4 Графовые задачи

Граф — это абстрактная математическая конструкция, которая применяется для моделирования реальной задачи путем ее разделения на множество связанных узлов. Каждый такой узел мы будем называть *вершиной*, а каждое соединение — *ребром*. Так, карту метро можно рассматривать как граф, представляющий транспортную сеть. Каждая из ее точек — это станция, а каждая из линий — маршрут между двумя станциями. В терминологии графов мы будем называть станции вершинами, а маршруты — ребрами.

В чем здесь польза? Графы не только помогают абстрактно представить задачу, но и позволяют задействовать несколько широко распространенных и эффективных методов поиска и оптимизации. Так, в примере с метро предположим, что мы хотим узнать кратчайший маршрут от одной станции до другой. Или нужно получить минимальное количество перегонов, необходимых для соединения всех станций. Графовые алгоритмы, которые вы изучите в данной главе, помогут решить обе эти задачи. Кроме того, такие алгоритмы применимы не только к транспортным сетям, но и к любым сетевым задачам, возникающим, например, в компьютерных, распределительных и инженерных сетях. Задачи поиска и оптимизации во всех этих пространствах могут быть решены с помощью графовых алгоритмов.

4.1. Карта как граф

В этой главе мы будем работать не с графом станций метро, а с городами Соединенных Штатов Америки и возможными маршрутами между ними. На рис. 4.1 представлена карта континентальной части США с 15 крупнейшими муниципальными

статистическими районами (Metropolitan Statistical Area, MSA) страны, согласно оценкам Бюро переписи США[1].

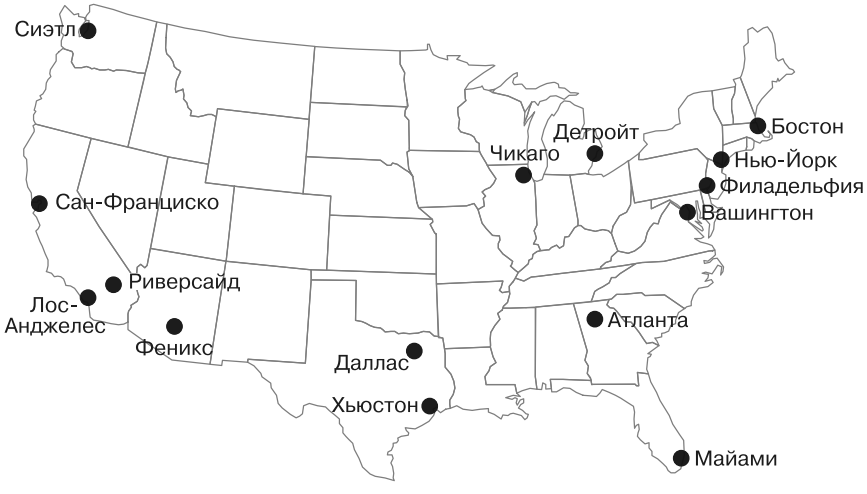

Рис. 4.1. Карта 15 крупнейших муниципальных статистических районов США

Известный предприниматель Илон Маск предложил построить высокоскоростную транспортную сеть, состоящую из капсул, перемещающихся в герметичных трубах. Маск утверждает, что капсулы будут передвигаться со скоростью 700 миль в час и позволят создать экономически эффективный междугородный транспорт для расстояний до 900 миль[2]. Маск называет эту новую транспортную систему Hyperloop. В этой главе мы рассмотрим классические графовые задачи в контексте построения такой транспортной сети.

Сначала Маск предложил идею Hyperloop для соединения Лос-Анджелеса и Сан-Франциско. Если бы потребовалось построить национальную сеть Hyperloop, то было бы целесообразно соединить крупнейшие мегаполисы Америки. На рис. 4.2 удалены границы штатов, которые были обозначены на рис. 4.1. Кроме того, каждый муниципальный статистический район связан с несколькими соседними. Это соседи не всегда являются ближайшими соседними MSA что делает граф намного интереснее..

