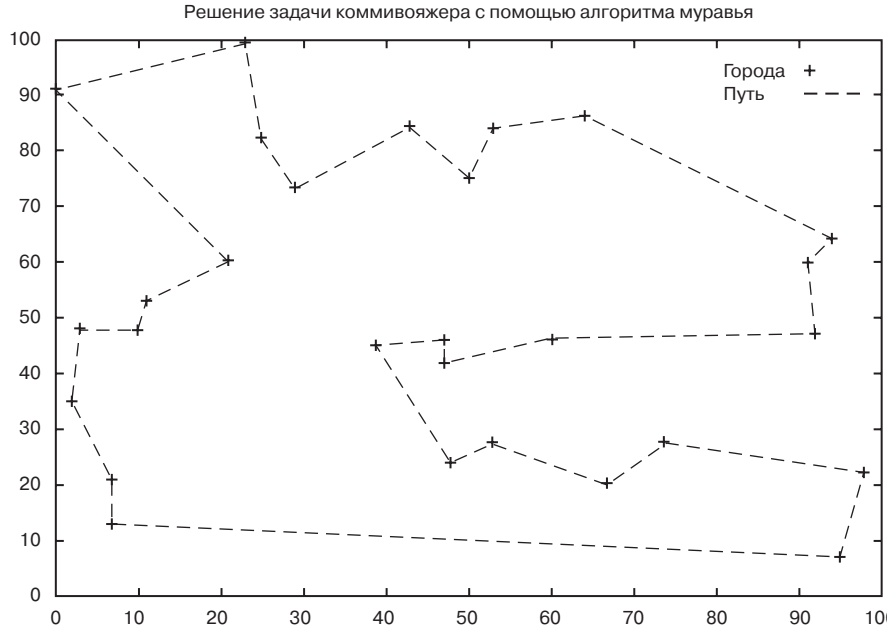


Рис. 4.7. Пример решения проблемы TSP для 30 городов

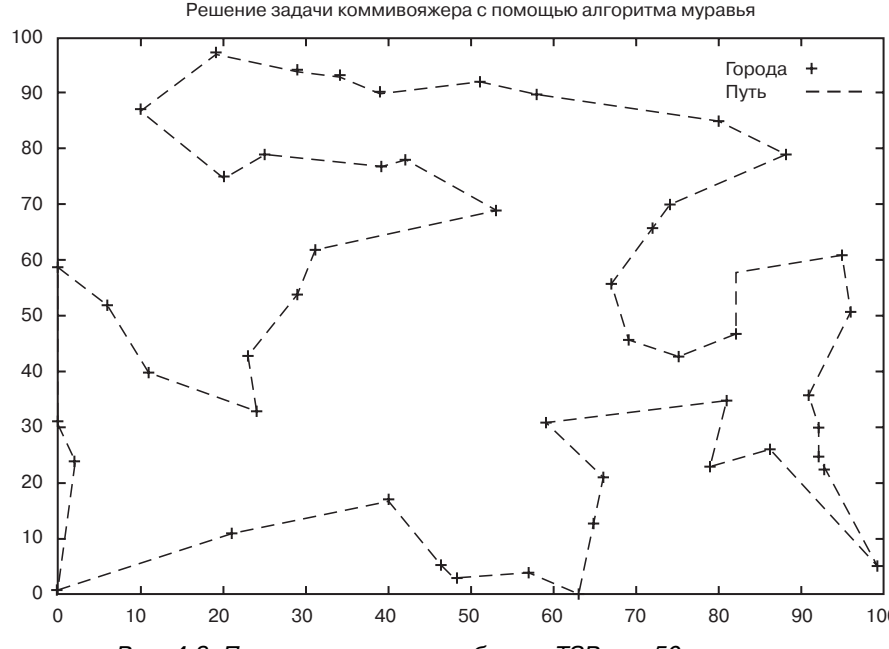


Рис. 4.8. Пример решения проблемы TSP для 50 городов

Изменение параметров алгоритма

Марко Дориго (изобретатель оптимизации по принципу муравьиной колонии) предлагает очень интересную дискуссию по параметрам алгоритма в статье «Система муравьев: оптимизация с помощью колонии сотрудничающих агентов» (The Ant System: Optimization by a Colony of Cooperating Agents). В этом разделе рассказывается о его предложениях по изменениям параметров алгоритма.

Alpha (α) / Beta (β)

Был открыт ряд комбинаций α/β, которые позволяют находить хорошие решения за небольшое время. Эти комбинации приведены в табл. 4.1.

Таблица 4.1. Комбинации параметров α/β

α	β
0,5	5,0
1,0	1,0
1,0	2,0
1,0	5,0

Параметр α ассоциируется с количеством фермента (из уравнения 4.1), а параметр β – с видимостью (длинной грани). Чем больше значение параметра, тем он важнее для вероятностного уравнения, которое используется при выборе грани. Обратите внимание, что в одном случае значимость параметров равна. Во всех других случаях видимость более важна при выборе пути.

Rho (ρ)

Параметр ρ представляет коэффициент, который применяется к распыляемому на пути ферменту, а (1,0 – ρ) представляет коэффициент испарения для существующего фермента. Были проведены тесты при $\rho > 0{,}5$, и все они показали интересные результаты. При установке значения $\rho < 0{,}5$ результаты были неудовлетворительными.

В первую очередь этот параметр определяет концентрацию фермента, которая сохранится на гранях.

Количество муравьев

Количество муравьев повлияло на качество полученных решений. Хотя увеличение количества муравьев может показаться хорошей идеей, наилучший результат достигается в том случае, если количество муравьев равно количеству городов.

Другие области применения

Алгоритм муравья может применяться для решения многих задач, таких как распределение ресурсов и работы.

При решении задачи распределения ресурсов (Quadratic Asigment Problem – QAP) необходимо задать группу ресурсов n для ряда адресатов m и при этом минимизировать расходы на перераспределение (то есть функция должна найти наилучший способ распределения ресурсов). Обнаружено, что алгоритм муравья дает решения такого же качества, как и другие, более стандартные способы.

Намного сложнее проблема распределения работы (Job-shop Sheduling Problem – JSP). В этой задаче группа машин M и заданий J (состоящих из последовательности действий, осуществляемых на машинах) должны быть распределены таким образом, чтобы все задания выполнялись за минимальное время. Хотя решения, найденные с помощью алгоритма муравья, не являются оптимальными, применение алгоритма для данной проблемы показывает, что с его помощью можно решать аналогичные задачи.

Алгоритм муравья применяется для решения других задач, например, прокладки маршрутов для автомобилей, расчета цветов для графиков и маршрутизации в сетях. Более подробно способы использования алгоритма муравья описываются в книге Марко Дориго «Алгоритмы муравья для абстрактной оптимизации» (Ant Algorithms for Discrete Optimization).

Итоги

В этой главе описывался метод оптимизации поиска путей, позаимствованный у природы. Алгоритм муравья моделирует поведение муравьев в их природной среде, чтобы определить оптимальный путь в пространстве (по графу или сети). Данная технология рассматривалась как средство для решения задачи коммивояжера (TSP). Кроме того, были описаны возможности изменения параметров алгоритма и представлены комбинации параметров, которые демонстрируют хорошие результаты.

Литература и ресурсы

1. Wolfram Research. Гамильтонов путь (Hamiltonian Path). Доступно по адресу http://mathworld.wolfram.com/HamiltonianPath.html.
2. Дориго М. Web-сайт Марко Дориго по оптимизации с помощью колонии муравьев, http://iridia.ulb.ac.be/~mdorigo/ACO/ACO.html.
3. Дориго М. Алгоритмы муравья для абстрактной оптимизации (Dorigo M. Ant Algorithms for Discrete Optimization, 1999). Доступно по адресу http://citeseer.nj.nec.con/420280.html.

4. Дориго М. Колонии муравьев как средство решения задачи коммивояжера (Dorigo M. Ant Colonies for the Travelling Salesman Problem // Biosystems, 43:73–81, 1996).
5. Дориго М. Система муравьев: оптимизация с помощью колонии сотрудничающих агентов (Dorigo M. The Ant System: Optimization by a Colony of Cooperating Agents // IEEE Transactions on Systems, Man and Cybernetics. – Part B26, (1):1–13, 1996).
6. Эпплгейт Д. История проблемы путешествующего коммивояжера (Applegate D. History of the Traveling Salesman Problem, 2003). Доступно по адресу http:/www.math.princeton.edu/tsp/histmain.html.

Глава 5. Введение в архитектуру нейронных сетей и алгоритм обратного распространения

В данной главе вводится понятие многослойных нейронных сетей, обучение в которых осуществляется с помощью алгоритма обратного распространения. Это, пожалуй, самый важный алгоритм обучения в нейронных сетях, внесший значительный вклад в развитие методов расчета, которые имеют естественное происхождение. После детального изучения нейронных сетей и алгоритма обратного распространения рассматривается использование нейронных сетей при разработке ИИ для игр.

Существует множество вариантов нейронных сетей и обучающих алгоритмов, однако в данной главе основное внимание уделяется многослойным сетям, в которых используется алгоритм обратного распространения. Сначала описываются компоненты нейронных сетей, обсуждается алгоритм обучения и ряд проблем, которые могут возникнуть при его применении. Приводится пример простой сети и последовательно разбирается работа алгоритма обратного распространения. Наконец, нейронные сети рассматриваются как средство создания «живых» персонажей в компьютерных играх.

Нейронные сети в биологической перспективе

Нейронные сети (Neural network) представляют собой упрощенную модель человеческого мозга. Мозг состоит из нейронов, которые являются индивидуальными процессорами. Нейроны соединяются друг с другом с помощью нервных окончаний двух типов: синапсов, через которые в ядро поступают сигналы, и аксонов, через которые нейрон передает сигнал далее. Человеческий мозг состоит примерно из 10^{11} нейронов. Каждый нейрон связан примерно с 1000 других нейронов (это не относится к коре головного мозга, где плотность нейронных связей намного выше). Структура мозга высокоциклична, но ее можно рассматривать и как многослойную (рис. 5.1). В очень упрощенном виде работу мозга можно представить так: внешний слой сети передает импульсы от сенсоров из внешней среды, средний слой (или кора головного мозга) обрабатывает импульсы, а «выходной» слой выдает результат (действие) обратно во внешнюю среду.

Искусственные нейронные сети имитируют работу мозга. Информация передается между нейронами, а структура и вес нервных окончаний определяют поведение сети.

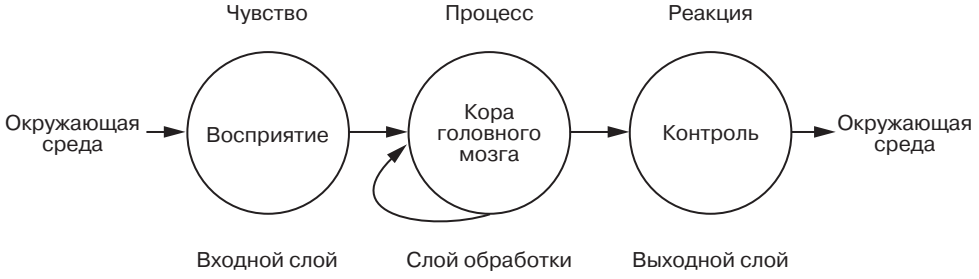


Рис. 5.1. Многослойная архитектура мозга

Однослойные перцептроны

Однослойный перцептрон (Single layer perceptron – SLP) представляет собой концептуальную модель, которая состоит из одного процессора. Каждое соединение от входа к ядру включает коэффициент, который показывает фактор веса, и обозначается с помощью веса w_i, который определяет влияние ячейки u_i на другую ячейку. Положительные веса показывают усиление, а отрицательные – запрещение. Совместно с входами в ячейку они определяют поведение сети. Схема однослойного перцептрона представлена на рис. 5.2.

Ячейка на рис. 5.2 включает три входа (u_1, u_2 и u_3). Кроме этого, есть вход смещения (w_0), о котором будет рассказано позже. Каждое входное соединение имеет вес (w_1, w_2 и w_3). Наконец, существует единый выход, O. Состояние нейрона обозначено как γ и определяется уравнением 5.1.

$$\gamma = \omega_0 + \sum_{i=1}^{3} u_i w_i \qquad (5.1)$$

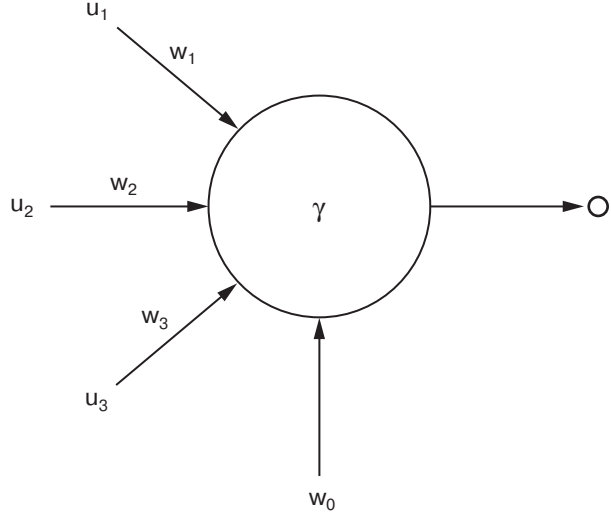


Рис. 5.2. Однослойный перцептрон

Выражение, показанное в уравнении 5.1, является функцией, которая суммирует сигналы на всех входах с учетом веса, а затем добавляет смещение. Затем результат передается в активационную функцию, которая может быть определена так, как показано в уравнении 5.2 (в данном случае функция является пороговой).

$$\gamma = -1, \text{ если } (\gamma \leq 0)$$
$$\gamma = 1, \text{ если } (\gamma > 0) \qquad (5.2)$$

Если значение состояния больше нуля, то выходное значение будет равно 1. Иначе оно составит –1.

Моделирование булевых выражений с помощью SLP

Хотя однослойный перцептрон является очень простой моделью, ее возможности весьма велики. Например, можно легко сконструировать базовые логические функции, как показано на рис. 5.3.

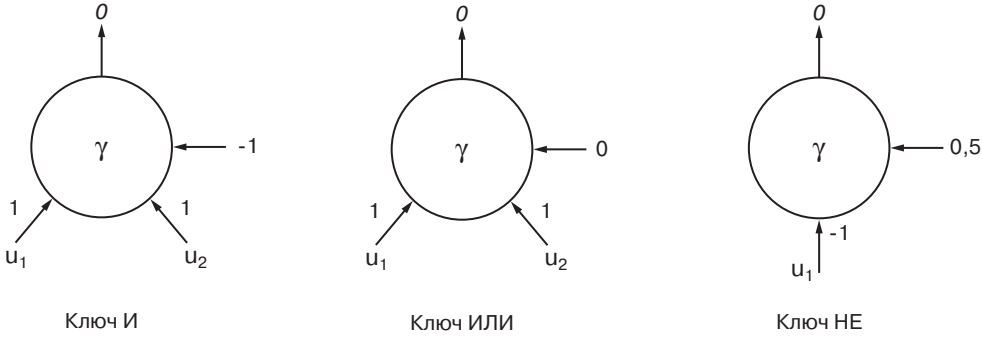


Рис. 5.3. Логические функции, построенные с помощью однослойных перцептронов

Вспомните, что функция И имеет значение 1, если оба входа равны 1, в противном случае функция возвращает значение 0. Поэтому если заданы оба входа (вектор u = (1,1)), то, используя активационную функцию из уравнения 5.2 в качестве порога, получим следующий результат:

γ = смещение + $u_1w_1 + u_2w_2$
или
1 = порог$(-1 + (1 \times 1) + (1 \times 1))$.

Теперь попробуем подставить вектор u = (0,1):

γ = смещение + $u_1w_1 + u_2w_2$
или
1 = порог$(-1 + (0 \times 1) + (1 \times 1))$.

Как показывают оба примера, модель простого перцептрона правильно реализует логическую функцию И (а также функции ИЛИ и НЕ). Однако однослойный

перцептрон не может смоделировать такую логическую функцию, как исключающее ИЛИ (XOR). Эта неспособность к моделированию функции XOR известна как проблема отделимости. Из-за нее Марвин Мински и Сеймур Паперт в 1960-е гг. уничтожили результаты своих разработок в области связей и стали заниматься стандартными подходами к изучению ИИ.

Проблема отделимости была легко решена путем добавления одного или нескольких слоев между входами и выходами нейронной сети (см. рис. 5.4). Это привело к созданию модели, известной как многослойные перцептроны (Multiple layer perceptron – MLP).

Многослойные сети

Многослойные сети позволяют создавать более сложные, нелинейные связи между входными данными и результатами на выходе. На рис. 5.4 многослойная сеть состоит из входного, промежуточного (или скрытого) и выходного слоев. Входной слой представляет входы в сеть и не состоит из ячеек (нейронов) в традиционном смысле слова. В этом примере для каждой ячейки задан идентификатор u_n. Две входные ячейки называются (u_1, u_2), две скрытые ячейки (u_3, u_4), а выходная ячейка – (u_5). Обозначение соединений в сети стандартизовано в форме $w_{1,3}$ и отображает связь с учетом веса между u_3 и u_1.

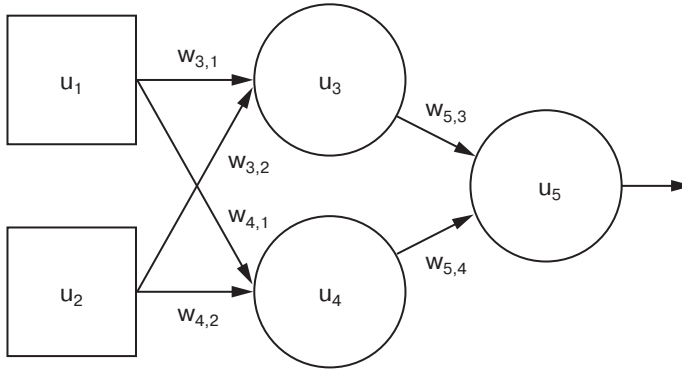


Рис. 5.4. Многослойные перцептроны

В то время как входные ячейки (u_1 и u_2) просто задают входное значение для сети, скрытые и выходные ячейки представляют собой функцию (уравнение 5.1). Результат суммирования дополнительно обрабатывается функцией сжатия (обычно сигмоид), результат которой выдается на выходе из ячейки. Функция сигмоида показана на рис. 5.5.

Теперь изучим полную картину ячейки в нейронной сети. На рис. 5.6 изображена выходная ячейка для сети, показанной на рис. 5.4. Выходная ячейка u_5 получает результат от двух скрытых ячеек (u_3 и u_4) через веса $w_{5,3}$ и $w_{5,4}$ соответственно.

Здесь важно отметить, что функция сигмоида должна быть применена и к скрытым узлам сети на рис. 5.6, но в данном случае они опускаются, чтобы

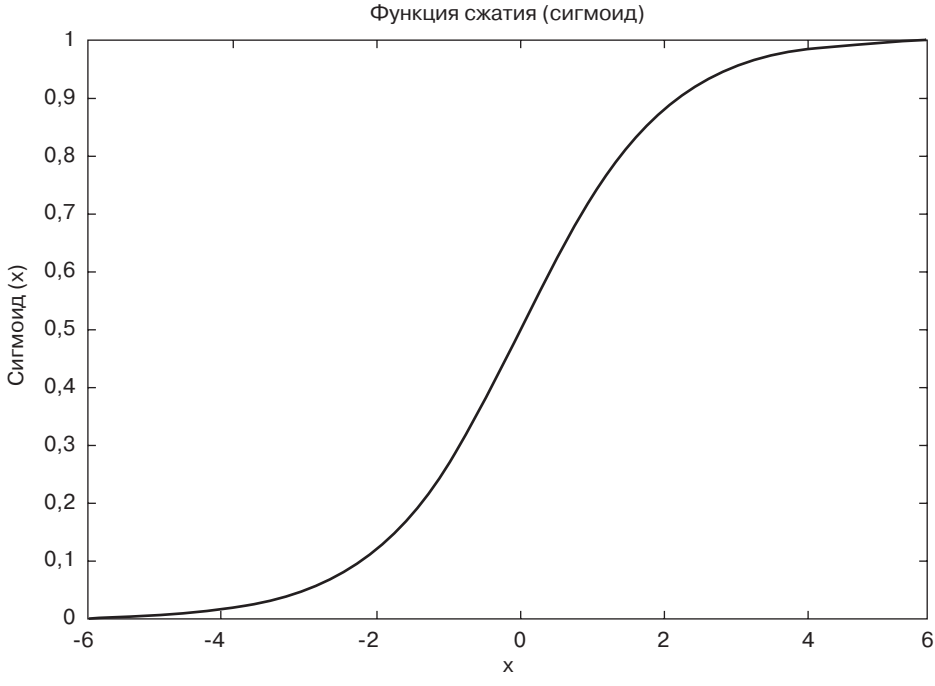


Рис. 5.5. Функция сигмоида, использующаяся для активации

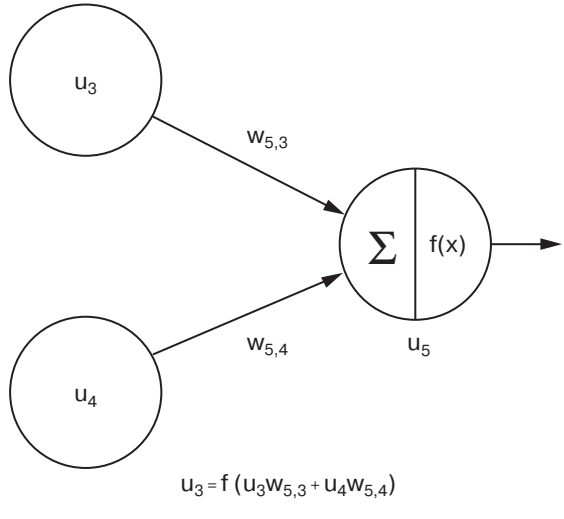


Рис. 5.6. Скрытый и выходной слои в нейронной сети

проиллюстрировать только обработку выходного слоя. Уравнение на рис. 5.6 показывает сумму результатов входов скрытого слоя с весами соединений. Функция f(x) представляет сигмоид, примененный к результату.

В сети со скрытым и выходным слоями сначала выполняется расчет скрытого слоя, а затем его результаты используются для расчета выходного слоя.

Обучение с помощью алгоритма обратного распространения

Обратное распространение (Backpropagation algorithm) – это самый популярный алгоритм для обучения с помощью изменения весов связей. Как ясно из названия, ошибка распространяется от выходного слоя к входному, то есть в направлении, противоположном направлению прохождения сигнала при нормальном функционировании сети. Хотя алгоритм достаточно простой, его расчет может занять довольно много времени в зависимости от величины ошибки.

Алгоритм обратного распространения

Выполнение алгоритма начинается с создания произвольно сгенерированных весов для многослойной сети. Затем процесс, описанный ниже, повторяется до тех пор, пока средняя ошибка на входе не будет признана достаточно малой:

1. Берется пример входного сигнала E с соответствующим правильным значением выхода C.
2. Рассчитывается прямое распространение E через сеть (определяются весовые суммы S_i и активаторы u_i для каждой ячейки).
3. Начиная с выходов, выполняется обратное движение через ячейки выходного и промежуточного слоя, при этом программа рассчитывает значения ошибок (уравнения 5.3 и 5.4):

$$\delta_0 = (C_i - u_5)u_5(1 - u_5) \text{ для выходной ячейки} \quad (5.3)$$

$$\delta_i = (\Sigma_{m:m>i}\, w_{m,j}\delta_0)u_i(1 - u_i) \text{ для всех скрытых ячеек} \quad (5.4)$$

   (Обратите внимание, что m обозначает все ячейки, связанные со скрытым узлом, w – заданный вектор веса, а u – активация).
4. Наконец, веса в сети обновляются следующим образом (уравнение 5.5 и 5.6):

$$w^*_{i,j} = w_{i,j} + \rho\delta_0 u_i$$
для весов соединений между скрытым слоем и выходом (5.5)

$$w^*_{i,j} = w_{i,j} + \rho\delta_i u_i$$
для весов соединений между скрытым слоем и входом (5.6)

Здесь ρ представляет коэффициент обучения (или размер шага). Это небольшое значение ограничивает изменение, которое может произойти при каждом шаге.

Совет *Параметр ρ можно определить таким образом, чтобы он указывал скорость продвижения алгоритма обратного распространения к решению. Лучше начать тестирование с небольшого значения (0,1) и затем постепенно его повышать.*

Продвижение вперед по сети рассчитывает активации ячеек и выход, продвижение назад – градиент (по отношению к данному примеру). Затем веса обновляются таким образом, чтобы минимизировать ошибку для данного входного сигнала. Коэффициент обучения минимизирует процент изменения, которое может произойти с весами. Хотя при небольшом коэффициенте процесс может занять больше времени, мы минимизируем возможность пропуска правильной комбинации весов. Если коэффициент обучения слишком велик, сеть может никогда не сойтись, то есть не будут найдены правильные веса связей.

Рассмотрим пример функционирования сети в процессе обучения.

Пример алгоритма обратного распространения

Изучим работу алгоритма обратного распространения, взяв в качестве примера сеть, показанную на рис. 5.7.

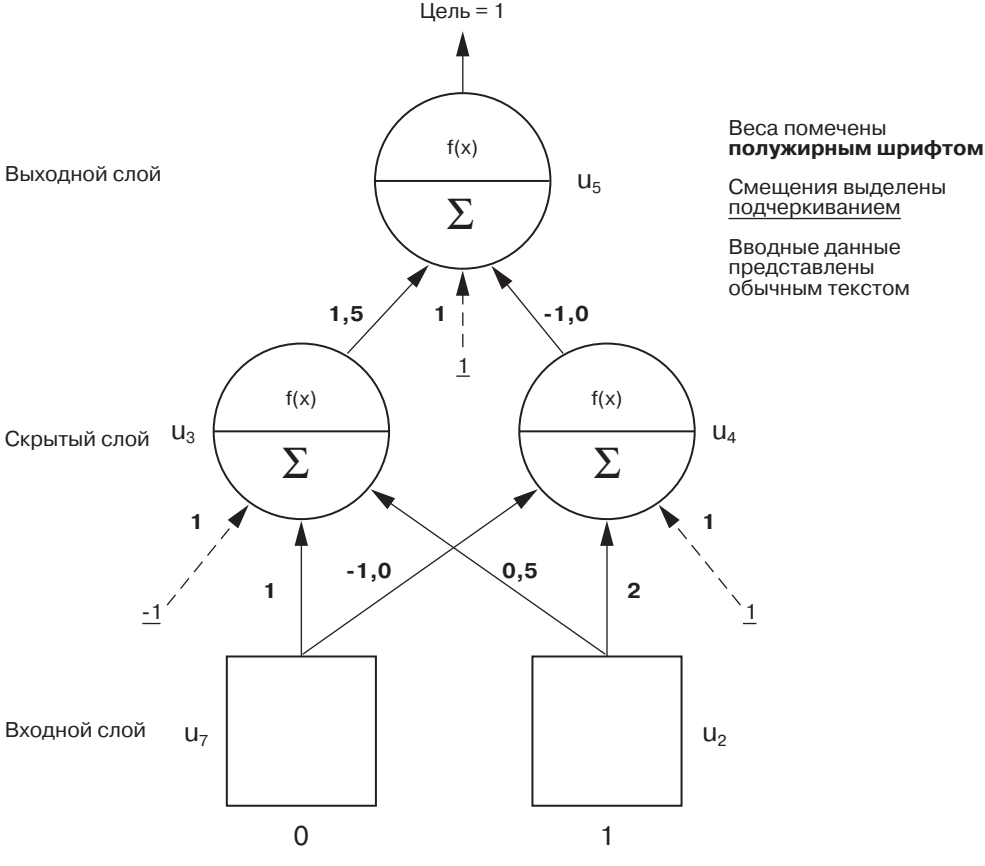


Рис. 5.7. Пример алгоритма обратного распространения

Развитие вперед

Сначала выполняется расчет движения входного сигнала по сети. Рассмотрим значения для скрытого слоя:

$u_3 = f(w_{3,1}u_1 + w_{3,2}u_2 + w_b \times \text{смещение})$
$u_3 = f(1 \times 0 + 0{,}5 \times 1 + 1 \times 1) = f(1{,}5)$
$u_3 = 0{,}81757;$

$u_4 = f(w_{4,1}u_1 + w_{4,2}u_2 + w_b \times \text{смещение})$
$u_4 = f(-1 \times 0 + 2 \times 1 + 1 \times 1) = f(3)$
$u_4 = 0{,}952574.$

Вспомните, что f(x) является активационной функцией, то есть функцией сигмоида (уравнение 5.7):

$$f(x) = 1/(1+e^{-x}) \tag{5.7}$$

Теперь сигнал дошел до скрытого слоя. Конечный шаг заключается в том, чтобы переместить сигнал из скрытого слоя в выходной слой и рассчитать значения на выходе из сети:

$u_5 = f(w_{5,3}u_3 + w_{5,4}u_4 + w_b \times \text{смещение})$
$u_5 = f(1{,}5 \times 0{,}81757 + -1{,}0 \times 0{,}952574 + 1 \times 1) = f(1{,}2195)$
$u_5 = 0{,}78139.$

Правильной реакцией нейронной сети на тестовый входной сигнал является 1,0; значение, рассчитанное сетью, составляет 0,78139. Это не так уж и плохо, однако можно уменьшить значение ошибки, применив для сети алгоритм обратного распространения.

Для коррекции весовых коэффициентов в сети обычно используется среднеквадратичная ошибка. Для одного узла ошибка определяется в уравнении 5.8:

$$err = 0{,}5 \times (0_{\text{требуемое}} - 0_{\text{реальное}})^2 \tag{5.8}$$

Поэтому ошибка составляет:

$err = 0{,}5 \times (1{,}0 - 0{,}78139)^2 = 0{,}023895.$

Алгоритм обратного распространения для ошибки

Теперь применим обратное распространение, начиная с определения ошибки в выходном и скрытых узлах. Используя уравнение 5.1, рассчитаем ошибку в выходном узле:

$\delta_0 = (1{,}0 - 0{,}78139) \times 0{,}78139 \times (1{,}0 - 0{,}78139)$
$\delta_0 = 0{,}0373.$

Теперь следует рассчитать ошибку для двух скрытых узлов. Для этого используется производная функция сигмоида (уравнение 5.5), которая показана в виде уравнения 5.9:

$$val = x \times (1{,}0 - x) \tag{5.9}$$

Используя уравнение 5.2, рассчитаем ошибки для скрытых узлов:

$$\delta_{u4} = (\delta_0 \times w_{5,4}) \times u_4 \times (1,0 - u_4)$$
$$\delta_{u4} = (0,0373 \times -1,0) \times 0,952574 \times (1,0 - 0,952574)$$
$$\delta_{u4} = -0,0016851;$$
$$\delta_{u3} = (\delta_0 \times w_{5,3}) \times u_3 \times (1,0 - u_3)$$
$$\delta_{u3} = (0,0373 \times 1,5) \times 0,81757 \times (1,0 - 0,81757)$$
$$\delta_{u3} = 0,0083449.$$

Изменения весов соединений

Теперь, когда рассчитаны значения ошибок для выходного и скрытого слоев, можно с помощью уравнений 5.3 и 5.4 изменить веса. Используем коэффициент обучения (ρ), равный 0,5. Сначала следует обновить веса для соединений между выходным и скрытым слоями:

$$w^*_{i,j} = w_{i,j} + \rho\delta_0 u_i$$
$$w_{5,4} = w_{5,4} + (\rho \times 0,0373 \times u_4)$$
$$w_{5,4} = -1 + (0,5 \times 0,0373 \times 0,952574)$$
$$w_{5,4} = -0,9882;$$
$$w_{5,3} = w_{5,3} + (\rho \times 0,0373 \times u_3)$$
$$w_{5,3} = 1,5 + (0,5 \times 0,0373 \times 0,81757)$$
$$w_{5,3} = 1,51525.$$

Теперь нужно обновить смещение для выходной ячейки:

$$w_{5,b} = w_{5,b} + (\rho \times 0,0373 \times \text{смещение}_5)$$
$$w_{5,b} = 1 + (0,5 \times 0,0373 \times 1)$$
$$w_{5,b} = 1,01865.$$

Для $w_{5,4}$ вес уменьшен, а для $w_{5,3}$ – увеличен. Смещение было обновлено для повышения возбуждения. Теперь нужно показать изменение весов для скрытого слоя (для входа к скрытым ячейкам):

$$w^*_{i,j} = w_{i,j} + \rho\delta_0 u_i$$
$$w_{4,2} = w_{4,2} + (\rho \times -0,0016851 \times u_2)$$
$$w_{4,2} = 2 + (0,5 \times -0,0016851 \times 1)$$
$$w_{4,2} = 1,99916;$$
$$w_{4,1} = w_{4,1} + (\rho \times -0,0016851 \times u_1)$$
$$w_{4,1} = -1 + (0,5 \times -0,0016851 \times 0)$$
$$w_{4,1} = -1,0;$$
$$w_{3,2} = w_{3,2} + (\rho \times 0,0083449 \times u_2)$$
$$w_{3,2} = 0,5 + (0,5 \times 0,0083449 \times 1)$$
$$w_{3,2} = 0,50417;$$
$$w_{3,1} = w_{3,1} + (\rho \times 0,0083449 \times u_1)$$
$$w_{3,1} = 1,0 + (0,5 \times 0,0083449 \times 1)$$
$$w_{3,1} = 1,0.$$

Последний шаг – это обновление смещений для ячеек:

$w_{4,b} = w_{4,b} + (\rho \times -0{,}0016851 \times \text{смещение}_4)$
$w_{4,b} = 1{,}0 + (0{,}5 \times -0{,}0016851 \times 1)$
$w_{4,b} = 0{,}99915;$
$w_{3,b} = w_{3,b} + (\rho \times 0{,}0083449 \times \text{смещение}_3)$
$w_{3,b} = 1{,}0 + (0{,}5 \times 0{,}0083449 \times 1)$
$w_{3,b} = 1{,}00417.$

Обновление весов для выбранного обучающего сигнала завершается. Проверим, что алгоритм действительно уменьшает ошибку на выходе, введя тот же обучающий сигнал еще раз:

$u_3 = f(w_{3,1}u_1 + w_{3,2}u_2 + w_b \times \text{смещение})$
$u_3 = f(1 \times 0 + 0{,}50417 \times 1 + 1{,}00417 \times 1) = f(1{,}50834)$
$u_3 = 0{,}8188;$
$u_4 = f(w_{4,1}u_1 + w_{4,2}u_2 + w_b \times \text{смещение})$
$u_4 = f(-1 \times 0 + 1{,}99916 \times 1 + 0{,}99915 \times 1) = f(2{,}99831)$
$u_4 = 0{,}952497;$
$u_5 = f(w_{5,3}u_3 + w_{5,4}u_4 + w_b \times \text{смещение})$
$u_5 = f(1{,}51525 \times 0{,}81888 + -0{,}9822 \times 0{,}952497 + 1{,}01865 \times 1) = f(1{,}32379)$
$u_5 = 0{,}7898;$
$\text{err} = 0{,}5 \times (1{,}0 - 0{,}7898)^2 = 0{,}022.$

Вспомните, что начальная ошибка была равна 0,023895. Текущая ошибка составляет 0,022, а это значит, что одна итерация алгоритма обратного распространения позволила уменьшить среднюю ошибку на 0,001895.

Расчет поведения ИИ для компьютерных игр

Алгоритм обратного распространения применяется при создании нейроконтроллеров для персонажей компьютерных игр. *Нейроконтроллерами* (Neurocontroller) обычно называются нейронные сети, которые используются при управлении. В этом приложении мы задействуем нейронную сеть, чтобы выбрать действие из доступного списка на основании того, в какой окружающей среде находится персонаж. Термины «персонаж» и «агент» далее употребляются как синонимы.

Причина использования нейронных сетей в качестве нейроконтроллеров заключается в том, что невозможно задать для ИИ персонажа игры поведение, которое охватывало бы все возможные в окружающей среде ситуации. Поэтому необходимо обучить нейронную сеть на ограниченном количестве примеров (то есть образцов поведения в зависимости от обстановки), а затем позволить ей самостоятельно генерировать поведение во всех прочих ситуациях. Способность генерировать правильную реакцию на различные ситуации, не входящие в набор обучающих, является ключевым фактором при создании нейроконтроллера.

Другим преимуществом нейроконтроллера является то, что он не является строго заданной функцией, которая обеспечивает взаимодействие между окружающей

средой и реакцией на нее. Незначительные изменения в окружающей среде могут вызвать различную реакцию у нейроконтроллера, отчего поведение персонажа выглядит более естественным. Фиксированные деревья поведения или конечные автоматы вызывают предсказуемую реакцию, что довольно плохо отражается на игре.

Как показано на рис. 5.8, окружающая среда предоставляет для персонажа определенную информацию, которая затем передается агенту. Процесс «восприятия» окружающей среды называется предчувствием. Нейроконтроллер обеспечивает возможность выбора действия, посредством которого персонаж взаимодействует с окружающей средой. Окружающая среда при этом изменяется, персонаж вновь обращается к восприятию, и цикл возобновляется.

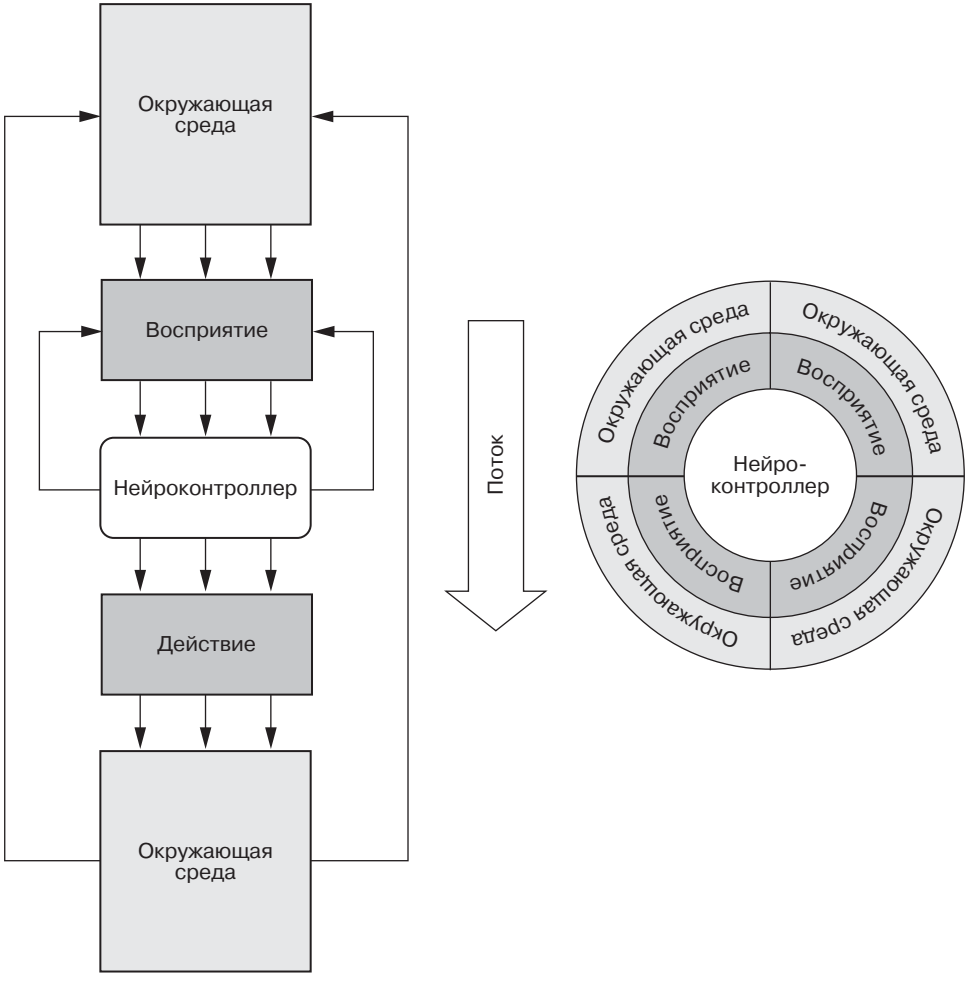


Рис. 5.8. Пример нейроконтроллера в окружающей среде

Архитектура нейроконтроллера

В предыдущем примере описывалась нейронная сеть с одним выходом. В компьютерных играх используется несколько иная архитектура – сеть, построенная по принципу «победитель получает все». Такие архитектуры полезны в том случае, если выходы должны быть разделены на несколько классов (рис. 5.9).

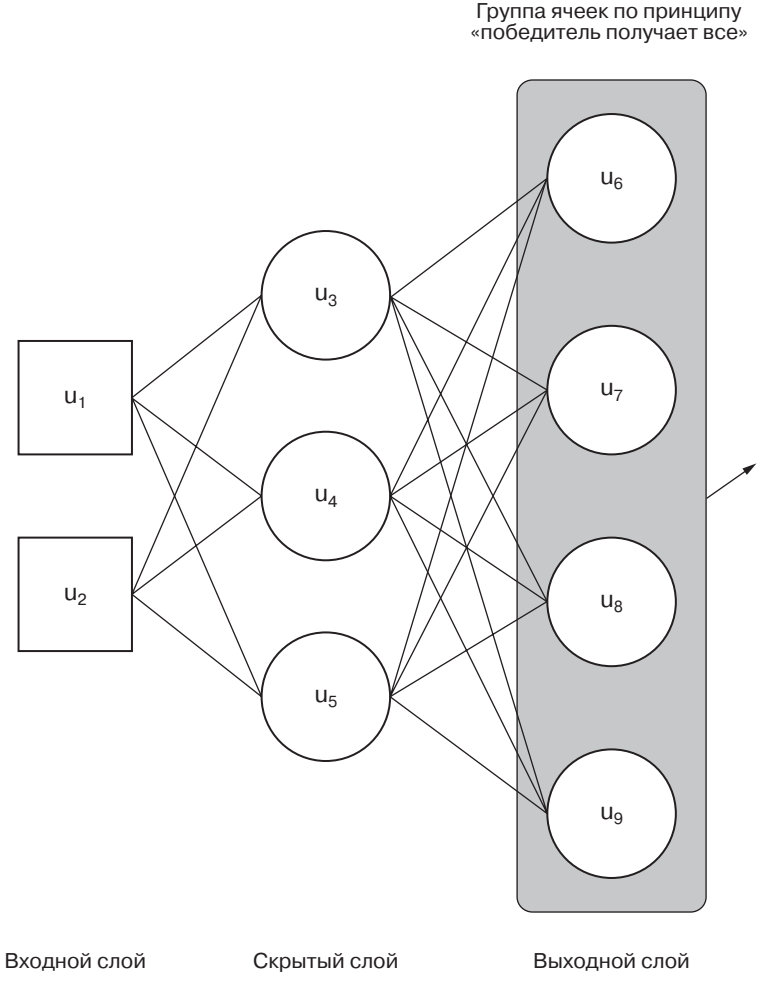


Рис. 5.9. Группа «победитель получает все»

В сети, созданной по принципу «победитель получает все», выходная ячейка с большей суммой весов является «победителем» группы и допускается к действию. В рассматриваемом приложении каждая ячейка представляет определенное поведение, которое доступно для персонажа в игре. В качестве примеров поведения

можно назвать такие действия, как выстрелить из оружия, убежать, уклониться и др. Срабатывание ячейки в группе по принципу «победитель получает все» приводит к тому, что агент выполняет определенное действие. Когда агенту вновь позволяется оценить окружающую среду, процесс повторяется.

На рис. 5.10 представлена сеть, которая использовалась для тестирования архитектуры и метода выбора действия. Четыре входа обозначают «здоровье персонажа» (0 – плохое, 2 – хорошее), «имеет нож» (1, если персонаж имеет нож, 0 – в противном случае), «имеет пистолет» (1, если персонаж имеет пистолет, 0 – в противном случае) и «присутствует враг» (количество врагов в поле зрения).

Выходы определяют поведение, которое выберет персонаж. Действие «атаковать» приводит к тому, что персонаж атакует врагов в поле зрения, «бежать»

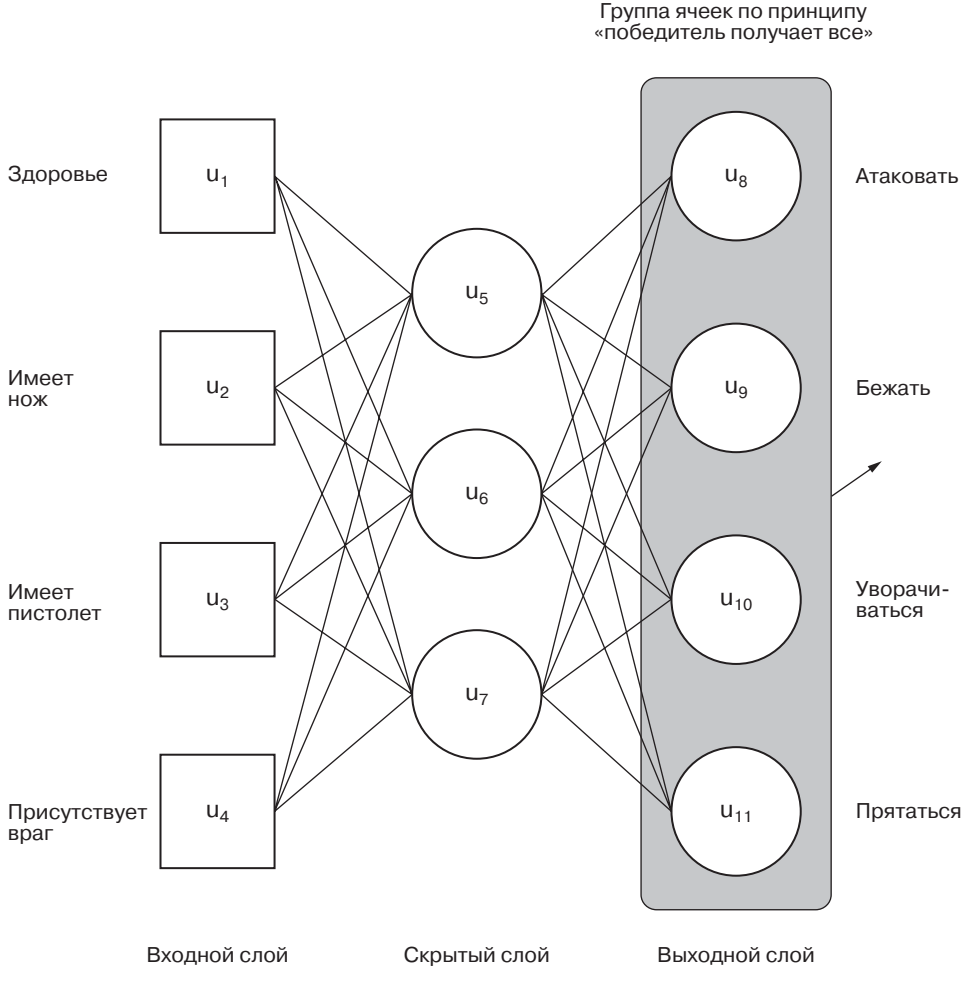


Рис. 5.10. Архитектура нейроконтроллера для компьютерных игр

вынуждает персонаж убегать, «уворачиваться» приводит к произвольному движению персонажа, а «прятаться» вынуждает персонаж искать укрытие. Это высокоуровневые образы поведения, и предполагается, что подсистема поведения будет выбирать действие и следовать ему.

Примечание *Выбранная здесь архитектура (три скрытые ячейки) была определена способом проб и ошибок. Три скрытые ячейки могут быть обучены для всех представленных примеров со 100% точностью. Уменьшение количества ячеек до двух или одной приводит к созданию сети, которая не может правильно классифицировать все примеры.*

Обучение нейроконтроллера

Нейроконтроллер в игровой среде представляет собой постоянный элемент персонажа. Дальше мы обсудим обучение нейроконтроллера в режиме реального времени.

Обучение нейроконтроллера состоит в предоставлении обучающих примеров из небольшой группы желательных действий. Затем следует выполнение алгоритма обратного распространения с учетом желаемого результата и действительного результата. Например, если персонаж имеет пистолет, здоров и видит одного врага, желаемое действие – атаковать. Однако если персонаж здоров, имеет нож, но видит двух врагов, то правильное действие – спрятаться.

Данные для тестирования

Данные для тестирования представляют собой несколько сценариев с набором действий. Поскольку требуется, чтобы нейроконтроллер вел себя так же, как настоящий человек, мы не будем обучать его для каждого случая. Сеть должна рассчитывать реакцию на входы и выполнять действие, которое будет похожим на обучающие сценарии. Примеры, которые использовались для обучения сети, представлены в табл. 5.1.

Таблица 5.1. Примеры, которые используются для обучения нейроконтроллера

Здоровье	Имеет нож	Имеет пистолет	Враги	Поведение
2	0	0	0	Уворачиваться
2	0	0	1	Уворачиваться
2	0	1	1	Атаковать
2	0	1	2	Атаковать
2	1	0	2	Прятаться
2	1	0	1	Атаковать
1	0	0	0	Уворачиваться
1	0	0	1	Прятаться
1	0	1	1	Атаковать

Таблица 5.1. Примеры, которые используются для обучения нейроконтроллера (окончание)

Здоровье	Имеет нож	Имеет пистолет	Враги	Поведение
1	0	1	2	Прятаться
1	1	0	2	Прятаться
1	1	0	1	Прятаться
0	0	0	0	Уворачиваться
0	0	0	1	Прятаться
0	0	1	1	Прятаться
0	0	1	2	Бежать
0	1	0	2	Бежать
0	1	0	1	Прятаться

Данные, приведенные в табл. 5.1, были переданы сети в произвольном порядке во время обучения с помощью алгоритма обратного распространения. График снижения средней ошибки показан на рис. 5.11.

В большинстве случаев сеть успешно проходит обучение на всех представленных примерах. При некоторых запусках один или два примера приводят

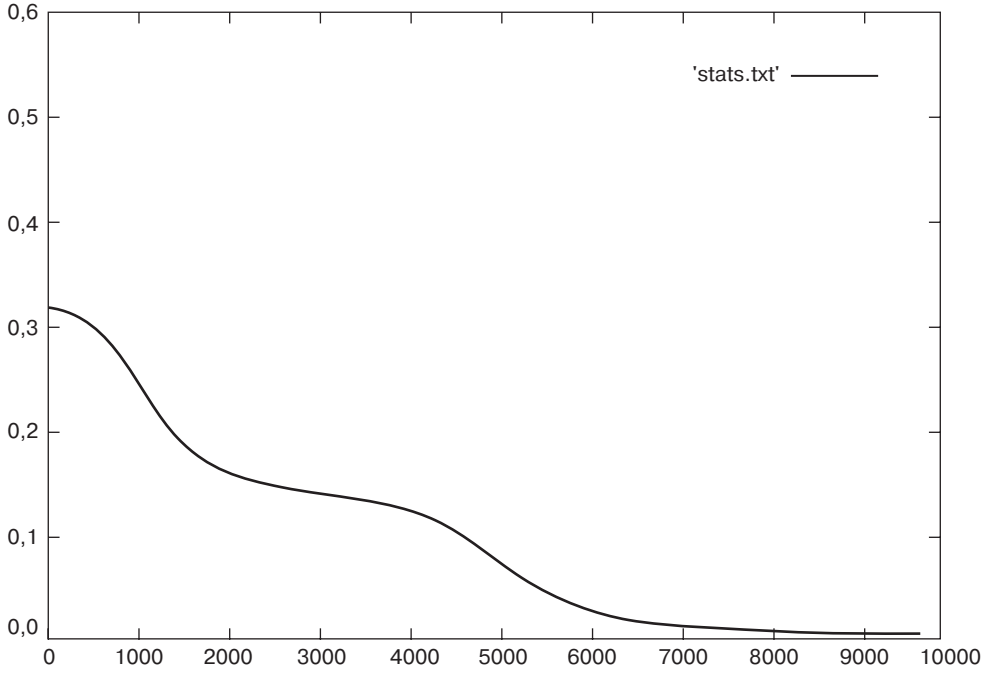


Рис. 5.11. Пример выполнения алгоритма обратного распространения для нейроконтроллера

к неправильным действиям. При увеличении количества скрытых ячеек обучение проходит идеально. Однако в этом случае для работы нейроконтроллера требуется намного больше компьютерных ресурсов. Поэтому были выбраны три скрытые ячейки с несколькими циклами обучения. Для обучения нейроконтроллера было выполнено столько запусков, сколько потребовалось для получения идеального результата на основании тестовых данных.

Чтобы протестировать нейроконтроллер, сети были представлены новые примеры. Это позволило определить, как сеть будет реагировать на сценарии, о которых ей ничего не известно. Данные тесты дают ответ на вопрос, насколько хорошо нейроконтроллер может генерировать и выполнять нужные действия, реагируя на непредвиденную ситуацию.

Если предложить нейроконтроллеру сценарий, в котором персонаж полностью здоров, владеет оружием двух видов и видит двух врагов (то есть 2:1:1:1), нейроконтроллер выберет действие «атаковать». Это разумная реакция на данную ситуацию. Теперь рассмотрим сценарий, в котором персонаж полностью здоров, владеет ножом и видит трех врагов (то есть 2:1:0:3). Нейроконтроллер выбирает действие «спрятаться», вполне разумный выбор в данной ситуации. Другие примеры показаны в табл. 5.2.

Таблица 5.2. Примеры, иллюстрирующие правильную генерацию действий

Здоровье	Имеет нож	Имеет пистолет	Враги	Поведение
Хорошее (2)	Да	Да	1	Атаковать
OK (1)	Да	Да	2	Прятаться
Плохое (0)	Нет	Нет	0	Уворачиваться
Плохое (0)	Да	Да	1	Прятаться
Хорошее (2)	Нет	Да	3	Прятаться
Хорошее (2)	Да	Нет	3	Прятаться
Плохое (0)	Да	Нет	3	Бежать

Таким образом, нейроконтроллер правильно генерирует действие из заданного набора в ответ на новую обстановку (табл. 5.2). Хотя его не обучали конкретно для этих примеров, он способен правильно на них реагировать.

Совет *Исходный код алгоритма обратного распространения вы можете найти в архиве с примерами на сайте издательства «ДМК Пресс» www.dmk.ru.*

Обсуждение исходного кода

Рассмотрим исходный код реализации алгоритма обратного распространения для конфигурируемой сети, а также код, который выполняет обучение и тестирование нейроконтроллера. Глобальные константы и переменные показаны в листинге 5.1.

Листинг 5.1. Глобальные константы и переменные для нейронной сети и алгоритма обратного распространения

```
#define INPUT_NEURONS       4
#define HIDDEN_NEURONS      3
#define OUTPUT_NEURONS      4

/* Веса */
/* Вход скрытых ячеек (со смещением) */
double wih[INPUT_NEURONS+1][HIDDEN_NEURONS];

/* Вход выходных ячеек (со смещением) */
double who[HIDDEN_NEURONS+1][OUTPUT_NEURONS];

/* Активаторы */
double inputs[INPUT_NEURONS];
double hidden[HIDDEN_NEURONS];
double target[OUTPUT_NEURONS];
double actual[OUTPUT_NEURONS];

/* Ошибки */
double erro[OUTPUT_NEURONS];
double errh[HIDDEN_NEURONS];
```

Веса определяются как веса соединений между входным и скрытым (`wih`), а также между скрытым и выходным слоями (`who`). Вес соединения между u_5 и u_1 (рис. 5.10) является весом входа в скрытый слой, представленный `wih[0][0]` (так как u_1 – это первая входная ячейка, а u_5 – первая скрытая ячейка, начиная с нуля). Данный вес обозначается как $w_{5,1}$. Вес $w_{11,7}$ (соединение между ячейкой u_{11} выходного слоя и u_7 скрытого слоя) равен `who[2][3]`. Веса смещения занимают последнюю строку в каждой таблице и идентифицируются с помощью значения +1 в массивах `wih` и `who`.

Значения сигналов хранятся в четырех массивах. Массив `inputs` определяет значение входных ячеек, массив `hidden` содержит выход для скрытых ячеек, массив `target` предоставляет желаемое значение сети для заданных входов, а массив `actual` отображает реальный результат работы сети.

Ошибки сети предоставляются в двух массивах. Массив `erro` хранит ошибку для каждой входной ячейки. Массив `errh` содержит ошибки скрытых ячеек.

Чтобы найти произвольные веса для инициализации сети, создается группа макросов, показанная в листинге 5.2.

