Вот и все! Графы состоят из узлов и ребер. Узел может быть напрямую соединен с несколькими другими узлами. Эти узлы называются *соседями*. На этом графе Рама является соседом Алекса. С другой стороны, Адит соседом Алекса не является, потому что они не соединены напрямую. При этом Адит является соседом Рамы и Тома.

Графы используются для моделирования связей между разными объектами. А теперь посмотрим, как работает поиск в ширину.

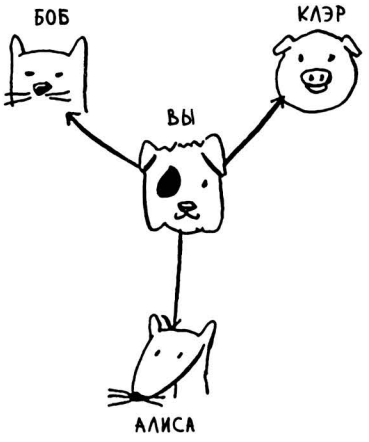
Поиск в ширину

В главе 1 уже рассматривался пример алгоритма поиска: бинарный поиск. Поиск в ширину также относится к категории алгоритмов поиска, но этот алгоритм работает с графами. Он помогает ответить на вопросы двух типов:

- тип 1: существует ли путь от узла A к узлу B?
- тип 2: как выглядит кратчайший путь от узла A к узлу B?

Вы уже видели пример поиска в ширину, когда мы просчитывали кратчайший путь из Твин-Пикс к мосту Золотые Ворота. Это был вопрос типа 2: как выглядит кратчайший путь? Теперь разберем работу алгоритма более подробно с вопросом типа 1: существует ли путь?

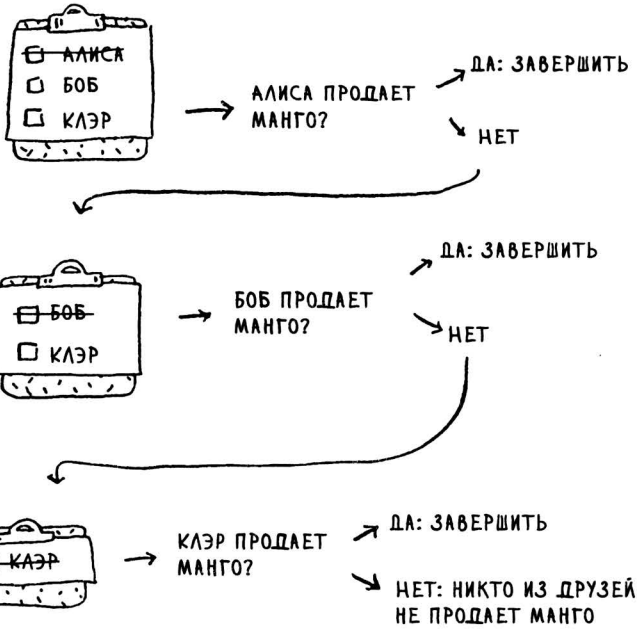
Представьте, что вы выращиваете манго. Вы ищете продавца, который будет продавать ваши замечательные манго. А может, продавец найдется среди ваших контактов на Facebook? Для начала стоит поискать среди друзей.



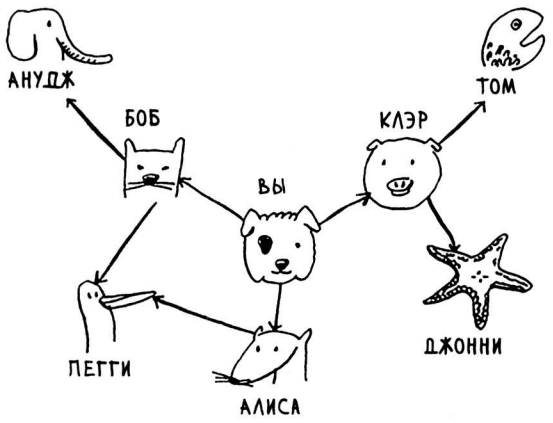
Поиск происходит вполне тривиально.

