

формацию и принимает решение о выполнении действий. Обучающий компонент использует информацию обратной связи от **критика** с оценкой того, как действует агент, и определяет, каким образом должен быть модифицирован производительный компонент для того, чтобы он успешнее действовал в будущем.



Рис. 2.7. Общая модель обучающегося агентов

Проект обучающего компонента во многом зависит от проекта производительного компонента. Осуществляя попытку спроектировать агента, который обучается определенным способностям, необходимо прежде всего стремиться найти ответ на вопрос: “Какого рода производительный компонент потребуется моему агенту после того, как он будет обучен тому, как выполнять свои функции?”, а не на вопрос: “Как приступить к решению задачи обучения его выполнению этих функций?” После того как спроектирован сам агент, можно приступить к конструированию обучающих механизмов, позволяющих усовершенствовать любую часть этого агента.

Критик сообщает обучающему компоненту, насколько хорошо действует агент с учетом постоянного стандарта производительности. Критик необходим, поскольку сами результаты восприятия не дают никаких указаний на то, успешно ли действует агент. Например, шахматная программа может получить результаты восприятия, указывающие на то, что она поставила мат своему противнику, но ей требуется стандарт производительности, который позволил бы определить, что это — хороший результат; сами данные восприятия ничего об этом не говорят. Важно, чтобы стандарт производительности был постоянным. В принципе этот стандарт следует рассматривать как полностью внешний по отношению к агенту, поскольку агент не должен иметь возможности его модифицировать так, чтобы он в большей степени соответствовал его собственному поведению.

Последним компонентом обучающегося агента является **генератор проблем**. Его задача состоит в том, чтобы предлагать действия, которые должны привести к получению нового и информативного опыта. Дело в том, что если производительный компонент предоставлен самому себе, то продолжает выполнять действия, которые являются наилучшими с точки зрения того, что он знает. Но если агент готов

к тому, чтобы немного поэкспериментировать и в кратковременной перспективе выполнять действия, которые, возможно, окажутся не совсем оптимальными, то он может обнаружить гораздо более лучшие действия с точки зрения долговременной перспективы. Генератор проблем предназначен именно для того, чтобы предлагать такие исследовательские действия. Именно этим занимаются ученые, проводя эксперименты. Галилей не считал, что сбрасывание камней с вершины Пизанской башни является самоцелью. Он не старался просто вдрызг разбить эти булыжники или оказать физическое воздействие на головы неудачливых прохожих. Его замысел состоял в том, чтобы изменить взгляды, сложившиеся в его собственной голове, сформулировав лучшую теорию движения объектов.

Для того чтобы перевести весь этот проект на конкретную почву, вернемся к примеру автоматизированного такси. Производительный компонент состоит из той коллекции знаний и процедур, которая применяется водителем такси при выборе им действий по вождению. Водитель такси с помощью этого производительного компонента выезжает на дорогу и ведет свою машину. Критик наблюдает за миром и в ходе этого передает соответствующую информацию обучающему компоненту. Например, после того как такси быстро выполняет поворот налево, пересекая три полосы движения, критик замечает, какие шокирующие выражения используют другие водители. На основании этого опыта обучающий компонент способен сформулировать правило, которое гласит, что это — недопустимое действие, а производительный компонент модифицируется путем установки нового правила. Генератор проблем может определить некоторые области поведения, требующие усовершенствования, и предложить эксперименты, такие как проверка тормозов на разных дорожных покрытиях и при различных условиях.

Обучающий компонент может вносить изменения в любой из компонентов “знаний”, показанных на схемах агентов (см. рис. 2.3–2.6). В простейших случаях обучение будет осуществляться непосредственно на основании последовательности актов восприятия. Наблюдение за парами последовательных состояний среды позволяет агенту освоить информацию о том, “как изменяется мир”, а наблюдение за результатами своих действий может дать агенту возможность узнать, “какое влияние оказывают мои действия”. Например, после того как водитель такси приложит определенное тормозное давление во время езды по мокрой дороге, он вскоре узнает, какое снижение скорости фактически было достигнуто. Очевидно, что эти две задачи обучения становятся более сложными, если среда наблюдаема лишь частично.

Те формы обучения, которые были описаны в предыдущем абзаце, не требуют доступа к внешнему стандарту производительности, вернее, в них применяется универсальный стандарт, согласно которому сделанные прогнозы должны быть согласованы с экспериментом. Ситуация становится немного сложнее, когда речь идет об агенте, основанном на полезности, который стремится освоить в процессе обучения информацию о полезности. Например, предположим, что агент, занимающийся вождением такси, перестает получать чаевые от пассажиров, которые в ходе утомительной поездки почувствовали себя полностью разбитыми. Внешний стандарт производительности должен информировать агента, что отсутствие чаевых — это отрицательный вклад в его общую производительность; в таком случае агент получает возможность освоить в результате обучения, что грубые маневры, утомляющие пассажиров, не позволяют повысить оценку его собственной функции полезности. В этом смысле стандарт производительности позволяет выделить определенную

часть входных результатов восприятия как **вознаграждение** (или **штраф**), непосредственно предоставляемое данными обратной связи, влияющими на качество поведения агента. Именно с этой точки зрения могут рассматриваться жестко закрепленные стандарты производительности, такие как боль или голод, которыми характеризуются жизнь животных. Эта тема рассматривается более подробно в главе 21.

Подводя итог, отметим, что агенты имеют самые различные компоненты, а сами эти компоненты могут быть представлены в программе агента многими способами, поэтому создается впечатление, что разнообразие методов обучения чрезвычайно велико. Тем не менее все эти методы имеют единый объединяющий их аспект. Процесс обучения, осуществляемый в интеллектуальных агентах, можно в целом охарактеризовать как процесс модификации каждого компонента агента для обеспечения более точного соответствия этих компонентов доступной информации обратной связи и тем самым улучшения общей производительности агента.

## 2.5. РЕЗЮМЕ

---

Эту главу можно сравнить с головокружительным полетом над обширным ландшафтом искусственного интеллекта, который мы рассматриваем как науку проектирования агентов. Ниже кратко приведены основные идеи, которые рассматривались в данной главе.

- **Агентом** является нечто воспринимающее и действующее в определенной среде. **Функция агента** определяет действие, предпринимаемое агентом в ответ на любую последовательность актов восприятия.
- **Показатели производительности** оценивают поведение агента в среде. **Рациональный агент** действует так, чтобы максимизировать ожидаемые значения показателей производительности, с учетом последовательности актов восприятия, полученной агентом к данному моменту.
- Спецификация **проблемной среды** включает определения показателей производительности, внешней среды, исполнительных механизмов и датчиков. Первым этапом проектирования агента всегда должно быть определение проблемной среды с наибольшей возможной полнотой.
- Варианты проблемной среды классифицируются по нескольким важным размерностям. Они могут быть полностью или частично наблюдаемыми, детерминированными или стохастическими, эпизодическими или последовательными, статическими или динамическими, дискретными или непрерывными, а также одноагентными или мультиагентными.
- **Программа агента** реализует функцию агента. Существует целый ряд основных проектов программ агента, соответствующих характеру явно воспринимаемой информации, которая используется в процессе принятия решения. Разные проекты характеризуются различной эффективностью, компактностью и гибкостью. Выбор наиболее подходящего проекта программы агента зависит от характера среды.
- **Простые рефлексивные агенты** отвечают непосредственно на акты восприятия, тогда как **рефлексивные агенты, основанные на модели**, поддерживают внутреннее

состояние, прослеживая те аспекты среды, которые не наблюдаются в текущем акте восприятия. **Агенты, действующие на основе цели**, организуют свои действия так, чтобы достигнуть своих целей, а **агенты, действующие с учетом полезности**, пытаются максимизировать свою собственную ожидаемую “удовлетворенность”.

- Все агенты способны улучшать свою работу благодаря обучению.

## БИБЛИОГРАФИЧЕСКИЕ И ИСТОРИЧЕСКИЕ ЗАМЕТКИ

Истоки представлений о центральной роли действий в интеллекте (сформулированных в виде понятия *практических рассуждений*) прослеживаются до *Никомаховой этики* Аристотеля. Практические рассуждения были также темой влиятельной статьи Маккарти *Programs with Common Sense* (Программы со здравым смыслом) [1009]. Такие области науки, как робототехника и теория управления, по самому своему характеру преимущественно касаются проблем конструирования физических агентов. Понятие **контроллера** в теории управления идентично понятию агента в искусственном интеллекте. И хотя на первый взгляд это может показаться удивительным, но на протяжении большей части истории развития искусственного интеллекта основные усилия в этой области были сосредоточены на исследовании отдельных компонентов агентов (в качестве примеров можно привести системы поиска ответов на вопросы, программы автоматического доказательства теорем, системы технического зрения и т.д.), а не самих агентов, рассматриваемых как единое целое. Важным исключением из этого правила, оказавшим значительное влияние на дальнейшее развитие данной области, стало обсуждение проблематики агентов в работе Генезерета и Нильссона [537]. В настоящее время в данной научной области широко применяется подход, основанный на изучении всего агента, и результаты, достигнутые в рамках этого подхода, стали центральной темой новейших работ [1146], [1227].

В главе 1 было показано, что корни понятия *рациональности* прослеживаются в философии и экономике. В искусственном интеллекте это понятие не привлекало значительного интереса до тех пор, пока в середине 1980-х не началось широкое обсуждение проблемы создания подходящих теоретических основ данной области. В статье Джона Дойла [410] было предсказано, что со временем проектирование рациональных агентов станет рассматриваться в качестве основного назначения искусственного интеллекта, притом что другие популярные ныне темы исследований послужат основой формирования новых дисциплин.

Стремление к тщательному изучению свойств среды и их влияния на выбор самого подходящего проекта рационального агента наиболее ярко проявляется в традиционных областях теории управления; например, в исследованиях по классическим системам управления [405] рассматриваются полностью наблюдаемые, детерминированные варианты среды; темой работ по стохастическому оптимальному управлению [865] являются частично наблюдаемые, стохастические варианты среды; а работы по гибридному управлению [649] касаются таких вариантов среды, которые содержат и дискретные, и непрерывные элементы. Описание различий между полностью и частично наблюдаемыми вариантами среды является также центральной темой работ по **динамическому программированию**, проводимых в области исследования операций [1244], которая будет обсуждаться в главе 17.

Рефлексные агенты были основной моделью для психологических бихевиористов, таких как Скиннер [1423], которые пытались свести все знания в области психологии организмов исключительно к отображениям “ввод–вывод” или “стимул–отклик”. В результате прогресса в области психологии, связанного с переходом от бихевиоризма к функционализму, который был по крайней мере отчасти обусловлен распространением на агентов трактовки понятия компьютера как новой метафоры в мировоззрении человека [923], [1246], возникло понимание того, что нужно также учитывать внутреннее состояние агента. В большинстве работ по искусственному интеллекту идея чисто рефлексных агентов с внутренним состоянием рассматривалась как слишком простая для того, чтобы на ее основе можно было добиться значительных успехов, но исследования Розеншайна [1308] и Брукса [189] позволили поставить под сомнение это предположение (см. главу 25). В последние годы значительная часть работ была посвящена поиску эффективных алгоритмов слежения за сложными вариантами среды [610]. Наиболее впечатляющим примером достигнутых при этом результатов является программа Remote Agent, которая управляла космическим аппаратом Deep Space One (описанная на с. 69) [744], [1108].

Понимание необходимости создания агентов на основе цели просматривается во всем, что стало источником идей искусственного интеллекта, начиная с введенного Аристотелем определения практического рассуждения и заканчивая ранними статьями Маккарти по логическому искусственному интеллекту. Робот Shakey [466], [1143] был первым воплощением логического агента на основе цели в виде автоматизированного устройства. Полный логический анализ агентов на основе цели приведен в книге Генезерета и Нильссона [537], а методология программирования на основе цели, получившая название *агентно-ориентированного программирования*, была разработана Шохемом [1404].

Кроме того, начиная с книги *Human Problem Solving* [1130], оказавшей чрезвычайно большое влияние на ход дальнейших исследований, в той области когнитивной психологии, которая посвящена изучению процедур решения задач человеком, ведущее место занял подход, основанный на понятии цели; на этом подходе базируется также вся последняя работа Ньюэлла [1125]. Цели, дополнительно подразделяемые на желания (общие замыслы) и намерения (осуществляемые в настоящее время), являются основой теории агентов, разработанной Братманом [176]. Эта теория оказала влияние и на работы в области понимания естественного языка, и на исследования мультиагентных систем.

Горвиц, наряду с другими исследователями [686], особо подчеркивал важность использования в качестве основы для искусственного интеллекта понятия *рациональности*, рассматриваемой как средство максимизации ожидаемой полезности. Книга Перла [1191] была первой работой в области искусственного интеллекта, в которой дан глубокий анализ проблем применения теории вероятности и теории полезности; описанные им практические методы проведения рассуждений и принятия решений в условиях неопределенности, по-видимому, послужили наиболее важной причиной быстрой смены направления исследований в 1990-х годах и перехода к изучению агентов, действующих с учетом полезности (подробнее об этом рассказывается в части V).

Общий проект для обучающихся агентов, приведенный на рис. 2.7, является классическим образцом такого проекта в литературе по машинному обучению [203], [1064]. Одним из первых примеров воплощения этого проекта в программах являет-

ся обучающаяся программа для игры в шашки Артура Самюэла [1349], [1350]. Обучающиеся агенты подробно рассматриваются в части VI.

В последние годы быстро растет интерес к агентам и к проектам агентов, отчасти в связи с расширением Internet и осознанием необходимости создания автоматизированных и мобильных **программных роботов** [447]. Сборники статей по этой теме можно найти в работах *Readings in Agents* [704] и *Foundations of Rational Agency* [1615]. В книге *Multiagent Systems* [1564] предоставлены надежные теоретические основы для многих аспектов проектирования агентов. К числу конференций, посвященных агентам, относятся *International Conference on Autonomous Agents*, *International Workshop on Agent Theories, Architectures, and Languages* и *International Conference on Multiagent Systems*. И наконец отметим, что в книге *Dung Beetle Ecology* [615] приведен большой объем интересной информации о поведении навозных жуков (о котором говорилось в данной главе).

## УПРАЖНЕНИЯ

---


- 2.1. Самостоятельно сформулируйте определения следующих понятий: агент; функция агента; программа агента; рациональность; автономность; рефлексный агент; агент, основанный на модели; агент на основе цели; агент на основе полезности; обучающийся агент.
- 2.2. Для измерения того, насколько успешно функционирует агент, используются и показатели производительности, и функция полезности. Объясните, в чем состоит различие между этими двумя критериями.
- 2.3. В этом упражнении исследуются различия между функциями агента и программами агента.
  - а) Может ли существовать больше чем одна программа агента, которая реализует данную функцию агента? Приведите пример, подтверждающий положительный ответ, или покажите, почему такая ситуация невозможна.
  - б) Есть ли такие функции агента, которые не могут быть реализованы никакими программами агента?
  - в) Верно ли, что каждая программа агента реализует точно одну функцию агента, при условии, что архитектура вычислительной машины остается неизменной?
  - г) Если в архитектуре предусмотрено  $n$  битов памяти, то сколько различных возможных программ агента может быть реализовано с ее помощью?
- 2.4. В этом упражнении исследуется рациональность различных функций агента для пылесоса.
  - а) Покажите, что простая функция агента—пылесоса, описанная в табл. 2.1, действительно рациональна, согласно предположениям, перечисленным на с. 79.
  - б) Опишите рациональную функцию агента для модифицированного показателя производительности, который предусматривает штраф в один пункт за каждое движение. Требуется ли поддержка внутреннего состояния для соответствующей программы агента?

- в) Обсудите возможные проекты агентов для случаев, в которых чистые квадраты могут стать грязными, а география среды неизвестна. Имеет ли смысл в таких случаях для агента обучаться на своем опыте? Если да, то что он должен изучать?

2.5. Для каждого из следующих агентов разработайте описание PEAS среды задачи:

- а) робот-футболист;
- б) агент, совершающий покупки книг в Internet;
- в) автономный марсианский вездеход;
- г) ассистент математика, занимающийся доказательством теорем.

2.6. Для каждого из типов агентов, перечисленных в упр. 2.5, охарактеризуйте среду в соответствии со свойствами, приведенными в разделе 2.3, и выберите подходящий проект агента.

 Все приведенные ниже упражнения касаются реализации вариантов среды и агентов для мира пылесоса.

2.7. Реализуйте имитатор среды измерения производительности для мира пылесоса, который изображен на рис. 2.2 и определен на с. 79. Предложенная реализация должна быть модульной, для того чтобы можно было заменять датчики и исполнительные механизмы, а также модифицировать характеристики среды (размер, форму, размещение мусора и т.д.). (Примечание. Для некоторых сочетаний языка программирования и операционной системы в оперативном репозитории кода уже имеются реализации.)

2.8. Реализуйте простого рефлексного агента для среды пылесоса, которая рассматривается в упр. 2.7. Вызовите на выполнение имитатор среды с этим агентом для всех возможных начальных конфигураций мусора и местоположений агента. Зарегистрируйте оценки производительности работы агента для каждой конфигурации и определите его общую среднюю оценку.

2.9. Рассмотрите модифицированную версию среды пылесоса из упр. 2.7, в которой агенту назначается штраф в один пункт за каждое движение.

- а) Может ли быть полностью рациональным простой рефлексный агент для этой среды? Обоснуйте свой ответ.
- б) А что можно сказать о рефлексном агенте с внутренним состоянием? Спроектируйте такого агента.
- в) Как изменятся приведенные вами ответы на вопросы а) и б), если акты восприятия агента позволяют ему иметь информацию о том, является ли чистым или грязным каждый квадрат в этой среде?

2.10. Рассмотрите модифицированную версию среды пылесоса из упр. 2.7, в которой неизвестны география среды (ее протяженность, границы и препятствия), а также начальная конфигурация расположения мусора. (Агент может совершать движения *Left* и *Right*, а также *Up* и *Down*.)

- а) Может ли быть полностью рациональным простой рефлексный агент для этой среды? Обоснуйте свой ответ.
- б) Может ли простой рефлексный агент с рандомизированной функцией агента превзойти по своей производительности простого рефлексного

агента? Спроектируйте такого агента и измерьте его производительность в нескольких вариантах среды.

- в) Можете ли вы спроектировать среду, в которой предложенный вами рандомизированный агент будет показывать очень низкую производительность? Продемонстрируйте полученные вами результаты.
- г) Может ли рефлексный агент с поддержкой состояния превзойти по своей производительности простого рефлексного агента? Спроектируйте такого агента и измерьте его производительность в нескольких вариантах среды. Сумеете ли вы спроектировать рационального агента этого типа?

2.11. Повторите упр. 2.10 для такого случая, в котором датчик местоположения заменен датчиком удара, позволяющим обнаруживать попытки агента пройти сквозь препятствие или пересечь границы среды. Предположим, что датчик удара перестал работать; как должен действовать агент?

2.12. Все варианты среды пылесоса, рассматриваемые в предыдущих упражнениях, были детерминированными. Обсудите возможные программы агента для каждого из следующих стохастических вариантов.

- а) Закон Мэрфи: в течение 25% времени применение действия *Suck* не позволяет очистить пол, если пол грязный, и приводит к появлению мусора на полу, если пол чистый. Как эти обстоятельства отразятся на предложенной вами программе агента, если датчик мусора дает неправильный ответ в течение 10% времени?
- б) Маленькие дети: в каждом интервале времени каждый чистый квадрат имеет 10%-ную вероятность стать грязным. Можете ли вы предложить проект рационального агента для этого случая?



## Часть II

# РЕШЕНИЕ ПРОБЛЕМ

Решение проблем посредством поиска	110
Информированный поиск и исследование пространства состояний	153
Задачи удовлетворения ограничений	209
Поиск в условиях противодействия	240

# 3 РЕШЕНИЕ ПРОБЛЕМ ПОСРЕДСТВОМ ПОИСКА

*В этой главе показано, каким образом агент может найти последовательность действий, позволяющую достичь его целей в тех условиях, когда единственного действия для этого недостаточно.*

Простейшими агентами, которые рассматривались в главе 2, были рефлексные агенты, функционирование которых основано на непосредственном отображении состояний в действия. Подобные агенты не могут успешно действовать в тех вариантах среды, где такие отображения были бы слишком большими, чтобы можно было обеспечить их хранение, а изучение отображений потребовало бы слишком много времени. Агенты на основе цели, с другой стороны, способны достичь успеха, рассматривая будущие действия и оценивая желательность их результатов.

Данная глава посвящена описанию одной разновидности агента на основе цели, называемой **агентом, решающим задачи**. Агенты, решающие задачи, определяют, что делать, находя последовательности действий, которые ведут к желаемым состояниям. Начнем изложение этой темы с точного определения элементов, из которых состоит “задача” и ее “решение”, и приведем несколько примеров для иллюстрации этих определений. Затем представим несколько алгоритмов поиска общего назначения, которые могут использоваться для решения подобных задач, и проведем сравнение преимуществ и недостатков каждого алгоритма. Эти алгоритмы являются **неинформированными** в том смысле, что в них не используется какая-либо информация о рассматриваемой задаче, кроме ее определения. В главе 4 речь идет об **информированных** алгоритмах поиска, в которых используются определенные сведения о том, где следует искать решения.

В данной главе применяются понятия из области анализа алгоритмов. Читатели, незнакомые с понятиями асимптотической сложности (т.е. с системой обозначений  $O()$  и NP-полноты, должны обратиться к приложению А.

## 3.1. АГЕНТЫ, РЕШАЮЩИЕ ЗАДАЧИ

Предполагается, что интеллектуальные агенты обладают способностью максимизировать свои показатели производительности. Как уже упоминалось в главе 2, реа-

лизация указанного свойства в определенной степени упрощается, если агент способен принять на себя обязанность достичь цели и стремиться к ее удовлетворению. Вначале рассмотрим, как и почему агент может приобрести такую способность.

Представьте себе, что некоторый агент находится в городе Арад, Румыния, и проводит свой отпуск в качестве туриста. Показатели производительности агента включают много компонентов: он хочет улучшить свой загар, усовершенствовать знание румынского языка, осмотреть достопримечательности, насладиться ночной жизнью (со всеми ее привлекательными сторонами), избежать неприятностей и т.д. Эта задача принятия решения является сложной; для ее выполнения необходимо учитывать множество компромиссов и внимательно читать путеводители. Кроме того, предположим, что у агента имеется не подлежащий возмещению билет для вылета из Бухареста на следующий день. В данном случае для агента имеет смысл стремиться к достижению цели попасть в Бухарест. Способы действий, не позволяющие вовремя попасть в Бухарест, могут быть отвергнуты без дальнейшего рассмотрения и поэтому задача принятия решения агентом значительно упрощается. Цели позволяют организовать поведение, ограничивая выбор промежуточных этапов, которые пытается осуществить агент. Первым шагом в решении задачи является **формулировка цели** с учетом текущей ситуации и показателей производительности агента.

Мы будем рассматривать цель как множество состояний мира, а именно тех состояний, в которых достигается такая цель. Задача агента состоит в том, чтобы определить, какая последовательность действий приведет его в целевое состояние. Прежде чем это сделать, агент должен определить, какого рода действия и состояния ему необходимо рассмотреть. Но если бы агент пытался рассматривать действия на уровне “перемещения левой ноги вперед на один сантиметр” или “поворота рулевого колеса на один градус влево”, то, по-видимому, так и не смог бы выехать с автомобильной стоянки, не говоря уже о своевременном прибытии в Бухарест, поскольку на таком уровне детализации мир обладает слишком большой неопределенностью, а решение должно состоять из слишком многих этапов. **Формулировка задачи** представляет собой процесс определения того, какие действия и состояния следует рассматривать с учетом некоторой цели. Этот процесс будет описан более подробно немного позднее. А на данный момент предположим, что агент будет рассматривать действия на уровне автомобильной поездки из одного крупного города в другой. Таким образом, состояния, рассматриваемые агентом, соответствуют его пребыванию в конкретном городе<sup>1</sup>.

Итак, наш агент поставил перед собой цель доехать на автомобиле до Бухареста и определяет, куда отправиться для этого из Арада. Из этого города ведут три дороги: в Сибиу, Тимишоару и Зеринд. Прибытие в какой-либо из этих городов не представляет собой достижение намеченной цели, поэтому агент, не очень знакомый с географией Румынии, не будет знать, по какой дороге он должен следовать<sup>2</sup>. Иными словами, агент не знает, какое из его возможных действий является наилучшим, поскольку не обладает достаточными знаниями о состоянии, возникающем в результа-

<sup>1</sup> Обратите внимание на то, что каждое из этих “состояний” фактически соответствует большому множеству состояний мира, поскольку в состоянии реального мира должен быть определен каждый аспект действительности. Важно всегда учитывать различие между состояниями, которые рассматриваются в проблемной области решения задач, и состояниями реального мира.

<sup>2</sup> Авторы предполагают, что большинство читателей окажутся в таком же положении и вполне могут представить себя такими же беспомощными, как и наш агент. Приносим свои извинения румынским читателям, которые не смогут воспользоваться преимуществами использованного здесь педагогического приема.

те выполнения каждого действия. Если агент не получит дополнительных знаний, то окажется в тупике. Самое лучшее, что он может сделать, — это выбрать одно из указанных действий случайным образом.

Но предположим, что у агента есть карта Румынии, либо на бумаге, либо в его памяти. Карта способна обеспечить агента информацией о состояниях, в которых он может оказаться, и о действиях, которые он способен предпринять. Агент имеет возможность воспользоваться этой информацией для определения последовательных этапов гипотетического путешествия через каждый из этих трех городов в попытке найти такой путь, который в конечном итоге приведет его в Бухарест. Найдя на карте путь от Арада до Бухареста, агент может достичь своей цели, осуществляя действия по вождению автомобиля, которые соответствуют этапам этого путешествия. Вообще говоря, *«агент, имеющий несколько непосредственных вариантов выбора с неизвестной стоимостью, может решить, что делать, исследуя вначале различные возможные последовательности действий, которые ведут к состояниям с известной стоимостью, а затем выбирая из них наилучшую последовательность»*.

Описанный процесс определения такой последовательности называется **поиском**. Любой алгоритм поиска принимает в качестве входных данных некоторую задачу и возвращает **решение** в форме последовательности действий. После того как решение найдено, могут быть осуществлены действия, рекомендованные этим алгоритмом. Такое осуществление происходит на стадии **выполнения**. Итак, для рассматриваемого агента можно применить простой проект, позволяющий применить принцип “сформулировать, найти, выполнить”, как показано в листинге 3.1. После формулировки цели и решаемой задачи агент вызывает процедуру поиска для решения этой задачи. Затем он использует полученное решение для руководства своими действиями, выполняя в качестве следующего предпринимаемого мероприятия все, что рекомендовано в решении (как правило, таковым является первое действие последовательности), а затем удаляет этот этап из последовательности. Сразу после выполнения этого решения агент формулирует новую цель.

**Листинг 3.1.** Простой агент, решающий задачу. Вначале он формулирует цель и задачу, затем ищет последовательность действий, позволяющую решить эту задачу, и, наконец, осуществляет действия одно за другим. Закончив свою работу, агент формулирует другую цель и начинает сначала. Обратите внимание на то, что при выполнении очередной последовательности действий агент игнорирует данные своих актов восприятия, поскольку предполагает, что найденное им решение всегда выполнимо

---

```

function Simple-Problem-Solving-Agent(percept) returns действие action
  inputs: percept, результат восприятия
  static: seq, последовательность действий, первоначально пустая
           state, некоторое описание текущего состояния мира
           goal, цель, первоначально неопределенная
           problem, формулировка задачи

  state ← Update-State(state, percept)
  if последовательность seq пуста then do
    goal ← Formulate-Goal(state)
    problem ← Formulate-Problem(state, goal)
    seq ← Search(problem)
  action ← First(seq)
  seq ← Rest(seq)
  return action

```

Вначале опишем процесс составления формулировки задачи, а затем посвятим основную часть данной главы рассмотрению различных алгоритмов для функции Search. В этой главе осуществление функций Update-State и Formulate-Goal дополнительно не рассматривается.

Прежде чем перейти к подробному описанию, сделаем краткую паузу, чтобы определить, в чем агенты, решающие задачи, соответствуют описанию агентов и вариантов среды, приведенному в главе 2. В проекте агента, показанном в листинге 3.1, предполагается, что данная среда является **статической**, поскольку этапы формулировки и решения задачи осуществляются без учета каких-либо изменений, которые могут произойти в данной среде. Кроме того, в этом проекте агента допускается, что начальное состояние известно; получить такие сведения проще всего, если среда является **наблюдаемой**. К тому же в самой идее перечисления “альтернативных стратегий” заключается предположение, что среда может рассматриваться как **дискретная**. Последней и наиболее важной особенностью является следующее: в проекте агента предполагается, что среда является **детерминированной**. Решения задач представляют собой единственные последовательности действий, поэтому в них не могут учитываться какие-либо неожиданные события; более того, решения выполняются без учета результатов каких-либо актов восприятия! Агент, выполняющий свои планы, образно говоря, с закрытыми глазами, должен быть вполне уверенным в том, что он делает. (В теории управления подобный проект именуется системой **с разомкнутой обратной связью**, поскольку игнорирование результатов восприятия приводит к нарушению обратной связи между агентом и средой.) Все эти предположения означают, что мы имеем дело с простейшими разновидностями среды, и в этом состоит одна из причин, по которой настоящая глава помещена в самом начале данной книги. В разделе 3.6 дано краткое описание того, что произойдет, если будут исключены допущения о наблюдаемости и детерминированности, а в главах 12 и 17 приведено гораздо более подробное изложение соответствующей темы.

## Хорошо структурированные задачи и решения

✎ **Задача** может быть формально определена с помощью четырех *компонентов*, описанных ниже.

- ✎ **Начальное состояние**, в котором агент приступает к работе. Например, начальное состояние для нашего агента в Румынии может быть описано как пребывание в Араде,  $In(Arad)$ .
- Описание возможных действий, доступных агенту. В наиболее общей формулировке<sup>3</sup> используется ✎ **функция определения преемника**. Если задано конкретное состояние  $x$ , то функция  $Successor-Fn(x)$  возвращает множество упорядоченных пар  $\langle action, successor \rangle$ , где каждое действие  $action$  представляет собой одно из допустимых действий в состоянии  $x$ , а каждый преемник  $successor$  представляет собой состояние, которое может быть достигнуто из  $x$  путем применения этого действия. Например, из состояния

<sup>3</sup> В альтернативной формулировке используется множество **операторов**, которые могут быть применены к некоторому состоянию для выработки преемников.

$In(Arad)$  функция определения преемника для данной задачи проезда по Румынии возвратит следующее:

```
<Go(Sibiu), In(Sibiu)>, <Go(Timisoara), In(Timisoara)>,
<Go(Zerind), In(Zerind)>
```

Начальное состояние и функция определения преемника, вместе взятые, неявно задают  $\simeq$  **пространство состояний** данной задачи — множество всех состояний, достижимых из начального состояния. Пространство состояний образует граф, узлами которого являются состояния, а дугами между узлами — действия. (Карта Румынии, показанная на рис. 3.1, может интерпретироваться как граф пространства состояний, если каждая дорога рассматривается как обозначающая два действия проезда на автомобиле, по одному в каждом направлении.)  $\simeq$  **Путем** в пространстве состояний является последовательность состояний, соединенных последовательностью действий.

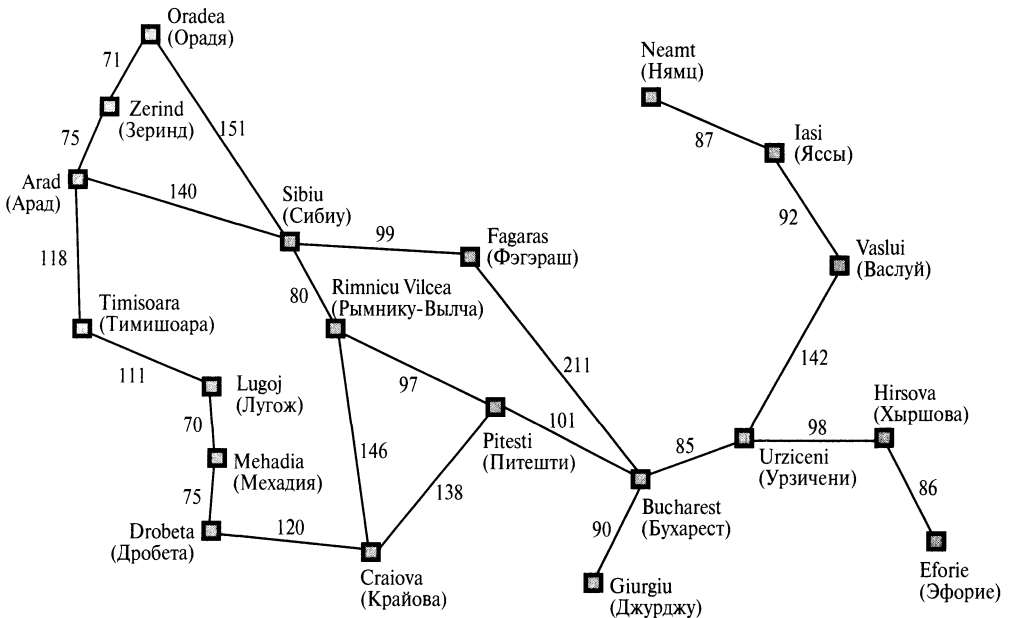


Рис. 3.1. Упрощенная дорожная карта части Румынии

- $\simeq$  **Проверка цели**, позволяющая определить, является ли данное конкретное состояние целевым состоянием. Иногда имеется явно заданное множество возможных целевых состояний, и эта проверка сводится просто к определению того, является ли данное состояние одним из них. Цель рассматриваемого агента в Румынии представляет собой одноэлементное множество  $\{In(Bucharest)\}$ . Иногда цель задается в виде абстрактного свойства, а не явно перечисленного множества состояний. Например, в шахматах цель состоит в достижении состояния, называемого “матом”, в котором король противника атакован и не может уклониться от удара.
- **Функция стоимости пути**, которая назначает числовое значение стоимости каждому пути. Агент, решающий задачу, выбирает функцию стоимости, кото-

рая соответствует его собственным показателям производительности. Для данного агента, пытающегося попасть в Бухарест, важнее всего время, поэтому стоимость пути может измеряться длиной в километрах. В настоящей главе предполагается, что стоимость пути может быть описана как сумма стоимостей отдельных действий, выполняемых вдоль этого пути.  $\Sigma$  **Стоимость этапа**, связанного с выполнением действия  $a$  для перехода из состояния  $x$  в состояние  $y$ , обозначается как  $c(x, a, y)$ . Стоимости этапов для Румынии показаны на рис. 3.1 в виде дорожных расстояний. Предполагается, что стоимости этапов являются неотрицательными<sup>4</sup>.

Описанные выше элементы определяют задачу и могут быть собраны вместе в единую структуру данных, которая передается в качестве входных данных в алгоритм решения задачи. **Решением** задачи является путь от начального состояния до целевого состояния. Качество решения измеряется с помощью функции стоимости пути, а  $\Sigma$  **оптимальным решением** является такое решение, которое имеет наименьшую стоимость пути среди всех прочих решений.

### Формулировка задачи

В предыдущем разделе была предложена формулировка задачи переезда в Бухарест в терминах начального состояния, функции определения преемника, проверки цели и стоимости пути. Эта формулировка кажется приемлемой, но в ней все же не учитываются слишком многие аспекты реального мира. Сравним простое выбранное нами описание состояния,  $In(Arad)$ , с действительным путешествием по стране, в котором состояние мира должно учитывать так много факторов: попутчиков, текущие радиопередачи, вид из окна, наличие поблизости представителей сил правопорядка, расстояние до ближайшей остановки на отдых, состояние дороги, погоду и т.д. Все эти соображения исключены из наших описаний состояния, поскольку они не имеют отношения к задаче поиска маршрута поездки в Бухарест. Процесс удаления деталей из представления называется  $\Sigma$  **абстрагированием**.

Кроме абстрагирования описаний состояний, необходимо абстрагировать сами действия. Любое действие по вождению влечет за собой много следствий. Такие действия не только изменяют местонахождение автомобиля и его пассажиров, но и занимают время, приводят к потреблению топлива, влекут за собой загрязнение окружающей среды и влияют на самого агента (как говорится, путешествия расширяют кругозор). В нашей формулировке учитывается только изменение местонахождения. Кроме того, существует много действий, которые мы вообще не рассматриваем: включение радиоприемника, осмотр окрестностей из окна, замедление движения по требованию дорожной полиции и т.д. К тому же, безусловно, действия здесь не задаются на уровне “поворота рулевого колеса влево на три градуса”.