На рис. 4.2 показан граф с вершинами, соответствующими 15 крупнейшим MSA Соединенных Штатов, и ребрами, обозначающими потенциальные маршруты Hyperloop между городами. Маршруты были выбраны в иллюстративных целях. Конечно, в состав сети Hyperloop могут входить и другие потенциальные маршруты.

---

[1] Данные предоставлены американским Бюро переписи США (American Fact Finder), https://factfinder.census.gov/.

[2] *Musk E.* Hyperloop Alpha, http://mng.bz/chmu.

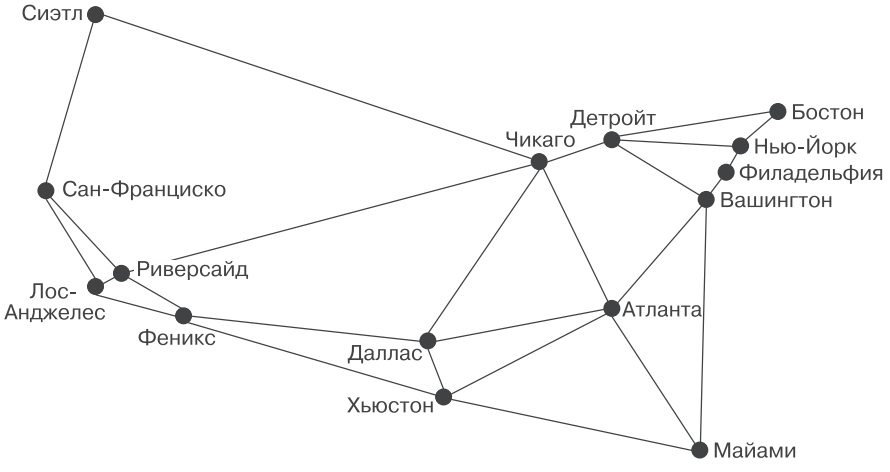

Рис. 4.2. Граф, в котором вершины представляют 15 крупнейших муниципальных статистических районов США, а ребра — потенциальные маршруты Hyperloop между ними

Такое абстрактное представление реальной задачи подчеркивает эффективность графов. Благодаря этой абстракции можно игнорировать географию Соединенных Штатов и сосредоточиться на изучении потенциальной сети Hyperloop только в контексте соединения городов. В сущности, мы можем представить задачу в виде любого графа, если только его ребра остаются неизменными. Например, на рис. 4.3 изменено положение Майами. Граф, изображенный здесь, будучи абстрактным представлением, позволяет решать те же фундаментальные вычислительные задачи, что и граф на рис. 4.2, несмотря на то что Майами на нем находится не там, где мы ожидаем. Но из соображений здравого смысла будем придерживаться представления, приведенного на рис. 4.2.

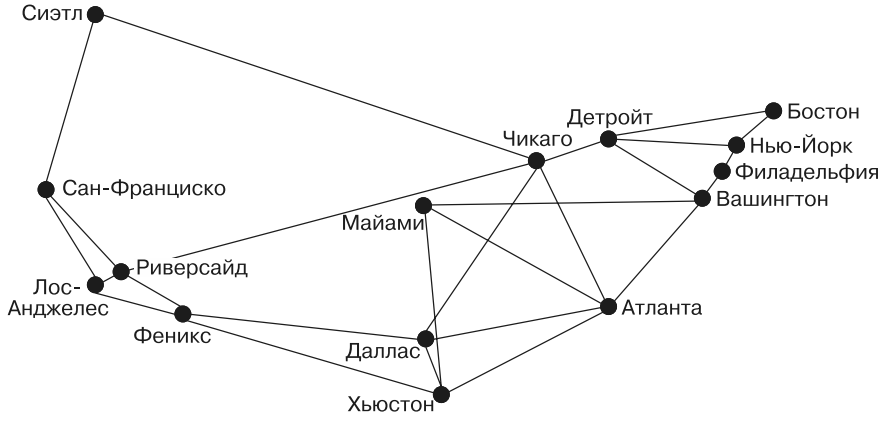

Рис. 4.3. График, эквивалентный показанному на рис. 4.2, с измененным положением Майами

