Министерство образования и науки Российской Федерации

САНКТ–ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Приоритетный национальный проект «Образование»
Национальный исследовательский университет

Ф. А. НОВИКОВ

СИСТЕМЫ ПРЕДСТАВЛЕНИЯ ЗНАНИЙ

Учебное пособие

Рекомендовано Учебно-методическим объединением
по университетскому политехническому образованию
в качестве учебного пособия для студентов высших учебных
заведений, обучающихся по направлениям подготовки
магистров «Системный анализ и управление»

Санкт-Петербург
Издательство Политехнического университета
2011

УДК
ББК

Рецензенты:

Заведующий кафедрой «Прикладная математика» физико-механического факультета СПбГПУ канд. техн. наук В.Е. Клавдиев

Директор по разработке ООО «ПРОКДИ» Золотарева Н.Н.

Главный научный сотрудник ИПС им. А.К.Айламазяна РАН, доктор физ.-мат. наук В.Б. Новосельцев

Новиков Ф. А. **Системы представления знаний**: Учеб. пособие. – СПб.: Изд-во Политехн. ун-та, 2010. – 240 с.

ISBN

В пособии излагаются основы теории искусственного интеллекта и методы решения слабо формализованных задач. Основное внимание уделяется методам представления знаний в компьютерах. Рассматривается представление знаний системами продукций и представления знаний формулами исчисления предикатов. Приводятся алгоритмы поиска решения в системах продукций и автоматического доказательства теорем.

Учебное пособие предназначено для студентов вузов, обучающихся по магистерской программе «Математическое и программное обеспечение компьютерных систем» направления подготовки магистров «Системный анализ и управление». Оно может быть также использовано при обучении в системах повышения квалификации и в учреждениях дополнительного профессионального образования».

Работа выполнена в рамках реализации программы развития национального исследовательского университета «Модернизация и развитие политехнического университета как университета нового типа, интегрирующего мультидисциплинарные научные исследования и надотраслевые технологии мирового уровня с целью повышения конкурентоспособности национальной экономики»

Печатается по решению редакционно-издательского совета Санкт-Петербургского государственного политехнического университета.

© Новиков Ф. А., 2011
© Санкт- Петербургский государственный политехнический университет, 2011

ISBN

ОГЛАВЛЕНИЕ

Введение……………………………………………………………......... 6
1. Прикладные системы искусственного интеллекта…………………......... 10
1.1. Обзор приложений искусственного интеллекта……………...... 14
1.1.1. Понимание естественного языка и машинный перевод…………………………………………………………………… 14
1.1.2. Интеллектуальные базы данных и вопрос-ответные системы……………………………………………........................... 25
1.1.3. Экспертные системы и автоматическое доказательство теорем………………………………………………………………….. 27
1.1.4. Автоматическое управление роботом и распознавание образов…………………………………………………………………… 32
1.1.5. Интеллектуальные игры………………………………….. 39
1.2. Место представления знаний в искусственном интеллекте…….. 41
1.2.1. Итеративный характер решения задач…………………... 41
1.2.2. Знание и незнание………………………………………… 43
1.2.3. Алгоритмы поиска решения и представление знаний…….. 47
1.3. История искусственного интеллекта…………………………... 49
1.3.1. Предыстория искусственного интеллекта………………. 51
1.3.2. Пионерские исследования……………………………….... 53
1.3.3. Становление и развитие……………………………….... 57
1.3.4. Научная консолидация и промышленное внедрение……… 59
1.4. Классификация прикладных систем искусственного интеллекта……………………………………………………………... 63
1.4.1. Виды знаний…………………………………………. 63
1.4.2. Классификация по степени использования различных видов знаний…………………………………………………………… 65
1.4.3. Классификация по форме представления знаний…………. 66
1.4.4. Классификация по виду ответа при решении задач……….. 68
1.4.5. Классификация по степени универсальности……………… 70
1.4.6. Классификация по архитектуре и инструментальным средствам………………………………………………………………… 72
Выводы……………………………………………………………….. 73
2. Представление знаний системами продукций………………………… 74
2.1. Системы продукций………………………………………………….. 74
2.1.1. Терминологические соглашения и содержательная интерпретация……………………………………………………………… 75
2.1.2. Структура системы неоднородных продукций……………..... 78
2.1.3. Алгоритм работы системы неоднородных продукций……. 86

2.1.4. Система продукций как логическое исчисление………….. 88
2.1.5. Игра в восемь……………………………………………… 92
2.1.6. Крестьянин, волк, коза и капуста………………………… 94
2.1.7. Ход конем…………………………………………………. 96
2.2. Стратегии управления……………………………………….. 99
2.2.1. Критерии сравнения стратегий…………………………... 101
2.2.2. Безвозвратный поиск…………………………………… 103
2.2.3. Поиск с возвратами……………………………………... 106
2.2.4. Поиск на графе………………………………………….... 109
2.3. Специальные системы продукций……………………………... 116
2.3.1. Обратные системы продукций…………………………… 116
2.3.2. Двусторонние системы продукций……………………… 119
2.3.3. Коммутативные системы продукций…………………..... 121
2.3.4. Разложимые системы продукций………………………… 126
Выводы………………………………………………………………... 131
3. Алгоритмы поиска решения……………………………………..... 132
3.1. Эвристический поиск………………………………………… 135
3.1.1. Общий алгоритм…………………………………………... 135
3.1.2. Эвристические алгоритмы……………………………….. 139
3.1.3. Оценочная функция……………………………………...... 141
3.2. Свойства алгоритма А*……………………………………… 144
3.2.1. Теорема о состоятельности……………………………… 145
3.2.2. Сравнение оценочных функций……………………….... 149
3.2.3. Монотонное ограничение………………………………… 151
3.2.4. Область применимости алгоритма А*………………… 152
3.3. Поиск на графах И/ИЛИ……………………………………... 154
3.3.1. Граф И/ИЛИ…………………………………………….…..... 155
3.3.2. Алгоритм поиска на графе И/ИЛИ……………………… 157
3.3.3. Пример применения процедуры поиска на графе И/ИЛИ… 161
3.4. Поиск на игровых деревьях……………………………………. 164
3.4.1. Игровые деревья…………………………………………... 164
3.4.2. Минимакс…………………………………………………... 166
3.4.3. α-β отсечение……………………………………………… 170
3.4.4. Эффективность α-β –отсечения…………………………... 174
Выводы………………………………………………………………... 176
4. Представление знаний формулами исчисления предикатов………... 177
4.1. Метод резолюций…………………………………………………... 177
4.1.1. Формальные теории………………………………………. 178
4.1.2. Язык исчисления предикатов первого порядка…………… 186
4.1.3. Предложения………………………………………………… 189
4.1.4. Правило резолюции………………………………………... 191

4.1.5. Унификация…………………………………………………….. 194
4.1.6. Опровержение методом резолюций………………………… 199
4.1.7. Программная реализация метода резолюций……………… 201
4.2. Стратегии поиска опровержения методом резолюций……......... 205
4.2.1. Полные стратегии…………………………………………… 205
4.2.2. Неполные стратегии………………………………………… 208
4.2.3. Хорновские предложения…………………………………… 209
4.2.4. Замечания по реализации…………………………………… 210
4.3. Извлечение результата…………………………………........ 211
4.3.1. Извлечение результата (да/нет)…………………………... 212
4.3.2. Извлечение результатов (факты)…………………………… 213
4.3.3. Извлечение результатов (термы)…………………………… 214
4.4. Системы дедукции на основе правил…………………………….. 215
4.4.1. Потеря импликативности…………………………………… 216
4.4.2. Размножение литералов…………………………………... 217
4.4.3. Естественное направление дедукции……………………….. 219
4.4.4. Форма И/ИЛИ…………………………………………………... 221
4.4.5. Прямая система дедукции…………………………………… 223
4.4.6. Обратная система дедукции………………………………… 227
4.4.7. Комбинация прямой и обратной систем…………………… 230
4.4.8. Метазнания в системах дедукции…………………………... 233
Выводы……………………………………………………………………….. 237
Заключение…………………………………………………………………….. 238
Библиографический список……………………………………………... 239

ВВЕДЕНИЕ

Можно было бы назвать эту книгу «Искусственный интеллект = представление знаний + методы поиска решений». Аллюзия с книгой Никлауса Вирта «Алгоритмы + структуры данных = программы» [1] намеренная. Мы искренне, в силу воспитания, накопленного опыта и врожденных предпочтений, придерживаемся идеологии пионеров и основоположников, подобных Вирту и другим, часто и много их цитируем и пересказываем.

Отбор материала для пособия вполне традиционный, можно сказать классический, но способ изложения сугубо авторский. Во-первых, на первое место мы ставим прикладные аспекты, то есть решение полезных задач на компьютере методами искусственного интеллекта, а в ответ на вопрос «может ли машина мыслить?» отшучиваемся. Теория без практики кажется нам скучной схоластикой, и мы ее избегаем. Во-вторых, мы считаем, что исследование способов представления «человеческих» знаний в компьютере намного существеннее в настоящее время, чем совершенствование «машинных» алгоритмов поиска решений, которым отдавался приоритет в предшествующие полвека. Именно поэтому представлению знаний у нас уделяется наибольшее внимание. В-третьих, мы предпочитаем объяснения на примерах и с помощью коротких программ, в противовес излишне распространенному в этой области наукообразию. Мы считаем, что предъявление текста программы, решающей задачу, часто бывает более убедительно, чем введение нумерованных определений и доказательство «математических» теорем.

Материал книги сконцентрирован вокруг ответа на вопрос: как знания и умения человека выразить в виде программы для компьютера? Прежде чем отвечать на этот технический вопрос, уместно дать ответ на вопрос идеологический: зачем? Зачем представлять знания в компьютере? Наш ответ прост: в последнее

время это стало экономически выгодно. Кроме того, это довольно трудно, а потому просто чрезвычайно интересно.

Современные тенденции в области применения компьютеров характеризуются возрастанием значения методов искусственного интеллекта (ИИ) в программном обеспечении. Можно сказать, в настоящее время происходит непрерывный процесс «интеллектуализации компьютеров»[1]. Мы имеем в виду следующую несомненную тенденцию: на заре своего развития компьютеры были «большими арифмометрами», а сейчас корень compute в слове компьютер является уже анахронизмом. Современный компьютер — это информационная машина, которая обрабатывает знания, представленные в виде данных. Эта тенденция иллюстрируется гистограммой на рис. В.1, на которой мы показали соотношения между вычислительными ресурсами, используемыми для инженерных расчетов и для обработки знаний сейчас и полвека тому назад. Инженерные расчеты — это собирательное название для таких приложений, как расчеты полеты ракеты, начисление зарплаты, и т. д., а обработка знаний — это информационный поиск в Интернете, телекоммуникации и т. д.[2] Заметим, что численные значения процентов и масштаб на этом рисунке довольно условны. Действительно, за полвека суммарная вычислительная мощность, доступная человечеству, выросла более чем в 10^{10} раз, а не в четыре раза, как на рисунке. Темпы роста вычислительных ресурсов не укладываются в голове, ни с чем подобным человечество ранее не

---

[1] Этот процесс идет не только снаружи, в области программного обеспечения, но и изнутри. Компьютеры становятся все мощнее, и методы ИИ, которые еще вчера представляли только академический интерес, поскольку были нереализуемы «в железе», сегодня уже используются в промышленности.

[2] Самым распространенным способом использования компьютеров является простой в выключенном состоянии, а вторым по частоте применения является игра в компьютерные игры. Эта два применения компьютеров мы оставляем за рамками данного рассмотрения.

сталкивалось. Всё бóльшая доля этой стремительно растущей мощи тратится именно на обработку знаний!

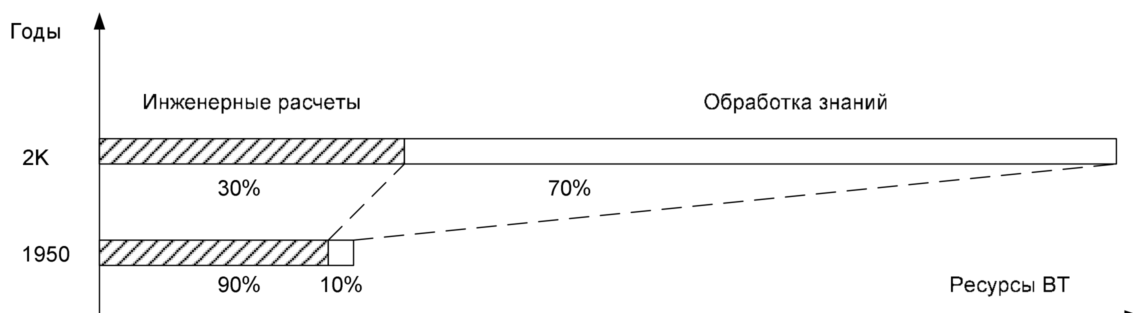

Рис. В.1. Перераспределение вычислительных ресурсов со временем

Направление искусственного интеллекта в информатике с самого начала вызвало большой, можно даже сказать, нездоровый интерес. Недобросовестные журналисты, падкие на сенсации, гиперболизировали робкие ростки новых идей, внушая необоснованно высокие ожидания широкой публике, и когда первые теоретические результаты не оправдывали завышенных ожиданий, те же журналисты участвовали в освистывании идей искусственного интеллекта. Но в XXI веке ситуация изменилась. За последнее время результаты, полученные в этом направлении, переросли рамки академических экспериментальных разработок и начали применяться в практических приложениях. Системы искусственного интеллекта с пользой применяются в реальной жизни, от медицинской диагностики до управления космическими аппаратами; к этому можно относиться как угодно, но это невозможно отрицать. Именно поэтому курсы «Методы обработки знаний», «Прикладной искусственный интеллект» и т. п. стали практически обязательными в учебных планах подготовки специалистов по информатике.

Инженеры, подготовленные по специальности «прикладная математика», и специалисты по информатике, работающие в наукоемких прикладных областях, должны владеть основными, уже устоявшимися методами поиска решения переборных задач,

разработанными в исследованиях по искусственному интеллекту. Необходимо подчеркнуть, что решающую роль в практической применимости методов решения переборных задач играют основные связанные с этими методами формализмы, используемые для предоставления знаний в компьютерах. Методы представления знаний, таким образом, являются ядром направления искусственного интеллекта. Именно это ядро, совершенно необходимое как начинающим студентам, так и опытным инженерам, является предметом книги.

Мы ожидаем, что студенты, прослушавшие данный курс, достигнут следующих результатов.

1. Знание основных методов поиска решения алгоритмически сложных задач; знание сравнительных достоинств и недостатков этих методов; знание основных принципов наиболее известных способов представления знаний: правил продукции, формул исчисления предикатов, семантических сетей.

2. Умение выбрать подходящий способ представления знаний для конкретной задачи в простых случаях; умение оценить адекватность использования для конкретной задачи того или иного метода поиска решения.

3. Свободное владение основными понятиями из области искусственного интеллекта и представления знаний; умение изучить по литературе другие методы искусственного интеллекта; умение оценить конкретную прикладную систему искусственного интеллекта и воспользоваться ею.

Методы искусственного интеллекта сейчас очень быстро развиваются и меняются, и невозможно дать заранее готовые рецепты на все случаи применения ИИ. Вместо этого в книге дается общее представление об основных принципах искусственного интеллекта, определяются наиболее распространенные методы представления знаний в компьютере.

1. ПРИКЛАДНЫЕ СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Проблемой человеческого интеллекта исследователи занимаются очень давно, хотя попытки структурировать это явление психики, или хотя бы приблизиться к более-менее строгим определениям явлений и процессов, реализующим человеческий интеллект, не всегда результативны. Авторы не являются специалистами ни в физиологии мозга, ни в психологии, равно как и не собираются «играть» на поле классической философии. Тем не менее (или, скорее, «в обоснование позиции»), мы позволим себе некоторые отступления и цитаты, иллюстрирующие ситуацию в упомянутых областях.

На сегодняшний день в психологии принято считать, что существует некий *общий интеллект как универсальная психическая способность познания*, в основе которой может лежать генетически обусловленное свойство нервной системы перерабатывать информацию с определенной прагматикой, скоростью и точностью. Считается, что многие виды умственной деятельности человека, например программирование, занятие математикой, проведение лекций, или чтение книг, подобных этой, требуют интеллекта.

К сожалению, до сих пор понимание того, что каждый из нас носит в черепной коробке, по большому счету, вряд ли можно считать достаточно полным. Если говорить о структуре мозга, как носителе интеллекта, то известно, что образующими элементами выступают клетки особого вида – нейроны. Доказано, что, по крайней мере, одна из форм взаимодействия нейронов реализуется электрохимическим путем, но вопрос, является ли эта форма единственной, остается открытым. Современная ситуация напоминает гипотетическую попытку кроманьонца понять, как устроен персональный компьютер, и что он может делать. Впрочем, что там говорить о мозге, если мы до

сих пор не в состоянии полностью осмыслить механики процесса хождения на двух конечностях.

Мы видим, что понятие «интеллект» для человека определяется косвенно, через предметную область, по некоторым результатам деятельности, а не напрямую указывается, что собой представляет и как реализуется этот аспект психики. Существующие определения, которые можно найти в энциклопедических изданиях, как правило, дают толкование, используя синонимичность и рекурсию. Например, энциклопедический словарь дает следующее краткое определение.

Интеллект *(от лат. intellectus — познание, понимание, рассудок) — способность мышления, рационального познания.*

Толковый психологический словарь близок, но отличается большей детализацией в определении.

Интеллект *(от лат. intellectus — понимание, познание) — способность к осуществлению процесса познания и к эффективному решению задач, в частности при овладении новым кругом жизненных проблем.*

Другими словами, определения ничего не определяют, но могут иногда стимулировать верные ассоциации, само же понятие «интеллект» является весьма расплывчатым и недостаточно адекватным. Невозможно, в частности, понять сущности *интуиции*, механизма *веры*, природы других *иррациональных отношений* (любви, антипатии) и так далее.

Само слово «интеллект» пробралось в термин «искусственный интеллект» не совсем законно. Дело в том, что исходный оригинальный термин artificial intelligence (см. параграф 1.3.3) имеет значительно более мягкий смысл. Слово intelligence все-таки скорее означает сообразительность, нежели рефлектирующий разум, как фундаментальное свойство человека. Но другой перевод на русский язык предложить трудно и уже поздно, поэтому мы пользуемся существующим переводом, вкладывая в него следующий смысл.

Предметом этой книги является *прикладной искусственный интеллект* (ИИ), и здесь мы рассуждаем о практически реализуемых и полезных проектах, избегая пикировки в стиле пресловутого «голого петуха» Платона-Диогена[3]. Понятие ИИ созвучно понятию интеллекта естественного, но имеет с последним, на самом деле, мало общего. Важно уяснить, что ИИ не является моделью человеческого мышления[4], так же как, например, резина не является химической моделью каучука, хотя обладает рядом схожих физических свойств. Действительно, программа с искусственным интеллектом использует некоторые алгоритмы, свойственные так же и человеческому мышлению (в конце концов, любая программа написана людьми). Но в реальной жизни человек будет решать ту же задачу иначе, чем это делает программа (производя другие операции, используя другие алгоритмы, задействуя структуры психики, пока что недостаточно изученные, а потому не поддающиеся переводу в математические формулы и программный код). Лобовой подход — изложение «человеческих» алгоритмов «машинным» языком малоэффективен, как мы увидим в дальнейшем. Другими словами, *если мы относим задачу к интеллектуальным, то любое ее решение*, по нашему определению, *свидетельствует о некотором интеллекте. Если решение добывается программным путем, то эта программа содержит элементы интеллекта.*

---

[3] Когда Платон дал определение, имевшее большой успех: «Человек есть животное о двух ногах, лишенное перьев», Диоген ощипал петуха и принес к Платону в школу, объявив: «Вот платоновский человек!». После этого к определению было добавлено: «И с широкими ногтями».

[4] Ведутся исследования, направленные на моделирование мышления человека в компьютере, именно с целью изучения мышления человека, а не с целью решения интеллектуальных задач на компьютере. В данной книге мы рассматриваем не моделирование мышления, а решение прикладных задач, что подчеркивается использованием прилагательного «прикладной» применительно к ИИ.

Программа, умеющая решать задачи в предметной области, которую традиция относит к интеллектуальным, называется ***прикладной системой с элементами***[5] ***искусственного интеллекта.***

Другое определение: ***прикладная система с элементами ИИ*** *— это программа, алгоритм которой (пока!) нельзя найти в вузовском учебнике.* Например, на заре программирования программы, которые могли строить другие программы (первые компиляторы), считались «интеллектуальными». Современные компиляторы не считаются интеллектуальными программами, хотя «знают» про составление программ значительно больше и применяют весьма изощренные алгоритмы решения задач компиляции и оптимизации. Можно приводить и другие примеры. В каждом отдельном случае конкретная прикладная система ИИ — это ноу-хау создателей такого программного обеспечения, но каждый раз любая новая идея в конкретной предметной области дает существенное продвижение в развитии искусственно-интеллектуальных систем в целом. При этом по мере их прогресса меняются и традиции отнесения предметных областей к интеллектуальной сфере, и меняется оценка степени «интеллектуальности» многих задач. Например, расчет конструкции Эйфелевой башни в конце XIX века потребовал огромного труда, недюжинного интеллекта и изобретательности, а сейчас подобный расчет может быть выполнен рядовым инженером-конструктором с помощью готового пакета программ быстро, надежно и без особых умственных усилий. Опыт и интеллект инженеров-конструкторов, исчерпывающие знания по

---

[5] Мы вставили в наше определение осторожное слово «элементы», чтобы подчеркнуть, что программа решает *некоторые* задачи в *определенной* области и не претендует на универсальную «способность мышления, рационального познания». Менее осторожные или более самонадеянные авторы употребляют оборот «прикладные системы с искусственным интеллектом». Мы также употребляем этот оборот, например, в названии первой главы, но оговариваем, что понимаем его в осторожном и минимальном смысле.

механике, сопротивлению материалов и другим инженерным дисциплинам в значительной степени аккумулированы в современных конструкторских пакетах. Такой пакет, вне всякого сомнения, демонстрирует интеллектуальное поведение, однако мы не считаем его прикладной системой ИИ, поскольку все расчеты в конструкторском пакете выполняются по хорошо известным, заранее заданным однозначным алгоритмам.

Для подкрепления изложенных тезисов и конкретизации основного определения прикладной системы с элементами ИИ рассмотрим некоторые из предметных областей, типичных для приложений искусственного интеллекта.

1.1. ОБЗОР ПРИЛОЖЕНИЙ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Некоторыми типичными приложениями, использующие элементы ИИ, являются:

- понимание естественного языка и машинный перевод;
- интеллектуальные базы данных и вопрос-ответные системы;
- экспертные системы и автоматическое доказательство теорем;
- планирование действий робота и распознавание образов;
- интеллектуальные игры.

Этот список не является ни исчерпывающим, ни ортогональным. Это просто набор наших излюбленных примеров.

1.1.1. Понимание естественного языка и машинный перевод

Процесс ***понимания естественного языка*** *— это процесс передачи знаний от одного носителя языка к другому.*

Несмотря на всю свою кажущуюся простоту и привычность, этот процесс чрезвычайно сложен. Он подразумевает наличие общей области знаний у обоих партнеров и согласованных механизмов представления знаний в виде данных, а также наличие обратного механизма извлечения знаний из данных.

Для компьютерной реализации процесса понимания естественного языка требуются мощные вычислительные ресурсы,

управляемые нетривиальными (интеллектуальными) программными комплексами. Рассмотрим, в чем здесь состоит проблема. Пусть у нас есть два интеллектуальных агента, для начала пусть это будут люди. Первый является источником знаний, назовем его *A*, а второй — приемником знаний, назовем его *B*. В процессе понимания задействовано два механизма. Во-первых, у *A* есть алгоритм, позволяющий преобразовывать гипотетические знания в данные, сигнал, (текст, речь или другие формы выражения естественного языка[6]) — *механизм представления знаний*. Во-вторых, у *B* есть механизм, который позволяет преобразовывать получаемый сигнал обратно в знания — *механизм извлечения (знаний)*. Если принять, что эти механизмы реализуются алгоритмами[7], то понятно, что у каждого из них есть область определения и область значений, вход и выход. Понимание происходит тогда, когда выход алгоритма представления знания и вход алгоритма извлечения знаний, по меньшей мере, пересекаются на общей интерпретации; а полное понимание достигается тогда, когда эти области совпадают.

Приведем несколько примеров. Пусть мы, автор — это агент *A*, а вы, уважаемый читатель, это агент *B*. Следующая фраза: «овелан аварпс ее иласипаз отсорп ым» — вызовет у вас, скорее всего, недоумение и непонимание в первое мгновение. Здесь агент *A* использует нестандартный способ представления знаний, и это создает затруднения при извлечении знаний агентом *B*. Еще пример. Если *A* использует язык, которым *B* не владеет (скажем, если бы текст этого абзаца прочел бы по-китайски для вас вслух китаец, знающий русский, при условии, что вы не знаете китайского), то происходит следующее: механизм представления знаний работает, механизм извлечения работает, но должного понимания не происходит.

[6] Язык жестов — это также естественный язык.

[7] Например, дар «говорить языками», который апостолы получили в день Пятидесятницы после сошествия на них Святого Духа (Деян. 2:3-11), явно неалгоритмической природы.

Русскоговорящий агент *B* смог бы оценить, к примеру, музыкальность речи, необычность звукосочетаний… и, в конце концов, прийти к выводу, что он, агент *B*, ничего (!) не понял, т. е. имеет место пустая общая интерпретация. А вот пример «ортогональной[8]» общей интерпретации. Русская девушка спрашивает своего парня (болгарина): Ты меня любишь? – Тот в ответ качает головой (от плеча к плечу). – Девушка рыдает от горя, хотя должна бы прыгать от счастья! – Дело в том, что у болгар наш подтверждающий кивок является жестом отрицания, а покачивание головой из стороны в сторону – означает подтверждение.

Мудрецу достаточно сказать одно слово, для того, чтобы другой мудрец его понял. Иными словами, для мудрецов общий объем передаваемых данных может быть значительно меньше, чем у среднестатистических людей, за счет более мощных механизмов представления и извлечения знаний.

***Общая интерпретация** — это одинаково трактуемый обеими сторонами набор семантических правил, связывающих языковые конструкции с их смысловой нагрузкой.*

Обратимся теперь к компьютерам. В компьютерах механизмы представления знаний и извлечение знаний тривиальны, понимание сводится к передаче данных. Поэтому, если компьютеры соединены каналом связи и форматы данных согласованы, то понимание достигается легко, а в противном случае, понимания нельзя достичь никоим образом. Критическим случаем, находящимся в фокусе нашего внимания, является взаимопонимание человека и компьютера. Существует два принципиально разных способа общения человека с компьютером (рис. 1.1). Первый из них — когда человек способен понимать компьютер. Это не заслуга компьютера, а способность человека, которого все окружающие называют программистом. Второй — когда компьютер способен понимать человека. Такой

---

[8] Пересечение не пусто, но смысл конструкций прямо противоположен.

компьютер обязательно должен обладать элементами ИИ. Поскольку в аппаратном обеспечении обычного компьютера трудно разместить интеллект, даже искусственный, ясно, что понимание должно обеспечиваться программно. И такие программы были созданы уже сравнительно давно, хотя все они без исключения весьма и весьма ограничены в понимании естественного языка или даже какого-либо более-менее широкого фрагмента языка.

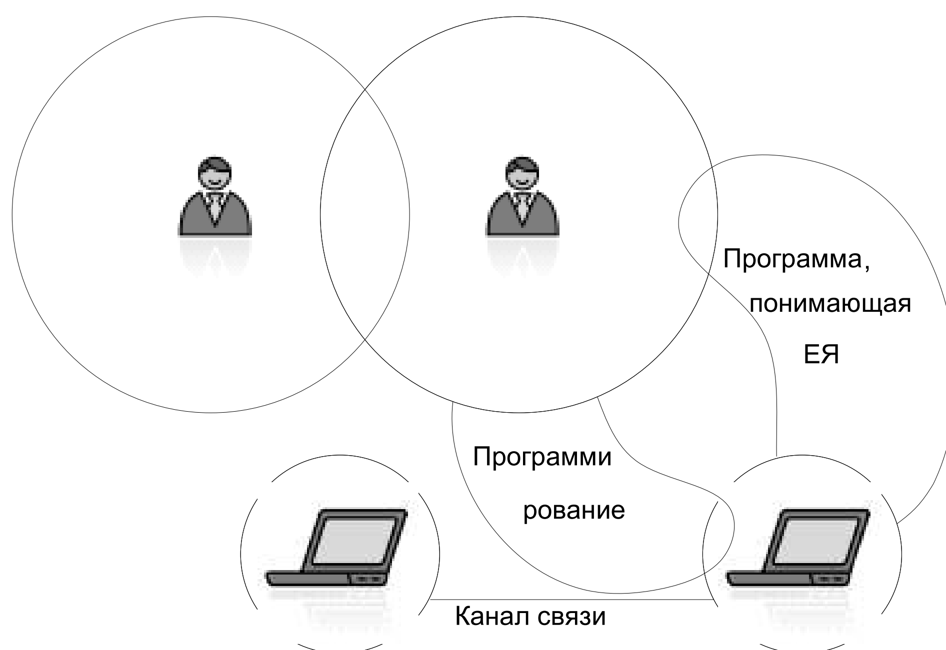

Рис. 1.1. Общий контекст понимания

Одной из первых прикладных систем ИИ, понимающих естественный язык, явилась программа ELIZA, созданная в 1966 году Дж. Вейценбаумом.

Позднее Терри Виноград (Terry Winograd) написал книгу «Программа, понимающая естественный язык», описывающую программу ELIZA [2]. Структура программы ELIZA была чрезвычайно проста и включала в себя всего несколько десятков эвристических правил, предназначенных для распознавания

грамматических конструкций (путем простого сравнения с образцом) и преобразования поступающих вопросов в «естественные» ответы. В результате эта программа могла вести вполне правдоподобный диалог, зачастую, при этом, отвечая вопросом на вопрос. Пример диалога с программой ELIZA:

— ELIZA, ты компьютерная программа.

— Почему Вы думаете, что я компьютерная программа?

— Потому что тебя составил Вейценбаум.

— Расскажите мне о Вейценбауме.

Несмотря на всю свою простоту, эта программа находит применение в психотерапии. Разговор психотерапевта с пациентом может продолжаться довольно длительное время, прежде чем достигается терапевтический эффект. ELIZA успешно имитировала диалог с роджерианским психологом в течение многих часов (суть этого метода заключается в том, что терапевт выслушивает пациента и задает наводящие вопросы, в результате чего психическое состояние последнего улучшается… или не улучшается). Многие пациенты отмечали, что эти сеансы оказались крайне полезны для них, и отказывались верить, что с ними разговаривала компьютерная программа, а не профессиональный психолог-человек.

Сам Вейценбаум, шокированный поведением своего детища, вскоре пришел к выводу, что «основная идеология развития искусственного интеллекта — искусственный разум — безнравственна». Джозеф Вейценбаум (Joseph Weizenbaum), 1923–2008 — известный деятель в области искусственного интеллекта, заслуженный профессор МТИ (Массачусетский технологический институт). Основатель организации «Компьютерные профессионалы за социальную ответственность». За свои взгляды на ответственность ученых-программистов перед человечеством снискал в среде специалистов по информатике репутацию диссидента и еретика. *«Существуют задачи, выполнение которых не следует*

поручать вычислительным машинам, независимо от того, можно ли добиться, чтобы вычислительные машины их решали».

Машинный перевод является частным уточнением проблемы понимания естественного языка. «Частность» определяется, главным образом, режимом обмена информацией — монолог вместо диалога.

Системы автоматического перевода текстов с одного естественного языка на другой образуют важный класс программных комплексов с элементами искусственного интеллекта.

Перевод с одного естественного языка на другой, несомненно, является интеллектуальной задачей. Автоматический машинный перевод привлек внимание исследователей едва ли не раньше других предметных областей, ввиду явной практической востребованности этой задачи.

Важно, что, как и в других случаях ИИ, в классе систем машинного перевода выделяется целый ряд категорий все возрастающей сложности:

- электронный словарь;
- подстрочник технических текстов;
- художественный перевод поэзии;
- синхронный перевод речи.

Ясно, что эти задачи не могут быть решены единообразно, а значит, их решения появляются в разное время и опираются на различные технологии.

Для каждой новой области применения ИИ наблюдается одна и та же характерная картина (рис. 1.2), которую мы проиллюстрируем на примере истории развития машинного перевода. После появления новой идеи к ней проявляется повышенный интерес, возникают завышенные ожидания, пользователи ждут чуда. Но чуда не происходит — методы ИИ трудоемки, сложны, и их развитие требует немалого естественного интеллекта. Наступает разочарование, новое направление уходит из сферы ажиотажного интереса и иногда подвергается даже необоснованной обструкции. Если идея была

плодотворна, то, несмотря на падения интереса в средствах массовой информации, подспудная кропотливая работа в академических кругах продолжается, и через некоторое время (не сразу!) начинает давать практические результаты. Интерес вновь появляется, но уже не праздный, а деловой и обоснованный.

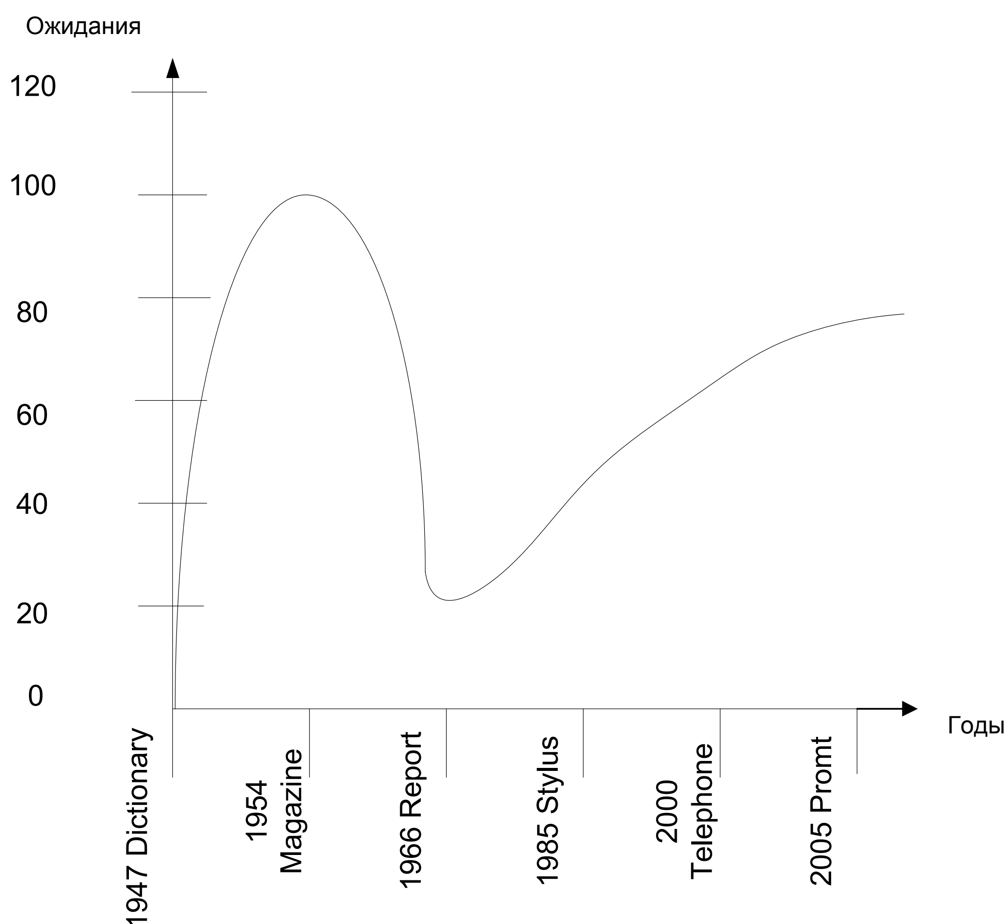

Рис. 1.2. История популярности машинного перевода

В марте 1947 г. Уоррен Уивер (Warren Weaver), 1894-1978, директор отделения естественных наук Рокфеллеровского фонда, в переписке с Эдрю Бутом (Andrew D. Booth) и Норбертом Винером впервые сформулировал концепцию машинного перевода, которую несколько позже (в 1949 г.) развил в своем меморандуме, адресованном Фонду.

В 1952 г. состоялась первая конференция по машинному переводу в Массачусетском технологическом институте, а в 1954 г в Нью-Йорке была представлена первая система машинного перевода — IBM Mark II, разработанная компанией IBM совместно с Джоржтаунским университетом (это событие вошло в историю как «Джорджтаунский эксперимент»). Была представлена очень ограниченная в своих возможностях программа (она имела словарь в 250 единиц и 6 грамматических правил), осуществлявшая перевод с русского языка на английский. После этого подобные разработки получили широкое распространение в Италии, Франции, Англии, Японии, а также был создан международный специализированный журнал.

В 1959 г. философ Й. Бар-Хиллел (Yohoshua Bar-Hillel) выступил с утверждением, что высококачественный полностью автоматический машинный перевод не может быть достигнут в принципе. Это выступление самым неблагоприятным образом отразилось на развитии машинного перевода в США. В 1966 г. был отмечен мощный спад, связанный с отчетом специально созданной Национальной Академией наук США комиссией ALPAC (Automatic Language Processing Advisory Committee). Комиссия пришла к заключению, что машинный перевод нерентабелен: соотношение стоимости и качества машинного перевода было явно не в пользу последнего, а для нужд перевода технических и научных текстов было достаточно человеческих ресурсов. Поэтому Министерство обороны решило прекратить финансирование такого рода проектов. Тема перестала быть модной, и работу продолжили только квалифицированные энтузиасты и профессионалы.

С развитием вычислительной техники в конце 70-х годов (появление микрокомпьютеров, развитие сетей, увеличение ресурсов памяти) машинный перевод вошел в эпоху «Ренессанса». При этом несколько сместились акценты: исследователи теперь ставили целью развитие «реалистических» систем машинного перевода,

предполагавших участие человека на различных стадиях процесса перевода. В результате в 80-х годах на рынке появились первые промышленные системы машинного перевода, ориентированные на массовое использование.

В 1990 г., когда системы машинного перевода снова стали одним из приоритетных направлений развития компьютерной отрасли и вышли на новый качественный уровень, пройдя непростой этап переосмысления и взаимной интеграции, Ларри Чайлдс (Larry Childs) предложил их классификацию. Он разделил все «электронные переводчики» на три группы:

1. FAMT (Fully-automated machine translation) — инструменты полностью автоматизированного машинного перевода.

входной текст → анализ / синтез текста → выходной текст

2. HAMT (Human-assisted machine translation) — приложения для автоматизированного машинного перевода текстов, выполняемого при участии человека.

входной текст → редактор → анализ / синтез текста → → редактор → выходной текст

3. MAHT (Machine-Assisted Human Translation) — вспомогательные средства для выполнения перевода человеком с использованием компьютера.

входной текст → переводчик → подстрочник → редактор → → выходной текст

Но жизненная практика свидетельствует о том, что в функционировании всех существующих на сегодняшний день систем машинного перевода в большей или меньшей степени необходимо участие человека-редактора.

Концептуально система машинного перевода имеет следующую структуру (рис. 1.3, слева): текст на исходном языке сначала анализируется, т. е. производится интерпретация данных с целью «понимания смысла», а затем понятый смысл представляется в виде текста на целевом языке. В идеале анализ производится «до полного

понимания», когда весь смысл исходного текста выражается на некотором гипотетическом универсальном метаязыке. Традиционно этот универсальный метаязык называют interlingua. Следует отметить, что к настоящему времени идеал так и не достигнут. Современные системы машинного перевода имеют трехслойную архитектуру (рис. 1.3, справа). Сначала проводится анализ входного языка до достижения *некоторого* уровня понимания, и это понимание выражается в построении различных структур над исходным текстом, часто специфических для определенного языка. Затем, используя эвристически алгоритмы, производится преобразование частично понятых структур одного языка в частично понимаемые структуры другого языка. Набор этих алгоритмов традиционно называется переносом (от английского transfer). Правила переноса имеют большой объем, используют огромные информационные ресурсы и весьма изощренные алгоритмы. Наконец, по перенесенным структурам целевого языка (они вовсе не обязательно изоморфны структурам исходного языка!) синтезируется текст на целевом языке. Разумеется, этап переноса является самым сложным, важным и наукоемким.

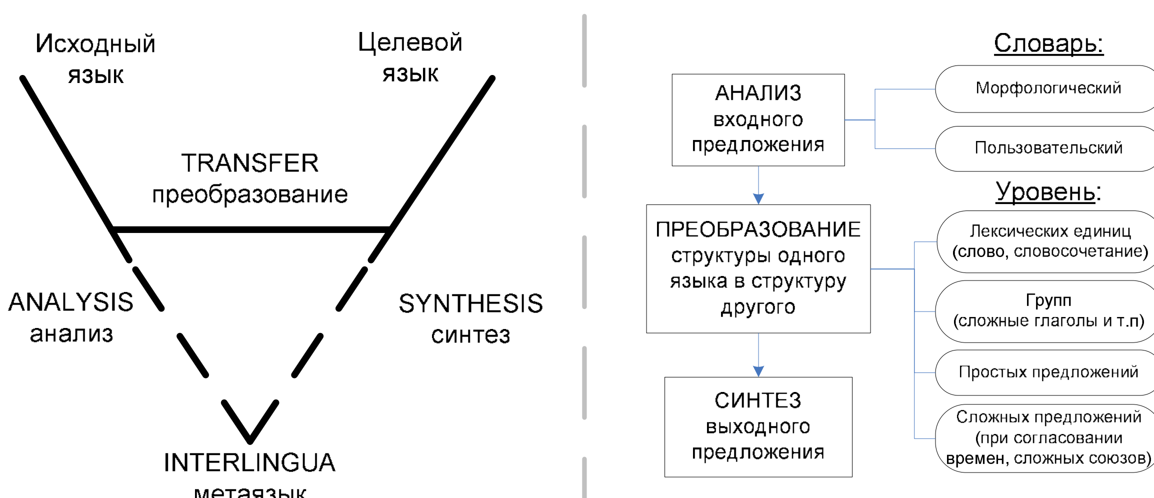

Рис. 1.3. Структура системы машинного перевода

Впрочем, мечты, с которыми род людской взялся полвека назад за задачу машинного перевода, в значительной мере остаются мечтами — высококачественный («художественный») перевод текстов широкой тематики по-прежнему недостижим. Однако несомненным является ускорение работы переводчика и повышение качества при использовании систем машинного перевода.

По мере появления все новых и новых алгоритмов, методик и программных решений, нацеленных на компьютерный перевод текстовой информации, достоверными фактами является наличие у таких систем целого ряда преимуществ перед обычным «рукотворным» переводом:

- высокая скорость перевода (использование машинного перевода позволяет значительно сократить время, требуемое для перевода больших объемов текста, по современным оценкам на 50-70 %);

- снижение стоимости перевода (с помощью машинного перевода можно сэкономить на оплате услуг профессиональных переводчиков, и при переводе «простых» текстов отказаться от них вовсе);

- конфиденциальность переводимой информации (участие компьютера в обработке конфиденциальных документов в гораздо большей степени гарантирует нераспространение их содержания, чем участие переводчика);

- универсализацию процесса перевода (даже самый квалифицированный переводчик, как правило, специализируется в определенных областях знаний и тематических направлениях, тогда как системы МП достаточно универсальны, и их работа зависит только от проведения необходимых настроек — подключения нужных электронных словарей и т. п.).

В настоящее время существует множество коммерческих проектов машинного перевода: Systran, IBM, L&H (Lernout & Hauspie), Language Engineering Corporation, Atril, Trados, Caterpillar

Co., LingoWare. В России большой вклад в развитие машинного перевода внесла группа под руководством проф. Р. Г. Пиотровского (Российский государственный педагогический университет им. Герцена, Санкт-Петербург). Сейчас заслуженное признание получил автоматический переводчик, разработанный компанией ПРОМТ (PROgrammer's Machine Translation), который удовлетворяет ожиданиям большинства пользователей.

1.1.2. Интеллектуальные базы данных и вопрос-ответные системы

На заре использования компьютеров форматы хранения данных и средства для манипулирования ими изобретались программистами индивидуально для каждого случая. Это неудобно и неэффективно, и вскоре были предложены унифицированные способы хранения данных (модели данных) и разработаны унифицированные системы доступа к данным (системы управления базами данных — СУБД). Рассматривая концепцию базы данных (БД) с самой общей точки зрения, можно отметить, что БД поддерживает три основных группы операций: занесение данных в базу, поиск данных в базе и извлечение данных из базы, причем извлекаются те данные, которые до этого были занесены[9]. Были предложены и используются несколько идей, к которым это ограничение неприменимо. Одной из них является идея интеллектуальной базы данных.

Интеллектуальная база данных *(Intelligent Database) предоставляет эффективный способ хранения, поиска и извлечения*

---

[9] Большинство пользователей СУБД воспринимают последнее утверждение как нечто, само собой разумеющееся, как закон природы, наподобие законов сохранения в физике. Какую информацию поместили в компьютер, такую и получим обратно! Но информация — вовсе не вещество и не энергия! Утверждение Михаила Васильевича Ломоносова «…сколько чего у одного тела отнимется, столько присовокупится к другому …» неприменимо к информации. Вы можете скачивать информацию из Интернета на свой компьютер сколько угодно, от этого в Интернете количество информации не уменьшится.

бо́льшего числа фактов, чем те, которые были изначально загружены в базу.

Рассмотрим простой случай. Реляционная база данных (см. табл. 1.1): таблица, строки которой соответствуют объектам, а столбцы — каким-то атрибутам объектов. Все объекты однотипны, и различаются только значениями атрибутов. Такая вещь позволяет отвечать на вопросы фактографического характера. Например, можно спросить: «В каком подразделении работает Новиков?», и получить ответ: Кафедра «Прикладная математика». Но если добавить туда информацию, представленную не как таблица, а как правило, можно будет отвечать на вопросы, ответы на которые не содержатся в базе данных. Идея состоит в том, что кроме обычных реляционных таблиц в базе данных хранятся и некоторые правила, например, такое:

```
if   (x.Должность = "Заведующий" &
     (x.Подразделение = y.Подразделение)
then x начальник y
```

База данных, снабженная такими правилами, является интеллектуальной.

Таблица 1.1

Таблица базы данных

ФИО	Должность	Подразделение
Новиков	Доцент	Кафедра «Прикладная математика»
Клавдиев	Заведующий	Кафедра «Прикладная математика»

Действительно, если поставить вопрос «Кто начальник Новикова?», то несложный встроенный механизм логического вывода легко найдет ответ: «Клавдиев». Приведенный пример позволяет выяснить отношения на уровне начальник-подчиненный. В данном случае, очевидно, что Клавдиев является начальником не только Новикова, но и любого другого сотрудника кафедры «Прикладная математика». Таким образом, интеллектуальная база данных может в качестве ответа выдать факт, который в ней не хранится. Более того, такие базы позволяют отвечать на вопросы, требующие дедуктивного

рассуждения, состоящего не из одного применения правила, а из многих применений, так что извлекаемые факты могут быть не так очевидны и весьма далеки от первоначально введенных.

Интеллектуальная база данных, которой можно задавать вопросы и получать ответы на (ограниченном) естественном языке, называется ***вопрос-ответной системой*** *(question-answering system).*

Первые вопрос-ответные системы появились еще в 60-х годах XX века. Вопросы и ответы в этих системах формулировались, разумеется, на весьма ограниченном подмножестве естественного языка, упрощенном как по словарю, так и по используемым грамматическим формам. Источник информации — то есть база данных, из которой извлекались ответы — был у каждой такой системы свой, локальный и настроенный на одну узкую предметную область. Полезность и область применения таких систем была невелика, хотя они и пользовались известной популярностью.

В настоящее время идея вопрос-ответных систем получила мощное развитие в форме интеллектуальных поисковых систем в Интернете. Опираясь на глобальные информационные ресурсы всего человечества, и используя достаточно изощренные новые алгоритмы анализа и синтеза текстов на ограниченном естественном языке, такие поисковые системы оказались более чем полезными. Без преувеличения можно сказать, что эти системы сейчас жизненно необходимы всем и каждому.

1.1.3. Экспертные системы и автоматическое доказательство теорем

Методы интеллектуальных баз данных получили дальнейшее развитие при разработке автоматических консультирующих систем. Такие экспертные системы (Expert Systems) способны предоставить компетентный ответ пользователю на конкретный вопрос. Первые экспертные системы были созданы для применения, например, в медицине (диагностика заболеваний) и в геологии (оценка месторождений).

Одна из самых главных проблем в этой области заключается в том, как следует представлять и использовать знания экспертов, которые по своей сути являются в достаточной степени разнородными и противоречивыми данными.

***Экспертные системы** — системы искусственного интеллекта, включающие в себя знания об определенной слабо структурированной и трудно формализуемой узкой предметной области, и способные предлагать и объяснять пользователю разумные решения.*

В настоящее время считается общепринятым, что структура экспертных систем состоит из пяти блоков (рис. 1.4). Совокупность трех блоков — Правила, Модель ситуации и Вывод — по сути, является интеллектуальной базой данных. Модули, позволяющие извлекать знания и объяснять выводы, превращают интеллектуальную базу данных в экспертную систему.

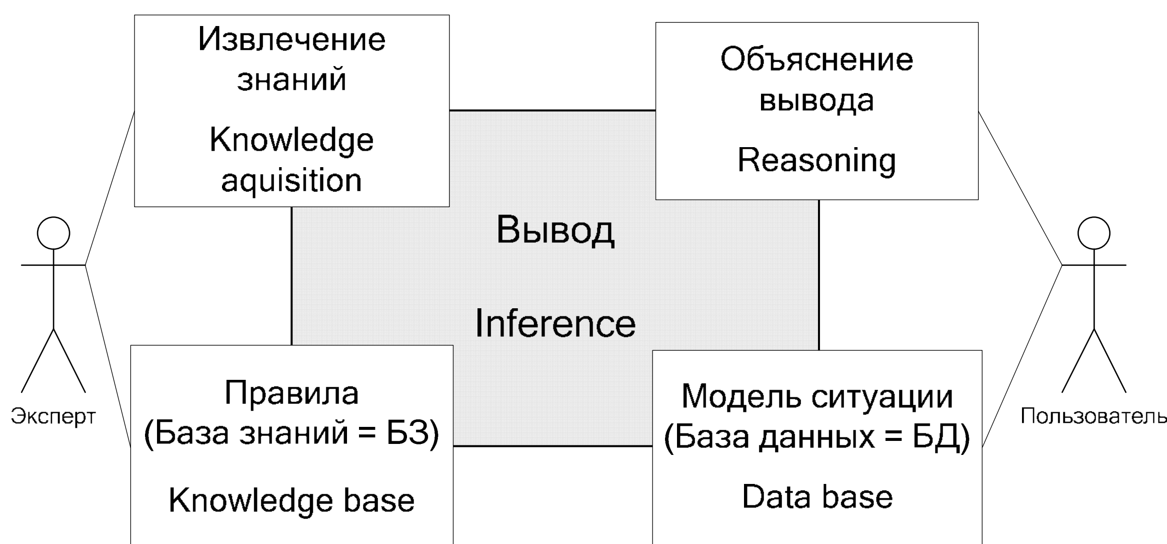

Рис. 1.4. Архитектура экспертной системы

Сейчас чаще всего база знаний для экспертной системы — это набор правил. Простейший случай, когда правила имеют

детерминированный характер[10] и позволяют из имеющихся фактов выводить новые факты, как в интеллектуальной базе данных. Такой вариант детально рассматривается во второй главе, где мы говорим о представлении знаний системами продукций.

Блок объяснений вывода фактически представляет машинный вывод результата в человеко-читаемой форме, так что пользователь может проследить, какие шаги рассуждений применяла экспертная система при решении задачи. Тем самым доверие пользователя к ответу возрастает.

Наиболее важным и сложным является блок извлечения знаний. Результат работы этого блока понятен — это пополнение и изменение базы знаний. А вот исходные данные могут быть различными. Не следует надеяться на то, что эксперт в предметной области просто так «из головы» сформулирует эффективные правила рассуждений, причем сразу в том формате, который требуется в системе. Это маловероятное чудо. Одним из наиболее надежных считается метод постепенного пополнения и изменения базы знаний, причем поводом к изменению является каждый случай, когда система не смогла решить задачу или решила ее неправильно. В таком случае эксперт анализирует с помощью блока объяснения ход машинных рассуждений, определяет, какое правило было неудачным или какого правила не хватает, и добавляет его в базу знаний посредством блока извлечения знаний. Можно сказать, что экспертная система, подобно человеку, учится на ошибках.

В центре структуры экспертных систем находится блок вывода. В этом блоке правила применяются к фактам из модели ситуации, в результате получаются новые факты, к которым также применяются

---

[10] Довольно часто встречается ситуация, когда правила имеют не детерминированный, а вероятностный характер: «если *A*, то *B* с вероятностью *p*». Большинство выводов, сделанных экспертной системой по вероятностным правилам, имеют вероятностный характер. На свой вопрос пользователь получает несколько различных ответов, каждому из которых приписана своя вероятность достоверности.

правила и так далее пока не будет достигнута цель, то есть требуемый факт. Этот процесс по существу является процессом поиска вывода в формальной теории. Действительно, исходные факты из модели ситуации — это аксиомы. Мы применяем к ним правила, получаем новые факты, причем каждый новый факт получается из ранее полученных фактов. В конце концов, мы доходим (или не доходим) до целевого факта, то есть, говоря языком логических теорий, получаем заключение целевой теоремы. Таким образом, ядро всякой экспертной системы можно рассматривать как систему автоматического поиска вывода в некоторой формальной теории, то есть как доказательство теорем.

Доказательство теорем безусловно предполагает затрату интеллектуальных усилий, требующих не только дедуктивных навыков, но и в значительной степени интуитивных догадок (например, какие леммы необходимы для той или иной теоремы). Другими словами, доказательство теорем подразумевает, в том числе, поиск уже доказанных теорем и решенных подзадач, которые могут оказаться полезны для достижения конечной цели. К настоящему времени разработано несколько программ с элементами ИИ, способных работать в этом направлении, то есть программ автоматического доказательства теорем (Automatic Theorem Proving).

При первом взгляде на предмет мы сразу отмечаем обстоятельства, известные из курса математической логики.

1. Язык исчисления предикатов первого порядка, обогащенный такими вещами, как равенство и внелогические предикаты и функции, является общепринятым средством для аксиоматизации в большинстве предметных областей (то есть практически любое утверждение может быть сформулировано как формула прикладного исчисления предикатов).

2. Алгоритм Тарского перечисления правильно построенных формул позволяет генерировать гипотезы исчерпывающим образом

(то есть все правильно построенные формулы могут быть перечислены).

3. Метод резолюций является частично корректным (то есть для любой доказуемой формулы вывод может быть найден автоматически, а для недоказуемой формулы алгоритм может не завершить свою работу).

Возникает следующая идея, подкупающая своей незатейливостью и простотой. Запишем аксиомы предметной области на языке исчисления предикатов первого порядка. Запустим алгоритм типа алгоритма Тарского, который будет генерировать новые гипотезы одну за другой. Каждую гипотезу будем пытаться автоматически доказывать методом резолюций, и если получится, то компьютер сам будет добывать все новые и новые знания. Процесс можно запустить на 24 часа в сутки, 7 дней в неделю, 365 дней в коду, и не на одном компьютере, а на всех незанятых в данный момент. Кстати, сейчас в среднем 90 % времени персональные компьютеры простаивают, и их суммарная неиспользуемая вычислительная мощность довольно велика, так что у человечества хватает неиспользуемых ресурсов.