Можно ли достичь большей точности определения приемлемого уровня абстракции? Будем считать, что выбранные нами абстрактные состояния и действия соответствуют большим множествам более подробных описаний состояния мира, а также более детализированным последовательностям действий. Теперь рассмотрим решение абстрактной задачи, например поиска пути от Арада до Бухареста через города Сибиу, Рымнику-Вылча и Питешти. Это абстрактное решение соответствует

<sup>4</sup> Последствия, связанные с применением отрицательных стоимостей, рассматриваются в упр. 3.17.

большому количеству более подробных инструкций по преодолению пути. Например, можно вести автомобиль между городами Сибю и Рымнику-Вылча с включенным радиоприемником, а затем выключить его на всю оставшуюся часть путешествия. Абстракция является действительной, если мы можем преобразовать любое абстрактное решение в такое решение, которое подходит для более детально описанного мира; достаточным условием является то, что для любого детализированного состояния, которое представляет собой “пребывание в городе Арад”, существует детализированный путь в некоторое состояние, соответствующее “пребыванию в городе Сибю”, и т.д. Абстракция является полезной, если осуществление каждого из действий, предусмотренных в решении, становится проще по сравнению с первоначальной задачей; в этом случае действия являются достаточно простыми для того, чтобы они могли быть выполнены без дальнейшего поиска или планирования со стороны обычного агента, занимающегося вождением автомобиля. Поэтому выбор хорошей абстракции сводится к исключению максимально возможного количества подробностей и вместе с тем к сохранению способности оставаться действительной и обеспечению того, чтобы абстрактные действия были легко осуществимыми. Если бы не было возможности создавать полезные абстракции, то интеллектуальные агенты не могли бы успешно действовать в реальном мире.

## 3.2. ПРИМЕРЫ ЗАДАЧ

Описанный подход к решению задач был применен в очень многих вариантах экспериментальной среды. В этом разделе перечислены некоторые из наиболее известных примеров решения задач, которые подразделяются на два типа — упрощенные и реальные задачи. ✎ **Упрощенная задача** предназначена для иллюстрации или проверки различных методов решения задач. Ей может быть дано краткое, точное описание. Это означает, что такая задача может легко использоваться разными исследователями для сравнения производительности алгоритмов. ✎ **Реальной задачей** называется такая задача, решение которой действительно требуется людям. Как правило, такие задачи не имеют единого приемлемого для всех описания, но мы попытаемся показать, как обычно выглядят их формулировки.

### Упрощенные задачи

В качестве первого примера рассмотрим мир пылесоса, впервые представленный в главе 2 (см. рис. 2.2). Деятельность в этом мире можно сформулировать в качестве задачи, как описано ниже.

- **Состояния.** Агент находится в одном из двух местонахождений, в каждом из которых может присутствовать или не присутствовать мусор. Поэтому существует  $2 \times 2^2 = 8$  возможных состояний мира.
- **Начальное состояние.** В качестве начального состояния может быть назначено любое состояние.
- **Функция определения приемника.** Эта функция вырабатывает допустимые состояния, которые являются следствием попыток выполнения трех действий (*Left*, *Right* и *Suck*). Полное пространство состояний показано на рис. 3.2.



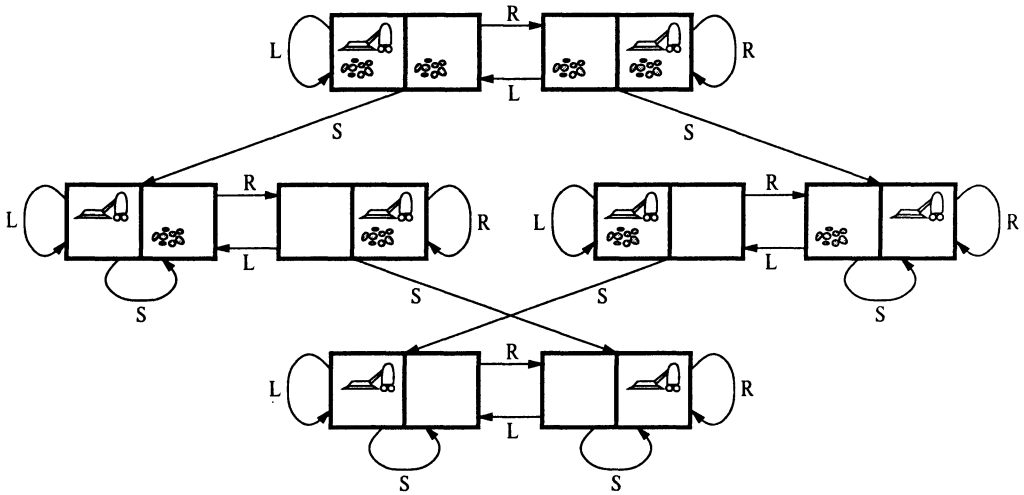


Рис. 3.2. Пространство состояний для мира пылесоса. Дуги обозначают действия: L=Left, R=Right, S=Suck

- **Проверка цели.** Эта проверка сводится к определению того, являются ли чистыми все квадраты.
- **Стоимость пути.** Стоимость каждого этапа равна 1, поэтому стоимость пути представляет собой количество этапов в этом пути.

По сравнению с задачей реального мира эта упрощенная задача характеризуется различными местонахождениями, возможностью определять наличие мусора, надежной очисткой, а также сохранением достигнутого состояния после очистки. (В разделе 3.6 некоторые из этих допущений будут исключены.) Необходимо учитывать, что состояние определяется и местонахождением агента, и наличием мусора. В более крупной среде с  $n$  местонахождениями имеется  $n \cdot 2^n$  состояний.

✎ **Задача игры в восемь,** экземпляр которой показан на рис. 3.3, состоит из доски  $3 \times 3$  с восемью пронумерованными фишками и с одним пустым участком. Фишка, смежная с пустым участком, может быть передвинута на этот участок. Требуется достичь указанного целевого состояния, подобного тому, которое показано в правой части рисунка. Стандартная формулировка этой задачи приведена ниже.

- **Состояния.** Описание состояния определяет местонахождение каждой из этих восьми фишек и пустого участка на одном из девяти квадратов.
- **Начальное состояние.** В качестве начального может быть определено любое состояние. Необходимо отметить, что любая заданная цель может быть достигнута точно из половины возможных начальных состояний (см. упр. 3.4).
- **Функция определения преемника.** Эта функция формирует допустимые состояния, которые являются результатом попыток осуществления указанных четырех действий (теоретически возможных ходов *Left*, *Right*, *Up* или *Down*).
- **Проверка цели.** Она позволяет определить, соответствует ли данное состояние целевой конфигурации, показанной на рис. 3.3. (Возможны также другие целевые конфигурации.)

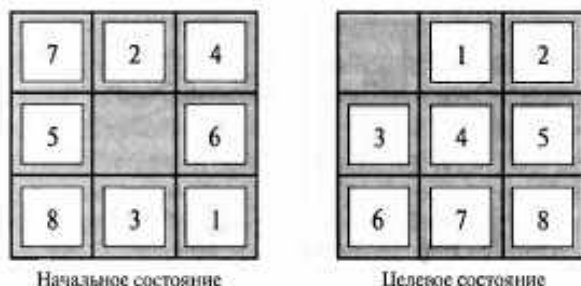


Рис. 3.3. Типичный экземпляр задачи игры в восемь

- **Стоимость пути.** Каждый этап имеет стоимость 1, поэтому стоимость пути равна количеству этапов в пути.

Какие абстракции здесь предусмотрены? Действия абстрагируются до уровня указания их начального и конечного состояний, при этом игнорируются промежуточные положения в процессе перемещения фишки. В ходе создания абстрактного описания исключены такие действия, как встряхивание доски, позволяющее передвинуть застрявшую фишку, или извлечение фишек с помощью ножа и повторное укладывание их в ящик. Исключено также описание правил игры, что позволяет обойтись без рассмотрения подробных сведений о физических манипуляциях.

Задача игры в восемь относится к семейству **задач со скользящими фишками**, которые часто используются в искусственном интеллекте для проверки новых алгоритмов поиска. Известно, что этот общий класс задач является NP-полным, поэтому вряд ли можно надеяться найти методы, которые в наихудшем случае были бы намного лучше по сравнению с алгоритмами поиска, описанными в этой и следующей главах. Задача игры в восемь имеет  $9!/2 = 181\,440$  достижимых состояний и легко решается. Задача игры в пятнадцать (на доске  $4 \times 4$ ) имеет около 1,3 триллиона состояний, и случайно выбранные ее экземпляры могут быть решены оптимальным образом за несколько миллисекунд с помощью наилучших алгоритмов поиска. Задача игры в двадцать четыре (на доске  $5 \times 5$ ) имеет количество состояний около  $10^{25}$ , и случайно выбранные ее экземпляры все еще весьма нелегко решить оптимальным образом с применением современных компьютеров и алгоритмов.

Цель **задачи с восемью ферзями** состоит в размещении восьми ферзей на шахматной доске таким образом, чтобы ни один ферзь не нападал на любого другого. (Ферзь атакует любую фигуру, находящуюся на одной и той же с ним горизонтали, вертикали или диагонали.) На рис. 3.4 показана неудачная попытка поиска решения: ферзь, находящийся на самой правой вертикали, атакован ферзем, который находится вверху слева.

Несмотря на то что существуют эффективные специализированные алгоритмы решения этой задачи и всего семейства задач с  $n$  ферзями, она по-прежнему остается интересной экспериментальной задачей для алгоритмов поиска. Для нее применяются формулировки двух основных типов. В **инкрементной формулировке** предусматривается использование операторов, которые дополняют описание состояния, начиная с пустого состояния; для задачи с восемью ферзями это означает, что каждое действие приводит к добавлению к этому состоянию еще одного ферзя.

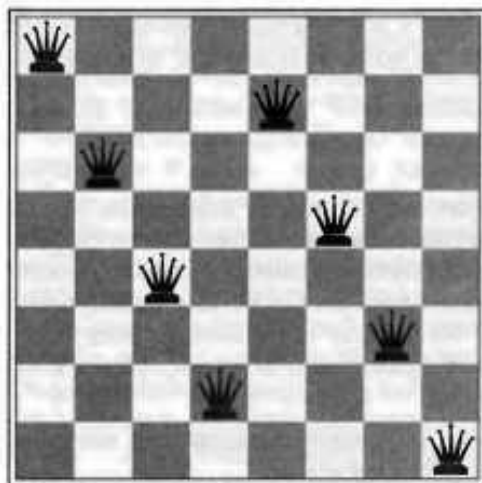


Рис. 3.4. Почти готовое решение задачи с восемью ферзями (поиск окончательного решения оставляем читателю в качестве упражнения)

✎ **Формулировка полного состояния** начинается с установки на доску всех восьми ферзей и предусматривает их дальнейшее перемещение. В том и другом случае стоимость пути не представляет интереса, поскольку важно лишь достигнуть конечного состояния. Первая инкрементная формулировка, которая может применяться при осуществлении попыток решения этой задачи, приведена ниже.

- **Состояния.** Состоянием является любое расположение ферзей на доске в количестве от 0 до 8.
- **Начальное состояние.** Отсутствие ферзей на доске.
- **Функция определения преемника.** Установка ферзя на любой пустой клетке.
- **Проверка цели.** На доске находится восемь ферзей, и ни один из них не атакован.

В этой формулировке требуется проверить  $64 \cdot 63 \cdot \dots \cdot 57 = 3 \times 10^{14}$  возможных последовательностей. В лучшей формулировке должно быть запрещено помещать ферзя на любую клетку, которая уже атакована, следующим образом.

- **Состояния.** Состояниями являются расположения с  $n$  ферзями ( $0 \leq n \leq 8$ ), по одному ферзю в каждой из находящихся слева  $n$  вертикалей, притом что ни один ферзь не нападает на другого.
- **Функция определения преемника.** Установка ферзя на любой клетке в находящейся слева пустой вертикали таким образом, чтобы он не был атакован каким-либо другим ферзем.

Эта формулировка позволяет сократить пространство состояний задачи с восемью ферзями с  $3 \times 10^{14}$  до 2057, и поиск решений значительно упрощается. С другой стороны, для 100 ферзей первоначальная формулировка определяет приблизительно  $10^{400}$  состояний, а улучшенная формулировка — около  $10^{52}$  состояний (см. упр. 3.5). Это — колоссальное сокращение, но улучшенное пространство состояний все еще слишком велико для того, чтобы с ним могли справиться алгоритмы, рассматривае-

мые в данной главе. В главе 4 описана формулировка полного состояния, а в главе 5 приведен простой алгоритм, который позволяет легко решить задачу даже с миллионным ферзей.

## Реальные задачи

Выше уже было показано, как можно определить задачу поиска маршрута в терминах заданных местонахождений и переходов по связям между ними. Алгоритмы поиска маршрута используются в самых разных приложениях, таких как системы маршрутизации в компьютерных сетях, системы планирования военных операций и авиапутешествий. Обычно процесс определения таких задач является трудоемким. Рассмотрим упрощенный пример задачи планирования авиапутешествий, который задан следующим образом.

- **Состояния.** Каждое состояние представлено местонахождением (например, аэропортом) и текущим временем.
- **Начальное состояние.** Оно задается в условии задачи.
- **Функция определения преемника.** Эта функция возвращает состояния, которые следуют из выполнения любого полета, указанного в расписании (возможно, с дополнительным указанием класса и места), отправления позже по сравнению с текущим временем с учетом продолжительности переезда внутри самого аэропорта, а также из одного аэропорта в другой.
- **Проверка цели.** Находимся ли мы в месте назначения к некоторому заранее заданному времени?
- **Стоимость пути.** Зависит от стоимости билета, времени ожидания, продолжительности полета, таможенных и иммиграционных процедур, комфортности места, времени суток, типа самолета, скидок для постоянных пассажиров и т.д.

В коммерческих консультационных системах планирования путешествий используется формулировка задачи такого типа со многими дополнительными осложнениями, которые требуются для учета чрезвычайно запутанных структур определения платы за проезд, применяемых в авиакомпаниях. Но любой опытный путешественник знает, что не все авиапутешествия проходят согласно плану. Действительно качественная система должна предусматривать планы действий в непредвиденных ситуациях (такие как страховочное резервирование билетов на альтернативные рейсы) в такой степени, которая соответствует стоимости и вероятности нарушения первоначального плана.

Задачи планирования обхода тесно связаны с задачами поиска маршрута, но с одним важным исключением. Рассмотрим, например, задачу: “Посетить каждый город, показанный на рис. 3.1, 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 первоначальным состоянием должно быть “В Бухаресте; посещен {Бухарест}”, а типичным промежуточным состоянием — “В Васлуй; посещены {Бухарест, Урзичени, Васлуй}”, тогда как проверка цели должна предусматривать определение того, находится ли агент в Бухаресте и посетил ли он все 20 городов.

Одной из задач планирования обхода является **задача коммивояжера** (Traveling Salesperson Problem — TSP), по условию которой каждый город должен быть посещен только один раз. Назначение ее состоит в том, чтобы найти самый короткий путь обхода. Как известно, эта задача является NP-трудной, но на улучшение возможностей алгоритмов TSP были затрачены колоссальные усилия. Кроме планирования поездок коммивояжеров, эти алгоритмы использовались для решения таких задач, как планирование перемещений автоматических сверл при отработке печатных плат и организация работы средств снабжения в производственных цехах.

Задача **компоновки СБИС** требует позиционирования миллионов компонентов и соединений на микросхеме для минимизации площади, схемных задержек, паразитных емкостей и максимизации выхода готовой продукции. Задача компоновки следует за этапом логического проектирования и обычно подразделяется на две части: **компоновка ячеек** и **маршрутизация каналов**. При компоновке ячеек простейшие компоненты схемы группируются по ячейкам, каждая из которых выполняет некоторую известную функцию. Каждая ячейка имеет постоянную форму (размеры и площадь) и требует создания определенного количества соединений с каждой из остальных ячеек. Требуется разместить ячейки на микросхеме таким образом, чтобы они не перекрывались и оставалось место для прокладки соединительных проводов между ячейками. При маршрутизации каналов происходит поиск конкретного маршрута для каждого проводника через зазоры между ячейками. Эти задачи поиска являются чрезвычайно сложными, но затраты на их решение, безусловно, оправдываются. В главе 4 приведены некоторые алгоритмы, позволяющие решать эти задачи.

**Задача управления навигацией робота** представляет собой обобщение описанной выше задачи поиска маршрута. В этой задаче вместо дискретного множества маршрутов рассматривается ситуация, в которой робот может перемещаться в непрерывном пространстве с бесконечным (в принципе) множеством возможных действий и состояний. Если требуется обеспечить циклическое перемещение робота по плоской поверхности, то пространство фактически может рассматриваться как двумерное, а если робот оборудован верхними и нижними конечностями или колесами, которыми также необходимо управлять, то пространство поиска становится многомерным. Даже для того чтобы сделать это пространство поиска конечным, требуются весьма развитые методы. Некоторые из этих методов рассматриваются в главе 25. Изначальная сложность задачи усугубляется тем, что при управлении реальными роботами необходимо учитывать ошибки в показаниях датчиков, а также отклонения в работе двигательных средств управления.

Решение задачи **автоматического упорядочения сборки** сложных объектов роботом было впервые продемонстрировано на примере робота Freddy [1044]. С тех пор прогресс в этой области происходил медленно, но уверенно, и в настоящее время достигнуто такое положение, что стала экономически выгодной сборка таких неординарных объектов, как электродвигатели. В задачах сборки цель состоит в определении последовательности, в которой должны быть собраны детали некоторого объекта. Если выбрана неправильная последовательность, то в дальнейшем нельзя будет найти способ добавления некоторой детали к этой последовательности без отмены определенной 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 дорогостоящих этапов решения задачи упорядочения сборки является

формирование допустимых преемников. Любой практически применимый алгоритм должен предотвращать необходимость поиска во всем пространстве состояний, за исключением крошечной его части. Еще одной важной задачей сборки является  $\sphericalangle$  **проектирование молекулы белка**, цель которой состоит в определении последовательности аминокислот, способных сложиться в трехмерный белок с нужными свойствами, предназначенный для лечения некоторых заболеваний.

В последние годы выросла потребность в создании программных роботов, которые осуществляют  $\sphericalangle$  **поиск в Internet**, находя ответы на вопросы, отыскивая требуемую информацию или совершая торговые сделки. Это — хорошее приложение для методов поиска, поскольку Internet легко представить концептуально в виде графа, состоящего из узлов (страниц), соединенных с помощью ссылок. Полное описание задачи поиска в Internet отложим до главы 10.

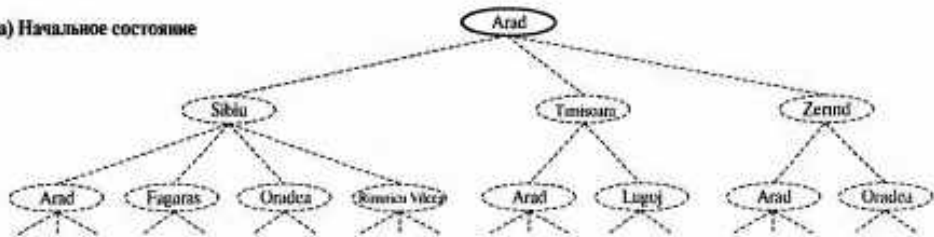
### 3.3. ПОИСК РЕШЕНИЙ

Сформулировав определенные задачи, необходимо найти их решение. Такая цель достигается посредством поиска в пространстве состояний. В настоящей главе рассматриваются методы поиска, в которых используются явно заданное  $\sphericalangle$  **дерево поиска**, создаваемое с помощью начального состояния и функции определения преемника, которые совместно задают пространство состояний. Вообще говоря, вместо дерева поиска может применяться граф поиска, если одно и то же состояние может быть достигнуто с помощью многих путей. Это важное дополнение рассматривается в разделе 3.5.

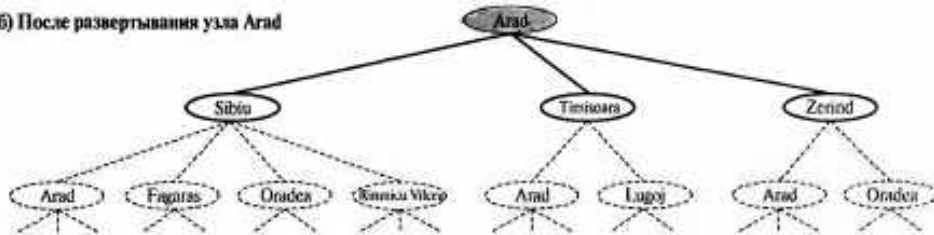
На рис. 3.5 показаны некоторые расширения дерева поиска, предназначенного для определения маршрута от Арада до Бухареста. Корнем этого дерева поиска является  $\sphericalangle$  **поисковый узел**, соответствующий начальному состоянию,  $In(Arad)$ . Первый этап состоит в проверке того, является ли это состояние целевым. Безусловно, что оно не является таковым, но необходимо предусмотреть соответствующую проверку, чтобы можно было решать задачи, содержащие в себе готовое решение, такие как “начав путешествие с города Арад, прибыть в город Арад”. А в данном случае текущее состояние не является целевым, поэтому необходимо рассмотреть некоторые другие состояния. Такой этап осуществляется путем  $\sphericalangle$  **развертывания** текущего состояния, т.е. применения функции определения преемника к текущему состоянию для  $\sphericalangle$  **формирования** в результате этого нового множества состояний. В данном случае будут получены три новых состояния:  $In(Sibiu)$ ,  $In(Timisoara)$  и  $In(Zerind)$ . Теперь необходимо определить, какой из этих трех вариантов следует рассматривать дальше.

В этом и состоит суть поиска — пока что проверить один вариант и отложить другие в сторону, на тот случай, что первый вариант не приведет к решению. Предположим, что вначале выбран город Сибиу. Проведем проверку для определения того, соответствует ли он целевому состоянию (не соответствует), а затем развернем узел *Sibiu* для получения состояний  $In(Arad)$ ,  $In(Fagaras)$ ,  $In(Oradea)$  и  $In(RimnicuVilcea)$ . После этого можно выбрать любое из этих четырех состояний либо вернуться и выбрать узел *Timisoara* или *Zerind*. Необходимо снова и снова выбирать, проверять и развертывать узлы до тех пор, пока не будет найдено решение или не останется больше состояний, которые можно было бы развернуть. Порядок, в котором происходит развертывание состояний, определяется  $\sphericalangle$  **стратегией поиска**. Общий алгоритм поиска в дереве неформально представлен в листинге 3.2.

а) Начальное состояние



б) После развертывания узла Arad



в) После развертывания узла Sibiu

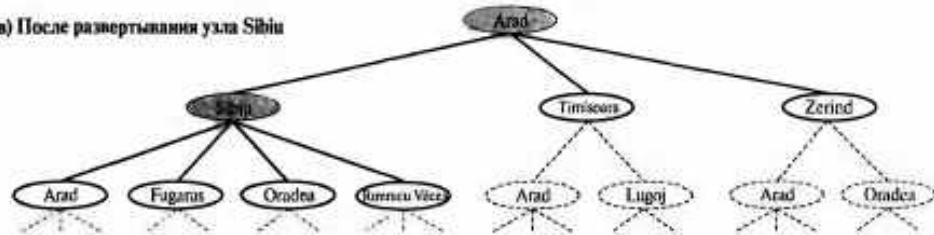


Рис. 3.5. Частично развернутые деревья поиска, предназначенные для определения маршрута от Арада до Бухареста. Развернутые узлы затенены; узлы, которые были сформированы, но еще не развернуты, выделены полужирным контуром; узлы, которые еще не были сформированы, обозначены тонкими штриховыми линиями

### Листинг 3.2. Неформальное описание общего алгоритма поиска в дереве

```

function Tree-Search(problem, strategy) returns решение solution
    или индикатор неудачи failure
    инициализировать дерево поиска с использованием начального
    состояния задачи problem
    loop do
        if нет кандидатов на развертывание then return индикатор
        неудачи failure
        выбрать листовой узел для развертывания в соответствии
        со стратегией strategy
        if этот узел содержит целевое состояние
            then return соответствующее решение solution
        else развернуть этот узел и добавить полученные узлы
        к дереву поиска
  
```

Необходимо учитывать различие между пространством состояний и деревом поиска. В пространстве состояний для задачи поиска маршрута имеется только 20 состояний, по одному для каждого города. Но количество путей в этом пространстве состояний является бесконечным, поэтому дерево поиска имеет бесконечное количество узлов. Напри-

мер, первыми тремя путями любой бесконечной последовательности путей являются маршруты Арад—Сибиу, Арад—Сибиу—Арад, Арад—Сибиу—Арад—Сибиу. (Безусловно, качественный алгоритм поиска должен исключать возможность формирования таких повторяющихся путей; в разделе 3.5 показано, как этого добиться.)

Существует множество способов представления узлов, но здесь предполагается, что узел представляет собой структуру данных с пятью компонентами, которые описаны ниже.

- **State.** Состояние в пространстве состояний, которому соответствует данный узел.
- **Parent-Node.** Узел в дереве поиска, применявшийся для формирования данного узла (родительский узел).
- **Action.** Действие, которое было применено к родительскому узлу для формирования данного узла.
- **Path-Cost.** Стоимость пути (от начального состояния до данного узла), показанного с помощью указателей родительских узлов, которую принято обозначать как  $g(n)$ .
- **Depth.** Количество этапов пути от начального состояния, называемое также *глубиной*.

Необходимо учитывать различие между узлами и состояниями. *Узел* — это учетная структура данных, применяемая для представления дерева поиска, а *состояние* соответствует конфигурации мира. Поэтому узлы лежат на конкретных путях, которые определены с помощью указателей Parent-Node, а состояния — нет. Кроме того, два разных узла могут включать одно и то же состояние мира, если это состояние формируется с помощью двух различных путей поиска. Структура данных узла показана на рис. 3.6.

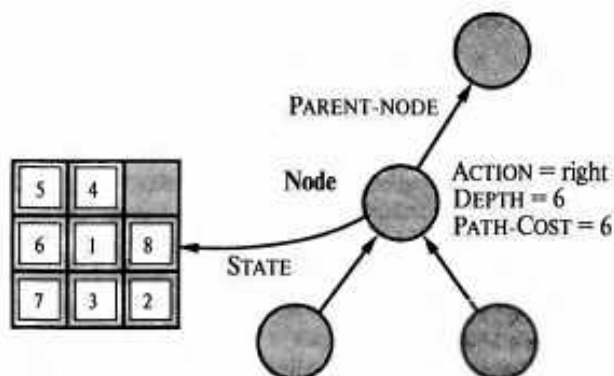


Рис. 3.6. Узлы представляют собой структуры данных, с помощью которых формируется дерево поиска. Каждый узел имеет родительский узел, содержит данные о состоянии и имеет различные вспомогательные поля. Стрелки направлены от дочернего узла к родительскому

Необходимо также представить коллекцию узлов, которые были сформированы, но еще не развернуты; такая коллекция называется **периферией**. Каждый элемент



периферии представляет собой  $\sphericalangle$  **листовой узел**, т.е. узел, не имеющий преемников в дереве. На рис. 3.5 периферия каждого дерева состоит из узлов с полужирными контурами. Простейшим представлением периферии может служить множество узлов. Тогда стратегия поиска должна быть выражена в виде функции, которая выбирает определенным образом из этого множества следующий узел, подлежащий разветвлению. Хотя данный подход концептуально является несложным, он может оказаться дорогостоящим с вычислительной точки зрения, поскольку функцию, предусмотренную в этой стратегии, возможно, придется применить к каждому элементу в указанном множестве для выбора наилучшего из них. Поэтому предполагается, что коллекция узлов реализована в виде  $\sphericalangle$  **очереди**. Операции, применимые к любой очереди, состоят в следующем.

- **Make-Queue(*element*, ...)**. Создает очередь с заданным элементом (элементами).
- **Empty?(*queue*)**. Возвращает истинное значение, только если в очереди больше нет элементов.
- **First(*queue*)**. Возвращает первый элемент в очереди.
- **Remove-First(*queue*)**. Возвращает элемент **First(*queue*)** и удаляет его из очереди.
- **Insert(*element*, *queue*)**. Вставляет элемент в очередь и возвращает результирующую очередь. (Ниже будет показано, что в очередях различных типов вставка элементов осуществляется в различном порядке.)
- **Insert-All(*elements*, *queue*)**. Вставляет множество элементов в очередь и возвращает результирующую очередь.

С помощью этих определений мы можем записать более формальную версию общего алгоритма поиска в дереве, показанную в листинге 3.3.

**Листинг 3.3.** Общий алгоритм поиска в дереве. Следует учитывать, что фактический параметр **fringe** (периферия) должен представлять собой пустую очередь, а порядок поиска зависит от типа очереди. Функция **Solution** возвращает последовательность действий, полученную путем прохождения по указателям на родительские узлы в обратном направлении, к корню

---

```
function Tree-Search(problem, fringe) returns решение solution
или индикатор неудачи failure
```

```
  fringe  $\leftarrow$  Insert(Make-Node(Initial-State[problem]), fringe)
  loop do
    if Empty?(fringe) then return индикатор неудачи failure
    node  $\leftarrow$  Remove-First(fringe)
    if Goal-Test[problem] применительно к State[node]
      завершается успешно
      then return Solution(node)
    fringe  $\leftarrow$  Insert-All(Expand(node, problem), fringe)
```

```
function Expand(node, problem) returns множество узлов successors
```

```
  successors  $\leftarrow$  пустое множество
  for each <action, result> in Successor-Fn[problem](State[node]) do
    s  $\leftarrow$  новый узел
```

```

State[s] ← result
Parent-Node[s] ← node
Action[s] ← action
Path-Cost[s] ← Path-Cost[node] + Step-Cost(node, action, s)
Depth[s] ← Depth[node] + 1
добавить узел s к множеству successors
return successors

```

---

## Измерение производительности решения задачи

Результатом применения любого алгоритма решения задачи является либо неудачное завершение, либо получение решения. (Некоторые алгоритмы могут входить в бесконечный цикл и не возвращать никакого результата.) Мы будем оценивать производительность алгоритма с помощью четырех показателей, описанных ниже.

- **Полнота.** Гарантирует ли алгоритм обнаружение решения, если оно имеется?
- **Оптимальность.** Обеспечивает ли данная стратегия нахождение оптимального решения, в соответствии с определением, приведенным на с. 115?
- **Временная сложность.** За какое время алгоритм находит решение?
- **Пространственная сложность.** Какой объем памяти необходим для осуществления поиска?

Временная и пространственная сложность всегда анализируются с учетом определенного критерия измерения сложности задачи. В теоретической компьютерной науке типичным критерием является размер графа пространства состояний, поскольку этот граф рассматривается как явно заданная структура данных, которая является входной для программы поиска. (Примером этого может служить карта Румынии.) В искусственном интеллекте, где граф представлен неявно с помощью начального состояния и функции определения преемника и часто является бесконечным, сложность выражается в терминах трех величин:  $b$  — коэффициент ветвления или максимальное количество преемников любого узла,  $d$  — глубина самого поверхностного целевого узла и  $m$  — максимальная длина любого пути в пространстве состояний.

Временная сложность часто измеряется в терминах количества узлов, вырабатываемых<sup>5</sup> в процессе поиска, а пространственная сложность — в терминах максимального количества узлов, хранимых в памяти.

Чтобы оценить эффективность любого алгоритма поиска, можно рассматривать только **стоимость поиска**, которая обычно зависит от временной сложности, но может также включать выражение для оценки использования памяти, или применять **суммарную стоимость**, в которой объединяется стоимость поиска и стоимость пути найденного решения. Для задачи поиска маршрута от Арада до Бухареста стоимость поиска представляет собой количество времени, затраченного на этот поиск, а стоимость решения выражает общую длину пути в километрах. Поэтому для вы-

---

<sup>5</sup> В некоторых работах вместо этого для измерения временной сложности применяется количество операций развертывания узлов. Эти два критерия различаются не больше, чем на коэффициент  $b$ . С точки зрения авторов, время выполнения операции развертывания узла растет пропорционально количеству узлов, вырабатываемых при этом развертывании.

числения суммарной стоимости нам придется складывать километры и миллисекунды. Между этими двумя единицами измерения не определен “официальный курс обмена”, но в данном случае было бы резонно преобразовывать километры в миллисекунды с использованием оценки средней скорости автомобиля (поскольку для данного агента важным является именно время). Это позволяет рассматриваемому агенту найти оптимальную точку компромисса, в которой дальнейшие вычисления для поиска более короткого пути становятся непродуктивными. Описание более общей задачи поиска компромисса между различными ценностями будет продолжено в главе 16.

### 3.4. СТРАТЕГИИ НЕИНФОРМИРОВАННОГО ПОИСКА

В данном разделе рассматриваются пять стратегий поиска, которые известны под названием **неинформированного поиска** (называемого также **слепым поиском**). Этот термин означает, что в данных стратегиях не используется дополнительная информация о состояниях, кроме той, которая представлена в определении задачи. Все, на что они способны, — выбатывать преемников и отличать целевое состояние от нецелевого. Стратегии, позволяющие определить, является ли одно нецелевое состояние “более многообещающим” по сравнению с другим, называются стратегиями **информированного поиска**, или **эвристического поиска**; они рассматриваются в главе 4. Все стратегии поиска различаются тем, в каком порядке происходит развертывание узлов.

#### Поиск в ширину

**Поиск в ширину** — это простая стратегия, в которой вначале развертывается корневой узел, затем — все преемники корневого узла, после этого развертываются преемники этих преемников и т.д. Вообще говоря, при поиске в ширину, прежде чем происходит развертывание каких-либо узлов на следующем уровне, развертываются все узлы на данной конкретной глубине в дереве поиска.

Поиск в ширину может быть реализован путем вызова процедуры `Tree-Search` с пустой периферией, которая представляет собой последовательную очередь (`First-In-First-Out` — `FIFO`), гарантирующую, что прежде всего будут развернуты узлы, которые должны посещаться первыми. Иными словами, к организации поиска в глубину приводит вызов процедуры `Tree-Search(problem, FIFO-Queue())`. В очереди `FIFO` предусмотрена вставка всех вновь сформированных преемников в конец очереди, а это означает, что поверхностные узлы развертываются прежде, чем более глубокие. На рис. 3.7 показан ход поиска в простом бинарном дереве.

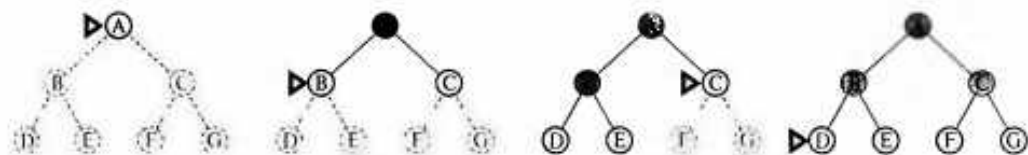


Рис. 3.7. Поиск в ширину в простом бинарном дереве. На каждом этапе узел, подлежащий развертыванию в следующую очередь, обозначается маркером

Проведем оценку поиска в ширину с использованием четырех критериев, описанных в предыдущем разделе. Вполне очевидно, что этот поиск является полным — если самый поверхностный целевой узел находится на некоторой конечной глубине  $d$ , то поиск в ширину в конечном итоге позволяет его обнаружить после развертывания всех более поверхностных узлов (при условии, что коэффициент ветвления  $b$  является конечным). Самый поверхностный целевой узел не обязательно является оптимальным; формально поиск в ширину будет оптимальным, если стоимость пути выражается в виде неубывающей функции глубины узла. (Например, такое предположение оправдывается, если все действия имеют одинаковую стоимость.)

До сих пор приведенное выше описание поиска в ширину не предвещало никаких неприятностей. Но такая стратегия не всегда является оптимальной; чтобы понять, с чем это связано, необходимо определить, какое количество времени и какой объем памяти требуются для выполнения поиска. Для этого рассмотрим гипотетическое пространство состояний, в котором каждое состояние имеет  $b$  преемников. Корень этого дерева поиска вырабатывает  $b$  узлов на первом уровне, каждый из которых вырабатывает еще  $b$  узлов, что соответствует общему количеству узлов на втором уровне, равному  $b^2$ . Каждый из них также вырабатывает  $b$  узлов, что приводит к получению  $b^3$  узлов на третьем уровне, и т.д. А теперь предположим, что решение находится на глубине  $d$ . В наихудшем случае на уровне  $d$  необходимо развернуть все узлы, кроме последнего (поскольку сам целевой узел не развертывается), что приводит к выработке  $b^{d+1} - b$  узлов на уровне  $d+1$ . Это означает, что общее количество выработанных узлов равно:

$$b + b^2 + b^3 + \dots + b^d + (b^{d+1} - b) = O(b^{d+1})$$

Каждый выработанный узел должен оставаться в памяти, поскольку он либо относится к периферии, либо является предком периферийного узла. Итак, пространственная сложность становится такой же, как и временная (с учетом добавления одного узла, соответствующего корню).

