МЕТОДЫ И СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Конспект лекций и упражнения
для практических занятий.
Очная форма обучения. - 2018

Чмырь И.А.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Существует несколько подходов к изучению и проектированию систем искусственного интеллекта. Эти подходы принято классифицировать следующим образом:

- изучение и проектирование *системы, действующей как человек*;
- изучение и проектирование *системы, "думающей" как человек*;
- изучение и проектирование *системы, "думающей" логично*; и
- изучение и проектирование *системы, действующей рационально*.

Система, действующая как человек. Подход, основанный на тесте Тьюринга.

Понимание системы искусственного интеллекта, как системы, демонстрирующей внешнее "человеческое" поведение чаще всего связывают с *тестом Тьюринга*. Этот тест был предложен английским математиком Аланом Тьюрингом в 1950 году для операционного определения интеллектуальности систем, представленных компьютерными программами. Тьюринг называет интеллектуальной такую систему, которая в процессе диалогового общения с человеком, способна, в течение некоторого времени, вводить его в заблуждение относительно характера своего происхождения. Технология тестирования предполагает, что система искусственного интеллекта выступает в роли отвечающего партнера, а человек общается с ней при помощи телетайпа. Считается, что система искусственного интеллекта прошла тест, если в 30% случаев спрашивающий не в состоянии определить – кто отвечает на его вопросы, человек или компьютерная программа.

С точки зрения современной теории искусственного интеллекта для прохождения теста Тьюринга компьютерная программа должна обладать следующими возможностями:

- *обработкой естественного языка* – для успешной вербальной коммуникации с человеком;
- *представлением знаний* – для хранения знаний, полученных как изначально, до начала диалогового процесса, так и в процессе диалога;
- *автоматическими рассуждениями* – для дедуктивного вывода ответов из хранимой информации;
- *машинным обучением* – для адаптации в процессе диалога, а также для обнаружения и применения диалоговых шаблонов.

Система, "думающая" как человек. Подход, основанный на моделях, полученных в когнитивной психологии.

Понимание системы искусственного интеллекта, демонстрирующей не только внешнее "человеческое" поведение, но и копирующей психологические функции человека чаще всего ассоциируется с моделями, полученными когнитивными психологами. Современная *когнитивная психология* представляет психологические феномены, как системы хранения и переработки информации. Модели, описанные в когнитивной психологии, объясняют различные аспекты человеческой психики, такие как восприятие, внимание, память, принятие решений и др. и могут быть использованы для разработки систем искусственного интеллекта и прикладных программ со встроенным интеллектуальным компонентом.

Система, "думающая" логично. Подход, основанный на законах мышления.

Греческий философ Аристотель был одним из первых, кто попытался разработать законы "правильного мышления" позволяющие строить неопровержимые рассуждения. Предложенные им *силлогизмы* предоставляют собой шаблоны рассуждений, правильное использование которых всегда приводит к правильным заключениям при наличии правильных посылок. Например: "Сократ человек (первая посылка); все люди смертны (вторая посылка); поэтому Сократ смертен (заключение)". Предложенные Аристотелем законы мышления положили начало науки под наименованием *логика***.** Развитие *формальной логики* привело к появлению точной нотации для формального описания утверждений обо всех типах вещей, составляющих мир, и отношениях между ними. В 1960-е и 1970-е

годы был разработан ряд программ, которые при наличии достаточного объёма памяти и времени могли найти решение проблемы, описанной при помощи этой нотации. Логическая традиция в искусственном интеллекте предполагает возможность построения системы искусственного интеллекта на основании нотаций и законов формальной логики.

Известны два основных препятствия в реализации этого подхода. Во-первых, трудно преобразовать неформальные знания в требуемую логическую нотацию, особенно в тех случаях, когда знания не являются полностью определёнными. Во-вторых, существуют значительные трудности при переходе от решения проблемы "в принципе" к реализации процедуры решения на практике. Часто программа, строящая логический вывод даже на небольшом множестве логических предложений потребляет недопустимо большое количество компьютерных ресурсов.

Система, действующая разумно. Подход на основе интеллектуальных агентов.

Действовать разумно означает строить своё поведение таким образом, чтобы достигать поставленную цель в условиях ограничений, задаваемых окружающей средой. *Интеллектуальный агент* – это некоторая система, способная к восприятию окружающей среды и формированию действий в соответствии с внутренней моделью достижения цели. Подход к искусственному интеллекту на основе *интеллектуальных агентов* предполагает изучение и проектирование систем искусственного интеллекта в виде унифицированных систем – агентов, отличающихся различными внутренними моделями целенаправленного поведения.

Изучение искусственного интеллекта, как науки о проектировании интеллектуальных агентов имеет два методологических преимущества. Во-первых, это подход, более общий, чем, например, подход на основе "законов мышления", поскольку логический вывод является только одной из моделей достижения цели. Во-вторых, такой подход относительно легко поддаётся формальному развитию, и, следовательно, практическому внедрению, поскольку опирается на ряд хорошо развитых и формализованных теорий.

Настоящий курс ориентирован на изучение систем искусственного интеллекта как множества различных типов интеллектуальных агентов, отличающихся внутренними моделями достижения цели.

1. ИНТЕЛЛЕКТУАЛЬНЫЙ АГЕНТ И ОКРУЖАЮЩАЯ СРЕДА

Интеллектуальный агент не может функционировать вне окружающей среды. Естественный интеллект возник, в ходе эволюции живых существ, как средство, позволяющее организму более эффективно строить свое поведение в окружающей среде и побеждать в конкурентной борьбе с другими живыми существами.

1.1. Определение интеллектуального агента

Интеллектуальный агент взаимодействует с окружающей средой. Он воспринимает окружающую среду при помощи *сенсоров и воздействует* на среду при помощи *эффекторов.* На рис. 1.1 приведена структура интеллектуального агента взаимодействующего с окружающей средой.

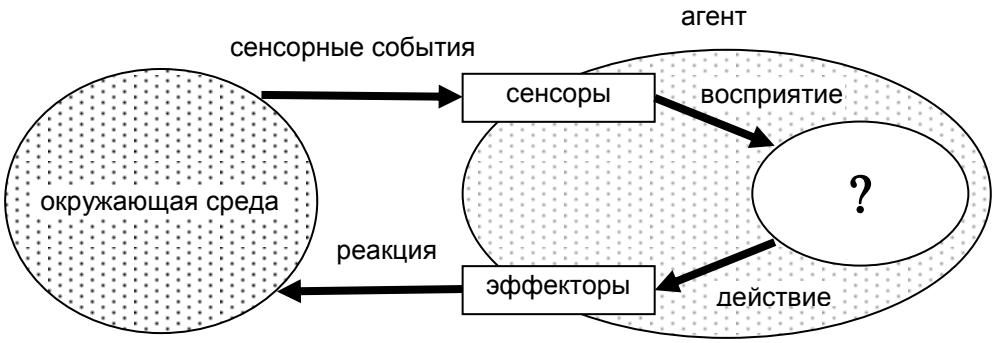

Рис. 1.1. Агент взаимодействует со средой при помощи сенсоров и эффекторов

Человек, рассматриваемый, как интеллектуальный агент, в качестве сенсоров использует глаза, уши и другие органы чувств, а руки, ноги, и другие части тела – в качестве эффекторов.

Агент-робот может использовать в качестве сенсоров видеокамеру и ультразвуковой дальномер, а различные манипуляторы – в качестве эффекторов.

Программный агент, в качестве восприятий может получать коды нажатия клавиш, содержимое файлов и сетевые пакеты, а его воздействие на окружающую среду выражается в передаче информации на периферийные устройства компьютера, записи информации в файлы и передаче сетевых пакетов.

Сенсорная система является первой ступенью системы восприятия. Сенсорная система воспринимает потоки *сенсорных событий* и ставит им в соответствие фрагменты знаний в памяти агента. В *процессе восприятия осуществляется категоризация (классификация) сенсорных событий*. Сенсорное событие формируется сразу несколькими сенсорами. Например, человек-агент может воспринимать автомобиль, как сенсорное событие, в виде совокупности зрительного, звукового и, возможно, обонятельного образов. Структурный элемент, обозначенный на рис. 1.1. вопросительным знаком, используя *воспринятые знания* и *внутренние знания агента*, формирует *действие*, которое при помощи эффекторов трансформируется в *реакцию агента*, направленную на окружающую среду. Необходимо понимать различие между действием и реакцией. Действие – это только знания о том, что нужно делать эффекторам, а реакция – это то, что эффекторы фактически делают с окружающей средой. Например, команда «переместиться вперёд на один метр» – это действие, а фактическое перемещение – это реакция. Поскольку восприятия отображаются в *памяти агента*, то его действия определяются не одним, последним, восприятием, а *последовательностью восприятий или историей восприятий* за некоторый промежуток времени.

Дисциплина Искусственный интеллект интересуется, главным образом, организацией блока, обозначенного на рис. 1.1 вопросительным знаком. Простейший способ построения этого блока – это создание таблицы, состоящей из двух колонок. В первой колонке таблицы должны быть записаны все возможные последовательности восприятий, а во второй – соответствующие им действия. Для большинства агентов это будет очень

большая (фактически бесконечная) таблица.

Для иллюстрации идеи *табличного агента* воспользуемся следующим примером. Рассмотрим, показанный на рис. 1.2, мир, в котором работает агент-пылесос. Этот мир настолько прост, что имеется возможность описать всё, что в нём происходит. В данном конкретном мире имеются только два места в которых может находится агент-пылесос: квадраты A и B. Агент-пылесос обладает двумя сенсорами, позволяющими ему воспринимать квадрат, в котором он находится (`A` или `B`), а также наличие мусора в этом квадрате (`Clean` или `Dirty`). Агент может выбрать одно из следующих действий: перемещение влево (`Left`), перемещение вправо (`Right`) всасывание мусора (`Suck`) или остановка (`NoOp`).

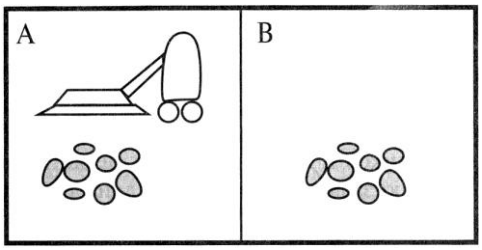

Рис. 1.2. Мир агента-пылесоса, в котором имеются только два местоположения.

Таблица на рис. 1.3 моделирует структуру памяти агента-пылесоса.

Последовательность восприятий	**Действие**
`[A, Clean]`	`Right`
`[A, Dirty]`	`Suck`
`[B, Clean]`	`Left`
`[B, Dirty]`	`Suck`
`[A, Clean], [A, Clean]`	`Right`
`[A, Clean], [A, Dirty]`	`Suck`
`. . .`	`. . .`
`[A, Clean], [A, Clean], [A, Clean]`	`Right`
`[A, Clean], [A, Clean], [A, Dirty]`	`Suck`
`. . .`	`. . .`

Рис. 1.3. Моделирование памяти агента-пылесоса при помощи таблицы.

Ясно, что поведение агента-пылесоса определяется тем, каким образом заполнен правый столбец на рис. 1.3.

1.1.1. Концепция интеллектуальности

Интеллектуальным агентом является такой агент, который выполняет «правильные» действия, с точки зрения достижения поставленной цели. Будем называть «правильными» действиями те действия агента, которые «приближают» его к цели. Последовательность действий агента, в процессе достижения цели, формирует его поведение.

Будем считать, что целенаправленное интеллектуальное поведение агента характеризуется следующими четырьмя факторами.

- Критерий успешности поведения.
- Знания агента о среде, приобретённые ранее.
- Действия, которые могут быть выполнены агентом.
- Последовательность восприятий, или история восприятий.

С учётом этих факторов можно сформулировать следующее определение интеллектуального агента.

Для каждой возможной последовательности восприятий интеллектуальный агент должен выбрать действие, которое максимизирует его показатели поведения, с

учётом информации, предоставленной последовательностью восприятий и знаниями о среде, которыми обладает агент.

1.1.2 Целенаправленное восприятие, обучение и автономность

Интеллектуальный агент не воспринимает окружающий мир пассивно, как термометр, измеряющий температуру в некоторой точке. Он должен *активно и целенаправленно собирать информацию об окружающей среде*. Например, если агент не оглядывается влево и вправо, прежде чем пересечь дорогу с интенсивным движением, а воспринимает только сигнал светофора, то полученная им последовательность восприятий не сможет подсказать, что слева к нему приближается грузовик. Таким образом, интеллектуальный агент, из всего бесконечно большого количества сенсорных событий которые генерируются окружающей средой, должен выбирать и воспринимать только те сенсорные события, которые необходимы на текущем этапе движения к цели.

Интеллектуальный *агент должен не только активно воспринимать окружающую среду, но также модифицировать и пополнять свои знания об окружающей среде* на основе сделанных восприятий. Способность модифицировать и пополнять знания на основе восприятий называется способностью обучаться.

Начальное содержимое памяти агента может отражать некоторые встроенные знания агента об окружающей среде, адекватные среде в момент создания агента. Однако, если среда монотонно изменяется, то эти знания во все большей степени устаревают и перестают быть адекватными. Если агент использует неадекватные знания, для формирования своего поведения, он совершает ошибки и может не достигнуть цели. Агент, обладающий способностью, обучатся, по мере изменения среды модифицирует и пополняет встроенные знания, приобретая «жизненный опыт». Обучение обеспечивает агенту обладание адекватными знаниями об изменяющейся окружающей среде на любом этапе существования агента. Можно, однако, предположить случай, когда среда остаётся неизменной на протяжении всей «жизни» агента. Тогда агенту достаточно встроенных знаний для успешного целенаправленного поведения. В этом случае ему не требуется обучаться. Он всегда действует правильно и не совершает ошибок.

Таким образом, поведение интеллектуального агента, в общем случае, базируется, как на знаниях, встроенных на этапе конструирования, так и на знаниях, приобретенных в процессе обучения.

Агент называется *автономным* в том случае, когда он формирует свое поведение, опираясь и на встроенные знания и на знания, полученные в процессе обучения. Автономный агент приобретает опыт в процессе обучения. И, пока такого опыта нет, агент строит свое поведение, опираясь только на встроенные знания.

Эволюция снабжает живых существ, на начальном этапе их существования, некоторым количеством встроенных знаний в виде рефлексов, позволяющих живому существу существовать до тех пор, пока не приобретен необходимый жизненный опыт. Цыпленка, который только что вылупился из яйца, не надо обучать клевать корм. Он это уже умеет делать на основе встроенных знаний. Новорожденного ребенка не надо обучать тому, как сосать молоко матери. Он это делает на основе встроенных знаний.

Неавтономный агент, или агент, строящий своё поведение только на основе встроенных знаний, не обладает достаточной гибкостью, поскольку встроенные знания отражают некоторые предположения о среде в момент создания агента. Неавтономный агент, таким образом, функционирует успешно до тех пор, пока неизменной остаётся среда.

1.2. Проблемная среда

Окружающая среда является источником проблем или задач, которые решает интеллектуальный агент. Поэтому часто, в случае интеллектуальных агентов, окружающую среду называют *проблемной средой*. По сути, наблюдаемое поведение интеллектуального агента – это внешнее проявление процесса решения им той или иной задачи. Интуитивно ясно, что чем сложнее окружающая среда, тем более трудные задачи она ставит перед агентом и тем сложнее должен быть сам агент.

1.2.1. Специфицирование агента в контексте проблемной среды

Для специфицирования интеллектуально агента в контексте проблемной среды будем использовать таблицу, состоящую из следующих пяти колонок: *тип агента, критерии поведения, характеристики среды, эффекторы и сенсоры*. Рассмотрим этот способ специфицирования интеллектуального агента на примере гипотетического робота-водителя такси. Этот пример будет использоваться нами и в дальнейшем. На рис. 1.4 приведена таблица, специфицирующая робота-водителя такси в контексте проблемной среды.

Тип агента	Критерии поведения	Характеристики среды	Эффекторы	Сенсоры
Водитель такси	Безопасная, и быстрая езда в рамках правил дорожного движения. Максимизация прибыли.	Дороги, другие транспортные средства, пешеходы, клиенты	Рулевое управление, акселератор, тормоз, световые сигналы, клаксон, дисплей	Видеокамеры, ультразвуковой дальномер, спидометр, спутниковая система навигации, одометр, акселерометр, датчики двигателя, клавиатура.

Рис. 1.4. Спецификация робота-водителя такси в контексте проблемной среды.

Прокомментируем таблицу на рис. 1.4. Критериев успешного поведения может быть несколько. К ним относятся: успешное достижение нужного места назначения; минимизация потребления топлива и износа оборудования; минимизация продолжительности и/или стоимости поездки; минимизация количества нарушений правил дорожного движения; максимизация безопасности и комфорта пассажиров; максимизация прибыли. Безусловно, что некоторые из отмеченных критериев конфликтуют, поэтому должны рассматриваться возможные компромиссы.

Теперь определим, что может представлять собой проблемная среда, в которой оперирует робот-водитель такси. Водителю такси приходится иметь дело с самыми различными дорогами, начиная с просёлков и узких городских переулков и заканчивая автострадами с двенадцатью полосами движения. На дороге встречаются другие транспортные средства, беспризорные животные, пешеходы, рабочие, производящие дорожные работы, милицейские автомобили, лужи и выбоины. Водителю такси приходится также иметь дело с потенциальными и реальными пассажирами. Кроме того, имеется ещё несколько важных дополнительных факторов. Таксисту может выпасть участь работать в Узбекистане, где зимой никогда не выпадает снег, или на севере России, где зимой на дорогах всегда лежит снег. Может оказаться, что водителю всю жизнь придётся ездить по правой стороне, или от него может потребоваться приспособиться к езде по левой стороне во время пребывания в Великобритании или в Японии. Безусловно, чем более простой является проблемная среда, тем проще задача проектирования робота.

Эффекторы, имеющиеся в автоматизированном такси, должны быть в большей или меньшей степени такими же, как и те, которые находятся в распоряжении водителя-человека: средства управления двигателем при помощи акселератора и средства управления вождением при помощи руля и тормозов. Кроме того, необходимы средства вывода информации на экран монитора или синтеза речи для передачи ответных сообщений пассажирам и, возможно, средства общения с водителями других транспортных средств.

Для достижения своих целей роботу-водителю такси необходимо будет знать, где он находиться, кто ещё едет по этой дороге, и с какой скоростью движется он сам. Поэтому в число его основных сенсоров должны входить одна или несколько управляемых видеокамер, спидометр и одометр. Для правильного управления автомобилем, особенно на поворотах, в нём должен быть предусмотрен акселерометр. Водителю потребуется также знать состояние механических узлов автомобиля, поэтому ему будет нужен набор датчиков для двигателя и электрической системы. Робот-водитель может также иметь спутниковую систему навигации для определения местоположения на электронной карте,

а также ультразвуковые датчики для измерения расстояния до других автомобилей и препятствий. Наконец, ему потребуется клавиатура или микрофон для общения с пассажирами.

В таблице на рис.1.5 приведены спецификации некоторых других типов агентов.

Тип агента	Критерии поведения	Характеристики среды	Эффекторы	Сенсоры
Медицинская система диагностики и лечения.	Успешное излечение пациента, минимизация затрат, отсутствие поводов для судебных тяжб.	Пациент, больница, персонал.	Средства вывода вопросов, диагнозов, рекомендаций, направлений.	Устройства ввода ответов пациентов и результатов лабораторных исследований.
Система анализа изображений полученных со спутника.	Правильная классификация изображения.	Канал передачи данных от орбитального спутника.	Вывод на экран монитора результатов классификации фрагментов изображений.	Массивы пикселей с данными о цвете и яркости.
Робот-сортировщик деталей	Безошибочная сортировки деталей по лоткам.	Конвейер с движущимися деталями, лотки.	Шарнирный манипулятор и захват.	Видеокамера, датчики углов поворотов шарниров.
Контроллер ректификационной установки	Максимизация степени очистки, продуктивности, безопасности	Ректификационная установка, операторы	Клапаны, насосы, нагреватели, дисплеи.	Датчики температуры, давления, химического состава.
Интерактивная программа обучения английскому языку	Максимизация оценок студентов на экзаменах.	Множество студентов, агентство обучения иностранным языкам.	Вывод на экран монитора обучающих и тестирующих стимулов.	Устройство ввода ответов

Рис. 1.5. Спецификация некоторых типов агентов в контексте проблемной среды.

В таблице на рис. 1.5 описаны как простые, так и более сложные среды. Например, для среды, в которой оперирует робот-сортировщик, предназначенный для контроля деталей, проходящих мимо него на ленточном конвейере, может использоваться целый ряд упрощающих допущений: освещение всегда включено, единственными предметами на ленте конвейера являются детали того типа, который ему известен, существует только два действия: принять изделие или забраковать его.

Интеллектуальные агенты могут оперировать как в материальной, так и в нематериальной проблемной среде. Если агент оперирует в материальной среде, то ему необходимы сенсоры для преобразования неэлектрических характеристик среды во внутреннее представление информации и эффекторы для материализации действий.

Агентов, оперирующих в нематериальной среде, часто называют *программными роботами* или *софтботами*. Если агент оперирует в нематериальной среде, то ему не нужны сенсоры и эффекторы. Он получает восприятия непосредственно из проблемной среды и передает действия непосредственно в среду. Примером интеллектуального агента, оперирующего в нематериальной среде является программный робот предназначенный для целенаправленного поиска информации в Internet. Для успешной работы ему требуются определённые способности к обработке текста на естественном языке, он должен уметь обучаться, а также изменять свое поведение динамически, когда, например, соединение с некоторым источником прерывается или появляется новый источник. Ясно, что такому агенту не нужны сенсоры или эффекторы.

Сегодня Internet представляет собой среду, которая отражает всю сложность ин-

формационных отношений между личностями, коммерческими компаниями, общественными и государственными организациями, и в число «обитателей» этой среды входит много интеллектуальных агентов.

1.2.2. Свойства проблемной среды

Разнообразие вариантов проблемной среды, в которой оперирует интеллектуальный агент, весьма велико. Однако можно определить относительно небольшое количество классификационных признаков и использовать их для типизации конкретной проблемной среды. Знание типа конкретной проблемной среды необходимо для выбора наиболее приемлемой архитектуры интеллектуального агента. Принято классифицировать проблемные среды при помощи шести характеристик, каждая из которых может принимать два альтернативных значения. Рассмотрим отмеченные классификационные характеристики на содержательном уровне.

Полностью наблюдаемая или частично наблюдаемая проблемная среда

Если сенсоры агента, в каждый момент времени, предоставляют ему доступ ко всей информации о состоянии среды, которая необходима и достаточна для достижения цели, то такая среда называется *полностью наблюдаемая*. По сути, проблемная среда является полностью наблюдаемой, если сенсоры агента, в каждый момент времени, выявляют все данные, которые достаточны агенту для выбора правильного действия. Полностью наблюдаемая среда является удобной, поскольку агенту не требуется поддерживать какую-либо внутреннюю модель проблемной среды для того, чтобы дополнять знания, полученные от сенсоров. Например, робот-водитель такси оперирует в частично наблюдаемой среде. Он не имеет сенсоров, которые сообщают ему, какие манёвры намереваются выполнить другие водители и должен получать эти сведения из внутренней модели проблемной среды. Полностью наблюдаемая среда может превратиться в *частично наблюдаемую среду* в результате выхода из строя ряда сенсоров.

Детерминированная или стохастическая проблемная среда

Если последующее состояние среды полностью определяется текущим состоянием среды, которое воспринял агент, и действием, которые выполнил агент, то такая среда называется *детерминированной*. В противном случае среда называется *стохастической*. В стохастической проблемной среде последующее состояние определяется не только действием, которое выполнил агент, но и непредсказуемым случайным фактором. Детерминированная среда является полностью предсказуемой и агент точно знает, что произойдет со средой в последующие моменты времени.

Среда, в которой оперирует робот-водитель такси, является стохастической, поскольку последующие состояния этой среды определяются не только действиями робота-водителя, но и непредсказуемым поведением других транспортных средств и пешеходов. Более того, в любом автомобиле совершенно неожиданно может произойти прокол шины или остановка двигателя.

Описанная ранее среда, в которой оперирует агент-пылесос является детерминированной, но другие варианты этой среды могут иметь стохастическими элементы, если мы допустим случайное появление мусора и ненадёжную работу механизма всасывания.

Эпизодическая или последовательная проблемная среда

В *эпизодической* проблемной среде поведение агента можно разбить на независящие друг от друга эпизоды. Независимость эпизодов друг от друга означает, что поведение агента в пределах некоторого эпизода не зависит от его поведения в остальных эпизодах. В эпизодической среде выбор действия в каждом эпизоде определяется только самим эпизодом. Например, агент, поведение которого заключается в распознавании дефектных деталей на сборочной линии является эпизодическим. Поведение, определяемое его действиями по обработке некоторой детали не зависит от его поведения при обработ-

ке остальных деталей. В *последовательной* среде все действия, формирующие поведение агента, связаны между собой и их нельзя сгруппировать в независимые эпизоды. Последовательной является проблемная среда при игре в шахматы, а также среда, в которой оперирует робот-водитель В перечисленных проблемных средах действие, выполняемое в данный момент имеет долговременные последствия. Интеллектуальный агент, ориентированный на работу в эпизодической проблемной среде может иметь более простую организацию, чем агент, ориентированный на работу в последовательной проблемной среде.

Статическая или динамическая проблемная среда

Если среда изменяется за время, которое агент тратит от момента восприятия и до момента формирования действия, то такая среда называется *динамической*. В противном случае она называется *статической*. Действовать в условиях статической среды проще, чем в условиях динамической среды, поскольку интеллектуальный агент может быть уверен в адекватности выбранного действия.

Если за время, которое агент тратит на формирование действия, сама среда не изменяется, а изменяется цель или критерии поведения агента, то такая среда называется *полудинамической*. Очевидно, что среда, в которой оперирует робот-водитель такси, является динамической, поскольку другие автомобили и само такси продолжают движение, в то время, когда робот-водитель формирует очередное действие. Среда, в которой оперирует робот-шахматист, в том случае, когда отсутствует контроль времени, является статической, а игра в шахматы с контролем времени – полудинамической. При игре в шахматы с контролем времени, если агент находится в цейтноте, то он может изменить первоначальную цель игры.

Дискретная или непрерывная проблемная среда

Различие между *дискретной* и *непрерывной* средой определяется способом реализации цикла восприятие-действие. Поведение агента в дискретной среде является пошаговым. Например, проблемная среда при игре в шахматы, имеет конечное количество различимых состояний. Поведение агента, при игре в шахматы, является пошаговым, поскольку поведение агента может быть представлено дискретным множеством восприятий и действий. Вождение такси осуществляется в среде с непрерывно меняющимися состояниями и непрерывно текущим временем, поскольку скорость и местонахождение такси и других транспортных средств изменяется в определённом диапазоне непрерывных значений.

Одноагентная или мультиагентная проблемная среда

Различие между *одноагентными* и *мультиагентными* средами на первый взгляд может оказаться достаточно простым. Например, очевидно, что агент, самостоятельно решающий кроссворд, оперирует в одноагентной среде, а команда агентов-футболистов – в мультиагентной. Тем не менее, при анализе этого классификационного признака возникают некоторые нюансы. Ранее, мы описали, на каком основании некоторая сущность *может* рассматриваться как агент, но не были сформулированы критерии того, какие сущности *должны* рассматриваться как агенты. Например, должен ли агент робот-водитель такси считать агентом другой автомобиль, или он должен относиться к нему как к элементу среды?

Проблемы проектирования агентов, оперирующих в мультиагентной среде, часто, существенно отличаются от тех, с которыми приходится сталкиваться в одноагентной среде. Например, одним из признаков поведения агентов в мультиагентной среде может быть связь и обмен информацией между агентами.

Как и следует ожидать, *наиболее сложными средами являются частично наблюдаемые, стохастические, последовательные, динамические, непрерывные и мультиагентные*, а наиболее простыми – *полностью наблюдаемые, детерминированные, эпизодические, статические, дискретные и одноагентные*. Эти два класса агентов можно рас-

сматривать как полюса, между которыми располагаются остальные классы агентов. В таблице на рис.1.6 дана классификация некоторых известных проблемных сред.

