2 Основы поиска

В этой главе

- ✓ Понятие планирования и поиска
- ✓ Определение задач, для решения которых подходят поисковые алгоритмы
- ✓ Представление пространства задач подходящим для обработки алгоритмами способом
- ✓ Понимание и проектирование базовых алгоритмов поиска для решения задач

Что мы планируем и ищем?

Способность планировать до начала действий — это отличительная черта интеллекта. Прежде чем отправиться в поездку в другую страну, начать новый проект или написать функцию в коде, мы составляем план действий. *Планирование* происходит на разных уровнях детализации, что позволяет получить наилучший возможный результат при выполнении задач, необходимых для достижения целей (рис. 2.1).

Планы редко реализуются так, как было изначально задумано. Мы живем в постоянно меняющейся среде, поэтому просто невозможно учесть все переменные и неизвестные. Мы практически всегда отклоняемся от начального плана из-за изменений в предметной области. Нам приходится менять планы, и иногда не раз, когда мы уже прошли несколько этапов, но происходят неожиданные

события, которые приходится учитывать для достижения цели. В результате фактический план, как правило, отличается от начального.

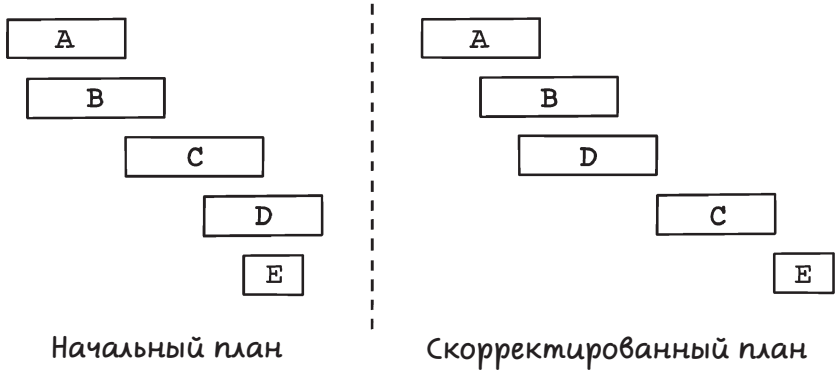


Рис. 2.1. Пример изменения плана проекта

Поиск — это способ управлять планированием путем разделения плана на этапы. Например, когда мы планируем путешествие, то ищем интересующие нас маршруты, оцениваем остановки на пути и сопутствующие возможности. Мы также ищем, где жить и чем заняться, исходя из предпочтений и бюджета. В зависимости от результатов всех этих поисков меняется и план.

Предположим, что мы задумали поездку на море, расстояние до которого около 500 километров, и решили сделать две остановки: одну в зоопарке и вторую в пиццерии. По прибытии мы планируем переночевать в домике на берегу и поучаствовать в трех мероприятиях. Путь до места назначения займет приблизительно 8 часов. Мы также решаем сократить путь по платной автостраде, съезд на которую будет после ресторана. Но эта дорога открыта только до 14:00.

Путешествие начинается, и все идет по плану. Мы приезжаем в зоопарк и любуемся удивительными животными. Затем продолжаем путь, и уже подходит время перекусить в ресторане, но по приезде оказывается, что он недавно закрылся. Теперь нужно скорректировать план и найти поблизости другой ресторан или кафе, которые нам подойдут.

Проехав еще какое-то расстояние, мы находим ресторан, с удовольствием съедаем пиццу и возвращаемся на трассу. При подъезде к частной автостраде мы понимаем, что время уже 14:20 и она закрыта. И здесь снова приходится менять план. Мы ищем объездной путь и выясняем, что он добавит к поездке еще 120 километров. Это означает, что нам нужно найти другое место для ночлега еще до прибытия на побережье. Мы находим такое место и меняем маршрут. Из-за потраченного времени мы успеваем поучаствовать только в двух из трех мероприятий. Нам пришлось искать альтернативные решения, и поэтому наш первоначальный план серьезно изменился, но в итоге поездка обернулась очень интересным приключением, и мы прекрасно провели время.

Этот пример показывает, как поиск используется для планирования и влияет на его итог для достижения намеченной цели. Меняются обстоятельства — меняются и цели. При этом путь к ним также неизбежно придется корректировать (рис. 2.2). Изменения в планах практически всегда внезапны и вносятся по мере необходимости.

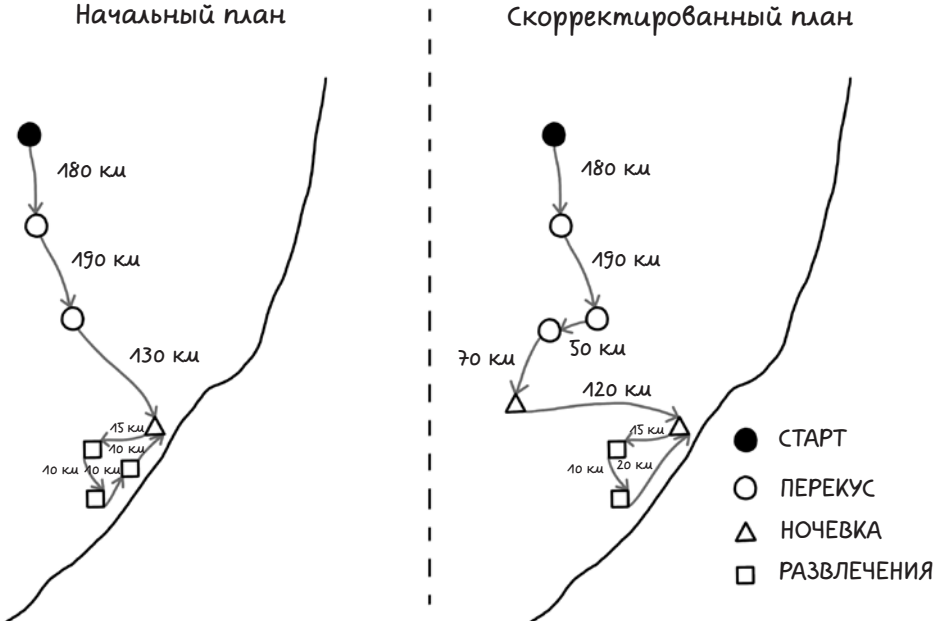


Рис. 2.2. Исходный план поездки и его измененный конечный вариант

Поиск подразумевает оценку будущих состояний на пути к цели для нахождения оптимального пути из этих состояний к намеченной точке. Эта глава посвящена различным подходам к поиску, различающимся по типу решаемых задач. Поиск — не новый, но мощный инструмент разработки интеллектуальных алгоритмов.

Стоимость вычислений: причина использования умных алгоритмов

В программировании функции состоят из операций, и согласно принципу работы современных компьютеров, разные функции требуют разного количества времени на обработку. Чем больше вычислений требуется, тем более дорогостоящей получается функция. Для описания сложности функции или алгоритма используется *нотация «О-большое»*. Эта нотация моделирует количество операций, необходимых при возрастающем размере ввода. Вот некоторые примеры и связанные с ними сложности:

- O(1) — это как пожать руку только одному человеку: занимает одно и то же время, независимо от того, сколько людей находится в комнате. В коде это может быть одна операция, которая выводит «`Hello World`».
- O(n) — это как пожать руку каждому из присутствующих в комнате: если в комнате 100 человек, потребуется 100 рукопожатий. В коде это может быть функция, которая проходит по списку и выводит каждый элемент.
- $O(n^2)$ — это когда каждый в комнате пожимает руку каждому: очень быстро перерастает в хаос. В коде это может быть функция, которая сравнивает каждый элемент списка с каждым другим элементом списка.

На рис. 2.3 отражена различная стоимость алгоритмов. Те из них, которые требуют увеличения количества операций по мере увеличения входных данных, выполняются медленнее всех. При этом алгоритмы, для которых требуется постоянное число операций по мере роста количества входных данных, выполняются быстрее.