Поэтому исследователи, выполняющие анализ сложности алгоритма, огорчаются (или восхищаются, если им нравится преодолевать трудности), столкнувшись с экспоненциальными оценками сложности, такими как  $O(b^{d+1})$ . В табл. 3.1 показано, с чем это связано. В ней приведены требования ко времени и к объему памяти при поиске в ширину с коэффициентом ветвления  $b=10$  для различных значений глубины решения  $d$ . При составлении этой таблицы предполагалось, что в секунду может быть сформировано 10 000 узлов, а для каждого узла требуется 1000 байтов памяти. Этим предположением приблизительно соответствуют многие задачи поиска при их решении на любом современном персональном компьютере (с учетом повышающего или понижающего коэффициента 100).

На основании табл. 3.1 можно сделать два важных вывода. Прежде всего, *при поиске в ширину наиболее сложной проблемой по сравнению со значительным временем выполнения является обеспечение потребностей в памяти*. Затраты времени, равные 31 часу, не кажутся слишком значительными при ожидании решения важной задачи с глубиной 8, но лишь немногие компьютеры имеют терабайт оперативной памяти, который для этого требуется. К счастью, существуют другие стратегии поиска, которые требуют меньше памяти.

Второй вывод состоит в том, что требования ко времени все еще остаются важным фактором. Если рассматриваемая задача имеет решение на глубине 12, то (с учетом при-

нятых предположений) потребуется 35 лет на поиск в ширину (а в действительности на любой неинформированный поиск), чтобы найти ее решение. Вообще говоря, *«Задачи поиска с экспоненциальной сложностью невозможно решить с помощью неинформированных методов во всех экземплярах этих задач, кроме самых небольших»*.

Таблица 3.1. Потребности во времени и объеме памяти для поиска в ширину. Приведенные здесь данные получены при следующих предположениях: коэффициент ветвления —  $b=10$ ; скорость формирования — 10 000 узлов/секунда; объем памяти — 1000 байтов/узел

Глубина	Количество узлов	Время	Память
2	1100	0,11 секунды	1 мегабайт
4	111 100	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

Поиск по критерию стоимости

Поиск в ширину является оптимальным, если стоимости всех этапов равны, поскольку в нем всегда разворачивается самый поверхностный неразвернутый узел. С помощью простого дополнения можно создать алгоритм, который является оптимальным при любой функции определения стоимости этапа. Вместо разворачивания самого поверхностного узла *«Поиск по критерию стоимости»* обеспечивает разворачивание узла  $n$  с наименьшей стоимостью пути. Обратите внимание на то, что, если стоимости всех этапов равны, такой поиск идентичен поиску в ширину.

При поиске по критерию стоимости учитывается не количество этапов, имеющих в пути, а только их суммарная стоимость. Поэтому процедура этого поиска может войти в бесконечный цикл, если окажется, что в ней развернут узел, имеющий действие с нулевой стоимостью, которое снова указывает на то же состояние (например, действие *NoOp*). Можно гарантировать полноту поиска при условии, что стоимость каждого этапа больше или равна некоторой небольшой положительной константе  $\epsilon$ . Это условие является также достаточным для обеспечения оптимальности. Оно означает, что стоимость пути всегда возрастает по мере прохождения по этому пути. Из данного свойства легко определить, что такой алгоритм разворачивает узлы в порядке возрастания стоимости пути. Поэтому первый целевой узел, выбранный для разворачивания, представляет собой оптимальное решение. (Напомним, что в процедуре *Tree-Search* проверка цели применяются только к тем узлам, которые выбраны для разворачивания.) Рекомендуем читателю попытаться воспользоваться этим алгоритмом для поиска кратчайшего пути до Бухареста.

Поиск по критерию стоимости направляется с учетом стоимостей путей, а не значений глубины в дереве поиска, поэтому его сложность не может быть легко охарактеризована в терминах  $b$  и  $d$ . Вместо этого предположим, что  $C^*$  — стоимость оптимального решения, и допустим, что стоимость каждого действия составляет, по меньшей мере,  $\epsilon$ . Это означает, что временная и пространственная сложность этого алгоритма в наихудшем случае составляет  $O(b^{1+\lceil C^*/\epsilon \rceil})$ , т.е. может быть намного больше, чем  $b^d$ . Это связано с тем, что процедуры поиска по критерию стоимости

могут и часто выполняют проверку больших деревьев, состоящих из мелких этапов, прежде чем перейти к исследованию путей, в которые входят крупные, но, возможно, более полезные этапы. Безусловно, если все стоимости этапов равны, то выражение  $b^{1+\lfloor c^*/e \rfloor}$  равняется  $b^d$ .

## Поиск в глубину

При **поиске в глубину** всегда разворачивается самый глубокий узел в текущей периферии дерева поиска. Ход такого поиска показан на рис. 3.8. Поиск непосредственно переходит на самый глубокий уровень дерева поиска, на котором узлы не имеют преемников. По мере того как эти узлы разворачиваются, они удаляются из периферии, поэтому в дальнейшем поиск “возобновляется” со следующего самого поверхностного узла, который все еще имеет неисследованных преемников.

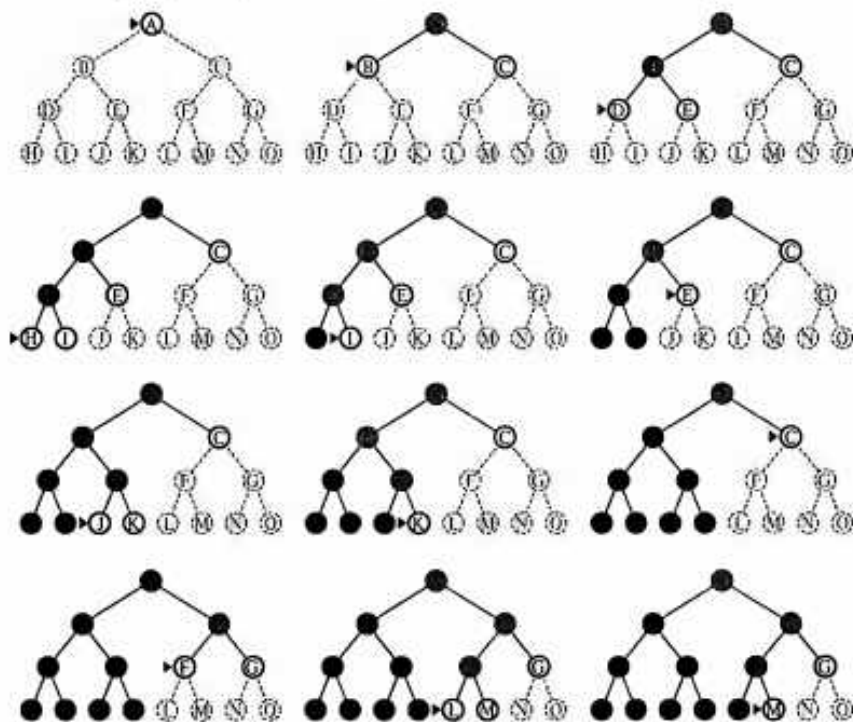


Рис. 3.8. Поиск в глубину в бинарном дереве. Узлы, которые были развернуты и не имеют потомков в этой периферии, могут быть удалены из памяти; эти узлы обозначены черным цветом. Предполагается, что узлы на глубине 3 не имеют преемников и единственным целевым узлом является M.

Эта стратегия может быть реализована в процедуре **Tree-Search** с помощью очереди **LIFO** (Last-In-First-Out), называемой также *стеком*. В качестве альтернативы способу реализации на основе процедуры **Tree-Search** поиск в глубину часто реализуют с помощью рекурсивной функции, вызывающей саму себя в каждом из дочерних узлов по очереди. (Рекурсивный алгоритм поиска в глубину, в котором предусмотрен предел глубины, приведен в листинге 3.4.)

Поиск в глубину имеет очень скромные потребности в памяти. Он требует хранения только единственного пути от корня до листового узла, наряду с оставшимися неразвернутыми сестринскими узлами для каждого узла пути. После того как был развернут некоторый узел, он может быть удален из памяти, коль скоро будут полностью исследованы все его потомки (см. рис. 3.8). Для пространства состояний с коэффициентом ветвления  $b$  и максимальной глубиной  $m$  поиск в глубину требует хранения только  $bm+1$  узлов. Используя такие же предположения, как и в табл. 3.1, и допуская, что узлы, находящиеся на той же глубине, что и целевой узел, не имеют преемников, авторы определили, что на глубине  $d=12$  для поиска в глубину требуется 118 килобайтов вместо 10 петабайтов, т.е. потребность в пространстве уменьшается примерно в 10 миллиардов раз.

В одном из вариантов поиска в глубину, называемом **поиском с возвратами**, используется еще меньше памяти. При поиске с возвратами каждый раз формируется только один преемник, а не все преемники; в каждом частично развернутом узле запоминается информация о том, какой преемник должен быть сформирован следующим. Таким образом, требуется только  $O(m)$  памяти, а не  $O(bm)$ . При поиске с возвратами применяется еще один прием, позволяющий экономить память (и время); идея его состоит в том, чтобы при формировании преемника должно непосредственно модифицироваться описание текущего состояния, а не осуществляться его предварительное копирование. При этом потребность в памяти сокращается до объема, необходимого для хранения только одного описания состояния и  $O(m)$  действий. Но для успешного применения данного приема нужно иметь возможность отменять каждую модификацию при возврате, выполняемом для формирования следующего преемника. При решении задач с объемными описаниями состояния, таких как роботизированная сборка, применение указанных методов модификации состояний становится важнейшим фактором успеха.

Недостатком поиска в глубину является то, что в нем может быть сделан неправильный выбор и переход в тупиковую ситуацию, связанную с прохождением вниз по очень длинному (или даже бесконечному) пути, притом что другой вариант мог бы привести к решению, находящемуся недалеко от корня дерева поиска. Например, на рис. 3.8 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  $J$ , менее приемлемый по сравнению с узлом  $C$ , то поиск в глубину возвратил бы в качестве решения именно его; это означает, что поиск в глубину не является оптимальным. Кроме того, если бы левое поддерево имело неограниченную глубину, но не содержало бы решений, то поиск в глубину так никогда бы и не закончился; это означает, что данный алгоритм — не полный. В наилучшем случае поиск в глубину формирует все  $O(b^m)$  узлов в дереве поиска, где  $m$  — максимальная глубина любого узла. Следует отметить, что  $m$  может оказаться гораздо больше по сравнению с  $d$  (глубиной самого поверхностного решения) и является бесконечным, если дерево имеет неограниченную глубину.

### Поиск с ограничением глубины

Проблему неограниченных деревьев можно решить, предусматривая применение во время поиска в глубину заранее определенного предела глубины  $\ell$ . Это означает, что узлы на глубине  $\ell$  рассматриваются таким образом, как если бы они не имели

преемников. Такой подход называется **поиском с ограничением глубины**. Применение предела глубины позволяет решить проблему бесконечного пути. К сожалению, в этом подходе также вводится дополнительный источник неполноты, если будет выбрано значение  $\ell < d$ , иными словами, если самая поверхностная цель выходит за пределы глубины. (Такая ситуация вполне вероятна, если значение  $d$  неизвестно.) Кроме того, поиск с ограничением глубины будет неоптимальным при выборе значения  $\ell > d$ . Его временная сложность равна  $O(b^\ell)$ , а пространственная сложность —  $O(b\ell)$ . Поиск в глубину может рассматриваться как частный случай поиска с ограничением глубины, при котором  $\ell = \infty$ .

Иногда выбор пределов глубины может быть основан на лучшем понимании задачи. Например, допустим, что на рассматриваемой карте Румынии имеется 20 городов. Поэтому известно, что если решение существует, то должно иметь длину не больше 19; это означает, что одним из возможных вариантов является  $\ell = 19$ . Но в действительности при внимательном изучении этой карты можно обнаружить, что любой город может быть достигнут из любого другого города не больше чем за 9 этапов. Это число, известное как **диаметр** пространства состояний, предоставляет нам лучший предел глубины, который ведет к более эффективному поиску с ограничением глубины. Но в большинстве задач приемлемый предел глубины остается неизвестным до тех пор, пока не будет решена сама задача.

Поиск с ограничением глубины может быть реализован как простая модификация общего алгоритма поиска в дереве или рекурсивного алгоритма поиска в глубину. Псевдокод реализации рекурсивного поиска с ограничением глубины приведен в листинге 3.4. Обратите внимание на то, что поиск с ограничением глубины может приводить к неудачным завершениям двух типов: стандартное значение *failure* указывает на отсутствие решения, а значение *cutoff* свидетельствует о том, что на заданном пределе глубины решения нет.

#### Листинг 3.4. Рекурсивная реализация поиска с ограничением глубины

---

```

function Depth-Limited-Search(problem, limit) returns решение result
    или индикатор неудачи failure/cutoff
    return Recursive-DLS(Make-Node(Initial-State[problem]),
                          problem, limit)

function Recursive-DLS(node, problem, limit) returns решение result
    или индикатор неудачи failure/cutoff
    cutoff_occurred?  $\leftarrow$  ложное значение
    if Goal-Test[problem](State[node]) then return Solution(node)
    else if Depth[node] = limit then return индикатор неудачи cutoff
    else for each преемник successor in Expand(node, problem) do
        result  $\leftarrow$  Recursive-DLS(successor, problem, limit)
        if result = cutoff then cutoff_occurred?  $\leftarrow$  истинное значение
        else if result  $\neq$  failure then return решение result
    if cutoff_occurred?
        then return индикатор неудачи cutoff
    else return индикатор неудачи failure

```

---



## Поиск в глубину с итеративным углублением

✎ **Поиск с итеративным углублением** (или, точнее, поиск в глубину с итеративным углублением) представляет собой общую стратегию, часто применяемую в сочетании с поиском в глубину, которая позволяет найти наилучший предел глубины. Это достигается путем постепенного увеличения предела (который вначале становится равным 0, затем 1, затем 2 и т.д.) до тех пор, пока не будет найдена цель. Такое событие происходит после того, как предел глубины достигает значения  $d$ , глубины самого поверхностного целевого узла. Данный алгоритм приведен в листинге 3.5. В поиске с итеративным углублением сочетаются преимущества поиска в глубину и поиска в ширину. Как и поиск в глубину, он характеризуется очень скромными требованиями к памяти, а именно, значением  $O(bd)$ . Как и поиск в ширину, он является полным, если коэффициент ветвления конечен, и оптимальным, если стоимость пути представляет собой неубывающую функцию глубины узла. На рис. 3.9 показаны четыре итерации применения процедуры *Iterative-Deepening-Search* к бинарному дереву поиска, где решение найдено в четвертой итерации.

**Листинг 3.5.** Алгоритм поиска с итеративным углублением, в котором повторно применяется поиск с ограничением глубины при последовательном увеличении пределов. Он завершает свою работу после того, как обнаруживается решение, или процедура поиска с ограничением глубины возвращает значение *failure*, а это означает, что решение не существует

---

```

function Iterative-Deepening-Search(problem) returns решение result
    или индикатор неудачи failure
    inputs: problem, задача

    for depth  $\leftarrow$  0 to  $\infty$  do
        result  $\leftarrow$  Depth-Limited-Search(problem, depth)
        if result  $\neq$  cutoff then return решение result

```

---

Поиск с итеративным углублением может на первый взгляд показаться расточительным, поскольку одни и те же состояния формируются несколько раз. Но, как оказалось, такие повторные операции не являются слишком дорогостоящими. Причина этого состоит в том, что в дереве поиска с одним и тем же (или почти одним и тем же) коэффициентом ветвления на каждом уровне большинство узлов находится на нижнем уровне, поэтому не имеет большого значения то, что узлы на верхних уровнях формируются многократно. В поиске с итеративным углублением узлы на нижнем уровне (с глубиной  $d$ ) формируются один раз, те узлы, которые находятся на уровне, предшествующем нижнему, формируются дважды, и т.д., вплоть до дочерних узлов корневого узла, которые формируются  $d$  раз. Поэтому общее количество формируемых узлов выражается следующей формулой:

$$N(\text{IDS}) = (d)b + (d-1)b^2 + \dots + (1)b^d$$

которая соответствует временной сложности порядка  $O(b^d)$ . Это количество можно сравнить с количеством узлов, формируемых при поиске в ширину:

$$N(\text{BFS}) = b + b^2 + \dots + b^d + (b^{d+1} - b)$$

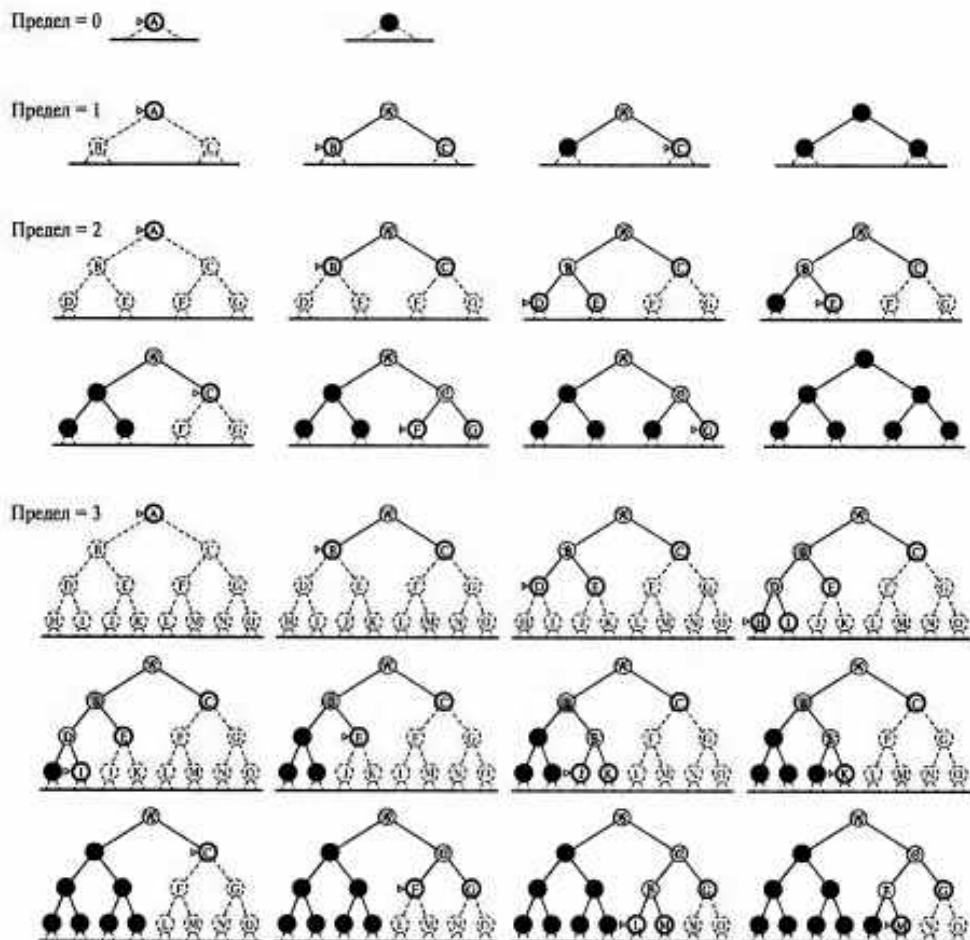


Рис. 3.9. Четыре итерации поиска с итеративным углублением в бинарном дереве

Следует отметить, что при поиске в ширину некоторые узлы формируются на глубине  $d+1$ , а при итеративном углублении этого не происходит. Результатом является то, что поиск с итеративным углублением фактически осуществляется быстрее, чем поиск в ширину, несмотря на повторное формирование состояний. Например, если  $b=10$  и  $d=5$ , то соответствующие оценки количества узлов принимают следующие значения:

$$N(\text{IDS}) = 50 + 400 + 3000 + 20\,000 + 100\,000 = 123\,450$$

$$N(\text{BFS}) = 10 + 100 + 1000 + 10\,000 + 100\,000 + 999\,990 = 1\,111\,100$$

Вообще говоря, итеративное углубление — это предпочтительный метод неинформированного поиска при тех условиях, когда имеется большое пространство поиска, а глубина решения неизвестна.

Поиск с итеративным углублением аналогичен поиску в ширину в том отношении, что в нем при каждой итерации перед переходом на следующий уровень исследуется полный уровень новых узлов. На первый взгляд может показаться целесообразным разработка итеративного аналога поиска по критерию стоимости, который

унаследовал бы от последнего алгоритма гарантии оптимальности, позволяя вместе с тем исключить его высокие требования к памяти. Идея состоит в том, чтобы вместо увеличивающихся пределов глубины использовались увеличивающиеся пределы стоимости пути. Результирующий алгоритм, получивший название **поиска с итеративным удлинением**, рассматривается в упр. 3.11. Но, к сожалению, было установлено, что поиск с итеративным удлинением характеризуется более существенными издержками, чем поиск по критерию стоимости.

### Двунаправленный поиск

В основе двунаправленного поиска лежит такая идея, что можно одновременно проводить два поиска (в прямом направлении, от начального состояния, и в обратном направлении, от цели), останавливаясь после того, как два процесса поиска встретятся на середине (рис. 3.10). Дело в том, что значение  $b^{d/2} + b^{d/2}$  гораздо меньше, чем  $b^d$ , или, как показано на этом рисунке, площадь двух небольших кругов меньше площади одного большого круга с центром в начале поиска, который охватывает цель поиска.

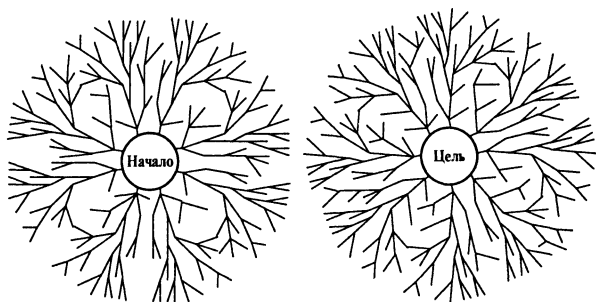


Рис. 3.10. Схематическое представление двунаправленного поиска в том состоянии, когда он должен вскоре завершиться успешно после того, когда одна из ветвей, исходящих из начального узла, встретится с ветвью из целевого узла

Двунаправленный поиск реализуется с помощью метода, в котором предусматривается проверка в одном или в обоих процессах поиска каждого узла перед его развертыванием для определения того, не находится ли он на периферии другого дерева поиска; в случае положительного результата проверки решение найдено. Например, если задача имеет решение на глубине  $d=6$  и в каждом направлении осуществляется поиск в ширину с последовательным развертыванием по одному узлу, то в самом худшем случае эти два процесса поиска встретятся, если в каждом из них будут развернуты все узлы на глубине 3, кроме одного. Это означает, что при  $b=10$  будет сформировано общее количество узлов, равное 22 200, а не 11 111 100, как при стандартном поиске в ширину. Проверка принадлежности узла к другому дереву поиска может быть выполнена за постоянное время с помощью хэш-таблицы, поэтому временная сложность двунаправленного поиска определяется как  $O(b^{d/2})$ . В памяти необходимо хранить по крайней мере одно из деревьев поиска, для того, чтобы можно было выполнить проверку принадлежности к другому дереву, поэтому пространственная сложность также определяется как  $O(b^{d/2})$ . Такие требования к пространству являются одним из наиболее существенных недостатков двунаправленного поиска. Этот алгоритм является полным и оптимальным (при единообразных стоимостях эта-

пов), если оба процесса поиска осуществляются в ширину; другие сочетания методов могут характеризоваться отсутствием полноты, оптимальности или того и другого.

Благодаря такому уменьшению временной сложности двунаправленный поиск становится привлекательным, но как организовать поиск в обратном направлении? Это не так легко, как кажется на первый взгляд. Допустим, что  $\Delta$  предшественниками узла  $n$ , определяемыми с помощью функции  $Pred(n)$ , являются все те узлы, для которых  $n$  служит преемником. Для двунаправленного поиска требуется, чтобы функция определения предшественника  $Pred(n)$  была эффективно вычислимой. Простейшим является такой случай, когда все действия в пространстве состояний обратимы таким образом, что  $Pred(n) = Succ^{-1}(n)$ , а другие случаи могут потребовать проявить значительную изобретательность.

Рассмотрим вопрос о том, что подразумевается под понятием “цель” при поиске “в обратном направлении от цели”. В задачах игры в восемь и поиска маршрута в Румынии имеется только одно целевое состояние, поэтому обратный поиск весьма напоминает прямой поиск. Если же имеется несколько явно перечисленных целевых состояний (например, два показанных на рис. 3.2 целевых состояния, в которых квадраты пола не содержат мусор), то может быть создано новое фиктивное целевое состояние, непосредственными предшественниками которого являются все фактические целевые состояния. Иным образом, формирования некоторых избыточных узлов можно избежать, рассматривая множество целевых состояний как единственное целевое состояние, каждым из предшественников которого также является множество состояний, а именно, множество состояний, имеющее соответствующего преемника в множестве целевых состояний (см. также раздел 3.6).

Наиболее сложным случаем для двунаправленного поиска является такая задача, в которой для проверки цели дано только неявное описание некоторого (возможно, большого) множества целевых состояний, например всех состояний, соответствующих проверке цели “мат” в шахматах. При обратном поиске потребовалось бы создать компактные описания “всех состояний, которые позволяют поставить мат с помощью хода  $t_1$ ”, и т.д.; и эти описания нужно было бы сверять с состояниями, формируемыми при прямом поиске. Общего способа эффективного решения такой проблемы не существует.

### Сравнение стратегий неинформированного поиска

В табл. 3.2 приведено сравнение стратегий поиска в терминах четырех критериев оценки, сформулированных в разделе 3.4.

## 3.5. ПРЕДОТВРАЩЕНИЕ ФОРМИРОВАНИЯ ПОВТОРЯЮЩИХСЯ СОСТОЯНИЙ

Вплоть до этого момента мы рассматривали все аспекты поиска, но игнорировали одно из наиболее важных усложнений в процессе поиска — вероятность появления непроизводительных затрат времени при развертывании состояний, которые уже встречались и были развернуты перед этим. При решении некоторых задач такая ситуация никогда не возникает; в них пространство состояний представляет собой дерево и поэтому имеется только один путь к каждому состоянию. В частности, эф-

фективной является формулировка задачи с восемью ферзями (в которой каждый новый ферзь помещается на самый левый пустой вертикальный ряд), и ее эффективность в значительной степени обусловлена именно этим — каждое состояние может быть достигнуто только по одному пути. А если бы задача с восемью ферзями была сформулирована таким образом, что любого ферзя разрешалось бы ставить на любую вертикаль, то каждого состояния с  $n$  ферзями можно было бы достичь с помощью  $n!$  различных путей.

Таблица 3.2. Оценка стратегий поиска, где  $b$  — коэффициент ветвления;  $d$  — глубина самого поверхностного решения;  $m$  — максимальная глубина дерева поиска;  $\ell$  — предел глубины. Предостережения, обозначенные строчными буквами, означают следующее: <sup>a</sup> — полный, если коэффициент ветвления  $b$  конечен; <sup>o</sup> — полный, если стоимость каждого этапа  $\geq \epsilon$  при некотором положительном значении  $\epsilon$ ; <sup>n</sup> — оптимальный, если стоимости всех этапов являются одинаковыми; <sup>r</sup> — применимый, если в обоих направлениях осуществляется поиск в ширину

Характеристика	Поиск в ширину	Поиск по критерию стоимости	Поиск в глубину	Поиск с ограничением глубины	Поиск с итеративным углублением	Двунаправленный поиск (если он применим)
Полнота	Да <sup>a</sup>	Да <sup>a, o</sup>	Нет	Нет	Да <sup>a</sup>	Да <sup>a, r</sup>
Временная сложность	$O(b^{d+1})$	$O(b^{1+\lceil c/\epsilon \rceil})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Пространственная сложность	$O(b^{d+1})$	$O(b^{1+\lceil c/\epsilon \rceil})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Оптимальность	Да <sup>n</sup>	Да	Нет	Нет	Да <sup>n</sup>	Да <sup>a, r</sup>

В некоторых задачах повторяющиеся состояния являются неизбежными. К ним относятся все задачи, в которых действия являются обратимыми, такие как задачи поиска маршрута и игры со скользящими фишками. Деревья поиска для этих проблем бесконечны, но если мы отсечем некоторые из повторяющихся состояний, то сможем уменьшить дерево поиска до конечного размера, формируя только ту часть дерева, которая охватывает весь граф пространства состояний. Рассматривая только часть дерева поиска вплоть до некоторой постоянной глубины, можно легко обнаружить ситуации, в которых удаление повторяющихся состояний позволяет достичь экспоненциального уменьшения стоимости поиска. В крайнем случае пространство состояний размером  $d+1$  (рис. 3.11, а) становится деревом с  $2^d$  листьями (рис. 3.11, б). Более реалистичным примером может служить **прямоугольная решетка**, как показано на рис. 3.11, в. В решетке каждое состояние имеет четырех преемников, поэтому дерево поиска, включающее повторяющиеся состояния, имеет  $4^d$  листьев, но существует приблизительно только  $2d^2$  различных состояний с  $d$  этапами достижения любого конкретного состояния. Для  $d=20$  это означает, что существует около триллиона узлов, но лишь примерно 800 различных состояний.

Таким образом, неспособность алгоритма обнаруживать повторяющиеся состояния может послужить причиной того, что разрешимая задача станет неразрешимой. Такое обнаружение обычно сводится к тому, что узел, подлежащий развертыванию, сравнивается с теми узлами, которые уже были развернуты; если обнаружено совпадение, то алгоритм распознал наличие двух путей в одно и то же состояние и может отбросить один из них.

При поиске в глубину в памяти хранятся только те узлы, которые лежат на пути от корня до текущего узла. Сравнение этих узлов с текущим узлом позволяет алгоритму обнаружить заклинивающие пути, которые могут быть немедленно отброшены. Это позволяет обеспечить, чтобы конечные пространства состояний не превращались в бесконечные деревья поиска из-за циклов, но, к сожалению, не дает возможности предотвратить экспоненциальное разрастание нециклических путей в задачах, подобных приведенным на рис. 3.11. Единственный способ предотвращения этого состоит в том, что в памяти нужно хранить больше узлов. В этом заключается фундаментальный компромисс между пространством и временем. *Алгоритмы, которые забывают свою историю, обречены на то, чтобы ее повторять.*

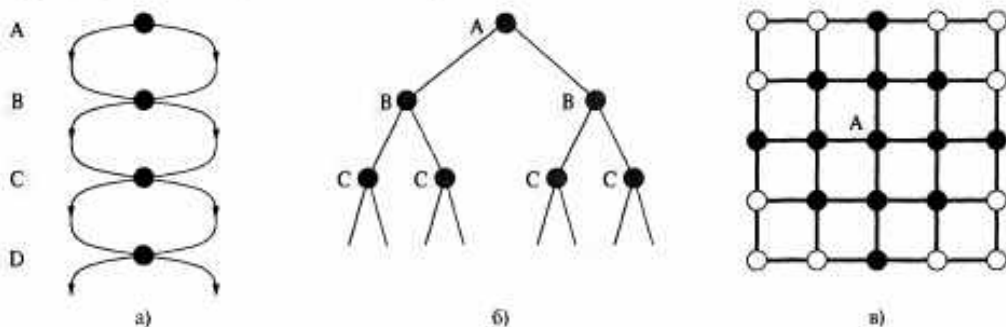


Рис. 3.11. Пространства состояний, которые формируют экспоненциально более крупные деревья поиска: пространство состояний, в котором имеются два возможных действия, ведущих от A к B, два — от B к C и т.д.; это пространство состояний содержит  $d+1$  состояний, где  $d$  — максимальная глубина (а); дерево поиска, которое имеет  $2^d$  ветвей, соответствующих  $2^d$  путям через это пространство (б); пространство состояний в виде прямоугольной решетки (в); состояния, находящиеся в пределах 2 этапов от начального состояния (A), обозначены серым цветом

Если некоторый алгоритм запоминает каждое состояние, которое он посетил, то может рассматриваться как непосредственно исследующий граф пространства состояний. В частности, можно модифицировать общий алгоритм *Tree-Search*, чтобы включить в него структуру данных, называемую  $\Sigma$  **закрытым списком**, в котором хранится каждый развернутый узел. (Периферию, состоящую из неразвернутых узлов, иногда называют  $\Sigma$  **открытым списком**.) Если текущий узел совпадает с любым узлом из закрытого списка, то не развертывается, а отбрасывается. Этому новому алгоритму присвоено название алгоритма поиска в графе, *Graph-Search* (листинг 3.6). При решении задач со многими повторяющимися состояниями алгоритм *Graph-Search* является намного более эффективным по сравнению с *Tree-Search*. В наихудшем случае предъявляемые им требования к времени и пространству пропорциональны размеру пространства состояний. Эта величина может оказаться намного меньше, чем  $O(b^d)$ .

Вопрос о том, оптимален ли поиск в графе, остается сложным. Выше было указано, что появление повторяющегося состояния соответствует обнаружению алгоритмом двух путей в одно и то же состояние. Алгоритм *Graph-Search*, приведенный в листинге 3.6, всегда отбрасывает вновь обнаруженный путь и оставляет первоначальный; очевидно, что если этот вновь обнаруженный путь короче, чем первоначальный, то алгоритм *Graph-Search* может упустить оптимальное решение. К счастью,

можно показать (упр. 3.12), что этого не может случиться, если используется либо поиск по критерию стоимости, либо поиск в ширину с постоянными стоимостями этапов; таким образом, эти две оптимальные стратегии поиска в дереве являются также оптимальными стратегиями поиска в графе. При поиске с итеративным углублением, с другой стороны, используется разветвление в глубину, поэтому этот алгоритм вполне может проследовать к некоторому узлу по неоптимальному пути, прежде чем найти оптимальный. Это означает, что при поиске в графе с итеративным углублением необходимо проверять, не является ли вновь обнаруженный путь к узлу лучшим, чем первоначальный, и в случае положительного ответа в нем может потребоваться пересматривать значения глубины и стоимости путей для потомков этого узла.

**Листинг 3.6. Общий алгоритм поиска в графе.** Множество *closed* может быть реализовано с помощью хэш-таблицы для обеспечения эффективной проверки повторяющихся состояний. В этом алгоритме предполагается, что первый найденный путь к состоянию *s* является наименее дорогостоящим (см. текст)

---

```

function Graph-Search(problem, fringe) returns решение
    или индикатор неудачи failure

    closed ← пустое множество
    fringe ← Insert(Make-Node(Initial-State[problem]), fringe)
    loop do
        if Empty?(fringe) then return индикатор неудачи failure
        node ← Remove-First(fringe)
        if Goal-Test[problem](State[node]) then return
            Solution(node)
        if State[node] не находится в множестве closed then
            добавить State[node] к множеству closed
            fringe ← Insert-All(Expand(node, problem), fringe)

```

---

Между прочим, использование закрытого списка *closed* означает, что поиск в глубину и поиск с итеративным углублением больше не имеют линейных требований к пространству. Поскольку в алгоритме Graph-Search каждый узел хранится в памяти, некоторые методы поиска становятся неосуществимыми из-за недостаточного объема памяти.

## 3.6. ПОИСК С ЧАСТИЧНОЙ ИНФОРМАЦИЕЙ

---

В разделе 3.3 было выдвинуто предположение, что среда является полностью наблюдаемой и детерминированной и что агент имеет информацию о том, каковы последствия каждого действия. Поэтому агент может точно вычислить, какое состояние становится результатом любой последовательности действий, и всегда знает, в каком состоянии он находится. Его восприятия не предоставляют новой информации после выполнения каждого действия. Но что произойдет, если знания о состояниях или действиях являются неполными? Авторы обнаружили, что разные типы неполноты приводят к трем перечисленным ниже типам проблем.

1. **Проблемы отсутствия датчиков** (называемые также **проблемами совместимости**). Если агент вообще не имеет датчиков, то (насколько ему известно) может находиться в одном из нескольких возможных начальных состояний и поэтому каж-

дое действие способно перевести его в одно из нескольких возможных состояний-преемников.

- 2. Проблемы непредвиденных ситуаций.** Если среда наблюдаема лишь частично или действия являются неопределенными, то акты восприятия агента предоставляют новую информацию после выполнения каждого действия. Каждое возможное восприятие определяет непредвиденную ситуацию, к которой необходимо подготовиться с помощью соответствующего плана. Проблема называется **обусловленной сторонним воздействием**, если неопределенность вызвана действиями другого агента.
- 3. Проблемы исследования.** Если состояния и действия в среде неизвестны, агент должен действовать так, чтобы их обнаружить. Проблемы исследования могут рассматриваться как крайний случай проблем непредвиденных ситуаций.

