Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования
«Иркутский государственный университет»
Институт математики, экономики и информатики

КОМПЬЮТЕРНЫЕ НАУКИ

В четырех частях

Часть 4

Е. А. Черкашин

Системы искусственного интеллекта

Учебное пособие

Иркутск 2014

УДК 681.3(075.8)
ББК 32.97я73
К63

Печатается по решению ученого совета ИМЭИ

Издание выходит в рамках Программы стратегического развития ФГБОУ ВПО «ИГУ» на 2012–2016 гг., проект Р121-02-001

Рецензенты:
д-р физ.-мат. наук *В. И. Сажин*, канд. техн. наук *А. О. Шигаров*

К63 Компьютерные науки : учеб. пособие. В 4 ч. – Иркутск : Из-во ИГУ, 2014.

Ч. 4. Системы искусственного интеллекта / Е. А. Черкашин – 108 с.

ISBN 978-5-9624-1255-9 (ч. 4)

ISBN 978-5-9624-1251-1

Первая часть учебного пособия включает разделы: «Информация и данные», «Вычислительная система», «Устройство персонального компьютера», «Системное и прикладное программное обеспечение», «Компьютерные сети»; вторая – «Программирование», третья – «Базы данных и СУБД», четвёртая часть – «Системы искусственного интеллекта».

Предназначено для студентов вузов, обучающихся по направлениям «Математика», «Прикладная математика и информатика», «Математическое обеспечение и администрирование информационных систем», «Информационная безопасность».

УДК 681.3 (075.8)
ББК 32.97я73

Outfit.....

ISBN 978-5-9624-1255-9 (ч. 4)
ISBN 978-5-9624-1251-1

Оглавление

Предисловие

Интерес математиков к решению сложных задач при помощи вычислительной техники постоянно возрастает. Разрабатываются программные пакеты автоматизации выполнения математических операций и даже решения стандартных задач в автоматическом режиме. Алгоритмы, реализованные в таких пакетах, используют современные и классические методы поиска решения. Решение представляется в виде комбинации различных вариантов преобразования сложных структур данных, входных и выходных переменных различных методов. Для того чтобы продуктивно пользоваться этими пакетами, необходимо изучать, как они работают, как получают требуемое решение.

В учебном пособии в форме тренинга (*tutorial*) представляются самые простые методики поиска решения дискретных задач, относящихся к различным классам искусственного интеллекта (ИИ): «логическое программирование», «планирование действий», «решение задач в терминах ограничений» и «символьные вычисления» («компьютерная алгебра»). Основная задача пособия состоит в выработке навыков автоматизации некоторых аспектов творческой деятельности математика при помощи вычислительных машин и классических средств реализации программ из области ИИ. Навыки решения таких задач позволят в будущем создавать собственные программные системы, реализующие новые методы, для которых еще не было разработано соответствующего программного обеспечения.

Пособие разработано для студентов, обучающихся по специальности «Математика», может быть использовано всеми заинтересованными программистами, желающими овладеть некоторыми методами искусственного интеллекта. Пособие включает подборку материала по курсам «Рекурсивно-логическое программирование», «Искусственный интеллект». Оно никоим образом не претендует на полноту излагаемого материала и базируется на личном опыте преподавания. Пособие следует воспринимать как путеводитель, и учащиеся в процессе обучения должны активно использовать литературу, на которую в тексте указаны ссылки. В цитируемом тексте в виде сносок автор позволяет себе высказывать свое отношение к изложенному.

Форма тренинга, примененная в пособии, позволяет рассматривать материал в процессе монолога с читателем. Ставится задача, обсуждаются возможные варианты ее решения. Затем рассматривается новый материал, необходимый для реализации выбранного решения. Если в процессе появляется необходимость в изучении дополнительного материала, то параллельно основному монологу создается новый. Если читатель знаком с излагаемым материалом, то его можно пропустить и перейти далее (по ссылке) к рассмотрению решения задачи. Форма изложения, принятая в пособии, предполагает, что материал осваивается с самого начала до самого конца. То есть эту книгу будет сложно использовать как справочник.

Практически все разделы пособия представляют собой авторскую разработку, за исключением вводной части, которая содержит компиляцию материала из различных книг и справочников.

Во всем тексте, по возможности, указаны ссылки на оригинальные источники информации. Автор не преследует цели коммерческого использования учебного пособия. В электронной версии ссылки на литературу в списке литературы – активные и ведут к найденным в Интернете электронным версиям книг. Читатель должен решать самостоятельно: скачивать их из Интернета, покупать в магазинах или искать в библиотеках.

Данное учебное пособие является свободной книгой (так же как и свободное программное обеспечение), которую можно при определенных условиях читать, копировать и дополнять новым материалом. Адрес исходного кода методического пособия: `https://github.com/eugeneai/ais/tree/ais-pure-math`. Исходный код разрешено использовать в соответствии с лицензией CC BY-NC-SA 4.0 (Attribution-NonCommercial-ShareAlike 4.0 International), которая позволяет включать материал в свои произведения (необходимо указывать автора оригинальных материалов), запрещает коммерческое использование материалов (ввиду наличия заимствований в первой части) и требует распространение производных материалов производить по этой же самой лицензии (по указанной выше же причине). Адрес лицензии: `http://creativecommons.org/licenses/by-nc-sa/4.0/`.

В тексте пособия использована следующая разметка:

- `моноширинным шрифтом` приводятся программы, фрагменты программ в основном тексте пособия, а также имена идентификаторов, т. е. все, что имеет какое-либо отношение к тексту программы;

- *наклонным шрифтом* выделяются новые термины, вводимые в текст и возникающие, например, в определениях, а также текст выделенных примеров;
- при помощи «кавычек» выделяются метафоры, значения, элементы текстов программ, цитаты, слова, использованные в переносном смысле, и т. д.

Автор пособия надеется, что материал, представленный в данной книге, будет полезным при решении практических и научных задач, а также что он вызовет интерес студента к разработке программного обеспечения в области искусственного интеллекта.

Доцент кафедры ИТ ИМЭИ ИГУ
канд. техн. наук Е. А. Черкашин

Адрес сайта с методическим материалом:
`http://eugeneai.github.io/ais/`

P. S. Автора искать по адресу `eugeneai@icc.ru`, в поле «тема» просим указывать «ИИ-2014».

1. Информатика и искусственный интеллект

Среди задач, которые решают современные программисты, выделяются задачи создания программных систем математического моделирования и прогнозирования, проектирования и реализации информационных систем и баз данных, системного программного обеспечения. Все перечисленные задачи объединяет одно общее свойство – для широкого практического класса задач можно построить детерминированную процедуру (например, алгоритм) их решения. Существует большой класс задач, для которых такую процедуру построить достаточно сложно, а порой и невозможно. Например, разработать игровую систему, способную играть в шахматы с человеком на высоком профессиональном уровне. К таким задачам относятся также и задачи поиска решения (планирование действий, или Problem Solving), распознавание образов, экспертные консультации, интеллектуальное управление сложными динамическими объектами и т. д. В каждой такой задаче четко вырисовывается их первое общее свойство – необходимость *автоматизации принятия некоторого решения*. В других задачах четко вырисовывается еще одно свойство – *обработка символьной информации*. Примерами задач, обработка информации в которых основывается на преобразовании строк символов, выступают следующие задачи: автоматический перевод текста с одного естественного языка на другой, автоматическое доказательство теорем. В той или иной мере оба выделенных свойства присутствуют в каждой из задач.

Средства искусственного интеллекта[1], в частности логическое программирование, позволяют представить решение таких задач, алгоритм, в рекурсивном виде или в виде некоторого переборного процесса. Такое представление обладает одним полезным свойством – оно компактно и достаточно близко к исходной математической модели задачи по сравнению с изученной ранее процедурной парадигмой программирования. Программисту не требуется определять все действия, необходимые для достижения результата. Как правило, достаточно рассказать транслятору, какие данные есть в наличии, объяснить, как они связаны друг с другом и постановкой задачи. Система постарается получить решение самостоятельно. Логическое программирование прежде всего направлено на решение задач ИИ, поэтому в данном учебном пособии необходимо ввести читателя в базовые концепции ИИ. Для начала рассмотрим, как можно определить, относится ли ваша задача к задачам ИИ.

Рассмотрим задачи планирования действий. Что есть решение в этих задачах? Это ответ на вопрос «Какие действия необходимо выполнить и в каком порядке их надо выполнять, чтобы достичь цели из некоторого начального состояния?». Получается, что ответ на этот вопрос есть некоторая конечная последовательность действий. Эта последовательность представляется в памяти компьютера в виде некоторого ряда чисел, кодирующего эту последовательность. Построить (найти) эту последовательность, выбрать последовательность из возможных альтернатив – это и есть принятие решения.

[1] В англоязычной литературе данный термин называется *Artifical Inelligence, AI*.

Есть еще один интересный аспект алгоритма – массовость, т. е. алгоритмы должны строиться для некоторого класса задач, а не для конкретных входных данных. Что это значит? На вход программы, реализующей алгоритм, подаются какие-либо входные данные, задающие конкретную задачу из класса решаемых алгоритмом задач. Теперь представим такую ситуацию, что на вход алгоритма невозможно подать все необходимые данные, т. е. имеет место *неполнота информации*. Или другой вариант: имеется два эксперимента с разными результатами, но с одинаковым набором исходных данных. Какие данные следует передавать на вход алгоритма? Этот случай связан с *противоречивой информацией*. Разрабатывая программное обеспечение, позволяющее функционировать в таких условиях, приходится создавать подпрограммы, принимающие решение, что следует делать. Например, во втором случае можно запустить алгоритм для каждого набора данных и проанализировать полученные результаты. Может получиться так, что эти результаты не будут сильно отличаться друг от друга, а может этого не получиться. В последнем случае одним из вариантов дальнейших действий является поиск нового атрибута (характеристики), который позволит различать варианты, находящиеся в противоречии.

Задачи, обладающие перечисленными свойствами, и методы их решения на ЭВМ в конечном счете составляют предмет исследования искусственного интеллекта – одного из разделов информатики (Computer Science).

1.1. Определения

В литературе можно найти целый спектр определений термина «искусственный интеллект», однако, насколько известно авторам, ни один из них не принят как стандарт.

Среди многих точек зрения доминируют три [3]. Согласно первой, исследования в области искусственного интеллекта являются фундаментальными исследованиями, в рамках которых разрабатываются модели и методы решения задач, традиционно считавшихся интеллектуальными и не поддававшихся ранее формализации и автоматизации. Согласно второй точке зрения, новое направление связано с новыми идеями решения задач на ЭВМ, с разработкой принципиально иной технологии программирования, с переходом к архитектуре ЭВМ, отвергающей классическую архитектуру, которая восходит еще к первым ЭВМ. Наконец, третья точка зрения, по-видимому, наиболее прагматическая, состоит в том, что в результате работ в области искусственного интеллекта рождается множество прикладных систем, решающих задачи, для которых ранее создаваемые системы были непригодны.

Достаточно простые определения *искусственного интеллекта* показаны в табл. 1.1 [11]. Выделяются несколько комбинаций двух пар ключевых терминов: «размышлять» и «вести себя», «как человек» и «рационально».

Искусственный интеллект как наука насчитывает уже около 60 лет. Задачей этой науки является воссоздание с помощью искусственных устройств (в основном с помощью ЭВМ) разумных рассуждений и действий [4].

Таблица 1.1

Несколько определений искусственного интеллекта

Системы, которые размышляют, как люди	Системы, которые размышляют рационально
Системы, которые ведут себя, как люди	Системы, которые ведут себя рационально

Искусственный интеллект – раздел информатики, изучающий методы, способы и приемы моделирования и воспроизведения с помощью ЭВМ разумной деятельности человека, связанной с решением задач [6].

1.1.1. Тест Тьюринга

В книге [11] вводится понятие *агента*. *Агент* – субъект, находящийся в среде, имеющий цель своего существования, взаимодействующий со средой или другими агентами с помощью *рецепторов* и *эффекторов*. Рецепторы воспринимают информацию о среде, а эффекторы – это способ воздействия на среду, которое меняет среду, а следовательно, и информацию о среде. Агентом может являться как программа, так и человек. Вводится понятие *интеллектуального агента* – агента, обладающего интеллектом.

Агенты взаимодействуют друг с другом. Примером такого взаимодействия выступают, например, общение человека с человеком или работа человека с компьютерной программой.

Тест Тьюринга предложен Аланом Тьюрингом (1950) и был разработан, чтобы представить действующее определение ин-

теллекта [11]. Тьюринг определял интеллектное поведение как возможность достижения человеческого уровня производительности во всех задачах, где возможно обмануть человека, задающего вопросы. Грубо говоря, предложенный им тест состоял в следующем. Компьютеру задает вопросы человек через удаленное устройство. Тест считается пройденным, если человек не может сказать, кто или что на другом конце устройства: компьютер или человек.

С точки зрения агентов, этот тест можно представить так: один интеллектуальный агент (человек) по информационному каналу, не позволяющему ему использовать иную информацию, кроме ответов на поставленные им вопросы, анализирует поступающую информацию (ответы собеседника) от другого агента (испытуемого). Если первый агент не в силах определить, кто на другом конце информационного канала – человек или устройство, тогда считается, что испытуемый агент обладает интеллектуальными свойствами.

1.1.2. Задачи ИИ

Всякая задача, для которой неизвестен алгоритм решения, априорно относится к ИИ. Перечислим некоторые направления (задачи) ИИ [3].

Восприятие и распознавание образов. К таким задачам относятся распознавание текста (как печатного, так и рукописного), компьютерное зрение.

Автоматическое доказательство теорем. В этом направлении решаются задачи автоматизации математических исследований, разработки формальных (математических) методов логического вывода для поддержки решения других задач ИИ. Это направление нашло применение в задачах верификации программного и аппаратного обеспечения.

Игры. Автоматизация решения игровых задач, например игры в шахматы, кала́х, реверси, а также других игр.

Решение задач (Problem Solving), планирование действий. В этих задачах предполагается наличие некоторого выбора из возможных путей решения, требуется найти первое, лучшее или оптимальное решение. Примеры: составление расписания работы учебного учреждения, планирование действий автономного необитаемого аппарата.

Понимание естественного языка. Как правило, системы понимания естественного языка являются составляющими информационных систем различного назначения: от автоматических систем заказа билетов до систем ввода экспертного знания.

Логическое программирование. Языки и системы программирования высокого выразительного уровня, построенные на основе результатов исследования формально-логических систем, теорий исчислений. Эта область ИИ носит, кроме прочего, инструментальный характер, т. е. логическое программирование является средством реализации систем ИИ.

Экспертные системы. Экспертные системы (ЭС) позволяют заменять человека-эксперта в некоторой предметной области программной системой, способной проводить экспертные консультации пользователя. ЭС нашли широкое применение в индустрии.

Интеллектные информационные системы. Эти системы объединяют разнородные интеллектные системы (например, системы речевого общения, решения задач) для организации интеллектного доступа, обработки информации. Они, как правило, предназначены для работы с конечным пользователем низкой квалификации. Пример: электронные переводчики и разговорники.

Восприятие и усвоение знаний. Одна из задач ИИ – это приобретение знаний[1] (обучение, наполнение базы знаний) от человека или самостоятельно из среды функционирования. Системы усвоения знаний используются как подсистемы других интеллектных систем.

Интеллектное управление [2]. Новое направление, появившееся на стыке ИИ и теории управления, в котором разрабатываются управляющие системы, основанные на тех или иных методах ИИ. В настоящее время наиболее развиты методы управления на основе нечеткой логики и искусственных нейронных сетей. В этом новом направлении ведутся научные разработки Института динамики систем и теории управления СО РАН (`http://www.idstu.irk.ru`).

[1] *Англ.* – Knowledge Acquisition.

Робототехника (Robotics). Собирательное направление исследований, задачей которого является автоматизация функционирования роботов, вплоть до полной независимости их от человека.

Определение задач ИИ в контексте пособия. К сфере искусственного интеллекта относятся задачи, обладающие следующими свойствами [4]:

- в них используется информация в символьной форме: буквы, слова, знаки, рисунки. Это отличает область ИИ от областей, в которых традиционно компьютерам доверяется обработка данных в числовой форме[1];
- в них предполагается наличие выбора; действительно, сказать, что не существует алгоритма[2], – это значит сказать, по сути дела, только то, что нужно сделать выбор между многими вариантами в условиях неопределенности, и этот недетерминизм, который носит фундаментальный характер, эта свобода действия являются существенной составляющей интеллекта.

Излагаемый в пособии курс подразумевает непосредственное изучение технических аспектов применения методов ИИ, таких как применимость того или иного метода в конкретной задаче, реализация программных модулей конкретного метода. Поэтому

[1] Например, математическое моделирование, базы данных.

[2] Интуитивное определение алгоритма: *алгоритм* – это конечная последовательность действий, каждое из которых выполняется за конечное время, приводящая к определенному результату.

в предлагаемом курсе мы будем использовать следующее определение искусственного интеллекта:

> *Искусственный интеллект* – область информатики, в которой разрабатываются и исследуются методы построения программных систем и решения задач, так или иначе связанных с принятием решения и обработкой символьной информации[1].

1.1.3. Данные и знания

Одним из фундаментальных терминов ИИ является термин *знание*. Данный термин также является сложным в смысле его конструктивного определения, понятного читателю. Как правило, авторы статей по ИИ сознательно уклоняются давать более или менее точные определения, предполагая, что читателю это уже известно.

Более формальный термин «данные» получил широкое распространение в научно-техническом обиходе, в особенности в практике использования ЭВМ для решения самых разнообразных задач [3]. При этом вся обрабатываемая информация называется данными: начальными, промежуточными или конечными, входными или выходными. Для предложений естественного языка более привычны термин «знание» и глагол «знать». Ни у кого не вызывает возражений использование этого слова в предложениях вроде «Я знаю, как решить задачу» или «Я знаю, что вчера Петя встречался с Наташей». Сомнению может подвер-

[1] Данное определение задает термин в достаточно узком смысле.

гаться лишь истинность подобных утверждений, но никак не возможность сочетания слова «знать» с фрагментами предложения, обозначающими любую информацию, о которой говорится, что она кому-то известна.

Вопрос о разделении информации на данные и знания возник при разработке систем ИИ, характеризуемых в последнее время как системы, основанные на знаниях[1]. Был предложен ряд определений, отражающих различные аспекты этих понятий, но касающихся скорее форм (см. раздел 1.2) представления данных и знаний, правил их использования, чем их сути.

Два подхода к разработке методов и средств ИИ

«Снизу вверх». Суть подхода выражена в фразе «*Давайте создадим механическое (вычислительное) устройство, похожее на (моделирующее) мозг человека, а затем посмотрим, как оно будет решать задачи ИИ*».