Сначала нужно построить список друзей для поиска.

Теперь нужно обратиться к каждому человеку в списке и проверить, продает ли этот человек манго.



Предположим, ни один из ваших друзей не продает манго. Теперь поиск продолжается среди друзей ваших друзей.



Каждый раз, когда вы проверяете кого-то из списка, вы добавляете в список всех его друзей.

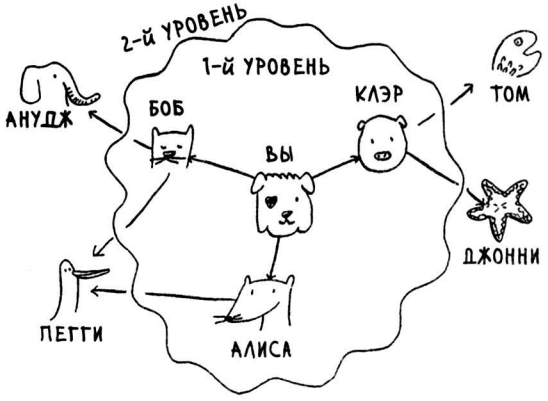
В таком случае поиск ведется не только среди друзей, но и среди друзей друзей тоже. Напомним: нужно найти в сети хотя бы одного продавца манго. Если Алиса не продает манго, то в список добавляются ее друзья. Это означает, что со временем вы проверите всех ее друзей, а потом их друзей и т. д. С этим алгоритмом поиск рано или поздно пройдет по всей сети, пока вы все-таки не наткнетесь на продавца манго. Такой алгоритм и называется поиском в ширину.

Поиск кратчайшего пути

На всякий случай напомню два вопроса, на которые может ответить алгоритм поиска в ширину:

- тип 1: существует ли путь от узла А к узлу В? (Есть ли продавец манго в вашей сети?)
- тип 2: как выглядит кратчайший путь от узла А к узлу В? (Кто из продавцов манго находится ближе всего к вам?)

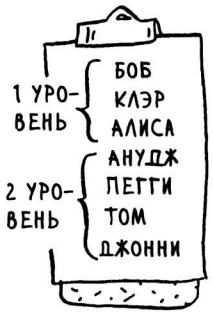
Вы уже знаете, как ответить на вопрос 1; теперь попробуем ответить на вопрос 2. Удастся ли вам найти ближайшего продавца манго? Будем считать, что ваши друзья — это связи первого уровня, а друзья друзей — связи второго уровня.



Связи первого уровня предпочтительнее связей второго уровня, связи второго уровня предпочтительнее связей третьего уровня и т. д. Отсюда следует, что поиск по контактам второго уровня не должен производиться, пока вы не будете полностью уверены в том, что среди связей первого уровня нет ни одного продавца манго. Но ведь поиск в ширину именно это и делает! Поиск в ширину распространяется от начальной точки. А это означает, что связи первого уровня будут проверены до связей второго уровня. Контрольный вопрос: кто будет проверен первым, Клэр или Анудж? Ответ:

Клэр является связью первого уровня, а Анудж — связью второго уровня. Следовательно, Клэр будет проверена первой.

Также можно объяснить это иначе: связи первого уровня добавляются в список поиска раньше связей второго уровня.



Вы двигаетесь вниз по списку и проверяете каждого человека (является ли он продавцом манго). Связи первого уровня будут проверены до связей второго уровня, так что вы найдете продавца манго, ближайшего к вам. Поиск в ширину находит не только путь из A в B, но и кратчайший путь.

Обратите внимание: это условие выполняется только в том случае, если поиск осуществляется в порядке добавления людей. Другими словами, если Клэр была добавлена в список до Ануджа, то проверка Клэр должна быть выполнена до проверки Ануджа. А что произойдет, если вы проверите Ануджа раньше, чем Клэр, и оба они окажутся продавцами манго? Анудж является связью второго уровня, а Клэр — связью первого уровня. В результате будет найден продавец манго, не ближайший к вам в сети. Следовательно, проверять связи нужно в порядке их добавления. Для операций такого рода существует специальная структура данных, которая называется *очередью*.