Таким образом, автоматическое доказательство теорем возможно, и его нетрудно запрограммировать, но такой прямолинейный подход оказывается безнадежно неэффективным. Сложные, интересные теоремы при лобовом подходе автоматически доказать не удается, потому что пространство перебора столь велико, что даже самые быстродействующие компьютеры не в состоянии ничего сделать за разумное время.

Однако существуют примеры, правду сказать, пока немногочисленные, когда в очень узком классе математических теорий, применяя очень специфические методы, удавалось автоматически находить и доказывать нетривиальные содержательные теоремы, которые были ранее неизвестны. Одному из таких примеров уже много-много лет [3]. Член-корреспондент РАН

В. М. Матросов придумал удивительный способ перечисления и доказательства математических фактов в одной узкой области, относящейся к теории дифференциальных уравнений. Коллеги-программисты составили программу, которая генерировала гипотезы и автоматически строила к ним доказательства. Таким образом, были найдены и доказаны некоторые действительно новые, нетривиальные теоремы. При этом результаты (теоремы и доказательства) публиковались в престижном математическом журнале, как новые научные результаты в математике. Мировое сообщество пока не знает, как отнестись к этому факту, однако прецедент есть.

Следует подчеркнуть, что *автоматическое доказательство теорем является основным инструментом всех прикладных систем с элементами искусственного интеллекта, которые в той или иной степени используют формулы логических исчислений для представления знаний.* Формулы исчисления предикатов используются практически всегда — это основной способ формальной записи любых утверждений, известный человечеству в настоящее время. Поэтому можно смело утверждать, что *автоматическое доказательство теорем — это фундаментальная тема ИИ*. В следующих главах показано, что все рассматриваемые нами методы ИИ, а возможно и все методы ИИ вообще, прямо или косвенно сводятся к автоматическому доказательству теорем.

1.1.4. Автоматическое управление роботом и распознавание образов

Ро́бот *(robot) — автоматическое или дистанционно управляемое устройство, которое частично или полностью заменяет человека при выполнении некоторых видов работ.*

Слово «робот» было придумано чешским писателем Карелом Чапеком и его братом Йозефом и впервые использовано в пьесе Чапека «Р.У.Р.» («Россумские универсальные роботы», 1920).

Роботы чаще всего используются при выполнении работ в опасных для жизни условиях или при относительной недоступности

объекта. Все наслышаны об успехах беспилотных луно- и марсоходов, военных роботов, обеспечивающих разведку и обезвреживание взрывоопасных предметов. Недавно (в 2009 году) в Японии прошла очередная международная выставка роботов, на которой большинством моделей были представлены бытовые и медицинские роботы (рис. 1.5). К примеру, вниманию публики был представлен «робот-пациент» для обучения дантистов, который способен реагировать на боль. Другой образец — «робот-полицейский», способный запоминать лица людей, которых разыскивает полиция. При установлении личности робот немедленно предупреждает об этом по радиосвязи сотрудников охранной службы. Все большую популярность приобретают роботы, созданные для развлечения. На той же выставке был представлен «робот-партнерша по танцам». Это «женщина» в человеческий рост, которая при помощи специальных датчиков чутко улавливает, в какую сторону движется танцор, и перемещается в такт с ним. Она может исполнить и короткий сольный танец, плавно перемещаясь и двигая руками, головой и корпусом. Начинает казаться, что мечты о будущем роботов, описанные фантастами XX века, начинают сбываться.

Рис. 1.5. Примеры роботов: робот, играющий в пинг-понг,
робот-танцор, робот-пациент

На самом деле, *управление роботом* (robot planning) — задача чрезвычайно сложная. Казалось бы, нет ничего проще, чем

передвигаться на двух ногах. Развитие любого человеческого существа начинается со способности видеть и ходить. При изучении временной шкалы (рис. 1.6) появляется мысль, что эти навыки вроде бы и не требуют значительных интеллектуальных усилий. На первый взгляд может показаться, что говорить, рассуждать и программировать значительно сложнее. Однако если взглянуть на развитие роботов и робототехники, можно легко убедиться в обратном. Роботы сначала научились программировать и рассуждать, потом говорить, затем ходить, и до сих пор не научились видеть как следует.

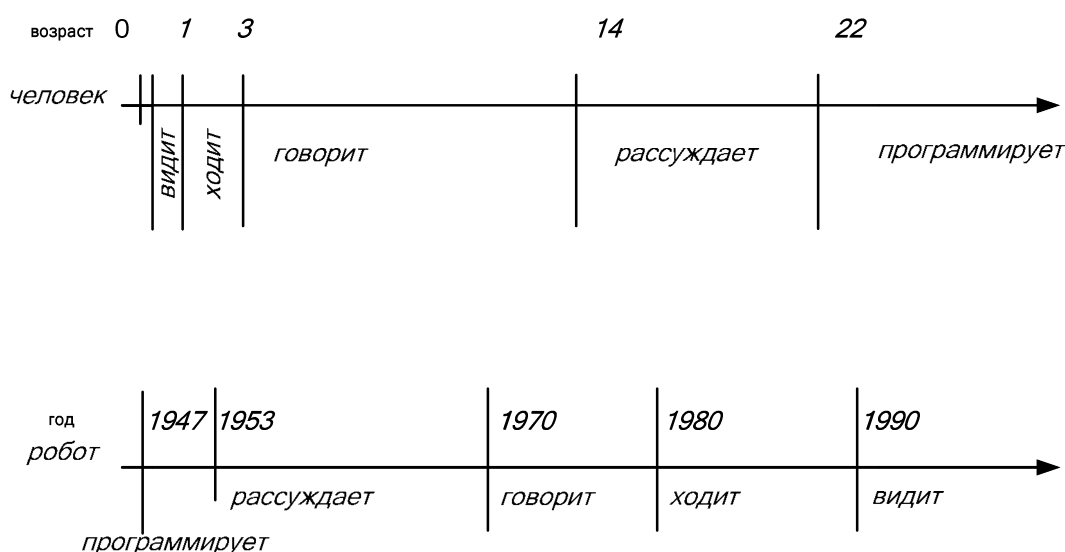

Рис. 1.6. Сравнение развития возможностей робота и человека

Успешное хождение двуного робота — достаточно сложная задача даже для современного уровня развития науки и техники. Дело в том, что когда мы ходим, центр тяжести нашего тела почти никогда не находится над площадью опоры. С точки зрения механики, мы не ходим, а непрерывно падаем, но не совсем, а удачно подставляем ногу в нужное место и продолжаем движение. Лобовой подход в стиле XIX века — составить механическую модель с нужным количеством степеней свободы и на каждом шаге решать соответствующую систему дифференциальных уравнений, — не работает. Даже самый

быстродействующий современный компьютер не успевает проводить нужные вычисления в реальном времени. Совершенно ясно, что мы, когда ходим, не решаем в уме систему уравнений. Также и для реализации хождения роботов применяются совсем другие приемы. Каждое новое достижение в этой области рассматривается как значительный успех, что видно из перечисленного выше состава экспонатов международной выставки. Что касается быстрого бега и прыжков (на уровне профессиональных спортсменов), то двуногий быстро бегающий и прыгающий робот — все еще почти нереальная задача на сегодняшний день[11]. Робот-терминатор, который может двигаться и стрелять так же ловко, как это делает Шварценеггер в известном фильме, пока что, к счастью, невозможен.

До сих пор не создано адекватного аппарата, позволяющего роботу «видеть» окружающий мир и реагировать на «визуальные» изменения в нем также хорошо, как это делает человек. То, как видим мир мы, — результат длительного обучения, основанного на интеллектуальной активности. Известно, что уже на третьей неделе развития ребенок обладает цветным стереоскопическим зрением. В результате взаимодействия с предметами и накопления опыта наблюдений у ребенка на основе цветного стереоскопического зрения формируется новая способность — предметное восприятие. Важно подчеркнуть, что эта способность объясняется не физиологическими, а интеллектуальными причинами[12]. С другой стороны, исследователи установили, что если поместить человека в совершенно непривычную искусственную среду (в которой ненормально устроено освещение, привычные предметы имеют непривычные размеры, процессы имеют ненормальный темп и т. д.), то способность к адекватному

---

[11] Впрочем, далеко не все представители человечества могут бегать и прыгать на уровне профессиональных спортсменов.

[12] Способность видеть мир объемным объясняется не только тем, что у нас два глава. В этом нетрудно убедиться: прикройте один глаз — способность видеть не плоский, а трехмерный мир, и определять расстояния до предметов сохранится!

восприятию трехмерной действительности человеком утрачивается. Таким образом, предметное восприятие — это способность психики, а не глаза. На самом деле, мы видим модель, которую очень быстро, за десятые доли секунды, строит мозг по информации, полученной из глаза. Иногда эта модель может несколько расходиться с реальностью (см. например рис. 1.7). Проблема технического зрения заключается не в разрешающей способности камер и не в их количестве, а в сложности алгоритмов, адекватно восстанавливающих трехмерную модель по совокупности плоских изображений, полученных от камер. Для конкретных условий и специальных задач есть неплохие технические решения, например, роботы, распознающие человеческие лица в охранных системах, или роботы, распознающие положение и ориентацию мелких деталей на конвейере часового производства, однако эти системы не взаимозаменяемы и не совместимы. Универсальной системы технического зрения, сопоставимой по широте возможностей с человеческими, пока еще не сконструировано[13].

Машинное зрение — это один из примеров широкой области применения методов ИИ, которая называется распознавание образов.

Распознавание образов *(pattern recognition) — раздел кибернетики, развивающий методы классификации и идентификации предметов, процессов и сигналов, которые характеризуются конечным набором свойств и признаков.*

К этой области относится, например, задача о переходе улицы по сигналам светофора. Распознавание цвета загоревшейся лампы светофора и знание правил дорожного движения позволяет принять правильное решение о том, можно или нельзя переходить улицу в данный момент.

---

[13] Заметим, что еще Конан Дойль в рассказах о Шерлоке Холмсе справедливо акцентировал внимание на том, что способности замечать и распознавать детали (наблюдательность) у людей сильно варьируются. Часто бывает, что люди смотрят на одно и то же, а видят совсем разные вещи.

Одной из первых искусственных нейронных сетей, способных к восприятию (перцепции) и формированию реакции на воспринятый стимул был *персептрон* Фрэнка Розенблатта. Персептрон рассматривался его автором не как конкретное техническое вычислительное устройство, а как модель работы мозга. Нужно заметить, что после нескольких десятилетий исследований современные работы по искусственным нейронным сетям редко преследуют такую цель.

На примере распознавания образов мы опять наблюдаем характерную кривую развития направления ИИ (см. рис. 1.2 и пояснения к нему). После появления пионерской работы Розенблатта наблюдался резкий подъем интереса к этой тематике, затем последовал холодный душ в виде книги М. Минского и С. Пейперта, в которой, среди прочего, показана принципиальная ограниченность возможностей персептрона Розенблатта. Казалось, перспективы утрачены, но сейчас вновь наблюдается увеличение интереса к этой теме. К настоящему времени первоначальная идея трансформировалась в нейронные сети и нейрокомпьютеры, которые в текущий момент более чем успешны.

Один из серьезных успехов в распознавании образов связан с созданием в 1982 г. фирмой Logica системы распознавания отпечатков пальцев, которая нашла широкое применение не только в криминалистике, но и в разнообразных системах защиты, например в ноутбуках. Другими примерами применения искусственно-интеллектуальных систем являются: распознавание речи, распознавание сканированного текста (OCR), распознавание автомобильных номеров или мониторинг среды по аэрофотоснимкам.

Необходимо хорошо понимать, что *синтез информационного объекта всегда значительно проще, чем анализ того же самого объекта*. Наглядный пример — рукописный текст. Его очень легко написать, но прочитать его иногда неимоверно трудно. Всем нам, надо полагать, знакома ситуация, когда в Интернете при регистрации

на сайте, отправке сообщения и тому подобных операциях необходимо «расшифровать» код на рисунке. Это считается достаточно надежным тестом, отличающим реального пользователя от робота. Современные программы распознавания не могут проанализировать сложное изображение, где контуры букв и цифр не замкнуты, линии причудливо искажены и присутствует большое количество «шума», а человеку это удается практически без усилий. Пока что распознавание образов — неоспоримое конкурентное преимущество человека по отношению к роботу — надолго ли?

Мы закончим этот раздел примерами изображений (рис. 1.7), которые показывают, что задача распознавания образов отнюдь не проста, даже для человека.

Голова осла или обнаженные фигуры?

Старик или влюбленная пара?

Одно лицо или два?

Молодая женщина или старуха?

Рис. 1.7. Примеры сложных изображений для распознавания образов

1.1.5. Интеллектуальные игры

Одно из самых известных и популярных применений прикладных систем с элементами искусственного интеллекта — это разнообразные *интеллектуальные игры* (intelligent games). Почти наверняка каждый сталкивался с искусственным интеллектом, использующимся именно в этой области. Шашки, шахматы, нарды встречаются и в мобильном телефоне, и в качестве более серьезных программ для достаточно мощных компьютеров.

Исторически случилось так, что из всех интеллектуальных игр внимание программистов-исследователей более всего привлекали шахматы. На данный момент лучшие компьютерные шахматные программы уже превосходят в игре известных гроссмейстеров мира. Можно сказать, что в шахматы компьютер переиграл человека. При этом следует подчеркнуть, что никаких чудес в шахматных программах нет. В них используются хорошо известные и тщательно реализованные алгоритмы, подобные тем, что рассматриваются в главе 3. Тем не менее, если мы принимаем утверждение, что игра в шахматы на гроссмейстерском уровне требует интеллекта, то мы вынуждены признать, что лучшие шахматные программы обладают интеллектом. При этом компьютерные шахматы — это не только развлечение, но и способ обучения новых шахматистов, создание шахматных баз данных, анализ окончаний, изменение правил[14]. Но для нас самое главное, что, в любом случае, это один из наиболее интересных случаев представления знаний.

Задача представления знаний в интеллектуальных играх является вызовом для естественного интеллекта программистов. Дело в том, что некоторые из игр чрезвычайно сложны с количественной

---

[14] Любопытный пример. Ранее в шахматах действовало правило, что партия заканчивается вничью, если за пятьдесят ходов не было взятий или превращений. С помощью компьютерного анализа было найдено результативное окончание, которое требовало 53 хода. В итоге шахматные правила были изменены.

точки зрения. Например, шахматы имеют около 10^{40} возможных позиций, и около 10^{120} возможных партий. Это огромные числа, бóльшие, чем число атомов во Вселенной (по разным оценкам от 10^{67} до 10^{80}) и чем число секунд, прошедших от начала существования Вселенной[15]. Никакие информационные устройства не могут и никогда не смогут просмотреть или запомнить все позиции и разыграть все партии — это физически невозможно. Так что у разработчиков программ для интеллектуальных игр впереди непочатый край работы!

Огромный вклад в алгоритмическое понимание шахмат внес М. М. Ботвинник. Михаил Моисеевич Ботвинник (1911–1995) — доктор технических наук, шестой чемпион мира по шахматам, носивший этот титул в общей сложности 13 лет. Первым уделил особое внимание вопросам тренировки шахматистов, создал свой метод подготовки к соревнованиям, в котором важное место уделялось физическим упражнениям и укреплению психологической устойчивости. Внёс ценный вклад в теорию многочисленных начал, разработал ряд оригинальных дебютных систем, обогатил ценными анализами теорию эндшпиля. *«Я отдал шахматам, по-моему, все, что было возможно. Сейчас у меня просто нет времени готовиться к крупным турнирам. Когда я был занят одной электротехникой, работу удавалось сочетать с шахматами. Но уже несколько лет я работаю еще и над проблемой игры в шахматы на вычислительной машине и так погрузился в эту проблему, что сочетать все три занятия стало невозможно».*

Закончив в 1970 спортивные выступления («матч века» сборная СССР — сборная мира), Ботвинник занимался проблемами искусственного интеллекта, работал над компьютерной шахматной программой «Пионер». Идея создания «электронного шахматиста» возникла у него еще в 1958 году. В 1968 году вышла книга

---

[15] С момента большого взрыва всего-то прошло от 13 до 14 миллиардов лет, т. е. $5*10^{17}$ секунд.

Ботвинника, в которой он сформулировал метод моделирования мышления шахматного мастера при выборе хода, построенного на позиционной оценке, отметающей все побочные и ненужные ходы и варианты. К сожалению, программа «Пионер» так и не была реализована, но ценные идеи и решения Ботвинника остались в теории шахмат, и нашли свое применение в позднейших разработках отечественных специалистов по ИИ.

Уже несколько десятилетий проводятся регулярные чемпионаты мира среди шахматных программ. В настоящее время Чемпионат проводится Международной ассоциацией компьютерных игр и открыт для всех типов компьютеров, у которых не более восьми ядер (то есть, исключая суперкомпьютеры и большие кластеры). Отрадно отметить, что на первых чемпионатах побеждала отечественная программа «Каисса», разработанная под руководством Г. М. Адельсона-Вельского и М. В. Донского. Чемпион последних трех лет — программа «Рыбка» («Rybka»), разработанная чешско-американским шахматистом Васиком Райлихом (Vasik Rajlich).

1.2. МЕСТО ПРЕДСТАВЛЕНИЯ ЗНАНИЙ В ИСКУССТВЕННОМ ИНТЕЛЛЕКТЕ

В интеллектуальной области невозможно создать универсальный алгоритм, подходящий для решения любой задачи. Этим интеллектуальные задачи отличаются от других, неинтеллектуальных, в которых существуют алгоритмы «делай раз, делай два, делай три…». *Метод решения задачи в интеллектуальной области — это всегда поиск новых подходов, новых путей, в конце концов, если ничего не удается придумать — банальный перебор всех возможностей в поисках решения.*

1.2.1. Итеративный характер решения задач

Общая схема решения проблемы с помощью компьютера изображена на рис. 1.8. Стрелочки сверху вниз — детерминированный процесс, стрелочки снизу вверх —

недетерминированный. Решение проблемы всегда носит, в целом, недетерминированный характер. Это метод проб и ошибок, метод поиска, в результате которого часто приходится возвращаться на предыдущие шаги.

Если обратная связь в таком процессе (стрелочки снизу вверх) реализуется через человека (а это происходит достаточно часто), то такой процесс не может называться компьютерной обработкой знаний. Для того чтобы можно было говорить именно о компьютерной обработке знаний, все или некоторые петли обратной связи должны замыкаться, соответственно, через компьютер.

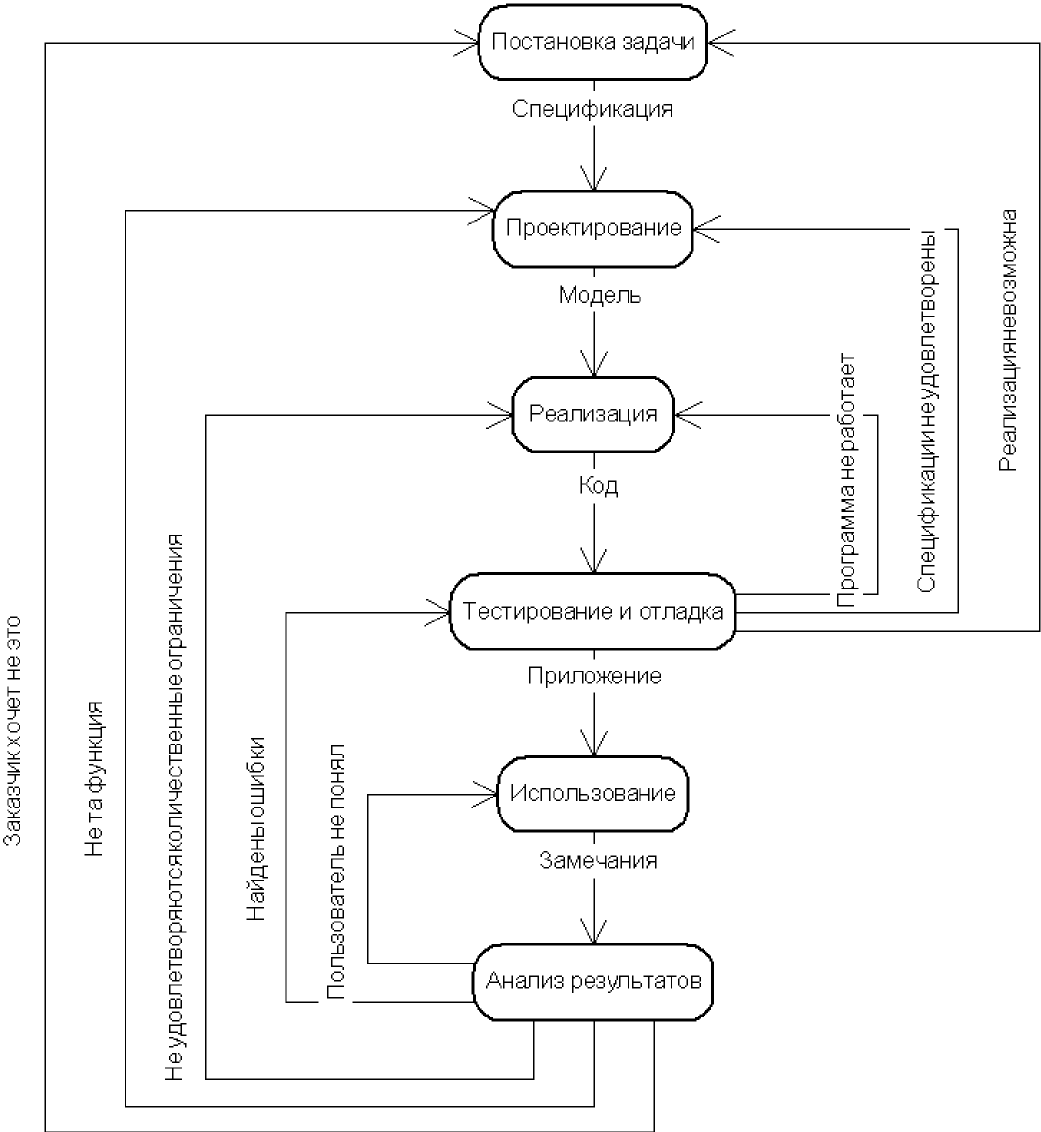

Рис. 1.8. Схема процесса решения задачи на компьютере

При традиционном процедурном программировании (сверху вниз), решение целиком зависит от задачи. Каждая задача имеет свое индивидуальное решение, и решение одной задачи не может быть непосредственно, без дополнительной работы, использовано для решения другой. В искусственном интеллекте (непроцедурное программирование) один и тот же метод поиска может применяться к целому набору проблем.

Наше резюме таково. *Если задача не имеет прямого алгоритма решения, решается перебором, и возвраты осуществляются не через голову программиста, а через саму программу, то это прикладная система с элементами искусственного интеллекта.* Если же задача имеет прямой алгоритм решения, который находит программист (может быть, за несколько попыток), но не сама программа, то искусственный интеллект отсутствует (возможно, присутствует естественный интеллект программиста). Но, фактически, непроходимой границы между этими случаями нет. Искусственная интеллектуальность программы не является дискретным свойством, принимающим только значения 0 или 1, всё или ничего. Напротив, искусственная интеллектуальность — это непрерывная величина, принимающая любые значения из интервала (0, 1). Чем больше петель обратной связи на рис. 1.8 проходит не через голову программиста, а через компьютер, тем выше мера интеллектуальности программы.

1.2.2. Знание и незнание

Явно выраженную границу между знанием и незнанием указать невозможно. На самом деле точной границы для представления знаний в компьютере нет — всякая программа и всякие данные отражают какие-то знания. С другой стороны, самые тонкие философские соображения можно (на естественном языке) записать в компьютер, и они будут там представлены бессмысленной последовательностью байтов. Байты присутствовать в компьютере будут, но философские знания?.. Точнее говоря*, ничто в компьютере*

не есть знания без нашей собственной интерпретации, и почти из всего можно извлечь знание, если как следует постараться. Более строго, везде в этой книге (если явно не оговаривается иное) мы подразумеваем следующую трактовку понятия «знание»:

Знания (компьютерные) *— это совокупность строго фиксированных информационных* ***структур*** *и вполне определенных* ***правил и стратегий*** *обработки этих структур (правил интерпретации).*

Рассмотрим это на модельном примере.

Допустим, мы хотим, чтобы компьютер отсортировал набор чисел в порядке их возрастания. Мы, люди, несомненно, знаем, как это сделать. Рассмотрим, можем ли мы научить это делать компьютер, то есть, можем ли мы представить необходимые знания в компьютере? Пусть нам заранее известно, что этих чисел три и они равны 3, 5 и 7. Можно научить компьютер решать данную задачу, написав всего один оператор:

```
Write (3, 5, 7)
```

Таким образом, мы представили в компьютер знания, достаточные для решения этой отдельной частной задачи. Знания представлены в форме одного оператора языка программирования.

Усложним задачу. Пусть чисел по-прежнему три, но при этом значение их произвольное. Составим процедуру, способную отсортировать эти числа по возрастанию:

```
proc Sort3 (x, y, z)
    if x > y then x :=: y end if
    if y > z then y :=: z end if
    if x > y then x :=: y end if
    Write (x, y, z)
end proc
```

Таким образом, мы научили компьютер решать довольно большое множество задач, связанных с сортировкой трех чисел по возрастанию, не слишком удлинив при этом запись. Во всяком случае, данная запись намного меньше, чем все возможные операторы

Write со всеми возможными комбинациями аргументов[16]. Знания представлены в форме процедуры с параметрами. Другими словами, механизм процедур в языках программирования — это уже метод представления знаний. Просто это достаточно старый метод, мы к нему привыкли, и не воспринимаем как механизм представления знаний.

Возникает еще один вопрос: откуда взялась процедура Sort3?

Можно сказать, что, как обычно, ее придумал программист. Но мы дадим не совсем привычный ответ, показав, как можно представить в программе знания, достаточные для того, чтобы компьютер сам придумал процедуру, подобную Sort3.

Так как в процедуре Sort3 есть фрагменты кода, похожие друг на друга, и их больше двух, следует применить специальный *рефакторинг* (преобразование текста программы, не меняющее выполняемую программой функцию), который называется запроцедуривание. *Запроцедуривание* — это такой рефакторинг, при котором повторяющиеся или похожие фрагменты программы заменяются вызовом вновь введенной процедуры. Переписав процедуру Sort3, введя процедуру Sort2, получаем процедуру Sort3'.

```
proc Sort3′ (x, y, z)
    Sort2(x, y)
    Sort2(y, z)
    Sort2(x, y)
    Write (x, y, z)
end proc
proc Sort2(u, v)
    if u > v then u :=: v end if
end proc
```

[16] Заметим, что поскольку количество различных чисел, представимых в компьютере, конечно, количество различных операторов Write с различными аргументами также конечно, хотя и довольно велико.

Но процедуру `Sort2` нетрудно специфицировать формально:

```
x, y { Sort2(x, y) } x < y
```

Здесь перед заголовком процедуры (который заключен в фигурные скобки) выписано предусловие, а после заголовка — постусловие. Процедура `Sort2` очень проста и очевидно полезна во многих случаях. Можно допустить, что эта процедура вместе со спецификацией (предусловие и постусловие) содержится в библиотеке готовых процедур, доступной транслятору.

Предусловие *— формальное утверждение о входных параметрах фрагмента на языке программирования (например, процедуры), которое должно быть выполнено до начала выполнения фрагмента для того, чтобы фрагмент проработал правильно.*

Постусловие *— формальное утверждение о входных и выходных параметрах фрагмента языка программирования, выполнение которого гарантируется после окончания выполнения фрагмента в случае, если было выполнено предусловие.*

Тогда, используя эту спецификацию, транслятор может логически вывести процедуру `Sort3'`:

```
x, y, z
{Sort2(x, y)} x < y, z
{Sort2(y, z)} y < z, x < z
{Sort2(x, y)} x < y < z.
```

Таким образом, мы немного усложнили механизм транслятора (допустив возможность логического вывода при трансляции) и получили систему автоматического синтеза программ. Другими словами, мы компактно записали не одну процедуру `Sort3`, а целый класс процедур. В этом случае знания представлены в виде специфицированной процедуры и алгоритма логического вывода, встроенного в транслятор.

Кажется, что понятия «знание» и «процедура» ортогональны. Если есть процедура, никакие знания не нужны — делай, как сказано в процедуре, и задача будет решена. Приведенный пример

показывает, что понятия «знание» и «процедура» (или, лучше сказать, алгоритм) не только не противоположны, но даже, можно сказать, неразделимы.

Любой алгоритм обязательно содержит знания, а любые знания представляются, в конечном счете, алгоритмами.

1.2.3. Алгоритмы поиска решения и представление знаний

Первый этап развития ИИ был связан с разработкой методов решения задач — алгоритмов поиска решения и построения логического вывода. В настоящее время это уже достаточно проработанный и изученный вопрос. Основные классические результаты изложены в третьей главе.

Однако для нетривиальных случаев трудоемкость поиска решения очень велика. Трудоемкость поиска решения — это быстрорастущая функция (в большинстве случаев растущая как экспонента или быстрее), аргументом которой является общий объем базы знаний. Рассматривая ниже конкретные алгоритмы, мы *доказываем точные оценки*, но здесь, на неформальном уровне, хотим *убедить* читателя в справедливости этого замечания. Пусть искомое решение является комбинацией конечного числа заранее заданных элементов, которых n. Тогда всего существует $2^n–1$ непустых комбинаций[17], если брать элементы без повторений, а если с повторениями, то еще больше. Заранее мы не знаем, какая комбинация является решением, в этом и состоит поисковый характер алгоритма, а значит, *в худшем случае* нам придется перебрать все комбинации!

Один из способов уменьшения трудоемкости поиска решения является изменение алгоритмов поиска с целью повышения их внутренней эффективности, не зависящей от решаемой задачи. Однако для очень многих задач, типичных для ИИ, доказана их *NP*-полнота. Построение эффективного (более эффективного, чем

---

[17] Это число непустых подмножеств множества из *n* элементов.

экспонента) алгоритма для них совершенно невероятно. Если для какой-то еще не исследованной задачи ИИ вдруг обнаруживается эффективный (скажем, полиномиальный) алгоритм, то задачу часто перестают квалифицировать как задачу ИИ, и переносят в область обычного программирования. Таким образом, в настоящий момент существенного улучшения в повышении внутренней эффективности алгоритмов поиска ожидать трудно.

Гораздо более продуктивным методом, позволяющим уменьшить трудоемкость поиска решения, является повышение эффективности представление знаний. Если выбор изощренного представления позволяет уменьшить объем базы знаний, то тот же стандартный алгоритм поиска становится значительно эффективнее в том смысле, что расширяются границы его применимости. Поясним это модельным числовым примером, сравнивающим два гипотетических способа представления знаний. Пусть одна и та же задача в первом представлении имеет размер n, а во втором – $n/2$. Пусть переборный алгоритм один и тот же, и имеет трудоёмкость 2^k, где k – размер задачи. Пусть, при этом, более компактное представление дается не даром: каждый шаг перебора выполняется в десять раз дольше и еще 100 единиц времени безвозвратно теряется на предварительную обработку. Фактически, мы сравниваем две функции: 2^n и $100 + 10*2^{n/2}$. При малых n первая функция меньше, но после $n = 8$ вторая функция становится меньше, и ее преимущество только возрастает с ростом n (рис. 1.9). Даже небольшой выигрыш в значении аргумента показательной функции перевешивает все другие проигрыши!

Компактность базы знаний — определяющий фактор эффективности искусственно-интеллектуальной системы и, следовательно, границ ее применимости. Именно это обстоятельство определяет структуру данной книги. Главы структурированы по методам представления знаний, а не по методам поиска решения.

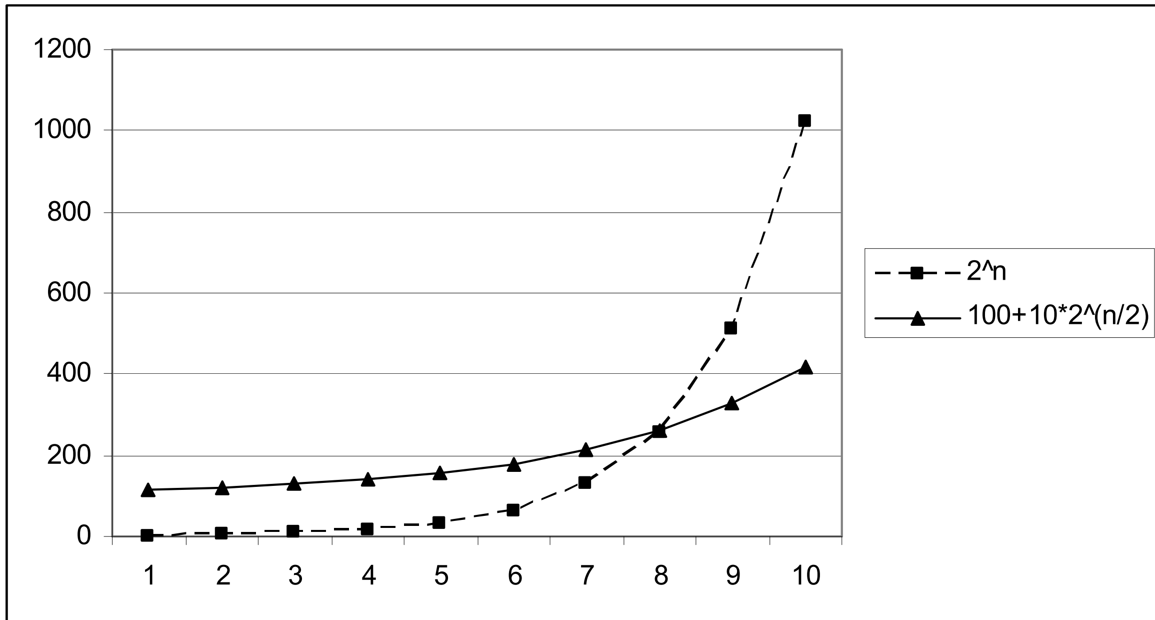

Рис. 1.9. Сравнение трудоемкости поиска в зависимости
от компактности базы знаний

При решении любой задачи в интеллектуальной области сначала следует выбрать представление знаний, потом подобрать подходящий алгоритм поиска решения. Дело в том, что если выбрано неудачное представление знаний, то никакие ухищрения с алгоритмами не помогут. Если же изобретено особо удачное представление знаний, то, скорее всего, самый обычный алгоритм поиска решения даст вполне удовлетворительные результаты.

1.3. ИСТОРИЯ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Экскурс в историю часто бывает совершенно необходим, иначе непонятно, почему рассматриваемые современные положения именно таковы, каковы они есть. Ведь положения и концепции ИИ носят во многом субъективный характер, объясняются историческими и психологическими причинами. Этим положения ИИ заметно отличаются от объективно обусловленных законов физики или химии. Мы разбили историю развития ИИ на несколько периодов (рис. 1.10):

- **доисторический**, когда даже термина «искусственный интеллект» еще не существо, но некоторые работы можно, ретроспективно и с некоторой натяжкой, отнести к данному направлению[18];

- **пионерский**, когда появление компьютеров и другие достижения техники XX века резко расширило горизонты возможных исследований, когда почти каждая предлагаемая идея ошеломляла своей новизной и открывала новые перспективы, причем идеи возникали как фейерверк, одна за другой;

- **академический**, когда направление ИИ оформилось как отдельное направление исследований в науке, со своей терминологией, авторитетами, исследовательскими центрами и образованием международного сообщества; когда исследования велись в основном в интересах развития самого направления, а практические приложения были интересны, прежде всего, самой принципиальной возможностью реализации, а не тактико-техническими характеристиками;

- **промышленный**, продолжающийся и развивающийся в наши дни, когда возросшие возможности техники открыли дорогу к массовому промышленному применению прикладных систем искусственного интеллекта.

Конечно, не следует думать, что эти периоды сменяли и сменяют друг друга строго последовательно, наподобие пятилетних планов развития социалистической экономики. И в пионерский, и в академический периоды были отдельные промышленные образцы, оставившие след в истории, и даже в наше меркантильное время случаются, хотя и не часто, научные прорывы, скачком ведущие к «сдвигу парадигмы».

---

[18] При появлении принципиально новых направлений, особенно новых направлений мысли, всегда стараются отыскать аналог или предтечу, причем, чем древнее историческая ссылка, тем уютнее новому направлению на первых порах.

1.3.1. Предыстория искусственного интеллекта

Достоверно зафиксированные **первые попытки создать машину, моделирующую некоторые аспекты человеческого разума**, предпринял Раймунд Луллий более 700 лет назад. Луллий полагал, что возможно открыть особый метод или искусство, посредством которого можно с разумной необходимостью вывести из общих понятий всякие истины, и, прежде всего — истины христианского вероучения. Основываясь на своих представлениях, он создал первую логическую машину в виде нескольких концентрических бумажных кругов, размеченных на отделения, в которых помещались общие понятия и категории. Вращая круги друг относительно друга, можно было получать различные более или менее сложные комбинации обозначенных понятий. Луллий полагал, что таким образом можно открыть новые реальные истины. Разумеется, никаких новых истин открыто не было, но идея, что рассуждение можно провести механически, прозвучала!

Сейчас мы знаем, что логические операции можно трактовать с алгебраической точки зрения и выполнять как арифметические, но до работ Дж. Буля (G. Boole) в XIX веке это было далеко не так очевидно. Поэтому внимание первых конструкторов интеллектуальных машин было направлено на построение машин, способных вычислять. Логические рассуждения казались более сложной задачей. Таких «вычислительных машин» было сконструировано несколько в XVII–XVIII веках, и здесь необходимо упомянуть арифметическую машину Лейбница, которая была первой в мире машиной, предназначенной для эффективного выполнения всех четырех действий арифметики.

Над этой машиной Г. Лейбниц начал работать в 1670 году; через два года он составил новое эскизное описание, на основе которого был, по-видимому, изготовлен тот экземпляр, который ученый продемонстрировал в феврале 1673 года на заседании лондонского Королевского общества. Лейбниц признал, что его инструмент еще

несовершенен, и обещал улучшить его, как только вернется в Париж. Действительно, в последующие годы он внес существенные усовершенствования в машину, но к ее окончательному варианту пришел лишь в 1694 году. Впоследствии Лейбниц еще несколько раз возвращался к своему изобретению, предлагая новые, усовершенствованные варианты.

Венцом предыстории компьютеров являются работы Чарльза Бэббиджа (Charles Babbage). Бэббидж начал с механического аппарата, предназначенного для автоматизации вычислений путём аппроксимации функций многочленами и вычисления конечных разностей. Возможность приближённого представления многочленами логарифмов и тригонометрических функций позволяет рассматривать эту машину как довольно универсальный вычислительный прибор. Эта *разностная машина* должна была уметь вычислять значения многочленов до шестой степени с точностью до 18-го знака. В том же 1822 году Бэббиджем была построена модель разностной машины, состоящая из валиков и шестерней, вращаемых вручную при помощи специального рычага. В следующем году правительство Великобритании предоставило ему субсидию в размере 1500 фунтов стерлингов. Разрабатывая машину, Бэббидж и не представлял всех технических и механический трудностей, связанных с её реализацией, и не только не уложился в обещанные три года, но спустя девять лет вынужден был приостановить свою работу. Однако часть машины все же начала функционировать, и производила вычисления даже с большей точностью, чем ожидалось.

В 1834 Бэббидж изобрел *аналитическую машину*. Она состояла из «склада» для хранения чисел (запоминающее устройство), «мельницы» для производства арифметических действий над числами (арифметическое устройство), «барабана», управлявшего в определенной последовательности операциями машины (устройство управления), а также устройств ввода и вывода данных. Регистровая память машины Бэббиджа была способна хранить как минимум

10 десятичных чисел по 40 знаков, теоретически же могла быть расширена до тысячи 50-разрядных. Арифметическое устройство имело аппаратную поддержку всех четырех действий арифметики. Машина производила сложение за 3 секунды, умножение и деление — за 2 минуты. На вход машины должны были поступать два потока перфокарт, которые Бэббидж назвал «операционными картами» и «картами переменных»: первые управляли процессом обработки данных, которые были записаны на вторых. Информация заносилась на перфокарты путем пробивки отверстий. Из операционных карт можно было составить библиотеку функций. Таким образом, все основные элементы архитектуры первых компьютеров были Бэббиджем предвосхищены в полной мере. Аналитическая машина так и не была реализована. Основная причина неудачи — технологическая. Технологии того времени не позволяли изготовить в достаточном количестве и достаточно точно необходимые сложные детали. Изобретатель писал в 1851 г.: «*Все разработки, связанные с аналитической машиной, выполнены за мой счет. Я провел целый ряд экспериментов и дошел до черты, за которой моих возможностей не хватает. В связи с этим я вынужден отказаться от дальнейшей работы*».

1.3.2. Пионерские исследования

Серьезное развитие искусственного интеллекта как научного направления началось в 40-х годах прошлого столетия, после появления ЭВМ. В 1948 году Норберт Винер опубликовал книгу «Кибернетика, или управление и связь в животном мире и машине», тем самым, основав новую науку — *кибернетику*.

Хотя в названии книги Винера мышление не упоминается, но одиозный вопрос «могут ли машины мыслить?»[19] впервые прозвучал в знаменитой статье Тьюринга уже два года спустя. (В этой статье, в

---

[19] Наш ответ таков: также как и люди. В некоторых случаях некоторые экземпляры могут, но далеко не все и не всегда.

частности, был впервые сформулирован конструктивный критерий наличия у машины ИИ, так называемый *тест Тьюринга*).

По отношению к этому вопросу мнения разделились не только у сторонних наблюдателей, журналистов и широкой публики, но и у самих ученых, начинавших работать в области ИИ. Конечно, в целом ответ у работающих исследователей был положительным. Тонкость заключается в том, что можно поставить два похожих вопроса:

Может ли машина мыслить как человек?

Может ли машина мыслить рационально?

Поиски конструктивного ответа на первый вопрос ведут к моделированию мышления человека, при котором считается важным не только получение ответа на интеллектуальный вопрос, но и анализ средств достижения ответа. Это направление получило название «нейрокибернетика».

Основным методом нейрокибернетики является математическое моделирование работы человеческого мозга. Еще в 1943 году американские ученые Уоррен Мак-Каллок и Уолтер Питтс предложили математическую модель работы нейрона человеческого мозга — *искусственный нейрон* — и способы комбинирования искусственных нейронов в нейронную сеть.

Затем последовали пионерские работы Розенблатта, Минского и других исследователей. Однако в 70-х годах работы в этом направлении несколько замедлились, что связано с невысокими количественными возможностями компьютеров того времени. Дело в том, что в мозгу человека параллельно работают от 10 до 14 миллиардов (10^9) нейронов, и каждый имеет в среднем около семи тысяч (10^3) связей с другими нейронами. Число искусственных нейронов в первых сетях измерялось сотнями, в лучшем случае тысячами, но столь большие количественные отличия (в миллионы раз) образуют качественную пропасть — на маленьких моделях невозможно исследовать большие системы. Заметим, что количественные показатели человеческого мозга до сих пор не

достигнуты в нейронных сетях[20], но масштабы постепенно становятся соизмеримыми

Отвечая на второй вопрос, мы приходим к кибернетике «черного ящика», когда наблюдаются только вход и выход, стимул и реакция, а внутреннее устройство «мыслящего агента» скрыто от наблюдения. В отличие от нейрокибернетики, для кибернетики «черного ящика» неважно, как устроено «мыслящие» устройство. Главное, чтобы на заданные входные воздействия оно реагировало так же, как человеческий мозг, то есть рационально.

Исследователи данного направления сразу же перешли от сверхсложных параллельных недетерминированных сетей нейрокибернетики к сравнительно простым алгоритмическим моделям, манипулирующим абстрактными символами, а не реальными сигналами. Именно поэтому мы предпочитаем термин *символический ИИ* (см. рис. 1.11).

Самой заметной пионерской работой этого направления была программа «Логик-теоретик», разработанная А. Ньюэллом, Г. Саймоном и Дж. Шоу в 1955–1956 годах. Программа работала в области автоматического доказательства теорем и смогла автоматически найти вывод для 38 из 52 теорем исчисления высказываний, приведенных в фундаментальной книги Рассела и Уйатхеда *Principia Mathematica*, причем в одном случае был найден более короткий, по сравнению с оригиналом, вывод.

По поводу этого несомненного достижения в литературе высказано столько различных, а иногда удивительных мнений и оценок, что мы считаем допустимым на примере программы «Логик-теоретик» дать свои выводы по универсальным программам автоматического доказательства теорем, а тем самым, по

---

[20] В рамках одного из лучших современных проектов Neurogrid построен тиражируемый нейрокомпьютер, который содержит 16 «нейроядер», каждое из которых содержит 65 536 программируемых искусственных нейронов, то есть всего около миллиона (10^6) нейронов.

универсальным интеллектуальным программам в целом (см. также параграф 1.1.3).

1. Автоматически доказанные теоремы не несут нового математического знания: в случае программы «Логик-теоретик» это простые технические теоремы исчисления высказываний. Их справедливость легко проверяется построением таблиц истинности. Задача состоит только в том, чтобы найти формальный вывод (последовательность формул) в заданной аксиоматике и с заданными правилами вывода. Поиск вывода не требует новых идей или методов, он опирается на вполне рутинные приемы. Среди обсуждаемых выводов нет выводов короче, чем в три применения правил вывода, и нет выводов длиннее, чем в две дюжины применений правил вывода. Результаты, полученные программой «Логик-теоретик», не имеют практической значимости.

2. Многолетние наблюдения автора за студентами, изучающими исчисление высказываний, и вынужденными в рамках учебного процесса искать и находить подобные выводы, позволяют утверждать, что средний человек «с улицы» без специальной подготовки не найдет ни одного из формальных доказательств, найденных программой «Логик-теоретик». Человек «с улицы» просто не поймет, что нужно сделать, потому что заведомо никогда ничего подобного не делал. Так что неподготовленный человек уступает в этом искусстве программе «Логик-теоретик». В то же время, прилежные студенты, после проведения трех-четырех сеансов упражнений с преподавателем, в состоянии найти любой из этих выводов, а также те выводы, которые программа найти не смогла. Следует оговориться, что выводы, найденные студентами, могут быть не такими элегантными, как у Рассела и Уайтхеда, и найдены будут не очень быстро. Но, тем не менее, можно утверждать, что человек, который в состоянии сдать вступительные экзамены в университет, также в состоянии научиться искать формальные выводы не хуже, чем это может делать программа «Логик-теоретик».

3. С качественной точки зрения, результаты, достигнутые программой «Логик-теоретик», за прошедшие полвека с лишним, были улучшены незначительно. Дело в том, что задача автоматического поиска вывода в исчислении высказываний имеет переборный характер (см. параграф 1.1.3). Современные компьютеры в миллионы и миллиарды раз мощнее того компьютера, которым пользовался Шоу, однако современные универсальные программы поиска логического вывода могут построить только в десятки, от силы в сотни раз более сложные выводы, поэтому на качественном уровне первые два замечания в этом списке остаются в силе и в наши дни, и применимы не только к программе «Логик-теоретик», а к любой современной программе, которая претендует на автоматическое доказательство произвольных теорем в достаточно богатом исчислении.

1.3.3. Становление и развитие

В 1956 году в Дартмутском колледже (один из старейших университетов США) был проведен двухмесячный семинар, в котором приняло участие десять ведущих исследователей нового направления:

- Джон Маккарти (John McCarthy), Дартмутский колледж;
- Марвин Минский (Marvin Minsky), Гарвардский университет;
- Клод Шеннон (Claude Shannon), Bell Laboratories;
- Натаниэль Рочестер (Nathaniel Rochester), IBM;
- Артур Самюэль (Arthur Samuel), IBM;
- Аллен Ньюэлл (Allen Newell), университет Карнеги-Меллон;
- Герберт Саймон (Herbert Simon), университет Карнеги-Меллон;
- Тренчард Мур (Trenchard Moore), Принстонский университет;
- Рей Соломонов (Ray Solomonoff), Массачусетский технологический институт;
- Оливер Селфридж (Oliver Selfridge), Массачусетский технологический институт.

Этот семинар принято считать началом искусственного интеллекта как самостоятельного направления, именно на нем был предложен и утвержден сам термин *искусственный интеллект* (artificial intelligence). Никаких особенных научных результатов Дартмутский семинар не дал. На нем произошли другие важнейшие события: искусственный интеллект был признан самостоятельной отраслью науки, были продемонстрированы первые впечатляющие результаты (подобные программе «Логик-теоретик») и было образовано научное сообщество в результате личного знакомства лидеров.

После этого исследования в области ИИ стали вестись широким фронтом, и не только в США. Стали появляться первые публикации, в которых делались попытки обобщения накопленного материала. Среди специалистов, выступивших с такими работами, были Дж. Маккарти, М. Минский, Э. Фейгенбаум (E. Feigenbaum, США), Д. Мичи (D. Michie, Великобритания), А. А. Ляпунов и В. М. Глушков (СССР). Разумеется, результаты не заставили себя ждать — было предложено множество идей, большинство из которых используются и в наши дни.

В конце 1950-х гг. была предложена модель поиска на графе. В рамках данного подхода, задача представляется в виде поиска пути в графе, вершинами которого являются состояния задачи, а дугами — все возможные допустимые преобразования. Решение задачи состоит в поиске пути от исходного состояния к целевому. В начале 1960-х гг. получило широкое распространение эвристическое программирование (см. главу 3). Эвристикой называется правило, обычно полученное эмпирическим путем, позволяющее сократить объем перебора в пространстве решений. Эвристическое программирование — разработка стратегий поиска решения на основе заранее известных эвристик. В конце 1960-х гг. были разработаны общие методы автоматического доказательства теорем в исчислениях первого порядка (в частности, метод резолюций Дж. Робинсона

(J. Robinson) и обратный метод С. Ю. Маслова, см. главу 4), и на их основе в начале 1970-х гг. были предложены многочисленные инструментальные средства программирования решения интеллектуальных задач. Так, в 1973 г. был создан язык Пролог, который является популярным средством логического программирования до сих пор. Практически весь запас идей и концепций, которыми оперируют в ИИ сейчас, был предложен или обозначен в этот период. Перечислить все идеи невозможно, да в этом и нет нужды, если ясна общая схема развития (см. рис. 1.10**)**.

1.3.4. Научная консолидация и промышленное внедрение

С нашей точки зрения, наиболее важным для ИИ событием 80-х гг. было появление концепции инженерии знаний. Эта концепция была выдвинута Э. Фейгенбаумом в 1982 г.

Инженерия знаний *(knowledge engineering) — область искусственного интеллекта, которая изучает методы и средства извлечения, представления, структурирования и использования знаний.*

Отделение методов представления знаний от методов поиска решения, построение все более эффективных специализированных приемов представления знаний в конкретных предметных областях позволило получать достаточно эффективные представления, чтобы имеющимися методами на реальных компьютерах решать практически значимые задачи. Одновременно с этим, обретя общий методологический базис, различные подходы и направления внутри ИИ стали теснее интегрироваться друг с другом, поскольку не нужно было доказывать «чистоту теории» или «оригинальность подхода», а нужно было любым способом получить практический результат.

В этот период стали появляться компьютерные системы, которые были весьма далеки от прохождения теста Тьюринга, зато положительно отвечали на такие вопросы: может ли машина решать некоторые интеллектуальные проблемы дешевле, чем человек? Быстрее, чем человек? Безотказнее, чем человек?

Возможности вычислительной техники заметно возросли, и стало ясно, что если оставить в стороне философские, этические и моральные вопросы «искусственного разума», то во многих конкретных узкоспециальных областях не только можно, но и выгодно заменить человека машиной. Прикладные системы искусственного интеллекта стали приносить прибыль. Немедленно стали увеличиваться инвестиции в академические исследования, которые стали давать новые результаты, которые внедрялись в промышленность, которая приносила доход, который частично вкладывался в исследования… Возникла положительная обратная связь, и период подъема ИИ еще не завершен!

Вторым значимым событием этого периода стало возрождение, после определенного затишья, нейрокибернетики с концепциями нейронных сетей и нейрокомпьютеров. Возобновление интереса не было вызвано каким-либо внезапным чудесным откровением свыше с популярным разъяснением, как именно устроен и работает мозг человека. Конечно, физиологи, психологи, лингвисты, философы и другие исследователи не сидели, сложа руки, а узнали и придумали про нашу человеческую природу много нового, но причина была не в этом. Внутри самой нейрокибернетики также не было сделано эпохальных открытий. В современных нейросетях используются математические модели, очень близкие к первоначальным, хотя, конечно, во многом развитые и усовершенствованные. Очевидная причина возрождения нейрокибернетики заключается в поразительных успехах микроэлектроники. В пионерских работах, подобных персептрону Розенблатта, количественные различия между искусственными и естественными нейронными сетями измерялись многими миллиардами раз, и ни о каком качественном сопоставлении речи не было. Сейчас количественные различия измеряются тысячами раз, и у конструкторов супернейрокомпьютеров появились невиданные ранее возможности. Следует ожидать качественных

изменений, к которым должны привести происходящие количественные изменения.

Третьим примечательным событием уже сравнительно недавнего прошлого мы считаем привнесение заслуженного философского понятия «онтология» на свежую почву информатики.

Онтология *(ontology) — это формализации некоторой области знаний с помощью концептуальной схемы. Обычно такая схема состоит из структуры данных, содержащей все релевантные классы объектов, их связи и ограничения, наложенные на объекты и связи.*

Онтологии используются в процессе программирования как форма представления знаний о реальном мире или его части, иначе говоря, как *модель предметной области*. Хотя термин «онтология» изначально философский[21], в информатике он принял самостоятельное значение. Здесь есть два существенных отличия:

- онтология в информатике является формой представления знаний, причем такой формой, которую компьютер сможет легко обработать;

- информационные онтологии создаются всегда с конкретными целями решения практических задач; они оцениваются больше с точки зрения применимости, чем полноты, замкнутости и корректности.

Автор совершенно убежден, что анализ, построение, развитие и использование моделей предметной области — самая важная и перспективная задача информатики в настоящее время, причем не только в ИИ, но и в других информационных технологиях. Достаточно упомянуть, например, концепцию систем управления бизнес-процессами (Business Process Management Systems — BPMS) в области информационно-управляющих систем, концепцию семантической паутины (Semantic Web) в Интернете или концепцию

[21] Онтология (древнегреческие корни «онто» — сущее, то, что существует и «логия» — учение, наука) — раздел философии, изучающий бытие.

проблемно-ориентированных языков (Domain Specific Languages) в прикладном программировании.

