Поиск в графах

Как найти узел в графе? Если структура графа не предоставляет никакой помощи в навигации, вам придется посетить каждую вершину, пока не обнаружится нужная. Есть два способа сделать это: выполнить обход графа в глубину и в ширину (рис. 5.2).

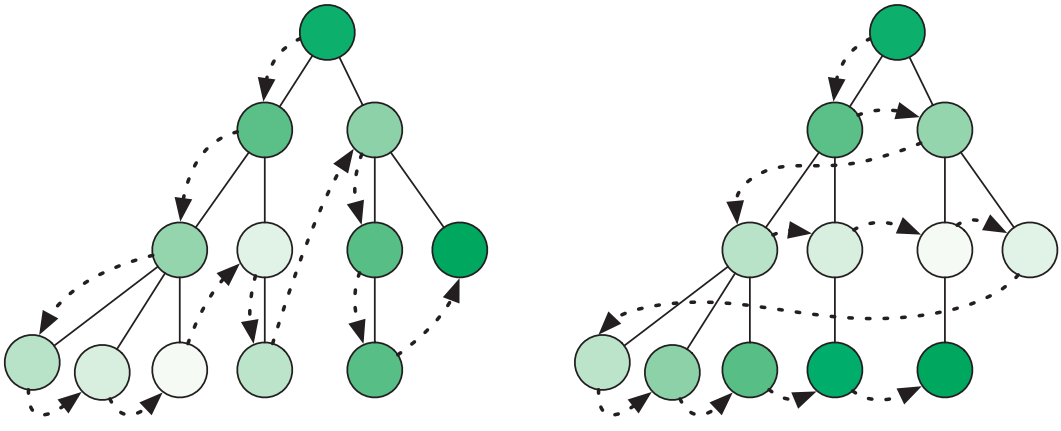

Рис. 5.2. Обход графа в глубину против обхода в ширину

Выполняя поиск в графе в глубину (DFS, depth first search), мы продвигаемся вдоль ребер, уходя все глубже и глубже в граф. Достигнув вершины без ребер, ведущих к каким-либо новым вершинам, мы возвращаемся к предыдущей и продолжаем процесс. Мы используем стек, чтобы запомнить путь обхода графа, помещая туда вершину на время ее исследования и удаляя ее, когда нужно вернуться. Стратегия поиска с возвратом (см. соответствующий раздел главы 3) выполняет обход решений точно так же.

```
function DFS(start_node, key)
    next_nodes ← Stack.new()
    seen_nodes ← Set.new()

    next_nodes.push(start_node)
    seen_nodes.add(start_node)
```

```
    while not next_nodes.empty
        node ← next_nodes.pop()
        if node.key = key
            return node
        for n in node.connected_nodes
            if not n in seen_nodes
                next_nodes.push(n)
                seen_nodes.add(n)
    return NULL
```

Если обход графа вглубь не кажется приемлемым решением, можно попробовать обход в ширину (BFS, breadth first search). В этом случае обход графа выполняется по уровням: сначала соседей начальной вершины, затем соседей его соседей и т. д. Вершины для посещения запоминаются в очереди. Исследуя вершину, мы ставим в очередь ее дочерние вершины, затем определяем следующую исследуемую вершину, извлекая ее из очереди.

```
function BFS(start_node, key)
    next_nodes ← Queue.new()
    seen_nodes ← Set.new()

    next_nodes.enqueue(start_node)
    seen_nodes.add(start_node)

    while not next_nodes.empty
        node ← next_nodes.dequeue()
        if node.key = key
            return node
        for n in node.connected_nodes
            if not n in seen_nodes
                next_nodes.enqueue(n)
                seen_nodes.add(n)
    return NULL
```

Обратите внимание, что алгоритмы `DFS` и `BFS` отличаются только способом хранения следующих исследуемых вершин: в одном случае это очередь, в другом — стек.

Итак, какой подход нам следует использовать? Алгоритм `DFS` более прост в реализации и использует меньше памяти: достаточно хра-

