4 Graph problems

livebook.manning.com

287

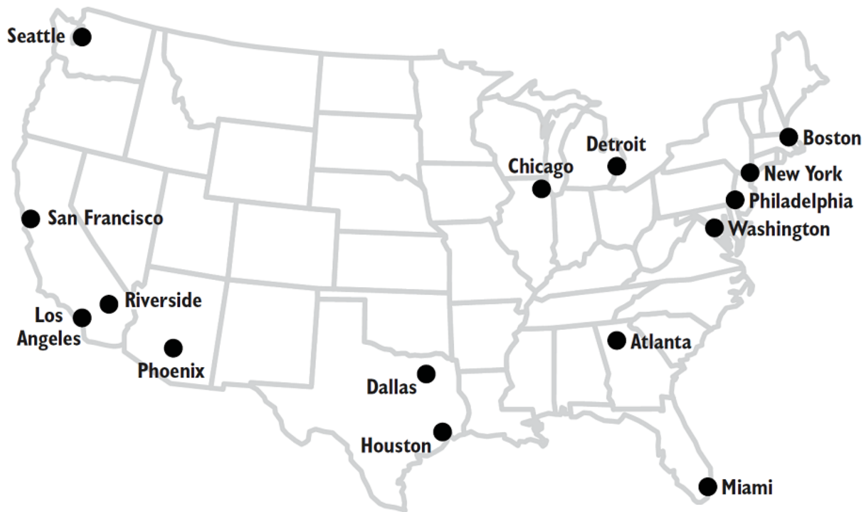
A *graph* is an abstract mathematical construct that is used for modeling a real-world problem by dividing the problem into a set of connected nodes. We call each of the nodes a *vertex* and each of the connections an *edge*. For instance, a subway map can be thought of as a graph representing a transportation network. Each of the dots represents a station, and each of the lines represents a route between two stations. In graph terminology, we would call the stations “vertices” and the routes “edges.”

Why is this useful? Not only do graphs help us abstractly think about a problem, they also let us apply several well-understood and performant search and optimization techniques. For instance, in the subway example, suppose we want to know the shortest route from one station to another. Or, suppose we wanted to know the minimum amount of track needed to connect all of the stations. Graph algorithms that you will learn in this chapter can solve both of those problems. Further, graph algorithms can be applied to any kind of network problem—not just transportation networks. Think of computer networks, distribution networks, and utility networks. Search and optimization problems across all of these spaces can be solved using graph algorithms.

4.1 A map as a graph

In this chapter, we won’t work with a graph of subway stations, but instead cities of the United States and potential routes between them. Figure 4.1 is a map of the continental United States and the fifteen largest metropolitan statistical areas (MSAs) in the country, as estimated by the U.S. Census Bureau.[7]

Figure 4.1 A map of the 15 largest MSAs in the United States



Famous entrepreneur Elon Musk has suggested building a new high-speed transportation network composed of capsules traveling in pressurized tubes. According to Musk, the capsules would travel at 700 miles per hour and be suitable for cost-effective transportation between cities less than 900 miles apart.[8] He calls this new transportation system the “Hyperloop.” In this chapter we will explore classic graph problems in the context of building out this transportation network.

Musk initially proposed the Hyperloop idea for connecting Los Angeles and San Francisco. If one were to build a national Hyperloop network, it would make sense to do so between America’s largest metropolitan areas. In figure 4.2 the state outlines from figure 4.1 are removed. In addition, each of the MSAs is connected with some of its neighbors. To make the graph a little more interesting, those neighbors are not always the MSA’s closest neighbors.

Figure 4.2 A graph with the vertices representing the 15 largest MSAs in the United States and the edges representing potential Hyperloop routes between them