«Сверху вниз». В основе этого подхода лежит *разработка методов моделирования процесса мышления человека (логических выводов, логических рассуждений)*.

Методы и системы ИИ, основанные на подходе «Снизу вверх», как правило, представляют собой сложную сеть взаимосвязанных, простых по сути, элементарных агентов. Эта сеть агентов формирует агента высокого уровня, направленного на решение конкретной задачи ИИ. Элементарные агенты сети вносят неболь-

[1] На самом деле это определение не охватывает такую важную отрасль ИИ, как нейронные сети.

шой персональный вклад в решение агента высокого уровня. Выделяют одно из достоинств этого подхода: *если задача «не решается» какими-то формальными методами, то ее «хоть какое-то» решение может быть получено методами «Снизу вверх»*. Как правило, схема применения описываемых методов и систем состоит из двух этапов: *обучение* на известном наборе «данные – решения» (данные и решения известны) и *решения* новых задач (данные известны, решения – нет).

Известным недостатком, присущим методам и системам «Снизу вверх», является неопределенность характеристик с точки зрения их практического применения: трудно ответить, например, на вопросы: «Сколько нужно агентов, чтобы решить конкретную задачу? Каковы должны быть связи между агентами?». В каждом конкретном случае требуются эмпирические исследования («сможет – не сможет»). Типичным представителем подхода «Снизу вверх» являются нейронные сети.

Моделирование логических выводов и рассуждений – основа подхода «Сверху вниз». В системах ИИ (агентах), основывающихся на этом подходе, как правило, *четко выделяют* функциональные блоки «Хранилище базы знаний», «Машина логического вывода» и интерфейс «Рецептор – Блок рассуждений – Эффектор». В задачу последнего блока входит преобразование информации в/из вид (-а), используемый (-ого) в первых двух блоках. Именно в этих методах и системах ИИ возникает задача представления знаний в некотором формализованном виде, удобном для осуществления их интерпретации и преобразований в блоке «Машина логического вывода». Примерами систем «Сверху вниз»

выступают язык программирования Пролог, экспертные системы, системы автоматического логического вывода.

Исходя из общих соображений, естественно определить данные как некоторые сведения об отдельных объектах, а знания – о мире в целом. В согласии с таким подходом будем считать, что:

> *данные* представляют информацию о существовании объектов с определенными комбинациями свойств (значений признаков), а *знания* – информацию о существующих в мире закономерных связях между признаками, запрещающих некоторые другие сочетания свойств у объектов.

Отсюда следует, что различие между данными и знаниями можно сформулировать так: *данные* – это информация о существовании объектов с некоторым набором свойств, а *знания* – информация о несуществовании объектов с некоторым набором свойств[1].

Используя логический формализм (см. далее) представления знаний, продемонстрируем эти понятия в формализованном виде. Пусть $H(x)$ обозначает высказывание «x является человеком», а $M(x)$ – «x – смертен». Теперь представим данные как утверждения с кванторами существования («существует x, x является человеком»):

$$A_1 = \exists x H(x),$$

[1] Вообще говоря, данное предложение не является определением, оно неконструктивно, т. е. не задает логических связей с известными объектами и терминами. В частности, термин «навык» тоже подходит под это же определение как нечто, отличное от «данных».

знания – утверждения с отрицанием существования («не существует бессмертных людей»):

$$A_2 = \neg\exists x\big(H(x)\&\neg M(x)\big),$$

легко преобразуемые в утверждения с квантором всеобщности («все люди смертны»):

$$A_3 = \forall x\big(H(x) \to M(x)\big).$$

Запись знаний с использованием логической связки отрицания перед квантором всеобщности (как в формуле A_2) не применяется на практике. Удобней и понятней форма A_3, т. е. форма с квантором всеобщности, утверждающим, что все объекты x, обладающие свойством H, будут обладать свойством M.

В формулах некоторые объекты удобно задавать именами. Например, утверждение «Сократ – человек» представляется в виде формулы $H(s)$, где s обозначает Сократа. Теперь продемонстрируем процесс обработки информации с использованием формализованного знания A_3 и исходного данного $H(s)$:

$$\Big(H(s)\&\forall x\big(H(x) \to M(x)\big)\Big) \to M(s).$$

Из того, что Сократ – человек и что все люди смертны, следует, что Сократ тоже смертен. Доказательство «от противного» предлагается построить читателю самостоятельно.

В конце книги [7] данные и знания (совместно с «умениями») охарактеризованы следующим образом.

Данные должны прежде всего храниться, а затем в порядке убывания приоритетов для непосредственной применимости

успешно находиться при нужде, проверяться, поддерживаться в порядке и обновляться при необходимости. Таким образом, они хранятся неизменными, пока не будут явно обновлены, и поэтому обычно внимание уделяют прежде всего сохранению, поддержанию их адекватности меняющемуся состоянию дел и целостности при необходимых изменениях.

Знания должны прежде всего преобразовываться. Далее, их нужно хранить, как и данные, они должны быть доступными, они должны конкретизироваться применительно к данной ситуации и обобщаться для целого класса применений. Они, конечно же, должны при необходимости пересматриваться. И, наконец, они должны переводиться с одного языка на другой.

Умения прежде всего применяются. Помимо этого, они преобразуются для обеспечения гибкости или приспособления к изменившимся условиям. Далее они обобщаются и пересматриваются.

В процессе исследований природных явлений ученые сначала накапливают информацию в виде данных, выделяя объекты и выявляя и измеряя их свойства. Знания – результат обработки данных и их обобщения. Классическим примером данных служат таблицы движения планет по небесному своду Тихо Браге, примером знаний – выведенные из них законы Иоганна Кеплера и затем, как обобщение результатов Кеплера, закон всемирного тяготения Исаака Ньютона.

1.2. Формализмы представления знаний

В интеллектуальных системах используются различные формализмы[1] представления знаний – *логический*, *сетевой*, *продукционный* и *фреймовый*.

Логический формализм традиционно применяется для реализации систем автоматического доказательства теорем, логических языков и сред программирования и экспертных систем. Сетевой формализм удобен в решении задач поддержки диалога с пользователем на естественном языке, формализации концептуального уровня предметной области (онтологий). Сетевой формализм совместно с логическим формализмом в настоящее время используется для верификации онтологий и построения логического вывода новых знаний и данных из указанного концептуального уровня.

Основная область применения продукционного формализма – описание баз знаний экспертных систем, систем поддержки принятия решений. Знания в продукционной модели представляются в виде утверждений «Если …, то …», база знаний – это множество таких утверждений. Отдельные утверждения в базе знаний можно удалять и добавлять новые, и если множество утверждений достаточно большое, то небольшие его изменения не влияют на общую работоспособность программной системы. То есть продукционный формализм удобен, если необходимо проводить эксперименты с логическим описанием предметной области *системы, основанной на знаниях*.

[1] Способы формального описания.

Фреймовый формализм в некоторой степени схож с объектами в прототипных объектноориентированных языках программирования, например JavaScript, Self. В основе формализма лежит идея декомпозиции предметной области на отдельные концепты и объекты, объединенные в иерархию, представления их в виде набора атрибутов. Каждый концепт и объект является набором слотов. Слоты состоят из описания и значения, значение должно соответствовать описанию или отсутствовать. Решение задачи распознавания на фреймовом формализме заключается в выявлении фрейма, лучшим образом соответствующего распознаваемому объекту.

Мы не будем далее подробно формально описывать формализмы в этом учебном пособии, познакомимся с их особенностями по мере необходимости и только в нужном объеме. Считается, что все формализмы эквивалентны, т. е. знания из одного формализма интерпретируются полноценно в другом формализме (тезис Черча).

2. Логические модели и логическое программирование

Самой известной системой ИИ, использующей логический формализм, является язык логического программирования Prolog (Пролог) [1], который будет предметом нашего дальнейшего изучения. Название «Пролог» образовано из слияния терминов: «ПРОграммирование в терминах ЛОГики». Пролог относится к классу языков, называемых *сентенциальными*. Классические реализации Пролога используются в вузах мира для обучения студентов методам автоматизации рассуждений. Специальные версии этого языка, например Prolog-III, являются дорогими коммерческими продуктами и предназначаются для решения комбинаторных задач с удовлетворением ограничений. К таким задачам относятся задачи раскроя материала, составление расписаний, задачи проектирования сетей изготовления и распространения продукции (задачи логистики).

Программа на языке Prolog представляет собой набор логических высказываний (утверждений), называемых *фразами* (*clauses*):

```
h(s).              % Сократ - человек.
m(X) :- h(X).      % Все люди смертны.
```

Каждая фраза завершается точкой[1] «.». В приведенном примере первое высказывание `h(s)` является *фактом*, которому интерпретатор языка «верит» по определению. Второе высказывание

[1] Со знака процента «%» начинается комментарий к тексту программы.

(«Все люди смертны») – это *правило*. Правило состоит из *головы* `m(X)` и *тела* `h(X)`. Знак «`:-`» обозначает логическую связку «←» и читается как «если». Правило связывает истинность высказывания, представленного в теле, с истинностью высказывания в голове: высказывание в голове правила истинно, если истинно высказывание в теле правила.

Идентификаторы в Prolog бывают двух видов. С маленькой буквы[1] начинаются идентификаторы, обозначающие что-то конкретное. В нашем примере идентификатор `s` обозначает Сократа, т. е. конкретный объект; `h` и `m` обозначают свойства аргумента «быть человеком» и «быть смертным» соответственно. Эти обозначения не меняются в процессе исполнения программы и в языке Prolog называются *атомами*[2]. Атомы также можно задавать при помощи строк в одинарных кавычках, например `'Российская Федерация'`. Такая форма записи позволяет использовать в качестве атомов любые строки.

Голова правила и атомарные высказывания, составляющие тело правила, строятся при помощи идентификаторов, обозначающих высказывания и называемых *предикатами*[3]. Высказывания `m(s)` и `h(s)` построены на основе предикатов $m(x)$ и $h(x)$.

[1] Разные реализации интерпретаторов задают различные ограничения на алфавит букв, из которых можно формировать идентификаторы. Одни интерпретаторы позволяют использовать только буквы латинского алфавита, другие – всю таблицу Unicode.

[2] В математической логике атомами обозначают неделимые высказывания, т. е. конструкции вида $m(x)$, $h(x)$. Это, конечно, приводит к некоторой путанице.

[3] Предика́т (от лат. *praedicatum* – сказуемое) в узком смысле – то же, что термин «свойство»; в широком смысле – отношение, т. е. общее свойство нескольких предметов. В логике – пропозициональная функция, т. е. выражение с неопределенными терминами (переменными), при выборе конкретных значений для этих терминов преобразующееся в осмысленное (истинное или ложное) высказывание.

В Prolog при обозначении предикатов переменные x, y и т. п. не используются, указывается только количество аргументов при помощи символа «/» (слеш): `h/1`, `m/1`.

Идентификаторы, начинающиеся с большой (прописной) буквы, – *переменные*. Значения переменных в общем случае меняются: в переменную в процессе исполнения программы подставляются объекты, структуры и другие переменные. В нашей программе в правиле используется одна переменная `X`. Существует специальный случай – переменная «_» (подчерк), при помощи которой обозначают никому не нужные значения, мы их будем использовать позже. С подчерка также начинаются названия переменных, например `_language`.

Приводимые в методическом пособии примеры демонстрируются на платформе интерпретатора SWI-Prolog [16]. Интерпретатор доступен в Интернете и является полностью свободным. Если интересно, то можно загрузить и изучить его исходный код. Домашняя страница находится по адресу: `http://www.swi-prolog.org/`. На одной из страниц сайта есть ссылка на версию SWISH-интерпретатора, функционирующую онлайн. Кроме того, на сайте размещена ссылка на обучающий материал по тематике логического программирования: `http://www.learnprolognow.org/`.

Чтобы запустить программу, необходимо ее загрузить в интерпретатор и сделать запрос:

```
?- ['socrates.pl'].    % Загрузка программы
% socrates.pl compiled 0.00 sec, 1 clauses
?- m(s).               % Запрос
true.
?- _
```

В приведенном примере «?-» обозначает приглашение к вводу пользователем запроса. Запросы – это тоже фразы языка, поэтому тоже заканчиваются точкой. Запрос специального вида `['socrates.pl']` загружает программу из файла `socrates.pl`, который должен находиться в той же папке, где был запущен интерпретатор. Фраза `m(s)` – это запрос к интерпретатору: «Правда ли, что Сократ смертен?» Ответ `true` или `yes` является результатом рассуждений интерпретатора над запросом, результатом доказательства его истинности. В последней строке примера подчерком показан курсор, т. е. интерпретатор находится в режиме ожидания ввода нового запроса. Запрос загрузки программы и последнюю строку с курсором далее не будем приводить в коде программы.

Интерпретатор Prolog не может вывести истинность некоторого запроса в двух случаях:

- в процессе поиска логического вывода невозможно найти подходящий факт, от которого зависит истинность нужного тела правила;
- логический вывод уходит в бесконечную рекурсию.

В первом случае Prolog выдает `false` или `no` и считает, что утверждение запроса ложно. Во втором случае необходимо прерывать работу интерпретатора при помощи комбинации клавиш Ctrl-C или Ctrl-Break в Windows-версии интерпретатора.

Процесс логического вывода запроса отслеживается средствами трассировки, встроенными в интерпретатор. Рассмотрим трассировку исполнения запроса к нашей программе:

```
?- trace.              % Включить трассировку.
true.

[trace]  ?- m(s).      % Запрос
   Call: (6) m(s) ?    % Нажать Enter.
   Call: (7) h(s) ?    % Enter
   Exit: (7) h(s) ?    % Enter
   Exit: (6) m(s) ?    % Enter
true.

[trace]  ?- nodebug. % Отключить трассировку.
true.
```

Предикаты `trace/0`, `notrace/0`, `debug/0` и `nodebug/0` управляют процессом трассировки и отладки. Текущий режим интерпретатора отображается строкой `[trace] ?-`. В режиме трассировки программа выполняется пошагово. Каждому шагу соответствует одна строка трассировки. В строке слова `Call:` и `Exit:` обозначают команды, выполняемые интерпретатором. Команда `Call:` показывает, истинность какого запроса (подзапроса) требуется доказать на очередном шаге; `Exit:` показывает, что соответствующий запрос успешно доказан.

Приведенный пример – это далеко не полный перечень возможностей языка программирования Prolog. Остальные его свойства будем рассматривать по мере необходимости при решении конкретных задач. Хорошим толковым учебником по языку является книга профессора Люблянского университета Ивана Братко [1]. Следующий абзац посвящен вопросам редактирования и загрузки программы. Если вы уже умеете это делать, то перехо-

дите сразу на стр. 32. Далее будем обозначать такие переходы следующим образом: ↪32.

Редактирование и загрузка программы. Многие ISO-совместимые реализации интерпретатора языка Prolog имеют встроенные графические оконные интерфейсы. При помощи этих средств можно редактировать и загружать программы, используя манипулятор мышь. В применяемом в данном пособии интерпретаторе SWI-Prolog также есть эти средства. В версии Windows редактирование программ Пролога доступно через меню оконного приложения. Встроенный редактор представляет собой упрощенную реализацию системы EMACS. Редактор вызывается из командной строки интерпретатора запуском предиката `emacs`, в том числе и в реализации Linux.

В классических реализациях языка Пролог введение списка утверждений в Пролог-систему осуществляется при помощи встроенного предиката `consult/1`. Единственным аргументом этого предиката `consult/1` является атом, который интерпретируется системой как имя файла, содержащего текст программы на Прологе. Файл открывается, и его содержимое записывается[1] в базу знаний. Если в файле встречаются управляющие команды, они сразу же выполняются. Возможен случай, когда файл не содержит ничего, кроме управляющих команд, например для загрузки других файлов. Для ускорения набора команды загрузки

[1] Фактически производится добавление новых знаний и фактов из загружаемого файла. Если в файле содержатся определения предикатов, которые уже имеются в рабочей памяти системы, то происходит обновление этих предикатов. Это может стать причиной неработоспособности программы. Рекомендуется иногда выходить из интерпретатора и очищать рабочую память.

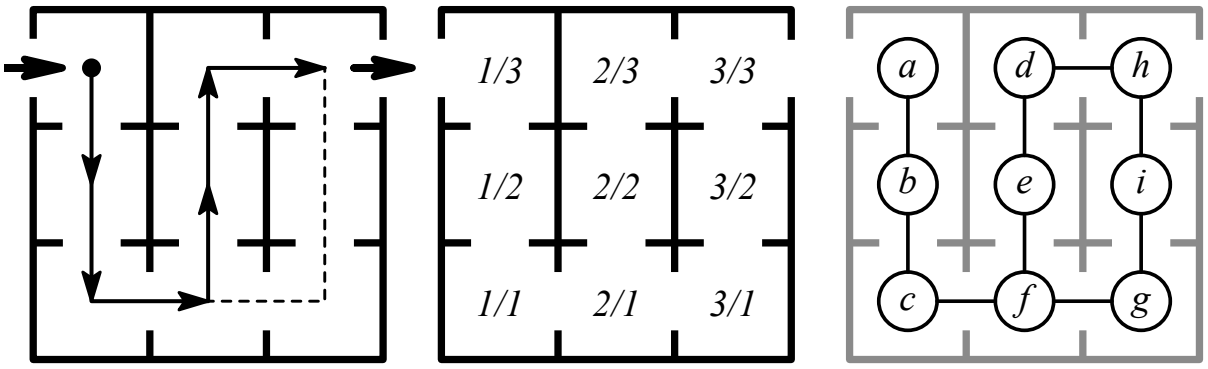
пользователи Пролога изобрели для себя следующую конструкцию, являющуюся синонимом предикату `consult/1`:

```
?- ['имя файла.pl'].
```

Работая с вышеупомянутым редактором, программу можно загрузить прямо из его окна при помощи комбинации клавиш Ctrl-c, Ctrl-b (Consult Buffer). Выход из редактора и закрытие его окна осуществляется комбинацией Ctrl-x, Ctrl-c, запись редактируемого буфера – Ctrl-x, Ctrl-s.