В качестве примера мы будем использовать среду мира пылесоса. Напомним, что пространство состояний имеет восемь состояний, как показано на рис. 3.12. Существуют три действия (*Left*, *Right* и *Suck*), и цель состоит в том, чтобы был убран весь мусор (состояния 7 и 8). Если среда наблюдаема, детерминирована и полностью известна, то эта задача решается тривиально с помощью любого из описанных нами алгоритмов. Например, если начальным является состояние 5, то последовательность действий [*Right*, *Suck*] обеспечивает достижение целевого состояния 8. В оставшейся части этого раздела рассматриваются версии данной задачи, в которых отсутствуют датчики и возникают непредвиденные ситуации. Проблемы исследования описаны в разделе 4.5, а проблемы, обусловленные сторонним воздействием, — в главе 6.

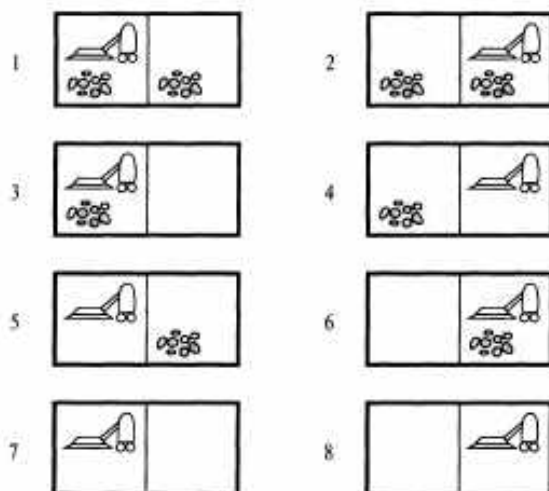


Рис. 3.12. Восемь возможных состояний мира пылесоса

### Проблемы отсутствия датчиков

Предположим, что агенту-пылесосу известны все последствия его действий, но он не имеет датчиков. В таком случае агент знает только, что его начальным состоянием является одно состояние из множества  $\{1, 2, 3, 4, 5, 6, 7, 8\}$ . На первый взгляд можно предположить, что попытки агента предсказать будущую ситуацию окажутся бесполез-



ными, но фактически он может сделать это вполне успешно. Поскольку агент знает, к чему приводят его действия, то может, например, вычислить, что действие *Right* вызовет переход его 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. Мы утверждаем, что агент может **заставить** перевести мир в состояние 7, даже если ему не известно, с какого состояния он начинается. Подведем итог: если мир не является полностью наблюдаемым, то агент должен рассуждать о том, в какое множество состояний (а не в единственное состояние) он может попасть. Мы называем каждое такое множество состояний **доверительным состоянием**, поскольку оно показывает, в каких возможных физических состояниях агент может считать себя находящимся в данный момент со всей уверенностью. (В полностью наблюдаемой среде каждое доверительное состояние содержит одно физическое состояние.)

Для решения проблемы отсутствия датчиков необходимо выполнять поиск в пространстве доверительных, а не физических состояний. Первоначальное состояние является доверительным состоянием, а каждое действие становится отображением из одного доверительного состояния в другое. Результат применения некоторого действия к некоторому доверительному состоянию определяется путем объединения результатов применения этого действия к каждому физическому состоянию из этого доверительного состояния. Теперь любой путь объединяет несколько доверительных состояний, а решением является путь, который ведет к такому доверительному состоянию, все члены которого представляют собой целевые состояния. На рис. 3.13 показано пространство достижимых доверительных состояний для детерминированного мира пылесоса без датчиков. Существует только 12 достижимых доверительных состояний, но все пространство доверительных состояний включает каждое возможное множество физических состояний, т.е.  $2^8=256$  доверительных состояний. Вообще говоря, если пространство физических состояний имеет  $S$  состояний, то пространство доверительных состояний имеет  $2^S$  доверительных состояний.

В приведенном выше описании проблем отсутствия датчиков предполагалось, что действия являются детерминированными, но этот анализ, по сути, остается неизменным, если среда — недетерминированная, т.е. если действия могут иметь несколько возможных результатов. Причина этого состоит в том, что в отсутствие датчиков агент не способен определить, какой результат достигнут фактически, поэтому различные возможные результаты становятся просто дополнительными физическими состояниями в доверительном состоянии-преемнике. Например, предположим, что среда подчиняется закону Мэрфи (или закону “подлости”): так называемое действие *Suck* иногда оставляет мусор на полу, но только если на нем еще не было мусора<sup>6</sup>. В таком случае, если действие *Suck* применяется в физическом состоянии 4 (см. рис. 3.12), то существуют два возможных результата: состояния 2 и 4. Теперь применение действия *Suck* в начальном доверительном состоянии,  $\{1, 2, 3, 4, 5, 6, 7, 8\}$ , приводит к доверительному состоянию, представляю-

<sup>6</sup> Мы предполагаем, что большинство читателей сталкиваются с аналогичными проблемами и поэтому выражают сочувствие нашему агенту. Авторы приносят свои извинения владельцам современных, эффективных бытовых приборов, которые не смогут извлечь урок из этого педагогического упражнения.

шему собой объединение множеств результатов для этих восьми физических состояний. Проведя эти вычисления, можно обнаружить, что новым доверительным состоянием снова становится  $\{1, 2, 3, 4, 5, 6, 7, 8\}$ . Таким образом, для агента без датчиков в мире закона Мэрфи действие *Suck* оставляет доверительное состояние неизменным! Это означает, что фактически данная задача неразрешима (см. упр. 3.18). Интуитивно можно понять, что причина этого состоит в том, что агент не может определить, является ли текущий квадрат грязным и поэтому не способен установить, приведет ли действие *Suck* к его очистке или оставит еще больше мусора.

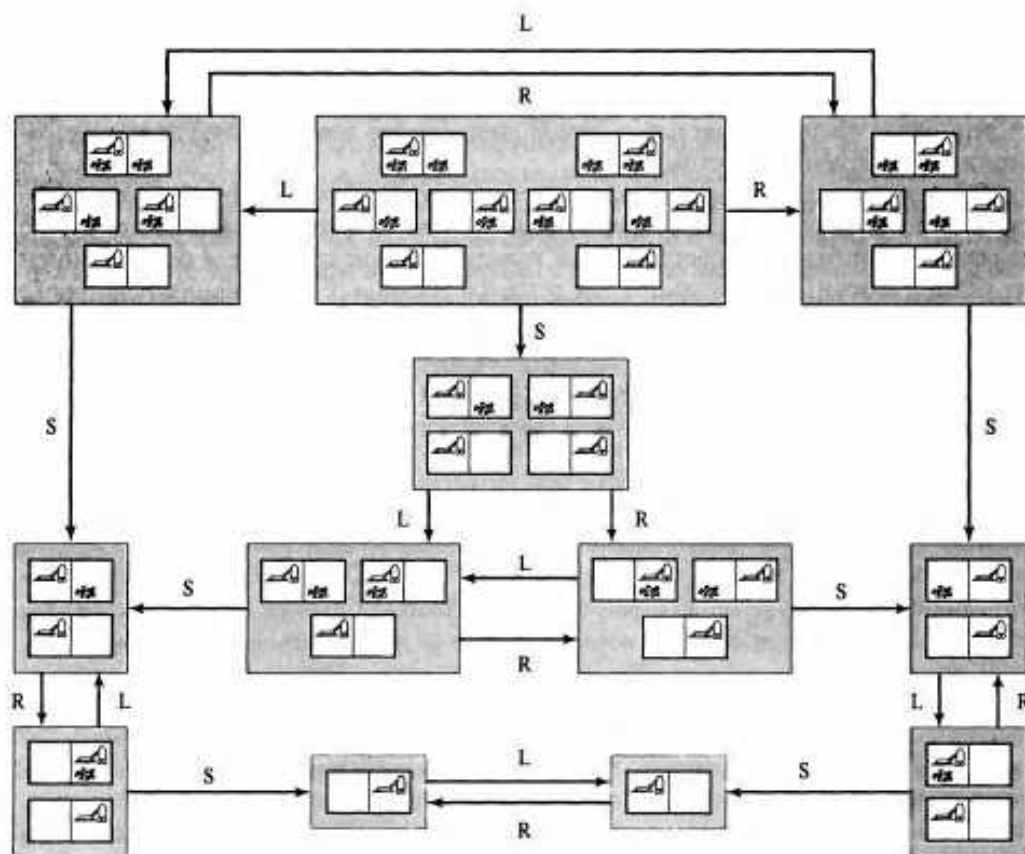



Рис. 3.13. Достижимая часть пространства доверительных состояний для детерминированного мира пылесоса без датчиков. Каждый затененный прямоугольник соответствует одному доверительному состоянию. В любой конкретный момент времени агент находится в одном конкретном доверительном состоянии, но не знает, в каком именно физическом состоянии 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 для упрощения рисунка

### Проблемы непредвиденных ситуаций

Если среда является таковой, что агент после выполнения действия может получить от своих датчиков в том же состоянии новую информацию, то агент сталкива-


ется с  проблемой непредвиденных ситуаций. Решение проблемы непредвиденных ситуаций часто принимает форму дерева, в котором каждая ветвь может быть выбрана в зависимости от результатов восприятий, полученных вплоть до этой позиции в дереве. Например, предположим, что агент находится в мире закона Мэрфи и имеет датчик положения и локальный датчик мусора, но не имеет датчика, способного обнаружить мусор в других квадратах. Таким образом, восприятие  $[L, Dirty]$  означает, что агент находится в одном из состояний  $\{1, 3\}$ . Агент может сформулировать последовательность действий  $[Suck, Right, Suck]$ . Всасывание должно было бы перевести текущее состояние в одно из состояний  $\{5, 7\}$ , и тогда передвижение вправо перевело бы данное состояние в одно из состояний  $\{6, 8\}$ . Выполнение заключительного действия *Suck* в состоянии 6 переводит агента в состояние 8, целевое, но выполнение его в состоянии 8 способно снова перевести агента в состояние 6 (согласно закону Мэрфи), и в этом случае данный план оканчивается неудачей.

Исследуя пространство доверительных состояний для этой версии задачи, можно легко определить, что ни одна фиксированная последовательность действий не гарантирует решение данной задачи. Однако решение существует, при условии, что мы не будем настаивать на фиксированной последовательности действий:

$[Suck, Right, \text{if } [R, Dirty] \text{ then } Suck]$

Тем самым дополняется пространство решений для включения возможности выбора действий с учетом непредвиденных ситуаций, возникающих во время выполнения. Многие задачи в реальном, физическом мире усложняются из-за наличия проблемы непредвиденных ситуаций, поскольку точные предсказания в них невозможны. По этой причине многие люди соблюдают предельную осторожность во время пеших прогулок или вождения автомобиля.

Иногда проблемы непредвиденных ситуаций допускают чисто последовательные решения. Например, рассмотрим полностью наблюдаемый мир закона Мэрфи. Непредвиденные ситуации возникают, если агент выполняет действие *Suck* в чистом квадрате, поскольку при этом в данном квадрате может произойти или не произойти отложение мусора. При том условии, что агент никогда не будет очищать чистые квадраты, не будут возникать какие-либо непредвиденные ситуации и поэтому из любого начального состояния можно будет найти последовательное решение (упр. 3.18).

Алгоритмы для решения проблем непредвиденных ситуаций являются более сложными по сравнению с алгоритмами стандартного поиска, приведенными в этой главе; они рассматриваются в главе 12. Кроме того, проблемы непредвиденных ситуаций требуют применения немного иного проекта агента, в котором агент может действовать до того, как найдет гарантированный план. Это полезно, поскольку вместо предварительного обдумывания каждой возможной непредвиденной ситуации, которая могла бы возникнуть во время выполнения, часто бывает лучше приступить к действию и узнать, какие непредвиденные ситуации действительно возникают. После этого агент может продолжать решать свою задачу, принимая в расчет эту дополнительную информацию. Такой тип  чередования поиска и выполнения является также полезным при решении проблем исследования (см. раздел 4.5) и при ведении игр (см. главу 6).

## РЕЗЮМЕ

В настоящей главе представлены методы, которые могут использоваться агентом для выбора действий в таких вариантах среды, которые являются детерминированными, наблюдаемыми, статическими и полностью известными. В таких случаях агент может формировать последовательности из действий, позволяющих ему достичь своих целей; такой процесс называется **поиском**.

- Прежде чем агент сможет приступить к поиску решений, он должен сформулировать **цель**, а затем использовать эту цель для формулировки **задачи**.
- Задача состоит из четырех частей: **начальное состояние**, множество **действий**, функция **проверки цели** и функция **стоимости пути**. Среда задачи представлена **пространством состояний**, а путь через пространство состояний от начального состояния до целевого состояния представляет собой **решение**.
- Для решения любой задачи может использоваться единый, общий алгоритм **Tree-Search**; конкретные варианты этого алгоритма воплощают различные стратегии.
- Алгоритмы поиска оцениваются на основе **полноты**, **оптимальности**, **временной** и **пространственной сложности**. Сложность зависит от коэффициента ветвления в пространстве состояний,  $b$ , и глубины самого поверхностного решения,  $d$ .
- При **поиске в ширину** для развертывания выбирается самый поверхностный неразвернутый узел в дереве поиска. Этот поиск является полным, оптимальным при единичных стоимостях этапов и характеризуется временной и пространственной сложностью  $O(b^d)$ . В связи с такой пространственной сложностью в большинстве случаев он становится практически не применимым. **Поиск по критерию стоимости** аналогичен поиску в ширину, но предусматривает развертывание узла с самой низкой стоимостью пути,  $g(n)$ . Он является полным и оптимальным, если стоимость каждого шага превышает некоторое положительное предельное значение  $\epsilon$ .
- При **поиске в глубину** для развертывания выбирается самый глубокий неразвернутый узел в дереве поиска. Этот поиск не является ни полным, ни оптимальным, и характеризуется временной сложностью  $O(b^m)$  и пространственной сложностью  $O(bm)$ , где  $m$  — максимальная глубина любого пути в пространстве состояний.
- При **поиске с ограничением глубины** на поиск в глубину налагается установленный предел глубины.
- При **поиске с итеративным углублением** вызывается поиск с ограничением глубины и каждый раз устанавливаются увеличивающиеся пределы, до тех пор, пока цель не будет найдена. Этот поиск является полным и оптимальным при единичных стоимостях этапов и характеризуется временной сложностью  $O(b^d)$  и пространственной сложностью  $O(bd)$ .
- **Двунаправленный поиск** способен чрезвычайно уменьшить временную сложность, но он не всегда применим и может потребовать слишком много пространства.

- Если пространство состояний представляет собой граф, а не дерево, то может оказаться целесообразной проверка повторяющихся состояний в дереве поиска. Алгоритм Graph-Search устраняет все дублирующие состояния.
- Если среда является частично наблюдаемой, то агент может применить алгоритмы поиска в пространстве **доверительных состояний**, или множестве возможных состояний, в которых может находиться агент. В некоторых случаях может быть сформирована единственная последовательность решения; в других случаях агенту требуется **план действий в непредвиденных ситуациях**, чтобы иметь возможность справиться со всеми неизвестными обстоятельствами, которые могут возникнуть.

## БИБЛИОГРАФИЧЕСКИЕ И ИСТОРИЧЕСКИЕ ЗАМЕТКИ

Большинство задач поиска в пространстве состояний, проанализированных в этой главе, имеют длинную историю, отраженную в литературе, и являются менее тривиальными, чем может показаться на первый взгляд. Задача с миссионерами и каннибалами, используемая в упр. 3.9, была подробно проанализирована Амарелем [24]. До него эту задачу ввели в проблематику искусственного интеллекта Саймон и Ньюэлл [1420], и в проблематику исследования операций — Беллман и Дрейфус [96]. Работы наподобие проведенных Ньюэллом и Саймоном над программами Logic Theorist [1127] и GPS [1129] привели к тому, что алгоритмы поиска считались основным оружием в арсенале исследователей искусственного интеллекта 1960-х годов, а сами процессы решения задач стали рассматриваться в качестве канонической проблемы искусственного интеллекта. К сожалению, в то время в области автоматизации этапа формулировки задачи было предпринято слишком мало усилий. Более современная трактовка подхода к представлению и абстрагированию задач, включая применение программ искусственного интеллекта, которые (отчасти) сами выполняют эти функции, изложена в книге Кноблока [807].

Задача игры в восемь — это “младшая сестра” задачи игры в пятнадцать, которая была изобретена знаменитым американским разработчиком игр Сэмом Ллойдом [955] в 1870-х годах. Игра в пятнадцать быстро приобрела в Соединенных Штатах огромную популярность, которую можно сравнить лишь с более современной сенсацией, вызванной изобретением кубика Рубика. Она также быстро привлекла внимание математиков [739], [1486]. Редакторы *American Journal of Mathematics* заявили: “Игра в пятнадцать в последние несколько недель занимает все внимание американской публики, и можно с уверенностью сказать, что ею интересуются девять из десяти людей всех полов и возрастов и всех общественных положений. Но не это побудило нас, редакторов, включить статьи, посвященные этой теме, в наш журнал *American Journal of Mathematics*, а тот факт, что...” (далее следует краткое описание аспектов игры в пятнадцать, представляющих интерес с точки зрения математики). Исчерпывающий анализ игры в восемь был проведен с помощью компьютера Шофилдом [1363]. Ратнер и Уормут [1268] показали, что общая версия игры (не в пятнадцать, а в  $n \times n - 1$ ) принадлежит к классу NP-полных задач.

Задача с восемью ферзями была впервые опубликована анонимно в немецком шахматном журнале *Schach* в 1848 году; позднее ее создание приписали некоему

Максу Беззелью. Она была повторно опубликована в 1850 году и на этот раз привлекала внимание выдающегося математика Карла Фридриха Гаусса, который попытался перечислить все возможные решения, но нашел только 72. Наук (Nauck) опубликовал все 92 решения позднее, в том же 1850 году. Нетто [1121] обобщил эту задачу до  $n$  ферзей, а Абрамсон и Янг [2] нашли алгоритм с оценкой  $O(n)$ .

Каждая из реальных задач поиска, перечисленных в данной главе, была предметом значительных усилий исследователей. Методы выбора оптимальных расписаний авиаперелетов по большей части остаются недоступными для общего пользования (закрытыми), но Карл де Маркен (Carl de Marcken) сообщил авторам (в личной беседе), что структуры формирования цен на авиабилеты и связанные с ними ограничения стали настолько сложными, что задача выбора оптимального авиарейса является формально неразрешимой. Задача коммивояжера (Traveling Salesperson Problem — TSP) — это стандартная комбинаторная проблема в теоретических компьютерных науках [898], [899]. Карп [772] доказал, что задача TSP является NP-трудной, но для нее были разработаны эффективные методы эвристической аппроксимации [933]. Ароа [41] разработал полностью полиномиальную схему аппроксимации для евклидовых вариантов задачи TSP. Обзор методов компоновки СБИС был сделан Шахукаром и Мазумдером [1390], а в журналах по сверхбольшим интегральным микросхемам (СБИС) появилось много статей, посвященных оптимизации компоновки. Задачи робототехнической навигации и сборки обсуждаются в главе 25.

Алгоритмы неинформированного поиска для решения задач являются центральной темой классических компьютерных наук [680] и исследования операций [417]; более современные результаты исследований приведены в книгах Део и Панга [390], а также Галло и Паллоттино [516]. Метод поиска в ширину применительно к решению задач с лабиринтами был сформулирован Муром [1078]. Метод **динамического программирования** [96], в котором предусматривается систематическая регистрация решений для всех подзадач с возрастающей длиной, может рассматриваться как форма поиска в ширину в графах. Алгоритм определения кратчайшего пути между двумя точками, предложенный Дейкстрой [399], является предшественником алгоритма поиска по критерию стоимости.

Одна из версий алгоритма поиска с итеративным углублением, предназначенная для эффективного ведения игры в шахматы с контролем времени, была впервые применена Слейтом и Аткином [1429] в программе ведения шахматной игры Chess 4.5, но ее применению для поиска кратчайшего пути в графе мы обязаны Корфу [835]. В некоторых случаях может также оказаться очень эффективным двуправленный поиск, который был предложен Полом [1219], [1221].

Частично наблюдаемые и недетерминированные варианты среды не были достаточно глубоко исследованы в этом подходе к решению задач. Некоторые проблемы эффективности при поиске в пространстве доверительных состояний рассматривались Генезеретом и Нурбакшем [538]. Кёниг и Симмонс [819] изучали навигацию робота из неизвестной начальной позиции, а Эрдман и Мэйсон [439] исследовали проблему робототехнического манипулирования без датчиков, используя непрерывную форму поиска в пространстве доверительных состояний. Поиск в условиях непредвиденных ситуаций изучался в этой подобласти планирования (см. главу 12). По большей части в подходе к изучению планирования и осуществления действий с неопределенной информацией используются инструментальные средства теории вероятностей и теории решений (см. главу 17).

Книги Нильссона [1141], [1142] являются хорошим общим источником информации о классических алгоритмах поиска. Исчерпывающий и более современный обзор можно найти в [838]. Статьи о новых алгоритмах поиска (которые, как это ни удивительно, продолжают изобретаться) публикуются в таких журналах, как *Artificial Intelligence*.


## УПРАЖНЕНИЯ

- 3.1. Самостоятельно сформулируйте определения следующих понятий: состояние, пространство состояний, дерево поиска, поисковый узел, цель, действие, функция определения преемника и коэффициент ветвления.
- 3.2. Объясните, почему составление формулировки задачи должно осуществляться вслед за составлением формулировки цели.
- 3.3. Предположим, что выражение  $\text{Legal-Actions}(s)$  обозначает множество действий, которые являются допустимыми в состоянии  $s$ , а выражение  $\text{Result}(a, s)$  обозначает состояние, которое следует из выполнения допустимого действия  $a$  в состоянии  $s$ . Определите функцию  $\text{Successor-Fn}$  в терминах выражений  $\text{Legal-Actions}$  и  $\text{Result}$ , и наоборот.
- 3.4. Покажите, что в задаче игры в восемь состояния подразделяются на два непесекающихся множества, таких, что ни одно состояние из первого множества не может быть преобразовано в состояние из второго множества, даже с применением сколь угодно большого количества ходов. (*Подсказка.* См. [102].) Разработайте процедуру, позволяющую узнать, к какому множеству относится данное состояние, и объясните, для чего нужно иметь под рукой такую процедуру, формируя состояния случайным образом.
- 3.5. Рассмотрите задачу с  $n$  ферзями, используя “эффективную” инкрементную формулировку, приведенную на с. 119. Объясните, почему размер пространства состояний равен по меньшей мере  $\sqrt[n]{n!}$ , и оцените наибольшее значение  $n$ , для которого является осуществимым исчерпывающее исследование. (*Подсказка.* Определите нижнюю границу коэффициента ветвления, рассматривая максимальное количество клеток, которые ферзь может атаковать в любом столбце.)
- 3.6. Всегда ли наличие конечного пространства состояний приводит к получению конечного дерева поиска? А что можно сказать о конечном пространстве состояний, которое является деревом? Можете ли вы указать более точно, применение пространств состояний каких типов всегда приводит к получению конечных деревьев поиска? (*Адаптировано из [99].*)
- 3.7. Укажите для каждой из следующих задач ее компоненты: начальное состояние, проверку цели, функцию определения преемника и функцию стоимости. Выберите формулировку, которая является достаточно точной, чтобы ее можно было успешно реализовать.
  - а) Необходимо раскрасить плоскую карту, используя только четыре цвета, таким образом, чтобы никакие два смежных региона не имели один и тот же цвет.

- б) Обезьяна, имеющая рост 3 фута, находится в комнате, где под потолком, на высоте 8 футов, подвешено несколько бананов. Обезьяна хочет получить бананы. В комнате находятся две проволочные корзины высотой по 3 фута каждая, которые можно передвигать, ставить друг на друга и на которые можно залезать.
- в) Некоторая программа выводит сообщение “недопустимая входная запись” после передачи ей некоторого файла, состоящего из входных записей. Известно, что обработка каждой записи происходит независимо от других записей. Требуется обнаружить, какая запись является недопустимой.
- г) Имеются три кувшина, с емкостью 12 галлонов, 8 галлонов и 3 галлона, а также водопроводный кран. Кувшины можно заполнять или опорожнять, выливая воду из одного кувшина в другой или на землю. Необходимо отмерить ровно один галлон.





3.8. Рассмотрите пространство состояний, в котором начальным состоянием является число 1, а функция определения преемника для состояния  $n$  возвращает два состояния: числа  $2n$  и  $2n+1$ .

- а) Нарисуйте часть пространства состояний, которая относится к состояниям 1–15.
- б) Предположим, что целевым состоянием является 11. Перечислите последовательности, в которых будут посещаться узлы при поиске в ширину, поиске с ограничением глубины, с пределом 3, и поиске с итеративным углублением.
- в) Будет ли подходящим для решения этой задачи двунаправленный поиск? Если да, то опишите подробно, как действовал бы этот метод поиска.
- г) Каковым является коэффициент ветвления в каждом направлении двунаправленного поиска?
- д) Не подсказывает ли вам ответ на вопрос упр. 3.8, в, что нужно найти другую формулировку этой задачи, которая позволила бы решить проблему перехода из состояния 1 в заданное целевое состояние почти без поиска?

3.9.  Задача с миссионерами и каннибалами обычно 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. Это — известная задача в искусственном интеллекте, поскольку она была темой первой статьи, в которой был применен подход к формулировке проблемы с аналитической точки зрения [24].

- а) Точно сформулируйте эту задачу, определяя только те различия, которые необходимы для обеспечения правильного решения. Нарисуйте схему полного пространства состояний.
- б) Реализуйте и решите эту задачу оптимальным образом, используя соответствующий алгоритм поиска. Действительно ли имеет смысл проверять наличие повторяющихся состояний?
- в) Почему, по вашему мнению, люди сталкиваются с затруднениями при решении этой головоломки, несмотря на то, что пространство ее состояний является чрезвычайно простым?



- 3.10.  Реализуйте следующие две версии функции определения преемника для задачи игры в восемь. Первая из них должна формировать сразу всех преемников, копируя и редактируя структуру данных игры в восемь, а вторая при каждом ее вызове должна формировать по одному новому преемнику и действовать по принципу непосредственной модификации родительского состояния (отменяя эти модификации в случае необходимости). Напишите версии процедуры поиска в глубину с итеративным углублением, в которых используются эти функции, и сравните данные об их производительности.
- 3.11.  На с. 135 упоминался поиск с итеративным удлинением, итеративный аналог поиска по критерию стоимости. Его идея состоит в том, что должны использоваться увеличивающиеся пределы стоимости пути. Если сформирован узел, стоимость пути для которого превышает текущий предел, этот узел немедленно отбрасывается. При каждой новой итерации предел устанавливается равным самому низкому значению стоимости пути для любого узла, отвергнутого в предыдущей итерации.
- а) Покажите, что этот алгоритм является оптимальным применительно к общим методам определения стоимости пути.
  - б) Рассмотрите однородное дерево с коэффициентом ветвления  $b$ , глубиной решения  $d$  и единичными стоимостями этапов. Какое количество итераций требуется при итеративном удлинении?
  - в) Рассмотрите стоимости этапов, взятые из непрерывного диапазона  $[0, 1]$  с минимальной положительной стоимостью  $\epsilon$ . Сколько итераций потребуется в самом неблагоприятном случае?
  - г) Реализуйте этот алгоритм и примените его к экземплярам задачи игры в восемь и задачи коммивояжера. Сравните производительность этого алгоритма с производительностью алгоритма поиска по критерию стоимости и прокомментируйте полученные результаты.
- 3.12. Докажите, что поиск по критерию стоимости и поиск в ширину с постоянными значениями стоимости этапа являются оптимальными при их использовании с алгоритмом Graph-Search. Продемонстрируйте такое пространство состояний с постоянными значениями стоимости этапа, в котором алгоритм Graph-Search с использованием итеративного углубления находит неоптимальное решение.
- 3.13. Опишите пространство состояний, в котором поиск с итеративным углублением характеризуется гораздо более низкой производительностью по сравнению с поиском в глубину (например,  $O(n^2)$ , в отличие от  $O(n)$ ).
- 3.14.  Напишите программу, которая принимает в качестве входных данных URL-локаторы двух Web-страниц и находит путь от одной к другой, состоящий из ссылок. В чем состоит подходящая для этого стратегия поиска? Действительно ли имеет смысл применять двунаправленный поиск? Можно ли использовать для реализации функции определения предшественника машину поиска?
- 3.15.  Рассмотрите задачу определения кратчайшего пути между двумя точками на плоскости, на которой имеются препятствия в виде выпуклых многоугольников, как показано на рис. 3.14. Это — идеализация задачи, которую должен

решать робот, прокладывая свой путь через среду, в которой очень мало свободного места.

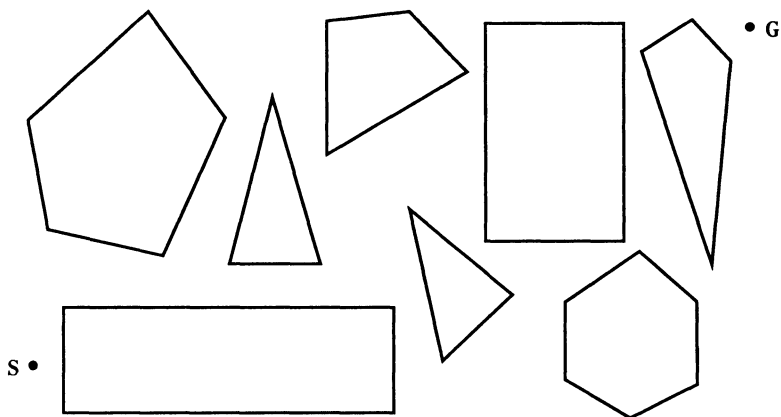



Рис. 3.14. Сцена с многоугольными препятствиями

- а) Предположим, что пространство состояний состоит из всех позиций  $(x, y)$  на плоскости. Каково количество состояний в этом пространстве? Каково в нем количество путей к цели?
  - б) Объясните в нескольких словах, почему в этой двухмерной сцене кратчайший путь от одной вершины многоугольника до любой другой должен состоять из прямолинейных отрезков, соединяющих некоторые вершины многоугольников. Теперь определите более приемлемое пространство состояний. Насколько велико это пространство состояний?
  - в) Определите функции, необходимые для реализации решения этой задачи поиска, включая функцию определения преемника, которая принимает в качестве входных данных координаты любой из вершин и возвращает множество вершин, достижимых по прямой линии от данной вершины. (Не забывайте при этом о соседних вершинах того же многоугольника.) Используйте в качестве значения эвристической функции расстояние между указанными точками по прямой.
  - г) Примените для решения ряда задач из этой области один или несколько алгоритмов, представленных в настоящей главе, и прокомментируйте данные об их производительности.
- 3.16.  Определите задачу обеспечения навигации робота, приведенной в упр. 3.15, можно преобразовать в определение среды следующим образом.
- Акт восприятия будет представлять собой список позиций видимых вершин, сформированный относительно агента. Акт восприятия не предусматривает определение позиции робота! Робот должен определять свою собственную позицию по карте; на данный момент можно предположить, что каждое местоположение имеет другое “представление”.
  - Каждое действие должно быть вектором, описывающим прямолинейный путь, по которому будет следовать робот. Если путь свободен, то действие выполняется успешно; в противном случае робот останавливается в той точке,

где его путь впервые сталкивается с препятствием. Если агент возвращает нулевой вектор движений и находится в целевой позиции (которая является постоянной и известной), то среда должна телепортировать этого агента в случайно выбранное местоположение (не находящееся в пределах препятствия).

- Показатель производительности предусматривает штрафование агента на 1 пункт за каждую единицу пройденного расстояния и награждение его 1000 пунктами каждый раз после достижения цели.

Ниже перечислены предлагаемые задания.

- а) Реализуйте описанную среду и агента, решающего задачи для этой среды. После каждой телепортации агент должен будет сформулировать новую задачу, которая предусматривает также определение его текущего местоположения.
- б) Зарегистрируйте данные о производительности предложенного вами агента (для этого предусмотрите выработку агентом соответствующих комментариев по мере его передвижения в среде) и составьте отчет о его производительности по данным, охватывающим больше 100 эпизодов.
- в) Модифицируйте среду так, чтобы в 30% случаев движение агента заканчивались в не предусмотренном им месте назначения (выбранном случайным образом среди других видимых вершин, если таковые имеются, а в противном случае соответствующем ситуации, в которой вообще не было выполнено никакого движения). Это — грубая модель ошибок при выполнении движений реального робота. Доработайте определение агента так, чтобы при обнаружении указанной ошибки он определял, где находится, а затем создавал план возвращения в то место, где он находился прежде, и возобновлял выполнение прежнего плана. Помните, что иногда попытка возвращения в прежнее место также может оканчиваться неудачей! Продемонстрируйте пример агента, который успешно преодолевает две последовательные ошибки движения и все равно достигает цели.
- г) Опробуйте две различные схемы возобновления работы после ошибки: во-первых, отправиться к ближайшей вершине из первоначального маршрута и, во-вторых, перепланировать маршрут к цели от нового местоположения. Сравните производительность всех схем возобновления работы. Влияет ли на результаты такого сравнения включение затрат на поиск?
- д) Теперь предположим, что есть такие местоположения, представления среды из которых являются идентичными. (Например, предположим, что мир — это решетка с квадратными препятствиями.) С какой проблемой теперь сталкивается агент? Как должны выглядеть решения?

3.17. На с. 115 было указано, что мы не будем принимать во внимание задачи с отрицательными значениями стоимости пути. В данном упражнении эта тема рассматривается немного более подробно.

- а) Предположим, что действия могут иметь произвольно большие отрицательные стоимости; объясните, почему такая ситуация может вынудить любой оптимальный алгоритм исследовать полное пространство состояний.
- б) Удастся ли выйти из этого положения, потребовав, чтобы стоимости этапов были больше или равны некоторой отрицательной константе  $c$ ? Рассмотрите и деревья, и графы.

- в) Предположим, что имеется множество операторов, образующих цикл, так что выполнение операторов этого множества в определенном порядке не приводит к какому-либо чистому изменению состояния. Если все эти операторы имеют отрицательную стоимость, то какие выводы из этого следуют применительно к оптимальному поведению агента в такой среде?
- г) Можно легко представить себе, что операторы с высокой отрицательной стоимостью имеются даже в таких проблемных областях, как поиск маршрута. Например, некоторые участки дороги могут оказаться настолько живописными, что стремление ознакомиться с ними намного перевесит обычные здравые рассуждения о стоимости, измеряемой в терминах затрат времени и топлива. Объясните, применяя точные термины, принятые в контексте поиска в пространстве состояний, почему все же люди не ведут свои автомобили неопределенно долго по живописным циклическим участкам пути, и укажите, каким образом нужно определить пространство состояний и операторы для задачи поиска маршрута, чтобы агенты с искусственным интеллектом также могли избежать попадания в цикл.
- д) Можете ли вы придумать пример такой реальной проблемной области, в которой стоимости этапов таковы, что могут вызвать возникновение цикла?

**3.18.** Рассмотрим мир пылесоса без датчиков, с двумя местоположениями, подчиняющийся закону Мэрфи. Нарисуйте пространство доверительных состояний, достижимых из начального доверительного состояния  $\{1, 2, 3, 4, 5, 6, 7, 8\}$ , и объясните, почему эта задача неразрешима. Покажите также, что если бы этот мир был полностью наблюдаемым, то существовала бы последовательность решения для каждого возможного начального состояния.

**3.19.**  Рассмотрим задачу в мире пылесоса, который определен на рис. 2.2.

- а) Какой из алгоритмов, определенных в этой главе, был бы подходящим для решения этой задачи? Должен ли этот алгоритм проверять наличие повторяющихся состояний?
- б) Примените выбранный вами алгоритм для вычисления оптимальной последовательности действий в мире с размером  $3 \times 3$ , в начальном состоянии которого в трех верхних квадратах имеется мусор, а агент находится в центре.
- в) Сконструируйте агента, выполняющего поиск в этом мире пылесоса, и оцените его работу в множестве миров с размером  $3 \times 3$ , характеризующемся вероятностью наличия мусора в каждом квадрате, равной 0.2. Включите в состав показателя производительности не только стоимость поиска, но и стоимость пути, используя для них подходящий “курс обмена”.
- г) Сравните вашего лучшего поискового агента с простым рандомизированным рефлексным агентом, который всасывает мусор, если последний имеется, а в противном случае выполняет случайно выбранные перемещения.
- д) Рассмотрите, что произошло бы, если ли бы мир расширился до размеров  $n \times n$ . Как производительность данного поискового агента и рефлексного агента зависит от  $n$ ?