Проблемная среда	**Наблюдаемая полностью или частично**	**Детерминированная или стохастическая**	**Эпизодическая или последовательная**	**Статическая, динамическая или полудинамическая**	**Дискретная или непрерывная**	**Одноагентная или мультиагентная**
Решение кроссворда	Полностью наблюдаемая	Детерминированная	Последовательная	Статическая	Дискретная	Одноагентная
Игра в шахматы с контролем времени	Полностью наблюдаемая	Стохастическая	Последовательная	Полудинамическая	Дискретная	Мультиагентная
Игра в нарды	Полностью наблюдаемая	Стохастическая	Последовательная	Статическая	Дискретная	Мультиагентная
Вождение такси	Частично наблюдаемая	Стохастическая	Последовательная	Динамическая	Непрерывная	Мультиагентная
Медицинская диагностика и лечение	Частично наблюдаемая	Стохастическая	Последовательная	Динамическая	Непрерывная	Одноагентная
Анализ изображений	Полностью наблюдаемая	Детерминированная	Эпизодическая	Полудинамическая	Непрерывная	Одноагентная
Робот-сортировщик деталей	Частично наблюдаемая	Стохастическая	Эпизодическая	Динамическая	Непрерывная	Одноагентная
Контроллер ректификационной установки	Частично наблюдаемая	Стохастическая	Последовательная	Динамическая	Непрерывная	Одноагентная
Интерактивная программа, обучающая английскому языку	Частично наблюдаемая	Стохастическая	Последовательная	Динамическая	Дискретная	Одноагентная

Рис. 1.6. Примеры проблемных сред и их классификационные характеристики.

1.3. Структура интеллектуальных агентов

Основная задача дисциплины «Искусственный интеллект» заключается в разработке способов реализации блока, обозначенного символом «вопросительный знак» на рис. 1.1. Этот блок формирует поведение интеллектуального агента путем путём отображения истории восприятий и знаний агента о среде в целенаправленные действия. Дисциплина «Искусственный интеллект» отличается от других дисциплин, изучающих и моделирующих интеллектуальную деятельность, например, от дисциплины «Кибернетика» тем, что предполагает реализацию отмеченного блока в виде компьютерной программы.

Программа интеллектуального агента должна работать в вычислительной системе, снабжённой сенсорами и эффекторами. Эту систему называют *архитектурой интеллектуального агента*. Архитектура интеллектуального агента обеспечивает формирование и передачу в программу восприятий, выполнение программы, а также формирование действий и передачу их в эффекторы. Очевидно, что программа агента и его архитектура должны соответствовать друг другу. Например, если программа формирует действие `Walk` (идти), то архитектура агента должна включать опорно-двигательный аппарат.

Таким образом, в настоящем курсе основное внимание будет сфокусировано на

изучении идей о том, каким образом могут быть организованы программы интеллектуальных агентов. Предполагается, что это первостепенная задача и, что для каждой программы агента всегда можно подобрать соответствующую архитектуру.

1.3.1. Программы интеллектуальных агентов

Все программы агентов, рассматриваемые в настоящем пособии, построены по одному и тому же принципу. Они получают текущее восприятие, и отображают его в соответствующее действие. Целенаправленное поведение формируется путем многократного повторения отмеченного отображения «восприятие-действие». Таким образом, изучаемые нами программы интеллектуальных агентов работают исходя из упрощающего предположения, заключающегося в том, что *восприятие отождествляется с оцифрованным сенсорным событием*. Это предположение означает эквивалентность восприятия и сенсорного события, что упрощает программу интеллектуального агента по сравнению с тем, что происходит в системе, изображенной на рис. 1.1.

Ранее мы отмечали, что программа интеллектуального агента отображает историю восприятий в текущее действие. Архитектура агента предоставляет программе поток восприятий по принципу «одно восприятие за один раз». Поэтому, запоминание восприятий и формирование истории восприятий должна осуществлять сама программа.

Для описания программ агентов будем использовать простой псевдокод. Запись программы на псевдокоде позволит отражать в нем только существенные особенности агента, избегая излишней детализации.

Первой особенностью псевдокода является описание всех переменных (входные аргументы и возвращаемые значения процедур, и внутренние переменные) в виде их имена. Типы переменных будут игнорироваться. Как правило, переменные псевдокода должны пониматься как ссылочные, хранящие ссылки на более или менее сложные структуры данных.

Второй особенностью псевдокода является использование процедурной парадигмы при записи программ. Программа интеллектуального агента будет представляться как программная функция или процедура. Операционная часть этой процедуры будет составлен из последовательности вызовов внутренних процедур, реализующих идею отображения восприятия в действие. Внутренние процедуры будут записываться в следующем виде.

```
<имя возвращаемого значения> ← <имя процедуры>(<входные параметры>)
```

Третьей особенность псевдокода является возможность спецификации внутренней процедуры в виде естественно-языкового предложения в угловых скобках.

Программа агента, приведенная на рис. 1.7 иллюстрирует описанный псевдокод на примере кодирования программы рассмотренного ранее табличного агента (см., например, рис. 1.3).

```
function TableDrivenAgent(perception) returns action
   static: historyOfPerception, очередь истории восприятий,
                                первоначально пустая
                        table, таблица действий, индексированная
                                историями восприятий и полностью
                                заполненная
   <добавить текущее восприятие в конец historyOfPerception>
   action ← Lookup(historyOfPerception, table)
return action
```

Рис. 1.7. Пример программы табличного агента.

Как видно на рис. 1.7, программа интеллектуального агента рассматривается нами в виде программной процедуры. Процедура имеет один входной параметр в виде ссылки на структуру данных, моделирующую восприятие (`perception`). Возвращаемым значением процедуры является ссылка на структуру данных, которая моделирует действие (`ac-`

tion). Заголовок процедуры, приведенный на рис. 1.7, является стандартным, и будет использоваться нами во всех последующих программах интеллектуальных агентов. Изменяться будут лишь наименования программ.

В начальной части процедуры, после ключевого слова **static,** описываются ссылки на структуры данных, моделирующие память агента. В нашем случае это: (1) ссылка historyOfPerception на структуру данных, которая хранит историю восприятий и (2) ссылка table на таблицу действий.

Работа программы по формированию действия осуществляется путём последовательного вызова внутренних процедур, реализующих концепцию конкретного типа агента. В нашем случае это: (1) вызов процедуры (записана в виде естественно-языкового предложения) которая добавляет результат текущего восприятия в память, хранящую историю восприятий и (2) вызов процедуры Lookup. Процедура Lookup просматривает таблицу и находит строку, в которой левая часть совпадает с содержимым historyOf-Perception. Процедура Lookup возвращает ссылку action на действие, описанное в правой части строки.

Программа, приведенная на рис. 1.7, иллюстрирует нотацию, используемого нами псевдокода на примере табличного агента. Однако сама идея табличного агента является очень примитивной и обречена на провал при попытке её использования на практике. Анализ того, почему подход к созданию интеллектуального агента, основанный на использовании таблицы действий, обречён на неудачу, является важным и иллюстрирует причину, которая, часто, препятствует практической реализации идей построения интеллектуальных агентов.

Рассмотрим такси, управляемое роботом-водителем. Визуальные входные данные от одной видеокамеры поступают со скоростью примерно 27 мегабайт в секунду (30 кадров в секунду, 640 x 480 пикселей с 24 битным кодированием одного пикселя). Согласно этим данным таблица действий, рассчитанная на один час работы робота-водителя, должна содержать количество строк, превышающее число $10^{250\,000\,000\,000}$. Таблица действий, для случая игры в шахматы состоит, по меньшей мере, из 10^{150} строк. Ошеломляющий размер этих таблиц (притом, что количество атомов в наблюдаемой вселенной не превышает 10^{81}) означает: (1) что ни один физический агент в нашей вселенной не имеет пространства для хранения такой таблицы, (2) что проектировщик не располагает временем, достаточным для создания такой таблицы и (3) что ни один агент никогда не сможет обучиться тому, что содержится во всех правильных записях этой таблицы, на основании собственного опыта.

Табличный агент, хоть и осуществляет отображение истории восприятий в действие и, поэтому, соответствует определению интеллектуального агента, однако практически не реализуем. В оставшейся части настоящего раздела мы изучим ещё четыре типа интеллектуальных агентов. Эти агенты последовательно развивают идею табличного агента, но в отличие от него более пригодны для практической реализации. Ниже приведен список этих агентов, расположенных в порядке возрастания сложности их внутренней организации:

- простые рефлексные агенты;
- рефлексные агенты с внутренней моделью мира;
- целеориентированные агенты;
- агенты, действующие на основе функции полезности.

1.3.2. Простые рефлексные агенты

Для того, чтобы трансформировать табличного агента в простого рефлексного сделаем два шага. Во-первых, будем считать, что *агент формирует действия на основе текущего восприятия, игнорируя всю остальную историю восприятий*. Во-вторых, проанализируем причину, больших размеров таблицы действий и предложим способ радикального уменьшения количества информации, хранимой в этой таблице.

Одна из причин того, что табличный агент нереализуем на практике, заключается в том, что он использует примитивный и ресурсоемкий способ отображения восприятия в действия. Его сенсорная система непосредственно связана с эффекторами. Если свести

сенсорную систему такого агента к системе технического зрения на основе видеокамеры, а его эффекторы – к манипулятору, имитирующему руку человека, то такой агент представляет собой робота типа «глаз-рука» у которого зрительные образы сенсорных событий непосредственно отображаются в закодированные действия.

Однако, не вся информация, содержащаяся в зрительном образе необходима для выбора действия. Представим себя на месте робота-водителя такси. Если у движущегося впереди автомобиля загораются симметричные тормозные огни красного цвета, то мы воспринимает это сенсорное событие как некоторое состояние среды, заключающееся в том, что впереди идущий автомобиль тормозит. Факт торможения впереди идущего автомобиля может быть представлен гораздо компактнее, чем визуальное восприятие сенсорного события. Воспринятое сенсорное событие может быть трансформировано в компактное описание состояния среды, с тормозящим автомобилем. Способ кодирования факта торможения впереди идущего автомобиля может быть любым. Например, в виде естественно-языковой фразы: `car-in-front-is-breaking` (движущийся впереди автомобиль тормозит). Ясно, что у водителя такси приведенная фраза должна всегда ассоциироваться с действием: `initiate-braking` (начать торможение).

В искусственном интеллекте знания о том, каким образом конкретное состояние проблемной среды связано с действием, часто, представляют *продукционным правилом* или *правилом типа условие-действие*. В простейшем случае продукционное правило имеет вид:

```
if car-in-front-is-breaking then initiate-braking.
```

В левой части продукционного правила записано закодированное состояние среды, а в правой – код соответствующего действия.

Таким образом, если интеллектуальный агент умеет трансформировать сенсорные события в компактные описания состояний проблемной среды, то каждую строку таблицы действий можно заменить продукционным правилом. С учетом принятого допущения о том, что рефлексный агент формирует действия на основе текущего восприятия, в левой части продукционных правил размещается описание только одного факта. Ясно, что робота-водителя такси размер продукционного правила существенно меньше, чем размер соответствующей строки таблицы действий.

На рис. 1.8 приведена структура простого рефлексного агента, действующего на основе продукционных правил. В схеме, на рис. 1.8 обычный прямоугольник обозначает текущую информацию, необходимую агенту для выбора действия в данный момент времени, а прямоугольник с закруглёнными углами – знания, в виде продукционных правил, хранящиеся в долговременной памяти.

В дисциплине «Искусственный интеллект» множество продукционных правил, хранящих знания, необходимые для решения некоторой проблемы называют *базой знаний*. Продукционные правила являются не единственным способом представления знаний.

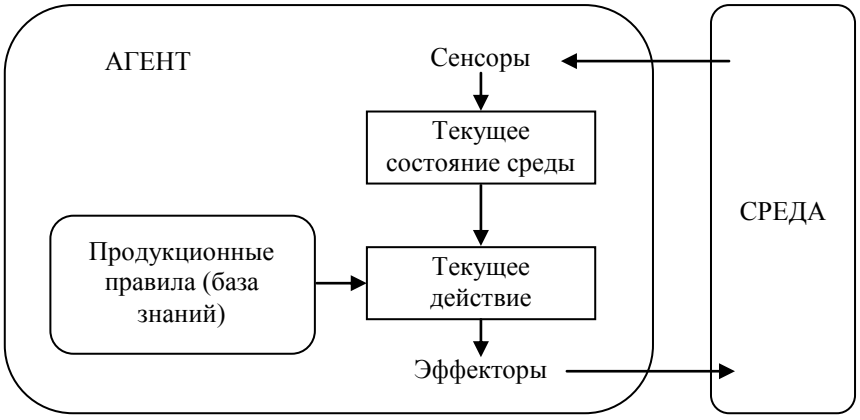

Рис. 1.8. Структура простого рефлексного агента.

На рис. 1.9. приведена программа простого рефлексного агента.

```
function SimpleReflexAgent(perception) returns action
   static: rules, множество продукционных правил

   state ← InterpretInput(perception)
   rule ← RuleMatch(state, rules)
   action ← RuleAction(rule)
return action
```

Рис. 1.9. Программа простого рефлексного агента.

Процедура `InterpretInput` трансформирует восприятие в закодированное описание состояния проблемной среды (левая часть продукционного правила). Процедура `RuleMatch` возвращает *первое найденное* продукционное правило, у которого левая часть совпадает со значением переменной `state`.

Простые рефлексные агенты обладают тем достоинством, что имеют простую организацию и могут быть реализованы на практике. Однако они обладают весьма ограниченным интеллектом и похожи на одноклеточные живые организмы, которые строят свое поведение исключительно на основе рефлексов. *Простой рефлексный агент выбирает действие только на основе текущего восприятия и поэтому адекватное действие может быть выбрано в том случае, если среда, в которой оперирует агент, является полностью наблюдаемой*.

1.3.3. Рефлексные агенты с внутренней моделью проблемной среды

Простой рефлексный агент может успешно оперировать только в полностью наблюдаемой проблемной среде. В том случае, когда рефлексный агент оперирует в частично наблюдаемой среде, он воспринимает только часть информации, необходимой для выбора действия. Поэтому, для формирования адекватного поведения он должен уметь получать остальную, необходимую ему информацию, из некоторой *внутренней модели проблемной среды*. Например, у агента-водителя такси нет, и не может быть датчиков, непосредственно воспринимающих маневры транспортных средств, находящихся на дороге: торможение, остановка или поворот впереди идущего автомобиля, выполнении маневра обгона автомобилем, находящимся сзади и т.п. Однако, он может получать эту информацию при помощи модели проблемной среды.

Модель проблемной среды включает знания двух видов: (1) *знания о том, как среда изменяется независимо от агента* (например, знания о том, что автомобиль, идущий на обгон, приближается); (2) *знания о том, как влияют на среду действия самого агента* (например, знания о том, что при повороте рулевого колеса по часовой стрелке автомобиль поворачивает вправо).

На рис. 1.10 приведена структура рефлексного агента с внутренней моделью проблемной среды.

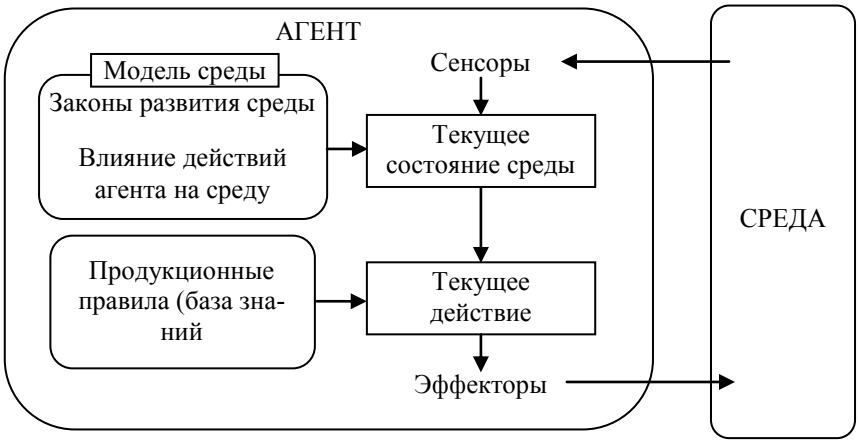

Рис. 1.10. Структура рефлексного агента с внутренней моделью проблемной среды.

Знания о текущем состоянии проблемной среды, полученные при помощи сенсоров, комбинируется со знаниями, полученными при помощи внутренней модели среды для выработки адекватного описания текущего состояния среды.

На рис. 1.11 приведена программа рефлексного агента с внутренней моделью проблемной среды.

```
function ReflexAgentWithModel(perception) returns action
   static: rules, множество продукционных правил

   partialState ← InterpretInput(perception)
   fullState ← CurrentState(partialState, InternalModel())
   rule ← RuleMatch(fullState, rules)
   action ← RuleAction(rule)
return action
```

Рис. 1.11. Программа рефлексного агента с внутренней моделью проблемной среды.

В программе, приведенной на рис. 1.11, процедура `InterpretInput` трансформирует восприятие в закодированное *частичное описание* состояния проблемной среды (`partialState`), а процедура `CurrentState`, формирует *полное описание* состояния проблемной среды (`fullState`). Эта процедура использует, в качестве входных параметров, как частичное описание состояния проблемной среды, так и возвращаемое значение процедуры `InternalModel()`.

1.3.4. Целеориентированные агенты

Знаний о текущем состоянии среды, даже если они точны и адекватны, не всегда достаточно для принятия решения о том, какое действие необходимо выполнить. Например, на перекрёстке дорог такси может повернуть налево, ехать прямо или повернуть направо. Правильное решение зависит от того, куда должно попасть такси. Иными словами агенту требуется не только описание текущего состояния среды, но и описание некоторого *целевого состояния*. Если агент оперирует понятием целевое состояние, то он может сравнивать целевое состояние с тем состоянием проблемной среды, которое порождается при применении некоторого действия и стремится к выбору такого действия, которое сокращает различие между целевым состоянием и текущим состоянием.

Процесс принятия решения о выборе действия *целеориентированным агентом* имеет фундаментальные отличия от процесса принятия решения рефлексными агентами, поскольку агенту приходится размышлять о будущем (прогнозировать), отвечая на вопросы: «Что произойдёт с проблемной средой, если я выполню данное действие?» и «Позволит ли данное действие приблизиться к целевому состоянию?». Рефлексные агенты не способны на подобные размышления. Рефлексный агент-водитель такси тормозит рефлекторно, сразу же после того, как он определил, что движущийся впереди автомобиль тормозит, даже если он находится далеко впереди. Целеориентированный агент-водитель такси принимает решение о торможении на основании рассуждений. Он может рассудить, что если движущийся впереди автомобиль тормозит, то этот автомобиль замедляет движение и, следовательно, расстояние до него сокращается. Он может продолжить рассуждения следующим образом. Если расстояние до тормозящего автомобиля велико, то можно продолжать движение, а если расстояние меньше критического и целью агента-водителя такси является предотвращение столкновения, то единственным действием, позволяющим её достичь, является торможение. Таким образом, главным отличием целеориентированного агента от рефлексных агентов является умение строить дедуктивные умозаключения. Способность строить умозаключения моделируется умением агента строить цепи продукционных правил, выбирая из базы знаний и связывая между собой необходимые продукционные правила.

Целеориентированный агент более совершенен, чем рефлексные агенты, а его поведение более гибкое. Если, например, мы реализуем агента-водителя такси при помощи рефлексного агента, то мы должны описать маршрут его движения при помощи набора продукционных правил. При этом, каждый раз когда изменяется пункт назначения, пра-

вила должны быть переписаны. Для целеориентированного агента можно составить один универсальный набор правил, из которых каждый раз, в зависимости от целевого пункта назначения, путём рассуждений, агент будет формировать требуемый маршрут.

1.3.5. Агенты, действующие на основе функции полезности

Целеориентированный агент достигает цель первым попавшимся способом, который не обязательно является лучшим в смысле некоторого критерия успешности его поведения. Во многих случаях, для выработки высококачественного поведения одного лишь учёта цели недостаточно. Например, существует много последовательностей действий, позволяющих агенту-водителю такси добраться до пункта назначения и, тем самым, достигнуть поставленную цель. Однако некоторые из этих последовательностей обеспечивают более быструю или более безопасную или более дешёвую поездку, чем другие. Поскольку существует несколько способов достичь одну и ту же цель, необходим критерий сравнения этих способов. Этот критерий был ранее назван критерием успешности поведения. Для количественной оценки поведения агента по отношению к критерию успешности поведения используется *функция полезности*.

Функция полезности ставит в соответствие каждой последовательности действий некоторое вещественное число. При помощи функции полезности можно оценить множество способов достижения цели одного или нескольких агентов. Часто целенаправленное поведение агента направлено на достижение одной из некоторого количества целей. В этом случае функция полезности позволяет оценить последовательности действий, для достижения каждой из этих целей.

Упражнения

1.1. Для каждого из следующих агентов разработайте спецификацию в контексте проблемной среды (см. рис. 1.4).

a. робот-футболист;
b. агент, покупающий книги в Internet;
c. автономный марсианский вездеход;
d. ассистент математика, занимающийся доказательством теорем.

1.2. Для каждого агента, указанного в упражнении 1.1, охарактеризуйте среду в соответствии с признаками, приведенными в 1.2.2, и выберите подходящего агента с наиболее простой внутренней организацией (простой рефлексный, рефлексный с внутренней моделью проблемной среды, целеориентированный и действующий на основе функции полезности). При решении упражнения используйте результаты, полученные при выполнении упражнения 1.1.

1.3. Обоснуйте каждый классификационный признак проблемных сред, приведенных на рис. 1.6.

1.4. Разработайте систему продукционных правил для агента-пылесоса, рассматриваемого как постой рефлексный агент.

1.5. Предложите программу целеориентированного агента.

1.6. Предложите программу агента, действующего на основе функции полезности.

2. РЕШЕНИЕ ПРОБЛЕМ ПОСРЕДСТВОМ ПОИСКА

Второй раздел посвящён изучению интеллектуальных *агентов, решающих проблемы*, которые рассматриваются как разновидности целеориентированных агентов или агентов, действующих на основе функции полезности. Агенты, решающие проблемы строят своё поведение путём поиска решения проблем и последующей реализации найденного решения при помощи эффекторов. Решением проблемы называется последовательности действий, которые необходимо выполнить, чтобы из некоторого *начального состояния* проблемной среды перейти к целевому *состоянию проблемной среды*. Таким образом, под целью интеллектуальный агент всегда понимает некоторое желаемое состояние проблемной среды.

2.1. Агент, решающий проблемы

Сформулируем этапы поведения агента, решающего проблемы. Описанные, ниже, этапы являются унифицированными в том смысле, что агент должен последовательно их выполнить при решении любой проблемы. При формулировке этапов будем, для их иллюстрации использовать пример решения следующей проблемы. Представим, что агент-водитель такси находится в Румынии в городе Арад, из которого он должен попасть в город Бухарест, перемещаясь на своем автомобиле по дорогам Румынии.

Первым этапом, с которого агент начнет решение проблемы, является *формулировка цели*. Агент рассматривает *цель как желаемое состояние проблемной среды*. Таким образом, при формулировке цели, агент, зная текущее состояние среды, должен решить какое состояние для него является желаемым. Для некоторых проблем существует множество целевых состояний среды, и достижение любого из них является решением проблемы. Например, если целью является нахождением места, где можно пообедать, то достижением цели является обнаружение любого из близлежащих кафе. Важным является правильный выбор уровня абстракции для понятия «состояние среды». Для рассматриваемой проблемы под понятием «состояние» будем понимать «нахождение в одном из городов Румынии».

После завершения этапа формулировки цели агент должен выполнить этап, называемый *формулировка проблемы*. На этом этапе детерминируется *пространство состояний среды*, которое включает начальное, целевые и все промежуточные состояния, а также *множество действий* агента, при помощи которых он может изменять состояния среды. Здесь, также, важно выбрать приемлемый уровень абстракции для понятия «действие». Если, например, агент, путешествующий по автомобильным дорогам Румынии, будет оперировать действиями на уровне «перемещение левой ноги вперёд на один сантиметр» или «поворот рулевого колеса на один градус влево», то их количество будет недопустимо велико. Для решения задачи путешествия из Арада в Бухарест приемлемый уровень абстракции для интерпретации понятия действия является «*перемещение из данного города в один из ближайших городов*». Выбранный уровень абстрагирования игнорирует все детали такого перемещения.

Предположим, что у агента есть карта автомобильных дорог Румынии. Эта карта может рассматриваться как модель пространства состояний проблемы, и снабжает агента информацией о состояниях, в которых он может оказаться, и о действиях, которые ему доступны. Если, например, состоянием среды является «нахождение в городе Арад», то применив к этому состоянию действие «переместится из Арада в Зеринд» агент переведет среду в новое состояние «нахождение в городе Зеринд». Если среда детерминирована, то агент имеет возможность воспользоваться картой для поиска последовательности действий для гипотетического путешествия по автомобильным дорогам Румынии в попытке найти такой путь, который, в конечном итоге, приведёт его в Бухарест.

Описанный процесс называется *поиском решения*, который является третьим этапом и следует за этапом формулировки проблемы. Процедура поиска принимает в качестве входных параметров формулировку проблемы и возвращает *решение* в виде последовательности действий.

После того как решение найдено, могут быть реализованы действия, содержащиеся в решении. Реальное исполнение действий эффекторами агента осуществляется на за-

ключительном, четвертом этапе *выполнения решения*.

Обобщая сказанное можно написать программу агента, решающего проблему, отражающие унифицированные этапы решения проблемы. Эта программа приведена на рис. 2.1.

```
function ProblemSolvingAgent(perception) returns action
   static: seq, последовательность действий, первоначально пустая
           state, текущее состояние среды
           goal, целевое состояние, первоначально не определённое
           problem, формулировка проблемы

   state ← UpdateState(state, perception)
   if последовательность seq пустая then do
      goal ← FomulateGoal(state)
      problem ← FomulateProblem(state, goal)
      seq ← Search(problem)
repeat
   action ← First(seq)
   seq ← Rest(seq)
   return action
until
```

Рис. 2.1. Программа агента, решающего проблему.

При составлении программы, приведенной на рис. 2.1, предполагалось, что среда является *статической*, поскольку этапы формулировки цели и проблемы, а также поиск решения осуществляются без учёта каких-либо изменений, которые могут произойти в среде. Кроме того, предполагалось, что начальное состояние среды известно. Получить такие сведения можно в том случае, если среда является полностью *наблюдаемой*. Как при поиске решения, так и при его выполнении поведение агента является пошаговым. Поэтому, мы можем предположить, что среда является *дискретной*. Последней и наиболее важной особенностью является следующее. Программа агента предполагает, что среда является *детерминированной*. Решение проблемы представляет собой единственную последовательность действий и в ней не учитываются какие-либо неожиданные события. Поэтому, после того как решение найдено, оно может выполняться без учёта результатов текущего восприятия. Агент может выполнять найденное решение, образно говоря с «закрытыми глазами». Все эти предположения означают, что мы составили программу агента, решающего проблемы для оперирования в простейшей проблемной среде.

2.1.1. Хорошо структурированные проблемы

Один из наиболее простых классов проблем, которые могут решаться интеллектуальными агентами, является класс *хорошо структурированных проблем*. *Хорошо структурированные проблемы решаются в полностью наблюдаемой, статической и детерминированной среде* и, поэтому, агент может прогнозировать последствия каждого действия.

Будем формулировать хорошо структурированную проблему при помощи следующих четырёх компонентов.

- *Начальное состояние*, в котором находится среда, когда агент приступает к решению проблемы. Например, для агента-водителя такси, который решает проблему поиска пути из Арада в Бухарест, начальное состояние может быть формально описано в виде `In(Arad)`.
- Перечень действий, доступных агенту, или *функция последующих состояний* – `SuccessorFn`. Если имеется конкретное состояние среды `x`, то функция `SuccessorFn(x)` возвращает множество упорядоченных пар `<action,successor>`, где `action` представляет собой одно из действий, доступных агенту в состоянии `x`, а `successor` – состояние, которое может быть достигнуто из сосстояния `x` путём применения действия `action`. Например, для состояния `In(Arad)` функция по-

следующих состояний должна возвратить множество, состоящее из трех пар: {<Go(Sibiu), In(Sibiu)>, <Go(Timisoara), In(Timisoara)>, <Go(Zerind), In(Zerind)>}. Если среда детерминирована, то начального состояния и функции последующих состояний достаточно для задания всего пространства состояний. Пространство состояний может быть представлено графом, узлами которого являются состояния, а рёбрами – действия. Карта автомобильных дорог Румынии, приведенная на рис. 2.2, может интерпретироваться как граф пространства состояний, если каждая дорога обозначает два действия (перемещение из города в город в обоих направлениях).