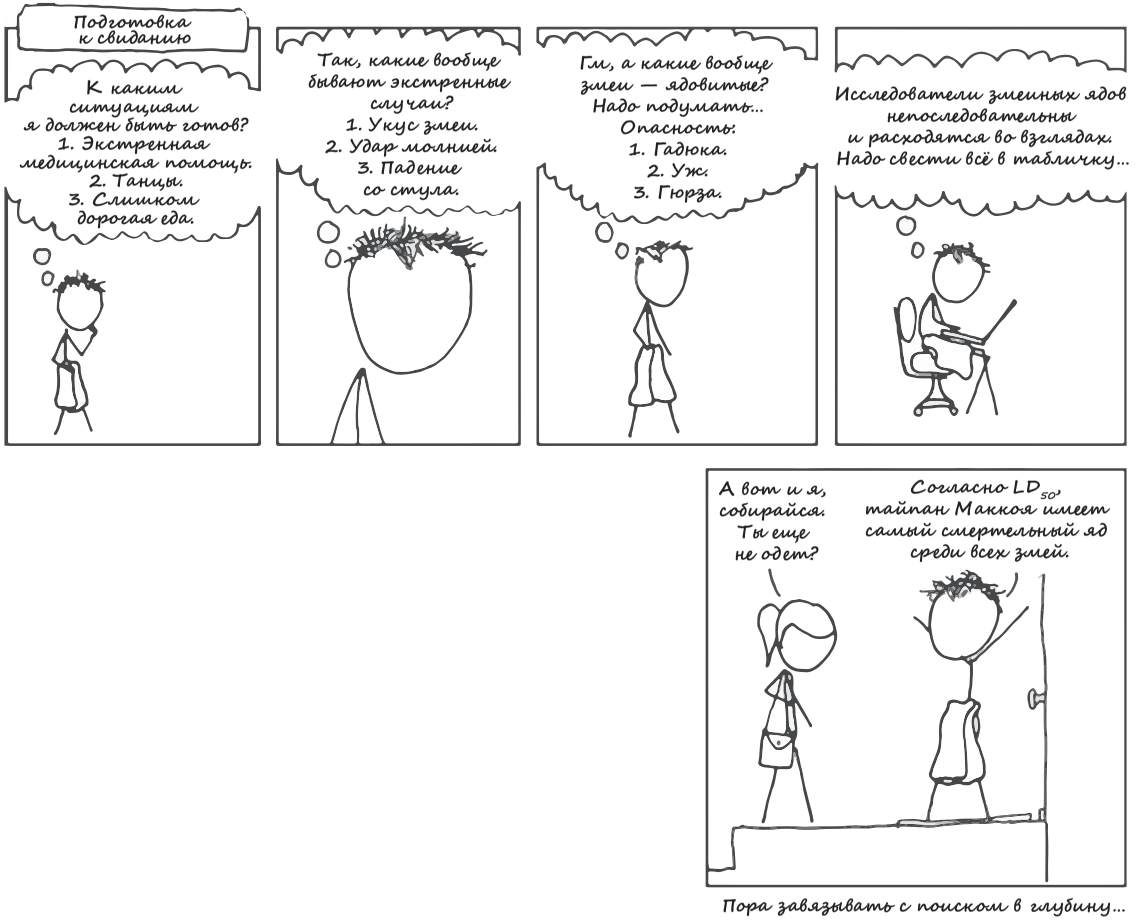

Рис. 5.3. Поиск в графе в глубину[1]

[1] Любезно предоставлено http://xkcd.com.

нить родительские вершины, ведущие к текущей исследуемой вершине. В BFS придется хранить всю границу процесса поиска. Если граф состоит из миллиона вершин, это может оказаться непрактичным.

Когда есть основания предполагать, что искомая вершина не находится многими уровнями ниже начальной, обычно имеет смысл заплатить более высокую стоимость BFS, потому что так вы, скорее всего, закончите поиск быстрее. Если нужно исследовать абсолютно все вершины графа, лучше придерживаться алгоритма DFS из-за его простой реализации и меньшего объема потребляемой памяти.

Рис. 5.3 показывает, что выбор неправильного метода обхода может иметь страшные последствия.

Раскраска графов

Задачи раскраски графов возникают, когда есть фиксированное число «красок» (либо любой другой набор меток) и вы должны назначить «цвет» каждой вершине в графе. Вершины, которые соединены ребром, не могут иметь одинаковый «цвет». В качестве примера давайте рассмотрим следующую задачу.

> **Помехи**  Дана карта вышек сотовой связи и районов обслуживания. Вышки в смежных районах должны работать на разных частотах для предотвращения помех. Имеется четыре частоты на выбор. Какую частоту вы назначите каждой вышке?

Первый шаг состоит в моделировании задачи при помощи графа. Вышки являются вершинами в графе. Если две из них расположены настолько близко, что вызывают помехи, соединяем их ребром. Каждая частота имеет свой цвет.

Как назначить частоты приемлемым способом? Можно ли найти решение, которое использует всего три цвета? Или два? Определение

минимально возможного количества цветов на самом деле является NP-полной задачей — для этого подходят только экспоненциальные алгоритмы.

Мы не покажем алгоритм для решения данной задачи. Используйте то, чему вы научились к настоящему моменту, и попробуйте решить задачу самостоятельно. Это можно сделать на сайте UVA[1] с онлайн-экспертом, который протестирует предложенное вами решение, выполнит ваш программный код и сообщит, работоспособен ли он. Если с кодом окажется все в порядке, эксперт также оценит время выполнения вашего кода в сравнении с тем, что написали другие люди. Дерзайте! Продумайте алгоритмы и стратегии решения данной задачи и испытайте их. Чтение книги может лишь подвести вас к решению. Взаимодействие с онлайновым экспертом даст вам практический опыт, необходимый для того, чтобы стать отличным программистом.

Поиск путей в графе

Поиск кратчайшего пути между узлами является самой известной графовой задачей. Системы навигации GPS проводят поиск в графе улиц и перекрестков для вычисления маршрута. Некоторые из них даже используют данные дорожного движения с целью увеличения веса ребер, представляющих улицы, где образовался затор.

Для поиска кратчайшего пути вполне можно использовать стратегии BFS и DFS, но это плохая идея. Одним из хорошо известных и очень эффективных способов поиска кратчайшего пути является *алгоритм Дейкстры*. В отличие от BFS, для запоминания просматриваемых вершин алгоритм Дейкстры использует очередь с приоритетом. Когда исследуются новые вершины, их связи добавляются в эту очередь. Приоритетом вершины является вес ребер, которые приводят ее в стартовую вершину. Благодаря этому следующая ис-

[1] Задача раскраски графа на сайте онлайновых экспертов UVA: https://code.energy/uva-graph-coloring.