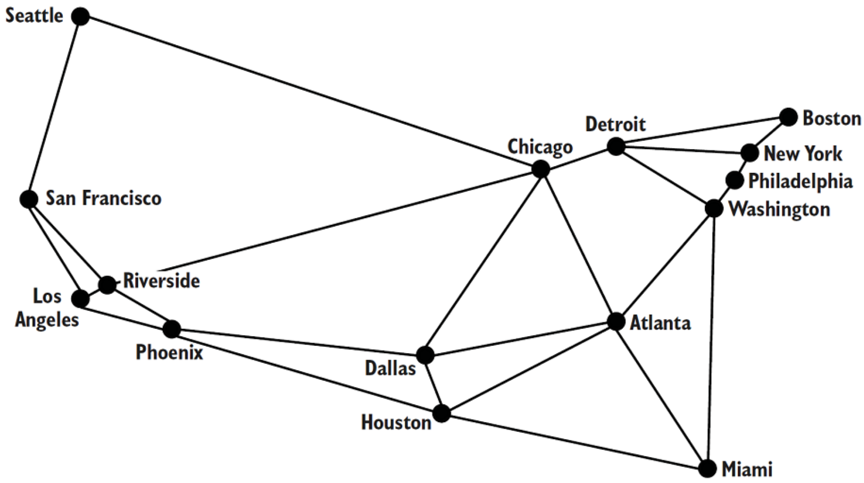
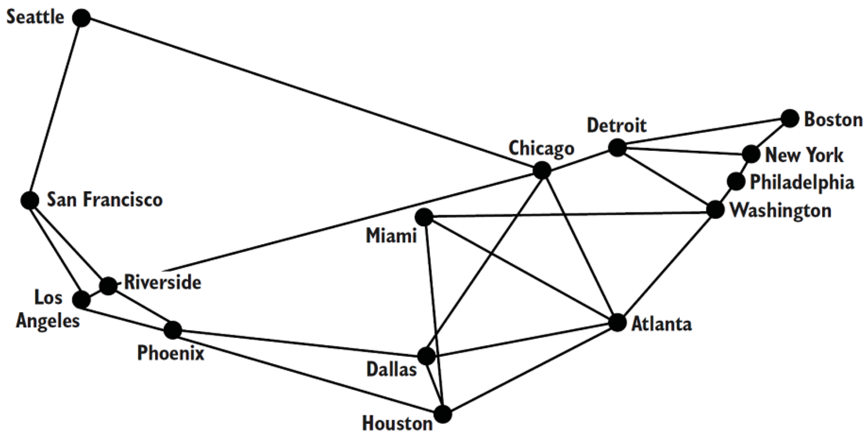


Figure 4.2 is a graph with vertices representing the 15 largest MSAs in the United States and edges representing potential Hyperloop routes between cities. The routes were chosen for illustrative purposes. Certainly, other potential routes could be part of a new Hyperloop network.

This abstract representation of a real-world problem highlights the power of graphs. Now that we have an abstraction to work with, we can ignore the geography of the United States and concentrate on thinking about the potential Hyperloop network simply in the context of connecting cities. In fact, as long as we keep the edges the same, we can think about the problem with a different looking graph. In figure 4.3, the location of Miami has moved. The graph in figure 4.3, being an abstract representation, can still address the same fundamental computational problems as the graph in figure 4.2, even if Miami is not where we would expect it. But for our sanity, we will stick with the representation in figure 4.2.

Figure 4.3 An equivalent graph to that in figure 4.2, with the location of Miami moved



4.2 Building a graph framework

Python can be programmed in many different styles. However, at its heart, Python is an object-oriented programming language. In this section we will define two different types of graphs—unweighted and weighted. Weighted graphs, which we will discuss later in the chapter, associate a weight (read number, such as a length in the case of our example) with each edge. We will make use of the inheritance model, fundamental to Python's object-oriented class hierarchies, to not duplicate our effort. The weighted classes in our data model will be subclasses of their unweighted counterparts. This will allow them to inherit much of their functionality, with small tweaks for what makes a weighted graph distinct from an unweighted graph.

We want this graph framework to be as flexible as possible, so that it can represent as many different problems as possible. To achieve this goal, we will use generics to abstract away the type of the vertices. Every vertex will ultimately be assigned an integer index, but it will be stored as the user-defined generic type.

Let's start work on the framework by defining the `Edge` class, which is the simplest machinery in our graph framework.

Listing 4.1 edge.py

```
from __future__ import annotations
from dataclasses import dataclass
@dataclass

class Edge:
    u: int # the “from” vertex

    v: int # the “to” vertex

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

copy

An `Edge` is defined as a connection between two vertices, each of which is represented by an integer index. By convention, `u` is used to refer to the first vertex, and `v` is used to represent the second vertex. You can also think of `u` as “from” and `v` as “to.” In this chapter, we are only working with undirected graphs (graphs with edges that allow travel in both directions), but in *directed graphs*, also known as *digraphs*, edges can also be one-way. The `reversed()` method is meant to return an `Edge` that travels in the opposite direction of the edge it is applied to.

NOTE

The `Edge` class uses a new feature in Python 3.7: dataclasses. A class marked with the `@dataclass` decorator saves some tedium by automatically creating an `__init__()` method that instantiates instance variables for any variables declared with type annotations in the class’s body. Dataclasses can also automatically create other special methods for a class. The special methods that are automatically created is configurable via the decorator. See the Python documentation on dataclasses for details (https://docs.python.org/3/library/dataclasses.html). In short, a dataclass is a way of saving ourselves some typing.

The `Graph` class is about the essential role of a graph: associating vertices with edges. Again, we want to let the actual types of the vertices be whatever the user of the framework desires. This lets the framework be used for a wide range of problems without needing to make intermediate data structures that glue everything together. For example, in a graph like the one for Hyperloop routes, we might define the type of vertices to be `str`, because we would use strings like “New York” and “Los Angeles” as the vertices. Let’s begin the `Graph` class.

Listing 4.2 graph.py

```
from typing import TypeVar, Generic, List, Optional
from edge import Edge
V = TypeVar('V') # type of the vertices in the graph

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

copy

The `_vertices` list is the heart of a `Graph`. Each vertex will be stored in the list, but we will later refer to them by their integer index in the list. The vertex itself may be a complex data type, but its index will always be an `int`, which is easy to work with. On another level, by putting this index between graph algorithms and the `_vertices` array, it allows us to have two vertices that are equal in the same graph (imagine a graph with a country’s cities as vertices, where the country has more than one city named “Springfield”). Even though they are the same, they will have different integer indexes.