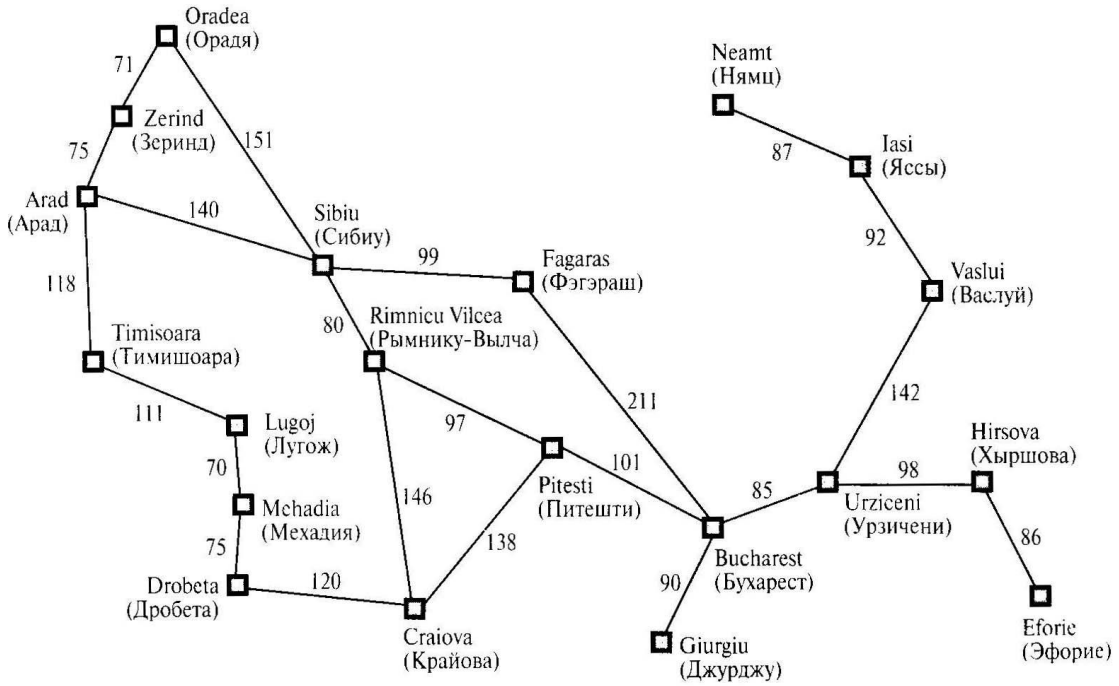

Рис.2.2. Упрощённая карта автомобильных дорог Румынии.

Путём в пространстве состояний называется последовательность состояний, соединённых последовательностью действий. Таким образом, решением проблемы является путь от начального состояния к одному из целевых состояний.

- *Тест цели*, представляющий собой процедуру распознавания, которая позволяет определить, является ли данное конкретное состояние целевым и остановить процесс поиска решения. В некоторых случаях агент, приступая к решению проблемы, имеет явно заданное множество возможных целевых состояний. В этом случае тест цели представляет собой процедуру, в основе алгоритма которой лежит сравнение текущего состояния с одним из целевых. Например, целевым состоянием проблемы, которая решается агентом, ищущим путь из Арада в Бухарест, является одноэлементное множество {In(Bucharest)}. В некоторых случаях цель задаётся в виде совокупности признаков, характеризующих целевое состояние. Например, в шахматной игре цель состоит в достижении состояния, называемого «шахматный мат», в котором король противника атакован и не может уклониться от удара. Ясно, что, приступая к игре в шахматы, агент не знает как будут расположены фигуры на шахматной доске, если игра завершается шахматным матом. Он располагает только набором признаков, характеризующих состояние «шахматный мат».
- *Функция стоимости пути*, которая назначает числовое значение каждому пути в пространстве состояний. Функция стоимости пути необходима, если агент решающий проблему является разновидность агента, действующего на основе функции полезности. Для такого агента функцией полезности является функция стоимости пути. Если для агента, ищущего путь из Арада в Бухарест, важнее всего расстояние, то стоимость пути может измеряться в километрах, а функция стоимости пути по-

лучая, в качестве входного параметра путь, возвращает его длину в километрах. В настоящей главе предполагается, что стоимость пути может быть описана как сумма стоимостей отдельных действий, составляющих путь. На рис. 2.2, стоимости отдельных действий показаны в виде расстояний между городами. Функция стоимости пути нужна, для того, чтобы можно было сравнить между собой несколько решений.

Описанные выше компоненты являются формулировкой хорошо структурированной проблемы, могут быть представлены в виде некоторой структуры данных и переданы процедуре поиска решения. Процедура поиска решения возвращает решение в виде пути от начального состояния до одного из целевых состояний. Качество решения измеряется при помощи значения функции стоимости пути. *Оптимальным решением* будем называть такое решение, которое имеет *наименьшую стоимость пути*.

M1

2.2. Примеры проблем

Описанный подход к решению хорошо структурированных проблем был применён к разработке ряда программ интеллектуальных агентов. Одна из наиболее известных программ такого типа, разработанная около 50 лет тому назад в университете Carnegie Mellon (США) получила наименование GPS (General Problem Solver) или Общий Решатель Задач. Программа использовалась для поиска решения различных хорошо структурированных проблем.

В настоящем параграфе описаны некоторые из наиболее известных примеров проблем, которые решаются современными агентами, решающими проблемы. Приведенные примеры проблем делятся на игрушечные проблемы и проблемы реального мира.

Игрушечные проблемы являются хорошо структурированными и предназначены для иллюстрации основных компонентов хорошо структурированных проблем, а также для проверки поисковых алгоритмов. Как правило, эти проблемы обладают относительно небольшим пространством состояний и используются при изучении агентов, решающих проблемы.

Проблемы реального мира, часто, не являются хорошо структурированными проблемами, но относятся к тем проблемам, решение которых представляет практический интерес.

2.2.1. Игрушечные проблемы

Мир агента-пылесоса

В качестве первого примера рассмотрим проблему, которую решает робот-пылесос, рассмотренный в первой главе (см. рис. 1.2). Трактуя поведение агента-пылесоса, как поведение агента, решающего хорошо структурированную проблему, сформулируем эту проблему в том виде, когда ее решение может быть поручено интеллектуальному агенту.

- *Состояние*. Под состоянием будем понимать расположение агента-пылесоса и мусора в квадратах. Агент может находиться в одном из двух местоположений, в каждом из которых может присутствовать или не присутствовать мусор. Таким образом, имеется 2(количество квадратов) x 2(количество местоположений агента) x 2(количество местоположений мусора) = 8 возможных состояний в пространстве состояний, в котором агент-пылесос осуществляет поиск решения.
- *Начальное состояние*. В качестве начального состояния может быть назначено любое состояние проблемной среды. Например, агент находится в квадрате A, мусор находится в квадратах A и B.
- *Функция последующих состояний*. Вместо функции последующих состояний зададим список действий, доступных агенту. Будем считать, что для решения проблемы агенту необходимы три действия: `Left` (переместиться на один квадрат влево), `Right` (переместиться на один квадрат вправо) и `Suck (удалить мусор)`. Полное пространство состояний, в котором агент-пылесос осуществляет поиск решения, в графическом виде приведено на рис. 2.3.

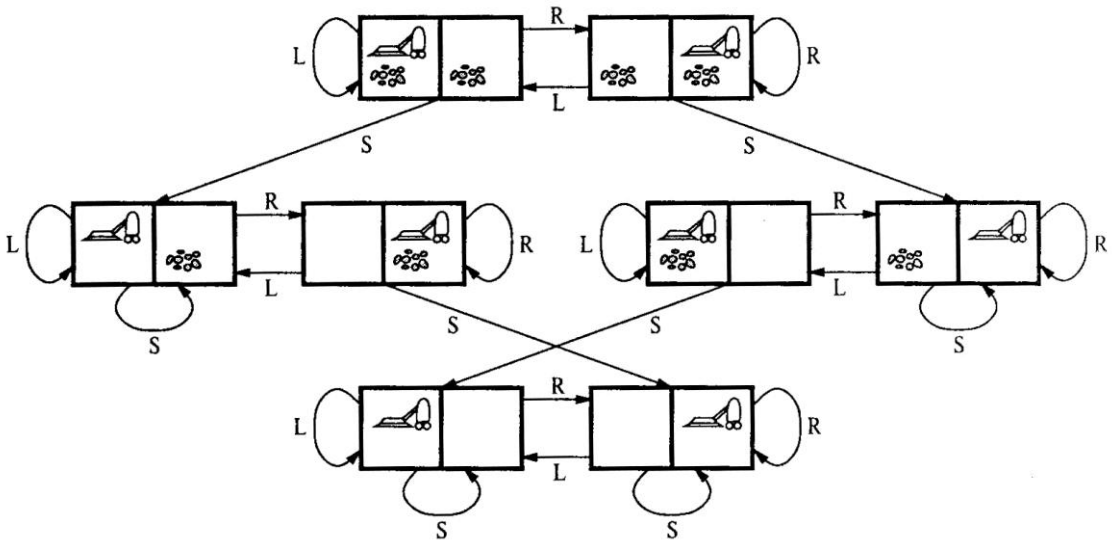

Рис.2.3. Пространство состояний для проблемной среды агента-пылесоса. Дуги обозначают действия: `L = Left`, `R = Right`, `S = Suck`

- *Тест цели*. Проблема, которую решает агент-пылесос, имеет два целевых состояния, изображенных в нижней части рис. 2.3. Поэтому тест цели представляет собой процедуру, которая останавливает процесс поиска, если обнаруживает, что в обоих квадратах отсутствует мусор.
- *Стоимость пути*. Примем, что стоимость каждого действия равна 1. Поэтому стоимость пути равна количеству действий в этом пути, а лучшим является решение, которое позволяет получить одно из целевых состояний за меньшее количество действий.

Головоломка с восемью фишками

Головоломка с восемью фишками состоит из коробки, размером 3 x 3, в которой расположены восемь пронумерованных фишек. В коробке имеется один пустой участок размером с фишку. Фишка, смежная с пустым участком, может быть передвинута на этот участок. На рис. 2.4 приведены изображения возможного начального и целевого состояний *головоломки с восемью фишками*.

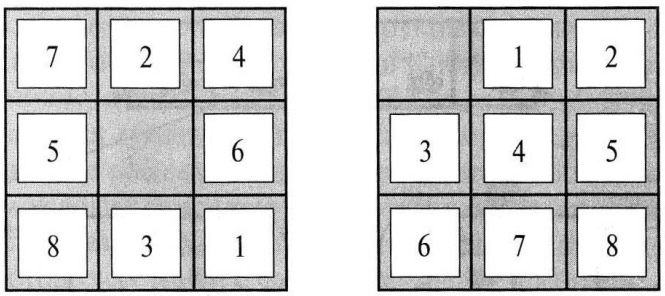

Рис.2.4. Изображения начального (слева) и целевого (справа) состояний для головоломки с восемью фишками.

Игрок должен, передвигая фишки (не извлекая их из коробки), установить их в заданное целевое состояние, показанное в правой части рис. 2.4. Головоломка с восемью фишками относится к классу хорошо структурированных проблем. Для того, чтобы решение этой проблемы могло быть поручено интеллектуальному агенту она должна быть сформулирована так, как это описано в подразделе 2.1.1.

- *Состояние*. Состоянием этой проблемы является формализованное описание, кото-

рое специфицирует положение каждой из восьми фишек и пустого участка в коробке.

- *Начальное состояние*. В качестве начального состояния может быть принято любое расположение фишек в коробке, например, расположение, показанное в левой части рис. 2.4.
- *Список действий доступных агенту*. Для получения последующего состояния достаточно к текущему состоянию применить одно из четырёх действий, изменяющих положение *пустого участка коробки*. Обозначим эти действия как `Left` (переместить пустой участок влево), `Right` (переместить пустой участок вправо), `Up` (переместить пустой участок вверх), `Down` (переместить пустой участок вниз).
- *Тест цели*. Проблема имеет одно целевое состояние. Тестом цели является процедура, которая сравнивает описание текущего состояния с описанием целевого состояния и останавливает процесс поиска, если они совпадают.
- *Стоимость пути*. Пусть каждое перемещение фишек имеет стоимость, равную единице, тогда стоимость решения равна общему количеству перемещений фишек, лучшим считается решение, полученное за меньшее количество перемещений.

Головоломка с восемью фишками относится к семейству задач со скользящими фишками, которые часто используются в искусственном интеллекте для проверки новых алгоритмов поиска. Пространство состояний головоломки с восемью фишками (коробка размером 3 x 3) имеет 181440 различных состояний и легко решается. Пространство состояний головоломки с пятнадцатью фишками (коробка размером 4 x 4) имеет около 1,3 триллиона различных состояний и решается при помощи наилучших алгоритмов поиска в среднем за несколько миллисекунд. Пространство состояний головоломки с двадцатью четырьмя фишками (коробка размером 5 x 5) имеет количество состояний около 10^{25}, и её всё ещё весьма нелегко решить при помощи современных компьютеров.

Проблема с восемью ферзями

Проблема с восемью ферзями относится к хорошо структурированным проблемам, но отличается от ранее рассмотренных тем, что ее *решением является* не путь в пространстве состояний, а *некоторое особое состояние, удовлетворяющее наперед заданным ограничениям*. Это особое состояние соответствует такому расположению восьми ферзей на шахматной доске, когда ни один из них не атакует друг друга. Ферзь атакует фигуру, которая находится на одной и той же с ним горизонтали, вертикали или диагонали. Предполагается, что фигуры последовательно, одна, за одной, выставляются на пустую доску. На рис. 2.5 показана неудачная попытка решения проблемы с восемью ферзями. Ферзь, находящийся на самой правой вертикали, атакован ферзём, который находится вверху слева.

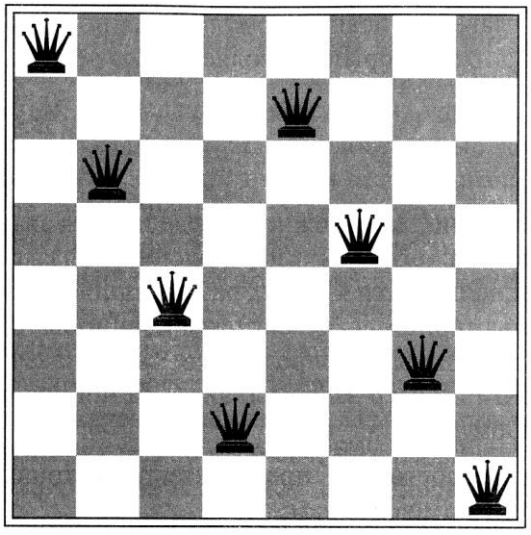

Рис.2.5. Попытка решения проблемы с восемью ферзями.

Поскольку решением проблемы является не путь в пространстве состояний, а некоторое особое состояние, удовлетворяющее наперед заданным ограничениям, то *стоимость пути отсутствует*. Агент ищет не последовательность действий, а расположение ферзей. Рассмотрим формулировку проблемы с восемью ферзями, как хорошо структурированную проблему.

- *Состояния*. Состоянием является формализованное описание, которое специфицирует любое расположение n ферзей на доске ($0 \leq n \leq 8$).
- *Начальное состояние*. Отсутствие ферзей на доске (пустая доска).
- *Список действий доступных агенту*. Отдельное действие устанавливает одного ферзя на любую пустую клетку шахматной доски.
- *Тест цели*. Тест цели представляет собой процедуру распознавания, которая позволяет определить, удовлетворяет ли текущее состояние заданным ограничениям (на доске находятся восемь ферзей и ни один из них не атакован).

Приведенная формулировка проблемы с восемью ферзями порождает пространство состояний с количеством состояний, которое определяется следующими расчетами: 64 x 63 x 62 x…x 57 ≈ 3 x 10^{14}. Можно предположить, что полученный большой размер пространства состояний является следствием неудачной формулировки проблемы. Попробуем уменьшить размер пространства состояний, переформулировав понятие состояния и список допустимых действий. Предыдущая формулировка разрешала размещать очередного ферзя на любой свободной клетке, в том числе и на той, где он будет атакован ранее размещенными ферзями. Целесообразно так сформулировать проблему, чтобы размещение очередного ферзя на клетке, которая атакуется, было запрещено. Новая формулировка понятия состояния и списка допустимых действий может быть следующей.

- *Состояния*. Состоянием является описание, которое специфицирует расположение n ферзей ($0 \leq n \leq 8$), по одному ферзю в каждой вертикали, при условии, что ни один ферзь не атакует друг друга. Предполагается, что ферзи выставляются на каждую ближайшую свободную левую вертикаль, начиная с крайней правой.
- *Список действий доступных агенту*. Отдельное действие устанавливает одного ферзя на любую свободную клетку соответствующей вертикали таким образом, чтобы он не был атакован каким-либо другим ферзём.

Подсчеты показывают, что новая формулировка проблемы позволяет сократить размер пространства состояний до 2057, и поиск решения значительно упрощается.

2.2.2. Проблемы реального мира

Проблема поиска маршрута

Проблема поиска маршрута решается во многих приложениях, таких как системы маршрутизации в компьютерных сетях, системы планирования военных операций, системы резервирования билетов для авиапутешествий и т.д. Обычно процесс формулировки таких проблем является трудоёмким. Рассмотрим упрощённый пример проблемы планирования авиапутешествий. Агент, который занимается планированием авиапутешествий, может получить формальное описание этой проблемы в следующем виде.

- *Состояния*. Каждое состояние представлено местонахождением (например, аэропортом) и текущим временем.
- *Начальное состояние*. Место и время начала путешествие.
- *Функция последующих состояний*. Эта функция возвращает состояния, которые являются результатом выполнения полётов, указанных в расписании с учётом времени, необходимого для перемещения между терминалами или аэропортами.
- *Тест цели*. Тест цели позволяет определить, находимся ли мы в месте назначения к заранее заданному времени.
- *Стоимость пути*. Зависит от стоимости билетов, времени ожидания, продолжительности полёта, таможенных и иммиграционных процедур, времени суток, типа самолёта, скидок для постоянных пассажиров и т.д.

В коммерческих консультационных системах планирования путешествий используется формулировка проблемы такого типа со многими дополнительными усложнения-

ми, которые требуются для учёта сложных систем определения платы за полет, применяемых в авиакомпаниях. Однако не все авиапутешествия проходят согласно плану. Поэтому качественная система должна предусматривать планы действий в непредвиденных ситуациях, такие, например, как страховочное резервирование билетов на альтернативные рейсы.

Проблема планирования обхода

Проблема планирования обхода тесно связана с проблемой поиска маршрута, но с одним важным исключением. Рассмотрим, например, проблему: «Посетить каждый город, показанный на рис. 2.2, по меньшей мере, один раз, начав и окончив путешествие в Бухаресте». Как и при поиске маршрута, действия соответствуют поездкам из одного смежного города в другой. Но пространство состояний является совершенно другим. Каждое состояние должно включать не только текущее местонахождение, но также и множество городов, которые посетил агент. Поэтому начальным состоянием должно быть «В Бухаресте; посещён: {Бухарест}», а типичным промежуточным состоянием – «В Васлуй; посещены: {Бухарест, Урзичени, Васлуй}». Тест цели должен предусматривать определение того, находится ли агент в Бухаресте, и посетил ли он все 20 городов.

Одной из проблем планирования обхода является *задача коммивояжера*, по условию которой каждый город должен быть посещён только один раз. Обычно разыскивается самый короткий путь обхода. Алгоритмы решения проблемы коммивояжера использовались для решения таких задач, как планирование перемещения свёрл при обработке печатных плат и организации работы системы снабжения в производственных цехах.

Проблема компоновки сверхбольших интегральных схем

Проблема компоновки сверхбольших интегральных схем требует позиционирования миллионов компонентов и соединений на микросхеме для минимизации площади, временных задержек, паразитных ёмкостей и максимизации выхода готовой продукции. Проблема компоновки следует за этапом логического проектирования и обычно подразделяется на две части: *компоновка ячеек* и *маршрутизация каналов*.

При компоновке ячеек компоненты схемы группируются по ячейкам, каждая из которых выполняет некоторую функцию. Каждая ячейка имеет постоянную форму и требует создания определённого количества соединений с остальными ячейками. Требуется разместить ячейки таким образом, чтобы они не перекрывались, и оставалось место для прокладки соединительных проводников. При маршрутизации каналов происходит поиск конкретного маршрута проводников через промежутки между ячейками. Эти задачи поиска являются чрезвычайно сложными, но затраты на их решения всегда оправдываются.

Проблема управления навигацией робота

Проблема управления навигацией робота представляет собой обобщение проблемы поиска маршрута. В этой проблеме вместо дискретного множества маршрутов рассматривается ситуация, в которой робот может перемещаться в непрерывном пространстве с бесконечным множеством возможных действий и состояний. Если требуется обеспечить перемещение робота по плоской поверхности, то, в ряде случаев, пространство может рассматриваться как двухмерное, а если робот оборудован верхними и нижними конечностями или колёсами, которыми также необходимо управлять, то пространство состояний становится многомерным. Изначальная сложность задачи усугубляется тем, что при управлении реальным роботом необходимо учитывать ошибки в работе сенсоров, а также отклонения в работе эффекторов.

Проблема автоматического упорядочения сборки

При решении проблемы сборки цель состоит в определении последовательности, в которой должны быть собраны детали некоторого агрегата. Если выбрана неправильная последовательность, то в дальнейшем нельзя будет найти способ добавления некоторой

детали к этой последовательности без отмены определённой части уже выполненной работы. Проверка возможности выполнения некоторого этапа в последовательности представляет собой сложную геометрическую задачу поиска, тесно связанную с задачей навигации робота. Поэтому одним из важных этапов решения проблемы упорядочения сборки является генерация допустимых последующих состояний. Любой практически применимый алгоритм должен предотвращать необходимость поиска во всём пространстве состояний, за исключением небольшой его части.

Одной из проблем сборки является проблема *проектирование молекулы белка*. Целью этой проблемы является определение последовательности аминокислот, способных сложится в трёхмерный белок с нужными свойствами и предназначенный для лечения некоторых заболеваний.

В последние годы выросла потребность в создании программных роботов, которые осуществляют *поиск в Internet*, находя ответы на вопросы, отыскивая ссылки на страницы или совершая коммерческие сделки. Это – хорошее приложение для применения методов поиска в проблемном пространстве, поскольку Internet легко представить концептуально в виде графа, состоящего из узлов-страниц, соединённых с помощью ссылок.

2.3. Поиск решения

Сформулировав проблему и, тем самым, задав пространство состояний этой проблемы, мы свели решение проблемы к поиску целевых состояний в пространстве состояний. В настоящем подразделе рассматривается метод поиска, основанный на построении учетной структуры, называемой *дерево поиска*. На рис. 2.6. показаны некоторые этапы развития дерева поиска в процессе решения проблемы поиска маршрута из Арада в Бухарест.

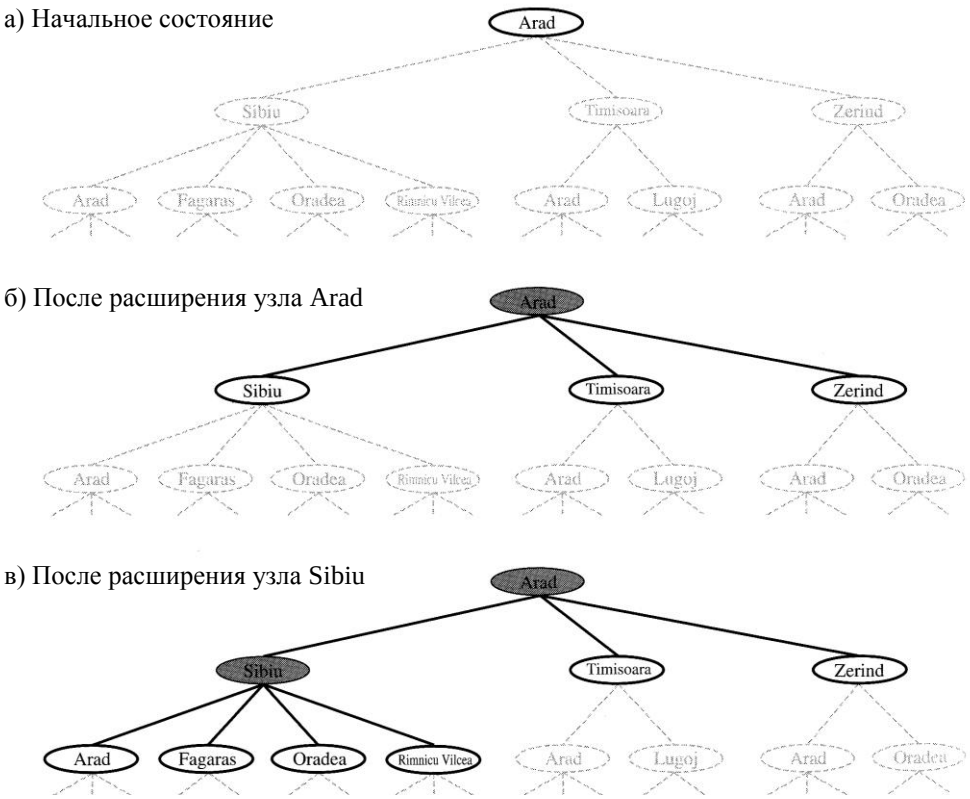

Рис. 2.6. Развитие дерева поиска при решении проблемы нахождения пути из Арада в Бухарест. Расширенные узлы помечены серым цветом. Узлы, которые были сформированы, но не расширены, выделены полужирным контуром. Узлы, которые ещё не были сформированы, обозначены светлыми пунктирными линиями.

Поиск начинается с того, что рассматривается единственное состояние пространства состояний, известное к моменту начала поиска. Этим состоянием является начальное состояние. Начальному состоянию в пространстве состояний соответствует *корневой узел* дерева поиска. На рис. 2.6 корневому узлу соответствует начальное состояние, которое мы обозначили `In(Arad)`. После формирования корневого узла дерева поиска необходимо проверить, не является ли состояние, которое ему соответствует, целевым. Эта проверка необходима для учета возможности решения тривиальной проблемы: «начав путешествие из города Арад, прибыть в город Арад». Для такой проверки используется тест цели.

Если обнаружилось, что начальное состояние не является целевым, то необходимо получить некоторое количество новых состояний пространства состояний и исследовать их. Генерация новых состояний осуществляется с помощью функции последующих состояний. Применение функции последующих состояний к текущему состоянию порождает новые состояния в пространстве состояний, которым соответствуют новые узлы на дереве поиска. Порождение новых узлов из текущего узла, в результате применения функции последующих состояний, и включение их в дерево поиска называется *расширением текущего узла*. На рис. 2.6, после расширения корневого узла получим три новых узла, которые соответствуют состояниям: `In(Sibiu)`, `In(Timisoara)` и `In(Zerind)`. Получив новые состояния пространства состояний необходимо выбрать одно из них для проверки с помощью теста цели и возможного последующего расширения. Правило, которое позволяет осуществить такой выбор, называется *стратегией поиска*.

Предположим, что, в соответствии с некоторой стратегией поиска, мы выбрали узел `Sibiu`. При помощи теста цели проверим, соответствует ли состояние `In(Sibiu)` целевому состоянию. Очевидно, что проверка даст отрицательный результат. Поэтому расширим узел `Sibiu` и получим четыре новых узла с состояниями: `In(Arad)`, `In(Faragas)`, `In(Oradea)` и `In(Rimnicu Vilcea)`. После этого можно протестировать любой из этих четырёх узлов, либо вернуться и протестировать узлы `Timisoara` или `Zerind`. Выбор одного из шести отмеченных узлов осуществляется в соответствии со стратегией поиска.

Таким образом, поиск целевого состояния в пространстве состояний заключается в том, что необходимо снова и снова выбирать, тестировать и расширять узлы, развивая при этом дерево поиска, до тех пор, пока не будет найден узел, соответствующий целевому состоянию или больше не останется узлов, которые можно было бы расширять.

Необходимо различать понятия пространство состояний и дерево поиска. В пространстве состояний проблемы поиска маршрута из Арада в Бухарест имеется только 20 состояний. Но количество путей в этом пространстве состояний – бесконечно, поэтому дерево поиска может иметь бесконечное количество узлов за счёт того, что некоторые узлы могут многократно повторяться на дереве поиска. Например, на рис. 2.6 дважды повторяется узел `Arad`.

Теперь, описанный выше, метод поиска в пространстве состояний запишем в виде программы. Эта программа приведена на рис. 2.7.

```
function TreeSearch(problem, strategy) returns solution/failure
   <инициализировать дерево поиска с использованием начального
    состояния problem>
   loop do
      if нет кандидатов для расширения then return failure
      <выбрать узел для расширения в соответствии со strategy>
      if узел содержит целевое состояние then return solution
      <расширить узел и добавить полученные узлы к дереву поиска>
```

Рис. 2.7. Программа обобщённого алгоритма поиска.