3. Планирование действий

Рассмотрим задачу поиска выхода из лабиринта, точнее задачу прокладки пути между двумя разными комнатами в лабиринте. В качестве примера будем использовать карту лабиринта, изображенную на рис. 3.1, *a*.



a) *b)* *c)*

Рис. 3.1. *Лабиринт*

Формализуем задачу. Сначала каждую комнату каким-нибудь образом обозначим. Первый вариант обозначения состоит в использовании двоек $\langle x, y \rangle$, x и y – координаты комнаты (рис. 3.1, *b*). В Prolog двойки удобно обозначать в форме `X/Y`, так как запятая используется в перечислении аргументов предикатов. Если факт наличия проема или двери (перехода) между двумя комнатами `A` и `B` обозначить предикатом `e(A,B)`, то следующая программа, набор фактов, будет представлять структуру нашего лабиринта:

```
e(1/3,1/2). e(1/2,1/1). e(1/1,2/1).
e(2/1,2/2). e(2/2,2/3). e(2/3,3/3).
e(2/1,3/1). e(3/1,3/2). e(3/2,3/3).
```

Другим вариантом обозначения комнат является использование имен («гостиная», «логово дракона») или просто букв *a*, *b*,

c и т. д. (рис. 3.1, *c*). Тогда набор фактов приобретет более лаконичный вид, но в этом случае теряется часть информации об относительном расположении комнат. Хорошо, что она нам не понадобится.

```
e(a,b). e(b,c). e(c,f).
e(f,e). e(e,d). e(d,h).
e(f,g). e(g,i). e(i,h).
```

Теперь надо системе Prolog объяснить, как прокладывать пути между комнатами, точнее что такое путь между двумя комнатами.

1. Между двумя комнатами существует путь, если между этими комнатами есть переход.

2. Между двумя комнатами `A` и `B` существует путь, если из комнаты `A` можно перейти в еще одну комнату `C`, а из нее построить путь в `B`.

Запишем это на языке Prolog, дополнив одну из предыдущих формализаций лабиринта:

```
path(A,B) :- e(A,B).
path(A,B) :- e(A,C), path(C,B).
```

Запрос, соответствующий решению задачи прокладки пути, будет следующим:

```
?- path(a,h).
true ;
true ;
false.
```

Программа выдала два положительных ответа и один `false`. Структура нашего лабиринта допускает наличие двух путей от входа к выходу. На каждый положительный ответ интерпретатор ожидает команды пользователя: требуется ли ему еще один ответ (`;`, Enter) или нет (Enter). В системе `gprolog` (GNU Prolog) [13] есть вариант `a`, который выдает на экран все решения. Использовать его надо с осторожностью, так как программа может порождать бесконечное количество вариантов решения задачи.

В программе появились новые синтаксические конструкции. Истинность высказывания в голове второго правила теперь зависит от двух предикатов `e/2` и `path/2`, соединенных запятой «`,`», которая обозначает конъюнкцию: надо, чтобы одновременно и `e(A,C)`, и `path(C,B)` были истинны. Далее, истинность `path(a,h)` выводится из двух разных правил. Пролог просматривает правила последовательно сверху вниз справа налево по тексту программы. Сначала будет осуществлена попытка вывести `path(a,h)` из первого короткого правила, что невозможно, так как `e(a,h)` отсутствует в наборе фактов. Затем Prolog попытается вывести `path(a,h)` из второго правила, что приведет к необходимости решать две связанные *подзадачи*, достигать две *подцели*, исполнять два *подзапроса*: `e(a,C)` и `path(C,h)`. Задача `e(a,C)` решается, если `C = b`, и тогда остается решить подзадачу `path(C,h)` при `C = b`, т. е. `path(b,h)`. Подзапрос `path(b,h)` доказывается аналогично запросу `path(a,h)`: интерпретатор рекурсивно строит решение задачи.

Рассмотрим трассировку задачи `path(a,c)`, решаемой за два перехода:

```
?- trace,path(a,c).
   Call: (7) path(a, c) ?
   Call: (8) e(a, c) ?
   Fail: (8) e(a, c) ?          % (1)
   Redo: (7) path(a, c) ?       % (2)
   Call: (8) e(a, _G1450) ?     % (3)
   Exit: (8) e(a, b) ?
   Call: (8) path(b, c) ?
   Call: (9) e(b, c) ?
   Exit: (9) e(b, c) ?
   Exit: (8) path(b, c) ?
   Exit: (7) path(a, c) ?
true ;                          % (4)
   Redo: (8) path(b, c) ?
   Call: (9) e(b, _G1450) ?
   Exit: (9) e(b, c) ?
   Call: (9) path(c, c) ?
   Call: (10) e(c, c) ?
   Fail: (10) e(c, c) ?
   Redo: (9) path(c, c) ?       % (5)
   . . . . .
```

В этом примере появились две новые команды `Redo:` и `Fail:`, соответственно обозначающие попытку вывести истинность запроса из следующего по списку правила и неудачное завершение поиска доказательства запроса. В точке (1) `Fail:` показывает, что утверждение `e(a,c)` отсутствует в списке фактов. Это приводит к необходимости (2) перейти к следующему правилу `Redo:`. Задача `e(a, _G1450)` то же самое, что и `e(a,C)`, только переменная `C` переименована в `_G1450`: каждый «запуск» правила требует разыменования используемых в нем переменных. В строке (4)

Prolog выдает первый положительный ответ, а пользователь просит предоставить еще один. Далее происходит достаточно длинный перебор вариантов переменной `_G1450`, который не приводит к новому решению, и интерпретатор выдает `false`.

Специальные запросы. В английском языке выделяются пять видов вопросов: общие (Do You …?), разделительные (You are …, ain't You?), специальные (What do You do?), к подлежащему (Who is …?) и «или»-вопрос (Do you like coffe or tea?). Первые два класса вопросов в Прологе представляются запросами, где в параметрах не используются переменные, например `path(a,h)` и `e(a,b)`. Специальные вопросы и вопросы к подлежащему, помимо ответа на вопрос «Так это или нет?», нацелены на получение дополнительной информации: «Кто?», «Когда?» и т. д. В Прологе такие вопросы формулируются в виде запросов с использованием переменных:

```
?- path(a,X).
X = b ;
X = c ;
X = f ;
X = e ;
X = g ;
. . . .
false.
```

Запрос соответствует вопросу «Куда можно попасть, войдя в лабиринт?». Дополнительная информация, получаемая пользователем, – значения переменной `X`, при которых запрос, ограниче-

ние на значение X, истинен. «Или»-вопросы в Prolog задавать и неестественно, и не принято ;-) .

Программа работает правильно, но нам мало, чтобы программа просто показывала, существует ли путь между двумя комнатами лабиринта. Хотелось бы, чтобы программа также показывала сам путь. Для того чтобы представить путь как последовательность комнат лабиринта, нужно к нашим знаниям добавить еще одну структуру данных – *список*. ↪39

3.1. Списки

Задачи, связанные с обработкой списков, на практике встречаются очень часто. Скажем, нам понадобилось составить список студентов, находящихся в аудитории. В Prolog список определяется как последовательность *термов*[1], заключенных в квадратные скобки. Приведем примеры списков Пролога:

```
[jack, john, fred, jill, john]
[name(john, smith), age(jack, 24), X]
[X, У, date(12, january, 1986), X]
[]
```

Запись специального вида [H | T] определяет новый список, где H – первый (левый) элемент нового списка, а T – его остальные элементы. Говорят, что H – голова, а T – хвост списка [H | T].

[1] Терм – структура, обозначающая объект [8]. Терм обобщает понятия константы, переменной и функции. Например, высказывание h(wife(s)) содержит терм wife(s), обозначающий жену Сократа.

На запрос

```
?- L=[a | [b, c, d]].
```

будет получен ответ

```
L=[a, b, c, d],
```

а на запрос

```
?- T=[a, b, c, d], L=[2 | T].
```

получим ответ

```
T=[a, b, c, d], L=[2, a, b, c, d].
```

Запись `[H | T]` используется и в обратную сторону, для того чтобы определить голову и хвост списка. Так, запрос

```
[H | T]=[a, b, c].
```

дает ответ

```
H = a, T = [b, c].
```

Заметим, что употребление только имен переменных `H` и `T` необязательно. Кроме записи вида `[H | T]`, для выборки термов используются переменные. Запрос

```
?- [a, X, Y]=[a, b, c].
```

определит значения

```
X=b, Y=c,
```

запрос

```
?- [person(X) | T]=[person(john), a, b].
```

значения

```
X=john, T=[a, b].
```

Можно отделять в голове списка несколько элементов, соответствующая запись выглядит так: `L=[H1, H2, H3 | T]`. Пустой список задается так: «`[]`».

Возвращаемся к лабиринту. Далее будем использовать списки для хранения пути в лабиринте. Для этого введем третий параметр в предикат `path/2`:

```
path(A,B, [A-B]) :- e(A,B).
path(A,B, [A-C|T]) :- e(A,C), path(C,B,T).
```

Новые правила можно добавлять к предыдущей программе, так как Prolog воспринимает одноименные предикаты с разным количеством аргументов как самостоятельные. Предикаты `path/2` и `path/3` формируют в данной программе свои независимые множества правил. Выполним запрос на поиск путей, соединяющих вход и выход лабиринта:

```
?- path(a,h,L).
L = [a-b, b-c, c-f, f-e, e-d, d-h] ;
L = [a-b, b-c, c-f, f-g, g-i, i-h] ;
false.
```

Результат совпадает с ожидаемым (рис. 3.1, *а*) и представляется списком переходов, формирующих путь. Каждый переход обозначен структурой вида A-B, где A и B – вершины графа структуры лабиринта, соединенные дугой. Можно представить результат просто списком вершин:

```
path2(A,B, [A,B]) :- e(A,B).
path2(A,B, [A|T]) :- e(A,C), path2(C,B,T).
```

Выполним запрос к новому предикату path2/3:

```
?- path2(a,h,L).
L = [a, b, c, f, e, d, h] ;
L = [a, b, c, f, g, i, h] ;
false.
```

Коллекционирование решений. Если есть необходимость собрать коллекцию всех решений, обладающих некоторым свойством, используются предикаты bagof/3, setof/3 и findall/3. Спецификации интерфейсов этих предикатов схожи, рассмотрим их, как обычно, на примере:

```
?- findall(X, path2(a,h,X), L).
L = [[a, b, c, f, e, d, h], [a, b, c, f, g, i, h]].
```

Предикат формирует список L путей-решений, т. е. список L таких X, что справедлив запрос path2(a,h,X). Предикат findall/3 истинен всегда, даже если нет ни одного X, такого, что path2(a,h,X) истинен. В последнем случае findall/3 выдает L=[] (пустой список), а вот bagof/3 и setof/3 потерпят неудачу (будут ложными). Предикат setof/3 отличается от других еще и тем, что из выходного списка будут удалены повторения.

3.2. Алгоритмы планирования действий

Задача поиска пути в лабиринте – простейший пример класса задач *планирования действий*. В общих словах задача относится к этому классу, если выполняются следующие условия:

1) выделяется не обязательно конечный набор *допустимых состояний*, т. е. состояний, не противоречащих природе и условиям задачи;

2) состояния соединяются *переходами* (не обязательно двунаправленными). Эти переходы задаются набором правил, не противоречащих условиям задачи;

3) задается *логическое условие* $P(x)$, *распознающее целевые состояния* x;

4) среди допустимых состояний выделяется одно, в котором система находится в начальный момент, т. е. *исходное состояние*.

Решением задачи планирования действий является построение последовательности логически связанных действий, которые необходимо выполнить, чтобы перевести систему из начального состояния в одно из целевых.

Математическая структура, позволяющая формализовать этот класс задач, строится на основе графа, называемого *графом пространства состояний* (*State Space Graph*):

$$G = \langle V, E, P(x), s \rangle,$$

где V – множество допустимых состояний (вершины графа), E – переходы (дуги графа, соединяющие вершины), а s – начальное состояние (стартовая вершина графа). В такой формализации решением задачи планирования действий является *путь*, соединяющий исходное состояние в один из целевых. При этом, как правило, интересует не просто поиск решения, а поиск лучшего или оптимального решения, характеризующегося, например, минимальным количеством переходов между смежными вершинами.

Алгоритмы поиска пути в графе с заданными свойствами определяются *стратегиями перебора* неизвестных вершин, видимых из области графа с уже известными вершинами (рис. 3.2, *a*).

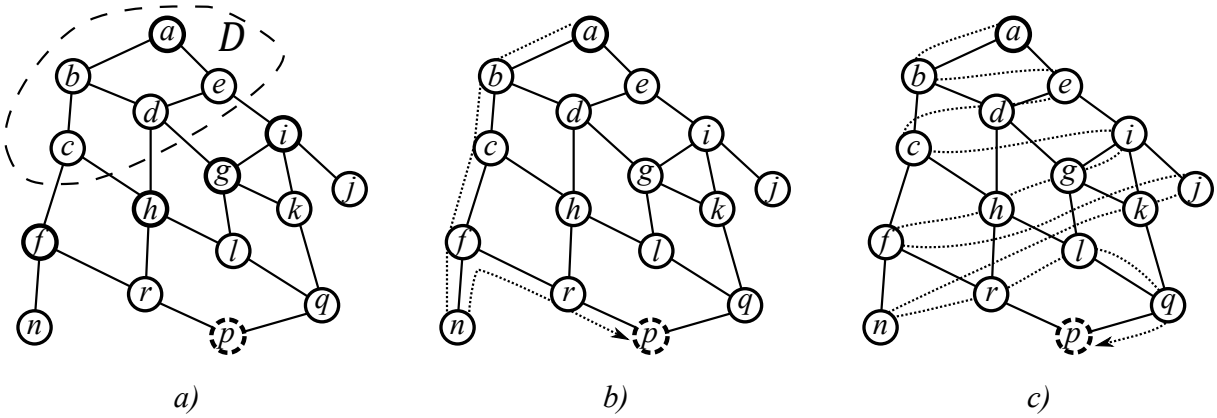


Рис. 3.2. *Стратегии перебора вершин*

Область D графа пространства состояний (ГПС) содержит вершины, которые к какому-то моменту уже были исследованы и информация о которых сохранена в оперативной памяти. Из этой области видны вершины f, h, g и i. *Стратегия перебора* определяет, какую из этих четырех вершин надо исследовать на очередном шаге. Выделим несколько основных стратегий:

1) случайный выбор вершины;
2) выбор вершины, максимально удаленной от стартового узла;

3) выбор вершины, ближайшей к стартовому узлу.

Самой простой процедурой поиска решения в ГПС есть случайный перебор. Начиная со стартовой вершины, всякий раз переходим в одну из смежных вершин и проверяем, не является ли она целевой. Если она таковой не является, то повторяем процедуру. Очевидно, что такой дезорганизованный процесс не дает гарантии, что за приемлемое время будет получен приемлемый результат: в связном ГПС он находит решение (останавливается) в общем случае в потенциальной бесконечности. Тем не менее такая процедура (а не алгоритм) очень легко реализуема программно.

3.2.1. Поиск в глубину

Вторая стратегия реализуется в алгоритмах поиска *в глубину* (*depth-first search*) (рис. 3.2, *b*). Примером такого поиска выступает рассмотренная ранее программа прокладки пути в лабиринте. Представим ее еще раз, только в более общем виде:

```
dfs(V,[]):- p(V).    % depth-first search
dfs(V,[V-N|T]):- \+ p(V), after(V,N), dfs(N,T).
p(h).
after(X,Y):- e(X,Y); e(Y,X).
```

Запрос к программе – неотъемлемая часть задачи:

```
?- dfs(a, S).
S = [a-b, b-c, c-f, f-e, e-d, d-h] ;
S = [a-b, b-c, c-f, f-e, e-d, d-e, e-d, d-h] ;
S = [a-b, b-c, c-f, f-e, e-d, d-e, e-d, d-e, ...]
```

ГПС задан переходами `after(X,Y)`, где `X` и `Y` – допустимые состояния. Множество допустимых состояний не задано в явном виде, оно порождается предикатом `after` по мере необходимости. Условие распознавания целевого состояния – `p(h)`. Начальное состояние указано в запросе первым аргументом: `dfs(a, S)`. В программе использованы две новые синтаксические конструкции: «`\+`» – логическая связка отрицания «¬»; и «`;`» – логическая связка «`V`» («или»). Применение логической связки «или» позволяет в некоторых случаях сокращать текст программы за счет объединения нескольких правил с одним и тем же заголовком в одно.

Результат запуска программы показывает, что количество порождаемых путей для нашего двунаправленного графа потенциально бесконечно. Действительно, если ГПС содержит циклы, то поиск в глубину может «зацикливаться», т. е. соответствующий алгоритм неполон даже на конечных ГПС. Но все же у поиска в глубину есть два замечательных свойства: а) алгоритм прост в реализации, что видно из предыдущего примера; б) для хранения состояния перебора, границы подграфа D, достаточно списка вершин, соединяющих стартовую вершину с текущей, т. е. оперативная память компьютера используется достаточно эффективно. Текст программы процедуры поиска в глубину, реализованной на языках общего назначения, также является лаконичным. В качестве хранилища состояния перебора, как правило, используется стек, организуемый непосредственно операционной системой:

```
// Файл dfc.c. Компиляция: gcc -o dfs dfs.c
#include "stdio.h"
int after[9][9]={                    // (1)
```

```
// a b c d e f h i g
  {0,1,0,0,0,0,0,0,0}, // a
  {1,0,1,0,0,0,0,0,0}, // b
  {0,1,0,0,0,1,0,0,0}, // c
  {0,0,0,0,1,0,1,0,0}, // d
  {0,0,0,1,0,1,0,0,0}, // e
  {0,0,1,0,1,0,0,0,1}, // f
  {0,0,0,1,0,0,0,1,0}, // h
  {0,0,0,0,0,0,1,0,1}, // i
  {0,0,0,0,0,1,0,1,0}};// g
int dfs(int x) {
  int y=0;
  if (x==6) { // h==6              // (2)
    return 1;
  } else {
    for (y=0;y<=9;y++) {          // (3)
      if (after[x][y]) {          // (4)
        if (dfs(y)) {             // (5)
          printf ("%i-%i\n",x,y);
          break;
        }}}}} // плохой стиль! Сэкономим пару строк в пособии.
int main(int argc, char ** argv) {
  if (dfs(0)) return 0; // a==0      (6)
  printf ("Нет решений\n");
  return 1;
}
```

В приведенном примере лабиринт представлен в виде матрицы смежности (1), перебор дочерних вершин реализован в цикле (3–5) по переменной `y`. Функция `dfs` возвращает «истину» 1, если решение найдено (2) и часть пути из текущей вершины `x` до целевой уже выведена на экран. Начальная вершина указывает-

ся при первом вызове `dfs` (6). Программа лаконичная[1], только она не останавливается – «зацикливается», бесконечно перемещаясь между вершинами `a (0)` и `b (1)`. Пример на языке Prolog сначала рассматривает все переходы «вперед», только потом вводит в рассмотрение переходы в обратном направлении. Эта особенность реализации программы и структура лабиринта позволяет-таки находить первое решение. Несмотря на это, будем по-прежнему считать и эту реализацию «ненадежной».