Очереди

Очередь работает точно так же, как и в реальной жизни. Предположим, вы с другом стоите в очереди на автобусной остановке. Если вы стоите ближе к началу очереди, то вы первым сядете в автобус. Структура данных очереди работает аналогично. Очереди чем-то похожи на стеки: вы не можете обращаться к произвольным элементам очереди. Вместо этого поддерживаются всего две операции: *постановка в очередь* и *извлечение из очереди*.

Если вы поставите в очередь два элемента, то элемент, добавленный первым, будет извлечен из очереди раньше второго. А ведь это свойство можно использовать для реализации списка поиска! Люди, добавленные в список первыми, будут извлечены из очереди и проверены первыми.

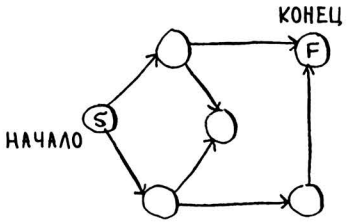
Очередь относится к категории структур данных FIFO: First In, First Out («первым вошел, первым вышел»). А стек принадлежит к числу структур данных LIFO: Last In, First Out («последним пришел, первым вышел»).

Теперь, когда вы знаете, как работает очередь, можно переходить к реализации поиска в ширину!

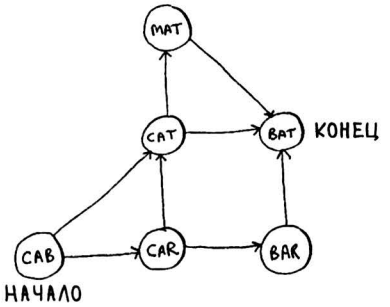
Упражнения

Примените алгоритм поиска в ширину к каждому из этих графов, чтобы найти решение.

6.1 Найдите длину кратчайшего пути от начального до конечного узла.

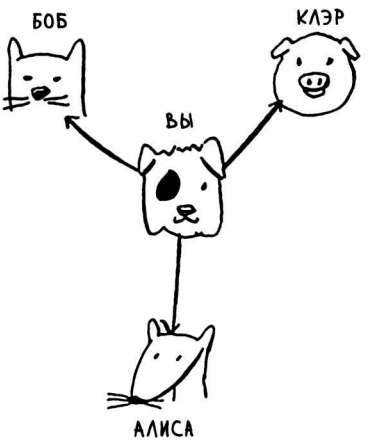


6.2 Найдите длину кратчайшего пути от «cab» к «bat».



Реализация графа

Для начала необходимо реализовать граф на программном уровне. Граф состоит из нескольких узлов. И каждый узел соединяется с соседними узлами. Как выразить отношение типа «вы –> боб»? К счастью, вам уже известна структура данных, способная выражать отношения: *хеш-таблица*!



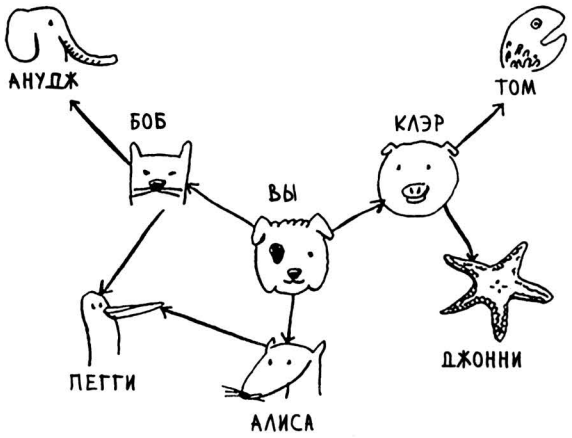
Вспомните: хеш-таблица связывает ключ со значением. В данном случае узел должен быть связан со всеми его соседями.

А вот как это записывается на Python:

```
graph = {}
graph["you"] = ["alice", "bob", "claire"]
```

Обратите внимание: элемент «вы» (you) отображается на массив. Следовательно, результатом выражения `graph["you"]` является массив всех ваших соседей.