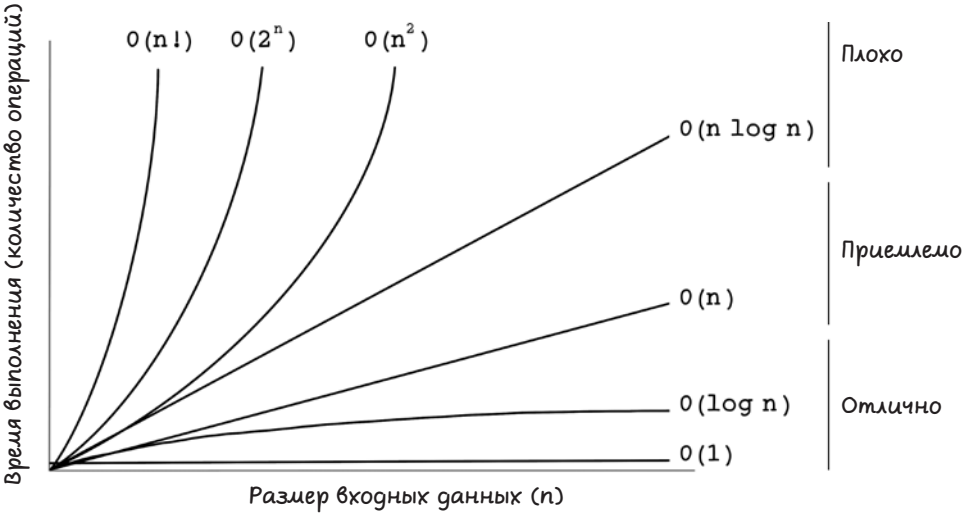


Рис. 2.3. Сложность «О-большое»

ПРИМЕЧАНИЕ Более медленный алгоритм будет работать быстрее, если его можно распараллелить на множестве процессоров (например, на GPU). Мы стремимся к низкой сложности по «О-большому», но также заботимся об использовании памяти и эффективности работы с аппаратным обеспечением.

Понимание того, что различные алгоритмы имеют различную вычислительную стоимость, очень важно, потому что в ее минимизации и состоит сама цель интеллектуальных алгоритмов, способных справляться с задачами быстро и эффективно. Теоретически почти любую задачу можно решить перебором возможных решений, пока не будет найдено лучшее. Но в реальности на по-

добные вычисления ушли бы часы, годы или даже тысячелетия, поэтому они не подходят для практического применения.

Задачи, подходящие для алгоритмов поиска

Практически любую задачу, требующую очередности решений, можно реализовать с помощью поисковых алгоритмов, которые подбираются согласно типу задачи и размеру области поиска. В зависимости от выбранного алгоритма и конфигурации можно найти оптимальное решение, или *лучшее из возможных*. Другими словами, решение может быть хорошим, но оно не обязательно лучшее. Когда мы говорим «хорошее решение» или «оптимальное решение», то подразумеваем степень его эффективности в достижении поставленной цели.

Один из сценариев, в котором полезны алгоритмы поиска, — это поиск кратчайшего пути к цели в лабиринте. Предположим, что мы находимся в квадратном лабиринте размером 10 × 10 клеток, как показано на рис. 2.4. В лабиринте есть цель, которую мы хотим достичь, и препятствия, на которые нельзя наступать или через которые нельзя перешагивать. Задача состоит в том, чтобы найти путь к цели, избегая препятствий и делая как можно меньше шагов, в процессе перемещения на север, юг, восток или запад. В этом примере игрок не может двигаться по диагонали.

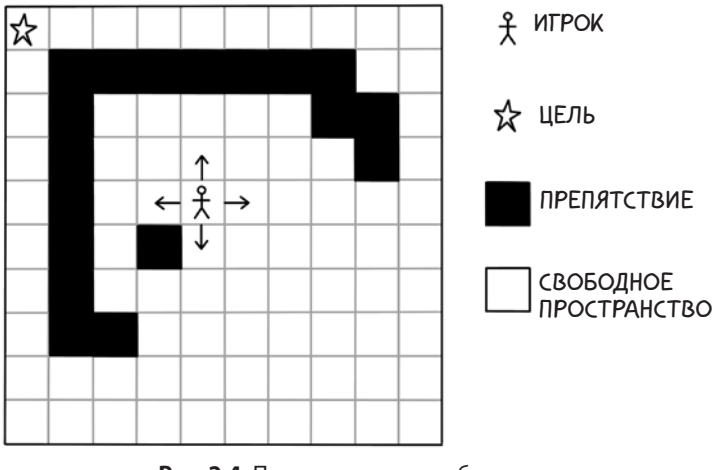


Рис. 2.4. Пример задачи с лабиринтом

Как найти кратчайший путь и обойти препятствия? Оценивая задачу с позиции человека, можно попробовать каждый вариант и подсчитать количество ходов. Таким образом, методом проб и ошибок можно найти кратчайшие пути, учитывая, что лабиринт относительно мал.

На рис. 2.5 показаны некоторые из возможных путей к цели, но обратите внимание, что в первом варианте мы ее не достигаем.