Ограничение по глубине. Самый простой способ решения проблемы «зацикливания» – это ввести ограничение по глубине поиска. Всякий раз, когда алгоритм входит в рекурсию, будем отнимать единицу от специальной переменной. Если значение этой переменной станет равным нулю и решение не будет найдено, то вернуть «ложь» `0`. Рассмотрим модификацию программы на языке Prolog, а совершенствование C-программы оставляем в качестве простого упражнения:

```
dfsl(V,[],_):-p(V),!.  % limited depth-first search
dfsl(V,[V-N|T],D):- D>0, after(V,N),
                    D1 is D-1, dfsl(N,T,D1).
```

Новый предикат `dfsl` по сравнению с `dfs` содержит третий аргумент, переменную – счетчик глубины входа в рекурсию. В первом правиле счетчик не используется, так как там распознается решение. Во втором правиле сначала выполняется проверка

[1] Программа на языке Prolog занимает значительно меньше места, чем программа на языке C. Это в основном связано с тем, что Prolog-интерпретатор представляет сам по себе механизм поиска решения «в глубину», а в C-программе его надо реализовать.

условия, что глубина допустима `D>0`, затем рассматривается дочерняя вершина, конструкция `D1 is D-1` вычисляет арифметическое выражение `D-1` и связывает результат с новой переменной `D1`. В конце правила осуществляется вход в рекурсию с новым значением глубины.

Первое правило использует анонимную переменную «_», она нужна здесь, чтобы «занимать место» третьего неиспользуемого аргумента, ее значение нам не важно. Во втором правиле убрана проверка `\+p(V)`, в ней нет особого смысла: решающие пути, неоднократно попадающие в целевые вершины, остаются решающими. Приведем пример работы программы:

```
?- dfsl(a, L, 8).
L = [a-b, b-c, c-f, f-e, e-d, d-h] ;
L = [a-b, b-c, c-f, f-e, e-d, d-e, e-d, d-h] ;
L = [a-b, b-c, c-f, f-e, e-f, f-e, e-d, d-h] ;
L = [a-b, b-c, c-f, f-e, e-f, f-g, g-i, i-h]
```

Видно, что результат программы немного отличается от предыдущего. Дальнейшим совершенствованием подхода является управление ограничением по глубине. Сначала необходимо попробовать найти решения длиной в один шаг. Если попытка потерпела неудачу, перейти к решениями длиной в два шага, если опять неудача, то перейти к решениям, состоящим из трех шагов и т. д. Такой алгоритм имеет специальное название IDA (*Iterative Deeping Algorithm*). Алгоритм обладает важным очевидным свойством: первое найденное решение является оптимальным. Реализацию этого алгоритма оставляем читателю.

Распознавание циклов. Другим популярным подходом к совершенствованию алгоритмов поиска решения в ГПС, основанных на стратегии «в глубину», является распознавание циклов. Для реализации распознавания необходимо хранить перечень пройденных вершин. Как было сказано выше, этот перечень, путь из стартовой вершины в текущую, сохраняется в стеке. Но ни в Prolog, ни в C к этому списку нет прямого доступа, к сожалению. Поэтому будем «тащить» третий аргумент в спецификации процедуры поиска, указывающий на этот путь:

```
dfsr(V,[],_):-p(V). % dfs with recognition
dfsr(V,[V-N|T],Q):- after(V,N), \+ member(N,Q),
                    dfsr(N,T, [N|Q]).
```

Библиотечный предикат `member(X,L)` истинный, если `X` является элементом списка `L`. Его реализация следующая[1]:

```
member(X,[X|_]).
member(X,[_|T]):-member(X,T).
```

Предикат интерпретируется следующим образом: «элемент списка находится либо в голове, либо в хвосте списка». Предикат демонстрирует, как применяются анонимные переменные. Тестировать работоспособность `member/2` можно запросами `member(b,[a,b,c])` и `member(X,[1,2,3])`. Но вернемся к нашей основной задаче и посмотрим, как программа строит решающие пути:

[1] Большинство реализаций языка Prolog не разрешают переопределять библиотечные предикаты. Если собираетесь экспериментировать, то выберите для него какое-нибудь другое имя.

```
?- dfsr(a, L, []).
L = [a-b, b-c, c-f, f-e, e-d, d-h] ;
L = [a-b, b-c, c-f, f-g, g-i, i-h] ;
false.
```

Программа находит всего два пути: а) решающие пути, б) пути без циклов. Распознавание циклов – процедура достаточно накладная, она имеет сложность $O(n/2)$ «в среднем», где n – длина пройденного пути. Но в сложных переборных задачах распознавание циклов и экономит объем использованной памяти, и отсекает заведомо худшие варианты решения.

3.2.2. Поиск в ширину

Стратегия поиска *в ширину* (*breadth-first search*) обладает таким же свойством, что и стратегия алгоритма IDA, т. е. первое найденное решение будет оптимальным. Вот только достигается это без повторного перебора исследованной области графа *D* (рис. 3.1, *a*, *c*). Процесс поиска хранит все так называемые *пути-кандидаты*, пути, претендующие на то, чтобы стать решением. Все пути-кандидаты начинаются в исходном узле и заканчиваются текущей вершиной. Если текущая вершина – целевая, то такой путь-кандидат становится решающим. Хранение списка путей-кандидатов не является лучшим решением по управлению оперативной памятью вычислительного процесса, но зато достаточно просто реализуем:

```
bfs([[X|T]|_],[X|T]):-p(X),!.  % (1)
bfs([[X|T]|Ways], S):-
        findall([Y,X|T],       % (2)
```

```
    (after(X,Y), \+ member(Y,[X|T])), L),
  append(Ways,L, NWays), % (3)
  bfs(NWays,S).  % Истина где-то рядом...
```

Первый аргумент `bfs/2` – список путей-кандидатов, второй – решающий путь. В обоих правилах `[X|T]` – путь-кандидат, находящийся на рассмотрении на данном шаге. Его голова `X` соответствует текущей вершине, начальная вершина находится в самом конце хвоста пути `T`. Путь-кандидат хранит решение в обратном порядке: новые вершины добавляются в начало списка (2).

Во втором правиле второй аргумент `findall/3` является выражением (конъюнкцией) в скобках, порождающим переходы к дочерним вершинам, но только такие, которые не приводят к циклам. Библиотечный предикат `append(L1,L2,L3)` добавляет слева от `L2` список `L1` и порождает список `L3`. Вот его не самая эффективная реализация [1]:

```
append([],L,L).
append([X|T1],L2,[X|T3]):-append(T1,L2,T3).
```

В первом правиле, в строке (1), использовано отсечение «`!`», в нашем случае запрещающее Prolog искать еще одно решение, если первое уже найдено. Запрос на поиск первого оптимального решения выглядит следующим образом:

```
?- bfs([[a]],S).
S = [h, d, e, f, c, b, a].
```

В результате действия отсечения интерпретатор выдал решение и без лишних вопросов прекратил дальнейшие поиски. В [1] приведен вариант алгоритма, где подграф *D* представляется в виде дерева, что значительно эффективнее списка путей-кандидатов.

Теперь давайте займемся более сложными задачами, которые порождают большое (в том числе бесконечное) пространство поиска. Поиск решений в этих задачах занимает много ресурсов компьютера, если решать их простыми переборными алгоритмами. Необходимо сделать так, чтобы область D была как можно меньше, а поиск пытаться вести где-то рядом с вероятным решающим путем.

3.3. Использование дополнительной информации

Обобщим задачу планирования действий: добавим третий аргумент `after/2` и получим `after(X,Y,V)`, где `V` – стоимость дуги, которая раньше была у всех дуг одинакова и равнялась 1. Теперь стоимость решающего пути будет складываться из стоимостей составляющих его индивидуальных переходов. Дальнейшие наши рассуждения также будут подразумевать использование поиска в ширину как базовой стратегии решения задач.

Практическая задача, обладающая в той или иной мере перечисленными свойствами, – это поиск маршрутов между городами. Граф дорожной сети состоит из вершин (городов), дуг (дорог, соединяющих два смежных города). Данные сети автомобильных дорог Российской Федерации в Интернете найти не удалось, зато в документе [12] содержится достаточно подробная, но не вполне корректная информация о расстояниях между станциями Российских железных дорог (РЖД). Все данные импортированы в базу данных SQLite `rzd.sqlite` и опубликованы на сервере по адресу: `https://github.com/eugeneai/ais/tree/cs/code`.

Импорт данных осуществлен при помощи сценария (*script*), реализованного на языке программирования Python [5, 9, 17]. Для исполнения дальнейшего программного кода необходимо скачать все файлы с указанного адреса к себе в рабочую папку.

В документе [12] среди всего множества станций выделяются станции, называемые *транзитными пунктами* (ТП). Судя по структуре документа, расстояния от «обычных» станций указываются именно до транзитных пунктов. Примерами транзитных пунктов являются станции `'Иркутск-Пассажирский'`, `'Тайшет'`, `'Лена'` (г. Братск), `'Улан-Удэ'`. Решать задачу поиска кратчайшего расстояния будем между транзитными пунктами, а задачу поиска ближайшего транзитного пункта от обычных станций оставим в качестве упражнения. Дополнительную информацию будем вычислять, исходя из физических параметров карты – географических координат транзитных пунктов. Координаты будем запрашивать в Интернете на сервисе OpenStreetMap (OSM), а также сохранять их в базе данных для обеспечения быстрого доступа. Рассмотрим, как SWI-Prolog взаимодействует с SQL – базами данных. ↪56

3.3.1. Доступ к базам данных

Вообще говоря, следует уточнить, о какой поддержке баз данных в Prolog идет речь. В интерпретаторе доступны:

- встроенная база данных, реализуемая при помощи динамических предикатов, набора фактов, перечень которых меняется при помощи предикатов `assert/1` и `retract/1` [1];

- внешние базы данных, в том числе SQL. Далее будем говорить именно об этом.

Как любой другой полноценный язык программирования, Prolog поддерживает организацию подпрограмм в отдельные модули. Для того чтобы загрузить правила и факты из модуля доступа к базам данных SQLite, надо выполнить инструкцию `:-use_module(library(prosqlite)).` Такие инструкции являются запросами и исполняются в процессе загрузки программы.

Модуль `prosqlite` не включен в дистрибутив SWI-Prolog по умолчанию, его необходимо загрузить из Интернета в виде одноименного пакета. Чтобы получить список доступных пакетов, содержащих в своем названии интересующий текст, надо выполнить поисковый запрос в интерпретаторе:

```
?- pack_list(prosqlite).
% Contacting server at http://www.swi-prolog.org/
%  pack/query ... ok
p db_facts@0.1.0  - Common db-tables-as-facts and
                    SQL layer for ODBC and SQLite.
p prosqlite@1.0   - An SWI-Prolog interface to
                    SQLite
```

Установка пакета, как и выполнение предыдущего запроса, требует наличия соединения с Интернетом[1]. В процессе установки интерпретатор может спросить, стоит ли продолжать установку, а также куда следует устанавливать библиотеку (Create directory for packages). Отвечать следует нажатием y.

[1] На самом деле можно загрузить архив пакета и выполнить установку и без подключения к Интернету. Процедура установки описана на сайте SWI-Prolog [16].

```
?- pack_install(prosqlite).
% Contacting server at ... ok
Install prosqlite@1.0 from http://stoics... Y/n?

Create directory for packages
   (1) * /home/eugeneai/lib/swipl/pack
   (2)   Cancel

Your choice? 1

% Contacting server at ... ok
% ”prosqlite-1.0.tgz” was downloaded 7 times
Package:                prosqlite
Title:                  An SWI-Prolog ... SQLite
Installed version:      1.0
Author:                 Sander Canisius, ...
Maintainer:             Nicos Angelopoulos
Packager:               Nicos Angelopoulos
Home page:              http://stoics.org.uk/...
Download URL:           http://stoics.org.uk/...
Install ”prosqlite-1.0.tgz” (740,988 bytes) Y/n?
% Found foreign libraries for target platform.
% Use ?- pack_rebuild(prosqlite). to rebuild ...
```

В заключение установки проверим, загружается ли наш модуль (библиотека):

```
?- [library(prosqlite)].
% library(prosqlite) compiled into prosqlite
% 0.01 sec, 116 clauses
```

Представление таблиц в виде предикатов. SWI-Prolog позволяет непосредственно выполнять SQL-запросы к базе данных, в результате запроса последний аргумент связывается со структурой `row(…)` для каждой возвращаемой записи. Аргументами данной структуры являются значения атрибутов записи:

```
?- sqlite_connect(rzd, db, [as_predicates(true)]).
?- sqlite_query(db,
        'SELECT * FROM station LIMIT 3', Row).
Row = row('Вентспилс (эксп.)','Лат',98306,0) ;
Row = row('Шаблиевская','С-Кав',518827,0) ;
Row = row('Салар','Узбк',720903,0) ;
false.
```

Первый запрос создает соединение с базой данных `rzd.sqlite`, при этом соединение ассоциируется с атомом db, который далее используется как глобальный идентификатор этого соединения. Третий аргумент настраивает дополнительные возможности библиотеки – представление таблиц базы данных в виде предикатов языка Prolog. Второй запрос к Prolog является `SELECT`-запросом к таблице данных о существующих станциях `station`. Prolog выдал три ответа, затем волнение запроса закончилось неудачей. Теперь воспользуемся дополнительными возможностями библиотеки и запросим перечень станций по-прологовски, а соответствующий запрос на выборку данных он синтезирует сам:

```
?- station(Name,Region,Code,TP).
Name = 'Вентспилс (эксп.)',
Region = 'Лат',         % Регион
Code = 98306,           % Ж/Д-код станции
TP = 0 ;                % Транзитный пункт?
```

```
Name = 'Шаблиевская',  % Название станции
Region = 'С-Кав',
Code = 518827,
TP = 0
```

Если требуется узнать только транзитные пункты, то надо просто указать значения соответствующих параметров:

```
?- station(Name,'В-Сиб',Code,1).
Name = 'Лена',
Code = 927105 ;
Name = 'Иркутск-Пассажирский',
Code = 930108 ;
Name = 'Улан-Удэ',
. . . . . .
```

Теперь запросы к базам данных можно комбинировать с другими предикатами программы. Соединение с базой данных разрывается предикатом `sqlite_disconnect(db)`. В таблице 3.1 приведены данные о структуре основных таблиц базы данных `rzd.sqlite`. Структуры остальных таблиц можно посмотреть при помощи команды «`.schema`», подключившись к базе данных `sqlite3 rzd.sqlite`.

Возвращаемся к задаче поиска оптимального пути. Итак, у нас есть база данных с необходимыми таблицами и данными. Теперь наша задача – построить алгоритм, осуществляющий поиск оптимального пути в ГПС сети железных дорог РЖД, эффективно использующий процессор. Сначала обобщим поиск в ширину – учтем тот факт, что дуги теперь нагружены (дороги, соединяющие города, разной длины):

Таблица 3.1

Основные таблицы базы данных структуры РДЖ

Таблица station		Таблица geocache	
name text,	Название станции	station int,	Код ТП
region int,	Филиал РДЖ	lon int,	Долгота
code int,	Код станции	lat int	Широта
transit int	Это ТП?	lat place_id	OSM-код объекта
Таблица route		Таблица dist	
a int,	Первый ТП	a int,	Код станции
b int,	Второй ТП	b int,	Код ТП
dist int	Расстояние	dist int	Расстояние до ТП

```
ucs([_-[Target|T] |_],Target,[Target|T]):-!.
ucs([G-[X|T]|Ways], Target, S):-
        Target\=X,
        findall(G1-[Y,X|T], after([X|T],G, Y,G1), L),
        append(L, Ways, NWays),
        keysort(NWays,SNWays),
        ucs(SNWays,Target,S).

after([S|R],SG, T,TG):-
        transdist(S,T,D),  % Расстояние D между
        \+ member(T,[S|R]),     % ТП S и T
        TG is SG + D.      % Приращение стоимости пути
```

В новой программе проверка вхождения дочерней вершины в пройденный путь и вычисления стоимости нового пути вынесены в предикат `after/4`, и поэтому в первый аргумент передается не вершина, а пройденный путь. Предикат `after/4` по сравнению с `after/2` дополнен двумя аргументами: `SG` – стоимость пройденного пути `[S|R]`, `TG` – стоимость пути `[T,S|R]`. Список

путей-кандидатов теперь оформлен в виде структуры G-W, где W – путь-кандидат, G – его стоимость. В ucs/3 добавлен новый аргумент – название целевой станции: в задаче целевая вершина всего одна, да и удобнее задавать конечную станцию в запросе, а не программе. Далее, предикат keysort(NWays, SNWays) сортирует список NWays двоек G-W по возрастанию значения G. Сортировка выполняется при помощи алгоритма быстрой сортировки над всем списком NWays. Алгоритм реализован на языке C как двоичный модуль и выполняется достаточно эффективно, поэтому левые два аргумента append/3 переставлены местами: длина (в элементах) списка L значительно меньше, чем длина списка Ways. Таким образом, первым элементом в отсортированном списке двоек SNWays оказывается двойка с минимальной стоимостью пути-кандидата.

Запрос на поиск пути между железнодорожными станциями – столицами двух российских регионов выглядит следующим образом[1]:

```
?- ucs([0-['Улан-Удэ']],'Новосибирск-Главный',S).
S = [Новосибирск-Главный,Сокур,Юрга I,Тайга, Анжерская,
     Ачинск I,Уяр, Тайшет, Иркутск-Пассажирский,Улан-Удэ]
```

В программе bfs/2 новые пути-кандидаты добавлялись в конец существующего списка путей-кандидатов, и список при этом оставался упорядоченным. В ucs/3 так сделать не получится, так как вставлять новые пути уже надо внутрь списка. Поэтому приходится использовать быструю сортировку на каждом шаге,

[1] В список станций не входит «Красноярск-Главный», так как в базе данных он не является транзитным пунктом.

сортировка вставкой не даст дополнительных преимуществ для рекурсивных списков в Prolog. Ускорить процесс вставки можно за счет использования сбалансированных деревьев для представления списка путей-кандидатов.

Представленный алгоритм называется «Поиск по критерию стоимости» (метод равных цен, *Uniform-Cost Search*, UCS). Если стоимости всех дуг в ГПС положительны, то UCS является полным и оптимальным, так же как и алгоритм поиска в ширину. Вместо перехода в ближайший к исходной вершине узел UCS переходит (раскрывает) в вершину с минимальной стоимостью пути до исходной вершины. Если все стоимости дуг одинаковы, UCS превращается в поиск в ширину.