Граф — всего лишь набор узлов и ребер, поэтому для представления графа на Python ничего больше не потребуется. А как насчет большего графа, например такого?



Код на языке Python выглядит так:

```
graph = {}
graph["you"] = ["alice", "bob", "claire"]
graph["bob"] = ["anuj", "peggy"]
graph["alice"] = ["peggy"]
graph["claire"] = ["thom", "jonny"]
graph["anuj"] = []
graph["peggy"] = []
graph["thom"] = []
graph["jonny"] = []
```

Контрольный вопрос: важен ли порядок добавления пар «ключ—значение»?

Важно ли, какую запись вы будете использовать, — такую:

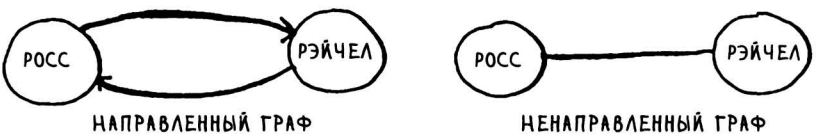
```
graph["claire"] = ["thom", "jonny"]
graph["anuj"] = []
```

или такую:

```
graph["anuj"] = []
graph["claire"] = ["thom", "jonny"]
```

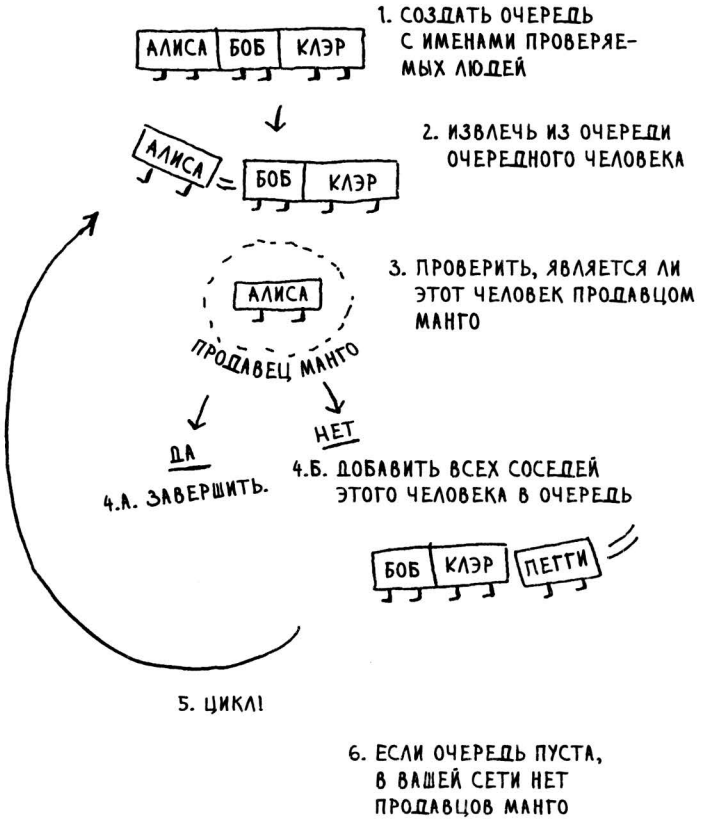
Вспомните предыдущую главу. Ответ: нет, не важно. В хеш-таблицах элементы не упорядочены, поэтому добавлять пары «ключ—значение» можно в любом порядке.

У Ануджа, Пегги, Тома и Джонни соседей нет. Линии со стрелками указывают на них, но не существует стрелок от них к другим узлам. Такой граф называется *направленным* — отношения действуют только в одну сторону. Итак, Анудж является соседом Боба, но Боб не является соседом Ануджа. В ненаправленном графе стрелок нет, и каждый из узлов является соседом по отношению друг к другу. Например, оба следующих графа эквивалентны.



Реализация алгоритма

Напомню, как работает реализация.