Листинг 5.2. Макросы и символьные константы для алгоритма обратного распространения

```
#define LEARN_RATE    0.2 /* Коэффициент обучения */
#define RAND_WEIGHT   ( ((float)rand() / (float)RAND_MAX) - 0.5)

#define getSRand() ((float)rand() / (float)RAND_MAX)
```

```
#define getRand(x) (int)((x) * getSRand())

#define sqr(x)          ((x) * (x))
```

Веса произвольно выбираются в диапазоне (от –0,5 до 0,5). Коэффициент обучения задается как 0,2. Диапазон весов и коэффициент обучения могут быть изменены в зависимости от проблемы и требуемой точности решения.

Существуют три вспомогательные функции, которые используются, чтобы задавать произвольные веса для сети, а также для работы алгоритма. Они приведены в листинге 5.3.

Листинг 5.3. Вспомогательные функции для алгоритма обратного распространения

```
void assignRandomWeights( void )
{
  int hid, inp, out;

  for (inp = 0 ; inp < INPUT_NEURONS+1 ; inp++) {
    for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {
      wih[inp][hid] = RAND_WEIGHT;
    }

  }
  for (hid = 0 ; hid < HIDDEN_NEURONS+1 ; hid++) {
    for (out = 0 ; out < OUTPUT_NEURONS ; out++) {
      who[hid][out] = RAND_WEIGHT;
    }
  }
}

double sigmoid( double val )
{
  return (1.0 / (1.0 + exp(-val)));
}

double sigmoidDerivative( double val )
{
  return ( val * (1.0 - val) );
}
```

Функция `assignRandomWeights` произвольно задает вес для всех соединений сети (включая все смещения). Функция `sigmoid` определяет значение функции сжатия (сигмоида), которая используется при прямом вычислении (уравнение 5.5). Функция `sigmoidDerivative` устанавливает значение производной функции `sigmoid` и используется при обратном распространении ошибки.

Следующая функция реализует фазу прямого вычисления алгоритма (листинг 5.4).

Листинг 5.4. Алгоритм прямого распространения

```
void feedForward( )
{
  int inp, hid, out;
  double sum;

  /* Вычислить вход в скрытый слой */
  for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {

    sum = 0.0;
    for (inp = 0 ; inp < INPUT_NEURONS ; inp++) {
      sum += inputs[inp] * wih[inp][hid];
    }

    /* Добавить смещение */
    sum += wih[INPUT_NEURONS][hid];

    hidden[hid] = sigmoid( sum );

  }

  /* Вычислить вход в выходной слой */
  for (out = 0 ; out < OUTPUT_NEURONS ; out++) {

        sum = 0.0;
        for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {
           sum += hidden[hid] * who[hid][out];
        }

    /* Добавить смещение */
    sum += who[HIDDEN_NEURONS][out];

    actual[out] = sigmoid( sum );

  }

}
```

Как показано в листинге 5.4, алгоритм прямого распространения начинает свою работу с расчета активации для скрытых слоев с учетом входов из входного слоя. Смещение добавляется в ячейку до вычисления функции сигмоида. Затем аналогичным образом рассчитывается выходной слой. Обратите внимание, что сеть может иметь одну или несколько выходных ячеек. Поэтому обработка выходных ячеек помещена в цикл, чтобы рассчитать все необходимые активации на выходе.

Алгоритм обратного распространения показан в листинге 5.5.

Листинг 5.5. Алгоритм обратного распространения

```
void backPropagate( void )
{
  int inp, hid, out;

  /* Вычислить ошибку выходного слоя (шаг 3 для выходных ячеек) */
  for (out = 0 ; out < OUTPUT_NEURONS ; out++) {
    erro[out] = (target[out]-actual[out])*
                sigmoidDerivative( actual[out]);
  }
  /* Вычислить ошибку скрытого слоя (шаг 3 для скрытого слоя) */
  for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {

    errh[hid] = 0.0;
    for (out = 0 ; out < OUTPUT_NEURONS ; out++) {
      errh[hid] += erro[out] * who[hid][out];
    }

    errh[hid] *= sigmoidDerivative( hidden[hid] );

  }

  /* Обновить веса для выходного слоя (шаг 4 для выходных ячеек) */
  for (out = 0 ; out < OUTPUT_NEURONS ; out++) {
    for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {
      who[hid][out] += (LEARN_RATE * erro[out] * hidden[hid]);
    }

    /* Обновить смещение */
    who[HIDDEN_NEURONS][out] += (LEARN_RATE * erro[out]);
  }

  /* Обновить веса для скрытого слоя (шаг 4 для скрытого слоя) */

  for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {
    for (inp = 0 ; inp < INPUT_NEURONS ; inp++) {
      wih[inp][hid] += (LEARN_RATE * errh[hid] * inputs[inp]);
    }

    /* Обновить смещение */
    wih[INPUT_NEURONS][hid] += (LEARN_RATE * errh[hid]);

  }

}
```

Данная функция следует алгоритму, описанному в разделе «Пример алгоритма обратного распространения». Сначала рассчитывается ошибка выходной ячейки (или ячеек) с использованием действительного и желаемого результатов. Далее определяются ошибки для скрытых ячеек. Наконец, обновляются веса для всех соединений в зависимости от того, находятся они во входном или выходном скрытом слое. Здесь важно отметить, что в алгоритме не рассчитывается вес для смещения, а просто применяется ошибка к самому смещению. В алгоритме прямого распространения смещение добавляется без использования для него веса.

Данный алгоритм позволяет рассчитать выход многослойной нейронной сети и ее изменения с помощью алгоритма обратного распространения. Рассмотрим исходный код примера, в котором этот алгоритм используется для реализации нейроконтроллера.

В листинге 5.6 приведена структура, задачей которой является отображение примеров для обучения. Структура задает входы (здоровье, нож, пистолет, враг), а также значение желаемого результата (массив `out`). Листинг 5.6 также включает инициализированные данные для обучения сети.

Листинг 5.6. Отображение данных для обучения нейроконтроллера

```
typedef struct {
  double health;
  double knife;
  double gun;
  double enemy;
  double out[OUTPUT_NEURONS];
} ELEMENT;

#define MAX_SAMPLES 18

/*  H    K    G    E     A    R    W    H  */
ELEMENT samples[MAX_SAMPLES] = {
  { 2.0, 0.0, 0.0, 0.0, {0.0, 0.0, 1.0, 0.0} },
  { 2.0, 0.0, 0.0, 1.0, {0.0, 0.0, 1.0, 0.0} },
  { 2.0, 0.0, 1.0, 1.0, {1.0, 0.0, 0.0, 0.0} },
  { 2.0, 0.0, 1.0, 2.0, {1.0, 0.0, 0.0, 0.0} },
  { 2.0, 1.0, 0.0, 2.0, {0.0, 0.0, 0.0, 1.0} },
  { 2.0, 1.0, 0.0, 1.0, {1.0, 0.0, 0.0, 0.0} },

  { 1.0, 0.0, 0.0, 0.0, {0.0, 0.0, 1.0, 0.0} },
  { 1.0, 0.0, 0.0, 1.0, {0.0, 0.0, 0.0, 1.0} },
  { 1.0, 0.0, 1.0, 1.0, {1.0, 0.0, 0.0, 0.0} },
  { 1.0, 0.0, 1.0, 2.0, {0.0, 0.0, 0.0, 1.0} },
  { 1.0, 1.0, 0.0, 2.0, {0.0, 0.0, 0.0, 1.0} },
  { 1.0, 1.0, 0.0, 1.0, {0.0, 0.0, 0.0, 1.0} },

  { 0.0, 0.0, 0.0, 0.0, {0.0, 0.0, 1.0, 0.0} },
  { 0.0, 0.0, 0.0, 1.0, {0.0, 0.0, 0.0, 1.0} },
```

```
  { 0.0, 0.0, 1.0, 1.0, {0.0, 0.0, 0.0, 1.0} },
  { 0.0, 0.0, 1.0, 2.0, {0.0, 1.0, 0.0, 0.0} },
  { 0.0, 1.0, 0.0, 2.0, {0.0, 1.0, 0.0, 0.0} },
  { 0.0, 1.0, 0.0, 1.0, {0.0, 0.0, 0.0, 1.0} }
};
```

Вспомните, что вход «здоровье» может иметь три значения (2 – здоров, 1 – не полностью здоров и 0 – нездоров). Входы «нож» и «пистолет» являются булевыми (1 – если предмет есть, 0 – если его нет), а вход «враг» показывает количество врагов в пределах видимости. Действия также являются булевыми, где ненулевое значение показывает выбранное действие.

Поскольку сеть построена по принципу «победитель получает все», следует закодировать простую функцию, которая будет определять выходную ячейку с самой большой суммой весов. Она выполняет поиск по вектору максимального значения и возвращает строку, которая отображает нужное действие (листинг 5.7). Возвращенное значение затем может использоваться в качестве индекса в массиве строк `strings` и позволяет вывести текст, показывающий реакцию.

Листинг 5.7. Функция для сети «победитель получает все»

```
char *strings[4]={"Attack", "Run", "Wander", "Hide"};

int action( double *vector )
{
  int index, sel;
  double max;

  sel = 0;
  max = vector[sel];

  for (index = 1 ; index < OUTPUT_NEURONS ; index++) {
    if (vector[index] > max) {
      max = vector[index]; sel = index;
    }
  }

  return( sel );
}
```

Наконец, в листинге 5.8 показана функция `main`, которая выполняет обучение и тестирование нейроконтроллера.

Листинг 5.8. Пример функции `main`, которая используется для обучения и тестирования нейроконтроллера

```
int main()
{
  double err;
```

```
  int i, sample=0, iterations=0;
  int sum = 0;

  out = fopen("stats.txt", "w");

  /* Инициализировать генератор случайных чисел */
  srand( time(NULL) );
  assignRandomWeights();

/* Обучить сеть */
while (1) {

    if (++sample == MAX_SAMPLES) sample = 0;

    inputs[0] = samples[sample].health;
    inputs[1] = samples[sample].knife;
    inputs[2] = samples[sample].gun;
    inputs[3] = samples[sample].enemy;

    target[0] = samples[sample].out[0];
    target[1] = samples[sample].out[1];
    target[2] = samples[sample].out[2];
    target[3] = samples[sample].out[3];

    feedForward();

    err = 0.0;
    for (i = 0 ; i < OUTPUT_NEURONS ; i++) {
      err += sqr( (samples[sample].out[i] - actual[i]) );
    }
    err = 0.5 * err;

    fprintf(out, "%g\n", err);
    printf("mse = %g\n", err);

    if (iterations++ > 100000) break;

    backPropagate();

  }

  /* Проверить сеть */
  for (i = 0 ; i < MAX_SAMPLES ; i++) {

    inputs[0] = samples[i].health;
    inputs[1] = samples[i].knife;
```

```
    inputs[2] = samples[i].gun;
    inputs[3] = samples[i].enemy;
    target[0] = samples[i].out[0];
    target[1] = samples[i].out[1];
    target[2] = samples[i].out[2];
    target[3] = samples[i].out[3];

    feedForward();

    if (action(actual) != action(target)) {

      printf("%2.1g:%2.1g:%2.1g:%2.1g %s (%s)\n",
           inputs[0], inputs[1], inputs[2], inputs[3],
           strings[action(actual)], strings[action(target)]);

    } else {
      sum++;
    }

}

printf("Network is %g%% correct\n",
        ((float)sum / (float)MAX_SAMPLES) * 100.0);

/* Выполнение тестов */

/*  Здоровье            Нож       Пистолет         Враг*/
inputs[0] = 2.0; inputs[1] = 1.0; inputs[2] = 1.0; inputs[3] = 1.0;
feedForward();
printf("2111 Action %s\n", strings[action(actual)]);

inputs[0] = 1.0; inputs[1] = 1.0; inputs[2] = 1.0; inputs[3] = 2.0;
feedForward();
printf("1112 Action %s\n", strings[action(actual)]);

inputs[0] = 0.0; inputs[1] = 0.0; inputs[2] = 0.0; inputs[3] = 0.0;
feedForward();
printf("0000 Action %s\n", strings[action(actual)]);

inputs[0] = 0.0; inputs[1] = 1.0; inputs[2] = 1.0; inputs[3] = 1.0;
feedForward();
printf("0111 Action %s\n", strings[action(actual)]);

inputs[0] = 2.0; inputs[1] = 0.0; inputs[2] = 1.0; inputs[3] = 3.0;
feedForward();
printf("2013 Action %s\n", strings[action(actual)]);
```

```
    inputs[0] = 2.0; inputs[1] = 1.0; inputs[2] = 0.0; inputs[3] = 3.0;
    feedForward();
    printf("2103 Action %s\n", strings[action(actual)]);

    inputs[0] = 0.0; inputs[1] = 1.0; inputs[2] = 0.0; inputs[3] = 3.0;
    feedForward();
    printf("0103 Action %s\n", strings[action(actual)]);

    fclose(out);

    return 0;
}
```

После инициализации генератора случайных чисел с помощью функции `srand` будут произвольно сгенерированы веса соединений сети. Затем для обучения сети выполняется цикл `while`. Примеры рассматриваются последовательно, а не в произвольном порядке. Выполняется расчет реакции сети на входной вектор, а затем – проверка средней ошибки. Наконец, запускается алгоритм обратного распространения, чтобы изменить веса соединений сети. После выполнения ряда итераций программа выходит из цикла, и сеть тестируется на точность на основе значений, заданных для обучения. После стандартного тестирования выполняется тестирование сети с примерами, которые не входили в начальную группу при обучении.

Если нейронная сеть будет применяться в компьютерной игре, веса могут быть выведены в файл для последующего использования. На этом этапе сеть можно оптимизировать, чтобы уменьшить количество ресурсов, необходимых для расчетов.

Обучение нейроконтроллера

Когда нейроконтроллер встраивается в игру, его обучение считается завершенным и не может продолжаться дальше. Чтобы дать персонажу возможность обучаться, вы можете добавить в игру элементы алгоритма обратного распространения. Это позволит изменять веса в зависимости от ситуаций игрового процесса.

Очень простой механизм может изменять веса нейроконтроллера на основании последнего действия, выполненного персонажем. Если действие привело к отрицательным последствиям, например, смерти персонажа, веса для действия в данной обстановке могут быть запрещены. Это снизит вероятность повторения указанного действия в будущем.

Далее все ИИ-персонажи игры могут получать одни и те же знания, пользуясь тем, что называется «эволюцией Ламарка», при которой дети учатся на ошибках своих родителей. Пройдя несколько игр, персонажи постепенно будут стремиться избегать отрицательных результатов.

Память нейроконтроллера

В нейронной сети может быть сформирована функция памяти. Для всех входов создаются линии задержки (при этом входной вектор расширяется в одном или двух измерениях). Первый вход от окружающей среды не исчезает, а становится частью другого входа в сеть. Данная функция может быть выполнена для нескольких элементов, что позволяет запрограммировать для персонажа некоторую память, а также многие другие возможности, свойственные разумным существам.

Этот метод также включает обратную связь. Последние действия могут направляться в сеть, что позволяет нейроконтроллеру помнить о них. Дополнительная обратная связь может включать информацию о здоровье персонажа, а также его эмоциональном состоянии. Данные механизмы способствуют генерации сложных действий, похожих на поступки человека.

Другие области применения

Алгоритм обратного распространения может использоваться для обучения нейронных сетей, которые имеют различное назначение. Ниже приведен краткий список возможных областей применения алгоритма:

- общее распознавание моделей;
- диагностика ошибок;
- мониторинг состояния пациентов врача;
- распознавание персонажей;
- фильтрация данных;
- анализ запахов и ароматов;
- распознавание фальшивых банкнот и документов.

Итоги

В этой главе рассматривались нейронные сети и алгоритм обратного распространения. Для иллюстрации алгоритма было выбрано обучение нейроконтроллеров в компьютерных играх. Рассказывалось, как алгоритм позволяет нейроконтроллеру правильно реагировать на непредвиденные ситуации. Из-за высоких системных требований алгоритм обратного распространения может не подходить для всех игровых архитектур, однако он позволяет добиться наибольшей реалистичности в поведении персонажа, добавляя нелинейные связи между окружающей средой и выбором действия.

Литература и ресурсы

1. Ваггонер Б., Спир Б. Жан-Баптист Ламарк (1744–1829) (Waggoner B., Speer B. Jean-Baptiste Lamarck (1744–1829)). Доступно по адресу http://www.ucmp.berkeley.edu/history/lamarck.html).
2. Галлант С. Обучение в нейронных сетях и экспертные системы (Gallant S. L., Neutral Network Learning and Expert Systems. – Cambridge, Mass.: MIT Press, 1994).
3. Мински М., Паперт С. Перцептроны: введение в компьютерную геометрию (Minsky M., Papert S. Perceptrons: An Introduction to Computational Geometry. – Cambridge, Mass.: MIT Press, 1969).

Глава 6. Введение в генетические алгоритмы

В этой главе моделирование эволюции рассматривается как средство решения компьютерных проблем. Генетический алгоритм, разработанный Джоном Холландом (John Holland), является поисковой программой, которая работает с группой закодированных решений заданной проблемы. Чтобы проиллюстрировать выполнение алгоритма, мы воспользуемся расчетом по эволюционной модели для выведения последовательности действий, которые представляют собой простой алгоритм. Джон Коза (John Koza) назвал этот процесс генетическим программированием.

Биологическое побуждение

Генетический алгоритм (Genetic algorithm) представляет собой технику оптимизации, которая моделирует феномен естественной эволюции (впервые открытый Чарльзом Дарвином). При естественной эволюции выживают и дают самое многочисленное потомство особи, наиболее адаптированные к сложным условиям окружающей среды. Степень адаптации, в свою очередь, зависит от набора хромосом конкретной особи, полученным от родителей. Это основа выживания сильнейшего – не только процесс выживания, но и участие в формировании следующего поколения. В природе выживание является определяющей и основной функцией.

Генетический алгоритм

Генетический алгоритм не пытается оптимизировать единственное решение. Он работает с группой решений, которые кодируются, подобно хромосомам. Отдельные гены хромосомы представляют собой уникальные переменные для изучаемой проблемы (рис. 6.1).

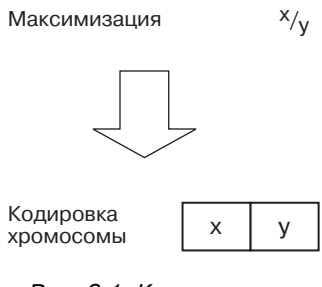


Рис. 6.1. Кодирование решений в хромосомы

Как видно на простом примере (рис. 6.1), за основу берутся параметры проблемы и создается хромосома, которая представляет собой два уникальных независимых параметра. Параметры могут быть группой битов, переменными с плавающей точкой или простыми двузначными числами в двоичном коде.

При отборе «здоровых» хромосом из популяции и использовании генетических операторов

(таких как рекомбинирование и мутация) в популяции остаются только те хромосомы, которые лучше всех приспособлены к окружающей среде, то есть наиболее полно отвечают задаче. Именно так работает теорема Холланда.

Выполнение генетического алгоритма включает три основных шага (причем для каждого шага предусмотрен большой разброс возможных вариантов). Эти шаги показаны на рис. 6.2.

Генетический алгоритм выполняется в три этапа (если не учитывать начальное создание популяции). Во время оценки определяется здоровье популяции. Далее производится отбор подгруппы хромосом на основании предварительно заданного критерия. Наконец, выбранная подгруппа рекомбинируется, в результате чего получается новая популяция. Алгоритм выполняется заново с новой популяцией. Процесс продолжается до тех пор, пока не будет достигнут определенный предел. Тогда работа алгоритма считается завершенной.

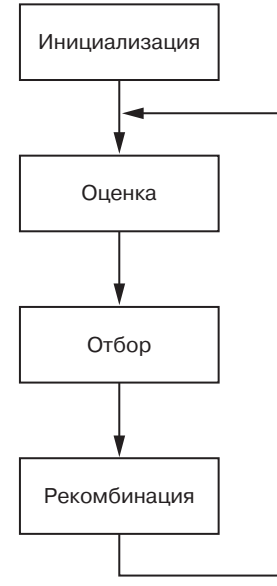


Рис. 6.2. Работа генетического алгоритма

Инициализация

Создание начальной популяции (Initialization) позволяет сформировать отправную точку для работы алгоритма. Обычно это выполняется путем произвольного создания хромосом, но также допускается добавлять в популяцию «здоровые» хромосомы (рис. 6.3).

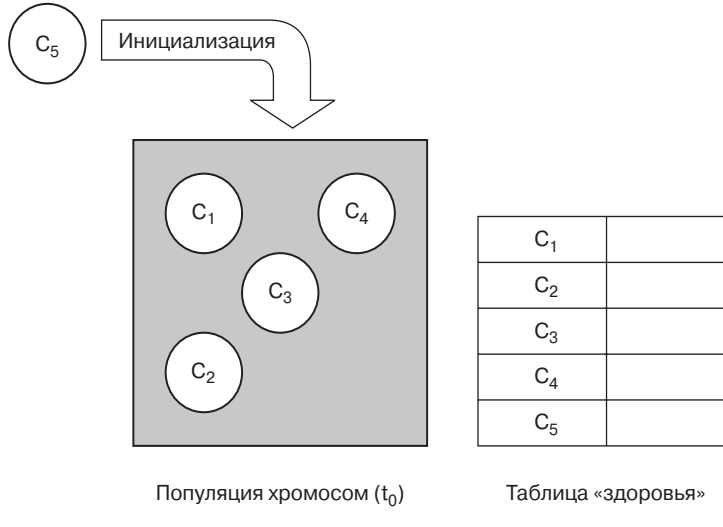


Рис. 6.3. Инициализация генетического фонда

Совет *Важное ограничение заключается в том, что начальная популяция должна быть разнообразной. Убедиться в этом можно при выполнении второго шага, определяющего, что все хромосомы различны. При отсутствии различия результаты работы алгоритма будут неудовлетворительными.*

Оценка

Этап *оценки* (Evaluation) дает возможность определить, как каждая хромосома (решение) справляется с данной проблемой. Алгоритм декодирует хромосому применительно к проблеме и проверяет результат решения проблемы с использованием новых параметров. Затем на основании результата рассчитывается «здоровье» хромосомы (рис. 6.4).

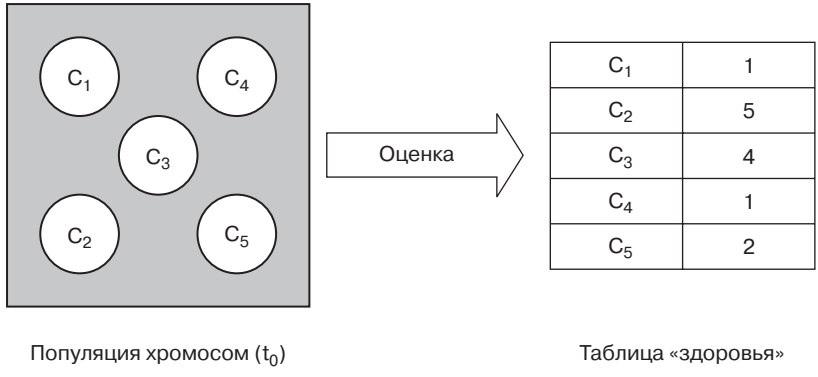


Рис. 6.4. Оценка популяции

Отбор

Отбор (Selection) является, вероятно, наиболее важным и самым трудным для понимания этапом генетического алгоритма. На этом этапе хромосомы выбираются для дальнейшего использования в другой популяции. Отбор осуществляется на основании здоровья хромосом (то есть того, насколько эффективно они решают данную проблему). Этот процесс является двусторонним, так как если включить в выбор только очень здоровые хромосомы, то решение становится слишком ограниченным по причине недостаточного разнообразия. Если выбор осуществляется произвольно, то нет гарантии, что здоровье последующих поколений будет улучшаться. В результате выбирается группа хромосом, которые будут участвовать в рекомбинировании (или скрещивании). Процесс выбора представлен на рис. 6.5.

Существует множество алгоритмов отбора. На рис. 6.5 показан алгоритм, известный как «Отбор по методу рулетки», или *вероятностный отбор*. При использовании этого метода отбор из популяции основывается на здоровье хромосомы. Чем лучше здоровье хромосомы, тем больше вероятность того, что она будет

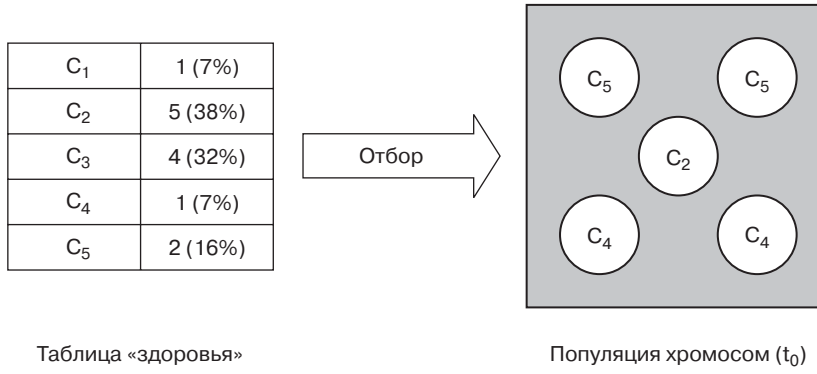


Рис. 6.5. Отбор хромосом на основании их здоровья

выбрана (или повторно выбрана) для формирования следующего поколения. Таким образом, вероятность выбора пропорциональна здоровью хромосомы. На рис. 6.5 хромосома 2 была выбрана для дальнейшего использования два раза, хромосома 3 – тоже два раза, а хромосома 5 – только один раз. Хромосомы 1 и 4 не были выбраны ни разу, поэтому они исчезают из последующей популяции.

Рекомбинирование

При *рекомбинировании* (Recombination) части хромосом перемещаются, может быть, даже изменяются, а получившиеся новые хромосомы возвращаются обратно в популяцию для формирования следующего поколения. Первая группа хромосом обычно называется родителями, а вторая – детьми. С одинаковой вероятностью могут применяться один или несколько генетических операторов. Доступные операторы включают мутацию и перекрестное скрещивание, которые являются аналогами одноименных генетических процессов. В следующем разделе рассматриваются примеры генетических операторов. Процесс рекомбинирования представлен на рис. 6.6.

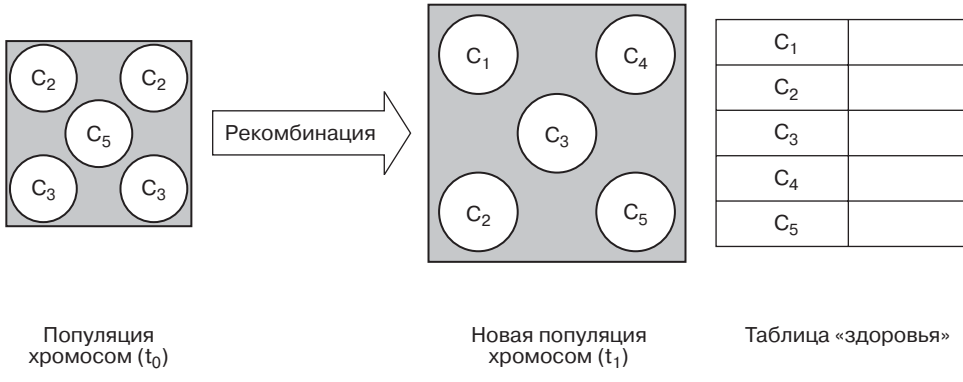


Рис. 6.6. Рекомбинирование хромосом для новой популяции

В результате рекомбинирования образуется новая популяция хромосом. Процесс повторяется заново с этапа оценки до тех пор, пока проблема не будет решена или пока не будет выполнено любое другое условие завершения алгоритма (например, максимально возможное количество поколений).

Генетические операторы

Существует огромное количество *генетических операторов* (Genetic operator), однако только несколько используется при решении общих задач. Перекрестное скрещивание и мутация, о которых рассказывается ниже, являются прямыми аналогами естественных процессов.

Перекрестное скрещивание

Оператор *перекрестного скрещивания* (Crossover) берет две хромосомы, разделяет их в произвольной точке (для каждой хромосомы), а затем меняет местами получившиеся «хвосты». При этом образуются две новые хромосомы. Разделение хромосомы в одной точке (получившее название перекрестного скрещивания в одной точке) является не единственной возможностью (рис. 6.7а). Также можно использовать разделение в нескольких точках (рис. 6.7б).

При перекрестном скрещивании в популяции не создается новый материал, выполняется просто ее изменение с целью создания новых хромосом. Это позволяет генетическому алгоритму выполнять поиск решения проблемы среди существующих решений. Оператор перекрестного скрещивания является самым важным и используется чаще всего. Второй оператор, мутация, предлагает возможность создания нового материала в популяции.

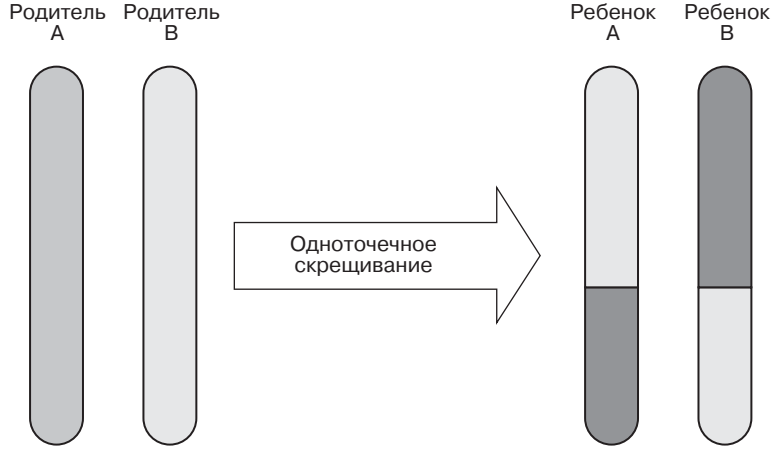


Рис. 6.7а. Перекрестное скрещивание в одной точке

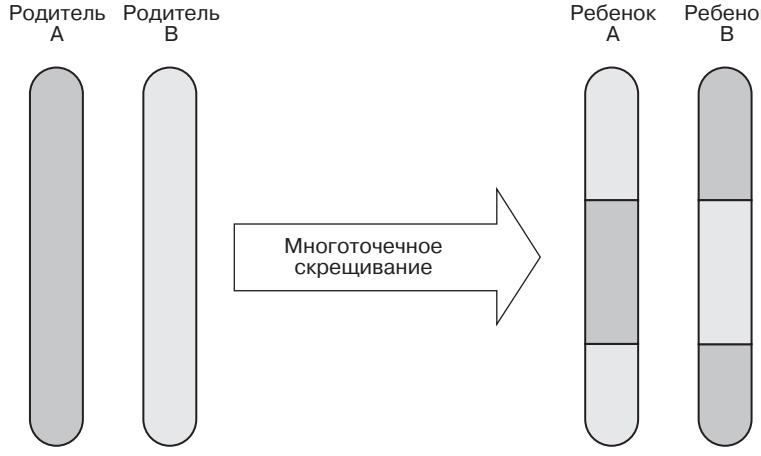


Рис. 6.7б. Перекрестное скрещивание в нескольких точках

Мутация

Оператор *мутации* (Mutation) вносит произвольное изменение в гены хромосомы (иногда даже несколько изменений в зависимости от частоты применения). Он позволяет создавать в популяции новый материал. Так как новые хромосомы просто перемешиваются с уже существующими, мутация предлагает возможность «перетряхнуть» популяцию и расширить область поиска решения (рис. 6.8).

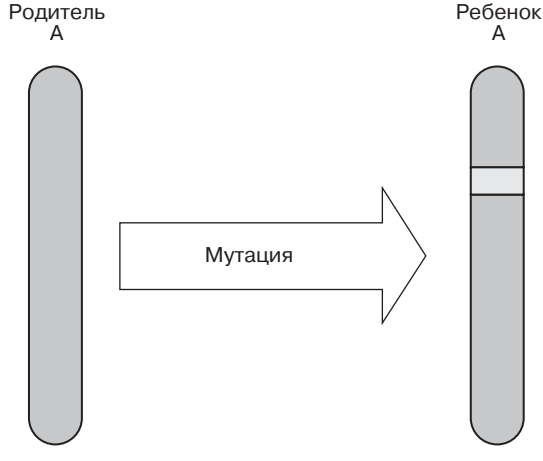


Рис. 6.8. Мутация одной хромосомы

Пример выполнения генетического алгоритма

До сих пор наше обсуждение было теоретическим. Теперь рассмотрим итерацию алгоритма, чтобы понять принцип его работы. В этом примере выполняется поиск параметров, которые позволяют максимизировать уравнение 6.1:

$$f(x,y)=\frac{1}{1+x^2+y^2} \tag{6.1}$$

Это уравнение графически представлено на рис. 6.9 (трехмерный график) и рис. 6.10 (контурный график).

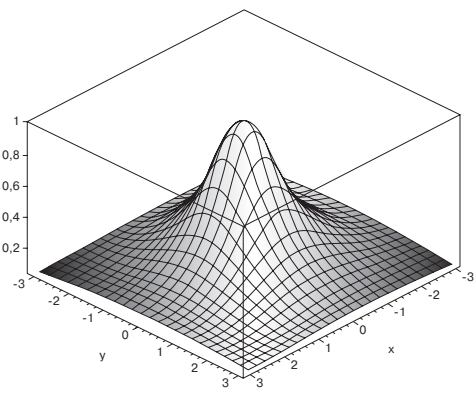


Рис. 6.9. Графическое представление уравнения 6.1

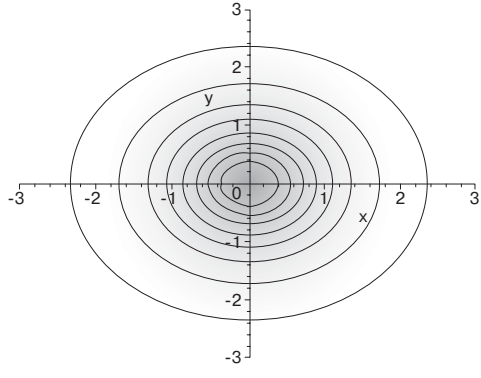


Рис. 6.10. Контурный график уравнения 6.1, показывающий z-измерение при помощи тени

Хромосомы в этом примере были построены из двух генов (или параметров), x и y. Сначала необходимо создать фонд хромосом, к которым будет применяться генетический алгоритм. В табл. 6.1 показана начальная популяция, которая была выбрана произвольно.

Таблица 6.1. Начальная популяция (t_0)

Хромосома	x	y	Здоровье
C_0	–1	2	?
C_1	–2	3	?
C_2	1,5	0	?
C_3	0,5	–1	?

Теперь, когда сформирована начальная популяция, можно приступать к оценке. Чтобы оценить хромосомы в популяции, следует определить здоровье каждой хромосомы. В этом случае здоровье равно значению уравнения 6.1 для соответствующих значений параметров. Чем больше значение уравнения, тем лучше здоровье хромосомы. В табл. 6.2 показаны значения здоровья для оцениваемых хромосом.

Таблица 6.2. Здоровье начальной популяции

Хромосома	x	y	Здоровье
C_0	–1	2	0,167
C_1	–2	3	0,007
C_2	1,5	0	0,31
C_3	0,5	–1	0,44

Хромосома C_1 очень маленькая, и поэтому вероятность ее выбора невелика. Необходимо выбрать две пары родителей для рекомбинирования. Каждый родитель при перекрестном скрещивании будет влиять на детей в следующем поколении. На основании показателей здоровья алгоритм выбирает хромосомы C_3 и C_2 в качестве первой пары родителей, а хромосомы C_3 и C_0 – в качестве второй пары. Поскольку в хромосоме только два гена, нужно осуществить разделение на генном уровне, что приводит к разделению элементов x и y хромосомы. В табл. 6.3 показана новая популяция (источники генов заключены в круглые скобки), а также здоровье каждой новой хромосомы.

Таблица 6.3. Новая популяция и ее здоровье (популяция t_1)

Хромосома	x	y	Здоровье
C_0 $(x_{c3}y_{c2})$	0,5	0	0,8
C_1 $(x_{c2}y_{c3})$	1,5	–1	0,24
C_2 $(x_{c3}y_{c0})$	0,5	2	0,19
C_3 $(x_{c0}y_{c3})$	–1	–1	0,33

Теперь сравним значения здоровья, представленные в табл. 6.2, со значениями здоровья новой популяции в табл. 6.3. Очевидно, что данные значения увеличились (наибольшее значение в t_0 было равно 0,44, а наибольшее значение в t_1 составляет 0,8). Это значит, что лучшее решение в новой популяции эффективнее, чем лучшее решение в предыдущей популяции. Среднее здоровье в первой популяции равно 0,231, а среднее здоровье в новой популяции составляет 0,39, что демонстрирует явное улучшение в популяции при использовании генетического алгоритма.

В этом примере не применялся оператор мутации. Для решения подобной проблемы можно использовать мутацию, как небольшие произвольные изменения значений генов (например, прибавление или вычитание небольшого значения, скажем, 0,1).

Примечание *Этот пример показывает, что с помощью оператора перекрестного скрещивания нельзя добиться идеального решения, поскольку для поиска решения он использует уже имеющийся генетический материал популяции. Существует ограниченное число комбинаций, поэтому для расширения области поиска следует использовать мутацию, которая позволяет внести в популяцию новый материал.*

Пример задачи

Освоив базовые принципы работы генетического алгоритма, воспользуемся им для решения проблемы оптимизации. Вместо того чтобы фокусироваться на традиционных числовых проблемах оптимизации, попробуем оптимизировать более символическую проблему с последовательностью указаний (относящуюся к области эволюционного программирования).

Обзор

Представим простой набор инструкций для архитектуры с нулевыми адресами (стековой архитектуры). Виртуальный компьютер не имеет регистров, только набор значений, которыми можно управлять посредством инструкций. Компьютер распознает только одну группу инструкций, которые представлены в табл. 6.4.

Таблица 6.4. Простой набор инструкций

Инструкция	Описание
DUP	Копирует верхний объект стека (стек: A ⇒ AA)
SWAP	Меняет два верхних элемента стека местами (стек: AB ⇒ BA)
MUL	Перемножает два верхних элемента стека (стек: 23 ⇒ 6)
ADD	Складывает два верхних элемента стека (стек: 23 ⇒ 5)
OVER	Копирует второй сверху объект стека (стек: AB ⇒ BAB)
NOP	Нет операции или пустая операция (заполнитель)

Эти инструкции очень просты и могут использоваться во многих функциях. Например, если требуется рассчитать площадь квадрата, можно использовать такую последовательность инструкций (предполагается, что в стек уже введено значение длины стороны):

```
DUP
MUL
```

Эта последовательность дуплицирует значение верхней ячейки стека, а затем перемножает два элемента ($X \times X$, или X^2).

Кодировка решения

Вспомните, что кодируется решение задачи, а не сама задача (так как сама задача используется в качестве критерия здоровья хромосом). Задача довольно проста, поскольку хромосома отображает набор инструкций в одном блоке. Для каждой инструкции нужно задать числовое значение (от `DUP` = 0 до `NOP` = 5). При этом кодировка будет являться группой байтов, представляющих набор инструкций.

Оценка здоровья

Оценка здоровья заданной хромосомы представляет собой выполнение набора инструкций, которые содержатся в хромосоме. Создается стек, в него загружаются

начальные значения, затем последовательно выполняются все инструкции до тех пор, пока программа не завершится или не достигнет инструкции END.

Затем здоровье рассчитывается как разница между результатом вычислений, хранящемся в стеке, и тем значением, которое ожидалось от функции (было задано до тестирования).

Рекомбинирование

Для решения этой проблемы используется как перекрестное скрещивание, так и мутация. Оператор перекрестного скрещивания разбивает поток инструкций в одной точке, а затем отделяет часть от каждого родителя. Мутация в произвольном порядке изменяет ген и создает новую инструкцию (так как хромосома представляет собой программу, каждый ген является инструкцией).

Совет *Исходный код содержится в архиве с примерами к книге, находящемся на сайте издательства «ДМК Пресс» www.dmk.ru.*

Обсуждение кода

Исходный код реализации генетического алгоритма очень прост, как и виртуальная машина, на которой выполняется код хромосомы. Сначала рассмотрим реализацию виртуальной машины.

Реализация виртуальной машины

Виртуальная машина (Virtual machine – VM) представляет собой очень простую архитектуру, которая поддерживает стек целочисленных значений и шесть инструкций, работающих со значениями. В листинге 6.1 показан код VM (файлы common.h и stm.c).

Листинг 6.1. Реализация виртуальной машины

```
#define DUP         0x00
#define SWAP        0x01
#define MUL         0x02
#define ADD         0x03
#define OVER        0x04
#define NOP         0x05

#define MAX_INSTRUCTION     (NOP+1)

#define NONE                0
#define STACK_VIOLATION     1
#define MATH_VIOLATION      2

#define STACK_DEPTH         25

int stack[STACK_DEPTH];
int stackPointer;
```

```
#define ASSERT_STACK_ELEMENTS(x) \
   if (stackPointer < x) { error = STACK_VIOLATION ; break; }

#define ASSERT_STACK_NOT_FULL \
   if (stackPointer == STACK_DEPTH) { error = STACK_VIOLATION ;
     break; }

#define SPUSH(x) (stack[stackPointer++] = x)
#define SPOP     (stack[==stackPointer])
#define SPEEK    (stack[stackPointer-1])

/*
* interpretSTM
*
* program - последовательность инструкций (программа)
* progLength - длина программы
* args - список аргументов
* argsLength - количество аргументов
/*

int interpretSTM(const int *program, int progLength,
                  const int *args, int argsLength)
{
   int pc = 0;
   int i, error = NONE;
   int a, b;

   stackPointer = 0;

   /* Загрузка аргументов в стек */
   for (i = argsLength-1 ; i >= 0 ; i--) {
     SPUSH(args[i]);
  }

/* Выполнение программы */
while ((error == NONE) && (pc < progLength)) {

switch(program[pc++]) {

   case DUP:
     ASSERT_STACK_ELEMENTS(1);
     ASSERT_STACK_NOT_FULL;
     SPUSH(SPEEK);
     break;

   case SWAP:
     ASSERT_STACK_ELEMENTS(2);
     a = stack[stackPointer-1];
     stack[stackPointer-1] = stack[stackPointer-2];
```

```
        stack[stackPointer-2] = a;
        break;

      case MUL:
        ASSERT_STACK_ELEMENTS(2);
        a = SPOP; b = SPOP;
        SPUSH(a * b);
        break;

      case ADD:
        ASSERT_STACK_ELEMENTS(2);
        a = SPOP; b = SPOP;
        SPUSH(a + b);
        break;

      case OVER:
        ASSERT_STACK_ELEMENTS(2);
        SPUSH(stack[stackPointer-2]);
        break;

      } /* Switch */

      } /* Цикл */

      return(error);
  }
```

В первой части листинга 6.1 представлены определения инструкций, которые поддерживаются виртуальной машиной. Также задается ряд других констант, которые позволяют понять, почему был выполнен выход из программы (если успешно, то возвращается значение `NONE`, если нет, то указывается причина ошибки). Кроме того, создается стек (массив `stack`) заданной глубины и ряд макросов, которые упрощают реализацию VM. Макросы `ASSERT_XXX` используются для идентификации нарушений в программе (например, выхода за пределы стека или использования объекта, который в него не входит). Макросы `SPEEK`, `SPUSH` и `SPOP` являются простейшими стековыми командами.

Функция `interpretSTM` представляет собой реализацию виртуальной машины. Она принимает на вход набор инструкций (`program`) и его длину (`progLength`), а также набор начальных значений (`args`) и их количество (`argsLength`). Первым действием VM будет сохранение начальных значений в стеке. При занесении значений в стек выполняется движение по списку от конца к началу, поэтому объект, указанный первым, будет располагаться наверху.

Программа выполняет все инструкции, а когда достигается конец программы, виртуальная машина возвращает текущее сообщение об ошибке. Если ошибка была получена в ходе работы программы, устанавливается соответствующий код ошибки и возврат выполняется автоматически.

Применение генетического алгоритма

Рассматриваемый генетический алгоритм будет следовать общим принципам формирования генетических алгоритмов, о которых рассказывалось ранее в этой главе (рис. 6.2). В данном разделе выполнение алгоритма изучается последовательно – функция `main`, затем функции, реализующие инициализацию, оценку, отбор и рекомбинирование.

Функция main

Функция `main`, представленная в листинге 6.2, иллюстрирует главный поток выполнения алгоритма.

Листинг 6.2. Функция `main` – главный поток генетического алгоритма

```
int main()
{
   int generation = 0, i;
   FILE *fp;
   extern float minFitness, maxFitness, avgFitness;
   extern int curCrossovers, curMutations;
   extern int   curPop;

   void printProgram( int, int );

   /* Инициализация генератора случайных чисел */
   srand(time(NULL));

   curPop = 0;

  fp = fopen("stats.txt", "w");

   if (fp == NULL) exit(-1);

   /* Инициализируем начальную популяцию и проверяем здоровье
    * хромосом в ней
    */
   initPopulation();
   performFitnessCheck( fp );

   /* Цикл до максимального количества поколений */
   while (generation < MAX_GENERATIONS) {

   curCrossovers = curMutations = 0;

   /* Выбрать 2-х родителей, и, скрестив их, создать 2-х детей */
   performSelection();

   /* Смена поколений */
   curPop = (curPop == 0) ? 1 : 0;
```

```
   /* Вычислить здоровье новой популяции */
   performFitnessCheck( fp );

   /* Вывести статистику (каждые 100 поколений) */
   if ((generation++ % 100) == 0) {
   printf("Generation %d\n", generation-1);
   printf("\tmaxFitness = %f (%g)\n", maxFitness, MAX_FIT);
   printf("\tavgFitness = %f\n", avgFitness);
   printf("\tminFitness = %f\n", minFitness);
   printf("\tCrossovers = %d\n", curCrossovers);
   printf("\tMutation   = %d\n", curMutations);
   printf("\tpercentage = %f\n", avgFitness / maxFitness);
  }

/* Проверить однородность популяции. Если популяция однородна, то
 * выйти из программы
 */
if ( generation > (MAX_GENERATIONS * 0.25) ) {
      if ((avgFitness / maxFitness) > 0.98) {
        printf("converged\n");
        break;
      }
    }

    if (maxFitness == MAX_FIT) {
      printf("found solution\n");
      break;
    }

  }

/* Вывести окончательную статистику */
printf("Generation %d\n", generation-1);
  printf("\tmaxFitness = %f (%g)\n", maxFitness, MAX_FIT);
  printf("\tavgFitness = %f\n", avgFitness);
  printf("\tminFitness = %f\n", minFitness);
  printf("\tCrossovers = %d\n", curCrossovers);
  printf("\tMutation   = %d\n", curMutations);
  printf("\tpercentage = %f\n", avgFitness / maxFitness);

/* Вывести окончательную статистику */
  for (i = 0 ; i < MAX_CHROMS ; i++) {

    if (populations[curPop][i].fitness == maxFitness) {
      int index;
      printf("Program %3d : ", i);
```

```
        for (index = 0 ; index < populations[curPop][i].progSize ;
              index++) {
          printf("%02d ", populations[curPop][i].program[index]);
        }
        printf("\n");
        printf("Fitness %f\n", populations[curPop][i].fitness);
        printf("ProgSize %d\n\n", populations[curPop][i].progSize);

        printProgram(i, curPop);

        break;
      }

    }

    return 0;
}
```

Функция `main` работает следующим образом. Сначала с помощью функции `srand` инициализируется генератор случайных чисел. Открывается файл, в который будет выводиться информация о здоровье хромосом (о нем будет рассказано в следующем разделе). Затем популяция инициализируется с помощью произвольно выбранных хромосом (функция `initPopulation`), и выполняется проверка здоровья хромосом (функция `performFitnessCheck`), поскольку здоровье каждой хромосомы важно для процесса отбора (функция `performSelection`). Дело в том, что алгоритм использует отбор, вероятность которого основывается на значении здоровья. В данном листинге рекомбинирование отдельно не показано, так как оно выполняется одновременно с процессом отбора.

После отбора обновляется переменная `curPop`, которая определяет текущую популяцию. Популяция хромосом представляет собой двумерный массив, который хранит две популяции (старую и новую) – см. листинг 6.3.

Листинг 6.3. Определение популяций

```
typedef struct population {
   float fitness;
   int progSize;
   int program[MAX_PROGRAM];

} POPULATION_TYPE;

POPULATION_TYPE populations [2] [MAX_CHROMS];
int curPop;
```

Популяция состоит из группы хромосом (задается как `MAX_CHROMS`), при этом каждая хромосома представляет собой саму программу (поле `program`), размер программы (поле `progSize`) и «здоровье» программы (поле `fitness`). Переменная `populations` хранит две популяции (старую и новую). Переменная

curPop указывает, какая популяция рассматривается в текущий момент; в нее вносятся все изменения. При рекомбинировании родители выбираются из текущей популяции (указывается переменной curPop), а новые хромосомы помещаются в другую популяцию (определяется выражением !curPop).

Главный цикл функции main (цикл поколений) выполняет отбор, проверку здоровья и перемещает популяции при каждой итерации. Остальные команды цикла выдают информацию о работе программы и выполняют проверку на условия выхода. Если среднее здоровье составляет 98% от максимального здоровья, то популяция считается однородной (то есть содержащей одинаковые хромосомы), и программа завершает работу. Если максимальное здоровье равно максимально допустимому здоровью для функции, работа также заканчивается, потому что обнаружено решение задачи. После того как решение было найдено, выводится дополнительная информация для пользователя, а также отображается программа (хромосома), которая имеет максимальное здоровье.

Инициализация

Инициализация популяции представляет собой довольно простой процесс. Программа проходит по всем хромосомам популяции и присваивает каждому гену произвольную инструкцию. Здоровье всех хромосом обнуляется, а их размер задается максимально допустимым значением (листинг 6.4).

Листинг 6.4. Инициализация популяции

```
void initMember( pop, index )
{
  int progIndex;

  populations[pop][index].fitness = 0.0;
  populations[pop][index].progSize = MAX_PROGRAM-1;

   /* Случайным образом создаем новую программу */
   progIndex = 0;
   while (progIndex < MAX_PROGRAM) {
     populations[pop][index].program[progIndex++] =
                                    getRand(MAX_INSTRUCTION);
   }

}

void initPopulation( void )
{
  int index;

/* Инициализируем каждую хромосому в популяции */
for (index = 0 ; index < MAX_CHROMS ; index++) {
    initMember(curPop, index);
  }
}
```

Из функции main вызывается функция initPopulation, которая, в свою очередь, вызывает функцию initMember, чтобы инициализировать каждую хромосому популяции.

Оценка здоровья

Оценка здоровья выполняется с помощью функции performFitnessCheck (см. листинг 6.5).

Листинг 6.5. Оценка здоровья популяции

```
float maxFitness;
float avgFitness;
float minFitness;

extern int stackPointer;
extern int stack[];

static int x = 0;

float totFitness;

int performFitnessCheck( FILE *outP )
{

  int chrom, result, i;
  int args[10], answer;

  maxFitness = 0.0;
  avgFitness = 0.0;
  minFitness = 1000.0;
  totFitness = 0.0;

  for ( chrom = 0 ; chrom < MAX_CHROMS ; chrom++ ) {

    populations[curPop][chrom].fitness = 0.0;

    for ( i = 0 ; i < COUNT ; i++ ) {

      args[0] = (rand() & 0x1f) + 1;
      args[1] = (rand() & 0x1f) + 1;
      args[2] = (rand() & 0x1f) + 1;

      /* Задача: x^3 + y^2 + z */
      answer = (args[0] * args[1]) +
               (args[1] * args[1]) + args[2];

  /* Вызов виртуальной машины для проверки программы (хромосомы) */
  result = interpretSTM(populations[curPop][chrom].program,
                        populations[curPop][chrom].progSize,
                          args, 3);
```

```
/* Если не было ошибки, то добавить к здоровью */
if (result == NONE) {
        populations[curPop][chrom].fitness += TIER1;
}

/* Если в стеке только одно значение, то прибавить его к здоровью
 * хромосомы
 */
if (stackPointer == 1) {
   populations[curPop][chrom].fitness += TIER2;
}

/* Если в стеке находится правильный ответ, то прибавить его
 * к здоровью хромосомы
 */
if (stack[0] == answer) {
   populations[curPop][chrom].fitness += TIER3;
}

/* Если здоровье этой хромосомы больше, чем ранее найденное
 * максимальное, то обновить статистику
 */
if (populations[curPop][chrom].fitness > maxFitness) {
      maxFitness = populations[curPop][chrom].fitness;
   } else if (populations[curPop][chrom].fitness < minFitness) {
      minFitness = populations[curPop][chrom].fitness;
    }

   /* Обновить значение общего здоровья популяции */
   totFitness += populations[curPop][chrom].fitness;

}

/* Вычислить среднее здоровье популяции */
avgFitness = totFitness / (float)MAX_CHROMS;

  if (outP) {
   /* Вывести статистику в файл */
   fprintf(outP, "%d %6.4f %6.4f %6.4f\n",
             x++, minFitness, avgFitness, maxFitness);
  }

  return 0;
}
```

Функция `performFitnessCheck` изучает все хромосомы в текущей популяции и рассчитывает их здоровье. Затем здоровье сохраняется как информация о хромосоме в структуре `populations`.

Алгоритм начинает работу с удаления всей информации о здоровье и подготовки к оценке текущей популяции. После удаления информации о здоровье

в цикле выполняется тест для всех хромосом. Чтобы избежать варианта, при котором хромосома выдаст правильный ответ, который будет работать только в одном случае, хромосомы проверяются несколько раз (в данном случае 10 раз). Массив `args` содержит аргументы задачи. Он загружается в стек виртуальной машины при вызове функции `evaluation`. В каждом случае массив `args` заполняется случайными значениями. Это позволяет удостовериться, что программа действительно решает поставленную задачу.

После того как для задачи были заданы аргументы, рассчитывается и помещается в переменную `answer` значение, которое должно получиться в результате. Затем вызывается функция `interpretSTM`, чтобы оценить здоровье (она представлена в листинге 6.1). На основании результата, возвращаемого функцией оценки, определяется здоровье хромосомы. Здоровье основывается на трех значениях. Если выход из программы прошел успешно (не было математической или программной ошибки), возвращается значение `TIER1`. Если в стеке осталось только одно значение, добавляется значение `TIER2`. Наконец, если в верхней ячейке стека находится правильное значение (ожидаемое значение `answer`), добавляется значение `TIER3`. Значения `TIERX` задаются таким образом, что значение `TIER3` является более важным, чем `TIER2`, а оно, в свою очередь, более важно, чем `TIER1`. Это условие дает генетическому алгоритму информацию для отбора. Генетический алгоритм работает наиболее эффективно при постепенном улучшении рассчитанного здоровья хромосомы по отношению к ожидаемому. Рассматриваемый пример не совсем четко иллюстрирует это правило, но тем не менее это так.

После завершения проверки здоровья текущей хромосомы программа определяет, является ли это значение самым большим или самым малым. Данная информация хранится в переменных `maxFitness` и `minFitness` соответственно. Затем рассчитывается значение `totFitness`, и после завершения работы цикла проверки всех хромосом определяется среднее здоровье популяции (хранится в переменной `avgFitness`).

Наконец, данные о текущем поколении сохраняются в файле вывода информации. Можно затем сравнить их с аналогичными данными следующих поколений, чтобы определить их прогресс.

Отбор и рекомбинирование

Во время отбора и рекомбинирования программа выбирает родителей из текущей популяции и создает из них новые хромосомы для следующего поколения. Этот процесс представлен в листинге 6.6.

Листинг 6.6. Выполнение отбора

```
int performSelection( void )
{
  int par1, par2;
  int child1, child2;
  int chrom;

/* Цикл по хромосомам с шагом 2 */
for (chrom = 0 ; chrom < MAX_CHROMS ; chrom+=2) {
     /* Выбираем 2-х родителей случайным образом */
```

```
      par1 = selectParent();
      par2 = selectParent();
      /* Дети помещаются в массив по текущему адресу */
      child1 = chrom;
      child2 = chrom+1;

      /* Рекомбинация родителей для получения потомства */
      performReproduction( par1, par2, child1, child2 );

    }
    return 0;
}
```

Данная функция работает с индексами в таблицах двух популяций. Переменные `par1` и `par2` являются индексами в текущей популяции, а переменные `child1` и `child2` – индексами в новой популяции. Алгоритм выбирает индексы `child` начиная с нуля и увеличивает их на два при каждом расчете. Индексы родителей определяются с помощью функции `selectParent`. После получения значений для четырех индексов вызывается функция `performReproduction`, которая выполняет рекомбинирование (см. листинг 6.7).