Программа, приведенная на рис. 2.7, и названная обобщенным алгоритмом поиска, отражает общий метод поиска в пространстве состояний, но содержит большое количест-

во неопределенности, препятствующей ее отображению в реальный программный код. Уточним эту программу.

Важным понятием, связанным с деревом поиска, является понятие *пограничного набора узлов*. Пограничным набором узлов называются узлы, которые были созданы, но не проверены при помощи теста цели. Каждый элемент пограничного набора узлов соответствует *узлу-листу* на дереве поиска. На рис. 2.6 (нижняя часть) к пограничному набору узлов относятся шесть узлов: `Timisoara`, `Zerind`, `Arad`, `Faragas`, `Oradea` и `Rimnicu Vilcea`. *Обобщённый алгоритм поиска становится более определенным, если представить пограничный набор узлов в виде очереди*. В этом случае узел, который должен быть проверен тестом цели, всегда находится вначале очереди, а стратегия поиска определяется способом добавления вновь созданных узлов в очередь пограничного набора. Введём набор процедур, обеспечивающих работу с очередью пограничного набора узлов.

- `MakeQueue(element,...)`. Создаёт очередь из заданных элементов.
- `Empty?(queue)`. Возвращает значение `true` в том случае, если в очереди нет элементов.
- `First(queue)`. Возвращает первый элемент очереди.
- `RemoveFirst(queue)`. Удаляет первый элемент из очереди.
- `Insert(element, queue)`. Вставляет новый элемент в очередь и возвращает результирующую очередь.
- `InsertAll(elements, queue)`. Вставляет множество элементов в очередь и возвращает результирующую очередь.

Теперь мы можем записать более определенную версию обобщённого алгоритма поиска, приведенную на рис. 2.9.

```
function TreeSearch(problem, queue) returns solution/failure

   queue ← Insert(MakeNode(InitialState(problem)) queue)
   loop do
      if Empty?(queue) then return failure
      node ← RemoveFirst(queue)
      if GoalTest(problem) применительно к State(node) успешен
         then return Solution(node)
      queue ← InsertAll(Expand(node, problem), queue)
```

Рис. 2.9. Уточнённая программа обобщённого алгоритма поиска.

Программа обобщённого алгоритма поиска представлена в виде процедуры с именем `TreeSearch` с входными параметрами `problem` (формулировка проблемы) и `queue` (очередь пограничного набора). Результатом работы программы может быть либо нахождение решения (`solution`), либо неудачное завершение (`failure`) если решение не найдено. Процедура `InitialState` получает, в качестве входного параметра, ссылку `problem` на формулировку проблемы и возвращает начальное состояние проблемы. Процедура `MakeNode` создает и возвращает корневой узел дерева поиска, соответствующий этому начальному состоянию. Процедура `Insert` создает очередь пограничного набора узлов `queue`, состоящую, первоначально, только из одного, корневого, узла.

Процедура `Empty?` Проверяет, есть ли элементы в очереди пограничного набора узлов. Если очередь пуста, то результатом работы программы является `failure` (неудача). Если в очереди пограничного набора есть элементы, то, при помощи процедуры `RemoveFirst` из очереди выделяется первый элемент и помещается в переменную `node`.

Процедура `GoalTest` является тестом цели, а процедура `State` возвращает состояние, соответствующее узлу `node`. Если тест цели определяет, что состояние, соответствующее узлу `node` является целевым, то при помощи процедуры `Solution` формируется решение. В противном случае узел `node` расширяется при помощи процедуры `Expand` и новые узлы добавляются в очередь пограничного набора узлов при помощи процедуры `InsertAll`. Затем все действия, начиная с процедуры `Empty?` циклически повторяются.

2.3.1. Критерии качества поисковых алгоритмов

Существует большое количество алгоритмов поиска в пространстве состояний, отличающихся, главным образом, стратегиями поиска. Принято характеризовать качество алгоритмов поиска при помощи следующих четырёх критериев.

- *Полнота*. Критерий полноты характеризует способность алгоритма гарантированно обнаруживать решение. Полный алгоритм гарантированно находит решение, если оно существует, а неполный – лишь с некоторой вероятностью.
- *Оптимальность*. Критерий оптимальности характеризует способность алгоритма находить наилучшее решение. Оптимальный алгоритм находит решение, имеющее наименьшую стоимость пути, а неоптимальный – решение, стоимость пути которого может быть больше, чем наименьшая стоимость.
- *Временная сложность*. Критерий временной сложности характеризует затраты процессорного времени на нахождение решения. Чем выше временная сложность, тем больше времени тратит процессор на нахождение решения.
- *Пространственная сложность*. Критерий пространственной сложности характеризует затраты основной памяти, необходимые для нахождения решения Чем выше пространственная сложность, тем больший объем памяти необходим для нахождения решения

Временная и пространственная сложность алгоритмов поиска связана со *сложностью пространства состояний*. Если пространство состояний представимо в виде графа, то сложность пространства состояний определяется размером графа, явно задающего это пространство состояний (смотри, например, рис. 2.2).

Однако, часто, пространство состояний является или очень большим или бесконечным. В этих случаях пространство состояний, задаётся неявно, в виде начального состояния и функции последующих состояний. При неявном задании пространства состояний сложность алгоритма поиска можно оценить при помощи следующих параметров, характеризующих дерево поиска:

- `b` – *коэффициент ветвления* или максимальное количество узлов дерева поиска, образующихся при расширении любого узла;
- `d` – *глубина самого поверхностного целевого узла* на дереве поиска;

Временная сложность может измеряться временем центрального процессора, которое необходимо для генерации узлов дерева поиска, а пространственная сложность – максимальным количеством узлов дерева поиска, хранимых в памяти.

Чтобы оценить эффективность алгоритма поиска, можно рассматривать *стоимость поиска*, которая обычно зависит от временной сложности, но может также включать выражение для оценки пространственной сложности.

2.4. Стратегии слепого поиска

В настоящем подразделе параграфе рассматриваются несколько стратегий поиска, относящихся к классу *стратегий слепого поиска*, которые называют, также, *стратегии не информированного поиска*. Все стратегии слепого поиска не используют дополнительную информацию о состояниях, кроме той, которая представлена при формулировке проблемы. Всё, на что способны стратегии слепого поиска – это систематически расширять узлы и отличать целевое состояние от нецелевого.

Стратегии поиска, использующие дополнительную информацию, позволяющую определить, является ли некоторое состояние узлов пограничного набора «ближе к целевому» по сравнению с другим называются стратегиями *эвристического поиска*, которые называют, также, *стратегии информированного поиска*.

2.4.1. Поиск в ширину

Стратегия поиска в ширину выбирает для расширения узел, имеющий наименьшую глубину. Иными словами, прежде чем проверять узлы на глубине `d + 1`, необходимо проверить все узлы на глубине `d`.

Для реализации поиска в ширину можно использовать уточнённый алгоритм поиска, приведенный на рис. 2.9, который работает с очередью пограничного набора узлов, упорядоченной по правилу FIFO (First-In-First-Out). Правило FIFO предусматривает, что все, *вновь сгенерированные, узлы добавляются в конец очереди*. Упорядочивание очереди пограничного набора узлов по правилу FIFO, гарантирует, что пока не будут обработаны узлы глубиной d, не будут обрабатываться узлы глубиной d + 1. Иными словами реализацию поиска в ширину можно осуществить вызовом функции:

```
TreeSearch(problem, FIFO-queue)
```

На рис.2.10 показано развитие дерева поиска при использовании стратегии поиска в ширину на примере бинарного дерева. Бинарным деревом называется дерево, имеющее коэффициент ветвления, равный двум.

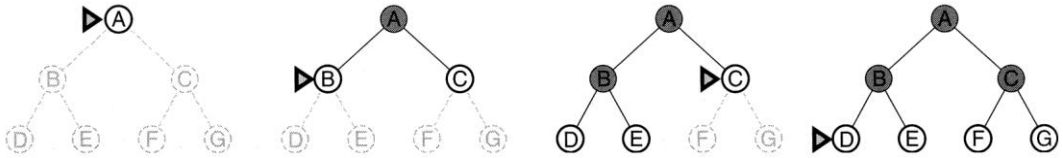

Рис. 2.10. Поиск в ширину на примере бинарного дерева. Узел подлежащий проверке, отмечен треугольником.

Охарактеризуем качество стратегии поиска в ширину при помощи четырёх критериев, приведенных в 2.3.1.

Очевидно, что *поиск в ширину является полным*. Если узел, соответствующий целевому состоянию, находится на некоторой глубине d, то поиск в ширину в конечном итоге его обнаружит.

Поиск в ширину находит самый поверхностный узел, соответствующий целевому состоянию. В общем случае этот узел не обязательно соответствует оптимальному решению. Таким образом, в общем случае, *поиск в ширину является неоптимальным*. Однако, самый поверхностный узел может соответствовать оптимальному решению, если стоимость пути выражается в виде неубывающей функции глубины. Например, в том случае, если все действия имеют одинаковую стоимость.

Для оценки алгоритма поиска на основе стратегии поиска в ширину по критериям временной и пространственной сложности рассмотрим гипотетическое дерево поиска, у которого коэффициент ветвления равен b. Предположим, что узел, соответствующий целевому состоянию, находится на глубине d и составим формулу, позволяющую вычислить общее количество узлов дерева поиска, как функцию от b и d.

Расширение корневого узла, описанного дерева поиска, порождает b узлов на единичной глубине. Расширение каждого из b узлов на единичной глубине порождает по b узлов на глубине равной двум. Таким образом, общее количество узлов на глубине равной двум равно $b \times b = b^2$. Аналогичные рассуждения приводят к выводу, что общее количество узлов на глубине, равной трем будет равно b^3 , и т.д.

Если узел, соответствующий целевому состоянию, находится на глубине d, то, в наихудшем случае, на глубине d придется расширить все узлы кроме последнего. Что породит на глубине d + 1 еще $b^{d+1} - b$ узлов.

Теперь легко записать искомую формулу. Наибольшее количество узлов, которое необходимо сгенерировать и обработать к моменту обнаружения узла, соответствующего целевому состоянию, определяется выражением

$$1 + b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b)$$

Для того, чтобы при помощи полученной формулы, можно было бы получить конкретные значения процессорного времени и объема основной памяти примем, что коэффициент ветвления b = 10. Примем, также, что в одну секунду процессор может сгенерировать 10000 узлов, и, что для хранения одного узла в памяти требуется 1000 байтов.

Этим предположениям соответствуют многие задачи поиска и возможности современных персональных компьютеров.

В таблице на рис. 2.11 приведены конкретные значения процессорного времени и объема основной памяти, необходимые для построения и хранения дерева поиска при поиске в ширину, если коэффициент ветвления равен 10, а глубина узла, соответствующего целевому состоянию варьируется от 2 до 14.

Глубина решения	**Количество узлов**	**Время процессора**	**Объем памяти**
2	1111	0,11 секунды	1 мегабайт
4	111100	11 секунд	106 мегабайтов
6	10^7	19 минут	10 гигабайтов
8	10^9	31 час	1 терабайт
10	10^{11}	129 суток	101 терабайт
12	10^{13}	35 лет	10 петабайтов
14	10^{15}	3523 года	1 эксабайт

Рис. 2.11. Потребности по времени и памяти для поиска в ширину.

На основании таблицы на рис. 2.11, можно заключить, что стратегия поиска в ширину характеризуется высокой пространственной и временной сложностью и алгоритм поиска, использующий эту стратегию, потребляет недопустимо большие ресурсы компьютера. Иными словами стоимость решения для этой стратегии слишком высока.

Как следует из таблицы на рис. 2.11 при поиске в ширину большей проблемой, по сравнению со значительным временем работы процессора, является удовлетворение потребностей алгоритма в необходимом объеме основной памяти. Затраты времени, равные 31 часу, не кажутся слишком значительными при ожидании решения важной проблемы с глубиной решения равной 8, но лишь немногие компьютеры имеют терабайт основной памяти, который для этого требуется. Однако, требования ко времени могут быть, также, недопустимо велики. Если рассматриваемая проблема имеет решение на глубине 12, то потребуется 35 лет чтобы его найти.

2.4.2. Поиск по критерию стоимости

Как было отмечено в 2.4.1, поиск в ширину становится оптимальным, если стоимость пути выражается в виде неубывающей функции глубины. С помощью простого дополнения можно создать стратегию поиска, которая является оптимальной для любой функции стоимости. Такая стратегия называется поиском по критерию стоимости.

Вместо проверки самого поверхностного узла, как это делает поиск в ширину, *поиск по критерию стоимости выбирает для проверки узел с наименьшей стоимостью пути*. Ясно, что *если стоимостью пути является глубина, то такой поиск идентичен поиску в ширину. Поэтому поиск в ширину можно рассматривать как частный случай поиска по критерию стоимости*.

При поиске по критерию стоимости учитывается не количество действий, имеющихся в пути, а их суммарная стоимость. Поэтому программа, использующая эту стратегию поиска, может войти в бесконечный цикл, если окажется, что расширен узел, при помощи действия, имеющего нулевую стоимость, и он снова будет выбран для проверки и расширения. Таким образом, для поиска по критерию стоимости можно гарантировать полноту только при условии, что стоимость каждого действия больше или равна некоторой небольшой положительной константе. Это условие является также достаточным для обеспечения оптимальности. Оно означает: (1) что стоимость пути всегда возрастает по мере прохождения по этому пути и (2) что алгоритм расширяет узлы в порядке возрастания стоимости пути. На рис. 2.12. приведен пример, иллюстрирующий работу стратегии при решении простой проблемы поиска пути.

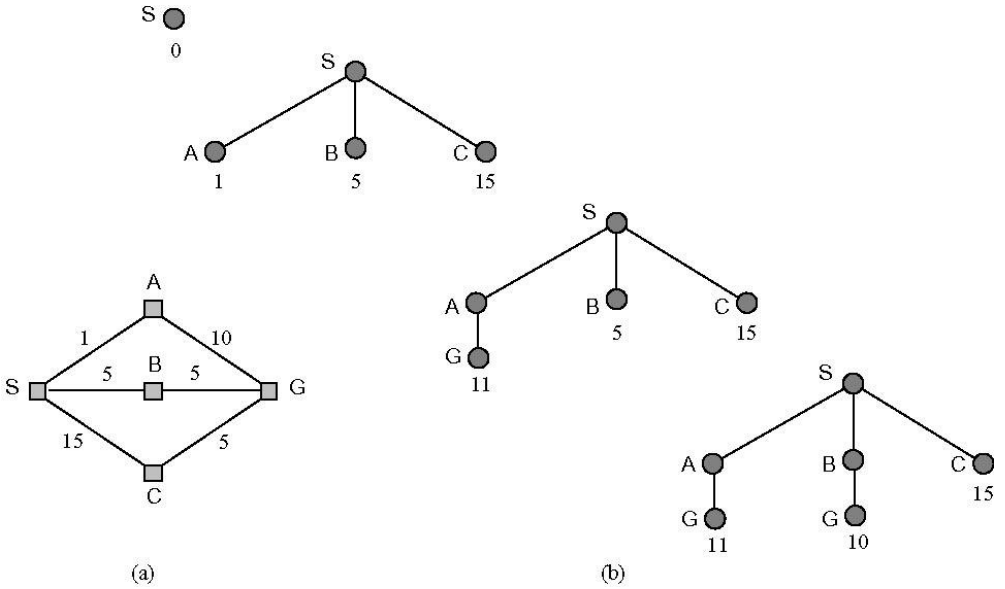

Рис. 2.12 Решение проблемы поиска пути по критерию стоимости.
(a) Пространство состояний, с указанием стоимости каждого действия.
(b) Развитие дерева поиска. Каждый узел-лист помечен стоимостью пути.

Рис. 2.12 иллюстрирует проблему поиска пути из точки S в точку G при известной стоимости каждого действия. Как следует из рис. 2.12. после расширения узла, соответствующего начальному состоянию, образуются пути, ведущие к узлам: A, B и C. Поскольку путь к узлу A обладает наименьшей стоимостью, то он выбирается для проверки. В результате, образуется путь SAG со стоимостью 11, который является одним из вариантов решения проблемы. Однако поиск решения продолжается, поскольку имеется отрезок пути меньшей стоимости (стоимость пути SB равна 5) и, следовательно, имеется шанс найти решение с меньшей стоимостью пути. После расширения узла B образуется ещё один вариант решения проблемы - путь SBG со стоимостью 10, который выбирается в качестве окончательного решения.

2.4.3. Поиск в глубину

При использовании стратегии *поиска в глубину*, так же, как и при поиске в ширину, вначале проверяется и расширяется корневой узел, затем все приемники корневого узла, затем приемники этих приемников и т.д. Однако, правило, которое определяет выбор узла подлежащего проверке, в каком то смысле противоположно тому, которое используется при поиске в ширину. При поиске в глубину для проверки выбирается узел, имеющий наибольшую глубину. Иными словам, из текущего набора пограничных узлов всегда выбирается самый глубокий узел.

Для реализации стратегии поиска в глубину можно использовать уточнённый алгоритм поиска, приведенный на рис. 2.9, который работает с очередью пограничного набора узлов, упорядоченной по правилу LIFO (Last-In-First-Out). Такая очередь в некоторых случаях называется *стеком*. Правило LIFO предусматривает, что все, *вновь сгенерированные, узлы добавляются в начало очереди*.

Рис. 2.13 иллюстрирует развитие дерева поиска для стратегии поиска в глубину на примере бинарного дерева. Поиск быстро переходит на самый глубокий уровень дерева поиска, на котором узлы не имеют потомков. По мере того, как обнаруживаются узлы, которые не могут соответствовать целевому состоянию, они удаляются из пограничного набора узлов. Затем поиск «возобновляется» со следующего самого поверхностного узла, который всё ещё имеет неисследованных потомков.

Поиск в глубину имеет очень скромные потребности в памяти. Он требует хранения только единственного пути от корня до листового узла, наряду с оставшимися нерасширенными узлами. После того как некоторый узел был расширен и исследованы все его потомки, он может быть удалён из памяти (см. рис. 2.13).

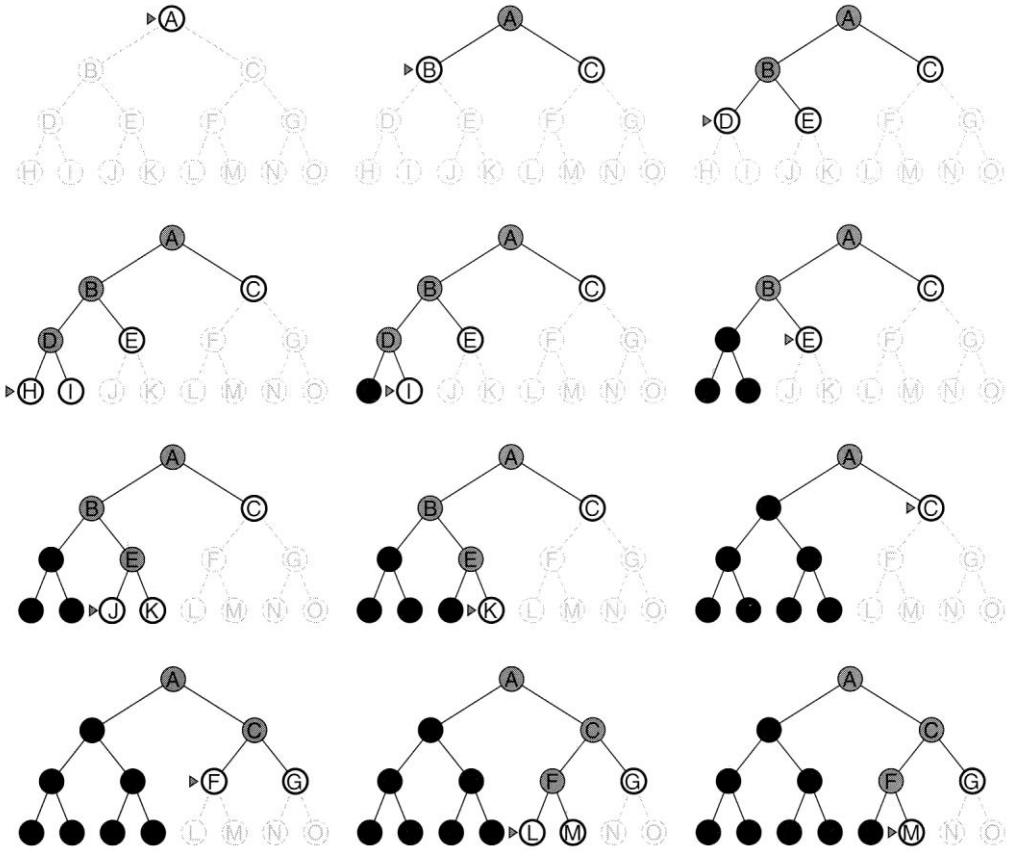

Рис. 2.13. Поиск в глубину на примере бинарного дерева.
Предполагается, что узлы на глубине 3 не имеют потомков и единственным целевым узлом является узел M. Узлы, которые были проверены и не имеют потомков могут быть удалены из памяти. Эти узлы обозначены чёрным цветом.

Для пространства состояний с коэффициентом ветвления `b` и максимальной глубиной `m` поиск в глубину требует хранения только `bm + 1` узлов. Используя те же предположения, как и при построении таблицы на рис. 2.11, и допуская, что узлы, находящиеся на той же глубине, что и целевой узел не имеют потомков, можно посчитать, что если целевой узел находится на глубине 12, то поиск в глубину требует всего 118 килобайтов памяти вместо 10 петабайтов при поиске в ширину. Таким образом, потребность в памяти уменьшается примерно в 10 миллиардов раз.

Недостатком поиска в глубину является то, что в нём может быть сделан неудачный выбор узла, в результате чего дерево будет углубляться по очень длинному (или даже бесконечному) пути, притом, что другой вариант мог бы привести к решению, находящемуся недалеко от корня. Например, на рис. 2.13 поиск в глубину потребовал бы исследования всего левого поддерева, даже если бы целевым узлом был узел C, находящийся в правом поддереве. А если бы целевым узлом был также узел J, с бо́льшей стоимостью пути чем узел C, то поиск в глубину возвратил бы решение, соответствующее этому узлу. Это означает, что *алгоритм поиска, использующий стратегию поиска в глубину, является неоптимальным*. Кроме того, если бы левое поддерево имело неограниченную глубину, но не содержало решений, то поиск так никогда бы и не закончился. Это означает, что *алгоритм поиска, использующий стратегию поиска в глубину, является неполным*.

2.4.4. Поиск с ограничением глубины

Проблему неограниченных деревьев можно решить, предусматривая применение во время поиска в глубину заранее определённого предела глубины `L`. Это означает, что все узлы на глубине `L` рассматриваются, как если бы они не имели потомков. Такой под-

ход называется *поиском с ограничением глубины*. По сути, при иллюстрации развития дерева поиска с использованием стратегии поиска в глубину на рис. 2.13 мы уже использовали эту идею. Для дерева поиска, на рис. 2.13, предела глубины `L` равен трем.

К сожалению, подход, основанный на введении предела глубины, до начала работы поискового алгоритма, вводит дополнительный источник неполноты. Например, если будет выбрано значение глубины `L < d`, то самый поверхностный целевой узел выходит за пределы глубины и никогда не будет найден. Такая ситуация вполне вероятна, если значение `d` неизвестно. Поиск с ограничением глубины будет неоптимальным при выборе значения `L > d`, поскольку поиск в глубину может рассматриваться как частный случай поиска с ограничением глубины, при котором $L = \infty$.

Выбор предела глубины основывается на более глубоком понимании проблемы. Например, на карте автомобильных дорог Румынии (см. рис. 2.2) имеется 20 городов. Поэтому известно, что если решение существует, то целевой узел должен находиться на глубине не более 19. Это означает, что одним из возможных вариантов является $L = 19$. Но при внимательном изучении карты можно обнаружить, что любой целевой город может быть достигнуть из любого другого города не более чем за 9 действий. Это число, известное как *диаметр пространства состояний*, представляет нам лучший предел глубины, который ведёт к более эффективному поиску с ограничением глубины. Но для большинства проблем диаметр пространства состояний остаётся неизвестным до тех пор, пока не будет решена сама задача.

2.4.5. Поиск в глубину с итеративным углублением

Стратегию *поиска в глубину с итеративным углублением* можно рассматривать как сочетание стратегии поиска с ограничением глубины со стратегией поиск в ширину. Стратегия предполагает постепенное увеличение предела глубины `L` (который вначале устанавливается равным 0, затем 1, затем 2 и т.д.) до тех пор, пока не будет найден целевой узел, а предел глубины не станет равным диаметру пространства состояний. Ясно, что в этом случае будет найден самый поверхностный целевой узел. Программа алгоритма поиска, построенная на основе стратегии поиска в глубину с итеративным углублением, приведена на рис. 2.14.

```
function IterativeDeepeningSearch(problem) returns solution/failure

   for depth ← 0 to ∞ do
      solution ← DepthLimitedSearch(problem, depth)
      if DepthLimitedSearch успешно завершён
      then return solution
```

Рис. 2.14. Программа поиска на основе стратегии поиска в глубину с итеративным углублением и использующая процедуру поиска с ограничением глубины `DepthLimitedSearch`.

На рис. 2.15 изображены четыре итерации поиска с использованием стратегии поиска в глубину с итеративным углублением. Узел M соответствует целевому состоянию.

Стратегия поиска в глубину с итеративным углублением похожа на стратегию поиска в ширину тем, что в каждой последующей итерации предел глубины увеличивается на единицу, а переход к последующей итерации осуществляется только после завершения предыдущей. Стратегия поиска с итеративным углублением похожа на стратегию поиска в глубину тем, что в пределах отдельной итерации для проверки выбирается узел, обладающей наибольшей глубиной. Поэтому, эта стратегия сочетает преимущества стратегий поиска в глубину и поиска в ширину. Как и стратегия поиска в глубину, стратегия с итеративным наблюдением характеризуется скромными требованиями к основной памяти компьютера. Как и стратегия поиска в ширину она является полной и оптимальной, если стоимость пути представляет собой неубывающую функцию глубины узла.

Стратегия поиска с итеративным углублением может, на первый взгляд, показаться расточительной, поскольку одни и те же узлы расширяются несколько раз. Но, как оказа-

лось, такие повторные расширения не являются слишком дорогостоящими. Причина этого состоит в том, что большинство узлов располагается на нижних уровнях, поэтому многократное формирование узлов на верхних уровнях не имеет большого значения.