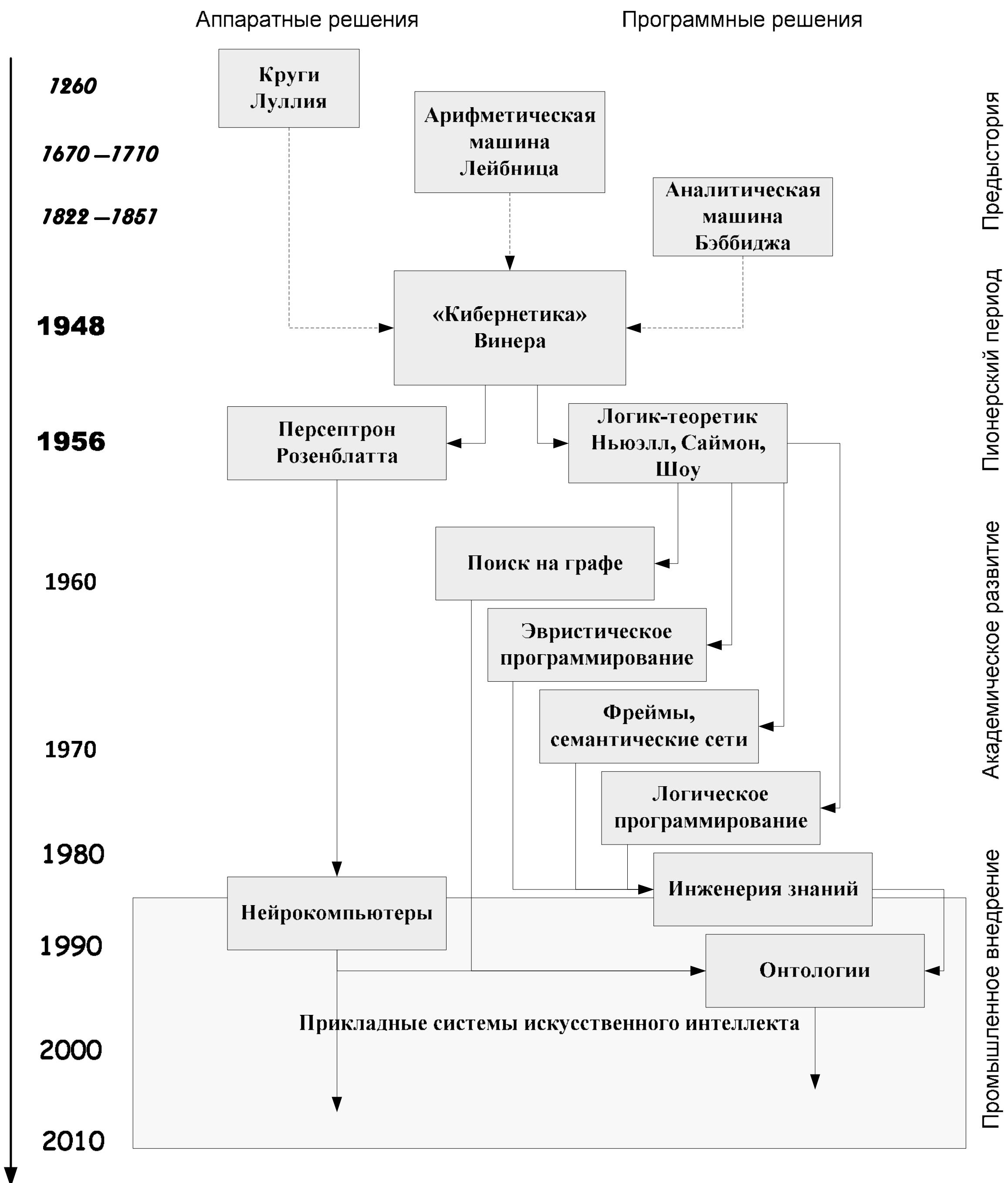

Рис. 1.10. История развития искусственного интеллекта

Все эти перспективные концепции апеллируют к понятию онтология[22].

Рис. 1.10 отражает наше видение истории развития ИИ и подводит итог разделу.

1.4. КЛАССИФИКАЦИИ ПРИКЛАДНЫХ СИСТЕМ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Прикладные системы и области, в которых используется искусственный интеллект, весьма разнообразны — раздел 1.1 дает только небольшой набор учебных примеров, но не энциклопедический обзор. На самом деле в реальной практике применяются тысячи прикладных систем ИИ, и каждая имеет свои особенности и отличительные черты. Такое разнородное собрание объектов и методов нуждается в классификации, даже в нескольких классификациях, одной не обойтись.

1.4.1. Виды знаний

Знания, которые мы собираемся представлять в компьютере, весьма разнообразны.

Следуя С. С. Лаврову [4], знания можно разделить на три принципиально разных вида:

1. *Понятийные, концептуальные знания*. К этой группе относятся понятия и взаимосвязи между ними, например, правила, законы, инвариантные соотношения, а также *эвристические знания* — неформальные правила рассуждений, отражающие практический опыт решения задач в проблемной области.

2. *Процедурные алгоритмические знания*. Сюда можно отнести различные умения, технологии, процедуры, алгоритмы, их

---

[22] Вместе с тем, нельзя не отметить, что апологеты концепции онтологии в информатике пока обещают больше, чем реально могут сделать. В этом они уподобляются пионерам ИИ 50-х гг., которые обещали появление «искусственного разума» через 10–20 лет. Завышенные обещания ведут к охлаждению интереса…

программные реализации и типовые процедуры решения задач — т. е. те формы знаний, которые фактически всегда явно используются для компьютерного решения любой задачи.

3. *Фактографические знания*. Это самые разнообразные количественные и качественные характеристики конкретных объектов, т. е. факты. Например, различные данные в базах данных, константы в программах и т. д.

Конечно, между видами знаний нет непроходимой границы (см. также параграф 1.2.2). Более того, иногда бывает так, что знание трудно отнести к одному конкретному виду. Например, алгоритмы вычисления математических функций (sin, cos, и т. д.) для эффективности представляют в виде набора коэффициентов разложения по специальным полиномам. Эти коэффициенты, конечно, являются данными, то есть фактографическим знанием, а используются как алгоритмическое знание. Тем не менее, указанная классификация знаний очень полезна и действенна при обсуждении прикладных систем искусственного интеллекта, поскольку позволяет сосредоточиться на важных для ИИ вопросах представления концептуальных знаний, используя приемы представления алгоритмических и фактографических знаний как готовый базис.

Классификация знаний по Лаврову принимается как основополагающая в этой книге.

Лавров Святослав Сергеевич, 1923–2004, — российский ученый, член-корреспондент РАН. Автор трудов по механике, автоматическому управлению, вычислительной математике и программированию. Лауреат Ленинской премии 1957 года. Научная биография Лаврова в определенном смысле уникальна. Совсем в молодом возрасте он стал основоположником ракетно-космической баллистики в СССР и неоспоримым авторитетом в области динамики управляемого полета и автоматического управления. Появление цифровой вычислительной техники привело к резкому повороту в

деятельности С. С. Лаврова и в течение нескольких лет сделало его классиком программирования в СССР.

1.4.2. Классификация по степени использования различных видов знаний

В зависимости от того, какие виды знаний используются, и каким образом это происходит, системы искусственного интеллекта можно классифицировать несколькими способами.

1. Если в программной системе используются только фактографические знания, то такую программную систему называют *базой данных* (БД) и в современных условиях обычно не считают содержащей искусственный интеллект.[23]

2. Если в программной системе используются главным образом алгоритмические знания, то такую программную систему принято называть *пакетом прикладных программ*. Отнесение такого пакета программ к системе ИИ обычно явно указывается. Если пакет прикладных программ используется просто как библиотека процедур, то его не относят к искусственно-интеллектуальным системам. Если же в нем используются методы ИИ, например, если программу решения конкретной задачи строит не пользователь, а автоматический планировщик, то такой пакет прикладных программ может считаться системой с искусственным интеллектом.

3. Если в программной системе в той или иной форме используются *концептуальные знания*, то такую систему считают искусственно-интеллектуальной.

Заметим, что, как правило, развитые пакеты прикладных программ имеют в своем составе или активно используют внешнюю базу данных, а прикладные системы ИИ почти всегда используют как пакеты программ, так и базы данных (рис. 1.11). Например, развитая шахматная программа имеет огромную базу данных дебютов и

---

[23] Следует заметить, что такие современные технологии баз данных, как *извлечение знаний из данных* (data mining), скорее следует относить к области ИИ, чем к области систем управления базами данных.

эндшпилей, специальный пакет программ для работы с позициями и ходами во время партии, и набор стратегических правил выбора хода, которые относятся к понятийным знаниям.

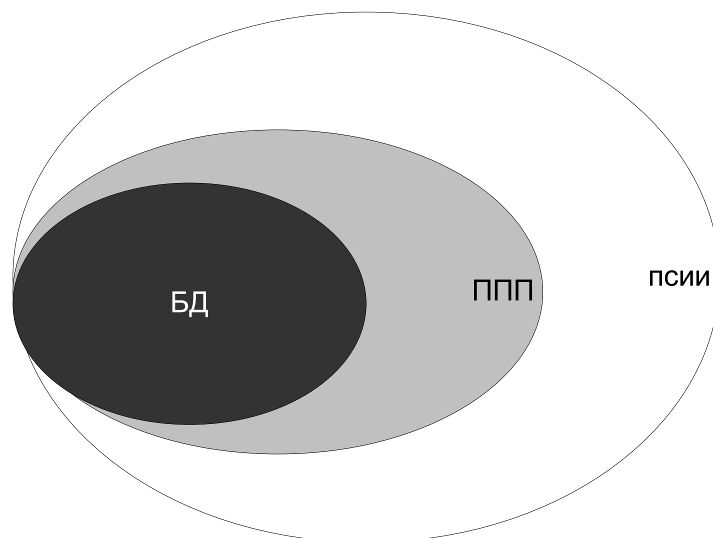

Рис. 1.11. Классификация прикладных систем с элементами искусственного интеллекта по типам используемых знаний

Предметом этой книги являются именно прикладные системы искусственного интеллекта. При этом мы считаем, что читатель в достаточной степени знаком с концепциями пакетов программ и баз данных, поэтому все внимание концентрируется на представлении понятийных знаний.

1.4.3. Классификация по форме представления знаний

Когда осознана и прочувствована необходимость совместного использования знаний всех трех типов, возникает потребность в выборе способа их представления в компьютере. Мощные механизмы, которые использует человек для представления знаний: естественные языки, математика, визуальные образы, — пригодны для представления всех видов знаний, но непригодны для реализации в компьютере. Точнее говоря, наоборот, современные компьютеры непригодны для использования этих механизмов — компьютеры слишком примитивны и неинтеллектуальны.

Таким образом, *возникает необходимость изобретения таких способов представления знаний, которые, с одной стороны, позволяли бы представлять знания всех видов, и, с другой стороны, были бы реализуемы в современных компьютерах*.

Механизмы для представления фактографических и алгоритмических знаний в компьютерах хорошо изучены и устоялись. Для фактографических знаний — это системы управления базами данных (СУБД), в современных условиях чаще всего реляционные. Для алгоритмических знаний — это библиотеки процедур, в современных условиях чаще всего объектно-ориентированные, то есть библиотеки типов и классов.

Таким образом, классификация по форме представления знаний рассматривает, прежде всего, представление концептуальных знаний.

Предлагается множество различных механизмов представления концептуальных знаний. Чаще всего сейчас применяются следующие три (рис. 1.12).

Правила, или продукции. Системы искусственного интеллекта, основанные на правилах, в настоящее время встречаются чаще всего. С математической точки зрения такие системы используют модели вычислимости, близкие к нормальным алгорифмам Маркова. Системы продукций детально рассматриваются во второй и третьей главах данной книги.

Формулы логических исчислений. Системы искусственного интеллекта, основанные на логических формулах, в настоящее время занимают второе место по распространенности. С математической точки зрения такие системы используют исчисления, близкие к исчислению предикатов первого порядка, а в некоторых случаях близкие к λ-исчислению Чёрча. Представление знаний с помощью формул логических исчислений рассматривается в четвертой главе.

Фреймы и семантические сети. Сейчас этот механизм используется сравнительно редко, хотя в недавнем прошлом это было

не так. С математической точки зрения такие системы используют аппарат теории графов.

Все три распространенных механизма сопоставимы с точки зрения выразительной силы и других общематематических свойств. Предпочтение тому или иному механизму представления знаний часто отдается исходя из соображений привычности, удобства программирования, знакомства с теорией разработчиков прикладной системы. Особенно важным является то, насколько компактным оказывается представление знаний о конкретной задаче, см. параграф 1.2.3.

Рис. 1.12. Механизмы представления различных видов знаний

1.4.4. Классификация по виду ответа при решении задач

Решая конкретную задачу, прикладная система ИИ получает на входе знания в той или иной форме, а на выходе выдает ответ, который также имеет некоторую форму и представляет собой

некоторое (новое) знание. В соответствии с введенной классификацией видов знаний искусственно-интеллектуальные системы можно классифицировать по уровню выдаваемого ответа.

0. **Логический ответ** (да или нет); не очень удобны в использовании, сейчас применяются редко.

1. **Фактографический ответ** (ответ-факт); если выдается конкретный ответ на один вопрос, то такие системы часто называют *информационными системами.* В настоящее время наиболее распространенный класс систем.

2. **Процедурный ответ**; решая задачу, система может создать и запустить процедуру (*система синтеза программ*, *автоматическое программирование*). Очень интересное и перспективное направление, привлекающее внимание большого числа исследователей.

3. **Понятийный ответ**; ответ-закон, строится не решение, а схема решения класса задач, может быть даже на компьютере не выполнимая. В настоящее время полномасштабные реализации пока неизвестны.

Приведем примеры ответов разных уровней, используя модельный пример с сортировкой чисел из параграфа 1.2.2 (напомним, задача состоит в следующем: мы хотим, чтобы компьютер отсортировал набор из трех чисел в порядке их возрастания).

(0) 2 5 7 → да
 5 2 7 → нет

(1) 5 2 7 → 2 5 7

(2) $\{x, y, z\}?\{x < y < z\}$ → `Sort3`

(3) $\{x_i\}$ **отсортировать** → $\forall\ i, j\ (\ i < j \rightarrow x_i < x_j\ )$

На уровне 0 мы предъявляем последовательность, а система только проверяет, отсортирована она или нет. На уровне 1 мы предъявляем последовательность, и система ее послушно сортирует. На уровне 2 система строит процедуру сортировки, примерно так, как это показано в параграфе 1.2.2. Наконец, на уровне 3 гипотетическая

система объясняет нам, что это значит — отсортировать массив чисел[24].

1.4.5. Классификация по степени универсальности

Вначале каждая новая идея представления знаний рождается, будучи привязанной к конкретной предметной области. Дело в том, что в области ИИ принято публикацию новых идей сопровождать хотя бы одним конкретным практическим примером, демонстрирующим преимущества новинки. Если идея хороша, она находит новые применения в смежных областях, вырастая в проблемно-ориентированный метод. В дальнейшем возникает соблазн сделать систему представления знаний как можно более универсальной для решения самых разнообразных задач. Однако это стремление практически бесполезно, поскольку с ростом универсальности резко падает применимость таких представлений знаний.

Более точно поясним используемую нами терминологию[25].

Универсальность *(данной системы представления знаний в данной предметной области) — доля задач предметной области, которые можно представить.*

Таким образом, *мы называем* ***универсальным*** *(в данной предметной области) такой способ представления знаний, который позволяет сформулировать все задачи.*

Применимость *(данной системы представления знаний в данной предметной области) — доля представимых задач предметной области, которые удается решить.*

---

[24] В настоящее время программной системы, которая могла бы дать ответ на вопрос, «что значит отсортировать массив?», видимо, не существует. Однако, как велика доля людей, которые могут дать ответ на этот вопрос, не прибегая к помощи жестов и примеров?

[25] Эта терминология не является пока общепринятой, это наше предложение.

Таким образом, *мы называем* ***всюду применимым****, или* ***тотальным*** *(в данной предметной области) такой способ представления знаний, который позволяет решить все задачи с учетом реальных количественных ограничений*. Понятно, что для достаточно широких предметных областей универсальные и одновременно тотальные системы представления знаний вряд ли возможны. Более того, наблюдения показывают, что между универсальностью и применимостью имеет место обратная зависимость (рис. 1.13)[26].

Концепция представления мира дискретными состояниями и поиска последовательности преобразований из исходного состояния в целевое удивительно широко применима. Алгоритм поиска на графе применим во всех случаях. Однако за эту универсальность приходится платить сугубой неэффективностью поиска.

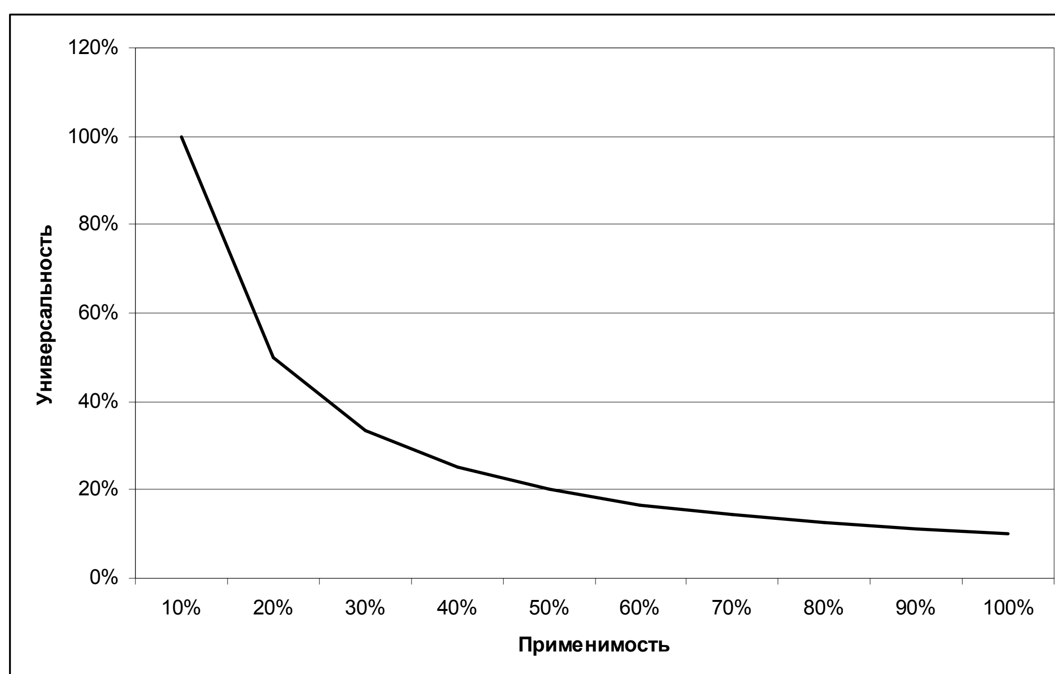

Рис. 1.13. Зависимость между универсальностью и применимостью

[26] Мы изобразили график обратно-пропорциональной зависимости, в качестве примера, поясняющего качественную картину. Его нельзя использовать для количественных оценок.

Заметим, что на практике ни в базах данных, ни в пакетах прикладных программ, ни в прикладных системах искусственного интеллекта, как правило, не ставится цель достижения универсальности. Обычно ставится цель увеличения применимости, даже ценой снижения универсальности. Причины достаточно просты. Во-первых, от предметной области зависят способы манипулирования знаниями, и только учитывая особенности предметной области можно надеяться построить компактное представление знаний, которое даст эффективное решение задач и повышает применимость. Во-вторых, в универсальных системах слабы метазнания (стратегия управления), т. е. знания о том, в каком порядке целесообразно применять конкретные правила, что важно, а на что можно не обращать внимания, решая задачу.

1.4.6. Классификация по архитектуре и инструментальным средствам

При анализе архитектуры прикладных систем с элементами искусственного интеллекта логично выделить:

- *объекты*, с которыми манипулирует система (другими словами, что дано);
- *действия*, посредством которых эти манипуляции производятся (что можно сделать с этими данными);
- *стратегию управления*, которая определяет когда и какие манипуляции следует производить (в каком порядке, к каким объектам, какие действия нужно применять).

Такое представление об архитектуре искусственно-интеллектуальной системы не догма — оно возникает при анализе *типичных* современных систем искусственного интеллекта. Узкоспециализированные системы иногда имеет другую архитектуру. Имеется виду логический анализ, а не анализ на уровне машинных кодов, где архитектурная структура уже неразличима.

Сейчас про эти три компонента (особенно, если они четко разделены в программной реализации) говорят: данные (свойства

объектов), алгоритмы (методы объектов в терминах объектно-ориентированного программирования) и *метазнания*.

Уже первые работы в области ИИ выявили потребность в специальных инструментальных средствах для разработки прикладных систем. Это связано с целым рядом особенностей программной реализации искусственно-интеллектуальных систем, часть их которых мы обсуждаем в третьей главе. Как бы то ни было, для разработки прикладных систем искусственного интеллекта применялись и применяются разные языки программирования и инструментальные средства и это не может не накладывать отпечаток на конкретную систему. Использованное при разработке системы ИИ инструментальное средство является классификационным признаком.

ВЫВОДЫ

1. Искусственный интеллект является научным направлением, имеющим явно выделенный предмет — моделирование мышления, понимаемого как рациональное поведение, использующее целый ряд методов информатики и математики.

2. История развития искусственного интеллекта свидетельствует, что период становления нового направления завершен и в настоящее время имеет место период практического применения.

3. Анализ достижений искусственного интеллекта показывает, что наиболее существенными для практического применения являются методы представления знаний.

2. ПРЕДСТАВЛЕНИЕ ЗНАНИЙ СИСТЕМАМИ ПРОДУКЦИЙ

Представление знаний системами продукций (Production Rules Knowledge Representation) является одним из наиболее часто используемых механизмов представления знаний в интеллектуальных системах. Этот механизм концептуально прост, его идея естественно воспринимается и легко реализуется программно.

На самом деле, нет резкой границы между прикладными системами с искусственным интеллектом и другим программным обеспечением. Системы продукций — это просто еще один способ организации выполнения компьютером некоторого процесса обработки информации. *Этот способ совершенно не похож на привычное процедурное и объектно-ориентированное программирование, но обладает не меньшими возможностями*.

2.1. СИСТЕМЫ ПРОДУКЦИЙ

Можно предложить несколько определений *систем продукций* (или *продукционных систем*, или *систем, основанных на правилах*)[27], рассматривая их с различных точек зрения. В конечном счете, различные определения ведут к одному и тому же объекту, хотя и

---

[27] Не только в русском языке системы продукций обозначаются многими, почти синонимичными, но все-таки различными способами. В английском языке также применяют множество различных, хотя и похожих словосочетаний. Например, в выражении «[production] [rules] [based] [inference] systems» почти любая пара или даже тройка слов, стоящих в квадратных скобках, может быть допустимым термином, имеющим прямое отношение к системам продукций. Поэтому при поиске информации на эту тему в Интернете мы рекомендуем провести поиск по нескольким различным словосочетаниям.

подходят с разных сторон и различаются в некоторых деталях. Мы рассмотрим три определения:

- системы продукций с точки зрения их структуры;
- системы продукций с алгоритмической точки зрения;
- системы продукций как логические исчисления.

Прежде чем переходить к систематическому изложению приемов и методов работы с продукциями как с информационными единицами, мы считаем необходимым навести некоторый порядок в сопутствующей терминологии, систематизировать разнообразные интерпретации и явным образом «расставить по местам» возможные области применения.

2.1.1. Терминологические соглашения и содержательная интерпретация

В названии параграфа присутствует слово «содержательная», которое обозначает сознательное отступление автора с позиций «строгости» и «алгоритмичности» в представлении материала, но это отступление временное — *в пределах данного параграфа*. Объясняется отступление желанием авторов подчеркнуть существующую в обсуждаемой области «неразбериху» в терминах, представлениях и обоснованиях, что, в свою очередь, является почти необходимым признаком «молодости» направления исследований и интенсивности его развития.

Содержательно (с позиций здравого смысла и естественного языка) с термином «продукция» чаще всего ассоциируют сложноподчиненное предложение условного вида «Если *посылка*, то *заключение*». Не вдаваясь в детали, зафиксируем пока именно эти термины: *посылка* и *заключение*. Опять же, без дополнительных уточнений, для удобства изложения будем записывать продукцию в символической форме

$$p \to q, \qquad (*)$$

где p и q представляют посылку и заключение, соответственно. Обращаем внимание читателя на то, что в символической записи, в отличие от словесного примера, явно зафиксирован третий и очень важный элемент продукции — *связка*, обозначенная стрелочкой. Каждый из трех элементов продукции может и должен иметь некоторую смысловую нагрузку (интерпретацию, семантику, означенность и т. д. — в разных литературных источниках используются различные варианты) для того, чтобы смысл можно было сопоставить всей продукции и соответствующим образом использовать её (продукцию) при решении конкретной проблемы. В зависимости от комбинации выбранных в каждом конкретном случае интерпретаций, может существенно изменяться применимый математический базис, а также алгоритмы и стратегии работы с продукциями. В то же время внимательный и дотошный читатель должен заметить, что вне зависимости от принимаемых интерпретаций *любая продукция определяет некоторое правило поведенческого или преобразующего характера* в конкретной предметной области.

Несмотря на заявленную в этом параграфе «содержательность», мы не станем анализировать всех возможных комбинаций семантических нагрузок, а ограничимся рамками тематики, определённой названием и назначением книги.

Продукции естественным образом можно разбить на два непересекающихся класса: *однородные* и *неоднородные*. Это разбиение определяется способом задания интерпретации посылки и заключения продукции. В первом случае p и q получают «схожий» смысл (слова в некотором общем алфавите, числовые значения, логические выражения и т. д.). Ко второму виду мы отнесем продукции, в которых посылка и заключение интерпретируются по разному (условие – действие, состояние среды – переход, стимул – реакция, другие сочетания).

Напомним еще один важный момент — все механизмы и приемы, составляющие содержание этой книги, предназначены для описания и анализа изменений в некоторой конкретной предметной области. Фиксированную ситуацию в предметной области мы, как правило, называем *текущим состоянием* (*окружения* или *окружающей среды*). Множество всех возможных состояний мы всегда (если явно не оговаривается иное) будем обозначать буквой ***D***, а отдельные состояния из этого множества будем обозначать буквами *d* с индексами и полагать, что это состояние описывается набором конкретных значений (утверждений) чего-либо (о чем-либо). Способ хранения (организации) текущего состояния *d* не принципиален (список, массив, набор таблиц). Механизм хранения текущего состояния мы также обозначаем буквой ***D*** (всегда легко отличить по контексту, о чем речь — о множестве состояний или о механизме представления множества в программе) и называем «*база фактов*» (реже — «*база данных*», а иногда «*пространство поиска*»).

Множество всех возможных состояний базы данных ***D*** *называется* ***пространством поиска*** *(search space).*

Отражая динамику развития (изменения ситуации) анализируемой области, мы также можем говорить о переходе из одного состояния в другое, при этом мы предполагаем наличие начального (обязательно!) и конечного (не всегда) состояний. Последовательность переходов, ведущих из начального состояния в конечное состояние, мы называем выводом в пространстве поиска, а акт перехода из одного состояния в непосредственно вытекающее — шагом вывода.

В табл. 2.1 мы приводим типичные примеры систем однородных и неоднородных продукций, указывая предметную область, формальную теорию, выступающую теоретическим базисом, расшифровывая формулу (*), а также, фиксируя цель и (при наличии) возникающие проблемы. Кое-где мы приводим ссылки на достойные

с нашей точки зрения источники (в том числе, на параграфы данной книги).

Таблица 2.1

Примеры систем продукций

Предметная область и ссылка	Математический базис и цель	Тип и формат продукции
Синтаксический анализ [5]	Нормальные алгорифмы Маркова, автомат-распознаватель	Однородная, цепочка → цепочка
Компьютерная алгебра	Системы подстановок термов, редукция выражений	Однородная терм → терм
Вычислительные модели	Интуиционистское исчисление высказываний, программа	Однородная, аргументы → → результаты
Системы логического вывода [6]	Исчисление предикатов первого порядка, доказательство теоремы	Однородная, предикаты → → предикаты
Экспертные системы (параграф 1.1.3)	Нет базиса, диагноз	Неоднородная, симптомы → диагноз
Системы управления	Нет базиса, гомеостаз или иная цель управления	Неоднородная, состояние + стимул → → реакция
Вопрос-ответные системы, (параграф 1.1.2)	Нет базиса, поддержка диалога	Неоднородная, вопрос + шаблон → → ответ

Мы видим, что для неоднородных систем нет общепринятого математического базиса. В лучшем случае, обоснованием систем второго класса могут выступить охраняемые команды Дейкстры [7], которые, впрочем, так и не стали строгой формальной теорией.

2.1.2. Структура системы неоднородных продукций

Генезис систем продукций. Как мы уже отмечали ранее, все (или почти все) прикладные системы с элементами ИИ, в той или иной степени, отражают (моделируют) *некоторые аспекты человеческого мышления*. Системы продукций не являются в этом

отношении исключением. Первоначально подход использовался при построении так называемых экспертных систем (см. параграф 1.1.3), имитирующих поведение эксперта в некоторой предметной области при разрешении разного рода ситуаций, будь то установление диагноза пациенту, выработка заключения о целесообразности разработки месторождения полезных ископаемых (на основании данных геологоразведки) или что-то иное. Человек, как правило, решая подобные задачи, выстраивает цепочки силлогизмов[28]. Подобный стиль мышления в быту часто называют дедуктивным, именно им блестяще владел и плодотворно его использовал один из самых великих литературных сыщиков — эксперт в области раскрытия преступлений Шерлок Холмс.

Оговорка «в быту» сделана не случайно — мы хотели подчеркнуть элементы субъективности экспертного заключения, получаемого человеком. Эта субъективность обуславливается множеством факторов:

- полнотой и подробностью анамнеза[29] (используя медицинскую терминологию) — кашель курильщика может вызывать подозрение онколога, а у некурящего спортсмена, скорее всего, банальное ОРЗ;

- опытностью эксперта — некоторые знают, что чайки, мирно гуляющие по берегу, предвещают шторм, другим неизвестно даже, что те же чайки, плавающие в море, гарантируют тихую и спокойную погоду;

- умением применить знания в наиболее подходящем порядке (использовать наилучшую стратегию вывода!) и т. д.

---

[28] *Силлогизм* — умозаключение, в котором на основании нескольких суждений с необходимостью выводится новое суждение, называемое заключением [8].

[29] *Анамнез* — описание прошлой жизни больного и история его болезни с самого начала ее, притом как внешних (объективных) проявлений ее, так и всех ненормальных ощущений, которыми она сопровождалась [8].

Вывод часто бывает не категоричным (неоднозначным) — «Пациент скорее жив, чем мёртв» (медицинский анекдот, а может и не анекдот). Нередко случается, что два разных эксперта из одних и тех же посылок строят различные (порой, противоречащие один другому) выводы – см. пример «ортогональной интерпретации» о болгарине и его девушке (параграф 1.1.1).

Все эти черты, в той или иной степени, присущи современным компьютерным экспертным системам, использующим механизм продукций в реализации.

С формальных позиций система неоднородных продукций является эклектичным объединением аппаратов нескольких независимых математических дисциплин: логики предикатов, теории вероятностей, теории графов, возможно, теории меры и чего-то еще… Аксиоматика большинства систем продукций (в части, отражающей конкретную предметную область — см. выше примеры про болгарина и чаек) может быть противоречивой, построенные заключения — недостоверными, а стоимость вывода — неоправданно высокой. Тем не менее (и это, порой, вызывает восхищение!) современные экспертные системы способны не просто облегчать жизнь медикам, геологам, химикам и специалистам других областей, но могут и «выдавать» абсолютно неожиданные и чрезвычайно продуктивные решения.

Следует также отметить, что в настоящее время ведутся интенсивные исследования по усовершенствованию механизмов представления и обработки знаний для систем продукций. В частности, в качестве логического аппарата (и примкнувшей к нему теории вероятностей) предлагается использовать так называемые модальные логики, явно включающие в себя кванторы возможности, достоверности и персонификации знания, а также реализующие свойство немонотонности вывода и семантику возможных миров (см. [9, 10]).

Система продукций *(СП) определяется четвёркой: база фактов, множество продукций, система управления, условие окончания построения экспертного заключения.*

В этой книге база данных систематически обозначается буквой $\boldsymbol{D}$, множество продукций — $\boldsymbol{R}$, стратегия (система) управления — $\boldsymbol{C}$, а условие окончания — $\boldsymbol{t}$. После того, как мы зафиксировали обозначения, приведенное выше определение может быть переписано в стандартной математической форме:

$$\text{СП} = \langle \boldsymbol{D}, \boldsymbol{R}, \boldsymbol{C}, \boldsymbol{t} \rangle . \qquad (**)$$

Условие окончания может быть не задано, поэтому ниже мы приведем некоторый уточняющий комментарий к последнему элементу четвёрки.

Таким образом, мы видим, что прикладные системы с элементами искусственного интеллекта, основанные на формализме продукций, имеют четкое деление на три части, упомянутые в предыдущей главе (см. параграф 1.3.4):

- фактографическое знание (база фактов, как правило, реализованная в форме базы данных);
- алгоритмическое знание (правила);
- концептуальное знание (стратегия управления).

Именно поэтому системы продукций так хорошо отражают прикладные системы ИИ в целом. Рассмотрим элементы системы продукций более детально.

Множество продукций $\boldsymbol{R}$. Элементы этого множества обычно называют *продукционными правилами*, или *правилами продукции* или просто *продукциями*.

Неоднородная продукция *(production) — это пара «условие – действие», которая фиксирует порцию знаний, необходимых для выполнения одного шага построения экспертного заключения.*

Правило продукции может быть записано в виде:

$$r =_{def} \textbf{if}\ p\,(d)\ \textbf{then}\ d := f\,(d)\ \textbf{end if}, \qquad (***)$$

где p — условие, а f — действие.

Тогда множество правил будет иметь вид:

$$\boldsymbol{R} = \{\ r \mid r = (p, f\,[w])\ \}.$$

Условная часть правила p определяет, когда это правило может быть применено в зависимости от состояния окружения (базы фактов). Реализация p, как правило, определяется истинностью или ложностью предиката над множеством достоверно установленных к данному моменту фактов. Часть продукции f задаёт отображение $f: \boldsymbol{D} \to \boldsymbol{D}$ и определяет *действие* над окружением[30]. Иногда используют вещественное число w, которое характеризует *цену* (или *вес, — weight*) применения правила. В таком случае можно, например, ставить задачу не просто найти какую-то последовательность применения продукций, ведущую к цели, а *найти последовательность с минимальной суммарной ценой.*

База фактов *D* содержит описание текущего состояния «мира» в процессе рассуждений. Это описание является образцом, который сопоставляется с условной частью продукции с целью выбора соответствующих действий при решении задачи. Если текущее содержимое базы данных удовлетворяет условию некоторого правила, то может выполняться действие, указанное в этом правиле. В таком случае говорят, что правило *применимо*. Но *возможность применить правило еще не означает необходимость его применять.* Будет ли применимое правило действительно применено, определяет стратегия управления.

Если правило применимо, и стратегия управления его выбирает, то правило применяется. После применения правила база данных меняет свое состояние.

Стратегия управления *C*. Управляющая структура продукционной системы проста: база фактов инициализируется

---

[30] В некоторых особо экзотических системах продукций f может также влиять на способ выбора и применения последующих продукций.

начальным описанием задачи. Текущее состояние решения задачи также отражается в базе фактов. Текущее содержимое базы фактов сопоставляется с условиями продукционных правил, что порождает подмножество применимых правил, иногда называемое *конфликтным множеством*[31]. Далее выбирается одна из продукций $r \in \boldsymbol{R}$ конфликтного множества (выполняется разрешение конфликта) и выполняется действие f этого правила. В результате применения правила меняется не только состояние базы фактов, но и, возможно, меняется конфликтное множество. Дело в том, что после изменения состояния базы условия некоторых правил могут перестать выполняться и эти правила нужно исключить из конфликтного множества. С другой стороны, условия ранее не включенных правил могут стать истинными, и их необходимо включить. При этом стратегии разрешения конфликтов (*стратегии управления*) могут быть как достаточно простыми, например, выбор первого применимого правила, так и достаточно сложными, например, использующими сложные эвристики. Правила применяются в соответствии со стратегией управления, например, до тех пор, пока либо не будет достигнуто условие окончания, либо уже не окажется больше применимых правил.

Условие окончания *t*. Обычно существует условие окончания процесса поиска решения $\boldsymbol{t} : \boldsymbol{D} \rightarrow$ **bool**, то есть предикат, аргументом которого являются факты из базы данных. Если условие $\boldsymbol{t}(d)$ выполнено, то считается, что больше применять продукции не нужно, и текущее состояние базы данных d является ответом. Иногда условие окончания задается не в виде формулы, а сразу в виде множества

---

[31] Условия продукционных правил вовсе не обязаны образовывать полную дизъюнктную систему предикатов в общем случае. Вполне может случиться так, что сразу несколько правил являются применимыми — в этом случае они конфликтуют друг с другом. С другой стороны, может статься, что ни одно правило не окажется применимым в некоторой ситуации — это означает, что система продукций не может продолжить работу и останавливается с результатом `fail` – неудача.

состояний базы данных, удовлетворяющих условию окончания. В таком случае это множество обычно обозначается $\boldsymbol{T}$:

$$\boldsymbol{T} = \{d \in \boldsymbol{D} \mid \boldsymbol{t}(d)\}.$$

Возможен случай, когда условие окончания не задается, потому что система должна работать постоянно. Таковыми, например, являются системы автоматического управления технологическими объектами и процессами непрерывного действия. Здесь имеет смысл разместить обещанное выше разъяснение относительно последнего элемента четверки в формуле (**). Последний элемент — то самое обсуждаемое условие окончания $\boldsymbol{t}$ — мы из записи удалить не можем (в силу существующих математических «условностей»), поэтому договоримся считать, что для обсуждаемых в этом абзаце систем продукций *постоянного действия* мы просто будем полагать $\boldsymbol{t}$ тождественно равным **false**.

Обсудив составные части систем продукций, обратимся к структуре системы продукций в целом. Организация прикладной системы ИИ в виде системы продукций обладает одним очень важным преимуществом — **модульностью** или **алгоритмичностью** знания. Отличие системы продукций от иерархически организованных программ заключается в следующем.

1. **Между правилами нет прямой связи по управлению**. Иными словами, правила не вызывают друг друга. Все управление правилами вынесено в стратегию управления $\boldsymbol{C}$.

2. **Между правилами нет и прямой связи по данным**. Правила ничего не сообщают друг другу, все данные находятся в базе данных $\boldsymbol{D}$.

Отсюда следует важный вывод — *правила являются функционально и информационно прочными модулями*.

В прикладной системе с элементами искусственного интеллекта модульность критична, так как для таких программ очень часто заранее не ясен алгоритм действия. А значит, нужна

модифицируемость, чтобы пробовать разные способы решения задачи. Именно этого можно достичь при использовании систем продукций, поскольку мы можем добавлять и удалять правила в множестве *R*, и это никак не скажется на других правилах! Более того, в пределе можно модифицировать множество *R* динамически, прямо в процессе решения задачи.

Гленфорд Майерс рекомендует для использования при разработке программного обеспечения только два высших по прочности класса модулей.

Функционально прочный модуль *— это модуль, выполняющий (реализующий) только одну какую-либо определенную функцию.*

При реализации этой функции такой модуль может использовать и другие модули.

Информационно прочный модуль *— это модуль, выполняющий (реализующий) несколько операций над одной и той же структурой данных (информационным объектом), которая считается неизвестной вне этого модуля.*

Информационно прочный модуль может реализовывать, например, абстрактный тип данных или класс в объектно-ориентированном программировании.

Сцепление модуля *— это мера его зависимости по данным от других модулей.*

Сцепление модулей характеризуется способом передачи данных. Чем слабее сцепление модуля с другими модулями, тем сильнее его независимость от других модулей. Для оценки степени сцепления Майерс предложил упорядоченный набор из нескольких видов сцепления модулей. Худшим видом сцепления модулей является *сцепление по содержимому*. Таким является сцепление двух модулей, когда один из них имеет прямые ссылки на содержимое другого модуля (например, на константу, содержащуюся в другом модуле). Такое сцепление модулей не рекомендуется использовать. *Сцепление по общей области* — это такое сцепление модулей, когда

несколько модулей используют одну и ту же область памяти[32]. Наилучшим видом сцепления модулей, который рекомендуется для использования современной технологией программирования, является *параметрическое сцепление* — это случай, когда данные передаются модулю либо при обращении к нему как значения аргументов, либо как результат его обращения к другому модулю для вычисления некоторой функции. Такой вид сцепления модулей реализуется в языках программирования при использовании обращений к процедурам и функциям.

2.1.3. Алгоритм работы системы неоднородных продукций

Рассмотрим вид прикладной системы ИИ, выдающий простейший тип результата — только ответ да/нет (уровень 0, см. параграф 1.3.5.), что соответствует тому, выполнено условие окончания $\boldsymbol{t}(d)$ или нет. При этом мы считаем, что последовательность применения правил нам неинтересна. Положительный ответ «да» здесь и далее в программах обозначается **OK**, отрицательный ответ «нет» — **fail**. В таком случае работа системы продукций задается алгоритмом 2.1.

Алгоритм 2.1. Интерпретатор системы продукций.

```
proc Production (d, R, t)
    while ¬t(d) do
        r : = Select (d, R)
        if r = nil then return (fail) end if
        d : = r.f (d)
    end while
    return (OK)
end proc
```

Этот алгоритм получает на вход начальное состояние базы d, множество продукций **R** и условие окончания **t**. Далее, пока не выполнено условие окончания, с помощью функции Select

[32] Именно этот вид сцепления используется в системах продукций.

выбирается применимое правило, и с его помощью меняется состояние базы данных, пока не окажется выполненным условие окончания. Если применимого правила не находится, то процесс прерывается и выдается отрицательный ответ.

Функция `Select` — сердце этой системы, именно она определяет выбор применяемого правила, то есть стратегию системы продукций ***C***. Как правило, в искусственно-интеллектуальной системе недостаточно информации для того, чтобы на каждом шаге применять наилучшее правило. Поэтому *процесс выбора правила есть процесс поиска*.

Например, самая грубая стратегия поиска — «первый подходящий», представленная в алгоритме 2.2. Примеры более изощренных стратегий приведены далее.

Алгоритм 2.2. Стратегия выбора правила «первый подходящий».

```
proc Select (d, R)
    for r ∈ R do
        if r.p (D) then return (r) end if
    end for
    return (nil)
end proc
```

Это хороший пример плохой стратегии. Трудно ожидать интеллекта от машины, если мы не закладываем в нее немного своего интеллекта.

Рассмотрим более пристально систему продукций с алгоритмической точки зрения. Фактически, в алгоритме 2.1 мы определили и зафиксировали способ интерпретации знаний, представленных в форме множества продукций. Алгоритм 2.1 носит чисто механический характер, в нем нет никаких признаков «интеллекта», ни искусственного, ни естественного, он просто исполняет множество правил R над входными данными d, никак не анализируя ход изменения состояния базы, не оценивания, приближаемся ли мы к решению или удаляемся от него. Совершенно аналогично ведут себя и обычные компьютеры: они бездумно

исполняют заложенную программу, не пытаясь понять, имеет ли какой-либо смысл их работа. Другими словами, алгоритм 2.1 является *эмулятором виртуальной машины*, а множество правил R — программа для этой машины. Особое место занимает функция `Select`. В этой функции и кроется «интеллект», то есть понятийные знания в терминологии параграфа 1.3.2. Именно эта функция должна направить ход вычислений (преобразований базы) в сторону решения, в то время как алгоритмические знания, сосредоточенные в множестве правил R, живут каждое в отдельности и ни за что не отвечают.

2.1.4. Система продукций как логическое исчисление

Само слово «продукция» в искусственный интеллект пришло из логики, причем не самым прямым путем. *Продукция*, или *правило вывода* — это логическое правило, которое утверждает, что если у нас есть посылки определенного вида, то из них следует заключение, также определенного вида. Например, правило Modes ponens утверждает, что если верны две посылки: утверждение A, и импликация, утверждающая, что из A следует B, то верно заключение B. В формульной записи такое правило записывают так:

$$\frac{A, A \to B}{B}\ .$$

Так вот, с точки зрения математической логики, *систему продукций можно рассматривать как формальную теорию* (понятие формальной теории подробно раскрыто в параграфе 4.1.1). В этой формальной теории:

- исходная база фактов — это часть набора аксиом (проблемно-ориентированных[33]) формальной теории вида $\vdash A(c_1,\ldots, c_k)$, где A — некоторый предикатный символ (обычно, сопоставленный свойству

[33] Т. е. аксиом, связанных с рассматриваемой предметной областью.

объектов рассматриваемой предметной области) — а все c_i — константные термы;

- продукции — это дополнение набора проблемно-ориентированных аксиом в импликативной форме вида $\vdash d \,\&\, p(d) \to$ $\to M(f(d))$, причем предикат p над набором фактов d — это своего рода «спусковой крючок» продукции, а запись $M(f(d))$ содержательно означает, что инициируется выполнение действий над базой фактов $\boldsymbol{D}$ (модификация базы), которые определяет функция f: $\boldsymbol{D} \to \boldsymbol{D}$, и успешное выполнение этих действий приписывает предикату $M()$ значение **true**;

- единственным правилом вывода выступает уже упоминавшееся правило Modes ponens;

- промежуточные состояния базы фактов — это наборы формул, полученные из проблемно-ориентированных аксиом в результате модификаций базы $\boldsymbol{D}$;

- состояние базы данных, удовлетворяющее условию окончания — это целевая теорема, которую нужно доказать в данной формальной теории.

Таким образом, решение некоторой задачи с помощью системы продукций можно трактовать как поиск доказательства соответствующей теоремы в специальной формальной теории. Данная интерпретация систем продукций еще раз подтверждает наблюдение, сделанное в главе 1 (см. параграф 1.1.5), о том, что *подавляющее большинство задач ИИ можно трактовать как задачи автоматического доказательства теорем*.

Всё это было бы хорошо, если бы не одно «но»! Без таких «но» не обходится, практически, ни один результат в области ИИ даже с учетом сделанных нами во Введении и первой главе ограничений. Перечисленные в параграфе 2.1.1 неоднородные системы продукций, помимо указанных там недостатков, доставляют разработчикам и пользователям дополнительную (и немалую!) головную боль, и связано это как с неоднородностью, так и с предметными областями,

для которых они разрабатываются. Дело в том, что в конкретной ситуации и факты верны не абсолютно, и применение того или иного правила оправдано лишь частично. Сошлемся на ту же медицинскую диагностику[34]. Как вы думаете, можно ли на основании наблюдаемого покраснения радужной и небольшого повышения температуры со стопроцентной уверенностью заявить, что пациент страдает, например, блефаритом? — Разумеется, нет! Такое заключение можно допустить лишь с некоторой вероятностью (в системах продукций для этого используют термины «коэффициент достоверности», «обоснованность», «индекс уверенности» и т. д., и обрабатываются эти коэффициенты, порой, самым невероятным способом). Скажем больше, — обоснованность применения конкретной продукции к текущему состоянию окружения также является не абсолютной.

Поясним на более формальном уровне применение одной из популярных в настоящее время техник работы с частично достоверной информации. Пусть $Q_{\mathbf{1}}(a_{\mathbf{1}}),\ldots, Q_{\boldsymbol{n}}(a_{\boldsymbol{n}})$ — факты из конкретного $D \in \boldsymbol{D}$, $v_{\boldsymbol{i}}$, $i = 1,\ldots, n$ — значения признаков обоснованности этих фактов из диапазона $(0,1)$, а $v_{\boldsymbol{r}}$ из того же диапазона — обоснованность (применения!) продукции r вида (***). Модифицирующая составляющая продукции $r.f$ в реальных системах определяется в общем случае некоторой функцией, которая, естественно, имеет аргументы (пусть это будут факты $Q_{\boldsymbol{i}}(a_i)$, $i = 1,\ldots, n$) и результат (допустим, $Q_{\mathbf{0}}(a_{\mathbf{0}})$). Проблема заключается в определении достоверности модифицированного факта $Q_{\mathbf{0}}$ — величины $v_{\mathbf{0}}$. Обращаем внимание читателя на использование слова «модифицированного» — $r.f$ может не только добавлять/удалять факт, но может и повышать/понижать уровень достоверности уже имеющегося! В случае удаления факта из базы вопросов не возникает. Обсуждая *собственно* модификацию,

---

[34] Как известно, в медицине разбираются все, так же как в проблемах образования и воспитания.

предыдущее значение обоснованности факта Q_0 мы будем обозначать символом $v_{0(pred)}$.

Чаще всего для определения новых значений достоверностей используют те или иные комбинации минимаксных и вероятностных оценок. Первое (и, пожалуй, наихудшее), что может прийти в голову, является назначение *новому* (добавленному) факту признака обоснованности как минимума (реже, максимума) $v_0 = \mathrm{MIN}(v_1, v_2,\ldots, v_n, v_r)$, достоверность *модифицированного* факта при этом подходе вычисляют по формуле: $v_0 = (v_{0(pred)} + \mathrm{MIN}(v_1, v_2,\ldots, v_n,v_r))/2$.

Более привлекательным выглядит использование вероятностного аппарата, при этом обоснованность *нового* факта может определяться, как условная вероятность $v_0 = \mathbf{P}(g(v_1, v_2,\ldots, v_n)|v_r) = v_r*(1 - g(v_1, v_2,\ldots, v_n))$, где g — любая более или менее разумная функция, не выводящая за пределы диапазона (0,1), это может быть MIN или MAX, среднее арифметическое, взвешенное среднее[35] или еще более экзотические варианты. Столь же разнообразны и методы определения достоверность *модифицированного* факта. Отметим весьма важное обстоятельство — мы не можем говорить о достоверности некоторого состояния базы фактов *целиком* (интегрально) — это все равно, что подсчитывать «среднюю температуру по больнице».

В последние годы разработчики все чаще ищут более строгий (и целостный) формальный базис для работы с неоднородными продукциями. Известны попытки использовать *нечеткие логик*и [11], *дескриптивные логики* [12] и другие теории.

Похвальное стремление разработчиков прикладных систем продукций, практически, полностью дезавуируется тем

---

[35] Взвешенное среднее определяется выражением вида: $(c_1*v_1 + c_2*v_2 + \ldots + c_n*v_n)/(c_1 + c_2 + \ldots + c_n)$, где c_i — некоторым образом заданные (экспертом) константы, определяющие «степень влияния» отдельного аргумента на конечный результат.

обстоятельством, что оценки достоверности исходных фактов и правил в конкретизированной предметной области сами не являются достоверными — их выставляют эксперты (люди с немалым опытом, но всё же люди!), которым, как известно «свойственно ошибаться». *Какой бы не была продуманной и сбалансированной реализация системы с элементами ИИ, и на какой бы сильный и строгий теоретический базис она не опиралась, но, если исходный материал* ***не вполне достоверен*** *и имеет оттенок* ***субъективности****, то найденное решение также будет носить характер возможного (но* ***не абсолютного****!).* Этот тезис является, пожалуй, одним из самых важных в книге, и если читатель его усвоит, это поможет избежать в дальнейшем глубочайших разочарований и уверенно чувствовать себя в «бурлящих водах океана искусственного интеллекта»!

2.1.5. Игра в восемь

Эффективность искусственно-интеллектуальной системы определяется в первую очередь тем, насколько удачно выбрано представление знаний для всех компонентов ***D***, ***R***, ***C***. Здесь мы рассмотрим несколько примеров простых головоломок, акцентируя внимание на удачных и неудачных выборах представления знаний.

Рассмотрим игру в восемь (пример взят из книги [13]), которая многим знакома по аналогии с известной игрой в пятнадцать. Суть игры заключается в перемещении фишек, пронумерованных от 1 до 8 так, чтобы они расположились по возрастанию по часовой стрелке. При этом одна из клеток поля 3 × 3 всегда свободна, и на нее можно перемещать любую соседнюю фишку. Фишки не могут «прыгать» друг через друга — их можно только передвигать на соседнее свободное место.

Допустим, у нас имеется следующие исходная (слева) и целевая (справа) позиции, представленные на рис. 2.1.

Представим знания об игре в восемь на основе систем продукций.

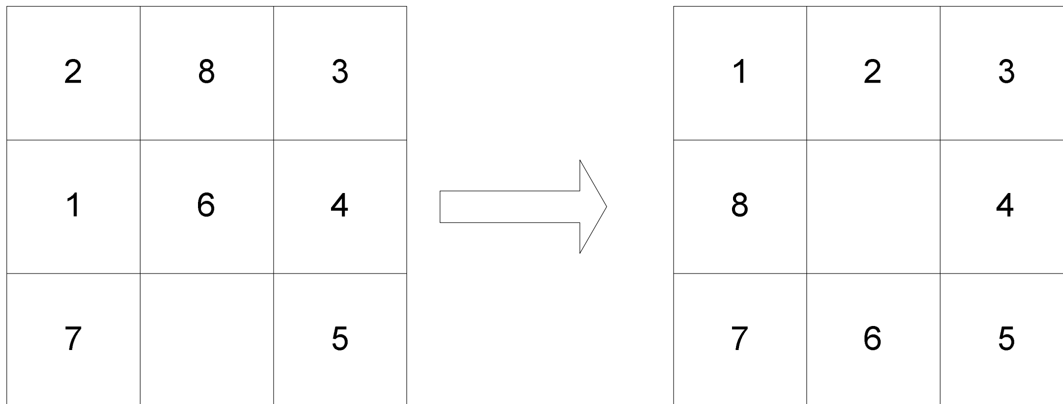

Рис. 2.1. Исходная и целевая позиции игры в восемь

Представление базы данных напрашивается из физического смысла. Обозначим пустую ячейку как фишку с номером 0. В результате ***D*** будет иметь вид:

```
D : array [1..3,1..3] of 0..8
```

Правила ***R*** тоже записываются достаточно ожидаемо. Например, правило «подвинуть фишку 1 вниз» имеет вид:

```
if D[i,j]=1 & i<3 & D[i+1,j]=0 then D[i+1,j]:=1;
D[i,j]:=0
```

Исходя из того, что любую из восьми фишек можно подвинуть максимум в четыре стороны, получается всего 32 правила. Это естественное, но не самое удачное представление знаний об игре.

Попробуем усовершенствовать первоначальный вариант. Основная идея заключается в том, что любое перемещение фишки можно рассматривать как перемещение пустой ячейки. Вот здесь мы и закладываем немного своего интеллекта.

Добавим в базу данных еще пару чисел `{i,j}` — координаты пустого места. Обозначим направления движения по сторонам света — север, запад, юг и восток.

```
if i>1 then D[i,j]:=D[i-1,j]; i:=i-1   (↑ N на север)
if j<3 then D[i,j]:=D[i,j+1]; j:=j+1   (→ E на восток)
if i<3 then D[i,j]:=D[i+1,j]; i:=i+1   (↓ S на юг)
if j>1 then D[i,j]:=D[i,j-1]; j:=j-1   (← W на запад)
```

В результате составлено представление знаний для игры в восемь, в котором всего 4 правила вместо 32. Это принципиальный результат, потому что число правил n (4 или 32) является аргументом экспоненциальной функции перебора для поиска решения. Для $n = 4$ мощности среднего компьютера наверняка хватит, чтобы быстро произвести полный перебор и найти решение, а для $n = 32$ может и не хватить.

2.1.6. Крестьянин, волк, коза и капуста

Второй пример — известная задача о крестьянине, которому необходимо перевезти на другой берег волка, козу и капусту так, чтобы они не съели друг друга (рис. 2.2). В лодку крестьянин может взять с собой не более одного объекта.

Рис. 2.2. Крестьянин, волк, коза и капуста: постановка задачи

Естественно, можно упростить данную задачу. Идея такова — крестьянин и лодка неразличимы, и составляют одно целое. Также можно считать любое перемещение с берега на берег мгновенным. Перенумеруем объекты: 0 — крестьянин, 1 — волк, 2 — коза, 3 — капуста. Присутствие объекта на правом (целевом) берегу обозначим

«1», на левом (исходном) — «0». В таком случае, очевидно, что база данных ***D*** должна иметь вид

```
D : array [0..3] of 0..1
```

При этом `D[0]` — это положение крестьянина, `D[1]` — волка, `D[2]` — козы и `D[3]` — капусты. Массив `D` мы будем записывать в виде четырехзначной битовой шкалы. Например, `0101` означает, что крестьянин с козой находятся на левом берегу, а волк с капустой — на правом. Тогда исходное состояние базы d_0 = `0000`, а целевое d_k = `1111`.