Листинг 6.7. Выбор родителя

```
int selectParent( void )
{
  static int chrom = 0;
  int ret = -1;
  float retFitness = 0.0;

/* Выбор случайным образом */
do {

   /* Получить значение коэффициента здоровья */
   retFitness = (populations[curPop][chrom].fitness / maxFitness);

    if (chrom == MAX_CHROMS) chrom = 0;
    /* Если случайное число превысит вычисленное здоровье хромосомы,
     * то выбираем эту хромосому
     */
if (populations[curPop][chrom].fitness > minFitness) {
      if (getSRand() < retFitness) {
        ret = chrom++;
        retFitness = populations[curPop][chrom].fitness;
        break;
      }
    }
    chrom++;

  } while (1);

  return ret;
}
```

Выбор родителей основывается на принципе, согласно которому шансы хромосомы быть выбранной пропорциональны ее здоровью в сравнении с общим здоровьем популяции. Эта вероятность (хранится в переменной `retFitness`) рассчитывается в начале цикла `do`. Затем выполняется проверка, которая позволяет убедиться, что здоровье текущей хромосомы выше, чем минимальное здоровье популяции. Другими словами, отсеиваются хромосомы, у которых самое низкое здоровье в популяции. Затем произвольное число (от 0 до 1) сравнивается со значением здоровья текущей хромосомы. Если произвольное значение меньше, программа выбирает родителя и позволяет функции вернуться. В противном случае программа переходит в начало цикла и рассчитывает значение здоровья для следующей хромосомы.

После того как оба родителя выбраны, выполняется рекомбинирование (см. листинг 6.8).

Листинг 6.8. Рекомбинирование родительских хромосом для создания двух новых детей

```
int performReproduction( int parentA, int parentB,
                         int childA, int childB )
{
  int crossPoint, i;
  int nextPop = (curPop == 0) ? 1 : 0;

  int mutate( int );

   /* Если применяем скрещивание, то генерируем точку скрещивания */
   if (getSRand() > XPROB) {
    crossPoint =
         getRand(MAX(populations[curPop][parentA].progSize-2,
                     populations[curPop][parentB].progSize-2))+1;
    curCrossovers++;
  } else {
    crossPoint = MAX_PROGRAM;
  }

   /* Выполнить скрещивание (дополнительно, случайным образом
    * выполняется мутация)
    */
   for (i = 0 ; i < crossPoint ; i++) {
    populations[nextPop][childA].program[i] =
        mutate(populations[curPop][parentA].program[i]);
    populations[nextPop][childB].program[i] =
        mutate(populations[curPop][parentB].program[i]);
  }

  for ( ; i < MAX_PROGRAM ; i++) {
    populations[nextPop][childA].program[i] =
        mutate(populations[curPop][parentB].program[i]);
```

```
      populations[nextPop][childB].program[i] =
          mutate(populations[curPop][parentA].program[i]);
    }

    /* Обновить длину программы для потомков */
    populations[nextPop][childA].progSize =
      populations[curPop][parentA].progSize;
    populations[nextPop][childB].progSize =
      populations[curPop][parentB].progSize;

    return 0;
}
```

Сначала выполняется проверка на необходимость использования оператора перекрестного скрещивания. Если случайное число больше, чем порог `XPROB`, рассчитывается точка пересечения (`crossPoint`) на основании максимальной длины хромосомы (рассматривается та из хромосом-родителей, длина которой больше). Размеры хромосом могут быть различными (хотя в данном примере они всегда максимальные). Точка пересечения не может быть первым или последним геном хромосомы (так как в этом случае пересечение не происходит). Если перекрестное скрещивание не будет выполняться, точка пересечения задается равной размеру программы.

Следующий шаг – выполнение пересечения двух хромосом. Родитель A копируется в ребенка A, а родитель B – в ребенка B и так вплоть до точки пересечения. В этой точке (второй цикл `for`) родитель A копируется в ребенка B, а родитель B – в ребенка A. Обратите внимание, что если точка пересечения равна размеру программы, то первый цикл `for` скопирует всю хромосому, а второму будет нечего копировать (то есть мы добьемся нужного результата).

Наконец, дети в новой популяции наследуют размеры своих родителей. В данном случае размер является постоянной величиной, но вы легко можете изменить эту установку во время инициализации алгоритма.

Обратите внимание, что переменная `nextPop` создается на основании `curPop`. Переменная `nextPop` задает популяцию, в которой будут созданы дети; переменная `curPop` показывает, из какой популяции брать родителей.

Последняя функция, `mutate`, изменяет ген в текущей хромосоме на новую инструкцию, которая зависит от вероятности мутации (листинг 6.9).

Листинг 6.9. Оператор мутации

```
int mutate(int gene)
{
  float temp = getSRand();

/* Если требуется мутация, то генерируем новую инструкцию
 * случайным образом
 */
```

```
if (temp > MPROB) {
      gene = getRand(MAX_INSTRUCTION);
      curMutations++;
  }

  return gene;
}
```

Примеры запуска

Рассмотрим несколько примеров запуска программы. Хотя при запуске выдается большое количество информации, включая тенденции изменения здоровья (в файле stats.txt), здесь описываются только результаты.

При первом тесте задача состояла в том, чтобы создать последовательность инструкций, которая решит уравнение 6.2:

$$x^8 \tag{6.2}$$

В результате получилась такая программа:

```
DUP MUL DUP MUL DUP MUL
```

Эта последовательность сначала рассчитывает квадрат для x, затем квадрат для полученного значения, затем еще раз – квадрат для полученного значения. Каждый раз используется инструкция `DUP`.

Следующий тест включил три переменные, как показано в уравнении 6.3:

$$(x \times 2) + (y \times 2) + z \tag{6.3}$$

В результате получилась такая программа:

```
ADD DUP ADD SWAP ADD
```

Данная программа является оптимизацией уравнения, при которой сначала суммируются x и y, а затем сумма умножается на 2 (через последовательность `DUP ADD`). Последняя инструкция `ADD` добавляет компонент z. Обратите внимание, что в этом случае была создана инструкция `SWAP`, которая не имеет значения, но присутствует в полученной программе.

Другой интересный пример показан в уравнении 6.4:

$$(x \times y) + (x \times y) + z \tag{6.4}$$

В результате получилась такая программа:

```
OVER ADD MUL ADD
```

Данная программа сначала копирует второй объект группы (y), затем помещает его в начало стека. Далее она суммирует x и y и умножает сумму на y. Наконец, с помощью последней инструкции `ADD` добавляется значение z. Это еще одно упрощение (оптимизация) начального уравнения.

Рассмотрим более сложный пример в уравнении 6.5:

$$x^3 + y^2 + z \tag{6.5}$$

Это уравнение было решено с помощью такой последовательности инструкций:

```
DUP DUP MUL MUL SWAP DUP MUL SWAP ADD SWAP SWAP ADD
```

Первые четыре инструкции (`DUP DUP MUL MUL`) рассчитывают значение для x^3. Далее переставляются два объекта в стеке; объект y теперь находится вверху. Последовательность `DUP MUL` рассчитывает значение для y^2. Наконец, остальные пять инструкций соединяют три значения. Обратите внимание, что инструкции `SWAP` являются излишними, однако они не разрушили последовательность и потому не были отброшены.

На рис. 6.11 представлен прогресс алгоритма в расчете последовательности инструкций для уравнения 6.5. Интересно отметить постепенное улучшение среднего здоровья на этом примере. После 10000 поколений график показывает постепенное улучшение здоровья вплоть до 20000 поколений. В этой точке возникает последовательное улучшение как максимального, так и среднего здоровья популяции. Затем максимальное здоровье популяции колеблется около максимальной отметки (2510) вплоть до 30000 поколений, пока задача окончательно не решается.

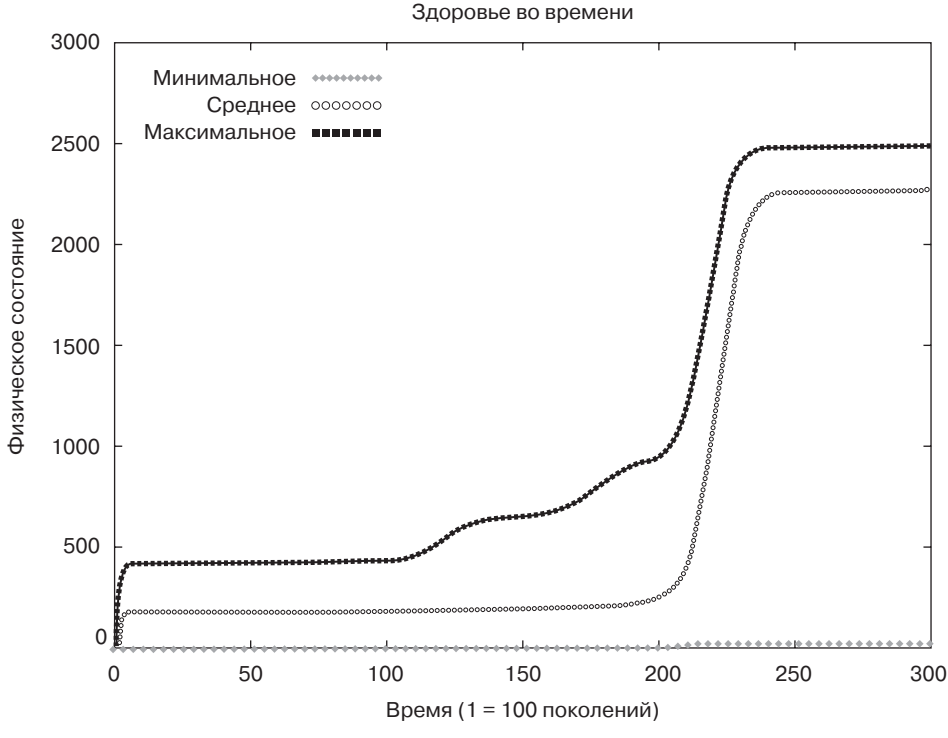


Рис. 6.11. График изменения здоровья во времени при решении уравнения 6.5

На графике представлен интересный момент – постепенное улучшение здоровья. Это доказывает, что генетический алгоритм полезен для решения символических проблем, подобных поставленной задаче. Несмотря на то что выбираются хромосомы, которые до конца не решают проблему, мутация и перекрестное скрещивание с другими хромосомами популяции ведут к повышению здоровья до тех пор, пока задача не будет полностью решена.

Настройка параметров и процессов

Можно не только изменять множество параметров генетического алгоритма (например, метод отбора или рекомбинирования), но и управлять коэффициентами применения генетических операторов. Получается, что не существует оптимального набора методов и параметров, который сумел бы решить любую задачу. Каждая задача должна рассматриваться отдельно, чтобы решить ее быстрее или более эффективным способом (а в некоторых случаях хотя бы решить). В этом разделе описываются способы настройки параметров генетического алгоритма для решения определенной задачи.

Метод отбора

Ранее рассматривался метод вероятностного выбора: чем выше здоровье хромосомы, тем больше вероятность, что она будет выбрана для формирования последующего поколения. Однако трудность вероятностного выбора состоит в том, что может отсеиваться даже здоровая хромосома. Чтобы гарантировать рекомбинирование самых здоровых хромосом, можно использовать *метод элиты*, который заключается в автоматическом переносе некоторого количества (например, 10%) самых здоровых хромосом в следующее поколение.

Существует еще один интересный механизм выбора, который называется *методом турнира*. Из популяции выбираются две или несколько хромосом, которые затем соревнуются за право попасть в следующее поколение. Побеждает та хромосома, здоровье которой выше. Турнир повторяется два раза, и в результате выбираются два родителя, которые переходят в следующее поколение.

Размер популяции

Размер популяции является весьма важным элементом генетического алгоритма. Если популяция слишком мала, генетического материала может не хватить для решения данной проблемы. Размер популяции также влияет на коэффициент применения операторов мутации и перекрестного скрещивания.

Генетические операторы

В предыдущих разделах были рассмотрены операторы мутации и перекрестного скрещивания. Существуют и другие операторы, с помощью которых можно решить данную проблему. Достаточно успешно показал себя оператор *инверсии* (inversion), который переставляет группу генов в хромосоме.

Другие механизмы

Создание популяции является интересной методикой, которая позволяет генетическому алгоритму найти подход к решению проблемы. Вместо того чтобы инициализировать популяцию со случайным набором хромосом, разработчик добавляет хромосомы, которые изначально представляют хорошие решения проблемы. Это может привести к быстрому схождению популяции (то есть потере разнообразия в генетическом фонде), поэтому при использовании подобной методики следует соблюдать повышенную осторожность.

Не менее интересно экспериментировать со схемами скрещивания, которые берут свое начало в садоводстве и разведении домашних животных. Р. Хольштейн (R. Hollstein) исследовал множество методов селекции (табл. 6.5).

Таблица 6.5. Методы селекции, исследованные Р. Хольштейном (таблица создана автором)

Метод	Описание
Тестирование потомства	От здоровья потомства зависит последующее скрещивание родителей
Индивидуальная селекция	Здоровье индивидуума определяет его последующий выбор как родителя
Выбор семьи	От здоровья семьи зависит использование ее членов в качестве родителей
Выбор в семье	Здоровье индивидуумов в семье определяет выбор родителей для скрещивания в семье
Комбинированная селекция	Комбинируются несколько вышеописанных методов

Последняя опция – это самостоятельная адаптация генетических операторов и/или коэффициентов их применения. Томас Бэк (Thomas Back) наблюдал интересные результаты при оценке коэффициентов мутации и перекрестного скрещивания хромосомы. Эта методика позволяет генетическому алгоритму не только найти верное решение поставленной задачи, но и применить правильные коэффициенты использования генетических операторов.

Вероятности

Вероятности применения генетических операторов имеют большое значение. Например, если оператор мутации имеет слишком большую вероятность применения, это приводит к разрушению полезного генетического материала популяции и переключению на обычный поиск в произвольном порядке. Если коэффициенты слишком малы, поиск решения данной проблемы может занять намного больше времени, чем нужно.

Если не используется самостоятельная адаптация (по Томасу Бэку), то правильные коэффициенты обычно находят путем эксперимента. Хорошая отправная точка – это установка коэффициента таким образом, чтобы к 70% выбранных

родителей применялось перекрестное скрещивание, а в другом случае производилась мутация.

В зависимости от того, какой метод кодировки был выбран для задачи, некоторые генетические операторы могут стать деструктивными. Поэтому для правильного использования генетических операторов необходимо хорошее понимание кодировки и эффектов использования генетических операторов.

Недостатки генетического алгоритма

Генетический алгоритм не лишен определенных недостатков. В этом разделе мы обсудим некоторые аспекты, которые следует учитывать при его применении.

Преждевременное схождение

Проблема преждевременного схождения связана с недостаточным разнообразием хромосом в популяции. Если большинство хромосом в популяции схожи между собой, то для селекции доступно меньше рабочего материала, и коэффициент увеличения здоровья снижается. Вы можете обнаружить эффект преждевременного схождения, сравнив среднее здоровье популяции с максимальным здоровьем. Если они слишком схожи, это значит, что произошло преждевременное схождение популяции.

Самой распространенной причиной преждевременного схождения является слишком маленький размер популяции. При увеличении размера популяции проблема легко решается. Другой причиной может быть алгоритм отбора, который вы применяете. При использовании метода элиты выбираются только хромосомы с самым высоким здоровьем, что приводит к сильному уменьшению размера популяции по сравнению с начальным. Если популяция слишком рано сошлась, и решение не было найдено, вам следует перезапустить алгоритм. При перезапуске в популяцию будет введен новый материал, в котором может не быть небольшого количества доминирующих хромосом.

Эпистазис

Эпистазисом называется внутренняя зависимость между переменными (генами), закодированными в хромосоме. Если ни один ген не связан с другими генами в хромосоме, считается, что эпистазис очень мал или не существует. Если гены зависят друг от друга, эпистазис высок и может создать проблемы для алгоритмов рекомбинирования.

Обычное решение этой проблемы состоит в том, чтобы сохранять гены (переменные), которые близко связаны друг с другом в хромосоме. При группировании зависимых генов существенно снижается вероятность того, что они будут разрушены при применении генетических операторов, например, перекрестного скрещивания.

Теорема «не бывает бесплатных обедов»

Теорема «не бывает бесплатных обедов» основывается на идее о том, что не существует совершенного метода оптимизации. Нельзя решить задачу с помощью любой кодировки, метода селекции и набора вероятностей, из всех имеющихся возможностей необходимо выбрать оптимальный способ с учетом особенностей поставленной задачи.

Другие области применения

Генетический алгоритм используется для решения многих задач оптимизации. Так как эффективность генетического алгоритма во многом зависит от представления решения, вы можете оптимизировать как числовые, так и символические задачи. Например, помимо простой функциональной оптимизации можно работать с такими символическими задачами, как задачи Ханойских Башен.

Генетические алгоритмы применяются для решения следующих проблем:

- создание дизайна с помощью компьютера;
- составление порядка решения задач;
- экономические задачи и задачи теории игр;
- другие задачи оптимизации.

В качестве составной части генетический алгоритм применялся в разработке, которая предназначена для вычисления абсолютного положения в пространстве при работе с телескопом. Таким образом было найдено соответствие между звездными четырехугольниками в каталоге и звездами в поле обзора телескопа, что позволило определить курс космического корабля.

Итоги

В этой главе рассматривался генетический алгоритм. После обзора принципов работы алгоритма подробно описывалось его выполнение – инициализация, оценка здоровья, селекция и рекомбинирование. Затем был представлен исходный код, который применяет алгоритм для вычисления последовательности инструкций, и с помощью этой программы было найдено решение нескольких уравнений. Поскольку алгоритму ничего не известно о самих уравнениях, то только при правильном решении он способен предлагать альтернативные варианты решения проблемы (с заданным набором инструкций), которые демонстрируют оптимизацию заданного алгоритма. В заключение рассказывалось, какие параметры генетического алгоритма можно настраивать, а также ряд трудностей, которые возникают при работе с данным алгоритмом.

Литература и ресурсы

1. Бэк Т. Взаимодействие коэффициента мутации, селекции и самостоятельной адаптации в генетическом алгоритме (Back T. The Interaction of Mutation Rate, Selection, and Self-Adaptation within a Genetic Algorithm. – Germany: University of Dortmund, 1992).
2. Коза Д. Web-сайт Genetic Programming Inc., http://www.genetic-programming.com.
3. Коза Д. Генетическое программирование: о компьютерном программировании с использованием естественного отбора (Koza J. Genetic Programming: On the Programming of Computers by Means of Natural Selection. – Cambridge, Mass.: MIT Press, 1992).
4. Холланд Д. Об эффективных системах адаптации (Holland J. Concerning efficient adaptive systems // Self-Organizing Systems.– Washington, D.C.: Spartan Books, pp. 215–230, 1962).
5. Холланд Д. Адаптация в естественных и искусственных системах (Holland J. Adaptation in Natural and Artificial Systems. Ann Arbor: The University of Michigan Press, 1975).
6. Хольштейн Р. Искусственная генетическая адаптация в компьютерных системах управления (Hollstein R. Artificial Genetic Adaptation in Computer Control Systems. Ph.D diss., University of Michigan, 1971).
7. Шаффер Р. Практическое руководство по генетическим алгоритмам (Shaffer R. Practical Guide to Genetic Algorithms, 1993). Доступно по адресу http://chemdiv-www.nrl.navy.mil/6110/6112/sensors/chemometrics/practga.html.

Глава 7. Искусственная жизнь

Искусственная жизнь (Artifical life) – это понятие, введенное Крисом Лангтоном (Chris Langton) для обозначения множества компьютерных механизмов, которые используются для моделирования естественных систем. Искусственная жизнь применяется для моделирования процессов в экономике, поведения животных и насекомых, а также взаимодействия различных объектов. В данной главе рассматривается теория построения искусственной жизни и модель, которая демонстрирует агентов, соревнующихся друг с другом в искусственной среде.

Введение

Искусственная жизнь представляет собой целую науку с множеством аспектов. Здесь рассматривается одно из ее направлений – *синтетическая наука о поведении* (Synthetic ethology). Ее очень четко описывает Брюс МакЛеннан (Bruce MacLennan):

«Синтетическая наука о поведении – это подход к изучению поведения животных, при котором простые синтетические организмы определенным образом действуют в синтетическом мире. Так как и мир, и организмы являются синтетическими, они могут быть сконструированы для особых целей, а именно для проверки определенных гипотез».

Искусственная жизнь может быть описана как теория и практика моделирования биологических систем. Разработчики, которые ведут исследования в данной сфере, надеются, что путем моделирования биологических систем мы сможем лучше понять, почему и как они работают. С помощью моделей разработчики могут управлять созданной средой, проверять различные гипотезы и наблюдать, как системы и среда реагируют на изменения.

Моделирование пищевых цепочек

Пищевая цепочка описывает иерархию живых организмов в экосистеме. Например, рассмотрим очень простую абстрактную пищевую цепочку, которая состоит из трех особей (рис. 7.1). В нижней части цепочки находятся растения. Они получают энергию из окружающей среды (дождя, почвы и солнца). Следующий уровень занимают травоядные животные, – для выживания они поедают растения. На верхней ступени находятся хищники. В этой модели хищники поедают травоядных

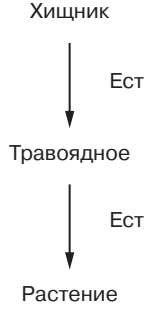


Рис. 7.1. Простая пищевая цепочка

животных, чтобы выжить. Если проигнорировать присутствие в среде мертвых травоядных и хищников, то цепочка будет выглядеть так, как показано на рис. 7.1.

Если рассматривать рис. 7.1 как график зависимости, видно, что между особями существует четко выраженный баланс. Что произойдет, если вдруг в результате засухи или по другой причине исчезнут все растения? При этом нарушится баланс выживания травоядных животных в среде, что приведет к сокращению их популяции. Это отразится на всей цепочке и повлияет на популяцию хищников. Данный баланс может моделироваться и изучаться в сфере искусственной жизни и науки о поведении.

Модель пищевой цепочки

Чтобы смоделировать простую пищевую цепочку, необходимо определить некоторые параметры: окружающую среду (физическое пространство, в котором взаимодействуют агенты), самих агентов (а также их восприятие и поведение в среде) и группу правил, которые определяют, как и когда происходит взаимодействие. Эти элементы будут описаны в следующих разделах.

Обзор

Как и описывалось выше, создаваемая модель будет состоять из среды и трех типов особей. Растения представляют собой неподвижный источник еды для травоядных животных. Травоядные животные являются мигрирующими агентами, которые определенным образом воспринимают окружающую среду и едят растения. Другими мигрирующими агентами в среде являются хищники, поедающие травоядных животных. Хищники могут есть только травоядных, а травоядные могут есть только растения. Если какой-либо агент живет в среде определенное время и не получает еды, он сам погибает от голода. Когда агент поглощает достаточное количество пищи, он может размножаться. Таким образом, в среде создается новый агент определенного типа. Происходит эволюция, при которой мутирует мозг агента (простая нейронная сеть).

Важно отметить, что агенты изначально не знают, как нужно выживать в среде. Они не знают, что поедание пищи позволит им прожить дольше. Также они не знают, что должны избегать тех, кто их ест. Агенты должны освоить все эти знания посредством эволюции.

В следующих разделах подробно рассматриваются элементы модели.

Окружающая среда

Агенты живут в мире, построенном по принципу сетки, грани которой соединены по аналогии с тороидом. Если агент перемещается за грань в определенном направлении, он появляется на другой стороне (рис. 7.2).

Растения занимают уникальные ячейки в среде, однако несколько агентов могут занимать одну и ту же ячейку (травоядное животное и/или хищник).

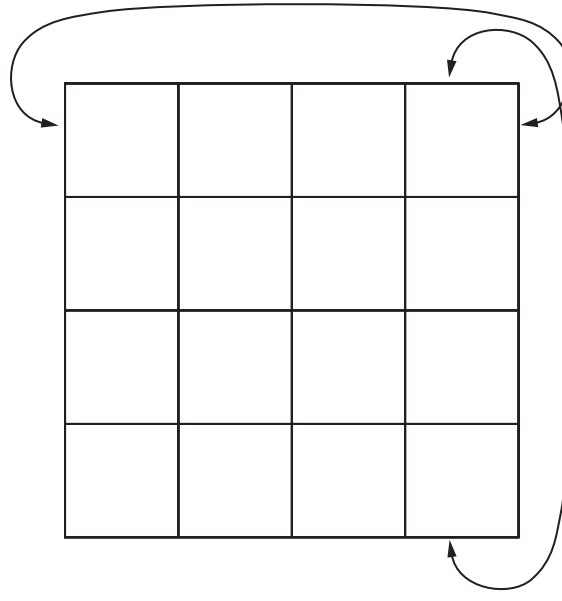


Рис. 7.2. Мир в виде сетки, построенной по принципу тороида, который будет использоваться для моделирования пищевой цепочки

Анатомия агента

Агент является генетической особью. Он может быть только определенного типа (травоядным или хищником), но метод изучения окружающей среды и образ действий для всех агентов одинаковы (рис. 7.3). Агента можно рассматривать как простую систему с набором входов (его ощущением мира), реакций на окружающий мир (его мозгом) и действий.

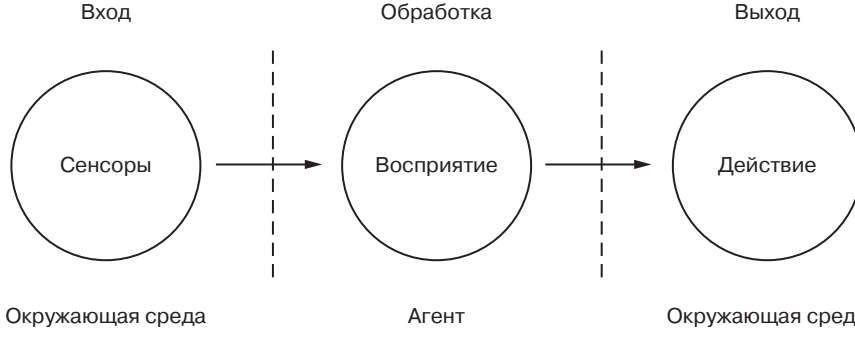


Рис. 7.3. Модель систем с агентами

Как показано на рис. 7.3, агент состоит из трех отдельных частей. Это сенсоры, ощущение (определение того, какое действие выбрать) и действие. Обратите внимание, что модель агента реагирует на окружающую среду. Агенты не могут планировать и обучаться. Даже в такой простой модели обучение происходит по принципу, который называется эволюцией Ламарка. При воспроизводстве характеристики родителя передаются потомству.

Примечание *Жан-Баптист Ламарк (1744–1829) предложил альтернативный механизм эволюции, который отличается от механизма, исследованного Чарльзом Дарвином. Ламарк утверждал, что вместо процесса естественного отбора, направляющего постепенную эволюцию особи, процессом эволюции управляет наследственность.*

Сенсоры

Агенты могут чувствовать, что происходит вокруг них в среде. Однако агент не видит всю среду, он реагирует только на группу ячеек вокруг него (рис. 7.4).

Локальная среда, которую может чувствовать агент, разделена на четыре отдельные области. Самая ближняя к агенту область называется *областью близости*, и это та область, в которой агент может действовать (скажем, съесть другой объект). Область впереди агента (5 ячеек) именуется *фронт*, а две ячейки слева и справа – *слева* и *справа*.

Агент может определять вид объектов в поле зрения. Поэтому для четырех областей предлагаются три числа, которые позволяют идентифицировать типы

Рис. 7.4. Область предчувствия агента. Агент «смотрит» на север

имеющихся объектов (растения, травоядные и хищники), то есть всего двенадцать входов.

Активаторы

Агент может выполнять ограниченное число действий в среде: перейти на одну ячейку (в заданном направлении), повернуться налево или направо или съесть объект, который находится в области «близости». Действие, которое производит агент, определяется его мозгом при оценке входов, полученных на уровне сенсоров.

Мозг агента

Мозг агента может быть одной из многочисленных компьютерных конструкций. Существующие симуляции искусственной жизни используют принцип конечных автоматов, системы классификации или нейронные сети. Чтобы сохранить аналогию с биологической мотивацией, в данном случае при моделировании используется простая нейронная сеть, построенная по принципу «победитель получает все» (см. главу 5), в качестве системы поведения агента. На рис. 7.5 показана полная сеть.

Вспомните, что входы сенсоров отображают количество агентов, которые находятся в поле зрения в определенной области. После того как все входы были получены из среды, программа «продвигает» их через сеть к выходам. Это делается с помощью уравнения 7.1:

$$o_j = b_j + \Sigma^n_{i=0}\, u_i w_{i,j} \qquad (7.1)$$

Другими словами, для каждой входной ячейки (o_j) сети суммируются результаты входных ячеек (u_i), которые умножаются на веса соединений от входных ячеек к выходным ($w_{i,j}$). Также добавляется смещение для выходной ячейки. В результате в выходных ячейках будет получен набор значений, которые затем используются элементом действия агента.

Начальные веса нейронной сети агента выбираются случайным образом. В результате воспроизведения веса должны быть настроены для выживания в среде.

Выбор действия агента

Вспомните, что агент может выполнять одно действие из четырех возможных, как указано выходными ячейками нейронной сети. Процесс выбора действия заключается в поиске выходной ячейки с наибольшим значением и выполнении соответствующего действия. Это и есть принцип «победитель получает все» применительно к сети. После выполнения действия окружающая среда изменяется (если на нее воздействовали), и процесс продолжается.

Энергия и метаболизм

Чтобы выжить в окружающей среде, агентам нужна адекватная энергия. Если внутренняя энергия агента становится равна нулю, агент умирает. Агенты создают энергию, съедая другие объекты в среде. Агент может съесть только тот объект, который допускается пищевой цепочкой. Хищники могут есть только травоядных,

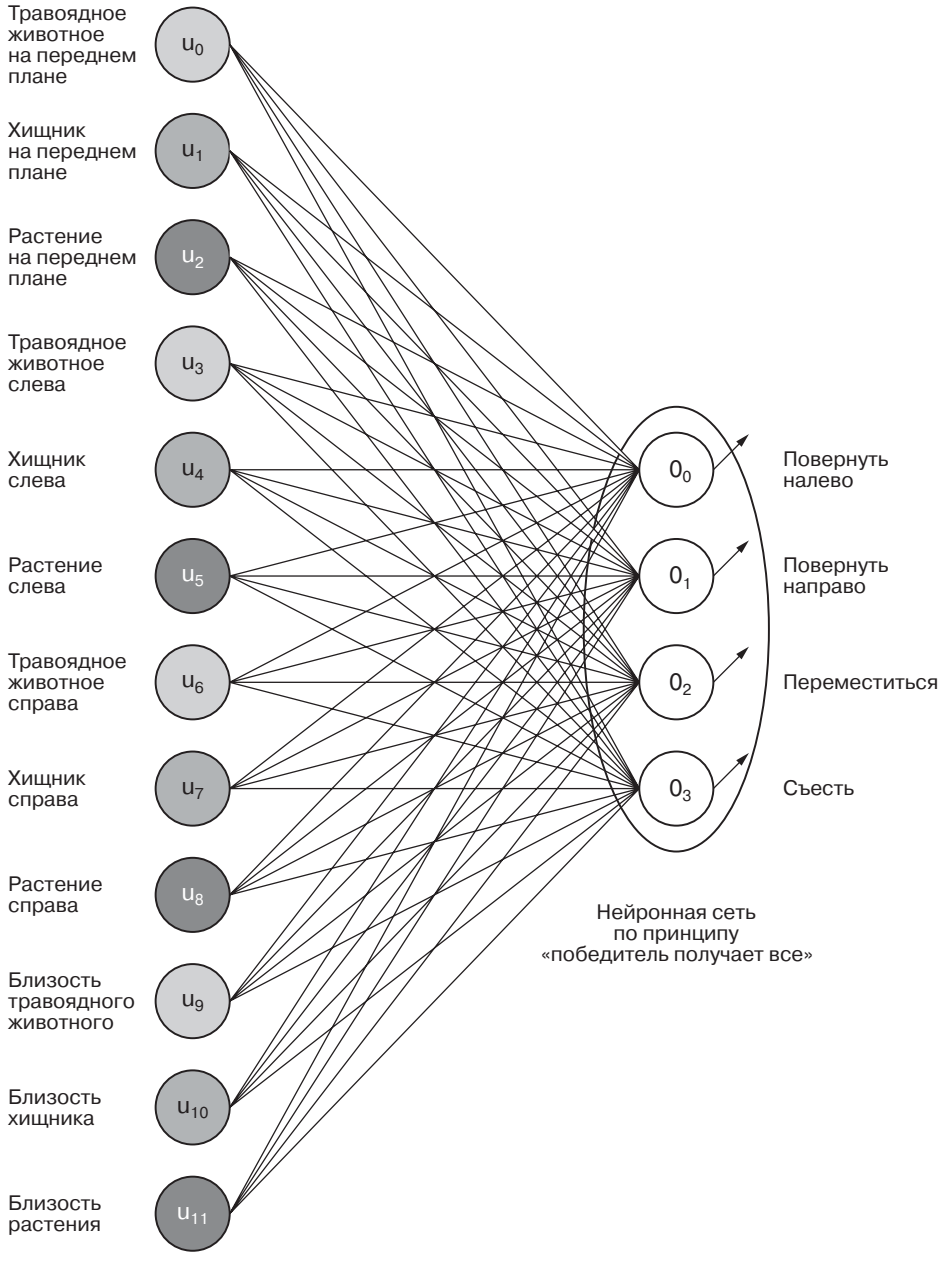


Рис. 7.5. «Победитель получает все»: нейронная сеть в качестве «мозга» агента

а травоядные – только растения. Агенты также обладают метаболизмом, то есть коэффициентом поглощения энергии, который позволяет им сохранить жизнь. За

каждую единицу времени хищники поглощают одну единицу энергии, а травоядные – две единицы. Это значит, что для сохранения жизни травоядным нужно съедать в два раза больше пищи, чем хищникам. Хотя хищникам не требуется так много еды, им еще нужно ее найти. Травоядные имеют преимущество, которое заключается в том, что их пища не перемещается по среде. Тем не менее им все равно нужно отыскать свою пищу.

Воспроизведение

Если агент поглощает достаточное количество пищи, чтобы достичь показателя 90% от максимального уровня энергии, он допускается к участию в воспроизведении. Воспроизведение позволяет агентам, которые смогли выжить в окружающей среде, создать потомство (естественный отбор). При создании потомства агенты изменяют веса своих нейронных сетей посредством произвольной мутации. Обучение в среде недоступно, однако то, что агент может воспроизводить себя, означает, что его нейронная сеть будет передана его ребенку. Это повторяет принцип эволюции Ламарка, поскольку характеристики агента передаются его потомству (ребенок наследует нейронную сеть своего родителя).

Воспроизведение имеет последствия: родитель и ребенок разделяют имеющуюся энергию родителя (энергия родителя делится пополам). При этом агент не сможет непрерывно воспроизводить себя.

Смерть

Агент может умереть двумя способами: либо он не сможет найти пищу и умрет от голода, либо его съест агент, который стоит выше в пищевой цепочке. В любом случае мертвый агент удаляется из модели.

Соревновательность

Во время симуляции происходит своеобразное соревнование. Хищники постепенно разрабатывают нейронные сети, которые подходят для обнаружения и поедания травоядных животных. В то же время травоядные совершенствуют нейронные сети, которые помогают находить растения в среде и избегать хищников. Хотя эти стратегии видны при изучении симуляции, анализ изменений в нейронных сетях позволяет сделать интересные выводы. Чтобы лучше понять мотивацию агентов, мы поговорим об этих изменениях в следующих разделах.

Пример итерации

Рассмотрим пример итерации для агента, связанной с выбором действия. В данном примере будет описано травоядное животное, которое развивается в ходе симуляции. Этому агенту удалось выжить в среде в течение 300 единиц времени, поскольку он находил и поедал растения, а также избегал хищников. На рис. 7.6 представлена нейронная сеть этого агента.

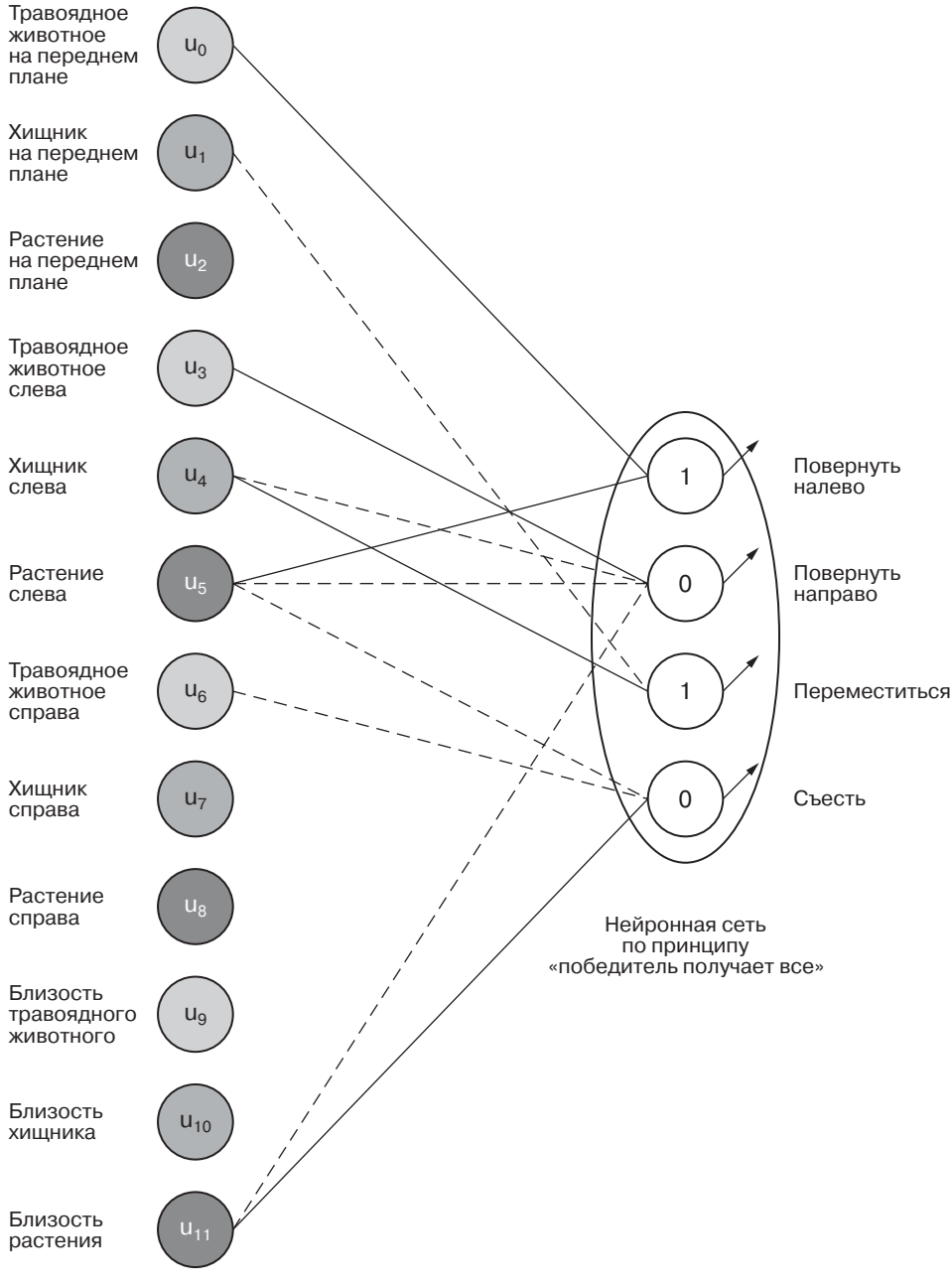


Рис. 7.6. Нейронная сеть травоядного животного

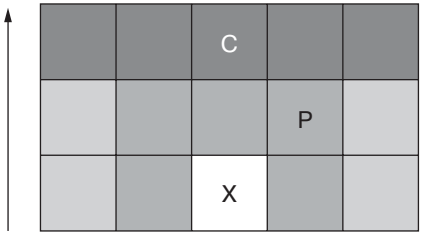
Сплошные линии в нейронной сети являются возбуждающими соединениями, а пунктирные линии – запретными соединениями. В выходных ячейках

находятся смещения, которые применяются к каждой выходной ячейке при ее активации. Возбуждающее соединение существует для действия «есть» при условии, что в области «близости» находится растение (растение можно съесть только в том случае, если оно находится вблизи от агента). Не менее интересно запретное соединение для действия «движение», которое срабатывает, когда в области «фронт» находится хищник. Это еще одно важное условие для выживания травоядного.

Действия агента не формируются только одним соединением. Вместо этого срабатывает действие с наибольшим весом (на основании комбинации входов сенсоров). Рассмотрим несколько итераций для травоядного животного, описанных нейронной сетью на рис. 7.6.

Вспомните (уравнение 7.1), что вектор весов (для определенного действия) умножается на вектор входов, а затем добавляется смещение.

В первом примере травоядное животное рассматривается в ситуации, которая показана на рис. 7.7. Различные зоны закрашены для удобства (как на рис. 7.4). На этой сцене символ «X» обозначает положение травоядного (его точку на сцене). Растение расположено в области «близости», а хищник – в области «фронт».




```
Влево       = 1 + 0 = 1
Вправо      = 0 + -1 = -1
Двигаться   = 1 + -1 = 0
Съесть      = 0 + 1 = 1
```

```
                  HF CF PF HF CL PL HR CR PR HR CR PP
веса_влево      = {1, 0,  0, 0,  0, 1,  0, 0,  0, 0,  0, 0}
веса_вправо     = {0, 0,  0, 0, -1,-1,  0, 0,  0, 0,  0,-1}
веса_двигаться  = {0, -1, 0, 0,  1, 0,  0, 0,  0, 0,  0, 0}
веса_съесть     = {1, 0,  0, 0,  0,-1, -1, 0,  0, 0,  0, 1}

смещения_влево      = 1
смещения_вправо     = 0
смещения_двигаться  = 1
смещения_съесть     = 0

входы           = {0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1}
```

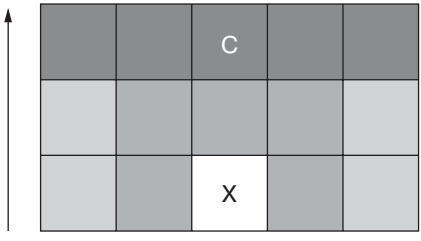
Рис. 7.7. Травоядное животное во время t_0

Сначала необходимо оценить сцену. Подсчитывается количество объектов каждого типа во всех четырех зонах. Как показано на рис. 7.7, веса и входы помечены по принципу тип/зона (HF обозначает «фронт для травоядного», CF – «фронт

для хищника», PP – «растение в области «близости» и т.д.). В этом примере входной вектор имеет значения, отличные от нуля, только в двух элементах: хищник в области «фронт» и растение в области «близости».

Чтобы определить, какое действие выбрать, нужно умножить входной вектор на вектор весов для определенного действия, а затем добавить относительное смещение. Этот процесс показан на рис. 7.7. Выбор поведения определяется действием с наибольшим значением. В данном примере программа берет наибольшее значение, которое появилось последним. При этом выполняется действие «есть» (нужное действие для текущей сцены).

Растение съедено, и оно исчезает со сцены. Окружающая среда изменилась, и перед травоядным животным предстает сцена, показанная на рис. 7.8.



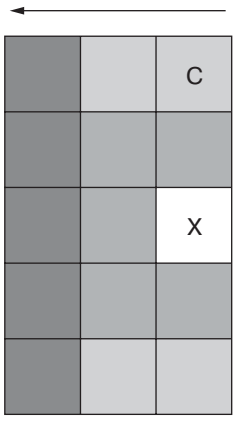

```
Влево       = 1 + 0 = 1
Вправо      = 0 + 0 = 0
Двигаться   = 1 + -1 = 0
Съесть      = 0 + 0 = 0
```

```
                  HF CF PF HF CL PL HR CR PR HR CR PP
веса влево      = {1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0}
веса вправо     = {0, 0, 0, 0,-1,-1, 0, 0, 0, 0, 0,-1}
веса двигаться  = {0, -1,0, 0, 1, 0, 0, 0, 0, 0, 0, 0}
веса съесть     = {1, 0, 0, 0, 0,-1,-1, 0, 0, 0, 0, 1}

смещения влево     = 1
смещения вправо    = 0
смещения двигаться = 1
смещения съесть    = 0

входы       = {0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

Рис. 7.8. Травоядное животное во время T_1

Сцена оценивается заново, причем растения больше нет, а хищник остался. Это видно по входам (изменения по сравнению с предыдущей итерацией показаны полужирным шрифтом). Программа еще раз рассчитывает выходные ячейки нейронной сети, умножая значения сигналов входного вектора на соответствующий вектор весов. В этом случае наибольшее значение ассоциируется с действием «влево». Учитывая данную ситуацию, это и есть наилучшее действие.

Наконец, на рис. 7.9 представлена последняя итерация. Обратите внимание, что поле зрения агента изменилось, поскольку в предыдущей итерации он выбрал другое направление движения. С учетом изменений в сцене были




```
Влево       = 1 + 0 = 1
Вправо      = 0 + 0 = 0
Двигаться   = 1 + 0 = 1
Съесть      = 0 + 0 = 0
```

```
                  HF CF PF HF CL PL HR CR PR HR CR PP
веса влево      = {1, 0,  0, 0, 0, 1, 0, 0, 0, 0, 0, 0}
веса вправо     = {0, 0,  0, 0,-1,-1, 0, 0, 0, 0, 0,-1}
веса двигаться  = {0, -1,0, 0, 1, 0, 0, 0, 0, 0, 0, 0}
веса съесть     = {1, 0,  0, 0, 0,-1,-1, 0, 0, 0, 0, 1}

смещения влево      = 1
смещения вправо     = 0
смещения двигаться  = 1
смещения съесть     = 0

входы = {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0}
```

Рис. 7.9. Травоядное животное во время T_2

переопределены и входы. Теперь хищник находится в области «справа», а не в области «фронт».

При расчете выходных ячеек получается, что действие «идти» имеет самое большое значение и, что более важно, является последним. Поведение травоядного позволяет ему отыскивать и поедать пищу, а также избегать хищника, который находится в поле зрения. Демонстрация нейронной сети травоядного наглядно показывает, почему оно смогло прожить в среде в течение длительного времени.

Совет *Исходный код программы симуляции искусственной жизни находится в архиве, который можно загрузить с сайта издательства «ДМК Пресс» www.dmk.ru.*

Исходный код

Исходный код программы моделирования искусственной жизни очень прост. Сначала рассмотрим те структуры, которые описывают окружающую среду, агентов и другие объекты.

В листинге 7.1 представлена структура данных, описывающая свойства агента. Большинство полей этой структуры говорят сами за себя: поле `type` определяет тип агента как травоядное животное или хищника, `energy` показывает энергию агента, `age` – возраст агента в «прожитых» итерациях, а поле `generation` – значение, характеризующее агента по количеству его предков, то есть поколение, к которому принадлежит агент.

Положение агента (заданное типом `locType`) показывает его координаты в среде по оси x и y. Массив `inputs` определяет значения входов в нейронную сеть на этапе восприятия окружающей среды. Массив `actions` представляет собой выходной слой нейронной сети, который задает следующее действие агента. Наконец, массивы `weight_oi` (значение веса от выхода до входа) и `biasso` показывают веса и смещения для выходного слоя сети.

Листинг 7.1. Типы и символика агентов

```
typedef struct {
  short type;
  short energy;
  short parent;
  short age;
  short generation;
  locType location;
  unsigned short direction;
  short inputs[MAX_INPUTS];
  short weight_oi[MAX_INPUTS * MAX_OUTPUTS];
  short biaso[MAX_OUTPUTS];
  short actions[MAX_OUTPUTS];
} agentType;

#define TYPE_HERBIVORE   0
#define TYPE_CARNIVORE   1
#define TYPE_DEAD        -1

typedef struct {
  short y_offset;
  short x_offset;
} locType;
```

Входной вектор задает входы как объект и область (например, травоядное животное и область «фронт»). Чтобы дать агенту возможность различать эти элементы, для каждого элемента определяется отдельный вход в нейронную сеть. Выходы также связаны с отдельными выходными ячейками выходного вектора, который представляет одно действие. В листинге 7.2 показаны символические константы для входных и выходных ячеек.

Листинг 7.2. Определения входной ячейки сенсора и выходной ячейки действия

```
#define HERB_FRONT          0
#define CARN_FRONT          1
```

```
#define PLANT_FRONT         2
#define HERB_LEFT           3
#define CARN_LEFT           4
#define PLANT_LEFT          5
#define HERB_RIGHT          6
#define CARN_RIGHT          7
#define PLANT_RIGHT         8
#define HERB_PROXIMITY      9
#define CARN_PROXIMITY      10
#define PLANT_PROXIMITY     11

#define MAX_INPUTS          12

#define ACTION_TURN_LEFT    0
#define ACTION_TURN_RIGHT   1
#define ACTION_MOVE         2
#define ACTION_EAT          3

#define MAX_OUTPUTS         4
```

Окружающая среда агента отображается в виде трехмерного куба. Для агентов доступны три плоскости, причем каждую плоскость занимает объект одного типа (растение, травоядное или хищник). Мир агента по-прежнему рассматривается как двумерная сетка, а три измерения применяются для более эффективного подсчета присутствующих объектов. В листинге 7.3 показаны типы данных и константы, которые используются для отображения среды.

Листинг 7.3. Определения входной ячейки сенсора и выходной ячейки действия

```
#define HERB_PLANE    0
#define CARN_PLANE    1
#define PLANT_PLANE   2

#define MAX_GRID      30

/* Среда имеет 3 измерения (независимые измерения для растений,
 * травоядных и хищников)
 */
int landscape[3][MAX_GRID][MAX_GRID];

#define MAX_AGENTS    36
#define MAX_PLANTS    35

agentType agents[MAX_AGENTS];
int agentCount = 0;

plantType plants[MAX_PLANTS];
int plantCount = 0;
```

Размер сетки, количество существующих агентов и растений – это параметры, которые вы можете изменять при решении разных задач. Заголовочный файл `common.h` включает раздел с параметрами, которые могут быть переопределены.

Наконец, рассмотрим последнюю группу макросов, которые представляют часто используемые функции, связанные с генерацией случайных чисел (листинг 7.4).

Листинг 7.4. Функции, которые используются для симуляции, представленные в виде макросов

```
#define getSRand() ((float)rand() / (float)RAND_MAX)
#define getRand(x) (int)((x) * getSRand())

#define getWeight()   (getRand(9)-1)
```

Функция `getSRand` возвращает случайное число от 0 до 1, а функция `getRand` –число от 0 до –1. Функция `getWeight` возвращает значение веса, которое используется для нейронных сетей агента. Она также используется для генерации смещения, которое применяется при расчетах в выходных ячейках.

Теперь обсудим исходный код собственно симуляции. Начнем с упрощенной функции `main`, из которой была удалена обработка параметров командной строки и сбор статистических данных.

Функция `main` инициализирует модель, а затем выполняет в цикле итерации, количество которых указано в заголовочном файле посредством константы `MAX_STEPS`. Функция `simulate` является первой вводной точкой симуляции, после которой начинают свою жизнь агенты и окружающая среда (листинг 7.5).

Листинг 7.5. Функция `main` для симуляции искусственной жизни

```
int main ( int argc, char *argv[])
{
   int i;

   /* Инициализация генератора случайных чисел */
   srand( time(NULL) );

   /* Инициализация модели */
   init();

   /*Главный цикл модели */
   for (i = 0; i < MAX_STEPS ; i++) {

   /* Выполнение одного действия для каждого агента */
   simulate();
}
```

```
    return 0
}
```

Функция `init` инициализирует среду и объекты в ней (растения, травоядных и хищников). Обратите внимание, что при инициализации агентов вводится тип каждого агента. Это делается для того, чтобы функция `initAgent` могла выбрать соответствующий алгоритм действий. После ввода типа агента функция `initAgent` получает информацию о том, с каким агентом она работает, и начинает действовать соответствующим образом (листинг 7.6).

Листинг 7.6. Использование функции `init` для инициализации модели

```
void init( void )
{

  /* Инициализация мира */
  bzero( (void *)landscape, sizeof(landscape) );

  bzero( (void *)bestAgent, sizeof(bestAgent) );

  /* Инициализация измерения растений */
  for (plantCount = 0 ; plantCount < MAX_PLANTS ; plantCount++) {
    growPlant( plantCount );
  }
  if (seedPopulation == 0) {

    /* Инициализация агентов случайным образом */
    for (agentCount = 0 ; agentCount < MAX_AGENTS ; agentCount++) {
      if (agentCount < (MAX_AGENTS / 2)) {
        agents[agentCount].type = TYPE_HERBIVORE;
      } else {
        agents[agentCount].type = TYPE_CARNIVORE;
      }

      initAgent( &agents[agentCount] );

    }


}
```

Сначала инициализируется плоскость растений. Для этого создаются растения в количестве, заданном константой `MAX_PLANTS`. Реализация этого действия возложена на функцию `growPlants` (листинг 7.7). Далее инициализируются агенты. Чтобы создать максимально допустимое количество агентов (согласно константе `MAX_AGENTS`), каждый раз резервируется половина пространства. Сначала инициализируются травоядные, а затем хищники. Инициализация агентов обеспечивается функцией `initAgent` (листинг 7.8).

Функция `growPlant` находит пустое место на плоскости растений и помещает в эту ячейку новое растение (листинг 7.7). Она также гарантирует, что в ячейке пока нет растения (это позволит контролировать количество растений в среде).

Листинг 7.7. Использование функции `growPlant` для инициализации плоскости растений

```
void growPlant( int i )
{
  int x,y;

  while (1) {

     /* Получить координаты агента случайным образом */
     x = getRand(MAX_GRID); y = getRand(MAX_GRID);

     /* Пока в этой точке нет растений */
     if (landscape[PLANT_PLANE][y][x] == 0) {

          /* Поместить растение в среду */
          plants[i].location.x = x;
          plants[i].location.y = y;
          landscape[PLANT_PLANE][y][x]++;
          break;

   }

  }

  return;
}
```

Далее инициализируются плоскости агентов (листинг 7.8). Программа проходит в цикле по массиву, хранящему описания агентов (листинг 7.6). Вспомните, что тип агента уже был задан. Сначала инициализируется поле `energy` для агента. Энергия устанавливается в значение, равное половине от максимума, чтобы при достижении определенного уровня энергии агент смог воспроизвестись. Когда уровень энергии для агента задается равным половине от максимума, это значит, что агент должен быстро найти еду в среде, чтобы быть допущенным к воспроизведению. Кроме того, для нового агента инициализируются возраст и поколение. В переменной `agentTypeCounts` подсчитывается количество агентов соответствующего типа. Это гарантирует, что в модели сохранится начальное соотношение 50/50 между травоядными и хищниками. Далее с помощью функции `findEmptySpot` определяется начальное положение агента (листинг 7.8). Она находит пустую ячейку в заданной плоскости (определенной типом агента) и сохраняет координаты агента в его структуре. Наконец, инициализируются веса и смещения для нейронной сети агента.

Листинг 7.8. Функции `initAgent`, предназначенная для инициализации агентов

```
void initAgent( agentType *agent )
{
  int i;

  agent->energy = (MAX_ENERGY / 2);
  agent->age = 0;
  agent->generation = 1;

  agentTypeCounts[agent->type]++;

  findEmptySpot( agent );

     for (i = 0 ; i < (MAX_INPUTS * MAX_OUTPUTS) ; i++) {
      agent->weight_oi[i] = getWeight();
     }

     for (i = 0 ; i < MAX_OUTPUTS ; i++) {
      agent->biaso[i] = getWeight();
     }

  }
  return;
}

  void findEmptySpot( agentType *agent )
{
  agent->location.x = -1;
  agent->location.y = -1;

  while (1) {

     /* Получить координаты агента случайным образом */
     agent->location.x = getRand(MAX_GRID);
     agent->location.y = getRand(MAX_GRID);

     /* Если ячейка пуста, то прервать цикл генерации координат */
     if (landscape [agent->type]
               [agent->location.y][agent->location.x] == 0)
      break;