4.2. Построение графовой структуры

Python позволяет программировать в самых разных стилях. Но по своей сути это объектно-ориентированный язык. В данном разделе мы построим два типа графов: невзвешенный и взвешенный. Во взвешенных графах, которые будут описаны позже, с каждым ребром ассоциируется вес — определенное число, такое как длина в нашем примере.

Мы используем модель наследования, которая является фундаментальной для объектно-ориентированных иерархий классов Python, поэтому не станем дублировать усилия. Взвешенные классы в нашей модели данных будут подклассами их невзвешенных аналогов. Это позволит таким классам наследовать бóльшую часть функциональности с небольшими изменениями, отличающими взвешенный граф от невзвешенного.

Мы хотим, чтобы графовая структура была настолько гибкой, насколько возможно, и могла представлять как можно больше различных задач. Для достижения этой цели применим параметризацию, чтобы абстрагироваться от типа вершин. В итоге каждой вершине присвоим целочисленный индекс, который будет сохранен как универсальный тип, определяемый пользователем.

Начнем построение структуры с того, что определим класс `Edge`, который является простейшим механизмом в графовой структуре (листинг 4.1).

Листинг 4.1. edge.py

```
from __future__ import annotations
from dataclasses import dataclass

@dataclass
class Edge:
    u: int # вершина "откуда"
    v: int # вершина "куда"

    def reversed(self) -> Edge:
        return Edge(self.v, self.u)

    def __str__(self) -> str:
        return f"{self.u} -> {self.v}"
```

`Edge` определяется как связь между двумя вершинами, каждая из которых представлена целочисленным индексом. По соглашению `u` определяет первую вершину, а `v` — вторую. Другими словами, `u` — «откуда», а `v` — «куда». В этой главе мы будем работать только с ненаправленными графами (графами с ребрами, которые допускают перемещение в обоих направлениях), но в направленных графах, также известных как диграфы, ребра могут быть односторонними. Метод `reversed()` должен возвращать ребро `Edge`, допускающему перемещение в направлении, противоположном направлению ребра, к которому применяется этот метод.

ПРИМЕЧАНИЕ

В классе Edge используется новая функциональность, которая появилась в Python 3.7, — классы данных. Класс, помеченный декоратором @dataclass, выполняет рутинную работу, автоматически создавая метод __init__(), который создает экземпляры переменных экземпляра для любых переменных, объявленных с аннотациями типов в теле класса. Классы данных также могут автоматически создавать другие специальные методы класса. То, какие именно специальные методы создаются автоматически, настраивается с помощью декоратора. Подробнее о классах данных читайте в документации по Python (https://docs.python.org/3/library/dataclasses.html). Проще говоря, класс данных — это способ сэкономить на вводе букв.

Класс `Graph` играет в графе важную роль — связывает вершины с ребрами. Здесь мы также хотим, чтобы действительные типы вершин могли быть любыми, какие только пожелает пользователь структуры. Это позволяет применять ее для решения широкого круга задач, не создавая промежуточных структур данных, которые склеивали бы все воедино. Например, на графе, подобном графу для маршрутов Hyperloop, можно определить тип вершин как `str`, потому что мы будем использовать в качестве вершин строки наподобие New York и Los Angeles. Начнем с класса `Graph` (листинг 4.2).