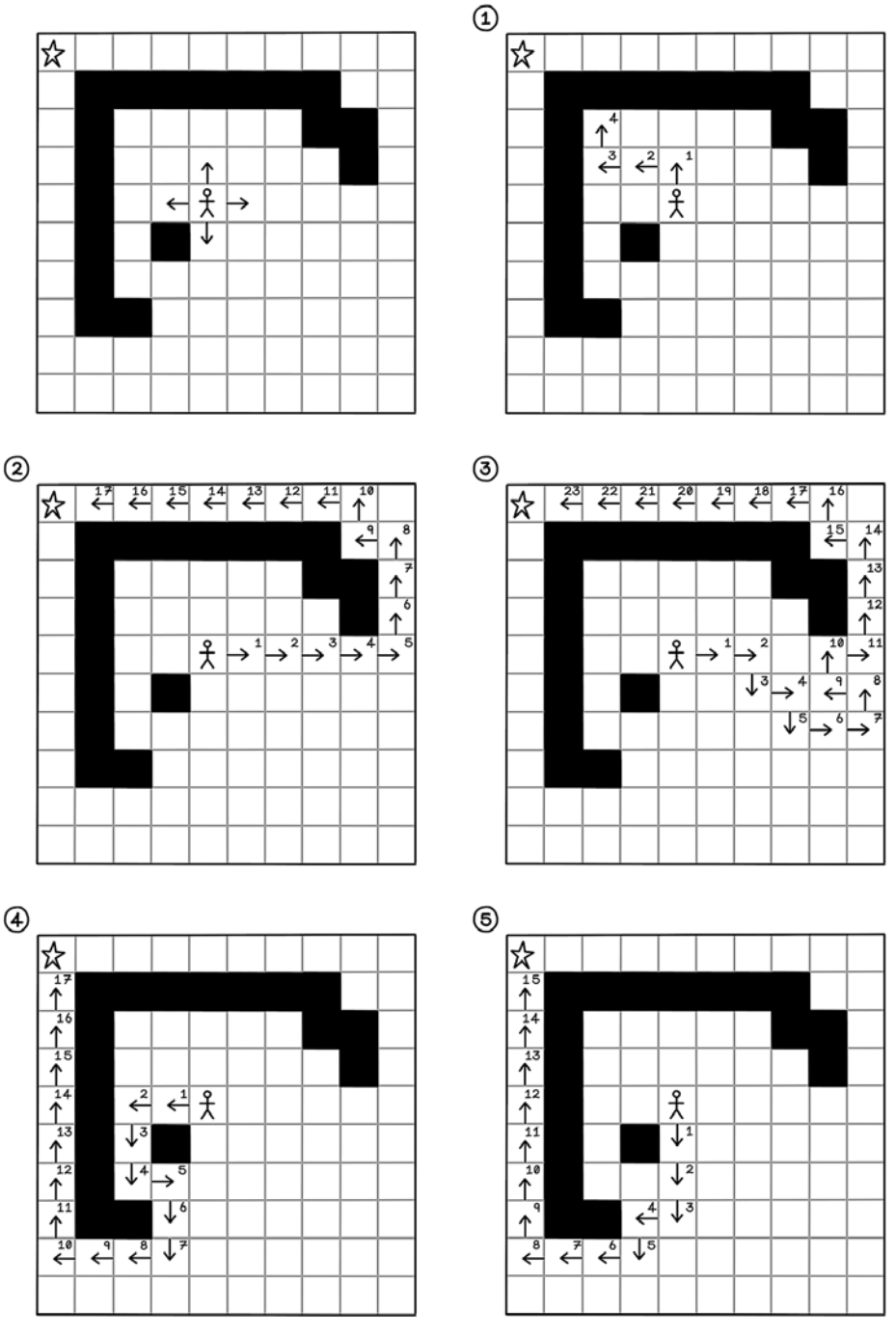


Рис. 2.5. Примеры возможных путей для решения задачи лабиринта

Рассматривая лабиринт и подсчитывая количество клеток в различных направлениях, можно найти несколько решений задачи. Чтобы найти четыре успешных варианта из неизвестного количества возможных, мы предприняли пять попыток. Для вычисления же всех возможных решений потребовалось бы очень много усилий:

- Попытка 1 не является подходящим решением. В ней выполняется 4 хода и цель при этом не достигается.
- Попытка 2 решает задачу, на что уходит 17 ходов.
- Попытка 3 решает задачу и требует совершения 23 ходов.
- Попытка 4 решает задачу снова за 17 ходов.
- Попытка 5 оказывается наилучшим возможным решением, так как требует всего 15 ходов. Несмотря на то что эта попытка оптимальна, она была найдена случайно.

Если бы лабиринт был намного больше, как, например, на рис. 2.6, то чтобы найти кратчайший путь вручную, потребовалось бы огромное количество времени. Тут-то и пригодятся поисковые алгоритмы.

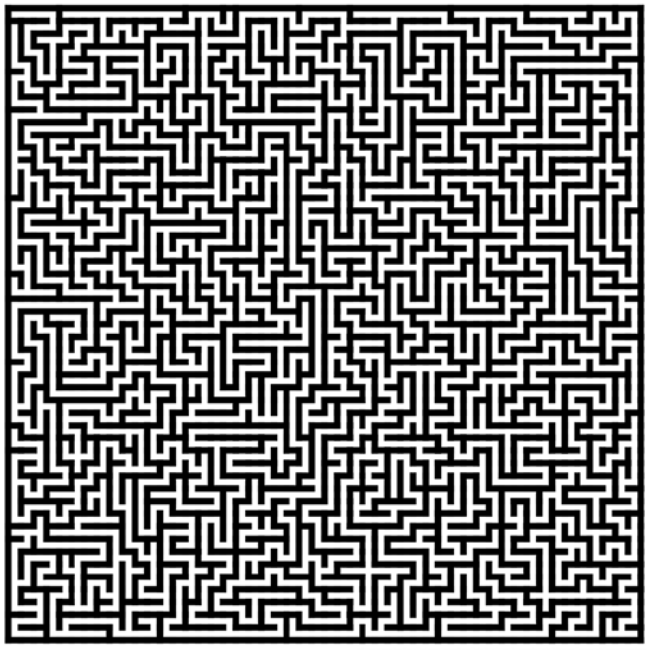


Рис. 2.6. Пример задачи с большим лабиринтом

У людей есть преимущество в том, что мы способны визуально воспринять задачу, понять ее и найти решение на основе заданных параметров. Мы понимаем и интерпретируем данные абстрактно. Компьютер пока что не способен

понимать обобщенную информацию в столь же естественной форме, как это делаем мы. Поэтому пространство задачи нужно представить в виде, который будет пригоден для вычисления и обработки поисковым алгоритмом.

Представление состояния: создание структуры для представления пространства задачи и решений

При выражении данных и информации в понятном для компьютера виде необходимо кодировать их логически, чтобы это понимание было объективным. Несмотря на то что данные будет субъективно кодировать человек, выполняющий эту задачу, необходимо предусмотреть краткий и согласованный способ их представления.

Проясним разницу между данными и информацией. *Данные* — это сырые факты о чем-либо, а *информация* — это интерпретация этих фактов, их анализ для конкретной области. Чтобы информация стала значимой, требуется контекст и обработка данных. Вот пример: каждое отдельное пройденное в лабиринте расстояние представляет собой данные, а сумма всех расстояний является информацией. В зависимости от точки восприятия, уровня детализации и желаемого результата классификация чего-либо как данных либо информации может зависеть от контекста (рис. 2.7).

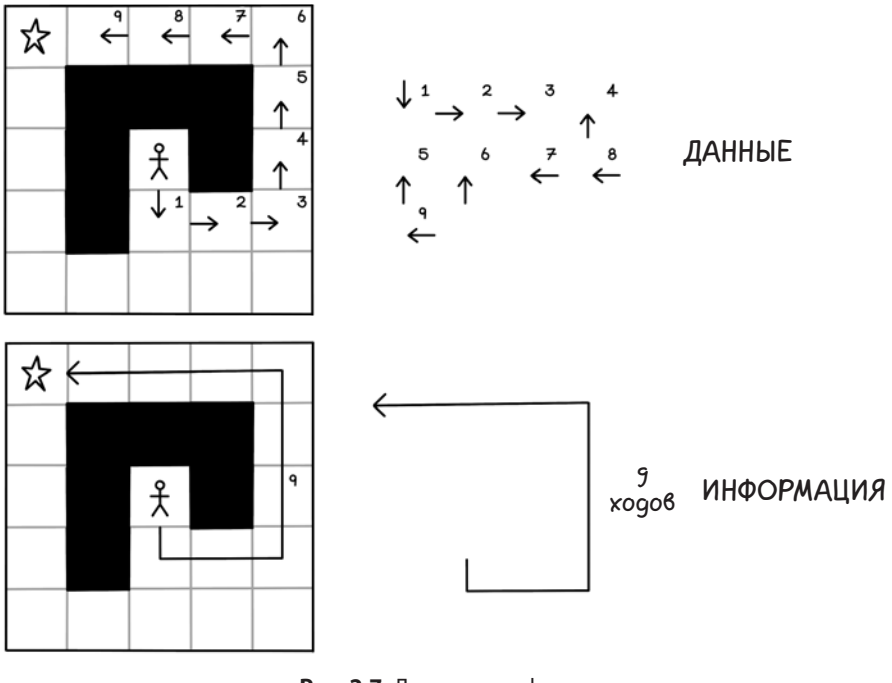


Рис. 2.7. Данные и информация

Структуры данных — это понятие computer science; с помощью структуры данные представляются в таком виде, чтобы их можно было обрабатывать с применением алгоритмов. Это абстрактный тип данных, состоящий из данных и операций, организованных определенным образом. При этом используемая структура определяется контекстом задачи и целью.

Примером такой структуры является *массив*, который представляет собой просто набор данных. Разные типы массивов обладают различными свойствами, которые делают их более подходящими для тех или иных целей. В зависимости от используемого языка программирования массив может содержать значения разных типов либо только одного типа; кроме того, в нем могут быть запрещены повторяющиеся значения. Различные виды массивов, как правило, называются тоже по-разному. Особенности и ограничения различных структур данных ведут к более эффективным вычислениям (рис. 2.8).