  }

   /* Сгенерировать направление движения агента */
   agent->direction = getRand(MAX_DIRECTION);
   landscape[agent->type][agent->location.y][agent->location.x]++;

   return;
}
```

Обратите внимание, что в функции `findEmptySlot` окружающая среда представлена в виде чисел. При этом записывается, присутствует ли объект в определенной ячейке сетки или нет. Когда объекты умирают или их съедают, переменная `landscape` изменяется, чтобы идентифицировать удаление объекта.

Теперь, после рассмотрения инициализации модели, перейдем собственно к самой симуляции. Вспомните, что функция `main` (листинг 7.5) вызывает функцию `simulate`, чтобы начать симуляцию. Функция `simulate` (листинг 7.9) позволяет каждому агенту выполнять одно действие в окружающей среде за один вызов. Вспомните, что сначала создаются травоядные, а потом – хищники. Это дает травоядным небольшое преимущество, но поскольку им приходится противостоять и голоду, и хищникам, такое преимущество лишь немного выравнивает шансы агентов.

Листинг 7.9. Функция `simulate`

```
void simulate( void )
{
  int i, type;

  /* Первыми действуют травоядные */

  for (type = TYPE_HERBIVORE ; type <= TYPE_CARNIVORE ; type++) {

    for (i = 0 ; i < MAX_AGENTS ; i++) {

      if (agents[i].type == type) {

        simulateAgent( &agents[i] );

      }

    }

  }

}
```

Функция `simulate` (листинг 7.9) вызывает функцию `simulateAgent` для просчета и выполнения одного действия агента. Она может быть разбита на четыре логические части. Это восприятие, обработка полученных данных об окружающей среде, выбор действия и проверка энергии агента.

Алгоритм восприятия является, вероятно, самым сложным этапом в симуляции. Вспомните (рис. 7.4), что поле зрения агента определяется направлением его движения и состоит из четырех отдельных областей (фронт, близость, слева и справа). Чтобы агент чувствовал среду, ему сначала необходимо идентифицировать координаты сетки, которые составляют его поле зрения (на основании направления движения агента), а затем разбить данную область на четыре отдельные зоны. Этот процесс отражается в команде переключения функции

simulateAgent (листинг 7.11). Здесь определяется направление, в котором смотрит агент. Каждый вызов функции percept суммирует объекты в определенной зоне. Обратите внимание, что при каждом вызове (HERB_<zone>) отображается первая плоскость для зоны (сначала травоядное, затем хищник и, наконец, растение).

При вызове функции percept в нее передаются текущие координаты агента, из массива inputs выбираются нужные данные о значениях на входах нейронной сети агента, а также список координат offsets и их смещение. Обратите внимание, что если агент смотрит на север, то в функцию передается набор координат north<zone>, а если агент смотрит на юг, то передается тот же набор координат, но со смещением –1. Этот процесс аналогичен и для области west<zone>. Смещения координат в каждой зоне определяются выбранным направлением, но они могут быть изменены на координаты противоположного направления. Чтобы лучше понять последнее утверждение, рассмотрим смещения координат в листинге 7.10.

Листинг 7.10. Смещения координат для суммирования объектов в поле зрения агента

```
const offsetPairType northFront[]=
      {{-2,-2}, {-2,-1}, {-2,0}, {-2,1}, {-2,2}, {9,9}};
const offsetPairType northLeft[]={{0,-2}, {-1,-2}, {9,9}};
const offsetPairType northRight[]={{0,2}, {-1,2}, {9,9}};
const offsetPairType northProx[]=
      {{0,-1}, {-1,-1}, {-1,0}, {-1,1}, {0,1}, {9,9}};

const offsetPairType westFront[]=
      {{2,-2}, {1,-2}, {0,-2}, {-1,-2}, {-2,-2}, {9,9}};
const offsetPairType westLeft[]={{2,0}, {2,-1}, {9,9}};
const offsetPairType westRight[]={{-2,0}, {-2,-1}, {9,9}};
const offsetPairType westProx[]=
      {{1,0}, {1,-1}, {0,-1}, {-1,-1}, {-1,0}, {9,9}};
```

Здесь представлены два набора координат для векторов смещения, один для севера и один для запада. Рассмотрим в качестве примера вектор northRight. Предположим, что агент имеет координаты <7,9> в среде (используя систему координат <x,y>). Используя вектор northRight в качестве смещения координат, программа рассчитывает две новые пары координат – <7,11> и <6,11> (координаты <9,9> представляют конец списка). Данные координаты отображают два положения в правой зоне для агента, который смотрит на север. Если бы агент смотрел на юг, программа бы инвертировала координаты northRight перед тем, как добавить их к текущему положению. В результате получилось бы следующее: <7,7> и <8,7>. Эти координаты представляют два положения в правой зоне при условии, что агент смотрит на юг.

Изучив пары координат при смещении, продолжим обсуждение функции simulateAgent (листинг 7.11).

Листинг 7.11. Функция `simulateAgent`

```
void simulateAgent( agentType *agent )
{
  int x, y;
  int out, in;
  int largest, winner;

/* Сокращаем имена переменных */
  x = agent->location.x;
  y = agent->location.y;

  /* Вычисление значений на входе в нейронную сеть агента */
  switch( agent->direction ) {

    case NORTH:
      percept( x, y, &agent->inputs[HERB_FRONT], northFront, 1 );
      percept( x, y, &agent->inputs[HERB_LEFT], northLeft, 1 );
      percept( x, y, &agent->inputs[HERB_RIGHT], northRight, 1 );
      percept( x, y, &agent->inputs[HERB_PROXIMITY], northProx, 1);
      break;

    case SOUTH:
      percept( x, y, &agent->inputs[HERB_FRONT], northFront, -1 );
      percept( x, y, &agent->inputs[HERB_LEFT], northLeft, -1 );
      percept( x, y, &agent->inputs[HERB_RIGHT], northRight, -1 );
      percept( x, y, &agent->inputs[HERB_PROXIMITY], northProx, -1 );
      break;

    case WEST:
      percept( x, y, &agent->inputs[HERB_FRONT], westFront, 1 );
      percept( x, y, &agent->inputs[HERB_LEFT], westLeft, 1 );
      percept( x, y, &agent->inputs[HERB_RIGHT], westRight, 1 );
      percept( x, y, &agent->inputs[HERB_PROXIMITY], westProx, 1 );
      break;

    case EAST:
      percept( x, y, &agent->inputs[HERB_FRONT], westFront, -1 );
      percept( x, y, &agent->inputs[HERB_LEFT], westLeft, -1 );
      percept( x, y, &agent->inputs[HERB_RIGHT], westRight, -1 );
      percept( x, y, &agent->inputs[HERB_PROXIMITY], westProx, -1 );
      break;

  }

  /* Вычисление в сети */
  for ( out = 0 ; out < MAX_OUTPUTS ; out++ ) {

    /* Инициализация входной ячейки сложением */
```

```
  agent->actions[out] = agent->biaso[out];
  /* Перемножаем значения на входе выходной ячейки
   * на соответствующие веса */
  for ( in = 0 ; in < MAX_INPUTS ; in++ ) {
   agent->actions[out] +=
      ( agent->inputs[in] *
        agent->weight_oi[(out * MAX_INPUTS)+in] );
  }
}
largest = -9;
winner = -1;

/* Выбор ячейки с максимальным значением (победитель
 * получает все)
 */
for ( out = 0 ; out < MAX_OUTPUTS ; out++ ) {
  if (agent->actions[out] >= largest) {
    largest = agent->actions[out];
    winner = out;
  }
}

/* Выполнение выбранного действия */
switch( winner ) {

  case ACTION_TURN_LEFT:
  case ACTION_TURN_RIGHT:
    turn( winner, agent );
    break;

  case ACTION_MOVE:
    move( agent );
    break;

  case ACTION_EAT:
    eat( agent );
    break;

}

/* Вычитаем "потраченную" энергию */
if (agent->type == TYPE_HERBIVORE) {
  agent->energy -= 2;
} else {
  agent->energy -= 1;
}

/* Если энергия агента меньше или равна нулю - агент умирает.
```

```
     * В противном случае проверяем, не является ли этот агент
     * самым старым
     */
    if (agent->energy <= 0) {
      killAgent( agent );
    } else {
      agent->age++;
      if (agent->age > agentMaxAge[agent->type]) {
        agentMaxAge[agent->type] = agent->age;
        agentMaxPtr[agent->type] = agent;
      }
    }

    return;
}
```

Обсудив процесс восприятия агентом окружающей среды, продолжим изучение трех других частей функции `simulateAgent`. Следующий этап заключается в том, чтобы «провести» переменные `inputs`, полученные на предыдущей стадии, в выходные ячейки нейронной сети агента. Этот процесс осуществляется с помощью уравнения 7.1. Результатом является набор значений выходных ячеек, которые рассчитаны на основании входных сигналов с использованием весов соединений между нейронами в сети. Затем (базируясь на том, какая выходная ячейка имеет наибольшее значение) программа выбирает действие, которое будет осуществлено агентом, по принципу «победитель получает все». Для выполнения действия используется оператор `case`. Как показано в листинге 7.11, для выбора доступны следующие действия: `ACTION_TURN_LEFT`, `ACTION_TURN_RIGHT`, `ACTION_TURN_MOVE` и `ACTION_EAT`.

Последний этап симуляции агента – это проверка его энергии. На каждом этапе агент теряет часть энергии (количество потерянной энергии различно для травоядных и хищников). Если энергия агента падает до нуля, он умирает от голода и выбывает из симуляции. Таким образом, если агент выживает, его возраст увеличивается, а противном случае вызывается функция `killAgent`, которая убирает агента из модели.

Теперь рассмотрим функции, используемые функцией `simulateAgent`, в том порядке, в котором они вызываются (`percept`, `turn`, `move`, `eat` и `killAgent`).

При предыдущем обсуждении вам могло показаться, что функция `percept` очень сложна, однако, прочитав листинг 7.12, вы поймете, что это не так. Дело в том, что большую часть функциональности обеспечивает структура данных; код просто следует структуре данных для получения нужного результата.

Листинг 7.12. Функция `percept`

```
void percept( int x, int y, short *inputs,
              const offsetPairType *offsets, int neg )
{
  int plane, i;
  int xoff, yoff;
```

```
/* Цикл по слоям "мира" */
for (plane = HERB_PLANE ; plane <= PLANT_PLANE ; plane++) {
/* Инициализация входов */
inputs[plane] = 0;
i = 0;

/* Пока не достигли конца списка смещений */
while (offsets[i].x_offset != 9) {

/* Вычисляем реальные координаты для текущей позиции */
xoff = x + (offsets[i].x_offset * neg);
yoff = y + (offsets[i].y_offset * neg);

/* "Закругление" координат (в соответствии с рис. 7.2) */
xoff = clip( xoff );
yoff = clip( yoff );

/* Если в полученной точке что-то есть, то увеличиваем счетчик
 * входов
 */
if (landscape[plane][yoff][xoff] != 0) {
  inputs[plane]++;
  }

  i++;
     }
  }

  return;
}

int clip( int z )
{
  if (z > MAX_GRID-1) z = (z % MAX_GRID);
  else if (z < 0) z = (MAX_GRID + z);
  return z;
}
```

Вспомните, что при каждом вызове функции `percept` выполняется расчет количества объектов в зоне видимости агента для всех трех плоскостей. Поэтому функция `percept` использует цикл для изучения всех плоскостей и рассчитывает суммы на основании информации о соответствующей плоскости. Для каждой плоскости программа перемещается по всем парам координат смещения (как указано аргументом смещения). Каждая пара координат смещения задает новый набор координат на основании текущего положения. Используя новые координаты, программа изменяет соответствующий элемент массива `inputs`, если в ячейке уже есть объект. Это значит, что агент не знает, сколько объектов существует в данной плоскости; ему известно только, что там есть, по крайней мере, один объект.

В листинге 7.12 представлена функция `clip`, которая используется функцией `percept`, чтобы достичь в сетке эффекта тороида.

Функция `turn`, показанная в листинге 7.13, очень проста, поскольку агент всего лишь изменяет направление, в котором «смотрит». В зависимости от текущего направления агента и направления для поворота устанавливается новое направление.

Листинг 7.13. Функция `turn`

```
void turn ( int action, agentType *agent )
{
/* В зависимости от направления поворота агента вычисляем новое
 * направление движения
 */
switch( agent->direction ) {

    case NORTH:
      if (action == ACTION_TURN_LEFT) agent->direction = WEST;
      else agent->direction = EAST;
      break;

    case SOUTH:
      if (action == ACTION_TURN_LEFT) agent->direction = EAST;
      else agent->direction = WEST;
      break;

    case EAST:
      if (action == ACTION_TURN_LEFT) agent->direction = NORTH;
      else agent->direction = SOUTH;
      break;

    case WEST:
      if (action == ACTION_TURN_LEFT) agent->direction = SOUTH;
      else agent->direction = NORTH;
      break;
  }

  return;
}
```

Функция `move` немного сложнее. Используя набор смещений для установки нового положения по осям координат, а также направление, которое помогает определить нужную пару смещений, программа рассчитывает новый набор координат. В листинге 7.14 представлено также управление средой. Перед перемещением агента изменяется среда для данной плоскости `plane` (как указано типом агента `type`), чтобы отобразить движение агента. После перемещения агента среда вновь изменяется, чтобы показать, что агент находится в другой ячейке на заданной плоскости.

Листинг 7.14. Функция `move`

```
void move( agentType *agent )
{
   /* Определяем смещение новой позиции в зависимости от направления
    * движения агента
    */
   const offsetPairType offsets[4]={{-1,0},{1,0},{0,1},{0,-1}};

   /* Удаляем агента со старого места */
   landscape[agent->type][agent->location.y][agent->location.x]--;

   /* Обновляем координаты агента */
   agent->location.x =
     clip( agent->location.x + offsets[agent->direction].x_offset );

  agent->location.y =
    clip( agent->location.y + offsets[agent->direction].y_offset );

   /* Помещаем агента в новое место */
   landscape[agent->type][agent->location.y][agent->location.x]++;

  return;
}
```

Выполнение функции `eat` разбито на два этапа: поиск объекта для съедания в области «близости» агента (если таковой существует), а затем запись об удалении съеденного объекта (листинг 7.15).

Листинг 7.15. Функция `eat`

```
void eat( agentType *agent )
{
  int plane=0, ax, ay, ox, oy, ret=0;

   /* Сначала определяем слой, объект в котором будет съеден */
   if (agent->type == TYPE_CARNIVORE) plane = HERB_PLANE;
  else if (agent->type == TYPE_HERBIVORE) plane = PLANT_PLANE;

   /* Сокращаем имена переменных */
   ax = agent->location.x;
   ay = agent->location.y;

   /* Выбираем съедаемый объект в зависимости от направления агента */
   switch( agent->direction ) {

     case NORTH:
       ret = chooseObject( plane, ax, ay, northProx, 1, &ox, &oy );
       break;

     case SOUTH:
```

```
        ret = chooseObject( plane, ax, ay, northProx, -1, &ox, &oy );
        break;
      case WEST:
        ret = chooseObject( plane, ax, ay, westProx, 1, &ox, &oy );
        break;

      case EAST:
        ret = chooseObject( plane, ax, ay, westProx, -1, &ox, &oy );
        break;

    }

    /* Объект нашли - съедаем его! */
    if (ret) {

      int i;

      if (plane == PLANT_PLANE) {

     /* Найти растение по его позиции */
     for (i = 0 ; i < MAX_PLANTS ; i++) {
          if ((plants[i].location.x == ox) &&
              (plants[i].location.y == oy))
            break;
       }

       /* Если растение найдено, то удаляем его и сажаем в другом
        * месте новое
        */
       if (i < MAX_PLANTS) {
              agent->energy += MAX_FOOD_ENERGY;
              if (agent->energy > MAX_ENERGY) agent->energy = MAX_ENERGY;
              landscape[PLANT_PLANE][oy][ox]--;
              if (noGrow == 0) {
                 growPlant( i );
       }

           } else if (plane == HERB_PLANE) {

  /* Найти травоядное в списке агентов (по его позиции) */
  for (i = 0 ; i < MAX_AGENTS ; i++) {
          if ((agents[i].location.x == ox) &&
              (agents[i].location.y == oy))
            break;
       }

  /* Если нашли, то удаляем агента */
  if (i < MAX_AGENTS) {
```

```
        agent->energy += (MAX_FOOD_ENERGY*2);
        if (agent->energy > MAX_ENERGY) agent->energy = MAX_ENERGY;
        killAgent( &agents[i] );
      }

    }

/* Если агент имеет достаточно энергии для размножения, то
 * позволяем ему сделать это
 */
if (agent->energy > (REPRODUCE_ENERGY * MAX_ENERGY)) {
      if (noRepro == 0) {
        reproduceAgent( agent );
        agentBirths[agent->type]++;
      }
    }
  }

  return;
}
```

Последний этап состоит в том, чтобы определить плоскость для поиска. Выбор основывается на типе агента, который съедает пищу. Если агент является травоядным, то поиск ведется на плоскости растений, в противном случае – на плоскости травоядных (для хищников).

Далее, используя направление движения агента, программа вызывает функцию `chooseObject` (листинг 7.16), чтобы вернуть координаты объекта интереса на нужную плоскость. Обратите внимание, что здесь вновь задействованы пары координат смещения (как и в листинге 7.11), но внимание уделяется только области «близости» в направлении движения агента. Если объект был найден, функция `chooseObject` возвращает значение, которое не равно нулю, и заполняет координаты `ox`/`oy` в соответствии с функцией `eat`.

Листинг 7.16. Функция `chooseObject`

```
int chooseObject( int plane, int ax, int ay,
                  const offsetPairType *offsets,
                  int neg, int *ox, int *oy )
{
  int xoff, yoff, i=0;

   /* Проходим по всему списку смещений */
   while (offsets[i].x_offset != 9) {

   /* Определяем координаты */
   xoff = ax + (offsets[i].x_offset * neg);
   yoff = ay + (offsets[i].y_offset * neg);
```

```
    xoff = clip( xoff );
    yoff = clip( yoff );

    /* Если объект найден, возвращаем его индекс    */
    if (landscape[plane][yoff][xoff] != 0) {
          *ox = xoff; *oy = yoff;
          return 1;
     }

    /* Проверить следующие координаты */
    i++;

   }

   return 0;
}
```

Функция `chooseObject` очень похожа на функцию `percept` (листинг 7.12), однако вместо того, чтобы суммировать объекты, расположенные в плоскости данной зоны, она возвращает координаты первого найденного объекта.

Следующий этап – съедание объекта. Если подходящий объект был найден, программа проверяет плоскость, в которой он был обнаружен. Для плоскости растений выполняется поиск в массиве `plants`, после чего растение удаляется из массива `landscape`. Затем создается новое растение, которое будет помещено в новую произвольную ячейку. Для плоскости травоядных животных программа идентифицирует съедаемое травоядное с помощью диапазона `agents` и «убивает» его функцией `killAgent` (листинг 7.17). Благодаря съеданию объекта увеличивается энергия текущего агента.

Наконец, если агент достиг уровня энергии, который необходим для воспроизводства, вызывается функция `reproduceAgent`, чтобы позволить агенту «родить» нового агента данного типа (листинг 7.18).

Уничтожение агента является, прежде всего, задачей, ориентированной на запись данных. Сначала программа удаляет агента из среды и записывает статистические данные (количество смертей для типа данного агента и общее количество агентов этого типа). Затем она сохраняет данные агента (его описание) при условии, если он является самым старым агентом этого типа.

После записи программа определяет, нужно ли инициализировать нового случайного агента (данного типа) вместо уничтоженного агента. Решение зависит от количества агентов данного типа, которое присутствует в модели. Поскольку эволюционный аспект симуляции является самым интересным, требуется сохранить количество открытых позиций для агентов, чтобы любой агент при достижении заданного уровня энергии мог воспроизвести себя. Поэтому новому случайному агенту позволяется занять место уничтоженного агента только при условии, если популяция данного типа агентов составляет менее 25% от общего количества агентов. Это позволяет сохранять 25% мест для агентов одного типа свободными для последующего воспроизводства.

Листинг 7.17. Функция `killAgent`

```
void killAgent( agentType *agent )
{
  agentDeaths[agent->type]++;

  /* Пришла смерть (или агента съели) */
  landscape[agent->type][agent->location.y][agent->location.x]==;
  agentTypeCounts[agent->type]==;

  if (agent->age > bestAgent[agent->type].age) {
    memcpy( (void *)&bestAgent[agent->type],
            (void *)agent, sizeof(agentType) ); }

  /* 50% ячеек резервируем для воспроизводства агентов, если
   * здесь произошла ошибка, то создаем фиктивного агента
   */
  if (agentTypeCounts[agent->type] < (MAX_AGENTS / 4)) {

      /* Создание нового агента */
      initAgent( agent );

    } else {

      agent->location.x = -1;
      agent->location.y = -1;
      agent->type = TYPE_DEAD;
  }

  return;
}
```

Последняя функция симуляции, `reproduceAgent`, является самой интересной, поскольку она вносит в модель аспект эволюции Ламарка. Когда агент воспроизводит себя, он передает свою нейронную сеть ребенку. Ребенок наследует нейронную сеть родителя, а потом производится с небольшой вероятностью мутации весов сети. Это позволяет внести в модель эволюционный аспект, а также повысить конкурентоспособность при выживании в среде. Функция `reproduceAgent` представлена в листинге 7.18.

Листинг 7.18. Функция `reproduceAgent`

```
void reproduceAgent( agentType *agent )
{
  agentType *child;
  int i;

  /* Не даем агенту одного типа занять более половины доступных
   * ячеек
   */
  if ( agentTypeCounts[agent->type] < (MAX_AGENTS / 2)) {
```

```
    /* Найти пустое место и скопировать агента. При этом происходит
     * мутация одного веса или смещение в нейронной сети агента
     */
    for (i = 0 ; i < MAX_AGENTS ; i++) {
      if (agents[i].type == TYPE_DEAD) break;
    }

    if (i < MAX_AGENTS) {

      child = &agents[i];

      memcpy( (void *)child, (void *)agent, sizeof(agentType) );

      findEmptySpot( child );

      if (getSRand() <= 0.2) {
        child->weight_oi[getRand(TOTAL_WEIGHTS)] = getWeight();
      }

      child->generation = child->generation + 1;
      child->age = 0;

      if (agentMaxGen[child->type] < child->generation) {
        agentMaxGen[child->type] = child->generation;
      }

      /* Репродукция уменьшает энергию родителя вдвое */
      child->energy = agent->energy = (MAX_ENERGY / 2);

      agentTypeCounts[child->type]++;
      agentTypeReproductions[child->type]++;

    }

  }

  return;
}
```

Сначала необходимо определить, есть ли свободное место для ребенка, проверив, заполнено пространство для агентов данного типа на 50% или нет. Это обеспечивает ровное распределение агентов в среде, что биологически не совсем корректно, но позволяет симулировать доминантное положение одной особи по отношению к другой в игровом режиме (который будет описан далее).

Если для ребенка было найдено свободное место, структура агента-родителя копируется для ребенка, а затем отыскивается свободная ячейка, которую займет ребенок. Далее посредством мутации изменяется один из весов в нейронной сети агента. Обратите внимание, что для поиска веса, который будет изменен,

используется символьная переменная TOTAL_WEIGHTS. Это позволяет учесть не только веса, но и смещения (поскольку они связаны со структурой агента). Затем выполняется запись данных, а энергия родителя делится поровну между ним и ребенком, что заставит родителя и ребенка передвигаться по среде в поисках пищи, пока они не наберут нужного количества энергии для дальнейшего воспроизведения.

Примеры функционирования модели

Рассмотрим несколько примеров работы модели. Симуляция может быть запущена без определения дополнительных параметров, например:

```
./sim
```

При этом симуляция будет запущена с параметрами, которые указаны в заголовочном файле common.h. Установки файла common.h по умолчанию включают 36 агентов (18 травоядных и 18 хищников), а также 30 растений в сетке 30×30. Максимальное количество шагов в симуляции составляет 1 млн. На рис. 7.10 показан график максимального возраста, который был достигнут для каждой особи.

Интересно отметить тенденцию увеличения возраста агентов. Если хищники находят интересную стратегию, которая позволяет продлить их существование,

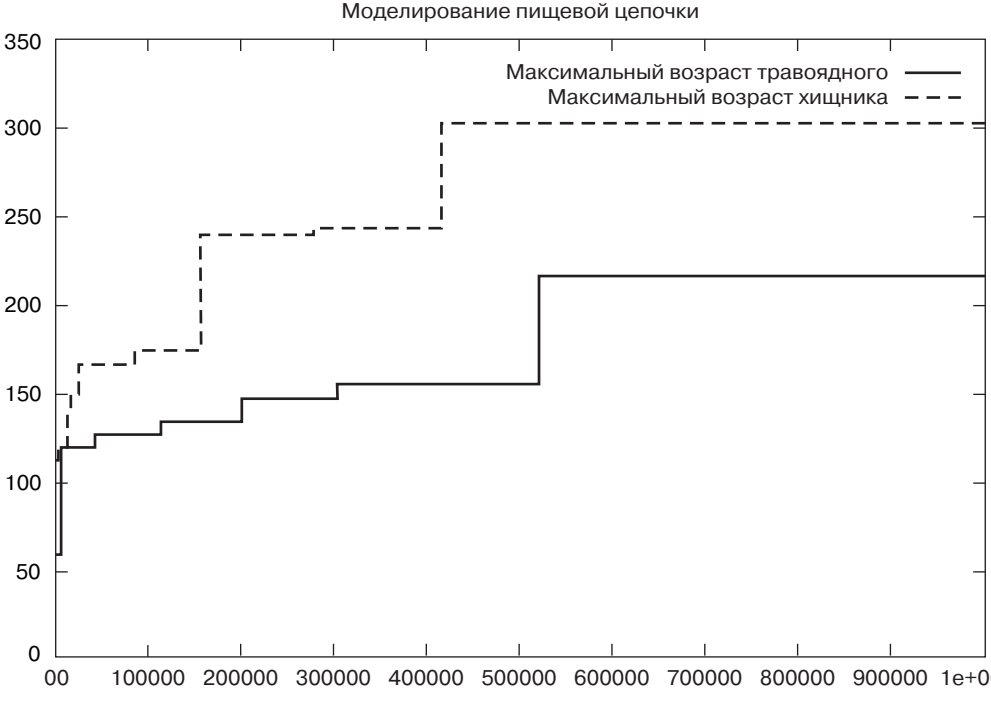


Рис. 7.10. Увеличение возраста агентов в примере симуляции

через короткое время травоядные вырабатывают другую стратегию, которая позволяет им прожить дольше. Определенным образом особи соревнуются друг с другом. Если один тип животных изобретает новую стратегию, другому типу приходится изобретать свою стратегию, чтобы противостоять ей.

После завершения работы симуляции описание двух лучших агентов (по одному от каждого типа) сохраняются в файле `agents.dat`. Затем они могут сразиться друг с другом в другой симуляции, которая называется `playback`. Этот режим не создает популяцию из случайного количества агентов, а начинает работать с лучшими агентами из предыдущего запуска. Вы можете запустить программу с помощью следующей команды:

```
./sim -prn
```

Аргумент `p` указывает, что требуется запустить режим `playback`. Аргумент `r` показывает, что необходимо сохранять информацию о тенденции, а аргумент `n` запрещает воспроизводство. В режиме `playback` записываются такие данные, как счет рождений и смертей агентов (для всех особей).

Используя агентов, полученных при первом запуске, программа выводит график для новой симуляции (рис. 7.11).

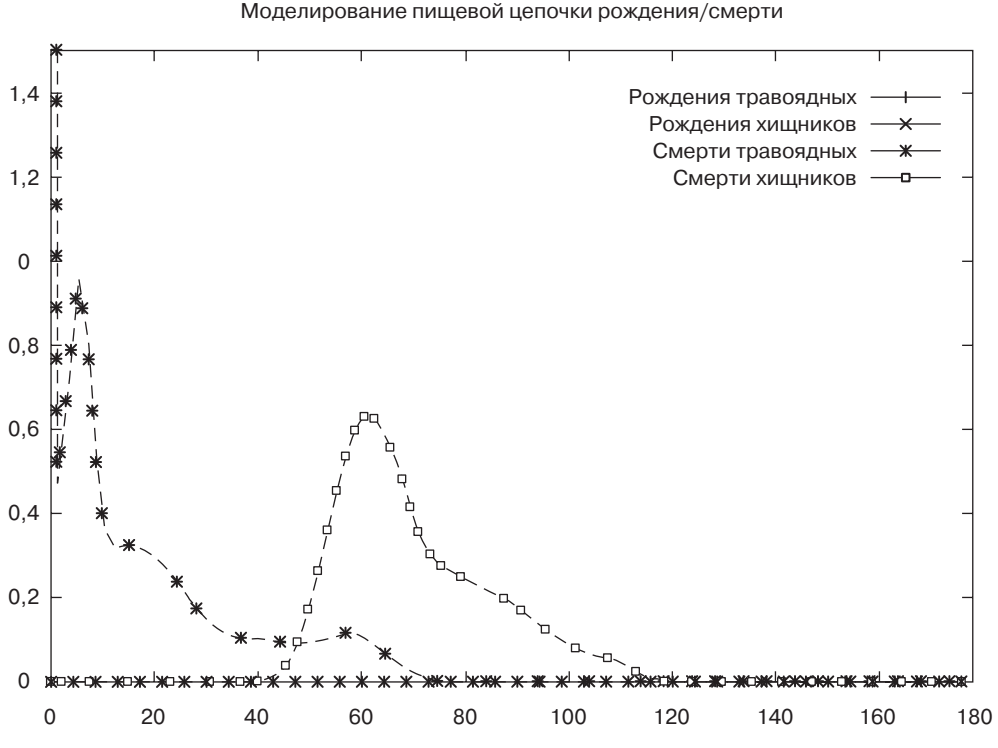


Рис. 7.11. График данных при запуске симуляции в режиме `playback`. Кривые рождений хищников и травоядных хотя и присутствуют на графике, но не видны из-за малого масштаба и частоты смертей как травоядных, так и хищников

Поскольку в этом решении запрещено воспроизводство, симуляция не показывает рождения, только смерти. Когда симуляция началась, среда была инициализирована с травоядными и хищниками. График показывает, что для хищников «время смерти» наступает тогда, когда в среде не остается травоядных, которых они могли бы съесть. При этом кривая смертей хищников возрастает, так как среди них наступает голод. Потеряв источник пищи, хищники вымирают.

Вы можете задавать параметры программы, представленные в табл. 7.1.

Таблица 7.1. Параметры программы симуляции

Опция	Описание
`-h`	Вывести справку
`-p`	Режим воспроизведения (агенты загружаются из файла agents.dat)
`-r`	Сохранение данных агентов (файл runtime.dat)
`-g`	Запретить «выращивание» съеденных растений
`-c`	Конвертировать хищников в пищу после смерти
`-n`	Запретить воспроизводство агентов
`-s`	Ручное управление (требуется нажатие клавиши **Enter**)

Интересный сценарий можно запустить с помощью следующей команды:

```
./sim -prncg
```

В среде создается «круговорот еды»: хищники охотятся на травоядных и поедают их, но после смерти сами становятся пищей для травоядных.

Интересные стратегии

Хотя данная модель очень проста, и агентам предоставляется минимальное количество входных сенсоров и доступных действий, в результате могут образоваться весьма любопытные стратегии поведения.

Интересная стратегия для травоядных животных – это инстинкт стада. Травоядное будет следовать за другим травоядным, если оно находится в области «фронт». Сила травоядных животных в их количестве, конечно, если это не то травоядное, которое идет впереди. Хищники нашли множество любопытных стратегий, одна из которых состоит в том, чтобы найти растения и подстерегать возле них травоядных. Эта стратегия была успешной, но только в течение короткого времени, так как травоядные быстро научились избегать хищников даже при условии, что в области «близости» находятся растения.

Изменение параметров

Размер среды, количество агентов и растений – это параметры, которые связаны между собой. Чтобы модель была сбалансированной, количество растений должно быть, по крайней мере, равным количеству травоядных (то есть составлять половину общего количества агентов). Если растений будет меньше, травоядные

быстро вымрут, а следом за ними вымрут и хищники. Количество агентов не должно быть слишком большим, чтобы они не переполнили среду. Если количество агентов и размеры сетки схожи, модель будет сбалансированной.

Параметры модели задаются в заголовочном файле `common.h`. Модель также можно настраивать с помощью параметров командной строки, представленных в табл. 7.1.

Итоги

В этой главе понятие искусственной жизни рассматривалось на примере моделирования простой пищевой цепочки. Искусственная жизнь предлагает платформу для изучения различных феноменов в биологических и социальных системах. Самое значительное преимущество искусственной жизни в сфере синтетической теории поведения – это возможность играть в ролевые игры, изменяя параметры модели и отслеживая результаты. В данной главе концепции синтетической теории поведения были продемонстрированы с помощью несложной симуляции хищник/жертва. В результате хищники и жертвы выработали ряд любопытных стратегий поведения.

Литература и ресурсы

1. CALResCo. Концепция исследования искусственной жизни для самоорганизующихся систем (The Complexity & Artificial Life Research Concept for Self-Organizing Systems). Доступно по адресу http://www.calresco.org.
2. Digital Life Lab at Caltech. Программное обеспечение для Aveda (Aveda Software). Доступно по адресу http://dllab.caltech.edu/avida.
3. Лангтон К. Что такое искусственная жизнь (Langton C. What Is Artificial Life). Доступно по адресу http://www.biota.org/papers/cgalife.html.
4. МакЛеннан Б. Домашняя страница Брюса МакЛеннана (MacLennan B. Bruce MacLennan's Home Page), http://www.cs.utk.edu/~mclennan/.
5. МакЛеннан Б. Искусственная жизнь и синтетическая теория поведения (MacLennan B. Artificial Life and Synthetic Ethology). Доступно по адресу http://www.cs.utk.edu/~mclennan/alife.html.
6. Международное общество изучения искусственной жизни. Web-сайт International Society for Artificial Life, http://www.alife.org.

Глава 8. Введение в системы, основанные на правилах

В этой главе мы поговорим об одной из оригинальных систем искусственного интеллекта, системе, основанной на знании. Эти системы также называются *экспертными* (или продукционными системами), знание в них кодируется в виде правил. Знание (или факты) сохраняется в рабочей памяти, а правила применяются к знанию, чтобы создать новое знание. Процесс продолжается до тех пор, пока не будет достигнута определенная цель. В данной главе рассматривается простая система, основанная на правилах, а также ее применение при построении контроллера, устойчивого к ошибкам.

Введение

Хотя существует большое количество систем, основанных на правилах, мы сфокусируемся на системах с *продукционными правилами* (Production rules). Продукционные правила состоят из двух частей. Первая часть правила (часть ЕСЛИ) описывает предпосылку, то есть условие применения правила. Вторая часть (часть ТО) содержит последствия использования правил. При применении правила могут произойти события двух типов: получение нового знания (поэтому правила и называются продукционными) и выполнение некоторого действия для изменения окружающей среды.

Архитектура системы, основанной на правилах

Система, основанная на правилах, состоит из группы отдельных элементов. Существует ряд правил, которые управляют фактами, сохраненными в рабочей памяти. Логика используется, чтобы идентифицировать правило, которое следует использовать (на основании предпосылки). После применения правила рабочая память изменяется (на основании последствий). На рис. 8.1 графически показана простая система, основанная на правилах.

Правила управляют фактами, которые хранятся в рабочей памяти. После того как было найдено соответствие для правила, оно вступает в действие; при этом рабочая память может быть изменена, а может остаться неизменной. Процесс продолжается, пока не будет достигнута определенная цель.

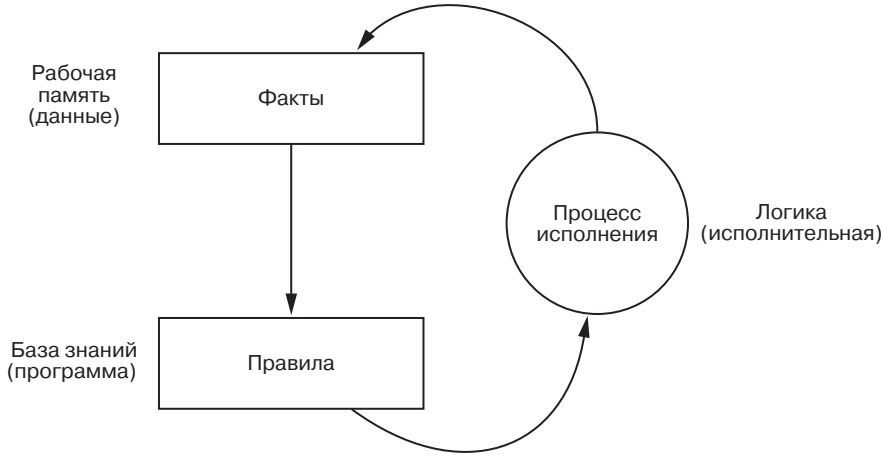


Рис. 8.1. Система, основанная на правилах

Рабочая память

Рабочая память (Working memory) представляет собой структуру, в которой хранятся известные на данный момент факты. Это постоянное пространство, которое может изменяться только с помощью последствия правила (причем правило может быть применено только в том случае, если найдено соответствие для предпосылки). Рассмотрим следующий пример рабочей памяти:

```
(sensor-failed sensor1)
(mode normal)
```

В данном примере известны два факта. Они закодированы в парах «имя – значение». Например, первый факт задается как `sensor-failed`, а его значение – как `sensor1`. Это означает, что `sensor1` является неисправным сенсором. Второй факт указывает, что режим `mode` является `normal`. Все факты кодируются подобным образом (хотя в коммерческих системах, основанных на правилах, обычно используется более сложная система кодировки).

База знаний

База знаний (Knowledge base) содержит группу правил, которые управляют фактами в рабочей памяти. Правила состоят из двух частей и включают предпосылку и последствия. Предпосылка определяет, какие факты должны быть истинными, чтобы правило вступило в действие. Последствие задает действия, которые должны быть выполнены при применении правила. Рассмотрим следующий пример:

```
(defrule sensor-check
(sensor-failed sensor1)
=>
(add (disable sensor1))
)
```

Команда `defrule` указывает, что для системы задается правило. За этой командой вводится текстовое название правила. Далее следует одно или несколько условий – предпосылок, в данном случае (`sensor-failed sensor1`). Символ «=>» разделяет предпосылку и последствия. Затем определяется одно или несколько последствий. Указанные в секции ТО действия производятся только при условии выполнения правила. Наконец, правило закрывается круглой скобкой. Действия делятся на две части. Первая часть – это команда, которая должна быть выдана, а вторая – параметр, на который влияет команда. В данном примере при выполнении правила в рабочую память будет добавлен новый факт (`disable sensor1`).

Система логического вывода

Система с правилами основывается на логике, которая определяет, какие правила должны быть применены, а также выполняет их. Этот процесс обычно называют «цикл соответствие-действие» (см. далее раздел «Фазы работы системы, основанной на правилах»).

Типы систем, основанных на правилах

Перед тем как обсуждать систему логического вывода, важно понять, что существуют два принципиально разных типа таких систем – прямого вывода и обратного вывода.

Система обратного вывода

Система обратного вывода (Backward chaining) представляет собой стратегию, при которой программа просматривает все правила, но выбирает те, последовательность выполнения которых позволяет достичь цели. Для каждого из этих правил проверяется, соответствуют ли первые операнды (предпосылки) информации в рабочей памяти. Если все предпосылки удовлетворяют этому условию, правило выполняется и задача решается. Если существует предпосылка, которая не соответствует информации в рабочей области, определяется новая подцель как «организация условий для удовлетворения этой предпосылки». Процесс выполняется рекурсивно. Если известны значения цели и их число невелико, то система обратного вывода вполне эффективна.

Система прямого вывода

Система прямого вывода (Forward chaining) начинает работу от известных фактов. Затем происходит обращение к базе знаний, чтобы идентифицировать правила, которые соответствуют фактам и, значит, могут внести в рабочую память новые факты. Процесс продолжается до тех пор, пока цель не будет достигнута или больше не будет обнаружено новых фактов. Это дедуктивный метод, который использует известные факты, чтобы при продвижении по рабочей памяти (и правилам) создавать новые факты.

В данной главе основное внимание уделяется системе прямого вывода, приводятся примеры ее применения и подробно рассматривается ее реализация.

Фазы работы системы, основанной на правилах

Рассмотрим фазы работы системы, основанной на правилах, на примере системы прямого вывода (рис. 8.2).

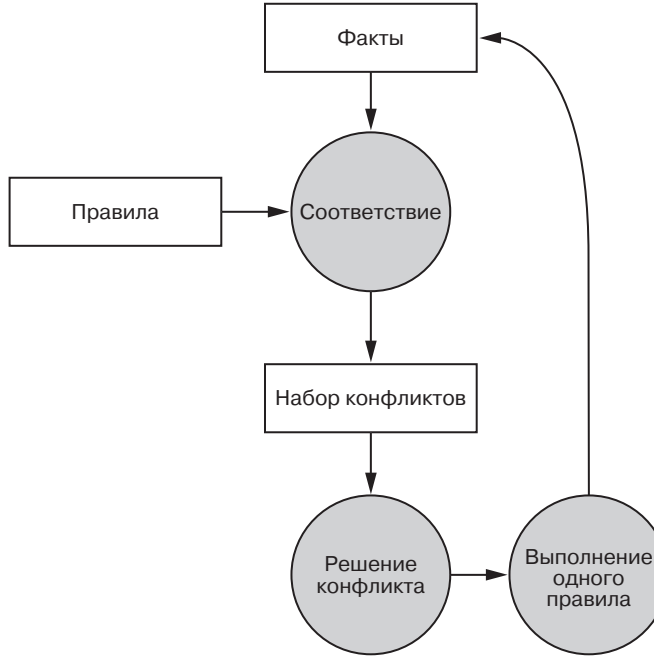


Рис. 8.2. Фазы системы, основанной на правилах

Фаза соответствия

Во время *фазы соответствия* (Match phase) каждое правило проверяется на соответствие между набором его предпосылок и фактами в рабочей памяти. Если соответствие найдено, правило добавляется в набор конфликтов. Правила, для предпосылок которых не было обнаружено соответствий, будут проигнорированы. После проверки всех правил набор конфликтов рассматривается в следующей фазе, – фазе разрешения конфликтов.

Фаза разрешения конфликтов

Задача *фазы разрешения конфликтов* (Conflict resolution phase) заключается в том, чтобы выбрать в наборе конфликтов правило, которое будет выполнено. Если в наборе только одно правило, то данный процесс очень прост. Если в наборе несколько правил, необходимо задать критерий выбора правила. Такой критерий

может быть сложным, например, выбор правила с наибольшим количеством предпосылок (или последствий), или простым, например, выбор первого правила. Выбрав правило, программа переходит в фазу действия.

Фаза действия

Фаза действия (Action phase) реализует последствия для выбранного правила. Они могут включать добавление фактов в рабочую память, удаление их из рабочей памяти или выполнение других действий. Например, если система, основанная на правилах, соединена с каким-либо устройством, посредством действий может осуществляться управление механизмом (например, можно двигать рукой робота или управлять выключателем).

Простой пример

Разберем простой пример с набором правил и группой фактов в рабочей памяти. Здесь используется подгруппа правил ZOOKEEPER, предложенная Патриком Генри Уинстоном (Patrick Henry Winston), профессором Департамента Искусственного Интеллекта и компьютерной науки Массачусетского Технологического Института.

Рассмотрим следующие правила (листинг 8.1).

Листинг 8.1. Задача, адаптированная для правил ZOOKEEPER

```
(defrule bird-test
   (has-feathers ?)
=>
   (bird ?)
)

(defrule mammal-test
   (gives milk ?)
=>
   (mammal ?)
)

(defrule ungulate-test1
   (mammal ?)
   (chews-cud ?)

=>
   (is-ungulate ?)
)

(defrule ungulate-test2
   (mammal ?)
   (has hoofs ?)
=>
   (is-ungulate ?)
)
```

Теперь предположим, что рабочая память содержит следующие факты:

```
(gives-milk animal)
(has-hoofs animal)
```

Начнем с фазы соответствия, в ходе которой программа попытается найти соответствие между фактами в рабочей памяти и предпосылками в наборе правил. Ни один факт не удовлетворяет предпосылкам первого правила (`bird-test`), но программа обнаружила соответствие во втором правиле (`mammal-test`). Правило сохраняется в наборе конфликтов, а поиск продолжается по остальным правилам. Других соответствий для правила в рабочей памяти нет, поэтому набор конфликтов будет включать только одно правило. Поскольку конфликт отсутствует, правило `mammal-test` выполняется (фаза действия), и рабочая память принимает следующий вид:

```
(gives-milk animal)
(has-hoofs animal)
(mammal animal)
```

Программа вновь начинает с фазы соответствия и проходит по набору правил в поисках соответствующих предпосылок. В результате набор конфликтов содержит два правила, `mammal-test` и `ungulate-test2`. В данном случае процедура разрешения конфликтов очень проста, поскольку здесь только одно правило влияет на рабочую память (`ungulate-test2`). Первое правило уже было выполнено ранее; ничто новое не может быть добавлено в рабочую память, следовательно, правило может не учитываться в наборе конфликтов. При переходе в фазу действия и реализации правила `ungulate-test2` рабочая память принимает следующий вид:

```
(gives-milk animal)
(has-hoofs animal)
(mammal animal)
(ungulate animal)
```

На этом простом примере система с помощью дедукции определила, что под знаком вопроса подразумевается копытное животное. После того как система узнала, что животное является млекопитающим (то есть дает молоко), она использовала эту информацию вместе с данными о том, что животное имеет копыта, и определила, что данное животное является копытным.

Разрешение конфликта в данном примере включает два базовых механизма выбора правила для выполнения. Для более сложных сценариев могут использоваться другие механизмы. Например, система при разрешении конфликта может выбрать для выполнения то правило из набора конфликтов, которое имеет наибольшее число предпосылок. Этот метод используется в самых сложных случаях и позволяет системе достичь нужного результата и проигнорировать более простые варианты, которые приведут к нежелательным результатам.

Пример использования

Теперь рассмотрим другой пример, который демонстрирует систему, устойчивую к ошибкам, а при его изучении подробно разберем варианты, содержащиеся в части ТО (то есть последствия).

Устойчивость к ошибкам

В данном примере будет создана простая система знаний по управлению сенсорами. Существует группа сенсоров, один из которых допускается задать как активный. Любой сенсор может быть исправен или неисправен, но в определенный момент времени только один сенсор может быть активным. Если все сенсоры неисправны, то ни один не может быть активным. Кроме того, режим должен соответственно изменяться, чтобы сигнализировать о существовании проблемы в подсистеме сенсоров.

Подобная архитектура напоминает доску объявлений (рис. 8.3). Она включает группу агентов, которые используют данные и помещают их на доску объявлений, являющуюся рабочим пространством для коммуникации с агентами. Агенты активируются данными на доске и могут управлять ими (добавлять, изменять или удалять). Допускается включать других агентов, которые примут участие в рабочем процессе.

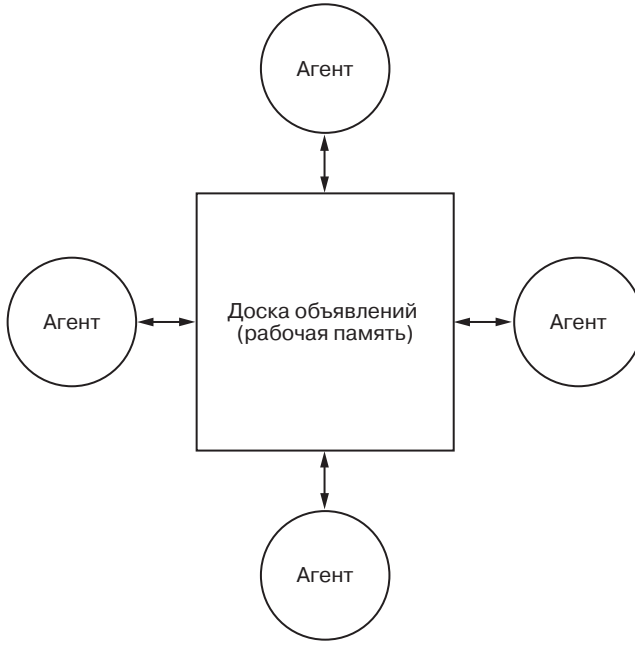


Рис. 8.3. Графическое представление архитектуры доски

Архитектура доски объявлений не только предоставляет агентам возможность взаимодействия друг с другом, но и позволяет им координировать и синхронизировать свои действия.

Определение правил

Файл правил представлен в листинге 8.2. Набор правил вводит группу новых последствий (действий), которые могут быть выполнены.

Листинг 8.2. Демонстрация правил при погрешности

```
(defrule init
   (true null)        ; предпосылка
=>
   (add (sensor-active none))     ; последствия
   (add (sensor-working sensor1))
   (add (sensor-working sensor2))
   (add (mode normal))
   (enable (timer 1 10))
   (print ("default rule fired!"))
(disable (self))
)

;
; Определение набора активных правил
;

(defrule sensor-failed
   (sensor-working ?)
   (sensor-failed ?)
=>
   (delete (sensor-working ?))
)

(defrule check-active
   (sensor-active ?)
   (sensor-failed ?)
=>
   (delete (sensor-active ?))
   (add (sensor-active none))
)

(defrule make-working
   (sensor-active none)
   (sensor-working ?)
=>
   (add (sensor-active ?))
   (delete (mode failure))
   (add (mode normal))
   (delete (sensor-active none))
)
```

```
(defrule failure
   (mode normal)
   (sensor-active none)
   (sensor-failed sensor1)
   (sensor-failed sensor2)
=>
   (add (mode failure))
   (delete (mode safe))
   (delete (mode normal))
)

; Используем триггеры для эмуляции событий
(defrule trigger1
   (timer-triggered 1)
=>
   (print ("Sensor 1 failure.\n"))
   (add (sensor-failed sensor1))
   (enable (timer 2 10))
   (delete (timer-triggered 1))
)

(defrule trigger2
   (timer-triggered 2)
=>
   (print ("Sensor 2 failure.\n"))
   (add (sensor-failed sensor2))
   (enable (timer 3 10))
   (delete (timer-triggered 2))
)

(defrule trigger3
   (timer-triggered 3)
=>
   (print ("Sensor 1 is now working.\n"))
   (delete (sensor-failed sensor1))
   (add (sensor-working sensor1))
   (enable (timer 4 10))
   (delete (timer-triggered 3))
)

(defrule trigger4
   (timer-triggered 4)
=>
   (print ("Sensor 2 is now working.\n"))
   (delete (sensor-failed sensor2))
   (add (sensor-working sensor2))
   (enable (timer 1 10))
   (delete (timer-triggered 4))
)
```

Этот пример включает девять правил, но только четыре из них являются работающим. Первым в листинге 8.2 вводится правило инициализации (`init`). Обратите внимание, что предпосылка (`true null`) всегда является истинной. Правило инициализации позволяет заполнить рабочую память начальным набором фактов (как указано командами `add`). Команда `enable` активирует таймер, который при включении может задействовать другое правило. В данном примере таймеры помогают эмулировать события (факты), поступающие из окружающей среды и помещенные в рабочую память другими агентами, которые действуют в среде (рис. 8.3).

Примечание *Таймеры данного примера были смоделированы после выхода игры «Век королей» компании Microsoft, в которой файл с правилами использовался в системе искусственного интеллекта.*

Команда `timer` использует два аргумента, числовой идентификатор таймера и число секунд, через которое таймер должен включиться (в данном случае таймер 1 сработает через 10 с). Команда `print` позволяет изучить работу системы. Особое значение имеет последняя команда, `disable`, которая предоставляет возможность удалить правило из группы доступных правил. Такое правило не может больше быть выполненным. Это требуется для того, чтобы исключить правила, предпосылка которых всегда истинна (например, правило `init`).

После того как правило `init` выполнено, рабочая память будет выглядеть так:

```
(sensor-active none)
(sensor-working sensor1)
(sensor-working sensor2)
(mode failure)
```

Затем выполняется правило `make-working` (в качестве параметра соответствия используется `sensor1`). Рабочая память примет следующий вид:

```
(sensor-working sensor1)
(sensor-working sensor2)
(sensor-active sensor 1)
(mode normal)
```

В данный момент другие правила не могут быть выполнены, и система сохраняет свое состояние вплоть до срабатывания таймера (через 10 с). При поступлении события от таймера система добавляет факт в рабочую память: (`timer-triggered 1`). Новый факт заставляет систему реализовать то правило, которое управляет таймером (в качестве предпосылки правила используется событие-триггер). Действие, активированное таймером, эмулирует сбой сенсора `sensor1`. Рабочая память примет следующий вид:

```
(sensor-working sensor1)
(sensor-working sensor2)
(sensor-active sensor1)
```

```
(mode normal)
(sensor-failed sensor1)
```

Затем выполняются правила `sensor-failed` и `check-active`. При этом рабочая память примет следующий вид:

```
(sensor-working sensor2)
(mode normal)
(sensor-failed sensor1)
(sensor-active none)
```

Наконец, выполняется правило `make-working`. Рабочая память принимает следующий вид:

```
(sensor-working sensor2)
(mode normal)
(sensor-failed sensor1)
(sensor-active sensor2)
```

В данный момент система работает с сенсором из резервной пары. Правила также позволяют работать без активированных сенсоров (это будет отражаться с помощью факта `mode`). Выполнение остальных правил вы сможете изучить самостоятельно.

Примечание *Приведенная база знаний может быть использована совместно с системой, основанной на правилах, исходный код которой находится в архиве с примерами к данной книге на сайте издательства «ДМК Пресс» www. dmk.ru.*

Обсуждение исходного кода

Рассмотрим простое применение системы, основанной на правилах, и фактов, о которых рассказывалось ранее.

Базовый принцип работы программы показан на рис. 8.4. После инициализации (очистки) рабочей памяти и базы знаний система обрабатывает файл правил, заданный пользователем, и загружает правила в базу знаний. Затем программа переходит к циклу, пытаясь найти правило для выполнения в соответствии с данными, находящимися в рабочей памяти. Если правило было обнаружено, оно выполняется. Независимо от того, было выполнено правило или нет, программа попробует найти другое правило, соответствующее текущим фактам. Это и есть упрощенный принцип работы системы, основанной на правилах.

Рассмотрим структуры данных, которые используются в программе (листинг 8.3).

Листинг 8.3. Структуры данных системы, основанной на правилах

```
#define MEMORY_ELEMENT_SIZE 80
#define MAX_MEMORY_ELEMENTS 40
```

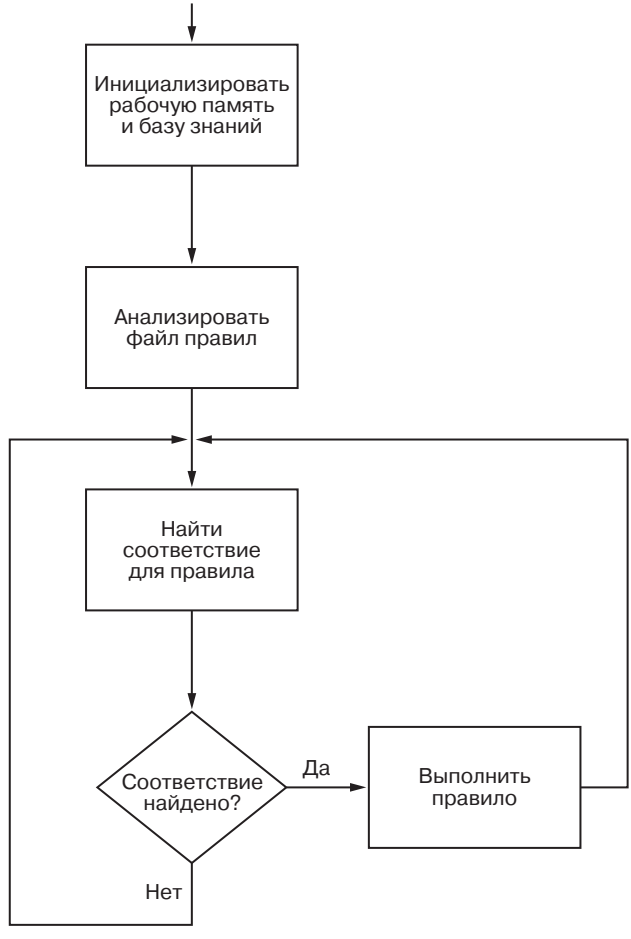


Рис. 8.4. Базовый принцип работы системы, основанной на правилах