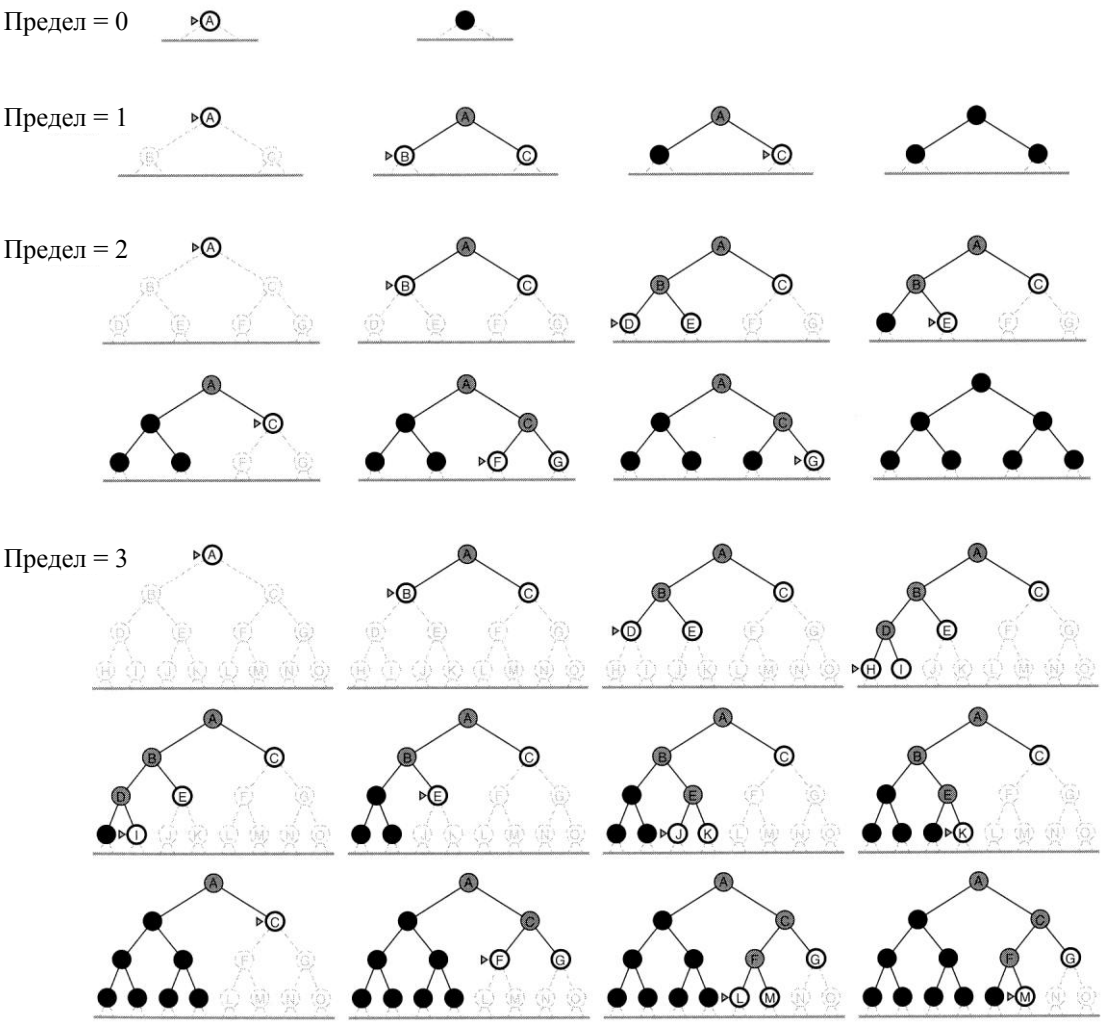

Рис. 2.15. Четыре итерации поиска с итеративным углублением на примере бинарного дерева

2.4.6. Двунаправленный поиск

В основе стратегии *двунаправленного поиска* лежит идея, заключающаяся в том, что можно одновременно проводить два поиска: (1) в прямом направлении от начального состояния и (2) в обратном направлении от целевого состояния. Поиск останавливается после того, как два процесса «встретятся». Встреча происходит в тот момент, когда оба процесса сгенерируют один и тот же узел. Рис. 2.16 иллюстрирует эту идею.

Двунаправленный поиск должен быть более экономичным, чем поиск в ширину, поскольку значение $b^{d/2} + b^{d/2}$ (количество узлов в двух деревьях при завершении поиска) значительно меньше, чем b^d (количество узлов в одном дереве при завершении поиска).

При реализации стратегии двунаправленного поиска необходимо осуществлять проверку каждого узла перед его расширением для определения того, не находится ли он в пограничном наборе противоположного дерева поиска. В случае положительного результата проверки, решение (путь от начального узла к целевому узлу) найдено. Например, если узел, соответствующий целевому состоянию находится на глубине $d = 6$ и в обоих направлениях используется стратегия поиска в ширину, то в худшем случае оба дерева встретятся, если в каждом из них будут расширены все узлы на глубине 3, кроме

одного. Это означает, что при `b = 10` на обоих деревьях будет сформировано 22200 узла, а не 11111100, как при обычном поиске в ширину.

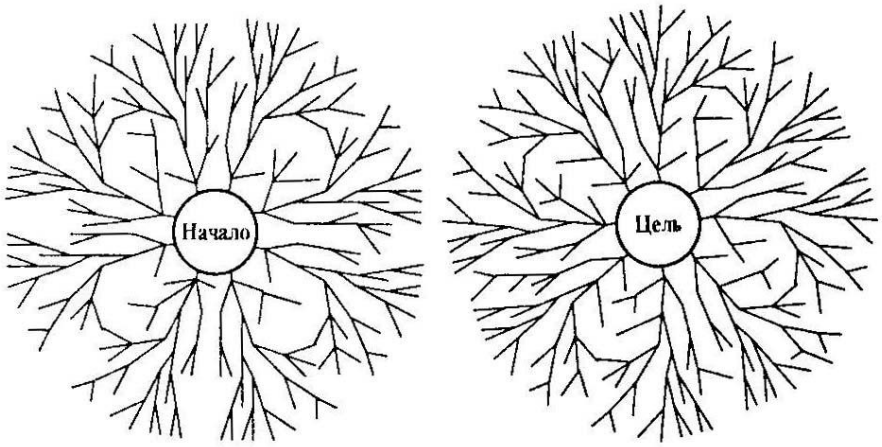

Рис. 2.16. Схематическое представление двунаправленного поиска на той стадии, когда он должен вскоре успешно завершиться после того, когда одна из ветвей, исходящих из начального узла встретиться с ветвью, исходящей из целевого узла.

Алгоритм поиска, использующий стратегию двунаправленного поиска, является полным и оптимальным, если оба процесса поиска используют стратегию поиска в ширину и используют одинаковую функцию стоимости пути. При различных стратегиях поиска и способов оценки стоимости пути в прямом и обратном направлениях стратегия двунаправленного поиска может характеризоваться отсутствием полноты, оптимальности или того и другого.

Организовать двунаправленный поиск не так легко, как кажется на первый взгляд. Допустим, что *предшественниками* узла `n`, определяемыми с помощью функции `Pred(n)`, являются все те узлы, для которых `n` служит потомком. Нахождение функции `Pred(n)`, в ряде случаев, является сложной задачей.

В головоломке с восемью фишками и при поиске маршрута из города Арад в город Бухарест имеется только одно целевое состояние, поэтому обратный поиск весьма напоминает прямой поиск. Если же проблема содержит несколько целевых состояний, то необходимо решить задачу выбора одного из целевых состояний для организации двунаправленного поиска.

2.5. Поиск с частичной информацией

Проведенные рассуждения относительно поиска в пространстве состояний базировались на предположении, что среда является полностью наблюдаемой и детерминированной и что агент располагает информацией о том, каковы последствия каждого действия. При таких предположениях агент может точно прогнозировать состояние, в котором он окажется в результате выполнения некоторой последовательности действий. Поэтому, после выполнения очередного действия его восприятия не предоставляют ему новой информации о состоянии среды, а лишь подтверждают прогноз. Но что произойдёт, если знания о состояниях среды или последствиях действий являются неполными. Разные типы неполноты знаний трансформируют хорошо структурированную проблему в одну из следующих проблем.

- *Проблема с отсутствующими сенсорами*. Если агент не имеет сенсоров, то он не знает в каком состоянии он находится и не может прогнозировать состояние, в котором он окажется после применения некоторого действия, но может предположить, что находиться в одном из нескольких возможных начальных состояний и, что каждое действие переводит его в одно из нескольких возможных последующих состояний.
- *Проблема с непредвиденными ситуациями*. Если среда наблюдаема лишь частично или последствия действий неизвестны агенту, то после каждого выполненного дей-

ствия восприятие агента определяет непредвиденную ситуацию, к которой необходимо подготовиться при помощи соответствующего плана. Проблема называется *обусловленной сторонним воздействием*, если непредвиденная ситуация вызвана действиями другого агента.

- *Исследовательская проблема.* Если состояния среды и действия агента в среде неизвестны, то агент должен действовать так, чтобы их обнаружить. Исследовательская проблема может рассматриваться как крайний случай проблемы с непредвиденными ситуациями.

2.5.1. Проблема с отсутствующими сенсорами

В качестве примера проблемы с отсутствующими сенсорами рассмотрим пространство состояний агента-пылесоса. Напомним, что пространство состояний проблемы, которую решает агент-пылесос, состоит из восьми состояний, изображенных на рис. 2.17.

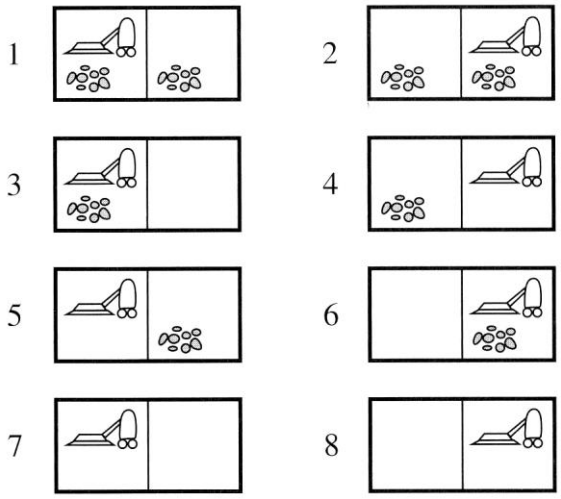

Рис. 2.17. Восемь возможных состояний мира агента-пылесоса.

Агенту доступны три действия (`Left`, `Right` и `Suck`), а цель состоит в том, чтобы был убран весь мусор, что эквивалентно достижению состояния 7 или 8. Если среда, в которой оперирует агент-пылесос, наблюдаема и детерминирована, то эта проблема легко решается при помощи любой из описанных стратегий поиска. Например, если начальным является состояние 5, то последовательность действий [`Right`, `Suck`] обеспечивает достижения целевого состояния 8.

Предположим, теперь, что среда является детерминированной, но агент *не имеет сенсоров*. В таком случае агент знает только, что его начальным состоянием является одно из состояний множества состояний {1, 2, 3, 4, 5, 6, 7, 8}. На первый взгляд может показаться, что попытки агента предсказать будущую ситуацию окажутся бесполезными, но на самом деле он может сделать это вполне успешно. Поскольку агент знает, к чему приводят его действия, то он может, например, определить, что действие `Right` вызовет переход в одно из состояний {2, 4, 6, 8}, а последовательность действий [`Right`, `Suck`] всегда оканчивается в одном из состояний {4, 8}. Наконец, последовательность действий [`Right`, `Suck`, `Left`, `Suck`] гарантирует достижение целевого состояния 7, независимо от того, каковым является начальное состояние. Агент может *принудительно перевести* среду в состояние 7, даже если ему не известно с какого состояния он начинает. Таким образом, если среда не является полностью наблюдаемой, то агент должен уметь прогнозировать, в какое множество состояний (а не в единственное состояние) он может попасть из текущего состояния. Будем называть каждое такое множество состояний *доверительным состоянием*.

Таким образом, для решения проблемы с отсутствующими сенсорами агент должен выполнять *поиск в пространстве доверительных состояний*. Первоначальное состояние является доверительным состоянием, а каждое действие отображает одно доверительное

состояние в другое. Любой путь объединяет несколько доверительных состояний, а решением является путь, который ведёт к такому доверительному состоянию, все элементы которого представляют собой целевые состояния. На рис. 2.18 показано пространство достижимых доверительных состояний для детерминированной среды агента-пылесоса у которого отсутствуют сенсоры. Как видно, на рис. 2.18, существует только 12 достижимых доверительных состояний.

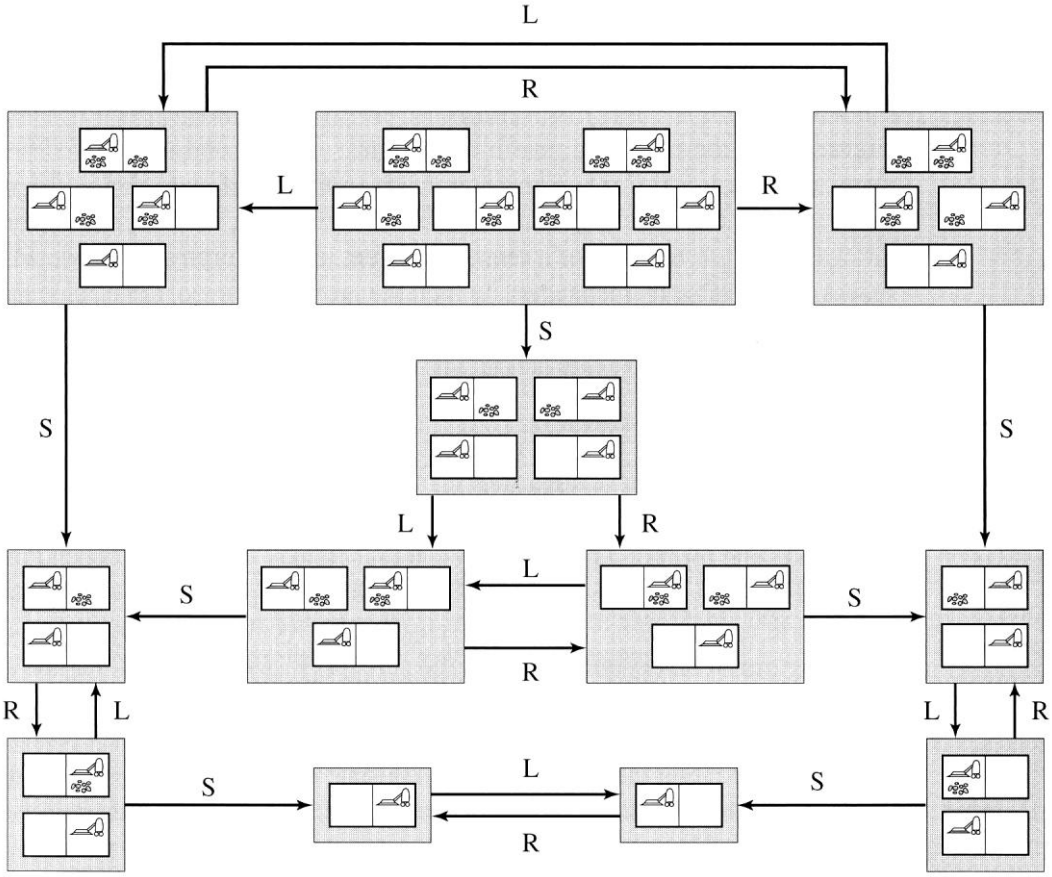

Рис. 2.18. Пространство доверительных состояний для проблемы, которую решает агент-пылесос, у которого отсутствуют сенсоры.

На рис. 2.18 каждый затенённый прямоугольник соответствует одному доверительному состоянию. В любой момент времени агент-пылесос знает в каком доверительном состоянии он находится, но не знает в каком именно физическом состоянии этого доверительного состояния он находится. Первоначальным доверительным состоянием (с полным незнанием ситуации) является верхний центральный прямоугольник. Действия обозначены дугами с метками, а петли, обозначающие возврат в одно и то же доверительное состояние опущены.

Упражнения

2.1. Сформулируйте для каждой из следующих проблем её компоненты: состояние, начальное состояние, функцию последующих состояний (или множество допустимых действий), тест цели, стоимость пути. Выберите формулировку, которая является достаточно точной, чтобы её можно было успешно реализовать.

a. Необходимо раскрасить плоскую карту, используя только четыре цвета, таким образом, чтобы никакие два смежных региона не имели один и тот же цвет.

b. Обезьяна, имеющая рост 90 сантиметров, находится в комнате, где под потолком, на высоте 2,4 метра, подвешено несколько бананов. В комнате находятся два ящика высотой по 75 сантиметров каждый, которые можно пере-

двигать, ставить друг на друга и на которые можно залезать. Обезьяна должна найти способ достать бананы.

c. Имеются три кувшина, с ѐмкостью 12 литров, 8 литров и 3 литра, а также водопроводный кран. Кувшины можно заполнять или опорожнять, выливая воду из одного кувшина в другой или на землю. Необходимо отмерить ровно один литр.

2.2. Рассмотрите пространство состояний, в котором начальным состоянием является число 1, а функция последующих состояний для состояния `n` возвращает два состояния: числа `2n` и `2n + 1`.

a. Нарисуйте часть дерева поиска соответствующее состояниям 1 - 15.

b. Предположим, что целевым состоянием является 11. Перечислите последовательности, в которых будут расширяться узлы при поиске в ширину, поиске с ограничением глубины, с пределом 3, и поиске с итеративным углублением.

c. Будет ли подходящим для решения этой проблемы двунаправленный поиск? Если да, то опишите подробно, как действовал бы этот метод поиска.

d. Каковым является коэффициент ветвления в каждом направлении двунаправленного поиска?

2.3. Задача «*миссионеры и каннибалы*» формулируется следующим образом. Три миссионера и три каннибала находятся на одной стороне реки, где также находится лодка, которая может выдержать одного или двух человек. Найдите способ перевезти всех на другой берег реки, никогда не оставляя где-либо группу миссионеров, которую превосходила бы по численности группа каннибалов.

a. Сформулируйте проблему «миссионеры и каннибалы», как хорошо структурированную проблему.

b. Постройте дерево поиска, используя стратегию поиска в ширину, и найдите решение проблемы «миссионеры и каннибалы».

2.4. Рассмотрите пространство состояний проблемы поиска маршрута в виде поверхности глобуса, на которую нанесена координатная сетка в виде параллелей и меридианов. Отдельная координата представляет собой состояние этого пространства состояний. Рассмотрите три случая, когда параллели и меридианы нанесены с точностью до: (1) десяти градусов, (2) одного градуса, и (3) одной минуты.

a. Сформулируйте эту задачу, как хорошо структурированную проблему.

b. Определите количество состояний этого пространства состояний для каждого из трѐх случаев.

c. Чему равен коэффициент ветвления для полюса и остальных состояний пространства состояний.

d. Для первого случая точности нарисуйте развитие дерева поиска в ширину. Считайте, что начальным состоянием является полюс, а искомое состояние находится на глубине 3.

3. СТРАТЕГИИ ЭВРИСТИЧЕСКОГО ПОИСКА И АЛГОРИТМЫ ЛОКАЛЬНОГО ПОИСКА

Алгоритмы поиска, использующие стратегии слепого поиска, позволяют находить решение проблемы путём систематической генерации новых состояний из пространства состояний и проверки этих состояний при помощи теста цели. К сожалению, полные стратегии слепого поиска, в большинстве случаев, обладают высокой стоимостью решения и потребляют значительные компьютерные ресурсы. В настоящем разделе рассматриваются стратегии эвристического поиска, использующие дополнительные знания относительно пространства состояний, которые обеспечивают более эффективный поиск решения проблемы.

3.1. Алгоритмы эвристического поиска

Будем изучать алгоритмы эвристического поиска, относящиеся к классу алгоритмов *поиска по первому наилучшему совпадению*. Поиск по первому наилучшему совпадению осуществляется обобщённым алгоритма поиска (см. рис. 2.7 и 2.9), в котором узел, подлежащий расширению выбирается в соответствии со значением функции `f(n)`, оценивающей «близость» текущего узла к целевому узлу. Для проверки и расширения выбирается узел с наименьшим значением этой функции.

Существует целое семейство алгоритмов поиска по первому наилучшему совпадению, отличающихся видом оценочной функции `f(n)`. Функция `f(n)`, как правило, состоит из нескольких компонентов. Во всех случаях ключевым компонентом функции `f(n)` является *эвристическая функция*, обозначаемая как `h(n)`

Эвристические функции (или просто *эвристики*) представляют собой наиболее общую форму, в которой к алгоритму поиска подключается дополнительная информация, позволяющая выбрать более вероятное направление поиска. Термин «эвристический» означает «полученный на основе опыта или интуиции». Для каждой проблемы, характер которой позволяет использовать эвристические функции, существуют свои специфические эвристики, которые, как правило, неприменимы при решении других проблем. Поскольку эвристическая функция `h(n)` возвращает значение, смысл которого – *оценка стоимости пути от узла `n` до целевого узла*, то ясно, что если `n` – целевой узел, то `h(n) = 0`. Отметим, также, что эвристическая функция принимает в качестве входного параметра некоторый узел, но зависит только от состояния, соответствующего данному узлу.

Например, для того, чтобы при решении проблемы поиска маршрута их города Арад в город Бухарест можно было бы использовать эвристический алгоритм поиска необходимо разработать эвристику, специфическую для этой проблемы. Такой эвристикой может быть функция, которая возвращает кратчайшее расстояние по прямой от текущего города до Бухареста.

3.1.1. Жадный поиск по первому наилучшему совпадению

Стратегия *жадного поиска по первому наилучшему совпадению* использует оценочную функцию в виде

`f(n) = h(n).`

Таким образом, в стратегии жадного поиска по первому наилучшему совпадению «близость» текущего узла к целевому оценивается только при помощи эвристической функции.

Проиллюстрируем на примере «работу» стратегии жадного поиска по первому наилучшему совпадению при решении проблемы поиска маршрута из Арада в Бухарест. В качестве эвристической функции используем функцию, возвращающую *расстояние по прямой (Straight Line Distance – SLD) от текущего города до Бухареста*. Обозначим эту эвристическую функцию как h_{SLD}. Тогда, например, если входным параметром этой функции будет `In(Arad)`, то она возвратит число `366`.

h_{SLD}(In(Arad)) = 366.

На рис. 3.1 функция h_{SLD} задано таблично.

Обозначение узла	Наименование города	Расстояние по прямой до Бухареста	Обозначение узла	Наименование города	Расстояние по прямой до Бухареста
Arad	Арад	366	Mehadia	Мехадия	241
Bucharest	Бухарест	0	Neamt	Нямц	234
Craiova	Крайова	160	Oradea	Орадя	380
Drobeta	Дробета	242	Pitesti	Питешти	100
Eforie	Эфорие	161	RimnicuVilcea	Рымнику-Вылча	193
Faragas	Фэгэраш	176	Sibiu	Сибиу	253
Giurgiu	Джурджу	77	Timisoara	Тимишоара	329
Hirsova	Хыршова	151	Urziceni	Урзичени	80
Iasi	Яссы	226	Vaslui	Васлуй	199
Lugoj	Лугож	250	Zerind	Зеринд	374

Рис. 3.1. Значения эвристической функции h_{SLD}.

На рис. 3.2 показан процесс развития дерева поиска маршрута из Арада в Бухарест при использовании стратегии жадного поиска по первому наилучшему совпадению с использованием эвристической функции h_{SLD}.

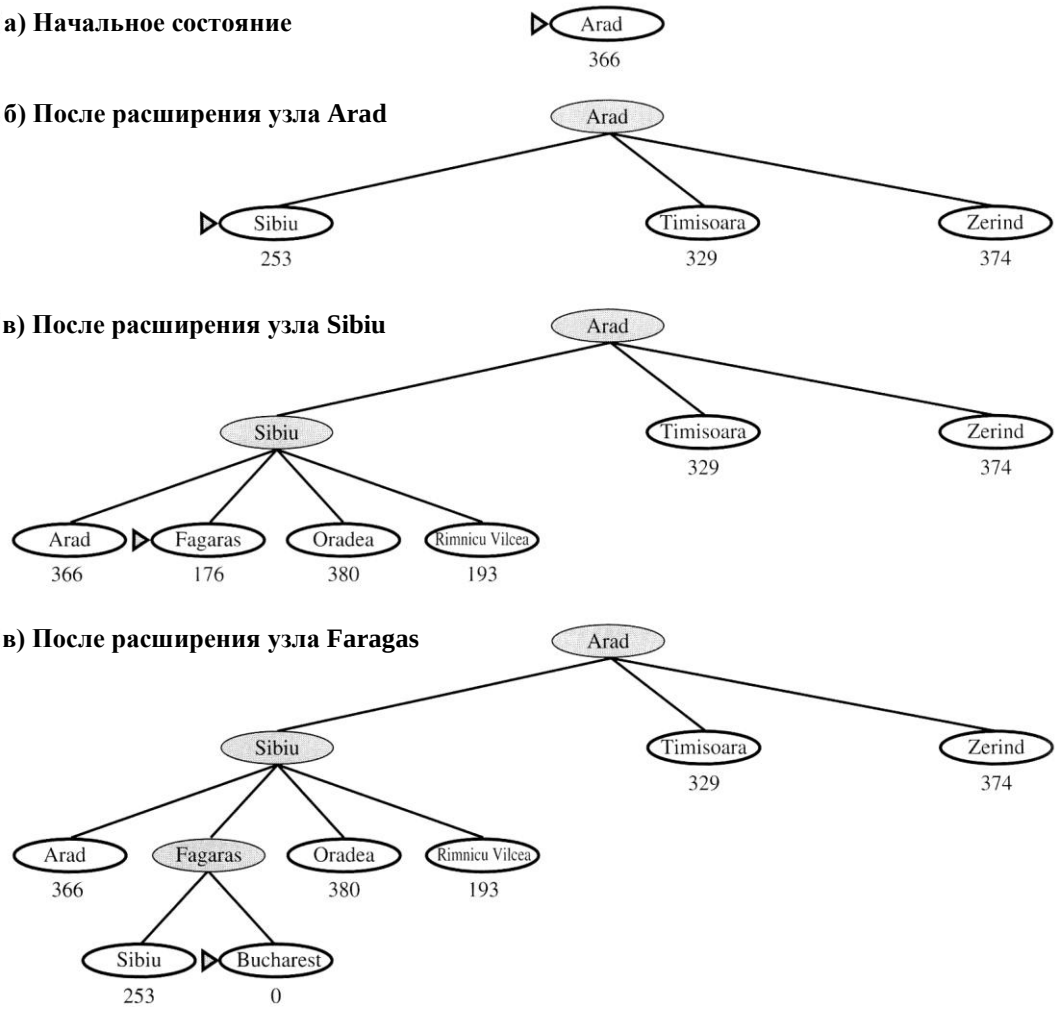

Рис. 3.2. Рост дерева поиска при решении проблемы поиска маршрута из Арада а Бухарест с использованием стратегии жадного поиска по первому наилучшему совпадению и эвристической функции h_{SLD}. Узлы обозначены значениями эвристической функции.

Стратегия поиска по первому наилучшему совпадению подсказывает, что после проверки и расширения узла Arad, необходимо проверять узел Sibiu, поскольку он обладает наименьшим значением h_{SLD}. Содержательно это означает, что город Сибиу находится ближе к Бухаресту, чем города Зеринд или Тимишоара. После проверки и расширения узла Sibiu должен быть проверен и расширен узел Fagaras, поскольку, теперь, он обладает наименьшим значением функции h_{SLD}. Расширение узла Fagaras порождает узел Bucharest, который является целевым.

Использование стратегии эвристического поиска при решении этой проблемы позволяет найти решение без проверки и расширения узлов, не находящихся на пути к целевому узлу. Однако, очевидно, что найденное решение не является наилучшим. Путь до Бухареста через города Сибиу и Фэгэраш на 32 километра длиннее, чем путь через города Рымнику-Вылча и Питешти (см. рис. 2.2). Это замечание объясняет, почему изучаемый алгоритм называется «жадным». На каждом шаге он пытается подойти к цели как можно ближе (фигурально выражаясь, «захватить как можно больше») учитывая оценку «расстояния до цели» только на основе значения эвристической функции для текущего узла и не учитывая оценку всего пути.

Стратегия жадного поиска по первому наилучшему совпадению напоминает стратегию поиска в глубину в том смысле, что этот алгоритм ведет поиск целевого узла по единственному пути, но *возвращается к предыдущим узлам после попадания в тупик*. Алгоритм обладает теми же недостатками, что и алгоритм поиска в глубину: он *не является оптимальным*, к тому же он – *не полный*, поскольку способен отправиться по бесконечному пути и не вернуться, чтобы опробовать другие возможности.

3.1.2 Поиск A*: минимизация суммарной оценки стоимости решения

Наиболее широко известная разновидность поиска по первому наилучшему совпадению называется поиском A*, читается как «*поиск А звёздочка*». В этой стратегии используется оценочная функция f(n), состоящая из двух компонентов: функции стоимости пути g(n) и эвристической функции h(n).

```
f(n) = g(n) + h(n)
```

Функция g(n) возвращает точную стоимость пути от начального узла до узла n, а функция h(n) – оценку стоимости пути от узла n до целевого узла.

Если эвристическая функция h(n) удовлетворяет некоторым условиям, то стратегия поиска A* становится и полной и оптимальной. Стратегия поиска A* будет оптимальной, при условии, что h(n) представляет собой *допустимую эвристику*. Допустимой эвристической функцией будем называть такую функцию, которая никогда не переоценивает стоимость достижения целевого узла. Иными словами всегда возвращает наименьшее значение этой стоимости. А поскольку g(n) – точная стоимость пути к узлу n, то из этого непосредственно следует, что функция f(n) никогда не переоценит истинную стоимость решения, проходящего через узел n.

Примером допустимой эвристической функции является функция, возвращающая кратчайшее расстояние по прямой h_{SLD}, которая уже использовалась для иллюстрации стратегии жадного поиска. Очевидно, что функция возвращающая расстояние по прямой является допустимой эвристикой, поскольку кратчайший путь между любыми двумя точками лежит на прямой, соединяющей эти точки, и, следовательно, h_{SLD} никогда не переоценивает длину пути.