# 4 ИНФОРМИРОВАННЫЙ ПОИСК И ИССЛЕДОВА- НИЕ ПРОСТРАНСТВА СОСТОЯНИЙ

*В этой главе показано, как можно с помощью информации о пространстве состояний не дать алгоритмам заблудиться в темноте.*

В главе 3 показано, что неинформированные стратегии поиска позволяют находить решения задач путем систематической выработки новых состояний и их проверки применительно к цели. К сожалению, в большинстве случаев эти стратегии являются чрезвычайно неэффективными. Как показано в настоящей главе, информированные стратегии поиска (в которых используются знания, относящиеся к конкретной задаче) обеспечивают более эффективный поиск решения. В разделе 4.1 описаны информированные версии алгоритмов главы 3, а в разделе 4.2 показано, как может быть получена необходимая информация, относящаяся к конкретной задаче. В разделах 4.3 и 4.4 представлены алгоритмы, которые выполняют исключительно **локальный поиск** в пространстве состояний, оценивая и модифицируя одно или несколько текущих состояний вместо систематического исследования путей из начального состояния. Эти алгоритмы применимы для решения задач, в которых стоимость пути не представляет интереса и требуется лишь найти состояние, соответствующее решению. К этому семейству алгоритмов локального поиска относятся методы, созданные под влиянием исследований в области статистической физики (**моделируемый отжиг**) и эволюционной биологии (**генетические алгоритмы**). Наконец, в разделе 4.5 рассматривается **поиск в оперативном режиме**, в котором агент сталкивается с полностью неизвестным пространством состояний.

## 4.1. СТРАТЕГИИ ИНФОРМИРОВАННОГО (ЭВРИСТИЧЕСКОГО) ПОИСКА

В данном разделе показано, как стратегия **информированного поиска** (в которой кроме определения самой задачи используются знания, относящиеся к данной конкретной проблемной области) позволяет находить решения более эффективно, чем стратегия неинформированного поиска.

Общий рассматриваемый здесь подход называется **поиском по первому наилучшему совпадению**. Поиск по первому наилучшему совпадению представляет собой разновидность общего алгоритма Tree-Search или Graph-Search, в котором узел для развертывания выбирается на основе **функции оценки**,  $f(n)$ . По традиции для развертывания выбирается узел с наименьшей оценкой, поскольку такая оценка измеряет расстояние до цели. Поиск по первому наилучшему совпадению может быть реализован в рамках описанной в данной книге общей инфраструктуры поиска с помощью очереди по приоритету — структуры данных, в которой периферия поиска поддерживается в возрастающем порядке  $f$ -значений.

Название “поиск по первому наилучшему совпадению” (best first search) узаконено традицией, но неточно. В конце концов, если бы мы действительно могли развертывать наилучший узел первым, то не было бы и поиска как такового; решение задачи представляло бы собой прямое шествие к цели. Единственное, что мы можем сделать, — это выбрать узел, который представляется наилучшим в соответствии с функцией оценки. Если функция оценки действительно является точной, то выбранный узел в самом деле окажется наилучшим узлом, но фактически функция оценки иногда оказывается малоприменимой и способной завести поиск в тупик. Тем не менее авторы будут придерживаться названия “поиск по первому наилучшему совпадению”, поскольку более подходящее название “поиск по первому совпадению, которое можно считать наилучшим” было бы довольно громоздким.

Существует целое семейство алгоритмов поиска по первому наилучшему совпадению, Best-First-Search, с различными функциями оценки<sup>1</sup>. Ключевым компонентом этих алгоритмов является **эвристическая функция**<sup>2</sup>, обозначаемая как  $h(n)$ :

$h(n)$  = оценка стоимости наименее дорогостоящего пути от узла  $n$   
до целевого узла

Например, в задаче поиска маршрута в Румынии можно было бы оценивать стоимость наименее дорогостоящего пути от Арада до Бухареста с помощью расстояний по прямой до Бухареста, измеряемых в узловых точках маршрута от Арада до Бухареста.

Эвристические функции (или просто эвристики) представляют собой наиболее общую форму, в которой к алгоритму поиска подключаются дополнительные знания о задаче. Эвристики рассматриваются более подробно в разделе 4.2, а на данный момент мы будем определять их как произвольные функции, относящиеся к конкретной проблеме, с одним ограничением: если  $n$  — целевой узел, то  $h(n) = 0$ . В остав-

<sup>1</sup> В упр. 4.3 предложено показать, что это семейство включает несколько знакомых читателю неинформированных алгоритмов.

<sup>2</sup> Эвристическая функция  $h(n)$  принимает в качестве входного параметра некоторый узел, но зависит только от состояния данного узла.

шейся части настоящего раздела рассматриваются два способа использования эвристической информации для управления поиском.

**Жадный поиск по первому наилучшему совпадению**

При **жадном поиске по первому наилучшему совпадению**<sup>3</sup> предпринимаются попытки развертывания узла, который рассматривается как ближайший к цели на том основании, что он со всей вероятностью должен быстро привести к решению. Таким образом, при этом поиске оценка узлов производится с использованием только эвристической функции:  $f(n) = h(n)$ .

Теперь рассмотрим, как используется этот алгоритм при решении задачи поиска маршрута в Румынии на основе эвристической функции определения **расстояния по прямой** (Straight Line Distance — SLD), для которой принято обозначение  $h_{SLD}$ . Если целью является Бухарест, то необходимо знать расстояния по прямой от каждого прочего города до Бухареста, которые приведены в табл. 4.1. Например,  $h_{SLD}(In(Arad)) = 366$ . Обратите внимание на то, что значения  $h_{SLD}$  не могут быть вычислены на основании описания самой задачи. Кроме того, для использования этой эвристической функции нужен определенный опыт, позволяющий узнать, каким образом значения  $h_{SLD}$  связаны с действительными дорожными расстояниями, а это означает, что данная функция исходит из практики.

Таблица 4.1. Значения  $h_{SLD}$  — расстояния по прямой до Бухареста

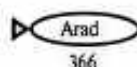
Обозначение узла	Название города	Расстояние по прямой до Бухареста	Обозначение узла	Название города	Расстояние по прямой до Бухареста
<i>Arad</i>	Арад	366	<i>Mehadia</i>	Мехадия	241
<i>Bucharest</i>	Бухарест	0	<i>Neamt</i>	Нямц	234
<i>Craiova</i>	Крайова	160	<i>Oradea</i>	Орадя	380
<i>Drobeta</i>	Дробета	242	<i>Pitesti</i>	Питешти	100
<i>Eforie</i>	Эфорие	161	<i>RimnicuVilcea</i>	Рымнику-Вылча	193
<i>Fagaras</i>	Фэгэраш	176	<i>Sibiu</i>	Сибиу	253
<i>Giurgiu</i>	Джурджу	77	<i>Timisoara</i>	Тимишоара	329
<i>Hirsova</i>	Хыршова	151	<i>Urziceni</i>	Урзичени	80
<i>Iasi</i>	Яссы	226	<i>Vaslui</i>	Васлуй	199
<i>Lugoj</i>	Лугож	244	<i>Zerind</i>	Зеринд	374

На рис. 4.1 показан процесс применения жадного поиска по первому наилучшему совпадению с использованием значений  $h_{SLD}$  для определения пути от Арада до Бухареста. Первым узлом, подлежащим развертыванию из узла *Arad*, является узел *Sibiu*, поскольку город Сибиу находится ближе к Бухаресту, чем города Зеринд или Тимишоара. Следующим узлом, подлежащим развертыванию, является узел *Fagaras*, поскольку теперь ближайшим к Бухаресту является город Фэгэраш. Узел

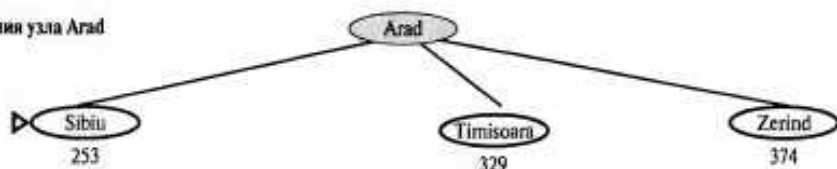
<sup>3</sup> В первом издании данной книги этот алгоритм поиска именовался просто **жадным поиском**; в книгах других авторов для него применяется название **поиск по первому наилучшему совпадению**. Авторы настоящей книги используют последний термин в более общем смысле, в соответствии с трактовкой Перла [1188].

*Fagaras*, в свою очередь, обеспечивает формирование узла *Bucharest*, который является целевым. Применение в процессе решения данной конкретной задачи алгоритма жадного поиска по первому наилучшему совпадению с использованием функции  $h_{SLD}$  позволяет найти решение без развертывания какого-либо узла, не находящегося в пути решения; это означает, что стоимость такого поиска является минимальной. Но само найденное решение не оптимально: путь до Бухареста через города Сибиу и Фэгрэш на 32 километра длиннее, чем путь через города Рымнику-Вылча и Питешти. Это замечание показывает, почему данный алгоритм называется “жадным”: на каждом этапе он пытается подойти к цели как можно ближе (фигурально выражаясь, “захватить как можно больше”).

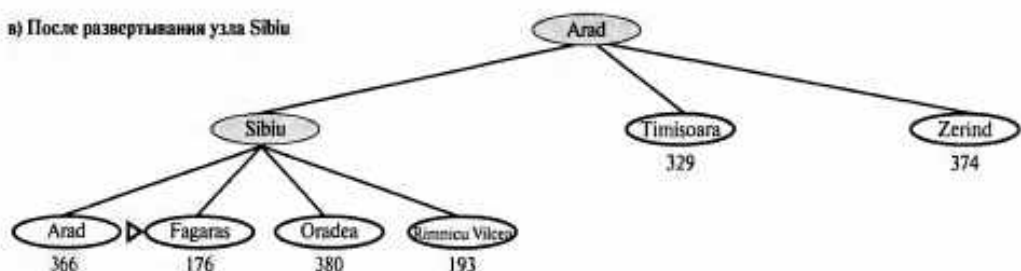
а) Начальное состояние



б) После развертывания узла *Arad*



в) После развертывания узла *Sibiu*



г) После развертывания узла *Fagaras*

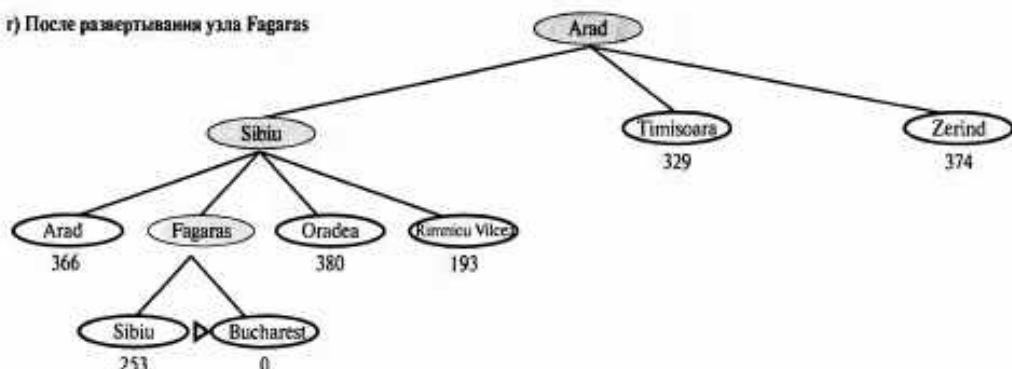


Рис. 4.1. Этапы жадного поиска пути до Бухареста по первому наилучшему совпадению с использованием эвристической функции  $h_{SLD}$ , определяющей расстояние по прямой. Узлы обозначены с помощью их  $h$ -значений

Процедура минимизации  $h(n)$  восприимчива к фальстартам (при ее использовании иногда приходится отменять начальные этапы). Рассмотрим задачу поиска пути



от города Яссы до города Фэгрэш. Эта эвристическая функция подсказывает, что в первую очередь должен быть развернут узел города Нямц, *Neamt*, поскольку он является ближайшим к узлу *Fagaras*, но этот путь становится тупиковым. Решение состоит в том, чтобы отправиться вначале в город Васлуй (а этот этап, согласно данной эвристической функции, фактически уводит дальше от цели), а затем продолжать движение через Урзичени, Бухарест и, наконец, в Фэгрэш. Поэтому в данном случае применение указанной эвристической функции вызывает развертывание ненужных узлов. Более того, если не будет предусмотрено обнаружение повторяющихся состояний, то решение так никогда не будет найдено — процедура поиска станет совершать возвратно-поступательные движения между узлами *Neamt* и *Iasi*.

Жадный поиск по первому наилучшему совпадению напоминает поиск в глубину в том отношении, что этот алгоритм предпочитает на пути к цели постоянно следовать по единственному пути, но возвращается к предыдущим узлам после попадания в тупик. Данный алгоритм страдает от тех же недостатков, что и алгоритм поиска в глубину: он не является оптимальным, к тому же он — не полный (поскольку способен отправиться по бесконечному пути, да так и не вернуться, чтобы опробовать другие возможности). При этом в наихудшем случае оценки временной и пространственной сложности составляют  $O(b^m)$ , где  $m$  — максимальная глубина пространства поиска. Однако хорошая эвристическая функция позволяет существенно сократить такую сложность. Величина этого сокращения зависит от конкретной задачи и от качества эвристической функции.

### Поиск А\*: минимизация суммарной оценки стоимости решения

Наиболее широко известная разновидность поиска по первому наилучшему совпадению называется **поиском А\*** (читается как “А звездочка”). В нем применяется оценка узлов, объединяющая в себе  $g(n)$ , стоимость достижения данного узла, и  $h(n)$ , стоимость прохождения от данного узла до цели:

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

Поскольку функция  $g(n)$  позволяет определить стоимость пути от начального узла до узла  $n$ , а функция  $h(n)$  определяет оценку стоимости наименее дорогостоящего пути от узла  $n$  до цели, то справедлива следующая формула:

$$f(n) = \text{оценка стоимости наименее дорогостоящего пути решения, проходящего через узел } n$$

Таким образом, при осуществлении попытки найти наименее дорогостоящее решение, по-видимому, разумнее всего вначале попытаться проверить узел с наименьшим значением  $g(n) + h(n)$ . Как оказалось, данная стратегия является больше чем просто разумной: если эвристическая функция  $h(n)$  удовлетворяет некоторым условиям, то поиск А\* становится и полным, и оптимальным.

Анализ оптимальности поиска А\* является несложным, если этот метод используется в сочетании с алгоритмом *Tree-Search*. В таком случае поиск А\* является оптимальным, при условии, что  $h(n)$  представляет собой **допустимую эвристическую функцию**, т.е. при условии, что  $h(n)$  никогда не переоценивает стоимость достижения цели. Допустимые эвристические функции являются по своей сути оптимистическими функциями, поскольку возвращают значения стоимости решения за-

дачи, меньшие по сравнению с фактическими значениями стоимости. А поскольку  $g(n)$  — точная стоимость достижения узла  $n$ , из этого непосредственно следует, что функция  $f(n)$  никогда не переоценивает истинную стоимость достижения решения через узел  $n$ .

Очевидным примером допустимой эвристической функции является функция определения расстояния по прямой  $h_{SLD}$ , которая уже использовалась в данной главе для поиска пути в Бухарест. Расстояние по прямой является допустимым, поскольку кратчайший путь между любыми двумя точками лежит на прямой; это означает, что длина прямого пути по определению не может представлять собой переоценку длины пути. На рис. 4.2 показан процесс поиска  $A^*$  пути в Бухарест с помощью дерева. Значения  $g$  вычисляются на основании стоимостей этапов, показанных на рис. 3.1, а значения  $h_{SLD}$  приведены в табл. 4.1. Следует, в частности, отметить, что узел *Bucharest* впервые появляется в периферии на этапе, показанном на рис. 4.2,  $d$ , но не выбирается для развертывания, поскольку его  $f$ -стоимость (450) выше, чем стоимость узла *Pitesti* (417). Иными словами эту ситуацию можно описать так, что может существовать решение, при котором путь проходит через город Питешти со стоимостью, достигающей 417, поэтому алгоритм не останавливается на решении со стоимостью 450. Данный пример может служить общим свидетельством того, что поиск  $A^*$  с использованием алгоритма Tree-Search является оптимальным, если функция  $h(n)$  допустима. Предположим, что на периферии поиска появился неоптимальный целевой узел  $G_2$ , а стоимость оптимального решения равна  $C^*$ . В таком случае, поскольку узел  $G_2$  неоптимален, а  $h(G_2) = 0$  (это выражение справедливо для любого целевого узла), можно вывести следующую формулу:

$$f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^*$$

Теперь рассмотрим периферийный узел  $n$ , который находится в оптимальном пути решения, например узел *Pitesti* в примере, приведенном в предыдущем абзаце. (Если решение существует, то всегда должен быть такой узел.) Если функция  $h(n)$  не переоценивает стоимость завершения этого пути решения, то справедлива следующая формула:

$$f(n) = g(n) + h(n) \leq C^*$$

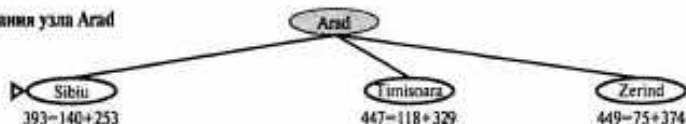
Таким образом, доказано, что  $f(n) \leq C^* < f(G_2)$ , поэтому узел  $G_2$  не развертывается и поиск  $A^*$  должен вернуть оптимальное решение.

Если бы вместо алгоритма Tree-Search использовался алгоритм Graph-Search, приведенный в листинге 3.6, то данное доказательство стало бы недействительным. Дело в том, что алгоритм Graph-Search способен отбросить оптимальный путь к повторяющемуся состоянию, если он не был сформирован в первую очередь, поэтому может возвращать неоптимальные решения (см. упр. 4.4). Существуют два способа устранения этого недостатка. Первое решение состоит в том, что алгоритм Graph-Search должен быть дополнен так, чтобы он отбрасывал наиболее дорогостоящий из любых двух найденных путей к одному и тому же узлу (см. обсуждение этой темы в разделе 3.5). Сопровождение необходимой для этого дополнительной информации связано с определенными трудностями, но гарантирует оптимальность. Второе решение состоит в обеспечении того, чтобы оптимальный путь к любому повторяющемуся состоянию всегда был первым из тех, по которым следует алгоритм, как в случае поиска по критерию стоимости.

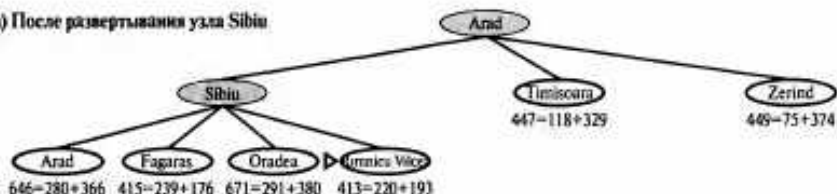
а) Начальное состояние



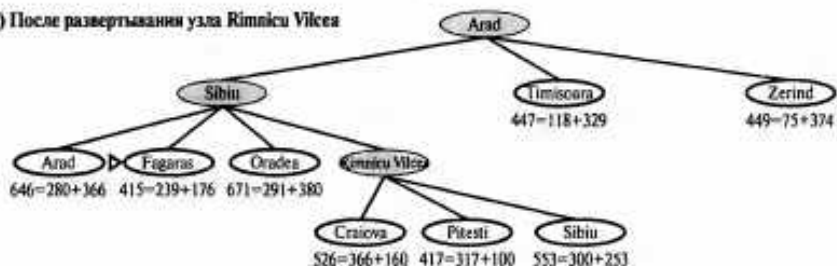
б) После развертывания узла Arad



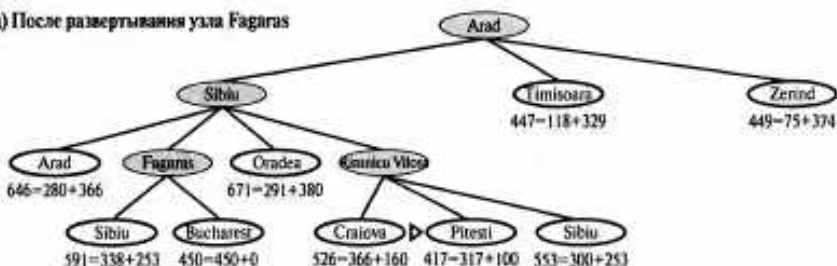
в) После развертывания узла Sibiu



г) После развертывания узла Rimnicu Vilcea



д) После развертывания узла Fagaras



е) После развертывания узла Pitesti

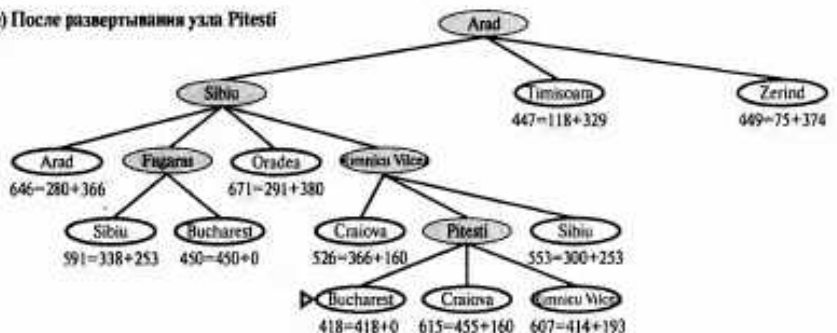


Рис. 4.2. Этапы поиска  $A^*$  пути в Бухарест. Узлы отмечены значениями  $f = g + h$ . Значения  $h$  представляют собой расстояния по прямой до Бухареста, приведенные в табл. 4.1

Такое свойство соблюдается, если на функцию  $h(n)$  налагается дополнительное требование, а именно требование обеспечения  $\nless$  **преемственности** эвристической функции (такое свойство называют также  $\nless$  **монотонностью** эвристической функции). Эвристическая функция  $h(n)$  является преемственной, если для любого узла  $n$  и для любого преемника  $n'$  узла  $n$ , сформированного в результате любого действия  $a$ , оценка стоимости достижения цели из узла  $n$  не больше чем стоимость этапа достижения узла  $n'$  плюс оценка стоимости достижения цели из узла  $n'$ :

$$h(n) \leq c(n, a, n') + h(n')$$

Это — форма общего  $\nless$  **неравенства треугольника**, которое указывает, что длина любой стороны треугольника не может превышать сумму длин двух других сторон. В данном случае треугольник образован узлами  $n$ ,  $n'$  и целью, ближайшей к  $n$ . Можно довольно легко показать (упр. 4.7), что любая преемственная эвристическая функция является также допустимой. Наиболее важным следствием из определения преемственности является такой вывод:  $\text{☞}$  *поиск  $A^*$  с использованием алгоритма Graph-Search является оптимальным, если функция  $h(n)$  преемственна.*

Несмотря на то что требование к преемственности является более строгим, чем требование к допустимости, весьма нелегко составить такие эвристические функции, которые были бы допустимыми, но не преемственными. Все допустимые эвристические функции, рассматриваемые в данной главе, являются также преемственными. Возьмем в качестве примера функцию  $h_{\text{SLD}}$ . Известно, что общее неравенство треугольника удовлетворяется, если длина каждой стороны измеряется с помощью расстояния по прямой, и что расстояние по прямой между  $n$  и  $n'$  не больше чем  $c(n, a, n')$ . Поэтому эвристическая функция  $h_{\text{SLD}}$  является преемственной.

Еще один важный вывод из определения преемственности является таковым:  $\text{☞}$  *если функция  $h(n)$  преемственна, то значения функции  $f(n)$  вдоль любого пути являются неубывающими.* Доказательство этого утверждения непосредственно вытекает из определения преемственности. Предположим, что узел  $n'$  — преемник узла  $n$ ; в таком случае для некоторого  $a$  справедливо выражение  $g(n') = g(n) + c(n, a, n')$  и имеет место такая формула:

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$$

На основании этого можно сделать вывод, что последовательность узлов, развернутых в поиске  $A^*$  с использованием алгоритма Graph-Search, находится в неубывающем порядке значений  $f(n)$ . Поэтому первый целевой узел, выбранный для развертывания, должен представлять собой оптимальное решение, поскольку все дальнейшие узлы будут, по меньшей мере, столь же дорогостоящими.

Тот факт, что  $f$ -стоимости вдоль любого пути являются неубывающими, означает также, что могут быть очерчены  $\nless$  **контуры** равных  $f$ -стоимостей в пространстве состояний, полностью аналогичные контурам равных высот на топографической карте. Пример подобной схемы приведен на рис. 4.3. Внутри контура, обозначенного как 400, все узлы имеют значения  $f(n)$ , меньшие или равные 400, и т.д. В таком случае, поскольку в поиске  $A^*$  развертывается периферийный узел с наименьшей  $f$ -стоимостью, можно видеть, как поиск  $A^*$  распространяется из начального узла, добывая узлы в виде концентрических полос с возрастающей  $f$ -стоимостью.

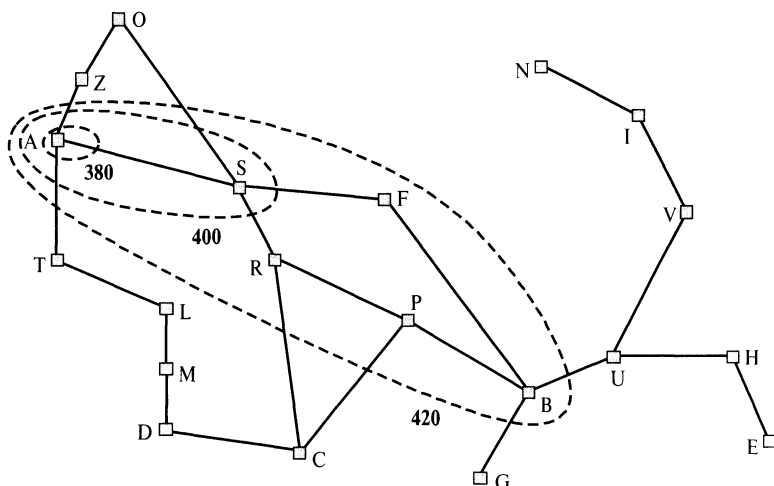


Рис. 4.3. Карта Румынии, на которой показаны контуры, соответствующие  $f=380$ ,  $f=400$  и  $f=420$ , причем  $A$  является начальным состоянием. Узлы в пределах данного конкретного контура имеют  $f$ -стоимости, меньшие или равные значению стоимости контура

При поиске по критерию стоимости (таковым является поиск  $A^*$  с применением  $h(n)=0$ ) эти полосы будут представлять собой “кольца” с центром в начальном состоянии. При использовании более точных эвристических функций полосы вытягиваются в направлении целевого состояния и становятся более узко сосредоточенными вокруг оптимального пути. Если  $C^*$  представляет собой стоимость оптимального пути решения, то можно утверждать следующее:

- в поиске  $A^*$  разворачиваются все узлы со значениями  $f(n) < C^*$ ;
- поэтому в поиске  $A^*$  могут разворачиваться некоторые дополнительные узлы, находящиеся непосредственно на “целевом контуре” (где  $f(n) = C^*$ ), прежде чем будет выбран целевой узел.

На интуитивном уровне представляется очевидным, что первое найденное решение должно быть оптимальным, поскольку целевые узлы во всех последующих контурах будут иметь более высокое значение  $f$ -стоимости и поэтому более высокое значение  $g$ -стоимости (поскольку все целевые узлы имеют значения  $h(n)=0$ ). Кроме того, на интуитивном уровне также очевидно, что поиск  $A^*$  является полным. По мере добавления полос с возрастающими значениями  $f$  мы должны в конечном итоге достичь полосы, в которой значение  $f$  будет равно стоимости пути к целевому состоянию<sup>4</sup>.

Следует отметить, что в поиске  $A^*$  узлы со значением  $f(n) > C^*$  не разворачиваются; например, как показано на рис. 4.2, не разворачивается узел *Timisoara*, даже несмотря на то, что является дочерним узлом корневого узла. Эту ситуацию принято обозначать так, что происходит **отсечение** поддерева, находящегося ниже узла

<sup>4</sup> Для соблюдения требования полноты необходимо, чтобы количество узлов со стоимостью, меньшей или равной  $C^*$ , было конечным; это условие соблюдается, если стоимости всех этапов превышают некоторое конечное значение  $\epsilon$ , а коэффициент ветвления  $b$  является конечным.

*Timisoara*; поскольку функция  $h_{SLD}$  является допустимой, рассматриваемый алгоритм может безопасно игнорировать это поддереву, гарантируя вместе с тем оптимальность. Понятие *отсечения* (под которым подразумевается исключение из рассмотрения некоторых вариантов в связи с отсутствием необходимости их исследовать) является важным для многих областей искусственного интеллекта.

Одно заключительное наблюдение состоит в том, что среди оптимальных алгоритмов такого типа (алгоритмов, которые разворачивают пути поиска от корня) поиск  $A^*$  является ~~на~~ **оптимально эффективным** для любой конкретной эвристической функции. Это означает, что не гарантируется разворачивание меньшего количества узлов, чем в поиске  $A^*$ , с помощью какого-либо иного оптимального алгоритма (не считая той возможности, когда осуществляется выбор на равных среди узлов с  $f(n) = C^*$ ). Это связано с тем, что любой алгоритм, который не разворачивает все узлы со значениями  $f(n) < C^*$ , подвержен риску потери оптимального решения.

Те соображения, что поиск  $A^*$ , как один из всех подобных алгоритмов, является действительно полным, оптимальным и оптимально эффективным, оставляют довольно приятное впечатление. Но, к сожалению, это отнюдь не означает, что поиск  $A^*$  может служить ответом на все наши потребности в поиске. Сложность заключается в том, что при решении большинства задач количество узлов в пределах целевого контура пространства состояний все еще зависит экспоненциально от длины решения. Хотя доказательство этого утверждения выходит за рамки настоящей книги, было показано, что экспоненциальный рост происходит, если ошибка эвристической функции растет не быстрее по сравнению с логарифмом фактической стоимости пути. В математических обозначениях условие субэкспоненциального роста состоит в следующем:

$$|h(n) - h^*(n)| \leq O(\log h^*(n))$$

где  $h^*(n)$  — истинная стоимость достижения цели из узла  $n$ . Почти для всех практически применяемых эвристических функций эта ошибка, по меньшей мере, пропорциональна стоимости пути, и происходящий в связи с этим экспоненциальный рост в конечном итоге превосходит возможности любого компьютера. По этой причине на практике стремление находить оптимальное решение часто не оправдано. Иногда вместо этого целесообразно использовать варианты поиска  $A^*$ , позволяющие быстро находить неоптимальные решения, а в других случаях — разрабатывать эвристические функции, которые являются более точными, но не строго допустимыми. В любом случае применение хорошей эвристической функции все равно обеспечивает поразительную экономию усилий по сравнению с использованием неинформированного поиска. Вопрос разработки хороших эвристических функций рассматривается в разделе 4.2.

Но большая продолжительность вычислений не является основным недостатком поиска  $A^*$ . Поскольку при поиске  $A^*$  все сформированные узлы хранятся в памяти (как и во всех алгоритмах Graph-Search), фактически ресурсы пространства исчерпываются задолго до того, как исчерпываются ресурсы времени. По этой причине поиск  $A^*$  не является практически применимым при решении многих крупномасштабных задач. Разработанные недавно алгоритмы позволяют преодолеть эту проблему пространства, не жертвуя оптимальностью или полнотой, за счет небольшого увеличения времени выполнения. Эти алгоритмы рассматриваются ниже.

### Эвристический поиск с ограничением объема памяти

Простейший способ сокращения потребностей в памяти для поиска  $A^*$  состоит в применении идеи итеративного углубления в контексте эвристического поиска. Реализация этой идеи привела к созданию алгоритма  $A^*$  с итеративным углублением (Iterative-Deepening  $A^*$  — IDA\*). Основное различие между алгоритмом IDA\* и стандартным алгоритмом итеративного углубления состоит в том, что применяемым условием останова развертывания служит  $f$ -стоимость ( $g+h$ ), а не глубина; на каждой итерации этим остановочным значением является минимальная  $f$ -стоимость любого узла, превышающая остановочное значение, достигнутое в предыдущей итерации. Алгоритм IDA\* является практически применимым для решения многих задач с единичными стоимостями этапов и позволяет избежать существенных издержек, связанных с поддержкой отсортированной очереди узлов. К сожалению, этот алгоритм характеризуется такими же сложностями, связанными с использованием стоимостей с действительными значениями, как и итеративная версия поиска по критерию стоимости, которая описана в упр. 3.11. В данном разделе кратко рассматриваются два более современных алгоритма с ограничением памяти, получивших названия RBFS и MA\*.

➤ **Рекурсивный поиск по первому наилучшему совпадению** (Recursive Best-First Search — RBFS) — это простой рекурсивный алгоритм, в котором предпринимаются попытки имитировать работу стандартного поиска по первому наилучшему совпадению, но с использованием только линейного пространства. Этот алгоритм приведен в листинге 4.1. Он имеет структуру, аналогичную структуре рекурсивного поиска в глубину, но вместо бесконечного следования вниз по текущему пути данный алгоритм контролирует  $f$ -значение наилучшего альтернативного пути, доступного из любого предка текущего узла. Если текущий узел превышает данный предел, то текущий этап рекурсии отменяется и рекурсия продолжается с альтернативного пути. После отмены данного этапа рекурсии в алгоритме RBFS происходит замена  $f$ -значения каждого узла вдоль данного пути наилучшим  $f$ -значением его дочернего узла. Благодаря этому в алгоритме RBFS запоминается  $f$ -значение наилучшего листового узла из забытого поддерева и поэтому в некоторый последующий момент времени может быть принято решение о том, стоит ли снова развертывать это поддерево. На рис. 4.4 показано, как с помощью алгоритма RBFS происходит поиск пути в Бухарест.