3.3.2. Алгоритм A*

Теперь посмотрим, как можно ускорить поиск, используя дополнительную информацию. На рисунке 3.3, *а* показана железнодорожная сеть (ГПС) Южного Урала, Западной Сибири и Казахстана. На примере этой сети продемонстрирован процесс UCS раскрытия вершин ГПС во время поиска решения. Задача состоит в том, чтобы найти путь со станции Тобол до станции Омск. Пунктирными окружностями обозначена область D на разных этапах поиска. Как видно, UCS не делает никаких предположений о местонахождении Омска и вынужден «ходить» вокруг начальной вершины, на каждом шаге постепенно увеличивая радиус окружности. На практике всегда возникает вопрос: «В какую сторону следует двигаться, чтобы исключить неподходящие вершины ГПС и быстрее добраться до целевой вершины?».

В нашем примере в качестве такого направления выступает направление на целевую вершину (рис. 3.3, *b*).

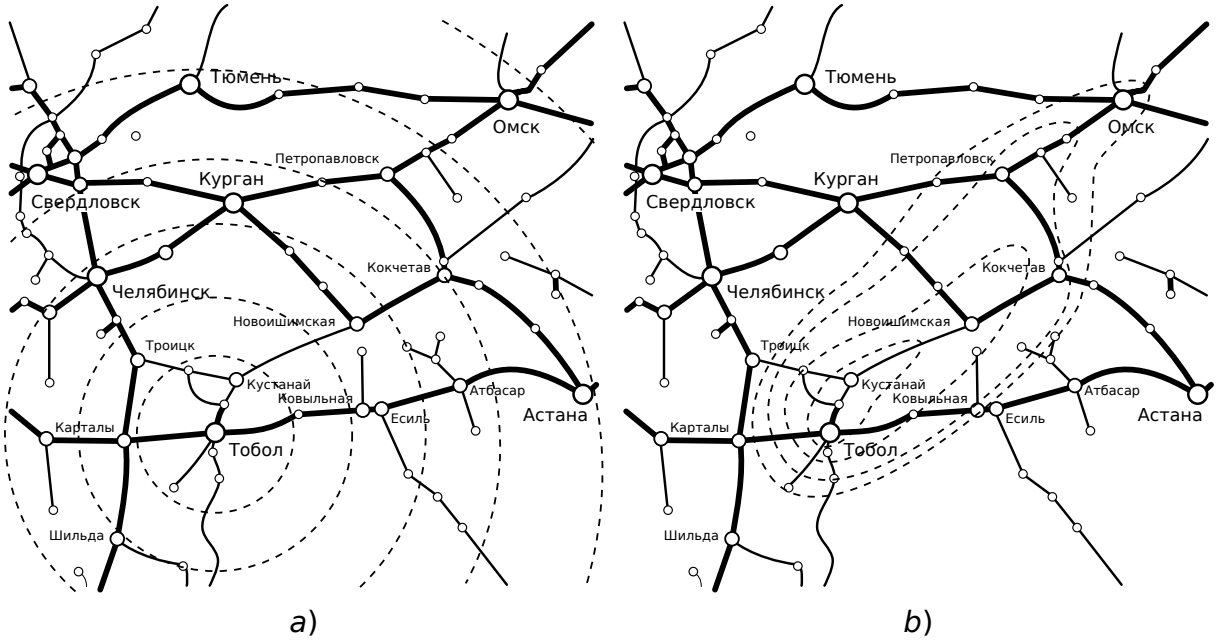


Рис. 3.3. *Сужение области поиска*

Зададим стоимость кратчайшего решающего пути, проходящего через некоторую вершину ГПС x, функцией $f: V \to R$ следующего вида:

$$f(x) = g(x) + r(x), \tag{3.1}$$

где $g(x)$ – стоимость кратчайшего пути из начальной вершины до вершины x, а $r(x)$ – стоимость пути из x до целевой вершины. Если x является целевой вершиной, то $r(x) = 0$ и $g(x) = f(x)$, т. е. значения функций равны стоимости решающего пути. Если x не является целевой, то значение $r(x)$ не известно. Если значение $f(x)$ можно было бы вычислить для любой вершины x, то тогда можно было бы разработать алгоритм поиска, который точно «знает», куда следует идти на каждом шаге построения решения. Несмотря на то что такое в общем случае невозмож-

но, кое-что все-таки сделать можно: будем оценивать величину $r(x)$ снизу некоторой функцией $h(x)$. Для всех x должно выполняться неравенство $r(x) \leqslant h(x)$. Значения функции $f(x)$ будем оценивать таким образом:

$$f(x) \geqslant g(x) + h(x). \tag{3.2}$$

В нашей задаче значения $h(x)$ соответствуют расстоянию между городами, т. е. по самой короткой линии на поверхности сферы. Для вычисления этого расстояния нам потребуется база данных `geocache` (табл. 3.1) и реализация подсистемы запроса географических координат на сервере OpenStreetMap, которым в этот раз займемся чуть позже.

Программа поиска, основывающегося на оценке $h(x)$, состоит из трех основных предикатов. Первый предикат – `after/6`, который получает еще два дополнительных параметра: `GPS` – координаты целевой вершины (входной параметр), `F` – оценка f(`T`) для дочерней вершины `T` (выходной параметр). Вторым предикатом выступает `bf1/3`, являющийся реализацией собственно алгоритма поиска. Третий предикат – это процедура `bf/3`, подготавливающая данные для основного предиката `bf1/3`:

```
bf1([_-s(G,[Target|T]) |_],          % (1)
        Target-_,s(G,[Target|T])):-!.
bf1([_-s(G,[X|T])|Ways], Target-GPS, S):-
        Target\=X,
        findall(F1-s(G1,[Y,X|T]),
           after([X|T],G,GPS, Y,G1,F1), L),
        append(L, Ways, NWays), keysort(NWays,SNWays),
        bf1(SNWays,Target-GPS,S).
```

```
after([S|R],SG, GPS, T,TG, F):-
        transdist(S,T,D),
        \+ member(T,[S|R]),
        TG is SG + D,
        geodist(T, GPS, GDist),     % (2)
        F is TG + GDist.

bf(Start, Target, Sol):-
        geocode(Target, Lon, Lat, _),
        bf1([0-s(0,[Start])],
          Target-ll(Lon,Lat), Sol). % (3)
```

Используемые в программе структуры также немного усложнены по сравнению с UCS. Элемент списка путей-кандидатов (1) теперь представляет собой структуру вида `F-s(G,[X|T])`, где `X` – текущая вершина, `G` – стоимость пройденного пути $g(X)$ до текущей вершины, `F` – оценка $f(X)$. Параметр `GPS` предиката `after/6` – структура `ll(Lon,Lat)`, `Lon` – долгота целевой вершины, `Lat` – ее широта. Оба значения заданы в градусах. Структура создается в строке (3) и передается в составе двойки `Target-GPS`. Оценка $h(T)$ вычисляется в `geodist/2` в строке (2). Приведем пример поиска кратчайшего пути между двумя станциями:

```
?- bf('Свердловск-Сортировочный', 'Новосибирск-Главный', S),
    write(S).
OSM: Пермь II
    lon: 56.1744706, lat:58.0050018    % (1)
S = s(1535,[Новосибирск-Главный,Обь,Татарская,Карбышево I,
    Войновка,Тюмень,Богданович,Шарташ,
    Свердловск-Сортировочный])
```

В результате запроса получается структура, которая содержит решающий путь и его стоимость (длина в км). В процессе построения решения программа запрашивает данные местоположения станций в Интернете. Результаты запросов выдаются на экран в виде отладочной информации (1).

Построенная нами реализация алгоритма A* основывается на непереоценивающей функции h, т. е. выдает первым оптимальное решение. Необходимо сделать пару замечаний.

1. Расстояние между городами вычисляется в пространстве на сфере, что не отражает структуры железнодорожной сети. Например, расстояние между Москвой и Иркутском составляет чуть меньше 4200 км, траектория полета самолета проходит вблизи города Сургута, находящегося значительно севернее Трассиба.

2. Запуская программу, можно наблюдать, что гарантированно получить правильные координаты станции у нас не всегда получится: сервис OSM не специализируется на предоставлении координат станций и может выдавать координаты объектов со схожим названием. Так, например, на момент публикации данного пособия не удалось правильно определить координаты станции Лена, которая находится в городе Братске Иркутской области.

После небольшого экскурса в обработку файлов XML (*eXtensible Markup Language*) и доступа в Интернет рассмотрим решение более общих задач на основе переборных алгоритмов. ↪70

3.3.3. Доступ в Интернет и обработка XML

Язык программирования Prolog представляет собой удобное средство обработки рекурсивных структур. Использованные нами в разделе 3.2.2 списки относятся к таким рекурсивным структурам. Другой известной рекурсивной структурой является *дерево*, которое лежит в основе структуры формата данных XML. Формат XML позволяет представлять информацию таким образом, чтобы она, с одной стороны, достаточно эффективно обрабатывалась алгоритмически (на компьютере), а с другой, была понятна человеку. Вот, например, ответ с сервера геокодирования Nominatim [14]:

```
<?xml version="1.0" encoding="UTF-8" ?>
<searchresults timestamp='Thu, 15 Jan 15 07:54:30 +0000'
               attribution='Data © OpenStreetMap contributors, ODbL
                            1.0. http://www.openstreetmap.org/copyright'
               querystring='Иркутск'
               oviewbox='-112.15,61.44,112.15,-61.44'
               polygon='false'
               exclude_place_ids='158888259'
               >
  <place
      place_id='158888259' osm_type='relation' osm_id='1430614'
      place_rank='16'
      boundingbox="52.2094352,52.4217324,104.0579985,104.462928"
      lat='52.2895979' lon='104.2805843'
      display_name='Иркутск, городской округ Иркутск, Иркутская
                    область, Сибирский федеральный округ, Российская
                    Федерация'
      class='place' type='city' importance='0.70744217496694'/>
</searchresults>
```

Основные элементы XML – это *теги*, *атрибуты* и текст, заполняющий пространство между тегами. В данном примере тегами выступают <searchresults> и <place>. Тег <place> расположен «внутри» тега <searchresults>. Тег <searchresults> – корневой, который по стандарту XML должен быть только один, остальные могут повторяться. В нашем примере сервер геокодирования может выдать несколько объектов, соответствующих запросу. В XML-ответе сервера это отразится наличием повторений тега <place> для каждого объекта.

Атрибуты ассоциированы с тегами и несут формализованную дополнительную информацию. В примере тегами lat и lon заданы географические координаты объекта (города Иркутска), полное наименование (display_name), тип объекта type и т. д. Каждый атрибут приравнивается к некоторому значению. Таким образом, тег place со своими атрибутами описывает объекты, а <searchresults> – результат обработки запроса, состоящий из одного или нескольких объектов.

Процедуры (предикаты) обработки XML, запросы к сайтам Интернета, а также обработка списка ключевых слов (атрибутов) реализованы в соответствующих библиотеках среды Prolog. В нашей программе необходимо импортировать эти библиотеки:

```
:- use_module(library(http/http_open)).
:- use_module(library(sgml)).   % Обработка XML
:- use_module(library(option)).
```

Предикат geocodequery/4 запрашивает на сервере объекты по названию Name и возвращает широту, долготу и идентификатор объекта. Библиотечная процедура http_open в качестве

первого аргумента принимает список ключевых слов в формате <название структуры>(<значение>). Элементы списка позволяют формировать строку URL-запроса к серверу из отдельных компонент. Важным элементом этого списка является search/1 со списком параметров запроса к скрипту search.php. Параметр format указывает серверу на тот факт, что ответ (перечень объектов) надо возвращать в виде XML. Остальные значения понятны на интуитивном уровне.

Переменная In получает идентификатор входного потока данных, представляющих собой ответ сервера. Этот поток передается в процедуру интерпретации XML в виде текста. Текст транслируется в древовидное представление и помещается в переменную S. Переменная S затем унифицируется со структурой, чтобы разобрать корневой тег на отдельные элементы. Унификация помещает в L список тегов <place>. Этот список анализируется в предикате geoplace/4, где изымаются требуемые данные о местоположении найденных объектов.

```
geocodequery(Name, Lon, Lat, ID):-
        http_open([
            host('nominatim.openstreetmap.org'),
            path('/search.php'),
            search([ q=Name,
                     lang=ru,
                     format=xml ])
            ], In, [] ),
        load_xml(In, S, []),
        S=[element(searchresults,_,[_|L])],
        close(In), geoplace(L, Lon,Lat,ID),
```

```
        writef("OSM: %w lon: %w, lat:%w",
                [Name, Lon, Lat]),nl.

geoplace(element(place,Attrs,_), Lon, Lat, Id):-
    option(lon(Lon1),Attrs), atom_number(Lon1,Lon),
    option(lat(Lat1),Attrs), atom_number(Lat1,Lat),
    option(place_id(Id),Attrs).
geoplace([X|_], Lon, Lat, Id):-
    geoplace(X, Lon, Lat, Id).
geoplace([_|T], Lon, Lat, Id):-
    geoplace(T, Lon, Lat, Id).
```

Предикат `geoplace` в качестве первого параметра принимает структуру `element(place, <список атрибутов>, [])`, соответствующую тегу `place`, или список таких структур. Предикат возвращает геоданные об объектах, представленных в списке тегов `place`. Предикат `option(lon(Val),L)` ищет в списке `L` элемент вида `lon=Val` или `lon(Val)` и унифицирует соответствующим значением переменную `Val`. Процедура `atom_number(X,Y)` переводит атомы `X` (строки) в целые числа `Y` (и наоборот), если такое возможно.

Следующий текст представляет собой программу обеспечения доступа предикату `after/6` к геоданным по названию станции `Station`:

```
geocode(Station, Lon, Lat, Id):-
    \+ number(Station),
    station(Station, _, Code, _),
    geocode(Code, Lon, Lat, Id).
```

```
geocode(Station, Lon, Lat, Id):-
    number(Station),
    geocache(Station, Lon, Lat, Id),!.

geocode(Station, Lon, Lat, Id):-
    number(Station),
    station(Name, _, Station, _),
    geocodequery(Name, Lon, Lat, Id),
    sqlite_query_f(db,
     'INSERT INTO geocache (station, lon, lat,
         place_id) values
         (%w,%w,%w,%w)'-[Station,Lon,Lat,Id], _),!.
```

Сначала `geocode/4` пытается найти данные в локальной базе данных координат `geocache/4`. Если попытка терпит неудачу, то осуществляется запрос к интернет-сервису Nominatim и запись результата в базу данных `geocache` для дальнейшего использования. В тексте SQL-запроса `INSERT` `%w` обозначает место вставки строкового представления очередного значения из списка `[Station,Lon,…]`. Предикат `sqlite_query_f/3` реализован в программе следующим образом:

```
sqlite_query_f(Conn, S-Args, ROW):-
    swritef(Query, S, Args),
    sqlite_query(Conn, Query, ROW).
```

Всеми преобразованиями `%w` в форматной строке `S` в конкретные значения из списка `Args` занимается библиотечный предикат `swritef/3`. Результат преобразования записывается в переменную `Query`.

Из приведенных примеров программного кода хорошо видно, что синтез двух реляционных языков программирования в рамках одного приложения производится достаточно просто, при этом не надо изобретать дополнительные конструкции, например циклы по строкам результата запроса. Поэтому, разрабатывая в следующий раз информационную систему со встроенными подсистемами математического моделирования, следует задуматься над вопросом о том, где в ней будут использоваться императивные языки программирования, а какие подсистемы следует реализовать на реляционном языке программирования.

4. Поиск решения на основе перебора

Иногда приходится сталкиваться с задачами, эффективный алгоритм решения которых очевидным образом реализовать не удается, либо недостаточно времени на анализ свойств задачи и поиск подходящего алгоритма. К ним относятся, например, *задачи с удовлетворением ограничений*[1]. Такие задачи являются математическими проблемами, определенными на конечном наборе объектов, чьи значения должны удовлетворять ряду ограничений, выраженных в виде неравенств и логических выражений. Исследования в области решения задач CSP ведутся достаточно давно, и по сей день актуальность этих исследований только повышается.

К задачам CSP сводятся многие задачи искусственного интеллекта, в частности планирование действий. Задачи удовлетворения ограничений довольно часто демонстрируют большую комбинаторную сложность, и практически для каждой индивидуальной задачи строятся собственные варианты эвристических алгоритмов[2] их решения. Примеры известных задач: «Восемь ферзей», «Раскраска карты», «Судоку», «Поиск выполняющего набора», «Составление расписания вуза».

Формально задачи CSP определяются следующим образом. Заданы вектор переменных $\vec{V} = \langle v_1, v_2, \dots, v_n \rangle$, где n – количество переменных (натуральное); множество множеств $D = \{d_1, d_2, \dots,$

[1] Англ. *Constraint satisfaction problems, CSP.*

[2] Алгоритмов, в которых на этапах выбора очередного направления поиска решения из нескольких альтернатив используется дополнительная (по отношению к исходной) информация для задания этим альтернативам некоторого предпочтения.

$d_n\}$, где d_i – непустое множество значений (домен), которые может принимать переменная v_i, $i = 1, 2, \ldots, n$. Задается также логическое условие $P(\vec{V}) = P(v_1, v_2, \ldots, v_n)$, которое истинно, если значения, присвоенные переменным v_i, соответствуют условиям правильной комбинации. Например, для задачи «Восемь ферзей» $P(\vec{V})$ истинно, если все ферзи, расставленные на доске, не бьют друг друга. Иногда говорят о системе ограничений и о таких значениях переменных, при которых все ограничения выполняются (истинны). Систему ограничений можно записать как конъюнкцию индивидуальных ограничений, т. е. свести опять же к единому логическому условию $P(\vec{V})$.

4.1. Алгоритм British Museum

Одним из простых алгоритмов, решающих задачи CSP, является алгоритм British Museum[1]. Алгоритм реализует самый общий подход в задачах поиска решения на основе последовательной проверки всех возможностей (одну за одной), начиная с самых простых решений.

Алгоритм реализует концептуальный, а не практический подход, оперируя огромным количеством возможных альтернатив. В частности, в теории он представляет способ найти самую короткую программу, которая решает конкретную задачу. Например, можно сгенерировать все возможные программы длиной в один символ, проверить каждую программу – решает ли она

[1] В советской литературе вариант этого подхода известен как метод «Отобразить и проверить» (источник информации, к сожалению, потерян). В настоящее время широко используется термин «Метод грубой силы» (*Brute Force Approach*) для обозначения данного подхода.

эту задачу[1]. Если среди односимвольных программ не найдено программы, решающей задачу, перейти к просмотру программ длиной в два символа, затем в три и т. д. В теории такой подход позволяет найти самую короткую программу, однако на практике перебор занимает недопустимо большое время вычислений (для многих задач больше, чем возраст Вселенной).