```
#define MAX_RULES           40
#define MAX_TIMERS          10

typedef struct memoryElementStruct *memPtr;

typedef struct memoryElementStruct {
  int active;
  char element[MEMORY_ELEMENT_SIZE+1];
  struct memoryElementStruct *next;
} memoryElementType;

typedef struct {
  int active;
  char ruleName[MEMORY_ELEMENT_SIZE+1];
```

```
  memoryElementType *antecedent;
  memoryElementType *consequent;
} ruleType;

typedef struct {
  int active;
  int expiration;
} timerType;
```

Структура `memoryElementType` используется для хранения как фактов в рабочей памяти, так и элементов предпосылок и последствий в правиле. Поле `element` определяет факт в контексте рабочей памяти. Тип данных `ruleType` отображает одно правило в наборе правил. Поле `active` показывает, является ли правило в системе активным. Поле `ruleName` представляет собой название правила (указанное после команды `defrule`). Название используется, в основном, в качестве комментария, но может и упростить работу с большим количеством правил. Элементы `antecedent` и `consequent` отображают связанные списки предпосылок и последствий соответственно. Вспомните, что тип `memoryElementType` включает элемент `next`, который может быть адресом следующего правила или равняться нулю (это значит, что правило – последнее в цепи). Структура `timerType` предназначена для отображения таймеров. Поле `active` используется в ней так же, как и для правил, то есть для идентификации работающего таймера. Поле `expiration` показывает, через какое время таймер должен сгенерировать событие.

В листинге 8.4 описываются рабочие переменные системы, основанной на правилах. Они являются общими, то есть используются различными подсистемами программы.

Листинг 8.4. Общие переменные программы

```
memoryElementType workingMemory[MAX_MEMORY_ELEMENTS];

ruleType ruleSet[MAX_RULES];

timerType timers[MAX_TIMERS];

int endRun = 0;
int debug = 0;
```

Массивы `workingMemory`, `ruleSet` и `timers` были рассмотрены при изучении структур данных (листинг 8.3). Здесь введены две новые переменные: `endRun` (позволяет набору правил завершить работу системы с помощью команды `quit`) и `debug` (указывает, нужно ли выводить отладочную информацию в процессе работы программы).

Функция `main` (листинг 8.5) инициализирует систему, а также реализует цикл «соответствие-выполнение». Она начинает работу с разбора командной строки (с помощью функции `getopt`). Поддерживаются три параметра командной строки – `h` (выдает вспомогательную информацию), `d` (разрешает вывод

отладочной информации) и `r` (указывает, какой файл с правилами следует использовать). Обратите внимание, что, если файл не указан (опция `r` не была найдена), по умолчанию выводится справочная информация и выполняется выход из программы.

Затем программа очищает рабочие структуры системы, сбрасывая для каждой записи поля `active`. Далее вызывается функция `parseFile` (листинг 8.6), чтобы прочитать файл правил и поместить его в массив `ruleSet`.

Листинг 8.5. Функция `main` системы, основанной на правилах

```
int main( int argc, char *argv[] )
{
  int opt, ret;
  char inpfile[80]={0};

  extern void processTimers( void );
  extern int parseFile( char * );
  extern void interpret( void );

  while ((opt = getopt(argc, argv, "hdr:")) != -1) {

    switch( opt ) {

      case 'h':
        emitHelp();
        break;

      case 'd':
        debug = 1;
        printf("Debugging enabled\n");
        break;

      case 'r':
        strcpy(inpfile, optarg);
        break;

    }

  }

  if (inpfile[0] == 0) emitHelp();

  bzero( (void *)workingMemory, sizeof(workingMemory) );
  bzero( (void *)ruleSet, sizeof(ruleSet) );
  bzero( (void *)timers, sizeof(timers) );

  ret = parseFile( inpfile );

  if (ret < 0) {
```

```
    printf("\nCould not open file, or parse error\n\n");
    exit(0);
  }

  while (1) {

    interpret();

    if (debug) {
      printWorkingMemory();
    }

    processTimers();

    if (endRun) break;

    sleep(1);

  }

  return 0;
}
```

Главный цикл является последним элементом функции `main`, представленной в листинге 8.5. Он реализует логику «соответствие-выполнение», вызывая функцию `interpret`. Если задан вывод отладочной информации, при выполнении правил отображается состояние рабочей памяти. Программа обрабатывает таймеры, вызывая функцию `processTimers`. Если была найдена команда `quit`, производится выход из цикла (при проверке значения переменной `endRun`), и работа программы завершается. Наконец, для эмуляции функционирования системы во времени вызывается функция `sleep` (замедляет работу системы, чтобы можно было ее изучить). В системе, имеющей практическое применение, она будет удалена, а вместо нее будет использована функция `processTimers`, которая инициализирует текущее время для выполнения нужного правила.

Анализатор файла (функция `parseFile`, листинг 8.6) берет правила, сохраненные в файле, и преобразует их в форму ЕСЛИ–ТО. Анализатор очень прост, поскольку сама структура файла элементарна. Файл правил состоит из одного или нескольких правил, формат которых указан ниже:

```
(defrule <rule-name>
     (antecedent-terms)
=>
     (consequent-terms)
)
```

Команда `defrule` задает новое правило. За ней вводится текстовое название правила. Далее следуют предпосылки и символ «=>», отделяющий группу последствий. Символ «)» закрывает правило. Примеры правил, построенных по этому принципу, показаны в листинге 8.2.

При работе с этой структурой разбор правил выполняется функцией с предсказанием. Анализатор ищет стартовую команду, а затем анализирует предпосылки. После того как была найдена команда-сепаратор, начинается анализ последствий. Наконец, программа выполняет проверку закрывающего символа, чтобы убедиться, что правило имеет правильную форму. Если все проверки проходят успешно, цикл продолжает поиск нового правила. В листинге 8.6 представлена функция `skipWhiteSpace`, которая используется, чтобы пропустить комментарии, пробелы и другой текст, не входящий в правило.

Листинг 8.6. Функция-анализатор файла (`parseFile`)

```
int parseFile( char *filename )
{
  FILE *fp;
  char *file, *cur;
  int fail = 0;

  extern int debug;

  file = (char *)malloc(MAX_FILE_SIZE);

  if (file == NULL) return -1;

  fp = fopen(filename, "r");

  if (fp == NULL) {
    free(file);
    return -1;
  }

  fread( file, MAX_FILE_SIZE, 1, fp);

  cur = &file[0];

  while (1) {

    /* Разбираем правило */
    /* Ищем начало правила (начинается с "(defrule" */

    cur = strstr( cur, "(defrule" );
    if (cur == NULL) {
      fail = 1;
      break;
    }

    if (!strncmp(cur, "(defrule", 8)) {
      int i=0;

      cur+=9;
```

```
    while (*cur != 0x0a) {
      ruleSet[ruleIndex].ruleName[i++] = *cur++;
    }
    ruleSet[ruleIndex].ruleName[i++] = 0;
    cur = skipWhiteSpace( cur );

    /* Разбираем предпосылку */
    cur = parseAntecedent( cur, &ruleSet[ruleIndex] );

    if (cur == NULL) {
      fail = 1;
      break;
    }

    /* Далее должен быть "=>" */
    if (!strncmp(cur, "=>", 2)) {
      cur = skipWhiteSpace( cur+2 );
      /* Разбираем последствия правила */
      cur = parseConsequent( cur, &ruleSet[ruleIndex] );
      if (cur == NULL) {
        fail = 1;
        break;
      }

      /* Убедимся, что правило закрыто */
      if (*cur == ')') {
        cur = skipWhiteSpace( cur+1 );
      } else {
        fail = 1;
        break;
      }

    } else {
      fail = 1;
      break;
    }

    ruleSet[ruleIndex].active = 1;
    ruleIndex++;

  } else {

    break;

  }

}
```

```
  if (debug) {
    printf("Found %d rules\n", ruleIndex);
  }

  free( (void *)file );

  fclose(fp);

  return 0;
}
```

Анализ предпосылок и последствий требует использования двух функций-анализаторов (одной функции для предпосылок и одной для последствий) – см. листинг 8.7. Для анализа текущего пункта файла каждый анализатор задействует функцию `parseElement`.

Листинг 8.7. Функции-анализаторы предпосылок и последствий

```
char *parseAntecedent( char *block, ruleType *rule )
{
  while (1) {

    block = skipWhiteSpace( block );

    if (*block == '(') {

      block = parseElement( block, &rule->antecedent );

    } else break;
  }
  return block;
}

char *parseConsequent( char *block, ruleType *rule )
{
  while (1) {

    block = skipWhiteSpace( block );

    if (*block == '(') {

      block = parseElement(block, &rule->consequent);

    } else break;

  }

  return block;
}
```

Обратите внимание на использование команды `skipWhiteSpace` до анализа элементов в обеих функциях. Она позволяет вводить комментарии после каждой предпосылки или последствия.

Функция `parseElement` извлекает факт из файла, учитывая круглые скобки (в зависимости от типа факта может быть использована одна пара скобок или две). Круглые скобки применяются здесь в качестве маркеров элемента, который уже был проанализирован целиком (и сохранен). После завершения анализа элемент добавляется в цепочку предпосылок или последствий (листинг 8.8).

Листинг 8.8. Функция `parseElement`

```
char *parseElement( char *block, memoryElementType **met )
{
  memoryElementType *element;
  int i=0;
  int balance = 1;

  element = (memoryElementType*)malloc(sizeof(memoryElementType));
  element->element[i++] = *block++;

  while (1) {

    if (*block == 0) break;

    if (*block == ')') balance—;
    if (*block == '(') balance++;

    element->element[i++] = *block++;

    if (balance == 0) break;
  }

  element->element[i] = 0;
  element->next = 0;

  if (*met == 0) *met = element;
  else {
    memoryElementType *chain = *met;
    while (chain->next != 0) chain = chain->next;
    chain->next = element;
  }

  return block;
}
```

Наконец, функция `skipWhiteSpace` завершает анализ файла. Она пропускает все неактивные символы правила и возвращает новое положение во входном файле, с которого будет продолжен анализ (листинг 8.9).

Листинг 8.9. Функция, которая используется для пропуска неактивных символов

```
char *skipWhiteSpace( char *block )
{
  char ch;

  while (1) {

    ch = *block;

    while ((ch != '(') && (ch != ')') && (ch != '=') &&
           (ch != 0) && (ch != ';')) {
      block++;
      ch = *block;
    }

    if (ch == ';') {

      while (*block++ != 0x0a);

    } else break;

  }

  return block;
}
```

После завершения анализа в массив `ruleSet` будут добавлены правила из файла. От кода, указанного в листинге 8.5, программа переходит к функции `interpret` (листинг 8.10). Данная функция двигается по списку активных правил и вызывает функцию `checkRule`, чтобы проверить соответствие предпосылок правила текущему состоянию рабочей памяти. Функция `checkRule` возвращает единицу, если правило выполнено, в противном случае возвращается ноль. Если правило выполнено, программа выходит из цикла и начинает заново с начала набора правил.

Листинг 8.10. Функция `interpret`

```
void interpret( void )
{
  int rule;
  int fired = 0;

  extern int checkRule( int );
  extern int debug;

  for (rule = 0 ; rule < MAX_RULES ; rule++) {

    fired = 0;

    if (ruleSet[rule].active) {
```

```
        fired = checkRule( rule );

     /* Если правило изменяет состояние рабочей памяти, то
      * выходим. Иначе ищем другое правило
      */
        if (fired) break;

      }

    }

    if (debug) {
      if (fired) printf("Fired rule %s (%d)\n",
                        ruleSet[rule].ruleName, rule);
    }

    return;
  }
```

Функция `checkRule` является довольно короткой и простой, но при этом отвечает за самую сложную часть работы системы, основанной на правилах (листинг 8.11). Она вызывает функцию `checkPattern`, чтобы попытаться найти соответствие между предпосылками текущего правила и фактами в рабочей памяти.

Листинг 8.11. Функция `checkRule`

```
int checkRule( int rule )
{
  int fire = 0;
  char arg[MEMORY_ELEMENT_SIZE]={0};

  extern int fireRule( int, char * );

  fire = checkPattern(rule, arg);

  if (fire == 1) {

    fire = fireRule( rule, arg );

  }

  return fire;
}
```

Функция `checkPattern` возвращает единицу при успешности сопоставления. Если найдено совпадение, то правило выполняется с помощью функции `fireRule`. Обратите внимание, что еще раз используется переменная, отвечающая за выполнение правила для результата вызова функции `fireRule`. Как уже говорилось ранее, данный результат зависит от того, было ли выполнено рассматриваемое правило. Когда программа находит соответствие для правила

(с помощью функции checkPattern), разрешается его выполнение. Если правило изменяет рабочую память, это значит, что оно действительно выполнено. Если правило не изменяет рабочую память, программа предполагает, что оно не было выполнено, и продолжает поиск, выдавая значение, равное нулю. Данное действие позволяет избежать повторного выполнения правила, поскольку оно производится перед анализом других правил, предпосылки которых соответствуют фактам в рабочей памяти.

Функция checkPattern отыскивает соответствия между предпосылками текущего правила и фактами в рабочей памяти. Хотя эта задача концептуально проста, давайте предположим, что произойдет, если правило будет выглядеть таким образом:

```
(defrule check-fully-charged
(fully-charged?)
=>
(trickle-charge ?)
)
```

Вспомните, что в этом случае первый объект является предпосылкой (fully-charged), а второй – последствием. Поэтому если рабочая память выглядит так:

```
(fully-charged battery-1)
```

то после выполнения правила будет добавлен в рабочую память следующий факт:

```
(trickle-charge battery-1)
```

Вследствие этого при поиске соответствия для правила программа сначала проверяет, присутствует ли во втором объекте предпосылки символ «?». Если да, то она сохраняет все первые объекты правил, чтобы использовать их при следующих совпадениях правил.

Функция checkPattern показана в листинге 8.12.

Листинг 8.12. Алгоритм поиска соответствия для правила. Функция checkPattern

```
int checkPattern( int rule, char *arg )
{
  int ret=0;
  char term1[MEMORY_ELEMENT_SIZE+1];
  char term2[MEMORY_ELEMENT_SIZE+1];
  memoryElementType *antecedent = ruleSet[rule].antecedent;

    while (antecedent) {

    sscanf( antecedent->element, "(%s %s)", term1, term2);
    if (term2[strlen(term2)-1] == ')') term2[strlen(term2)-1] = 0;

    if (term2[0] == '?') {
      int i;
      char wm_term1[MEMORY_ELEMENT_SIZE+1];
      char wm_term2[MEMORY_ELEMENT_SIZE+1];
```

```
    for (i = 0 ; i < MAX_MEMORY_ELEMENTS ; i++) {

      if (workingMemory[i].active) {

        sscanf( workingMemory[i].element, "(%s %s)",
                wm_term1, wm_term2 );
        if (wm_term2[strlen(wm_term2)-1] == ")")
                wm_term2[strlen(wm_term2)-1] = 0;

        if (!strncmp(term1, wm_term1, strlen(term1)))
          addToChain(wm_term2);

      }
    }
  }

  antecedent = antecedent->next;

}

 /* Теперь мы имеем строку для подстановки. Проверяем правило,
  * подставляя строку в случае необходимости
  */

do {

  memoryElementType *curRulePtr, *temp;

  curRulePtr = ruleSet[rule].antecedent;

  while (curRulePtr) {

    sscanf( curRulePtr->element, "(%s %s)", term1, term2 );
    if (term2[strlen(term2)-1] == ')') term2[strlen(term2)-1] = 0;

    if (!strncmp( term1, "true", strlen(term1))) {
      ret = 1;
      break;
    } else {
      if ((term2[0] == '?') && (chain)) {
       strcpy(term2, chain->element);
    }
     }

    ret = searchWorkingMemory( term1, term2 );

    if (!ret) break;

    curRulePtr = curRulePtr->next;
  }
```

```
    if (ret) {

      /* Очищаем цепочку из строк подстановки */
      while (chain) {
        temp = chain;
        chain = chain->next;
        free(temp);
      }

      strcpy(arg, term2);

    } else {

      if (chain) {
        temp = chain;
        chain = chain->next;
        free(temp);
      }

    }

  } while (chain);

  return ret;
}
```

Первая часть функции `checkPattern` проходит по списку предпосылок текущего правила и отыскивает все элементы, во втором объекте которых имеется символ «?». Затем выполняется поиск первого объекта данного элемента в рабочей памяти. Когда объект найден, второй объект факта сохраняется с помощью функции `addToChain`. Это называется строкой замещения (действительное значение в рабочей памяти для символа «?» во втором объекте). После завершения цикла программа получает цепочку элементов, представляющих второй объект фактов в памяти, которые соответствуют правилам с символом «?» во втором объекте. С помощью этой цепочки второй цикл проходит по списку предпосылок текущего правила, используя все сохраненные вторые объекты в списке цепочки. После того как предпосылка была полностью проверена с помощью данного элемента цепочки и в рабочей памяти было найдено совпадение для правила, сохраненный второй объект (заданный текущим элементом в цепочке) копируется и возвращается функцией. Аргумент (например, `battery-1` в предыдущем примере) используется позже при работе с последствиями правила.

Функция `addToChain`, которая рассматривалась ранее, строит цепочку объектов, чтобы использовать их при поиске соответствия (листинг 8.13). Функция `searchWorkingMemory` (листинг 8.14) пытается найти соответствие между

двумя объектами предпосылки и фактом в рабочей памяти. Найденной соответствие возвращается (в противном случае возвращается ноль).

Листинг 8.13. Построение цепочки элементов с помощью функции `addToChain`

```
void addToChain( char *element )
{
  memoryElementType *walker, *newElement;;

  newElement =
   (memoryElementType *)malloc(sizeof(memoryElementType));

  strcpy( newElement->element, element );

  if (chain == NULL) {
    chain = newElement;
  } else {
    walker = chain;
    while (walker->next) walker = walker->next;
    walker->next = newElement;
  }

  newElement->next = NULL;
}
```

Листинг 8.14. Поиск совпадений между предпосылками и фактами в рабочей памяти с помощью функции `searchWorkingMemory`

```
int searchWorkingMemory( char *term1, char *term2 )
{
  int ret = 0;
  int curMem = 0;
  char wm_term1[MEMORY_ELEMENT_SIZE+1];
  char wm_term2[MEMORY_ELEMENT_SIZE+1];

  while (1) {

    if (workingMemory[curMem].active) {

      /* Читаем элемент из рабочей памяти */
      sscanf(workingMemory[curMem].element, "(%s %s)",
            wm_term1, wm_term2);
      if (wm_term2[strlen(wm_term2)-1] == ')')
            wm_term2[strlen(wm_term2)-1] = 0;

      if ((!strncmp(term1, wm_term1, strlen(term1))) &&
          (!strncmp(term2, wm_term2, strlen(term2)))) {

        ret = 1;
```

```
          break;

        }
      }
      curMem++;
      if (curMem == MAX_MEMORY_ELEMENTS) {
        break;
      }

    }

    return ret;
}
```

Если было найдено соответствие между правилом и рабочей памятью, функция `checkPattern` возвращает единицу в функцию `checkRule` (листинг 8.11). Правило может быть исполнено, и функция `fireRule` вызывается с индексом, указывающим, какое правило следует выполнить, а также с аргументом, сохраненным в функции `checkPattern`. Вспомните, что это значение представляет собой объект с символом «?», для которого было найдено соответствие в рабочей памяти.

Выполнение правила намного проще, чем поиск соответствия для него в рабочей памяти. Чтобы выполнить правило, программа проходит по списку последствий для него и выполняет команду, закодированную каждым элементом (листинг 8.15).

Листинг 8.15. Выполнение правила с помощью функции `fireRule`

```
int fireRule( int rule, const char *arg )
{
  int ret;
  memoryElementType *walker = ruleSet[rule].consequent;
  char newCons[MAX_MEMORY_ELEMENTS+1];

  while (walker) {

    if (!strncmp(walker->element, "(add", 4)) {

      constructElement( newCons, walker->element, arg );

      ret = performAddCommand( newCons );

    } else if (!strncmp(walker->element, "(delete", 7)) {

      constructElement( newCons, walker->element, arg );
      ret = performDeleteCommand( newCons );
```

```
    } else if (!strncmp(walker->element, "(disable", 8)) {
      ruleSet[rule].active = 0;
      ret = 1;
    } else if (!strncmp(walker->element, "(print", 6)) {

      ret = performPrintCommand( walker->element );

    } else if (!strncmp(walker->element, "(enable", 7)) {

      ret = performEnableCommand( walker->element );

    } else if (!strncmp(walker->element, "(quit", 5)) {

      extern int endRun;

      endRun = 1;

    }

    walker = walker->next;

  }

  return ret;
}
```

Функция `fireRule` проходит по списку всех последствий и декодирует команды, которые в них содержатся. Она сравнивает значение второго объекта последствия с известными типами команд и в случае совпадения вызывает специальную функцию, которая выполняет данную команду. Существует два случая, когда происходит изменение рабочей памяти (добавление и удаление), поэтому следует правильно сформировать факт. Вспомните, что если для предпосылки имеется нечеткое совпадение (например, `(fast-charge ?)`), то используется переданный аргумент (`arg`), чтобы сформировать факт, который будет применен к рабочей памяти. Эта функция показана в листинге 8.16.

Листинг 8.16. Формирование нового факта с помощью функции `constructElement`

```
void constructElement( char *new, const char *old, const char *arg )
{

  /* Найти шаблон */
  old++;
  while (*old != '(') old++;

  while ((*old != 0) && (*old != '?')) *new++ = *old++;

  /* Это полное правило (нет символа '?') */
```

```
  if (*old == 0) {

    *(--new) = 0;
    return;
  } else {

    /* Копируем в arg */
    while (*arg != 0) *new++ = *arg++;
    if ( *(new-1) != ")") *new++ = ')';
    *new = 0;

  }

  return;
}
```

Функция `constructElement` просто создает новый факт на основании модели факта (последствия) и переданного аргумента. Если модель факта включает символ «?», то переданный аргумент заменяет символ «?» и возвращается в виде нового факта. Поэтому если использовался аргумент `battery-1`, а последствие имело вид `(fast-charge ?)`, то новый факт будет выглядеть как `(fast-charge battery1)`.

Теперь рассмотрим функции, которые выполняют доступные команды. Первая команда, `add`, добавляет новый факт в рабочую память (листинг 8.17). Команда `add` для нечеткого соответствия имеет следующий формат:

```
(add (fast-charge ?))
```

Также она может отображать абсолютный факт, например:

```
(add (fast-charge battery-2))
```

Листинг 8.17. Добавление нового факта в рабочую память

```
int performAddCommand( char *mem )
{
  int slot;

   /* Проверяем, что этот элемент не находится в рабочей памяти */
   for (slot = 0 ; slot < MAX_MEMORY_ELEMENTS ; slot++) {

    if (workingMemory[slot].active) {

      if (!strcmp( workingMemory[slot].element, mem )) {
        /* Файл в памяти уже есть - выходим */
        return 0;

      }

    }

  }
```

```
  /* Добавляем элемент в рабочую память */

  slot = findEmptyMemSlot();

  if (slot < MAX_MEMORY_ELEMENTS) {

    workingMemory[slot].active = 1;
    strcpy( workingMemory[slot].element, mem );

  } else {
    assert(0);
  }

  return 1;
}
```

Команда `add` сначала проверяет, существует ли данный факт в рабочей памяти. Если да, то команда возвращает ноль (значение ошибки) и завершает работу. В противном случае находится пустая ячейка (с помощью функции `findEmptyMemSlot`, см. листинг 8.18) и факт копируется в рабочую память. Здесь возвращается единица; это означает, что рабочая память изменилась. Обратите внимание, что статус данной функции постоянно возвращается в функцию `ruleFire`.

Листинг 8.18. Поиск пустой ячейки в рабочей памяти

```
int findEmptyMemSlot( void )
{
  int i;
  for (i = 0 ; i < MAX_MEMORY_ELEMENTS ; i++) {
    if (!workingMemory[i].active) break;
  }
  return i;
}
```

Команда `delete` удаляет факт из рабочей памяти (листинг 8.19). Она пытается отыскать соответствие между сформированным фактом и фактом в рабочей памяти. Если факт в рабочей памяти найден, то он будет удален (станет неактивным; при этом строка факта обнуляется). Команда `delete` выглядит следующим образом:

```
(delete (fast-charge ?))
```

Листинг 8.19. Удаление факта из рабочей памяти

```
int performDeleteCommand( char *mem )
{
  int slot;
  int ret = 0;
  char term1[MEMORY_ELEMENT_SIZE+1];
  char term2[MEMORY_ELEMENT_SIZE+1];
```

```
  char wm_term1[MEMORY_ELEMENT_SIZE+1];
  char wm_term2[MEMORY_ELEMENT_SIZE+1];

  sscanf( mem, "(%s %s)", term1, term2 );

  for ( slot = 0 ; slot < MAX_MEMORY_ELEMENTS ; slot++ ) {

    if ( workingMemory[slot].active ) {
      sscanf(  workingMemory[slot].element, "(%s %s)",
               wm_term1, wm_term2 );

      if (!strncmp(term1, wm_term1, strlen(term1)) &&
          !strncmp(term2, wm_term2, strlen(term2))) {

        workingMemory[slot].active = 0;
        bzero( workingMemory[slot].element, MEMORY_ELEMENT_SIZE );

        ret = 1;

      }

    }

  }

  return ret;
}
```

Как и для команды `performAddCommand`, результат этой функции постоянно возвращается в функцию `ruleFire`.

Команда `print` выводит второй объект факта на печать (листинг 8.20). Он представляет собой строку, заключенную в двойные кавычки. Функция `performPrintCommand` ищет первый символ «"» и выводит все символы до следующего символа «"». Факты вывода строк на печать имеют следующий формат:

```
(print ("this is a test."))
```

Листинг 8.20. Вывод строки на печать

```
int performPrintCommand( char *element )
{
  char string[MAX_MEMORY_ELEMENTS+1];
  int i=0, j=0;

  /* Find initial '"' */
  while ( element[i] != '"') i++;
  i++;

  /* Copy until we reach the end */
  while ( element[i] != '"') string[j++] = element[i++];
  string[j] = 0;
```

```
  printf("%s\n", string);

  return 1;
}
```

Команда `enable` используется для запуска таймера (листинг 8.21). Факт для запуска таймера имеет следующий формат:

```
(enable (timer N T))
```

Здесь символ «N» представляет собой индекс таймера, а символ «T» – время включения таймера.

Листинг 8.21. Включение таймера

```
int performEnableCommand( char *element )
{
  char *string;
  int timer, expiration;

  void startTimer( int, int );

  string = strstr( element, "timer" );

  sscanf( string, "timer %d %d", &timer, &expiration );

  startTimer( timer, expiration );

  return 1;
}
```

Две последние команды, `disable` и `quit`, выполняются в функции `fireRule` (листинг 8.15). Команда `disable` используется, чтобы запретить реализацию правила, а `quit` завершает работу системы. Они выглядят следующим образом:

```
(disable (self))
(quit null)
```

Работа таймеров осуществляется с помощью функции `processTimers` (листинг 8.22), которая сокращает время завершения работы для активных таймеров до тех пор, пока оно не достигает значения активации (в данном случае это ноль). Для отработки события от таймера вызывается функция `fireTimer` (листинг 8.23).

Листинг 8.22. Обработка списка таймеров

```
void processTimers( void )
{
  int i;

  for (i = 0 ; i < MAX_TIMERS ; i++) {

    if (timers[i].active) {
```

```
        if (--timers[i].expiration == 0) {

          fireTimer( i );

        }
      }
    }

    return;
}
```

Вспомните, что включение таймера подразумевает добавление в рабочую память факта, который будет играть роль триггера для активного правила. При включении таймера в рабочую память (символ `X` отображает индекс таймера) добавляется факт, например, `(timer-triggered X)`. Функция добавления (`performAddCommand`) используется для помещения факта триггера в рабочую память. Предполагается, что другое правило в наборе будет иметь предпосылку для этого факта, чтобы обработать поступившее от таймера событие.

Листинг 8.23. Включение таймера

```
int fireTimer( int timerIndex )
{
  int ret;
  char element[MEMORY_ELEMENT_SIZE+1];

  extern int performAddCommand( char *mem );

  sprintf( element, "(timer-triggered %d)", timerIndex );

  ret = performAddCommand( element );

  timers[timerIndex].active = 0;

  return ret;
}
```

Запуск таймера осуществляет команда `enable`. Функция, выполняющая эту команду, приводится в листинге 8.24, чтобы объединить все функции таймера в одном файле.

Листинг 8.24. Запуск таймера

```
void startTimer( int index, int expiration )
{
  timers[index].expiration = expiration;
  timers[index].active = 1;

  return;
}
```

Построение базы правил

Построение базы правил является сложной задачей, но ее можно упростить с помощью нескольких способов.

Правила должны быть выстроены в логическом порядке на основании их контекста. Например, правила, имеющие схожие предпосылки, могут быть сгруппированы вместе, поскольку они анализируют одни и те же или сходные факты из рабочей памяти. Аналогично, правила со схожими последствиями тоже могут быть сгруппированы вместе, так как ведут к схожим действиям (на основании такого же анализа).

При работе со сложными базами правил вы можете использовать алгоритмы ступени вместе с объектами. Например, рассмотрим два правила, представленные в листинге 8.25.

Листинг 8.25. Пример правил ступени

```
(defrule stage1-check
     (stage stage1)
     (battery-temp low)
=>
)
     (add (check battery-temp))
     (delete (stage stage1))
     (add (stage stage2))
)

(defrule stage2-check
     (stage stage2)
     (battery-temp low)
=>
     (add (check battery-pressure))
     (delete (stage stage2))
)
```

Факт `(stage stageX)` был введен в качестве маркера. Вместо того чтобы выполнять роль обычной информации в рабочей памяти, эти факты являются ограничителями для правил. Ни одно из правил не рассматривается, если не будет найдено соответствие для правила ступени.

Наконец, ни один метод по эффективности не сравнится с тестированием. Для полного тестирования должен существовать способ внести в систему стимул. Помимо простого введения фактов с помощью правила `init` вы можете использовать правила-триггеры для произвольного добавления фактов. Вспомните (листинг 8.2) использование таймеров для эмуляции устойчивости к ошибкам.

Область применения

Системы, основанные на правилах, часто используются в качестве экспертных. Однако, как показывает данная глава, эти системы могут применяться и для

решения простых задач, связанных с программированием, когда знание о проблеме кодируется с помощью языка программирования. Кроме того, этот алгоритм позволяет создавать системы управления, устойчивые к ошибкам. Одно из основных отличий между двумя подходами заключается в том, что при стандартном программировании мы кодируем определенное знание с помощью имеющихся конструкций. А системы, основанные на правилах, требуют сформировать знание в виде правил, которые определяют предпосылки и последствия. В некоторых случаях это приводит к тому, что информация становится проще и воспринимается легче.

Недостатки систем, основанных на правилах

Самый большой недостаток данных систем заключается в том, что на поиск соответствия между правилом и фактами в рабочей памяти часто тратится очень много времени. Вспомните, что существует не только безусловное соответствие между правилами и фактами, но и варианты, основанные на неполных совпадениях. Поэтому вместо того, чтобы просто искать абсолютное соответствие между правилом и фактами в рабочей памяти, придется выполнить ряд итераций с каждым правилом. Если соответствие не было найдено, поиск будет продолжаться для следующего правила.

Проблему можно решить с помощью *алгоритма Рете*. Идея состоит в том, чтобы поделить промежуточную информацию между правилами и сократить количество соответствий, проверку которых предстоит выполнить. Это реализуется с помощью ацикличного графа, отражающего предпосылки правил. Например, если два правила имеют одинаковый объект в предпосылке, каждое правило разделит узел в графе для этого элемента. От данного узла будет вестись построение остальных элементов соответствующих предпосылок. Таким образом, поиск соответствия для указанного элемента выполняется только один раз, что, возможно, и приводит к увеличению количества соответствий для правила, однако, расчет начального соответствия не повторяется.

Итоги

В этой главе вы изучили одну из ранних архитектур искусственного интеллекта – систему, основанную на правилах. Здесь рассматривалась архитектура прямого логического вывода, в которой применяется метод дедукции. Она очень проста и может использоваться для построения разнообразных систем, включая добавление в другие системы для управления средой. Описывался не только классический пример ее реализации (набор правил ZOOKEEPER), но и пример системы управления, устойчивой к ошибкам. Рассматриваемые методы применяются в системах, которые взаимодействуют с пользователем (например, классических диагностических системах), и даже в интегрированных системах управления.

Литература и ресурсы

1. Уинстон П. Н. Искусственный разум (Winston P. H. Artificial Intelligence, 3rd ed. Addison Wesley, 1993).
2. Форги С. Л. Рете: быстрый алгоритм для решения проблемы поиска соответствия между многими моделями для большого количества объектов (Forgy C. L. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem // Artificial Intelligence, 19:17–37, 1982).s

Глава 9. Введение в нечеткую логику

Логическую систему, о которой пойдет речь в данной главе, в 1963 г. разработал Лотфи Заде (Lotfi Zadeh), чтобы преодолеть несоответствие между системами настоящего мира и их компьютерными воплощениями. Булевы значения истина-ложь воплощают в программах уникальные элементы системы с двумя значениями. В реальности мы редко имеем дело с такими системами, поскольку многие условия могут быть частично истинными, а частично ложными (или теми и другими одновременно). *Нечеткая логика* (Fuzzy logic) была придумана для того, чтобы позволить программам работать в диапазоне различных степеней истины. Вместо двоичных систем, отображающих только истину и ложь, были введены степени истины, которые действуют в диапазоне от 0,0 до 1,0 включительно.

Введение

Чтобы полностью понять принцип, предложенный доктором Заде, рассмотрим несколько примеров, которые дают больше информации о нечеткой логике и ее воплощениях.

Пример нечеткой логики

Преимущества нечеткой логики проявляются, когда система анализируется с помощью лингвистики. В качестве примера рассмотрим алгоритм QoS (алгоритм качества обслуживания), который управляет выходом данных по определенному каналу связи. Его задача состоит в том, чтобы обеспечить для программы постоянную пропускную способность канала. Если программа пытается отправить слишком много данных, необходимо снизить скорость передачи. С точки зрения управления можно выделить три элемента. Первый элемент – это скорость поступления данных из программы, второй элемент – измеряемый коэффициент использования канала, а третий – шлюз, который контролирует поток данных между программой и каналом связи (рис. 9.1).

Задача шлюза заключается в том, чтобы определять, когда и сколько пакетов данных может быть пропущено через канал. Человек оценивает данную задачу с помощью простых правил:

«Если коэффициент использования канала программой слишком высок, для нее следует уменьшить скорость пропуска пакетов через шлюз».

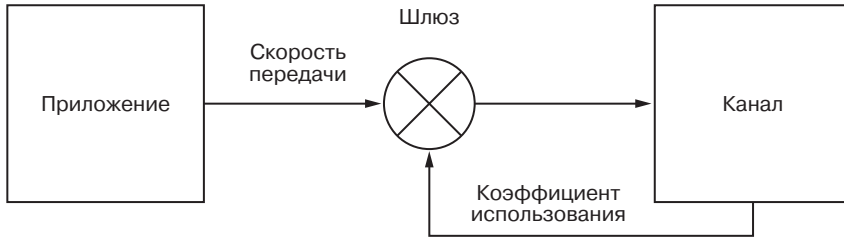


Рис. 9.1. Алгоритм QoS управления передачей данных

И наоборот:

«Если коэффициент использования канала программой слишком низок, для нее следует увеличить скорость пропуска пакетов через шлюз».

Эти правила показывают, что существует «мертвая зона» между высоким и низким использованием пропускной способности.

Вопрос заключается в том, как определить понятия «высоко» или «низко» для программы? Нечеткая логика определяет эти понятия с помощью функций принадлежности (рис. 9.2).

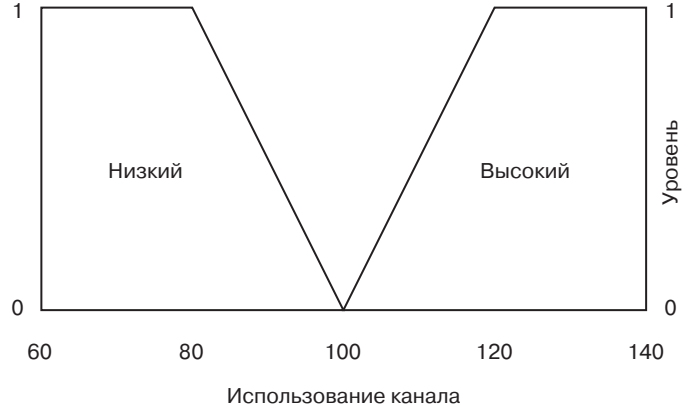


Рис. 9.2. Функция принадлежности для скорости пропуска данных

Функции принадлежности

Функция принадлежности (Membership function) определяет степень принадлежности (степень истины) заданного условия. Алгоритм QoS позволяет получить функцию принадлежности для использования канала (рис. 9.2).

На рис. 9.2 представлено два сегмента, между которыми находится третий (цель). Низкое значение может быть задано, как показано в выражении 9.1:

```
m_low(x) = {

0,                 если rate(x) >= 100
(100-rate(x))/20   если rate(x) > 80 и rate(x) < 100
1,                 если rate(x) <= 80.
}
```

(9.1)

Высокое значение может быть задано так, как показано в выражении 9.2:

```
m_high(x) = {

0,                 если rate(x) <= 100
(rate(x)-100)/20   если rate(x) > 100 и rate(x) < 120
1,                 если rate(x) >= 120.
}
```

(9.2)

Нечеткое управление

Табл. 9.1 иллюстрирует функции принадлежности. В ней представлены значения скорости прохождения данных, а также их степень принадлежности для двух функций.

Таблица 9.1. Пример степени принадлежности

Скорость передачи пакетов	m_low(x)	m_high(x)
10	1,0	0,0
85	0,75	0,0
90	0,5	0,0
95	0,25	0,0
100	0,0	0,0
105	0,0	0,25
110	0,0	0,5
115	0,0	0,75
180	0,0	1,0

С помощью двух функций принадлежности можно задать степень принадлежности для определенной скорости пропуска данных. Как же с помощью этой степени управлять скоростью пропуска пакетов данных и сохранять постоянный уровень сервиса? Очень простой механизм позволяет включить степень принадлежности в коэффициент, который используется алгоритмом шлюза.

Функция определяет, сколько пакетов данных может пройти через шлюз в заданный промежуток времени (например, за одну секунду). Каждую секунду количество пакетов данных, которым разрешается пройти через шлюз, регулируется

таким образом, чтобы поддерживалось постоянство. Если коэффициент использования слишком высок, количество пакетов уменьшается. В противном случае количество пакетов увеличивается. Вопрос в том, насколько?

Механизм состоит в том, чтобы использовать степень принадлежности в качестве коэффициента, который применяется к дельте от количества пакетов данных. Рассмотрим уравнение 9.3:

$$\text{rate} = \text{rate} + (\text{m\_low}(\text{rate}) \times \text{pdelta}) - (\text{m\_high}(\text{rate}) \times \text{pdelta}) \quad (9.3)$$

Переменная `rate` – это текущее количество пакетов, которым разрешается проходить через шлюз в заданное время (одну секунду). Константа `pdelta` – изменяемый параметр, определяющий максимальное количество пакетов, которое может быть добавлено в коэффициент или удалено из него. Функции принадлежности представляют степень принадлежности, которая применяется в качестве коэффициента для алгоритма изменения скорости.

Рассмотрим несколько примеров, для которых используется константа `pdelta`, равная 10. Хотя коэффициент является нецелым числом, для настройки алгоритма он будет изменен.

Если значение `rate` равно 110, как при этом алгоритм изменит коэффициент для механизма шлюза? Используя уравнение 9.3, получаем:

$$\text{rate} = 110 + (0 \times 10) - (0{,}5 \times 10) = 105.$$

Если при следующей итерации программа продолжает работать с коэффициентом 110, будет получен следующий результат:

$$\text{rate} = 105 + (0 \times 10) - (0{,}25 \times 10) = 102{,}5.$$

Процесс будет продолжаться до тех пор, пока не будет получено значение, равное 100, которое указывает, что дальнейшие изменения коэффициента производиться не будут.

Если коэффициент был невысоким, уравнение 9.3 примет следующий вид:

$$\text{rate} = 80 + (1 \times 10) - (0 \times 10) = 90.$$

Таким образом, при использовании степени принадлежности в качестве коэффициента изменения скорости получается простой механизм управления.

Визуальный пример нечеткой логики

Рассмотрим другой пример использования нечеткой логики. В простом двумерном мире определены два агента, один является хищником, другой – жертвой. Задача заключается в том, чтобы создать набор функций принадлежности, которые позволят хищнику выследить жертву и поймать ее. Для управления функции принадлежности будут использовать угол ошибки (отображающий разницу между текущим направлением движения хищника и текущим направлением движения жертвы).

График функции принадлежности для хищника показан на рис. 9.3. Задаются семь отдельных групп, которые идентифицируют степень ошибки и степень необходимой корректировки.

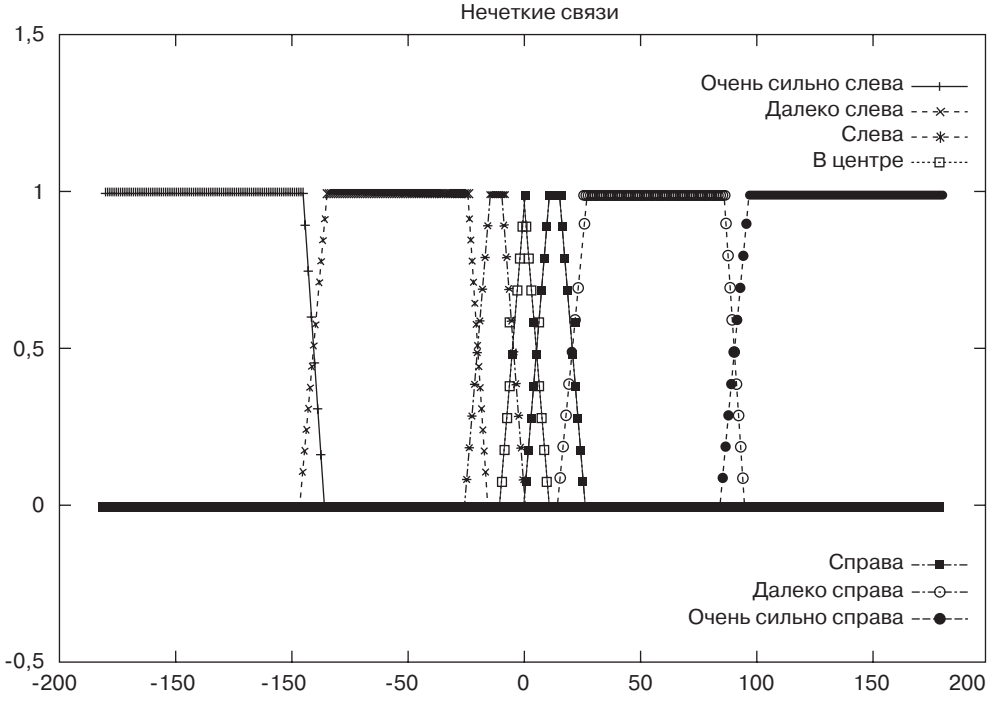


Рис. 9.3. Функции принадлежности для хищника

Центральная группа является единственной группой, в которой направление движения хищника не изменяется. Левая и правая группы – это, соответственно, [+1, –1]. Дальше находятся группы [+8, –8] и, наконец, группы [+15, –15]. Если угол ошибки хищника попадает в область определения какой-либо функции принадлежности, то выполняется корректировка направления движения хищника, и процесс повторяется. При каждом действии хищник перемещается на одну ячейку в скорректированном направлении.

На рис. 9.4 представлен график действий хищника, определяемых функцией принадлежности с рис. 9.3.

Хищник начинает движение в ячейке с координатами [100, 100], а жертва – в произвольной ячейке (здесь [84, 30]). Жертва перемещается по диагонали (под углом в 45°) от начального положения. Как показано, хищник изменяет направление своего движения, чтобы поймать жертву (приблизиться к ней на расстояние в 5 ячеек).

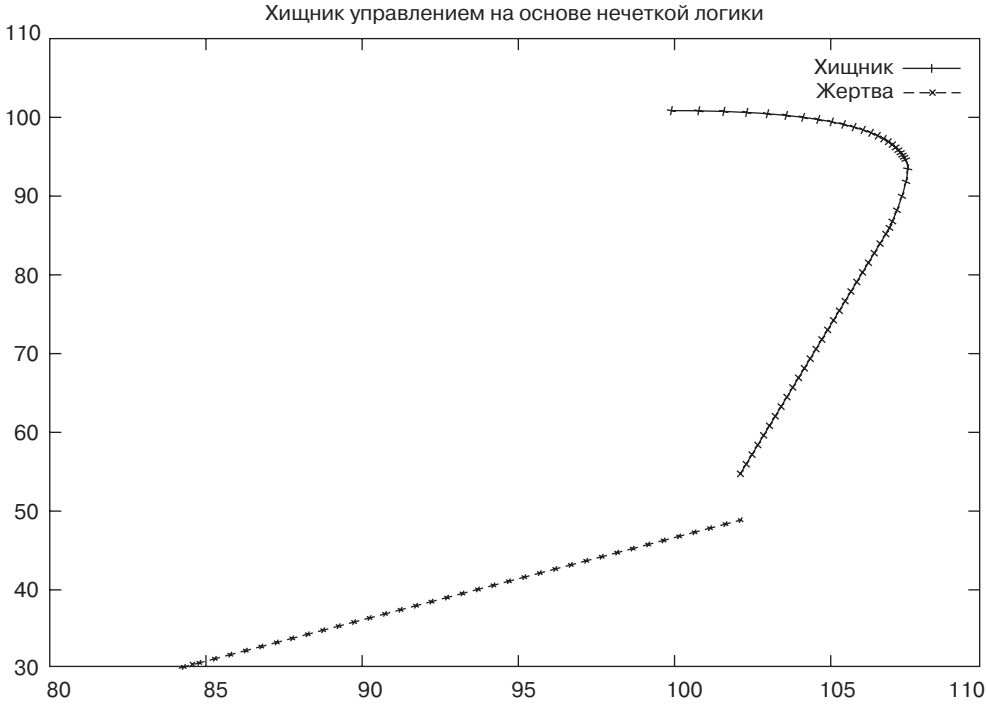


Рис. 9.4. График движения хищника и жертвы

Аксиомы нечеткой логики

Аналогично булевой логике, нечеткая логика имеет набор базовых операторов. Они совпадают с булевыми, но действуют по-другому. Аксиомы нечеткой логики представлены в табл. 9.2.

Таблица 9.2. Аксиомы нечеткой логики

Оператор	Формула вычисления
Truth(A OR B)	MIN(truth(A), truth(B))
Truth(A AND B)	MAX(truth(A), truth(B))
Truth(NOT A)	1,0 – truth(A)

Эти операторы обеспечивают основу для операций нечеткой логики. Пример условия:

```
if (m_warm(board_temperature) AND m_high(fan_speed)) then …
```

Эта форма не очень удобна для чтения, но помогает точнее оценивать состояние системы.

Функции ограничения

Важным элементом систем нечеткой логики являются *ограничители* (Hedge) функций принадлежности. Они предоставляют системе нечеткой логики дополнительные лингвистические конструкции при описании правил и позволяют поддерживать математическое постоянство. Рассмотрим функции-ограничители `VERY` и `NOT_VERY`. Они используются вместе с функциями принадлежности и изменяют их значения в зависимости от поставленных задач. Функции ограничения показаны в уравнениях 9.4 и 9.5:

$$VERY(m\_x) = m\_x^2 \tag{9.4}$$

$$NOT\_VERY(m\_x) = m\_x^{0,5} \tag{9.5}$$

Рассмотрим использование этих функций вместе с функцией принадлежности, показанной на рис. 9.2. При коэффициенте скорости 115 значение `m_high` будет равно 0,75. Если применить функцию ограничения `VERY` к функции принадлежности (`VERY(m_high(rate)`), то полученное значение будет составлять 0,5625 (другими словами, недостаточно большое значение). Если коэффициент равен 119, `m_high` будет составлять 0,95. При применении функции `VERY` к этому значению получаем результат 0,903 (или большое значение).

Зачем использовать нечеткую логику

Нечеткая логика имеет огромное значение при создании программного обеспечения, которое использует вместо четких значений приблизительные. Поскольку человеческая логика сама по себе является приблизительной, при проектировании систем легче использовать нечеткую логику, чем детерминированные алгоритмы.

Пример использования

Рассмотрим применение нечеткой логики на примере простой модели зарядного устройства для батарей (опустив ряд деталей).

Зарядное устройство работает в среде, где существует напряжение заряда (например, от солнечных батарей), и нагрузка. Напряжение позволяет заряжать батарею, в то время как нагрузка ее разряжает. Зарядное устройство имеет два режима работы: режим подзарядки и режим быстрой зарядки. В режиме подзарядки в батарею поступает только очень небольшое количество тока, что приводит к неполной зарядке батареи. В режиме быстрой зарядки весь доступный ток направляется в зарядное устройство.

С точки зрения систем управления следует определить, когда нужно переходить в режим быстрой зарядки, а когда – в режим подзарядки. При зарядке температура батареи повышается. Если батарея заряжена полностью, дополнительный ток, проходящий через нее, будет приводить к ее нагреву. Поэтому, если батарея нагревается, можно считать, что она полностью заряжена, а значит,

следует перейти в режим подзарядки. Кроме того, можно измерить напряжение батареи, чтобы определить, достигло ли оно предела, и затем переключиться в режим подзарядки. Если батарея не нагрелась и не достигла предела по напряжению, следует перейти в режим полной зарядки. Это упрощенные правила, поскольку кривая температуры батареи является оптимальным показателем ее зарядки.

Управление зарядкой батареи с помощью нечеткой логики

Как уже говорилось, зарядное устройство имеет два режима работы: режим подзарядки и режим быстрой зарядки. Состояние батареи отслеживают два датчика: датчик напряжения и датчик температуры. Для управления зарядкой будут использоваться следующие правила нечеткой логики:

```
if m_voltage_high ( voltage )
then mode = trickle_charge
if m_temperature_hot ( temperature )
then mode = trickle_charge
if ( ( not(m_voltage_high ( voltage ))) AND
( not(m_temperature_hot( temperature ))) )
then mode = fast_charge
```

Обратите внимание, что эти правила не являются оптимальными, так как последнее правило может выполняться для всех вариантов. Здесь они используются для описания большего количества операторов нечеткой логики.

Сначала нужно идентифицировать правила нечеткой логики. Идентификация проводится с помощью анализа проблемы, или же для этой цели вызывается эксперт. Следующая задача состоит в том, чтобы определить соответствие между словесным выражением правил и понятиями реального мира. Для этого необходимо создать функции принадлежности.

Функции принадлежности при зарядке батареи с помощью нечеткой логики

При создании функций принадлежности нужно взять лингвистические правила нечеткой логики и определить их связь с понятиями реального мира в заданной области. В этом примере заданы две переменные: напряжение и температура. Графики принадлежности для напряжения и температуры (отображающие функции принадлежности) показаны на рис. 9.5 и 9.6.

График функции принадлежности для напряжения определяет в области напряжения три функции принадлежности: низкое, среднее и высокое. Аналогично задаются три функции принадлежности для области температуры: холодно, тепло и горячо. Эти значения используются только для демонстрации и не учитывают какую-либо технологию производства батарей.

Совет *Вы легко можете увидеть шаблоны в графическом представлении функций принадлежности. В исходном коде примера предлагается ряд полезных функций, которые упрощают создание функций принадлежности (включая модель шипа, не представленную на этих графиках).*

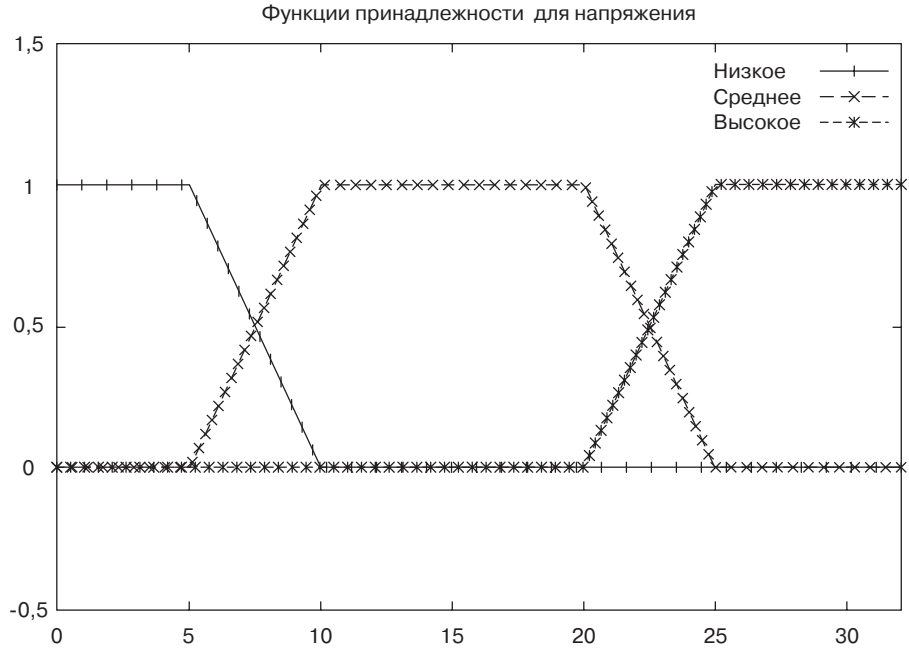


Рис. 9.5. График функции принадлежности для напряжения

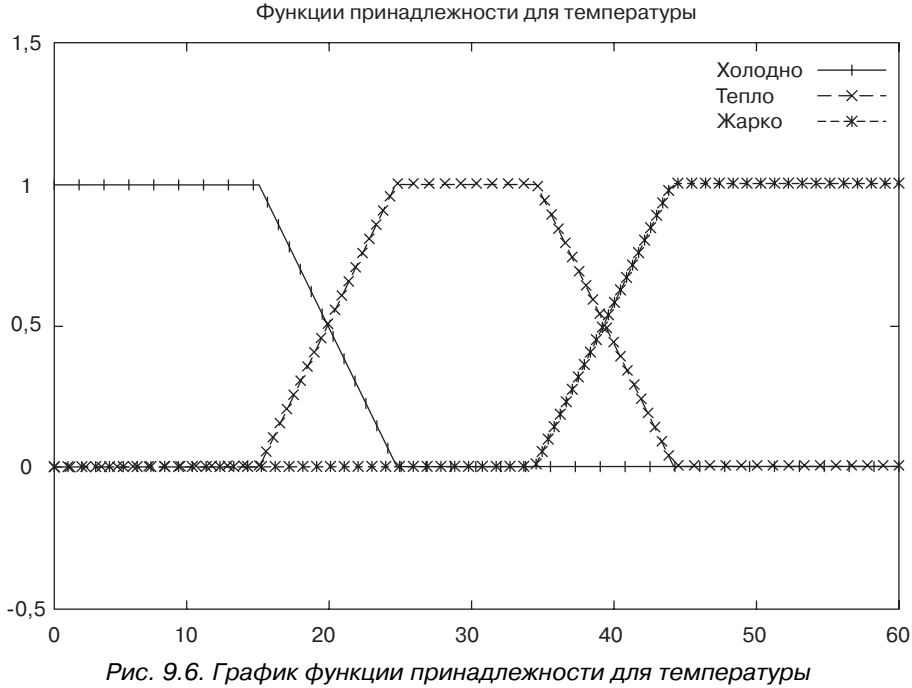


Рис. 9.6. График функции принадлежности для температуры

Примечание *Исходный код примера для модели зарядного устройства на основе нечеткой логики находится в архиве с примерами, который вы можете загрузить с сайта издательства «ДМК Пресс» www.dmk.ru.*

Обсуждение исходного кода

Код, представленный в данном разделе, предлагает ряд моделей для создания программного обеспечения с помощью нечеткой логики. Кроме того, здесь приведен пример использования этих моделей, позволяющий продемонстрировать их возможности. Сначала рассматриваются общие функции нечеткой логики, а затем рассказывается о примере модели.

Механизм нечеткой логики

Разработчики программного обеспечения обычно используют два элемента нечеткой логики: операторы нечеткой логики и функции принадлежности.

Операторы нечеткой логики представляют собой обычные функции `AND`, `OR` и `NOT`, модифицированные для нечеткой логики (листинг 9.1).

Листинг 9.1. Операторы нечеткой логики