Каково должно быть множество правил? Если подойти к решению этой задачи прямолинейно, в лоб, то можно рассуждать следующим образом. Всего имеется 16 возможных состояний базы[36]. Рассмотрим пары состояний базы, такие, что из первого состояние можно перейти во второе за один ход. Например, `1011`→`0011` — крестьянин переплыл с правого берега на левый. Таких пар оказывается, как нетрудно видеть, 16 * 4 = 64. Заметим, что некоторые состояния базы допустимы, например, `0101`, а другие состояния недопустимы, например, `0011` — коза съест капусту. Легко проверить, что допустимых состояний 10, а недопустимых — 6. Отбросим те пары состояний, в которых целевое состояние является недопустимым. Оставшиеся пары, а их 40 штук, являются правилами нашей системы.

Нетрудно видеть, что это чисто механическая, формальная работа, которую можно поручить компьютеру. Однако результат окажется не очень хорошим (четыре десятка правил). Такое представление знаний нельзя назвать удачным.

Попробуем подойти к задаче творчески. В данном случае творчество состоит в привлечении общих знаний, которые в исходной постановке задачи не содержались.

---

[36] При $n = 4$ имеется $2^4 = 16$ различных битовых шкал.

1. Если начальное состояние допустимо и все правила сохраняют допустимость, то любое достижимое состояние допустимо (принцип полной индукции).

2. Для решения этой задачи достаточно рассматривать только допустимые состояния базы (недопустимые не могут войти в решение даже на промежуточных этапах).

Заметим, что начальное состояние допустимо (так же, как и конечное). Осталось выделить правила, сохраняющие допустимость. Для этого заметим, что в задаче, по существу, есть всего четыре возможных хода: движение крестьянина без объекта, движение крестьянина с волком, с козой и с капустой. Теперь осталось выписать простые условия допустимости этих ходов и получить множество правил R:

```
if D[1]<>D[2] & D[2]<>D[3]    then    D[0]:= 1 - D[0]
if D[2]<>D[3] & D[0]=D[1]     then    D[0]:= 1 - D[0];
                                   D[1]:= 1 - D[1]
if D[0]=D[2]                  then    D[0]:= 1 - D[0];
                                   D[2]:= 1 - D[2]
if D[1]<>D[2] & D[0]=D[3]     then    D[0]:= 1 - D[0];
                                   D[3]:= 1 - D[3]
```

Таким образом, в данном случае творческий подход к задаче улучшил представление знаний в десять раз.

2.1.7. Ход конем

В шахматах конь может перемещаться «буквой Г» — на два поля по горизонтали или вертикали и на одно поле в перпендикулярном направлении. При этом он не должен выходить за границы шахматной доски. Таким образом, в самом общем случае существуют не более восьми возможных ходов конем из данной позиции (рис. 2.3). Если конь находится на краю доски, возможных ходов меньше.

Традиционная формулировка задачи заключается в том, чтобы найти такую последовательность ходов конем, начиная с произвольного поля, при которой конь становится на каждое поле доски только один раз.

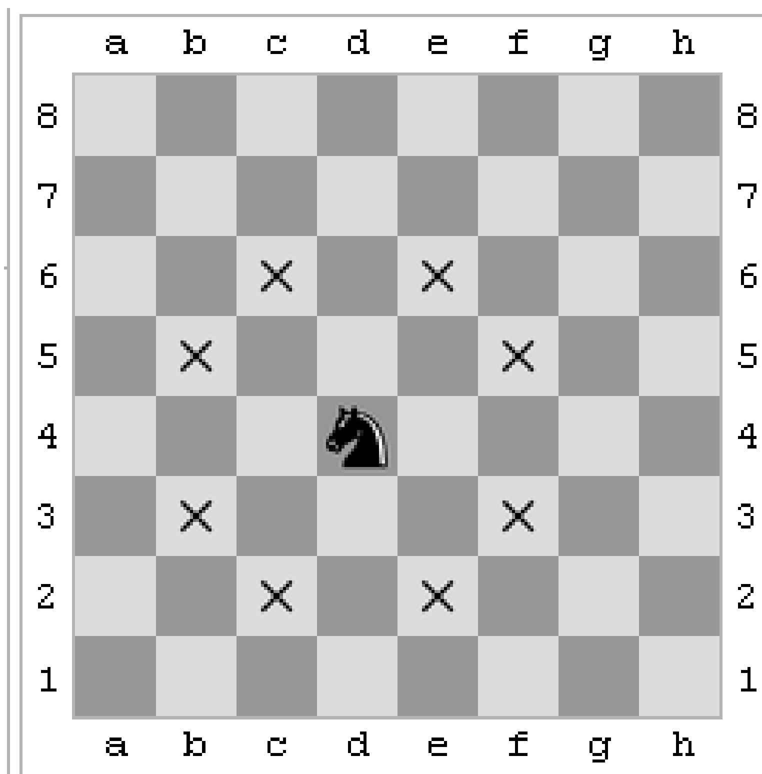

Рис. 2.3. Возможные ходы конем

Тогда база данных ***D*** будет иметь вид:

```
D : array [1..8,1..8] of 1..0,
```

где `D[i, j] = 1` означает, что конь был на поле `(i,j)`, а `0` — не был.[37]

Тогда исходное состояние базы D_0 — все элементы массива `D` нули, а целевое D_k — все элементы `D` единицы.

---

[37] Для упрощения обозначений при написании алгоритмов мы отходим от традиционной шахматной нотации, обозначая как горизонтали, так и вертикали цифрами, а при указании ходов в шахматном смысле используем привычные обозначения вертикалей латинскими буквами.

Подсчитаем, сколько всего различных ходов конем существует в шахматах (на пустой доске). Для каждого из 16 полей в центре доски существует по 8 ходов. Для полей примыкающего слоя — по 6 ходов, за исключением полей b2, b7, g2, g7, в которых только по 4 хода. Далее, из полей на краю доски возможны по 4 хода, за исключением угловых полей, из которых существует только 2 хода, и полей a2, b1, a7, b8, g8, h7, h2, g1, из которых конь может прыгнуть на 3 других поля. Итого имеем: 16 * 8 + 16 * 6 + 4 * 4 + 16 * 4 + 8 * 3 + 4 * 2 = 336 ходов конем.

Очевидно, не имеет смысла нумеровать все реально возможные ходы и сопоставлять им правила: их слишком много. Заменим реальные ходы набором из восьми продукций, генерирующих допустимые ходы конем, руководствуясь рис. 2.3. При этом мы можем прямо в правила внести проверку условия задачи о том, что конь может попадать на данное поле только один раз. Текущие координаты коня хранятся в переменных `i` и `j`.

```
if i <= 6 & j <= 7 & D[i+2,j+1] = 0 then
  D[i+2,j+1] = 1 & i = i+2 & j = j+1
if i <= 7 & j <= 6 & D[i+1,j+2] = 0 then
  D[i+1,j+2] = 1 & i = i+1 & j = j+2
if i >= 1 & j <= 6 & D[i-1,j+2] = 0 then
  D[i-1,j+2] = 1 & i = i-1 & j = j+2
if i >= 2 & j <= 7 & D[i-2,j+1] = 0 then
  D[i-2,j+1] = 1 & i = i-2 & j = j+1
if i >= 2 & j >= 1 & D[i-2,j-1] = 0 then
  D[i-2,j-1] = 1 & i = i-2 & j = j-1
if i >= 1 & j >= 2 & D[i-1,j-2] = 0 then
  D[i-1,j-2] = 1 & i = i-1 & j = j-2
if i <= 7 & j >= 2 & D[i+1,j-2] = 0 then
  D[i+1,j-2] = 1 & i = i+1 & j = j – 2
if i <= 6 & j >= 1 & D[i+2,j-1] = 0 then
  D[i+2,j-1] = 1 & i = i+2 & j = j-1
```

Экономия по сравнению с «лобовым» подходом составляет 336:8 = 42 раза.

2.2. СТРАТЕГИИ УПРАВЛЕНИЯ

Можно предложить огромное множество *стратегий управления* (control strategies) для систем продукций. Самое важное наблюдение состоит в том, что *не существует и не может существовать наилучшей стратегии управления на все случаи жизни. Одна и та же стратегия может хорошо работать для одного класса задач и быть совершенно неудовлетворительной для другого*.

Предваряя основной материал раздела, мы сделаем ряд замечаний. На первый взгляд эти замечания могут показаться несущественными, однако мы последовательно пытаемся придерживаться принципа *необходимости* (*нужности*, *полезности*) всего материала, включённого в книгу — это в равной мере касается примечаний, добавлений и шуток, встречающихся в тексте.

Поговорим о *естественности* терминов, *однозначности* определений и *единственности* трактовок. Начнем, пожалуй, с графов, точнее, с класса, важного для понимания материала книги, называемого классом ориентированных деревьев. Зададимся вопросом, *куда растет дерево — вверх или вниз*? Разумеется, первым в голову приходит ответ «вверх», но так ли это очевидно и правильно? А что, если встать на голову (поменять систему координат)? или вспомнить о корнях[38] (рассмотреть более полную картину)? Наконец, можно просто переопределить понятия *верха* и

---

[38] Известно просто замечательное растение — баньян — индийский фикус, священное дерево буддистов. С его горизонтальных ветвей к земле опускаются отростки, которые превращаются в корни, разрастаются, приобретают колоннообразный вид, и одно дерево становится целой рощей.

низа[39], взаимно поменяв их смысл (исходя, скажем, из соображений удобства[40] при изображении объекта заранее неизвестного размера).

Примеров подобного толка существует огромное количество[41], и все они убеждают лишь в одном — *направления* (равно как и их *именование*) являются весьма относительными, и каждый раз, если это не следует из контекста, необходимо фиксировать систему отсчета.

Вооружившись этими наблюдениями, вернемся к стратегиям управления выводом. Напомним, что речь идет о системах продукций, где базовым объектом выступает выражение вида (*). В записи продукции $p \to q$ присутствует символ «$\to$», который некоторым образом задаёт систему отчета: движение от аргументов к результатам (от данных к цели) осуществляется, как бы, в прямом направлении, а от цели к данным — соответственно, в обратном. Такая классификация стратегий характерна, в основном, для следующих областей искусственного интеллекта: экспертные системы, автоматический синтез программ, системы верификации и ряд других. В то же время в «более фундаментальной» области автоматического доказательства теорем (её еще называют логическим программированием) принята прямо противоположная классификация: целеориентированные стратегии[42] (от целевой

---

[39] Символ '**Ж**' ('живете') древнерусского алфавита как раз и изображает дерево (Жизни — *авт.*), при этом он (символ) обладает как центральной, так и осевыми симметриями!

[40] Имеется в виду «удобство» для тех, кто привык писать слева направо, а не, например, по-арабски.

[41] Вспомним противоположность направлений отчета угловых мер в геодезии и математике — «*посолонь*» и «*противосолонь*», и «*естественность*» этих направлений для жителей Северного и Южного полушарий.

[42] Их предложили сначала М. Дэвис и Х. Патнем [14], а затем развил и усовершенствовал, разработав широко известный метод резолюции, Дж. Робинсон [15].

формулы к аксиомам) называют прямыми, а ведущие от аксиом к целевому утверждению — обратными. К последним относится чрезвычайно интересный и перспективный *обратный метод*, предложенный ленинградским логиком С. Ю. Масловым [16].

Таким образом, (с некоторыми допущениями, разумеется) можно считать, что *обратный* метод Маслова и *прямая* волна реализуют одну и ту же стратегию, несмотря на радикальное расхождение в названиях.

2.2.1. Критерии сравнения стратегий

Раз стратегии различны, нужны критерии для их сравнения. Критериев для сравнения также немало, и мы приведем здесь те, которые нам импонируют.

Некоторые варианты стратегий управления приведены на рис. 2.4. Здесь используются два критерия:

- какая доля информации, уже полученной в ходе поиска решения, сохраняется для последующего использования;
- до какой степени разрешены возвраты (перевод базы данных происходит не в новое, а в одно из старых состояний).

Рис. 2.4. Варианты стратегии управления

Заметим, что на самом деле эти критерии не являются вполне независимыми. Действительно, чтобы вернуть базу в одно из уже пройденных состояний, нужно, по меньшей мере, сохранить информацию о пройденном состоянии.

Очевидно, что стратегия «первый подходящий», приведенная ранее (см. параграф 2.1.2, алгоритм 2.2), простейшая и соответствует левому нижнему углу на плоскости стратегий на рис. 2.4. Эта стратегия относится к группе стратегий *безвозвратный поиск*.

Кроме безвозвратного поиска, в этой книге рассматриваются еще две популярные группы стратегий:

1. *Поиск на графе*, при котором сохраняется вся доступная информация и возможны любые возвраты, хоть в начальное состояние.

2. *Поиск с возвратами* — промежуточный вариант, когда некоторая информация сохраняется, но не вся, и используются возвраты, но не произвольные.

Любую стратегию можно оценить по двум критериям — по затратам на выбор правил и по затратам на применение правил. При этом, естественно, возможно два крайних случая — тупая, но быстрая стратегия и умная, но медленная стратегия. Слишком быстрая стратегия не тратит много времени на выбор правил, а сразу бежит вперед, применяя их. Но может быть, это бег в ложном направлении, который ведет к возрастанию затрат на бесцельное применение правил. Слишком умная стратегия семь раз отмеряет, прежде чем отрезать и применить правило. Может быть, это хорошее правило, прямо ведущее к цели, но каковы будут затраты на топтание на месте перед его выбором? Как всегда, *есть и золотая середина, где общая трудоемкость процесса будет минимальной* (рис. 2.5). Дело в том, что в обоих случаях затраты растут не линейно, а более быстрым образом, то есть функции затрат выпуклы вниз, как показано на рис. 2.5, поэтому у суммы таких функций обязательно есть минимум.

Всё искусство составления успешной стратегии состоит в том, чтобы отыскать этот минимум в каждом конкретном случае.

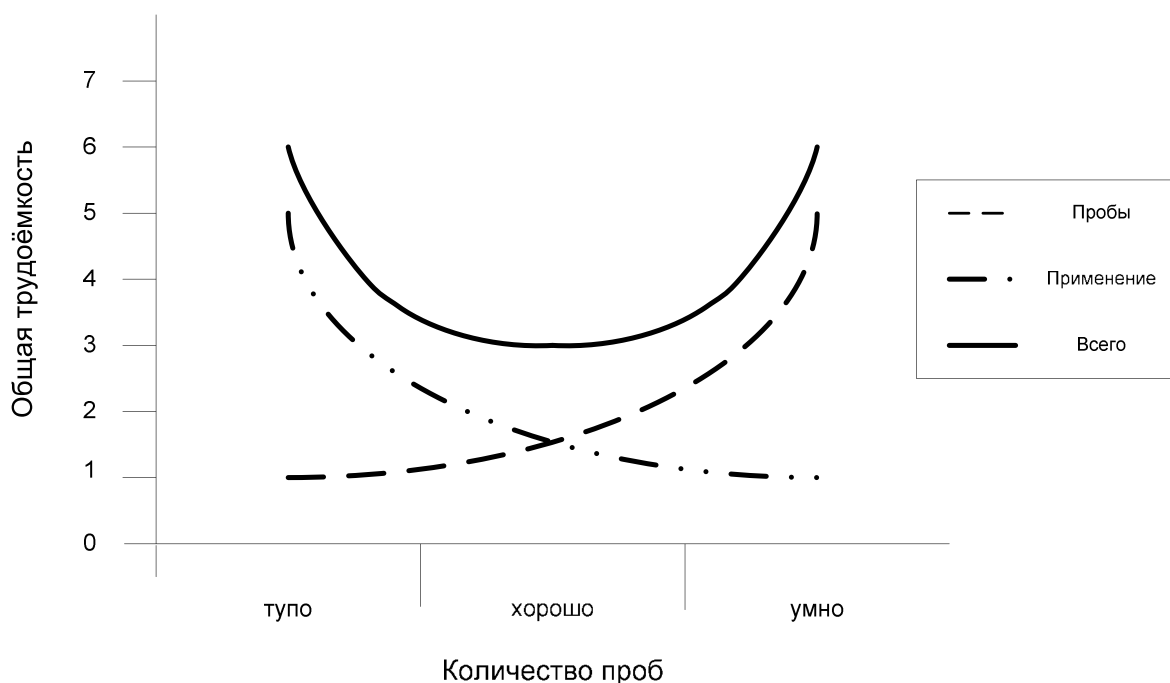

Рис. 2.5. Составляющие затрат на применение стратегии управления

2.2.2. Безвозвратный поиск

***Безвозвратный поиск**, как следует из его названия, — это такой алгоритм поиска, когда мы не собираемся возвращаться к предыдущим состояниям базы данных, а значит, можем не хранить информацию об этих состояниях.*

Простейшим случаем безвозвратного поиска является стратегия «первый подходящий», описанная выше, в параграфе 2.1.2. В этом случае поиск ведется вслепую, наудачу, и результат существенным образом зависит от того, в каком порядке перебираются (а значит, и применяются) правила при проверке. Если мы ничего не знаем о задаче, а пространство поиска велико, то не стоит рассчитывать на то, что стратегия «первый подходящий» даст хороший результат.

Также как и в других случаях, в случае безвозвратного поиска можно ввести *оценочную функцию* $F : \boldsymbol{D} \to \mathrm{P}^{+}$, где P^{+} — множество

неотрицательных вещественных чисел. Запись $F(d)$ — оценка состояния базы данных d. Функция F подбирается таким образом, чтобы на терминальных узлах (тех состояниях базы, где выполняется условие окончания) функция принимала минимальное значение

$$\boldsymbol{t}(d_k) =_{def} F(d_k) = \min \{d \in \boldsymbol{D} \mid F(d)\}.$$

В этом случае функция `Select`, реализующая стратегию «первый подходящий, уменьшающий оценку», может быть записана, как указано в алгоритме 2.3. Ясно, что такая стратегия, по меньшей мере, не хуже стратегии «первый подходящий», а если оценочная функция подобрана удачно, то такая стратегия будет намного лучше.

Алгоритм 2.3. Стратегия выбора правила «первый подходящий, уменьшающий оценку».

```
proc Select (d, R)
    for r ∈ R do
        if r.p(d) & F(r.f(d)) ≤ F(d) then
            return (r)
        end if
    end for
    return (nil)
end proc
```

Можно улучшить процедуру, выбирая такое правило, которое не просто уменьшает оценку, а уменьшает ее в наибольшей степени. Такую стратегию часто называют *методом наискорейшего спуска*, или *градиентным методом*, который относится к классу *жадных алгоритмов* (greedy search). В алгоритме 2.4 представлена простейшая реализация метода наискорейшего спуска.

Алгоритм 2.4. Стратегия выбора правила «наискорейший спуск».

```
proc Select (d, R)
    m := F(d)
    r′ := nil
    for r ∈ R do
        if r.p (d) & F(r.f(d)) < m then
```

```
            r':= r
            m := F(r.f(d))
        end if
    end for
    return (r')
end proc
```

Из вычислительной математики хорошо известно, у методов этого типа есть существенный недостаток — иногда в процессе поиска решения возможны попадания в локальные минимумы и плато, которые в общем случае достаточно трудно обнаружить, для этого требуются очень дорогие вычисления. Попав в локальный минимум, метод наискорейшего спуска остановится в нем, не найдя решения. Попав на плато, метод приведет к бесцельным случайным блужданиям по плато.

Метод наискорейшего спуска хорошо работает, если оценочная функция *унимодальна*, то есть имеет единственный глобальный минимум и не имеет плато. Но *чтобы подобрать унимодальную оценочную функцию, нужно исчерпывающим образом знать свойства пространства поиска, а они, как правило, неизвестны*.

Рассмотрим пример. В случае игры в восемь возьмем оценочную функцию F — это число шашек, находящихся не на своем месте. В результате граф поиска этой игры будет выглядеть следующим образом (рис. 2.6).

Здесь узлы графа обозначают состояния базы данных, а числа в узлах — это значение оценочной функции в данном состоянии.

В данном случае оценочная функция оказалась очень удачной — метод наискорейшего поиска быстро нашел решение. Но здесь есть элемент везения: из узла с оценкой 3 поиск пошел на север, а не на запад, просто потому, что правило движения на север в нашем списке правил оказалось раньше правила движения на запад.

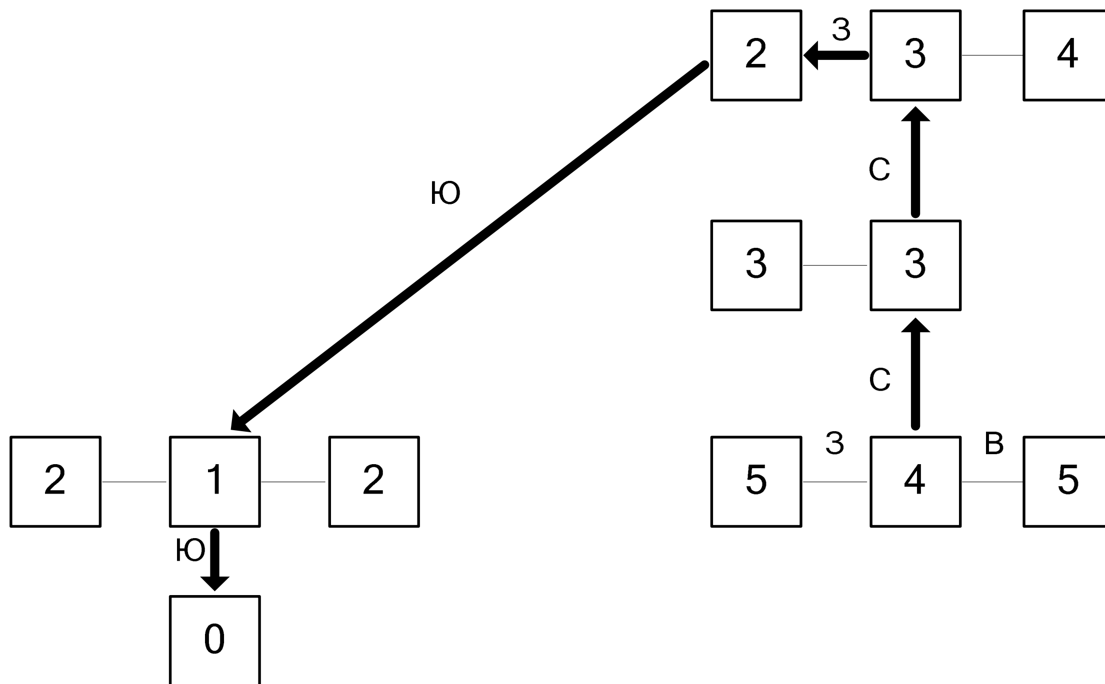

Рис. 2.6. Оценочная функция *F*: число шашек не на своем месте

2.2.3. Поиск с возвратами

Безвозвратный поиск подразумевает, что в случае применения неподходящего правила решение либо становится принципиально невозможным, либо процесс поиска значительно замедляется. Можно положить в основу алгоритма поиска следующую идею: если в текущем состоянии базы данных видно, что при продолжении начатого решения, либо невозможно найти, либо этот поиск обещает быть слишком долгим, тогда можно попробовать вернуться назад и попробовать другой путь.

Стратегия поиска, в которой происходит возвращение назад и применяется другое правило, если применение правила оказывается неподходящим, называется ***поиск с возвратами***[43] *(Backtracking).*

Рассмотрим процедуру `Backtracking` (алгоритм 2.5).

---

[43] Иногда применяют неудачный термин «обратное прослеживание». Этот термин является точным переводом с английского, но плохо отражает суть дела.

Алгоритм 2.5. Процедура поиска с возвратами для системы продукций.

```
proc Backtracking (d, R, t)
    if t(d) then return (OK) end if
    if Deadend (d) then return (fail) end if
    R′ := SelectAll (d, R)
    for r ∈ R′ do
        if Backtracking (r.f(d), R, t) = OK then
            return (OK)
        end if
    end for
    return (fail)
end proc
```

Идея процедуры поиска с возвратами очень проста и состоит в следующем. Если достигнуто какое-то из условий окончания, то работу нужно прекратить и вернуть результат. В противном случае следует определить множество правил, применимых в данном состоянии (конфликтное множество, см. параграф 2.1.2) с помощью функции `SelectAll`. Далее следует в цикле по применимым правилам рекурсивно обратиться к той же процедуре поиска с возвратами, но в качестве состояния базы данных использовать состояние, которое получается после применения выбранного правила.

В данную процедуру целесообразно включить функцию `Deadend` — обрыв «мертвых» путей (тупиков), иначе может произойти зацикливание программы.

Процедура `Deadend` может проверять самые разные условия, например:

- совпадение текущего состояния базы данных ***D*** с уже построенным ранее;

- ограничение глубины поиска и т. д.

Пример поиска с возвратами для задачи о волке, козе и капусте представлен на рис. 2.7. На этом рисунке серым отмечены узлы, из

которых происходит возврат, потому что они идентичны уже построенным узлам. Отдельно следует упомянуть узел 0001. Это тупиковый узел — продолжение поиска из него невозможно. К счастью, продолжение и не нужно, так как к этому моменту алгоритм находит целевой узел 1111.

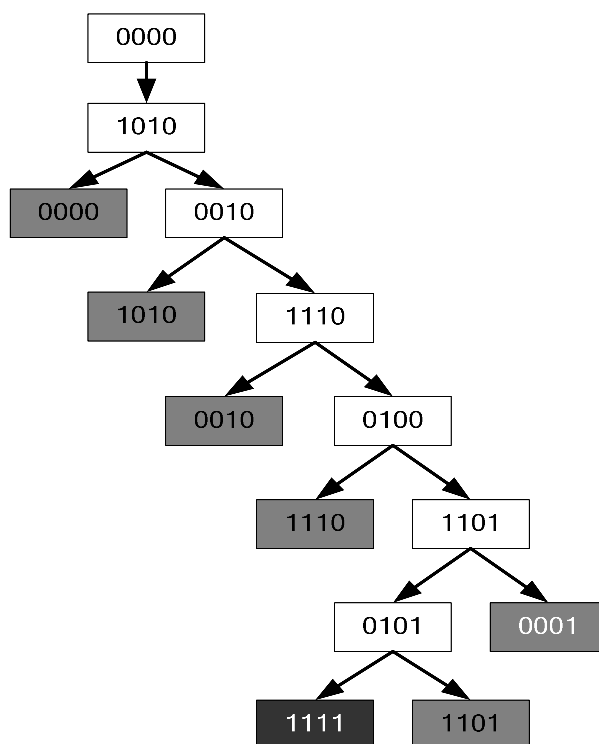

Рис. 2.7. Поиск возвратами для задачи о волке, козе и капусте

Здесь в качестве функции `Deadend` используется проверка совпадения нового состояния базы с построенным ранее на этом пути.

Известны различные модификации поиска с возвратами. Например, когда в процессе поиска сохраняются не все, а только ключевые состояния базы данных, и возврат выполняется не на один шаг, а на несколько шагов назад к последнему сохраненному состоянию.

При поиске с возвратами происходит обход дерева, причем в памяти хранится не все дерево, а только текущий построенный путь в этом дереве. Движение вниз по дереву соответствуют рекурсивные

вызовы. Движению вверх соответствуют возвраты. На рис. 2.8 представлен фрагмент дерева поиска с возвратами для задачи о ходе конем.

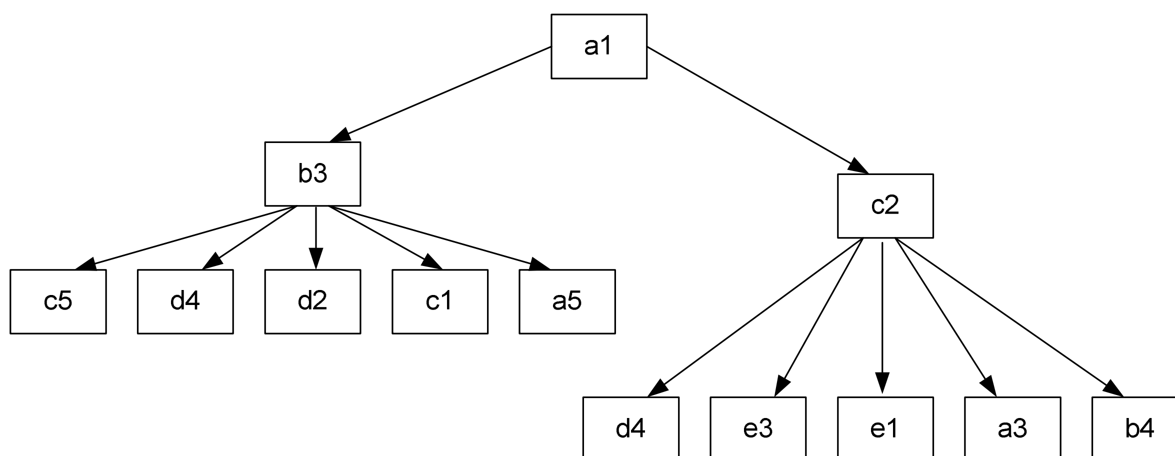

Рис. 2.8. Фрагмент дерева поиска с возвратами для задачи о ходе конем

Ставим коня на произвольное поле доски. В данном случае в качестве первого поля взято угловое поле a1. Поочередно переставляя коня на различные поля, ищем такое, на котором ранее конь не был. Если таких полей вообще нет, то возвращаемся на предыдущее поле и так далее.

Заметим, что сама структура правил предполагает хранение информации о тех полях доски, на которых конь уже побывал (см. параграф 2.1.6). Поэтому здесь в функции `Deadend` нет нужды проверять совпадение нового состояния базы с построенным ранее на этом пути. *Ценой хранения некоторой дополнительной информации в базе данных, можно уменьшить вычислительную нагрузку на процедуру поиска с возвратами*.

2.2.4. Поиск на графе

Поиск на графе (graph search) относится к группе информированных стратегий, при которых хранится вся доступная полученная ранее информация. Для этого создается так называемый граф поиска.

Граф поиска *— это ориентированный граф G (V, E), в котором множество узлов V — это совокупность*[44] *состояний базы данных, а множество дуг E — это совокупность применений правил.*

В этом графе есть начальный узел d_0 и множество (возможно, пустое) терминальных узлов $\boldsymbol{T} = \{d \in D \mid \boldsymbol{t}(d)\}$. Элементы множества терминальных узлов будем обозначать d_t, $d_t \in \boldsymbol{T}$. Дуга из узла d_1 в узел d_2 проводится в том и только в том случае, когда существует правило, применимое в состоянии d_1 и в результате своего применения переводящее базу в состояние d_2.

$$d_1 \rightarrow d_2 \in E \Leftrightarrow \exists\, r \in \boldsymbol{R}\ (r.p(d_1)\ \&\ r.f\,(d_1) = d_2).$$

В общем случае в графе поиска могут быть циклы, петли, мультидуги. Он может быть не связан и т. д.

Для поиска на графе используются следующие две основные процедуры, приведенные в алгоритмах 2.6 и 2.7:

Алгоритм 2.6. Основная процедура поиска на графе.

```
proc GraphSearch (d, R, t)
    O := {d} { список «открытых» узлов }
    C := ∅  { список «закрытых» узлов }
    while ¬ t(d) do
        d := Select (O)
        if d = nil then return (fail) end if
        Open (d, R);
    end while
    return (OK)
end proc
```

Алгоритм 2.7. Процедура раскрытия узла.

```
proc Open (d, R)
```

[44] Мы используем здесь термин «совокупность», чтобы подчернуть, что все узлы графа поиска индивидуальны и различны, но разные узлы графа могут соответствовать одному и тому же состоянию базы, аналогично, все дуги графа индивидуальны и различны, но разные дуги могут соответствовать применению одного правила.

```
    for r ∈ R do
        if r.p(d) then
            if r.f(d) ∉ O & r.f(D) ∉ C then
                O := O + r.f (D)
            end if
        end if
    end for
    O := O - {D}
    C := C + {D}
end proc
```

Здесь знак + означает добавление элемента в список, а знак – означает удаление элемента из списка.

Список *O* традиционно называется списком открытых узлов (от английского Open), а список *C* традиционно называется списком закрытых узлов (от английского Close). Между тем эти традиционные названия могут вводить в заблуждение. На самом деле список *O* — это список узлов, которые мы еще не открывали, но собираемся открыть (to be opened), а список *C* — это список узлов, которые мы уже открыли, и больше открывать не будем (already opened).

Существуют два принципиальных способа поиска на графе — поиск в ширину и поиск в глубину.

Если список O — это стек, то есть для раскрытия выбирается последний положенный узел, то в этом случае говорят о ***поиске в глубину***.

Если список O — это очередь, то есть для раскрытия выбирается первый положенный узел, то в этом случае говорят о ***поиске в ширину***.

У поиска в ширину и в глубину есть свои достоинства и недостатки.

Главное преимущество поиска в ширину состоит в том, что решение всегда будет найдено, если оно существует. Поиск в ширину просматривает все без исключения узлы, постепенно удаляясь от начального узла. С другой стороны, поиск в глубину в

случае, если граф поиска бесконечен или очень велик, может пройти мимо решения, уйти в глубину и не вернуться.

Главное преимущество поиска в глубину состоит в том, что он иногда может оказаться быстрее. Действительно, пусть, например, искомый узел находится на расстоянии *k* от начального узла. В такой ситуации поиск в ширину *обязательно* сначала раскроет *все* узлы, которые находятся на расстоянии, меньшем, чем *k*, и только после этого начнет раскрывать узлы на расстоянии *k*, среди которых находится искомый. В то время как поиск в глубину, если повезет, может пойти к нужному узлу и раскрыть минимальное число узлов на прямом пути до цели (в лучшем случае всего раскрывается *k*–1 узел). Вообще говоря, можно показать, что в среднем поиск в глубину находит решение примерно вдвое быстрее, чем поиск в ширину. На рис. 2.9 приведена наглядная иллюстрация причин этого явления. На этом рисунке черным выделены узлы, которые будет раскрыты, а извилистая стрелка показывает порядок раскрытия узлов. Хорошо видно, что поиск в ширину, двигаясь по спирали, может пройти рядом с решением и уйти на «штрафной круг», а поиск в глубину, если повезет, может почти сразу «наткнуться» на решение.

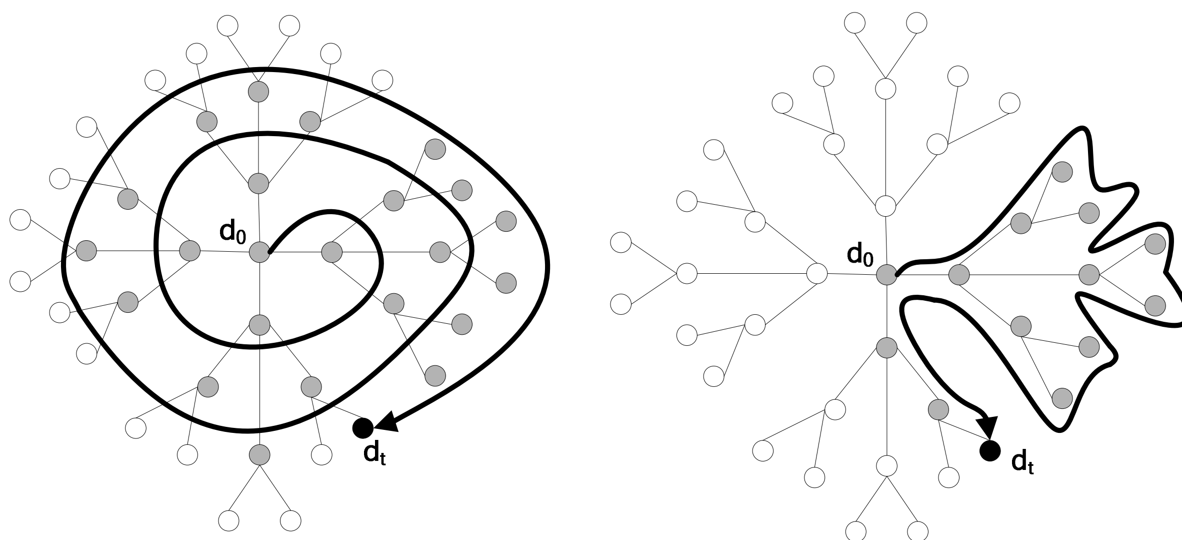

Рис. 2.9. Поиск в ширину (слева) и в глубину (справа)

Обратим внимание на самый внутренний оператор **if** в алгоритме 2.7. При решении любой задачи методом поиска на графе всегда существует дилемма — проверять или не проверять дублирование узлов? Дублирование может возникнуть, поскольку одно и то же состояние базы может получиться как результат разных последовательностей применений правил. Если проверять, то будет тратиться лишнее время на проверку совпадения, зато не будет дублирующих узлов и не будет лишней работы на их повторное открытие. Если не проверять, то наоборот, не будем терять времени на проверку, но, может быть, будем терять время на повторную работу с теми узлами, которые уже встречались.

Рассмотрим поиск в ширину для задачи о крестьянине, волке, козе и капусте (рис. 2.10).

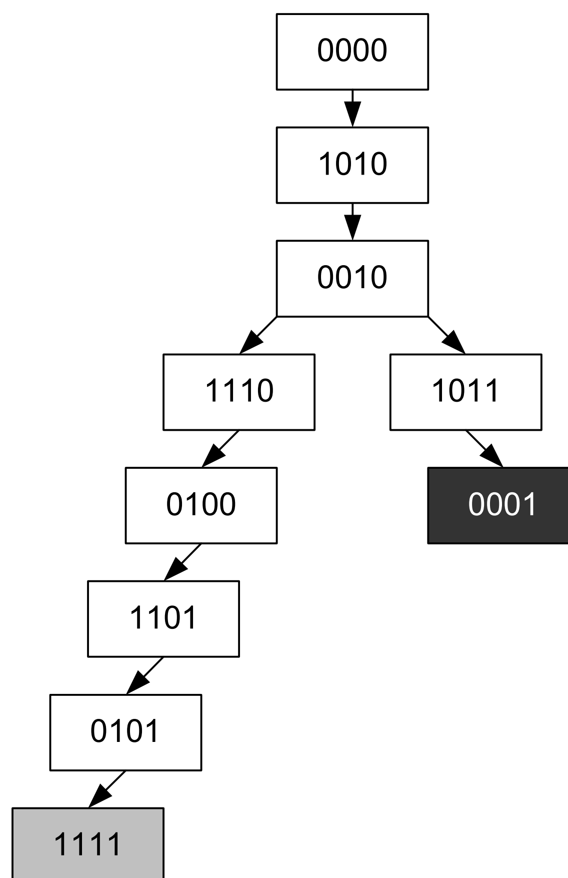

Рис. 2.10. Граф поиска для задачи о волке, козе и капусте с проверкой совпадения узлов

Заметим важное обстоятельство. Вопреки распространенному предрассудку, поиск в ширину может строить меньше узлов, чем поиск с возвратами, если проверять совпадения (сравните рис. 2.7 и рис. 2.10, в первом случае построено 14 узлов, а во втором — только 10). Другими словами, *поиск в ширину на графе (полный перебор) может в некоторых случаях быть эффективнее поиска с возвратами!*

Но если не проверять совпадения, то может быть построено очень много лишних узлов, повторяющих уже имеющиеся. На рис. 2.11 приведен фрагмент соответствующего графа поиска для этой задачи, который, вообще говоря, является бесконечным. Троеточиями на этом рисунке обозначены не показанные продолжения графа. Поиск в ширину найдет решение, хотя и затратит много лишней работы, а поиск в глубину может «промахнуться» мимо решения.

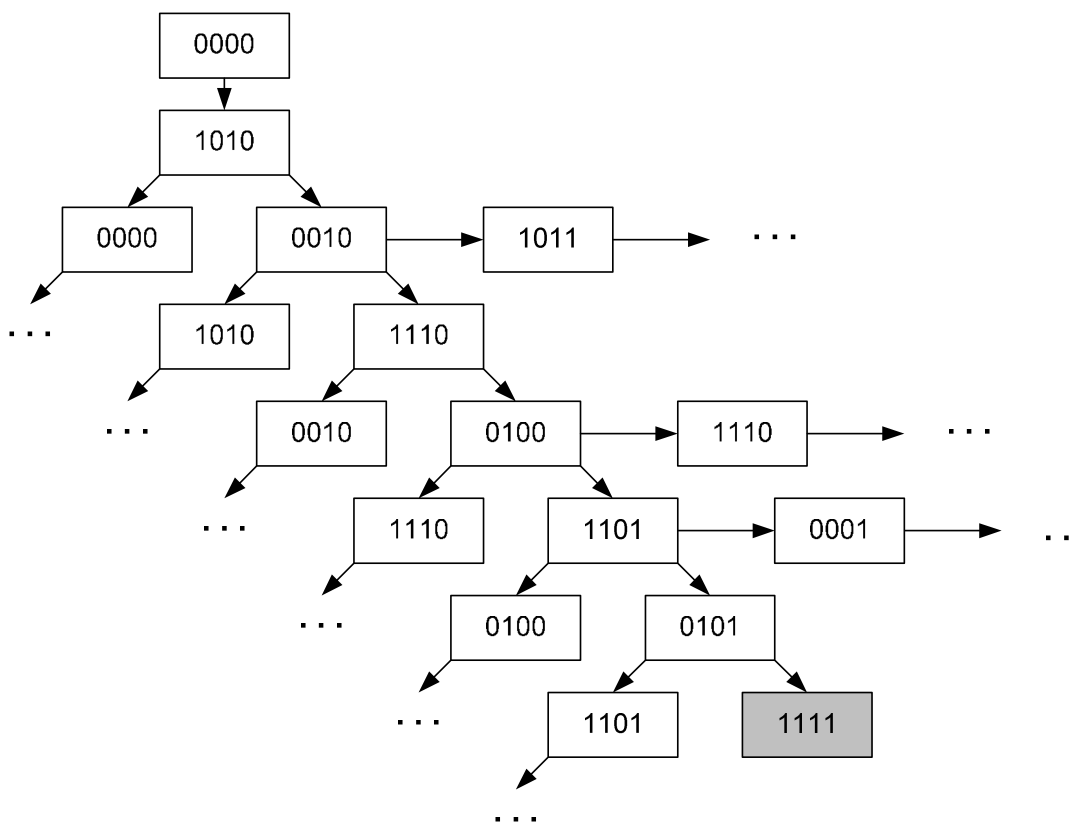

Рис. 2.11. Граф поиска для задачи о волке, козе и капусте без проверки совпадения узлов

Общей математически строгой рекомендации относительно того, нужно или не нужно проверять совпадения, дать невозможно. Неформально можно предположить, что если совпадений немного, то их лучше не проверять, а если много, как в задаче про крестьянина, волка, козу и капусту, то лучше проверять.

Еще одно важное наблюдение о свойствах графа поиска можно сделать на примере игры в восемь. Как известно, существует 8! = 40 320 возможных перестановок восьми фишек и, значит, существует 8!9 = 9! = 362 880 состояний базы данных (число перестановок фишек, умноженное на число положений пустой ячейки). Можно доказать, что все состояния БД разбиваются на два множества: с четным числом инверсий и с нечетным числом инверсий, причем ходы обязательно сохраняют четность. Таким образом, граф поиска для игры в восемь состоит из двух компонент связности, причем каждая компонента сильно связана. Если начальное и целевое состояния базы принадлежат разным компонентам связности, то решение невозможно, а если принадлежат одной компоненте, то решение обязательно существует.

Мы видим, что в общем случае конечное множество правил ***R*** неявно задает, возможно, бесконечный граф поиска, причем он может быть как связным, так и несвязным, содержать циклы или не содержать циклов. По виду отдельных правил никакого заключения о свойствах графа поиска сделать нельзя.

В терминах графа поиска образно можно сказать, что поиск с возвратами — это щуп, который мы засовываем в граф поиска, в надежде наткнуться на d_t. В результате происходит проявление части графа, а часть графа остается скрытой. Поиск на графе — это систематическое постепенное проявление всего графа, когда ничего не остается скрытым.

2.3. СПЕЦИАЛЬНЫЕ СИСТЕМЫ ПРОДУКЦИЙ

Здесь мы обсуждаем различные частные случаи систем продукции. Использование специальных свойств частных случаев может дать и, как правило, дает неожиданный выигрыш.

2.3.1. Обратные системы продукций

Рассмотренные системы продукций всегда начинали поиск от исходного состояния и работали до тех пор, пока в результате не получали целевое состояние. Такие системы продукций *обычно* (см. начало раздела 2.2) называются *прямыми*. Логично поступать и наоборот — применять обратные правила и двигаться от целевого состояния к исходному. Такие системы часто называют *обратными*. Возможны случаи, когда обратная система более эффективна, возможны случаи, когда прямая (рис. 2.12). Образно говоря, это зависит от того, где цель больше — в бóльшую мишень легче попасть!

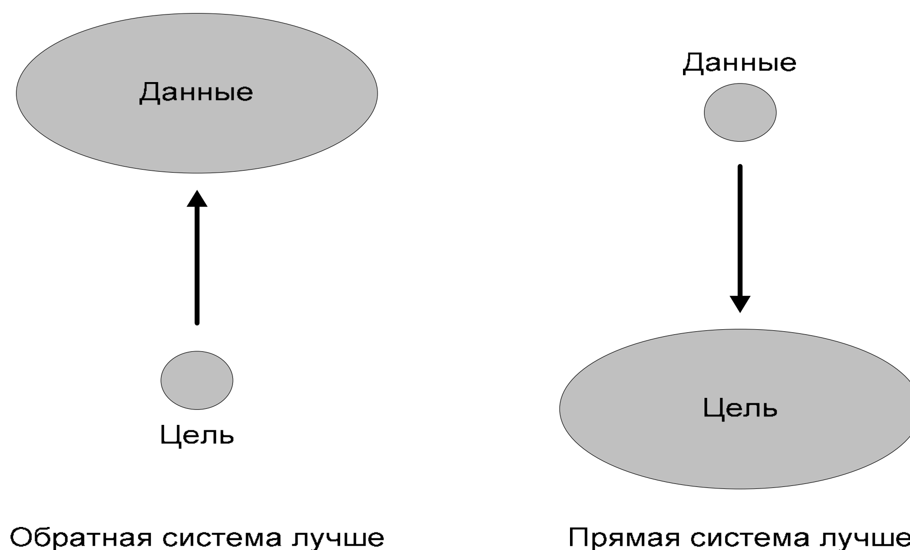

Рис. 2.12. Прямые и обратные системы продукций

Для сравнения прямой и обратной систем продукций рассмотрим следующую модельную задачу из теории чисел. В этой

задаче нужно определить, разлагается ли заданное число на заданные множители.

Пусть целевое число равно 60, а множители — 2, 3, 5.

Прямая система выглядит таким образом.

```
d0 := {2,3,5}, t(d) := 60∈d
R := if x∈d & y∈d & xy∉d & xy<61 then d:= d∪{xy}
```

Поиск в ширину дает следующие результаты (19 применений правил).

```
2    3    5                              (исходные множители)
4    6    9    10   15   25              (попарные произведения)
8    12   18   20   30   50              (вторая строка *2)
27   45                                  (вторая строка *3)
                                         (вторая строка *5 – нет новых)
16   24   36   40   60                   (третья строка *2 – достигнута
цель)
```

Поиск в глубину дает следующие результаты (4 применения правил).

```
2    3    5
15   45   30   60
```

Обратная система будет такой.

```
d0 := {60}, t(d) := 1∈d
R := {   if x∈d & x:2 then d := d\{x}∪{x/2}
         if x∈d & x:3 then d := d\{x}∪{x/3}
         if x∈d & x:5 then d := d\{x}∪{x/5}   }
```

Поиск в глубину дает следующие результаты (4 применения правил).

```
60
30   15   5    1
```

Поиск в ширину дает следующие результаты (11 применений правил).

```
60
30   20   12
```

```
15   10   6    4
5    3    2
1
```

Здесь цель — единственная, а данных — три, поэтому, как и следовало ожидать, обратная система дает чуть лучшие, чем прямая, но сопоставимые результаты.

Но если поставить другую, более расплывчатую цель: какие числа, меньшие или равные 60, можно получить из множителей 2, 3, 5, то сравнение систем покажет совсем другую картину. Действительно, прямая система при этом отличается отсутствием условия окончания и должна работать, пока не остановится ввиду отсутствия применимых правил. Накопленное состояние базы данных будет ответом на вопрос задачи.

```
d0 := {2,3,5}
R := if x∈d & y∈d & xy∉d & xy<61 then d:= d∪{xy}
```

Поиск в ширину дает следующие результаты (22 применения правил).

2	3	5				(исходные множители)
4	6	9	10	15	25	(попарные произведения)
8	12	18	20	30	50	(вторая строка *2)
27	45					(вторая строка *3)
						(вторая строка *5)
16	24	36	40	60		(третья строка *2)
54						(третья строка *3)
32	48					(шестая строка *2)

Поиск в глубину даст такие же результаты, только числа будут найдены в другом порядке. Это следует из того, что в данном случае оба вида поиска обойдут один тот же граф, посетив каждый узел только один раз. Заметим, что количественные результаты расширенной постановки нашей задачи практически совпадают с результатами исходной — в первом случае алгоритмом поиска в ширину всего лишь три узла не были построены. То есть в исходной постановке задачи для поиска в ширину прямая система не была

направлена на цель, фактически система продукций решала более общую задачу, а не ту, что требовалось.

Теперь постараемся ответить на вопрос, какие числа, меньшие или равные 60, можно получить из множителей 2, 3, 5, используя обратную систему продукций. Сразу ясно, что придется 60 раз задать вопрос, представимо ли очередное число. При этом, используя, например, поиск в глубину, будет найден ответ для каждого числа, причем число применений правил минимум = 0 (если проверяемое число простое), максимум = 5. В среднем — около двух применений правил для проверки одного числа. Это означает, что обратная система в данном случае на порядок хуже, и нет необходимости загромождать книгу утомительным описанием ее работы, — столбцами ненужных чисел. С другой стороны, если потребовать просто разложить число 60 на множители, то обратная система, очевидно, будет более эффективна.

Вывод из этого примера таков: в искусственно-интеллектуальных системах очень важно учитывать детали задач. *Иногда учет деталей важнее следования общим принципам*.

2.3.2. Двусторонние системы продукций

Прямые и обратные системы называются *односторонними*, так как поиск двигается строго в одном направлении — к цели или от нее. Однако, если существует только одно целевое состояние и одно исходное состояние, то направления эквивалентны (например, при игре в восемь). В таком случае можно одновременно двигаться от данных к цели и от цели к данным, проверяя, не появилось ли общее состояние, которое свидетельствует о том, что решение найдено. Это принцип действия *двусторонней системы продукций*.

Можно ожидать, что если используется малоинформированная стратегия управления, то двусторонняя система продукций окажется эффективнее односторонней, поскольку суммарно меньше узлов будет раскрыто, например, при поиске в ширину (рис. 2.13).

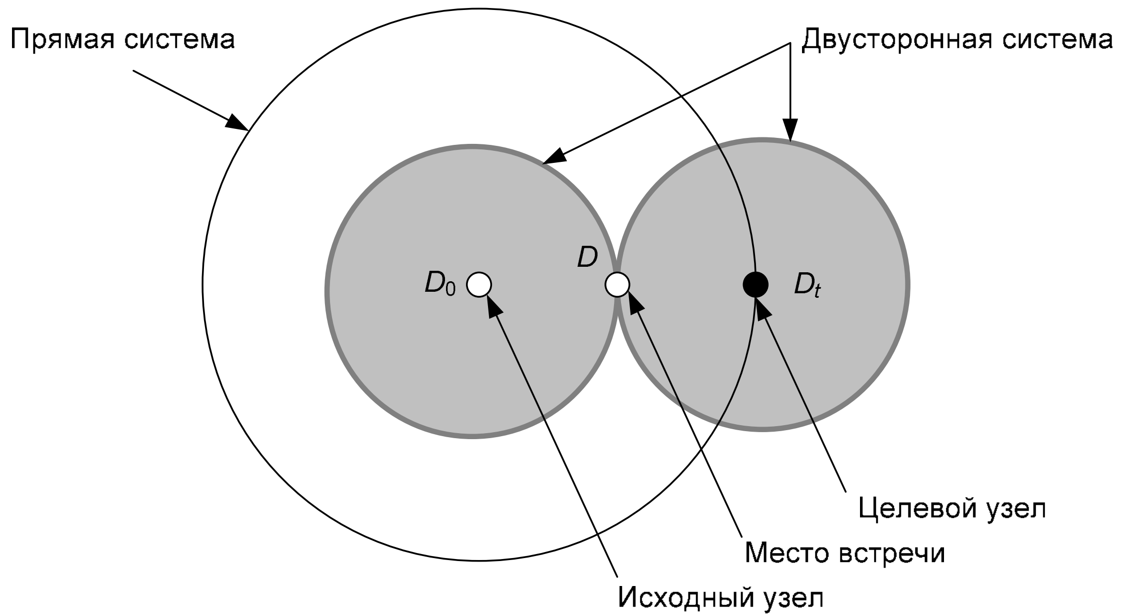

Рис. 2.13. Преимущество двусторонней системы продукций

Однако это не всегда так. Если используются очень хорошие эвристики, которые целенаправленно ведут от данных к цели и от цели к данным, но при этом разными путями, то может оказаться, что двусторонняя система хуже, чем любая односторонняя (рис. 2.14)!

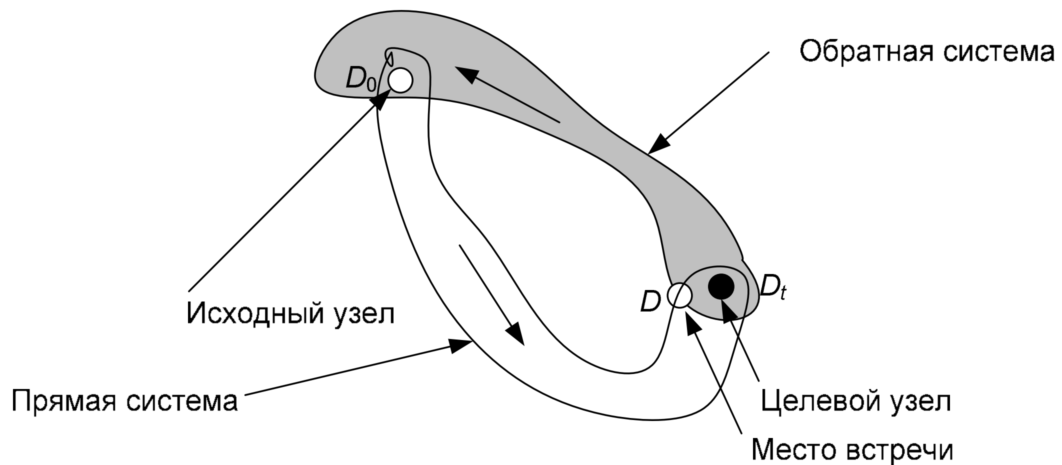

Рис. 2.14. Двусторонняя система с направленными эвристиками

2.3.3. Коммутативные системы продукций

*Система продукций с множеством правил **R** называется **коммутативной**, если выполнены следующие три условия:*

$$\forall d \in \boldsymbol{D}\ (r1.p\ (d)\ \&\ r2.p\ (d) \rightarrow r2.p\ (r1.f\ (d)))$$

- если два правила применимы, то при применении одного, другое остается применимым;

$$\forall d \in \boldsymbol{D}\ (r.p(d)\ \&\ \boldsymbol{t}(d) \rightarrow \boldsymbol{t}(r.f(d)))$$

- если цель достигнута, то навсегда, применение правил не отменяет достижения цели;

$$\forall d \in \boldsymbol{D}\ (r1.p(d)\ \&\ r2.p(d) \rightarrow r2.f(r1.f(d)) = r1.f(r2.f(d)))$$

- если два правила применимы, то их можно применять в любом порядке.

Рассмотренная в параграфе 2.3.1 система продукций для чисел является коммутативной. Другой пример коммутативной системы продукций — игра в восемь, а вот задача о крестьянине, волке, козе и капусте — это некоммутативная система.