#### Листинг 4.1. Алгоритм рекурсивного поиска по первому наилучшему совпадению

---

```

function Recursive-Best-First-Search(problem) returns решение result
    или индикатор неудачи failure
    RBFS(problem, Make-Node(Initial-State[problem]),  $\infty$ )

function RBFS(problem, node, f_limit) returns решение result
    или индикатор неудачи failure и новый предел  $f$ -стоимости f_limit
    if Goal-Test[problem](State[node]) then return узел node
    successors  $\leftarrow$  Expand(node, problem)
    if множество узлов-преемников successors пусто
        then return failure,  $\infty$ 
    for each s in successors do
         $f[s] \leftarrow \max(g(s)+h(s), f[node])$ 
    repeat
        best  $\leftarrow$  узел с наименьшим  $f$ -значением в множестве successors
        if  $f[best] > f\_limit$  then return failure,  $f[best]$ 

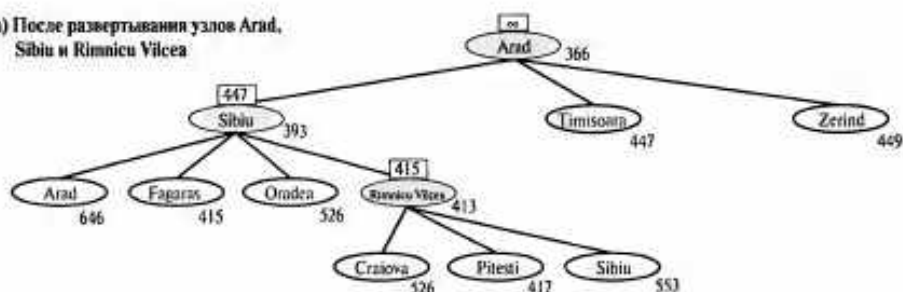
```

```

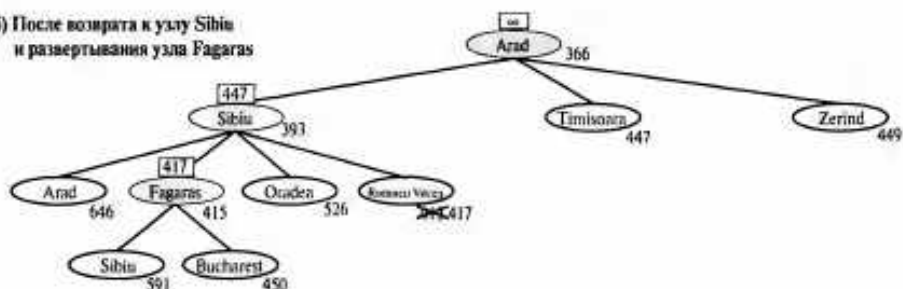
alternative ← второе после наименьшего f-значение
                в множестве successors
result, f[best] ← RBFS(problem, best,
                        min(f_limit, alternative))
if result ≠ failure then return result

```

а) После развертывания узлов Arad, Sibiu и Rimnicu Vilcea



б) После возврата к узлу Sibiu и развертывания узла Fagaras



в) После переключения снова на узел Rimnicu Vilcea и развертывания узла Pitesti

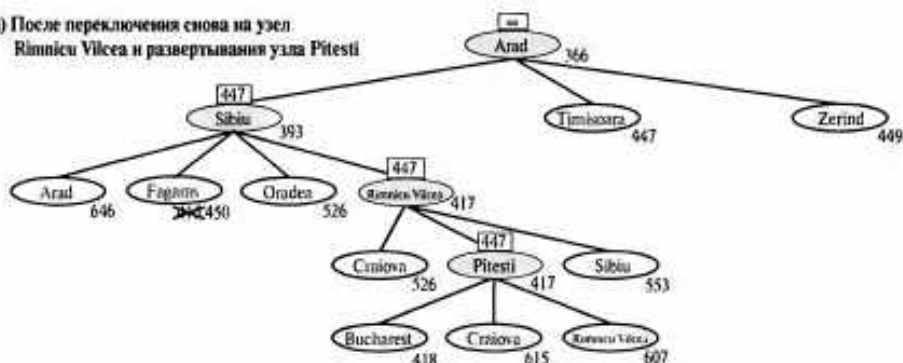


Рис. 4.4. Этапы поиска кратчайшего маршрута в Бухарест с помощью алгоритма RBFS. Значение  $f$ -предела для каждого рекурсивного вызова показано над каждым текущим узлом; путь через узел Rimnicu Vilcea, который следует до текущего наилучшего листового узла (Pitesti), имеет значение, худшее по сравнению с наилучшим альтернативным путем (Fagaras) (а); рекурсия продолжается и значение наилучшего листового узла забытого поддерева (417) резервируется в узле Rimnicu Vilcea; затем развертывается узел Fagaras, в результате чего обнаруживается наилучшее значение листового узла, равное 450 (б); рекурсия продолжается и значение наилучшего листового узла забытого поддерева (450) резервируется в узле Fagaras; затем развертывается узел Rimnicu Vilcea; на этот раз развертывание продолжается в сторону Бухареста, поскольку наилучший альтернативный путь (через узел Timisoara) стоит, по меньшей мере, 447 (в)



Алгоритм RBFS является немного более эффективным по сравнению с IDA\*, но все еще страдает от недостатка, связанного со слишком частым повторным формированием узлов. В примере, приведенном на рис. 4.4, алгоритм RBFS вначале следует по пути через узел *RimnicuVilcea*, затем “меняет решение” и пытается пройти через узел *Fagaras*, после этого снова возвращается к отвергнутому ранее решению. Такие смены решения происходят в связи с тем, что при каждом развертывании текущего наилучшего пути велика вероятность того, что его  $f$ -значение увеличится, поскольку функция  $h$  обычно становится менее оптимистической по мере того, как происходит развертывание узлов, более близких к цели. После того как возникает такая ситуация, особенно в больших пространствах поиска, путь, который был вторым после наилучшего, может сам стать наилучшим путем, поэтому в алгоритме поиска приходится выполнять возврат, чтобы проследовать по нему. Каждое изменение решения соответствует одной итерации алгоритма IDA\* и может потребовать многих повторных развертываний забытых узлов для воссоздания наилучшего пути и развертывания пути еще на один узел.

Как и алгоритм поиска A\*, RBFS является оптимальным алгоритмом, если эвристическая функция  $h(n)$  допустима. Его пространственная сложность равна  $O(bd)$ , но охарактеризовать временную сложность довольно трудно: она зависит и от точности эвристической функции, и от того, насколько часто происходила смена наилучшего пути по мере развертывания узлов. И алгоритм IDA\*, и алгоритм RBFS подвержены потенциальному экспоненциальному увеличению сложности, связанной с поиском в графах (см. раздел 3.5), поскольку эти алгоритмы не позволяют определять наличие повторяющихся состояний, отличных от тех, которые находятся в текущем пути. Поэтому данные алгоритмы способны много раз исследовать одно и то же состояние.

Алгоритмы IDA\* и RBFS страдают от того недостатка, что в них используется слишком мало памяти. Между итерациями алгоритм IDA\* сохраняет только единственное число — текущий предел  $f$ -стоимости. Алгоритм RBFS сохраняет в памяти больше информации, но количество используемой в нем памяти измеряется лишь значением  $O(bd)$ : даже если бы было доступно больше памяти, алгоритм RBFS не способен ею воспользоваться.

Поэтому представляется более разумным использование всей доступной памяти. Двумя алгоритмами, которые осуществляют это требование, являются поиск  $\approx$  MA\* (Memory-bounded A\* — поиск A\* с ограничением памяти) и  $\approx$  SMA\* (Simplified MA\* — упрощенный поиск MA\*). В данном разделе будет описан алгоритм SMA\*, который действительно является более простым, чем другие алгоритмы этого типа. Алгоритм SMA\* действует полностью аналогично поиску A\*, развертывая наилучшие листовые узлы до тех пор, пока не будет исчерпана доступная память. С этого момента он не может добавить новый узел к дереву поиска, не уничтожив старый. В алгоритме SMA\* всегда уничтожается наихудший листовой узел (тот, который имеет наибольшее  $f$ -значение). Как и в алгоритме RBFS, после этого в алгоритме SMA\* значение забытого (уничтоженного) узла резервируется в его родительском узле. Благодаря этому предок забытого поддерева позволяет определить качество наилучшего пути в этом поддереве. Поскольку имеется данная информация, в алгоритме SMA\* поддерево восстанавливается, только если обнаруживается, что все другие пути выглядят менее многообещающими по сравнению с забытым путем. Иными словами, если все потомки узла  $n$  забыты, то неизвестно, каким

путем можно следовать от  $l$ , но все еще можно получить представление о том, есть ли смысл куда-либо следовать от  $l$ .

Полный алгоритм слишком сложен для того, чтобы его можно было воспроизвести в данной книге<sup>5</sup>, но заслуживает упоминания один его нюанс. Как уже было сказано выше, в алгоритме SMA\* разворачивается наилучший листовой узел и удаляется наихудший листовой узел. А что происходит, если все листовые узлы имеют одинаковое  $f$ -значение? В таком случае может оказаться, что алгоритм выбирает для удаления и разворачивания один и тот же узел. В алгоритме SMA\* эта проблема решается путем разворачивания самого нового наилучшего листового узла и удаления самого старого наихудшего листового узла. Эти два узла могут оказаться одним и тем же узлом, только если существует лишь один листовой узел; в таком случае текущее дерево поиска должно представлять собой единственный путь от корня до листового узла, заполняющий всю память. Это означает, что если данный листовой узел не является целевым узлом, то решение не достижимо при доступном объеме памяти, даже если этот узел находится в оптимальном пути решения. Поэтому такой узел может быть отброшен точно так же, как и в том случае, если он не имеет преемников.

Алгоритм SMA\* является полным, если существует какое-либо достижимое решение, иными словами, если  $d$ , глубина самого поверхностного целевого узла, меньше чем объем памяти (выраженный в хранимых узлах). Этот алгоритм оптимален, если достижимо какое-либо оптимальное решение; в противном случае он возвращает наилучшее достижимое решение. С точки зрения практики алгоритм SMA\* вполне может оказаться наилучшим алгоритмом общего назначения для поиска оптимальных решений, особенно если пространство состояний представляет собой граф, стоимости этапов не одинаковы, а операция формирования узлов является более дорогостоящей в сравнении с дополнительными издержками сопровождения открытых и закрытых списков.

Однако при решении очень сложных задач часто возникают ситуации, в которых алгоритм SMA\* вынужден постоянно переключаться с одного пути решения на другой в пределах множества возможных путей решения, притом что в памяти может поместиться только небольшое подмножество этого множества. (Такие ситуации напоминают проблему **пробуксовки** в системах подкачки страниц с жесткого диска.) В таком случае на повторное формирование одних и тех узлов затрачивается дополнительное время, а это означает, что задачи, которые были бы фактически разрешимыми с помощью поиска A\* при наличии неограниченной памяти, становятся трудноразрешимыми для алгоритма SMA\*. Иными словами, из-за **ограничений в объеме памяти некоторые задачи могут становиться трудноразрешимыми с точки зрения времени вычисления**. Хотя отсутствует теория, позволяющая найти компромисс между затратами времени и памяти, создается впечатление, что зачастую избежать возникновения этой проблемы невозможно. Единственным способом преодоления такой ситуации становится частичный отказ от требований к оптимальности решения.

## Обучение лучшим способам поиска

Выше было представлено несколько стратегий поиска (поиск в ширину, жадный поиск по первому наилучшему совпадению и т.д.), которые были разработаны уче-

<sup>5</sup> Грубый набросок этого алгоритма был приведен в первом издании настоящей книги.

ными и специалистами по компьютерным наукам. Но может ли сам агент обучаться лучшим способам поиска? Ответ на этот вопрос является положительным, а применяемый при этом метод обучения опирается на важную концепцию, называемую *пространством состояний*, рассматриваемым на *метауровне*, или **метауровневым пространством состояний**. Каждое состояние в метауровневом пространстве состояний отражает внутреннее (вычислительное) состояние программы, выполняющей поиск в пространстве состояний, рассматриваемом на уровне объектов, или в **объектно-уровневом пространстве состояний**, таком как карта Румынии. Например, внутреннее состояние алгоритма  $A^*$  включает в себя текущее дерево поиска. Каждое действие в метауровневом пространстве состояний представляет собой этап вычисления, который изменяет внутреннее состояние, например, на каждом этапе вычисления в процессе поиска  $A^*$  разворачивается один из листовых узлов, а его преемники добавляются к дереву. Таким образом, рис. 4.2, на котором показана последовательность все больших и больших деревьев поиска, может рассматриваться как изображающий путь в метауровневом пространстве состояний, где каждое состояние в пути является объектно-уровневым деревом поиска.

В настоящее время путь, показанный на рис. 4.2, имеет пять этапов, включая один этап (разворачивание узла *Fagaras*), который нельзя назвать слишком полезным. Может оказаться, что при решении более сложных задач количество подобных ненужных этапов будет намного больше, а алгоритм **метауровневого обучения** может изучать этот опыт, чтобы в дальнейшем избегать исследования бесперспективных поддеревьев. Методы, используемые при обучении такого рода, описаны в главе 21. Целью обучения является минимизация **суммарной стоимости** решения задач, а также поиск компромисса между вычислительными издержками и стоимостью пути.

## 4.2. ЭВРИСТИЧЕСКИЕ ФУНКЦИИ

В этом разделе будут рассматриваться эвристические функции для задачи игры в восемь, что позволяет лучше продемонстрировать характерные особенности всех эвристических функций в целом.

Головоломка “игра в восемь” была одной из первых задач эвристического поиска. Как было указано в разделе 3.2, в ходе решения этой головоломки требуется передвигать фишки по горизонтали или по вертикали на пустой участок до тех пор, пока полученная конфигурация не будет соответствовать целевой конфигурации (рис. 4.5).

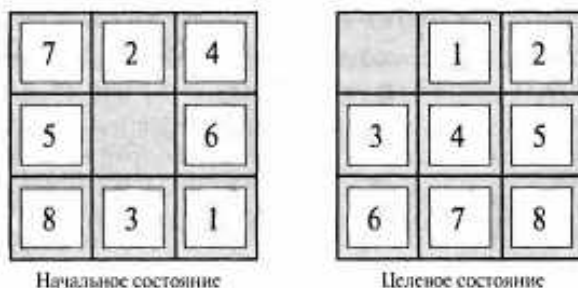


Рис. 4.5. Типичный экземпляр головоломки “игра в восемь”. Решение имеет длину 26 этапов

Средняя стоимость решения для сформированного случайным образом экземпляра головоломки “игра в восемь” составляет около 22 этапов. Коэффициент ветвления примерно равен 3. (Если пустой квадрат находится в середине коробки, то количество возможных ходов равно четырем, если находится в углу — двум, а если в середине одной из сторон — трем.) Это означает, что при исчерпывающем поиске на глубину 22 приходится рассматривать примерно  $3^{22} \approx 3.1 \times 10^{10}$  состояний. Отслеживая повторяющиеся состояния, это количество состояний можно сократить приблизительно в 170 000 раз, поскольку существует только  $9! / 2 = 181\,440$  различных состояний, которые являются достижимыми (см. упр. 3.4.) Это количество состояний уже лучше поддается контролю, но соответствующее количество для игры в пятнадцать примерно равно  $10^{13}$ , поэтому для такой головоломки с более высокой сложностью требуется найти хорошую эвристическую функцию. Если нужно находить кратчайшие решения с использованием поиска  $A^*$ , то требуется эвристическая функция, которая никогда не переоценивает количество этапов достижения цели. История исследований в области поиска таких эвристических функций для игры в пятнадцать является довольно долгой, а в данном разделе рассматриваются два широко используемых кандидата на эту роль, которые описаны ниже.

- $h_1$  = количество фишек, стоящих не на своем месте. На рис. 4.5 все восемь фишек стоят не на своем месте, поэтому показанное слева начальное состояние имеет эвристическую оценку  $h_1=8$ . Эвристическая функция  $h_1$  является допустимой, поскольку очевидно, что каждую фишку, находящуюся не на своем месте, необходимо переместить по меньшей мере один раз.
- $h_2$  = сумма расстояний всех фишек от их целевых позиций. Поскольку фишки не могут передвигаться по диагонали, рассчитываемое расстояние представляет собой сумму горизонтальных и вертикальных расстояний. Такое расстояние иногда называют **расстоянием, измеряемым в городских кварталах**, или **манхэттенским расстоянием**. Эвристическая функция  $h_2$  также является допустимой, поскольку все, что может быть сделано в одном ходе, состоит лишь в перемещении одной фишки на один этап ближе к цели. Фишки от 1 до 8 в рассматриваемом начальном состоянии соответствуют такому значению манхэттенского расстояния:  $h_2=3+1+2+2+2+3+3+2=18$ .

Как и можно было предположить, ни одна из этих функций не переоценивает истинную стоимость решения, которая равна 26.

### Зависимость производительности поиска от точности эвристической функции

Одним из критериев, позволяющих охарактеризовать качество эвристической функции, является **эффективный коэффициент ветвления  $b^*$** . Если общее количество узлов, вырабатываемых в процессе поиска  $A^*$  решения конкретной задачи, равно  $N$ , а глубина решения равна  $d$ , то  $b^*$  представляет собой коэффициент ветвления, который должно иметь однородное дерево с глубиной  $d$  для того, чтобы в нем содержалось  $N+1$  узлов. Поэтому справедлива следующая формула:

$$N + 1 = 1 + b^* + (b^*)^2 + \dots + (b^*)^d$$

Например, если алгоритм  $A^*$  находит решение на глубине 5 с использованием 52 узлов, то эффективный коэффициент ветвления равен 1,92. Эффективный коэффи-

циент ветвления может изменяться от одного экземпляра одной и той же задачи к другому, но обычно в случае достаточно трудных задач остается относительно постоянным. Поэтому экспериментальные измерения коэффициента  $b^*$  на небольшом множестве задач могут служить хорошим критерием общей полезности рассматриваемой эвристической функции. Хорошо спроектированная эвристическая функция должна иметь значение  $b^*$ , близкое к 1, что позволяет быстро решать довольно большие задачи.

Для проверки эвристических функций  $h_1$  и  $h_2$  авторы сформировали случайным образом 1200 экземпляров задачи с длиной решения от 2 до 24 (по 100 экземпляров для каждого четного значения длины) и нашли их решения с помощью поиска с итеративным углублением и поиска в дереве по алгоритму  $A^*$  с применением эвристических функций  $h_1$  и  $h_2$ . Данные о среднем количестве узлов, развернутых при использовании каждой стратегии и эффективном коэффициенте ветвления, приведены в табл. 4.2. Эти результаты показывают, что эвристическая функция  $h_2$  лучше чем  $h_1$  и намного лучше по сравнению с использованием поиска с итеративным углублением. Применительно к найденным авторами решениям с длиной 14 применение поиска  $A^*$  с эвристической функцией  $h_2$  становится в 30 000 раз более эффективным по сравнению с неинформированным поиском с итеративным углублением.

Таблица 4.2. Сравнение значений стоимости поиска и эффективного коэффициента ветвления для алгоритмов *Iterative-Deepening-Search* и  $A^*$  с  $h_1$ ,  $h_2$ . Данные усреднялись по 100 экземплярам задачи игры в восемь применительно к различным значениям длины решения

$d$	Стоимость поиска			Эффективный коэффициент ветвления		
	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2,45	1,79	1,79
4	112	13	12	2,87	1,48	1,45
6	680	20	18	2,73	1,34	1,30
8	6384	39	25	2,80	1,33	1,24
10	47127	93	39	2,79	1,38	1,22
12	3644035	227	73	2,78	1,42	1,24
14	—	539	113	—	1,44	1,23
16	—	1301	211	—	1,45	1,25
18	—	3056	363	—	1,46	1,26
20	—	7276	676	—	1,47	1,27
22	—	18094	1219	—	1,48	1,28
24	—	39135	1641	—	1,48	1,26

Интерес представляет вопрос о том, всегда ли эвристическая функция  $h_2$  лучше чем  $h_1$ . Ответ на этот вопрос является положительным. На основании определений этих двух эвристических функций можно легко прийти к выводу, что для любого узла  $n$  справедливо выражение  $h_2(n) \geq h_1(n)$ . Таким образом, можно утверждать, что эвристика  $h_2$  доминирует над  $h_1$ . Доминирование напрямую связано с эффективностью: при поиске  $A^*$  с использованием функции  $h_2$  никогда не происходит развертывание большего количества узлов, чем при поиске  $A^*$  с использованием  $h_1$  (возможно, за исключением нескольких узлов с  $f(n) = C^*$ ). Доказательство этого ут-

верждения является несложным. Напомним приведенное на с. 161 замечание, что каждый узел со значением  $f(n) < C^*$  наверняка должен быть развернут. Это аналогично утверждению, что должен быть наверняка развернут каждый узел со значением  $h(n) < C^* - g(n)$ . Но поскольку для всех узлов значение  $h_2$ , по крайней мере, не меньше значения  $h_1$ , то каждый узел, который должен быть наверняка развернут в поиске  $A^*$  с  $h_2$ , будет также наверняка развернут при поиске с  $h_1$ , а применение эвристической функции  $h_1$  может к тому же вызвать и развертывание других узлов. Поэтому всегда лучше использовать эвристическую функцию с более высокими значениями, при тех условиях, что эта функция не переоценивает длину пути решения и что время вычисления этой эвристической функции не слишком велико.

### Составление допустимых эвристических функций

Выше было показано, что эвристические функции  $h_1$  (в которой используется количество стоящих не на своем месте фишек) и  $h_2$  (в которой используется манхэттенское расстояние) являются довольно неплохими эвристическими функциями для задачи игры в восемь и что функция  $h_2$  — из них наилучшая. Но на основании чего именно была предложена функция  $h_2$ ? Возможно ли, чтобы компьютер мог составить некоторую эвристическую функцию механически?

Эвристические функции  $h_1$  и  $h_2$  представляют собой оценки оставшейся длины пути для задачи игры в восемь, но они, кроме того, возвращают идеально точные значения длины пути для упрощенных версий этой игры. Если бы правила игры в восемь изменились таким образом, чтобы любую фишку можно было передвигать куда угодно, а не только на соседний пустой квадрат, то эвристическая функция  $h_1$  возвращала бы точное количество этапов в кратчайшем решении. Аналогичным образом, если бы любую фишку можно было перемещать на один квадрат в любом направлении, даже на занятый квадрат, то точное количество этапов в кратчайшем решении возвращала бы эвристическая функция  $h_2$ . Задача с меньшим количеством ограничений на возможные действия называется **ослабленной задачей**. *Стоимость оптимального решения ослабленной задачи представляет собой допустимую эвристику для первоначальной задачи.* Такая эвристическая функция является допустимой, поскольку оптимальное решение первоначальной задачи, по определению, служит также решением ослабленной задачи и поэтому должно быть, по меньшей мере, таким же дорогостоящим, как и оптимальное решение ослабленной задачи. Поскольку значение производной эвристической функции представляет собой точную стоимость решения ослабленной задачи, эта функция должно подчиняться неравенству треугольника и поэтому должна быть **преемственной** (см. с. 160).

Если определение задачи записано на каком-то формальном языке, то существует возможность формировать ослабленные задачи автоматически<sup>6</sup>. Например, если действия в игре в восемь описаны следующим образом:

<sup>6</sup> В главах 8 и 11 будут описаны формальные языки, подходящие для этого назначения; возможность автоматизировать формирование ослабленных задач появляется при наличии формальных описаний, с которыми могут проводиться манипуляции, а пока для этого назначения будет использоваться естественный язык.

Фишка может быть передвинута из квадрата  $A$  в квадрат  $B$ , **если** квадрат  $A$  является смежным по горизонтали или по вертикали с квадратом  $B$  и квадрат  $B$  пуст

то могут быть сформированы три ослабленные задачи путем удаления одного или обоих из приведенных выше условий.

- а) Фишка может быть передвинута из квадрата  $A$  в квадрат  $B$ , если квадрат  $A$  является смежным с квадратом  $B$ .
- б) Фишка может быть передвинута из квадрата  $A$  в квадрат  $B$ , если квадрат  $B$  пуст.
- в) Фишка может быть передвинута из квадрата  $A$  в квадрат  $B$ .

На основании ослабленной задачи а) можно вывести функцию  $h_2$  (манхэттенское расстояние). В основе этих рассуждений лежит то, что  $h_2$  должна представлять собой правильную оценку, если каждая фишка передвигается к месту ее назначения по очереди. Эвристическая функция, полученная на основании ослабленной задачи б), обсуждается в упр. 4.9. На основании ослабленной задачи в) можно получить эвристическую функцию  $h_1$  (фишки, стоящие не на своих местах), поскольку эта оценка была бы правильной, если бы фишки можно было передвигать в предназначенное для них место за один этап. Обратите внимание на то, что здесь существенной является возможность решать ослабленные задачи, создаваемые с помощью указанного метода, фактически без какого-либо поиска, поскольку ослабленные правила обеспечивают декомпозицию этой задачи на восемь независимых подзадач. Если бы было трудно решать и ослабленную задачу, то был бы дорогостоящим сам процесс получения значений соответствующей эвристической функции<sup>7</sup>.

Для автоматического формирования эвристических функций на основе определений задач с использованием метода “ослабленной задачи” и некоторых других методов может применяться программа под названием Absolver [1237]. Программа Absolver составила для задачи игры в восемь новую эвристическую функцию, лучшую по сравнению со всеми существовавшими ранее эвристическими функциями, а также нашла первую полезную эвристическую функцию для знаменитой головоломки “кубик Рубика”.

Одна из проблем, возникающих при составлении новых эвристических функций, состоит в том, что часто не удается получить эвристическую функцию, которая была бы “лучшей во всех отношениях” по сравнению с другими. Если для решения какой-либо задачи может применяться целая коллекция допустимых эвристических функций  $h_1 \dots h_m$  и ни одна из них не доминирует над какой-либо из других функций, то какая из этих функций должна быть выбрана? Оказалось, что такой выбор делать не требуется, поскольку можно взять от них всех самое лучшее, определив такой критерий выбора:

$$h(n) = \max\{h_1(n), \dots, h_m(n)\}$$

В этой составной эвристике используется та функция, которая является наиболее точной для рассматриваемого узла. Поскольку эвристические функции, входящие в состав эвристики  $h$ , являются допустимыми, сама эта функция также является до-

---

<sup>7</sup> Следует отметить, что идеальную эвристическую функцию можно также получить, просто разрешив функции  $h$  “тайком” выполнять полный поиск в ширину. Таким образом, при создании эвристических функций всегда приходится искать компромисс между точностью и временем вычисления.

пустимой; кроме того, можно легко доказать, что функция  $h$  преэстивенна. К тому же  $h$  доминирует над всеми эвристическими функциями, которые входят в ее состав.

Допустимые эвристические функции могут быть также выведены на основе стоимости решения  $\approx$  подзадачи данной конкретной задачи. Например, на рис. 4.6 показана подзадача для экземпляра игры в восемь, приведенного на рис. 4.5. Эта подзадача касается перемещения фишек 1, 2, 3, 4 в их правильные позиции. Очевидно, что стоимость оптимального решения этой подзадачи представляет собой нижнюю границу стоимости решения полной задачи. Как оказалось, такая оценка в некоторых случаях является намного более точной по сравнению с манхэттенским расстоянием.

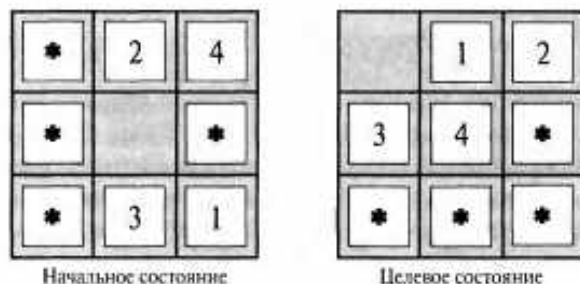


Рис. 4.6. Подзадача для экземпляра игры в восемь, показанного на рис. 4.5. Задание заключается в том, чтобы передвинуть фишки 1, 2, 3 и 4 в их правильные позиции, не беспокоясь о том, что произойдет с другими фишками

В основе  $\approx$  баз данных с шаблонами лежит такая идея, что указанные точные стоимости решений нужно хранить для каждого возможного экземпляра подзадачи — в нашем примере для каждой возможной конфигурации из четырех фишек и пустого квадрата. (Следует отметить, что при выполнении задания по решению этой подзадачи местонахождение остальных четырех фишек не нужно принимать во внимание, но ходы с этими фишками следует учитывать в стоимости решения.) После этого вычисляется допустимая эвристическая функция  $h_{\text{гв}}$  (здесь DB — Data Base) для каждого полного состояния, встретившегося в процессе поиска, путем выборки данных для соответствующей конфигурации подзадачи из базы данных. Сама база данных заполняется путем обратного поиска из целевого состояния и регистрации стоимости каждого нового встретившегося шаблона; затраты на этот поиск окупаются за счет успешного решения в дальнейшем многих новых экземпляров задачи.

Выбор фишек 1–2–3–4 является довольно произвольным; можно было бы также создать базы данных для фишек 5–6–7–8, 2–4–6–8 и т.д. По данным каждой базы данных формируется допустимая эвристическая функция, а эти эвристические функции можно составлять в общую эвристическую функцию, как было описано выше, применяя их максимальное значение. Составная эвристическая функция такого вида является намного более точной по сравнению с манхэттенским расстоянием; количество узлов, вырабатываемых при решении сформированных случайным образом экземпляров задачи игры в пятнадцать, может быть уменьшено примерно в 1000 раз.

Представляет интерес вопрос о том, можно ли складывать значения эвристических функций, полученных из баз данных 1–2–3–4 и 5–6–7–8, поскольку очевид-



но, что соответствующие две подзадачи не перекрываются. Будет ли при этом все еще получена допустимая эвристическая функция? Ответ на этот вопрос является отрицательным, поскольку в решениях подзадачи 1-2-3-4 и подзадачи 5-6-7-8 для данного конкретного состояния должны наверняка присутствовать некоторые общие ходы. Дело в том, что маловероятна ситуация, при которой фишки 1-2-3-4 можно было бы передвинуть на свои места, не трогая фишек 5-6-7-8, и наоборот. Но что будет, если не учитывать эти ходы? Иными словами, допустим, что регистрируется не общая стоимость решения подзадачи 1-2-3-4, а только количество ходов, в которых затрагиваются фишки 1-2-3-4. В таком случае можно легко определить, что сумма этих двух стоимостей все еще представляет собой нижнюю границу стоимости решения всей задачи. Именно эта идея лежит в основе **баз данных с непересекающимися шаблонами**. Применение таких баз данных позволяет решать случайно выбранные экземпляры задачи игры в пятнадцать за несколько миллисекунд — количество формируемых узлов сокращается примерно в 10 000 раз по сравнению с использованием манхэттенского расстояния. Для задачи игры в 24 может быть достигнуто ускорение приблизительно в миллион раз.

Базы данных с непересекающимися шаблонами успешно применяются для решения головоломок со скользящими фишками, поскольку сама задача может быть разделена таким образом, что каждый ход влияет лишь на одну подзадачу, так как одновременно происходит перемещение только одной фишки. При решении таких задач, как кубик Рубика, подобное разделение не может быть выполнено, поскольку каждый ход влияет на положение 8 или 9 из 26 элементов кубика. В настоящее время нет полного понимания того, как можно определить базы данных с непересекающимися шаблонами для подобных задач.

## Изучение эвристических функций на основе опыта

Предполагается, что эвристическая функция  $h(n)$  оценивает стоимость решения, начиная от состояния, связанного с узлом  $n$ . Как может некоторый агент составить подобную функцию? Одно из решений было приведено в предыдущем разделе, а именно: агент должен сформулировать ослабленные задачи, для которых может быть легко найдено оптимальное решение. Еще один подход состоит в том, что агент должен обучаться на основании полученного опыта. Здесь под “получением опыта” подразумевается, например, решение большого количества экземпляров игры в восемь. Каждое оптимальное решение задачи игры в восемь предоставляет примеры, на основе которых можно изучать функцию  $h(n)$ . Каждый пример складывается из состояния, взятого из пути решения, и фактической стоимости пути от этой точки до решения. На основе данных примеров с помощью алгоритма **индуктивного обучения** может быть сформирована некоторая функция  $h(n)$ , способная (при благоприятном стечении обстоятельств) предсказывать стоимости решений для других состояний, которые возникают во время поиска. Методы осуществления именно такого подхода с использованием нейронных сетей, деревьев решения и других средств описаны в главе 18. (Применимы также методы обучения с подкреплением, которые рассматриваются в главе 21.)

Методы индуктивного обучения действует лучше всего, когда в них учитываются **характеристики** состояния, релевантные для оценки этого состояния, а не просто общее описание состояния. Например, характеристика “количество стоящих не на

своих местах фишек” может быть полезной при предсказании фактического удаления некоторого состояния от цели. Назовем эту характеристику  $x_1(n)$ . Например, можно взять 100 сформированных случайным образом конфигураций головоломки игры в восемь и собрать статистические данные об их фактических стоимостях решений. Допустим, это позволяет обнаружить, что при  $x_1(n)$ , равном 5, средняя стоимость решения составляет около 14, и т.д. Наличие таких данных позволяет использовать значение  $x_1$  для предсказания значений функции  $h(n)$ . Безусловно, можно также применять сразу несколько характеристик. Второй характеристикой,  $x_2(n)$ , может быть “количество пар смежных фишек, которые являются также смежными в целевом состоянии”. Каким образом можно скомбинировать значения  $x_1(n)$  и  $x_2(n)$  для предсказания значения  $h(n)$ ? Общепринятый подход состоит в использовании линейной комбинации, как показано ниже.

$$h(n) = c_1 x_1(n) + c_2 x_2(n)$$

Константы  $c_1$  и  $c_2$  корректируются для достижения наилучшего соответствия фактическим данным о стоимостях решений. Предполагается, что константа  $c_1$  должна быть положительной, а  $c_2$  — отрицательной.

### 4.3. АЛГОРИТМЫ ЛОКАЛЬНОГО ПОИСКА И ЗАДАЧИ ОПТИМИЗАЦИИ

Рассматривавшиеся до сих пор алгоритмы поиска предназначались для систематического исследования пространств поиска. Такая систематичность достигается благодаря тому, что один или несколько путей хранится в памяти и проводится регистрация того, какие альтернативы были исследованы в каждой точке вдоль этого пути, а какие нет. После того как цель найдена, путь к этой цели составляет также искомое решение данной задачи.

Но при решении многих задач путь к цели не представляет интереса. Например, в задаче с восемью ферзями (см. с. 1) важна лишь окончательная конфигурация ферзей, а не порядок, в котором они были поставлены на доску. К этому классу задач относятся многие важные приложения, такие как проектирование интегральных схем, разработка плана цеха, составление производственного расписания, автоматическое программирование, оптимизация сети связи, составление маршрута транспортного средства и управление портфелем акций.

Если путь к цели не представляет интереса, то могут рассматриваться алгоритмы другого класса, в которых вообще не требуются какие-либо данные о путях. Алгоритмы **локального поиска** действуют с учетом единственного **текущего состояния** (а не многочисленных путей) и обычно предусматривают только переход в состояние, соседнее по отношению к текущему состоянию. Как правило, информация о путях, пройденных в процессе такого поиска, не сохраняется. Хотя алгоритмы локального поиска не предусматривают систематическое исследование пространства состояний (не являются систематическими), они обладают двумя важными преимуществами: во-первых, в них используется очень небольшой объем памяти, причем обычно постоянный, и, во-вторых, они часто позволяют находить приемлемые решения в больших или бесконечных (непрерывных) пространствах состояний, для которых систематические алгоритмы не применимы.

Кроме поиска целей, алгоритмы локального поиска являются полезным средством решения чистых  $\simeq$  **задач оптимизации**, назначение которых состоит в поиске состояния, наилучшего с точки зрения  $\simeq$  **целевой функции**. Многие задачи оптимизации не вписываются в “стандартную” модель поиска, представленную в главе 3. Например, природа предусмотрела такую целевую функцию (пригодность для репродукции), что дарвиновская эволюция может рассматриваться как попытка ее оптимизации, но в этой задаче оптимизации нет компонентов “проверка цели” и “стоимость пути”.

Авторы пришли к выводу, что для понимания сути локального поиска очень полезно рассмотреть  $\simeq$  **ландшафт пространства состояний** (подобный показанному на рис. 4.7). Этот ландшафт характеризуется и “местонахождением” (которое определяется состоянием), и “возвышением” (определяемым значением эвристической функции стоимости или целевой функции). Если возвышение соответствует стоимости, то задача состоит в поиске самой глубокой долины —  $\simeq$  **глобального минимума**, а если возвышение соответствует целевой функции, то задача заключается в поиске высочайшего пика —  $\simeq$  **глобального максимума**. (Минимум и максимум можно поменять местами, взяв их с обратными знаками.) Алгоритмы локального поиска исследуют такой ландшафт. Алгоритм **полного** локального поиска всегда находит цель, если она существует, а **оптимальный** алгоритм всегда находит глобальный минимум/максимум.

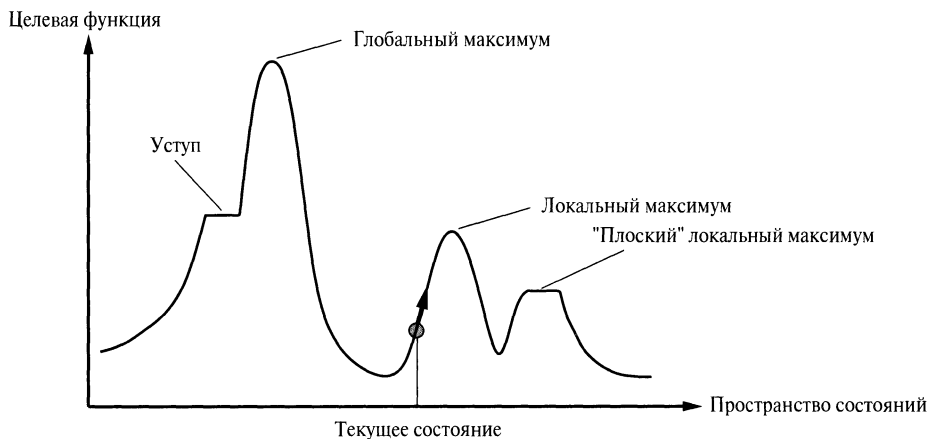


Рис. 4.7. Ландшафт одномерного пространства состояний, в котором возвышение соответствует целевой функции. Задача состоит в поиске глобального максимума. Как обозначено стрелкой, в процессе поиска по принципу “подъема к вершине” осуществляются попытки модификации текущего состояния в целях его улучшения. Различные топографические особенности ландшафта определены в тексте

## Поиск с восхождением к вершине

Алгоритм поиска  $\simeq$  **с восхождением к вершине** показан в листинге 4.2. Он представляет собой обычный цикл, в котором постоянно происходит перемещение в направлении возрастания некоторого значения, т.е. подъем. Работа этого алгоритма заканчивается после достижения “пика”, в котором ни одно из соседних состояний не имеет более высокого значения. В данном алгоритме не предусмотрено сопрово-

ждение дерева поиска, поэтому в структуре данных 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 непосредственно соседними по отношению к текущему состоянию. Это напоминает попытку альпиниста, страдающего от амнезии, найти вершину горы Эверест в густом тумане.