Листинг 4.2. graph.py

```
from typing import TypeVar, Generic, List, Optional
from edge import Edge

V = TypeVar('V') # тип вершин графа

class Graph(Generic[V]):
    def __init__(self, vertices: List[V] = []) -> None:
        self._vertices: List[V] = vertices
        self._edges: List[List[Edge]] = [[] for _ in vertices]
```

Список `_vertices` — это сердце графа. Все вершины сохраняются в списке, а впоследствии мы будем ссылаться на них по их целочисленному индексу в этом списке. Сама вершина может быть сложным типом данных, но ее индекс всегда имеет тип `int`, с которым легко работать. На другом уровне, между графовыми алгоритмами и массивом `_vertices`, этот индекс позволяет получить две вершины с одним и тем же именем в одном графе (представьте себе граф, в котором вершинами являются города некоей страны, причем среди них несколько носят название «Спрингфилд»). Несмотря на одинаковые имена, они будут иметь разные целочисленные индексы.

Есть много способов реализации структуры данных графа, но самыми распространенными являются использование матрицы вершин и списков смежности. В матрице вершин каждая ячейка представляет собой пересечение двух вершин графа, и значение этой ячейки указывает на связь (или ее отсутствие) между этими

вершинами. В нашей структуре данных графа задействуются списки смежности. В таком представлении у каждой вершины есть список вершин, с которыми она связана. В нашем конкретном представлении применяется список списков ребер, поэтому для каждой вершины существует список ребер, которыми она связана с другими вершинами. Этот список списков называется `_edges`.

Далее представлена остальная часть класса `Graph` во всей своей полноте (листинг 4.3). Обратите внимание на использование коротких, в основном однострочных методов с подробными и понятными именами. Благодаря этому остальная часть класса становится в значительной степени очевидной, однако она снабжена краткими комментариями, чтобы не оставалось места для неправильной интерпретации.

Листинг 4.3. graph.py (продолжение)

```
@property
def vertex_count(self) -> int:
    return len(self._vertices) # Количество вершин

@property
def edge_count(self) -> int:
    return sum(map(len, self._edges)) # Количество ребер

# Добавляем вершину в граф и возвращаем ее индекс
def add_vertex(self, vertex: V) -> int:
    self._vertices.append(vertex)
    self._edges.append([]) # Добавляем пустой список для ребер
    return self.vertex_count - 1 # Возвращаем индекс по добавленным вершинам

# Это ненаправленный граф,
# поэтому мы всегда добавляем вершины в обоих направлениях
def add_edge(self, edge: Edge) -> None:
    self._edges[edge.u].append(edge)
    self._edges[edge.v].append(edge.reversed())

# Добавляем ребро, используя индексы вершин (удобный метод)
def add_edge_by_indices(self, u: int, v: int) -> None:
    edge: Edge = Edge(u, v)
    self.add_edge(edge)

# Добавляем ребро, просматривая индексы вершин (удобный метод)
def add_edge_by_vertices(self, first: V, second: V) -> None:
    u: int = self._vertices.index(first)
    v: int = self._vertices.index(second)
    self.add_edge_by_indices(u, v)

# Поиск вершины по индексу
def vertex_at(self, index: int) -> V:
    return self._vertices[index]
```

```
# Поиск индекса вершины в графе
def index_of(self, vertex: V) -> int:
    return self._vertices.index(vertex)

# Поиск вершин, с которыми связана вершина с заданным индексом
def neighbors_for_index(self, index: int) -> List[V]:
    return list(map(self.vertex_at, [e.v for e in self._edges[index]]))

# Поиск индекса вершины; возвращает ее соседей (удобный метод)
def neighbors_for_vertex(self, vertex: V) -> List[V]:
    return self.neighbors_for_index(self.index_of(vertex))

# Возвращает все ребра, связанные с вершиной, имеющей заданный индекс
def edges_for_index(self, index: int) -> List[Edge]:
    return self._edges[index]

# Поиск индекса вершины; возвращает ее ребра (удобный метод)
def edges_for_vertex(self, vertex: V) -> List[Edge]:
    return self.edges_for_index(self.index_of(vertex))

# Упрощенный красивый вывод графа
def __str__(self) -> str:
    desc: str = ""
    for i in range(self.vertex_count):
        desc += f"{self.vertex_at(i)} -> {self.neighbors_for_index(i)}\n"
    return desc
```