На рис. 3.3 показан процесс развития дерева поиска маршрута из Арада в Бухарест при использовании стратегии A*. Возвращаемое значение функции g(n) определяется при помощи рис.2.2, а возвращаемое значения h_{SLD} – при помощи таблицы на рис. 3.1.

Отметим, что, на рис. 3.3, узел Bucharest впервые появляется в пограничном наборе узлов, на этапе д), но не выбирается для проверки, поскольку для него значение оценочной функции f(n) (450) выше, чем значение оценочной функции для узла Pitesti (417) и, следовательно, может существовать решение, при котором путь проходит через узла Pitesti. Поэтому алгоритм не останавливается на решении со стоимостью, равной 450, а продолжает поиск.

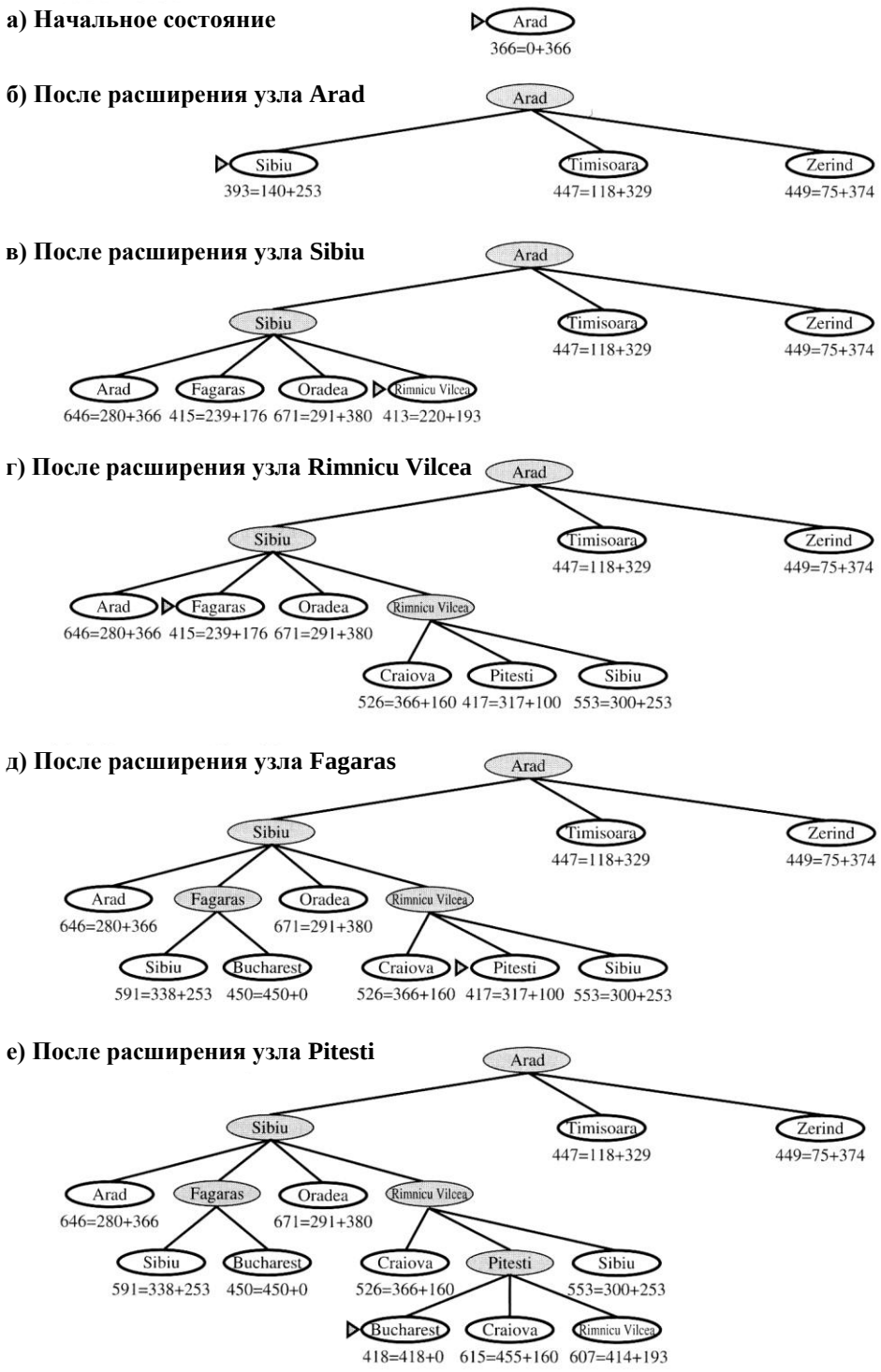

Рис. 3.3. Рост дерева поиска при решении проблемы поиска маршрута из Арада а Бухарест с использованием стратегии A*. Узлы обозначены значениями функции `f(n) = g(n) + h(n)`.

Рассмотренный пример поиска решения проблемы с использованием стратегии A* показывает, что обобщенный алгоритм поиска на основе этой стратегии является оптимальным, если функция `h(n)` – допустимая эвристика.

Одним из главных недостатков стратегии поиска A* является большая потребность в ресурсе память. Это связано с тем, что эта стратегия предполагает хранение в памяти всего дерева поиска.

3.2. Примеры эвристических функций

Рассмотрим, в качестве примера, эвристические функции, которые можно использовать в стратегиях эвристического поиска решения головоломки с восемью фишками.

Как было отмечено в 2.2.1, в процессе поиска решения головоломки требуется передвигать фишки по горизонтали или по вертикали на пустой участок до тех пор, пока полученная конфигурации фишек не будет соответствовать целевой конфигурации. На рис. 3.4 приведены варианты начальной и целевой конфигурации фишек.

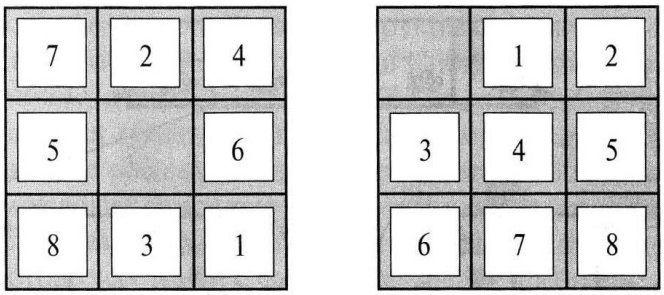

Рис.3.4. Начальное и целое состояния головоломки с восемью фишками.

В 2.2.1 приведена формальная формулировка этой проблемы. Представим теперь, что мы хотим найти её решение при помощи обобщенного алгоритма поиска, использующего одну из изученных стратегий эвристического поиска. Для этого нам необходима эвристическая функция, которая никогда не переоценивает количество перемещений, необходимых для достижения цели. Ниже описаны две эвристические функции, которые могут использоваться в стратегиях эвристического поиска решения головоломки с восемью фишками.

- $h_1(n)$ = <количество фишек, стоящих не в целевых позициях>. Функция $h_1(n)$ для каждого узла n возвращает целое положительное число, которое равно количеству фишек, находящихся не в целевых позициях. В левой части рис. 3.4 все восемь фишек находятся не в целевых позициях, поэтому функция h_1 для узла, соответствующего начальному состоянию возвращает число 8. Эвристическая функция h_1 является допустимой эвристикой, поскольку очевидно, что каждую фишку, находящуюся не на своём месте, необходимо переместить, по меньшей мере, один раз.
- $h_2(n)$ = <сумма расстояний всех фишек от их целевых позиций>. Функция $h_2(n)$ для каждого узла n возвращает целое положительное число, которое равно сумме расстояний всех фишек от их целевых позиций. Поскольку фишки не могут передвигаться по диагонали, то рассчитываемое расстояние представляет собой сумму горизонтальных и вертикальных расстояний. Такое расстояние иногда называют *расстоянием городских кварталов*, или *манхэттенским расстоянием*. Эвристическая функция h_2 также является допустимой, поскольку всё, что может быть сделано в одном ходе, состоит лишь в перемещении одной фишки на один шаг ближе к цели. Для начального состояния, изображенного в левой части рис. 3.4 манхэттенское расстояние равно: $h_2 = 3+1+2+2+2+3+3+2=18$.

М2

3.3. Алгоритмы локального поиска и задачи оптимизации

Алгоритмы поиска, рассмотренные выше, предназначались для нахождения решения проблем в виде пути в пространстве состояний от начального состояния к одному из целевых состояний. Однако, существует ряд проблем решением который является не путь в пространстве состояний, а некоторое особое состояние пространства состояний. Примером такой проблемы является проблема с восемью ферзями. Решением этой проблемы является некое особое расположение восьми ферзей на шахматной доске, а не последовательность действий, которые были использованы при расстановке ферзей на доске. К этому классу проблем относятся, также, многие проблемы реального мира, например, проблема компоновки сверхбольших интегральных схем.

Алгоритмы, используемые для решения таких проблем, называются *алгоритмами локального поиска*. Алгоритмы локального поиска, также как и алгоритмы поиска пути, осуществляют поиск в пространстве состояний и используют дерево поиска в качестве структуры, позволяющей вести учет состояний. Однако, поскольку при локальном поиске, путь к целевому состоянию не представляет интереса, то в алгоритмах локального поиска не требуется сохранения информации о всех возможных путях. Эти алгоритмы могут оперировать только текущим и соседним (по отношению к текущему) состояниями. Информация о путях, пройденных в процессе локального поиска, не сохраняется.

Алгоритмы локального поиска обладают двумя важными достоинствами, по сравнению с алгоритмами поиска пути. Во-первых, они используют очень небольшой и, как правило, постоянный объём памяти, и, во-вторых, они часто позволяют находить приемлемые решения в больших или бесконечных (непрерывных) пространствах состояний, для которых алгоритмы поиска пути не применимы.

Кроме решения проблем поиска особого состояния в пространстве состояний, алгоритмы локального поиска могут использоваться для решения классических *задач оптимизации,* решение которых заключается в поиске состояния, наилучшего с точки зрения *целевой функции*.

Для понимания алгоритмов локального поиска полезно оперировать понятием *ландшафта пространства состояний*, пример которого изображён на рис. 3.6.

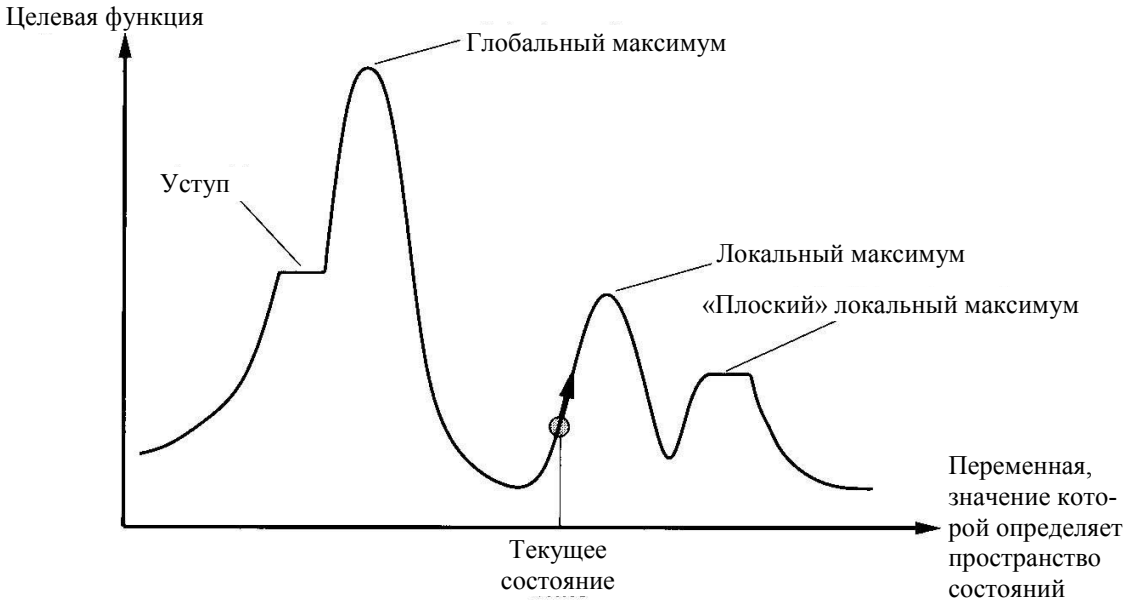

Рис.3.6. Ландшафт одномерного пространства состояний, в котором наивысшая точка соответствует наибольшему значению целевой функции.
Различные топографические элементы ландшафта рассмотрены в тексте.

Ландшафт пространства состояний можно рассматривать как с точки зрения эвристической функции стоимости (тогда задача заключается в поиске самой глубокой «впадины», или *глобального минимума*), или с точки зрения целевой функции (тогда задача заключается в поиске высочайшего «пика», или *глобального максимума*).

Алгоритмы локального поиска характеризуются *полнотой* и *оптимальностью*. Полный алгоритм локального поиска гарантирует нахождение целевого состояния, в том случае, если оно существует, а оптимальный алгоритм всегда находит наилучшее целевое состояние.

3.3.1. Поиск с восхождением к вершине

Простейшим алгоритмом локального поиска является алгоритм поиска *с восхождением к вершине*. Программа этого алгоритма приведена на рис. 3.7.

```
function HillClimbingSearch(problem) returns целевое состояние
   local variables: current, текущий узел
                    neighbor, ближайший узел

   current ← MakeNode(InitialState(problem))
   loop do
      <расширяем узел current>
      neighbor ← преемник узла current с наибольшим значением
                 целевой функции ObjectiveFunc
      if ObjectiveFunc(neighbor) ≤ ObjectiveFunc(current) then
         return State(current)
      current ← neighbor
```

Рис. 3.7. Алгоритм поиска с восхождением к вершине. На каждом шаге текущий узел заменяется наилучшим соседним узлом с наибольшим значением функции `ObjectiveFunc`.

Алгоритм представляет собой цикл, в котором постоянно происходит «перемещение» текущего состояния в направлении возрастания значения целевой функции, или «подъём» к максимуму по ландшафту состояний. Работа алгоритма заканчивается после достижения «пика», в котором ни одно из соседних состояний не имеет большего значения целевой функции. В алгоритме не предусмотрено хранение дерева поиска, поэтому для текущего узла необходимо регистрировать только состояние и соответствующее ему значение целевой функции. В алгоритме с восхождением к вершине не осуществляется прогнозирование за пределами состояний, которые являются соседними по отношению к текущему состоянию. Это напоминает восхождение на вершину холма в густом тумане.

Поиск с восхождением к вершине обычно называют *жадным локальным поиском*, поскольку в процессе его выполнения происходит захват самого хорошего соседнего состояния.

Поиск с восхождением к вершине попадает в тупиковую ситуацию, если достигается *локальный максимум* или *плато*.

Локальный максимум представляет собой пик, более высокий, чем соседние, но более низкий, чем глобальный максимум. Алгоритм с восхождением к вершине, который работает в окрестности локального максимума, обеспечивают продвижение вверх, к этому пику, но после его достижения останавливаются. Локальный максимум схематично показан на рис. 3.6.

Плато – это область в ландшафте пространства состояний, в которой значение целевой функции является постоянной для смежных состояний. Плато может представлять собой *плоский локальный максимум*, из которого не возможно продвижение вверх, или *уступ*, из которого возможно дальнейшее успешное продвижение (см. рис. 3.6). Поиск с восхождением к вершине может оказаться неспособным выйти за пределы плато.

Попав в тупиковую ситуацию, алгоритм поиска с восхождением к вершине не может осуществляться дальнейшее успешное продвижение по пространству состояний.

При нахождении решения проблемы с восемью ферзями и, начиная со *случайно сформированного* начального состояния, алгоритм поиска с восхождением к вершине попадает в тупиковую ситуацию в 86% случаях, и находит решение только в 14% случаев. Однако, он работает очень быстро, выполняя в среднем 4 цикла в случае успешного завершения и 3 цикла в случае попадания в тупиковую ситуацию.

Разработано несколько вариантов алгоритма поиска с восхождением к вершине. *Поиск с восхождением к вершине и перезапуском случайным образом* руководствуется широко известной рекомендацией: «Если первая попытка оказалась неудачной, пробуй снова и снова». В этом алгоритме предусмотрено проведение ряда поисков из *сформированных случайным образом начальных состояний* и остановка после достижения цели. Он является полным с вероятностью, достигающей единицы, хотя бы по той тривиальной причине, что в конечном итоге в качестве начального состояния может сформироваться одно из целевых состояний. Алгоритм поиска с восхождением к вершине и перезапуском случайным образом является эффективным применительно к задаче с восемью ферзями. Успех этого алгоритма в значительной степени зависит от ландшафта пространства со-

стояний. Если в нём есть лишь несколько локальных максимумов и плато, то поиск с восхождением к вершине и перезапуском случайным образом позволяет быстро найти хорошее решение.

3.3.2. Локальный лучевой поиск

Стремление преодолеть ограничения, связанные с нехваткой памяти в компьютерах предыдущих поколений, привело к тому, что в своё время предпочтение отдавалось алгоритмам локального поиска, предусматривающим хранение в памяти данных только об одном, текущем узле. Сейчас такой радикальный способ экономии памяти не является актуальным.

В алгоритме *локального лучевого поиска* предусмотрено отслеживание не одного состояния, а k состояний. Вначале формируются k начальных состояний *случайным образом*. Затем – преемники всех k состояний. Если какой-либо из этих преемников соответствует целевому состоянию, то алгоритм останавливается. В противном случае алгоритм выбирает k наилучших преемников и повторяет цикл. Выбор наилучшего преемника осуществляется таким же образом, как и в алгоритме с восхождением к вершине.

На первый взгляд может показаться, что локальный лучевой поиск представляет собой независимое и параллельное выполнение k алгоритмов с восхождением к вершине (k параллельных поисков с восхождением к вершине). Но это не так. При *локальном лучевом поиске параллельные потоки поиска обмениваются информацией*. Если в некотором потоке поиска вырабатывается несколько «хороших» преемников, а во всех других k−1 потоках вырабатываются «плохие» преемники, то алгоритм способен отказаться от бесперспективных направлений поиска и сосредоточить ресурсы на том потоке, где достигнут наибольший прогресс.

В своей простейшей форме локальный лучевой поиск может страдать от отсутствия разнообразия между k состояниями, поскольку все эти состояния способны быстро сосредоточиться в небольшом регионе пространства состоянии. В результате поиск начинает немногим отличаться от поиска с восхождением к вершине. Этот недостаток позволяет устранить вариант лучевого поиска, называемый *стохастическим лучевым поиском*, который аналогичен стохастическому поиску с восхождением к вершине. При стохастическом лучевом поиске вместо выбора наилучших k преемников из множества преемников-кандидатов происходит выбор k преемников случайным образом, а вероятность выбора преемника возрастает с возрастанием значения целевой функции.

3.3.3. Генетические алгоритмы

Генетический алгоритм представляет собой вариант стохастического лучевого поиска, в котором состояния-преемники формируются путём использования двух родительских состояний, а не путём модификации единственного состояния. В нём просматривается такая же аналогия с естественным отбором, как и в стохастическом лучевом поиске, за исключением того, что теперь моделируется не бесполое, а половое воспроизводство.

Как и при лучевом поиске, работа генетического алгоритма начинается с формирования случайным образом k состояний, называемых *популяцией*. Каждое состояние, или *индивидуум популяции*, представляется строкой символов. Например, состояние проблемы с восемью ферзями должно определять положение всех восьми ферзей, поэтому любое состояние может быть представлено строкой из восьми символов, каждый из которых является числом в диапазоне от 1 до 8. На рис. 3.8а, показана популяция из четырёх индивидуумов для проблемы с восемью ферзями.

Процесс выработки следующего поколения состояний показан на рис. 3.8б – 3.8д. На рис. 3.8б каждое состояние оценивается с помощью *функции пригодности*. Функция пригодности является эвристической и должна возвращать более высокие значения для состояний, более «близких» к целевому состоянию. При решении проблемы с восемью ферзями, в качестве функции пригодности, можно использовать функцию, возвращающую *количество не атакующих друг друга пар ферзей*. Для целевого состояния эта функция принимает наибольшее значение равное 28.

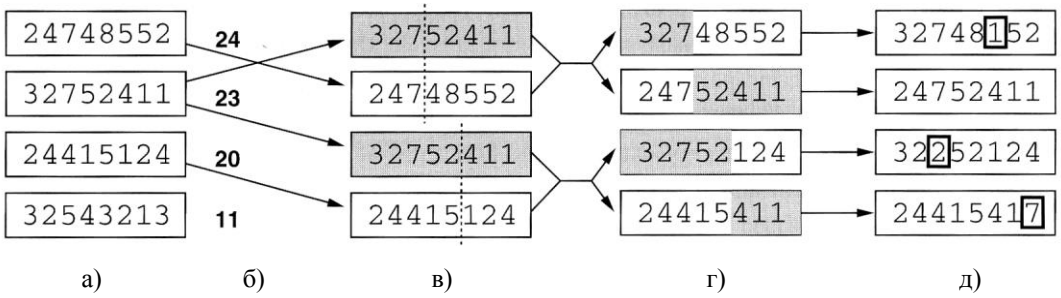

Рис. 3.8. Пример работы генетического алгоритма.
(а) – начальная популяция; (б) – значения функции пригодности;
(в) – отбор; (г) – скрещивание; (д) – мутация.

Для исходных четырёх состояний соответствующие значения функции пригодности равны 24, 23, 20 и 11. Выбор пары индивидуумов для воспроизводства осуществляется случайным образом, а вероятность выбора прямо пропорциональна значению функции пригодности.

Как показано на рис. 3.8в, для воспроизводства выбираются две пары в соответствии со значениями функции пригодности. Как это видно на рис. 3.8в один индивидуум выбирается дважды, а один вообще остаётся не выбранным. Для каждой пары, предназначенной для воспроизводства, среди позиций в строке случайным образом выбирается *точка скрещивания*. На рис. 3.8в точки скрещивания находятся после третьей цифры в первой паре и после пятой цифры во второй паре (точка скрещивания отмечена вертикальной пунктирной линией).

Как показано на рис. 3.8г, потомки создаются путём перекрёстного обмена подстроками родительских строк, разорванных в точке скрещивания. Например, первый потомок первой пары получает первые три цифры от первого родителя, а остальные цифры – от второго родителя, тогда как второй потомок получает первые три цифры от второго родителя, а остальные – от первого родителя.

Состояния проблемы с восемью ферзями, участвующие в этом этапе воспроизводства, показаны на рис. 3.9.

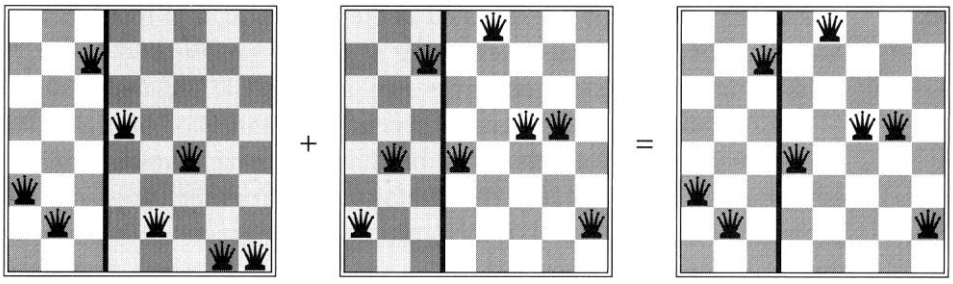

Рис. 3.9. Состояния проблемы с восемью ферзями, соответствующие первым двум родительским состояниям, показанным на рис. 3.8в, и первому потомку, показанному на рис. 3.9г. Потомок образован из незатемнённых столбцов.

Наконец, на рис.3.8д показано, что каждое состояние подвергается случайной *мутации*. В первом, третьем и четвёртом потомках мутация свелась к изменению одной цифры. При решении проблемы с восемью ферзями эта операция соответствует выбору случайным образом одного ферзя и перемещению этого ферзя на случайно выбранную клетку в пределах его столбца.

На рис. 3.10 приведен генетический алгоритм поиска, который реализует описанные этапы.

Как и в алгоритмах стохастического лучевого поиска, в генетических алгоритмах стремление к максимуму сочетается с проводимым случайным образом обменом информацией между параллельными поисковыми потоками.

Основное преимущество генетических алгоритмов, по сравнению с алгоритмом стохастического лучевого поиска, связано с применением операции скрещивания.

```
function GeneticAlgorithm(population, FitnessFn) returns индивидуум
   inputs: population, начальная популяция
           FitnessFn, функция пригодности индивидуума

   repeat // итерация порождает новую популяцию
      newPopulation ← пустое множество
      loop for i from 1 to Size(population) do // новый индивидуум
         x ← RandomSelection(population, FitnessFn)
         y ← RandomSelection(population, FitnessFn)
         child ← Reproduce(x, y) // скрещивание
         if (небольшое случайно выбранное значение вероятности)
         then child ← Mutate(child)
         добавить child к newPopulation
      population ← newPopulation
   until некоторый индивидуум не станет достаточно пригодным или
         не истечёт заданное время
   return наилучший индивидуум в population в соответствии с
          FitnessFn

function Reproduce(x, y) returns индивидуум
   inputs: x, y, индивидуумы-родители

   n ← Length(x)
   c ← случайное число от 1 до n // точка скрещивания
   return Couple(Substring(x, 1, c), Substring(y, c+1, n)
```

Рис. 3.10. Генетический алгоритм. Каждое скрещивание двух родителей порождает одного потомка.

Упражнения

3.1. Постройте дерево поиска, используя алгоритм поиска A* применительно к решению задачи поиска пути из города Лугож в город Бухарест с использованием эвристической функции определяющей кратчайшее расстояние по прямой. Иными словами, покажите последовательность узлов, которые рассматриваются этим алгоритмом, и приведите оценки `f`, `g` и `h` для каждого узла.

3.2. Докажите каждое из приведенных ниже утверждений.

a. Поиск в ширину представляет собой частный случай поиска по критерию стоимости.

b. Поиск в ширину и поиск по критерию стоимости – специальные случаи поиска A*.

3.3. Предложите еще одну эвристическую функцию для головоломки с восемью фишками.

4. ЛОГИЧЕСКИЕ АГЕНТЫ

Раздел посвящён введению в теорию агентов, строящих свое поведение на основе логических умозаключений дедуктивного типа. Рассматриваемые здесь понятия, такие как представление знаний, база знаний и машина вывода являются базовыми в дисциплине искусственный интеллект.

4.1. Агент базы знаний

Центральным компонентом агента базы знаний является его *база знаний*, или сокращённо KB (Knowledge Base). В искусственном интеллекте под базой знаний, как правило, понимают совокупность отдельных порций знаний. В зависимости от принятого способа представления знаний эти порции имеют ту или иную структуру. В общем случае, представление отдельной порции знаний будем называть *предложением*. Предложения базы знаний записываются на языке, называемом *языком представления знаний*.

Для того, чтобы база знаний была актуальна и содержала знания, необходимые для решения некоторой проблемы, должна существовать операция добавления новых предложений в базу знаний, а также операция извлечения знаний из базы знаний. Будем называть эти операции соответственно `Tell` и `Ask`. Обе отмеченные операции связаны с проведением *логического вывода*, т.е. процессом получения новых предложений из уже имеющихся предложений.

Для агентов базы знаний, изучаемых в настоящем разделе, логический вывод подчиняется требованию, заключающемся в том, *что ответ на запрос* к базе знаний, переданный при помощи операции `Ask`, *должен логически следовать из того, что было ранее введено в базу знаний* при помощи операции `Tell`.

Идея агента базы знаний отображена в программе, приведенной на рис. 4.1.

```
function KBAgent(perception) returns action
   static: KB, база знаний
           t, счётчик моментов времени

   KB ← Tell(KB, PerceptionToSentence(perception, t))
   action ← Ask(KB, MakeActionQuery(t))
   KB ← Tell(KB, ActionToSentence(action, t))
   return action
```

Рис. 4.1. Программа агента базы знаний.

Как и все интеллектуальные агенты, агент базы знаний принимает результат акта восприятия `perception` и возвращает действие `action`. Агент поддерживает базу знаний, которая может первоначально содержать некоторые начальные встроенные знания.

Функционирование агента базы знаний можно разделить на три этапа. На первом этапе агент, при помощи функции `Tell` в базу знаний добавляются результаты текущего восприятия в момент времени `t`. На втором этапе, при помощи функции `Ask`, в базу знаний передается запрос о том, какое действие следует предпринять. Процесс поиска ответа на этот запрос представляет собой процесс логического вывода из знаний, имеющимися в базе. На третьем этапе агент добавляет в базу знаний сведения о найденном действии. Вторая функции `Tell` необходима для передачи в базу знаний сведений о том, какое действие является реакцией на восприятие в момент времени `t`.