Заметим, что третье условие, условие коммутативности, не означает, что можно произвольно переупорядочить последовательность правил, то есть утверждение $\forall d\ (r2.f(r1.f(d)) = r1.f(r2.f(d)))$, вообще говоря, неверно. Но *те правила, которые применимы в данный момент, можно применять в любом порядке*.

Одним из важных достоинств коммутативной системы продукций является возможность всегда использовать безвозвратный режим, так как применение правила ничего не портит.

Теорема. Всякую СП можно преобразовать в коммутативную.

Доказательство. Пусть задана система продукций S = < **D** , **R**, **C** > с начальным состоянием базы d_0. Рассмотрим ее граф

поиска G. Построим новую систему продукций S' = < **D'**, **R'**, **C'** >. Базой данных системы S' будет граф поиска системы S. Правила **R'** новой системы — это различные способы преобразования графа поиска, которое можно делать с помощью стратегии **C** в старой системе (раскрытие узла). Новая система продукций коммутативна. Действительно:

- если в старой системе два узла находятся в списке *O*, то раскрытие одного узла не препятствует раскрытию второго узла, так что первое условие коммутативности выполнено;

- если в старой системе в графе поиска построен терминальный узел, то раскрытие других узлов не исключит из графа терминального узла, так что второе условие коммутативности выполнено;

- если в старой системе два узла находятся в списке *O*, то их можно раскрывать в любом порядке, получится один и тот же результат, так что третье условие коммутативности выполнено. **Ч.т.д.**

Доказанная теорема мало что дает на практике. Хотя полученная новая система коммутативна, но преобразование резко увеличивает её объем по сравнению с исходной. Выигрыш за счет применения безвозвратного поиска оказывается меньше, чем проигрыш за счет разрастания систем при преобразовании.

Подкрепим общие оценочные суждения примером коммутативной системы для упрощения алгебраических выражений, построенных из: переменной x, натуральных коэффициентов, операций сложения, умножения и возведения в натуральную степень (полиномы одной переменной). Имеется набор правил преобразования:

1. *Раскрыть скобки*
2. *Привести подобные*
3. *Упорядочить по степеням*

Эти правила всем известны со школы, поэтому мы не стали записывать их формально. Попытаемся упростить выражение $1 + x(1 + x(3 + x + x))$ (рис. 2.15).

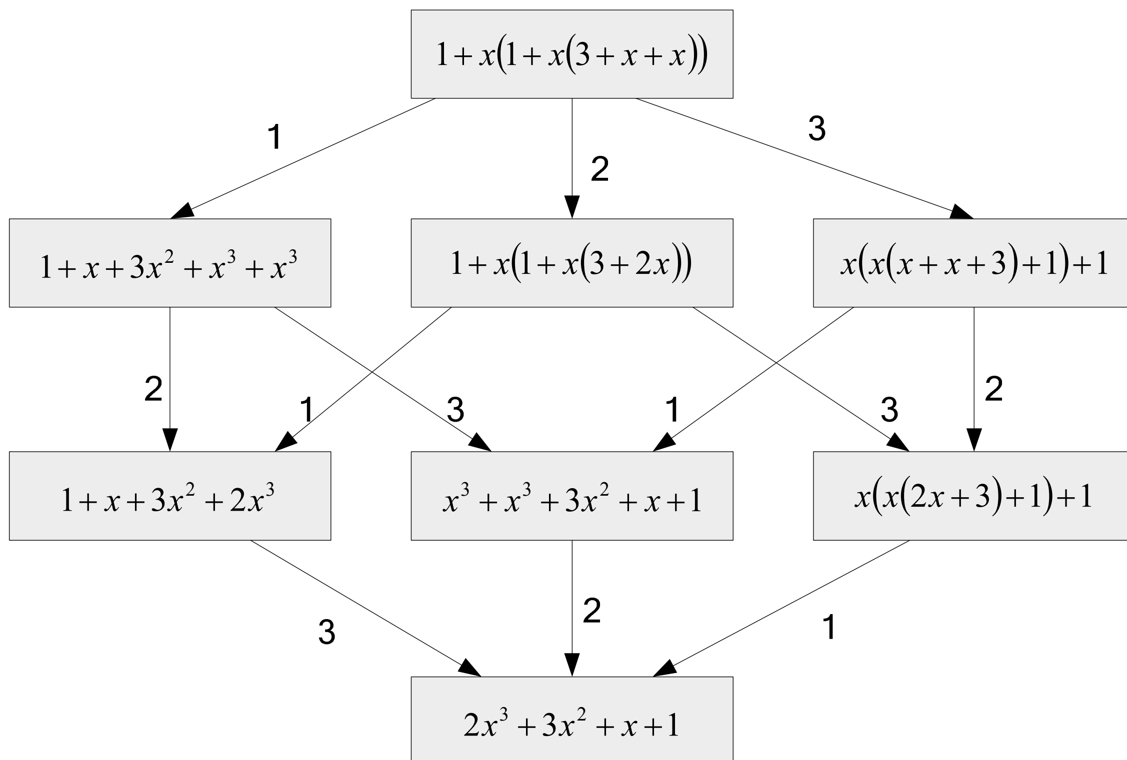

Рис. 2.15. Упрощение выражений

Если правило что-то меняет — применяем его. Для полиномов окончательный результат не зависит от порядка применения правил. Фактически здесь система продукций не содержала условий — это пример *системы подстановки термов*, или *системы правил переписывания* (terms rewriting rules), которые относятся к однородным системам продукций.

Система правил переписывания *(Term-rewriting system) — конечное множество правил переписывания R, определяющее на множестве термов бинарное отношение → (отношение редукции).*

Говорят, что система правил переписывания R обладает свойством Чёрча-Россера, если для любых термов t_1 и t_2 выполняется равносильность $t_1 = t_2$ в том и только в том случае, когда существует единственный терм t_3, такой, что $t_1 \rightarrow t_3$ и $t_2 \rightarrow t_3$. В этом случае, СПТ называется ***полной*** *или* ***канонической****, а*

результат применения правил R к исходной формуле — ***канонической формой***.

В данном случае система обладает свойством Чёрча-Россера. Это очень сильное, полезное и редкое свойство, которое позволяет легко решить автоматически многие задачи, такие как проверка равенств, решение уравнений и т. д. Однако канонические формы существуют не для всех классов алгебраических выражений. Существуют и, наверное, известны читателю канонические формы для полиномов одной переменной и для булевых формул. В то же время, например, для элементарных функций математического анализа (дробно-рациональные функции, логарифм, экспонента и их суперпозиции) нормальной формы не существует[45].

Покажем существенность свойства Черча-Россера путем, на первый взгляд, незначительного усложнения рассматриваемого примера упрощения алгебраических выражений. Попытаемся упростить выражение $x(1 + 1/x)$, которое в результате должно превратиться в $x + 1$. Для этого в наш пример необходимо добавить операцию деления и два правила:

4. *Привести к общему знаменателю*

5. *Сократить числитель и знаменатель*

Легко видеть, что теперь результат зависит от порядка применения правил! То есть, система продукций уже не является коммутативной. Более того, в некоторых случаях возможны тупики в поиске. В результате один из путей не редуцируется правилами 1-5 (рис. 2.16).

---

[45] Это объясняет, почему задачи, предлагаемые на экзаменах по элементарной математике («решить уравнение», «упростить выражение»), действительно иногда являются задачами, которые для своего решения требуют изобретательности и фантазии, а не только механического применения заученных правил. Доказательство отсутствия канонической формы для выражений в элементарных функциях было найдено сравнительно недавно и принадлежит к числу важных математических результатов последнего времени.

Рис. 2.16. Тупики при поиске

Для решения этой проблемы, связанной с образованием тупиковых ветвей, добавим обратные операции:

6. *Вынести за скобки*

Если ввести только правило 6 (вынесение за скобки), то будут два взаимно-обратных правила: раскрытие скобок и вынесение за скобки, а это значит, что возможно зацикливание, то есть свойство Черча-Россера не выполняется по-прежнему. Чтобы тупик на рис. 2.16 сделать редуцируемым, необходимо добавить правило 7:

7. *Разложить дробь*

$$(u + v) / w = u / w + v / w$$

Но тогда может появиться бесконечное зацикливание, такое, которое при безвозвратном поиске невозможно распознать (на рис. 2.17 один из возможных циклов обведен овалом).

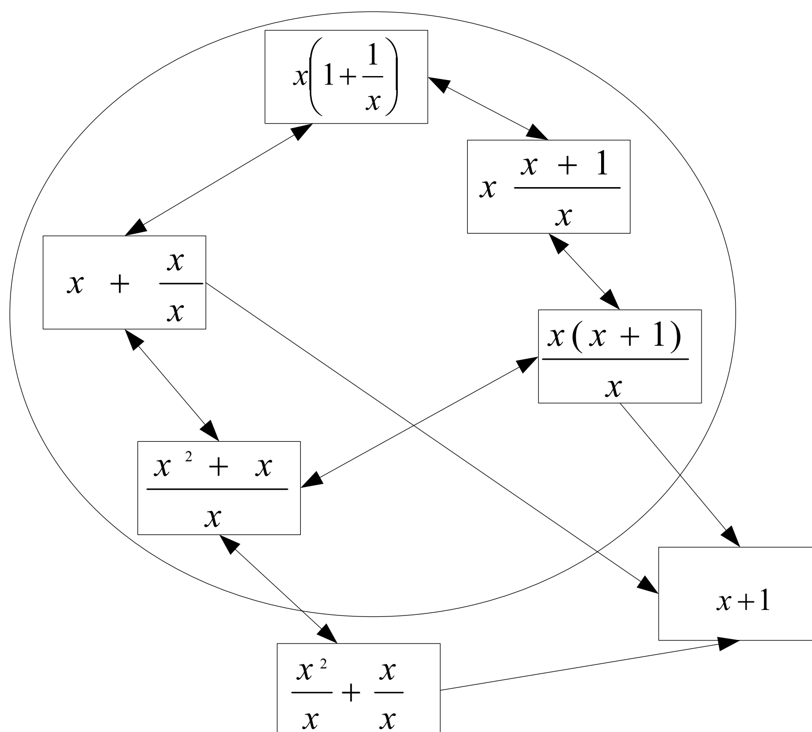

Рис. 2.17. Циклы при поиске

Вывод по данному примеру: коммутативность системы продукций — очень мощное и полезное свойство. *Нужно стремиться строить коммутативные системы продукций, а в тех случаях, когда это невозможно, стоит проверить, нельзя ли немного ограничить задачу, но так, чтобы стало возможным построить коммутативную систему.*

2.3.4. Разложимые системы продукций

Коммутативность системы продукций предоставляет свободу выбора применяемого правила. Это не единственное свойство, которое дает такую свободу.

*Если исходная система продукций может быть **разложена** (splitting) в сумму систем продукций, которые могут рассматриваться независимо, то такую систему называют **разложимой системой продукций**.*

Хороший пример разложимой системы продукций дают формальные порождающие грамматики. Рассмотрим простой пример на эту тему. Пусть имеются правила

```
A → BC
A → CT
C → TT.
```

Вопрос к системе продукций: можно ли из слова `AC` получить цепочку, содержащую только символ `T`? Ниже представлены графы поиска для обычной (рис. 2.18) и разложимой (рис. 2.19) системы продукций. В первом случае мы осуществляем все возможные переходы из начального выражения. Во втором случае мы сначала разлагаем исходную базу данных (слово `AC`) на две части (слово `A` и слово `C`), а затем строим графы поиска для каждой части. Здесь представлены полные графы поиска, и из них видно, что решение получить можно. Действительно, на рис. 2.18 существуют пути (их три, листовые узлы выделены цветом), ведущие из начального узла в терминальный, а на рис. 2.19 имеется *одна пара* путей, ведущий в терминальные узлы для *каждой* из двух частей разложения. Узлы, входящие в решение, на рис. 2.19 обведены овалом.

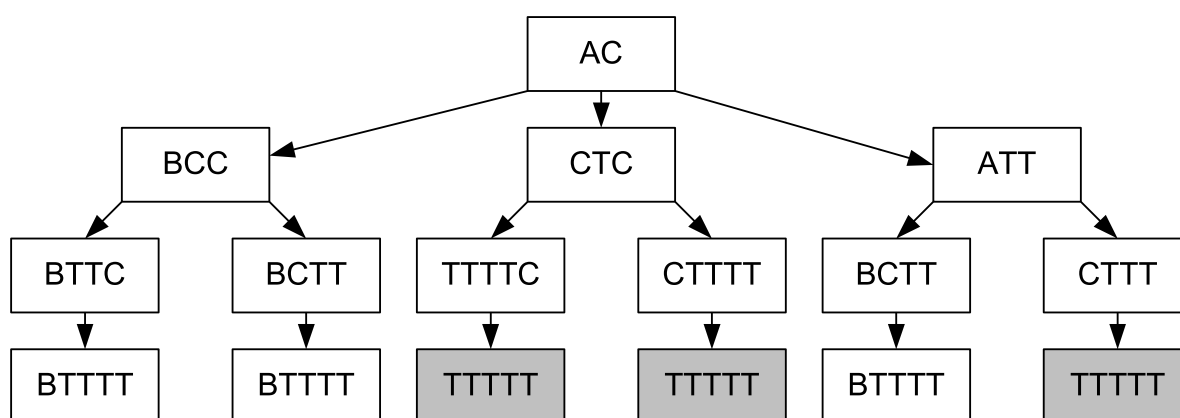

Рис. 2.18. Граф поиска для обычной системы продукций

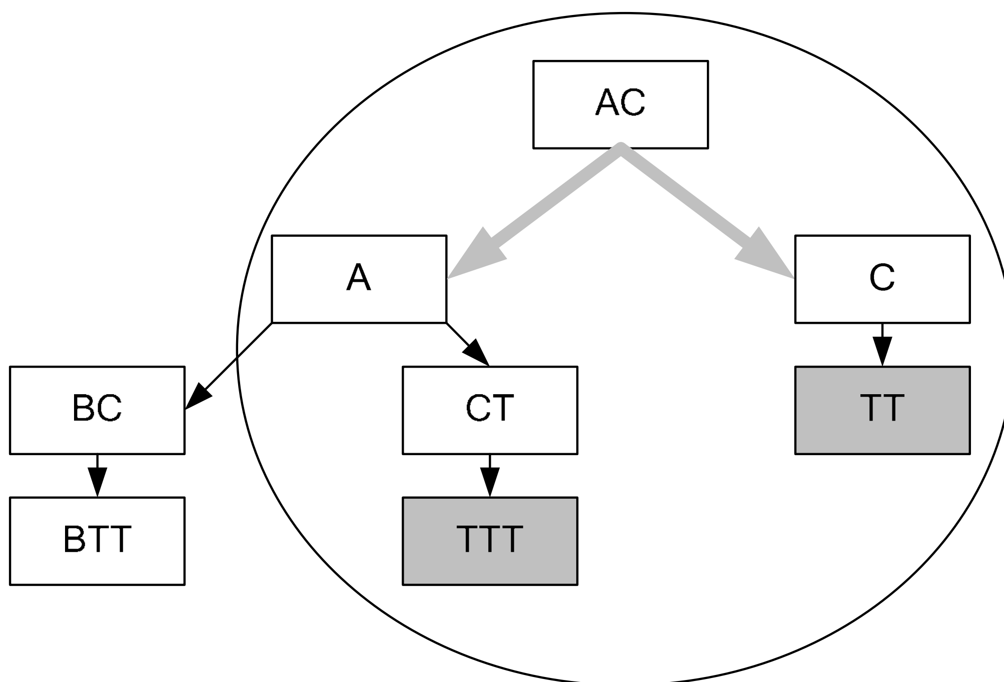

Рис. 2.19. Граф поиска для разложимой системы продукций

Если сравнить рис. 2.18 и рис. 2.19, то легко заметить, что в первом случае проделывается много лишней работы: результат применения правил в левой ветви приводит в тупик. Использование разложения в этом примере сокращает граф поиска более чем в два раза!

Основная процедура для разложимой системы продукций представлена в алгоритме 2.8.

Алгоритм 2.8. Интерпретатор разложимой системы продукций.

```
proc Split (d, R, t)
    Δ := Decompose (d)
    while true do
        for d∈Δ do
            if t(d) then Δ := Δ \ {d} end if
        end for
        if Δ = ∅ then return (OK) end if
        d := SelectD (Δ)
```

```
        Δ:= Δ \ {d}
        r := Select (d, R)
        if r = nil then return (fail) end if
        Δ := Δ ∪ Decompose (r.f(d))
    end while
end proc
```

Это аналог процедуры `Production`, самой первой и общей (см. параграф 2.1.2). Процедура `Decompose` разлагает базу данных на несколько частей (или оставляет без изменения, если в данном состоянии базу разложить невозможно). В известном смысле это похоже на операцию раскрытия узла, но все-таки отличается от нее, поэтому на рис. 2.19 дуги, соответствующие разложению, а не раскрытию, выделены серым цветом. Далее, функция `SelectD` аналогична функции `Select`, она фактически выбирает узел для дальнейшей работы.

Сделаем важное замечание. С точки зрения теории графов, та дуга, которая появляется в графе поиска в результате применения правила разложения, является гипердугой — дуга, исходящая из разлагаемой базы и ведущая сразу в несколько баз, являющихся частями разложения. Для того чтобы прийти к решению, нужно найти решение для первой части И для второй части И так далее. Поэтому такая гипердуга называется И-дугой. В случае если из узла выходят обычные дуги, соответствующие применению правил, то они имеют следующий смысл: чтобы найти решение, нужно применить первое правило, ИЛИ второе правило, ИЛИ и так далее. Поэтому такие дуги называются ИЛИ-дугами. Если в графе применяются только такие дуги, он называется *графом И/ИЛИ*. В следующих двух главах мы вернемся к этим понятиям и обсудим их более детально, см. параграфы 3.3 и 4.5.1.

Для описания языков программирования и других искусственных языков с успехом используется теория формальных грамматик, разработанная А. Н. Хомским. Вкратце ее суть

заключается в следующем. Рассматривается конечный алфавит A (множество символов) и язык над этим алфавитом. Здесь язык — это просто множество последовательностей символов алфавита (цепочек). Другими словами, среди всех возможных цепочек (которых заведомо бесконечно много) нужно иметь возможность выделить те и только те, которые являются цепочками языка (еще говорят «входят в язык», «принадлежат языку», «являются выражениями языка»).

Для описания (бесконечного) множества цепочек Хомский предложил использовать формальную *порождающую грамматику* — конечное множество правил вида $\alpha \rightarrow \beta$, где α и β конечные цепочки, составленные из символов алфавита A (которые называются терминальными символами) и из символов некоторого дополнительного алфавита N (эти символы принято называть нетерминальными). В левую часть правила (т. е. в цепочку α) обязательно должен входить хотя бы один нетерминальный символ. Среди всех нетерминальных символов выделяется один, который называется начальным символом (или аксиомой) грамматики.

Далее все очень просто. Взяв за основу аксиому грамматики, разрешается строить новые цепочки, заменяя в уже построенной цепочке любое вхождение левой части любого правила на правую часть этого правила. И так до тех пор, пока в цепочке не останутся одни терминальные символы. Всякая цепочка терминальных символов, которую можно построить таким процессом, по определению входит в язык, а если цепочку нельзя так построить, то она не входит в язык.

Разумеется, все это так просто, пока не поставлены разные каверзные вопросы. Всякий ли язык может быть определен таким способом? Если дана грамматика и конкретная терминальная цепочка, то, как узнать, принадлежит она языку или нет? Однозначен ли процесс порождения цепочки? Зависят ли ответы на предыдущие вопросы от вида (формы) правил?

Но если основная идея ясна, то можно оставить эти (весьма трудные) вопросы математикам и принять тот факт, что синтаксис подавляющего большинства языков программирования легко описывается подклассом формальных грамматик, которые называются контекстно-свободными (левая часть всех правил состоит из одного нетерминального символа). Для этого подкласса грамматик все вопросы, связанные с распознаванием принадлежности цепочки языку (это называется синтаксическим анализом, или разбором) надежно решены.

ВЫВОДЫ

1. Системы продукций — это исторически первый и до сих пор часто применяемый способ представления знаний.

2. Системы продукций эффективно программно реализуемы и являются основой многих практических программных систем.

3. Использование систем продукций может принести пользу только при полном понимании всех сопутствующих им ограничений и недостатков.

3. АЛГОРИТМЫ ПОИСКА РЕШЕНИЯ

Алгоритмы поиска решения столь многочисленны и разнообразны, что использование этого термина без надлежащих комментариев может ввести в заблуждение. В общем случае *алгоритм поиска ищет решение поставленной задачи в некотором пространстве поиска*. Пространство поиска может быть дискретным, а может быть непрерывным; может быть конечным, а может быть потенциально бесконечным; метрические свойства пространства могут быть заданы, а могут быть неизвестны; характеристики искомого решения могут быть фиксированы, а могут меняться в процессе поиска. Во всех этих и во многих других случаях применяются самые разнообразные алгоритмы, которые, порой, имеют весьма мало общего. По нашему мнению, исчерпывающий обзор всех алгоритмов поиска вообще вряд ли возможен технически, да и ценность такого гипотетического обзора была бы сомнительна. Поэтому мы ограничиваемся одной конкретной областью — *алгоритмами поиска решения в прикладных системах с элементами искусственного интеллекта*.

Даже при таком ограничении подробная аналитика всех используемых алгоритмов представляется чрезмерно объемной. Мы позволим себе еще больше сузить область рассмотрения, ограничившись анализом наиболее популярных и представительных подходов к *поиску вывода в системах, опирающихся на продукции*, т. е. использующих в качестве базовых конструкции вида $p \to q$, введённые во второй главе. В сделанных соглашениях пространство поиска естественно представлять графом (или сетью) специального вида, в котором узлы представляют состояния базы фактов, а дугам

соответствует применение продукций[46]. *Результатом поиска решения на графе всегда является некоторый маршрут (цепь, вывод), ведущий от начального узла в целевой узел*. Новое сужение области исследований, разумеется, упрощает нашу задачу, однако и теперь *мы не можем выделить одного наилучшего универсального алгоритма поиска решения!* Дело в том, что очень важную (если не определяющую) роль при поиске решения играют конкретные свойства узлов и дуг графа поиска, определяемые прикладным назначением, а, следовательно, и природой самих продукций.

Таким образом**,** *задачу поиска решения в системе продукций можно свести к задаче поиска маршрута на графе*. Подбирая подходящий алгоритм для решения конкретной задачи на графе, мы должны принимать во внимание множество характеристик графа поиска (за расшифровкой терминов направляем, например, к книге [17]), а именно: является ли граф ориентированным (орграф), допускаются ли в нем циклы (и какой эти циклы природы), могут ли два узла соединяться более чем одной дугой (мультиграф), является ли граф многодольным (и какой степени), что нам известно о связности и степенях вершин графа, какова мощность множеств узлов и дуг (является ли граф конечным), является ли граф размеченным, взвешенным (и какова комбинация этих свойств) и многое другое. Важно понимать, что все эти свойства задаются постановкой задачи, точнее выбором представления для поставленной задачи, и не являются произвольными. Например, если у нас есть хороший алгоритм для ациклических графов, но граф поиска для данной задачи содержит циклы, то от этого алгоритма мало толка.

---

[46] Представление пространства поиска в виде ориентированного графа действительно является естественным и широко распространенным, но оно ни в коем случае не является обязательно наилучшим или единственно возможным. Существуют и используются альтернативные представления, некоторые из них рассматриваются в последующих главах этой книги.

В последнее время значительное внимание привлекают задачи, характеризующиеся в самом общем виде, как *задачи построения оптимального маршрута*. Примерами соответствующих прикладных областей могут выступать логистика (минимизация времени транспортировки грузов между складами), управление движением роботов (в среде с ограничениями и препятствиями), стратегические игры и многое другое. Графы поиска для подобных задач являются «вполне приличными» — ориентированными однодольными орграфами — узлом графа обычно выступает просто имя (или точка пространства, как правило, метрического и небольшой размерности), плотность невысока (далеко не все узлы связаны друг с другом непосредственно), степень узла d всегда оценивается ограниченной константой ($\deg(d) < const \ll n$, где n — число узлов графа), вес на дуге определяет только стоимость перехода из одного узла в другой, исходный и целевые узлы однозначно определены. При этом считается, что необходимая для решения задачи построенная часть графа поиска может быть целиком размещена в «быстрой» (оперативной) памяти компьютера. Основная рассматриваемая в этой главе задача состоит в построении оптимального (минимизирующего ресурсные затраты) маршрута от исходного узла к целевому.

В главе рассмотрены пять алгоритмов:

- общий алгоритм эвристического поиска решения для систем неоднородных продукций (раздел 0);
- формальные свойства исторически первого и самого известного алгоритма поиска на графе (раздел 3.2);
- изящный способ повышения эффективности и практической применимости общего алгоритма поиска решения (раздел 3.3)
- алгоритм поиска на графах И/ИЛИ, который является специализацией общего алгоритма, возникающей, например, при использовании разложимых систем продукций (раздел 3.4);
- алгоритмы поиска на игровых деревьях, которые являются, с одной стороны, дальнейшей специализацией общих алгоритмов,

поскольку игровые деревья — это частный случай графов И/ИЛИ, и, с другой стороны, являют собой наиболее проработанные на сегодняшний день практические алгоритмы реальных программ с элементами ИИ указанных в начале главы классов (раздел 3.5).

Такой набор рассматриваемых алгоритмов образует систему, достаточную для того, чтобы читатель после прочтения главы мог уверенно ориентироваться в море разнообразных конкретных алгоритмов поиска — основные маяки мы обозначили.

3.1. ЭВРИСТИЧЕСКИЙ ПОИСК

Исторически одним из первых и самым известным в ИИ алгоритмом для поиска оптимальных (по суммарному весу или длине) путей в графе пространства поиска считается алгоритм A* (читается как «А-звездочка», или «А-стар»). Этот алгоритм был впервые описан в 1968 году Питером Хартом, Нильсом Нильсоном и Бертрамом Рафаэлем [18]. В их работе он упоминается как «алгоритм A». В классическом варианте алгоритм A* применим в дискретном пространстве поиска, представленном графом поиска с нагруженными дугами. Этот эвристический поиск сортирует узлы графа по значению функции, которая оценивает длину кратчайшего пути, проходящего через данный узел, и идущего от начального узла к одному из целевых. Таким образом, этот *алгоритм сочетает в себе учет длин оптимальных путей из уже исследованной части графа поиска из алгоритма Дейкстры с эвристикой наискорейшего спуска*[47] *для еще неисследованной части графа*.

3.1.1. Общий алгоритм

Для уточнения дальнейшего изложения повторим определение графа поиска, сделаем несколько важных замечаний и выпишем общий вид алгоритма поиска на графе.

[47] Метод наискорейшего спуска описан в параграфе 2.2.2.

Граф поиска *— это ориентированный граф G (V, E), в котором множество узлов* $V = \{d \mid d \in \mathbf{D}\}$ *— это множество возможных состояний базы фактов (в смысле предыдущей главы), а множество дуг E определяется следующим образом:*

$$E = \{ d_1 \to d_2 \mid d_1, d_2 \in \boldsymbol{D} \;\&\; \exists\, r \in R \;(\, r.p(d_1) \;\&\; d_2 = r.f(d_1)\,)\, \}.$$

В случае, когда граф поиска существенно конечен, его можно задать явно с помощью обычных способов представления графов в компьютере. Зачастую, в подобных случаях именно граф поиска выбирается в качестве механизма представления знаний — пометки на дугах-переходах являются продукциями, отдельно задавать множество продукций нет необходимости.

В случае бесконечного (или очень большого) графа мы не можем использовать указанные простые представления, просто потому, что весь граф не помещается в памяти компьютера. Вместо этого мы, *начиная с исходного состояния базы данных* d_0*, с помощью конечного множества правил R порождаем некоторую часть графа поиска, до тех пор, пока либо не будут исчерпаны выделенные ресурсы, либо порожденная часть не будет содержать целевое состояние базы данных* d_t. Таким образом, используется неявное (имплицитное) представление графа в компьютере. Неявно граф может быть задан с помощью операции раскрытия узла.

Считается, что каждому правилу *r* сопоставлена стоимость его применения, обозначаемая *r.w* и называемая *стоимостью* или *весом* (**w**eight) применения правила. В этом случае, в графе поиска каждая дуга будет иметь вес. Другими словами, дуги нагружены, причем величину нагрузки можно считать положительной:

$$r = (p, f, w), \qquad w > 0.$$

Стоимость пути в графе определяется как сумма стоимостей входящих в него дуг. Общий алгоритм поиска на графе, который

вычисляет стоимость путей от исходной вершины и (при выполнении определенных условий!) определяет путь минимальной стоимости[48], выглядит следующим образом.

Алгоритм 3.1. Общий алгоритм поиска на графе.

```
proc GS (d0, R, t)
    O := {d0};   g(d0) := 0
    C := ∅;      P(d0) := nil
    while true do
        d := Select (O)
        if d = nil then return (fail) end if
        if t(d) then return (OK) end if
        Open (d)
    end while
end proc

proc Open (d)
for r ∈ R do
        if r.p(d) then {правило применимо}
            d′ := r.f(d)
            c′ := g(d)+r.c
            if d′ ∈ O ∨ d′ ∈ C then {старый узел}
                if c′ < g(d′) then {путь лучше}
                    g(d′) := c′
                    P(d′) := d
                end if
            else { новый узел }
                O := O+{d′}
                g(d′) := c′
                P(d′) := d
            end if
```

---

[48] Можно использовать геометрическую терминологию. В этом случае величину нагрузки на дугу называют *длиной дуги*. Сумма длин дуг, входящих в путь, называется *диной пути*, а задача формулируется как задача отыскания *кратчайшего пути*.

```
        end if
end for
O := O - {d}
C := C + {d}
end proc
```

где O — список «открытых», C — список «закрытых» узлов (см. параграф 2.2.4).

Помимо обычных параметров и уже обсуждавшихся списков открытых и закрытых узлов, процедура использует еще две структуры данных. Первая из них — $g(d)$ — длина кратчайшего пути от d_0 до d из тех, что исследованы к этому моменту. Заметим, что $g(d_0) = 0$, для прочих узлов в начале работы $g(d)$ неизвестно и по мере работы алгоритма значение $g(d)$ для данного узла d может разве что уменьшаться.

Особого обсуждения заслуживает структура данных $P(d)$ — эта структура хранит для узла d предшествующий узел на найденном кратчайшем пути от d_0 до d. Такой прием позволяет эффективно хранить множество путей, ведущих от узла d_0 сразу ко всем исследованным узлам.

В этом алгоритме поиска на графе есть две функции — `Select` и `Open`. Неформально говоря, первая из них есть «поиск», а вторая — «на графе».

Функция `Select` каким-то образом определяет очередной узел для раскрытия. Если функция `Select (O)` работает по принципу FIFO (First In First Out), то есть, если список O — это очередь, то происходит поиск в ширину. Этот метод гарантирует нахождение пути к целевому узлу, если такой путь существует. Если функция `Select (O)` работает по принципу LIFO (Last In First Out), то есть, если список O — это стек, то происходит поиск в глубину; этот метод подобен поиску с возвратами, но хранится уже не один текущий, а все просмотренные пути.

Поиск в ширину и поиск в глубину — это примеры неинформированных процедур поиска. Если про задачу ничего

неизвестно, то приходится применять неинформированные процедуры, но на многое при этом рассчитывать не приходится. Для эффективного поиска решения необходимо использование дополнительной информации о задаче. Такая дополнительная информация часто называется эвристической. Смысл этого термина раскрывается в следующем параграфе.

3.1.2. Эвристические алгоритмы

Древнегреческое слово «эвристика» (ευρίσκω — отыскиваю, открываю) имеет много близких, но различных значений и способов употребления в современном языке. В литературе можно, среди прочих, найти следующие толкования этого понятия.

Эвристические методы — специальные методы решения задач, которые обычно противопоставляются формальным методам решения, опирающимся на точные математические модели. Использование эвристических методов не гарантирует, что получаемые решения являются наилучшими, неудачная эвристика может привести и к получению наихудшего результата! В то же время известно множество удачных эвристик (для решения конкретных задач), обеспечивающих[49] получение приемлемых решений за допустимую цену.

Эвристическая деятельность — организация процесса продуктивного творческого мышления. В этом смысле эвристика понимается как совокупность присущих человеку механизмов

---

[49] Подчеркнем еще раз тот факт, что мы можем даже не вполне понимать, *как работает* та или иная «удачная» эвристика и *почему она обеспечивает* «приемлемость» решения! Всё это, в некоторой степени, сродни волшебству, но волшебству конкретному, действующему, а потому — *полезному*!

решения творческих задач. Эти механизмы универсальны по своему характеру и не зависят от конкретной решаемой проблемы.[50]

Эвристическое программирование — способ написания программ, при котором программист не кодирует готовый математический метод решения, а пытается формализовать тот интуитивно понимаемый метод решения задачи, которым, по его мнению, пользуется человек при решении подобных задач. Как и эвристические методы, эвристические программы не дают абсолютной гарантии достижения поставленной цели и оптимальности получаемого результата.

Эвристическое обучение — специальный метод обучения (*сократические беседы*) или коллективного решения проблем, исторически восходящий к Сократу. Метод состоит в задании обучающимся серии наводящих вопросов и примеров, в результате чего они как бы сами отыскивают решение.

В нашем случае мы понимаем термин «эвристика» в более узком смысле.

Эвристика *— эмпирическое*[51] *правило, упрощающее или ограничивающее перебор вариантов при поиске решения в конкретной предметной области.*

Более того, поскольку здесь мы рассматриваем совершенно конкретный способ поиска решения в данной предметной области — поиск на графе для системы продукций, мы можем еще более конкретизировать определение.

---

[50] Эврика! (буквально — я нашел!), согласно преданию, восклицание Архимеда при открытии им основного закона гидростатики. В переносном смысле это выражение радости, удовлетворения при решении какой-либо сложной задачи или возникновении новой идеи.

[51] Это слово в определении является чрезвычайно важным! Оно подчеркивает тот факт, что правило *не является формально обоснованным* и обеспечивает «упрощение» и «ограничение» *по чьему-то мнению и вовсе не обязательно*.

Эвристика *— набор правил выбора очередного узла для раскрытия, который должен, по мнению автора эвристики, вести к решению проблемы.*

К сожалению, эвристика опирается на догадки и интуицию, а не на точные знания, эвристика использует доступную, но ограниченную информацию, такую, как описание множества текущих открытых узлов, и не может гарантировать, что выбор действительно наилучший. Эвристика — это лишь предложение для выбора следующего шага, основывающееся на опыте и здравом смысле.

Алгоритм, использующий эвристики, называется ***эвристическим алгоритмом****.*

Эвристики уменьшают вычислительные затраты на применение правил за счет того, что сокращают общее количество применений правил. Но применение самих эвристик также требует каких-то вычислений и привносит свои затраты. Кроме того, манипуляции с графом поиска, особенно если он велик, также требуют затрат (еще раз см. рис. 2.5).

На самом деле, нас интересует общая сумма, складывающаяся из затрат на применение правил, затрат на проверку и раскрытие узлов и затрат на эвристику. Точнее говоря, мы хотим минимизировать среднее значение этой суммы по всем случаям применения эвристического алгоритма.

Чтобы поставить задачу точно, необходимо дать определения и ввести обозначения, что мы и делаем в следующем параграфе.

3.1.3. Оценочная функция

Эвристическая информация может иметь любую форму. В ИИ принято представлять эвристическую информацию в виде *оценочной функции* $F(d)$.

Оценочная функция F определена на множестве допустимых состояний среды (базы фактов) и принимает вещественные числовые значения $F\colon \boldsymbol{D} \to \Upsilon$. Причем обычно выбирают оценочную функцию таким образом, чтобы на терминальных узлах ее значение было

минимально $F(d_t) = \min \{d \in \boldsymbol{D} \mid F(d)\}$. Если задана такая оценочная функция, то можно записать функцию `Select`, используя идею наискорейшего спуска (алгоритм 3.2).

Алгоритм 3.2. Функция выбора узла для раскрытия.

```
Proc Select (O)
    m := ∞
    d′ := nil
    for d ∈ O do
        if F(d) < m then
            d′ := d
    m := F(d)
        end if
    end for
    return (d′)
end proc
```

Или, короче:

```
Select (O) : = min F(d), d ∈ O
```

Для оценочной функции F предложено много различных идей реализации. Слово «оценочная» означает, что F (предположительно) близка к каким-либо важным характеристикам поиска. Ниже приведен список некоторых характерных свойств, которые часто пытаются использовать при выборе оценочной функции:

- величина, обратная вероятности того, что d принадлежит оптимальному пути;
- расстояние от узла d до целевого множества узлов T;
- вес пути от исходной вершины d_0 до целевой вершины d_t через текущую вершину d.

В этом разделе мы рассматриваем именно последний случай.

Будем рассматривать оценочную функцию в следующей форме:

$$F(d) = g(d) + h(d),$$

где g(d) — фактическая стоимость пути от исходной вершины d_0 до d (уже определена нами в процедуре `Open`); h(d) — эвристическая оценка стоимости пути от d до целевой вершины d_t.

Рассмотрим процесс выбора оценочной функции на примере игры в 8 [13] (уже обсуждавшейся нами в параграфе 2.1.4). Введем оценочную функцию F(d) как сумму глубины в дереве (соответствует функции g(d)) и числа шашек не на своем месте (соответствует функции h(d)). Мы видим (рис. 3.1), что с такой оценочной функцией поиск на графе оказывается весьма эффективным и быстро приводит к решению.

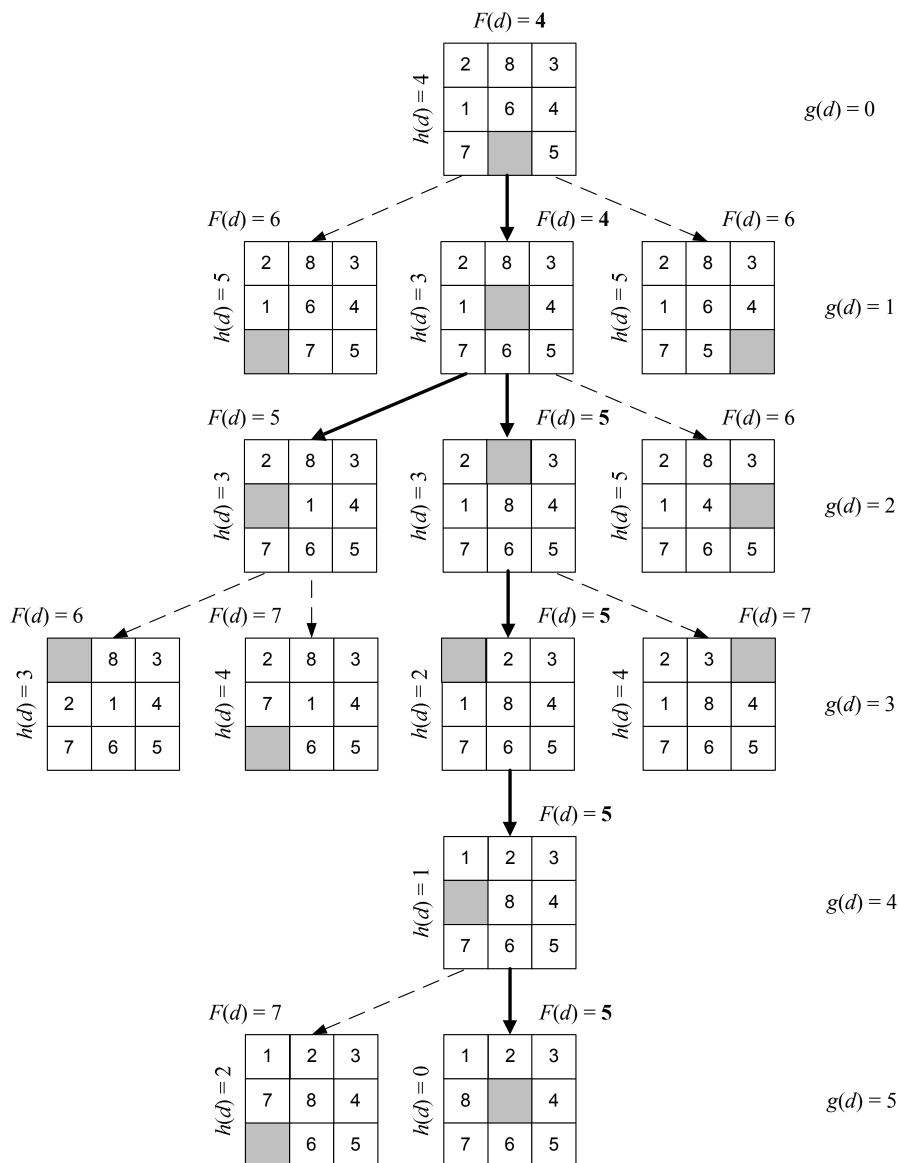

Рис. 3.1. Граф поиска для игры в 8 с оценочной функцией *F*

Сделаем несколько наблюдений по этому примеру:

- $h(d_t) = 0$, так как в целевой позиции все шашки находятся на своем месте по определению;

- $h(d)$ не превосходит минимального числа ходов, которые необходимо совершить, чтобы перейти из d в d_t;

- значение функции $F(d)$ для раскрываемых узлов не убывает; если взять функцию F только как глубину в дереве, то есть положить $h = 0$, то получится просто поиск в ширину.

В следующем разделе мы обобщим и истолкуем эти эмпирические наблюдения.

3.2. СВОЙСТВА АЛГОРИТМА А*

Алгоритм А* обладает целым рядом примечательных и полезных свойств, которые объясняют его широкую применимость. Самым важным свойством является то обстоятельство, что, несмотря на эвристический характер алгоритма, известно достаточное[52] условие, при выполнении которого алгоритм А* гарантировано находит оптимальное решение, если оно существует. Это строго показано в первом параграфе данного раздела. Во втором параграфе показано, что алгоритм А*, образно говоря, «добро помнит» — чем более информирована предоставленная алгоритму оценочная функция, тем «лучше» ведет себя алгоритм. Наконец, в третьем параграфе раздела показано, что если оценочная функция выбрана разумно, если она согласована с метрическими свойствами пространства поиска, то алгоритм А* не только находит оптимальное решение в конце, но и все промежуточные состояния лежат на оптимальном пути. Другими словами, в *некоторых случаях алгоритм А* ведет себя просто идеально:* наилучшим возможным образом. В последнем параграфе мы констатируем, что идеальное

---

[52] Условие именно достаточное, но не необходимое. Даже если условие не будет выполнено, алгоритм А* может найти решение, если повезет, но гарантировать этого нельзя.

поведение не всегда гарантировано: иногда А* оставляет желать много лучшего.

3.2.1. Теорема о состоятельности

Дадим необходимые определения и введем обозначения. Длину кратчайшего пути[53] из узла d_1 в узел d_2 обозначим $s(d_1, d_2)$. Пусть

$$F^*(d) = g^*(d) + h^*(d)$$

есть точная длина кратчайшего пути от узла d_0 к узлу d_t, проходящего через узел d. Здесь $g^*(d)$ — длина кратчайшего пути от d_0 к d, $h^*(d)$ — длина кратчайшего пути от d к d_t, то есть $g^*(d) = s(d_0, d)$, $h^*(d) = s(d, d_t)$.

Исходя из определений, легко показать, что $\forall d\ F^*(d) \geq s(d_0, d_t)$. Действительно, путь, составленный из двух кратчайших отрезков и проходящий через заданный узел d, не может быть короче самого кратчайшего пути. Обозначим $M = F^*(d_0) = h^*(d_0) = s(d_0, d_t)$ — это оптимум, длина кратчайшего пути, без всяких ограничений.

Заметим, что функции F^*, g^*, h^* и число M нам, вообще говоря, неизвестны.

Напомним, что мы рассматриваем оценочную функцию в следующей форме:

$$F(d) = g(d) + h(d),$$

где функция g — уже определена нами в процедуре `Open`. В процессе работы алгоритма А* для $g(d)$ справедлива оценка: $g(d) \geq g^*(d)$, так как возможно еще не все пути построены, и при раскрытии узлов и построении новых путей оценка $g(d)$ может уменьшаться, но не может стать меньше $g^*(d)$.

---

[53] Иногда в популярной литературе по ИИ эту величину называют *расстоянием*. Строго говоря, это некорректно, потому что расстояние должно быть симметричной функцией, а достижимость в орграфе вовсе не обязана быть симметричным отношением.

Под вопросом остается функция $h(d)$ — эвристическая оценка расстояния от текущего положения до решения. Ясно, что в *общем случае мы не можем математически точно указать эту функцию, мы можем только придумывать ее в каждом конкретном случае, более или менее удачно*. Например, функция, выбранная в параграфе 3.1.3 для игры в 8, оказалась очень удачной. Она позволила быстро найти решение.

Насколько случайно это обстоятельство? Какими свойствами должна обладать оценочная функция, определяющая порядок раскрытия узлов, чтобы алгоритм гарантированно находил решение? Ответ на эти вопросы дает приведенная ниже теорема[54].

Говорят, что алгоритм поиска ***состоятелен*** *(или частично корректен), если он находит оптимальный путь, при условии что такой путь существует.*

Теорема. *Если* $\forall d\ h(d) \le h^*(d)$, *то алгоритм* A* *состоятелен.*

Доказательство теоремы основано на серии лемм. Все леммы доказываются при условии, что $h \le h^*$. Лемма 1 и теорема доказываются при условии, что $\forall r \in R\ (r.w > 0)$, то есть вес применения правила строго положителен. Из этого следует, что по смыслу задачи все функции F^*, g^*, h^*, F, g, h и число M неотрицательны.

Лемма 1. *Если алгоритм* A* *не заканчивает свою работу, то оценка раскрываемого узла неограниченно возрастает. То есть* $\forall N\ \exists k\ F(d_k) > N$.

Замечание. Здесь k — это номер шага алгоритма A*, а d_k — узел, раскрываемый на этом шаге.

Доказательство. Пусть $e = \min\ \{r \in R \mid r.w\} > 0$ — минимальный вес правила, а $m = |R|$ — количество правил. Пусть p —

[54] Эту теорему иногда называют теоремой о состоятельности алгоритма A*. Доказательство можно найти, например, в учебнике одного из авторов теоремы [13]. Приводимое здесь доказательство использует идеи авторов теоремы, но отличается структурой и обозначениями.

глубина[55] узла d_k в дереве. Заметим, что самую меньшую глубину узел имеет, когда все ранее раскрытые узлы лежат на предыдущих ярусах дерева и при этом ярусы полностью заполнены. В предельном случае в $p - 1$ ярусе m-ичного дерева в совокупности содержится $1 + m + m^2 + \ldots + m^{p-1} = (m^p - 1)/(m - 1)$ узлов. Следовательно, $k < (m^p - 1)/(m - 1) < m^p$ (при $m > 2$). Тогда $p > \log_m k$. Возьмем теперь $k > m^{N/e}$. Имеем: $F(d_k) > g(d_k) > e\,p > e \log_m k > e \log_m m^{N/e} = N$. **Ч.т.д.**

Лемма 2. *Если существует оптимальный путь, то в списке открытых узлов всегда есть узел, принадлежащий этому пути, оценка которого не превосходит M.*

Доказательство. Пусть $A = <d_0, \ldots, d_t>$ — оптимальный путь. Далее индукция по шагам алгоритма.

База индукции: $d_0 \in A$ и $F(d_0) = h(d_0) < h^*(d_0) = M$.

Индукционный переход. Открытие узла из A оставляет наследника в O. Значит, в списке O всегда есть узел из A, по меньшей мере, один, но может быть и несколько. Пусть теперь $d' \in O$, $d' \in A$, и d' — самый левый узел из таких узлов (см. рис. 3.2), то есть все узлы левее d' на пути уже открыты и находятся в списке C. Тогда $g(d') = g^*(d')$, поскольку отрезок оптимального пути также оптимален. Имеем $F(d) = g(d') + h(d') = g^*(d')+h(d') \leq g^*(d') + h^*(d') = F^*(d') = M$, так как $\forall d \in A\ F^*(d) = M$. **Ч.т.д.**

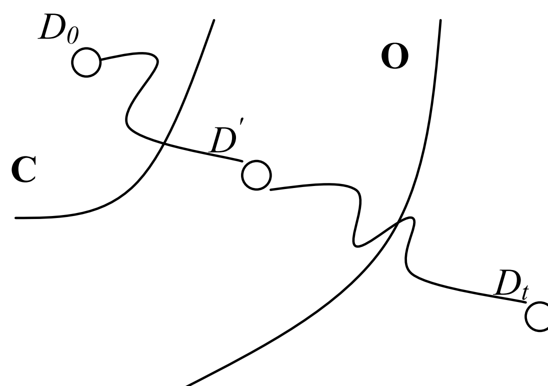

Рис. 3.2. К доказательству Леммы 2

---

[55] *Глубиной узла* в корневом дереве называется число дуг на пути от корня до узла.

Лемма 3. *Если существует оптимальный путь, то открывается всегда узел, оценка которого не превосходит M.*

Доказательство. По лемме 2 $\exists d \in O$ $F(d) \le M$, но функция `Select` всегда выбирает минимум. **Ч.т.д**.

Доказательство теоремы. Доказательство от противного за три шага.

Первый шаг. Пусть $\exists A$ — оптимальный путь и алгоритм не заканчивает свою работу. Тогда по лемме 1 $\exists k$ $F(d_k) > M$, что противоречит лемме 3. Значит, алгоритм заканчивает свою работу.

Второй шаг. Алгоритм может закончиться с результатом **fail** или **OK**. Результат **fail** возможен, только если список O пуст. Но по лемме 2 список O не пуст, значит, алгоритм заканчивается с результатом **OK**.

Третий шаг. Пусть алгоритм A* закончил работу с результатом **OK**, но найденный путь P — не оптимальный путь. Рассмотрим последний вызов функции `Select`, перед тем как алгоритм закончил работу. Пусть функция `Select` выбрала в качестве раскрываемой вершины d_t. Но $h^*(d_t) = 0$, следовательно, $h(d_t) = 0$. Отсюда

$$F(d_t) = g(d_t) + h(d_t) = g(d_t) > g^*(d_t) = F^*(d_t) = M,$$

то есть была выбрана вершина с оценкой больше M, что противоречит лемме 3**.**

Таким образом, доказано, что алгоритм A* заканчивает работу с результатом **OK** и находит при этом оптимальный путь. **Ч.т.д**.

Следствие 1. *Любая вершина $d \in O$, для которой $F(d) < M$, будет раскрыта.*

Доказательство. Последним выбирается для раскрытия узел d_t с оценкой $F(d_t) = M$, а на каждом шаге раскрывается узел с минимальной оценкой. **Ч.т.д.**

Следствие 2. *Поиск в ширину состоятелен.*

Доказательство. При поиске в ширину: $h = 0$, $F = g$, и выполнено условие состоятельности для h: $0 \le h^*$. **Ч.т.д.**

Рассмотрим пример с игрой в 8, приведенный в параграфе 3.1.3. Можно заметить, что предложенная эвристическая функция удовлетворяет условию состоятельности. Действительно, предложенная функция h — число шашек не на своем месте — не может превосходить числа шагов, которые нужны, чтобы поставить все шашки на свои места. Ведь за один ход мы ставим на место не более чем одну шашку!

3.2.2. Сравнение оценочных функций

Пусть даны две оценочные функции:

$F_1(d) = g_1(d) + h_1(d)$,

$F_2(d) = g_2(d) + h_2(d)$.

Причем выполнены условия теоремы о состоятельности:

$h_1(d) \leq h^*(d)$,

$h_2(d) \leq h^*(d)$.

Будем говорить, что функция $h_2(d)$ более информированная, чем $h_1(d)$, если $\forall d \neq dt\ h_2(d) > h_1(d)$. Пусть $h_2(d)$ — более информированная функция, чем $h_1(d)$. Тогда в случае применения F_2 будет раскрыто не более узлов, чем в случае применения F_1, то есть более информированная оценочная функция быстрее находит решение.

Как пример рассмотрим игру восемь, представленную в параграфах 2.1.4 и 3.1.3. В нашей терминологии $g(d)$ — глубина в дереве игры, $h(d)$ — оценочная функция числа шашек не на своем месте. Так выбранная функция $h(d)$ в игре 8, очевидно, более информированная, чем функция $h(d)$ при поиске в ширину ($h = 0$).

Теорема. *Если существует оптимальный путь, и функция F_2 более информированная, чем функция F_1, то к моменту окончания поиска всякий узел, раскрытый по F_2, раскрыт и по F_1.*

Доказательство. Индукция по k — глубине узла d в дереве поиска по F_2 (рис. 3.3).

База индукции. Глубина равна нулю (d_0). В этом случае либо оба алгоритма раскрывают узел d_0, либо оба не раскрывают (при $d_0 = d_t$).

Индукционный переход. Пусть d — узел, выбранный для раскрытия по F_2 на шаге k. Пусть S_1 и S_2 множества раскрытых до d узлов по F_1 и F_2 соответственно. По индукционному предположению S_1 содержит узлов не меньше, чем S_2, значит в S_1 возможно больше узлов, из которых можно попасть в d, чем в S_2 (рис. 3.3). Поэтому $g_1(d) \leq g_2(d)$. Далее от противного. Пусть алгоритм закончен и d не раскрыт по F_1. По следствию 1 к теореме о состоятельности $F_1(d) \geq M$. Но по F_2 узел d раскрыт, значит $F_2(d) \leq M$. Имеем $g_1(d) + h_1(d) \geq M \geq g_2(d) + h_2(d) \geq g_1(d) + h_2(d)$, откуда $h_1(d) \geq h_2(d)$, что противоречит условию. **Ч.т.д.**

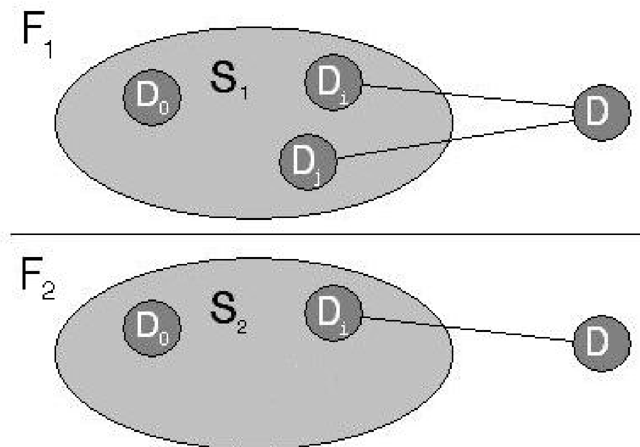

Рис. 3.3. К доказательству теоремы о сравнении оценочных функций

Рассмотрим еще раз пример с игрой в 8 в параграфе 3.1.3. Можно заметить, что предложенная эвристическая функция очень хороша: она не только удовлетворяет условию состоятельности, как показано в предыдущем параграфе, но она и «предельно информированная». Действительно, предложенную функцию h — число шашек не на своем месте — невозможно существенно улучшить, то есть, невозможно предложить эвристику h', такую, что $\forall d\, h'(d) > h(d)$. Действительно, из рис. 3.1 видно, что бывают такие узлы d, в которых $h(d) = h^*(d)$, а значит гипотетическая мажоранта h' нарушает условие состоятельности.

3.2.3. Монотонное ограничение

Говорят, что функция h удовлетворяет *монотонному ограничению* (или *неравенству треугольника*), если

$$\forall d\ h(d) \le h(r.f(d)) + r.w\ \&\ h(d_t) = 0,$$

где r — правило для перехода из узла d в узел $r.f(d)$.

Другими словами, эвристическая функция локально согласована с весом дуг (правил).

Теорема. *Если выполнено условие состоятельности, оценочная функция удовлетворяет монотонному ограничению и существует оптимальный путь, то алгоритм* A* *всегда выбирает для раскрытия вершину, к которой уже найден оптимальный путь, т. е.* $g(d) = g^*(d)$.