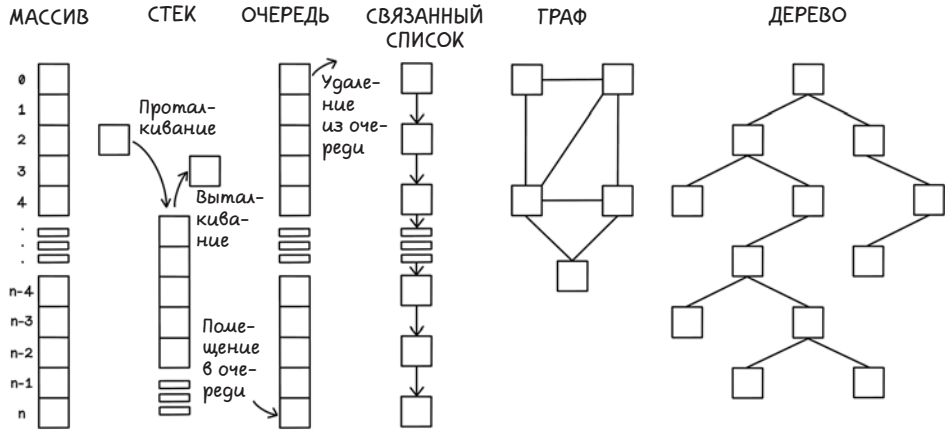


Рис. 2.8. Структуры данных, используемые с алгоритмами

Графы: представление задач поиска и решений поиска

Граф — это структура данных, содержащая несколько связанных между собой состояний. Каждое состояние называется *вершиной* (или *узлом*), а связь между двумя состояниями — *ребром*. Понятие графа происходит из *теории графов* в математике и используется для моделирования связей между объектами. Это полезные структуры данных, которые благодаря простоте визуального представления легко понятны человеку, а благодаря их логической природе очень удобны для обработки с помощью различных алгоритмов (рис. 2.9). Мы рассмотрим структуры графа на страницах этой книги.

На рис. 2.10 приведен граф поездки на море, о которой мы недавно говорили. Каждая остановка представлена в виде вершины, ребра между вершинами отражают точки, между которыми мы перемещались, а веса каждого ребра указывают пройденное расстояние.

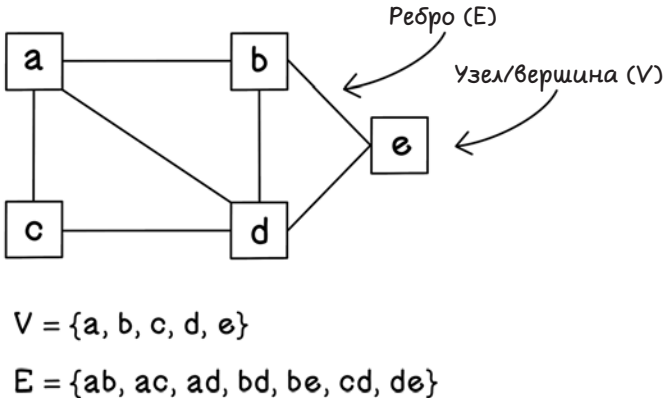


Рис. 2.9. Обозначения, используемые для записи графов

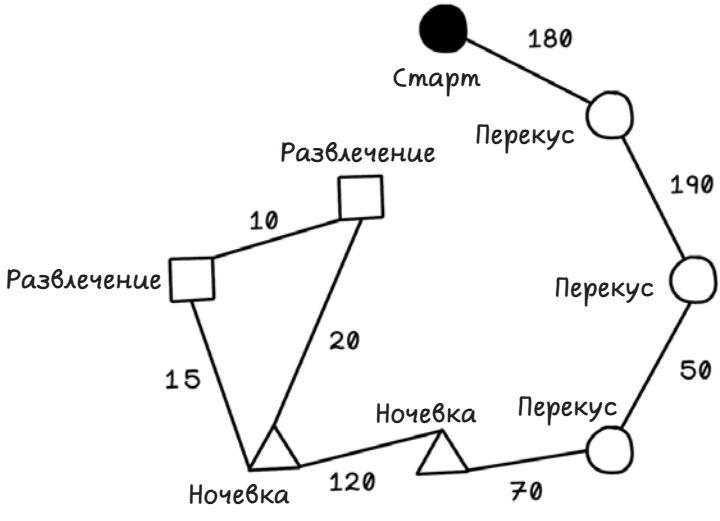


Рис. 2.10. Пример описания поездки в виде графа

Представление графа как конкретной структуры данных

Для эффективной обработки алгоритмами граф можно представить несколькими способами. По сути, его можно отразить с помощью массива массивов, указывающего связи между вершинами (рис. 2.11). Иногда полезно создать еще один массив, просто перечисляющий все вершины графа, чтобы отдельные вершины не приходилось выводить на основе связей.

К другим вариантам представлений графов относятся матрица инцидентности, матрица смежности и список смежности. Из их названий понятно, что здесь важна смежность вершин. *Смежная вершина* — это вершина, которая напрямую связана с другой вершиной.

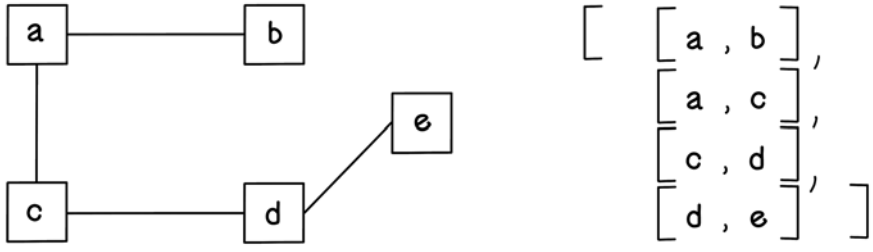
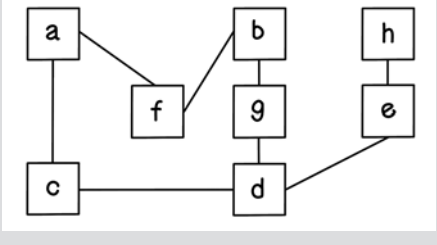


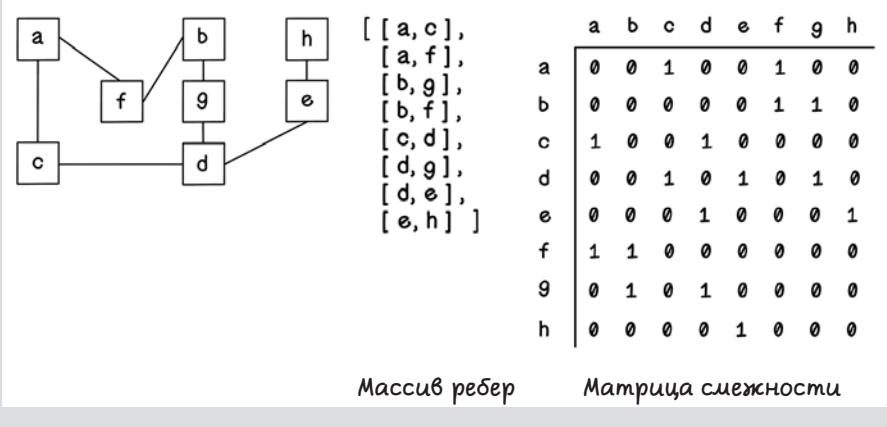
Рис. 2.11. Представление графа как массива массивов

Упражнение: представьте граф в виде матрицы

Как бы вы представили этот граф, используя массивы ребер?



Решение:



Дерево: структура, используемая для представления решений поиска

Дерево — это популярная структура данных, которая симулирует иерархию значений или объектов. *Иерархия* — это упорядоченность элементов, в которой один объект связан с несколькими другими нижестоящими объектами. Дерево — это *связный ациклический граф*, то есть граф, где каждая вершина связана ребром с другой вершиной без образования цикла.