Название данный алгоритм получил после высказывания Аллена Ньювэлла, Дж. С. Шоу и Герберта А. Симона в 1958 году: «Вполне уместно предположить, что если посадить обезьян за печатные машинки, можно через некоторое время воспроизвести все книги в известном Британском музее в Лондоне».

Несмотря на всю идеалистичность подхода, для задач небольшой размерности алгоритм вполне пригоден. Для начала, конечно, нет необходимости порождать программы, достаточно порождать варианты значений переменных v_i, затем проверять выполнимость $P(\vec{V})$. Рассмотрим пример задачи.

Пример 4.1. *Разработать программу поиска номеров счастливых билетов, содержащих шесть цифр. Подсчитать их количество.*

```
num(X) :- member(X, [0,1,2,3,4,5,6,7,8,9]).
gen([]).
gen([X|T]) :- num(X), gen(T).

p([A,B,C, D,E,F]) :-
        A + B + C =:= D + E + F.
```

[1] Существует фундаментальная проблема остановки, которая делает такую проверку, в общем случае, невозможной.

```
lucky([A,B,C, D,E,F]) :-
        gen([A,B,C, D,E,F]),
        p([A,B,C, D,E,F]).
```

Программа при помощи предиката `gen/1` порождает идентификаторы билетов. Предикат `p/1` проверяет, является ли билет счастливым. Процедура порождения списка счастливых билетов оформлена в виде предиката `lucky/1` и в комментариях не нуждается. Для запуска программы выполним команду:

```
?- lucky(L).

L = [0,0,0,0,0,0] ? ;
L = [0,0,1,0,0,1] ? ;
L = [0,0,1,0,1,0] ? ;
L = [0,0,1,1,0,0] ?

yes.
```

Для подсчета количества счастливых билетов создадим еще одно вспомогательное правило:

```
count(N) :- findall(Ticket, lucky(Ticket),
    Tickets), length(Tickets, N).
```

Данное правило позволяет подсчитывать количество счастливых билетов, но не выводить их полный список на экран.

Выполним запрос (GNU-Prolog) [13]:

```
?- count(N).
N = 55252

(630 ms) yes.
```

Приведенные программы являются также примерами использования стандартных `member/2` и `length/2` предикатов обработки списков.

Сужение области поиска. Программа перебирает 10^6 вариантов, из которых только около $5,5 \cdot 10^4$ относятся к решению задачи. То есть примерно один из двадцати билетов – счастливый. Возникает вопрос: можно ли усовершенствовать программу, чтобы уменьшить количество неправильных вариантов и сэкономить время решения задачи на проверке этих неправильных вариантов?

Первым делом давайте попробуем вычислить значение переменной `F is A+B+C-D-E`. Добавим к программе следующий код:

```
lucky2([A,B,C, D,E,F]) :-
        gen([A,B,C, D,E]),
        F is A+B+C-D-E,
        num(F),
        p([A,B,C, D,E,F]).

count2(N) :- findall(Ticket, lucky2(Ticket),
        Tickets), length(Tickets, N).
```

```
?- count2(N).
N = 55252

(133 ms) yes.
```

Получено такое же количество решений, но за время, в пять раз меньшее. Вычисленное значение `F` может быть отрицательным и больше 9, что противоречит условиям задачи, поэтому

в новую процедуру порождения билетов необходимо добавить дополнительную проверку `num(F)`, которая выполняется, если `F` находится в требуемом диапазоне. Теперь порождается в 10 раз меньше билетов, даже с учетом тех, где `F` находится вне диапазона. То есть каждый второй сгенерированный билет – счастливый. Если убрать уже ненужную повторную проверку `p/1`, то скорость исполнения программы увеличится еще на 30 % до 106 микросекунд, т. е. уже более чем в 6 раз быстрее первоначальной:

```
?- count2(N).
N = 55252

(103 ms) yes.
```

Дополнительное ускорение. Теперь попробуем найти два последних числа. Выражение `A+B+C-D` изменяется в пределах -9, $-8, \ldots, 0, 1, \ldots, 26, 27$: от `0+0+0-9` до `9+9+9-0`. Варианты, когда результат выражения – отрицательный, заведомо неподходящие, так же как если этот результат больше 18, `9+9+9-9`. Можно еще усовершенствовать алгоритм, но оставим это в качестве упражнения. Теперь надо разработать подпрограммы, которые для диапазона $0, 1, \ldots, 18$ будут решать просто отдельную переборную задачу: задано число `S`$\in 0, 1, \ldots, 18$, найти два слагаемых `E` и `F`, дающих в сумме `S`. Дополним программу следующим кодом:

```
lucky3([A,B,C, D,E,F]) :-
        gen([A,B,C, D]),
        S is A+B+C-D,
        S >= 0, S=<18,
        gen2(S, E,F).
```

```
count3(N) :- findall(Ticket, lucky3(Ticket),
        Tickets), length(Tickets, N).

gen2(0,0,0):-!.  % Выделим отдельно наглядные
gen2(18,9,9):-!. % тривиальные варианты.
gen2(N,A,B):-N<10, !, igen(N,A), B is N - A.
gen2(N,A,B):-D is N - 9, Z is 9 - D,
        igen(Z, A1), A is A1 + D, B is N - A.

% igen(N, A) для A порождает последовательности
% 0,1,2,...,N
igen(N, A) :- N>=1, M is N - 1, igen(M, A).
igen(N, N).
```

Запускаем запрос:

```
?- count3(N).
N = 55252

(47 ms) yes.
```

Теперь программа работает в 13 с лишним раз быстрее первоначальной и в два раза быстрее предыдущей, т. е. примерно один из трех билетов не является счастливым. Конечно, программу можно совершенствовать дальше: перейти к порождению первых трех цифр и, отталкиваясь от их суммы, по аналогии с последним примером порождать соответствующие последовательности. Однако необходимо заметить, что программа[1] постепенно становится сложной, а текст значительно хуже воспринимаемым.

[1] Автор пособия не ставил целью найти самую эффективную и короткую программу для решения этой задачи. Задача – продемонстрировать ход рассуждений.

5. Компьютерная алгебра

Старшее поколение преподавателей недовольно уровнем знаний студентов вузов в области математического анализа, особенно в части знаний оператора дифференцирования. Студентам не разрешают пользоваться популярными системами компьютерной алгебры. По-видимому, сказывается недостаточная практика в этом направлении. Действительно, прежде чем *автоматизировать* некоторую *творческую деятельность*, необходимо детально в ней разобраться, и, самое главное, разбираться в случае необходимости.

Давайте продвинемся в решении двух проблем сразу – повторим дифференцирование и разработаем ядро своей компьютерной алгебры. В качестве побочного продукта получим навыки *обработки символьной информации* – решения важного класса задач ИИ.

5.1. Символьное дифференцирование

Частную производную функции $f(x, y, ...)$ по переменной x обозначают как

$$f'_x,\ f_x,\ \frac{\partial}{\partial x}f,\ \frac{\partial f}{\partial x},\ \text{или}\ \frac{d_x f}{dx}.$$

В языке Prolog обозначим частную производную предикатом `d(F,X,DF)`, где `F` – функция, производная `DF` которой «берется» по переменной `X`. Программу нахождения производных из функций (выражений) начнем писать с самых простых вариантов: производной переменной, константы и атома.

```
d(Y,X,1):-var(X),var(Y),
    Y==X,!.
d(Y,X,0):-
    var(Y),var(X),!.
d(C,_,0):-atomic(C),!.
```

Предикат «==/2» позволяет определять, не с одной ли той же переменной унифицированы X и Y. Предикат var/1 проверяет, является ли аргумент все еще переменной или нет. Выполним тестовые запросы:

```
?- d(X,X,D).     |    ?- X=Y,d(Y,X,D).
D = 1.           |    X = Y,
                 |    D = 1.
?- d(Y,X,D).     |
Y = A,           |    ?- d(1,X,D).
X = B,           |    D = 0.
D = 0.           |
                 |
?- d(a,X,D).     |
D = 0.           |
```

Следующий этап – реализация правила преобразования арифметических операций:

```
d(U+V,X,DU+DV):-!,    |   d(U*V,X,DU*V+DV*U):-!,
        d(U,X,DU),    |           d(U,X,DU),
        d(V,X,DV).    |           d(V,X,DV).
                      |
d(U-V,X,DU-DV):-!,    |   d(U/V,X,(DU*V-DV*U)/(V^2)):-!,
        d(U,X,DU),    |           d(U,X,DU),
        d(V,X,DV).    |           d(V,X,DV).
```

Теперь рассмотрим суперпозицию функций, формула является рекурсивной (цепной):

$$\frac{\partial}{\partial x} f(g(x)) = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}.$$

Соответствующее правило Prolog будет следующим:

```
d(E,X, DExpF*DExp):-
    E=..[Atom, Exp],      % (1)
    atom(Atom),!,         % (2)
    d(Exp,X,DExp),        % (3)
    df(Atom, Exp, DExpF). % (4)
```

Сначала надо удостовериться, что `E` – это функция. Для этого попробуем представить `E` в виде списка `[Atom, Exp]` (1,2), где `Atom` – это название функции, а `Exp` – выражение (аргумент). Затем в строке (3) производим построение производной из аргумента, а в (4) – обращение к набору правил соответствия функций выражениям-производным:

```
df(sin,E, cos(E)).
df(cos,E, -sin(E)).
df(ln,E, 1/E).
df(exp,E, exp(E)).
```

Предикат `df(Fun,Exp, DExp)` переводит функцию `Fun` и ее аргумент `Exp` в выражение, представляющее производную `DExp`. В этот список необходимо самостоятельно добавить другие известные функции.

Перейдем к дифференцированию конкретных выражений. Производная степенной функции, например, реализуется так:

```
d(E^N,X, N*E^(N1)*DE):-!,
        d(E,X,DE),
        N1 is N-1.
```

Проверим программу на нескольких выражениях:

```
?- d(X^2, X, D).
D = 2*X^ (2-1)*1.

?- d(sin(X^2), X, D).
D = cos(X^2)* (2*X^ (2-1)*1).

?- d(sin(cos(X^2))*ln(X+Y), X, D).
D = cos(cos(X^2))* (-sin(X^2)*
    (2*X^ (2-1)*1))*ln(X+Y)+1/
    (X+Y)* (1+0)*sin(cos(X^2)).
```

Результат требует дальнейшего совершенствования – сокращения выражений. Для этого реализуем два набора правил: фасадный[1] предикат `sim/2`, предназначенный для сокращения выражений, и предикат `r/2`, представляющий собой элементарные упрощающие преобразования. Если элементарное преобразование удалось, то `r/2` истинен, иначе он терпит неудачу.

```
sim(X, X):-     % Переменная не сокращается.
    var(X),!.
sim(E, R):-     % Сделать один шаг упрощения.
    r(E,E1),!, % Если удалось, то
    sim(E1,R). % сократить результат еще.
sim(E,E).       % Выражение не сокращается.
```

[1] Фасадный предикат предназначен для запуска пользователем. За фасадом здания скрывается вся его сложная конструкция, доступная для понимания только специалистам.

Реализация `sim/2` сильно походит на сортировку списка методом «пузырька». Теперь рассмотрим элементарные преобразования:

```
r(A+B, A):-B=@=0,!.  % A+0 → A
r(B+A, A):-B=@=0,!.  % 0+A → A
r(A*B, A):-B=@=1,!.  % A*1 → A
r(B*A, A):-B=@=1,!.  % 1*A → A
r(_*B, 0):-B=@=0,!.  % A*0 → 0
r(B*_, 0):-B=@=0,!.  % 0*A → 0
r(A^B, A):-B=@=1,!.  % A^1 → A
r(A^B, 1):-B=@=0,A\=@=0,!.  % (1)
r(A/B, A):-B=@=1,!.  % A/1 → A
r(B/A, A^(-1)):-B=@=1,!.    % (2)
```

В строке (1) рассматривается случай $a^0 = a$, при этом исключается неопределенность $0^0 = \frac{0}{0}$. В строке (2) $\frac{1}{x}$ заменяется на x^{-1}. В перечисленных правилах используется специальный вид унификации `=@=/2`, который проверяет структурную эквивалентность аргументов. Представим несколько примеров[1] [16]:

```
1    a =@= A     false
2    A =@= B     true
3    x(A,A) =@= x(B,C)  false
4    x(A,A) =@= x(B,B)  true
5    x(A,A) =@= x(A,B)  false
6    x(A,B) =@= x(C,D)  true
```

То есть в сокращаемых выражениях нам важно распознавать правильно, где переменная, а где константа во входных пара-

[1] Специальные варианты унификаций SWI-Prolog рассматриваются на странице `http://www.swi-prolog.org/pldoc/man?section=unifyspecial`.

метрах. Реализация первого правила `r/2` в виде `r(A+0, A):-!` приводит к неправильному ответу на запрос[1]:

```
?- r(2+X, R).
X = 0,
R = 2.
```

Автоматическая унификация Prolog по умолчанию при сопоставлении двух выражений подставляет переменные, символы, числа и другие термы вместо переменных в правое и в левое выражения так, чтобы они стали одинаковыми. В данном примере овеществляется унификация `2+X=A+0`. Если вместо `A` подставить `2`, а вместо `X` – `0`, то получим выражение `2+0=2+0`. В нашем редукторе необходимо сокращаемые переменные сохранить как переменные Prolog, чтобы потом можно было вычислять значения производных в точке:

```
r(A+B, AB):-number(A),number(B),!,AB is A+B.
r(A*B, AB):-number(A),number(B),!,AB is A*B.
r(A+B, B+A):-number(B),!.
r(A*B, B*A):-number(B),!.

r(A+B+C,AB+C):-number(A),number(B),!, AB is A+B.
r(A*B*C,AB*C):-number(A),number(B),!, AB is A*B.
r(A+D, A+B+C):- \+ D=@=_, D=(B+C),!.
r(A*D, A*B*C):- \+ D=@=_, D=(B*C),!.
```

Эти правила предназначены для переупорядочения слагаемых и сомножителей таким образом, чтобы можно было сократить числовые выражения. Два последних правила раскрывают

[1] Последовательность слагаемых в запросе имеет принципиальное значение.

«ненужные скобки», перестраивая древовидное представление формулы. Набор правил не полон, есть возможность их пополнить, в частности можно алгоритмизировать операции вычитания и деления. На самом деле цепочки последовательных операций сложения и вычитания, умножения и деления следует представить в виде списков слагаемых и сомножителей. Затем надо производить упорядочение элементов этого списка согласно правилам представления полиномов, поиск и сокращение однородных членов. Оставим эту задачу как упражнение.

Следующая группа правил пытается просто вычислить выражение, если такое возможно. Если не получается вычислить, то нужно попытаться сократить аргументы:

```
r(E,R):-
    compound(E),
    ground(E),!,
    R is E.

r(E,R):-
    compound(E),
    E=..[F|Args],!,
    r(Args,SArgs),
    R=..[F|SArgs],
    E\=@=R.

r([],[]):-!.
r([X|T],[SX|ST]):-!,
    sim(X,SX),
    r(T,ST).
```

Предикат `compound/1` проверяет, является ли его аргумент сложным выражением, а `ground/1` – выражением, не содержащим свободных переменных. Первое правило, вообще говоря, реализовано некорректно: оно будет порождать исключительную ситуацию, если в выражении встретится символ или список, то есть что-то, над чем невозможно выполнить арифметическую операцию.

Второе правило преобразует структуру к списку, аналогично тому, как мы делали с производной суперпозиции функции. Затем производится сокращение выражений в аргументах функции. В конце выражение собирается из отдельных компонент в результат сокращения. Если получилась структура, отличная от исходной, то правило завершается удачно.

Последние два правила рекурсивно обрабатывают список аргументов. Теперь рассмотрим пример использования компьютерной алгебры в программе расчета оптимального управления.

5.2. Оптимальное управление

Разработка программ численного решения дифференциальных уравнений, расчеты оптимального управления связаны с необходимостью аналитических вычислений формул производных различных функций и их реализации в виде программного кода. В принципе, вместо реализации этого этапа разработки программ можно использовать численное дифференцирование и ограничиться только реализацией исходных функций. Но, как практически любой численный метод, численное дифференци-

рование а) реализуется как итеративная процедура, которая затрачивает процессорное время на вычисление значения функции в нескольких точках; б) вносит дополнительную погрешность вычисления, которая заложена уже в самом методе. Использование аналитического вычисления производных в целом избавляет программный код от этих недостатков.

В [10] рассмотрен метод вычисления оптимального управления, названного в честь его автора, академика Льва Семеновича Понтрягина, «Принцип максимума Л. С. Понтрягина». В книге вводится обозначение

$$\boldsymbol{\psi}^T\boldsymbol{x} = \psi_i x^i = \psi_1 x^1 + \psi_2 x^2 + \ldots + \psi_n x^n, \quad i = 1, 2, \ldots, n;$$

$$\boldsymbol{\psi} = \boldsymbol{\psi}(t) = \begin{pmatrix} \psi_1, \\ \psi_2, \\ \vdots \\ \psi_n \end{pmatrix}, \qquad \boldsymbol{x} = \boldsymbol{x}(t) = \begin{pmatrix} x_1, \\ x_2, \\ \vdots \\ x_n \end{pmatrix}.$$

В этом обозначении x^1 – i-я фазовая переменная, $\boldsymbol{\psi}^T\boldsymbol{x}$ – матричное произведение двух векторов, $\psi_i x^i$ – скалярное произведение двух векторов. Фазовые переменные описывают состояние объекта[1] (рис. 5.1) в пространстве, одна переменная – одна координата, составляющая вектора скорости и т. п. Фазовые переменные x^i – это вещественнозначные функции ($x^i \in \mathbb{R}$) времени $x^i(t)$. Верхний индекс в именах переменных и функций используется здесь и далее ввиду того, что производные функций по переменным будут указываться в нижнем индексе. Переменные $\psi_i = \psi_i(t)$ – обобщение импульса, задающее влияние фазовой переменной $x^i(t)$ на управление в определенные моменты времени. Вектор

[1] Самолета, корабля, автомобиля, природного ресурса и т. д.

переменных $\boldsymbol{x} = \boldsymbol{x}(t)$ называется *фазовым вектором* объекта (рис. 5.1), а вектора, аналогичные $\boldsymbol{x}(t)$, – *вектор-функциями*.

В фазовый вектор объекта входят его координаты и вектор скорости, а также другие физические величины, характеризующие состояние этого объекта в заданный момент времени $t \in [t_o, t_1]$, где t_0 – начальный момент времени, а t_1 – конечный. На рисунке 5.1 изображен самолет и приведен его упрощенный фазовый вектор. Переменные $x^1, x^2, \ldots, x^6$ задают координаты самолета и величины скоростей по осям, x^7, x^8 – угол рыскания и скорость его изменения. Далее, аналогично x^9, x^{10} – угол горизонта, x^{11}, x^{12} – угол крена, x^{13} – величина подъемной силы. Переменная x^{14} задает величину силы тяжести, x^{15} – величина сил, тормозящих самолет.