Все начинается с создания очереди. В Python для создания *двусторонней очереди* (дека) используется функция deque:

```
from collections import deque
search_queue = deque()          <···· Создание новой очереди
search_queue += graph["you"]    <···· Все соседи добавляются в очередь поиска
```

Напомню, что выражение `graph["you"]` вернет список всех ваших соседей, например `["alice", "bob", "claire"]`. Все они добавляются в очередь поиска.

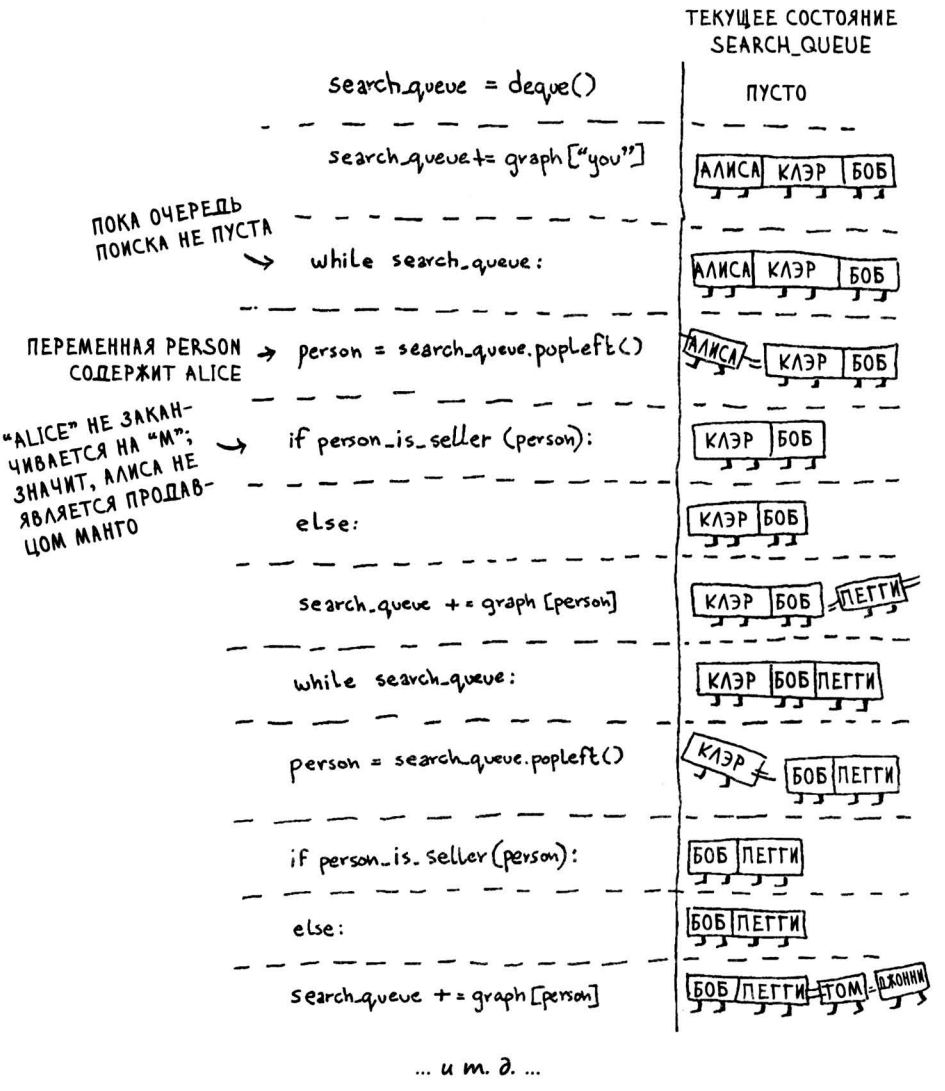
А теперь рассмотрим остальное:

```
while search_queue:  <···· Пока очередь не пуста...
    person = search_queue.popleft()  <···· из очереди извлекается первый человек
    if person_is_seller(person): <···· Проверяем, является ли этот человек
                                       продавцом манго
        print person + " is a mango seller!"  <···· Да, это продавец манго
        return True
    else:
        search_queue += graph[person]  <···· Нет, не является. Все друзья этого человека добавляются в очередь поиска
return False  <···· Если выполнение дошло
                    до этой строки, значит,
                    в очереди нет продавца манго
```

И последнее: нужно определить функцию `person_is_seller`, которая сообщает, является ли человек продавцом манго. Например, функция может выглядеть так:

```
def person_is_seller(name):
    return name[-1] == 'm'
```

Эта функция проверяет, заканчивается ли имя на букву «m», и если заканчивается, этот человек считается продавцом манго. Проверка довольно глупая, но для нашего примера сойдет. А теперь посмотрим, как работает поиск в ширину.