В дереве значение или объект, представленный в конкретной точке, называется *узлом*. Как правило, у деревьев есть один корневой узел с некоторым количеством дочерних узлов, которые могут содержать поддеревья.

Давайте разберемся еще с несколькими терминами. Когда у узла есть связанные с ним узлы, корневой узел называется *родительским*. Это рассуждение можно применять рекурсивно. Дочерний узел может иметь свои собственные дочерние узлы, которые также могут содержать поддеревья или собственные дочерние узлы. Каждый дочерний узел имеет ровно один родительский узел. Узел, не имеющий дочерних узлов, называется концевым узлом (листом). У деревьев также есть *общая высота*. При этом уровень конкретных узлов называется их *глубиной*.

> **ПРИМЕЧАНИЕ** Терминология, описывающая связи внутри семейства узлов, активно используется при работе с деревьями. Так что рекомендую запомнить ее, поскольку это поможет связывать понятия в структуре данных дерева.

На рис. 2.12 глубина измеряется, начиная с 0 у корневого узла. Корневой узел имеет 4 дочерних узла, высота дерева равна 4, а глубина в этом примере равна 2. Глубина всегда определяется тем, на каком уровне дерева находится алгоритм во время обхода.

Верхний узел называется *корневым*. Узел, непосредственно связанный с одним или более узлом, называется *родительским*. Узлы, связанные с родительским, называются *дочерними* или *соседними*. Узлы, имеющие одного родителя, называются *братьями*. Связь между двумя узлами называется *ребром* (или *ветвью*).

Путь — это последовательность узлов и связывающих их ребер, которые не соединены напрямую. Узел, связанный с другим узлом по исходящему от корневого узла пути, называется *потомком*. И наоборот, узел, связанный с другим узлом через путь, ведущий к корню дерева, называется *предком*. Узел, не имеющий потомков, называется *концевым узлом (листом)*. Для описания количества потомков узла используется термин *степень*, из чего следует, что степень концевого узла равна 0.

На рис. 2.13 показан путь от начальной точки до цели в задаче с лабиринтом. Этот путь содержит девять узлов, представляющих совершаемые ходы.

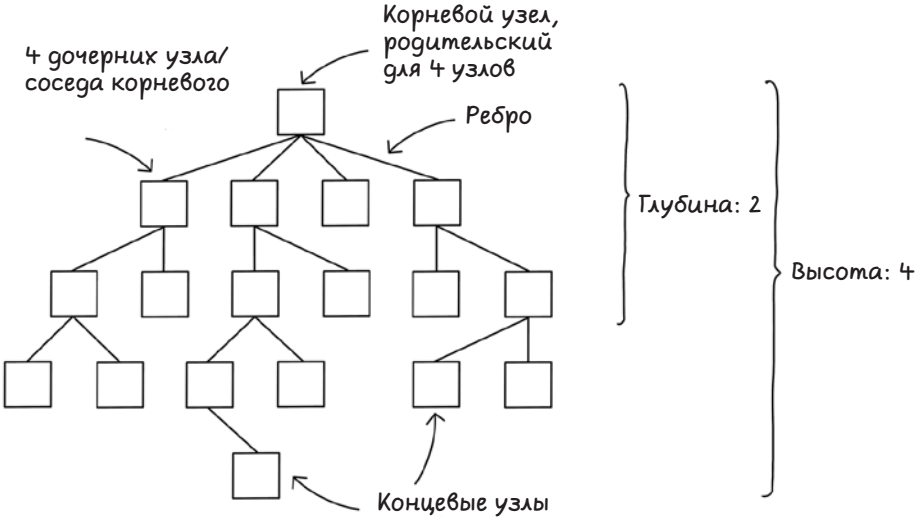


Рис. 2.12. Основные атрибуты дерева

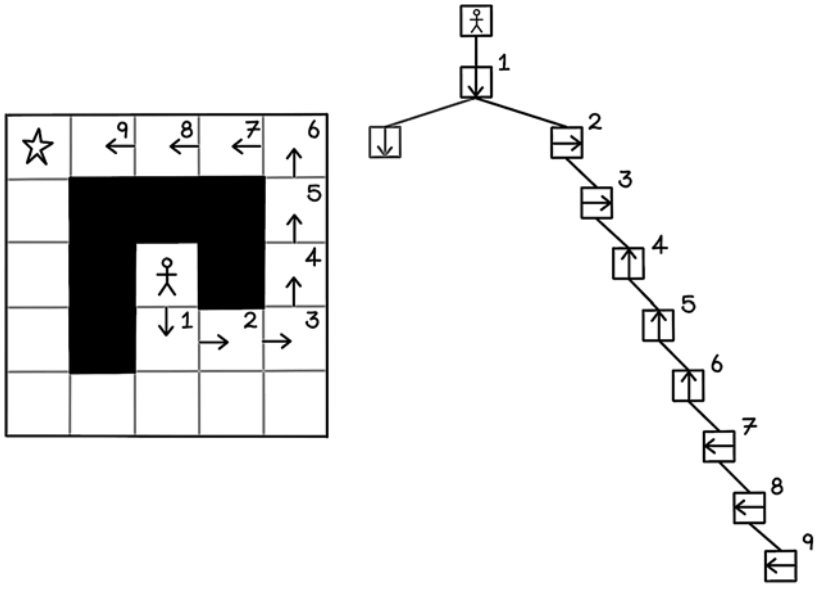


Рис. 2.13. Решение задачи с лабиринтом, представленное в виде дерева

Деревья — это фундаментальная структура данных для поисковых алгоритмов, к которым мы перейдем далее.

СОВЕТ Для более эффективных вычислений и решения конкретных задач применяется также сортировка алгоритмов. Если вы хотите узнать о ней больше, обратитесь к книге «Grokking Algorithms» Адитьи Бхаргавы (Aditya Y. Bhargava)[1].

Неинформированный поиск: слепой поиск решений

Неинформированный поиск известен также как *слепой поиск* или *поиск методом грубой силы (брутфорс)*. В неинформированных поисковых алгоритмах отсутствует информация о предметной области задачи, помимо представления самой задачи, которое обычно имеет форму дерева.

Подумайте, например, как вы изучаете что-то новое. Кто-то берет несколько разных тем и изучает основы каждой, а другие выбирают одну узкую тематику и углубляются во все ее подразделы. В этом заключается разница между поиском в ширину (breadth-first search, BFS) и поиском в глубину (depth-first search, DFS). *Поиск в глубину* подразумевает исследование конкретного пути от начальной точки вглубь до достижения цели. *Поиск в ширину*, в свою очередь, исследует все возможные варианты на определенной глубине пути, после чего переходит на следующий уровень и повторяет перебор вариантов.

Вернемся к лабиринту (рис. 2.14).

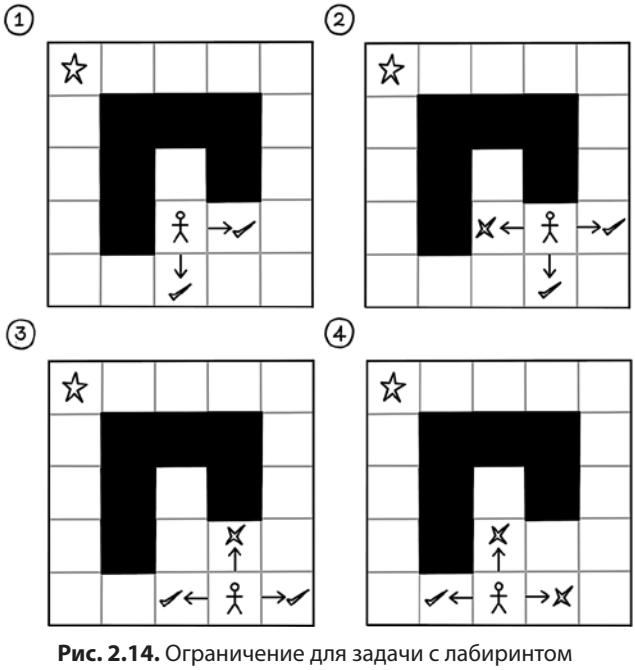


Рис. 2.14. Ограничение для задачи с лабиринтом

[1] *Бхаргава А.* «Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих». СПб., издательство «Питер».