Давайте ненадолго вернемся назад и посмотрим, почему большинство методов этого класса существует в двух версиях. Из определения класса мы знаем, что список `_vertices` содержит элементы типа `V`, который может быть любым классом Python. Таким образом, у нас есть вершины типа `V`, которые хранятся в списке `_vertices`. Но если мы хотим получить эти вершины и впоследствии ими манипулировать, то нужно знать, где они хранятся в данном списке. Следовательно, с каждой вершиной в этом массиве связан индекс (целое число). Если индекс вершины неизвестен, то его нужно найти, просматривая `_vertices`. Именно для этого нужны две версии каждого метода: одна оперирует внутренними индексами, другая — самим `V`. Методы, которые работают с `V`, ищут соответствующие индексы и вызывают основанную на индексе функцию. Поэтому их можно считать удобными.

Назначение большинства функций очевидно, но `neighbors_for_index()` заслуживает небольшого разъяснения. Эта функция возвращает соседей данной вершины. Соседи вершины — это все вершины, которые напрямую связаны с ней посредством ребер. Например, на рис. 4.2 Нью-Йорк и Вашингтон являются единственными соседями Филадельфии. Чтобы найти соседей вершины, нужно перебрать концы (значения `v`) всех выходящих из нее ребер:

```
def neighbors_for_index(self, index: int) -> List[V]:
    return list(map(self.vertex_at, [e.v for e in self._edges[index]]))
```

`_edges [index]` — это список смежности, то есть список ребер, посредством которых рассматриваемая вершина связана с другими вершинами. В списковом включении, передаваемом функции `map()`, `e` означает одно конкретное ребро, а `e.v` — индекс соседней вершины, с которой соединено данное ребро. `map()` возвращает все вершины, а не только их индексы, потому что `map()` применяет метод `vertex_at()` к каждому индексу `e.v`.

Еще один важный момент, на который стоит обратить внимание, — это то, как работает `add_edge()`. Сначала `add_edge()` добавляет ребро в список смежности вершины «откуда» (`u`), а затем добавляет обратную версию ребра в список смежности вершины «куда» (`v`). Второй шаг необходим, потому что этот граф ненаправленный. Мы хотим, чтобы каждое ребро было добавлено в обоих направлениях, это означает, что `u` будет соседней вершиной для `v`, а `v` — соседней для `u`. Ненаправленный граф можно представить себе как двунаправленный, если это поможет вам помнить, что каждое его ребро может быть пройдено в любом направлении:

```
def add_edge(self, edge: Edge) -> None:
    self._edges[edge.u].append(edge)
    self._edges[edge.v].append(edge.reversed())
```

Как уже упоминалось, в этой главе мы рассматриваем именно ненаправленные графы. Помимо того что графы могут быть направленными или ненаправленными, они могут быть также взвешенными или невзвешенными. Взвешенный граф — это такой граф, у которого с каждым из ребер связано некое сопоставимое значение, обычно числовое. В потенциальной сети Hyperloop мы могли бы представить вес ребер как расстояния между станциями. Однако сейчас будем иметь дело с невзвешенной версией графа. Невзвешенное ребро — это просто связь между двумя вершинами, следовательно, классы `Edge` и `Graph` являются невзвешенными. Другой способ показать это: о невзвешенном графе известно только то, какие вершины связаны, а о взвешенном — то, какие вершины связаны, и кое-какая дополнительная информация об этих связях.

4.2.1. Работа с Edge и Graph

Теперь, когда у нас есть конкретные реализации `Edge` и `Graph`, можем создать представление потенциальной сети Hyperloop. Вершины и ребра в `city_graph` соответствуют вершинам и ребрам, представленным на рис. 4.2. Используя параметризацию, мы можем указать, что вершины будут иметь тип `str` (`Graph[str]`). Другими словами, тип `str` заменяет переменную типа `V` (листинг 4.4).