**Листинг 4.2. Алгоритм поиска с восхождением к вершине (версия с наиболее крутым подъемом), который представляет собой самый фундаментальный метод локального поиска. На каждом этапе текущий узел заменяется наилучшим соседним узлом; в данной версии таковым является узел с максимальным значением Value, но если используется эвристическая оценка стоимости  $h$ , то может быть предусмотрен поиск соседнего узла с минимальным значением  $h$**

---

```
function Hill-Climbing(problem) returns состояние, которое
    представляет собой локальный максимум
inputs: problem, задача
local variables: current, узел
                 neighbor, узел

current ← Make-Node(Initial-State[problem])
loop do
    neighbor ← приемник узла current с наивысшим значением
    if Value[neighbor] ≤ Value[current] then return
        State[current]
    current ← neighbor
```

---

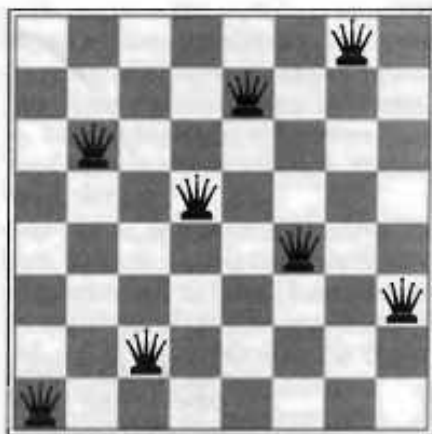
Для иллюстрации поиска с восхождением к вершине воспользуемся **задачей с восемью ферзями**, которая представлена на с. 118. В алгоритмах локального поиска обычно применяется **формулировка полного состояния**, в которой в каждом состоянии на доске имеется восемь ферзей, по одному ферзю в каждом столбце. Функция определения приемника возвращает все возможные состояния, формируемые путем перемещения отдельного ферзя в другую клетку одного и того же столбца (поэтому каждое состояние имеет  $8 \times 7 = 56$  приемников). Эвристическая функция стоимости  $h$  определяет количество пар ферзей, которые атакуют друг друга либо прямо, либо косвенно (атака называется косвенной, если на одной горизонтали, вертикали или диагонали стоят больше двух ферзей). Глобальный минимум этой функции становится равным нулю, и это происходит только в идеальных решениях. На рис. 4.8, *a* показано состояние со значением  $h=17$ . На этом рисунке также показаны значения всех приемников данного состояния, притом что наилучшие приемники имеют значение  $h=12$ . Алгоритмы с восхождением к вершине обычно предусматривают случайный выбор в множестве наилучших приемников, если количество приемников больше одного.

Поиск с восхождением к вершине иногда называют **жадным локальным поиском**, поскольку в процессе его выполнения происходит захват самого хорошего соседнего состояния без предварительных рассуждений о том, куда следует отправиться дальше. Жадность считается одним из семи смертных грехов, но, как оказалось, жадные алгоритмы часто показывают весьма высокую производительность. Во время поиска с восхождением к вершине зачастую происходит очень быстрое продвижение в направлении к решению, поскольку обычно бывает чрезвычайно легко

улучшить плохое состояние. Например, из состояния, показанного на рис. 4.8, а, достаточно сделать лишь пять ходов, чтобы достичь состояния, показанного на рис. 4.8, б, которое имеет оценку  $h=1$  и очень близко к одному из решений.

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♚	13	16	13	16
♚	14	17	15	♚	14	16	16
17	♚	16	18	15	♚	15	♚
18	14	♚	15	15	14	♚	16
14	14	13	17	12	14	12	18

а)

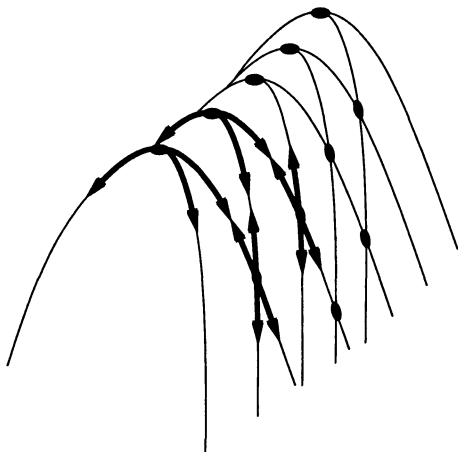


б)

Рис. 4.8. Пример применения алгоритма с восхождением к вершине: диаграмма состояния задачи с восемью ферзями, характеризующегося эвристической оценкой стоимости  $h=17$ ; на этой диаграмме показано значение  $h$  для каждого возможного приемника, полученное путем передвижения ферзя в пределах своего столбца; отмечены наилучшие ходы (а); локальный минимум в пространстве состояний задачи с восемью ферзями; это состояние имеет оценку  $h=1$ , но каждый приемник характеризуется более высокой стоимостью (б)

К сожалению, поиск с восхождением к вершине часто заходит в тупик по описанным ниже причинам.

- **Локальные максимумы.** Локальный максимум представляет собой пик, более высокий по сравнению с каждым из его соседних состояний, но более низкий, чем глобальный максимум. Алгоритмы с восхождением к вершине, которые достигают окрестностей локального максимума, обеспечивают продвижение вверх, к этому пику, но после этого заходят в тупик, из которого больше некуда двигаться. Такая проблема схематически показана на рис. 4.7. Более конкретный пример состоит в том, что состояние, показанное на рис. 4.8, б, фактически представляет собой локальный максимум (т.е. локальный минимум для оценки стоимости  $h$ ); задача еще не решена, а при любом передвижении отдельно взятого ферзя ситуация становится еще хуже.
- **Хребты.** Пример хребта показан на рис. 4.9. При наличии хребтов возникают последовательности локальных максимумов, задача прохождения которых для жадных алгоритмов является очень трудной.
- **Плато.** Это область в ландшафте пространства состояний, в которой функция оценки является плоской. Оно может представлять собой плоский локальный максимум, из которого не существует выхода вверх, или *до углуб*, из которого возможно дальнейшее успешное продвижение (см. рис. 4.7). Поиск с восхождением к вершине может оказаться неспособным выйти за пределы плато.



*Рис. 4.9. Иллюстрация того, почему хребты вызывают сложности при восхождении к вершине. На хребет, возвышающийся слева направо, налагается решетка состояний (полужирные дуги), создавая последовательность локальных максимумов, которые не являются непосредственно связанными друг с другом. В каждом локальном максимуме все доступные действия направлены вниз*

В каждом из этих случаев рассматриваемый алгоритм достигает такой точки, из которой не может осуществляться дальнейшее успешное продвижение. Начиная со случайно сформированного состояния с восемью ферзями, алгоритм поиска с восхождением к вершине по самому крутому подъему заходит в тупик в 86% случаях, решая только 14% экземпляров этой задачи. Но он работает очень быстро, выполняя в среднем только 4 этапа в случае успешного завершения и 3 этапа, когда заходит в тупик. Это не очень плохой результат для пространства состояний с  $8^8 \approx 17$  миллионами состояний.

Алгоритм, приведенный в листинге 4.2, останавливается, достигнув плато, на котором наилучший приемник имеет такое же значение, как и в текущем состоянии. Имеет ли смысл продолжать движение, разрешив **движение в сторону** в надежде на то, что это плато в действительности представляет собой уступ, как показано на рис. 4.7? Ответ на этот вопрос обычно является положительным, но необходимо соблюдать осторожность. Если будет всегда разрешено движение в сторону, притом что движение вверх невозможно, могут возникать бесконечные циклы после того, как алгоритм достигнет плоского локального максимума, не являющегося уступом. Одно из широко применяемых решений состоит в том, что устанавливается предел количества допустимых последовательных движений в сторону. Например, можно разрешить, допустим, 100 последовательных движений в сторону в задаче с восемью ферзями. В результате этого относительное количество экземпляров задачи, решаемых с помощью восхождения к вершине, возрастает с 14 до 94%. Но за этот успех приходится платить: алгоритм в среднем выполняет приблизительно 21 этап при каждом успешном решении экземпляра задачи и 64 этапа при каждой неудаче.

Разработано много вариантов поиска с восхождением к вершине. При **стохастическом поиске с восхождением к вершине** осуществляется выбор слу-

чайным образом одного из движений вверх; вероятность такого выбора может зависеть от крутизны движения вверх. Обычно этот алгоритм сходится более медленно по сравнению с вариантом, предусматривающим наиболее крутой подъем, но в некоторых ландшафтах состояний он находит лучшие решения. При **поиске с восхождением к вершине с выбором первого варианта** реализуется стохастический поиск с восхождением к вершине путем формирования преемников случайным образом до тех пор, пока не будет сформирован преемник, лучший по сравнению с текущим состоянием. Это — хорошая стратегия, если любое состояние имеет большое количество преемников (измеряемое тысячами). В упр. 4.16 предлагается исследовать этот алгоритм.

Алгоритмы с восхождением к вершине, описанные выше, являются неполными, поскольку часто оказываются неспособными найти цель, притом что она существует, из-за того, что могут зайти в тупик, достигнув локального максимума. **Поиск с восхождением к вершине и перезапуском случайным образом** руководствуется широко известной рекомендацией: “Если первая попытка оказалась неудачной, пробуйте снова и снова”. В этом алгоритме предусмотрено проведение ряда поисков из сформированных случайным образом начальных состояний<sup>8</sup> и остановов после достижения цели. Он является полным с вероятностью, достигающей 1, даже по той тривиальной причине, что в нем в конечном итоге в качестве начального состояния формируется одно из целевых состояний. Если вероятность успеха каждого поиска с восхождением к вершине равна  $p$ , то ожидаемое количество требуемых перезапусков составляет  $1/p$ . Для экземпляров задачи с восемью ферзями, если не разрешено движение в сторону,  $p \approx 0.14$ , поэтому для нахождения цели требуется приблизительно 7 итераций (6 неудачных и 1 успешная). Ожидаемое количество этапов решения равно стоимости одной успешной итерации, которая складывается с увеличенной в  $(1-p)/p$  раз стоимостью неудачи, или составляет приблизительно 22 этапа. Если разрешено движение в сторону, то в среднем требуется  $1/0.94 \approx 1.06$  итераций и  $(1 \times 21) + (0.06/0.94) \times 64 \approx 25$  этапов. Поэтому алгоритм поиска с восхождением к вершине и перезапуском случайным образом действительно является очень эффективным применительно к задаче с восемью ферзями. Даже для варианта с тремя миллионами ферзей этот подход позволяет находить решения меньше чем за минуту<sup>9</sup>.

Успех поиска с восхождением к вершине в значительной степени зависит от формы ландшафта пространства состояний: если в нем есть лишь немного локальных максимумов и плато, то поиск с восхождением к вершине и перезапуском случайным образом позволяет очень быстро найти хорошее решение. С другой стороны, многие реальные задачи имеют ландшафт, который больше напоминает семейство дикобразов на плоском полу, где на вершине иглы каждого дикобраза живут другие миниатюрные дикобразы и т.д., до бесконечности. NP-трудные задачи обычно имеют экспоненциальное количество локальных максимумов, способных завести

<sup>8</sup> Может оказаться трудной даже сама задача формирования случайным образом некоторого состояния из неявно заданного пространства состояний.

<sup>9</sup> В [958] доказано, что в некоторых случаях лучше всего осуществлять перезапуск рандомизированного алгоритма поиска по истечении конкретной, постоянной продолжительности времени и что такой подход может оказаться гораздо более эффективным по сравнению с тем, когда разрешено продолжать каждый поиск до бесконечности. Примером такого подхода является также запрещение или ограничение количества движений в сторону.

алгоритм в тупик. Несмотря на это, часто существует возможность найти достаточно хороший локальный максимум после небольшого количества перезапусков.

### Поиск с эмуляцией отжига

Для любого алгоритма с восхождением к вершине, который никогда не выполняет движения “вниз по склону”, к состояниям с более низкой оценкой (или более высокой стоимостью), гарантируется, что он окажется неполным, поскольку такой алгоритм всегда способен зайти в тупик, достигнув локального максимума. В отличие от этого алгоритм с чисто случайным блужданием (т.е. с перемещением к приемнику, выбираемому на равных правах случайным образом из множества приемников) является полным, но чрезвычайно неэффективным. Поэтому представляется разумной попытка скомбинировать каким-то образом восхождение к вершине со случайным блужданием, что позволит обеспечить и эффективность, и полноту. Алгоритмом такого типа является алгоритм с **эмуляцией отжига**. В металлургии **отжигом** называется процесс, применяемый для отпуска металла и стекла путем нагревания этих материалов до высокой температуры, а затем постепенного охлаждения, что позволяет перевести обрабатываемый материал в низкоэнергетическое кристаллическое состояние. Чтобы понять суть эмуляции отжига, переведем наше внимание с восхождения к вершине на **градиентный спуск** (т.е. минимизацию стоимости) и представим себе, что наше задание — загнать теннисный шарик в самую глубокую лунку на неровной поверхности. Если бы мы просто позволили шарiku катиться по этой поверхности, то он застрял бы в одном из локальных минимумов. А встряхивая поверхность, можно вытолкнуть шарик из локального минимума. Весь секрет состоит в том, что поверхность нужно трясти достаточно сильно, чтобы шарик можно было вытолкнуть из локальных минимумов, но не настолько сильно, чтобы он вылетел из глобального минимума. Процесс поиска решения с эмуляцией отжига заключается в том, что вначале происходит интенсивное встряхивание (аналогичное нагреву до высокой температуры), после чего интенсивность встряхивания постепенно уменьшается (что можно сравнить с понижением температуры).

Самый внутренний цикл алгоритма с эмуляцией отжига (листинг 4.3) полностью аналогичен циклу алгоритма с восхождением к вершине, но в нем вместо наилучшего хода выполняется случайно выбранный ход. Если этот ход улучшает ситуацию, то всегда принимается. В противном случае алгоритм принимает данный ход с некоторой вероятностью, меньшей 1. Эта вероятность уменьшается экспоненциально с “ухудшением” хода — в зависимости от величины  $\Delta E$ , на которую ухудшается его оценка. Кроме того, вероятность уменьшается по мере снижения “температуры”  $T$ : “плохие” ходы скорее всего могут быть разрешены в начале, когда температура высока, но становятся менее вероятными по мере снижения  $T$ . Можно доказать, что если в графике *schedule* предусмотрено достаточно медленное снижение  $T$ , то данный алгоритм позволяет найти глобальный оптимум с вероятностью, приближающейся к 1.

На первых порах, в начале 1980-х годов, поиск с эмуляцией отжига широко использовался для решения задач компоновки СБИС. Кроме того, этот алгоритм нашел широкое применение при решении задач планирования производства и других крупномасштабных задач оптимизации. В упр. 4.16 предлагается сравнить его производительность с производительностью поиска с восхождением к вершине и перезапуском случайным образом при решении задачи с  $n$  ферзями.



**Листинг 4.3.** Алгоритм поиска с эмуляцией отжига, который представляет собой одну из версий алгоритма стохастического поиска с восхождением к вершине, в которой разрешены некоторые ходы вниз. Ходы вниз принимаются к исполнению с большей вероятностью на ранних этапах выполнения графика отжига, а затем, по мере того как проходит время, выполняются менее часто. Входной параметр *schedule* определяет значение температуры *T* как функции от времени

---

```

function Simulated-Annealing(problem, schedule) returns состояние
    решения
    inputs: problem, задача
             schedule, отображение между временем и "температурой"
    local variables: current, узел
                     next, узел
                     T, "температура", от которой зависит вероятность
                       шагов вниз

    current  $\leftarrow$  Make-Node(Initial-State[problem])
    for t  $\leftarrow$  1 to  $\infty$  do
        T  $\leftarrow$  schedule[t]
        if T = 0 then return current
        next  $\leftarrow$  случайно выбранный преемник состояния current
         $\Delta E \leftarrow$  Value[next] - Value[current]
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next с вероятностью только  $e^{\Delta E/T}$ 

```

---

### Локальный лучевой поиск

Стремление преодолеть ограничения, связанные с нехваткой памяти, привело к тому, что в свое время предпочтение отдавалось алгоритмам, предусматривающим хранение в памяти только одного узла, но, как оказалось, такой подход часто является слишком радикальным способом экономии памяти. В алгоритме ~~э~~ **локального лучевого поиска**<sup>10</sup> предусмотрено отслеживание *k* состояний, а не только одного состояния. Работа этого алгоритма начинается с формирования случайным образом *k* состояний. На каждом этапе формируются все преемники всех *k* состояний. Если какой-либо из этих преемников соответствует целевому состоянию, алгоритм останавливается. В противном случае алгоритм выбирает из общего списка *k* наилучших преемников и повторяет цикл.

На первый взгляд может показаться, что локальный лучевой поиск с *k* состояниями представляет собой не что иное, как выполнение *k* перезапусков случайным образом, но не последовательно, а параллельно. Тем не менее в действительности эти два алгоритма являются полностью разными. При поиске с перезапуском случайным образом каждый процесс поиска осуществляется независимо от других. *☞* *А в локальном лучевом поиске полезная информация передается по k параллельным потокам поиска.* Например, если в одном состоянии вырабатывается несколько хороших преемников, а во всех других *k*-1 состояниях вырабатываются плохие преемники, то возникает такой эффект, как если бы поток, контролирующий первое состояние, сообщил другим потокам: "Идите все сюда, здесь

---

<sup>10</sup> Локальный лучевой поиск является адаптацией **лучевого поиска**, который представляет собой алгоритм, основанный на использовании пути.

трава зеленее!” Этот алгоритм способен быстро отказаться от бесплодных поисков и перебросить свои ресурсы туда, 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 состояний, в результате чего этот поиск начинает ненамного отличаться от дорогостоящей версии поиска с восхождением к вершине. Этот недостаток позволяет устранить вариант, называемый **стохастическим лучевым поиском**, который аналогичен стохастическому поиску с восхождением к вершине. При стохастическом лучевом поиске вместо выбора наилучших  $k$  преемников из пула преемников-кандидатов происходит выбор  $k$  преемников случайным образом, притом что вероятность выбора данного конкретного преемника представляет собой возрастающую функцию значения его оценки. Стохастический лучевой поиск имеет некоторое сходство с процессом естественного отбора, в котором “преемники” (потомки) некоторого “состояния” (организма) образуют следующее поколение в соответствии со “значением их оценки” (в соответствии с их жизненной пригодностью).

## Генетические алгоритмы

**Генетический алгоритм** (Genetic Algorithm — GA) представляет собой вариант стохастического лучевого поиска, в котором состояния-преемники формируются путем комбинирования двух родительских состояний, а не посредством модификации единственного состояния. В нем просматривается такая же аналогия с естественным отбором, как и в стохастическом лучевом поиске, за исключением того, что теперь мы имеем дело с половым, а не бесполом воспроизводством.

Как и при лучевом поиске, работа алгоритмов GA начинается с множества  $k$  сформированных случайным образом состояний, называемых **популяцией**. Каждое состояние, или **индивидуум**, представлено в виде строки символов из конечного алфавита, чаще всего в виде строки из нулей (0) и единиц (1). Например, состояние задачи с восемью ферзями должно определять позиции восьми ферзей, каждый из которых стоит на вертикали с 8 клетками, и поэтому для его представления требуется  $8 \times \log_2 8 = 24$  бита. Иным образом, каждое состояние может быть представлено в виде восьми цифр, каждая из которых находится в диапазоне от 1 до 8. (Как будет показано ниже, эти две кодировки проявляют себя в ходе поиска поразному.) На рис. 4.10, *а* показана популяция из четырех восьмисимвольных строк, представляющих состояния с восемью ферзями.

Процесс выработки следующего поколения состояний показан на рис. 4.10, *б–д*. На рис. 4.10, *б* каждое состояние классифицируется с помощью функции оценки, или (в терминологии GA) **функции пригодности**. Функция пригодности должна возвращать более высокие значения для лучших состояний, поэтому в задаче с восемью ферзями используется количество неатакующих друг друга пар ферзей, которое в любом решении имеет значение 28. Для этих четырех состояний соответствующие значения равны 24, 23, 20 и 11. В данном конкретном варианте генетического алгоритма вероятность выбора для воспроизводства прямо пропорциональна оценке пригодности; соответствующие вероятности в процентах показаны рядом с исходными оценками.

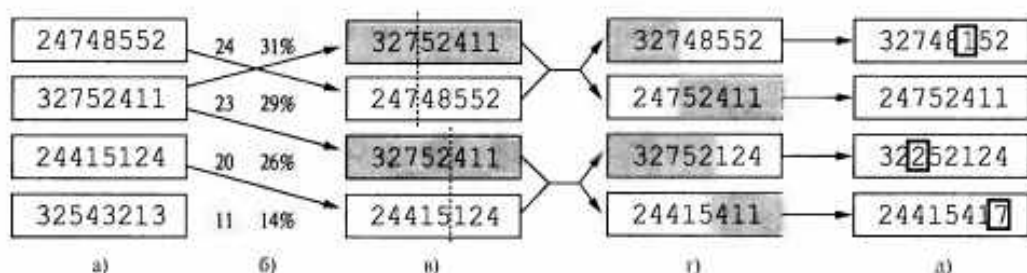


Рис. 4.10. Генетический алгоритм: начальная популяция (а); функция пригодности (б); отбор (в); скрещивание (г); мутация (д). Начальная популяция классифицируется с помощью функции пригодности, в результате чего формируются пары для скрещивания. Эти пары производят потомков, которые в конечном итоге подвергаются мутации

Как показано на рис. 4.10, в, для воспроизводства случайным образом выбираются две пары в соответствии с вероятностями, показанными на рис. 4.10, б. Обратите внимание на то, что один индивидум выбирается дважды, а один вообще остается не выбранным<sup>11</sup>. Для каждой пары, предназначенной для воспроизводства, среди позиций в строке случайным образом выбирается точка  $\times$  скрещивания. На рис. 4.10 точки скрещивания находятся после третьей цифры в первой паре и после пятой цифры во второй паре<sup>12</sup>.

Как показано на рис. 4.10, г, сами потомки создаются путем перекрестного обмена подстроками родительских строк, разорванных в точке скрещивания. Например, первый потомок первой пары получает три первые 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 задачи с восемью ферзями, участвующие в этом этапе воспроизводства, показаны на рис. 4.11. Данный пример иллюстрирует тот факт, что если два родительских состояния являются весьма различными, то операция скрещивания способна выработать состояние, которое намного отличается и от одного, и от другого родительского состояния. Часто происходит так, что популяция становится весьма разнообразной на самых ранних этапах этого процесса, поэтому скрещивание (как и эмуляция отжига) в основном обеспечивает выполнение крупных этапов в пространстве состояний в начале процесса поиска и более мелких этапов позднее, когда большинство индивидуумов становятся весьма похожими друг на друга.

Наконец, на рис. 4.10, д показано, что каждое местонахождение подвергается случайной  $\times$  мутации с небольшой независимой вероятностью. В первом, третьем и четвертом потомках мутация свелась к изменению одной цифры. В задаче с восемью ферзями эта операция соответствует выбору случайным образом одного ферзя и пе-

<sup>11</sup> Существует много вариантов этого правила выбора. Можно показать, что при использовании метода **отсеивания**, в котором отбрасываются все индивидуумы с оценками ниже заданного порога, алгоритм сходится быстрее, чем при использовании версии со случайным выбором [81].

<sup>12</sup> Кодировка начинает играть решающую роль в процессе решения именно на этом этапе. Если вместо 8 шифр используется 24-битовая кодировка, то вероятность попадания точки скрещивания в середину цифры становится равной 2/3, а это приводит к тому, что мутация этой цифры по сути становится произвольной.

ремонтированию этого ферзя на случайно выбранную клетку в его столбце. В листинге 4.4 приведен алгоритм, который реализует все эти этапы.

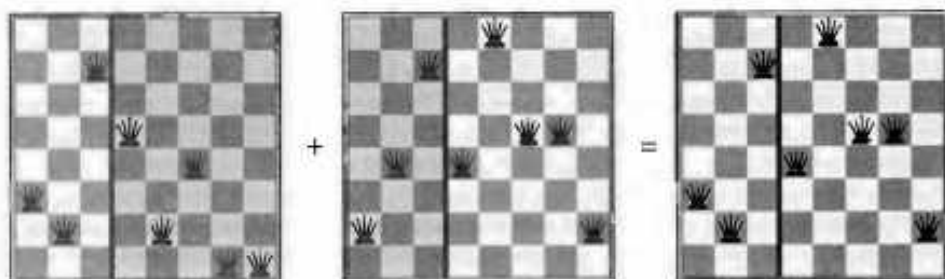


Рис. 4.11. Состояния задачи с восемью ферзями, соответствующие первым двум родительским состояниям, показанным на рис. 4.10, в, и первому потомку, показанному на рис. 4.10, г. Затененные столбцы на этапе скрещивания теряются, а незатененные сохраняются

**Листинг 4.4.** Генетический алгоритм. Этот алгоритм аналогичен тому, который показан схематически на рис. 4.10, за одним исключением: в этой более широко применяемой версии алгоритма каждое скрещивание двух родителей приводит к получению только одного потомка, а не двух

```
function Genetic-Algorithm(population, Fitness-Fn) returns индивидиум
    individual
    inputs: population, множество индивидиумов (популяция)
           Fitness-Fn, функция, которая измеряет пригодность
                индивидиума

    repeat
        new_population ← пустое множество
        loop for i from 1 to Size(population) do
            x ← Random-Selection(population, Fitness-Fn)
            y ← Random-Selection(population, Fitness-Fn)
            child ← Reproduce(x, y)
            if (небольшое случайно выбранное значение вероятности)
                then child ← Mutate(child)
            добавить дочерний индивидиум child
                к множеству new_population
        population ← new_population
    until некоторый индивидиум не станет достаточно пригодным
        или не истечет достаточное количество времени
    return наилучший индивидиум individual в популяции population,
        в соответствии с функцией Fitness-Fn

function Reproduce(x, y) returns индивидиум individual
    inputs: x, y, индивидиумы-родители

    n ← Length(x)
    c ← случайное число от 1 до n
    return Append(Substring(x, 1, c), Substring(y, c+1, n))
```

## ЭВОЛЮЦИЯ И ПОИСК

Теория эволюции была изложена Чарльзом Дарвином в его книге *On the Origin of Species by Means of Natural Selection* (Происхождение видов путем естественного отбора) [325]. Основная идея этой теории проста: при воспроизводстве возникают вариации (известные как **мутации**), которые сохраняются в следующих поколениях с частотой, приблизительно пропорциональной тому, как они влияют на пригодность к воспроизводству.

Дарвин разрабатывал свою теорию, не зная о том, благодаря чему происходит наследование и модификация характерных особенностей организмов. Вероятностные законы, управляющие этими процессами, были впервые обнаружены Грегором Менделем [1034], монахом, проводившим эксперименты с душистым горошком с использованием метода, названного им *искусственным оплодотворением*. Гораздо позже Уотсон и Крик [1559] выявили структуру молекулы ДНК (дезоксирибонуклеиновой кислоты) и ее алфавит, состоящий из аминокислот АГТЦ (аденин, гуанин, тимин, цитозин). В предложенной ими стандартной модели вариации возникают и в результате точечных мутаций в последовательности этих аминокислот, и в результате “скрещивания” (при котором ДНК потомка формируется путем объединения длинных секций ДНК от каждого родителя).

Аналогия между этим процессом и алгоритмами локального поиска уже была описана выше; принципиальное различие между стохастическим лучевым поиском и эволюцией состоит в использовании полового воспроизводства, в котором потомки формируются с участием нескольких организмов, а не только одного. Однако фактически механизмы эволюции намного богаче по сравнению с тем, что допускает большинство генетических алгоритмов. Например, мутации могут быть связаны с обращением, дублированием и перемещением больших фрагментов ДНК; некоторые вирусы заимствуют ДНК из одного организма и вставляют в другой; кроме того, существуют взаимозаменяемые гены, которые лишь копируют самих себя в геноме много тысяч раз. Есть даже такие гены, которые отравляют клетки потенциальных родителей, не несущие ген этого типа, повышая тем самым шансы на воспроизводство данного гена. Наиболее важным является тот факт, что гены сами кодируют те механизмы, с помощью которых геном воспроизводится и транслируется в организме. В генетических алгоритмах такие механизмы представляют собой отдельную программу, не закодированную в строках, с которыми осуществляются манипуляции.

На первый взгляд может показаться, что механизм дарвиновской эволюции является неэффективным, поскольку в его рамках на Земле за многие тысячи лет было вслепую сформировано около  $10^{45}$  или примерно столько организмов, притом что за такое время поисковые эвристики этого механизма не улучшились ни на йоту. Но за пятьдесят лет до Дарвина французский натуралист Жан Ламарк [884], получивший известность в другой области, предложил теорию эволюции, согласно которой характерные особенности организма, приобретенные в результате его адаптации на протяжении срока жизни этого организма, передаются его потомкам. Такой процесс был бы очень эффективным, но в природе, по-видимому, не происходит. Намного

позднее Джеймс Болдуин [64] предложил теорию, которая внешне кажется аналогичной; согласно этой теории, поведение, которому обучился организм на протяжении своей жизни, способствует повышению скорости эволюции. В отличие от теории Ламарка, теория Болдуина полностью согласуется с дарвиновской теорией эволюции, поскольку в ней учитываются давления отбора, действующие на индивидуумов, которые нашли локальные оптимумы среди множества возможных вариантов поведения, допустимых согласно их генетической природе. Современные компьютерные модели подтвердили, что “эффект Болдуина” является реальным, при условии, что “обычная” эволюция способна создавать организмы, внутренние показатели производительности которых каким-то образом коррелируют с их фактической жизненной пригодностью.

Как и в алгоритмах стохастического лучевого поиска, в генетических алгоритмах тенденция к стремлению к максимуму сочетается с проводимым случайным образом исследованием и с обменом информацией между параллельными поисковыми потоками. Основное преимущество генетических алгоритмов (если таковое действительно имеется) связано с применением операции скрещивания. Тем не менее можно доказать математически, что если позиции в генетическом коде первоначально устанавливаются в случайном порядке, то скрещивание не дает никаких преимуществ. На интуитивном уровне можно предположить, что преимущество была бы способность комбинировать в результате скрещивания большие блоки символов, которые были независимо сформированы в ходе эволюции для выполнения полезных функций, что позволило бы повысить степень детализации, с которой действует этот алгоритм поиска. Например, преимущество достигалось бы, если в качестве полезного блока была выделена строка символов, предусматривающая размещение первых трех ферзей в позициях 2, 4 и 6 (где они не атакуют друг друга), после чего можно было бы объединять этот блок с другими блоками для формирования решения.

В теории генетических алгоритмов описано, как действует этот подход, в котором используется идея **схемы**, представляющей собой такую подстроку, где некоторые из позиций могут оставаться не заданными. Например, схема 246\*\*\*\*\* описывает такие состояния всех восьми ферзей, в которых первые три ферзя находятся соответственно в позициях 2, 4 и 6. Строки, соответствующие этой схеме (такие как 24613578), называются **экземплярами** схемы. Можно доказать, что если средняя пригодность экземпляров некоторой схемы находится выше среднего, то количество экземпляров этой схемы в популяции будет расти со временем. Очевидно, что данный эффект вряд ли окажется существенным, если смежные биты полностью не связаны друг с другом, поскольку в таком случае будет лишь немного непрерывных блоков, которые предоставляли бы какое-то постоянное преимущество. Генетические алгоритмы работают лучше всего в сочетании со схемами, соответствующими осмысленным компонентам решения. Например, если строка представляет какую-либо радиотехническую антенну, то схемы могут соответствовать компонентам этой антенны, таким как рефлекторы и дефлекторы. Хороший компонент, по всей вероятности, будет оставаться хорошим во многих разных проектах. Из этого следует, что для успешного использования генетических алгоритмов требуется тщательное конструирование представления задачи.

На практике генетические алгоритмы оказали глубокое влияние на научные методы, применяющиеся при решении таких задач оптимизации, как компоновка электронных схем и планирование производства. В настоящее время уже не так ясно, вызвана ли притягательность генетических алгоритмов их высокой производительностью или обусловлена эстетически привлекательными истоками в теории эволюции. Но все еще необходимо проделать большой объем работы для выяснения условий, при которых генетические алгоритмы обладают наиболее высокой производительностью.

## 4.4. ЛОКАЛЬНЫЙ ПОИСК В НЕПРЕРЫВНЫХ ПРОСТРАНСТВАХ

В главе 2 было описано различие между дискретными и непрерывными вариантами среды, а также указано, что большинство реальных вариантов среды являются непрерывными. Но еще ни один из описанных выше алгоритмов не способен действовать в непрерывных пространствах состояний, поскольку в этих алгоритмах в большинстве случаев функция определения преемника возвращала бы бесконечно большое количество состояний! В настоящем разделе приведено очень краткое введение в некоторые методы локального поиска, предназначенные для нахождения оптимальных решений в непрерывных пространствах. Литература по этой теме весьма обширна; многие из этих основных методов впервые были созданы в XVII веке после разработки первых математических исчислений Ньютоном и Лейбницем<sup>13</sup>. Применение данных методов рассматривается в нескольких главах настоящей книги, включая главы, касающиеся обучения, машинного зрения и робототехники. Короче говоря, эти методы касаются всего, что связано с реальным миром.

Начнем изложение этой темы с примера. Предположим, что где-то в Румынии требуется найти место для размещения трех новых аэропортов таким образом, чтобы сумма квадратов расстояний от каждого города на карте (см. рис. 3.1) до ближайшего к нему аэропорта была минимальной. В таком случае пространство состояний определено координатами аэропортов:  $(x_1, y_1)$ ,  $(x_2, y_2)$  и  $(x_3, y_3)$ . Это — шестимерное пространство; иными словами можно выразить данную мысль так, что состояния определяются шестью переменными. (Вообще говоря, состояния определяются  $n$ -мерным вектором переменных,  $\mathbf{x}$ .) Перемещение в этом пространстве соответствует переносу одного или нескольких из этих аэропортов в другое место на карте. Целевую функцию  $f(x_1, y_1, x_2, y_2, x_3, y_3)$  после определения ближайших городов можно вычислить довольно легко, но гораздо сложнее составить общее выражение, соответствующее искомому решению.

Один из способов предотвращения необходимости заниматься непрерывными задачами состоит в том, чтобы просто дискретизировать окрестности каждого состояния. Например, можно предусмотреть перемещение одновременно только одного аэропорта в направлении либо  $x$ , либо  $y$  на постоянную величину  $\pm\Delta$ . При наличии шести переменных это соответствует двенадцати возможным преемникам для каждого состояния. После этого появляется возможность применить любой из алгоритмов локального поиска, описанных выше. Кроме того, алгоритмы стохастиче-

<sup>13</sup> Для чтения этого раздела желательно обладать элементарными знаниями о системах исчисления в многомерном пространстве и векторной арифметике.

ского поиска с восхождением к вершине и поиска с эмуляцией отжига можно применять непосредственно, без дискретизации этого пространства. Такие алгоритмы выбирают преемников случайным образом, что может быть осуществлено путем формирования случайным образом векторов с длиной  $\Delta$ .

Имеется также много методов, в которых при осуществлении попыток найти максимум используется  $\approx$  **градиент** ландшафта. Градиент целевой функции представляет собой вектор  $\nabla f$ , позволяющий определить величину и направление наиболее крутого склона. Для рассматриваемой задачи справедливо следующее соотношение:

$$\nabla f = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right)$$

В некоторых случаях можно найти максимум, решая уравнение  $\nabla f = 0$ . (Это можно сделать, например, при размещении только одного аэропорта; решение представляет собой арифметическое среднее координат всех городов.) Но во многих случаях это уравнение не может быть решено в замкнутой форме. Например, при наличии трех аэропортов выражение для градиента зависит от того, какие города являются ближайшими к каждому аэропорту в текущем состоянии. Это означает, что мы можем вычислить этот градиент локально, но не глобально. Даже в таком случае остается возможность выполнять поиск с восхождением к вершине по самому крутому склону, обновляя текущее состояние с помощью следующей формулы:

$$\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x})$$

где  $\alpha$  — небольшая константа. В других случаях целевая функция в дифференцируемой форме может оказаться вообще не доступной, например, значение конкретного множества местонахождений аэропортов может быть определено в результате вызова на выполнение какого-то пакета крупномасштабного экономического моделирования. В таких случаях можно определить так называемый  $\approx$  **эмпирический градиент**, оценивая отклик на небольшие увеличения и уменьшения каждой координаты. Поиск по эмпирическому градиенту является аналогичным поиску с восхождением к вершине по самому крутому склону в дискретизированной версии данного пространства состояний.