В попытке найти оптимальный путь к цели введем следующее простое ограничение, которое позволит предотвратить застревание в бесконечном цикле и формирование циклов в дереве: *игрок не может перемещаться в клетку, которую он уже занимал*. Поскольку неинформированные алгоритмы перебирают каждый возможный вариант каждого узла, зацикливание приведет к фатальному сбою.

Заданное ограничение исключает возникновение циклов на пути к цели. Но при этом оно может вызвать проблему, если в другом лабиринте существуют другие правила, по которым подобное перемещение необходимо, чтобы найти оптимальное решение.

На рис. 2.15 представлены все возможные пути и все доступные варианты. Это дерево содержит семь путей, которые ведут к цели, и один, ведущий к неверному решению. При этом работает условие запрета на возврат к ранее занятой клетке. Для такого небольшого лабиринта можно изобразить все варианты сразу. Тем не менее задача поискового алгоритма — выводить деревья поиска поочередно, поскольку генерировать все дерево возможностей сразу неэффективно ввиду высоких вычислительных затрат.

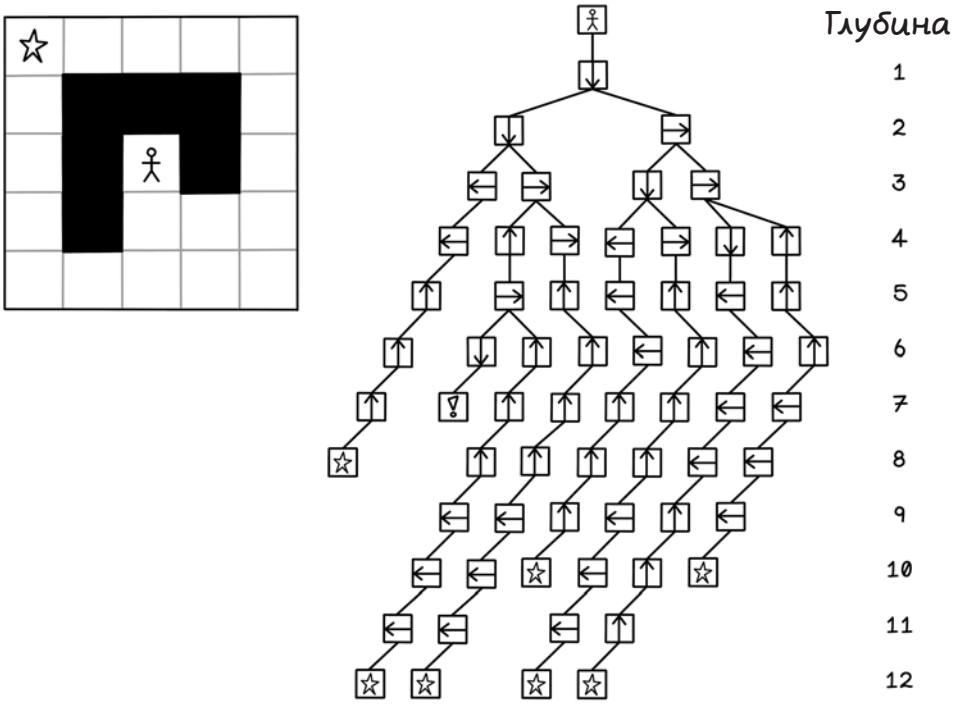


Рис. 2.15. Все возможные варианты перемещения, представленные в виде дерева

Также важно отметить, что термин *посещение* используется для обозначения разных понятий. Игрок посещает клетку в лабиринте. Алгоритм также посещает узлы в дереве. Порядок выбранных игроком шагов будет влиять на порядок посещения узлов дерева. В примере с лабиринтом приоритетным порядком перемещения является север, юг, восток, а затем запад.

Поиск в ширину: смотреть сначала вширь, потом вглубь

Теперь, когда мы понимаем идеи, лежащие в основе деревьев и примера с лабиринтом, давайте рассмотрим, как алгоритмы поиска могут строить деревья, которые находят пути к цели. Поиск в ширину (BFS) — это алгоритм, используемый для обхода или построения дерева. Этот алгоритм начинается с определенного узла, обычно корневого, и исследует все узлы на данной глубине, прежде чем перейти к следующей глубине. Представьте себе BFS как камень, брошенный в воду. Круги расходятся равномерно во всех направлениях, касаясь всего, что находится в метре, затем всего, что в двух метрах, и т. д. Алгоритм имитирует это расширяющееся кольцо, посещая всех соседей на текущей глубине, прежде чем двигаться дальше. Это гарантирует, что во взвешенных графах (графах, где все ребра равноценны) будет найден кратчайший путь.

Алгоритм BFS лучше всего реализуется с помощью очереди «первым вошел — первым вышел» (FIFO, first-in — first-out), в которой обрабатываются узлы на текущей глубине дерева, а их дочерние узлы ставятся в очередь для последующей обработки. Именно такой порядок обработки нам и нужен при реализации этого алгоритма. На рис. 2.16 приводится блок-схема, отражающая последовательность шагов алгоритма поиска в ширину.