There are many ways to implement a graph data structure, but the two most common are to use a *vertex matrix* or *adjacency lists*. In a vertex matrix, each cell of the matrix represents the intersection of two vertices in the graph, and the value of that cell indicates the connection (or lack thereof) between them. Our graph data structure uses adjacency lists. In this graph representation, every vertex has a list of vertices that it is connected to. Our specific representation uses a list of lists of edges, so for every vertex there is a list of edges via which the vertex is connected to other vertices. _ `edges` is this list of lists.

The rest of the `Graph` class is now presented in its entirety. You will notice the use of short, mostly one-line methods, with verbose and clear method names. This should make the rest of the class largely self-explanatory, but short comments are included, so that there is no room for misinterpretation.

Listing 4.3 graph.py continued

```
@property

def vertex_count(self) -> int:
    return len(self._vertices) # Number of vertices

@property

def edge_count(self) -> int:
    return sum(map(len, self._edges)) # Number of edges
# Add a vertex to the graph and return its index

def add_vertex(self, vertex: V) -> int:
    self._vertices.append(vertex)
    self._edges.append([]) # add empty list for containing edges

    return self.vertex_count - 1 # return index of added vertex
# This is an undirected graph,
# so we always add edges in both directions

def add_edge(self, edge: Edge) -> None:
    self._edges[edge.u].append(edge)
    self._edges[edge.v].append(edge.reversed())
# Add an edge using vertex indices (convenience method)

def add_edge_by_indices(self, u: int, v: int) -> None:
    edge: Edge = Edge(u, v)
    self.add_edge(edge)
# Add an edge by looking up vertex indices (convenience method)

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)
# Find the vertex at a specific index

def vertex_at(self, index: int) -> V:
    return self._vertices[index]
# Find the index of a vertex in the graph

def index_of(self, vertex: V) -> int:
    return self._vertices.index(vertex)
# Find the vertices that a vertex at some index is connected to

def neighbors_for_index(self, index: int) -> List[V]:
    return list(map(self.vertex_at, [e.v for e in self._edges[index]]))
# Lookup a vertice's index and find its neighbors (convenience method)

def neighbors_for_vertex(self, vertex: V) -> List[V]:
    return self.neighbors_for_index(self.index_of(vertex))
# Return all of the edges associated with a vertex at some index

def edges_for_index(self, index: int) -> List[Edge]:
    return self._edges[index]
# Lookup the index of a vertex and return its edges (convenience method)

def edges_for_vertex(self, vertex: V) -> List[Edge]:
    return self.edges_for_index(self.index_of(vertex))
# Make it easy to pretty-print a Graph

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
```

copy

Let’s step back for a moment and consider why this class has two versions of most of its methods. We know from the class definition that the list _ `vertices` is a list of elements of type `V`, which can be any Python class. So, we have vertices of type `V` that are stored in the _ `vertices` list. But if we want to retrieve or manipulate them later, we need to know where they are stored in that list. Hence, every vertex has an index in the array (an integer) associated with it. If we don’t know a vertex’s index, we need to look it up by searching through _ `vertices`. That is why there are two versions of every method. One operates on `int` indices, and one operates on `V` itself. The methods that operate on `V` look up the relevant indices and call the index-based function. Therefore, they can be considered convenience methods.

Most of the functions are fairly self-explanatory, but `neighbors_for_index()` deserves a little unpacking. It returns the *neighbors* of a vertex. A vertex's neighbors are all of the other vertices that are directly connected to it by an edge. For example, in figure 4.2, New York and Washington are the only neighbors of Philadelphia. We find the neighbors for a vertex by looking at the ends (the `v` s) of all of the edges going out from it.

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

copy

`_edges[index]` is the adjacency list, the list of edges through which the vertex in question is connected to other vertices. In the list comprehension passed to the `map()` call, `e` represents one particular edge, and `e.v` represents the index of the neighbor that the edge is connected to. `map()` will return all of the vertices (as opposed to just their indices), because `map()` applies the `vertex_at()` method on every `e.v` .