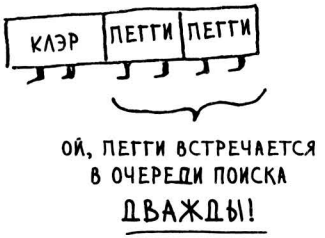
И так далее. Алгоритм продолжает работать до тех пор, пока:

- не будет найден продавец манго,

или

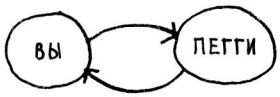
- очередь не опустеет (в этом случае продавца манго нет).

У Алисы и Боба есть один общий друг: Пегги. Следовательно, Пегги будет добавлена в очередь дважды: при добавлении друзей Алисы и при добавлении друзей Боба. В результате Пегги появится в очереди поиска в двух экземплярах.



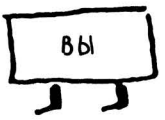
Но проверить, является ли Пегги продавцом манго, достаточно всего один раз. Проверяя ее дважды, вы выполняете лишнюю, ненужную работу. Следовательно, после проверки человека нужно пометить как проверенного, чтобы не проверять его снова.

Если этого не сделать, может возникнуть бесконечный цикл. Предположим, граф выглядит так:



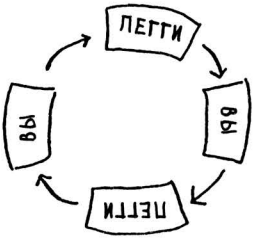
В начале очередь поиска содержит всех ваших соседей.

Теперь вы проверяете Пегги. Она не является продавцом манго, поэтому все ее соседи добавляются в очередь поиска.



Вы проверяете себя. Вы не являетесь продавцом манго, поэтому все ваши соседи добавляются в очередь поиска.

И так далее. Возникает бесконечный цикл, потому что очередь поиска будет поочередно переходить от вас к Пегги.



Прежде чем проверять человека, следует убедиться в том, что он не был проверен ранее. Для этого мы будем вести список уже проверенных людей.

А вот окончательная версия кода поиска в ширину, в которой учтено это обстоятельство:

```
def search(name):
    search_queue = deque()
    search_queue += graph[name]
    searched = []    <-- Этот массив используется для отслеживания уже проверенных людей
    while search_queue:
        person = search_queue.popleft()
        if not person in searched:    <-- Человек проверяется только в том случае, если он не проверялся ранее
            if person_is_seller(person):
                print person + " is a mango seller!"
                return True
            else:
                search_queue += graph[person]
                searched.append(person)    <-- Человек помечается как уже проверенный
    return False

search("you")
```

Попробуйте выполнить этот код самостоятельно. Замените функцию `person_is_seller` чем-то более содержательным и посмотрите, выведет ли она то, что вы ожидали.

Время выполнения

Если поиск продавца манго был выполнен по всей сети, значит, вы прошли по каждому ребру (напомню: ребром называется соединительная линия или линия со стрелкой, ведущая от одного человека к другому). Таким образом, время выполнения составляет как минимум $O(количество\ ребер)$.

Также в программе должна храниться очередь поиска. Добавление одного человека в очередь выполняется за постоянное время: $O(1)$. Выполнение операции для каждого человека потребует суммарного времени $O(количество\ людей)$. Поиск в ширину выполняется за время $O(количество\ людей + количество\ ребер)$, что обычно записывается в форме $O(V+E)$ (V — количество вершин, E — количество ребер).