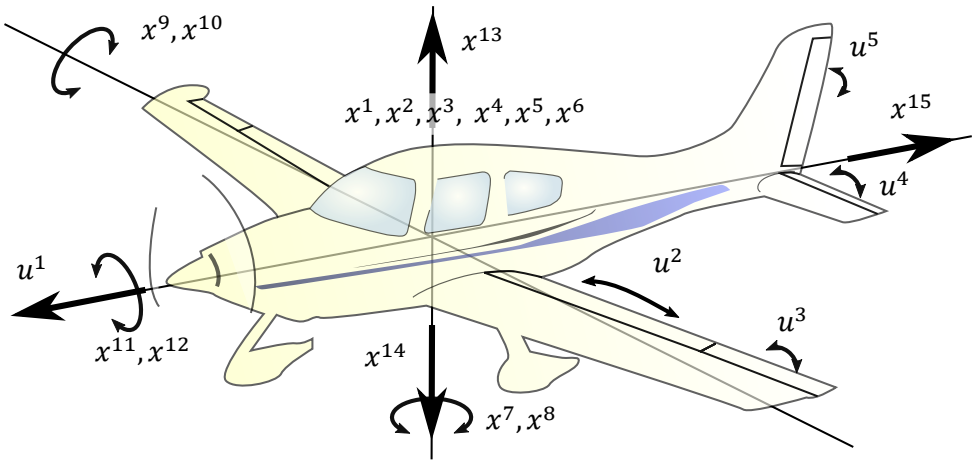


Рис. 5.1. *Фазовый вектор самолета*

Изменение переменных $\dot{x}^i(t)$ во времени в общем случае задается системой дифференциальных уравнений в частных производных:

$$\dot{x}^i = \frac{dx^i}{dt} = f^i(x^1, x^2, \ldots, x^n) = f^i(\boldsymbol{x}); \quad \dot{\boldsymbol{x}} = \frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}(\boldsymbol{x}), \qquad (5.1)$$

где вторая форма записи – векторная, причем для краткости параметр t опущен. Уравнения задают, например, влияние геометрии оперения и силы тяги на вектор скорости самолета, влияние вектора скорости и силы тяжести на координаты в трехмерном пространстве и т. п. Дифференциальные связи второго порядка, например влияние координаты объекта на величину силы тяжести искусственного спутника Земли, также задаются при помощи уравнений (5.1).

Задав $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$ в момент времени t_0 и решив уравнение (5.1), можно рассчитать траекторию изменения объекта (перемещение его в пространстве, например), т. е. решить задачу Коши. Теперь обобщим уравнение (5.1), преобразуем его в следующий вид:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}), \tag{5.2}$$

где $\boldsymbol{u} = \boldsymbol{u}(t)$ – *управление*, которое также зависит от t. Предполагается, что $\boldsymbol{f}$ непрерывна по совокупности всех переменных и имеет непрерывные производные по каждому x^i и u^i.

Как только в параметры $\boldsymbol{f}$ добавили $\boldsymbol{u}$, решение дифференциального уравнения (5.2) стало опять многозначным, т. е. траектория зависит теперь не только от $\boldsymbol{x}_0$, но и от того, как объектом управляют $\boldsymbol{u} = \boldsymbol{u}(t)$. На рисунке 5.1 в вектор управления $\boldsymbol{u}$ входят величина силы тяги u^1, а также $u^1, u^2, \ldots, u^5$, обозначающие величины сил, вызванных соответствующей геометрией крыла и хвостового оперения: закрылков, элеронов, рулей высоты и направления. Геометрия крыла и хвостового оперения изменяется системой управления, например пилотом. Далее будем предполагать, что самолетом управляет техническое устройство, например

компьютер с системой датчиков для получения значений элементов фазового ветра, а также выработанный им вектор $\boldsymbol{u}$ каким-либо образом меняет тягу и геометрию корпуса самолета.

Для того чтобы «выбрать» из бесконечного количества возможных решений одно, необходимо из всех возможных траекторий найти в некотором смысле лучшую. Какая траектория лучше, а какая хуже, определяет функция-критерий, отображающая траектории (или параметры, ее полностью определяющие) на некоторое число из $\mathbb{R}$. В оптимальном управлении такая функция называется *функционалом*.

$$I(\boldsymbol{x},\boldsymbol{u}) = \int_{t_0}^{t_1} f^0(\boldsymbol{x},\boldsymbol{u})dt \to \min. \qquad (5.3)$$

В формуле (5.3) функция $f^0(\boldsymbol{x},\boldsymbol{u})$ вычисляет (формализует) значение критерия оценки качества траектории тела в момент времени $t \in [t_0, t_1]$. Предполагается, что функция f^0 дифференцируема по переменным из $\boldsymbol{x},\boldsymbol{u}$ (так же как и $\boldsymbol{f}$). Значение функции f^0 в конечный момент времени t_1 не зависит от управления, так как в этот момент управлять[1] объектом уже нет смысла. В связи с этим перепишем функционал следующим образом:

$$I(\boldsymbol{x},\boldsymbol{u}) = \int_{t_0}^{t_1} f^0(\boldsymbol{x},\boldsymbol{u})dt + F(t_1,\boldsymbol{x}(t_1)) \to \min. \qquad (5.4)$$

Теперь функция $F(t_1,\boldsymbol{x}(t_1))$ будет обозначать качество траектории в этот последний момент времени t_1, а интегральное выражение – качество основной части траектории, где объектом еще можно управлять.

[1] Переводить управляемый объект в новое состояние «волевым усилием» функции $\boldsymbol{u}$.

Принцип максимума определяется через функцию Гамильтона (гамильтониан), которая имеет следующий вид:

$$H(t, \boldsymbol{x}, \boldsymbol{\psi}, \boldsymbol{u}) = f^0(\boldsymbol{x}, \boldsymbol{u}) + \boldsymbol{\psi}^T \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}). \quad (5.5)$$

Управление $\boldsymbol{u}$ будет оптимальным $\boldsymbol{u}^*$ с соответствующей оптимальной траекторией $\boldsymbol{x}^*$, если выполняется условие

$$H(t, \boldsymbol{x}^*, \boldsymbol{\psi}, \boldsymbol{u}^*) \geqslant H(t, \boldsymbol{x}^*, \boldsymbol{\psi}, \boldsymbol{u}) \quad (5.6)$$

для любого допустимого $\boldsymbol{u}$. Если привести (5.2)–(5.6) в одну систему и переписать все уравнения через H, то получим следующую задачу оптимального управления [15]:

$$\begin{cases} \dot{\boldsymbol{x}} = \boldsymbol{H}_{\boldsymbol{\psi}}, & \boldsymbol{x}(t_0) = \boldsymbol{x}_0, \\ \dot{\boldsymbol{\psi}} = -\boldsymbol{H}_{\boldsymbol{x}}, & \boldsymbol{\psi}(t_1) = F(t_1, \boldsymbol{x}(t_1)). \end{cases} \quad (5.7)$$

То есть $\boldsymbol{x}$ определяется своим левым $\boldsymbol{x}(t_0)$ концом, а $\boldsymbol{\psi}$ – правым $\boldsymbol{\psi}(t_1)$. Иногда $\boldsymbol{\psi}$ называют также функцией, сопряженной $\boldsymbol{x}$.

5.2.1. Дискретный вариант задачи улучшения управления

Использование численных расчетов предполагает постановку задачи в дискретном времени. Для этого разобьем интервал $[t_0, t_1]$ на N равных частей с шагом $h = (t_1 - t_0)/N$, $N \in \mathbb{N}$. Дискретные моменты времени можно обозначать t_0, $t_0 + h$, $t_0 + 2h$,…, $t_1 - h$, t_1 или, как будет далее использовано в пособии, t_0 t_1, t_2,…t_j,…t_N. Функцию $\dot{\boldsymbol{x}}(t)$ в левой части (5.2) заменим на $\boldsymbol{x}(t_{j+1})$ или еще короче $\boldsymbol{x}_{j+1}$ (так же как и остальные переменные) и получим дискретный вариант уравнения движения следующего вида:

$$\boldsymbol{x}_{j+1} = \boldsymbol{g}_j = \boldsymbol{g}(t_j, \boldsymbol{x}_j, \boldsymbol{u}_j), \quad j \in \{0, 1, \ldots, N\}; \quad (5.8)$$

$$\boldsymbol{x}(t_0) = \boldsymbol{x}_0, \quad \boldsymbol{x}_j \in \mathbb{R}^n, \quad \boldsymbol{u}_j \in \mathbb{R}^m. \quad (5.9)$$

Формула (5.8) уже не является дифференциальным уравнением, и преобразование из (5.2) в (5.8) – теперь задача пользователя программы, которую мы разработаем далее. Самый простой способ – применить простую схему интегрирования Эйлера:

$$\boldsymbol{x}_{j+1} = \boldsymbol{x}_j + h\boldsymbol{f}(t_j, \boldsymbol{x}_j, \boldsymbol{u}_j).$$

Далее будем предполагать, что пользователь сам задает (5.8). Качество траекторий оценим следующим функционалом:

$$I(\boldsymbol{x}, \boldsymbol{u}) = \sum_{j=0}^{N-1} f^0(t_j, \boldsymbol{x}_J, \boldsymbol{u}_j) + F(t_N, \boldsymbol{x}_N) \to \min. \qquad (5.10)$$

Принцип максимума в данном случае определяется при помощи дискретного варианта функции Гамильтона

$$H_j = H(t_j, \boldsymbol{x}_j, \boldsymbol{u}_j) = f^0(t_j, \boldsymbol{x}_j, \boldsymbol{u}_j) + \boldsymbol{\psi}^T(t_{j+1})\boldsymbol{g}(t_j, \boldsymbol{x}_j, \boldsymbol{u}_j). \qquad (5.11)$$

Выпишем теперь принцип максимума

$$H(t_j, \boldsymbol{x}_j^*, \boldsymbol{\psi}_{j+1}, \boldsymbol{u}_j^*) \geqslant H(t_j, \boldsymbol{x}_j^*, \boldsymbol{\psi}_{j+1}, \boldsymbol{u}_j) \qquad (5.12)$$

для всех допустимых $\boldsymbol{u}_j$. Сопряженная вектор-функция $\boldsymbol{\psi}_j$ вычисляется по схеме

$$\begin{aligned} \boldsymbol{\psi}_N^T &= F_{\boldsymbol{x}}(t_N, \boldsymbol{x}_N), \\ \boldsymbol{\psi}_j^T &= \boldsymbol{\psi}_{j+1}^T + \boldsymbol{H}_{\boldsymbol{x}}(t_j, \boldsymbol{x}_j^*, \boldsymbol{\psi}_{j+1}^T, \boldsymbol{u}_j^*). \end{aligned} \qquad (5.13)$$

Условиям (5.5) и (5.12) соответствует минимум $I(\boldsymbol{x}, \boldsymbol{u})$. Поиск этого минимума с использованием численного метода основывается на вычислении градиента $I'(\boldsymbol{u})$ (по переменным $\boldsymbol{u}$):

$$I'(\boldsymbol{u}) = \boldsymbol{H}_{\boldsymbol{u}}(t, \boldsymbol{x}, \boldsymbol{\psi}, \boldsymbol{u}). \qquad (5.14)$$

5.2.2. Схема алгоритма улучшения первого порядка

Теперь мы знаем, как вычислять градиент. Используя эти знания, построим алгоритм улучшения $I(\boldsymbol{x}, \boldsymbol{u})$. Обозначим через $\langle \boldsymbol{x}^I(i), \boldsymbol{u}^I(i) \rangle$ некоторую допустимую траекторию и управление на всем отрезке времени $[t_0, t_1]$. Цель шага улучшения – найти такую траекторию $\langle \boldsymbol{x}^{II}(i), \boldsymbol{u}^{II}(i) \rangle$, чтобы $I(\boldsymbol{x}^{II}, \boldsymbol{u}^{II}) < I(\boldsymbol{x}^I, \boldsymbol{u}^I)$. Общая схема улучшения выглядит следующим образом:

1. Задается начальное управление $\boldsymbol{u}^I(t)$. Из уравнения (5.8) и условий (5.9) определяется $\boldsymbol{x}^I(t)$. Вычисляется $I(\boldsymbol{x}^I, \boldsymbol{u}^I)$.

2. Из системы $\boldsymbol{\psi} = \boldsymbol{H}_{\boldsymbol{x}},\ \boldsymbol{\psi}_N = -\boldsymbol{F}_{\boldsymbol{x}}(t_N, \boldsymbol{x}_N)$ находим $\boldsymbol{\psi}$, где $\boldsymbol{H}(t_j, \boldsymbol{x}_j, \boldsymbol{\psi}_{j+1}, \boldsymbol{u}_j) = \boldsymbol{\psi}_{j+1}^T \boldsymbol{g}(t_j, \boldsymbol{x}_j, \boldsymbol{u}_j) - f^0(t_j, \boldsymbol{x}_j, \boldsymbol{u}_j)$, производная $\boldsymbol{H}_{\boldsymbol{u}}$ находится в точке $\left(t, \boldsymbol{x}_j^I, \boldsymbol{\psi}_{j+1}, \boldsymbol{u}_j^I\right)$.

$$\boldsymbol{H}_{\boldsymbol{u}}(t_j, \boldsymbol{x}_j, \boldsymbol{\psi}_{j+1}, \boldsymbol{u}_j) = \boldsymbol{g}_{\boldsymbol{u}}(t_j, \boldsymbol{x}_j, \boldsymbol{u}_j)\boldsymbol{\psi}_{j+1} - f_{\boldsymbol{u}}^0(t_j, \boldsymbol{x}_j, \boldsymbol{u}_j)$$

   из (5.9). Задается параметр α.

3. Из системы $\boldsymbol{x}_{j+1} = \boldsymbol{g}(t_j, \boldsymbol{x}_j, \boldsymbol{u}_j^{II}),\ \boldsymbol{x}(t_0) = \boldsymbol{x}_0$, где $\boldsymbol{u}^{II} = \boldsymbol{u}^I + \alpha \boldsymbol{H}_{\boldsymbol{u}}$, вычисляется $\boldsymbol{x}^{II}$.

4. Новое управление и значение параметра α подсчитываются из решения задачи одномерной минимизации для функционала $I(\boldsymbol{x}^{II}, \boldsymbol{u}^{II}) \to \min\limits_{\alpha}$.

5. Если $I\left(\boldsymbol{x}^{II}, \boldsymbol{u}^{II}\right) \geqslant I\left(\boldsymbol{x}^I, \boldsymbol{u}^I\right)$ (улучшение не произошло), то уменьшаем α и переходим к следующей итерации, начиная с пункта 3.

6. Иначе, если $I\left(\boldsymbol{x}^{I}, \boldsymbol{u}^{I}\right) - I\left(\boldsymbol{x}^{II}, \boldsymbol{u}^{II}\right) > \varepsilon$, то переходим к следующей итерации, начиная с пункта 2. Значение ε – параметр точности.

Шестой шаг алгоритма не относится непосредственно к схеме улучшения, но он логичен, так как создает итеративный процесс поиска минимума $I(\boldsymbol{x}, \boldsymbol{u})$, т. е. функции $\boldsymbol{u}$ оптимального управления объектом.

5.2.3. Реализация программы

Алгоритм реализуем на простом и компактном языке программирования Python [5, 9, 17]. Язык является объектноориентированным, его дистрибутив содержит огромное количество библиотек. Кроме того, в Интернете находится еще множество других библиотек и приложений, в том числе библиотеки `numpy` и `sympy`. Библиотека `numpy` включает в себя операции с векторами данных и матрицами. При их помощи будем задавать фазовый вектор $\boldsymbol{x}$ в виде переменной-массива `X` (`X[j]`=$\boldsymbol{x}_j$), траектории, моменты времени, функцию управления и т. п. В библиотеке `sympy` содержатся операции вычисления производных из выражений, функции преобразования и сокращения выражений, а также процедуры преобразования выражений в байт-код виртуальной машины Python (компиляции выражений в программный код).

Программу начнем реализовывать с импорта `numpy` и `sympy`, библиотеки специальных структур-итераторов `itertools`, системной библиотеки `os`.

```
import math
import numpy, sympy
import itertools
from sympy import symbols, diff, Symbol
import numpy.linalg
from sympy.utilities.lambdify import lambdify
import os

TupleType=type((1,))
ListType=type([])
```

Сначала запрограммируем абстрактный класс, представляющий модель управляемого объекта. В классе необходимо представить $\boldsymbol{x}_0$, определить размерности n и m вектор-функций $\boldsymbol{x}$ и $\boldsymbol{u}$. Вектор `U0` ($\boldsymbol{u}^I$) необходимо задать как начальное приближение синтезируемого управления:

```
class Model(object):

    def __init__(self, X0, U0):
        self.X0=atleast_1d(X0)  # x_0 должен быть вектором
        self.U0=U0
        self.N=self.X0.shape[0] # Размерность x
        self.M=U0.shape[1]      # Размерность u

    def F(self, x):
        return 0.0              # Заглушка по умолчанию

    def g(self, t, x, u):
        raise RuntimeError("метод реализуется в подклассе")

    def f0(self, t, x, u):
        return 0.0              # Заглушка по умолчанию
```

Реализация схемы улучшения – это класс Process, наследующий все свойства класса VFCalc, реализующего компьютерную алгебру над вектор-функциями. Реализацию VFCalc рассмотрим далее, а пока сосредоточимся на схеме улучшения.