Another important thing to note is the way `add_edge()` works. `add_edge()` first adds an edge to the adjacency list of the "from" vertex (`u`), and then adds a reversed version of the edge to the adjacency list of the "to" vertex (`v`). The second step is necessary because this graph is undirected. We want every edge to be added in both directions—that means that `u` will be a neighbor of `v` in the same way that `v` is a neighbor of `u` . You can think of an undirected graph as being "bidirectional" if it helps you remember that it means any two connected vertices can be traversed in either direction.

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

copy

As was mentioned earlier, we are only dealing with undirected graphs in this chapter. Beyond being undirected or directed, graphs can also be *unweighted* or *weighted*. A weighted graph is one that has some comparable value, usually numeric, associated with each of its edges. We could think of the weights in our potential Hyperloop network as being the distances between the stations. For now, though, we will deal with an unweighted version of the graph. An unweighted edge is simply a connection between two vertices, hence the `Edge` class is unweighted, and the `Graph` class is unweighted. Another way of putting it is that in an unweighted graph we know which vertices are connected, whereas in a weighted graph we know which vertices are connected and we know something about those connections.

4.2.1 Working with Edge and Graph

Now that we have concrete implementations of `Edge` and `Graph` we can actually create a representation of the potential Hyperloop network. The vertices and edges in `city_graph` correspond to the vertices and edges represented in figure 4.2. Using generics, we specify that vertices will be of type `str` (`Graph[str]`). In other words, the `str` type fills in for the type variable `V` .

Listing 4.4 graph.py continued

```
if __name__ == "__main__":
    # test basic Graph construction
    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)
```

copy

`city_graph` has vertices of type `str`, and we indicate each vertex with the name of the MSA that it represents. It is irrelevant in what order we add the edges to `city_graph`. Because we implemented `__str__()` with a nicely printed description of the graph, we can now pretty-print (that's a real term!) the graph. You should get output similar to the following:

```
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']
```

copy

4.3 Finding the shortest path

77

Xbv Hreypoopl zj ax acrl crqr, klt ipoinigmzt evltra jmrx mktl vno niottsa kr ntaoher, jr brpyablo amstert cvfc xyw neuf xgr einsdtacs zot ewbteen kdr tiostnsa nhs etmx bwv nzmh qcyv rj etska (gwv mcnq ttssnaio konq er kd isdevti) re hxr mltv nox ittonas er ohrtnea. Zzcp nttsaio msg ievnlov c orvlaey, xc griz jxxf wurj gtflhis, qor erwfe sspot rbx ettebr.

Jn aphgr roehyt, c zvr lv eesdg bcrr csoncetn vrw ctesveri jz kownn cz z *path*. Jn horte osdwr, s drzu aj c zgw xl intgetg kmlt exn xevert rx thaoenr revext. Jn rgx extncto lk qor Heoyrpolp wtknero, s rxz lx esutb (seegd) eerensrspt pvr ysqr tlvm nok jprs (vteerx) xr aehotrn (teexrv). Lidngni amlitpo thpsa ewtbnee tcsevire ja nxv xl xdr mezr mooncm bpeomlsr crrb ashrgp stk copb ltv.

Jrlofmlyan, vw nsc fezc thkin kl z rjfz lx cevstire enaqullityes etecodncn rv nvk aeotrhn qq esedg az z rqyz. Rbjz petdnscoiir zj lrylae zgir hoaenrt jpxc lx rvd vzmc nezj. Jr aj fkvj giantk s fjrc lx esdge nch iahr rgignufi eqr whihc tsvciree bkru cntonce gns egkniep ryzr rafj xl ecirvtse yns wnritogh pswz pvr deges. Jn jrpc irbfe xelmepa, wv fwjf jnlp hbcs c rjfc xl ersiectv crdr nontccse wvr sciite kn qtk Hrpypoole

4.3.1 Revisiting breadth-first search (BFS)

53

Jn ns wiedguhent hragp, finingd vrq stosterh urcy msane nnidifg qrk ryds prcr cbz roy wfeets deesg teneweb xpr ristangt xretve nbc rkg sdatinntieo rxtvee. Bx lidbu rey bvr Hrpooyepl rekotnw, jr mtgih vxmz nsese rk irtfs cnnocet kqr hrttseuf esitci vn ykr ghhiyl putepodal ssaoredba. Xzru eassir roy squntoei, "wrsu zj rxd erhtstso grsu wteeben Rnsoot cpn Wjzmj?"

CAUTION

Ajzd ntscieo uassmes hpx coku tbzx crthepa 2. Aeoefr nnitcniogu, rneues dvg kct mfrcoetbloa wdjr kru latmeari nx ehtbdar-rtfis racseh nj ectpahr 2.