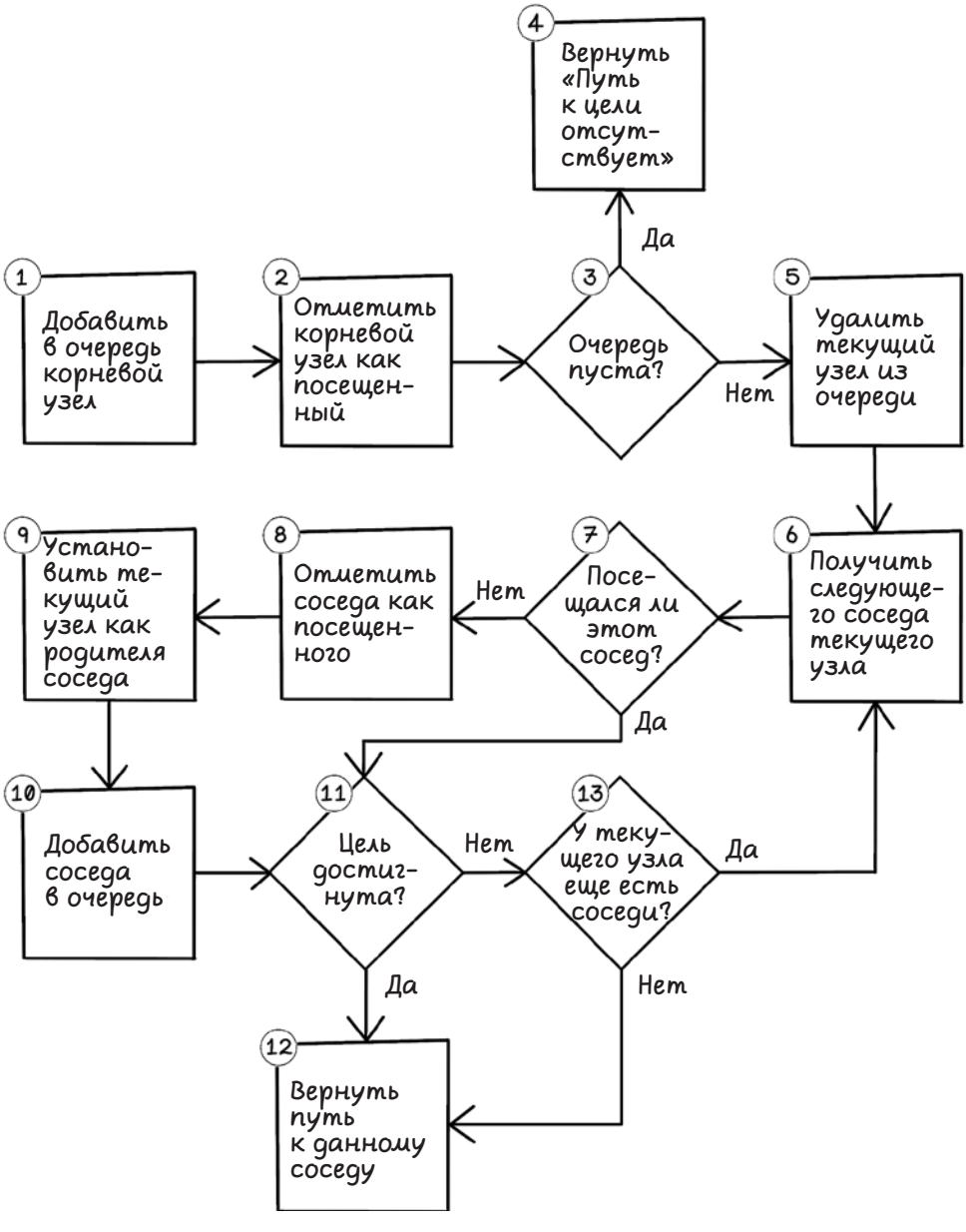


Рис. 2.16. Поток алгоритма поиска в ширину

Шаги алгоритма поиска в ширину

Давайте пройдемся по каждому шагу алгоритма, чтобы разобраться, какие операции выполняются:

1. *Добавить в очередь корневой узел.* Алгоритм поиска в ширину лучше всего реализуется с помощью очереди. Объекты обрабатываются в той последовательности, в какой добавляются в очередь. Этот процесс также называется FIFO. Первый шаг — это добавление в очередь корневого узла, который будет представлять начальную позицию игрока на карте.
2. *Отметить корневой узел как посещенный.* После добавления корневого узла в очередь для обработки он отмечается как посещенный, что исключает его повторное посещение.
3. *Очередь пуста?* Если очередь оказывается пуста (спустя множество итераций все узлы были обработаны) и на 12-м шаге алгоритма не был возвращен конечный путь, значит, путь к цели отсутствует. Если же в очереди остались узлы, алгоритм может продолжить ее поиск.
4. *Вернуть «Путь к цели отсутствует».* Это сообщение — возможный выход из алгоритма в случае отсутствия пути к цели.
5. *Удалить текущий узел из очереди.* Извлекая следующий объект из очереди и рассматривая его как текущий узел, изучить его возможности. В начале выполнения алгоритма текущим узлом является корневой узел.
6. *Получить следующего соседа текущего узла.* Совершение следующего возможного хода в лабиринте, для чего определяются доступные направления движения: на север, юг, восток или запад.
7. *Посещался ли этот сосед?* Если текущий сосед не посещался, значит, он еще не изучен и может быть обработан.
8. *Отметить соседа как посещенного.* Этот шаг указывает, что соседний узел был посещен.
9. *Установить текущий узел как родителя соседа.* Установка узла-источника как родителя текущего соседа. Этот шаг важен для отслеживания пути от текущего соседа к корневому узлу. С позиции карты источник — это клетка, из которой игрок перемещается, а текущий сосед — это клетка, куда он перемещается.
10. *Добавить соседа в очередь.* Соседний узел ставится в очередь, что позволяет далее исследовать его потомков. Такой механизм очереди позволяет обрабатывать узлы на каждом уровне глубины в нужном порядке.
11. *Цель достигнута?* Этот шаг определяет, содержит ли текущий сосед искомую алгоритмом цель.
12. *Вернуть путь к данному соседу.* Поочередно ссылаясь на родителя текущего соседа, затем его родителя и т. д., описывается путь от цели к корню. Корневой узел при этом окажется узлом без родителя.
13. *У текущего узла еще есть соседи?* Если от текущего узла возможны еще ходы — перейти к шагу 6 и сделать ход.

Теперь рассмотрим, как все это должно выглядеть в простом дереве. Обратите внимание, что по мере исследования дерева и добавления узлов в очередь FIFO эти узлы обрабатываются в нужном порядке, устанавливаемом очередью (рис. 2.17 и 2.18).

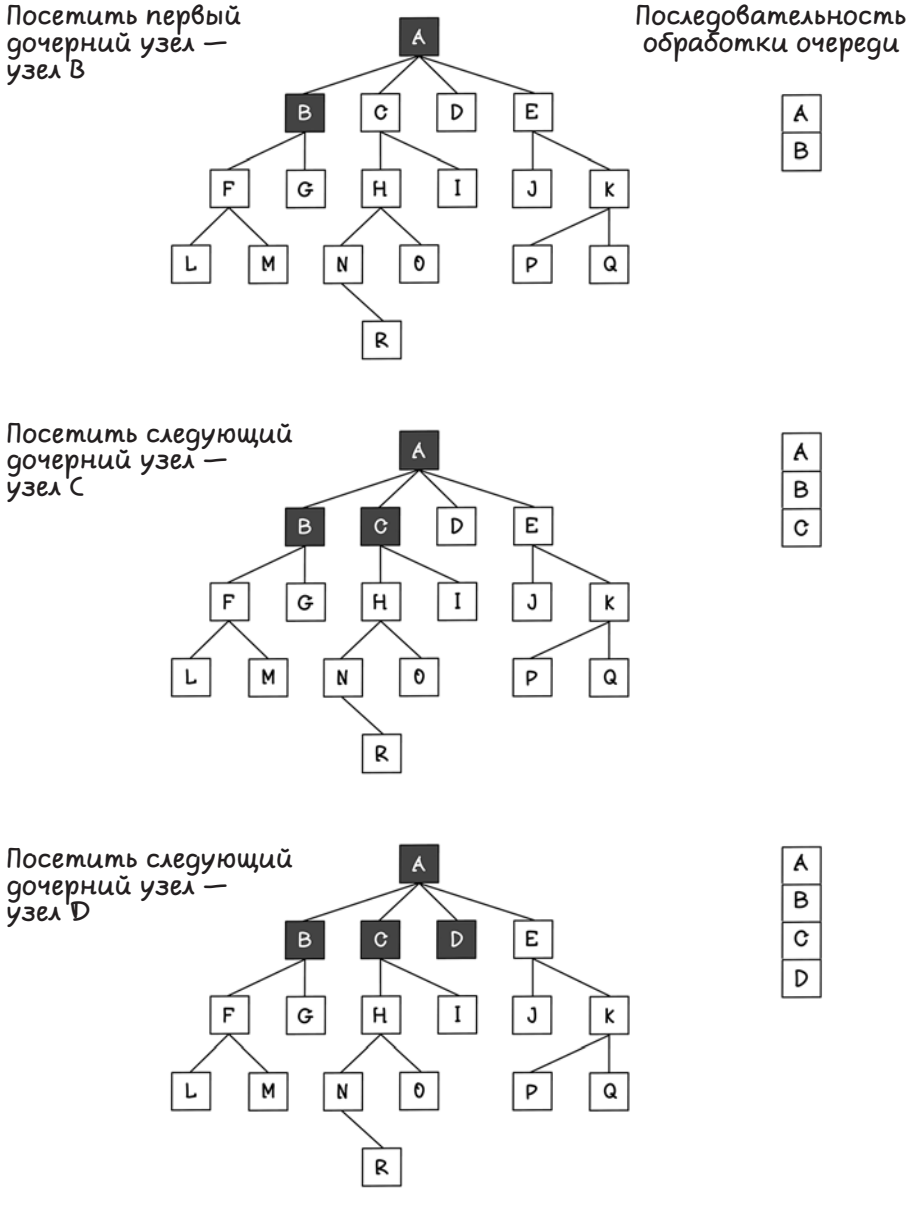


Рис. 2.17. Последовательность обработки дерева поиском в ширину (часть 1)