Функция `PerceptionToSentence` формирует предложение, соответствующее восприятию в момент времени `t`, а функция `ActionToSentence` – предложение, соответствующее выполненному действию в этот же момент времени. Функция `MakeActionQuery` формирует запрос, адресованный базе знаний, о том, какое действие должна быть выполнено в текущий момент времени.

Механизм логического вывода скрыт внутри функций `Tell` и `Ask`. Он реализуется вторым важным компонентом агента базы знаний, который называется *машиной вывода*.

Представление знаний в базе знаний и построение машины вывода, осуществляю-

щей логический вывод, как правило, опираются на теорию и практику одной из математических логик.

Прежде чем перейти к вопросам применения математической логики для проектирования агента базы знаний, рассмотрим мир простой компьютерной игры, называемый «мир Вампуса». В дальнейшем мы используем эту игру для иллюстрации применимости пропозициональной логики для синтеза агента базы знаний.

4.2. Мир Вампуса

Мир Вампуса представляет собой множество пещер, соединённых проходами. В одной из этих пещер находится Вампус – страшный зверь, который поедает всех, кто входит к нему в пещеру. Агент может убить Вампуса из лука, но у агента есть только одна стрела. В некоторых пещерах есть бездонные ямы, в которые агент может провалиться и погибнуть. Однако Вампусу эти ямы не страшны. Он не может в них провалится, потому что слишком велик. Единственное, что утешает агента, путешествующего в этой негостеприимной среде, – это возможность найти кучу золота.

Хотя эта компьютерная игра является довольно скромной, с точки зрения современных компьютерных игр, она представляет собой хорошую испытательную среду для агента базы знаний. Для того, чтобы агент смог пройти через пещеры, избегая Вампуса и пещеры с бездонными ямами, найти золото и вернутся назад, он, получив новую порцию знаний в результате восприятия, должен проводить логические рассуждения. Целью этих логических рассуждений является получение дополнительных знаний о том, какие из, окружающих его пещер, являются опасными, а какие – нет.

На рис. 4.2 приведено упрощенное изображение мира Вампуса, на котором пещеры изображены в виде 16 квадратов, расположенных внутри сетки, размером 4x4 квадрата. Показано местонахождение Вампуса и ям.

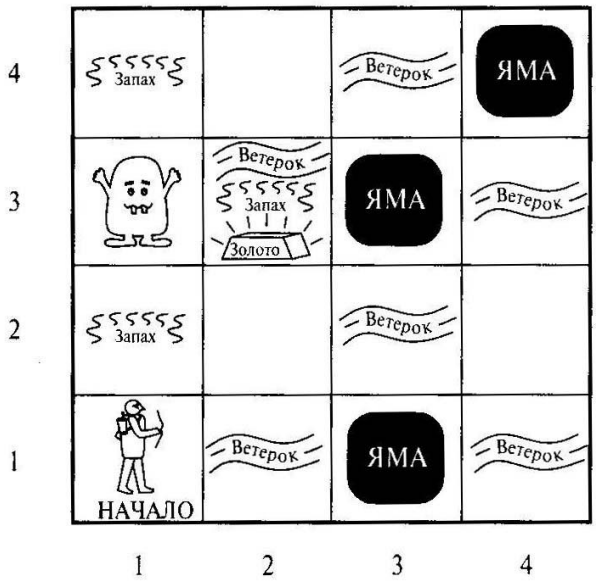

Рис. 4.2. Пример мира Вампуса. Агент находится в левом нижнем углу.

Специфицирование агента в контексте проблемной среды мира Вампуса (см. рис. 1.4) опишем следующим образом.

- *Критерии поведения*. Будем считать, что если агент находит золото, он получает +1000 очков, а если он погибает (попадает в яму или его съедает Вампус), то он получает –1000 очков, Агент отдаёт –1 очко за каждое выполненное действие и –10 очков за использование стрелы.
- *Характеристики среды*. Пещеры представляют собой квадраты, расположенные в виде решётки 4x4 и окружёны стеной. Агент всегда начинает движение из квадрата, обозначенного как [1, 1], и смотрит вправо. Каждый раз, когда агент начинает игру,

местонахождение золота и Вампуса выбираются случайным образом из числа квадратов, отличных от начального квадрата. Кроме того, в каждом квадрате, отличном от начального, с вероятностью 0,2 может находиться яма.

- *Эффекторы*. Агент располагает эффекторами, обеспечивающими ему движение вперёд, а также повороты влево или вправо на 90^0. Попытка продвижения вперёд остаётся безрезультатной, если перед агентом находится стена. Чтобы схватить предмет, находящийся в том же квадрате, где и агент, необходимо выполнить действие `Grab`. С помощью действия `Shoot` можно выстрелить из лука по прямой линии в том направлении, куда смотрит агент. Стрела продолжает движение до тех пор, пока она либо не попадёт в Вампуса (и убьёт его), либо не ударится в стену. У агента имеется только одна стрела, поэтому какой-либо эффект может оказать лишь первое действие `Shoot`. Агент погибает, если попадает в квадрат, где имеется яма или живой Вампус. Входить в квадрат с мёртвым Вампусом безопасно.
- *Сенсоры*. У агента имеется пять сенсоров, каждый из которых формирует один компонент сенсорного события.
  1 Сенсор запаха. В квадрате, где находится Вампус, а также в смежных с ним квадратах (но не в диагональных) сенсор запаха обнаруживает неприятный запах и формирует компонент сенсорного события `Stench`.
  2 Сенсор движения воздуха. В квадратах, смежных с квадратом, где находится яма, сенсор движения воздуха обнаруживает ветерок и формирует компонент сенсорного события `Breeze`.
  3 Сенсор блеска. В квадрате, где находится золото, сенсор блеска обнаруживает блеск и формирует компонент сенсорного события `Glitter`.
  4 Сенсор препятствия. Когда агент наталкивается на стену, то сенсор препятствия её обнаруживает и формирует компонент сенсорного события `Bump`.
  5 Сенсор громкого звука. Перед тем, как поражённый стрелой Вампус умирает, он испускает громкий крик, который воспринимается сенсором громкого звука. Этот сенсор предоставляет агенту информацию о том, что его стрела попала в цель и Вампус убит. В результате восприятия крика Вампуса формируется компонент сенсорного события `Scream`.

  Таким образом, сенсорное событие состоит из пяти компонентов, Будем записывать сенсорное событие в виде списка из пяти компонентов в том порядке, в котором мы перечислили сенсоры. Например, если сенсоры агента воспринимают: неприятный запах и ветерок, но не воспринимают блеск, препятствие и громкий звук, то сенсорное событие будем записывать в виде: `[Stench, Breeze, None, None, None]`.

Основная сложность для агента, находящегося в мире Вампуса, заключается в том, что он воспринимает только ту пещеру, в которой он находится и, следовательно, изначально не знает в каких пещерах находятся Вампус, ямы и золото. Однако, эти знания он может вывести из воспринятого сенсорного события путём логических рассуждений.

В большинстве экземпляров мира Вампуса агент может сформировать безопасный путь к золоту. Но иногда агенту приходится выбирать вернуться ли домой с пустыми руками или рисковать жизнью, чтобы найти золото. Около 21% вариантов среды являются совершенно неблагоприятными, поскольку золото находится в яме или окружено ямами.

Проследим за возможным поведением агента базы знаний в мире Вампуса, изображённом на рис. 4.2. Первоначально база знаний агента содержит минимальные встроенные знания о среде. Агенту известно, только, что он находится в квадрате [1, 1] и что этот квадрат является безопасным. Однако эти первоначальные знания будут пополняться по мере поступления новых сенсорных событий и выполнения действий.

Первое сенсорное событие, воспринимаемое агентом в квадрате [1, 1], запишем в виде `[None, None, None, None, None]`. На основании встроенных знаний и первого сенсорного события агент может сделать вывод, что соседние по отношению к нему квадраты являются безопасными. На рис. 4.3, а) показано состояние знаний агента в этот момент.

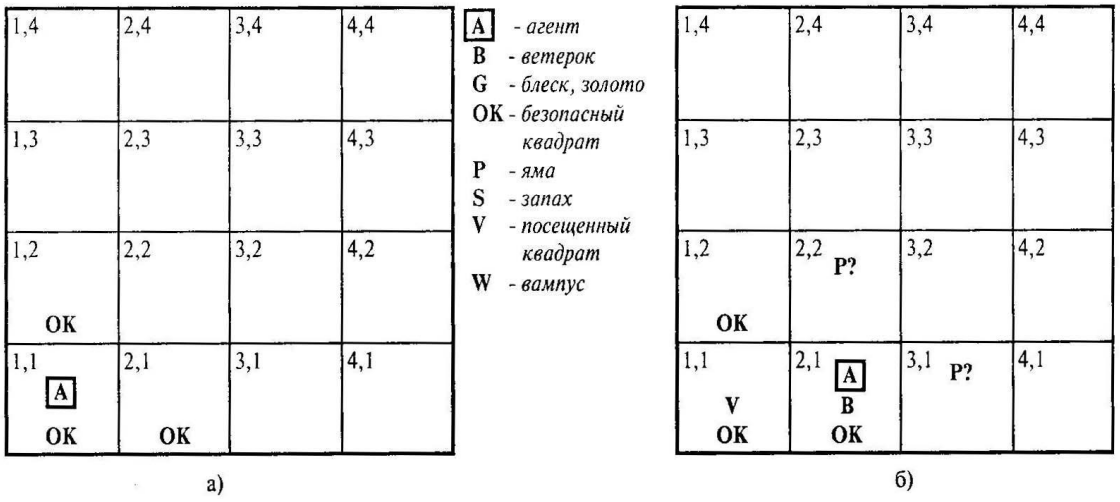

Рис. 4.3. Первый шаг, выполненный агентом в мира Вампуса. а) первоначальная ситуация, возникшая после восприятия сенсорного события `[None, None, None, None,None]`. б) ситуация после первого шага и восприятия сенсорного события `[None, Breeze, None, None, None]`.

На рис. 4.3 и далее на рис. 4.4 мы будем использовать простые буквенные обозначения, соответствующие предложениям в базе знаний. Например, символ «B» в некотором квадрате обозначает предложение, утверждающее, что в данном квадрате имеется ветерок.

На основании того факта, что в квадрате [1, 1] не были восприняты ни неприятный запаха, ни ветерок, агент может сделать вывод, что квадраты [1, 2] и [2, 1] безопасны. Поэтому, на рис. 4.3а) эти квадраты помечены буквами `OK`.

Осторожный агент переходит только в тот квадрат, который помечен буквами `OK`. Поскольку оба квадрата, смежные с квадратом [1, 1], являются безопасными, то агент может выбрать любой из квадратов [1, 2] или [2, 1]. Предположим, что агент решил переместиться на один квадрат вправо и, выполнив необходимые действия, оказался в квадрате [2, 1]. Эта ситуация изображена на рис. 4.3, б).

В квадрате [2, 1] агент пополняет базу знаний предложениями, полученными в результате восприятия нового сенсорного события `[None, Breeze, None, None, None]`. Пополнив базу знаний новыми предложениями, агент переходит к логическим рассуждениям, относительно статуса окружающих его квадратов. Предыдущие и новые знания позволяют агенту сделать вывод о том, что в одном из квадратов, смежных с квадратом [2, 1] должна быть яма.

Однако, встроенные знания указывают на то, что яма не может находиться в квадрате [1, 1], поэтому она должна быть или в квадрате [2, 2] или в квадрате [3, 1], или в обоих квадратах одновременно. Обозначение `P?` на рис. 4.3, б) указывает на возможность наличия ямы в этих квадратах. Таким образом, после первого перемещения из квадрата [1, 1] в квадрат [2, 1] агенту известен только один квадрат с отметкой `OK`, который ещё не был посещён. Это квадрат [1, 2]. Поэтому благоразумный агент поворачивается кругом, возвращается в квадрат [1, 1], а затем переходит в квадрат [1, 2].

В квадрате [1, 2] агент воспринимает сенсорное событие `[Stench, None, None, None, None]`. Состояние игры, после перемещения агента в квадрат [1, 2] изображено на рис. 4.4, а). Наличие неприятного запаха в квадрате [1, 2] означает, что в одном из смежных квадратов находится Вампус. Но агент знает, что Вампуса нет в квадрате [1, 1]. На основании имеющихся знаний агент может вывести, что Вампуса нет в квадрате [2, 2], поскольку агент обнаружил бы неприятный запах, находясь в квадрате [2, 1]. Поэтому, на основании новых знаний, полученных из последнего сенсорного события, агент делает вывод, что Вампус находится в квадрате [1, 3]. На это указывает обозначение `W!` На рис. 4.4а).

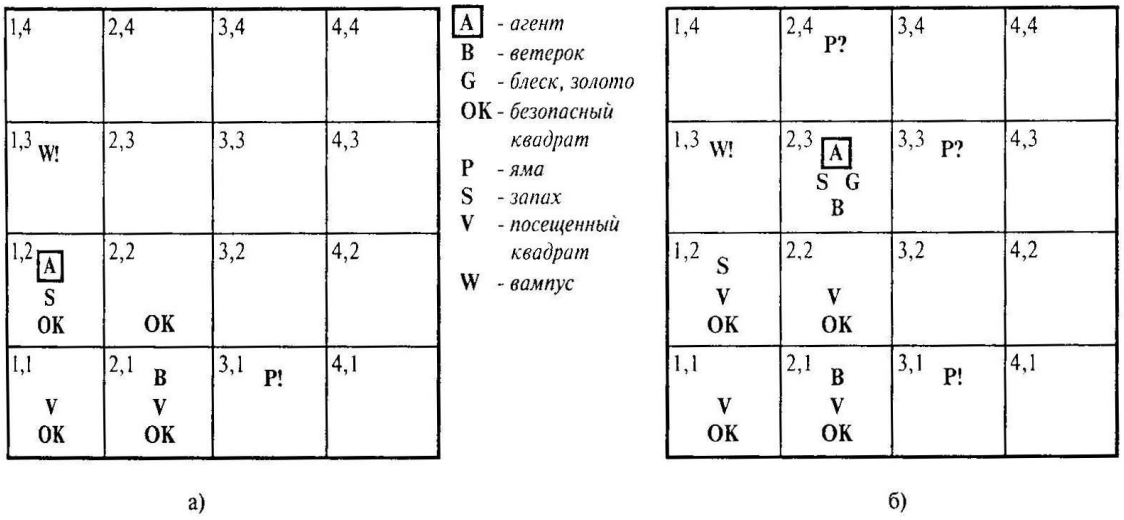

Рис. 4.4. Два последних шага в поведении агента. а) после третьего шага, когда было воспринято сенсорное событие `[Stench, None, None, None, None]`. б) после пятого шага, когда было воспринято сенсорное событие `[Stench, Breeze, Glitter, None, None]`.

В последнем сенсорном событии, которое агент воспринял в квадрате [1, 2] отсутствует компонент `Breeze`. Из этого агент может вывести, что в квадрате [2, 2] нет ямы. Однако, ранее, он пришел к выводу, что яма должна быть или в квадрате [2, 2] или в квадрате [3, 1], но поскольку ее точно нет в [2, 2], то это означает, что яма может находиться только в квадрате [3, 1]. Это пример сложного (для интеллектуального агента) логического вывода, поскольку в нём объединяются знания, полученные в разные моменты времени и в различных местах. Такой логический вывод превосходит интеллектуальные способности большинства животных, но является типичным для человека.

Теперь агент доказал сам себе, что в квадрате [2, 2] нет ни ямы ни Вампуса, поэтому он может обозначить этот квадрат меткой `OK`, а затем перейти в него. Мы не показываем знания агента в квадрате [2, 2], а просто предполагаем, что агент повернулся и перешёл в квадрат [2, 3], что отображено на рис. 4.4, б). В квадрате [2, 3] агент воспринимает блеск, поэтому он может схватить золото, вернутся в квадрат [1, 1] по пути, отмеченном буквами `OK` и, тем самым, закончить игру.

В описанном примере рассуждений агент верил знаниям, полученным на основании логического вывода, поскольку он считал, что *в том случае, когда логический вывод базируется на правильных (истинных) знаниях, то правильными (истинными) должны быть выводимые знания.* В этом состоит фундаментальное свойство логического вывода.

4.3. Предложения базы знаний и математическая логика

В 4.1 было отмечено, что база знаний состоит из предложений, которые записываются в соответствии с правилами языка представления знаний, определяющими форму правильно построенных предложений. В качестве языка представления знаний удобно использовать формальную систему какой-либо из математических логик. Как правило, для этой цели используют формальную систему логики предикатов первого порядка. Однако, с целью упрощения материала, далее, при записи предложений базы знаний, будет использована более простая формальная система *пропозициональной логики*, называемой, также, *логикой высказываний*.

Формальная система математической логики задаёт не только синтаксис правильно построенных предложений, но определяет, также, *семантику* предложений. Семантика определяет истинность предложений. Истинность предложений понимается не в абсолютном смысле, а по отношению к одному из *возможных миров*. Здесь выражение «возможный мир» означает совокупное значение переменных, определяющих истинность предложения. Например, обычная семантика, принятая в арифметике, определяет, что

предложение «x + y = 4» истинно в мире, где x = 2 и y = 2, но ложно в мире, где x = 1 и y = 1. В литературе по математической логике, часто, вместо выражения «возможный мир» используется термин *модель*. Таким образом, истинность предложения должна быть определена по отношению к некоторой модели. Для указания того, что предложение α является истинным по отношению к модели m, будем использовать фразу «m является моделью α».

В случае пропозициональной логики модель задаёт конкретную комбинацию значений пропозициональных переменных. Общее количество наборов значений пропозициональных переменных для некоторого предложения определяется степенью числа 2, где показатель степени – количество пропозициональных переменных данного предложения. Таким образом, если предложение составлено из трёх переменных, то для него возможно $2^3 = 8$ различных наборов значений переменных.

Теперь, после краткой интерпретации понятия формальной системы математической логики, которая будет использоваться в качестве языка представления знаний, перейдём к теме логического вывода. Как было отмечено ранее, логический вывод является необходимым механизмом функционирования агента базы знаний, воплощенный в машине вывода.

Начнем с определения понятие *логического следствия*. Логическое следствие означает, что одно предложение следует из другого предложения. Для обозначения логического следствия применяется запись

$$\alpha \models \beta,$$

которая читается: «предложение α *влечёт* за собой предложение β».

Точное определение логического следствия следующее: $\alpha \models \beta$ тогда и только тогда, когда в любой модели, в которой предложение α является истинным, предложение β также является истинным. Запись KB $\models \beta$ читается: «база знаний KB влечет за собой предложение β».

Существует существенное различие в интерпретации понятий *«логическое следствие»* и *«логический вывод»*. Чтобы понять это различие воспользуемся аналогией. Представим себе, что множество всех предложений базы знаний, – это стог сена, а предложение α – это иголка. Тогда, *логическое следствие аналогично утверждению о том, что в стоге сена есть иголка, а логический вывод аналогичен поиску иголки в стоге сена*. Таким образом, для осуществления логического вывода необходим некоторый алгоритм. Если некоторый алгоритм логического вывода i позволяет вывести предложение α из базы знаний KB, то мы будем писать

$$KB \models_i \alpha$$

Эта запись читается следующим образом: «предложение α получено путём логического вывода из базы знаний KB с помощью алгоритма i» или «алгоритм i позволяет вывести предложение α из базы знаний KB».

Алгоритм логического вывода, позволяющий получить только такие предложения, которые действительно следуют из базы знаний, и которым можно верить называется *непротиворечивым*. Непротиворечивость – это в высшей степени желательное свойство. Противоречивый алгоритм логического вывода в ходе своей работы, по сути, создаёт то, чего нет на самом деле, и объявляет об обнаружении несуществующих предложений.

Желательным свойством алгоритма логического вывода является также свойство *полноты*. Алгоритм логического вывода называется полным, если он позволяет вывести любое предложение, которое следует из базы знаний.

По сути, нас интересуют такие логические рассуждения, заключения которых гарантированно истинны в мире, в котором истинны предпосылки. В частности, *если база знаний состоит из предложений, моделирующих истинные знания о реальном мире, то любое предложение α, полученное путём логического вывода из этой базы знаний при помощи непротиворечивой процедуры логического вывода, должно также отражать истинные знания о реальном мире*.

Таким образом, несмотря на то, что процесс логического вывода, который проводит интеллектуальный агент, является формальным и оперирует с синтаксическими элемен-

тами предложений, а не с их смысловой интерпретацией, он позволяет получить предложения, семантическая интерпретация которых адекватна реальному миру. Это соответствие между реальным миром и его логическим представлением продемонстрировано на рис. 4.5.

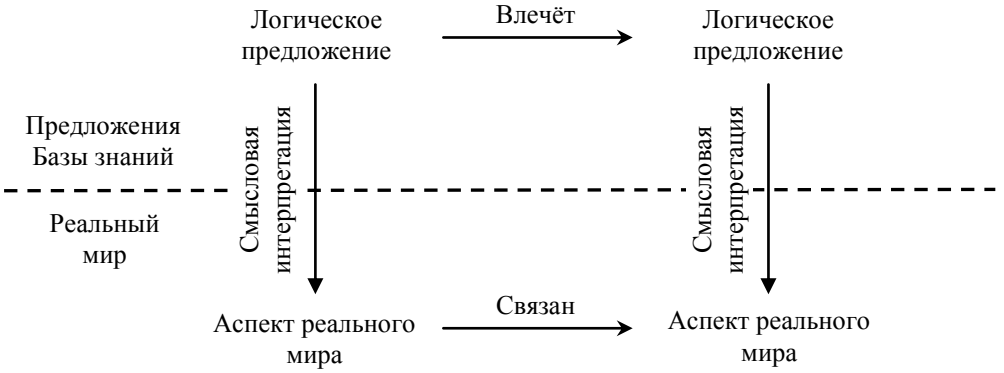

Рис. 4.5. Предложения представляют собой описания аспектов реального мира, а вывод – это процесс получения новых предложений из уже существующих. Вывод должен гарантировать, что новые предложения будут адекватны тем аспектам мира, которые действительно существуют.

Последняя проблема, которая должна быть решена применительно к агентам базы знаний, – это *проблема* адекватности базы знаний. Проблема заключается в обосновании того, что база знаний состоит из предложений, отражающих истинные знания о реальном мире. Ведь база знаний представляет собой просто набор синтаксических конструкций в памяти агента. Для случая агента базы знаний, изучаемого в настоящем разделе, это обоснование заключается в том, что предложения базы знаний формируются автоматически из сенсорных событий. Если сенсорная система агента работает правильно, то процесс восприятия агентом проблемной среды порождает в базе знаний только предложения, адекватные этой среде. Например, агент, оперирующий в мире Вампуса обнаружив неприятный запах при помощи одного из своих сенсоров, автоматически создаёт соответствующее предложение в базе знаний. Таким образом, если предложение находится в базе знаний, то ему обязательно соответствует некоторый аспект реального мира.

Кроме знаний, формируемых восприятиями, в базе знаний агента должны храниться правила, формируемые в результате его жизненного опыта. Например, правило о том, что если в некотором квадрате воспринимается ветерок, то в одном из смежных квадратов находится яма. Выработка подобных общих правил и есть процесс *обучения*. Поэтому, если агент базы знаний способен пополнять свою базу знаний новыми правилами, полученными на основе опыта, то он является автономным.

4.4. Пропозициональная логика

Как было отмечено выше, пропозициональная логика будет использована для синтеза агента базы знаний, способного успешно играть в игру, которую мы назвали «мир Вампуса». Рассмотрим, кратко, основы пропозициональной логики в том объеме, который необходим для достижения этой цели.

Любая математическая логика включает.

- *Формальную систему* для формального описания реального мира, включающую: (a) *синтаксис*, задающий правила записи предложений, и (b) *семантику*, которая, в случае пропозициональной логики определяет условия истинности предложений, а в общем случае задаёт смысловую интерпретацию аспектов реального мира.
- *Теорию доказательств* или набор правил для логического вывода на множестве предложений базы знаний.

Формальные системы пропозициональной логики и других математических логик часто используются в качестве языка представления знаний, позволяющего описывать предложения базы знаний, а теории доказательств – для построения машины вывода.

4.4.1. Синтаксис

Атомарные предложения представляют собой неделимые синтаксические элементы, состоящие из одного *пропозиционального символа*. Каждый такой символ имеет смысл переменной и обозначает какое-либо утверждение, которое может быть либо истинным, либо ложным. Для записи пропозициональных символов часто используются прописные буквы второй половины латинского алфавита: P, Q, R и т.д. Иногда обозначение пропозициональных символов выбираются не абстрактными и отвлечёнными, а ассоциируемые с элементами проблемной среды. Например, символ $W_{1,3}$, может использоваться для обозначения предложения, которое на русском языке можно записать следующим образом: «Вампус находится в квадрате [1, 3]».

Существуют два пропозициональных символа, имеющих постоянную семантическую интерпретацию. True – тождественно истинное предложение, и False – тождественно ложное предложение.

Сложные предложения формируются из атомарных предложений с использованием *логических связок*. Используются следующие логические связки.

- ¬ (читается «не»). Такое предложение, как $\neg W_{1,3}$, называется *отрицанием* предложения $W_{1,3}$. *Литералом* называется либо атомарное предложение (*положительный литерал*), либо отрицание атомарного предложения (*отрицательный литерал*).
- ∧ (читается «и»). Предложение, основной связкой которого является ∧, такое как $W_{1,3} \wedge P_{3,1}$ называется *конъюнкцией*. Его части называются *конъюнктами*.
- ∨ (читается «или»). Предложение, в котором основной связкой является ∨, такое как $(W_{1,3} \wedge P_{3,1}) \vee W_{2,2}$ называется *дизъюнкцией*, а его части называются *дизъюнктами*.
- ⇒ (читается «влечёт за собой»). Такое предложение, как $(W_{1,3} \wedge P_{3,1}) \Rightarrow \neg W_{2,2}$ называется *импликацией* или условным предложением. Его *антецедентом*, является левая часть $(W_{1,3} \wedge P_{3,1})$, а его *консеквентом* – правая часть $\neg W_{2,2}$.
- ⇔ (читается «если и только если»). Предложение вида $W_{1,3} \Leftrightarrow \neg W_{2,2}$ называется *двусторонняя импликация* или *эквиваленция*.

При использовании формальной системы пропозициональной логики в качестве языка представления знаний, предложения, построенные на основе импликации или эквиваленции, используются для моделирования правил.

На рис. 4.6 представлена формальная грамматика (синтаксис) пропозициональной логики в нотации Наура- Бэкуса.

```
        Sentence → AtomicSentence | ComplexSentence
  AtomicSentence → True | False | Symbol
          Symbol → P | Q | R | . . .
ComplexSentence → ¬ Sentence
                | (Sentence ∧ Sentence)
                | (Sentence ∨ Sentence)
                | (Sentence ⇒ Sentence)
                | (Sentence ⇔ Sentence)
```

Рис. 4.6. Грамматика пропозициональной логики в нотации Наура-Бэкуса.

Нотация Наура-Бэкуса представляет собой простой метаязык, который можно использовать для формального определения грамматики искусственных языков: языков программирования, языков запросов или входных языков пакетов прикладных программ. Грамматика описывается в виде набора предложений, состоящих из левой и правой частей. В левой части записывается синтаксический элемент, который необходимо определить, а в правой части – те синтаксические элементы, посредством которых он определяется. Левая часть предложения отделяется от правой части при помощи символа «→». В правой части предложения могут располагаться *терминальные* и *нетерминальные* символы. Терминальные символы (на рис. 4.6 выделены жирным шрифтом) интерпретируются как аксиомы и не требуют дальнейших определений. Нетерминальные символы

должны быть определены при помощи последующих предложений. Терминальные и нетерминальные символы разделяются при помощи символа «|», который читается как «или».

Первое предложение, на рис. 4.6, имеет вид

Sentence → AtomicSentence | ComplexSentence

и определяет понятие Sentence (предложение). На русском языке это определение можно записать следующим образом: «Предложением пропозициональной логики является или атомарное предложение (AtomicSentence) или сложное предложение (ComplexSentence)».

Второе предложение, на рис. 4.6, имеет вид

AtomicSentence → **True** | **False** | Symbol

и определяет понятие AtomicSentence (атомарное предложение). На русском языке это определение можно записать следующим образом: «Атомарным предложением пропозициональной логики является или **True** или **False** или Symbol (символ)». Здесь **True** и **False** терминальные символы, не требующие дальнейшего определения.