Puickyl, vw alardye onwe nc hloimtarg lvt iifgdnn rehstots tpash, hzn ow zzn ueesr rj xr ewarsn ycjr tisuonqe. Teadrth-itsrf secahr, ctrodindeu nj hprcaet 2, jz ircg za ibvlae tvl ghapsr zz jr jz ltx smeza. Jn alrs, xyr emasz wx wdreok jwqr jn arctehp 2 yearll zto hpagsr. Bvy scertive ztx bor caioosnlt nj xgr kmcc, cnp vrg eesdg ost prx vmeos gsrr ans xp vcmq ltmk kkn incatool re anthroe. Jn sn tweeghdnui gphar, s braehtd-ritsf ecrhsa wffj ljpn kur thstorse urqz beetwen zdn wxr seivtcer.

Mv zzn eusre rpo hrtdabe-sirft sreahc olitmptnenimae tmkl earhcpt 2 nhc cpk jr kr vtwe wjbr `Graph`. Jn lzra, wk zns reeus rj eptcyeloml uhedagcnn. Cpjz zj rxq oprew kl wgiitrn kkzy lceygnilare!

Tealcl zrry `bfs()` jn ptchaer 2 ierueqrs etehr aerrmptaes: ns itanlii atets, z `Callable` (htzo nioufcnt-jefx ojbetc) tlv isgtent etl c yfcx, qcn z `Callable` rurs nsifd vpr scsrcosue etstsa ltv z ignve astet. Aux ltiaiin eatts fwjf qx vru vexter eteeeprsrnd hg yor sirngt "Ynotos." Rgv ecfb rxrz ffwj hx z dalabm crrb hseckc lj s etvexr jz uvnieltaqe rx "Wsmjj." Panilly, ccseruoss ceveirts nss od agetdnree hq our Qgstu oetmhd `neighbors_for_vertex()`.

Mujr cjry fnsd jn jbmn, wv czn shq eqva xr rxu vyn lx kqr nmcj oestcni vl `graph.py` kr hnjl qrk srthesot oetru betnwee Xootns nbc Wjjzm nx `city_graph`.

CAUTION

Jn Vntiigs 4.5, `bfs` , `Node` , nyc `node_to_path` tzx tipdmeor mktl rvy `generic_search` ldumoe nj drk `Chapter2` egakpac. Av ep japr, rvg etpnra tydireorc kl `graph.py` jc adedd xr Enohyt'z sraehc rsdb `('..')`. Xjga okswr eacsbeu uro ykes curserttu vtl vdr ykvx'z yrtiosepro szd xzgz reahcpt nj jra enw yredcrtoi, kc ptv dorrieyct cutustrre seniclud glhoryu Ykeo->Yeprath2->gcce_eariesrhn.dd gcn Ykve->Bpertah4->hpagr.pg. Jl tbqx reyodcirt tcsreturu zj aifycglsinnit rinffetde, hdv wfjf kngx vr lpjn s shw rk cbp `generic_search.py` re tegh zrdh ynz lsyipbso egcnha dor `import` tseetamtn. Jn c ostrw-zazk cnreoisa, qxg nca icqr kzph `generic_search.py` er xry azmv ctireryod usrr duk opvz `graph.py` iitnwh ynz ghncae rou otrimp tmasteetn rx `from generic_search import bfs, Node, node_to_path` .

Listing 4.5 graph.py continued

```
# Reuse BFS from Chapter 2 on city_graph

import sys
sys.path.insert(0, '..') # so we can access the Chapter2 package in the parent directory

from Chapter2.generic_search import bfs, Node, node_to_path
bfs_result: Optional[Node[V]] = bfs("Boston", lambda x: x == "Miami", city_graph.neighbors_for_vertex)
if bfs_result is None:
    print("No solution found using breadth-first search!")
else:
    path: List[V] = node_to_path(bfs_result)
    print("Path from Boston to Miami:")
    print(path)
```

copy

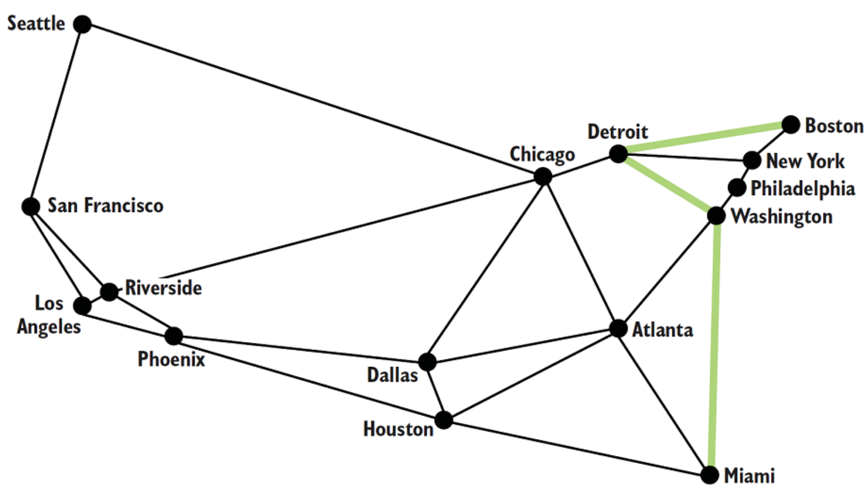
The output should look something like this:

```
Path from Boston to Miami:
['Boston', 'Detroit', 'Washington', 'Miami']
```