```
#define MAX(a,b)   ((a>b) ? a : b)
#define MIN(a,b)   ((a<b) ? a : b)

fuzzyType fuzzyAnd( fuzzyType a, fuzzyType b )
{
   return MAX(a,b);
}

fuzzyType fuzzyOr( fuzzyType a, fuzzyType b )
{
  return MIN(a,b);
}

fuzzyType fuzzyNot( fuzzyType a )

{
  return( 1.0 - a );
}
```

Эти функции следуют аксиомам нечеткой логики, представленным на рис. 9.5.

Следующие программные конструкции используются при создании функции принадлежности (листинг 9.2). Они позволяют разработчику задать границы функции принадлежности с помощью группы значений, представляющих ее параметры.

Листинг 9.2. Функции, которые используются для создания функций принадлежности

```
fuzzyType spikeProfile( float value, float lo, float high )
{
  float peak;
```

```
  value += (-lo);

  if      ((lo < 0) && (high < 0)) {
    high = -(high - lo);
  } else if ((lo < 0) && (high > 0)) {
    high += -lo;
  } else if ((lo > 0) && (high > 0)) {
    high -= lo;
  }

  peak = (high / 2.0);
  lo = 0.0;
  if (value < peak) {
    return( value / peak );
  } else if (value > peak) {
    return( (high-value) / peak );
  }

  return 1.0;
}

fuzzyType plateauProfile( float value, float lo, float lo_plat,
                          float hi_plat, float hi )
{
  float upslope;
  float downslope;

  value += (-lo);

  if (lo < 0.0) {
    lo_plat += -lo;  hi_plat += -lo;
    hi      += -lo;  lo       = 0;
  } else {
    lo_plat -= lo;  hi_plat -= lo;
    hi      -= lo;  lo       = 0;
  }

  upslope = (1.0 / (lo_plat - lo));
  downslope = (1.0 / (hi - hi_plat));

  if (value < lo) return 0.0;
  else if (value > hi) return 0.0;
  else if ((value >= lo_plat) && (value <= hi_plat)) return 1.0;
  else if (value < lo_plat) return ((value-lo) * upslope);
  else if (value > hi_plat) return ((hi-value) * downslope);

  return 0.0;
}
```

Первая функция, `spikeProfile`, задает обычную функцию принадлежности в виде треугольника (например, для центральной функции принадлежности на рис. 9.3). Разработчик указывает значения `lo` и `hi`, которые определяют базовые вершины треугольника. Высшая точка задается как `hi/2`.

Вторая функция, `plateauProfile`, задает функцию принадлежности в форме трапеции (пример – функция принадлежности для температуры на рис. 9.6). Затем с помощью функции `plateauProfile` дополнительно создаются те функции принадлежности, которые распространяются до границ (например, функции `холодно` и `жарко` на рис. 9.6).

Их задача заключается в том, чтобы определить степень принадлежности для заданного значения и аргументов функции.

Функции принадлежности для модели зарядного устройства

Рассмотрим исходный код, реализующий модель зарядного устройства для батарей. Сначала изучим функции принадлежности. Они используют описанные выше вспомогательные функции, чтобы построить графики, представленные диаграммами принадлежности.

В первую группу входят функции принадлежности для напряжения (листинг 9.3).

Листинг 9.3. Функции принадлежности для напряжения

```
fuzzyType m_voltage_low( float voltage )
{
  const float lo = 5.0;
  const float lo_plat = 5.0;
  const float hi_plat = 5.0;
  const float hi = 10.0;

  if (voltage < lo) return 1.0;
  if (voltage > hi) return 0.0;

  return plateauProfile( voltage, lo, lo_plat, hi_plat, hi );
}

fuzzyType m_voltage_medium( float voltage )
{
  const float lo = 5.0;
  const float lo_plat = 10.0;
  const float hi_plat = 20.0;
  const float hi = 25.0;

  if (voltage < lo) return 0.0;
  if (voltage > hi) return 0.0;

  return plateauProfile( voltage, lo, lo_plat, hi_plat, hi );
}
```

```
fuzzyType m_voltage_high( float voltage )
{
  const float lo = 25.0;
  const float lo_plat = 30.0;
  const float hi_plat = 30.0;
  const float hi = 30.0;

  if (voltage < lo) return 0.0;
  if (voltage > hi) return 1.0;

  return plateauProfile( voltage, lo, lo_plat, hi_plat, hi );
}
```

Все функции принадлежности в листинге 9.3 используют функцию `plateauProfile`, чтобы построить график. Каждая из них принимает значение напряжения и затем возвращает значение, которое соответствует ее степени принадлежности. Каждая функция сначала проверяет переданное значение на соответствие диапазону функции принадлежности. Если значение не выходит за рамки диапазона, оно передается в функцию `plateauProfile`. При этом ее сигнатура задается как вектор [`lo`, `lo_plat`, `hi_plat`, `hi`], а затем пользователю возвращается результат.

Функции принадлежности, описанные в листинге 9.3, изображены на рис. 9.5.

В листинге 9.4 представлены функции принадлежности для температуры, которые используются в модели зарядного устройства для батарей.

Листинг 9.4. Функции принадлежности для температуры

```
fuzzyType m_temp_cold( float temp )
{
  const float lo = 15.0;
  const float lo_plat = 15.0;
  const float hi_plat = 15.0;
  const float hi = 25.0;

  if (temp < lo) return 1.0;
  if (temp > hi) return 0.0;

  return plateauProfile( temp, lo, lo_plat, hi_plat, hi );
}

fuzzyType m_voltage_low( float voltage )
{
  const float lo = 5.0;
  const float lo_plat = 5.0;
  const float hi_plat = 5.0;
  const float hi = 10.0;

  if (voltage < lo) return 1.0;
  if (voltage > hi) return 0.0;
```

```
  return plateauProfile( voltage, lo, lo_plat, hi_plat, hi );
}

fuzzyType m_voltage_medium( float voltage )
{
  const float lo = 5.0;
  const float lo_plat = 10.0;
  const float hi_plat = 20.0;
  const float hi = 25.0;

  if (voltage < lo) return 0.0;
  if (voltage > hi) return 0.0;

  return plateauProfile( voltage, lo, lo_plat, hi_plat, hi );
}
```

Функции принадлежности для значения температуры, представленные в листинге 9.4, проиллюстрированы на рис. 9.6.

Функция управления в модели зарядного устройства для батарей

Теперь можно свести задачу управления зарядным устройством к применению правил нечеткой логики, о которых рассказывалось ранее. Функция `chargeControl` (листинг 9.5) позволяет управлять процессом зарядки батареи.

Листинг 9.5. Функция, управляющая зарядкой батареи

```
void chargeControl()
{

  static unsigned int i = 0;

  extern float voltage, temperature;

  if ( (i++ % 10) == 0 ) {

    if (normalize( m_voltage_high( voltage ) ) ) {
      chargeMode = TRICKLE_CHARGE;
      *timer = 0.0;
    } else if (normalize( m_temp_hot( temperature ) ) ) {
      chargeMode = TRICKLE_CHARGE;
      *timer = 0.0;
    } else if (normalize(
                  fuzzyAnd(
                  fuzzyNot( m_voltage_high( voltage ) ),
                  fuzzyNot( m_temp_hot( temperature ) ) ) ) ) {
      chargeMode = FAST_CHARGE;
      *timer = 0.0;
    }
  }
}
```

Используя правила нечеткой логики и функции принадлежности, указанная функция в зависимости от значений напряжения и температуры изменяет режим зарядки батареи.

Главный цикл модели

Наконец, главный цикл выполняет функции управления процессом зарядки батареи, основываясь на заданных параметрах напряжения и температуры (листинг 9.6).

Листинг 9.6. Главный цикл

```
int main()
{
  int i;

  extern float timer;
  extern int simulate(void);
  extern void chargeControl( float * );

  extern float voltage;
  extern float temperature;
  extern int chargeMode;

  for (i = 0 ; i < 3000 ; i++) {

    simulate();

    chargeControl( &timer );

    timer += 1.0;
    printf("%d, %f, %f, %d\n", i,
           voltage,
           temperature,
           chargeMode
    );
  }

  return 0;
}
```

Программа вызывает функцию, которая эмулирует процесс собственно зарядки/разрядки батареи, а затем позволяет функции управления зарядкой установить нужный режим для зарядного устройства. Сама функция эмуляции не показана в тексте, но вы можете найти ее в архиве с примерами.

Пример выполнения кода представлен на рис. 9.7. Этот график показывает напряжение, температуру и режим зарядки. Наличие входного напряжения зарядного устройства определяется при попадании 50% солнечного света на солнечные

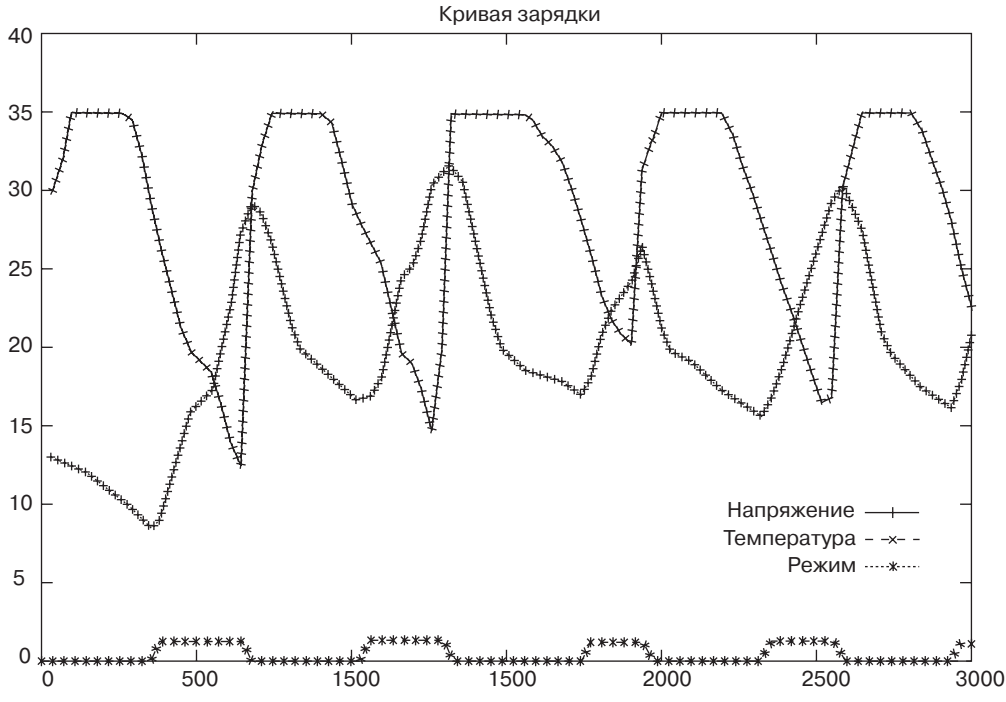


Рис. 9.7. Изменение кривых при моделировании управления зарядкой батареи

батареи. Очевидно, что зарядное устройство правильно выбирает режим зарядки для батареи с учетом имеющейся нагрузки и наличия входного тока.

Представленная модель не отражает настоящего процесса зарядки батареи, однако она демонстрирует учет ограничений по напряжению и температуре.

Преимущества использования нечеткой логики

Операторы нечеткой логики очень схожи с обычными булевыми операторами. Функции принадлежности и правила нечеткой логики, подвергнутые лингвистической модификации, позволяют значительно расширить возможности системных операторов.

Разработчики могут намного упростить сложность систем, используя нечеткую логику, поскольку она позволяет моделировать комплексные программы с большим количеством входов и выходов.

С помощью нечеткой логики можно добиться снижения системных требований, а значит, сократить расходы на аппаратные средства. Во многих случаях сложное математическое моделирование предпочтительнее заменить функциями принадлежности и правилами нечеткой логики и с их помощью управлять системой. При сокращении объемов информации размеры кода уменьшаются, поэтому

система работает быстрее. Кроме того, это позволяет использовать менее совершенные аппаратные средства.

Другие области применения

Нечеткая логика используется в самых разнообразных приложениях. Наиболее очевидная область ее применения – системы управления, которым нечеткая логика уже обеспечила коммерческий успех. Нечеткая логика используется в устройстве видеокамер и фотоаппаратов с автофокусом, системах смешивания цемента, автомобильных системах (например, системах АБС) и даже системах, основанных на правилах.

Наверное, самые полезные области применения все еще остаются неизвестными. Само название «нечеткая логика» не внушает особого доверия, хотя давно известно, что это надежный метод. Как и многие другие методики ИИ, нечеткая логика в настоящее время все чаще используется в устройствах повседневного применения, где она больше не ассоциируется с искусственным интеллектом.

Итоги

В этой главе было введено понятие нечеткой логики. Фундаментальные операторы нечеткой логики рассматривались на примере трех очень разных программ (потока пакетов данных, отслеживания цели и управления зарядкой батареи). Также обсуждались функции-ограничители, которые используются в качестве модификаторов для функций принадлежности. Затем была проанализирована модель зарядного устройства для батарей, демонстрирующая создание функций принадлежности и элементов управления. Для иллюстрации этой концепции использовался пример программы, которая включает набор механизмов нечеткой логики. Наконец, мы обсудили ряд преимуществ нечеткой логики, включая упрощение кода и снижение расходов на аппаратные средства.

Литература и ресурсы

1. Aptronix, Inc. Зачем использовать нечеткую логику (WhyUse Fuzzy Logic, 1996). Доступно по адресу http://www.aptronix.com/fide/whyfuzzy.htm.
2. Брюл Дж. Ф. Системы нечеткой логики – руководство (Brule J. F. Fuzzy Systems – A Tutorial, 1985). Доступно по адресу http://www.austinlinks.com/Fuzzy/tutorial.html.
3. Заде Л. А. Группы данных для нечеткой логики (Zadeh L. A. Fuzzy sets // Information and Control, 8:338–53, 1965).
5. Янтцен Я. Руководство по нечеткой логике (Jantzen J. Tutorial to Fuzzy Logic // Technical Report no 98–E 868. – University of Denmark, Department of Automation, 1998).

Глава 10. Модель состояний

В данной главе рассматривается *модель Маркова* (Markov Model), имя которой дал российский математик Андрей Марков), а также ее версия, которая называется *модель состояний* (Bigram Model). Эти модели могут быть очень полезными при описании процессов, которые включают разные состояния, а также возможностей, связанных с различными переходами между ними. Для данных моделей существует ряд весьма интересных областей применения. Мы изучим использование модели состояний для автоматической генерации текста.

Введение

Цепочка Маркова (Markov Chain) представляет собой процесс, который состоит из нескольких состояний и вероятностей, связанных с переходами между ними.

Рис. 10.1 иллюстрирует произношение слова «tomorrow» (завтра). На диаграмме доступны два различных способа произношения. Вероятность произношения слова как «tahmorrow» равна 0,5, вероятность произношения слова как «tuwmorrow» также равна 0,5.

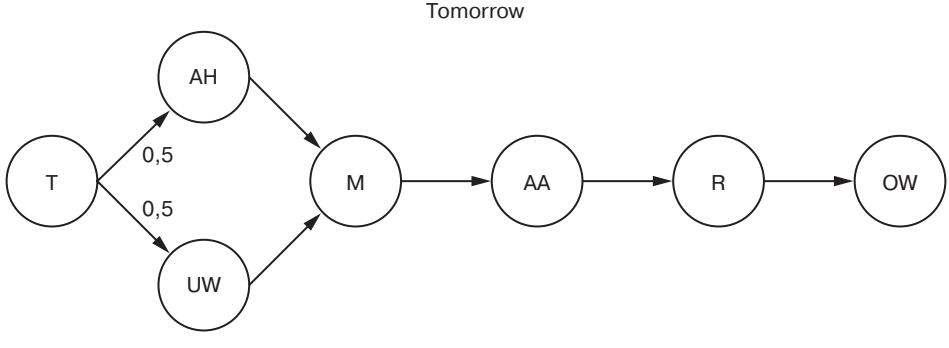


Рис. 10.1. Пример цепочки Маркова

Это очень простой пример, так как он подразумевает принятие решения только в одной точке цепи. Каждое состояние включает формирование фонемы. В конце цепочки нам доступно полное произношение. В разделе, посвященном применению модели, будут представлены задачи, которые могут решаться с помощью таких цепочек.

Рассмотрим другое применение модели. Представьте программу работы с электронной почтой, которая отслеживает поведение пользователя. Когда приходит сообщение, программа определяет, как с ним поступил пользователь, и использует эту информацию для работы со следующими подобными сообщениями (см. рис. 10.2).

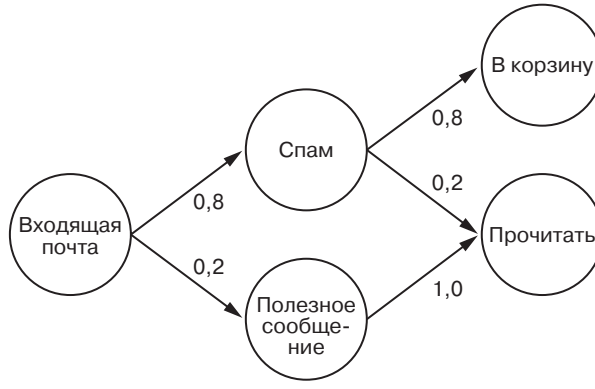


Рис. 10.2. Программа работы с электронной почтой отслеживает поведение пользователя

Агент электронной почты заметил, что 8 из 10 сообщений являются спамом, а остальные 2 – письмами. Затем агент запоминает, что в 80% случаев пользователь удаляет сообщения, не прочитав их, а в 20% случаев сообщения бывают прочитаны. Исходя из этих вероятностей, агент может предположить, что пользователь скорее удалит сообщение, чем прочитает его. Применение вероятностей позволяет агенту упростить работу с электронной почтой.

Примечание *В этих примерах используется ограниченное количество состояний и соединений. Цепочка Маркова может поддерживать очень большое количество состояний с множеством соединений и моделировать комплексные процессы.*

В приведенных примерах текущее состояние всегда является функцией предыдущего состояния, что допускает вероятность соединения. Эта особенность называется свойством Маркова. Кроме того, в данных примерах определенное состояние достигается не только из одного предыдущего состояния (например, состояние «Прочитать» на рис. 10.2). Такие модели известны как *скрытые модели Маркова* (Hidden Markov Model – HMM), или *скрытые цепочки Маркова* (Hidden Markov Chain).

Скрытые модели Маркова

Обратите внимание на то, что в предыдущих примерах текущее состояние цепочки является исключительно функцией предыдущего состояния с заданной

вероятностью. Это свойство известно как *биграмма* (Bigram), то есть последовательность двух слов. Если бы текущее состояние не являлось функцией предыдущего состояния, то выбор состояния превратился бы в случайный процесс. Состояние, зависящее от двух предыдущих состояний, называется триграммой. Хотя увеличение зависимости от предыдущего состояния (или контекста) может повысить полезность цепочки (как показано в разделе о применениях модели), системные требования к такими моделям могут стать очень высокими. В качестве примера рассмотрим табл. 10.1, которая показывает количество элементов, необходимых для словаря в 100 слов.

Таблица 10.1. Размер контекста и количество уникальных элементов

n	Тип	Количество элементов
2	Биграмма	10000
3	Триграмма	1000000
4	Тетраграмма	100000000

Словарь из 100 слов очень невелик, но создание более крупных биграмм может стать весьма трудоемким делом.

Интересные области применения

Цепочки Маркова для скрытых моделей могут применяться в самых разнообразных областях, например, при распознавании речи. Этот метод используется при обработке человеческой речи. Два следующих примера являются не столько практическими, сколько теоретическими, но они позволят вам лучше понять возможности цепочек Маркова.

Распознавание речи

Вспомните, что одно и то же слово может произноситься по-разному в зависимости от диалекта или происхождения говорящего (рис. 10.1). Это делает построение систем распознавания речи очень сложной задачей, так как система должна работать с различными вариантами произношения для одного и того же слова.

Скрытые цепочки Маркова позволяют упростить системы распознавания речи путем вероятностного анализа речевых фонем. Предположим, что система настроена на распознавание группы слов, два из которых – это «tomorrow» (завтра) и «today» (сегодня). Когда система в первый раз слышит фонему «tah», произносимое слово может быть как «tomorrow», так и «today». Далее система анализирует фонему «m»; теперь вероятность того, что произносится слово «today», равна нулю. При работе со скрытой моделью фонема «m» является основным состоянием, поэтому программа переходит к обработке следующей фонемы. Используя вероятности при переходах, система распознавания речи выбирает оптимальный путь по цепи, чтобы определить ту фонему, которая, скорее всего, последует.

Данный пример можно расширить от фонем к словам. Получив набор распознаваемых слов, можно создать цепочку Маркова, способную идентифицировать

вероятность того, что одно слово последует за другим. Это позволит системе лучше понимать речь, ориентируясь по контексту (рис. 10.3).

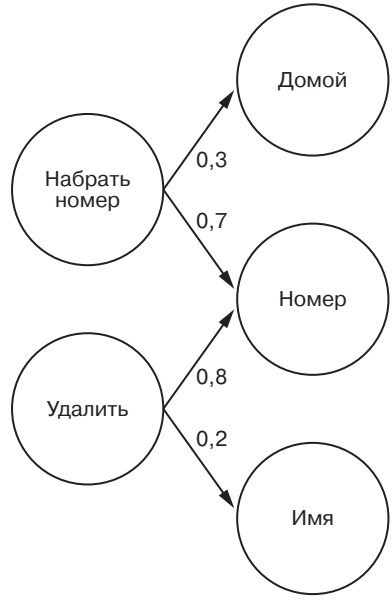


Рис. 10.3. Более сложная цепочка Маркова, использующаяся для системы распознавания речи

Как показывают приведенные примеры, скрытые модели Маркова могут упростить такие задачи, как распознавание и понимание речи. В этом примере ввод фонем привел к переходам от одного состояния к другому в пределах модели, что позволило распознать слова. Далее ввод слов привел к изменениям, помогающим понять предложение по контексту. В следующем разделе рассматривается применение скрытых моделей Маркова для генерирования символов на основании заданных или полученных в результате обучения вероятностей перехода.

Моделирование текста

В предыдущих примерах цепочка Маркова использовалась для вероятностной идентификации следующего состояния с учетом текущего состояния и внешнего стимула. Теперь мы рассмотрим ряд примеров, в которых отсутствует фактор внешнего стимула. Это значит, что переходы от одного состояния к другому в цепочке основываются только на заданных вероятностях.

На рис. 10.4 показан пример цепочки Маркова для двух предложений. Единственное неуникальное слово в исходных данных – это слово «is» (есть). С равной вероятностью за этим словом может следовать как «a», так и «the». Обратите внимание, что теперь в цепочке Маркова могут быть сгенерированы четыре предложения (показаны в нижней части на рис. 10.4).

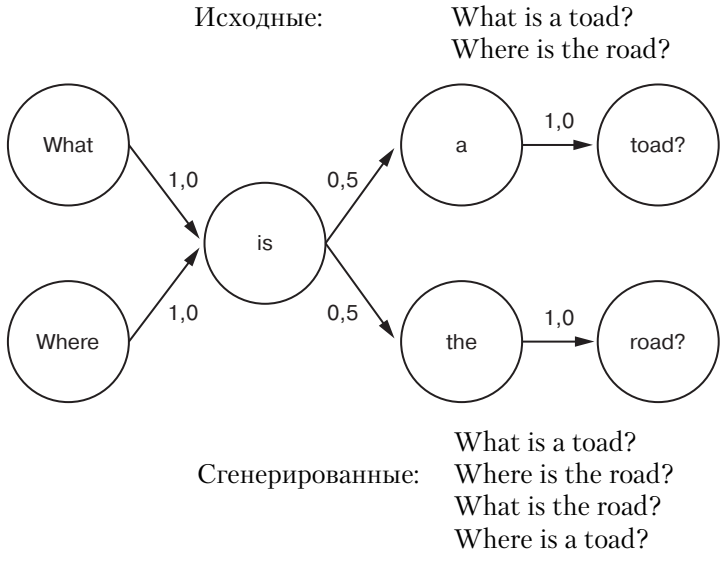


Рис. 10.4. Пример биграммы со словарем из семи уникальных слов

Моделирование музыки

Так же, как и со словами, скрытая модель Маркова работает с нотами при моделировании музыки. Затем вы сможете использовать ее при вероятностной генерации нот в соответствии со стилем данного композитора. Также модель допускается обучать работе с произведениями нескольких композиторов. Из комбинаций их творений вполне реально создавать целые симфонии.

Пример применения

Некоторые разработчики считают, что, используя вероятностные возможности скрытых моделей Маркова, можно повторить произведения даже таких великих писателей, как Шекспир. После обучения на примере авторского текста скрытая модель Маркова позволяет генерировать последовательности слов, которые схожи со стилем оригинального произведения.

В следующем разделе рассматривается обучение модели с помощью заданного текста. Затем рассказывается, как скрытая модель Маркова применяется для генерации произвольных последовательностей символов.

Примечание *Исходный код для скрытой модели Маркова содержится в архиве с примерами к данной книге, который можно загрузить с сайта издательства «ДМК Пресс» www.dmk.ru.*

Обсуждение исходного кода

Исходный код программы, анализирующей текст с помощью цепочек биграмм и последующим построением предложений из них, довольно прост. Биграмма

реализуется в виде обычного двумерного массива. Каждое измерение представлено уникальными словами, извлеченными из текста. Первое измерение – первое слово биграммы, второе – слова, которые следуют за первым. При этом содержимое двумерного массива показывает, сколько раз второе слово повторилось за первым.

Листинг 10.1 содержит символьные константы и глобальные переменные, которые используются в программе.

Листинг 10.1. Символьные константы и глобальные переменные

```
#define MAX_WORD_LEN  40
#define MAX_WORDS     1000

#define FIRST_WORD    0
#define MIDDLE_WORD   1
#define LAST_WORD     2

#define START_SYMBOL  0
#define END_SYMBOL    1

static int curWord = 2;

char wordVector[MAX_WORDS][MAX_WORD_LEN];

int  bigramArray[MAX_WORDS][MAX_WORDS];
int  sumVector[MAX_WORDS];
```

Максимальная длина слов составляет 40 символов (`MAX_WORD_LEN`), а максимальное количество уникальных слов равно 1000 (`MAX_WORDS`). Переменная `curWord` идентифицирует индекс текущего слова в массивах `wordVector` и `bigramArray`. Константы `START_SYMBOL` и `END_SYMBOL` представляют индексы для начального и конечного состояния цепочки Маркова. Наконец, константы `FIRST_WORD`, `MIDDLE_WORD` и `LAST_WORD` используются для идентификации контекста определенного слова (а также определяют, будет ли оно влиять на начало или конец цепочки Маркова).

Главная программа демонстрации биграммы представлена в листинге 10.2. Она выполняет два основных действия: анализ входного файла, который включает словарь (`parseFile`), и вывод предложения, построенного на основе данных, которые получены в результате обработки обучающего файла (`buildSentence`).

Листинг 10.2. Программа демонстрации биграммы

```
int main( int argc, char *argv[] )
{
  char filename[40];
  int  debug = 0;

   /* Разбор параметров командной строки */
   parseOptions( argv, argc, &debug, filename );
```

```
/* Инициализация */
srand(time(NULL));

  bzero(bigramArray, sizeof(bigramArray));

  strcpy(wordVector[0], "<START>");
  strcpy(wordVector[1], "<END>");

/* Анализ входных данных */
parseFile( filename );

  if (debug) emitMatrix();

  printf("unique = %d\n", curWord);

   /* Генерируем предложения */
   buildSentence();

  return 0;
}
```

Остальные элементы программы включают анализ параметров командной строки, позволяющий изменять поведение программы, и инициализацию рабочих переменных. Здесь важно проследить, как происходит инициализация массива `wordVector`. Вспомните, что массив `wordVector` представляет собой список уникальных слов, которые были обнаружены во время анализа авторского текста. В него помещаются два символа, которые позволяют идентифицировать начало и конец цепочки. Начало цепочки всегда задается нулем, а конец – единицей. Все последующие значения отображают уникальные слова, обнаруженные в тексте.

Программа принимает две опции, `-f` позволяет указать имя файла для обучающего текста, а `-v` включает режим отображения дополнительной информации (отладочный режим). В этом режиме выводится массив биграммы, и вы можете посмотреть, какая информация была извлечена из текста (листинг 10.3).

Листинг 10.3. Функция `parseOptions`

```
void parseOptions( char *argv[], int argc, int *dbg, char *fname )
{
  int opt, error = 1;

  *dbg = 0;

  if (argc > 1) {

    while ((opt = getopt(argc, argv, "vf:")) != -1) {

      switch(opt) {

        case 'v':
          *dbg = 1;
```

```
            break;

          case 'f':
            strcpy(fname, optarg);
            error = 0;
            break;

          default:
            error = 1;

        }
      }
    }

    if (error) {
      printf("\nUsage is : \n\n");
      printf("\t%s -f <filename> -v\n\n", argv[0]);
      printf("\t\t -f corpus filename\n\t\t -v verbose mode\n\n");
      exit(0);
    }

    return;
  }
```

Функция `parseOptions` применяет стандартную функцию `getopt`, чтобы упростить анализ параметров командной строки.

Функция `parseFile` используется, чтобы извлечь связи между словами из текста, заданного пользователем (листинг 10.4).

Листинг 10.4. Функция `parseFile`

```
void parseFile( char *filename )
{
  FILE *fp;
  int  inp, index = 0;
  char word[MAX_WORD_LEN+1];
  int  first = 1;

  fp = fopen(filename, "r");

  while (!feof(fp)) {

    inp = fgetc(fp);

    if (inp == EOF) {

      if (index > 0) {
/* Обновить матрицу для последнего слова */
        word[index++] = 0;
        loadWord(word, LAST_WORD);
```

```
        index = 0;
      }

    } else if (((char)inp == 0x0d) || ((char)inp == 0x0a) ||
               ((char)inp == ' ')) {

      if (index > 0) {
        word[index++] = 0;
          if (first) {
         /* Первое слово в последовательности */
          loadWord(word, FIRST_WORD);
          index = 0;
          first = 0;
        } else {
         /* Слово в середине последовательности */
          loadWord(word, MIDDLE_WORD);
          index = 0;
        }
      }

    } else if (((char)inp == '.') || ((char)inp == '?')) {

/* Обработка знаков препинания на конце текущей последовательности */
word[index++] = 0;
      loadWord(word, MIDDLE_WORD);
      loadWord(word, LAST_WORD);
      index = 0;
      first = 1;

    } else {

/* Пропуск пробелов */
if (((char)inp != 0x0a) && ((char)inp != ',')) {
        word[index++] = (char)inp;

      }
    }
  }

  fclose(fp);
}
```

Функция `parseFile` принимает название файла с обучающим текстом. Ее задача заключается в том, чтобы извлечь отдельные слова из файла и внести их в массивы `wordVector` и `bigramArray` (с использованием функции `loadWord`). Это значит, что при помещении слова в массив следует учитывать тот порядок, в котором оно было найдено. Было ли слово первым в предложении, последним или стояло в середине? В зависимости от порядка слов вызывается функция `loadWord` со значением, показывающим порядок, в котором слово было найдено.

Она также анализирует контекст слова с помощью маркеров конца файла, символов перевода строки, а также других пунктуационных знаков.

Функция `loadWord` (листинг 10.5) обновляет структуры биграммы для слова, а также информацию о его положении.

Листинг 10.5. Функция `loadWord`

```
void loadWord( char *word, int order )
{
  int wordIndex;
  static int lastIndex = START_SYMBOL;

  /* Сначала проверим, что слово уже сохранено */
  for (wordIndex = 2 ; wordIndex < curWord ; wordIndex++) {
    if (!strcmp(wordVector[wordIndex], word)) {
      break;
    }
  }

  if (wordIndex == curWord) {

    if (curWord == MAX_WORDS) {
      printf("\nToo may words, increase MAX_WORDS\n\n");
      exit(-1);
    }

    /* Слова нет в списке - добавляем его */
    strcpy(wordVector[curWord++], word);
  }

  /* Здесь мы имеем индекс слова в массиве */

  if (order == FIRST_WORD) {

     /* Сохраняем слово в качестве начала последовательности */
     bigramArray[START_SYMBOL][wordIndex]++;
     sumVector[START_SYMBOL]++;

  }  else if (order == LAST_WORD) {
     /* Сохраняем слово в качестве конца последовательности */
     bigramArray[wordIndex][END_SYMBOL]++;
     bigramArray[END_SYMBOL][wordIndex]++;
     sumVector[END_SYMBOL]++;

  } else {
     /* Сохраняем слово в качестве середины последовательности */
     bigramArray[lastIndex][wordIndex]++;
```

```
      sumVector[lastIndex]++;

    }

    lastIndex = wordIndex;

    return;
}
```

Сначала функция `loadWord` должна определить, является ли текущее слово уникальным (то есть не встречалось раньше). Это можно сделать путем быстрого сканирования массива `wordVector`. Если слово новое, программа создает новую ячейку для него в массиве `wordVector` и соответствующим образом обновляет информацию о границах массива (функция `curWord`).

Теперь индекс слова включен в массив `wordVector`. Используя аргумент порядка (определяющий порядок, в котором слово встретилось в тексте), программа обновляет массив `bigramArray`. Если слово было первым в предложении, она обновляет строку `START_SYMBOL` (первую строку в таблице) и изменяет смещение для столбца, заданного индексом `wordIndex`. Кроме того, увеличивается значение в массиве `sumVector`, которое позволяет рассчитать относительные вероятности для каждого слова.

Если текущее слово было последним в предложении, обновляется ячейка массива `bigramArray` по адресу, определяемому индексом слова, которое является первым измерением для массива, и символом `LAST_SYMBOL` для второго индекса. Наконец, если слово находится в середине предложения, последнее слово используется в качестве первого индекса, а текущее слово – в качестве второго. Последнее слово всегда сохраняется в функции в переменной `lastIndex`. Пример предложения представлен на рис. 10.5.

Итак, анализ и заполнение массива биграммы завершены. Две следующие функции обеспечивают генерацию предложения на основе биграммы, использующейся в качестве модели.

Функция `buildSentence` проходит по структуре `bigramArray` и с помощью массива `sumVector` определяет, какой путь выбрать и какие слова использовать для построения массива (листинг 10.6).

Листинг 10.6. Функция `buildSentence`

```
int buildSentence( void )
{
  int word = START_SYMBOL;
  int max = 0;

  printf("\n");

   /* Начнем со случайно выбранного слова */
   word = nextWord(word);
```

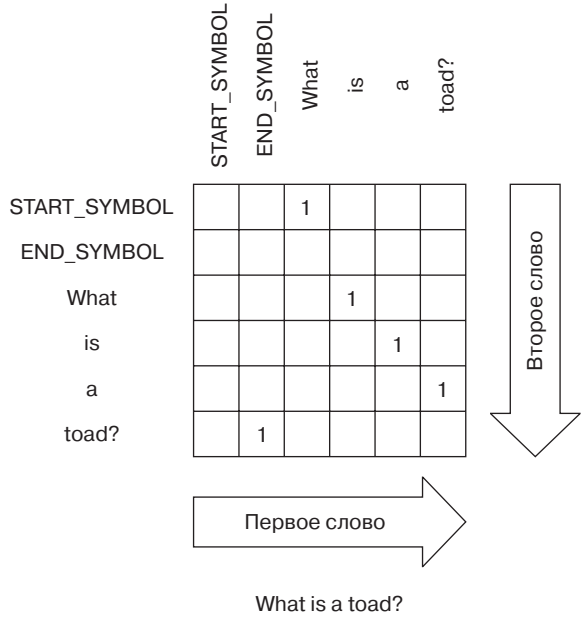


Рис. 10.5. Массив `bigramArray` *для предложения-примера*

```
    /* Продолжаем, пока не достигнем конца случайной последовательности */
    while (word != END_SYMBOL) {

    /* Выводим текущее слово */
    printf("%s ", wordVector[word]);

    /* Выбираем следующее слово */
    word = nextWord(word);

    /* Начинаем со случайно выбранного слова */
    max += getRand(12) + 1;

    /* Если набрали максимум возможных слов - останавливаемся */
    if (max >= 100) break;

   }

    /* Удаляем концевой пробел, ставим точку и конец строки */
    printf("%c.\n\n", 8);

   return 0;
}
```

Функция `buildSentence` создает путь через массив `bigramArray` с помощью вероятностей, которые определяются содержимым массива. Вспомните, что

пересечение двух слов в нем показывает, сколько раз второе слово повторилось после первого. В процесс поиска выполняется случайный расчет значения переменной max, который позволяет ограничить количество слов.

Функция buildSentence использует функцию nextWord, чтобы определить следующее слово в цепочке (еще один шаг по пути модели Маркова) – см. листинг 10.7.

Листинг 10.7. Функция nextWord

```
int nextWord( int word )
{
  int nextWord = (word + 1);
  int max = sumVector[word];
  int lim, sum = 0;

   /* Вычисляем ограничитель выбора */
   lim = getRand(max)+1;

  while (nextWord != word) {
   /* Остаемся в границах массива */
   nextWord = nextWord % curWord;

   /* Считаем сумму вероятностей */
   sum += bigramArray[word][nextWord];

   /* Если мы привнесем ограничитель - возвращаем текущее выбранное
    * слово
    */
   if (sum >= lim) {
       return nextWord;
    }

   /* Переходим к следующей колонке */
   nextWord++;

  }

  return nextWord;
}
```

Функция nextWord случайным образом определяет следующий шаг на основании вероятностей. Граница задается в виде произвольного числа из массива sumVector для строки (общая сумма всех слов, которые следуют за текущим). С помощью произвольной границы (lim) программа проходит через строку, изменяя при этом переменную суммы. Если переменная суммы становится равной границе или превышает ее, программа использует текущее слово в качестве следующего этапа пути. В противном случае она продолжает суммировать до тех пор, пока не достигнет границы. С точки зрения вероятности

это значит, что слова с большими суммами будут с большей вероятностью выбираться при нахождении пути.

Наконец, в листинге 10.8 представлена функция вывода информации, `emitMatrix`, которая используется для вывода массивов `bigramArray` и `sumVector`.

Листинг 10.8. Функция `emitMatrix`

```
void emitMatrix( void )
{
  int x, y;
  printf("\n");
  for (x = 0 ; x < curWord ; x++) {
    printf("%20s : ", wordVector[x]);
    for ( y = 0 ; y < curWord ; y++) {
      printf("%d ", bigramArray[x][y]);
    }
    printf(" : %d\n", sumVector[x]);
  }
}
```

Примеры

Рассмотрим ряд примеров работы алгоритма, в которых используется разный текст.

В первом примере в качестве файла для обучения использовано несколько цитат из трудов Альберта Эйнштейна. Набор для обучения состоит из 13 цитат с 377 уникальными словами (см. листинг 10.9).

Листинг 10.9. Вывод текста на основании цитат из трудов Альберта Эйнштейна

```
[root@plato bigram]./bigram -f einstein
Imagination is shipwrecked by language and other symbolic
devices.
[root@plato bigram]./bigram -f einstein
We all ruled in the authority of imagination.
```

Каждая из сгенерированных цитат листинга 10.9 имеет смысл и кажется вполне осмысленной.

Теперь рассмотрим текст, который был сгенерирован на основе текста книги Альберта Камю «Человек абсурда» (листинг 10.10).

Листинг 10.10. Вывод текста на основе отрывка из книги Альберта Камю «Человек абсурда»

```
[root@plato bigram]./bigram -f absurdman
It is to speak only truth that is the breath of future actions.
[root@plato bigram]./bigram -f absurdman
"My field" said Goethe "is time and unfolds in a disservice to make.
```

Наконец, рассмотрим цепочку Маркова, которая была создана из двух разных книг (листинг 10.11). Первая книга – это поэма «Ода радости» неизвестного автора, а вторая – «Одиссея» Гомера.

Листинг 10.11. Вывод текста из нескольких книг

```
[root@plato bigram]./bigram -f combo
From chaos and beasts and hostile shores! From the woods the
hunter strayed.
```

Авторство

Цепочка Маркова может использоваться как для создания симфонии, так и для генерации текста, который напоминает другие литературные произведения. Возникает вопрос: кто является автором нового произведения? Цепочка Маркова может создавать музыку или текст, который моделируется на основе исходных данных. Поскольку результат очень схож с изначальным творением автора, его следует считать новым статистическим представлением авторской работы.

Если для моделирования авторского произведения используется триграмма или более сложная модель, то результат может быть чрезвычайно интересным — пожалуй даже эксперты не смогут отличить сгенерированный текст от авторского.

Итоги

В этой главе рассматривались цепочки Маркова, а также модели биграммы, которые используются для генерации текста. Цепочки Маркова применяются при решении различных задач – от проверки правописания до подтверждения авторства неизвестного текста. Здесь рассказывалось о распознавании текста с помощью цепочек, а также моделировании текста и музыки. Была представлена реализация генератора текста, использующего биграмму для создания новых фраз. Наконец, было уделено внимание вопросу авторства новых творений, сгенерированных с помощью цепочек Маркова. Это очень важно, поскольку новые произведения полностью базируются на других авторских работах.

Литература и ресурсы

1. Балди П. и Брунак С. Биоинформатика: подход к обучению машин (Balid P. and Brunak S. Bioinformatics: The Machine Learning Approach. – Cambridge, Mass.: MIT Press, 1998).
2. Бейкер Д. К. Стохастическое моделирование для автоматического понимания речи (Baker J. J. In R. Reddy (ed.) Stochastic Modeling for Automatic Speech Understanding // Speech Recognition: Academic Press, pp. 521–42, 1975).
3. Дженг Б. Т. Курс 4190.515: Биоинформатика (обучение машин) (Zheng B. T. Course 4190:515: Bioinformatics (Machine Learning) // Seoul National

University School of Computer Science and Engineering, 2001). Доступно по адресу http://bi.snu.ac.kr/Courses/g-ai01/g-ai01.html.

4. Хенке Дж. Статистическое заключение: модели n-gram и разбросанные данные (Henke J. Statistical Inference: n-gram Models over Sparse Data, TDM Seminar, 1999). Доступно по адресу http://www.sims.berkeley.edu/courses/is296a-4/f99/Lectures/henke-ch6.ppt.
5. Шеннон С. Е. Математическая теория коммуникации (Shannon C. E. A mathematical theory of communication // Bell System Technical Journal 27 (July and October), pp. 379–423 and 623–56, 1948).
6. Шеннон С. Е. Предсказание и энтропия литературного английского языка (Shannon C. E. Prediction and Entropy of Printed English // Bell System Technical Journal 30:50–64, 1951).

Глава 11. Программное обеспечение, основанное на использовании агентов

Данная глава посвящена программированию с использованием агентов (Agent). Здесь рассматриваются различные программные агенты, их применение, а также их атрибуты и модели. Чтобы продемонстрировать возможности агентов, мы создадим агента, фильтрующего новости. Для предоставления нужной информации пользователю он задействует два стандартных Internet-протокола.

Что представляет собой агент

Алан Кэй (Alan Kay), начавший первым продвигать теорию агентов, определил агента как программу, которая после получения задания способна поставить себя на место пользователя. Если агент попадает в тупик, он может задать пользователю вопрос, чтобы продолжить действовать. Недавно эта концепция была применена для разработки так называемых ботов, которые существуют в сети Internet и предоставляют пользователям новые возможности.

Агентов также называют умными агентами (Intelligent agent), так как разумность является ключевым фактором при их создании. Хотя агенты могут принимать различные формы, многие считают их суррогатами, реализующими какие-то полезные функции за человека. Агент может получить задачу, выполнить которую необходимо его пользователю, и принимать нужные решения в процессе коммуникации с другими агентами (возможно, даже с агентами, представляющими других пользователей). Например, агент-продавец способен находить агентов-покупателей на аукционе и продавать им товары, которые принадлежат его пользователю. Эти агенты одновременно могут играть и роль покупателей, определяя стоимость схожих товаров и соответствующим образом изменяя цену своих товаров.

Агенты используются при создании множества программ. Поэтому мы не будем сразу давать определение агентам, а сначала обсудим их атрибуты, а к вопросу строения агентов вернемся позже.

Свойства агентов

Агенты могут иметь одно или несколько свойств, как показано в табл. 11.1. В этом разделе рассказывается обо всех возможных свойствах агентов.

Таблица 11.1. Общие свойства программных агентов

Свойство	Определение
Автономный	Может действовать независимо от пользователя
Адаптивный	Способен к обучению во время работы
Коммуникативный	Способен к коммуникации с пользователем или другими агентами
Способный к сотрудничеству	Работает с другими агентами для достижения цели
Персонифицированный	Ведет себя естественно (проявляет эмоции)
Мобильный	Может перемещаться по окружающей среде

Одним из основных свойств, которые ассоциируются с программными агентами, является автономность (Autonomy). Агент считается автономным, если он способен действовать без прямого управления человека. Понятие автономности включает также наличие у агента нескольких целей, которых он может достигнуть. Хотя нацеленность на решение задачи может рассматриваться как отдельное свойство, оно самым непосредственным образом связано с автономностью и поэтому считается зависимым. Наличие задач также способствует тому, что агент получает возможность планировать. Это подводит нас к понятию адаптивности.

Агент является адаптивным (Adaptivity), если он может изменять свое поведение на основании опыта. Он должен учиться и делать выводы в зависимости от состояния окружающей среды и своих знаний. Хотя этот аспект является самым интересным, он еще и самый сложный, поэтому его можно рассматривать только на примере специфических агентов.

Еще одной важной характеристикой агентов является способность к коммуникации (Communicative). Агент должен уметь общаться с пользователем, чтобы определять свои задачи или идентифицировать начальную информацию. Кроме того, агенту необходимо общаться с окружающей средой. Например, поисковый агент, который находит интересные Web-страницы, должен уметь общаться с помощью протокола HTTP, чтобы подключаться к серверам и получать от них информацию (искомые страницы). Возможность коммуникации с другими агентами является зависимым свойством.

Агент способен к сотрудничеству (Collaborative), если он может общаться с другими агентами для решения своих задач. Например, агенты, которые действуют от имени пользователей на аукционах, могут участвовать в заключении совместных сделок, ориентируясь на информацию, полученную от пользователей. Подобные агенты обычно входят в системы с большим количеством агентов, так как для сотрудничества требуется множество агентов. Умение сотрудничать очень важно для программных агентов, поэтому в этом направлении проводятся тщательные исследования.

Для некоторых агентов наиболее существенной является способность к персонализации (Personality). Программы с их применением обычно используются в развлекательных целях. Поэтому, если персонализация агента будет изменять его в зависимости от передаваемой информации, пользователь сможет понять то,

что агент пытается сообщить. Например, когда рассказчик о чем-либо повествует и выражение его лица при этом не изменяется, слушателю может быть трудно его понять. Информация, закодированная в человеческих чувствах, позволяет получить дополнительные сведения, поэтому возможность персонализации может быть важной для агентов определенного класса.

Наконец, некоторые агенты обладают мобильностью (Mobility), или возможностью перемещения по окружающей среде. Это свойство является очень полезным при создании определенных программ. Представьте агента, который получает информацию из удаленной базы данных. При обычных условиях хост принимает информацию от базы данных, собирает нужные сведения, а затем фильтрует их с помощью критериев, заданных пользователем. При получении мобильности агент может быть направлен в удаленную базу данных, чтобы автоматически отфильтровать результаты, а затем вернуть только те, которые необходимы пользователю. Это позволяет как сэкономить на пропускной способности канала, так и упростить архитектуру, особенно в том случае, если два хоста могут быть отсоединены друг от друга.

Агент необязательно должен обладать всеми свойствами. В большинстве случаев используются всего два. В примере, о котором будет рассказано в этой главе, агент, фильтрующий новости, имеет свойства автономности и сотрудничества.

Строение агентов

Вернемся к описанию агентов и идентифицируем ряд высокоуровневых агентов, которые имеют указанные выше свойства (рис. 11.1).

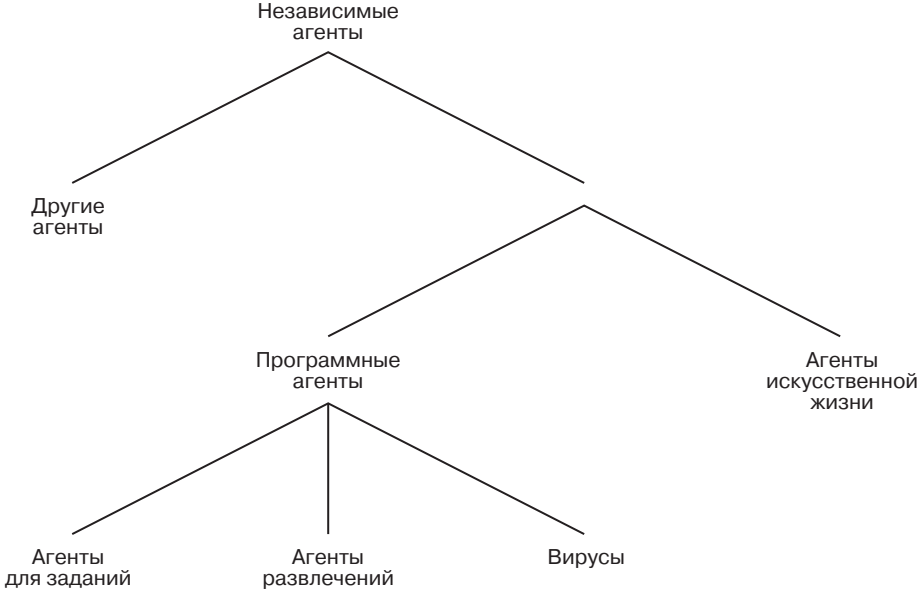


Рис. 11.1. Иерархия агентов по Франклину и Граессеру

Поскольку здесь рассказывается только об автономных агентах, будет использоваться классификация по Франклину и Граессеру (Franklin and Graesser). Есть и другие методы классификации, однако мы сфокусируемся на определении программных агентов с помощью данной методики.

Агенты, использование которых зависит от задания

Агент, использование которого зависит от задания, применяется для решения определенной проблемы. Примером может служить поисковый агент в сети Internet. Кроме поиска по заданным критериям, агент также следит за тем, какие из найденных ссылок откроет пользователь. Это позволяет создать шкалу ценностей поиска, которая способствует лучшей сортировке результатов, а также ускоряет получение пользователем нужного результата. Поисковый агент может работать в фоновом режиме, повторяя поиск, чтобы найти и предоставить пользователю новую информацию.

Поисковые агенты обладают не только автономностью, но и адаптивностью, поскольку они изучают наиболее интересные для пользователя темы, основываясь на его реакции (то есть ссылках, которые открывает пользователь).

Другим примером может служить агент, который используется для работы на аукционе. Он способен действовать от имени пользователя, покупая товары в сети Internet по самой низкой цене. Он может делать это в сотрудничестве с другими агентами. Найдя агентов, которые пытаются купить похожие товары, можно совершить совместную оптовую покупку группы товаров по сниженной цене.

Развлекательные агенты

Развлекательные агенты относятся к типу, который используется для создания умного агента. Агенты данного типа полезны для взаимодействия в виртуальном мире или представления персонажа в качестве интерфейса для пользователя. Например, агент Ананова (Ananova) компании Ananova Ltd. представляет собой говорящую голову, которая читает новости, используя графическую синхронизацию губ. Агент также обладает функцией имитации естественного движения головы, которая помогает ему казаться более реалистичным.

Агент Ананова имеет персональные и коммуникативные характеристики. По своей природе он обладает личностными свойствами, хотя ему трудно проявить их во время чтения новостей. Агент Ананова также является коммуникативным: его губы двигаются в соответствии с произносимым текстом, что усиливает его сходство с обычным диктором.

Вирусы

Вирус по-другому можно назвать опасным мобильным программным агентом. Хотя вирус не обязательно должен быть адаптивным, он является автономным и, что важнее, мобильным. Вместо того чтобы использовать специальный протокол для перемещения по сети, вирус применяет стандартные протоколы и чаще всего протокол передачи почты.

Структура мобильного агента используется и для перемещения безопасных агентов. Например, к ним относится структура Aglet компании IBM, которую можно бесплатно загрузить из сети Internet (см. раздел «Литература и ресурсы»).

Как сделать агентов разумными

Хотя сделать агента разумным довольно сложно, существует множество методов, которыми вы можете пользоваться, чтобы придать агенту способность к принятию разумных решений. О некоторых из них рассказывается в данной книге (рис. 11.2).

Рис. 11.2. Как придать агентам способность к принятию разумных решений

В главе 5 обсуждалось создание «умных» персонажей для компьютерных игр, использующих нейронную сеть с алгоритмом обратного распространения. Одним из основных параметров нейронных сетей является то, что они могут не только генерировать удачные решения для непредвиденных ситуаций, но и модифицировать свое поведение в зависимости от изменений в окружающей среде.

В главе 8 рассматривались системы, основанные на правилах, в которых используется схема прямого логического вывода. Они позволяют агенту делать заключения об окружающей среде с помощью набора правил и фактов, которые ее описывают. Важным свойством подобных систем является то, что они могут отображать процесс принятия решения.

Алгоритмы кластеризации (например, теория адаптивного резонанса, описанная в главе 3) тоже очень полезны при создании умных агентов. Они позволяют агенту идентифицировать свои связи в среде, чтобы он мог изучать новые модели без каких-либо указаний (произвольное обучение).

Могут применяться различные другие методики, в том числе и те, которые были описаны в предыдущих главах. Например, в качестве элемента умных агентов рекомендуется использовать нечеткую логику (описанную в главе 9) или алгоритм отжига (глава 2).

Пример применения

Рассмотрим очень простой пример агента, который предоставляет сервис фильтрации. Задача агента состоит в том, чтобы взаимодействовать со службами новостей в сети Internet и собирать информацию на основании критериев, заданных пользователем. Затем агент-фильтр предоставит ее пользователю в том порядке, который более всего соответствует запросу (начиная с данных, имеющих самый высокий рейтинг по условиям поиска).

Разработка Web-агента

Web-агент использует стандартные протоколы и передает пользователю информацию через обычный браузер. Для общения с пользователем агент содержит HTTP-сервер, для связи с внешними серверами является HTTP-клиентом, а для получения новостей функционирует как клиент протокола NNTP (рис. 11.3).

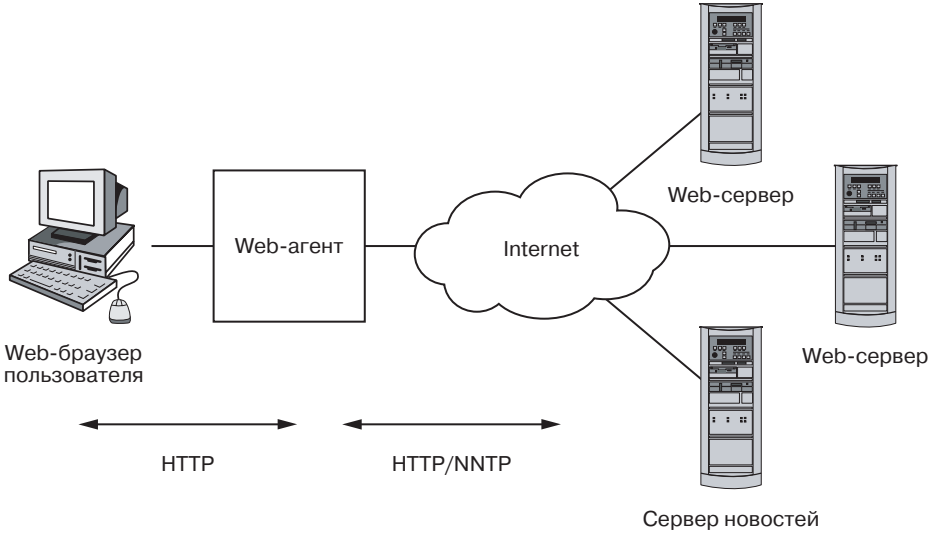


Рис. 11.3. Архитектура Web-агента

Web-агент будет играть роль приложения, которое осуществляет взаимодействие между пользователем и сетью Internet при чтении новостей. Пользователь изменяет параметры агента с помощью простого файла конфигурации (подробнее о нем будет рассказано позже). В нем пользователь указывает Web-страницы

для мониторинга (для выдачи сообщения в случае их изменения) и группы новостей для просмотра и определяет специфические критерии поиска. Когда агент находит объекты, которые соответствуют заданному критерию, он сохраняет их и выдает пользователю в виде Web-страницы, ссылки на которой отсортированы в нужном порядке. Пользователь может щелкнуть по ссылке на определенный Web-сайт или строке новостей и перейти к нужной статье.

Задача агента заключается в том, чтобы упростить чтение новостей, отфильтровать ненужную информацию и представить новости в порядке, при котором наиболее интересные данные стоят на первом месте.

Свойства Web-агента

Web-агент очень прост и не имеет способности к обучению. Он обладает всего двумя свойствами: автономностью (способен работать в фоновом режиме без прямого управления со стороны пользователя) и способностью к коммуникации с внешними серверами при сборе информации.

Сенсорами агента являются стандартные протоколы, которые позволяют ему собирать нужную информацию по заданным критериям поиска. Для передачи результатов поиска пользователю используется протокол HTTP. Затем пользователь подключается к Web-агенту, как к любому другому Web-серверу.

Примечание *Исходный код Web-агента содержится в архиве с примерами, который находится на сайте издательства «ДМК Пресс» www.dmk.ru.*

Обсуждение исходного кода

Теперь рассмотрим исходный код Web-агента. Чтобы лучше понять структуру агента, изучим схему потоков данных внутри него (см. рис. 11.4).

Сбор данных повторяется через каждые 10 мин. Данные помещаются в специальный репозиторий. Используя критерии поиска, сохраненные в файле конфигурации (заданном пользователем), программа сокращает количество данных, то есть удаляет те статьи, которые не соответствуют критериям поиска. Когда пользователь запрашивает текущую информацию, генератор HTML создает Web-страницу, которая передается обратно через HTTP-сервер (рис. 11.5).

Затем пользователь может просмотреть новости, щелкнув по ссылке (рис. 11.6).

Когда пользователь завершит чтение интересующих его статей, он может нажать кнопку **Пометить как прочитанные**, чтобы очистить текущий список новостей. Сохранение выполняется через Web-агента; при этом данная подборка новостей повторно показываться не будет.

Конфигурация задается в файле, который находится вне Web-агента, однако пользователь может просматривать установки конфигурации через самого агента. Текущие установки показаны в файле config.html (рис. 11.7).

В следующих разделах рассматривается исходный код Web-агента по схеме потоков данных, представленных на рис. 11.4. Слой интерфейсов сети Internet

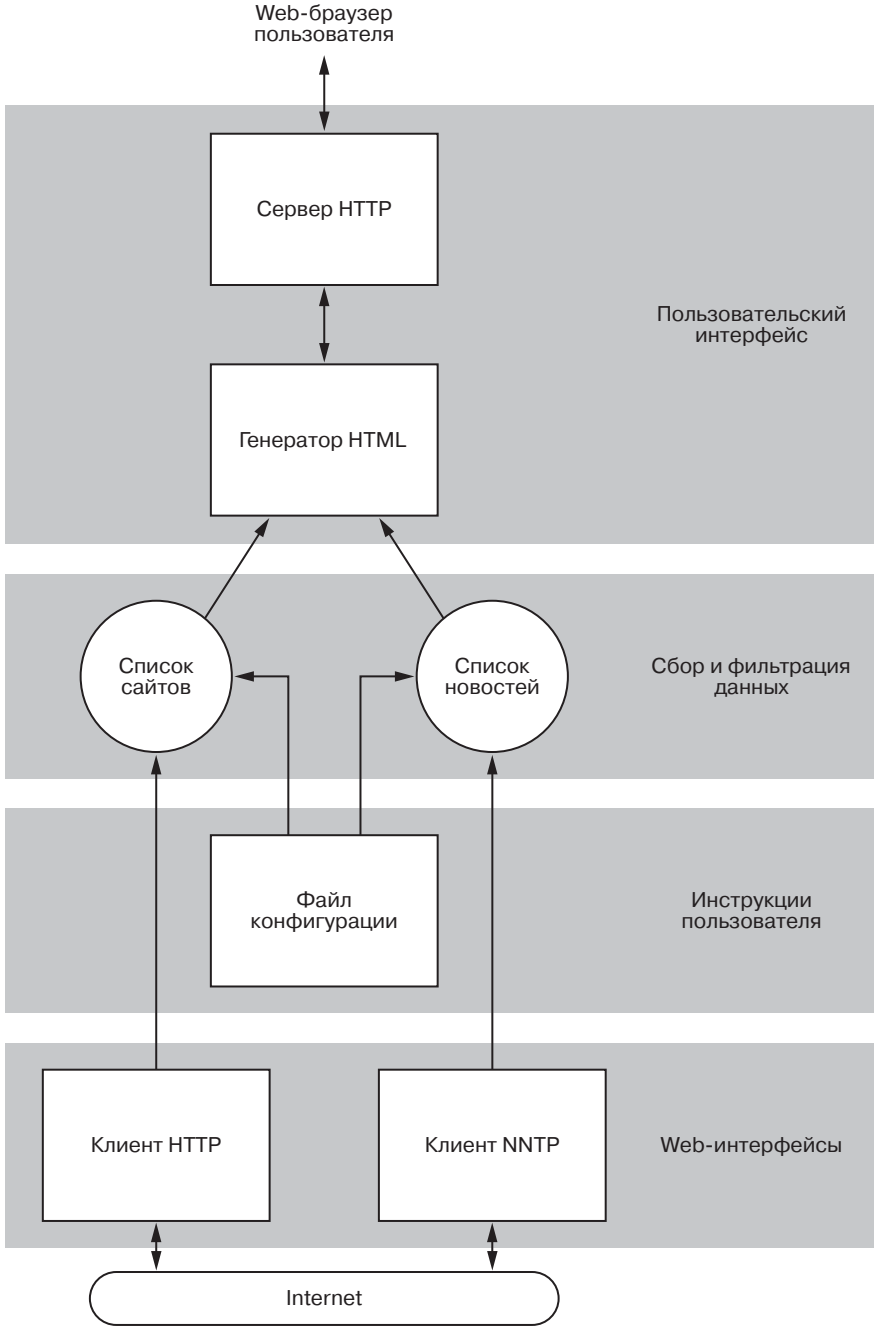


Рис. 11.4. Схема потоков данных внутри Web-агента

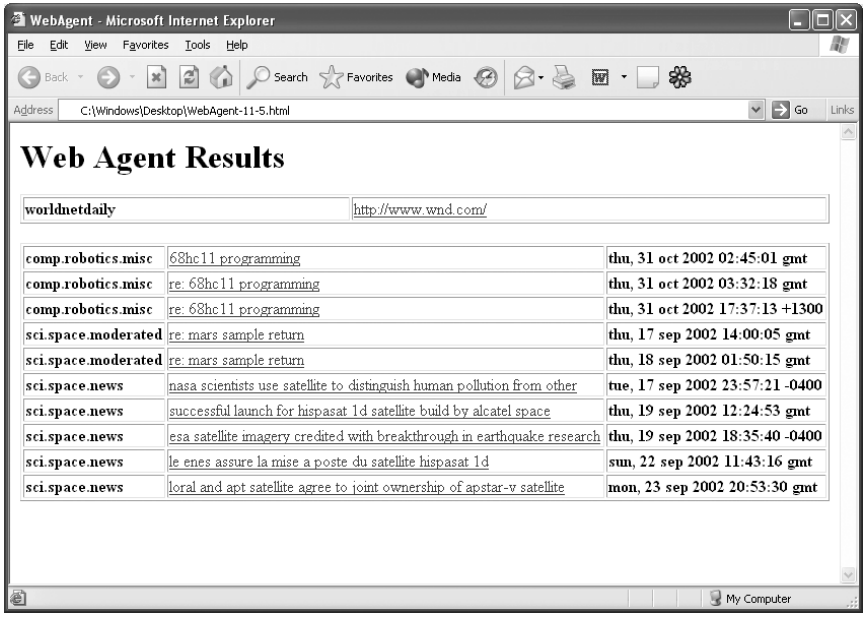


Рис. 11.5. Пример Web-страницы, выданной Web-агентом

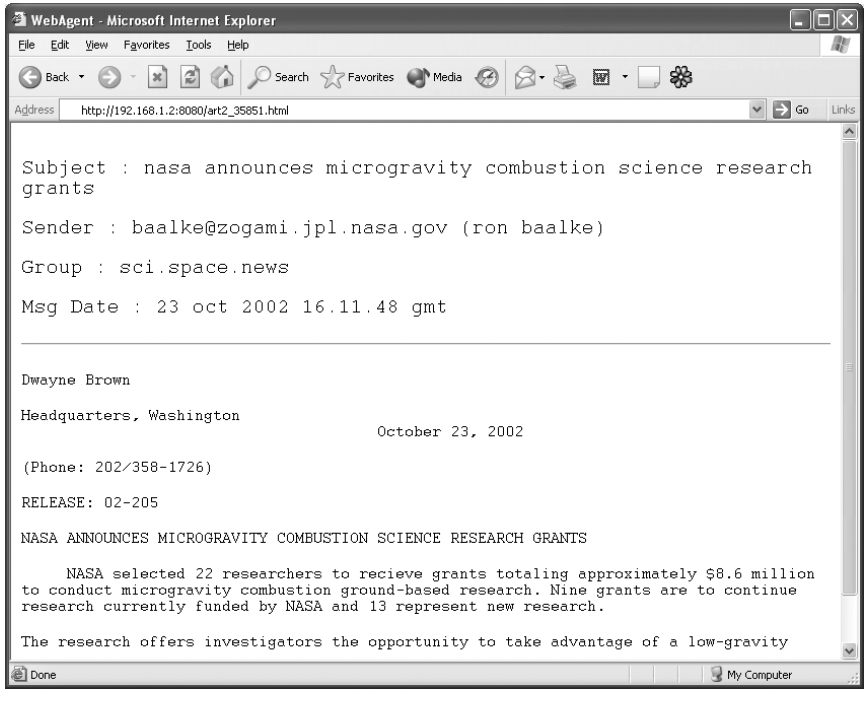


Рис. 11.6. Пример новостной статьи

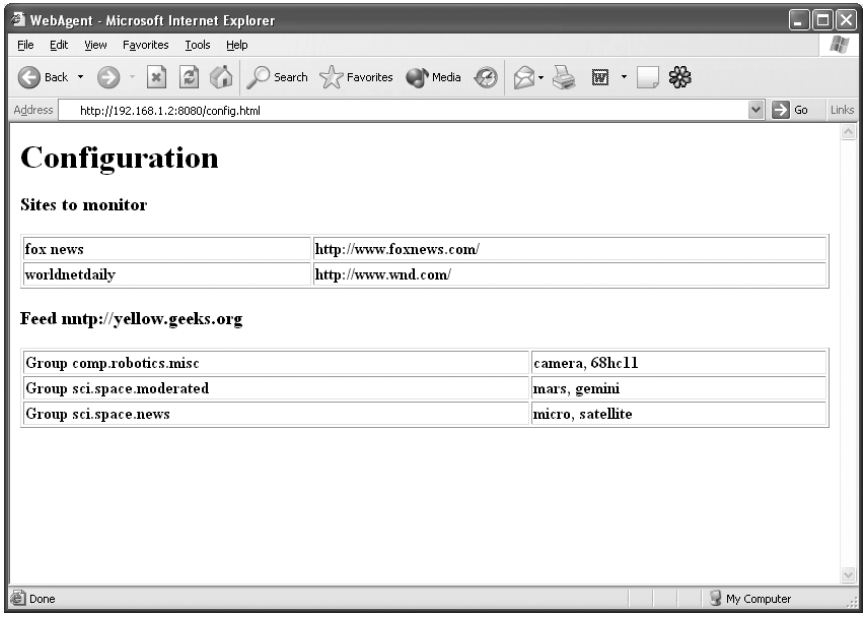


Рис. 11.7. Просмотр конфигурации агента сети Internet

позволяет осуществлять коммуникацию с внешними серверами с помощью стандартных протоколов. Слой инструкций пользователя определяет, как пользователь передаст агенту ограничения поиска. Слой сбора и фильтрации данных выполняет фильтрацию входящих данных в соответствии с указаниями пользователя. Наконец, слой пользовательского интерфейса обеспечивает связь с HTTP-сервером для просмотра отфильтрованных новостей.

Web-интерфейсы

Web-агент применяет простые варианты интерфейса клиента NNTP и интерфейса клиента HTTP. Также используется HTTP-сервер, о котором будет рассказано в разделе, описывающем реализацию пользовательского интерфейса.

Простой клиент протокола HTTP

Задача клиента протокола HTTP – мониторинг сайта, иными словами, программа должна определять изменения Web-сайта и сообщать о них пользователю. Для этого интерфейс клиента HTTP применяет простую версию запроса `GET`. Целью запроса является получение файла с удаленного сервера. Программу интересует только заголовок, особенно элемент `content-length`, который сообщает о размере содержимого (то есть файла). Размер выступает в качестве индикатора изменений, вносимых в файл. Это не идеальный способ, но, к сожалению, не все серверы посылают измененный заголовок файла.

Функция `monitorSite` (см. листинг 11) использует массив структур, чтобы определить, мониторинг каких сайтов следует выполнять (данная структура будет рассматриваться позже).

Листинг 11.1. Простой интерфейс клиента HTTP