Доказательство. Индукция по длине оптимального пути.

База индукции: $g(d_0) = g^*(d_0) = 0$.

Индукционный переход. Пусть выбрана для раскрытия вершина $d \ne d_0$ и пусть $A = d_0, d_1, \ldots, d_i, \ldots, d$ — оптимальный путь. Имеем

$$\forall i\ g^*(d_i) + h(d_\mathrm{i}) \le g^*(d_\mathrm{i}) + w(d_i, d_{i+1}) + h(d_{i+1})$$

по монотонному ограничению. Далее

$$d_i, d_{i+1} \in A \to g^*(d_i) + w(d_i, d_{i+1}) = g^*(d_{i+1}).$$

Имеем

$$\forall i\ g^*(d_i) + h(d_i) \le g^*(d_{i+1}) + h(d_{i+1}),$$

и по транзитивности

$$\forall i\ g^*(d_i) + h(d_i) \le g^*(d) + h(d).$$

Пусть теперь d_k — первая открытая вершина на пути A, т. е. $d_k \in A,\ d_k \in O\ \&\ \forall j < k\ d_j \in C$. Такой узел существует, так как $d_0 \in C$, $d \in O$. Все вершины на пути A левее d_k, т. е. все закрытые вершины, предшествующие d_k на пути A, уже раскрыты, значит $g(d_k) = g^*(d_k)$.

Так как алгоритм A* при раскрытии очередного узла из двух узлов d и d_k выбирает узел d, то $F(d) \le F(d_k)$. Итак, имеем:

$$g(d) + h(d) = F(d) \le F(d_k) = g(d_k) + h(d_k) = g^*(d_k) + h(d_k) \le g^*(d) + h(d).$$

Следовательно, $g(d) \le g^*(d)$. Но $\forall d \ \ g(d) \ge g^*(d)$ и значит $g(d) = g^*(d)$. **Ч.т.д.**

Следствие. *Если оценочная функция удовлетворяет монотонному ограничению, то последовательность значений оценочной функции для выбираемых узлов не убывает.*

Доказательство. Пусть раскрыты узлы d_1, d_2 (подряд). Если в момент раскрытия узла d_1 уже $d_2 \in O$, то по построению A* имеем $F(d_1) \le F(d_2)$. Иначе d_2 должен попадать в O при раскрытии d_1. Имеем:

$$F(d_2) = g(d_2) + h(d_2) = g^*(d_2) + h(d_2) = g^*(d_1) + w(d_1, d_2) + h(d_2) =$$
$$= g(d_1) + w(d_1,d_2) + h(d_2) \ge g(d_1) + h(d_1) = F(d_1).$$

Ч.т.д.

3.2.4. Область применимости алгоритма A*

При всех своих достоинствах и заслугах алгоритм A* имеет целый ряд имманентно присущих ему особенностей, которые при определенных условиях могут проявлять себя как недостатки.

1. **В худшем случае алгоритм A* имеет экспоненциальные оценки сложности по времени и по памяти**. Действительно, пусть в нашем распоряжении нет информированной оценочной функции, удовлетворяющей условию состоятельности. В таком случае придется применять поиск в ширину, $h = 0$. Допустим (худший случай!) что в любом состоянии все правила применимы, и пусть этих правил n. Пусть также количество дуг[56] на оптимальном пути от d_0 к d_t равно s. Тогда, как нетрудно видеть, общее количество построенных узлов графа поиска составит не менее $1 + n + n^2 + \ldots + n^{s-1} = (n^{s-1} - 1)/(n - 1)$.

---

[56] Речь идет именно о количестве дуг, а не о длине пути, т. е. не о сумме весов дуг.

2. **Эффективность алгоритма критическим образом зависит от эвристической функции *F***. Действительно, выше мы отметили, что, при отсутствии эвристики, алгоритм в худшем случае совершает количество шагов, экспоненциально зависящее от числа дуг оптимального пути. В то же время, если в нашем распоряжении есть предельно информированная оценочная функция, такая что $\forall d\ (h(d) = h^*(d))$, и вдобавок выполнено монотонное ограничение, то алгоритм А* (в худшем случае!) построит $ns + 1$ узлов, поскольку, по теореме предыдущего параграфа, в этом случае раскрываться будут подряд узлы на оптимальном пути. Таким образом, в случае идеальной[57] эвристики алгоритм А* может иметь трудоемкость, линейную по числу дуг оптимального пути.

3. **Алгоритм А* не учитывает структуру базы фактов и использует ее всегда целиком**. Это чрезвычайно расточительно по памяти и для очень многих практических задач оказывается неприемлемым. Например, при решении задачи о движении робота в среде с препятствиями база фактов очень невелика — координаты текущего положения робота, и алгоритм А* показывает неплохие результаты. Но если А* применить в экспертной системе над большой реляционной базой данных, то формально говоря, каждому узлу в графе соответствует полный слепок (дамп) базы данных, причем соседние отличаются, может быть, только одной записью.

4. **Алгоритм А* не учитывает структуру условий в правилах и повторно вычисляет условия, которые, возможно, уже проверялись на предыдущих шагах**. Это весьма расточительно по времени. В рассмотренном выше примере, при проверке положения робота, вычисления, выполненные для предыдущего положения, нельзя повторно использовать, поскольку координаты изменились. Но при проверке выполнения некоторого сложного условия для большой

---

[57] Строго говоря, если $h = h^*$, то функцию h уже нельзя считать эвристической, поскольку она совпадает с математически точной функцией h^*.

реляционной базы данных, 99 % работы уже было выполнено при проверке этого условия в предыдущем состоянии, поскольку большая часть данных не изменилась.

5. **Алгоритм A* непригоден для случая динамического поиска**. Действительно, если продукции, или цель, или веса правил могут динамически меняться в процессе поиска[58], то все рассуждения предыдущих трех параграфов утрачивают силу. Состоятельность алгоритма A* основана на том, что выбор дальнейшего пути из некоторого узла не зависит от того, как мы попали в этот узел. В случае динамического поиска это не так: даже длина пути от текущего узла до целевого узла может быть различной в зависимости от того, как мы попали в текущий узел.

Сказанное в этом параграфе отнюдь не означает, что алгоритм A* никуда не годен и его не следует использовать. Напротив, алгоритм A* — это лучшее известное решение, если нет никакой дополнительной конкретизирующей информации о свойствах задачи. Если же дополнительная информация есть, то A* можно и нужно модифицировать с учетом конкретной задачи. Одна из самых элегантных модификаций обсуждается в следующем разделе.

3.3. ПОИСК НА ГРАФАХ И/ИЛИ

При рассмотрении разложимых систем продукции было введено понятие графов И/ИЛИ. В таком графе вершины — это состояния базы данных или их части, которые получаются двумя способами: либо применением правила, либо разложением базы данных на составляющие. Это именно граф, а необязательно только дерево поиска. Одна и та же вершина d может появиться и в результате разложения d_1 и применением правила к d_2. Графы И/ИЛИ находят

[58] Конечно, при игре в 8 правила не могут меняться при движении шашек. Но при управлении войсками противник вполне может изменить свою тактику и стратегию по ходу боевых действий, например, введя в сражение новый вид оружия.

широкое применении в ИИ. Алгоритмы поиска, подобные А*, специализируются и уточняются для применения в случае графов И/ИЛИ. Кроме того, для отдельных частных случаев гиперграфов известны еще более эффективные алгоритмы.

3.3.1. Граф И/ИЛИ

Уточним определения.

Граф И/ИЛИ *— это гиперграф, то есть пара, состоящая из множества узлов* $\{d\}$ *и множества k-дуг, причем k-дуга — это пара* $d \to (d_1,\ldots,d_k)$, *где* d *— начальный узел k-дуги, а* $\{d_1,\ldots,d_k\}$ *— множество концевых узлов.*

С точки зрения этого определения, обычный ориентированный граф — это гиперграф, в котором каждая дуга — это 1-дуга. Так же как и в обычном орграфе, в гиперграфе можно определить понятие пути (гиперпути).

Гиперпуть *в гиперграфе — это последовательность множеств узлов,* $<S_1, \ldots, S_n>$, *причем для каждого узла* $d \in S_i$, $1 \le i \le n-1$, *существует ровно одна k-дуга* $d \to (d_1,\ldots,d_k)$, *такая, что* $\forall 1 \le j \le k$ $(d_j \in S_{i+1})$.

Определение гиперпути является прямым обобщением определения пути, но гиперпуть в гиперграфе будет выглядеть как граф, а не как одна чередующаяся последовательность узлов и дуг. Из каждой вершины выбирается ровно одна *k*-дуга. Из каждой полученной вершины снова выбирается ровно одна *k*-дуга т. д.

Среди всевозможных гиперпутей нам интересны те, которые начинаются в исходном состоянии базы данных и заканчиваются в терминальных состояниях.

Граф решения $G(d_0, T)$ *— это гиперпуть из* d_0 *во множество терминальных узлов* T.

Можно дать эквивалентное рекурсивное определение графа решения $G(d_0, T)$:

1. Если $d_0 \in T$, то $G = d_0$.

2. Если $d_0 \notin T$, и из d_0 исходит k-дуга $d_0 \rightarrow (d_1,\ldots, d_k)$, причем существуют графы решения $G_1, G_2, \ldots, G_k$ для узлов $d_1, d_2, \ldots, d_k$, то

$$G = d_0 \rightarrow (d_1,\ldots, d_k) \cup G_1 \cup G_2 \cup \ldots \cup G_k .$$

Стоимость (вес) графа решения (то есть длина пути в гиперграфе) определяется с помощью индуктивного определения:

$$w(G(d,T)) = w(d \rightarrow (d_1,\ldots, d_k)) + \Sigma\, w(G(d_i, T)).$$

Таким образом, одна и та же k-дуга при подсчете длины может учитываться много раз.

Рассмотрим пример (рис. 3.4). Положим, что узлы 5 и 6 терминальные, а узел 0 начальный. Можем утверждать, что здесь имеются два графа решения (на рисунке они обведены).

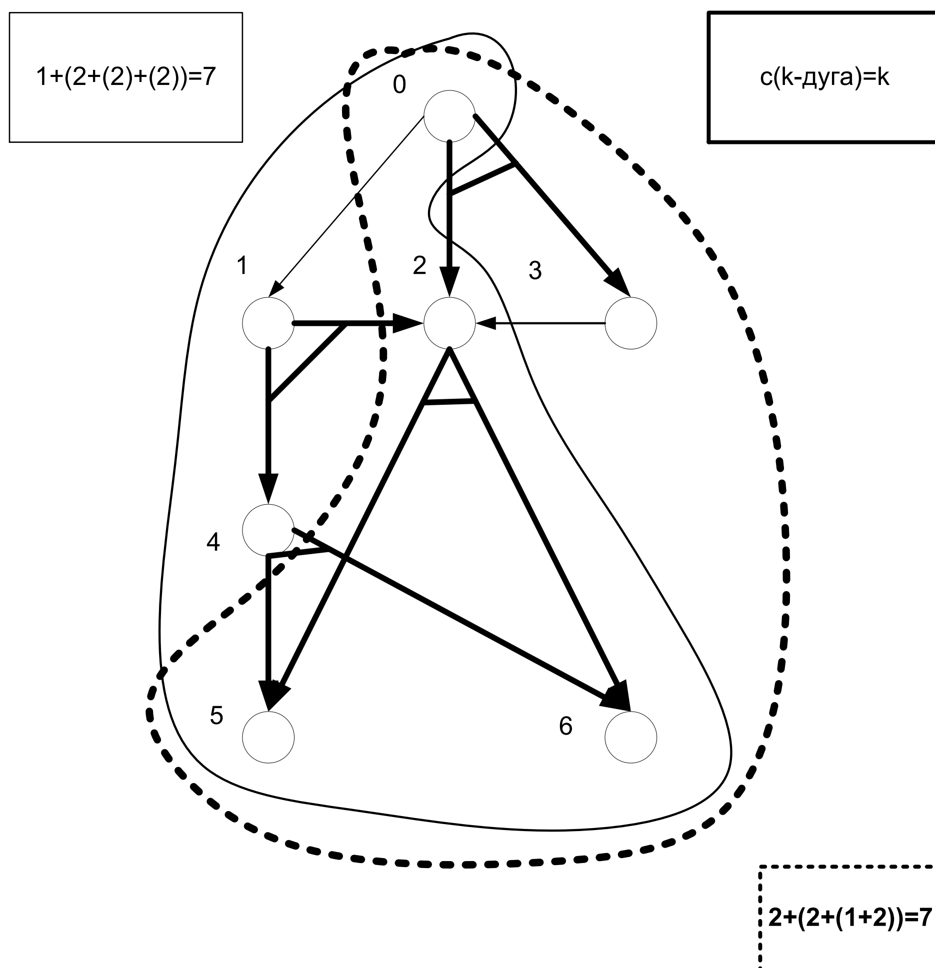

Рис. 3.4. Гиперграфы решения и k-дуги

Если считать, что длина k-дуги равна k, то

- длина первого графа решения = 1 + (2 + (2) + (2)) = 7,
- длина второго графа решения = 2 + (2 + (1 + 2)) = 7,

так как согласно индуктивному определению некоторые k-дуги могут учитываться несколько раз.

3.3.2. Алгоритм поиска на графе И/ИЛИ

Для поиска путей в гиперграфе существуют алгоритмы, обобщающие обходы в глубину и в ширину на обычных графах. Принципиальным отличием обобщающих алгоритмов является то, что при достижении отдельного терминального узла процедура поиска не останавливается. Нам нужно не просто достичь отдельного терминального узла, а построить граф решения, все ветви которого заканчиваются в терминальных узлах.

Вообще говоря, алгоритмы поиска на графе И/ИЛИ могут работать без дополнительной информации. Однако при наличии оценочной функции могут быть использованы алгоритмы эвристического поиска.

Алгоритм эвристического поиска на графе И/ИЛИ заключается в следующем.

Пусть граф ациклический, имеется оценочная функция F, и h удовлетворяет условию монотонности (монотонному ограничению):

$$h(d) \le w(d,(d_1,\dots, d_k)) + \Sigma\, h(d_i)$$

и условию состоятельности (то есть является нижней оценкой точного решения):

$$h(d) \le h^*(d).$$

Процедуры поиска на графах И/ИЛИ определяются следующим образом.

Алгоритм 3.3. Основная процедура поиска на графе И/ИЛИ.

```
proc GS и/или (d0, R, T)
     G := d0  {частичный граф решения}
```

```
    s(d0)  := d0 ∈ T {отметка узла}
    F(d0):= h(d0)
    while ¬s(d0) do
        d := Select (G)
        if D = nil then return (fail) end if
        Open (d, G)
    end while
end proc
```

Здесь `s(d): bool` — отметка узла, обозначающая, найден ли для этого узла гиперпуть, ведущий в терминальные вершины. Если `s(d)=true`, гиперпуть найден, то узел называется *разрешенным* (от английского слова solved).

Процедура `Select` на основе построенного графа поиска рассматривает частичный граф решения и выбирает для раскрытия на этом графе нераскрытый узел минимальной стоимости.

Алгоритм 3.4. Процедура выбора узла для раскрытия.

```
proc Select (G)
    O : = {d0}
    for d∈ O do
        for q = (d→d1,…,dk) ∈  Q(d) do
            if x(d→d1,…,dk) then {дуга отмечена}
                O:=O - {d} {удаляем начальный}
                O:=O + ({d1,…,dk}\T)  {концевые}
            end if
        end for
    end for
    m := ∝
    d' := nil
    for d ∈ O do
        if F(d)<m then
            d' := d
            m := F(d)
        end if
    end for
```

```
    return (d')
end proc
```

Здесь `O` — концевые вершины частичного графа решения, `Q(d)` — множество дуг узла `d`, `x(q)` — отметка дуги `q`.

В этой процедуре фактически каждый раз заново прогоняется фронт решения от d_0 к концевым вершинам по отмеченным *k*-дугам.

Процедура `Open` значительно сложнее. Она раскрывает выбранную вершину, пересчитывает оценки вершин, отмечает *k*-дуги и отмечает разрешенные вершины, то есть вершины, для которых найден гиперпуть в терминальные вершины. Эти изменения могут затронуть весь граф.

Алгоритм 3.5. Процедура Open.

```
proc Open (d,G){раскрытие вершины}
    for r∈ R do
        if r.p(d) then
            G := G + (d→ d1,…,dk)
            for i from 1 to k do
                if di∉ G then F(di):=h(di) end if
                s(di) := di ∈ T
            end for
        end if
    end for
    {Пересчет оценок и отметок}
    C := {d}
    while C ≠ ∅ do
        {выбор вершин для пересчета}
        for d'∈ C do
            if P(d') ∩ C = ∅ then d:=d' end if
        end for
        C := C - {d}
        m := ∝
        for q=(d→ d1,…,dk) ∈ Q(d) do
            w':=w(q) + Σ F(di)
```

```
            if w′ < m then
                x(q(d)) := false
                x(q) := true
                s(d):=&s(di)
                m := w′
            end if
        end for
        if s(d) ∨ F(d)≠m then
            F(d):=m
            C := C + U(d)
        end if
    end while
end proc
```

Здесь используются следующие обозначения.

Q(d) — множество исходящих из d дуг (в G),

P(d) — множество потомков d в G,

q(d) — отмеченная дуга из d,

x(q) — отметка дуги,

U(d) — множество предков d в G вдоль отмеченных дуг,

s(d) — отметка узла, означающая что из него построен гиперпуть до цели.

Повторим еще раз основные выводы по этим трем процедурам. Процедура GS и/или попеременно вызывает процедуры Select и Open пока либо окажется так, что никакой узел не найден для раскрытия (результат **fail**), либо исходный узел d_0 окажется разрешенным. Процедура Select строит фронт текущего частичного графа решения по отмеченным дугам и выбирает узел с наименьшей оценкой в этом фронте. Процедура Open сначала раскрывает выбранную вершину, а затем пересчитывает оценки узлов, отмечает дуги и вершины. При этом данная процедура может переоценить весь граф. В этих процедурах мы считаем, что понятие правила обобщает как применение правила, так и расщепление базы данных с помощью *k*-дуги.

3.3.3. Пример применения процедуры поиска на графе И/ИЛИ

На рис. 3.5 - 3.9 приведен пример работы алгоритма поиска на графе И/ИЛИ применительно к графу, представленному на рис. 3.4.

На этих рисунках используются следующие обозначения. Состояния базы, то есть узлы графа И/ИЛИ, обозначаются кружками. Если кружок закрашен, то это означает, что для него получено условие `s(d) = 1`, то есть, найден граф решения из данного узла в терминальные узлы (которыми в данном примере являются узлы 5 и 6). 1-дуги обозначаются обычными стрелками, *k*-дуги (в этом примере $k = 2$) обозначаются более плотными стрелками и имеют небольшую черточку, соединяющую ветви *k*-дуги. Число рядом с узлом является текущей оценкой этого узла. При этом считается, что функция h для терминальных узлов имеет значение 0, а для всех остальных — 1. Такая функция удовлетворяет условию состоятельности, очевидно. Длина *k*-дуги равна *k*. Отмеченная дуга, исходящая из данного узла, отмечена жирной полоской. Следующая серия рисунков образует пояснение к процедурам поиска на графе И/ИЛИ.

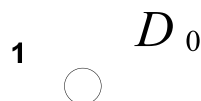

Рис. 3.5. Шаг 1. Начальное состояние

В начальном состоянии граф решения состоит из единственного узла d_0, этот же узел образует фронт решения, именно его выбирает функция `Select`, потому что больше выбирать некого, а процедура `Open` раскрывает этот узел.

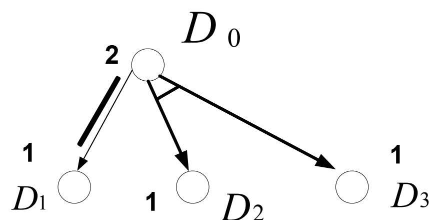

Рис. 3.6. Шаг 2. Первое раскрытие узла

После первого шага при пересчете оценок в узел d_0 возвращаются:

- слева = 1 (оценка узла d_1) + 1 (стоимость дуги $d_0 \to d_1$) = 2;

- справа = 1 (оценка узла d_2) + 1 (оценка узла d_3) + 2 (стоимость дуги $d_0 \to (d_2,d_3)$) = 4.

Оценка слева предпочтительнее, поэтому отмечается дуга $d_0 \to d_1$, и следующим будет открываться узел d_1.

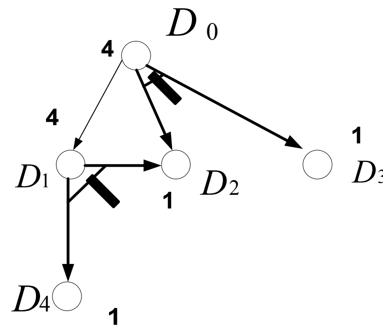

Рис. 3.7. Шаг 3. Второе раскрытие узла

После второго шага при пересчете оценок в узел d_0 возвращаются:

- слева = 1 (оценка узла d_4) + 1 (оценка узла d_2) + 2 (стоимость дуги $d_1 \to (d_4,d_2)$) + 1 (стоимость дуги $d_0 \to d_1$) = 5 — оценка возрастает;

- справа = 1 (оценка узла d_2) + 1 (оценка узла d_3) + 2 (стоимость дуги $d_0 \to (d_2, d_3)$) = 4 — оценка не меняется.

Теперь оценка справа предпочтительнее, поэтому отмечается дуга $d_0 \to (d_2, d_3)$, и следующим будет открываться узел d_2.

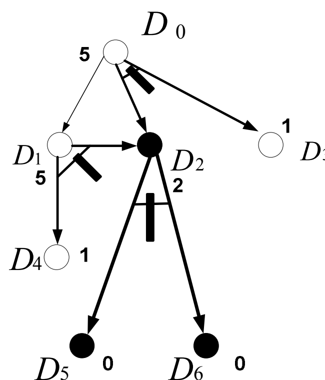

Рис. 3.8. Шаг 4. Третье раскрытие узла

После третьего шага мы попадаем в терминальные узлы d_5, d_6. Поскольку это оба узла, в которые ведет 2-дуга $d_2 \to (d_5, d_6)$, узел d_2 также отмечается как разрешенный. При этом его оценка равна 2 (стоимость исходящей 2-дуги). При пересчете оценок в узел d_0 возвращаются:

- слева = 1 (оценка узла d_4) + 1 (оценка узла d_2) + 2 (стоимость дуги $d_1 \to (d_4, d_2)$) + 1 (стоимость дуги $d_0 \to d_1$) = 5 – оценка не меняется;

- справа = 2 (оценка узла d_2) + 1 (оценка узла d_3) + 2 (стоимость дуги $d_0 \to (d_2, d_3)$) = 5 — оценка не меняется, а потому отметка дуги $d_0 \to (d_2, d_3)$ сохраняется.

Следующим будет открываться узел d_3.

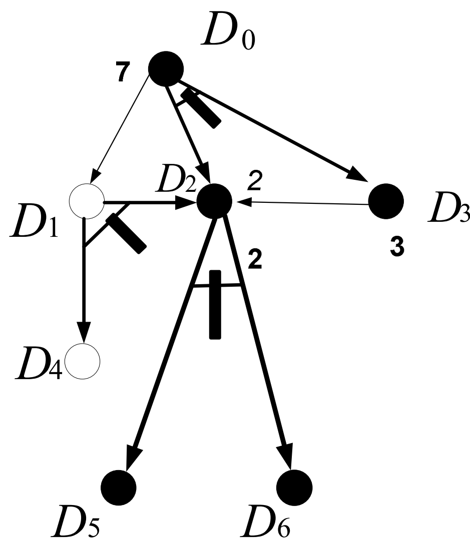

Рис. 3.9. Шаг 5. Построение графа решения

На последнем шаге открывается узел d_3. Поскольку дуга $d_3 \to d_2$ ведет в уже разрешенный узел, узел d_3, а за ним и узел d_0 получают отметку `s = 1`. При пересчете оценок в узел d_0 возвращается: 2 (оценка узла d_4) + 3 (оценка узла d_3) + 2 (стоимость дуги $d_0 \to (d_2, d_3)$) = 7. Работа алгоритма завершена.

3.4. ПОИСК НА ИГРОВЫХ ДЕРЕВЬЯХ

Идея поиска на графах И/ИЛИ особенно широко применяется при программировании интеллектуальных игр. В этом случае обычно рассматривается особый частный случай графа И/ИЛИ, который называется *игровым деревом*.

3.4.1. Игровые деревья

Допустим, имеется антагонистическая (есть соперник) дискретная (число позиций не более чем счетное) игра с полной информацией (все правила заранее известны и позиция открыта). В ней принимают участие два игрока **Max** и **Min**, которые ходят по очереди, причем **Max** ходит первым.

Попробуем составить выигрышную стратегию для игрока **Max**. Для этого построим граф поиска, узлами которого будут допустимые позиции игры. В этом графе дуга ведет из узла, соответствующего позиции A, в узел, соответствующий позиции B, если B получается из A применением некоторого правила игры к позиции A. Корню дерева соответствует начальная позиция игры, а листьям — позиции, в которых однозначно определен исход игры (т. е. определен победитель или зафиксирована ничья). Тогда задача определения выигрышной стратегии для игрока **Max** сводится к поиску пути в дереве игры от корня к листу, соответствующему позиции, победной для **Max**. Причем из позиции для некоторого хода **Max** достаточно найти хотя бы одну разрешимую вершину (потому, что мы составляем стратегию для игрока **Max** и можем выбрать тот ход, который хотим), тогда как для хода **Min** все потомки должны быть разрешимыми (потому, что мы не можем повлиять на выбор хода игроком **Min** и должны принимать во внимание все возможные ходы). Таким образом, дерево игры является графом И/ИЛИ с 1-дугами из узлов для первого игрока **Max** (И) и *k*-дугами из узлов для второго игрока **Min** (ИЛИ).

Правилами игры задана оценка F^* всех конечных позиций: $F^* = +\infty$ если победил **Max**, $F^* = -\infty$ если победил **Min** и $F^* = 0$ если игра закончилась вничью.

Обычно игру представляют не просто графом И/ИЛИ, а именно гипердеревом, так что в каждую позицию, кроме начальной, входит ровно одна дуга. При этом, очевидно, может происходить некоторая потеря памяти, потому что совпадающие позиции, которые получились в результате различных последовательностей ходов, будут храниться в нескольких экземплярах. Однако практические наблюдения показывают, что в наиболее распространенных интеллектуальных играх такие ситуации встречаются сравнительно редко. С другой стороны, для игровых деревьев предложено несколько весьма эффективных алгоритмов поиска решения (см. следующие параграфы), которые не применимы к графам общего вида. Поэтому при программировании интеллектуальных игр обычно жертвуют памятью ради повышения быстродействия алгоритмов.

Рассмотрим в качестве примера упрощенную игру Ним[59]. В кучке лежит пять монет. За ход разрешается взять одну, две или три монеты. Пропускать ход нельзя. Проигрывает тот, кто берет последнюю монету. На рис. 3.10 приведено полное дерево игры. Под терминальными позициями подписано, кто выиграл.

---

[59] Эта игра попала в Европу в XVI веке из Китая. Имя «Ним» было дано игре американским математиком Чарльзом Бутоном (Chalres Bouton), описавшим в 1901 году выигрышную стратегию игры. В классическом варианте игры есть три кучки монет, за один ход может быть взято любое количество монет (большее нуля) из одной кучки, и выигравшим считается игрок, взявший последний предмет.

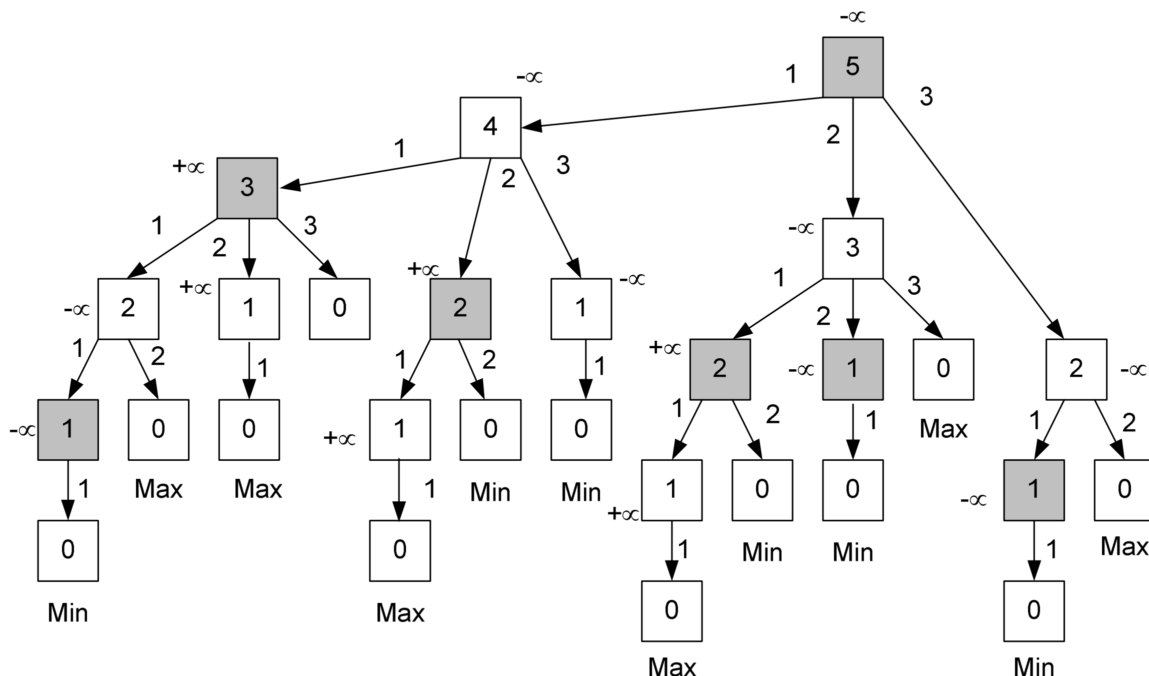

Рис. 3.10. Дерево игры Ним

3.4.2. Минимакс

Один из алгоритмов поиска решения на игровых деревьях известен как метод Минимакс (MiniMax), предложенный еще в 1945 году О. Моргенштерном (O. Morgenstern) и Дж. Фон Нейманом (J. Von Neumann). Дж. Фон Нейман наиболее известен как праотец современной архитектуры компьютеров (архитектура фон Неймана), а также как создатель теории игр и концепции клеточных автоматов.

Если бы у нас было полное дерево игры, мы бы начали снизу, с терминальных вершин, оценка которых задана правилами игры (функция F^* для терминальных позиций), и все было бы достаточно просто — для хода игрока **Max** мы выбираем дочернюю вершину с максимальным значением оценочной функции, и переносим эту оценку вверх, а для хода **Min** — с минимальным. Если в результате работы алгоритма начальная вершина имела бы оценку +∝, то победил **Max**, если –∝, то **Min**. Таким образом, игра была бы решена.

Например, на рис. 3.10 рядом с узлами написаны статические оценки, полученные указанным образом. На рисунке видно, что в

данной простой игре **Min** всегда может выиграть, если будет правильно играть.

Но полное дерево игры обычно слишком велико. К примеру, для шашек это 10^{40} позиций, а для шахмат — 10^{120} (параграф 1.1.8). *Для практически интересных игр полное дерево игры построить физически невозможно*. Но можно эту идею (начать снизу) применить и к частичному дереву. Для этого нужно построить частичное дерево, оценить узлы на нижнем ярусе при помощи оценочной функции, и вернуться по дереву назад, т. е. вверх.

Метод Минимакс заключается в следующем. Пусть есть некоторая статическая оценочная функция позиции F, причем если позиция считается выгодной для **Max**, то $F > 0$, если позиция считается выгодной для **Min**, то $F < 0$. Если бы **Max** выбирал среди концевых вершин, то он выбирал бы позицию с наибольшей оценкой, а если выбирал бы **Min**, то — с наименьшей оценкой. Также и в методе Минимакс. В построенном частичном дереве игры (если полное дерево построить невозможно) листовые позиции оцениваются придуманной оценочной функцией (поскольку правила игры не предусматривают оценку промежуточных позиций), а далее вверх по дереву распространяются оценки в точности так, как это делается при статической оценке полного дерева. Если оценочная функция придумана удачно, а дерево построено достаточно глубоко, то метод Минимакса часто дает действительно оптимальное решение, или решение, близкое к оптимальному.

Рассмотрим в качестве примера игру в крестики-нолики на доске 3×3. Игроки по очереди ставят в пустые клетки крестики и нолики. Выигрывает тот, кто первым поставит три своих значка подряд: в одной горизонтали, вертикали или диагонали.

Применение метода Минимакс с глубиной частичного дерева игры равной 2 приведено на рис. 3.11 («крестики» обозначены цифрами 1, а «нолики» — цифрой 0). Число после знака «=» в позиции — это ее оценка.

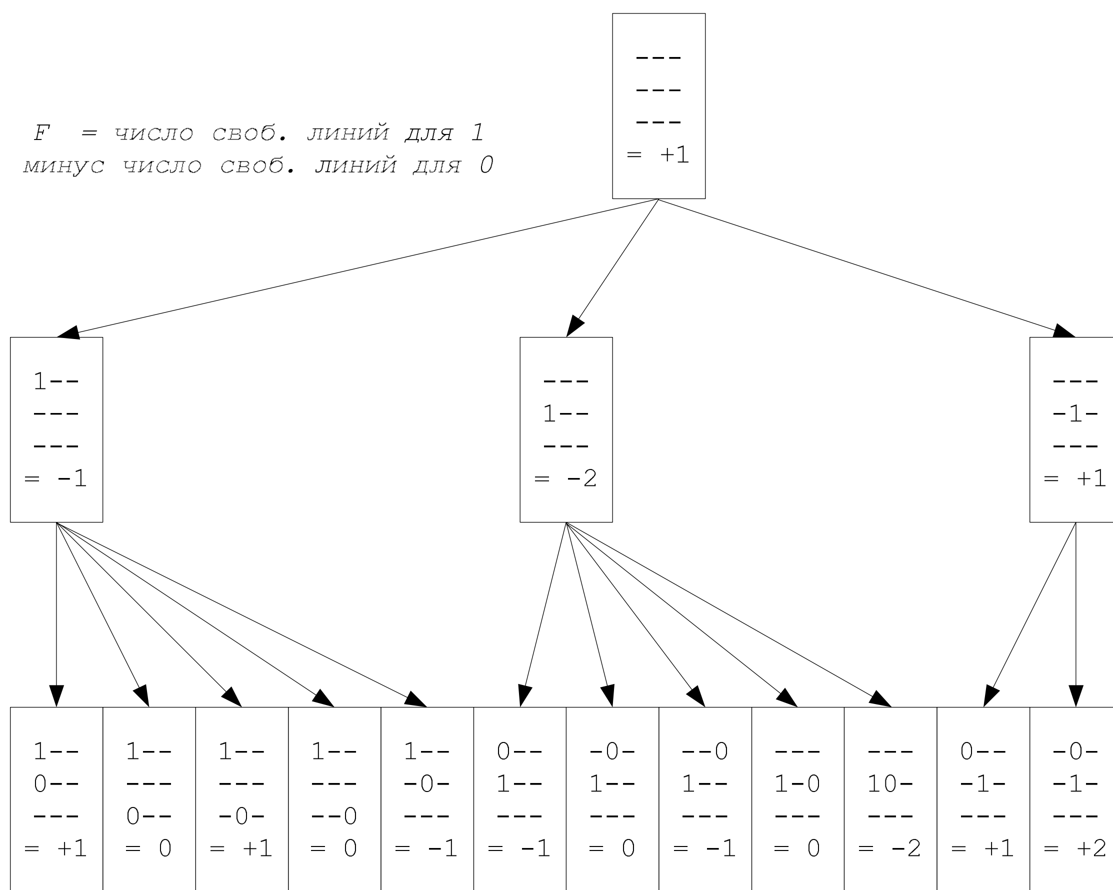

Рис. 3.11. Часть дерева игры крестики-нолики

Мы видим, что наилучшим ходом для **Max** при оценке по методу Минимакс с глубиной 2 оказывается ход в центральную клетку. Как нетрудно сообразить, это действительно выигрышная стратегия для **Max**.

Однако далеко не всегда метод Минимакс может выбрать действительно лучший или даже хороший ход, все зависит от того, насколько удачно составлена оценочная функция. Рассмотрим в качестве примера небольшое усложнение игры Ним из предыдущего параграфа. Пусть допустимые ходы и условие окончания остаются теми же, а исходное число монет не фиксировано и равно n. Нам нужно придумать оценочную функцию. Обычно это делается на основе наблюдений. Рассмотрим рис. 3.10 в качестве базы наблюдений. Мы видим, что при $n = 1$ **Max** проигрывает, при $n = 2, 3, 4$ **Max** выигрывает, а при $n = 5$ снова проигрывает. Значит, при $n = 1$ и

$n = 5$ оценочная функция должна быть отрицательна, а при $n = 2, 3, 4$ — положительна. Возникает искушение подобрать по возможности простую формулу, согласующуюся с наблюдательными данными, и объявить ее «законом природы», в данном случае — оценочной функцией[60]. Искушение становится просто непреодолимым, как только из глубины памяти всплывают остаточные знания курса школьной алгебры. Действительно, функция имеет, по меньшей мере, два корня, один между 1 и 2, а второй между 4 и 5. Как учили в школе, рассмотрим произведение $(n - 1{,}5)(n - 4{,}5)$ и после округления функцию $F(n) = -n^2 + 6n - 7$, которая демонстрирует желаемое поведение, полностью согласованное с имеющимися наблюдениями: $F(1) = -2$, $F(2) = +1$, $F(3) = +2$, $F(4) = +1$, $F(5) = -2$. Применим эту оценочную функцию в методе Минимакс для игры Ним при $n = 7$ с раскрытием дерева на глубину 2 (рис. 3.12). На этом рисунке указаны исходные оценки, заданные выбранной функцией, вычисленные оценки по методу Минимакс, и утолщены стрелки ходов, которые выбирает метод. Мы видим, что метод подсказывает правильный ход!

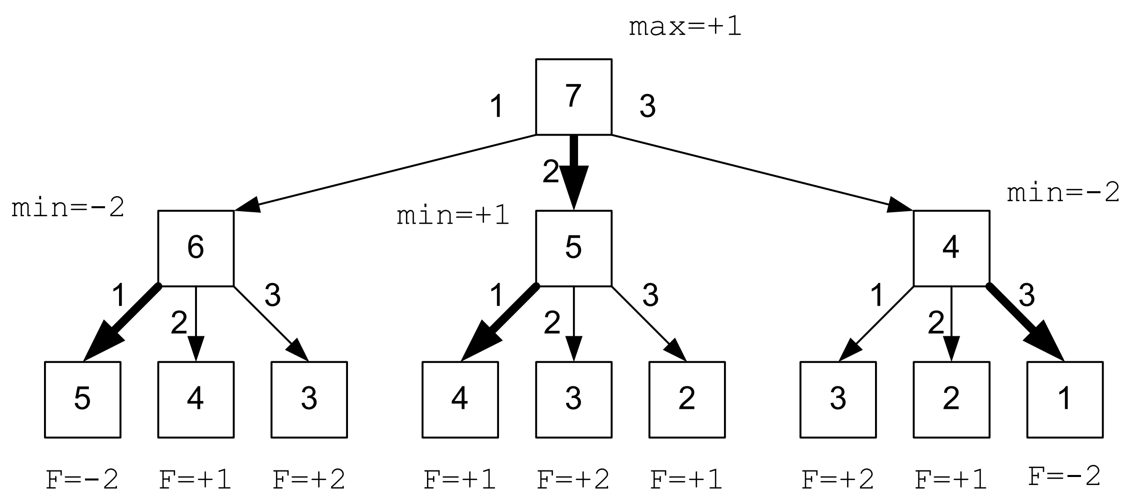

Рис. 3.12. Удачное применение метода Минимакс в игре Ним

[60] Читателю, который не потерял интереса к нашему примеру, мы настоятельно рекомендуем в этом месте прервать чтение и самостоятельно решить задачу. Мы даже подскажем ответ: **Max** проигрывает тогда и только тогда, когда n **mod** $4 = 1$.

Однако, если мы попытаемся экстраполировать оценочную функцию и выйти за пределы исходной наблюдательный базы, мы получим совсем негодный результат. На рис. 3.13 представлено дерево для случая $n = 10$ в тех же обозначениях, что и на рис. 3.12. Минимакс дает плохой совет, а тот ход, который на самом деле выигрывает, по методу Минимакс оценивается как худший!

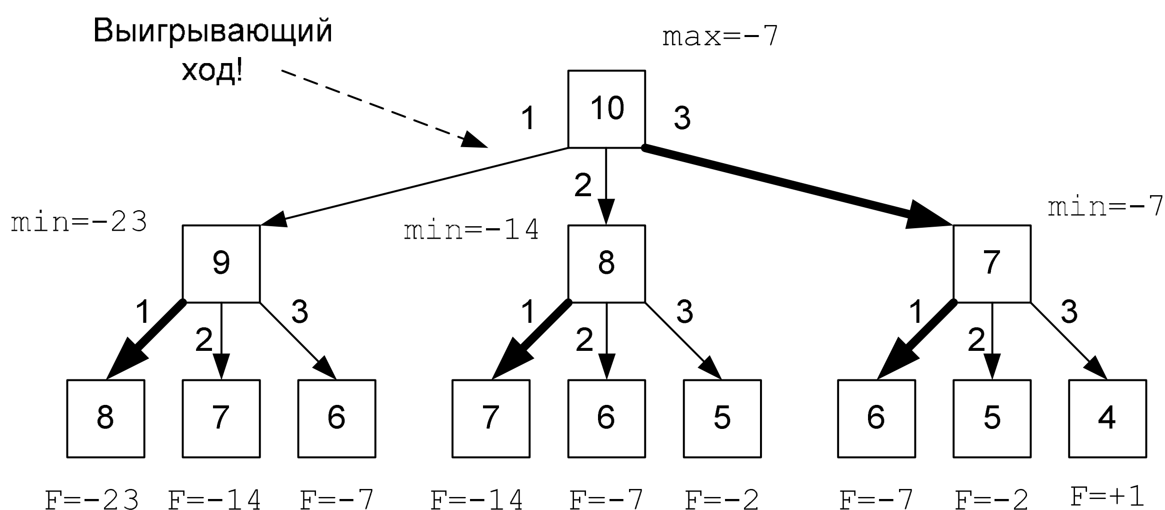

Рис. 3.13. Неудачное применение метода Минимакс в игре Ним

3.4.3. α-β отсечение

В Минимаксе порождение позиций и их оценка отделены друг от друга. Сначала происходит порождение, а только потом это порождение оценивается. Можно производить одновременную оценку прямо во время порождений, чтобы не порождать ничего лишнего. Такая процедура называется α-β *отсечение*.

Этот механизм, видимо, был настолько необходим и актуален в середине XX века, что его разработками занялись многие ученые по всему миру. Сейчас сложно сказать, кто является автором этой процедуры, судя по всему, она изобреталась неоднократно. Самуэль (A. Samuel), Ричардс (D. J. Richards), Харт (T. P. Hart) и другие независимо предлагали ранние версии этого алгоритма. Джон Маккарти (J. McCarthy) так же выдвигал подобные идеи на

Дармудском семинаре в 1956 году. Наш соотечественник Александр Львович Брудно независимо открыл алгоритм и опубликовал свои результаты в 1963 году.

Рассмотрим простой модельный пример, поясняющий основную идею α-β *отсечения*. Пусть имеется некоторая игра, поддерево поиска которой имеет вид как на рис. 3.14 (слева). Предположим, мы раскрыли и оценили только левое поддерево рассматриваемого узла (в нем есть единственная позиция, и ее оценка равна 7). Поскольку игрок **Max** всегда выбирает узел с наибольшим значением оценочной функции, то в независимости от нераскрытого поддерева справа оценка корневой позиции будет ≥ 7. Теперь рассмотрим выбор хода для игрока **Min** (рис. 3.14, справа). Пусть из двух возможных позиций раскрыта только одна (ее оценка 4). Так как игрок **Min** выбирает узел с наименьшим значением функции, то оценка выбранного хода будет ≤ 4.

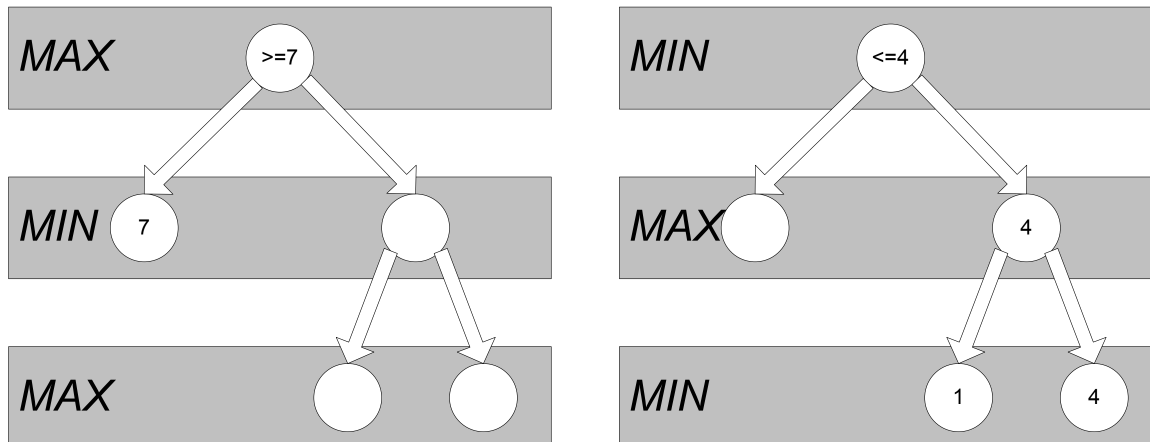

Рис. 3.14. Поддеревья некоторой игры

Продолжим эти рассуждения и рассмотрим рис. 3.15.

После рассмотрения правого поддерева узла, соответствующего ходу Min, известно, что его оценка не превзойдет 4. Однако при рассмотрении части левого поддерева оказывается, что оценка его корня будет ≥ 7. Это значит, что проводить дальнейшее раскрытие

узлов в левом поддереве не имеет смысла, так как у игрока **Min** уже есть заведомо лучший вариант. Поэтому происходит *отсечение* оставшейся части левого поддерева.

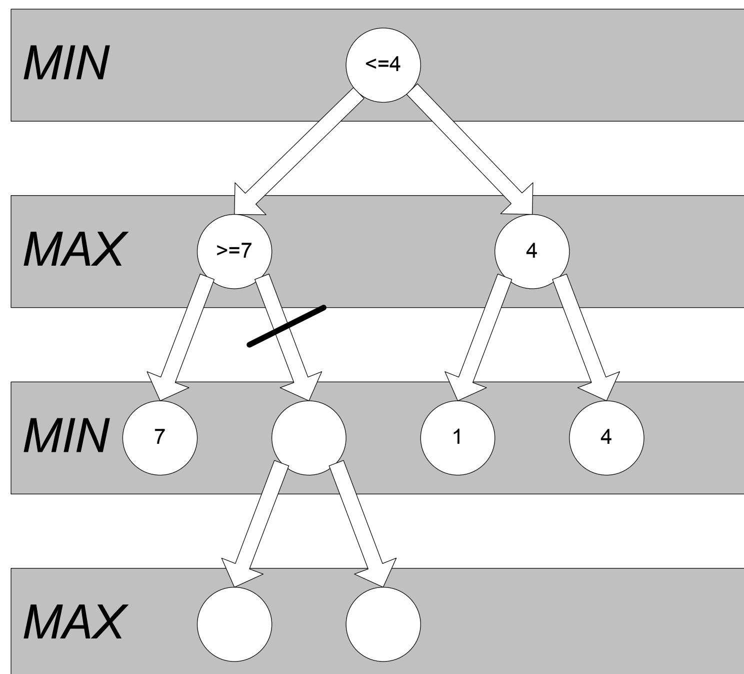

Рис. 3.15. Пример отсечения поддерева некоторой игры

Приведенный пример формализуется следующим образом.

Пусть α — граница снизу для значения оценки в данном узле, β — граница сверху для значения оценки в данном узле.

Заметим, что значение α для **Max** узла не могут уменьшаться, а значение β для **Min** не могут увеличиваться при любом раскрытии узлов. Следовательно, имеют место следующие правила.

1. Можно не искать из данного узла **Min**, если его верхняя оценка β меньше или равна нижней оценке α для любого его **Max** родителя.

2. Можно не искать из данного узла **Max**, если его нижняя оценка α больше или равна верхней оценке β для любого его **Min** родителя.

Обратите внимание, что формульная запись проверяемого неравенства для отсечения формально одна и та же: в первом случае $\beta \le \alpha$, во втором случае $\alpha \ge \beta$, но смысл эти формулы имеют различный.

В первом случае, для узла **Min**, мы берем только что полученную оценку β и сравниваем ее с оценками α для вышележащих узлов **Max**. Во втором случае, для узла **Max**, мы берем только что полученную оценку α и сравниваем ее с оценками β для вышележащих узлов **Min**. Но в обоих случаях выполнение неравенства является основанием для отсечения.

Пример для игры в крестики-нолики приведен на рис. 3.16.

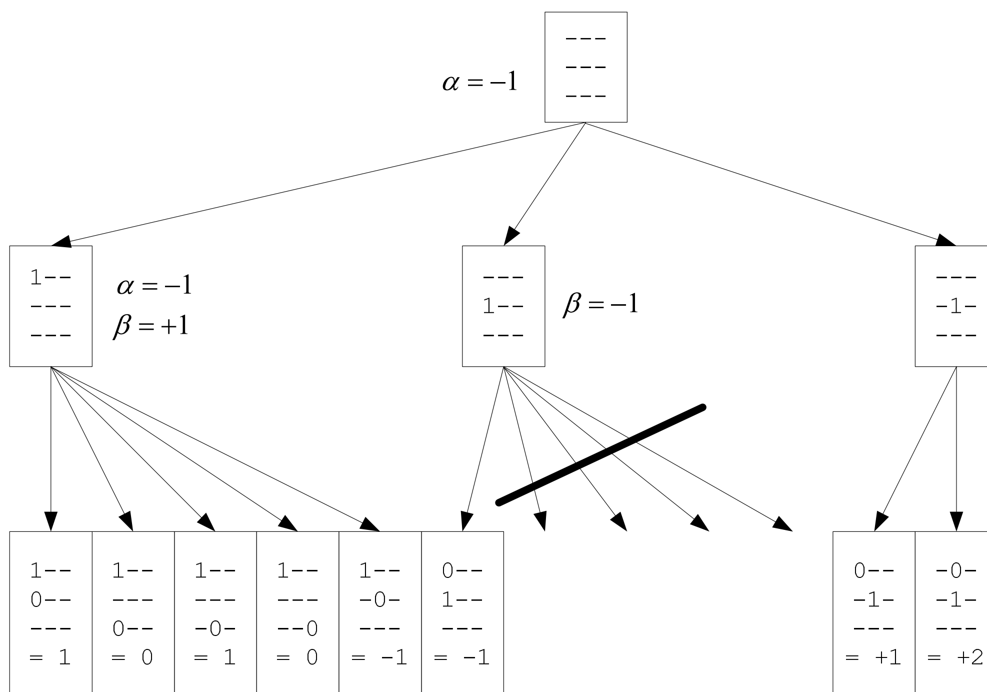

Рис. 3.16. α-β отсечение для игры в крестики-нолики

Порождение и оценивание производятся следующим образом. Сначала производится порождение трех узлов второго уровня. Потом производится порождение пяти узлов третьего уровня в левом поддереве и получается оценка –1 для узла второго уровня. При этом

можно для корневого узла первого уровня сразу определить нижнюю оценку $\alpha = -1$. Далее начинается порождение узлов третьего уровня в среднем поддереве. Первый же узел имеет оценку –1, что дает верхнюю оценку $\beta = -1$ и выполнение неравенства отсечения. Мы видим, что четыре узла на третьем уровне можно не порождать — их порождение не изменит оценки остальных узлов. В правом поддереве никакого отсечения не происходит — неравенство не выполняется.

3.4.4. Эффективность α-β -отсечения

Насколько эффективно α-β отсечение? Попробуем ответить на этот вопрос. В разобранном примере мы сэкономили 4 узла из 16, или 25 % работы. Является ли этот показатель предельным? Всегда ли он достигается?

Сначала ответим на второй вопрос, он более простой. Ответ: нет, не всегда. Действительно, пусть (худший случай!) дочерние узлы для узла **Min** порождаются в порядке убывания, а для узла **Max** в порядке возрастания. Например, если на рис. 3.16 узлы третьего уровня в среднем поддереве будут порождены в таком порядке: 0, 0, –1, –1, –2, то отсечены будут только два узла (12,5 % экономии), если оценочная функция будет неудачна и даст оценки 0, 0, 0, 0, 0, то вообще никакого отсечения не произойдет.

Теперь ответим на первый вопрос: что может дать α-β отсечение в наилучшем случае. Пусть глубина рассматриваемого дерева (без отсечений) равна h, а каждый узел имеет ровно d сыновей. Тогда внизу имеется ровно $N = d^h$ узлов. Пусть (наилучший случай!) вершины для узлов **Min** порождаются в порядке возрастания, а для **Max** — в порядке убывания значений оценочной функции.

Тогда можно показать, что при использовании α-β отсечения число порожденных узлов будет равняться:

$$N = O(d^{h/2}) = \begin{cases} 2d^{h/2}, \text{если } h \text{ четно} \\ d^{(h+1)/2} + d^{(h-1)/2} - 1, \text{если } h \text{ нечетно} \end{cases}$$

Поэтому ответ на наш вопрос таков: в идеальном случае α-β процедура позволяет удвоить глубину поиска по сравнению с Минимаксом, и в любом случае дает результат не хуже.

Таким образом, мы видим, что порядок порождения позиций и порядок рассмотрения ходов являются одним из ключевым факторов программирования интеллектуальных игр. К сожалению, (а может быть, и к счастью?) никаких общих методов ранжирования ходов не существует. Методы перебора во всех игровых программах примерно одинаковы и близки к изложенным (давно опубликованным и хорошо изученным), а вот эвристики порядка рассмотрения ходов для каждой игры свои (и составляют ноу-хау автора игровой программы).

Практика показывает, что сила игры программы прямо пропорциональна глубине поиска. Например, начинающий шахматист редко просматривает позицию больше чем на три-четыре полухода вперед. Т. е. глубина дерева поиска у новичка примерно 3. Шахматные мастера строят в уме дерево на дюжину полуходов, а если намечается форсированный вариант, то и больше. Гроссмейстеры и чемпионы видят еще дальше, на два десятка полуходов. Находясь в миттельшпиле, они уже могут предсказать эндшпиль.

Современные шахматные программы, используя резко возросшие возможности компьютеров, строят и оценивают огромные игровые деревья, включающие миллионы позиций. Каким же образом люди еще иногда не проигрывают компьютерам в шахматы?[61] Дело в том, что шахматист-человек строит в уме гораздо более узкое дерево. В сложной шахматной позиции сильный шахматист сразу видит 2-3 наилучших хода из двух десятков возможных ходов и только их

---

[61] Кстати, не проигрывают уже только гроссмейстеры и чемпионы. Средний любитель шахмат проиграет лучшей современной шахматной программе с вероятностью 100 %. Даже Каспаров уже проигрывает, дальше будет только хуже! Можно утверждать, что в скором времени соревнование человека (любого!) с программно-аппаратным комплексом в шахматах станет бесперспективным!