Листинг 4.4. graph.py (продолжение)

```
if __name__ == "__main__":
    # тест простейшей графовой конструкции
    city_graph: Graph[str] = Graph(["Seattle", "San Francisco", "Los
     Angeles", "Riverside", "Phoenix", "Chicago", "Boston", "New York",
     "Atlanta", "Miami", "Dallas", "Houston", "Detroit", "Philadelphia",
     "Washington"])
```

```
city_graph.add_edge_by_vertices("Seattle", "Chicago")
city_graph.add_edge_by_vertices("Seattle", "San Francisco")
city_graph.add_edge_by_vertices("San Francisco", "Riverside")
city_graph.add_edge_by_vertices("San Francisco", "Los Angeles")
city_graph.add_edge_by_vertices("Los Angeles", "Riverside")
city_graph.add_edge_by_vertices("Los Angeles", "Phoenix")
city_graph.add_edge_by_vertices("Riverside", "Phoenix")
city_graph.add_edge_by_vertices("Riverside", "Chicago")
city_graph.add_edge_by_vertices("Phoenix", "Dallas")
city_graph.add_edge_by_vertices("Phoenix", "Houston")
city_graph.add_edge_by_vertices("Dallas", "Chicago")
city_graph.add_edge_by_vertices("Dallas", "Atlanta")
city_graph.add_edge_by_vertices("Dallas", "Houston")
city_graph.add_edge_by_vertices("Houston", "Atlanta")
city_graph.add_edge_by_vertices("Houston", "Miami")
city_graph.add_edge_by_vertices("Atlanta", "Chicago")
city_graph.add_edge_by_vertices("Atlanta", "Washington")
city_graph.add_edge_by_vertices("Atlanta", "Miami")
city_graph.add_edge_by_vertices("Miami", "Washington")
city_graph.add_edge_by_vertices("Chicago", "Detroit")
city_graph.add_edge_by_vertices("Detroit", "Boston")
city_graph.add_edge_by_vertices("Detroit", "Washington")
city_graph.add_edge_by_vertices("Detroit", "New York")
city_graph.add_edge_by_vertices("Boston", "New York")
city_graph.add_edge_by_vertices("New York", "Philadelphia")
city_graph.add_edge_by_vertices("Philadelphia", "Washington")
print(city_graph)
```

У `city_graph` есть вершины типа `str` — мы указываем для каждой название MSA, который она представляет. Последовательность, в которой добавляем ребра в `city_graph`, не имеет значения. Поскольку мы реализовали `__str__()` с красиво напечатанным описанием графа, теперь можно структурно распечатать (это настоящий термин!) граф. У вас должен получиться результат, подобный следующему:

```
Seattle -> ['Chicago', 'San Francisco']
San Francisco -> ['Seattle', 'Riverside', 'Los Angeles']
Los Angeles -> ['San Francisco', 'Riverside', 'Phoenix']
Riverside -> ['San Francisco', 'Los Angeles', 'Phoenix', 'Chicago']
Phoenix -> ['Los Angeles', 'Riverside', 'Dallas', 'Houston']
Chicago -> ['Seattle', 'Riverside', 'Dallas', 'Atlanta', 'Detroit']
Boston -> ['Detroit', 'New York']
New York -> ['Detroit', 'Boston', 'Philadelphia']
Atlanta -> ['Dallas', 'Houston', 'Chicago', 'Washington', 'Miami']
Miami -> ['Houston', 'Atlanta', 'Washington']
Dallas -> ['Phoenix', 'Chicago', 'Atlanta', 'Houston']
Houston -> ['Phoenix', 'Dallas', 'Atlanta', 'Miami']
Detroit -> ['Chicago', 'Boston', 'Washington', 'New York']
Philadelphia -> ['New York', 'Washington']
Washington -> ['Atlanta', 'Miami', 'Detroit', 'Philadelphia']
```