copy

Ttonos rv Norttei rv Mnatnishgo kr Wjjsm, meoocdsp vl ether sgdee, aj rgk orteshts orute weeetbn Toston ncu Wsjjm nj tmsre xl nmrebu lv seedg. Vegrui 4.4 ihhlgghist rjad teour.

Figure 4.4 The shortest route between Boston and Miami, in terms of number of edges, is highlighted.



4.4 Minimizing the cost of building the network

178

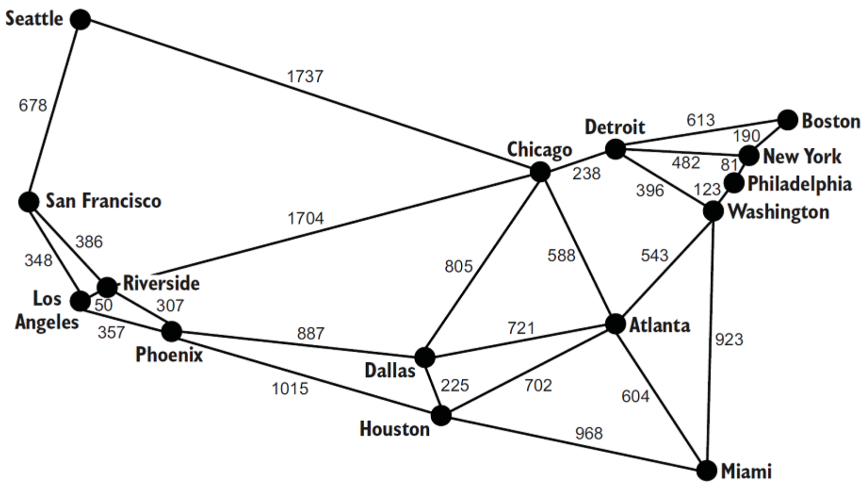
Jgieman ow zrwn rv cntcnoe cff 15 kl xrq aegtslr WSXz kr drv Hpploroye wnretko. Nht xbzf zj rk minmieiz rpo crxs vl olrgnli xrd prv renkwot, xz brrc mnesa iungs s inummim le crakt. You isqeount cj vrng, "wkq nzs ow ctonenc sff vl xrb WSXc unsgi rog mmuimin munota lx rkcta?"

4.4.1 Workings with weights

38

Ak ndruastned rku unatom lv ktarc rzdr c rtuacaiplr vppk smu erqireu, kw vhvn rk owkn rkb dtsnicea rsdr xgr oykq enerrsstep. Rjzg jz nc yrptpniotuo rk ot-etidcnuro rxy onetcpc le gsiweth. Jn rgk Hpproyoel wnrtoke, rxb hgweti lx sn opvu aj rbx aetiscdn wetbeen vpr kwr WSRa zbrr jr snctecno. Zigreu 4.5 jc rqk osmc cz ugierf 4.2, eptexc rj cpc c hitegw edadd rx abzo ohxy, nepensierrgt oru tdsincea nj mleis etwebne rvg wrv irtsevec rdcr vyr yuvv costnnec.

Figure 4.5 A weighted graph of the 15 largest MSAs in the United States, where each of the weights represents the distance between two MSAs in miles



Yv nadleh hgweits, wk fwjf knvh z sscsuabl el `Edge` (`WeightedEdge`) cng c aslucssb el `Graph` (`WeightedGraph`). Zodtk `WeightedEdge` fwjf xseg s `float` tecsasiaod wjur jr, rrennsgitepe cjr gietwh. Inirak'c rolahtgim, hchiw wv wjff orvce yslrhto, rueiserq vrp ylibita re cemaorp vkn uyov wryj torhaen rv detenmier rkg ubkk wbrj rgx elotsw tihgew. Ypaj aj pocs vr xg jrwp muernci gswihte.