Схема улучшения получает в качестве входных данных в конструктор __init__ два параметра: model – класс-потомок класса Model и alpha – параметр α. Для вычисления производных VFCalc необходимо знать размерности $\boldsymbol{x}$ и $\boldsymbol{u}$. Конструктор также сохраняет функции $\boldsymbol{f}$, f и F в виде специальных структур-выражений sympy, над которыми можно производить операции символьного дифференцирования:

```
class Process(VFCalc):
    def __init__(self, model, alpha=1.0):
        VFCalc.__init__(self, model.N, model.M)
        t,x,u=self.v.t,self.v.x,self.v.u
        self.model=model; self.alpha=alpha
        self.v.g=model.g(t, x, u)     # Получить выражение g
        self.v.f0=model.f0(t, x, u)   # Получить выражение f^0
        self.v.F=model.F(x)           # Получить выражение F

    def trajectory(self, U):          # Расчет траектории x(t)
        x0=self.model.X0; X = [x0]    # по известному u
        for t, u in enumerate(U):     # (0, u0), (1, u1)
            xn=self.model.g(t, X[t], u); X.append(xn)
        return array(X)

    def I(self, X, U):                # Вычисление I(x,u)
        def _a(acc, t):
            return acc + self.model.f0(t, X[t], U[t])
        return reduce(_a, range(len(X)-1), self.model.F(X[-1]))
```

```
def optimize(self, t, eps=0.001, iters=1000): # Схема
    Up=self.model.U0              # (x^I, u^I)
    Xp=self.trajectory(Up)
    Ip=self.I(Xp,Up); it = 1      # I(x^I, u^I)
    while True:
        alpha = self.alpha
        Psi=self.Psi(t, Xp, Up)   # ψ(t)
        _H_u=self.H((self.v.u,), t[:-1], Xp[:-1], Up, Psi)
        while True:
            _dU=_H_u*alpha        # H_u α
            Un = Up + _dU
            Xn = self.trajectory(Un) # (x^II, u^II)
            In = self.I(Xn, Un)   # I(x^II, u^II)
            dI = Ip-In
            if abs(dI)<eps:       # Решение найдено
                return In, Xn, Un, it, "opt"
            if iters<=0:          # Решение не найдено
                return In, Xn, Un, it, "nonoptimal"
            iters-=1; it+=1
            if In>=Ip:
                alpha/=2          # Новый параметр
                continue          # шага улучшения
            else:
                Xp, Up, Ip = Xn, Un, In
                break
```

В предыдущем отрезке программного кода переменные `X`, `U` и их аналоги – это массивы значений $\boldsymbol{x}(t)$ и $\boldsymbol{u}(t)$ для каждого момента времени t на интервале $[t_0, t_1]$.

Вычисления $\boldsymbol{\psi}$ и нужной производной $\boldsymbol{H}$ реализуются при помощи специальных методов класса `Process`, которые описаны

далее. Операция `dot(A,B)` выполняет матричное умножение аргументов. Если в качестве параметра этой операции передать два вектора-строки, то второй вектор будет автоматически транспонирован в столбец. Эта особенность реализации в библиотеке `numpy` позволяет тривиальные матричные умножения векторов делать без дополнительной явной операции транспонирования. В результате операции `X[:-1]` возвращается последний «-1» элемент `X`, т. е. $\boldsymbol{x}(t_1)$:

```
    def Psi(self, t, X, U):
        v=self.v
        psie = -self.fun(v.F,(v.x,), t[-1], X[-1], U[-1])
        psi=[psie]; X=X[:-1]; t=t[:-1]
        _f0_x=self.fun(v.f0, (v.x,), t, X, U) # f_x^0
        _g_x =self.fun(v.g, (v.x,), t, X, U)  # g_x
        j=len(t)-1; p=psie   # Начать с конца интервала
        while j>=1:
            i=t[j]; pp=p
            pn = dot(pp, _g_x[i]) - _f0_x[i]
            psi.append(pn); p=pn; j-=1
        psi=array(psi)
        return psi[::-1]     # Переставить в обратном порядке

    def H(self, vars, T, X, U, Psi):
        g=self.fun(self.v.g, vars, T, X, U)
        f0=-self.fun(self.v.f0, vars, T, X, U)
        H = alpha * f0
        for psi,_H,_g,i in zip(Psi, H, g, range(len(H))):
            _H += dot(psi,_g); H[i]=_H
        return H
```

В приведенном отрезке кода в H передается список векторов переменных, по которым вычисляются производные $\boldsymbol{H}$, значения $\boldsymbol{x}$ и $\boldsymbol{u}$ на всем интервале времени, а также сам интервал времени и параметр α.

Функции $\boldsymbol{g}$, f^0 и F, задаваемые пользователем нашей библиотеки решения задач оптимального управления, в программе могут получить значения как в определенный момент времени (например, X[:-1]), так и на всем интервале (X). Кроме того, формулы производных от g^i могут «вырождаться» в константы, т. е. g может выдать не вектор значений, когда это надо, а просто одно числовое значение. Для того чтобы меньше зависеть от этих случаев, необходимо реализовать специальную функцию, которая возвращала бы адекватный по типу результат типу переданных ей параметров. А вот внутри функции должно вычисляться значение нужной нам производной нужной нам функции. Такой функцией в Process является метод fun.

```
def fun(self, f, vars, T, X, U):
    code,df=self.code(f, *vars) # Найти производную
                                # и скомпилировать ее.
    X=numpy.atleast_1d(X)       # Входной параметры X и U
    U=numpy.atleast_1d(U)       # должны быть векторами.
    if X.ndim>1:
        Xs=[X[:,i:i+1] for i in range(X.shape[1])]
        Us=[U[:,i:i+1] for i in range(U.shape[1])]
        args=(T,)+tuple(Xs+Us)
    else:
        args=(T,)+tuple(X)+tuple(U)
    rc=code(*args)  # Вычислить функцию (производную)
    rct=type(rc)
```

```
    rc=numpy.atleast_1d(rc)
    if type(T)==numpy.ndarray:
        try: # Попробовать транспонировать список
            if rct in [TupleType,ListType]:
                rc=rc.reshape(rc.shape[:-1])
                rc=rc.T
                return rc
        except ValueError: pass # ... не получилось
        if T.shape[0]!=rc.shape[0]: # Константа → вектор
            nrc=numpy.zeros((len(T),)+rc.shape,dtype=float)
            nrc[:]=rc; rc=nrc
    return rc
```

5.2.4. Дифференцирование вектор-функций

Теперь рассмотрим подсистему вычисления производных вектор-функций. Для этого разработаем специальный класс, экземпляры которого настроены на поддержку нашей программы, реализующей схему улучшения.

Экземпляры прежде всего должны знать, какого размера `N` фазовый вектор $\boldsymbol{x}$ и вектор управления $\boldsymbol{u}$ (`M`). Эти параметры передаются в конструктор класса `VFCalc` при создании экземпляра. Далее порождается ряд символов, специальных структур библиотеки `sympy`, обозначающих идентификаторы в выражениях. При помощи этих символов записываются функции, из которых будут вычисляться производные.

Класс `VFCalc`, кроме сервиса вычисления производных, предоставляет возможности компиляции полученных выражений в байт-код виртуальной машины среды исполнения `Python`. Так

как класс VFCalc является базовым для класса Process, то его определение в тексте программы должно быть помещено перед определением Process.

```
class Helper():
    pass

class VFCalc(object):
    def __init__(self, N,M):
        self.N=N; self.M=M; self.v=Helper()
        self.v.x=[Symbol('x'+str(i+1)) for i in range(self.N)]
        self.v.u=[Symbol('u'+str(i+1)) for i in range(self.M)]
        self.v.t=Symbol('t')  # Переменные для представления
                              #    выражений
    def diff1(self, f, var):  # Шаг вычисления производной
        if type(f) in [TupleType,ListType]:
            df=tuple([self.diff1(fi, var) for fi in f])
        else:
            df=tuple([diff(f, vi) for vi in var])
        if len(df)==1: df=df[0] # Лишние скобки
        return df

    def diff(self, f, *vars): # Производная вектор-функции
        cf=f
        for v in vars:        # Выполнить пошагово
            cf=self.diff1(cf, v)
        return cf

    def subs(self, f, s):     # Подстановка в вектор-функцию
        if type(f) not in [TupleType,ListType]:
            return f.subs(s)
        return tuple([self.subs(fi,s) for fi in f])
```

```
    def lambdify(self, f):     # Компилирование вектор-функции
        l=[self.v.t]           # Формирование списка
        l.extend(self.v.x)     #   параметров
        l.extend(self.v.u)
        fl=lambdify(l, f, "numpy") # Использовать арифметику
        return fl              #  из пакета numpy

    def code(self, f, *vars): # Вычислить производную
        df=self.diff(f, *vars)
        c=self.lambdify(df)   # и скомпилировать ее.
        return c,df           # Возвратить оба результата
```

Тестирование класса произведем при помощи следующего программного кода:

```
d=VFCalc(2,2)
x1,x2=Symbol('x1'),Symbol('x2')
u1,u2=Symbol('u1'),Symbol('u2')
y1=x1**2*u1+x2*u2**2
y2=x1**2*x2**2*u1**2*u2**2
res=(d.diff([y1,y2],
    [x1,x2], [u1,u2]))
pprint (res)
return
```

В приведенном коде x1 и x2 – переменные, формирующие вектор переменных $\boldsymbol{x}$, u1 и u2 – переменные, формирующие вектор переменных $\boldsymbol{u}$, функции y1 и y2 формируют вектор-функцию $\boldsymbol{f}$, из которой берется производная $\boldsymbol{f}_{\boldsymbol{xu}}$. В результате получаем следующую матрицу функций:

```
(((2*x1, 0), (0, 2*u2)),
 ((4*u1*u2**2*x1*x2**2,
```

```
   4*u1**2*u2*x1*x2**2),
  (4*u1*u2**2*x1**2*x2,
   4*u1**2*u2*x1**2*x2)))
```

5.2.5. Тестирование программы

Тестирование программы проведем на простой модели:

$$g_1 = x_1(t_j) + hu_1(t_j), \quad x_1(t_0) = 1.0,$$
$$f^0 = h(x_1^2 + u_1^2), \quad F = 0.0,$$
$$h = 0.01, \quad t_0 = 0.0, \quad t_i = t_0 + h(i-1), \quad t_N = t_1 = 1.0,$$
$$\boldsymbol{u}^I = 0.0.$$

Модель реализуем в виде класса, унаследовав его от `Model`:

```
class LinModel1(Model):
    def __init__(self):
        X0=(1.0,)
        self.h = 0.01
        self.num = int((1.0-0.0) / self.h) # num=N
        self.T = linspace(start=0.0, stop=1.0, num=self.num)
        self.t = arange(len(self.T))  # 0,1,2,3,4,5...
        Model.__init__(self, X0=X0, U0=self.start_control())

    def start_control(self):  # u^I=0.0
        U = [(0.0,) for t in self.t[:-1]]
        return array(U)

    def F(self, x):
        return 0.0
```

```
    def g(self, t, x, u):
        x0=x[0]; u0=u[0]
        return (x0+self.h*u0,)    # Схема Эйлера для ẋ₁ = u₁.

    def f0(self, t, x, u):
        x0=x[0]; u0=u[0]
        return self.h * (x0*x0+u0*u0)
```

Теперь создадим экземпляр задачи, передадим его в качестве параметра в процесс улучшения, который тоже является экземпляром своего класса. Проведем тестовый запуск:

```
def test_1():
    m = LinModel1()           # Модель
    p1=Process(m, alpha=1.0) # Процесс улучшения
    iters=2000    # Максимальное количество итераций
    eps=0.001     # Точность аппроксимации минимума I
    rc=I1, X1, U1, it1, rstr =
        p1.optimize(m.t, eps=eps, iters=iters)
    print (I1, "iters:", it1)

test_1()          # Запуск теста
```

В результате запуска вычислений получим результат следующего вида:

```
0.776664425757 iters: 47.
```

Получается, что за 47 итераций получены управление и траектория, оцениваемая минимальным функционалом со значением 0,777. Траектория и управление изображены на рис. 5.2. На рисунке нижняя линия – это синтезированное оптимальное управление, а верхняя – траектория.

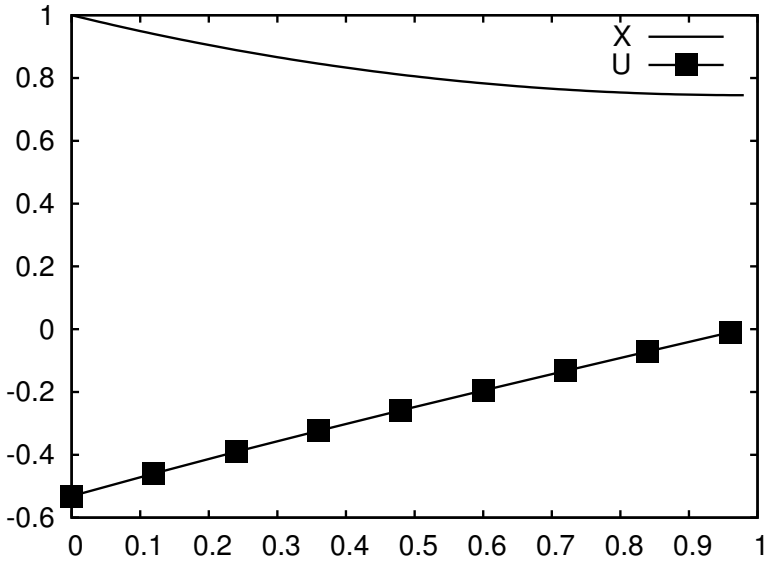


Рис. 5.2. *Результат расчета оптимального управления задачи*

5.2.6. Резюме

Мы рассмотрели практическое приложение процедур вычисления частных производных из функций. На самом деле данные процедуры не являются в полной мере реализациями какого-либо метода искусственного интеллекта, но они демонстрируют, как при помощи синтеза (гибридизации) численного моделирования и символьных вычислений разрабатываются полезные приложения, а также экономится труд математиков и программистов: поскольку нет необходимости в выписывании и программировании процедур вычисления производных вектор-функций.

Существенным недостатком разработанной программы является низкая производительность функции `fun`, так как она вычисляет производные заново при каждом обращении. Сделано это для того, чтобы не усложнять тексты программ в пособии. Проблема решается при помощи реализации механизма кеширования: вычислив производную из какой-либо функции, надо

сохранить ее копию в кеше. Дальнейшим развитием изложенного подхода к разработке программного обеспечения численных методов является порождение программ, реализующих метод на основе анализа или интерпретации Python-программы. Интересным также является разработка библиотеки или программного пакета, при помощи которого можно было бы задавать уравнение движением в непрерывной форме вместе со схемой ее перевода в дискретный вид для численного решения, а по полученной схеме опять же генерировать высокопроизводительную программу, реализация которой учитывает особенности микропроцессора.

Заключение

В учебном пособии приведен ряд задач искусственного интеллекта, решения которых представлены в виде программ на логическом языке программирования Prolog. Представлены как классические, относящиеся к планированию действий, поиску решения с удовлетворением ограничений, так и задачи, апеллирующие к специфике численных методов решения дифференциальных уравнений и поиска оптимального управления движением. Отдельно рассмотрено применение логического языка для организации символьных вычислений. Свойства логического языка программирования позволяют программисту развить навыки логического мышления и серьезное отношение к своей программе не просто как к некоторому набору операторов, а как к строгой логической конструкции. Причем это касается не только языков программирования высокого уровня, таких как Prolog, Refal, и языков функционального программирования, но и языков императивных: C/C++/C#, Pascal, Visual Basic и т. д.

За рамками пособия остались такие интересные темы, как «Эволюционные вычисления», «Решение игровых задач», «Экспертные системы», «Анализ данных» и др.

Надеемся, что задача пособия – рассказать, на каких принципах работают современные системы компьютерной алгебры, выполнена, а студенты-читатели с интересом знакомились с разделами пособия и выполняли задания. Надеемся также, что тематика учебного пособия теперь будет представлять научный интерес для читателей.

Задачи, рассмотренные в пособии, всегда присутствуют в любом производственном процессе, связанном с обработкой информации, но, к сожалению, они остаются незамеченными. Достаточно просто распознать вычислительную задачу, труднее – задачу, связанную с математическим моделированием, и совсем трудно – задачу автоматизации рассуждений и принятия решения. Читателю следует и далее уделять некоторое время развитию навыков программирования систем искусственного интеллекта и анализу бизнес-процессов предприятия с целью выявления и решения этих необычных задач.

Успехов в изучении методов искусственного интеллекта!

http://eugeneai.github.io/ais/

Рекомендуемая литература

1. Братко И. Программирование на языке ПРОЛОГ для искусственного интеллекта : пер. с англ. / И. Братко. – М. : Мир, 1990. – 560 с.: ил.
2. Васильев С. Н. Интеллектное управление динамическими системами / С. Н. Васильев, А. К. Жерлов, Е. А. Федосов, Б. Е. Федунов. – М. : Физматлит, 2000. – 352 с: ил.
3. Искусственный интеллект : в 3 кн. / под ред. Э. В. Попова. – М. : Радио и связь, 1990. – 464 с.: ил.
4. Лорьер. Ж.-Л. Системы искусственного интеллекта : пер. с франц. / Ж.-Л. Лорьер. – М. : Мир, 1991. – 568 с.: ил.
5. Лутц М. Изучаем Python, 4-е издание.: пер. с англ. / М. Лутц. СПб.:Символ-Плюс, 2011. 1280 с., ил.
6. Математический энциклопедический словарь / гл. ред. Ю. В. Прохоров. – М. : Сов. энциклопедия, 1988. – 847 с.
7. Непейвода Н. Н. Основания программирования / Н. Н. Непейвода, И. Н. Скопин. – Москва; Ижевск : Институт компьютерных исследований, 2003 – 880 с.: ил.
8. Непейвода Н. Н. Прикладная логика : учеб. пособие / Н. Н. Непейвода. – 2-е изд. – Новосибирск : Изд-во Новосиб. ун-та, 2000. – 521 с.: ил.
9. Основы программирования на Python. [Электронный ресурс]: сайт. URL:`http://younglinux.info/sites/default/files/python_structured_programming.pdf`. (дата обращения: 11.01.2015).
10. Понтрягин Л. С. Принцип максимума в оптимальном управ-

лении / Л. С. Понтрягин. 2-е изд. М. : Едиториал УРСС, 2004. – 64 с.

11. Рассел С. Искусственный интеллект: современный подход : пер. с англ. / С. Рассел, П. Новриг. 2-е изд. – М. : Изд. дом «Вильямс», 2006. – 1408 с.: ил.
12. Тарифное руководство № 4. Книга 2. Часть 1. Алфавитный список железнодорожных станций. [Электронный ресурс]: сайт. http://mapservis.ru/docs/tar_ruc_4.htm (дата обращения: 06.05.2015).
13. The GNU Prolog web site [Электронный ресурс]: сайт. URL:http://www.gprolog.org/. (дата обращения: 28.11.2013).
14. OpenStreetMap Nominatim: Search. [Электронный ресурс]: сайт. URL:http://wiki.openstreetmap.org/wiki/Nominatim. (дата обращения: 11.01.2015).
15. Sethi S. P., Thomson G. L. Optimal Control Theory: Applications to Management Science and Economics. 2nd Edition. 2005. 506 pp.
16. SWI-Prolog's home [Электронный ресурс]: сайт. URL:http://www.swi-prolog.org/. (дата обращения: 28.11.2013).
17. Welcome to Python.org. [Электронный ресурс]: сайт. URL:https://www.python.org/ (дата обращения: 11.01.2015).