анализирует и просчитывает в глубину. А шахматная программа, даже применяя наилучшие известные приемы, подобные α-β отсечению, отбрасывает в среднем только половину возможных ходов. Если условно принять, что у человека ширина ветвления в дереве игры постоянна и равна 2, а у программы постоянна и равна 10, то для поиска методом Минимакса на глубину 10 человеку нужно просмотреть $2^{11} = 2048$ позиций, а программе $10^{11} = 100\,000\,000\,000$ позиций.

ВЫВОДЫ

1. Существует большое количество алгоритмов поиска решения, в том числе и для систем продукций. Каждый из алгоритмов имеет область применимости, где он наиболее эффективен и полезен.

2. В искусственном интеллекте применяются эвристические алгоритмы поиска решения, причем эффективность алгоритма критически зависит от качества оценочной функции, которая собственно и определяет эвристику.

3. Для представления пространства поиска чаще всего используются ориентированные графы и графы (деревья) И/ИЛИ.

4. ПРЕДСТАВЛЕНИЕ ЗНАНИЙ ФОРМУЛАМИ ИСЧИСЛЕНИЯ ПРЕДИКАТОВ

Материал этой главы базируется, в основном, на математической логике, основные понятия которой считаются известными читателю. Тем не менее, все необходимые определения и некоторые теоремы кратко повторяются в первых разделах. И все-таки, мы хотим предуведомить читателя, что эта глава никак не является кратким учебником по математической логике. Многие важные, интересные и технически сложные теоремы и понятия формальной математической логики здесь либо опускаются, либо не доказываются, либо даются в упрощенной формулировке. Причина в том, что мы в этой главе рассматриваем математическую логику и ее основные объекты — логические исчисления — ни в коем случае не как цель, а только как средство представления знаний в компьютере. Наше отношение к высокому искусству математической логики в этой главе крайне прагматично: мы просто используем готовый язык и аппарат, не задумываясь о том, какой ценой он достался человечеству. Естественно, применение логических исчислений в ИИ рассматривается исключительно на конкретных примерах. Формальные построения в основном тексте главы сведены к минимуму. Читателя, которому формальные построения необходимы, мы отсылаем к классическому учебнику [19].

4.1. МЕТОД РЕЗОЛЮЦИЙ

Представления знаний с помощью логических формул базируется на следующей идее. Мы будем строить определенные формальные теории, проводить в них поиск формальных доказательств, и интерпретировать полученные результаты в предметной области, где работает интеллектуальная программа. В качестве формальных теорий мы используем прикладные исчисления

на базе исчисления предикатов первого порядка, а в качестве метода поиска доказательства — метод резолюций. Существует, и с успехом применяются, другие классы формальных теорий и другие методы поиска доказательства, и мы указываем на них в дополнениях и в отступлениях, но в основном тексте мы главы мы ограничиваемся этим классическим случаем.

4.1.1. Формальные теории

Исторически понятие формальной теории было разработано в период интенсивных исследований в области оснований математики для формализации собственно логики и теории доказательства. Огромный, может быть, решающий вклад в становление и развитие аппарата формальных теорий внес выдающийся немецкий математик Давид Гильберт. Сейчас этот аппарат широко используется при создании специальных исчислений для решения конкретных прикладных задач в области ИИ. Давид Гильберт писал: «Теория доказательства … дает нам возвышенное чувство убеждения в том, что, по крайней мере, для математического ума не поставлены никакие границы, и что он сам в состоянии проследить законы собственного мышления…».

Используемое нами далее исчисление предикатов является примером формальной теории, поэтому мы начнем с краткого описания общего понятия формальной теории.

Формальная теория *— это структура, имеющая следующие четыре необходимые составляющие:*

- множество A символов, образующих алфавит формальной теории;

- множество F слов в алфавите A, $F \subset A^*$*, которые называются формулами теории;*

- подмножество B формул, $B \subset F$*, которые называются аксиомами формальной теории;*

- множество отношений R на множестве формул, $R \subset F^{n+1}$*, которые называются правилами вывода формальной теории.*

Множество символов A может быть конечным или бесконечным (счётным). Обычно для образования символов используют конечное множество букв, к которым, если нужно, приписываются в качестве индексов натуральные числа.

Множество формул F обычно задается индуктивным определением, например с помощью порождающей формальной грамматики. Как правило, это множество бесконечно. Наряду с термином «формула» могут использоваться термин «предложение». Предложения, отвечающие строгому определению, в отличие от произвольных комбинаций символов из A (просто предложений или цепочек), иногда называют *правильно построенными предложениями*. В некоторых формальных теориях статус правильно построенного предложения может зависеть от контекста или внешних обстоятельств. Ниже, если не оговаривается иное, под *формулой* понимается *правильно построенная формула рассматриваемой формальной теории.*

Множества A и F в совокупности определяют язык, или *сигнатуру формальной теории*, традиционно обозначаемую как

$$\Sigma = (A, F).$$

Множество аксиом B может быть конечным или бесконечным (но счётным). Если множество аксиом бесконечно, то обычно оно задаётся с помощью конечного множества *схем аксиом* и правил порождения конкретных аксиом из схемы аксиом. Самым распространенным правилом порождения конкретных аксиом из схемы аксиом является *правило подстановки*.

Приведем поясняющий пример. В учебнике [19] сказано (цитируем с небольшими сокращениями и в тех обозначениях, которые приняты в этой книге): «Каковы бы ни были формулы A, B и C, следующие формулы суть аксиомы:

(A1) $(A \to (B \to A))$

(A2) $((A \to (B \to C)) \to ((A \to B) \to (A \to C)))$

(A3) $((\neg B \to \neg A) \to ((\neg B \to A) \to B))$ »

Это следует понимать в следующем смысле. Пусть строчные латинские буквы, связки $\neg$ и $\rightarrow$, а также скобки (и) входят в алфавит рассматриваемой формальной теории, а прописные латинские буквы не входят. Тогда написанные выражения, помеченные именами (A1), (A2) и (A3), строго говоря, не являются формулами формальной теории, а значит, не могут быть и аксиомами. Это именно схемы аксиом. Вместо параметров A, B и C можно подставить любые формулы, причем вместо всех вхождений какого-либо параметра должна быть подставлена одна и та же формула, и полученное выражение, которое уже будет являться формулой данной формальной теории (заметим в скобках, что это исчисление высказываний), объявляется аксиомой. В частности, если в схему A1 подставить формулу $(b \rightarrow a)$ вместо параметра A, и формулу a вместо параметра B, (это кратко обозначают следующим образом: $\{(b \rightarrow a)//A, a//B\}$), то получится формула $((b \rightarrow a) \rightarrow (a \rightarrow (b \rightarrow a)))$, которая является правильно построенной формулой и аксиомой исчисления высказываний. Если к той же схеме A1 применить подстановку $\{(a \rightarrow a)//A, a//B\}$, то получится формула $((a \rightarrow a) \rightarrow (a \rightarrow (a \rightarrow a)))$, которая также является аксиомой исчисления высказываний. Множество аксиом исчисления высказываний бесконечно, хотя и может быть задано всего тремя схемами аксиом.

Для прикладных теорий аксиомы зачастую делятся на два вида:

- **логические аксиомы**, прямо заимствованные из базовой формальной теории (в нашем случае — из исчисления предикатов первого порядка);

- **внелогические аксиомы** (также говорят *предметные* или *собственные* аксиомы), отражающие прикладную специфику и содержание конкретной прикладной области.

Введем еще два понятия, относящиеся к формулам и аксиомам.

Если существует алгоритм, который по предъявленной формуле теории (цепочке знаков алфавита теории) за конечное

время позволяет сделать вывод о принадлежности (или не принадлежности) этой цепочки классу правильно построенных формул, то теория называется ***распознаваемой***.

Если при этом время ограничено полиномом от длины записи формулы, то говорят, что теория *эффективно распознаваема*.

Если существует алгоритм, который по предъявленной формуле за конечное время позволяет сделать вывод о том, что эта формула может быть отнесена или не отнесена к множеству аксиом, то формальная теория называется ***аксиоматизируемой***.

Если при этом время ограничено полиномом от длины записи формулы, то говорят, что теория *эффективно аксиоматизируема*.

Множество правил вывода R чаще всего конечно.

Сигнатура, аксиомы и правила вывода в общем случае могут быть выбраны произвольным образом, с учетом ограничений, указанных в определении. Однако конструирование формальных теорий только ради самих формальных теорий малопродуктивно[62]. В контексте данной книги нас интересуют такие формальные теории, которые описывают какие-то реальные объекты и связи между ними. Совокупность реальных объектов и связей между ними часто называют *предметной областью*. *Три важных понятия — выводимость, логическое следование и интерпретация — образуют круг концепций, через которые происходит формальное описание неформальных вещей*.

Говорят, что формула G ***непосредственно выводима*** *из формул множества $F_1, \dots, F_n$, если среди правил вывода существует такое правило R, что $(F_1, \dots, F_n, G) \in R$.*

Формулы $F_1, \dots, F_n$ называют *посылками*, а формулу G — *заключением* правила вывода R. В этой книге непосредственную

---

[62] «Малопродуктивно» — исключительно, в контексте *прикладных исследований*. В «чистой» математической логике практика построения систем, не имеющих ровным счётом никакого отношения к «реальному миру», является обычной и вполне продуктивной.

выводимость мы, как правило, будем записывать, разделяя посылки и заключение горизонтальной чертой и указывая, если нужно, название правила вывода:

$$\frac{F_1, \dots, F_n}{G} R .$$

*Говорят, что формула G **выводима** из формул множества $F_1, \dots, F_n$ в данной формальной теории, если существует последовательность формул*

$$<A_1, A_2, \dots, A_i, \dots, A_n>,$$

*в которой $A_n = G$[63], а каждая формула A_i является либо аксиомой, либо одной из формул $F_1, \dots, F_n$, либо **непосредственно выводима** из некоторых предшествующих формул по одному из правил вывода R.*

Тот факт, что формула G выводима из формул $F_1, \dots, F_n$, записывают так:

$$F_1, \dots, F_n \vdash G.$$

Формулы $F_1, \dots, F_n$ называются *гипотезами* вывода, а сама последовательность $<A_1, A_2, \dots, A_i, \dots, A_n>$ называется (формальным) *выводом*[64].

---

[63] Имеется в виду — и это очень важно! — *синтаксическое равенство* (совпадение цепочек символов), в отличие от «привычных» равенств вида $2 + 3 = 5$ или $y = \sin(x)$. В логике для этого принято использовать специальный символ «$\doteq$», но мы решили его не использовать, а просто в необходимых случаях подчёркивать факт синтаксического совпадения словесно.

[64] В математической логике вывод — это *вся* последовательность шагов рассуждения, а не только заключительное утверждение, как в обыденной жизни.

Если формула G выводима только из аксиом — без использования гипотез, то G называется ***теоремой формальной теории****. Этот факт обозначается записью* $\vdash G$.

Выше мы определили понятия (эффективной) *распознаваемости* и (эффективной) *аксиоматизируемости*. Полагаем необходимым сделать аналогичное замечание относительно понятия *теоремы*, практически дословно цитируя Э. Мендельсона [19]: «*Даже в случае эффективно аксиоматизируемой теории понятие теоремы не обязательно эффективно, ибо, вообще говоря, может и не существовать процедуры (алгоритма), позволяющей узнавать по данной формуле, существует ли её вывод в теории. Теория, для которой такой алгоритм существует, называется разрешимой, в противном случае теория называется неразрешимой. Грубо говоря, разрешимая теория — это такая теория, для которой можно изобрести машину, испытывающую формулы на свойство быть теоремой этой теории, в то время как для выполнения той же задачи в неразрешимой теории требуются всё новые и новые независимые акты изобретательства*».

Мы видим, что понятия выводимости, вывода и теоремы являются чисто синтаксическими, они не апеллируют к семантическому понятию истинности. Если в формальной теории утверждается, что теорема выводима из аксиом, то это совсем не значит, что теорема в каком-то содержательном смысле «истинна» или «верна». Это просто значит, что удалось найти формальную последовательность формул, которая начинается определенным образом (аксиомами), заканчивается определенным образом (теоремой), и все промежуточные формулы связаны определенным образом (правилами вывода). Для перехода к семантике нужно ввести понятие интерпретации.

Интерпретация **I** *формальной теории в предметную область M — это множество функций, которые отображают конструкции*

формальной теории во множество объектов и утверждений относительно объектов предметной области M.

Множество функций, а не одна функция, нужно потому, что для каждого сорта формальных конструкций, используемых в формулах, нужна своя функция. Например, при интерпретации формул исчисления предикатов (см. следующий параграф) требуется каждой формальной константе сопоставить фактический объект, формальным функциональным символам — конкретные отображения, формальным предикатам — конкретные отношения. После этого формула становится содержательным утверждением, и с содержательной точки зрения мы можем попытаться проверить, истинно утверждение или ложно.

Истинностное значение формулы A в интерпретации **I** мы будем обозначать $A|_{\mathbf{I}}$ или, если из контекста однозначно понятно, что речь идет именно об интерпретации **I**, просто именем самой формулы с чертой интерпретации: $A|$. Истинностное значение формулы всегда определяется одной из констант **true** или **false**, чтобы не связывать себя записями типа $A|_{\mathbf{I}}$=**true**, мы будем писать $A|_{\mathbf{I}}$ ($\neg A|_{\mathbf{I}}$), подразумевая, что формула имеет значение **true** (**false**), или же просто $A|$ ($\neg A|$) — с учётом сделанного выше замечания об однозначности.

*Формула называется **общезначимой** (или **тавтологией**), если она истинна в любой интерпретации:* $\forall\mathbf{I}$ ($A|_{\mathbf{I}}$).

Формула называется выполнимой*, если существует, по крайней мере, одна интерпретация* **I***, в которой она истинна:* $\exists\mathbf{I}$ ($A|_{\mathbf{I}}$).

*Формула называется **невыполнимой** (или **противоречием**), если она ложна в любой интерпретации:* $\forall\mathbf{I}$ ($\neg A|_{\mathbf{I}}$).

Пусть есть некоторая формальная теория, некоторая предметная область, и задана интерпретация **I**. Обозначим буквой T множество теорем формальной теории, а буквой P множество истинных утверждений предметной области. Если оказывается, что множества T и P взаимно-однозначно соответствуют друг другу при интерпретации **I**, то лучшего и пожелать нельзя: все формально

выводимые утверждения содержательно истинны, а все содержательно истинные утверждения формально выводимы. В таком случае говорят, что формальная теория *адекватна* предметной области. Адекватность — сильное и редкое свойство. На практике чаще приходится довольствоваться более слабыми свойствами.

Говорят, что предметная область является ***моделью формальной теории****, если существует интерпретация* **I**, *при которой все выводимые формулы истинны в этой интерпретации:* $\forall G\,(\vdash G \rightarrow\ (G|_{\mathbf{I}}))$.

Говорят, что формальная теория ***полна*** *для предметной области, если существует интерпретация* **I**, *при которой все истинные формулы выводимы:* $\forall G\,((G|_{\mathbf{I}}) \rightarrow\ \vdash G)$.

Таким образом, адекватная теория имеет модель и полна в своей модели.

Необходимо ясно понимать, что формальные теории не являются божественными откровениями или имманентно врожденными законами человеческого разума — формальные теории придумываются людьми для формализации конкретных предметных областей, и эти придумки оказываются более или менее удачными, в зависимости от изобретательности придумывающего. Для одной и той же предметной области можно придумать разные формализации, и часто так и происходит: разные формализации конкурируют друг с другом и меняют друг друга[65].

---

[65] Самым известным примером, является, видимо, геометрия. «Начала» Евклида — исторически, наверное, самая первая формальная теория, которая служила образцом два тысячелетия, хотя и не была построена достаточно строго и формально с современной точки зрения. Многие математики пытались улучшить построения Евклида. Лобачевского, например, такие попытки привели к открытию неевклидовых геометрий. Чуть более ста лет тому назад Давид Гильберт написал книгу «Основания геометрии», и эта работа стала новым эталоном качества формальной теории. Уже в конце XX века отечественный математик А. В. Погорелов предложил свою аксиоматику геометрии,

Семантическое понятие интерпретации ведет к понятию логического следования.

Логическое следование. *Говорят, что формула G логически следует из формул* $F_1, ..., F_n$ *(обозначение* $F_1, ..., F_n \Rightarrow G$*), если формула G истинна во всякой интерпретации, в которой истинны формулы* $F_1, ..., F_n$*:*

$$F_1|_{\mathbf{I}} \& \ldots \& F_n|_{\mathbf{I}} \rightarrow G|_{\mathbf{I}}.$$

Логическое следование — это семантическое понятие. Выводимость — это синтаксическое понятие. Они могут быть связаны, и они связаны однозначно, если теория адекватна, а могут и не быть связаны так прямо. Все зависит от правил вывода R.

Говорят, что правила вывода R ***состоятельны****, если из формальной выводимости получается логическое следование:* $F_1, ..., F_n \vdash_R G \rightarrow F_1, ..., F_n \Rightarrow G$.

Говорят, что правила вывода R ***полны***[66]*, если любое логическое следование подтверждено формальной выводимостью:* $F_1, ..., F_n \Rightarrow G \rightarrow F_1, ..., F_n \vdash_R G$.

4.1.2. Язык исчисления предикатов первого порядка

Основным инструментом представления знаний, рассматриваемым в этой главе, являются формулы прикладного исчисления предикатов первого порядка. Еще Давид Гильберт указал, что «исчисления предикатов первого порядка достаточно во всех

весьма удачную и современную, так что даже рассматривался вопрос о введении этого варианта формальной теории в качестве основного в курс средней школы. Все эти формальные теории различны, но предметная область — множество содержательно истинных утверждений о свойствах геометрических объектов — одна!

[66] Чаще понятие состоятельности не используется, а полнота трактуется как эквивалентность доказуемости и следования: $F_1, \ldots, F_n \Rightarrow G \equiv F_1, \ldots, F_n \vdash_R G$. Мы сочли целесообразным отдельно называть и рассматривать те две импликации, которые образуют эту эквивалентность.

разумных случаях». Мы рассматриваем здесь исчисление предикатов как язык, как систему обозначений для записи логических утверждений. Более того, мы считаем, что этот язык знаком читателю на уровне таких понятий, как логические связки («И», «ИЛИ», «ЕСЛИ … ТО») и кванторы («для всякого», «существует»)[67].

Мы рассматриваем язык, в котором участвуют логические связки, кванторы и атомарные формулы, построенные над множеством функциональных символов, предметных переменных и констант. Ниже приведена формальная грамматика языка исчисления предикатов первого порядка. При этом считаются заданными следующие множества отличимых друг от друга символов: множество символов констант (обозначение `Конст`), множество символов переменных (обозначение `Перем`), множество символов функций (обозначение `Функ`) и множество символов предикатов (обозначение `Предикат`).

```
Терм:             Конст | Перем | Функ (Список термов)
Список термов:        Терм | Терм, Список термов
Атом:             Предикат | Предикат (Список термов)
Литерал:          Атом | ¬ Атом
Формула:          Атом | ¬ ( Формула ) |
                  (Формула Связка Формула) |
                  Квантор Перем ( Формула )
Связка:           ∧ | ∨ | →
Квантор:              ∀ | ∃
```

Понятие *литерала* (атом или отрицание атома), вообще говоря, грамматически излишне. Мы ввели его, чтобы иметь под рукой этот термин. Предикат без аргументов также излишен. Мы ввели его для того, чтобы исчисление высказываний синтаксически было частным случаем исчисления предикатов.

---

[67] В предыдущем параграфе мы уже использовали некоторые обозначения исчисления предикатов (кванторы ∀, ∃ и связку →) без определений, именно в расчете на то, что читатель с ними знаком.

Необходимо сделать несколько замечаний, касающихся грамматики этого языка.

1. Во-первых, в этом языке мы считаем, что наши функциональные и предикатные символы имеют фиксированные имена и фиксированное число аргументов, как в обычных системах программирования. Это довольно сильное допущение, и можно, в принципе, так не считать. В таком случае нам пришлось бы иметь дело с λ-выражениями и смешанными вычислениями, что выходит за рамки данного курса.

2. Во-вторых, язык, который мы используем, в логической иерархии классифицируется, как язык первого порядка. Другими словами, это означает, что под кванторами могут стоять только предметные переменные, т. е. не функциональные символы и не предикаты. В противном случае, то есть если допускается, что квантор может связывать не только предметные переменные, то это будет уже исчисление не первого порядка. В реальных современных системах ИИ используются именно такие исчисления. Например, модное направление Model Checking основано на темпоральных логиках, которые выглядят как логики первого порядка, а на самом деле сводятся к стандартным логикам не первого порядка.

3. В-третьих, приведенная грамматика языка исчисления предикатов предполагает формальную запись утверждений в определенном синтаксисе, со всеми необходимыми скобками и т. д. В нашем рассмотрении примеров мы, естественно, сильно упростим формальный синтаксис, и будем записывать формулы как удобнее, однако так, чтобы всегда можно было догадаться, какие синтаксические упрощения используются в том, или ином случае. В частности, используется обычный приоритет связок и кванторов, лишние скобки систематически опускаются.

4. В-четвертых, известно, что чистое исчисление предикатов первого порядка имеет ограниченную выразительную силу. Другими словами, многие содержательные теории невыразимы или трудно

выразимы в чистом исчислении первого порядка, необходимы внелогические допущения. Естественно, что мы допускаем константы, функциональные символы и предикаты, которые нужны для описания предметной области. Но помимо этого, считается, что натуральные числа и другие общеизвестные понятия можно использовать в любых формулах, не пытаясь аксиоматически описать, что на самом деле это такое.

5. В-пятых, замкнутых формул вполне достаточно, и можно не рассматривать незамкнутые формулы, потому, что если есть незамкнутая формула, то она будет выполнима тогда и только тогда, когда выполнимо соответствующее замыкание. Поэтому формулы записываются в открытой форме, при этом считается, что все свободные переменные замыкаются квантором всеобщности.

4.1.3. Предложения

В 1965 году для доказательства общезначимости логических формул Джон Алан Робинсон предложил *метод резолюций*. Идея этого метода основана на работах Эрбрана, но поскольку тот не думал о применении своего метода на компьютерах (что было не удивительно в 1930 г.), метод Эрбрана оказался не очень подходящим с вычислительной точки зрения. Робинсон сделал его пригодным для реального применения и разработал алгоритм унификации, являющийся основой метода.

Метод резолюций применим в логических исчислениях, в которых используются формулы особого вида, называемые предложениями.

Предложение *(clause) — это бескванторная дизъюнкция литералов.*

Любую формулу исчисления предикатов первого порядка можно свести к предложениям с помощью последовательного применения следующих действий:

1. Элиминация импликации. Импликация заменяется дизъюнкцией с отрицанием первого операнда. $A \to B = \neg A \vee B$.

2. Протаскивание отрицания. Отрицания оставляются только перед атомами. Осуществляется по правилам де Моргана с использованием свойства кванторов и инволютивность отрицания. $\neg(A\&B) = (\neg A\vee\neg B)$, $\neg(A\vee B) = (\neg A\&\neg B)$, $\neg\forall x(\mathrm{A}) = \exists x(\neg\mathrm{A})$, $\neg\exists x(A) = \forall x(\neg A)$, $\neg\neg\mathrm{A} = \mathrm{A}$.

3. Разделение связанных переменных. Связанные переменные переименовываются таким образом, чтобы они случайно не совпадали в разных кванторах. $\forall x(\ldots\exists x(\ldots x\ldots)\ldots) = \forall x(\ldots\exists y(\ldots y\ldots)\ldots)$, и аналогично для любых других комбинаций кванторов.

4. Приведение к предваренной форме. Преобразование формулы к такому виду, когда она начинается с кванторного префикса, за которым следует бескванторная матрица. $A\vee\forall x(B) = \forall x(A\vee B)$, и аналогично для любых других комбинаций кванторов и связок.

5. Сколемизация. Элиминация кванторов существования с помощью ввода новых функциональных символов и констант. $\exists x(A(\ldots x\ldots)) = A(\ldots a\ldots)$, $\forall x(\exists y(A(\ldots y\ldots)))= \forall x(A(\ldots f(x)\ldots))$, где a — новая константа, а f — новый функциональный символ.

6. Элиминация кванторов всеобщности. Поскольку рассматриваются только замкнутые формулы, оставшиеся кванторы всеобщности опускаются, так как они подразумеваются. $\forall x(A(\ldots x\ldots)) = A(\ldots x\ldots)$.

7. Приведение к конъюнктивной нормальной форме. Главной операцией в конъюнктивной нормальной форме является конъюнкция, она соединяет группы дизъюнкций. $A\vee(B\&C) = (A\vee B)\&(B\vee C)$.

8. Разбиение на предложения. Формула разбивается на множество предложений. $A\&B = A, B$.

Это не единственно возможный способ сведения формул исчисления предикатов к предложениям, существует и другие эквивалентные способы. Сведение к предложениям не является

логически эквивалентным, но справедливо следующее утверждение, которое мы приводим без доказательства.

Теорема. *Те формулы, которые получились в результате сведения исходных формул к предложениям, образуют противоречивое множество тогда и только тогда, когда исходное множество формул не выполняется*.

Рассмотрим примеры сведения формул исчисления предикатов первого порядка к предложениям с использованием последовательности действий, указанного выше.

Сведем к предложениям формулу: $\forall x\ (P(x) \rightarrow \exists y\ Q(x, y))$.

1) $\forall x\ (\neg P(x) \vee \exists y\ Q(x, y))$
2) формула без изменений
3) формула без изменений
4) $\forall x\ \exists y\ (\neg P(x) \vee Q(x, y))$
5) $\forall x\ (\neg P(x) \vee Q(x, f(x)))$
6) $\neg P(x) \vee Q(x, f(x))$
7) формула без изменений
8) формула без изменений

Сведем теперь к предложениям **отрицание** предыдущей формулы: $\neg\forall x\ (P(x) \rightarrow \exists y\ Q(x, y))$.

1) $\neg\forall x\ (\neg P(x) \vee \exists y\ Q(x, y))$
2) $\exists x\ (P(x)\ \&\ \forall y\ (\neg Q(x, y)))$
3) формула без изменений
4) $\exists x\ \forall y\ (P(x)\ \&\ \neg Q(x, y))$
5) $\forall y\ (P(a)\ \&\ \neg Q(a, y))$
6) $P(a)\ \&\ \neg Q(a, y)$
7) формула без изменений
8) $P(a), \neg Q(a, y)$

4.1.4. Правило резолюции

В методе резолюций используется не только особый вид формул, но и особое правило вывода.

Правило резолюции *— правило вывода новой формулы (предложения) из двух предложений.* ***Резольвента*** *— новое предложение, полученное в результате применения правила резолюции.*

Правило резолюции применимо только к предложениям, содержащим *контрарные унифицируемые литералы* (P и $\neg P$) и состоит в следующем. Если даны два предложения (содержащие контрарные унифицируемые литералы), из этой пары предложений можно вывести новое предложение, объединяя исходные предложения и выбрасывая (элиминируя) контрарные литералы:

$$\frac{P \vee Q_1 \vee \ldots \vee Q_n, \quad \neg P \vee R_1 \vee \ldots \vee R_m}{Q_1 \vee \ldots \vee Q_n \vee R_1 \vee \ldots \vee R_m}.$$

Напомним, что мы очень любим записывать правила вывода «многоэтажно» т. е. следующим образом: над чертой размещаются посылки (*антецедент*) правила, под чертой заключение (*сукцедент*).

Резольвента есть объединение (точнее, дизъюнктивное соединение) всех литералов за исключением пары контрарных. При этом как индекс n, так и индекс m могут иметь *любое* значение. В частности, если резольвируются предложения, которые содержат только контрарные литералы, то резольвента — *пустое предложение*. Заметим, что по определению пустое предложение считается тождественно ложным.

Правило резолюции — очень мощное правило! Многие правила (например, Modus ponens[68]) являются частными проявлениями правила резолюции. Ниже в табл. 4.1 приведены некоторые известные правила в традиционной форме и записанные в ограничениях метода резолюции, когда посылки и заключения являются бескванторными

---

[68] Латинское словосочетание modus ponendo ponens, обычно сокращаемое до modus ponens, сейчас на русский язык переводят как «правило отделения». По средневековой традиции логические термины используются на языке оригинала, без перевода.

дизъюнкциями литералов. Из этих примеров видно, что правило резолюции не слабее традиционных, и все, что можно вывести традиционным способом с помощью modus ponens, можно вывести и с помощью правила резолюции!

Таблица 4.1

Традиционная и резолютивная форма правил вывода

Название правила	Традиционная форма	Резолютивная форма
Modus ponens	$\dfrac{A,\quad A\to B}{B}$	$\dfrac{A,\quad \neg A\vee B}{B}$
Транзитивность	$\dfrac{A\to B,\quad B\to C}{A\to C}$	$\dfrac{\neg A\vee B,\quad \neg B\vee C}{\neg A\vee C}$
Слияние	$\dfrac{A\vee B,\quad A\to B}{B}$	$\dfrac{A\vee B,\quad \neg A\vee B}{B}$

Правило резолюции само по себе в одиночку образует полную и состоятельную систему правил (см. параграф 4.1.1). Докажем это утверждение для случая исчисления высказываний, дав подходящие упрощенные определения.

Правило вывода ***состоятельно****, если заключение является логическим следствием посылок.*

Теорема. *Правило резолюции состоятельно*.

Доказательство. Пусть **I** — произвольная интерпретация, C_1 и C_2 — это предложения, которые резольвируются по правилу резолюции, P — тот литерал, по которому ведется резольвирование. Пусть, для определенности, предикат P входит в предложение C_1 положительно, а в C_2 с отрицанием. Пусть C_1' и C_2' — это то, что получается после выбрасывания из предложений контрарных литералов.

По условию теоремы $\mathbf{I}(C_1)\ \&\ \mathbf{I}(C_2)$. Тогда если $\mathbf{I}(P)$, то $C_1' \neq \varnothing\ \&\ \mathbf{I}(C_1')$, а значит $\mathbf{I}(C_1' \vee C_2)$. Если же $\mathbf{I}(\neg P)$, то $C_2' \neq \varnothing\ \&\ \mathbf{I}(C_2')$, а значит все равно $\mathbf{I}(C_1' \vee C_2')$. **Ч.т.д.**

Правило вывода ***полно****, если любая общезначимая формула может быть выведена по этому правилу.*

Теорема. *Правило резолюции полно*.

Доказательство. Как известно, правило modus ponens для исчисления высказываний полно и является частным случаем резолюции, как видно из табл. 4.1. **Ч.т.д.**

4.1.5. Унификация

Для того чтобы применить метод резолюций, нам нужно найти контрарные литералы. В случае исчисления высказываний, такой проблемы нет. Любая переменная и та же переменная с отрицанием — это и есть пара контрарных литералов. В случае исчисления предикатов первого порядка это не так просто. Даже когда мы нашли два вхождения одного предиката, положительное и с отрицанием, это еще не значит, что найдена контрарная пара. После предикатной буквы могут стоять разные аргументы, и они могут отличаться в том и в другом случае. В некоторых случаях путем подстановки можно литералы *унифицировать*, то есть сделать такие замены переменных, что атомы литералов сделаются одинаковыми, но в других случаях этого сделать нельзя. Существуют алгоритмы, в частности, алгоритм 4.1, который проверяет, можно ли унифицировать две формулы (в частности, два атома) и находят набор унифицирующих подстановок, если унификация возможна. Уточним определения.

Частный случай формулы *— результат подстановки в заданную формулу некоторой формулы вместо всех вхождений определенной переменной в исходную формулу.*

Унификация *— процедура вычисления* ***общего*** *частного случая для двух формул, (если он существует).*

Унификатор *— это такой* ***набор подстановок*** *определенных формул вместо переменных, при применении которого к заданным формулам, получается их общий частный случай.*

Наиболее общий унификатор *(Least Common Unifier) — это минимальный набор подстановок, который унифицирует формулы.*

Сделаем несколько общих замечаний к определениям.

1. Во многих формальных теориях **частный случай формулы является ее логическим следствием**. В частности, это справедливо для формул исчисления высказываний и для предложений (бескванторных дизъюнкций литералов).

2. **В подавляющем большинстве случаев формулы не имеют общего частного случая**. Поэтому при унификации важно как можно быстрее получать отрицательный ответ.

3. Вообще говоря, вместо переменной можно подставлять терм, содержащий эту же переменную. Однако **такая подстановка никогда не может привести к унификации**, поэтому в унификаторах подобные подстановки запрещены.

4. Если σ — наиболее общий унификатор (стандартное сокращение Н.О.У.), а θ — любой унификатор, то существует такая подстановка ρ, что $\theta = \sigma^\circ\rho$, где° — знак композиции подстановок.

Например, две формулы $P(x, f(a))$ и $P(g(b), y))$ унифицируемы,

$$\text{Н.О.У. } [P(x, f(a)), P(g(b), y))] = \{g(b)//x, f(a)//y\}.$$

В то же время формулы $P(x, f(a))$ и $P(y, g(b))$ не унифицируемы, потому что $f(a) \neq g(b)$ при любых подстановках формул вместо переменных. Также не унифицируемы формулы $P(x, f(a))$ и $P(g(b), x))$, поскольку для унификации требуется подставить вместо переменной x формулу $f(a)$ и формулу $g(b)$, что, очевидно, невозможно.

Мы приводим классический алгоритм унификации (алгоритм 4.1), предложенный Робинсоном, в несколько упрощенной формулировке. Сразу заметим, что этот алгоритм — далеко не самый эффективный их известных в настоящее время. На вход алгоритма подаются два выражения `A` и `B`, записанные в префиксной (функциональной) форме, на выходе получается значение **`false`**, если выражения не унифицируемы, или значение **`true`** и Н.О.У σ.

Алгоритм 4.1. Унификация двух формул.

```
proc Unify (in A, B; out σ) : bool
    σ : = ∅
```

```
    while A≠B do
        [A′,B′] : = FisrtNeqTerms(A, B)
        if ¬ IsSubst(A′,B′,t,v) then return (false)
        end if
        σ : = σ°{t//v}
        A : = Subst(A, t, v)
        B : = Subst(B, t, v)
    end while
    return (true)
end proc
```

В этом алгоритме используются следующие функции и операции.

1. Функция `FisrtNeqTerms` сравнивает свои аргументы слева направо и находит первые подформулы, в которых сравниваемые формулы отличаются. Например, сравнивая формулы $P(x, f(a))$ и $P(g(b), y))$ функция `FisrtNeqTerms` вернет пару $[x, g(b)]$.

2. Функция `IsSubst(A′,B′,t,v)` проверяет, образует ли первая пара аргументов `A′` и `B′` допустимую подстановку, то есть, является ли один из этих аргументов некоторой переменной `v`, а второй — формулой `t`, не содержащей вхождений этой переменной `v`. Например, если первый и второй аргументы функции равны x и $g(b)$ соответственно, то функция вернет значение true, причем третий аргумент `t` получит значение $g(b)$, а четвертый аргумент `v` получит значение x.

3. Функция `Subst(A, t, v)` подставляет в формулу `A` формулу `t` вместо всех вхождений переменной `v`. Например, если аргументы функции $P(x, f(a))$, $g(b)$, x соответственно, то результатом функции является формула $P(g(b), f(a))$.

4. Операция ° является композицией подстановок. Выполнение ее состоит в том, что все подстановки правого аргумента применяются ко всем подставляемым формулам левого аргумента,

после чего множества подстановок объединяются, а совпадающие и тривиальные подстановки удаляются.

Рассмотрим пример — протокол работы алгоритма 4.1 для пары формул $P(a, x, f(g(y)))$ и $P(z, f(z), f(u))$. В этом примере в каждой строке выписаны текущие значения переменных `A` и `B`, причем найденные функцией `FirstNeqTerms` несовпадающие подформулы подчеркнуты. Справа выписан построенный унификатор.

$P(\underline{a}, x, f(g(y)))$ $P(\underline{z}, f(z), f(u))$ $\sigma = \{a//z\}$

$P(a, \underline{x}, f(g(y)))$ $P(a, \underline{f(a)}, f(u))$ $\sigma = \{a//z, f(a)//x\}$

$P(a, f(a), f(\underline{g(y)}))$ $P(a, f(a), f(\underline{u}))$ $\sigma = \{a//z, f(a)//x, g(y)//u\}$

$P(a, f(a), f(g(y)))$ $P(a, f(a), f(g(y)))$

Мы видим, что всего за четыре шага алгоритм заканчивает работу и находит наиболее общий унификатор. Однако не следует обольщаться и считать, что задача унификации — это простая задача. На самом деле, все «элементарные» операции, использованные в этом алгоритме — нахождение несовпадающих подформул, проверка допустимости подстановки, выполнение подстановки и даже вычисление композиции подстановок — в реализации отнюдь не элементарны, и их фактическая трудоемкость кардинально зависит от выбранного способа представления формул. Между тем, представление формул для алгоритма 4.1 не определено.

Рассмотрим еще один алгоритм унификации, в котором структура формул фиксирована более явно. Применительно к рассматриваемому методу резолюций, нам нужно проверять унифицируемость атомарных формул, синтаксис которых описан в параграфе 4.1.2. Чтобы не затуманивать идею алгоритма, мы оставим в языке только функциональные символы и переменные[69].

---

[69] Предикатные символы с точки зрения унификации ничем не отличаются от функциональных символов, а константы, хотя и требуют в коде алгоритма большого числа отдельных проверок, по существу подчиняются одному простому правилу: константа унифицируется только с переменной.

В алгоритме 4.2 предполагается, что функция `op` осуществляет синтаксический разбор формулы и возвращает знак главной операции (имя предиката, функциональный символ или имя переменной), функция var проверяет, является ли ее аргумент именем переменной, а функция `args` возвращает список аргументов главной операции. Набор подстановок (унификатор) представлен в виде глобального массива `S`, значениями индекса которого являются имена переменных, а значениями элементов — формулы, которые подставляются вместо соответствующих переменных.

Алгоритм 4.2 основан на следующей идее. При любых подстановках формул вместо переменных главная операция (предикат или функциональный символ) формулы остается неизменной. Поэтому, если главные операции формул различны, то формулы заведомо не унифицируемы. В противном случае, то есть, если главные операции совпадают, формулы унифицируемы тогда и только тогда, когда унифицируемы все подформулы, являющиеся аргументами главной операции. Эта рекурсия заканчивается, когда сопоставление доходит до переменных. Переменная унифицируется с любой формулой, не содержащей этой переменной (в частности, с другой переменной) простой подстановкой этой формулы вместо переменной. Но подстановки для всех вхождений одной переменной должны совпадать.

Алгоритм 4.2. Рекурсивная унификация двух формул.

```
proc Unify (in A, B) : bool
    a : = f(A); b : = f(B)
    // обе формулы суть переменные
    if var(a) & var(b) then
        if a = b then return true end if
        if S[a]=∅ & S[b]≠∅ then S[a] : = S[b]
                                return true end if
        if S[b]=∅ & S[a]≠∅ then S[b] : = S[a]
                                return true end if
        if S[a]≠∅ & S[b]≠∅ then Unify(S[a], S[b])
```

```
        end if
        S[a] : = b             // или S[b] : = a
    end if
    // одна формула — переменная, а другая — нет
    if var(a)∨var(b) then
        if var(a) & a∈B then return false else
            if S[a] = ∅ then S[a] : = B;
                                return true
            else    return (S[a] = B)
            end if
        end if
        if var(b) & b∈A then return false else
            if S[b] = ∅ then S[b] := A;
                                return true
            else return (S[b] = A)
            end if
        end if
    end if
    // обе формулы не переменные
    if a≠b then return false end if
    // главные операции различны
    for a∈args(A) || b∈args(B) do
        if ¬Unify(a, b) then return false end if
    end for
    return true
end proc
```

В данном алгоритме заголовок цикла **for** `a∈args(A) || b∈args(B)` **do** означает параллельный цикл по спискам аргументов.

4.1.6. Опровержение методом резолюций

Процедура доказательства по методу резолюций на самом деле является процедурой опровержения, то есть вместо доказательства общезначимости формулы доказывается, что ее отрицание

противоречиво, в значительной степени, такой подход выбран, исходя из соображений «технического удобства». Это типичная схема доказательства от противного. Докажем теорему, обосновывающую правильность этой схемы для случая исчисления высказываний.

Теорема. *Если* $A, \neg G \vdash F$, *где* F *противоречие, то* $A \vdash G$.

Доказательство. По теореме дедукции[70] если $A, \neg G \vdash F$, то $\vdash A \& \neg G \to F$, и значит $A \& \neg G \to F$ — тавтология. Но $A \& \neg G \to F = \neg(A \& \neg G) \vee F = \neg A \vee G = A \to G$. Значит $A \to G$ — тавтология и $\vdash A \to G$, откуда, еще раз применяя теорему дедукции, имеем $A \vdash G$. **Ч.т.д.**

Поскольку мы рассматриваем дизъюнктивные формы, то пустые дизъюнкции следует рассматривать как тождественно ложные. Значит, в качестве невыполнимой формулы F, которая упомянута в предыдущей теореме, удобно использовать пустую дизъюнкцию, не содержащую никаких дизъюнктивных слагаемых. Именно так традиционно делается в методе резолюций, а пустая дизъюнкция традиционно обозначается знаком $\square$.

Следствие. *Если* $A, \neg G \vdash \square$, *где* $\square$ — *пустая формула*, то $A \vdash G$.

Таким образом, можно сформулировать определение метода резолюций.

Метод резолюций *— это метод автоматического доказательства теорем, основанный на опровержении множества посылок A и отрицания целевой теоремы G с использованием правила резолюции.*

Рассмотрим порядок применения метода резолюций:

Пусть необходимо установить выводимость $A \vdash G$.

---

[70] Теорема дедукции утверждает, что $A \vdash B \Leftrightarrow A \to B$.

1. Каждая формула множества формул A и формула $\neg G$ **независимо** преобразуются во множества предложений.

2. В полученном множестве предложений отыскиваются **резольвируемые** предложения.

3. К ним применяется правило резолюций, и резольвента добавляется во множество до тех пор, пока не будет получено **пустое предложение**.

При этом возможны следующие исходы:

1. В результате применения метода резолюций получается пустая формула. Это означает, что теорема доказана. Мы построили вывод формулы G из формул A.

2. В результате применения метода резолюций не получается пустая формула, и среди текущего множества предложений нет резольвируемых. Это означает, что мы опровергли теорему, то есть что формула G невыводима из множества формул A.

3. Процесс не заканчивается. Правило резолюции применяется, множество предложений пополняется, среди них нет пустых, и есть резольвируемые. *Это ничего не означает*.

Другими словами, *метод резолюций является частично корректной процедурой*. Третий исход нельзя назвать зацикливанием, поскольку нет никакого способа определить, почему метод резолюции продуцирует новые резольвенты, но при этом не получается пустого предложения. Означает ли это, что теорема верна, но мы просто не дождались завершения? Не известно. Означает ли это, что в результате так никогда и не получится пустой формулы? И этого мы тоже не знаем. Это логически неразрешимая задача. В принципе нельзя узнать, как долго нужно ждать для того, чтобы получить ответ на этот вопрос.

4.1.7. Программная реализация метода резолюций

Все сказанное выше можно записать в виде алгоритма 4.3.

Алгоритм 4.3. Метод резолюций.

```
proc R(S)
```

```
    C : = S
    while □ ∉ C do
        Select(C, p1, p2, l1, l2, s)
        if p1,p2 = nil then return fail end if
        p : = Res(p1, p2, l1, l2, s)
        C : = C + p
    end while
    return OK
end proc

proc Res (p1, p2, l1, l2, s)
    p1 : = p1{s}; p2 := p2{s}
    p1 : = p1 – l1; p2 := p2 – l2
    return p1 ∨ p2
end proc

proc Select(C, p1, p2, l1, l2, s)
Выбрать в множестве предложений C два предложения p1,
p2, содержащие контрарные литералы l1, l2,
унифицируемые наиболее общим унификатором s
end proc
```

При записи этих процедур используется наша стандартная практика составления алгоритмов для решения переборных задач — то, что можно фиксировать, фиксировано. В частности, процедура `Res` фиксирована. Нефиксированной процедурой является процедура `Select`. Стратегия управления перемещена в эту функцию, которая выбирает два предложения для резольвирования.

Пример

Всякий программист знает метод резолюций.

Студенты не знают метода резолюций.

Некоторые студенты умные люди.

Следовательно, некоторые умные люди не являются программистами.

Логически последнее утверждение верно. Проверим его механическим путем с помощью метода резолюций.

Введем множество предикатов, описывающих высказывания, используемые в вышеприведенных утверждениях:

П — «быть программистом»,

Р — «знать метод резолюций»,

С — «быть студентом»,

У — «быть умным человеком».

На языке формул исчисления предикатов первого порядка исходные утверждения (посылки) будут выглядеть следующим образом:

$\forall x$ П(x) $\to$ Р(x),

$\forall x$ С(x) $\to \neg$Р(x),

$\exists x$ С(x)У(x).

Целевая формула: $\exists x$ У(x) $\neg$П(x).

Отрицание целевой формулы: $\forall x$ $\neg$У(x) $\vee$П(x).

Опровергнем ее с помощью метода резолюций.

1. Сводим к предложениям исходные формулы:

$\forall x$ П(x) $\to$ Р(x) преобразуется в $\neg$ П(x) $\vee$ Р(x),

$\forall x$ С(x) $\to \neg$Р(x) преобразуется в $\neg$С(x)$\vee\neg$Р(x),

$\exists x$ С(x)У(x) преобразуется в С(a) и У(a),

$\forall x$ $\neg$У(x) $\vee$П(x) преобразуется в $\neg$У(x) $\vee$П(x).

Таким образом, мы получили следующее множество предложений S:

$$S = \{\neg \text{П}(x) \vee \text{Р}(x),\ \neg\text{С}(x)\vee\neg\text{Р}(x),\ \text{С}(a),\ \text{У}(a),\ \neg\text{У}(x) \vee\text{П}(x)\}$$

2. Применяем правило резолюции к 4-му и 5-му предложениям и добавляем их к множеству предложений:

$$S = \{\neg \text{П}(x) \vee \text{Р}(x),\ \neg\text{С}(x)\vee\neg\text{Р}(x),\ \text{С}(a),\ \text{У}(a),\ \neg\text{У}(x) \vee\text{П}(x),\ \text{П}(a)\}$$

3. Применяем правило резолюции к 1-му и 6-му предложениям и добавляем их к множеству предложений:

S = {¬ П(x) ∨ Р(x), ¬С(x)∨¬Р(x), С(a), У(a), ¬У(x) ∨П(x), П(a), Р(a)}

4. Применяем правило резолюции к 7-му и 2-му предложениям и добавляем их к множеству предложений:

S = {¬ П(x) ∨ Р(x), ¬С(x)∨¬Р(x), С(a), У(a), ¬У(x) ∨П(x), П(a), Р(a), ¬ С(a)}

5. Применяем правило резолюции к 8-му и 3-му предложениям и добавляем их к множеству предложений:

S = {¬ П(x) ∨ Р(x), ¬С(x)∨¬Р(x), С(a), У(a), ¬У(x) ∨П(x), П(a), Р(a), ¬ С(a), □}

Мы получили пустое предложение, следовательно, получено опровержение отрицания целевой формулы, т. е. целевая формула доказана.

Обычно вывод по методу резолюций записывают в сокращенной форме: предложения нумеруют, множество выписывают в виде последовательности добавляемых резольвент, а для каждой резольвенты указывают номера резольвируемых предложений. Например, в сокращенной форме построенный вывод выглядит следующим образом:

1. ¬ П(x) ∨ Р(x)
2. ¬С(x)∨¬Р(x)
3. С(a)
4. У(a)
5. ¬У(x) ∨П(x)
6. П(a) (4, 5)
7. Р(a) (1, 6)
8. ¬С(a) (2, 7)
9. □ (3, 8)

В последующих примерах выводы методом резолюций записываются в сокращенной форме.

4.2. СТРАТЕГИИ ПОИСКА ОПРОВЕРЖЕНИЯ МЕТОДОМ РЕЗОЛЮЦИЙ

Один из самых важных вопросов, касающихся метода резолюций — откуда взять функцию `Select`, которая выбирает пару резольвирумых предложений? Это ключевой вопрос. *Удачно выбранная функция может быстро привести к цели. Неудачная будет порождать все новые и новые непустые резольвенты, не давая решения.*

*Способ выбора предложений для резольвирования называется **стратегией метода резолюций**.*

4.2.1. Полные стратегии

Ключевое понятие, относящееся к стратегиям — это идея полноты. К понятию «*полноты*» — в разных её ипостасях — мы возвращаемся постоянно и делаем это совершенно осознанно. Мы абсолютно уверены в том, что использование того или иного инструмента (в любой области человеческой деятельности и в программировании — в том числе) всегда должно сопровождаться чётким пониманием всех возможностей и ограничений, свойственных этому инструменту. Мы очень рассчитываем на то, что после внимательного ознакомления с этой книгой читатель сможет использовать замечательные результаты, полученные в области обработки знаний, не как «волшебную палочку», но с пониманием всех тонкостей функционирования инструмента и возможностью, в случае необходимости, внесения новых еще более замечательных усовершенствований.

Полная стратегия *— это такая стратегия поиска опровержения, про которую известно, что если существует какое-либо опровержение по методу резолюции, то существует и опровержение по данной стратегии.*

Рассмотрим две известнейшие полные стратегии.

1. *Стратегия поиска в ширину* (*полный перебор*).

В этом случае на каждом шаге строятся *все* возможные резольвенты (стратегия очевидно полна, так как любой вывод будет содержаться в полном переборе).

Рассмотрим полный перебор на примере задачи о программистах и студентах, которые знают или не знают метод резолюций (см. параграф 4.1.7).

$\neg$П(x)$\vee$Р(x)	$\neg$С(x)$\vee\neg$Р(x)	С(a)	У(a)	$\neg$У(x)$\vee$П(x)
$\neg$П(x) $\vee\neg$С(x)	Р(x) $\vee\neg$У(x)	$\neg$Р(a)	П(a)	
$\neg$П(a)	$\neg$С(x) $\vee\neg$У(x)	Р(a)	$\neg$С(a) $\vee\neg$П(a)	$\neg$У(a)
$\neg$С(a)	□			

Первая строчка — это исходные пять предложений. Напомним, это предложения, которые получились путем преобразования в форму предложений условий задачи и отрицания целевой теоремы.

Во второй строчке написан полный перебор всех предложений, которые получаются путем резольвирования первой строчки (исходных предложений). В третьей строчке написаны все новые предложения, которые получаются путем резольвирования тех предложений, которые написаны в первых двух строчках. В четвертой строчке появится пустой квадратик при резольвировании У(a) из первой строки и $\neg$У(a) из третьей строки.

В результате после третьего пополнения исходного множества предложений будет построено пустое предложение и закончен вывод, но перед этим будет построено довольно много предложений, которые в выводе не участвуют, они, безусловно, лишние. Это и есть стратегия полного перебора. Она не эффективна, как и большинство полных стратегий, однако всегда заканчивается положительным для нас результатом — мы получаем ответ, если он существует.

2. Линейная стратегия.

В случае линейной стратегии на каждом шаге, кроме первого, одно из резольвируемых предложений — это последнее полученное

предложение, последняя резольвента. Самое первое предложение называется верхним. Предложение, полученное из предыдущего шага (на первом шаге это верхнее предложение, на последующих шагах — это последняя полученная резольвента), называется *центральным*. Второе резольвируемое предложение называется *боковым*. Таким образом, боковое предложение — это либо исходное предложение, либо одно из центральных, полученных ранее.

Линейная резолюция выглядит для той же задачи более эффективной, чем полный перебор. Выберем в качестве верхнего предложения первое предложение из исходного множества предложений

¬П(x)∨Р(x)   ¬С(x)∨¬Р(x)   С(a) У(a) ¬У(x)∨П(x).

В качестве бокового предложения каждый раз будем выбирать первое подходящее, записывая вывод в виде последовательности пар предложений: слева центральное, справа — боковое.

¬П(x)∨Р(x)	¬С(x)∨¬Р(x)		
¬П(x)∨¬С(x)	С(a)		
¬П(a)	¬П(x)∨Р(x)	¬П(a)	¬У(x) or П(x)
Р(a)	¬С(x)∨¬Р(x)	У(a)	
	¬У(a)		
¬С(a)	С(a)	□	

Здесь выписаны все предложения, которые получаются по ходу решения. Их несколько меньше (11), чем в случае полного перебора (16). Но в этом случае пустое предложение появится только на шестом уровне пополнения. Однако это ничего не значит. В другом случае полный перебор может оказаться эффективнее.

Естественно, есть и другие примеры полных стратегий, которые широко представлены в литературе [6].

4.2.2. Неполные стратегии

Неполная стратегия — *это стратегия, не обладающая свойством полноты.*

Если алгоритм метода резолюций с неполной стратегией не заканчивает свою работу, то это означает не то, что доказательства не существует, а то, что с помощью выбранной стратегии его найти не удается.

Тем не менее, такие стратегии достаточно часто применяются, потому что неполные стратегии зачастую являются более эффективными, чем полные стратегии. Действительно, неполные стратегии могут не получить доказательства, даже если оно существует, но зато, когда получают, они срабатывают быстрее.

Кроме того, неполная в общем случае стратегия может оказаться полной, если мы ограничим исходный класс предложений.

Рассмотрим три примера.

1. *Единичная резолюция* (*поиск от данных*).

Это стратегия, в которой одно из резольвируемых предложений является положительным литералом (фактом).

2. *Отрицательная резолюция* (*поиск от цели*).

Это стратегия, в которой одно из резольвируемых предложений является отрицательным литералом (целью).

3. *Входная резолюция.*

Это стратегия, в которой одно из резольвируемых предложений выбирается из входных, т. е. заданных первоначально предложений.

В общем случае все указанные стратегии неполны. В частности, входная стратегия, которая является частным случаем полной линейной стратегии, сама не является полной.

Пример, демонстрирующий неполноту входной стратегии:

1) $P \vee Q$

2) $P \vee \neg Q$

3) $\neg P \vee Q$

4) $\neg P \vee \neg Q$

5) P (1, 2)

6) Q (5, 3)

7) ¬ P (6, 4)

8) □ (7, 5)

Обратим внимание на то, что вывод построен, но он не является выводом по входной стратегии — дизъюнкт P не является входным. Легко видеть, что для множества предложений 1–4 нет опровержения по входной стратегии.

4.2.3. Хорновские предложения

Существует подкласс предложений, который очень важен с точки зрения практического применения метода резолюций. Давайте попробуем синтаксически описать общие свойства следующих предложений:

A	Факты
$A_1 \& \ldots \& A_N \to B$	Правила
B	Цели

После перевода в форму предложений эти формулы имеют такой вид:

A

$\neg A_1 \vee \ldots \neg A_N \vee B$

$\neg B$

Хорн в свое время заметил, что три эти формулы в форме предложений обладают следующей синтаксической характеристикой: в них содержится не более одного положительного литерала. Выше приведены все три возможных случая: один положительный литерал (факт), один положительный литерал и несколько отрицательных (правило), нет положительных литералов (одни отрицательные — цель).

Предложения, содержащие не более одного положительного литерала, называются ***хорновскими***.

Кроме этого, множество таких предложений (хорновских формул) замкнуто относительно правила резолюции. Если мы

резольвируем хорновские предложения, мы снова получаем хорновское предложение, потому что мы вычеркиваем один положительный и один отрицательный литерал, в сумме в резольвенте оказывается не более одного положительного литерала.

Подход для применения метода резолюций в программировании был разработан не Хорном, это было сделано позднее Робертом Ковальским. Оказалось, что метод поиска от целей и метод поиска от данных, которые не являются полными в общем случае, полны в хорновском случае. Если существует какой-либо вывод, то существует вывод от целей для хорновского случая.