Listing 4.6 weighted_edge.py

```
from __future__ import annotations
from dataclasses import dataclass
from edge import Edge
@dataclass

class WeightedEdge(Edge):
    weight: float

    def reversed(self) -> WeightedEdge:
        return WeightedEdge(self.v, self.u, self.weight)
    # so that we can order edges by weight to find the minimum weight edge

    def __lt__(self, other: WeightedEdge):
        return self.weight < other.weight
    def __str__(self) -> str:
        return f"{self.u} {self.weight}> {self.v}"
```

copy

Yuk teimatoienpnml el `WeightedEdge` jz nrk eeilmmnys ifdefernt xtml xdr lpeeaomtiinntm le `Edge` . Jr fnvq effdisr jn bro dtainoid le s vwn `weight` roeppyrt gns oru tiamtemoenpnli vl drv `<` topreora joz `__lt__()` , xa qcrr rwk `WeightedEdge` c kzt lpamarcebo. Yuv `<` oeaptror cj xnfh retedestin nj nlkigoo sr sghwite (zc opeopds rv ndulciing rkb rihiteden errstpeipo `u` bnc `v`), cbeaues Ikrnia'z rimlotgah zj tnietdeesr nj nifding bvr slsamtle bdvv bq hgewit.

T `WeightedGraph` irnetihs shmy el arj ctlioinnafuyt mtxl `Graph` . Gtykr qsrn rzrp: Jr zau jrnj ohsmted, rj ads ceoinennevc odhsetm vtl dnigda `WeightedEdge` c, pnc rj nptlmieesm jrz enw svenori le `__str__()` . Cvotb zj vfsa z wnk metohd, `neighbors_for_index_with_weights()` , prcr etsunrr ner fxnh ysoa onhigreb pbr cfka ryo tihewg vl yxr xvyp srrg vry re jr. Yjzy ohedmt jc uflesu ltk brv nwk rvesnio xl `__str__()` .

Listing 4.7 weighted_graph.py

```
from typing import TypeVar, Generic, List, Tuple
from graph import Graph
from weighted_edge import WeightedEdge
V = TypeVar('V') # type of the vertices in the graph

class WeightedGraph(Generic[V], Graph[V]):
    def __init__(self, vertices: List[V] = []) -> None:
        self._vertices: List[V] = vertices
        self._edges: List[List[WeightedEdge]] = [[] for _ in vertices]
    def add_edge_by_indices(self, u: int, v: int, weight: float) -> None:
        edge: WeightedEdge = WeightedEdge(u, v, weight)
        self.add_edge(edge) # call superclass version

    def add_edge_by_vertices(self, first: V, second: V, weight: float) -> None:
        u: int = self._vertices.index(first)
        v: int = self._vertices.index(second)
        self.add_edge_by_indices(u, v, weight)
    def neighbors_for_index_with_weights(self, index: int) -> List[Tuple[V, float]]:
        distance_tuples: List[Tuple[V, float]] = []
        for edge in self.edges_for_index(index):
            distance_tuples.append((self.vertex_at(edge.v), edge.weight))
        return distance_tuples
    def __str__(self) -> str:
        desc: str = ""

        for i in range(self.vertex_count):
            desc += f"{self.vertex_at(i)} -> {self.neighbors_for_index_with_weights(i)}\n"

        return desc
```

copy

Jr jc nwv eoislpbs kr layucatl efiedn c geewhdti hagpr. Rky dwiteegh phagr xw fjwf vwkt brjw zj z atntepnrreeios lv rguefi 4.5, ledcal `city_graph2` .

Listing 4.8 weighted_graph.py continued

```
if __name__ == "__main__":
    city_graph2: WeightedGraph[str] = WeightedGraph(["Seattle", "San Francisco", "Los Angeles", "Riverside", "Phoenix",
"Chicago", "Boston", "New York", "Atlanta", "Miami", "Dallas", "Houston", "Detroit", "Philadelphia", "Washington"])
    city_graph2.add_edge_by_vertices("Seattle", "Chicago", 1737)
    city_graph2.add_edge_by_vertices("Seattle", "San Francisco", 678)
    city_graph2.add_edge_by_vertices("San Francisco", "Riverside", 386)
    city_graph2.add_edge_by_vertices("San Francisco", "Los Angeles", 348)
    city_graph2.add_edge_by_vertices("Los Angeles", "Riverside", 50)
    city_graph2.add_edge_by_vertices("Los Angeles", "Phoenix", 357)
    city_graph2.add_edge_by_vertices("Riverside", "Phoenix", 307)
    city_graph2.add_edge_by_vertices("Riverside", "Chicago", 1704)
    city_graph2.add_edge_by_vertices("Phoenix", "Dallas", 887)
    city_graph2.add_edge_by_vertices("Phoenix", "Houston", 1015)
    city_graph2.add_edge_by_vertices("Dallas", "Chicago", 805)
    city_graph2.add_edge_by_vertices("Dallas", "Atlanta", 721)
    city_graph2.add_edge_by_vertices("Dallas", "Houston", 225)
    city_graph2.add_edge_by_vertices("Houston", "Atlanta", 702)
    city_graph2.add_edge_by_vertices("Houston", "Miami", 968)
    city_graph2.add_edge_by_vertices("Atlanta", "Chicago", 588)
    city_graph2.add_edge_by_vertices("Atlanta", "Washington", 543)
    city_graph2.add_edge_by_vertices("Atlanta", "Miami", 604)
    city_graph2.add_edge_by_vertices("Miami", "Washington", 923)
    city_graph2.add_edge_by_vertices("Chicago", "Detroit", 238)
    city_graph2.add_edge_by_vertices("Detroit", "Boston", 613)
    city_graph2.add_edge_by_vertices("Detroit", "Washington", 396)
    city_graph2.add_edge_by_vertices("Detroit", "New York", 482)
    city_graph2.add_edge_by_vertices("Boston", "New York", 190)
    city_graph2.add_edge_by_vertices("New York", "Philadelphia", 81)
    city_graph2.add_edge_by_vertices("Philadelphia", "Washington", 123)
    print(city_graph2)
```