```
typedef struct {
  int  active;
  char url[MAX_URL_SIZE];
  char urlName[MAX_SEARCH_ITEM_SIZE+1];
  int  length;
  int  changed;
  int  shown;
} monitorEntryType;

int monitorSite( int siteIndex )
{
  int ret=0, sock, result, len;
  struct sockaddr_in servaddr;
  char buffer[MAX_BUFFER+1];
  char fqdn[80];

  extern monitorEntryType monitors[];

  /* Создать новый клиентский сокет */
  sock = socket(AF_INET, SOCK_STREAM, 0);

  prune( monitors[siteIndex].url, fqdn );

  memset(&servaddr, 0, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons( 80 );

  /* Проверяем адрес */
  servaddr.sin_addr.s_addr = inet_addr( fqdn );

  /* Если у нас не IP-адрес, тогда это должно быть имя домена.
   * Попробуем преобразовать его в IP-адрес с помощью DNS
   */
  if ( servaddr.sin_addr.s_addr == 0xffffffff ) {
    struct hostent *hptr =
      (struct hostent *)gethostbyname( fqdn );
    if ( hptr == NULL ) {
      close(sock);
      return -1;
    } else {
      struct in_addr **addrs;
```

```
        addrs = (struct in_addr **)hptr->h_addr_list;
        memcpy( &servaddr.sin_addr, *addrs, sizeof(struct in_addr) );
      }
    }

    /* Подключаемся к HTTP-серверу */
    result = connect(sock,
                 (struct sockaddr *)&servaddr, sizeof(servaddr));

    if (result == 0) {

      /* Отсылаем простую команду GET */
      strcpy(buffer, "GET / HTTP/1.0\n\n");

      len = write(sock, buffer, strlen(buffer) );

      if ( len == strlen(buffer) ) {
        char *cur;

        len = grabResponse( sock, buffer );

        cur = strstr(buffer, "Content-Length:");

        if (cur != NULL) {
          int curLen;

          sscanf(buffer, "Content-Length: %d", &curLen);

          if (len != monitors[siteIndex].length) {
            monitors[siteIndex].shown = 0;
            monitors[siteIndex].changed = 1;
            monitors[siteIndex].length = len;
            ret = 1;
          }
        }

      }
    }

  close(sock);

  return(ret);
  }
```

Структура `monitorEntryType` включает поле с адресом сервера (URL) и поле `urlName`, содержащее название Web-сайта, для которого выполняется мониторинг. Поле `length` включает информацию о длине запрашиваемой страницы, по которой программа определяет, была изменена страница или нет.

Примечание *Функция* `monitorSite` *является очень простым приложением, работающим через сокет. Сокет создается с помощью функции* `socket`. *Сокращение адреса URL выполняется с помощью функции* `prune`, *которая берет адрес в форме http://www.mjtones.com/ и переводит его в форму www.mtjones.com (не показано в тексте книги, см. полный исходный код в архиве с примерами, который вы можете загрузить с сайта издательства «ДМК Пресс» www.dmk.ru).*

Получившийся адрес, который также называют полностью квалифицированным именем домена, может быть разрешен, то есть преобразован в IP-адрес. Используя этот IP-адрес, программа может подключиться к удаленному серверу. Обратите внимание, что адрес сервера может быть задан как IP-адрес и как полное доменное имя. Поэтому сначала используется функция `inet_addr` для конвертации текстового IP-адреса в числовой IP-адрес. Если она не срабатывает, то функция `gethostbyname` конвертирует имя в IP-адрес с помощью DNS-сервера.

Получив IP-адрес (в структуре `servaddr`), программа подключается к удаленному серверу с помощью функции `connect`. Эта функция создает двустороннее соединение между двумя конечными точками, которое может использоваться для связи. Поскольку выполняется подключение к HTTP-порту на удаленном сервере (порт 80), программа знает, что для этого сокета в качестве протокола используется HTTP. Она посылает команду `GET` и ждет, когда сервер ответит. Получив ответ (с помощью функции `grabResponse`, листинг 11.2), помещенный в буфер, программа ищет элемент заголовка `Content-length` и сохраняет указанное в нем значение. Если длина изменяется по сравнению с сохраненным значением, для значения делается пометка об изменении, которая отобразится в фильтрующем слое.

Листинг 11.2. Получение ответа от HTTP-сервера

```
int grabResponse ( int sock, char *buf)
{
     int i, len, stop, state, bufIdx;

     if (buf == NULL) return -1;

     len = bufIdx = state = stop = 0;

     while (!stop) {

     if (bufIdx+len > MAX_BUFFER -80) break;

     len = read( sock, &buf[bufIdx], (MAX_BUFFER-bufIdx) );

     /* Ищем идентификатор конца сообщения в буфере */
    for ( i = bufIdx ; i < bufIdx+len ; i++ ) {
```

```
        if      ( (state == 0) && (news->msg[i] == 0x0d) ) state = 1;
        else if ( (state == 1) && (news->msg[i] == 0x0a) ) state = 2;
        else if ( (state == 2) && (news->msg[i] == 0x0d) ) state = 1;
        else if ( (state == 2) && (news->msg[i] ==  ".") ) state = 3;
        else if ( (state == 3) && (news->msg[i] == 0x0d) ) state = 4;
        else if ( (state == 4) && (news->msg[i] == 0x0a) ) { stop = 1;
           break; }
        else state = 0;
      }

      bufIdx += len;

    }
    bufIdx -= 3;

    news->msg[bufIdx] = 0;

    return bufIdx;
}
```

Ответ HTTP-сервера заканчивается двумя парами символов CR/LF. Простая машина состояний (листинг 11.2) считывает данные из сокета до тех пор, пока не находит указанную комбинацию символов. После этого цикл завершается, и в конец буфера добавляется символ с кодом 0.

Таким образом устроен простой HTTP-клиент. Для каждого отслеживаемого сайта создается сокет и отправляется запрос по протоколу HTTP. Затем программа сохраняет ответ, чтобы с его помощью проанализировать размер содержимого сайта. По этому значению определяется, изменялся ли сайт со времени последней проверки.

Простой NTTP-клиент

Клиент NTTP реализует набор интерфейсов для взаимодействия с новостными серверами. Эти интерфейсы позволяют приложению соединяться с сервером новостей (`nttpConnect`), задавать группы новостей по интересам (`nttpSetGroup`), переходить к заголовку статьи (`nttpPeek`), считывать всю статью (`nttpRetrieve`), анализировать сообщение (`nttpParse`), пропускать текущую новость (`nttpSkip`) и прерывать связь с сервером новостей (`nttpClose`).

Протокол NTTP является интерактивным протоколом, который полностью базируется на ASCII-командах. При открытии сессии протокола Telnet по порту 119 NTTP-сервера (листинг 11.3) осуществляется диалог с сервером. Данные, вводимые пользователем, выделены полужирным шрифтом.

Листинг 11.3. Пример работы с NTTP-сервером

```
root@plato /root]# telnet localhost 119
S: 201 plato.mtjones.com DNEWS Version 5.5d1, SO, posting OK
C: list
```

```
S: 215 list of newsgroups follows
S: control 2 3 y
S: control.cancel 2 3 y
S: my.group 10 3 y
S: new.group 6 3 y
S: .
C: group my.group
S: 211 8 3 10 my.group selected
C: article 3
S: 220 3 <3C36AF8E. 1BD3047E@mtjones.com> article retrieved
S: Message-ID: <3C36AF8E. 1BD3047E@mtjones.com>
S: Date: Sat, 05 Jan 2002 00:47:27 -0700
S: From: "M. Tim Jones" <mtj@mtjones.com>
S: X-Mailer: Mozilla 4.74 [en] (Win98; U)
S: X-Accept-Language: en
S: MIME-Version: 1.0
S: Newsgroups: my.group
S: Subject: this is my post
S: Content-Type: text/plain; charset=us-ascii
S: Content-Transfer-Encoding: 7bit
S: NNTP-Posting-Host: sartre.mtjones.com
S: X-Trace: plato.mtjones.com 1010328764 sartre.mtjones.com (6
Jan 2002 07:52:44 -0700)
S: Lines: 6
S: Path: plato.mtjones.com
S: Xref: plato.mtjones.com my.group:3
S:
S:
S: Hello
S:
S: This is my post.
S:
S:
S: .
C: date
S: 111 20020112122419
C: quit
S: 205 closing connection - goodbye!
```

В листинге 11.3 показано, как соединение с NTTP-сервером создается с помощью Telnet-клиента. Сервер новостей отвечает приветствием, в котором указывается тип сервера и другая подобная информация. Теперь через соединение можно давать команды. Существует два базовых типа ответов, которые можно ожидать от сервера – из одной строки и из нескольких строк. Ответ из одной строки прост (см. представленную выше команду `date`). Программе будет легче работать с ответом из нескольких строк, если используется универсальный символ для завершения запроса. Чтобы идентифицировать конец ответа, протокол NNTP, как и протокол SMTP, применяет символ «.» в строке (см. ответ на команды `list` и `article`).

Теперь, когда вы освоили основы протокола NNTP, следует приступить к изучению функций, которые будут использоваться для коммуникации с сервером новостей.

Обычно при работе с протоколом NNTP применяется тип `news_t`, который описывает сообщение (листинг 11.4).

Листинг 11.4. Базовая структура сообщения, `news_t`

```
typedef struct {
  char *msg;
  int  msgLen;
  int  msgId;
  char subject[MAX_LG_STRING+1];
  char sender[MAX_SM_STRING+1];
  char msgDate[MAX_SM_STRING+1];
  char *bodyStart;
} news_t;
```

Структура `news_t` включает непроанализированный буфер (`msg`), длину непроанализированного сообщения (`msgLen`) и числовой идентификатор сообщения (`msgId`). Также из заголовка сообщения выделяется такая информация, как тема (`subject`), отправитель (`sender`) и дата сообщения (`msgDate`). Наконец, поле `bodyStart` указывает на тело сообщения.

Перед запуском новой сессии протокола NNTP необходимо обратиться к функции `nntpConnect` (листинг 11.5). Она создает соединение с NNTP-сервером, используя переданный ей адрес сервера (IP-адрес или доменное имя).

Листинг 11.5. Функция `nntpConnect` в структуре API

```
 int nntpConnect ( char *nntpServer )
{
  int result = -1;
  struct sockaddr_in servaddr;

  if (!nntpServer) return -1;

  sock = socket( AF_INET, SOCK_STREAM, 0 );

  bzero( &servaddr, sizeof(servaddr) );
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons( 119 );

  servaddr.sin_addr.s_addr = inet_addr( nntpServer );

  if ( servaddr.sin_addr.s_addr == 0xffffffff ) {
    struct hostent *hptr =
            (struct hostent *)gethostbyname( nntpServer );
    if ( hptr == NULL ) {
      return -1;
```

```
    } else {
      struct in_addr **addrs;
      addrs = (struct in_addr **)hptr->h_addr_list;
      memcpy( &servaddr.sin_addr, *addrs, sizeof(struct in_addr) );
    }
  }
  printf( "Connecting to %s\n", nntpServer );

  result = connect( sock,
              (struct sockaddr *)&servaddr, sizeof(servaddr) );

  if ( result >= 0 ) {

    buffer[0] = 0;
    result = dialog( sock, buffer, "201", 3 );

    if (result < 0) nntpDisconnect();

  }

  return ( result );
}
```

Функция `nntpConnect` сначала определяет адрес, полученный от пользователя (`nntpServer`). Это может быть IP-адрес или полное доменное имя, поэтому обработка каждого имени выполняется отдельно (см. описание функции `monitorSite`). Затем для соединения с удаленным сервером используется функция `connect`. Для всех ответов протокола NNTP возвращается число (идентификатор ID), которое определяет статус запроса (с помощью функции `dialog`). При первом соединении NNTP-сервер должен возвратить код 201, означающий успешное соединение (см. листинг 11.3). Если этот код найден, возвращается сообщение об успешном соединении, в противном случае сессия NNTP прерывается и возвращается сообщение об ошибке (–1).

Функция `dialog` применяется всеми функциями, работающими с NNTP-сервером, для анализа ответа сервера (листинг 11.6). Программа передает в функцию идентификатор сокета, буфер, в котором хранится ответ, код ожидаемого ответа и длину кода. Для хранения данных используется собственный буфер программы, поскольку протокол TCP может «перепаковывать» пакеты данных. Даже несмотря на то что сервер способен выводить строку, включающую информацию о статусе, и строку с данными, стек сети может комбинировать эти строки перед отправкой. Поэтому требуется сохранить ответ в буфере, который содержит дополнительные данные для последующего использования.

Листинг 11.6. Функция взаимодействия с NNTP-сервером

```
int dialog( int sd, char *buffer, char *resp, int rlen )
{
  int ret, len;
```

```
  if ((sd == -1) || (!buffer)) return -1;

  if (strlen(buffer) > 0) {
    len = strlen( buffer );
    if ( write( sd, buffer, len ) != len ) return -1;
  }

  if (resp != NULL) {
    ret = read( sd, buffer, MAX_LINE );
    if (ret >= 0) {
      buffer[ret] = 0;
      if (strncmp( buffer, resp, rlen )) return -1;
  } else {
      return -1;
    }
  }

  return 0;
}
```

Поскольку не всегда можно послать команду серверу, программа проверяет аргумент `buffer`, чтобы определить, есть ли в нем команда (с помощью функции `strlen`). Если команда найдена, она посылает ее через сокет с помощью функции `write`. Ответ не всегда соответствует тому, что ожидалось. Если при вызове буфер передан с кодом возврата, то функция читает из сокета данные и проверяет наличие в ответе сервера «правильного» статуса. Если нужный статус найден, то возвращается код успешного выполнения (0), в противном случае возвращается сообщение об ошибке (–1).

После подключения к серверу новостей требуется определить нужную группу. Это выполняется с помощью функции `nntpSetGroup`. Программа указывает название интересующей группы (например, `comp.ai.alife`) и читает последнее сообщение группы. При инициализации вызывающая функция передает значение –1 в параметре `lastRead`. Это значит, что ни одно сообщение не было прочитано. В противном случае можно указать последнее прочитанное сообщение, что позволит клиенту протокола NNTP проигнорировать те сообщения, которые были прочитаны ранее (листинг 11.7).

Листинг 11.7. Функция установки группы новостей `nntpSetGroup`

```
int nntpSetGroup( char *group, int lastRead )
{
  int result = -1;
  int numMessages = -1;

  if ((!group) || (sock == -1)) return -1;

  snprintf( buffer, 80, "group %s\n", group );
```

```
  result = dialog( sock, buffer, "211", 3 );

  if (result == 0) {
    sscanf( buffer, "211 %d %d %d ",
             &numMessages, &firstMessage, &lastMessage );

    if (lastRead == -1) {
      curMessage = firstMessage;
    } else {
      curMessage = lastRead+1;
      numMessages = lastMessage - lastRead;
    }

    printf("Set news group to %s\n", group);

  }

  return( numMessages );
}
```

Для определения новостной группы в протоколе NNTP используется соответствующая команда. Ответ будет состоять из кода 211 (если подключение произошло успешно) и трех цифр: общее количество сообщений, а также номер первого и последнего сообщений. Эта информация сохраняется клиентом и используется при последующих вызовах. Функция возвращает количество непрочитанных сообщений.

После того как программа соединилась с группой новостей, пользователь может загрузить сообщения с сервера с помощью идентификатора, связанного с каждым сообщением. Для считывания сообщений с сервера используются две функции: `nntpPeek` и `nntpRetrieve`.

Функция `nntpPeek` считывает только заголовок сообщения (листинг 11.8), в то время как функция `nttpRetrieve` читает все сообщение (заголовок и тело сообщения).

Листинг 11.8. Функция загрузки заголовка сообщения `nntpPeek`

```
int nntpPeek ( news_t *news, int totalLen )
{
  int result = -1, i, len=0, stop, state, bufIdx=0;

  if ((!news) || (sock == -1)) return -1;

  if ((curMessage == -1) || (curMessage > lastMessage)) return -2;

  /* Сохраним ID сообщения */
  news->msgId = curMessage;

  snprintf( buffer, 80, "head %d\n", curMessage );
```

```
  result = dialog( sock, buffer, "21", 3 );

  if (result < 0) return -3;

  /* Пропускаем префикс +OK и выделяем данные (пока не встретим
   * CRLF)
   */
  len = strlen( buffer );
  for ( i = 0 ; i < len-1 ; i++ ) {
    if ( (buffer[i] == 0x0d) && (buffer[i+1] == 0x0a) ) {
      len -= i-2;
      memmove( news->msg, &buffer[i+2], len );
      bufIdx = len;
      break;
    }
  }

  state = stop = 0;

  while (!stop) {

    if (bufIdx+len > totalLen - 80) break;

    len = read( sock, &news->msg[bufIdx], (totalLen-bufIdx) );

   /* Ищем признак конца сообщения в буфере */
    for ( i = bufIdx ; i < bufIdx+len ; i++ ) {
      if      ( (state == 0) && (news->msg[i] == 0x0d) ) state = 1;
      else if ( (state == 1) && (news->msg[i] == 0x0a) ) state = 2;
      else if ( (state == 2) && (news->msg[i] == 0x0d) ) state = 1;
      else if ( (state == 2) && (news->msg[i] ==  ".") ) state = 3;
      else if ( (state == 3) && (news->msg[i] == 0x0d) ) state = 4;
      else if ( (state == 4) && (news->msg[i] == 0x0a) ) {
                  stop = 1;  break; }
      else state = 0;
    }

    bufIdx += len;

  }

  bufIdx -= 3;
  news->msg[bufIdx] = 0;
  news->msgLen = bufIdx;

  return bufIdx;
}
```

Первая задача функции `nntpPeek` заключается в том, чтобы выдать команду `head` через сокет на сервер NNTP. NNTP-сервер должен ответить кодом статуса 221, который обозначает, что исполнение команды прошло успешно. Затем в буфер копируются (`news->mag`) другие данные, которые могут сопровождать код статуса от NNTP-сервера. Наконец, программа читает дополнительные данные из сокета до тех пор, пока не будет найден индикатор конца сообщения (символ «`.`» в пустой строке). В этот момент сообщение (сохраненное в поле `news->msg`) включает только заголовок сообщения и может анализироваться соответствующим образом (см. описание функции `nntpParse`).

Функция `nntpRetrieve` очень схожа с функцией `nntpPeek` за исключением того, что она загружает все сообщение, а не только его заголовок (листинг 11.9).

Листинг 11.9. Функция загрузки сообщения `nntpRetrieve` API

```
int nntpRetrieve ( news_t *news, int totalLen )
{
  int result = -1, i, len=0, stop, state, bufIdx=0;

  if ((!news) || (sock == -1)) return -1;

  if ((curMessage == -1) || (curMessage > lastMessage)) return -1;

  /* Сохраняем ID сообщения */
  news->msgId = curMessage;

  snprintf( buffer, 80, "article %d\n", curMessage++ );

  result = dialog( sock, buffer, "220", 3 );

  if (result < 0) return -1;

  len = strlen(buffer);
  for ( i = 0 ; i < len-1 ; i++ ) {
    if ( (buffer[i] == 0x0d) && (buffer[i+1] == 0x0a) ) {
      len -= i-2;
      memmove( news->msg, &buffer[i+2], len );
      break;
    }
  }

  state = stop = 0;

  while (!stop) {

    if (bufIdx+len > totalLen - 80) break;

    /* Ищем индикатор конца сообщения в буфере */
```

```
    for ( i = bufIdx ; i < bufIdx+len ; i++ ) {
      if      ( (state == 0) && (news->msg[i] == 0x0d) ) state = 1;
      else if ( (state == 1) && (news->msg[i] == 0x0a) ) state = 2;
      else if ( (state == 2) && (news->msg[i] == 0x0d) ) state = 1;
      else if ( (state == 2) && (news->msg[i] ==  ".") ) state = 3;
      else if ( (state == 3) && (news->msg[i] == 0x0d) ) state = 4;
      else if ( (state == 4) && (news->msg[i] == 0x0a) ) {
                stop = 1; break; }
      else state = 0;
    }

    bufIdx += (i-bufIdx);

    if (!stop) {

      len = read( sock, &news->msg[bufIdx], (totalLen-bufIdx) );

      if ( (len <= 0) || (bufIdx+len > totalLen) ) {
        break;
      }

    }

  }

  bufIdx -= 3;
  news->msg[bufIdx] = 0;
  news->msgLen = bufIdx;

  return bufIdx;
}
```

Функция `nntpRetrieve` использует команду `article` протокола NNTP, чтобы запросить сообщение целиком. Вспомните, что функция `nntpPeek` использует команду `head`, чтобы запрашивать заголовки сообщений. Протокол NNTP возвращает заголовок и целое сообщение аналогичным способом, используя символ «`.`» в пустой строке в качестве индикатора конца сообщения. Поэтому команды `nntpPeek` и `nntpRetrieve` имеют схожие свойства, но различаются количеством данных, которое получается в итоге. Функция `nntpPeek` не выдает текущее сообщение в отличие от функции `nntpRetrieve`. Это объясняется следующим образом: функция `nntpPeek` используется для того, чтобы определить, нужно загружать все сообщение или нет. Если пользователь не желает загружать сообщение, он может задействовать функцию `nntpSkip`, чтобы перейти к следующему сообщению (листинг 11.10).

Листинг 11.10. Функция `nntpSkip` API

```
 void nntpSkip( void )
{
```

```
    curMessage++;
  }
```

Вспомните, что переменная `curMessage` является статической в NNTP-клиенте и инициализируется при вызове команды `nntpSetGroup`.

После того как сообщение (или заголовок) было загружено с NNTP-сервера, оно помещается в структуру `news_t` (листинг 11.4). Затем структура может быть передана в функцию `nntpParse` (листинг 11.11), чтобы выделить тему, дату и отправителя сообщения. Кроме того, функция находит начало сообщения (без учета заголовков NNTP), а затем загружает его в поле `bodyStart`.

Листинг 11.11. Функция анализа сообщения `nntpParse`

```
 int nntpParse( news_t *news, unsigned int flags )
{
  int result;

  if (!news) return -1;

  result = parseEntry( news, "Subject:", news->subject );
  if (result < 0) return -1;

  result = parseEntry( news, "Date:", news->msgDate );
  if (result < 0) return -2;

  result = parseEntry( news, "From:", news->sender );
  if (result < 0) return -3;
  fixAddress( news->sender );

  if (flags == FULL_PARSE) {
    result = findBody( news );
  }

  return result;
}
```

Вы должны указать, желаете ли вы анализировать только заголовок или сообщение целиком. Если нужен только заголовок, можно известить об этом программу с помощью команды `HEADER_PARSE`. Для полного анализа (включающего идентификацию тела сообщения) необходимо использовать константу `FULL_PARSE`.

Совет *Функция `nntpParse` использует вспомогательные функции `parseEntry` и `findBody`. Они не отражены в тексте, но содержатся в архиве с примерами к книге, который можно загрузить с сайта издательства «ДМК Пресс» www.dmk.ru.*

Функция `parseEntry` анализирует заголовок сообщения, которое было передано в функцию. Функция `findBody` находит начало тела сообщения (если передано сообщение целиком и установлен соответствующий флаг).

Последняя функция для работы с протоколом NNTP, `nntpDisconnect`, прерывает соединение с NNTP-сервером (листинг 11.12.).

Листинг 11.12. Функция `nntpDisconnect` API

```
int nntpDisconnect ( void )
{
  if (sock == -1) return -1;
  close(sock);
  sock = curMessage = firstMessage = lastMessage = -1;
  return 0;
}
```

Кроме закрытия сокета, связанного с сессией протокола NNTP, она сбрасывает внутренние переменные состояния, которые будут использоваться при открытии новой сессии.

Эти семь функций позволяют Web-агенту считывать данные с NNTP-сервера и фильтровать их на основании критериев, которые определяет пользователь.

Инструкции пользователя

С помощью простого текстового файла конфигурации пользователь передает Web-агенту критерии фильтрации сообщений. Этот файл имеет следующий формат (см. листинг 11.13.).

Листинг 11.13. Пример файла конфигурации для агента сети Internet

```
#
# Sample config file
#
[monitor]
http://www.foxnews.com;Fox News
http://www.wnd.com/;WorldNetDaily

[feeds]
nntp://yellow.geeks.org

[groups]
comp.robotics.misc;camera;68HC11
sci.space.moderated;mars;gemini
sci.space.news;micro;satellite
```

Файл конфигурации состоит из трех частей, причем каждая часть является опциональной, то есть необязательной. Первый раздел (секция `[monitor]`) содержит адреса Web-сайтов для мониторинга. Web-сайты должны указываться в полном формате, включая спецификацию протокола http://. После адреса вводится точка с запятой, которая позволяет отделить текстовое название сайта (используется только для показа информации).

Web-агент поддерживает только один источник, который указывает, откуда агент может получать информацию о новостях (секция `[feeds]`). Источник

задается в виде ссылки, включающей спецификацию протокола (в данном случае nntp://, что позволяет указать протокол NNTP). Здесь вводится адрес сервера новостей, к которому может подключаться агент. В качестве примера использовался адрес бесплатного (и надежного) сервера новостей.

После получения информации об источнике новостей можно задать одну или несколько групп (секция `[groups]`). Каждая строка может включать определение для группы новостей, отделенное точкой с запятой. Каждое слово, следующее за спецификацией группы новостей, представляет собой ключевое слово для поиска. Агент сети Internet с его помощью будет определять, какую статью следует показать пользователю. Чем больше слов было найдено в теме статьи, тем выше ее рейтинг, значит, она будет занимать одну из верхних позиций в списке.

Рассмотрим процесс анализа файла. Вспомните описание структуры `monitorEntryType`, используемой для мониторинга Web-сайтов (листинг 11.1). В листинге 11.14 представлена структура `feedEntryType`, содержащая источник новостей и группы новостей, которые следует проверять.

Листинг 11.4. Структуры `feedEntryType` и `groupEntryType`

```
#define MAX_URL_SIZE            80
#define MAX_SEARCH_ITEM_SIZE    40
#define MAX_SEARCH_STRINGS      10

#define MAX_GROUPS    20

typedef struct {
  int  active;
  char groupName[MAX_URL_SIZE+1];
  int  lastMessageRead;
  char searchString[MAX_SEARCH_STRINGS][MAX_SEARCH_ITEM_SIZE+1];
  int  numSearchStrings;
} groupEntryType;

typedef struct {
  char           url[MAX_URL_SIZE];
  groupEntryType groups[MAX_GROUPS];
} feedEntryType;
```

Структура `feedEntryType` включает адрес самого источника новостей (полученный из секции `[feeds]` файла конфигурации) и массив структур, описывающих группы новостей и критерии поиска в них сообщений (из секции `[groups]`). Информация о группе включает ее название, последнее прочитанное сообщение в группе и набор строк для поиска. Эти связанные структуры определяют рабочее состояние функции мониторинга новостей для агента.

Первым шагом при анализе файла конфигурации является вызов функции `parseConfigFile`. Это главная функция анализа, которая проверяет все три части файла конфигурации (листинг 11.15).

Листинг 11.15. Функция анализа главного файла конфигурации

```
int parseConfigFile( char *filename )
{
  FILE *fp;
  char line[MAX_LINE+1], *cur;
  int parse, i;

  bzero( &feed, sizeof(feed) );
  bzero( monitors, sizeof(monitors) );

  fp = fopen(filename, "r");

  if (fp == NULL) return -1;

  while( !feof(fp) ) {

    fgets( line, MAX_LINE, fp );

    if (feof(fp)) break;

    if      (line[0] == "#") continue;
    else if (line[0] == 0x0a) continue;

    if        (!strncmp(line, "[monitor]", 9)) {
      parse = MONITOR_PARSE;
    } else if (!strncmp(line, "[feeds]", 7)) {
      parse = FEEDS_PARSE;
    } else if (!strncmp(line, "[groups]", 8)) {
      parse = GROUPS_PARSE;
    } else {

      if (parse == MONITOR_PARSE) {

        if (!strncmp(line, "http://", 7)) {
          cur = parseURLorGroup( line, monitors[curMonitor].url );
          parseString( cur, monitors[curMonitor].urlName );
          monitors[curMonitor].active = 1;
          curMonitor++;
        } else return -1;

      } else if (parse == FEEDS_PARSE) {

        if (!strncmp(line, "nntp://", 7)) {
          cur = parseURLorGroup( line, feed.url );
        } else return -1;

      } else if (parse == GROUPS_PARSE) {
```

```
            cur = parseURLorGroup( line,
                    feed.groups[curGroup].groupName );

            i = 0;
            while (*cur) {
                cur = parseString(
                          cur, feed.groups[curGroup].searchString[i] );

              if (strlen(feed.groups[curGroup].searchString[i])) i++;
              if (i == MAX_SEARCH_STRINGS) break;
            }
            feed.groups[curGroup].numSearchStrings = i;
            feed.groups[curGroup].active = 1;

            curGroup++;

          }
        }

      }

      readGroupStatus();

      return 0;
    }
```

После инициализации базовой структуры агента открывается файл конфигурации. Затем последовательно читается каждая строка файла. Если строка начинается с символа «#» или символа (0×0a – пустая строка), то она игнорируется, и цикл переходит к следующей строке. В противном случае программа проверяет строку, чтобы определить, указывает ли она на новую секцию (для мониторинга, источника новостей или группы). Если да, то переменная `parse` задается равной соответствующему значению, чтобы указать, каким образом будут анализироваться следующие строки файла.

Если строка не содержит маркер секции, программа анализирует ее в соответствии с текущим статусом анализа (заданным переменной `parse`).

При анализе секции мониторинга строка сначала проверяется на предмет содержания в ней адреса для протокола HTTP. Если адрес найден, вызывается функция `parseURLorGroup` для анализа адреса в строке. Затем вызывается функция `parseString` для анализа текстового имени Web-сайта, который представлен адресом. Эти функции описаны в листинге 11.16. Наконец, следует указать, что только что заполненный элемент активен (содержит адрес Web-сервера для мониторинга) и увеличить переменную `curMonitor` для анализа следующей строки конфигурационного файла.

При анализе источника новостей программа работает со строкой, которая определяет используемый NNTP-сервер. Функция `parseURLorGroup` анализирует

адрес в этой строке и сохраняет его в поле `url` переменной `feed`. В дальнейшем агент будет подключаться к этому адресу для получения новостей.

Анализ групп новостей очень схож с анализом списка Web-сайтов для мониторинга, за исключением того, что строки поиска повторяют название группы новостей. Можно использовать до 10 строк поиска (листинг 11.14). Если в файле конфигурации строк больше, они игнорируются. Количество строк поиска сохраняется в поле `numSearchStrings`.

Листинг 11.16. Функции `parseURLorGroup` и `parseString`

```
char *parseURLorGroup( char *line, char *url )
{
  int i = 0;

  /* Пропускаем ';' или ' ' (пробел) */
  while ((*line != ' ') && (*line != ';') && (*line != 0x0a)) {
    url[i++] = *line++;
    if (i == MAX_URL_SIZE-1) break;
  }
  url[i] = 0;

  while ((*line != ';') && (*line != 0) && (*line != 0x0a)) i++;

  return( line );
}

char *parseString( char *line, char *string )
{
  int j=0;

  if (*line != ';') {
    *line = 0;
    return line;
  }
  line++;

  while (*line == ' ') line++;

  while ((*line != ';') && (*line != 0x0a)) {
    string[j++] = tolower(*line++);
    if (j == MAX_SEARCH_ITEM_SIZE-1) break;
  }
  string[j] = 0;

   while ((*line != ';') && (*line != 0)) line++;

  return( line );
}
```

Функция `parseURLorGroup` используется для анализа адреса Web-сервера или новостной группы из строки. Для нее безразлично, чем является входная строка — адресом сайта или именем группы новостей, так как функция просто ищет символ-разделитель (пробел, точку с запятой или новую строку). Все символы, которых нет в списке символов-разделителей, копируется в массив символов `url`, переданный вызывающей функцией. Обнаружив разделитель, программа пропускает все найденное пустое пространство, чтобы приготовиться к следующей функции анализа.

Функция `parseString` аналогична функции `parseURLorGroup`. Найденная строка копируется вплоть до символа-разделителя или новой строки. При копировании символы конвертируются в строчные. Поэтому строки поиска не зависят от регистра и для них проще искать соответствие. Когда функция находит символ-разделитель, новая строка удаляется, и пустое место пропускается для подготовки следующего возможного вызова функции `parseString`.

Процесс анализа (листинг 11.15) продолжается до тех пор, пока функция не достигнет конца файла (больше нет новых строк в файле конфигурации). В этот момент вызывается особая функция `readGroupStatus` (листинг 11.17), задача которой – прочитать сохраненную ранее информацию о последнем прочитанном сообщении групп новостей из файла конфигурации.

Листинг 11.17. Использование функции `readGroupStatus` для чтения состояния групп новостей

```
void readGroupStatus( void )
{
  FILE *fp;
  int  i, curMsg;
  char line[80];

  for (i = 0 ; i < MAX_MONITORS ; i++)
     feed.groups[i].lastMessageRead = -1;
  }

  fp = fopen(GRPSTS_FILE, "r");

  while (!feof(fp)) {

     fscanf( fp, "%s : %d\n", line, &curMsg );

     for (i = 0 ; i < MAX_MONITORS ; i++) {

      if (feed.groups[i].active) {

        if (!strcmp(feed.groups[i].groupName, line)) {

          feed.groups[i].lastMessageRead = curMsg;
          break;

        }
```

```
        }

      }

    }

    return;
}
```

Задача функции `readGroupStatus` заключается в том, чтобы установить номер последнего прочитанного сообщения для каждой группы новостей (если такое сообщение существует). Если группа только что была добавлена в файл конфигурации, то первое сообщение отсутствует, и агент будет читать первое доступное сообщение (полученное функцией `nntpSetGroup`). Формат файла представлен в листинге 11.18.

Листинг 11.18. Формат файла статуса новостных групп

```
comp.robotics.misc : 96000
sci.space.history : 135501
```

Формат файла статуса новостных групп (имеющий название `group.sts`) содержит в каждой строке имя группы и номер последнего прочитанного сообщения. Сначала в строке идет название группы новостей, далее символ «:», а затем следует номер последнего прочитанного сообщения группы.

При чтении статуса групп новостей сначала очищается поле `lastMessageRead` для всех групп в структуре `feed`. Затем программа проходит по всем строкам файла статуса новостных групп и анализирует их, чтобы найти название группы и номер сообщения. После анализа строки она ищет имя группы в списке групп новостного сервера, чтобы определить, что название группы существует (так как пользователь мог его удалить). Если группа найдена, поле `lastMessageRead` для нее обновляется с помощью номера сообщения (`curMsg`), прочитанного из файла. О том, как и когда создается этот файл, рассказывается далее.

Так завершается процесс конфигурации агента. Теперь рассмотрим основной процесс работы агента: сбор и фильтрацию данных.

Сбор и фильтрация новостей

Сбор новостей и их фильтрация по критериям, указанным пользователем, производит функция `checkNewsSources` (листинг 11.19). Она проходит по списку активных групп в структуре `feed` и вызывает функцию `checkGroup` для проверки каждой группы.

Листинг 11.19. Функция `checkNewsSources`

```
void checkNewsSources( void )
{
  int i;

  extern feedEntryType feed;
```

```
  for (i = 0 ; i < MAX_GROUPS ; i++) {

    if ( feed.groups[i].active ) {

      checkGroup( i );
    }

  }

  return;
}
```

Функция `checkGroup` использует функции работы с сервером новостей, описанные ранее, для сбора новостей на основании указаний пользователя (см. листинг 11.20).

Листинг 11.20. Функция `checkGroup`

```
void checkGroup( int group )
{
  int result, count, index = 0;
  char fqdn[80];
  news_t news;

  news.msg = (char *)malloc(MAX_NEWS_MSG+1);
  bzero( news.msg, MAX_NEWS_MSG+1 );
  news.msgLen = MAX_NEWS_MSG;

  prune( feed.url, fqdn );

  /* Подключаемся к серверу новостей */
  count = nntpConnect( fqdn );

  if (count == 0) {

    /* Устанавливаем нужную группу */
    count = nntpSetGroup( feed.groups[group].groupName,
                          feed.groups[group].lastMessageRead );

    index = 0;

    if (count > 100) count = 100;

    while (count-- > 0) {

      result = nntpPeek( &news, MAX_NEWS_MSG );

      if (result > 0) {

      result = nntpParse( &news, HEADER_PARSE );
```

```
            if (result == 0) {

              testNewsItem( group, &news );

            }

          }

          feed.groups[group].lastMessageRead = news.msgId;
          nntpSkip();

        }

      }

      free( news.msg );

      nntpDisconnect();

     return;
    }
```

Функция `checkGroup` сначала сокращает название NNTP-сервера с помощью функции `prune`, которая удаляет спецификацию протокола из адреса сервера (nntp://) и любые символы (/) в конце адреса, если они существуют. Далее создается буфер для новостных сообщений. Функции работы с NNTP-сервером требуют, чтобы пользователь указал буфер, который будет использоваться для сбора новостей (этот буфер должен быть достаточно велик, чтобы загрузить определенные сообщения). Здесь программа выделяет 64 Кб для буфера сообщения и устанавливает указатель `msg` структуры `news` на него. Размер буфера также помещается в поле `msgLen`, чтобы функции работы с NNTP-сервером не вышли за границу буфера при получении сообщения.

С помощью функции `nntpConnect` создается сессия на заданном NNTP-сервере. Если возвращенное значение показывает, что подключение прошло успешно, указывается текущая группа новостей. Обратите внимание, что группа здесь определяется на основании переданного индекса группы в списке. Функция возвращает количество сообщений, которые доступны для чтения. Чтобы не тратить слишком много времени на обработку всех доступных сообщений, следует ограничить их количество двумя сотнями сообщений (если их больше 200).

Затем выполняется цикл, который читает заданное количество сообщений для текущей группы. При этом используется функция `nntpPeek`, поскольку интерес представляет только информация из заголовка сообщения, а особенно соответствие между темой сообщения и критериями поиска для группы. Для анализа темы (и других полей) заголовка вызывается функция `nntpParse`. Получив уведомление об успешной загрузке заголовка сообщения и его анализа от функций

nntpPeek и nntpParse, программа передает его в функцию testNewsItem, чтобы проверить, соответствует ли сообщение критериям поиска.

После проверки сообщения программа обновляет поле lastMessageRead в соответствии с номером текущего сообщения, а затем переходит к следующему. Вспомните, что при использовании только функции nntpPeek необходимо также вызвать функцию nntpSkip, чтобы перейти к следующему доступному сообщению.

После завершения цикла программа освобождает буфер (news.msg) и отключается от NNTP-сервера с помощью функции nntpDisconnect.

Проверка новостного сообщения на соответствие текущему критерию поиска представлена в листинге 11.21.

Листинг 11.21. Функция testNewsItem

```
void testNewsItem( int group, news_t *news )
{
  int i, count=0;
  char *cur;

  if (feed.groups[group].numSearchStrings > 0) {

    for ( i = 0 ; i < feed.groups[group].numSearchStrings ; i++ ) {

      cur = strstr( news->subject,
                    feed.groups[group].searchString[i] );

      if (cur) count++;

    }

  } else {

    count = -1;

  }

  if (count) {

    insertNewsItem( group, count, news );

  }

  return;
}
```

Этот простой алгоритм позволяет определить соответствие между сообщением и критерием поиска. Если в теме сообщения была найдена хотя бы одна

строка поиска, заданная для группы, новость сохраняется для дальнейшего представления пользователю. Проверка осуществляется с помощью функции `strstr`, которая выполняет поиск подстроки. Поиск по теме для каждой строки текущей группы выполняется посредством функции `strstr`. Если возвращенное значение не равно `NULL`, это значит, что было найдено соответствие. При этом программа увеличивает на единицу переменную `count`, отображающую количество строк поиска, для которых было найдено соответствие. Данная переменная служит рейтингом сообщения (чем больше ее значение, тем выше будет сообщение в списке). Если значение `count` не равно нулю (или пользователь не задал строк поиска), то сообщение добавляется в список для дальнейшего представления пользователю.

Список новостей – это список объектов, которые служат для хранения новостей. Тип `elementType` показан в листинге 11.22. Данная структура содержит всю необходимую информацию для описания сообщения таким образом, чтобы впоследствии его можно было бы восстановить полностью и показать пользователю.

Листинг 11.22. Структура `elementType`, которая используется для хранения объектов новостей

```
typedef struct elementStruct *elemPtr;


typedef struct elementStruct {
  int  group;
  int  rank;
  int  msgId;
  char subject[MAX_LG_STRING+1];
  char msgDate[MAX_SM_STRING+1];
  char link[MAX_SM_STRING+1];
  int  shown;
  struct elementStruct *next;
} elementType;
```

Назначение многих полей этой структуры очевидно по их названиям; об остальных полях будет рассказано при обсуждении функции `insertNewsItem` (листинг 11.23).

Листинг 11.23. Функция `insertNewsItem`

```
// Define the head of the news list.
ElementType head;
void insertNewsItem( int group, int count, news_t *news )
{
  elementType *walker = &head;
  elementType *newElement;

  newElement = (elementType *)malloc(sizeof(elementType));
```

```
    newElement->group = group;
    newElement->rank = count;
    newElement->msgId = news->msgId;
    strncpy( newElement->subject, news->subject, MAX_LG_STRING );
    strncpy( newElement->msgDate, news->msgDate, MAX_SM_STRING );
    newElement->shown = 0;
    sprintf(newElement->link, "art%d_%d", group, news->msgId);
    newElement->next = (elementType *)NULL;

    while (walker) {
      /* Если нет следующего элемента, то добавляем в конец */
      if (walker->next == NULL) {
        walker->next = newElement;
        break;
      }

      /* Otherwise, insert in rank order (descending) */
      if (walker->next->rank < newElement->rank) {
        newElement->next = walker->next;
        walker->next = newElement;
        break;
      }

      walker = walker->next;