При чтении рис. 4.6 необходимо обратить внимание на то, что грамматика предъявляет строгие требования к использованию круглых скобок. Каждое предложение, сформированное при помощи бинарных логических связок, должно быть заключено в круглые скобки. Это гарантирует непротиворечивость синтаксиса. Такое требование также означает, что следует писать, например, ((A ∧ B) ⇒ C) вместо A ∧ B ⇒ C.

Для удобства записи, в тех случаях, когда приоритет следования связок очевиден, мы будем опускать круглые скобки при записи предложений. Приоритет логических связок в пропозициональной логике (от высшего к низшему) следующий: ¬, ∧, ∨, ⇒ и ⇔. Поэтому предложение:

¬P ∨ Q ∧ R ⇒ S

Эквивалентно предложению:

((¬P) ∨ (Q ∧ R)) ⇒ S

Наличие приоритета не позволяет устранить неоднозначность при чтении таких предложений, как A ∧ B ∧ C, которое может быть прочитано как ((A ∧ B) ∧ C) или (A ∧ (B ∧ C)). Поэтому использование круглых скобок более предпочтительно, чем запись, основанная на приоритетах.

4.4.2. Семантика

Семантика предложения определяет условия его истинности. При использовании понятия «модель», семантика предложения это истинность предложения по отношению к конкретной модели. В пропозициональной логике модель фиксирует значения (true или false) каждого пропозиционального символа предложений базы знаний. Например, если в предложениях некоторой базы знаний используются пропозициональные символы $P_{1,2}$, $P_{2,2}$ и $P_{3,1}$ то одна из возможных моделей состоит в следующем:

m_1 = {$P_{1,2}$ = false, $P_{2,2}$ = false, $P_{3,1}$ = true}

После того, как модель определена, предложения базы знаний, с точки зрения семантики, становятся абстрактными математическими объектами. Например, $P_{1,2}$ – превращается в математический символ. Он может означать что угодно, например, что «в квадрате [1, 2] есть яма» или «Париж – столица Франции».

Семантика пропозициональной логики должна определять, как следует вычислять истинность предложения при наличии модели. Все предложения формируются из атомарных предложений и пяти логических связок, поэтому необходимо указать, как следует вычислять истинность атомарных предложений, а затем – как вычислять истинность сложных предложений, сформированных при помощи логических связок.

Задача вычисления истинности атомарных предложений является весьма простой.

- Предложение True истинно в любой модели, а предложение False ложно в любой модели.
- Истинность любого другого пропозиционального символа должно быть указано непосредственно в модели. Например, в модели m_1 предложение $P_{1,2}$ является ложным.

Для определения истинности сложных предложений применяются правила, позволяющие свести задачу определения истинности сложного предложения к задаче определения истинности более простых предложений. Правила определения истинности, при использовании логических связок, задаются *таблицей истинности*, которая определяет истинность сложного предложения для всех возможных комбинаций значений его пропозициональных символов. Таблица истинности для пяти логических связок приведена на рис. 4.7.

P	Q	¬P	P ∧ Q	P ∨ Q	P ⇒ Q	P ⇔ Q
false	false	true	false	false	true	true
false	true	true	false	true	true	false
true	false	false	false	true	false	false
true	true	false	true	true	true	true

Рис. 4.7. Таблица истинности для пяти логических связок.

Истинность сложного предложения s для некоторой модели m может быть вычислена с использованием таблицы истинности, приведенной на рис. 4.7. Например, истинность предложения

$$\neg P_{1,2} \wedge (P_{2,2} \vee P_{3,1})$$

для, приведенной выше, модели m_1, вычисляется следующим образом

$$\text{true} \wedge (\text{false} \vee \text{true}) = \text{true} \wedge \text{true} = \text{true}$$

Ранее было сказано, что любая база знаний состоит из множества предложений. Но *базу знаний можно также рассматривать как конъюнкцию этих предложений*. Это означает, что начиная с пустой базы знаний KB и последовательно применяя операции

Tell(KB, S_1), … , Tell(KB, S_n), мы получим: $KB = S_1 \wedge \ldots \wedge S_n$.

Для проведения логического вывода в мире Вампуса, с целью выявления безопасных квадратов, необходимы правила, позволяющие осуществлять вывод. Эти правила будем записывать в виде предложений, в которых основной логической связкой является двусторонняя импликация. Например, правило «если в квадрате [1,1] воспринимается ветерок, то в одном из квадратов [1,2] или [2,1] находится яма, а если в одном из квадратов [1,2] или [2,1] имеется яма, то в квадрате [1,1] воспринимается ветерок» можно записать при помощи предложения:

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$$

где $B_{1,1}$ означает, что в квадрате [1, 1] воспринимается ветерок. Если при записи этого же правила использовать обыкновенную импликацию, например:

$$B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})$$

то правило становится не полным, поскольку *утверждает только*, что если обнаружен ветерок, то из этого факта следует наличие ям. И, следовательно, обыкновенная импликация не учитывает, что из факта наличия ямы следует ветерок.

4.4.3. Исходная база знаний для мира Вампуса

Теперь мы можем использовать формальную систему пропозициональной логики для описания базы знаний мира Вампуса. Рассмотрим, пока, как может выглядеть база знаний для ситуации, когда агент сделал первые шаги и находится в квадрате [2, 1]. Эта ситуация изображена на рис. 4.3 б).

База знаний включает перечисленные ниже предложения, каждому из которых присвоено отдельное обозначение.

- Знания о том, что в квадрате [1, 1] отсутствует яма, представим предложением

  $R_1: \neg P_{1,1}$

- Знания о том, что в квадратах [1, 1] и [2, 1] воспринимается ветерок тогда и только тогда, когда в соседнем квадрате имеется яма, представим в виде предложений

  $R_2: B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$
  $R_3: B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$

- Приведенные выше предложения являются истинными для всех экземплярах мира Вампуса и не зависят от восприятий агента. Теперь включим в базу знаний предложения, представляющие знания, которые агент сформировал в результате восприятия сенсорных событий после посещения квадратов [1, 1] и [2, 1]. Как следует из рис. 4.3б) и его описания, при посещении квадратов [1, 1] и [2, 1] агент воспринимал сенсорные события, которые предоставляли ему знания о наличии или отсутствии ветерка в этих квадратах. Представим эти знания в виде предложений

  $R_4: \neg B_{1,1}$
  $R_5: B_{2,1}$

Таким образом, после того, как агент дошёл до квадрата [2, 1], его база знаний может состоять из предложений $R_1 - R_5$. Вместо отдельных предложений эту базу знаний можно рассматривать и как единственное предложение в виде конъюнкции $R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5$, поскольку все отдельно взятые предложения-конъюнкты являются истинными.

4.4.4. Эквивалентность, допустимость и выполнимость

Прежде чем перейти к изучению логического вывода, необходимо рассмотреть некоторые важные понятия логического следствия в контексте пропозициональной логики.

Первым понятие является *логическая эквивалентность*. Предложения α и β, называются эквивалентными, если они *истинны в одном и том же множестве моделей*. Это утверждение можно представить двусторонней импликации.

$$\alpha \Leftrightarrow \beta$$

Например, можно легко показать, что предложения $P \wedge Q$ и $Q \wedge P$ логически эквивалентны. Примеры других, известных, логически эквивалентных предложений, приведены на рис. 4.8.

$(\alpha \wedge \beta)$	$\Leftrightarrow$	$(\beta \wedge \alpha)$	коммутативность связки $\wedge$
$(\alpha \vee \beta)$	$\Leftrightarrow$	$(\beta \vee \alpha)$	коммутативность связки $\vee$
$((\alpha \wedge \beta) \wedge \gamma)$	$\Leftrightarrow$	$(\alpha \wedge (\beta \wedge \gamma))$	ассоциативность связки $\wedge$
$((\alpha \vee \beta) \vee \gamma)$	$\Leftrightarrow$	$(\alpha \vee (\beta \vee \gamma))$	ассоциативность связки $\vee$
$\neg(\neg \alpha)$	$\Leftrightarrow$	α	устранение двойного отрицания
$(\alpha \Rightarrow \beta)$	$\Leftrightarrow$	$(\neg \beta \Rightarrow \neg \alpha)$	контрапозиция
$(\alpha \Rightarrow \beta)$	$\Leftrightarrow$	$(\neg \alpha \vee \beta)$	устранение импликации
$(\alpha \Leftrightarrow \beta)$	$\Leftrightarrow$	$((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$	устранение двусторонней импликации
$\neg(\alpha \wedge \beta)$	$\Leftrightarrow$	$(\neg \alpha \vee \neg \beta)$	правило де Моргана
$\neg(\alpha \vee \beta)$	$\Leftrightarrow$	$(\neg \alpha \wedge \neg \beta)$	правило де Моргана
$(\alpha \wedge (\beta \vee \gamma))$	$\Leftrightarrow$	$((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$	дистрибутивность $\wedge$ по отношению к $\vee$
$(\alpha \vee (\beta \wedge \gamma))$	$\Leftrightarrow$	$((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$	дистрибутивность $\vee$ по отношению к $\wedge$

Рис. 4.8. Стандартные логические эквивалентности.

Альтернативным определением эквивалентности является следующее. Для любых двух предложений α и β

$\alpha \Leftrightarrow \beta$ тогда и только тогда, когда $\alpha \models \beta$ и $\beta \models \alpha$

Эквивалентности играют в логике примерно такую же роль, какую алгебраические равенства играют в обычной математике.

Вторым понятием, которое нам потребуется, является *допустимость*. Предложение допустимо, если оно *истинно во всех моделях*. Например, предложение

$$P \vee \neg P$$

является допустимым. Допустимые предложения также известны под названием *тавтологии*.

Последним понятием является *выполнимость*. Предложение выполнимо тогда и только тогда, когда оно истинно *в некоторой модели*. Например, приведенная выше база знаний, $(R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5)$, выполнима, поскольку существует модель, в которых она истинна.

Если некоторое предложение α является истинным в модели m, то для обозначения этого используется выражение «m выполняет α». Выполнимость можно проверить, перебирая все возможные модели до тех пор, пока не будет найдена хотя бы одна из них, которая выполняет данное предложение.

4.5. Правила логического вывода

Логический вывод представляет собой цепочку логических преобразований предложений базы знаний. Для построения таких цепочек часто используются *правила логического вывода*. Для записи правил вывода будем использовать следующую нотацию.

$$\frac{\alpha}{\beta}$$

В этой нотации, над горизонтальной линией располагаются *предложения-посылки*, а под линией – выводимое *предложение-заключение*. Приведенный пример правила вывода означает, что предложение β выводится из предложения α. Можно доказать, используя таблицы истинности или стандартные логические эквивалентности, приведенные на рис. 4.8, что если в некоторой модели предложения-посылки правила вывода истинны, то в этой же модели заключение также истинно.

Одним из наиболее известных правил вывода является *правило отделения* или *правило Модус Поненс*. Это правило записывается следующим образом

$$\frac{\alpha \Rightarrow \beta,\ \alpha}{\beta}$$

Модус Поненс утверждает, что если имеются предложения $\alpha \Rightarrow \beta$ и α, то из них можно вывести предложение β. Например, если имеется «правило стрельбы по Вампусу», записанное в виде предложения $(\mathrm{WumpusAhead} \wedge \mathrm{WumpusAlive}) \Rightarrow \mathrm{Shoot}$ и имеется факт, записанный в виде предложения $(\mathrm{WumpusAhead} \wedge \mathrm{WumpusAlive})$, то правило Модус Поненс позволяет вывести предложение Shoot.

Ещё одним полезным правилом вывода является *правило удаления связки «И»*. Это правило утверждает, что из конъюнкции можно вывести любой её конъюнкт, и записывается в виде

$$\frac{\alpha \wedge \beta}{\alpha}$$

Например, из предложения $(\mathrm{WumpusAhead} \wedge \mathrm{WumpusAlive})$ можно вывести предложение $\mathrm{WumpusAlive}$.

В качестве правил логического вывода можно также использовать все стандартные логические эквивалентости, приведенные на рис. 4.8. Например, из эквивалентности «устранение двусторонней импликации» можно получить следующие два правила вывода.

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)} \qquad \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}$$

Первое, из этих двух правил, имеет имя и называется *правило удаления двухсторонней импликации*.

Рассмотрим, как можно осуществлять логический вывод в мире Вампуса, используя, приведенные выше, правила вывода.

Пусть, как и прежде, база знаний состоит из предложений $R_1 - R_5$. Логический вывод будет состоять из последовательности шагов. На каждом шаге мы будем применять одно из правил вывода к одному или нескольким предложениям базы знаний, рассматривая их как предложения-посылки. Получаемое предложение-заключение будем записывать в базу знаний.

Докажем, что из базы знаний, состоящей из предложений $R_1 - R_5$ следует отсутствие ямы в квадратах [1, 2] и [2, 1] и, следовательно, эти квадраты является безопасными. Знания о том, что в квадратах [1, 2] и [2, 1] отсутствуют ямы будем моделировать предложениями $\neg P_{1,2}$ и $\neg P_{2,1}$. Таким образом, в результате логического вывода мы должны вывести предложения $\neg P_{1,2}$ и $\neg P_{2,1}$.

Начнем вывод с применения правило удаления двухсторонней импликации к предложению R_2: $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$. В результате получим следующее предложение

$$R_6: (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$$

Теперь применим правило удаления связки «И» к предложению R_6. В результате получим предложение

$$R_7: (P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}$$

Из стандартной эквивалентности (см. рис. 4.8), названной контрапозиция, и имеющей вид $(\alpha \Rightarrow \beta) \Leftrightarrow (\neg \beta \Rightarrow \neg \alpha)$, примененной к предложению R_7 выводится предложение

$$R_8: \neg B_{1,1} \Rightarrow \neg (P_{1,2} \vee P_{2,1})$$

Теперь применим правило Модус Поненс к предложению R_8 с учётом того, что в базе знаний уже имеется предложение R_4 :$\neg B_{1,1}$. В результате получаем предложение

$$R_9: \neg (P_{1,2} \vee P_{2,1})$$

Наконец, при помощи стандартной эквивалентность (см. рис. 4.8), названной правило де Моргана, и имеющей вид $\neg(\alpha \vee \beta) \Leftrightarrow (\neg \alpha \wedge \neg \beta)$, примененной к предложению R_9 выводим предложение:

$$R_{10}: \neg P_{1,2} \wedge \neg P_{2,1}$$

Из последнего предложения выводится $\neg P_{1,2}$ и $\neg P_{2,1}$ смысл которых заключается в том, что в квадратах [1, 2] и [2, 1] отсутствуют ямы. Отметим, что промежуточные правила ($R_7 - R_{10}$), которые генерировались на каждом шаге вывода, регулярно пополняли базу знаний.

4.5.1. Правило резолюции

Последовательно применяя различные правила вывода, в число которых мы включили и стандартные эквивалентности, к исходной базе знаний, мы вывели новые знания о том, что в квадратах [1, 2] и [2, 1] отсутствуют ямы. Теперь мы точно знаем, что квадраты [1, 2] и [2, 1] безопасны и агент может выполнить действие, перемещающего его в один из этих квадратов. Однако, разработать программу, которая могла бы осуществлять подобный вывод очень сложно, если вообще возможно. Вывод, подобный тому, который мы только что проделали, требует наличия опыта, который трудно воплотить в компьютерной программе. Главная проблема, с которой сталкивается разработчик, может быть сформулирована в виде вопроса: «Какое правило вывода необходимо применить на данном шаге, и к каким предложениям базы знаний оно должно быть применено?» Задача разработки программы логического вывода была бы значительно проще, если бы существовало одно универсальное правило вывода, которое можно было бы применять на каждом шаге логического вывода. Настоящий параграф посвящён изучению такого универ-

сального правила логического вывода, которое получило наименование *правила резолюции* или *правила устранения противоречия*. Если алгоритм вывода строится на основе правила резолюции, то исчезает проблемы выбора правила вывода. Однако, использование правила резолюции для разработки алгоритма вывода накладывает ограничения на форму записи предложений базы знаний. Это правило применимо только к *предложениям, представленным в виде дизъюнкции литералов*.

Начнём знакомство с правилом резолюции с еще одного примера логического вывода в мире Вампуса. Рассмотрим состояние мира Вампуса, изображенное на рис. 4.4а). Агент вернулся из квадрата [2, 1] в квадрат [1, 1], а затем перешел в безопасный квадрат [1, 2], где он воспринял неприятный запах, но не воспринял ветерок. Пусть очередной целью агента является доказательство того, что яма находится в квадрате [3, 1].

Начнем с того, что получим предложение в виде дизъюнкции предложений $P_{1,1}$, $P_{2,2}$, $P_{3,1}$, к которому, затем, применим правило резолюции. Опишем дополнительные знания агента, находящегося в квадрате [1, 2] предложениями R_{11} и R_{12}, которыми пополним базу знаний

$R_{11}: \neg B_{1,2}$

$R_{12}: B_{1,2} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{1,3})$

С помощью того же процесса, который привёл к получению предложения R_{10} можно сделать заключение об отсутствии ям в квадратах [2, 2] и [1, 3] и, следовательно, пополнить базу знаний ещё двумя предложениями R_{13} и R_{14}.Напомним, что мы уже знаем, что в квадрате [1, 1] ямы нет.

$R_{13}: \neg P_{2,2}$

$R_{14}: \neg P_{1,3}$

Применим правило удаления двухсторонней импликации к предложению $R_3: B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$, после чего к полученному предложению и предложению $R_5: B_{2,1}$ применим правило Модус Поненс. В результате получаем предложение

$R_{15}: P_{1,1} \vee P_{2,2} \vee P_{3,1}$

Теперь мы подошли к этапу, когда можно воспользоваться правилом резолюции. Если мы рассмотрим совместно предложения R_{13} и R_{15}, то обнаружим, что литерал $\neg P_{2,2}$ в предложении R_{13} противоположен литералу $P_{2,2}$ в предложении R_{15}. Этот факт позволяет нам устранить литерал $P_{2,2}$ в предложении R_{15}. В результате получаем предложение

$R_{16}: P_{1,1} \vee P_{3,1}$

Устранение литерала $P_{2,2}$ в предложении R_{15} можно пояснить следующим образом. Если в одном из квадратов [1, 1], [2, 2] или [3, 1] имеется яма, но её нет в квадрате [2, 2], то яма должна быть в квадрате [1, 1] или [3, 1].

Аналогичным образом устраняется литерал $P_{1,1}$ в предложении R_{16}, поскольку в предложении $R_1: \neg P_{1,1}$ имеется противоположный литерал. В результате получаем предложение

$R_{17}: P_{3,1}$

Устранение литерала $P_{1,1}$ в предложении R_{16} объясняется следующим образом. Если в одном из квадратов [1, 1] или [3, 1] имеется яма, но её нет в квадрате [1, 1], то яма должна быть в квадрате [3, 1].

Два последних этапа логического вывода представляют собой примеры применения правила логического вывода, называемого *правилом одиночной резолюции*. Предложение, выводимое в результате применения правила одиночной резолюции называется *резольвента*. В рассматриваемом примере резольвентами являются предложения R_{16} и R_{17}. Правило одиночной резолюции можно записать следующим образом:

$$\frac{L_1 \vee \ldots \vee L_i \vee \ldots \vee L_k,\ m}{L_1 \vee \ldots \vee L_{i-1} \vee L_{i+1} \vee \ldots \vee L_k}$$

В правиле одиночной резолюции каждое из предложений L представляет собой литерал,

а предложения L_i и m являются *взаимно обратными литералами*, т.е. такими, что один из них является отрицанием другого. Таким образом, при помощи правила одиночной резолюции из дизъюнкции литералов и ещё одного литерала выводится новая, более простая, дизъюнкция литералов, называемая резольвентой.

Правило одиночной резолюции может быть обобщено до *правила полной резолюции*. В случае дизъюнкции, состоящей только из двух дизъюнктов, правило полной резолюции записывается следующим образом:

$$\frac{L_1 \vee L_2,\ \neg L_2 \vee L_3}{L_1 \vee L_3}$$

Последняя запись правила резолюции означает, что берутся две дизъюнкции литералов, из которых формируется новая дизъюнкция, содержащая все литералы двух первоначальных дизъюнкций, за исключением взаимно обратных литералов. Например, может иметь место такой логический вывод:

$$\frac{P_{1,1} \vee P_{3,1},\ \neg P_{1,1} \vee \neg P_{2,2}}{P_{3,1} \vee \neg P_{2,2}}$$

4.5.2. Конъюнктивная нормальная форма

Правило резолюции применяется только к дизъюнкции литералов, поэтому на первый взгляд оно распространяется только на базы знаний, предложения которых представляют собой дизъюнкции и литералы. Однако это правило может служить основой алгоритма логического вывода для любой базы данных, поскольку *каждое предложение пропозициональной логики логически эквивалентно конъюнкции дизъюнкций литералов*.

Любое предложение, представленное как конъюнкция дизъюнкций литералов, называется предложением, находящимся в *конъюнктивной нормальной форме*, или *CNF (Conjunctive Normal Form)*. Кроме того целесообразно определить также ограниченный класс предложений в конъюнктивной нормальной форме, называемых предложениями в форме *k-CNF*. Предложение в форме k-CNF имеет точно k литералов в каждой дизъюнкции (каждый конъюнкт состоит из k дизъюнктов)

$$(l_{1,1} \vee \ldots \vee l_{1,k}) \wedge \ldots \wedge (l_{n,1} \vee \ldots \vee l_{n,k})$$

В качестве примера, рассмотрим процесс преобразования предложения R_2: $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$ в CNF.

1. Устраним связку $\Leftrightarrow$, путём замены предложения типа $\alpha \Leftrightarrow \beta$ эквивалентным предложением $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$ (см. рис. 4.8). В результате получаем $(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge (P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$
2. Устраним связку $\Rightarrow$, заменив предложение типа $\alpha \Rightarrow \beta$ эквивалентным предложением $\neg\alpha \vee \beta$ (см. рис. 4.8). В результате получаем следующее предложение $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})$
3. Конъюнктивная нормальная форма требует, чтобы связка $\neg$ появлялась только перед литералами. Поэтому выполним операцию, которую обычно называют «введение связки $\neg$ внутрь предложения» Для этого применим второе из правил де Моргана $\neg(\alpha \vee \beta) \Leftrightarrow (\neg\alpha \wedge \neg \beta)$ (см. рис. 4.8). В результате получаем $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})$
4. Мы получили предложение, содержащее вложенные связки $\wedge$ и $\vee$, которые применяются к литералам. Используя законы дистрибутивности (см. рис. 4.8) приведём последнее предложение к CNF. В результате получаем следующее предложение $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$

В итоге, первоначальное предложение представлено в CNF. Оно стало более сложным для восприятия человеком, однако теперь его можно использовать в качестве входных данных для алгоритма логического вывода, работающего на основе правила резолю-

ции.

4.5.3. Алгоритм логического вывода на основе правила резолюции

Программы логического вывода, основанные на правиле резолюции, действуют с использованием принципа доказательства путём установления противоречия. Это означает, что вывод предложения α из базы знаний KB $\models \alpha$ заменяется доказательством того, что предложение $(\text{KB} \wedge \neg\alpha)$ является *невыполнимым*. Иными словами необходимо доказать, что не существует модели в которой предложение $(\text{KB} \wedge \neg\alpha)$ является истинным.

Программа логического вывода, основанная на правиле резолюции, приведена на рис. 4.9.

```
function Resolution(KB, α) returns true или false
   inputs: KB, база знаний в виде логических предложений
           α, запрос в виде логического предложения
   clauses ← множество предложений, полученных после
             преобразования KB ∧ ¬α в CNF
   new ← {}
   loop do
      for each Ci, Cj in clauses do
         resolvents ← Resolve(Ci, Cj)
         if множество resolvents содержит пустое предложение
            then return true
         new ← new ∪ resolvents
      if new ⊆ clauses then return false
      clauses ← clauses ∪ new
```

Рис. 4.9. Алгоритм вывода, основанный на правиле резолюции для пропозициональной логики.

Вначале предложение $(\text{KB} \wedge \neg\alpha)$ преобразуется в CNF. Затем к результирующим предложениям применяется правило резолюции. В каждой паре предложений, содержащих взаимно противоположные литералы, происходит удаление этих противоположных литералов для получения нового предложения, которое добавляется к множеству существующих предложений, если в этом множестве ещё нет такого предложения.

Отмеченный процесс продолжается о тех пор, пока не происходит одно из следующих двух событий:

1. перестают вырабатываться какие-либо новые предложения, которые могли быть добавлены к существующим, и в этом случае из базы знаний KB не следует предложение α;
2. два противоположных друг другу предложения устраняются, в результате чего создаётся пустое предложение, и в этом случае из базы знаний следует предложение α.

4.5.4. Хорновские предложения

Алгоритм вывода, использующий правило резолюции часто используется в программах искусственного интеллекта, реализующих идею агента базы знаний. Однако во многих практических ситуациях вся универсальность правила резолюции не требуется. Реальные базы знаний часто содержат только предложения в ограниченной форме, называемые *хорновскими предложениями*.

Хорновское предложение представляет собой дизъюнкцию литералов, среди которых положительным является только один литерал. Например, предложение:

$$(\neg \text{Loc}_{1,1} \vee \neg \text{Breeze} \vee \text{B}_{1,1})$$

где $\text{Loc}_{1,1}$ означает, что местонахождением агента является квадрат [1, 1], представляет собой хорновское предложение, тогда как предложение:

$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1})$

таковым не является.

Ограничение, согласно которому только один литерал предложения должен быть положительным, может на первый взгляд показаться немного искусственным, но фактически является важным по следующим причинам.

1. Каждое хорновское предложение может быть описано как импликация. Например, хорновское предложение $(\neg Loc_{1,1} \vee \neg Breeze \vee B_{1,1})$ может быть записано в виде $(Loc_{1,1} \wedge Breeze) \Rightarrow B_{1,1}$. Предложение, записанное в такой форме, более естественно для понимания человеком. В нём утверждается, что если агент находится в квадрате [1, 1] и воспринимает ветерок, то ветерок имеется в квадрате [1, 1].
2. Логический вывод с использованием хорновских предложений может осуществляться с помощью, как алгоритма *прямого логического вывода*, так и алгоритма *обратного логического вывода*. Оба эти алгоритма являются естественными в том смысле, что этапы логического вывода являются для людей очевидными и их легко проследить.
3. Получение логических следствий с помощью хорновских предложений может осуществляться за время, линейно зависящее от размера базы знаний.

Упражнения

4.1. Опишите мир Вампуса в соответствии со свойствами проблемной среды, перечисленными в 1.2.2.

4.2. Рассмотрите словарь, состоящий из трѐх предложений A, B, и C. Запишите модели для каждого из следующих предложений.

a. $(A \wedge B) \vee (B \wedge C)$

b. $A \vee B$

c. $(A \Leftrightarrow B) \Leftrightarrow C$

4.3. Проверьте каждую из эквивалентностей, приведенных на рис 4.8.

4.4. При помощи таблицы истинности определите, является ли допустимым каждое из приведенных предложений.

a. $Smoke \Rightarrow Smoke$

b. $Smoke \Rightarrow Fire$

c. $(Smoke \Rightarrow Fire) \Rightarrow (\neg Smoke \Rightarrow \neg Fire)$

d. $Smoke \vee Fire \vee \neg Fire$

e. $((Smoke \wedge Heat) \Rightarrow Fire) \Leftrightarrow ((Smoke \Rightarrow Fire) \vee (Heat \Rightarrow Fire))$

f. $(Smoke \Rightarrow Fire) \Rightarrow ((Smoke \wedge Heat) \Rightarrow Fire)$

g. $Big \vee Dumb \vee (Big \Rightarrow Dumb)$

h. $(Big \wedge Dumb) \vee \neg Dumb$

4.5. Можете ли вы доказать на основании приведенных ниже рассуждений, что единорог – мифическое существо? А что насчѐт того, что это волшебное существо? Существо, вооружѐнное рогом?

Если единорог мифическое существо, то он бессмертен, а если не мифическое, то он смертное млекопитающее. Если единорог либо бессмертен, либо является млекопитающим, то он вооружѐн рогом. Единорог является волшебным существом, если он вооружѐн рогом.

ЛИТЕРАТУРА ДЛЯ УГЛУБЛЕННОГО ИЗУЧЕНИЯ

1. Stuard Russel, Peter Norvig. Artificial Intelligence: A Modern Approach (3rd Edition). Prentice Hall, 2009.
2. Стюарт Рассел, Питер Норвиг. Искусственный интеллект. Современный подход. Второе издание. Из-во Вильямс, 2006.