На основе хорновского случая был разработан язык Пролог, который иногда называют *логическим программированием*. Однако понятия Пролога и логического программирования не следует отождествлять. Язык Пролог — это частный случай логического программирования, а именно, это реализация метода резолюций для хорновского случая со стратегией «поиск от целей». Можно использовать другие подходы: либо применить другую стратегию, либо взять другой подкласс формул, либо вообще рассматривать исчисление не первого порядка, — и все это будет логическое программирование, но не будет являться Прологом.

4.2.4. Замечания по реализации

Разработать прямолинейную систему логического программирования или систему автоматического доказательства теорем достаточно легко. Однако все такие наивные реализации оказываются практически бесполезными. Это происходит из-за того, что не учитывается несколько важных факторов:

1. Важен не выбор стратегии резольвирования, важно качество процедуры унификации. Процедура унификации проверяет, унифицируемы ли два литерала, и если они унифицируемы, выдает подстановку, которая их унифицирует. Более 90 % машинных ресурсов системы логического программирования уходит на

унификацию. Приведенная в разделе 4.1.7 процедура унификации совсем не эффективна.

2. Большинство программистов пытается написать программу унификации, которая как можно скорее выдает ответ — набор подстановок. А в данном случае (для унификации) требуется создать программу, которая как можно скорее *не* получала бы ответ, потому что большинство литералов *не* унифицируемы. Нужно как можно скорее выдать **fail**, оборвать выполнение процедуры, не нужно вычислять ответ. Его, скорее всего, вычислять не придется. Это требует тонких программистских хитростей.

3. Большинство операций и отношений, которые встречаются в реальных задачах, обладают специальными свойствами: коммутативностью, транзитивностью и т. д. Если их не использовать, то окажется, что не унифицируемо то, что на самом деле, по существу, унифицируемо. А если их использовать на логическом уровне, то количество аксиом увеличивается в несколько раз. Конечно, учет таких вещей, как транзитивность и коммутативность, нужно делать на уровне унификации термов, а не на уровне логических аксиом.

4.3. ИЗВЛЕЧЕНИЕ РЕЗУЛЬТАТА

Метод резолюций строит доказательство теоремы. С помощью специальных приемов из доказательства можно извлекать ответы (знания) любого типа. Специальные приемы нужны по следующей причине. Метод резолюций работает только с предложениями, но пользователь ставит реальную задачу не в форме предложений, а в той форме, в которой ему удобно. Перед применением метода резолюций приходится преобразовывать задачу в форму предложений. Выдавать в качестве ответа голое доказательство не очень лояльно по отношению к пользователю Нужно извлечь из доказательства то, что ожидает пользователь в качестве ответа.

4.3.1. Извлечение результата (да/нет)

Стратегии метода резолюции, которые мы рассмотрели, пока давали ответ на вопрос — выводимо или не выводимо решение, то есть, получался бинарный ответ на вопрос — да или нет.

Пример

Рассмотрим следующие утверждения.

1) Все студенты группы 5057 знают метод резолюций.

Власенко[71] студент группы 5057.

Знает ли Власенко метод резолюций?

При переводе в форму предложений и резольвировании получаем следующий вывод:

1) $\neg$ 5057(x) $\vee$ P(x)

2) 5057(Власенко)

3) $\neg$P(Власенко)}

4) P(Власенко)}   (1, 2)

5) □   (3, 4)

Ответ — **да**, поскольку мы получили пустое предложение.

2) Все студенты группы 5057 знают метод резолюций.

Халепский не знает метод резолюций.

Является ли Халепский студентом группы 5057?

1) $\neg$ 5057(x) $\vee$ P(x)

2) $\neg$ P(Халепский)

3) $\neg$ 5057(Халепский)

4) $\neg$ 5057(Халепский) (1, 2) — совпадает с 3

Ответ — **нет**, поскольку нет резольвируемых предложений и нет пустого предложения.

---

[71] Власенко, Халепский и Королев — фамилии реальных студентов, которые были среди первых слушателей учебного курса, положенного в основу этой книги. Их присутствие (или отсутствие) в аудитории оказало заметное влияние на подбор примеров и стиль аргументации, и в знак признательности мы решились оставить в тексте реальные фамилии.

Это и есть извлечение результата. С помощью выбранной стратегии мы смогли получить ответ на поставленный вопрос.

Может показаться, что ответ «да–нет» — это маргинальный случай, рассматриваемый только в учебных целях. На самом деле это далеко не так. Вот несколько примеров, экспертных систем, которые выдают ответ простой «да–нет», являясь при этом и вполне реальными и весьма сложными.

1) Мониторинг окружающей среды. Получены такие-то данные наблюдений. Имеется ли угроза, которая требует эвакуации населения?

2) Система противоракетной обороны. Засечен движущийся объект. Является ли этот объект вражеской ракетой?

3) Геологоразведочная система. Собраны косвенные данные о некотором районе. Имеются ли в этом районе нефтеносные пласты?

4.3.2. Извлечение результатов (факты)

А теперь попробуем поставить вопрос так, чтобы получить ответ не в форме да-нет, а в форме факта. Для этого нужно модифицировать метод резолюций введением специального предиката — традиционно называемого предикатом ANS[72]. Предикат ANS — это предикат, в котором накапливается значение ответа. Считается, что этот предикат эквивалентен пустому предложению. Если есть предложение, содержащее только предикаты ANS, то это пустое предложение.

Пример:

Все студенты группы 5057 знают метод резолюций.

Власенко студент группы 5057.

Королев студент группы 5057.

Кто знает метод резолюций?

Теперь целевое предложение P(x) выглядит так: если P(x), то ANS(x), то есть если x знает метод резолюций, то x является ответом.

---

[72] От английского Answer — ответ.

В формальной записи в форме предложений вопрос имеет вид $\neg P(x) \vee ANS(x)$. Напоминаем, что с точки зрения управления выводом ANS = □.

Мы проводим резолютивный вывод и получаем 6 предложений, последнее из которых ANS(Власенко), то есть, получаем ответ:

1) $\neg$ 5057(x) $\vee$ P(x)

2) 5057(Власенко)

3) 5057(Королев)

4) $\neg$ P(x) $\vee$ ANS(x)

5) $\neg$ 5057(x) $\vee$ ANS(x) (1, 4)

6) ANS(Власенко) (5, 2)

Можно остановиться, а можно продолжать построение новых резолютивных выводов, как и делают системы логического программирования, основанные на языке Пролог. Тогда получатся все ответы, которые можно получить из условий задачи.

7) ANS(Королев) (5, 3)

4.3.3. Извлечение результатов (термы)

Используя технику предикатов ANS, можно получать не только факты, но и результаты процедурного уровня. Основываясь на этом, можно синтезировать выражение (терм), вычисляющее некоторую функцию.

Пример:

Имеется отношение отец и дед. Аксиомы, описывающие эту ситуацию, заключаются в следующем: у любого человека есть отец, и отец отца — это дед.

$O(x, y) = x$ отец y

D(x, y) = x дед y

$\forall x, y, z\ O(x, y) \ \&\ O(y, z) \rightarrow D(x, z)$

$\forall y \exists x\ O(x, y)$.

Дальше ставим вопрос — кто является дедом x?

? $D(y, x) \rightarrow ANS(y)$

В результате применения метода резолюций получается вывод:

1) $\neg O(x, y) \vee \neg O(y, z) \vee D(x, z)$

2) $O(f(y), y)$

3) $\neg D(y, x) \vee ANS(y)$

4) $\neg O(y, z) \vee D(f(y), z)$ $(1, 2, f(y)//x)$

5) $D(f(f(y)), y)$ $(4, 2, f(y)//y,\ y//z)$

6) $ANS(f(f(x)))$ $(3, 5)$

где f — это функция вычисления отца, которая была получена при сколемизации (элиминации существования) второй аксиомы этого примера.

Получен ответ — дедом x является $f(f(x))$.

Таким образом, метод резолюций с точки зрения представления знаний является вполне адекватным средством. Он позволяет получать содержательные ответы на содержательные вопросы.

4.4. СИСТЕМЫ ДЕДУКЦИИ НА ОСНОВЕ ПРАВИЛ

В данном разделе мы попробуем на конкретном примере рассмотреть *совместно* различные способы представления знаний, которые мы разобрали в предыдущих главах, то есть представления знаний на основе системы продукции и представления знаний на основе исчисления предикатов первого порядка. На самом деле между ними нет никакой непреодолимой границы. Это все сводимые друг к другу вариации на одну и ту же тему, что дает повод считать, что действительно есть предмет «представление знаний». Если бы это были абсолютно разрозненные вещи, то было бы совершенно непонятно, почему они объединяются в рамках одного курса. Мы сейчас легко убедимся, что они совсем не разрозненные[73].

Сначала мы разберем, что же мотивирует нас объединить разные системы представления знаний. Затем еще раз с других

---

[73] При этом за рамками данной книги остается аналогичная ситуация, которая имеет место и в других случаях — например, если мы используем семантические сети или фреймы.

позиций рассмотрим форму И/ИЛИ, которая уже дважды встречалась в данном учебнике под разными названиями: первое название было «разложимые системы продукции», а второе — «поиск на игровых деревьях». Мы еще раз рассмотрим этот аспект представления знаний и на трех примерах попытаемся показать, что во всем этом есть общее содержание и глубокие внутренние взаимосвязи между приемами и техниками, которые здесь используются. В конце этой главы продемонстрированы три варианта конкретной техники, которая называется «система дедукции на основе правил».

4.4.1. Потеря импликативности

Разберем недостатки, присущие методу резолюций. У этого метода есть две существенные имманентно присущие ему особенности, которые могут рассматриваться в некоторых случаях как недостатки.

Первая особенность состоит в том, что метод резолюции работает с приложениями — с бескванторными дизъюнкциями литералов. В этих дизъюнкциях нет направления. На самом деле то, в каком порядке буквы находятся в предложении, очень важно, потому что все алгоритмы линейно просматривают информацию и анализируют ее в том порядке, в котором буквы попадаются в формулах. Этот порядок может быть важен.

Если сравнить метод резолюций с привычным методом доказательства на основе правила modus ponens, то в самом этом правиле заложено явное направление рассуждений. Правило modus ponens выглядит следующим образом:

$$\frac{A, \quad A \to B}{B}$$

Понятно, что в этом случае нужно начинать с A, и тогда мы придем к B. Есть явное направление рассуждений, заданное самой стрелкой. Но когда мы элиминируем стрелки импликаций при

сведении формул к предложениям, это направление может быть потеряно.

Пример

$(\neg A \wedge \neg B) \to C$ $\quad\quad\quad\quad$ $A \vee B \vee C$

$(\neg B \wedge \neg C) \to A$ $\quad\quad\quad\quad$ $A \vee B \vee C$

Выше приведены два совершенно разных импликативных правила, которые отражали разную семантику, а после преобразования к предложениям эта разница исчезла. Метод резолюции не знает, с чего нужно начинать и куда нужно двигаться. У него все буквы одинаковы, он их просто механически перебирает, пытаясь вывести пустое предложение, но в какую сторону надо двигаться, чтобы вывести пустое предложение — этой информации нет, она утрачена, хотя, возможно, присутствовала в исходной постановке задачи.

Постановки задачи для системы автоматического доказательства — это дело эксперта, который наполняет базу знаний. Он-то знает, в какую сторону нужно использовать импликацию, и мог бы подсказать системе. Например, эксперт может знать, что вместо прямого правила $\neg A \to B$ следует использовать контрапозитивную форму $\neg B \to A$. Но при сведении к предложениям подсказки забываются. Метод резолюций видит только формулу $A \vee B$. Это первый недостаток метода резолюций. Называется это свойство — *потеря импликативности*.

4.4.2. Размножение литералов

Второй недостаток данного метода заключается в следующем. При преобразовании в предложения мы используем конъюнктивную форму, и при этом формула может расширяться, то есть общее количество вхождений предикатов в формулы может увеличиваться. Другими словами, может произойти так называемое *размножение литералов*. В методе резолюций может оказаться (и в худшем случае

оказывается), что коэффициент расширения, который здесь наблюдается, носит экспоненциальный характер.

Пример:

Власенко отлично учится в группе 5057.

Все, кто учится в данной группе, являются студентами, все студенты знают метод резолюций.

Отличники умные.

Попробуем показать следующий факт: существует студент, который знает метод резолюций и является либо программистом, либо умником.

Введем соответствующие предикаты и запишем формулы.

0) $\exists x$ Р(x) $\wedge$ (П(x) $\vee$ У(x)) цель

1) 5057(Власенко) факт

2) О(Власенко) факт

3) $\forall x$ 5057(x) $\rightarrow$ С(x) правило

4) $\forall x$ С(x) $\rightarrow$ Р(x) правило

5) $\forall x$ О(x) $\rightarrow$ У(x) правило

Преобразуем формулы в предложения. По правилу метода резолюций к цели спереди надо приписать отрицание. При этом приписывании первое утверждение разделяется на два, и предикат Р, который в исходной постановке присутствовал один раз, после преобразования появляется два раза. А теперь начинаем резольвирование.

1) $\neg$Р(x) $\vee$ $\neg$П(x)

2) $\neg$Р(x) $\vee$ $\neg$У(x)

3) $\neg$С(x) $\vee$ Р(x)

4) $\neg$5057(x)$\vee$ С(x)

5) 5057(Власенко)

6) $\neg$(x)О $\vee$ У(x)

7) О(Власенко)

8) $\neg$П(x) $\vee$ $\neg$С(x) (1, 3)

9) $\neg$П(x) $\vee$ 5057(x) (8, 4)

10) $\neg$П(x) (9, 5)
11) $\neg$У(x) $\vee$ $\neg$С(x) (2, 3)
12) $\neg$У(x) $\vee$ $\neg$5057(x) (11, 4)
13) $\neg$У(Власенко) (12, 5)
14) $\neg$О(Власенко) (13, 6)
15) $\square$ (14, 7)

Пункты 8, 9 и 10 — это результативные выводы, резольвенты, полученные самым простым образом. Берем два первых подходящих предложения и начинаем резольвировать, 8, 9 и 10 пункты в этом примере — это первые подходящие, и легко сообразить, что полученное $\neg$П(x) нам совсем не нужно. Этот результат потом негде будет использовать. Программисты вообще в исходных данных не упоминались. Это бессмысленно потраченная машинное время и память на выполнение вывода, который не нужен. Начинать следовало сразу с пункта 11 (резольвируя предложения 2 и 3), первое предложение не надо было рассматривать. Атомарная формула $\neg$Р(x), которая появилась в предложении 1, произошла в результате размножения литералов и, безусловно, оказалась вредна для решения задачи.

Вывод построен, но по дороге был ложный ход, который вызван тем, что произошло размножение литералов, вытекающее из преобразования в конъюнктивную нормальную форму. В результате количество вхождений литералов в предложения увеличилось, а вывод удлинился. В худшем случае такое увеличение может носить экспоненциальный характер.

4.4.3. Естественное направление дедукции

Посмотрим на ту же задачу с другой стороны. Не будем сводить исходные формулы к предложениям, и применять правило резолюции, а применим обычное правило modus ponens. Не станем использовать доказательство от противного, применим прямую

систему дедукции. В результате все получается гораздо лучше, проще и быстрее:

1) $\exists x$ Р(x) $\wedge$ (П(x) $\vee$ У(x)) цель
2) 5057(Власенко) факт
3) О(Власенко) факт
4) $\forall x$ 5057(x) $\rightarrow$ С(x) правило
5) $\forall x$ С(x) $\rightarrow$ Р(x) правило
6) $\forall x$ О(x) $\rightarrow$ У(x) правило
7) С(Власенко) Modus ponens 1, 3
8) Р(Власенко) Modus ponens 6, 4
9) У(Власенко) Modus ponens 2, 5
10) П(Власенко) $\vee$ У(Власенко) правило $\vee$+, 8
11) Р(Власенко) $\wedge$ (П(Власенко) $\vee$ У(Власенко)) правило &+,7,9
12) $\exists x$ Р(x) $\wedge$ (П(x) $\vee$ У(x)) правило $\exists$+, 10

Предложения 1 и 3 по Modus ponens дают С (Власенко), 6 и 4 дают Р (Власенко), 2 и 5 дают У(Власенко), и, применяя правила, аналога которым в методе резолюций нет, но которые, очевидно, состоятельны, получаем цель. Мы видим, что прямая система (обычное рассуждение, а не метод доказательства от противного) — простое рассуждение по правилу Modus ponens, и введение дополнительных связок гораздо быстрее, чем в первом случае, позволяет получить тот же самый ответ.

Вот в этом и состоит основная идея. Мы хотим сохранить преимущества, которые нам дает представление знаний с помощью предложений, и избавиться от недостатков, которое дает это представление. Мы хотим избавиться от размножения литералов и сохранить направление импликации, чтобы иметь возможность проводить такие эффективные и короткие выводы, как во втором случае, по сравнению с неэффективными и длинными выводами в первом случае.

4.4.4. Форма И/ИЛИ

Попробуем преобразовать исходные формулы, с помощью которых записано наше знание о предметной области (формулы исчисления первого порядка) не в форму предложений, а в так называемую *форму И/ИЛИ*, просто проводя меньшее количество шагов преобразования. Так же, как при сведении к предложениям в методе резолюций элиминируем импликацию, протаскиваем отрицание, элиминируем квантор существования, строим предваренную форму, убираем квантор всеобщности и разделяем переменные в главных конъюнктах (см. параграф 4.1.3).

Если вы обратили внимание, пропущен этап построения конъюнктивной формы. Какая форма была, такая форма и осталась, мы только избавились от кванторов и разделили переменные. Легко видеть, что в результате элиминации импликации, протаскивания отрицания, элиминации квантора существования, построения предваренной формы, элиминации всеобщности и переименования переменных, количество литералов сохраняется.

Заметим, что *форма И/ИЛИ может быть получена по любой формуле исчисления предикатов*.

Пример:

Дана формула (рис. 4.1):

$$\exists u\ \forall v\ (\mathrm{Q}(v, u) \wedge \neg\ ((R(v) \vee P(v)) \wedge S(u, v)).$$

Для использования формы И/ИЛИ предлагается представить формулу в виде гиперграфа следующим образом. Там, где главная операция $\wedge$, ставится две раздельных 1-дуги, там, где главная операция $\vee$, ставится одна 2-дуга. Ход рассуждения достаточно прост. Когда известно, что $A \wedge B$, можно повести рассуждение в одну сторону, используя то, что дано A, а можно в другую, используя то, что дано B. Когда дано $A \vee B,$ мы не знаем, в какую сторону конкретно двигаться, нужно двигаться сразу в обе стороны.

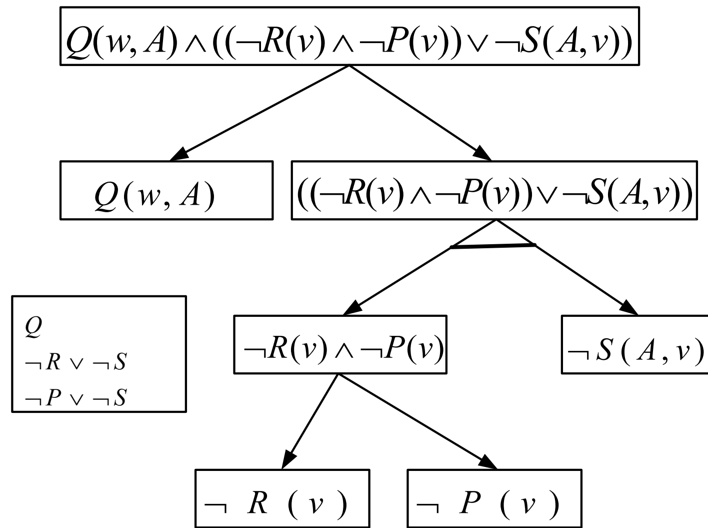

Рис. 4.1. Представления формы И/ИЛИ в виде гиперграфа

Очевидно, что множество предложений, в которые преобразуется корневая формула, может быть прочитано, как множество концевых вершин всех графов решения, которые есть на этом гиперграфе (рис. 4.1). На этом гиперграфе есть три графа решения. Один идет из корня в Q, второй идет из корня на следующий уровень и дальше обязательно завязывается на S, потому что там 2-дуга. Нас интересуют концевые вершины. Если исходную форму свести к предложениям, то их получится три: Q, $\neg R \vee \neg S$, $\neg P \vee \neg S$.

Заметим это обстоятельство, форма И/ИЛИ хранит ту же информацию, которой мы располагали в методе резолюций, а также еще и другую информацию.

Рассмотрим следующий способ построения машины доказательства — ядра интеллектуальной системы логического вывода в форме системы дедукции на основе правил.

4.4.5. Прямая система дедукции

Так же, как это принято в логическом программировании, мы должны разделить все множество формул, всю информацию, которая у нас есть в данной предметной области на три класса: факты, правила и цели.

При этом накладываются следующие синтаксические ограничения:

- то, что мы будем считать фактами, на самом деле ничем не ограничено и может быть представлено в любой форме И/ИЛИ;
- то, что мы будем считать правилами, обязательно должно иметь вид $L \to W$, где L — литерал, а W — любая формула, которая будет представлена в форме И/ИЛИ;
- цель должна быть дизъюнкцией литералов, то есть должна быть предложением. Другими словами, можно сформулировать условие остановки: целевая формула является предложением, граф решения заканчивается в целевых литералах.

Смысл процедуры наращивания графа И/ИЛИ с помощью правил прост. Возьмем факт (или множество фактов) и представим их в форме И/ИЛИ. Получится гиперграф. Начнем к листовым узлам этого графа прицеплять правила. Каждый листовой узел построенного графа унифицируется с левой частью правила. Если унификация успешна, мы наращиваем граф (или дерево) И/ИЛИ правой частью правила, пока в этом графе не появится граф решения, листья которого попадают в целевые узлы (то есть унифицируются с целевыми литералами). Когда происходит отождествление левой части правила с листовым узлом в графе, возможно, потребуется выполнение некоторых подстановок для унификации. Понятно, что если в разных графах решения применяются разные правила, то они применяются независимо. Но в одном графе решения невозможно одновременно подставить вместо одной и той же переменной разные термы. Значит, такие случаи должны быть исключены. Фактически, механизм, который сейчас описывается, является частным случаем

метода резолюций, который проводится с помощью перестроения графа.

Рассмотрим это на примере.

Факт: Власенко умный и знает метод резолюций или он не студент:

$\neg$С(В)$\vee$(Р(В)$\wedge$У(В)).

Правило 1: Все учащиеся группы 5057 являются студентами:

$\neg$С(x) $\to$ $\neg$5057(x).

Правило 2: Тот, кто знает метод резолюций, является отличником:

Р(y) $\to$ О(y).

Цель: Существует некто, кто или не учится в группе 5057 или является отличником:

$\exists z$ $\neg$5057(z) $\vee$ О(z).

На примере правила 1 видно преимущество прямой системы на основе правил. Конструктор системы представления знаний для конкретной предметной области это правило – «все учащиеся группы 5057 являются студентами» — может записать сразу в контрапозитивном виде: «если он не студент, то он не учится в группе 5057». Тем самым конструктор системы представления знаний (человек) дает подсказку программе логического вывода: в дедуктивных системах рассуждать выгоднее от общего к частному, а не от частного к общему. Именно это и сделано в правиле 1. Вот почему сохранение импликативности так важно и полезно.

Мы строим прямое дедуктивное рассуждение, а не используем метод доказательства от противного. В отличие от метода резолюций, отрицание к цели применять не следует. Цель должна использоваться в том виде, в котором находится в формулировке задачи. В данном случае видно, что после сколемизации цель будет бескванторной дизъюнкцией литералов, что и требуется в синтаксическом условии прямой системы дедукции.

Ход рассуждения таков (рис. 4.2).

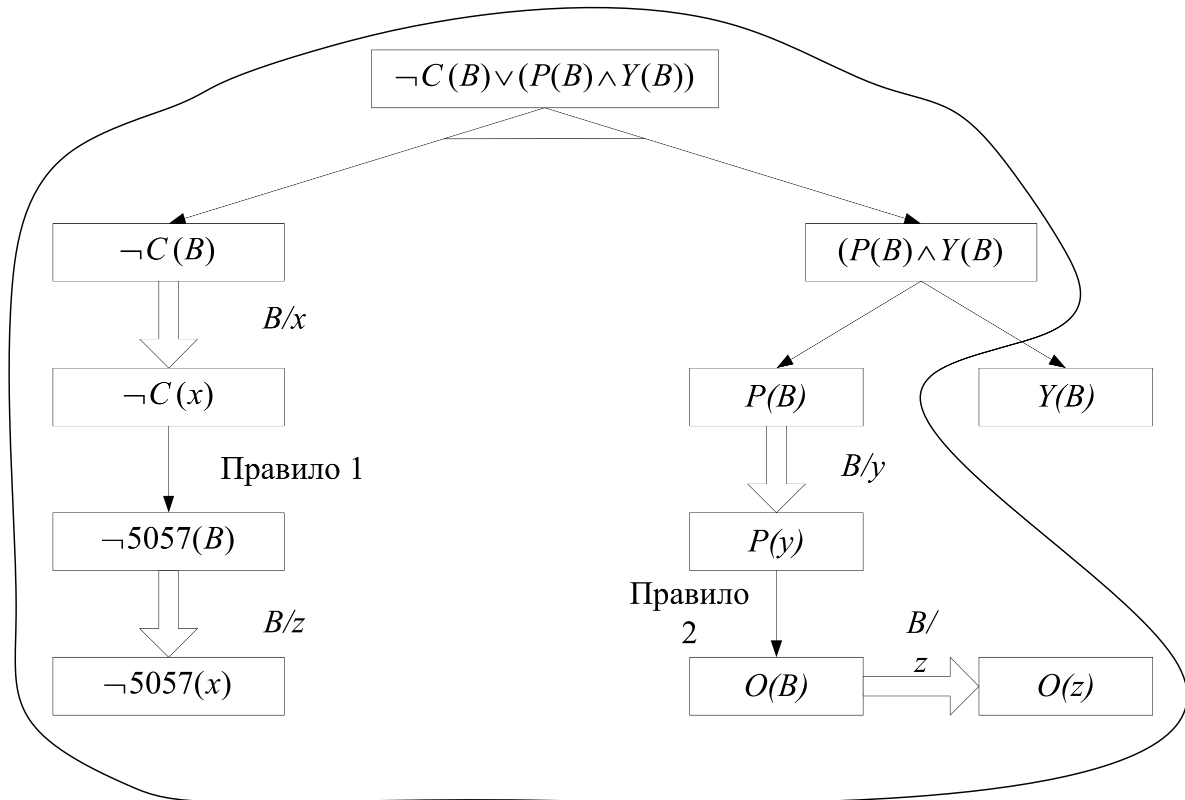

Рис. 4.2. Расширение формы И/ИЛИ применением правил

В начальный момент граф фактов имеет следующий список листовых узлов: ¬С(В), Р(В) и У(В). Используем наше правило 1 для наращивания дерева. Левая часть правила унифицируется с подстановкой «В» вместо *x*. Дерево наращивается правой частью, которая унифицируется с первым целевым литералом. Это еще не решение, поскольку мы не построили граф решения, который заканчивается целевыми литералами. Поэтому продолжаем применять правила. Срабатывает правило 2, и результат отождествляется со вторым целевым литералом.

Построен граф решения, который заканчивается целевыми литералами (на рисунке обведен). Чтобы сохранить информацию о подстановках, мы вводим *дуги соответствия* (на рисунке они изображены толстыми незакрашенными стрелками), соединяющие заменяемые литералы и левые части правил.

Рассмотрим еще один пример.

Дан факт: $P(x) \vee Q(x)$

Правило 1: $P(A) \rightarrow R(A)$

Правило 2: $Q(B) \rightarrow R(B)$, где A и B – конкретные константы.

Цель: $R(A) \vee R(B)$

Заметим, что цель не является логическим следствием фактов. Этот пример объясняет, в чем значения ограничения, говорящего о том, что подстановки на дугах соответствия в одном графе решения должны быть согласованы (рис. 4.3).

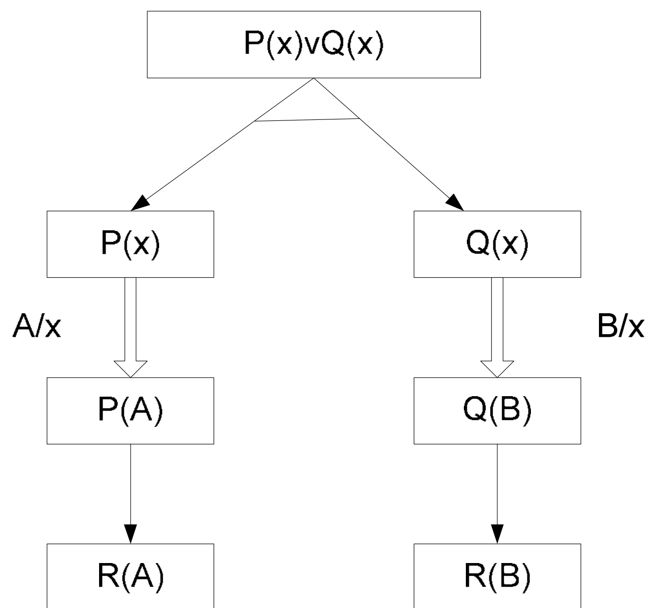

Рис. 4.3. Несогласованные подстановки на дугах соответствия

Видно, что подстановки не совместимы: нужно одновременно подставить А вместо *x* и В вместо *x*. В результате подстановки не согласованы, граф решения не построен, как и должно быть, потому что цель не является логическим следствием фактов и правил.

Разобранный в предыдущем разделе пример был достаточно показательный. Мы провели в технике графов резолютивный вывод, но применимость этого приема ограничена тем, что у нас есть синтаксические ограничения на вид фактов, синтаксические ограничения на вид правил и синтаксические ограничения на вид целей.

4.4.6. Обратная система дедукции

Напомним некоторые определения и факты, известные читателю из курса дискретной математики.

Если $f(x_1, \ldots, x_n)$ *— булева функция, то булева функция*

$$f^*(x_1, \ldots, x_n) = \neg f(\neg x_1, \ldots, \neg x_n)$$

называется ***двойственной булевой функцией***.

Пользуясь тем фактом, что дизъюнкция и конъюнкция двойственны, а отрицание самодвойственно, наряду с методом резолюции можно построить *двойственный метод резолюции*, где элементами, подобными предложениям, будут не дизъюнкции литералов, а конъюнкции литералов. В двойственном методе резолюций сколемизации будет подвергаться не существование, а всеобщность, и будет использоваться не конъюнктивная нормальная форма, а дизъюнктивная нормальная форма (k-связка для конъюнкции, 1-связка для дизъюнкции). Главной операцией будет дизъюнкция. В двойственном методе резолюции мы будем приводить отрицание целевой формулы вместе с исходными к тождественно истинной формуле, которая тоже будет иметь пустую форму, потому, что пустая конъюнкция является тождественно истинной.

Очевидные технические детали этих формальных логических построений мы опускаем, потому что нашей целью в этом разделе является не работа с логическими формулами, а работа с графами И/ИЛИ.

Теперь мы можем построить также синтаксически ограниченную, но уже с другой стороны, систему дедукции на основе правил. Можно идти от фактов к цели, а можно, и иногда это удобнее, идти от целей обратно к фактам по *обратным правилам*. В результате мы видим, что теперь у нас не ограничена форма целей — это любая форма И/ИЛИ. У нас ограничена форма правил, но совершенно другим образом. В правилах теперь правая часть должна быть литеральной, а левая может быть сколь угодно сложной. Кроме этого,

ограничена форма фактов. Это должна быть, конечно, конъюнкция литералов, а не дизъюнкция, как в случае прямой системы.

Условие остановки то же самое: факт является конъюнкцией литералов, согласованный граф решения заканчивается в фактах. Мы можем наращивать исходную цель, представленную в форме И/ИЛИ, с помощью обратных правил, до тех пор, пока она не обопрется на факты.

Рассмотрим пример: построим обратную систему дедукции на основе правил, которая будет решать неравенства.

П1: $x > 0 \mathbin{\&} y > 0 \quad \rightarrow \quad xy > 0$

П2: $x > 0 \mathbin{\&} y > z \quad \rightarrow \quad x + y > z$

П3: $x > w \mathbin{\&} y > z \quad \rightarrow \quad x + y > w + z$

П4: $x > 0 \mathbin{\&} y > z \quad \rightarrow \quad xy > xz$

П5: $1 > w \mathbin{\&} x > 0 \quad \rightarrow \quad x > xw$

П6: $x > (wz + yz) \quad \rightarrow \quad x > (w + y)z$

П7: $x > wy \mathbin{\&} y > 0 \quad \rightarrow \quad x/y > w$

Имеется семь правил. В каждом из них в правой части стоит один литерал, поскольку вообще в задаче рассматривается только один предикат. Слева может быть один предикат или конъюнкция предикатов. Таким образом, эти правила по своей синтаксической форме подходят для применения в обратной системе дедукции.

Имеется конъюнкция фактов, заключающаяся в следующем.

Факты: A, B, C, D > 0 C > D

Целью является такое неочевидное утверждение.

Цель: B(A+C)/D > B

Ход рассуждений представлен на рис. 4.4.

Давайте посмотрим на первое неравенство. Здесь главная операция — деление. При унификации, очевидно, сработает правило 7, других правил с делением в правой части нет. Это первое подходящее правило. Правило 7 имеет вид

$$x > wy \mathbin{\&} y > 0 \rightarrow x/y > w,$$

где x отождествится с B(A + C), при этом y отождествится с D; w отождествится с B. Подставляем правую часть правила. Получаем две новых цели: D > 0 (это дано) и B(A + C) > BD. Здесь будет применено четвертое правило, в котором есть умножение в правой части.

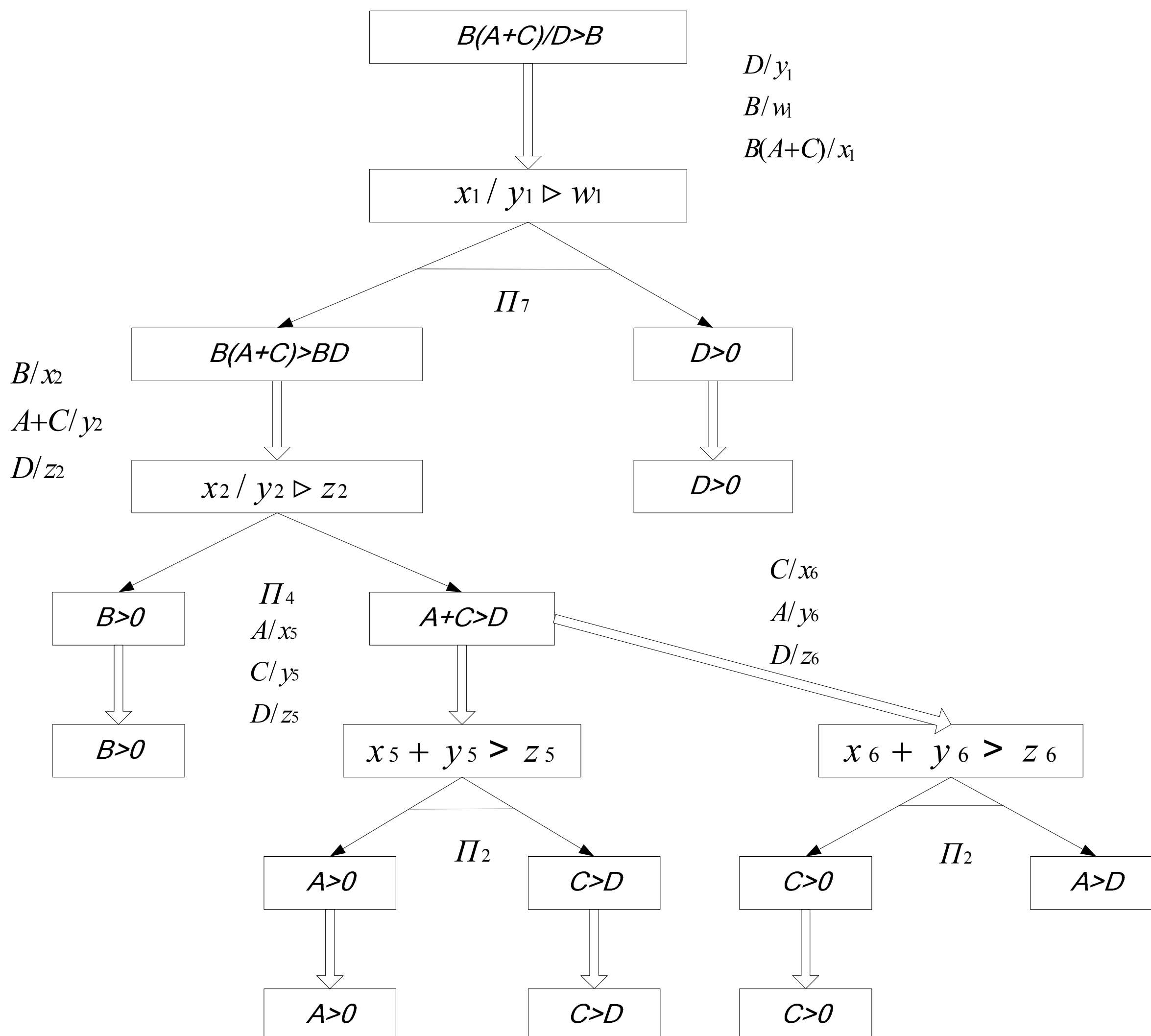

Рис. 4.4. Решение неравенства с помощью обратной системы дедукции

Появляются два подцели: B > 0 и A + C > D. Ко второй части применимы два правила — третье и второе. При этом третье правило сводится ко второму, если w = 0, а у нас всего три переменных в подцели A + C > D, значит, применение любого из этих правил будет равносильным. Проблема состоит в другом — мы по-разному можем

провести унификацию ввиду свойства коммутативности. Первая подстановка ведет к результату, а вторая заводит в тупик, и с этим ничего не сделать. Проверяем, что все подстановки согласованы, значит, построен граф решения. Обратите внимание, что мы использовали каждый экземпляр правил со своими личными переменными, чтобы случайно не оказалось одинаковых переменных под разными кванторами.

4.4.7. Комбинация прямой и обратной систем

Мы рассмотрели один ограниченный случай (прямая система) и другой ограниченный случай (обратная система). Возникает естественная идея: объединить эти два ограниченных случая в надежде получить неограниченный.

В результате объединения прямой и обратной систем дедукции на основе правил получается следующее: имеются факты, представленные в форме И/ИЛИ, имеются цели, представленные в форме И/ИЛИ, и имеются два сорта правил: прямые правила в форме $L \to W$, обратные правила в форме $W \to L$. Они будут навстречу друг другу наращивать графы решения, пока эти графы не столкнутся. Другими словами, мы вводим три вида резолюции:

1. Резолюция фактов и прямых правил, являющаяся частным случаем метода резолюций Робинсона, которую мы уже обсудили;
2. Резолюция целей и обратных правил, которая является частным случаем двойственного метода резолюций;
3. Резолюция фактов и целей.

Последняя называется правилом гашения.

Правило гашения *заключается в следующем: две вершины гасят друг друга, если они, либо унифицируются, либо гасят исходящие из них k-связки.*

Условие окончания выглядит в этом случае так: корень фактов гасит корень целей, и подстановки на дугах соответствия согласованы.

Рассмотрим позитивный пример, чтобы понять преимущества комбинированной прямой и обратной системы (это пример уже рассматривался в разделе 4.1.7):

Факт:	существуют умные студенты
Обратное правило:	программисты знают метод резолюций
Прямое правило:	студенты не знают метода резолюций
Цель:	существуют умные не программисты

Мы сами решаем, какое правило прямое, а какое правило обратное. Это накладывает дополнительную ответственность на того эксперта, который заносит эту информацию в базу знаний, но с другой стороны позволяет то, что не позволяет метод резолюций: сохранить импликативность, показать, в какую сторону надо применять рассуждение.

В результате решение находится достаточно легко (рис. 4.5). На этом рисунке гашение обозначено более жирными стрелками.

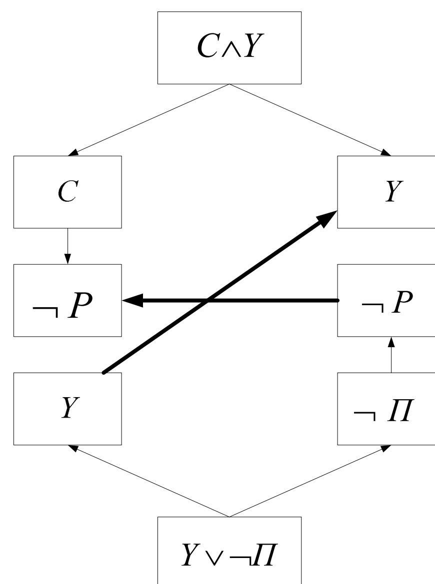

Рис. 4.5. Комбинация прямой и обратной систем дедукции

Одно применение правила для наращивания фактов, одно применение обратного правила для наращивания целей — и сразу же гашение. Другими словами, в отличие от метода резолюций, эта

система сработает за три шага, выполняя всего три унификации. При этом ей не надо думать, какие правила применять, не потребовалось никакой стратегии. Все применимые правила были применены, и в результате получено решение.

К сожалению, в этом достаточно эффектном примере есть сложность. Правило резолюции состоятельно и полно, и правило применения прямых правил как частный случай правила резолюции состоятельно и полно, и правило применения обратных правил как частный случай двойственного метода резолюции состоятельно и полно. Однако с правилом гашения все обстоит не так просто и хорошо. Оно не универсально, т. е. оно состоятельно, но не полно. Существуют такие выводы, которые не могут быть получены по правилу гашения (рис. 4.6).

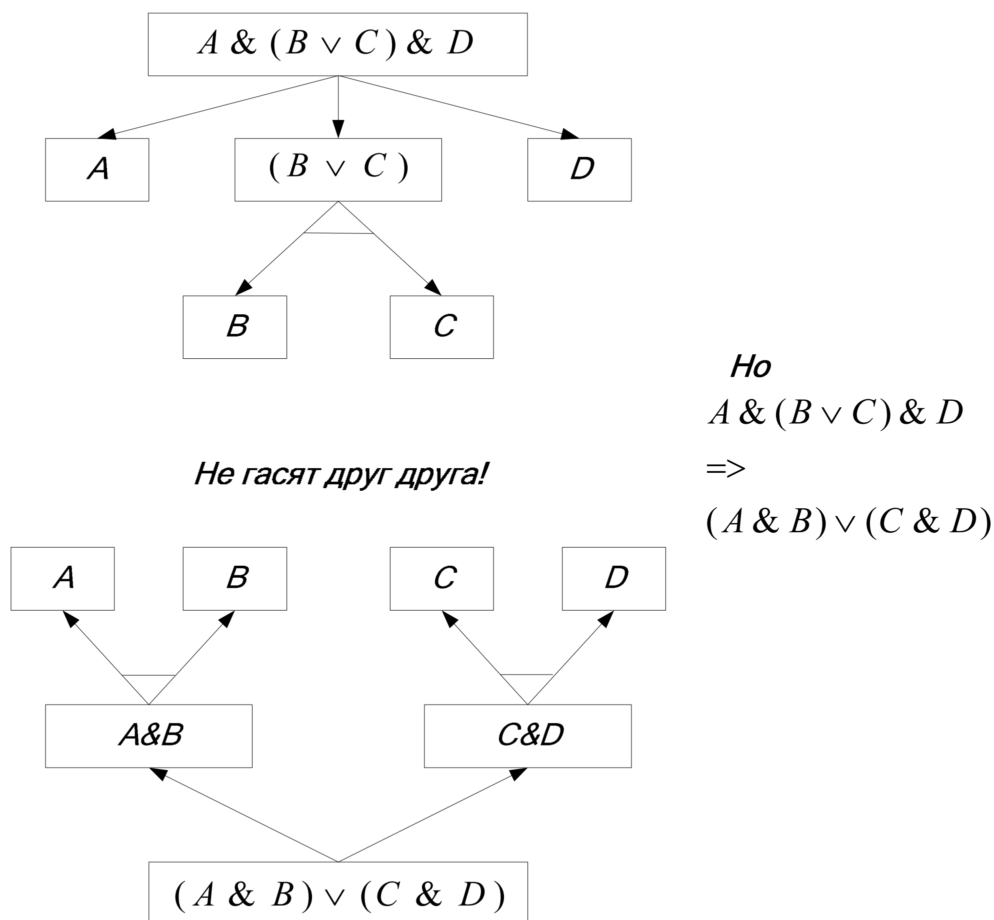

Рис. 4.6. Неполнота правила гашения

Нижняя формула является логическим следствием верхней формулы

$$(A \ \& \ B) \vee (C \ \& \ D) \Rightarrow A \ \& \ (B \vee C) \ \& \ D.$$

Более того, очень хорошо вырастают деревья навстречу друг другу, очень хочется их погасить — буквы одни и те же в обоих случаях. Но этого сделать нельзя, поскольку индуктивное определение следующее: либо унифицируемые, либо гасятся их *k*-связки. А в данном случае *k*-связки как раз и не гасятся. Это разные структуры по количеству дуг: наверху три графа решения, а внизу два.

В данном случае комбинированная система дедукции не сработала, и причина состоит в том, что само представление И/ИЛИ не носит всеобщего характера. Все же что-то мы теряем, когда общую формулу исчисления предикатов переводим в форму И/ИЛИ. Такого замечательного факта, который имеет место в методе резолюций, что после сведения к предложениям, объединение исходных посылок и отрицание теоремы невыполнимо тогда и только тогда, когда целевая теорема следует из посылок, здесь, к сожалению, нет.

Остается открытым вопрос: можно ли подправить форму И/ИЛИ таким образом, чтобы резолюция фактов и целей сделалась состоятельной и полной так же, как резолюция фактов и правил и резолюция правил и целей.

4.4.8. Метазнания в системах дедукции

Системы дедукции на основе правил хорошо подходят для представления фактов, правил и целей и дают почти полную свободу при решении вопроса, что считать фактами, что правилами, а что целями.

Однако есть управляющие знания (метазнания), очень важные для ПСИИ, которые трудно вместить в системы дедукции на основе

правил. Мы приводим три характерных примера метазнаний. Часть из них неявно была применена в примерах ранее.

1. Учет потенциальной ширины ветвления

Допустим, что в системе дедукции есть группа правил классификации. Например,

собака(x) $\to$ животное(x)

кошка(x) $\to$ животное(x)

и так далее, таких правил может быть много. Формально, такие правила можно применять как в прямом, так и в обратном направлении. В каком направлении более целесообразно? Это зависит от предметной области, от типа решаемых системой задач. Знанием о целесообразном направлении применения правил обладает эксперт, который строит систему дедукции. Например, эксперт знает, что правила классификации не следует применять как обратные правила. Дело в том, что применение таких правил в обратном направлении даст очень большую ширину ветвления при разрастании графа целей. Такие правила нужно применять в прямом направлении.

Если эксперт считает, что правила классификации в данной системе дедукции все же нужно применять в обратном направлении, то ему следует переписать их в контрапозитивной форме:

$\neg$животное (x) $\to$ $\neg$собака (x)

$\neg$животное (x) $\to$ $\neg$кошка (x), и так далее.

2. Порядок применения правил

Допустим, что в системе дедукции есть прямые правила вида

$$P \to (Q_1 \vee \ldots \vee Q_k)$$

или обратные правила вида

$$(Q_1 \,\&\, \ldots \,\&\, Q_k) \to P$$

Применение таких правил наращивает граф k-дугами. Большое значение имеет порядок, в котором далее будут рассматриваться узлы

Q_1 & … & Q_k. Например, в обратной системе дедукции на основе правил для решения неравенств (см. параграф 4.4.6, рис. 4.4) после применения правила 4 первым нужно рассматривать выражение B > 0, а не выражение A + C > D. Дело в том, что это выражение *уже* имеет форму факта. Эксперт понимает, что если нам необходим для доказательства какой-то конкретный факт, то возможны два случая: либо этот факт дан в условиях задачи, либо нет. Из воздуха факт не взять. Если факт дан (как в примере 4.4.6), то следует продолжать рассматривать другие выражения, в надежде свести их к данным фактам. Если же факта нет, он не подтверждается, то нужно прекратить рассмотрение других выражений, так как их рассмотрение ничего не даст – данная k-связка все равно не войдет в граф решения.

Такого рода информация является метазнаниями, которые вносятся в дедуктивные системы с помощью специальных приемов. Например, с помощью оценочных функций, как в продукционных системах.

3. Условное применение правил

Рассмотрим еще раз пример с решением неравенств (см. параграф 4.4.6, рис. 4.4). Из соображений транзитивности хотелось бы сказать, что факт C > O — лишний. Возникает соблазн вставить восьмое правило: $x > y$ & $y > z \rightarrow x > z$. Очень опасное правило! Дело в том, что его правая часть применима всегда, и она унифицируема с чем угодно в этой системе. Здесь две простые переменные связаны единственным предикатом, который есть в системе. Какой бы узел ни стоял в графе И/ИЛИ, его всегда можно унифицировать с правой частью правила транзитивности. А это означает, что такое правило будет использоваться постоянно. Можно сказать, что правило транзитивности слишком применимо! Такие правила не стоит использовать непосредственно. Их можно спрятать в унификацию (так обычно и поступают), либо (очень красивое решение) добавить в наш алгоритм метазнание, говорящее о том, что есть правила, которые можно применять всегда, когда они

применимы, а есть правила, которые применяются только при особых условиях. Это некие условия о структуре текущей базы знаний, которая управляет применением или неприменением правил. Метазнание для транзитивности такое: правило транзитивности можно применять тогда и только тогда, когда один из конъюнктов в левой части является фактом. Если это не так, то в результате применения правила транзитивности размножаются цели. Если же это так (если один из конъюнктов, стоящих в левой части правила транзитивности, сопоставим с фактом), то в результате происходит просто передвижение по фактам. При этом дерево решения остается линейным. Дерево не будет ветвиться, а будет просто виться по цепочке транзитивности, что и требуется от применения правила транзитивности. При применении его просто так, порождается y, с которым неизвестно что делать. Можно согласиться на порождение этого y, только если удается взамен избавиться от x.

Вывод из приведенных примеров метазнаний состоит в следующем: метазнания – это не знания о предметной области, для которой строится система дедукции, это знания о знаниях, о том, как применять и использовать знания о предметной области, какими свойствами обладают знания о предметной области и т. д.

Очень хорошие результаты дает двухуровневая система дедукции на основе правил. Такая система на верхнем уровне содержит метазнания, также представленные, например, в форме правил. Фактами в системе верхнего уровня является описание текущего состояния системы нижнего уровня. Целью является построение (синтез) стратегии управления для системы нижнего уровня. На втором уровне находится обычная система дедукции, правила, факты и цели которой берутся из предметной области. Система верхнего уровня определяет для системы нижнего уровня, в каком направлении нужно применять правила, в каком порядке рассматривать узлы, при каких условиях можно применять правила и т. д.

ВЫВОДЫ

1. Исчисления предикатов первого порядка с внелогическими функциями и предикатами достаточно для представления знаний во всех разумных случаях

2. Методы поиска доказательства являются полуразрешающими процедурами, эффективность которых зависит от применяемой стратегии

3. Графы И/ИЛИ и подобные структуры можно использовать для поиска логического вывода так же, как и для систем продукций

ЗАКЛЮЧЕНИЕ

За полвека развития искусственного интеллекта было предложено и проверено на практике достаточно много различных подходов к представлению и обработке знаний. Среди них естественным образом выделяются три группы, три кластера, различающиеся используемым базисом — формальным аппаратом, явно или неявно используемым при реализации конкретных приложений[74]. Эти три группы суть:

- системы продукций, основанные на теории графов;
- логические исчисления, основанные на математической логике;
- другие (например, нейронные и эволюционные системы).

Именно эти три группы методов положены в основу структуры книги. Следует заметить, что все новые, предлагаемые в последнее время подходы к представлению знаний, явным образом тяготеют к одной из этих групп. В то же время между системами продукций и логическим выводом нет непроходимой границы: мы приводим в последней главе несколько примеров сведе́ния методов одной группы к методам другой группы. С другой стороны, другие методы представления знаний, такие как фреймы, семантические сети, нейронные сети и им подобные, по своей природе эклектичны и, в конечном счете, опираются на разобранный в пособии базовый математический аппарат.

Таким образом, знакомство с идеями представления и обработки знаний в рамках выделенных подходов является вполне достаточным для того, чтобы быть в курсе основных тенденций искусственного интеллекта.

---

[74] Это не исчерпывающая таксономия и не закон природы. Так сложились обстоятельства к настоящему времени, и мы исходим из существующего положения вещей.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Вирт Н. Алгоритмы + структуры данных = программы. М.: Мир, 1985.

2. Виноград Т. Программа, понимающая естественный язык. М.: Мир, 1976.

3. В. М. Матросов, С. Н. Васильев, В. Г. Каратуев и др. Алгоритмы вывода теорем метода векторных функций Ляпунова. Ред. В. М. Матросов. Новосибирск: Наука, Сиб. Отделение, 1981.

4. Лавров С. С. Представление и использование знаний в автоматизированных системах // Микропроцессорные средства и системы, 1986, № 3.

5. А. Ахо, Дж. Ульман. Теория синтаксического анализа перевода и компиляции. (в двух томах). М.: Мир, 1978.

6. Чень Ч. и Ли Р. Математическая логика и автоматическое доказательство теорем. М.: Наука, 1983.

7. Дал У., Дейкстра Э., Хоар К. Структурное программирование. М.: Мир, 1975.

8. Энциклопедический словарь Брокгауза и Ефрона. Санкт-Петербург, 1890–1907.

9. Фейс Р. Модальная логика. М.: Наука, 1974.

10. Тейз А., Грибомон П., Ж. Луи и др. Логический подход к искусственному интеллекту. т. 1. От классической логики к логическому программированию. М.: Мир, 1990.

11. Заде Л. Понятие лингвистической переменной и его применение к принятию приближенных решений. М.: Мир, 1976. – 166 с.

12. F. Baader. The Description Logic Handbook. New York: Cambridge University Press, 2003.

13. Нильсен Н. Принципы искусственного интеллекта. М.: Радио и связь, 1985.

14. Davis M., Putnem H. A computing procedure for quantification theory. J. ACM., 1960, 7, No. 3, pp. 201-215.

15. Robinson J. A. Theorem proving on the computer. J. ACM, 1963, 10, No. 2, pp.163-174.

16. Маслов С. Ю. Обратный метод установления выводимости в классическом исчислении предикатов. // ДАН СССР, 1964. вып. 159, № 1. – с. 17-20.

17. Касьянов В. Н., Евстигнеев В. А. Графы в программировании: обработка, визуализация и применение. СПб., БХВ-Петербург, 2003. — 1104 с.

18. Hart P. E., Nilsson N. J., Raphael B. A. Formal Basis for the Heuristic Determination of Minimum Cost Paths // IEEE Transactions on Systems Science and Cybernetics SSC4, 1968. № 2. — С. 100 – 107.

19. Мендельсон Э. Введение в математическую логику. М.: Наука, 1976.

20. Рассел С., Норвиг П. Искусственный интеллект: современный подход. — Киев, Вильямс, 2006.

Новиков Федор Александрович

СИСТЕМЫ ПРЕДСТАВЛЕНИЯ ЗНАНИЙ

Учебное пособие

Лицензия ЛР № 020593 от 07.08.97

Налоговая льгота – Общероссийский классификатор продукции
ОК 005-93, т. 2; 95 3005 – учебная литература

Подписано в печать . .2011. Формат 60×84/16 Печать цифровая
Усл. печ. л. . Уч.-изд. л. . Тираж . Заказ

Отпечатано с готового оригинал-макета, предоставленного автором
в цифровом типографском центре Издательства Политехнического
университета.
195251, Санкт-Петербург, Политехническая ул., 29.
Тел. (812) 540-40-14
Тел./факс: (812) 927-57-76