За фразой “ $\alpha$  — небольшая константа” скрывается огромное разнообразие методов определения значения  $\alpha$ . Основная проблема состоит в том, что если значение  $\alpha$  слишком мало, то требуется слишком много этапов поиска, а если слишком велико, то в поиске можно проскочить максимум. Попытка преодолеть эту дилемму предпринимается в методе  $\approx$  **линейного поиска**, который предусматривает продолжение поиска в направлении текущего градиента (обычно путем повторного удвоения  $\alpha$ ) до тех пор, пока  $f$  не начнет снова уменьшаться. Точка, в которой это происходит, становится новым текущим состоянием. Сформировалось несколько научных школ, в которых доминируют разные взгляды на то, каким образом в этой точке следует выбирать новое направление.

Для многих задач наиболее эффективным алгоритмом является почтенный метод  $\approx$  **Ньютона–Рафсона** [1132], [1266]. Это — общий метод поиска корней функций, т.е. метод решения уравнений в форме  $g(x) = 0$ . Этот алгоритм действует на основе вычисления новой оценки для корня  $\mathbf{x}$  в соответствии с формулой Ньютона:

$$\mathbf{x} \leftarrow \mathbf{x} - g(\mathbf{x}) / g'(\mathbf{x})$$



Чтобы найти максимум или минимум  $f$ , необходимо найти такое значение  $\mathbf{x}$ , для которого градиент равен нулю (т.е.  $\nabla f(\mathbf{x}) = \mathbf{0}$ ). Поэтому функция  $g(x)$  в формуле Ньютона принимает вид  $\nabla f(\mathbf{x})$ , и уравнение обновления состояния может быть записано в матрично-векторной форме следующим образом:

$$\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}_f^{-1}(\mathbf{x}) \nabla f(\mathbf{x})$$

где  $\mathbf{H}_f(\mathbf{x})$  — представляет собой  $\nabla^2 f$  гессиан (матрицу вторых производных), элементы которого  $H_{ij}$  задаются выражением  $\partial^2 f / \partial x_i \partial x_j$ . Поскольку гессиан имеет  $n^2$  элементов, то метод Ньютона–Рафсона в многомерных пространствах становится дорогостоящим, поэтому было разработано множество способов приближенного решения этой задачи.

Локальные максимумы, хребты и плато создают затруднения в работе методов локального поиска не только в дискретных пространствах состояний, но и в непрерывных пространствах состояний. Для преодоления этих затруднений могут применяться алгоритмы с перезапуском случайным образом и с эмуляцией отжига, которые часто оказываются достаточно полезными. Тем не менее многомерные непрерывные пространства характеризуются очень большим объемом, в котором легко потеряться.

Последней темой, с которой необходимо кратко ознакомиться, является  $\nabla$  **оптимизация с ограничениями**. Задача оптимизации называется задачей оптимизации с ограничениями, если решения в ней должны удовлетворять некоторым жестким ограничениям на значения каждой переменной. Например, в рассматриваемой задаче с размещением аэропортов можно ввести ограничения, чтобы места для аэропортов находились в пределах Румынии, причем на суше (а не, допустим, на середине озера). Сложность задач оптимизации с ограничениями зависит от характера ограничений и от целевой функции. Наиболее широко известной категорией таких задач являются задачи  $\nabla$  **линейного программирования**, в которых ограничения должны представлять собой линейные неравенства, образующие выпуклую область, а целевая функция также является линейной. Задачи линейного программирования могут быть решены за время, полиномиально зависящее от количества переменных. Кроме того, проводились исследования задач с другими типами ограничений и целевых функций: квадратичное программирование, коническое программирование второго порядка и т.д.

## 4.5. ПОИСКОВЫЕ АГЕНТЫ, ДЕЙСТВУЮЩИЕ В ОПЕРАТИВНОМ РЕЖИМЕ, И НЕИЗВЕСТНЫЕ ВАРИАНТЫ СРЕДЫ

До сих пор в данной книге изложение было сосредоточено на описании агентов, в которых используются алгоритмы  $\nabla$  **поиска в автономном режиме**. Эти агенты вычисляют полное решение, прежде чем ступить в реальный мир (см. листинг 3.1), а затем выполняют это решение, не обращаясь к данным своих восприятий. В отличие от этого любой агент, выполняющий  $\nabla$  **поиск в оперативном режиме**<sup>14</sup>,

<sup>14</sup> В компьютерных науках термин “работающий в оперативном режиме” обычно используется по отношению к алгоритмам, которые должны обрабатывать входные данные по мере их получения, а не ожидать, пока станет доступным все множество входных данных.

функционирует по методу **чередования** вычислений и действий: вначале предпринимает действие, затем обозревает среду и вычисляет следующее действие. Поиск в оперативном режиме целесообразно применять в динамических или полудинамических проблемных областях; таковыми являются проблемные области, в которых назначается штраф за то, что агент ведет себя пассивно и вычисляет свои действия слишком долго. Еще более оправданным является использование поиска в оперативном режиме в стохастических проблемных областях. Вообще говоря, результаты любого поиска в автономном режиме должны сопровождаться экспоненциально большим планом действий в непредвиденных ситуациях, в котором учитываются все возможные варианты развития событий, тогда как при поиске в оперативном режиме необходимо учитывать лишь то, что действительно происходит. Например, агент, играющий в шахматы, должен быть настолько хорошо проконсультирован, чтобы мог сделать свой первый ход задолго до того, как станет ясен весь ход игры.

Применение поиска в оперативном режиме является необходимым при решении любой **задачи исследования**, в которой агенту не известны состояния и действия. Агент, находящийся в таком положении полного неведения, должен использовать свои действия в качестве экспериментов для определения того, что делать дальше, и поэтому вынужден чередовать вычисления и действия.

Каноническим примером применения поиска в оперативном режиме может служить робот, который помещен в новое здание и должен его исследовать, чтобы составить карту, которую может затем использовать для перехода из точки *A* в точку *B*. Примерами алгоритмов поиска в оперативном режиме являются также методы выхода из лабиринтов (как известно, такие знания всегда были нужны вдохновляющим нас на подвиги героям древности). Однако исследование пространства — это не единственная форма познания окружающего мира. Рассмотрим поведение новорожденного ребенка: в его распоряжении есть много возможных действий, но он не знает, к чему приведет выполнение какого-либо из них, а эксперименты проводит лишь в немногих возможных состояниях, которых он может достичь. Постепенное изучение ребенком того, как устроен мир, отчасти представляет собой процесс поиска в оперативном режиме.

### Задачи поиска в оперативном режиме

Любая задача поиска в оперативном режиме может быть решена только агентом, выполняющим и вычисления, и действия, а не осуществляющим лишь вычислительные процессы. Предполагается, что агент обладает только описанными ниже знаниями.

- Функция  $\text{Actions}(s)$ , которая возвращает список действий, допустимых в состоянии  $s$ .
- Функция стоимости этапа  $c(s, a, s')$ ; следует отметить, что она не может использоваться до тех пор, пока агент не знает, что результатом является состояние  $s'$ .
- Функция  $\text{Goal-Test}(s)$ .

Следует, в частности, отметить, что агент не может получить доступ к преемникам какого-либо состояния, иначе чем путем фактического опробования всех действий в этом состоянии. Например, в задаче с лабиринтом, показанной на рис. 4.12,

агент не знает, что переход в направлении *Up* из пункта (1, 1) приводит в пункт (1, 2), а выполнив это действие, не знает, позволит ему действие *Down* вернуться назад в пункт (1, 1). Такая степень неведения в некоторых приложениях может быть уменьшена, например, робот-исследователь может знать, как работают его действия по передвижению, и оставаться в неведении лишь в отношении местонахождения препятствий.

Мы будем предполагать, что агент всегда может распознать то состояние, которое он уже посещал перед этим, кроме того, будем руководствоваться допущением, что все действия являются детерминированными. (Последние два допущения будут ослаблены в главе 17.) Наконец, агент может иметь доступ к некоторой допустимой эвристической функции  $h(s)$ , которая оценивает расстояние от текущего состояния до целевого. Например, как показано на рис. 4.12, агент может знать местонахождение цели и быть способным использовать эвристику с манхэттенским расстоянием.

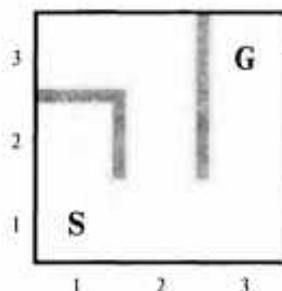


Рис. 4.12. Простая задача с лабиринтом. Агент начинает движение с квадрата *S* и должен достичь квадрата *G*, но ничего не знает о самой среде

Как правило, назначение агента состоит в том, чтобы достичь целевого состояния, минимизируя при этом стоимость. (Еще одно возможное назначение агента может заключаться в том, чтобы он просто исследовал всю эту среду.) Стоимость представляет собой суммарную стоимость пути, фактически пройденного агентом. Обычно принято сравнивать эту стоимость со стоимостью пути, по которому следовал бы агент, если бы он заранее знал пространство поиска, т.е. со стоимостью фактического кратчайшего пути (или кратчайшего полного обследования). В терминологии алгоритмов поиска в оперативном режиме это соотношение называется **коэффициентом конкурентоспособности**; желательно, чтобы этот коэффициент был как можно меньше.

Хотя такое требование по минимизации коэффициента конкурентоспособности может показаться резонным, можно легко доказать, что в некоторых случаях наилучший достижимый коэффициент конкурентоспособности (competitive ratio) является бесконечным. Например, если некоторые действия необратимы, то поиск в оперативном режиме может в конечном итоге перейти в тупиковое состояние, из которого не достижимо целевое состояние. Возможно, читатель сочтет выражение “в конечном итоге” неубедительным; в конце концов, должен же существовать такой алгоритм, который окажется способным не упираться в тупик в ходе проведения исследований с его помощью! Поэтому уточним приведенное выше утверждение таким образом: **ни один алгоритм не позволяет избежать тупиков во всех возможных**

пространствах состояний. Рассмотрим два пространства состояний с тупиками, показанные на рис. 4.13, а. Для алгоритма поиска в оперативном режиме, который посетил состояния  $S$  и  $A$ , оба эти пространства состояний представляются идентичными, поэтому он должен принять одинаковое решение в обоих из них. Поэтому в одном из этих состояний алгоритм потерпит неудачу. Это — один из примеров **⚡ возражения противника** (adversary argument), поскольку легко себе представить, как противник формирует пространство состояний в ходе того, как это пространство исследуется агентом, и может размещать цели и тупики везде, где пожелает.

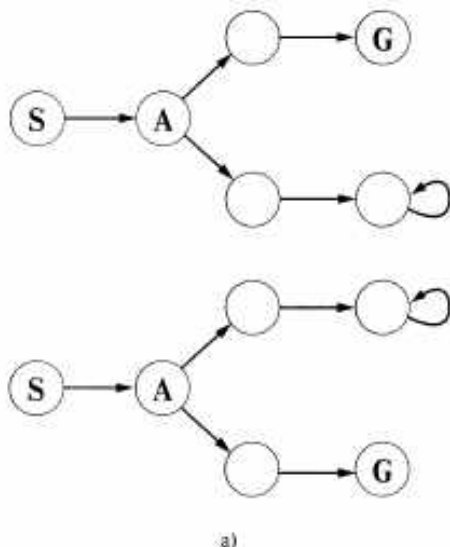


Рис. 4.13. Примеры ситуаций, характеризующихся бесконечным значением коэффициента конкурентоспособности: два пространства состояний, которые способны завести агента, выполняющего поиск в оперативном режиме, в тупик; любой конкретный агент потерпит неудачу, по меньшей мере, в одном из этих пространств (а); двухмерная среда, которая может вынудить агента, выполняющего поиск в оперативном режиме, следовать по маршруту к цели, который может оказаться сколь угодно неэффективным; какой бы выбор ни сделал агент, противник блокирует его маршрут еще одной длинной, тонкой стеной, чтобы путь, по которому следует агент, стал намного длиннее по сравнению с наилучшим возможным путем (б)

Тупики представляют собой одну из реальных сложностей в области исследований, проводимых с помощью робота, поскольку лестницы, пандусы, обрывы и многие другие формы естественного ландшафта создают предпосылки для необратимых действий. Для того чтобы добиться прогресса, мы будем предполагать, что данное пространство состояний является **⚡ безопасно исследуемым**, т.е. что некоторые целевые состояния достижимы из каждого достижимого состояния. Пространства состояний с обратимыми действиями, такие как лабиринты и задачи игры в восемь, могут рассматриваться как неориентированные графы и, вполне очевидно, являются безопасно исследуемыми.

Но даже в безопасно исследуемых вариантах среды нельзя гарантировать достижение какого-либо ограниченного значения коэффициента конкурентоспособности, если в них имеются маршруты с неограниченной стоимостью. Это утверждение легко доказать применительно к вариантам среды с необратимыми действиями, но

фактически оно остается истинным также и для обратимого случая (см. рис. 4.13, б). По этой причине обычно принято описывать производительность алгоритмов поиска в оперативном режиме с учетом размеров всего пространства состояний, а не только глубины самой поверхностной цели.

### Агенты, выполняющие поиск в оперативном режиме

После каждого действия агент, выполняющий поиск в оперативном режиме, получает результаты акта восприятия, с помощью которых может узнать, какого состояния он достиг; на основании этой информации агент может дополнить карту своей среды. Текущая карта используется для принятия решения о том, куда двигаться дальше. Такое чередование планирования и выполнения действий означает, что алгоритмы поиска в оперативном режиме весьма значительно отличаются от алгоритмов поиска в автономном режиме, которые рассматривались выше. Например, алгоритмы поиска в автономном режиме, такие как  $A^*$ , обладают способностью развертывать узел в одной части пространства, а затем немедленно развертывать узел в другой части пространства, поскольку для развертывания узла требуются моделируемые, а не реальные действия. С другой стороны, любой алгоритм поиска в оперативном режиме позволяет агенту развертывать только тот узел, который он физически занимает. Для предотвращения необходимости путешествовать через все дерево, чтобы развернуть следующий узел, представляется более удобным развертывание узлов в локальном порядке. Именно таким свойством обладает поиск в глубину, поскольку (за исключением операции возврата) следующий развертываемый узел является дочерним узлом ранее развернутого узла.

Алгоритм агента, выполняющего поиск в глубину в оперативном режиме, приведен в листинге 4.5. Агент хранит составленную им карту в таблице `result[a, s]`, в которой регистрируются состояния, явившиеся следствием выполнения действия `a` в состоянии `s`. Этот агент предпринимает попытку выполнения из текущего состояния любого действия, которое еще не было исследовано в этом состоянии. Сложности возникают после того, как агент опробовал все действия в некотором состоянии. При поиске в глубину в автономном режиме это состояние удаляется из очереди, а при поиске в оперативном режиме агент должен выполнить возврат физически. При поиске в глубину это равносильно возврату в последнее по времени состояние, из которого агент перешел в текущее состояние. Соответствующая организация работы обеспечивается за счет ведения такой таблицы, где для каждого состояния перечисляются состояния-предшественники, к которым агент еще не выполнял возврат. Если агент исчерпал список состояний, к которым может выполнить возврат, это означает, что проведенный агентом поиск является полным.

**Листинг 4.5.** Агент, выполняющий поиск в оперативном режиме, который проводит исследование с использованием поиска в глубину. Этот агент может применяться только в пространствах двуправленного поиска

---

```
function Online-DFS-Agent(s') returns действие action
  inputs: s', восприятие, позволяющее идентифицировать
           текущее состояние
  static: result, таблица, индексированная по действиям и
           состояниям, первоначально пустая
```

---

*unexplored*, таблица, в которой для каждого посещенного состояния перечислены еще не опробованные действия  
*unbacktracked*, таблица, в которой для каждого посещенного состояния перечислены еще не опробованные возвраты  
*s*, *a*, предыдущие состояние и действие, первоначально неопределенные

```

if Goal-Test(s') then return stop
if s' представляет собой новое состояние
    then unexplored[s'] ← Actions(s')
if s не является неопределенным then do
    result[a, s] ← s'
    добавить s в начало таблицы unbacktracked[s']
if элемент unexplored[s'] пуст then
    if элемент unbacktracked[s'] пуст then return stop
    else a ← действие b, такое что
        result[b, s'] = Pop(unbacktracked[s'])
else a ← Pop(unexplored[s'])
s ← s'
return a

```

Рекомендуем читателю провести трассировку хода выполнения процедуры Online-DFS-Agent применительно к лабиринту, показанному на рис. 4.12. Можно довольно легко убедиться в том, что в наихудшем случае агент в конечном итоге пройдет по каждой связи в данном пространстве состояний точно два раза. При решении задачи обследования такая организация работы является оптимальной; а при решении задачи поиска целей, с другой стороны, коэффициент конкурентоспособности может оказаться сколь угодно неэффективным, если агент будет отправляться в длительные экскурсии, притом что целевое состояние находится непосредственно рядом с начальным состоянием. Такая проблема решается с использованием варианта метода итеративного углубления для работы в оперативном режиме; в среде, представляющей собой однородное дерево, коэффициент конкурентоспособности подобного агента равен небольшой константе.

В связи с тем что в алгоритме Online-DFS-Agent используется метод с возвратами, он может применяться только в таких пространствах состояний, где действия являются обратимыми. Существуют также немного более сложные алгоритмы, применимые в пространствах состояний общего вида, но ни один из подобных алгоритмов не характеризуется ограниченным коэффициентом конкурентоспособности.

### Локальный поиск в оперативном режиме

Как и поиск в глубину, поиск с восхождением к вершине обладает свойством локализации применительно к операциям развертывания в нем узлов. В действительности сам поиск с восхождением к вершине уже можно считать алгоритмом поиска в оперативном режиме, поскольку предусматривает хранение в памяти только одного текущего состояния! К сожалению, в своей простейшей форме этот алгоритм не очень полезен, так как оставляет агента в таком положении, что последний не может покинуть локальный максимум и отправиться куда-то еще. Более того, в этом алго-

ритме не может использоваться перезапуск случайным образом, поскольку агент не способен перенести самого себя в новое состояние.

Вместо перезапуска случайным образом для исследования среды может быть предусмотрено использование **случайного блуждания** (random walk). В методе случайного блуждания просто выбирается случайным образом одно из действий, доступных из текущего состояния; предпочтение отдается действиям, которые еще не были опробованы. Легко доказать, что метод случайного блуждания позволяет агенту в конечном итоге найти цель или выполнить свою задачу исследования, при условии, что пространство является конечным<sup>15</sup>. С другой стороны, этот процесс может оказаться очень продолжительным. На рис. 4.14 показана среда, в которой для поиска цели с помощью метода случайного блуждания может потребоваться количество этапов, измеряемое экспоненциальной зависимостью, поскольку на каждом этапе вероятность шага назад вдвое превышает вероятность шага вперед. Безусловно, что этот пример является надуманным, но существует много реальных пространств состояний, топология которых способствует возникновению “ловушек” такого рода при случайном блуждании.

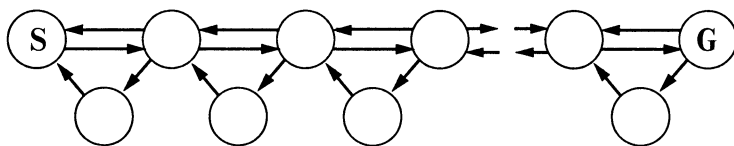


Рис. 4.14. Среда, в которой применение метода случайного блуждания для поиска цели требует выполнения такого количества этапов, которое определяется экспоненциальной зависимостью

Как оказалось, более эффективным подходом является дополнение метода поиска с восхождением к вершине способностью запоминать, а не способностью выбирать следующее действие случайным образом. Основная идея состоит в том, что необходимо хранить в памяти “текущую наилучшую оценку”  $H(s)$  стоимости достижения цели из каждого состояния, которое уже было посещено. На первых порах  $H(s)$  представляет собой только эвристическую оценку  $h(s)$  и обновляется по мере того, как агент приобретает опыт в исследовании пространства состояний. На рис. 4.15 показан простой пример одномерного пространства состояний. На рис. 4.15, а показано, что агент, по-видимому, зашел в тупик, попав в плоский локальный минимум, соответствующий затененному состоянию. Вместо того чтобы оставаться там, где находится, агент должен последовать по тому пути, который может рассматриваться как наилучший путь к цели согласно текущим оценкам стоимостей для соседних состояний. Оценка стоимости достижения цели через соседнее состояние  $s'$  равна оценке стоимости достижения  $s'$ , которая складывается с оценкой стоимости достижения цели из  $s'$ , т.е. равна  $c(s, a, s') + H(s')$ . В данном примере имеются два действия, с оценками стоимости 1+9 и 1+2, поэтому, по всей видимости, лучше всего двигаться вправо. После этого становится очевидно, что оценка стоимости для

<sup>15</sup> Этот вариант бесконечного пространства фактически является гораздо более сложным. Методы случайного блуждания являются полными в бесконечных одно- и двухмерных решетках, но не в трехмерных! В последнем случае вероятность того, что блуждающий агент в конечном итоге возвратится в начальную точку, составляет лишь около 0,3405 (для ознакомления с общим введением в эту тему см. [703]).

этого затененного состояния, равная 2, была слишком оптимистической. Поскольку стоимость наилучшего хода равна 1 и он ведет в состояние, которое находится по меньшей мере на расстоянии 2 шагов от цели, то затененное состояние должно находиться по меньшей мере на расстоянии 3 шагов от цели, поэтому его оценка  $H$  должна быть обновлена соответствующим образом, как показано на рис. 4.15, б. Продолжая этот процесс, агент перейдет вперед и назад еще дважды, каждый раз обновляя оценку  $H$  и “сглаживая” локальный минимум до тех пор, пока не вырвется из него вправо.

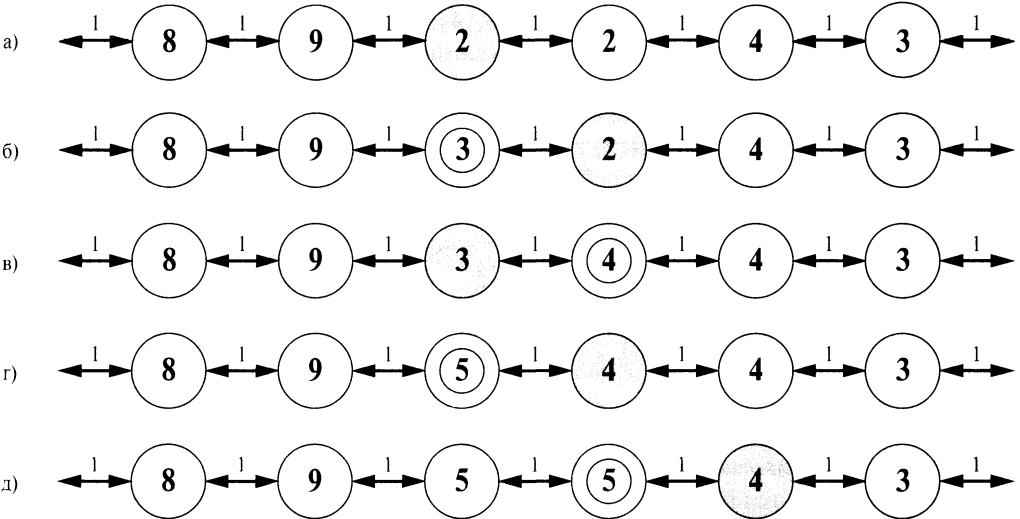


Рис. 4.15. Пять итераций алгоритма  $LRTA^*$  в одномерном пространстве состояний. Каждое состояние обозначено значением  $H(s)$ , текущей оценкой стоимости достижения цели, а каждая дуга обозначена соответствующей ей стоимостью этапа. Затененное состояние показывает местонахождение агента, а значения, обновленные после каждой итерации, обозначаются двойным кружком

Алгоритм агента, в котором реализована эта схема, получивший название поиска  $A^*$  в реальном времени с обучением (Learning Real-Time  $A^*$  —  $\approx LRTA^*$ ), показан в листинге 4.6. Как и в алгоритме  $Online\text{-}DFS\text{-}Agent$ , в данном алгоритме составляется карта среды с использованием таблицы *result*. Этот алгоритм обновляет оценку стоимости для состояния, которое он только что оставил, а затем выбирает ход, “который представляется наилучшим” в соответствии с его текущими оценками стоимости. Следует отметить одну важную деталь, что действия, которые еще не были опробованы в состоянии  $s$ , всегда рассматриваются как ведущие непосредственно к цели с наименьшей возможной стоимостью, а именно  $h(s)$ . Такой **оптимизм в отношении неопределенности** побуждает агента исследовать новые пути, которые могут действительно оказаться перспективными.

Листинг 4.6. Функция  $LRTA^*\text{-}Agent$  выбирает действие в соответствии со значениями соседних состояний, которые обновляются по мере того, как агент передвигается в пространстве состояний

```
function LRTA*-Agent( $s'$ ) returns действие  $a$ 
  inputs:  $s'$ , восприятие, позволяющее идентифицировать
           текущее состояние
```



```

static: result, таблица, индексированная по действиям и состояниям,
        первоначально пустая
        H, таблица оценок стоимостей, индексированная по состояниям,
        первоначально пустая
        s, a, предыдущие состояние и действие, первоначально
        неопределенные

if Goal-Test(s') then return stop
if s' является одним из новых состояний (отсутствующим в H)
    then H[s']  $\leftarrow h(s')$ 
unless s является неопределенным
    result[a, s]  $\leftarrow s'$ 

    H[s]  $\leftarrow \min_{b \in \text{bActions}(s)} \text{LRTA}^* \text{-Cost}(s, b, \text{result}[b, s], H)$ 

    a  $\leftarrow$  одно из действий b среди действий Actions(s'), которое
    минимизирует значение  $\text{LRTA}^* \text{-Cost}(s', b, \text{result}[b, s'], H)$ 
    s  $\leftarrow s'$ 
return a

function LRTA*-Cost(s, a, s', H) returns оценка стоимости
    if s' является неопределенным then return h(s)
    else return c(s, a, s') + H[s']

```

---

Гарантируется, что агент LRTA\* найдет цель в любой конечной, безопасно исследуемой среде. Однако, в отличие от A\*, в бесконечных пространствах состояний применяемый в этом агенте алгоритм становится неполным, поскольку в некоторых случаях этот алгоритм может вовлечь агента в бесконечные блуждания. В наихудшем случае данный алгоритм позволяет исследовать среду с  $n$  состояниями за  $O(n^2)$  этапов, но чаще всего действует намного лучше. Агент LRTA\* представляет собой лишь один пример из большого семейства агентов, выполняющих поиск в оперативном режиме, которые могут быть определены путем задания правила выбора действия и правила обновления различными способами. Это семейство агентов, которое было первоначально разработано для стохастических вариантов среды, рассматривается в главе 21.

## Обучение в ходе поиска в оперативном режиме

То, что агенты, выполняющие поиск в оперативном режиме, на первых порах находятся в полном неведении, открывает некоторые возможности для обучения. Во-первых, агенты изучают “карту” своей среды (а точнее, результаты каждого действия в каждом состоянии), регистрируя результаты каждого из своих опытов. (Обратите внимание на то, что из предположения о детерминированности среды следует, что для изучения любого действия достаточно проведения одного эксперимента.) Во-вторых, агенты, выполняющие локальный поиск, получают все более точные оценки значения каждого состояния, используя локальные правила обновления, как в алгоритме LRTA\*. В главе 21 будет показано, что в конечном итоге такие обновления сходятся к точным значениям для каждого состояния, при условии, что агент исследует пространство состояний правильным способом. А после того как станут известными точные значения, оптимальные решения могут быть приняты путем перемещения к прежнему с наи-

высшим значением, т.е. в таком случае оптимальной стратегией становится метод поиска с восхождением к вершине в чистом виде.

Если читатель выполнил нашу рекомендацию провести трассировку поведения алгоритма Online-DFS-Agent в среде, показанной на рис. 4.12, то должен был заметить, что этот агент не слишком умен. Например, после того как агент обнаружил, что действие *Up* ведет из пункта (1, 1) в пункт (1, 2), он еще не знает, что действие *Down* возвратит его назад в пункт (1, 1) или что следующее действие *Up* приведет его из пункта (2, 1) в пункт (2, 2), из пункта (2, 2) в пункт (2, 3) и т.д. Вообще говоря, было бы желательно, чтобы агент освоил в результате обучения, что действие *Up* приводит к увеличению координаты *y*, если на этом пути нет стены, и что действие *Down* приводит к уменьшению этой координаты, и т.д. Для того чтобы это произошло, требуются две составляющие. Во-первых, необходимо формальное и явно манипулируемое представление общих правил такого рода; до сих пор мы скрывали эту информацию внутри “черного ящика”, называемого *функцией определения преемника*. Данному вопросу посвящена часть III. Во-вторых, нужны алгоритмы, позволяющие формировать подходящие общие правила из конкретных наблюдений, сделанных агентом. Эта тема рассматривается в главе 18.

## РЕЗЮМЕ

В данной главе рассматривалось применение **эвристических функций** для уменьшения стоимости поиска. В ней исследовался целый ряд алгоритмов, в которых используются эвристические функции, и было установлено, что даже при использовании хороших эвристических функций достижение оптимальности связано с увеличением стоимости поиска.

- **Поиск по первому наилучшему совпадению** сводится просто к применению алгоритма Graph-Search, в котором для развертывания выбираются неразвернутые узлы с минимальной стоимостью (в соответствии с некоторым критерием). В алгоритмах поиска по первому наилучшему совпадению обычно используется **эвристическая** функция  $h(n)$ , которая оценивает стоимость достижения решения из узла  $n$ .
- При **жадном поиске по первому наилучшему совпадению** развертываются узлы с минимальным значением  $h(n)$ . Этот поиск не оптимален, но часто является эффективным.
- При **поиске A\*** развертываются узлы с минимальным значением  $f(n) = g(n) + h(n)$ . Алгоритм A\* является полным и оптимальным, при условии, что гарантирована допустимость (для алгоритма Tree-Search) или преэминентность (для алгоритма Graph-Search) функции  $h(n)$ . Пространственная сложность алгоритма A\* все еще остается слишком высокой.
- Производительность алгоритмов эвристического поиска зависит от качества эвристической функции. Иногда хорошие эвристические функции можно составить путем ослабления определения задачи, предварительного вычисления стоимости решения подзадач и сохранения этой информации в базе данных шаблонов или обучения на основе опыта решения данного класса задач.

- Алгоритмы **RBFS** и **SMA\*** представляют собой надежные, оптимальные алгоритмы поиска, в которых используются ограниченные объемы памяти; при наличии достаточного количества времени эти алгоритмы могут решать такие задачи, которые не позволяет решать алгоритм **A\***, поскольку исчерпывает доступную память.
- Такие методы локального поиска, как **поиск с восхождением к вершине**, действуют на основе формулировок полного состояния и предусматривают хранение в памяти лишь небольшого количества узлов. Было разработано также несколько стохастических алгоритмов, включая поиск с **эмуляцией отжига**, которые возвращают оптимальные решения при наличии подходящего графика “охлаждения” (т.е. графика постепенного уменьшения величины случайного разброса). Кроме того, многие методы локального поиска могут использоваться для решения задач в непрерывных пространствах.
- **Генетический алгоритм** представляет собой стохастический поиск с восхождением к вершине, в котором сопровождается большая популяция состояний. Новые состояния формируются с помощью **мутации** и **скрещивания**; в последней операции комбинируются пары состояний, взятых из этой популяции.
- **Проблемы исследования** возникают, если агент не имеет никакого представления о том, каковы состояния и действия в его среде. Для безопасно исследуемых вариантов среды агенты, выполняющие **поиск в оперативном режиме**, могут составить карту и найти цель, если она существует. Эффективным методом выхода из локальных минимумов является обновление эвристических оценок на основе опыта.

## БИБЛИОГРАФИЧЕСКИЕ И ИСТОРИЧЕСКИЕ ЗАМЕТКИ

Пример использования эвристической информации при решении задач впервые появился в одной из ранних статей Саймона и Ньюэлла [1419], но само выражение “эвристический поиск” и обоснование применения эвристических функций, которые оценивают расстояние до цели, появились немного позже [932], [1126]. Доран и Мичи [404] осуществили обширные экспериментальные исследования в области применения эвристического поиска к решению ряда задач, в частности задач игры в восемь и игры в пятнадцать. Хотя Доран и Мичи провели теоретический анализ длины пути и “проникновения” (penetrance) (отношения длины пути к общему количеству узлов, исследованных к данному моменту) в процессе эвристического поиска, они, по-видимому, игнорировали ту информацию, которую может предоставить текущая длина пути. Алгоритм **A\***, предусматривающий использование в эвристическом поиске текущего значения длины пути, был разработан Хартом, Нильссоном и Рафаэлем [623], с некоторыми дальнейшими поправками [624]. Декстер и Перл [371] продемонстрировали оптимальную эффективность алгоритма **A\***.

В указанной выше оригинальной статье, посвященной алгоритму **A\***, было впервые представлено условие преемственности (consistency condition), которому должны удовлетворять эвристические функции. В качестве более простой замены этого условия Поллом [1223] было предложено условие монотонности, но Перл [1188] показал, что эти два условия эквивалентны. Аналоги открытых и закрытых списков использовались

во многих алгоритмах, явившихся предшественниками алгоритма  $A^*$ ; к ним относятся алгоритмы поиска в ширину, поиска в глубину и поиска по критерию стоимости [97], [399]. Важность применения аддитивных стоимостей путей для упрощения алгоритмов оптимизации особо ярко была показана в работах Беллмана [97].

Пол [1220], [1223] впервые провел исследования связи между ошибками в эвристических функциях и временной сложностью алгоритма  $A^*$ . Доказательство того, что работа алгоритма  $A^*$  завершается за линейное время, если ошибка в эвристической функции ограничена некоторой константой, можно найти в работах Пола [1223] и Гашнига [522]. Перл [1188] развил этот результат, что позволило учитывать логарифмический рост ошибки. Применение “эффективного коэффициента ветвления” в качестве критерия эффективности эвристического поиска было предложено Нильссоном [1141].

Существует много вариантов алгоритма  $A^*$ . Пол [1222] предложил использовать в алгоритме  $A^*$  динамическое взвешивание (dynamic weighting), в котором в качестве функции оценки применяется взвешенная сумма текущей длины пути и эвристической функции  $f_w(n) = w_g g(n) + w_h h(n)$ , а не просто сумма  $f(n) = g(n) + h(n)$ . Веса  $w_g$  и  $w_h$  корректируются динамически по мере развития поиска. Можно доказать, что алгоритм Пола является  $\epsilon$ -допустимым (т.е. гарантирует нахождение решений с отклонением от оптимального решения на коэффициент  $1+\epsilon$ ), где  $\epsilon$  — параметр, предусмотренный в алгоритме. Таким же свойством обладает алгоритм  $A_\epsilon^*$  [1188], который способен выбрать из периферии любой узел, при условии, что его  $f$ -стоимость находится в пределах коэффициента  $1+\epsilon$  от стоимости периферийного узла с наименьшей  $f$ -стоимостью. Выбор соответствующего коэффициента может быть сделан таким образом, чтобы существовала возможность минимизировать стоимость поиска.

Алгоритм  $A^*$  и другие алгоритмы поиска в пространстве состояний тесно связаны с методами ветвей и границ, которые широко используются в исследованиях операций [901]. Соотношения между алгоритмами поиска в пространстве состояний и методами ветвей и границ были глубоко исследованы в [867], [869], [1115]. Мартелли и Монтанари [990] показали связь между динамическим программированием (см. главу 17) и некоторыми типами поиска в пространстве состояний. Кумар и Канал [868] предприняли попытку “великой унификации” методов эвристического поиска, динамического программирования, а также методов ветвей и границ под общим названием CDP (Composite Decision Process — комплексный процесс принятия решений).

Поскольку компьютеры в конце 1950-х — начале 1960-х годов имели не больше нескольких тысяч слов оперативной памяти, темой ранних исследовательских работ часто служил эвристический поиск с ограничением памяти. Одна из самых первых программ поиска, Graph Traverser [404], фиксирует свои результаты в виде некоторого оператора после выполнения поиска по первому наилучшему совпадению вплоть до заданного предела объема памяти. Алгоритм  $IDA^*$  [835], [836] стал одним из первых широко применяемых оптимальных алгоритмов эвристического поиска с ограничением памяти, после чего было разработано большое количество его вариантов. Анализ эффективности алгоритма  $IDA^*$  и сложностей, возникающих при его применении с эвристическими функциями, которые встречаются в реальных задачах, приведен в работе Патрика и др. [1182].

Алгоритм RBFS [840], [841] фактически является лишь немного более сложным по сравнению с алгоритмом, приведенным в листинге 4.1. Последний вари-