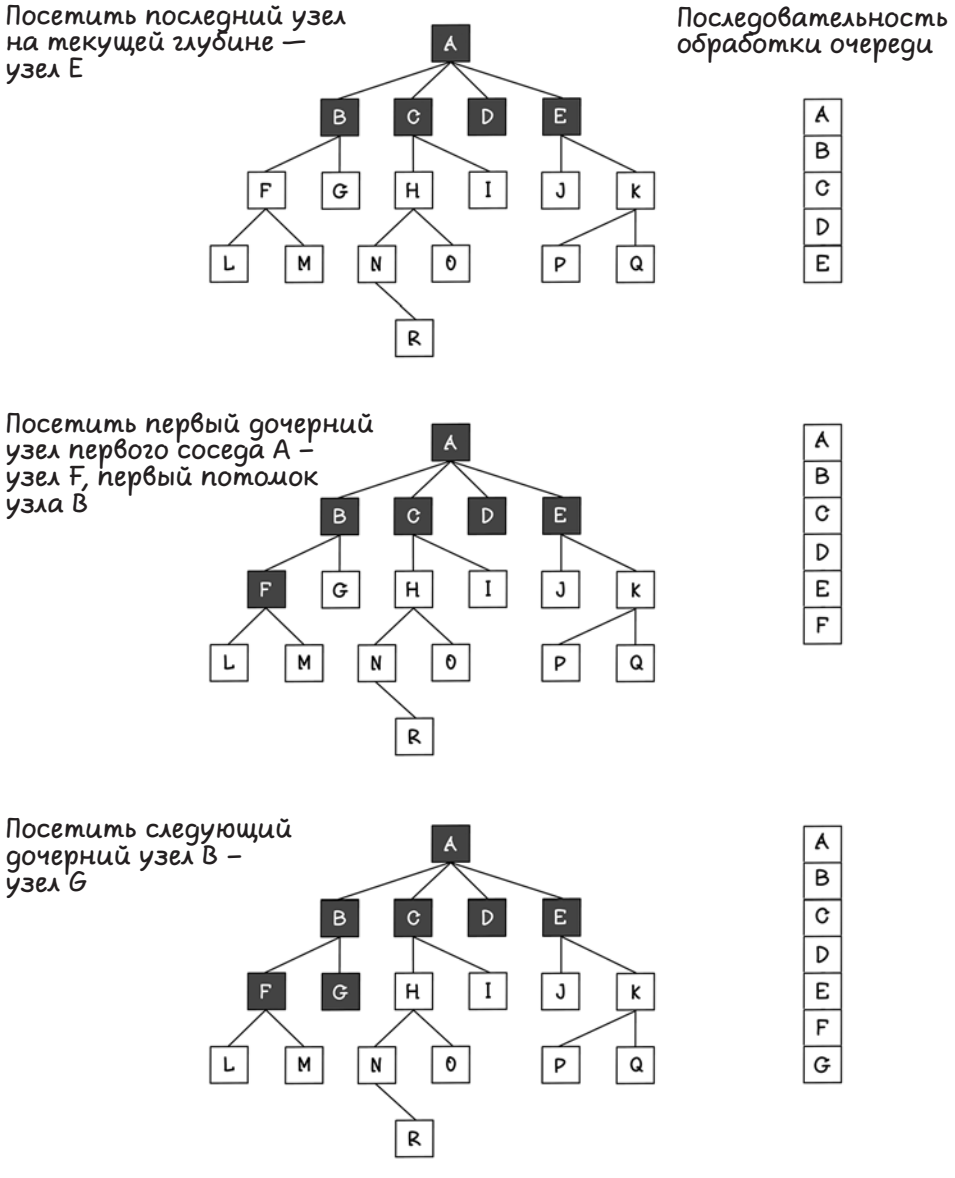
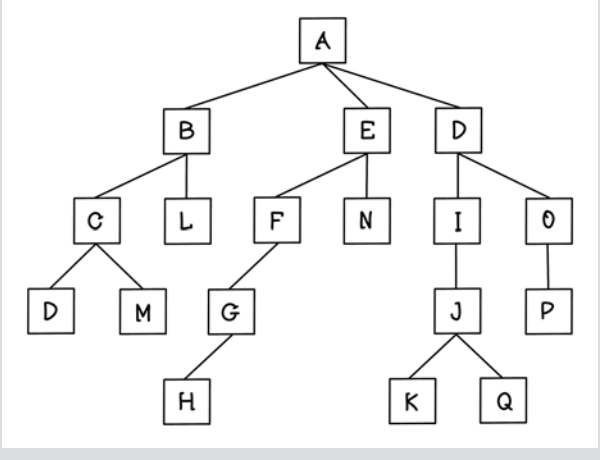


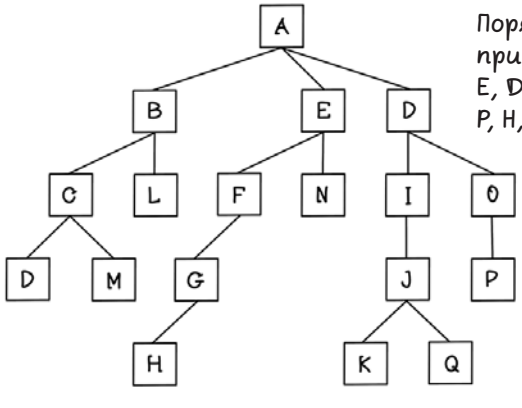
Рис. 2.18. Последовательность обработки дерева поиском в ширину (часть 2)

Упражнение: найдите путь к решению

Каким будет порядок исследования узлов следующего дерева при поиске в ширину?



Решение:



Порядок исследования узлов при поиске в ширину: A, B, E, D, C, L, F, N, I, O, D, M, G, J, P, H, K, Q

В примере с лабиринтом алгоритму нужно понять текущую позицию игрока, оценить все возможные направления и повторить эту логику для каждого направления, пока не будет достигнута цель. Таким образом алгоритм генерирует дерево с единственным путем к цели.

Важно понять, что с помощью посещений происходит генерация узлов в дереве. Мы просто находим через этот механизм связанные узлы.

Каждый путь к цели состоит из серии ходов. Количество ходов является расстоянием до цели по этому пути, которое мы будем называть *стоимостью*. Количество ходов также равняется количеству посещаемых на пути узлов, начиная от корневого и заканчивая концевым, в котором находится цель. Алгоритм перемещается вниз по дереву, переходя все глубже и глубже, пока не находит цель. Затем в качестве решения он возвращает первый найденный путь, который к ней привел. При этом может существовать и другой — оптимальный — маршрут, но поскольку поиск неинформированный, нет гарантии, что будет найден именно он.

> **ПРИМЕЧАНИЕ** В примере с лабиринтом все алгоритмы поиска завершаются при обнаружении решения. Можно позволить им находить несколько решений, внеся небольшую доработку, но лучше всего вести поиск именно одного пути, поскольку перебор всех возможных вариантов будет дорогостоящей процедурой.

На рис. 2.19 показана генерация дерева в процессе перемещения по лабиринту. Алгоритм поиска в ширину исследовал дерево до глубины 5. На самой карте в результате поиска на данный момент обнаружилось два пути к цели.

Поскольку дерево строится с помощью поиска в ширину, алгоритм полностью обрабатывает каждый уровень глубины, прежде чем перейти к следующему. На рис. 2.20 показано все дерево возможностей (в учебных целях). На глубине 7 алгоритм нашел путь к цели, поэтому наше решение — юг, юг, запад, запад, север, север, север.