    }

    return;
}
```

При добавлении новости в список сначала необходимо создать новый объект новостей (типа `elementType`). Для этого нужно выделить блок памяти и привести указатель на нее к типу `elementType`. Затем название группы, в которой было найдено сообщение, помещается в поле `group`, и значение поля `rank` (описывающего положение сообщения относительно других объектов в списке) устанавливается в соответствии с рейтингом, определенным как количество строк поиска, которые обнаружены при анализе сообщения. Идентификатор сообщения сохраняется в поле `msgID` (уникальный идентификатор сообщения в группе), а также копируются тема `subject` и дата `msgDate`.

Поле `shown` показывает, были ли определенные объекты представлены пользователю. Это важно, поскольку пользователь имеет возможность очистить список уже просмотренных сообщений. При очистке списка удаляются только те объекты, которые пользователь уже видел (а не сообщения, которые были получены, но не показаны пользователю). При отображении сообщения флаг `shown` становится равным единице. Если рассматривается новый объект, это поле равно нулю.

Поле `link` является специальным элементом, который применяется агентом для уникальной идентификации статьи. Его значение используется при создании HTML-ссылки. Когда HTTP-сервер агента получает запрос для данной ссылки, он понимает, как следует идентифицировать сообщение, которое желает просмотреть пользователь. Например, если индекс группы равен 7, а идентификатор сообщения составляет 20999, то будет создана ссылка `art7_20999`. Благодаря этой информации агент знает, как прочитать нужную статью.

И наконец, последнее поле `next`. Программа инициализирует его значением `NULL` (конец списка), поскольку пока не может определить, куда поместить текущий объект.

Итак, в данный момент имеется новая структура `elementType` с полями, которые были инициализированы на основании аргументов, переданных при вызове функции. Теперь нужно просто добавить объект в список на позицию, определенную рейтингом. Чем больше рейтинг, тем выше будет положение сообщения в списке. В начале работы функции переменная `walker` устанавливается на верхнюю позицию списка сообщений. Заголовок списка представляет собой пустой объект без данных, который существует исключительно для управления списком (рис. 11.8).

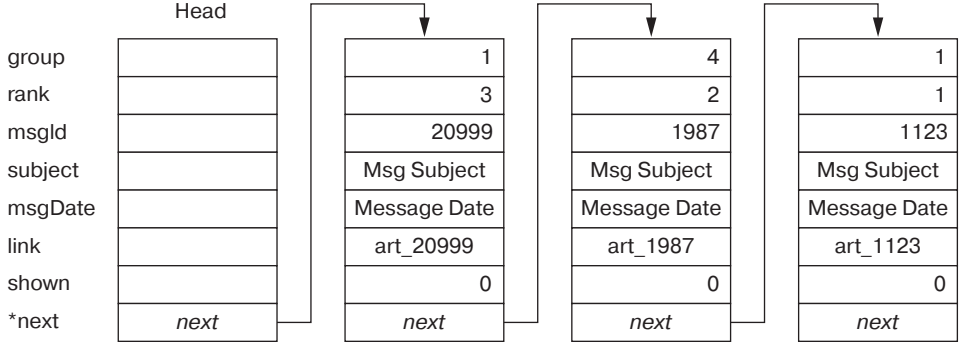


Рис. 11.8. Пример списка новостей

Из листинга 11.23 видно, что алгоритм проходит по списку объектов, обрабатывая один объект, а затем переходя к следующему. Добавить новый объект в связанный список (а именно так организован список сообщений) можно единственным способом – манипулировать указателями на следующий объект в списке.

Первый случай, который следует рассмотреть, – отсутствие следующего объекта в списке. Текущий объект добавляется в конец списка (указатель на следующий элемент списка в текущем объекте будет установлен на добавляемый элемент). Затем программа выходит из цикла и возвращается. В противном случае рейтинг следующего объекта в списке сравнивается с рейтингом добавляемого объекта. Если объект для добавления имеет более высокий рейтинг, чем следующий объект в списке, то его нужно вставить в этом месте (между текущим объектом

и следующим). Чтобы добавить объект, следует поместить в его поле `next` указатель на следующий объект в списке, а затем перевести указатель на следующий элемент текущего объекта, который требуется вставить.

Если проверка рейтинга прошла неудачно, то есть рейтинг добавляемого объекта меньше, чем у следующего в списке, программа переходит к следующему объекту (при этом он получает статус текущего) и повторяет проверку так, как было описано выше. Так создается список новостей, расположенных в порядке понижения их рейтинга, который затем будет представлен пользователю. В следующем разделе рассматривается процесс представления данных.

Пользовательский интерфейс

Пользовательский интерфейс Web-агента представляет собой HTTP-сервер, доступ к которому осуществляется с помощью браузера. В этом разделе мы поговорим о HTTP-сервере, а также о том, как он представляет данные, полученные с сервера новостей.

HTTP-сервер должен быть инициализирован с помощью функции `initHttpServer`. Она вызывается основной программой только один раз, чтобы создать HTTP-сервер и сокет, к которому могут подключаться клиенты (листинг 11.24).

Листинг 11.24. Функция `initHttpServer`

```
int initHttpServer( void )
{
  int on=1, ret;
  struct sockaddr_in servaddr;

  if (listenfd != -1) close( listenfd );

  listenfd = socket( AF_INET, SOCK_STREAM, 0 );

  /* Делаем порт доступным для повторного использования */
  ret = setsockopt( listenfd, SOL_SOCKET,
                    SO_REUSEADDR, &on, sizeof(on) );
  if (ret < 0) return -1;

  /* Устанавливаем доступность сокета с любого адреса по порту
   * 8080
   */
  bzero( (void *)&servaddr, sizeof(servaddr) );
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = htonl( INADDR_ANY );
  servaddr.sin_port = htons( 8080 );

  /* Связываем сокет со структурой servaddr */
  ret = bind( listenfd,
              (struct sockaddr *)&servaddr, sizeof(servaddr) );
  if (ret < 0) return -1;
```

```
    listen(listenfd, 1);

    return 0;
}
```

Функция `initHttpServer` создает серверный сокет, который другие функции будут проверять на предмет подключения к нему клиентов. Этот сервер работает не так, как обычный сервер, поскольку программа не будет терять время на ожидание подключения. Вместо этого она применит специальную функцию, чтобы определить, когда к серверу подключается клиент.

При создании сокета (листинг 11.24) указывается опция `SO_REUSEADDR`, которая позволяет привязаться к порту 8080. Если ее не использовать, потребуется ждать две минуты между запусками агента (время тайм-аута сокета). Затем программа соединяется с портом 8080 и позволяет принимать подключения с любых адресов (указав константу `INADDRY_ANY`). После выполнения функции `bind` вызывается функция `listen`, чтобы перевести сокет в состояние ожидания и разрешить клиентам подключение.

Так как агент используется для решения многих задач (сбора новостей, мониторинга Web-сайтов и т.д.), необходимо сделать так, чтобы при ожидании взаимодействия пользователя с HTTP-сервером он мог периодически выполнять сбор данных. Для этого потребуется проверить, есть ли какие-либо подключения клиентов. Через определенные промежутки времени программа с помощью функции `select` проверяет, нужно ли собирать данные (листинг 11.25).

Листинг 11.25. Функция `checkHttpServer`

```
void checkHttpServer( void )
{
  fd_set rfds;
  struct timeval tv;
  int ret = -1;
  int connfd;
  socklen_t clilen;
  struct sockaddr_in cliaddr;

  FD_ZERO( &rfds );
  FD_SET( listenfd, &rfds );

  tv.tv_sec = 1;
  tv.tv_usec = 0;

  ret = select( listenfd+1, &rfds, NULL, NULL, &tv );

  if (ret > 0) {

    if (FD_ISSET(listenfd, &rfds)) {
      clilen = sizeof(cliaddr);
```

```
        connfd = accept( listenfd,
                         (struct sockaddr *)&cliaddr, &clilen );

        if (connfd > 0) {
          handleConnection( connfd );
          close( connfd );
        }

      } else {

        /* При ошибке переинициализировать сервер */
        initHttpServer();

      }

    } else if (ret < 0) {

        /* При ошибке переинициализировать сервер */
        initHttpServer();

    } else {

      // Тайм-аут - ничего не делаем

    }

    return;
}
```

Вспомните (листинг 11.24), что в переменной `listenfd` хранится идентификатор сокета HTTP-сервера. При вызове функции `select` он сообщает о том, что произошло подключение клиента. Вызов этой функции предоставляет возможность работать с временными интервалами, поэтому если в течение определенного времени не происходит подключения клиента, то функция разблокирует вызов и возвращает управление вызывающей функции. Таким образом Web-агент выполняет периодический сбор данных. После завершения сбора данных агент возвращается к функции `checkHttpServer`, чтобы проверить, подключился ли клиент для получения данных. Подробное описание вызова функции `select` выходит за рамки данной книги, но в разделе «Литература и ресурсы» вы сможете найти несколько полезных ссылок.

Функция `select` использует структуру `rfds`, чтобы создать карту сокетов, для которых следует выполнять мониторинг. В данном случае работа производится только с одним сокетом, `listenfd`, который указывается в структуре `rfds` с помощью макроса `FD_SET`. Также инициализируется структура `timeval` с временным интервалом, равным одной секунде. Затем выполняется вызов функции `select`. При этом указывается, что ожидается процедура чтения (для которой необходимо подключение клиента), а также определяется время ожидания.

Вызывающая функция будет уведомлена о том, какое из представленных событий произошло.

После выполнения функции `select` программа проверяет возвращенное значение. Если оно меньше нуля, значит, произошла какая-то ошибка и необходимо повторно инициализировать HTTP-сервер, вызвав функцию `initHttpServer`. Если значение равно нулю, следовательно, время ожидания закончилось, но пользователь не присоединился (это определяется значением временного интервала). В этом случае программа не предпринимает никаких действий и просто выходит из функции. Наконец, если значение больше нуля, то произошло подключение пользователя к серверу. Макрос `FD_ISSET` позволяет определить, к какому сокету подключился клиент (это должен быть сокет `listenfd`, поскольку другие сокеты не конфигурировались). Если функция `FD_ISSET` подтверждает, что клиент ждет подключения к сокету `listenfd`, программа выполняет подключение с помощью команды `accept` и вызывает функцию `handleConnection` для обслуживания запроса. В противном случае программа определяет, что произошла внутренняя ошибка и выполняет повторную инициализацию с помощью функции `initHttpServer`.

Функция `handleConnection` обрабатывает один запрос для HTTP-сервера. Сначала программа анализирует HTTP-запрос, чтобы определить, о чем просит клиент. В зависимости от результатов вызывается функция, которая сгенерирует ответ HTTP-сервера (листинг 11.26).

Листинг 11.26. Функция `handleConnection`

```
 void handleConnection( int connfd )
{
  int len, max, loop;
  char buffer[MAX_BUFFER+1];
  char filename[80+1];

  /* Стираем HTTP-запрос */
  max = 0; loop = 1;
  while (loop) {
    len = read(connfd, &buffer[max], 255); buffer[max+len] = 0;
    if (len <= 0) return;
    max += len;
    if ((buffer[max-4] == 0x0d) && (buffer[max-3] == 0x0a) &&
        (buffer[max-2] == 0x0d) && (buffer[max-1] == 0x0a))
            loop = 0;
  }
  /* Определяем тип HTTP-запроса */
  if (!strncmp(buffer, "GET", 3)) {

    getFilename(buffer, filename, 4);

    /* В нашем сервере имя файла, полученное из запроса,
     * определяет функцию, которую надо вызвать для генерации
```

```
        * ответа
        */
       if (!strncmp(filename, "/index.html", 11))
            emitNews( connfd );
       else if (!strncmp(filename, "/config.html", 12))
            emitConfig( connfd );
       else if (!strncmp(filename, "/art", 3))
            emitArticle( connfd, filename );
       else
            write(connfd, notfound, strlen(notfound));

     } else if (!strncmp(buffer, "POST", 4)) {

       getFilename(buffer, filename, 5);

       /* Для запроса типа POST переданное имя файла указывает, что
        * надо сделать. В нашем сервере доступно только действие
        * "Пометить как прочитанное", которое очищает список для
        * показа
        */
       if (!strncmp(filename, "/clear", 6)) {
         clearEntries();
         emitHTTPResponseHeader( connfd );
         strcpy(buffer, "<P><H1>Click Back and Reload to "
                        "refresh page.</H1><P>\n\n");
         write(connfd, buffer, strlen(buffer));

       } else {
         write(connfd, notfound, strlen(notfound));
       }

     } else {

       strcpy(buffer, "HTTP/1.1 501 Not Implemented\n\n");
       write(connfd, buffer, strlen(buffer));

     }

     return;
   }
```

При работе с новым HTTP-соединением прежде всего нужно прочесть HTTP-запрос. Начальный цикл читает запрос и ищет пустую строку. Пустая строка, которая следует за запросом, – это способ уведомить о конце запроса в протоколе HTTP. После того как запрос целиком был помещен в буфер, программа проверяет, какая информация была запрошена пользователем. Запрос `GET` позволяет получить файл с HTTP-сервера; в данном случае все файлы генерируются динамически на основании имени файла, который был запрошен. Запрос `POST` указывает, что пользователь щелкнул по кнопке на странице, а далее

следует имя файла (обычно это имя файла CGI-скрипта: интерфейс CGI обеспечивает для HTTP-сервера скрипты, которые позволяют серверу выполнять различные действия). Программа читает имя файла, вызванного пользователем с помощью запроса `POST`, и использует его, чтобы определить, какое действие следует выполнить.

При работе с запросами `GET` сервер знает, как обслуживать три имени файла. По умолчанию используется файл `/index.html`, который представляет главную страницу поиска новостей и мониторинга Web-сайтов (рис. 11.5). Когда браузер клиента запрашивает данный файл, для создания страницы и выдачи ее пользователю вызывается функция `emitNews`. Файл `/config.html` отображает страницу конфигурации агента (то есть текущую конфигурацию) – см. рис. 11.7. Если название файла начинается с `/art`, это значит, что клиент запросил определенную новость (рис. 11.6). При изучении функции `emitNews` вы увидите, как статьи связаны с ссылками на странице новостей. Для выполнения данного запроса используется функция `emitArticle`. Наконец, если HTTP-сервер не распознает названия запрошенного файла, создается HTTP-сообщение об ошибке (код 404, файл не найден).

При работе с запросами `POST` название файла извлекается из HTTP-запроса для анализа. Если имя запрошенного файла представляет собой `/clear`, это значит, что пользователь потребовал удалить уже просмотренные новости (это делается щелчком по кнопке на странице новостей). Прочитанные сообщения удаляются с помощью функции `clearEntries`, а затем функция `emitHTTPResponseHeader` выводит заголовок HTTP-сообщения. Наконец, программа пишет пользователю короткое сообщение (которое отображается в окне браузера) с просьбой нажать кнопку **Назад** (Back) и обновить список текущих новостей. Если имя запрошенного файла отличается от рассмотренного, это значит, что произошла ошибка. Генерируется сообщение об ошибке.

Последний случай выполнения функции `handleConnection` связан с получением неизвестного запроса. Если HTTP-запрос не относится к типу `GET` или `POST`, выдается сообщение об ошибке, которое информирует пользователя о том, что его запрос не будет реализован.

Рассмотрим несколько функций, которые используются функцией `handleConnection`. Сначала для получения имени файла из HTTP-запроса применяется функция `getFilename` (листинг 11.27). Она вычленяет имя файла, пропуская тип HTTP-запроса и копируя все символы до ближайшего символа пробела. В том случае, если имя запрошенного файла состоит из одного символа «/», функция `getFileName` возвращает страницу показа новостей `/index.html`.

Листинг 11.27. Вспомогательная функция `getFileName`

```
void getFilename(char *inbuf, char *out, int start)
{
  int i=start, j=0;

  /* Пропустить пробелы в начале строки */
  while (inbuf[i] == ' ') i++;
  for ( ; i < strlen(inbuf) ; i++) {
```

```
    if (inbuf[i] == ' ') {
      out[j] = 0;
      break;
    }
    out[j++] = inbuf[i];
  }

  if (!strcmp(out, '/')) strcpy(out, "/index.html");

  return;
}
```

Следующая функция, `emitHTTPResponseHeader`, используется для генерации простого заголовка для HTTP-ответа (листинг 11.28). В этом ответе, направленном браузеру клиента, сообщается о том, что запрос был понят и ответ будет в HTML-формате (с помощью элемента `Content-type`).

Листинг 11.28. Вспомогательная функция `emitHTTPResponseHeader`

```
void emitHTTPResponseHeader( int connfd )
{
  char line[80];

  strcpy( line, "HTTP/1.1 200 OK\n" );
  write( connfd, line, strlen(line) );

  strcpy( line, "Server: tinyHttp\n" );
  write( connfd, line, strlen(line) );

  strcpy( line, "Connection: close\n" );
  write( connfd, line, strlen(line) );

  strcpy( line, "Content-Type: text/html\n\n" );
  write( connfd, line, strlen(line) );

  return;
}
```

При вызове этой функции программа передает дескриптор сокета текущего HTTP-соединения. Дескриптор использует функция `emitHTTPResponseHeader`, чтобы отправить ответ назад.

Следующая функция, вызываемая из функции `handleConnection`, – `clearEntries` (листинг 11.29). Она позволяет удалить все новости, которые клиент уже просмотрел. При сборе новостей флаг, показывающий, было ли текущее сообщение представлено пользователю, инициализируется нулем. После того как пользователю была показана страница, содержащая данную новость, флажок устанавливается в единицу. Когда пользователь щелкает по кнопке **Пометить как прочитанное** (Mark Read) на странице, эта функция вызывается для удаления прочитанных сообщений.

Листинг 11.29. Функция `clearEntries`

```
void clearEntries( void )
{
  elementType *walker = &head;
  elementType *temp;
  int i;

  extern monitorEntryType monitors[];

  /* Очистка цепочки новостей (для элементов, которые были
   * прочитаны пользователем)
   */
  while (walker->next) {

    if (walker->next->shown) {
      temp = walker->next;
      walker->next = walker->next->next;
      free(temp);
    } else {
      walker = walker->next;
    }

  }

  /* Очистка списка сайтов, которые были просмотрены
   * пользователем
   */
  for (i = 0 ; i < MAX_MONITORS ; i++) {

    if ((monitors[i].active) && (monitors[i].shown)) {
      monitors[i].changed = 0;
      monitors[i].shown = 0;
    }

  }

  emitGroupStatus();
}
```

Функция `clearEntries` действует аналогично функции `insertNewsItem` (показанной в листинге 11.23). Она проходит по списку новостей в поисках сообщения, для которого отмечено поле `shown`. Обратите внимание на то, что программа проверяет следующее сообщение, начиная с верхней позиции списка, то есть с пустого объекта. Это единственный способ удалить ненужные объекты. Указатель `next` текущего объекта требуется поместить на указатель `next` следующего объекта, что позволяет эффективно удалить объект из цепи. Объект сохраняется в структуре `temp` типа `ElementType`, а память, занимаемая им после обновления списка, освобождается.

Функция `clearEntries` также сбрасывает флажок `shown` для Web-сайтов, мониторинг которых выполняется агентом. Если Web-сайт был помечен как измененный и представленный пользователю, флажки сбрасываются. Такой сайт не будет отображаться при следующем запросе страницы новостей, если он опять не изменился. Наконец, функция `clearEntries` вызывает функцию `emitGroupStatus`, чтобы записать файл статуса для групп новостей. В нем хранится последнее прочитанное сообщение для всех текущих групп новостей (листинг 11.30).

Листинг 11.30. Функция `emitGroupStatus`

```
void emitGroupStatus( void )
{
  FILE *fp;
  int  i;

  fp = fopen(GRPSTS_FILE, "w");

  for (i = 0 ; i < MAX_MONITORS ; i++) {

    if (feed.groups[i].active) {

      fprintf( fp, "%s : %d\n",
                   feed.groups[i].groupName,
                   feed.groups[i].lastMessageRead );
    }

  }

  fclose(fp);

  return;
}
```

Функция `emitGroupStatus` выводит название группы и номер последнего прочитанного сообщения для всех активных групп в структуре `feed`. Если после ее выполнения работа агента была остановлена, она может быть возобновлена без пропуска новых или повтора старых сообщений.

Продолжим изучение реализации функций пользовательского интерфейса, введя три функции, которые генерируют данные для обслуживания через HTTP-сервер. Вспомните, что после идентификации HTTP-запроса `GET` программа анализирует название файла, чтобы передать запрос функции, способной его выполнить (листинг 11.26).

Функция `emitConfig` показывает текущую конфигурацию агента пользователю (листинг 11.31). С помощью Web-страницы конфигурацию изменять нельзя. Для этого следует отредактировать файл конфигурации и перезапустить агента.

Листинг 11.31. Функция `emitConfig`

```
const char *prologue={
  "<HTML><HEAD><TITLE>WebAgent</TITLE></HEAD>"
  "<BODY TEXT=\"#000000\" bgcolor=\"#FFFFFF\" link=\"#0000EE\""
  "vlink=\"#551A8B\" alink=\"#FF0000\">"
  "<BR><font face=\"Bauhaus Md BT\"><font color=\"#000000\">"
};

const char *epilogue={
  "</BODY></HTML>\n"
};

void emitConfig( int connfd )
{
  char line[MAX_LINE+1];
  int i, j;

  extern monitorEntryType monitors[];
  extern feedEntryType feed;

  emitHTTPResponseHeader( connfd );

  write( connfd, prologue, strlen(prologue));

  strcpy(line, "<H1>Configuration</H1></font></font><BR><BR>");
  write( connfd, line, strlen(line));

  strcpy(line, "<font size=+2>Sites to Monitor</font><BR><BR>");
  write( connfd, line, strlen(line));

  strcpy(line, "<center><table BORDER=3 WIDTH=100% NOSAVE><tr>\n");
  write( connfd, line, strlen(line));

   for (i = 0 ; i < MAX_MONITORS ; i++) {

     if (monitors[i].active) {

       sprintf(line, "<tr><td><font size=+1>%s</font></td><td>"
                     "<font size=+1>%s<font></td></tr>\n",
               monitors[i].urlName, monitors[i].url);
       write( connfd, line, strlen(line));

     }

  }

  strcpy(line, "</tr></table></center><BR><BR>\n");
  write( connfd, line, strlen(line));
```

```
    sprintf(line,
      "<H2>Feed %s</H2><BR><BR>\n", feed.url);
    write( connfd, line, strlen(line));

    strcpy(line, "<center><table BORDER=3 WIDTH=100% NOSAVE><tr>\n");
    write( connfd, line, strlen(line));

    for (i = 0 ; i < MAX_GROUPS ; i++) {

      if (feed.groups[i].active) {

        sprintf(line, "<tr><td><font size=+1>Group %s</font></td>\n",
                 feed.groups[i].groupName);
        write( connfd, line, strlen(line));

        strcpy(line, "\n<td><font size=+1>");
        if (feed.groups[i].numSearchStrings > 0) {
          for (j = 0 ; j < feed.groups[i].numSearchStrings ; j++) {
            if (j > 0) strcat(line, ", ");
            strcat(line, feed.groups[i].searchString[j]);
          }
        } else {
          strcat(line, "[*]");
        }
        strcat(line, "</font></td></tr>\n");
        write( connfd, line, strlen(line) );

      }

    }

    strcpy(line, "</tr></table></center><BR><BR>\n");
    write( connfd, line, strlen(line));

    write( connfd, epilogue, strlen(epilogue));

    return;
  }
```

Прежде всего нужно обратить внимание на две строковые константы, которые содержат информацию о заголовке и конце HTML-сообщения. Строка `prologue` задает цветовую гамму, размер шрифта и заголовок страницы. Функция `epilogue` используется для завершения HTML-страницы.

При обслуживании HTML-страницы через сервер сначала нужно вывести заголовок HTML-ответа (с помощью функции `emitHTTPResponseHeader`). Далее следует вывод строки `prologue`. Операции вывода используют дескриптор

сокета connfd, который передан в функцию. Все, что отправляется через сокет, будет получено и интерпретировано клиентом.

Перед выводом информации по мониторингу Web-сайта с помощью тэга <table> создается таблица. Затем все элементы таблицы помещаются в строки с помощью тэга <tr>. Программа проходит по массиву monitors и ищет активные объекты, которые затем выводит с помощью соответствующих тэгов. Для этого создается строка таблицы, разбитая на два столбца. В первый столбец помещается название сайта, а во второй – его адрес. После того как программа обработает все элементы массива, таблица закрывается с помощью тэга </table>.

Далее адрес сервера новостей выводится в виде одной строки. С помощью функции sprintf создается буфер, в который помещается нужная строка. Затем она выводится в сокет с помощью стандартной функции write.

Вывод информации о группах новостей схож с методом, который использовался для вывода массива monitors. Создается новая таблица, и генерируются все ее строки. Каждая строка разбивается на два столбца, один для названия группы, а другой для строк, являющихся критериями поиска сообщений в данной группе. Название группы представляется в виде одного объекта, но строки поиска являются независимыми элементами. Поэтому выполняется цикл, который позволяет собрать в одну большую строку данные о критериях поиска, причем строка будет включать символ разделения «,». Если строки поиска отсутствуют, выводится символ [*]. Обратите внимание, что при этом пользователю показываются все сообщения. После завершения цикла таблица закрывается с помощью тэга </table>.

Наконец, в сокет выводится строка epilogue, которая заканчивает HTML-страницу. Эта функция сообщает браузеру о том, что он может представить страницу пользователю.

Следующая функция, emitNews (листинг 11.32), почти аналогична функции emitConfig. Будут проиллюстрированы только различия (все общие элементы уже обсуждались в разделе, посвященном функции emitConfig).

Листинг 11.32. Функция emitNews

```
void emitNews( int connfd )
{
  int  i;
  char line[MAX_LINE+1];
  elementType *walker;

  extern monitorEntryType monitors[];
  extern feedEntryType feed;
  extern elementType head;

  emitHTTPResponseHeader( connfd );

  write( connfd, prologue, strlen(prologue));
```

```
strcpy(line,
        "<H1>Web Agent Results</H1></font></font><BR><BR>");

write( connfd, line, strlen(line));

strcpy(line, "<center><table BORDER=3 WIDTH=100% NOSAVE><tr>\n");
write( connfd, line, strlen(line));

for (i = 0 ; i < MAX_MONITORS ; i++) {

  if ((monitors[i].active) && (monitors[i].changed)) {

    sprintf(line, "<tr><td><font size=+1>%s</font></td>\n"
                  "<td><font size=+1><a href=\"%s\">%s</a>"
                  "</font></td></tr>\n",
          monitors[i].urlName, monitors[i].url,
          monitors[i].url);

    write( connfd, line, strlen(line));

    monitors[i].shown = 1;

  }

}

strcpy(line, "</tr></table></center><BR><BR>\n");
write( connfd, line, strlen(line));

walker = head.next;

if (walker) {

  strcpy(line, "<center><table BORDER=3 WIDTH=100% NOSAVE><tr>\n");
  write( connfd, line, strlen(line));

  while (walker) {

    sprintf(line, "<tr><td><font size=+1>%s</font></td>\n"
                  "<td><font size=+1><a href=\"%s\">"
                    %s</a></font></td>"
                  "<td><font size=+1>%s</font></td></tr>",
                  feed.groups[walker->group].groupName,
                  walker->link,
                  walker->subject,
                  walker->msgDate );
```

```
        write( connfd, line, strlen(line));
        walker->shown = 1;
        walker = walker->next;

      }

      strcpy(line, "</tr></table></center>\n");
      write( connfd, line, strlen(line));
    }

    strcpy(line, "<FORM METHOD=\"POST\" ACTION=/clear\">");
    write( connfd, line, strlen(line));

    strcpy(line, "<BR><BR><INPUT TYPE=\"submit\" "
                 "VALUE=\"Mark Read\"><BR>\n");
    write( connfd, line, strlen(line));

    write( connfd, epilogue, strlen(epilogue));

    return;
  }
```

Прежде всего, следует обратить внимание на раздел, который выводит массив `monitors` (цикл `monitors`). Выводятся только те строки, которые активны (флаг `active`), а также изменились с последнего просмотра пользователем (флаг `changed`). После отображения объекта устанавливается флажок в поле `shown`. Это означает, что объект можно удалять при условии, что пользователь пожелает очистить список просмотренных новостей (листинг 11.29). Аналогичным образом флажок в поле `shown` устанавливается для новостей, которые были представлены пользователю.

Наконец, в листинге 11.32 вам следует обратить внимание на применение ссылки, которая позволяет клиенту просмотреть новостные сообщения. Для ее создания используется HTML-тэг <a href>. Поле `link` структуры `elementType` также необходимо для создания ссылки (вспомните описание полей структуры `elementType`, представленной в листинге 11.23).

Последняя функция, `emitArticle`, немного более сложна, поскольку она должна связываться с NNTP-сервером, чтобы получить тело сообщения. Вспомните, что изначально считывается лишь заголовок новости. Это делается с целью экономии времени и места. Когда пользователь запрашивает для просмотра всю статью, для ее считывания создается соединение с NNTP-сервером (листинг 11.33).

Листинг 11.33. Функция `emitArticle`

```
 void emitArticle( int connfd, char *filename )
{
  int group, article, count, result;
  news_t news;
```

```
  char line[MAX_LINE+1];
  extern feedEntryType feed;

sscanf(filename, "/art%d_%d", &group, &article);

news.msg = (char *)maloc(MAX_NEWS_MSG+1);
bzero(news.msg, MAX_NEWS_MSG+1);
news.msgLen = MAX_NEWS_MSG;

  emitHTTPResponseHeader( connfd );

  write( connfd, prologue, strlen(prologue));

  prune( feed.url, line );

  count = nntpConnect( line );

  if (count == 0) {

    count = nntpSetGroup( feed.groups[group].groupName,
                          article-1 );

    if (count > 0) {

      result = nntpRetrieve( &news, MAX_NEWS_MSG );

      if (result > 0) {

        result = nntpParse( &news, FULL_PARSE );

        if (result == 0) {

         /* Вывод страницы */
         sprintf( line,
         "<font size=+1>Subject  : %s\n</font><BR><BR>",
                   news.subject );

          write( connfd, line, strlen(line) );
          sprintf( line,
                  "<font size=+1>Sender: %s\n</font><BR><BR>",
                   news.sender );
          write( connfd, line, strlen(line) );

          sprintf( line,
               "<font size=+1>Group : %s\n</font><BR><BR>",
                    feed.groups[group].groupName );
          write( connfd, line, strlen(line) );

          sprintf( line, "<font size=+1>Msg Date : %s\n</font>"
                         "<BR><BR><hr><PRE>", news.msgDate );
```

```
            write( connfd, line, strlen(line) );
            write( connfd, news.bodyStart, strlen(news.bodyStart) );
            sprintf(line, "</PRE><BR><BR>End of Message\n<BR><BR>");
            write( connfd, line, strlen(line) );

          } else {

            /* Вывод сообщений об ошибке */
            printf("Parse error\n");

          }

        }

      }

    write( connfd, epilogue, strlen(epilogue));

    free( news.msg );

    nntpDisconnect();

    return;
  }
```

Чтобы отобразить статью, сначала следует идентифицировать, какую именно статью запросил пользователь. Эта информация указана в имени файла (полученного из HTTP-запроса функцией `handleConnection`). Программа анализирует имя файла, чтобы получить индекс новостной группы и номер статьи, который является числовым идентификатором статьи на NNTP-сервере.

Далее создается буфер (переменная типа `news_t`), который предназначен для получения всего сообщения. Адрес источника сообщения анализируется и используется для соединения с NNTP-сервером с помощью функции `nntpConnect`. При успешном соединении нужно указать группу новостей, которая была получена из имени файла `/art` (`feed.groups[group]`). Обратите внимание, что последнее прочитанное сообщение задается как `article-1`. Это значит, что после завершения работы функции `nntpSetGroup` первым будет считано именно то сообщение, которое интересует пользователя. Затем программа вызывает функцию `nntpRetrieve` и анализирует результаты с помощью функции `nntpParse`. Отличие от ранее рассмотренной функции, обрабатывающей новостные сообщения, состоит в том, что константа `FULL_PARSE` передается в функцию `nntpParse`, чтобы выделить тело сообщения.

Осталось представить полученную информацию пользователю через сокет `connfd`. Большинство данных, которые будут выведены, ранее уже рассматривались. Новый объект здесь – это `bodyStart`, который указывает на начало тела

сообщения. Функция `emitArticle` завершает работу, передав значение `epilogue`, освободив выделенную для сообщения память и отключившись от NNTP-сервера с помощью функции `nntpDisconnect`.

Основная функция

Теперь сведем всю информацию по Web-агенту воедино с помощью функции `main`, которая обеспечивает основной цикл работы агента (листинг 11.34).

Листинг 11.34. Функция `main()` Web-агента

```
int main()
{
  int timer=0, ret, i;

  extern monitorEntryType monitors[];

  /* Разбор конфигурационного файла */
  ret = parseConfigFile( "config" );

  if (ret != 0) {
    printf("Error reading configuration file\n");
    exit(0);
  }

  /* Инициализация HTTP-сервера */
  initHttpServer();

  while (1) {

    /* Проверка новостей и мониторинг сайтов каждые 10 минут */
    if ((timer % 600) == 0) {
      printf("Checking News...\n");

      /* Проверка новостей */
      checkNewsSources();
      printf("Monitoring...\n");

      /* Проверяем, не изменился ли один из проверяемых сайтов */
      for (i = 0 ; i < MAX_MONITORS ; i++) {
        if (monitors[i].active) monitorSite( i );
      }

    }

    /* Проверяем, не пришел ли какой-нибудь запрос от пользователя */
    checkHttpServer();

    timer++;

  }
}
```

Сначала выполняется инициализация агента: программа читает и анализирует файл конфигурации с помощью функции `parseConfigFile`. Затем функция `initHttpServer` запускает HTTP-сервер. Далее программа может войти в бесконечный цикл, в котором вызываются две базовые функции. Первая функция – это сбор данных, а вторая – проверка подключений клиентов к HTTP-серверу.

Сбор данных выполняется каждые 10 мин. Функция `checkNewsSources` проверяет доступные новости, которые соответствуют критериям поиска, заданным пользователем. Функция `monitorSite` выполняет проверку Web-сайтов, чтобы найти изменения. Она вызывается в цикле, проходящем по всем элементам массива `monitors`.

В конце цикла функция `checkHttpServer` проверяет, есть ли входящие подключения клиентов к HTTP-серверу. При этом она блокируется на одну секунду в ожидании подключения. Если за указанное время подключения не происходит, функция возвращается, и программа выполняет проверку на наличие запросов. Если за это время выполняется подключение и программа получает запрос, она блокирует соединение, чтобы использовать его при следующем вызове функции `checkHttpServer`.

Другие области применения

Технология агентов применяется в самых разнообразных областях. Использование агентов предполагает другие пути развития программного обеспечения вне рамок создания «разумных» программ. Очень интересный материал по способам применения агентов предлагается вашему вниманию на Web-страницах UMBC Agent и BOTSpot (см. раздел «Литература и ресурсы»). В табл. 11.2 приведены несколько примеров использования агентов в настоящее время.

Таблица 11.2. Текущие сферы использования агентов

Тип агента	Описание
Агенты поиска	Выполняют поиск и фильтрацию информации в сети Internet, группах новостей, базах данных и т.п.
Агенты распределения	Распределяют ресурсы на основании заданных динамических ограничений
Агенты планирования	Создают план с учетом ресурсов, шкалы времени и ограничений
Агенты аукциона	Эффективно обмениваются ресурсами
Персональные агенты	Выполняют роль посредников для пользователей

Итоги

В этой главе рассматривалась технология так называемых умных агентов, представляющих интересный материал для изучения в рамках ИИ, поскольку их использование позволяет придать программам определенную долю разумности. После описания базовых свойств рассказывалось о классификации умных агентов. Затем были представлены многочисленные методы, которые позволяют сделать агентов разумными (подобные методы уже обсуждались в других главах

данной книги). Наконец, была подробно разобрана реализация простого агента-фильтра и рассмотрены различные сетевые интерфейсы, которые использовались при его создании.

Литература и ресурсы

1. Ananova Ltd. Агент Ананова (Ananova Agent). Доступно по адресу http://www.ananova.com/video.
2. BotSpot, http://www.botspot.com.
3. SourceForge. Набор для разработки программного обеспечения Aglet (Aglet Software Development Kit). Доступно по адресу http://sourceforge.net/projects/aglets.
4. Брэдшоу Д. Программные агенты (Bradshaw J. Software Agents. AAAI Press / MIT Press, 1997).
5. Кинг Д. Умные агенты: как оживить хорошие вещи (King J. Intelligent Agents: Bringing Good Things to Life // AI Expert: 17–19, February, 1995). Доступно по адресу http://coqui.lce.org/cedu5100/Intelligent_Agents.htm.
6. Кэй А. Компьютерное программное обеспечение (Kay A. Computer Software // Scientific American 251(3): 53–59, September, 1994).
7. Университет Балтимор Каунт, Мэриленд. Агент в сети Internet (University of Maryland Baltimore Count. Agent Web). Доступно по адресу http://agents.umbc.edu/.
8. Франклин С. и Граессер А. Это агент или просто программа? Иерархия автономных агентов (Franklin S. and Graesser A. Is It an Agent or Just a Program? A Taxonomy for Autonomous Agents // Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages. – New York: Springer Verlag, 1996).

Глава 12. Искусственный интеллект сегодня

В этой заключительной главе мы поговорим о том, как в настоящее время продвигается изучение ИИ. Искусственный интеллект является классическим примером технологии, которая изначально казалась простой, но при более внимательном исследовании выяснилось, насколько она сложна. Ранние предсказания дальнейшей судьбы ИИ оказались ошибочными, что делает любые прогнозы будущего ИИ, как минимум, недостоверными. В данной главе описываются интересные разработки в области ИИ, которые ведутся в настоящее время.

Сверху вниз и снизу вверх

Методы получения искусственного разума могут быть разделены на две категории: ведение исследования сверху вниз и снизу вверх. Категория «сверху вниз» является синонимом традиционного подхода к ИИ, когда во главу угла ставилась задача создания ИИ и мало внимания придавалось деталям, позволяющим добиться этой цели. Категория «снизу вверх» схожа с моделью нейронной сети: она почти полностью повторяет структуру человеческого мозга. С данной точки зрения познавательная способность разума зависит от работы огромного количества простых элементов. В этом подходе также используются эволюционные алгоритмы и искусственная жизнь.

Представим человеческий мозг. Нам еще предстоит понять, какие структуры мозга отвечают за то, что мы называем разумом или сознанием. Процесс работы миллионов нейронов каким-то образом создает разум на глобальном уровне. Простой процесс действия нейрона на микроуровне способствует формированию гораздо более сложного процесса на макроуровне.

ИИ начинал свое развитие на уровне «сверху вниз», причем разработки в области его связей были минимальны. После того как Марвин Мински и Сеймур Паперт опубликовали книгу «Перцептроны», исследования в области нейронных сетей были почти полностью прекращены. Однако разработчики быстро поняли, что проблемы, описанные в данной книге, легко поддаются решению. Как считают сегодня, методика «снизу вверх» связана с будущим ИИ. Главный вопрос в области ИИ формулируется так: можем ли мы создать ИИ, который будет копировать человеческий разум, или мы опишем наши задачи и позволим ИИ на

основе их решения обрести разум. Результаты изучения в этих сферах показывают, что нам следует руководствоваться методом «снизу вверх».

Построение искусственной жизни

Алан Тьюринг первым предложил идею «Машины-ребенка», принцип которой состоит в том, что разумная машина не станет разумной в одно мгновение, а будет постепенно учиться, как это делают дети. Стремление к обучению будет запрограммировано, но знания машины будут улучшаться с течением времени.

Другие исследователи предположили, что верный подход к проблеме – это изучение и построение искусственных животных. Сможем ли мы, к примеру, создать искусственное насекомое, которое сможет повторять поведение настоящего насекомого и учиться так же, как оно? Эта задача, разумеется, намного проще, чем создание разума, подобного человеческому, но, очевидно, ее решение поможет нам при построении искусственного разума.

Свое логическое развитие идея создания искусственных насекомых, животных и людей нашла при построении роботов. Эта задача требует инноваций в сфере развития и проектирования микродатчиков и приводов, а также программных структур, способствующих неограниченному обучению системы.

Разумные рассуждения и проект CYC

Марвин Мински определил одну из основных проблем, встающих перед экспертными системами в настоящее время: обладая огромными знаниями в определенной области, такие системы не имеют обычного здравого смысла. Рассмотрим данное утверждение на примере экспертной системы и программы, которая играет в шахматы. И та, и другая достаточно разумны в своей узкой области существования, но при этом экспертная система ничего не знает о шахматах, а программа для шахмат умеет только играть в шахматы. Иными словами, эти разумные программы совершенно бесполезны в других областях. Может быть, если придать данным программам разумность, они смогут общаться друг с другом или даже сотрудничать?

Одним из самых известных проектов в области разумности в настоящее время является проект CYC компании Cycorp. Он ведется Дугом Ленатом (Doug Lenat); это имя еще встретится вам в разделе, посвященном научным открытиям. Первой задачей CYC было создать базу знаний, которая включала бы понятие здравого смысла. Помимо простых фактов, база знаний должна была содержать управляющие ими правила. В CYC входят «микротеории», которые связывают правила для определенной области знаний, чтобы поддерживать и оптимизировать процесс умозаключения. Движок программы позволяет принимать решения о фактах в базе знаний.

Недавно компания Cycorp выпустила проект OpenCyc, который является открытой версией технологии CYC. Он включает базу знаний (6000 концепций и 60000 правил), движок принятия решений CYC, а также ряд языковых правил и механизмов API, которые позволяют поддерживать развитие программного обеспечения с помощью базы знаний.

Автономное программирование

На четвертом Семинаре по искусственной жизни в 1994 г. Джеффри О. Кепхарт (Jeffrey O. Kephart) из компании IBM представил работу «Биологическая иммунная система для компьютеров» (A Biologically Inspired Immune System for Computers). В этой работе Джеффри описал антивирусную архитектуру, которая моделирует иммунную систему для компьютеров и компьютерных сетей. Другие идеи с биологической мотивацией в этой работе подразумевали использование принципа самовоспроизведения для борьбы с вирусами.

С тех пор IBM существенно продвинулась вперед, используя агрессивную стратегию построения самоуправляющих компьютерных систем, основанных на принципе автономии. Ниже приводится список свойств, определяющих самоуправляющие системы:

- автоматическая конфигурация, адаптация к динамическим изменениям в окружающей среде;
- самоизлечение, диагностика и предотвращение проблем;
- самостоятельная оптимизация, автоматическая настройка и балансировка;
- самозащита, обнаружение угрозы и защита от нее.

Задачи этого исследования многочисленны. Они включают упрощение систем в будущем, снижение потребности в специалистах в области IT, экономию средств, а также обеспечение сотрудничества систем и людей при решении сложных проблем.

Хотя пока не ясно, как построить и использовать автономное программирование, сама идея звучит интересно и может способствовать развитию надежных программных систем.

ИИ и научные открытия

Представьте себе, что вы используете компьютер для создания человека или открытия новых интересных теорий в разнообразных областях. Данная сфера изучения не является новой, но продолжает вызывать интерес, поскольку включает как понимание творческого процесса, происходящего в человеческом мозгу, так и построение компьютеров, способных к формированию новых теорий и идей.

Новые программы были разработаны для того, чтобы создавать новые концепции в элементарной математике, теоретизировании и теории графов. Программа Дуга Лената AM и построенная по тому же принципу программа Eurisko повторно открыли фундаментальные математические аксиомы, а также разработали некоторые новые концепции.

Методы научных открытий с применением искусственного разума использовались и в других научных областях, включая химию, физику частиц и орбитальную механику. Система BACON.3, разработанная П. Лэнгли (P. Langley), повторно открыла третий закон Кеплера (квадраты периодов планет пропорциональны кубам их полуосей). Хотя повторное открытие и не создает нового знания, сам его

факт показывает, что компьютеры могут создавать сложные теории, что дает надежду на последующие открытия.

Программирование эмоций

В настоящее время все больше исследователей верит, что эмоции являются важной составляющей частью разума. Более того, многие верят, что истинный разум невозможно создать без учета эмоций в модели. Группа эмоционального программирования в MIT, которой руководит Розалинд Пикард (Rosalind Picard), находится в лидерах среди разработчиков в этой сфере. Группа фокусируется на широком спектре областей изучения, от эффекта воздействия компьютеров на людей до синтеза эмоций в компьютерах.

Понимание эмоций пользователя является важным аспектом при создании пользовательских интерфейсов будущего. Например, если бы компьютер чувствовал, что пользователь огорчен, он смог бы изменить способ предоставления информации. Автомобиль, оснащенный такой системой управления, смог бы почувствовать, что водитель начинает засыпать или находится под воздействием алкоголя или наркотиков. Группа разработчиков использует различные датчики для определения эмоционального состояния пользователя. Например, гальванический датчик сопротивления кожи (GSR) может определить состояние кожи человека, что позволит системе оценить эмоции пользователя. Датчик дыхания способен определить частоту и глубину дыхания пользователя и идентифицировать тревогу. Наконец, датчик электромиограммы может измерять активность мускулов и определять их сокращение. Группа исследователей научилась определять даже степень сжатия зубов человека, что позволяет выяснять степень его гнева.

Почувствовать внешние эффекты эмоций – одна задача, но идентифицировать то, что они представляют для конкретного человека, – это совсем другая задача. Группа MIT также изучает распознавание эмоциональных состояний (и обучение) с помощью группы видимых маркеров. Поскольку маркеры могут быть различными для разных людей, распознавание должно включать обучение, которое позволит привязывать физические измерения параметров к определенным чувствам.

Синтез эмоций в разумных машинах является другой интересной идеей в разработках группы MIT. Изучается не только синтез эмоций, проявляющихся внешне, но и моделирование эмоций в виде внутреннего механизма. Представьте подсистему космического корабля, для которой были смоделированы различные эмоции. Если корабль потерял курс на Землю, он синтезирует страх и таким образом умножает свои усилия по поиску правильного курса. Также представьте, что в таком состоянии корабль, понимая, что потерял курс, растратит на генерацию эмоций весь запас энергии в батареях. В этом случае потеря курса становится критичной, поскольку может привести корабль к гибели (смерти). Страх, «почувствованный» кораблем, приведет к более активным действиям по поиску правильного курса. Когда курс будет найден, страх резко снизится, а чувство счастья усилится. Когда корабль ощущает страх, он меньше будет концентрироваться

на проблемах, которые не подразумевают спасения (например, научных экспериментах), а больше будет заниматься решением задач, ведущих к уменьшению страха и увеличению счастья. Хотя этот сценарий пока является воображаемым, использование эмоциональных эффектов может быть полезным при изучении работы по исправлению неполадок в системе.

Семантическая сеть Internet

Internet вмещает огромную массу информации, однако он, прежде всего, представлен в виде HTML и может использоваться только людьми. Чтобы сделать Internet доступным для компьютеров, исследователи создали семантическую сеть Internet. Она применяется не только для того, чтобы сделать поиск в сети более эффективным, но и для того, чтобы предоставлять информацию в виде знания, которое может использоваться всеми.

Семантическая сеть Internet по-прежнему стоит в самом начале формирования, но она быстро растет, так как ее развитие очень важно. Эта сеть была разработана Тимом Бернерс-Ли (Tim Berners-Lee), создателем обычной сети Internet и протокола HTTP. Семантическая сеть Internet построена на двух базовых технологиях, которые тоже являются относительно новыми. Первая технология, XML, представляет собой схему кодирования, которая позволяет управлять информацией с помощью идентифицирующих тэгов. Например, Web-страница включает следующую информацию:

```
<BR>
Part Number: 2N2222
<BR>
Type: Transistor
<BR>
Leads: 3
<BR>
Cost: $0.25
```

Эту информацию в XML можно представить так:

```
<discrete-part>
  <part-number>2N2222</part-number>
  <type>Transistor</type>
  <lead-count>3</lead-count>
  <price-usd>0.25</price-usd>
</discrete-part>
```

В данном случае не может быть никакой двусмысленности в том, что именно представляет собой информация. Информацию из XML легко анализировать, используя тэги в виде маркеров для данных. Если изменить в начальном представлении данных в виде HTML «Leads» на «Pins», то значение будет нарушено. Кроме того, в записи HTML непонятно, указана цена в долларах США или австралийских долларах. Использование XML-тэгов позволяет избежать двусмысленности.

Приведенный пример показывает, что при помощи информационной схемы можно добиться предоставления данных в более универсальной форме. Используя XML, можно даже строить схемы, в которых, например, транзистор будет представлен в виде уникальной группы тэгов, что еще больше упрощает процесс.

Вторая технология, на которой построена семантическая сеть Internet, называется RDF. Она представляет собой язык программирования на базе XML, который позволяет сделать информацию в сети Web доступной для компьютеров, однозначной и имеющей значимость. RDF кодируется в триплеты, при этом каждый элемент триплета представляет собой универсальный идентификатор ресурса – URI. URL-адрес, который чаще всего используется в качестве адреса для Web-страницы, тоже является URI. Элементы триплета RDF могут принимать форму субъекта, глагола и объекта. Благодаря этому информацию в сети Internet можно связывать и создавать схожие параметры или ассоциации.

Как говорилось ранее, схемы допускается задавать таким образом, чтобы термины имели общие значения. Используя технологию RDF, можно добиться единства в онтологии. Онтология представляет собой документ (на который ссылаются триплеты RDF), определяющий классы объектов и связи между ними. Онтологии предлагают механизм для осмысления данных в какой-либо области и формирования связей между ними. Это позволяет создавать новое знание на основе уже существующего в семантической сети Internet. В данной области еще очень много нерешенных проблем, но семантическая сеть Internet может стать одной из самых полезных разработок ИИ.

Литература и ресурсы

1. OpenCyc, http://www.opencyc.org.
2. Scientific Discovery, http://www.aaai.org/AITopics/html/discovery.html.
3. Semantic Web Community Portal, http://www.semanticweb.org/.
4. Web-сайт автономного программирования IBM, http://www-3.ibm.com/autonomic/index.html.
5. Web-сайт группы программирования эмоций в MIT (Affective Computing Group at MIT), http://affect.media.mit.edu/.
6. Web-сайт компании Cycorp, http://www.cyc.com.
7. Бернерс-Ли Т., Хендлер Д., Лассила О. Семантическая сеть Internet (Bernesr-Lee T., Hendler J., Lassila O. The Semantic Web // Scientific American, 2001, May). Доступно по адресу http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21.
8. Вагман М. Процесс научного открытия для людей и компьютеров: теория и разработка в психологии и искусственном разуме (Wagman M. Scientific Discovery Processes in Humans and Computers: Theory and Research in Psychology and Artificial Intelligence. Praeger Publishers, 2000).
9. Кепхарт Д. Иммунные системы с биологической мотивацией для компьютеров // Работы четвертого Международного семинара по синтезу и симуляции живых систем (Kephart J. A Biologically Inspired Immune Systems for

Computers // in Artificial Life 4: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems. – Cambridge, Mass: MIT Press).

10. Мински М. Будущее технологии AI (Minsky M. Future of AI Technology // Toshiba Review 47, №7, 1992). Доступно по адресу http://web.media.mit.edu/~minsky/papers/CausalDiversity.html.

Приложение. Архив с примерами

Архив с примерами к этой книге, который вы можете загрузить с сайта издательства «ДМК Пресс» www.dmk.ru, включает полный код примеров реализации рассмотренных в книге алгоритмов. В данном разделе кратко описаны представленные в архиве алгоритмы и программы.

Алгоритм отжига

Алгоритм отжига (моделирование системы охлаждения раскаленного металла до состояния твердого тела) описан в главе 2. Его возможности показаны на примере решения задачи N-ферзей. Исходный код программы содержится в папке software/ch2.

Теория адаптивного резонанса

Теория адаптивного резонанса (или ART1) представляет собой алгоритм работы с кластерами, который позволяет классифицировать (распределять по кластерам) объекты в группе данных. При этом образуются небольшие кластеры, включающие схожие друг с другом данные. Алгоритм выполняет всю работу самостоятельно, поэтому он часто находит такие свойства данных, которые пользователь мог и не заметить. В главе 3 ART1 используется для решения практической проблемы персонализации в системе выдачи рекомендаций. Алгоритм и программа содержатся в папке software/ch3.

Алгоритмы муравья

Алгоритмы муравья – это сравнительно новый метод, который может использоваться для поиска оптимальных путей по графу. Он описан в главе 4. Данные алгоритмы симулируют движение муравьев в окружающей среде и используют модель ферментов для коммуникации с другими агентами. Для демонстрации алгоритма муравья используется теоретическая задача коммивояжера (или TSP). Вы можете найти алгоритм и программу в папке software/ch4.

Алгоритм обратного распространения

Алгоритм обратного распространения, описанный в главе 5, является одним из основных методов, которые используются для обучения многослойных нейронных

систем. Для его выполнения потребуются набор для обучения и группа необходимых результатов. При обратном распространении ошибки отправляются обратно к узлам сети, что позволяет изменять веса соединений на основании ошибки в сравнении текущего и требуемого результата. Работа алгоритма демонстрируется при решении такой интересной проблемы, как обучение нейроконтроллеров (ИИ в компьютерных играх). Имея группы входов и доступных действий (известных как набор данных для обучения), алгоритм обучает простую нейронную сеть выполнять нужное действие в окружающей среде. Пройдя обучение, нейроконтроллер уже сам генерирует и реализует нужные действия в непредвиденной ситуации. Алгоритм обратного распространения и программа нейроконтроллера находятся в папке software/ch5.

Генетические алгоритмы и генетическое программирование

Генетические алгоритмы предлагают модель оптимизации, которую можно применять при решении как числовых, так и символических задач. В главе 6 генетическое программирование используется при создании последовательности инструкций. Подобные последовательности применяются при решении математических задач. Генетический алгоритм и программа содержится в папке software/ch6.

Искусственная жизнь и разработка нейронных сетей

Искусственная жизнь изучается в контексте нейронных сетей: рассматривается развитие простых организмов в синтетической среде (см. главу 7). Только избегая хищников и находя пищу, организмы выживают в среде. Воспроизводство агентов допускается только в том случае, если они выживают и достигают определенного уровня внутренней энергии. Это позволяет получать более здоровое и совершенное потомство. В качестве нейроконтроллеров для агентов выступают многослойные нейронные сети. Простые пищевые цепочки создаются с помощью двух различных типов организмов (хищника и травоядного животного). Программу искусственной жизни вы можете найти в папке software/ch7.

Экспертные системы

В главе 8 обсуждаются системы, основанные на правилах, при этом акцент делается на системах прямого логического вывода. В качестве правил и начальных фактов используется ряд примеров, что позволяет в результате встроить систему с правилами в более крупную систему и задействовать ее для создания системы

управления сенсорами, устойчивой к ошибкам. Исходный код системы содержится в папке software/ch8.

Нечеткая логика

В главе 9 описывается нечеткая логика, а также построенные на ее основе системы управления. Кроме того, рассматривается применение механизма нечеткой логики в других программах. Функции, реализованные в примере, содержат не только стандартные операторы нечеткой логики, но и вспомогательные функции, которые поддерживают создание функций нечеткой логики. Алгоритм нечеткой логики и программа-пример (модель зарядного устройства для батарей) находятся в папке software/ch9.

Скрытые модели Маркова

Модели биграммы (тип скрытой модели Маркова) являются темой главы 10. Они представляют собой сети, включающие состояния и переходы со связанными возможностями. В результате генерируется состояние на основании ассоциированного распределения вероятностей. Действие выполняется, и его результат отображается, хотя внутреннее состояние скрыто (поэтому модель и называется скрытой). Скрытые модели Маркова имеют множество областей применения. В главе 10 рассматривается одна из них – процесс создания текста с помощью модели обучения. Вы можете найти модель HMM и программу для создания текста в папке software/ch10.

Умные агенты

Умные агенты – последняя тема этой книги. В главе 11 рассматриваются агенты поиска и фильтрации. Умные агенты применяются различными способами; здесь описывается агент-фильтр, использующийся для фильтрации информации из Internet. Параметры поиска Web-агента задаются в простом файле конфигурации. Затем агент автономно собирает новости через протокол NNTP и предоставляет их пользователю с помощью HTTP-протокола (действуя аналогично Web-серверу). Программа Web-агента находится в папке software/ch11.

Системные требования

Для работы с примерами программ компьютер должен быть оснащен операционной системой Windows 95, 98, 2000, Me или XP и библиотекой Cygwin UNIX (вы можете бесплатно загрузить ее по адресу www.cygwin.com) или Linux (Red Hat 6.1, более поздней версии или другой совместимой с Linux средой); процессором 486 или более новым. Кроме того, компьютер должен иметь 64 Мб оперативной памяти, 60 Мб свободного пространства на диске, привод CD-ROM, а также доступ в Internet и Web-браузер (для работы с Web-агентом).

Предметный указатель

З

И

М

Н

П

Р

С

Т

Ф

Ц

Э

A

B

E

F

G

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС-КНИГА» наложенным платежом, выслав открытку или письмо по почтовому адресу: **123242, Москва, а/я 20** или по электронному адресу: **post@abook.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в Internet-магазине: **www.dmk.ru**, **www.abook.ru**.

Оптовые закупки: тел. **(495) 258-91-94, 258-91-95**; электронный адрес **abook@abook.ru**.

М. Тим Джонс

Программирование искусственного интеллекта в приложениях

Второе издание

Главный редактор	*Мовчан Д. А.*
Перевод	*Осипов А. И.*
Научный редактор	*Нилов М. В.*
Выпускающий редактор	*Морозова Н. В.*
Верстка	*Захарова Е. П.*
Графика	*Салимонов Р. В.*
Дизайн обложки	*Дудатий А. М.*

Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 25,35. Тираж 1000 экз. Зак. №

Подписано в печать 24.03.2011

Издательство «ДМК Пресс»