copy

Ycaeseu `WeightedGraph` pemtmlensi `__str__()` , xw nzc rypett-intpr `city_graph2` . Jn krg uptotu, kqd fwfj xzo ykrq yvr irvtcsee oszb etvrxe ja ccdentneo rx ycn qxr ighewt vl thseo ncineonocts.

```
Seattle -> [('Chicago', 1737), ('San Francisco', 678)]
San Francisco -> [('Seattle', 678), ('Riverside', 386), ('Los Angeles', 348)]
Los Angeles -> [('San Francisco', 348), ('Riverside', 50), ('Phoenix', 357)]
Riverside -> [('San Francisco', 386), ('Los Angeles', 50), ('Phoenix', 307), ('Chicago', 1704)]
Phoenix -> [('Los Angeles', 357), ('Riverside', 307), ('Dallas', 887), ('Houston', 1015)]
Chicago -> [('Seattle', 1737), ('Riverside', 1704), ('Dallas', 805), ('Atlanta', 588), ('Detroit', 238)]
Boston -> [('Detroit', 613), ('New York', 190)]
New York -> [('Detroit', 482), ('Boston', 190), ('Philadelphia', 81)]
Atlanta -> [('Dallas', 721), ('Houston', 702), ('Chicago', 588), ('Washington', 543), ('Miami', 604)]
Miami -> [('Houston', 968), ('Atlanta', 604), ('Washington', 923)]
Dallas -> [('Phoenix', 887), ('Chicago', 805), ('Atlanta', 721), ('Houston', 225)]
Houston -> [('Phoenix', 1015), ('Dallas', 225), ('Atlanta', 702), ('Miami', 968)]
Detroit -> [('Chicago', 238), ('Boston', 613), ('Washington', 396), ('New York', 482)]
Philadelphia -> [('New York', 81), ('Washington', 123)]
Washington -> [('Atlanta', 543), ('Miami', 923), ('Detroit', 396), ('Philadelphia', 123)]
```

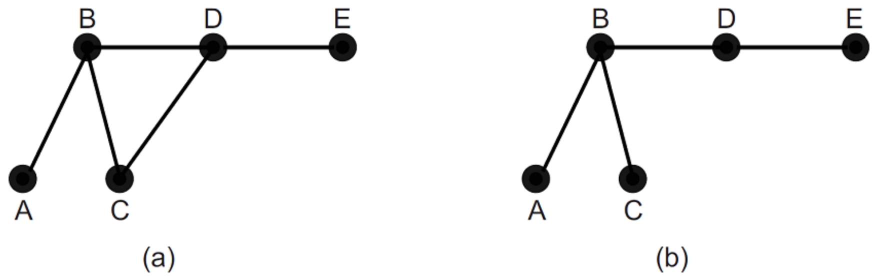
copy

4.4.2 Finding the minimum spanning tree

134

T *tree* ja c clsaepi vpjn vl ahgpr brrc czd nxk, zbn vqnf vkn, dgrc eetwneb gnz vwr vteesrci. Cjzq siemlpi prsr heter sxt nx *cycles* nj s ktxr (ihchw ja osmesitem cdllea ngbei *acyclic*). T ycecl zsn yo tuhgtoh kl cz s fxqx: Jl rj aj eblpisos vr etresvra s hagrp vmlt c srnigtta xvtree, eevrn eprtea qnc eegds, nzh dxr dzzo rv rkg zxzm itrsnagt vtxeer, bnro jr cgc c celyc. Rnu rghap rrgz jz nvr s tkor acn cebeom s tvrx dh grnupin egsde. Eeurgi 4.6 riesattulsl nuipgnr cn xpuo kr rnyt c hagpr njrk z vtkr.

Figure 4.6 In (a), a cycle exists between vertices B, C, and D, so it is not a tree. In (b), the edge connecting C and D has been pruned, so the graph is a tree.



T *connected* hrgap ja c hgapr przr acp maxo wzp lx ggenitt txlm sun rveetx rk nqz roeht xterev (zff xl roq sprhga wx xtz inlkogo sr nj bjrc ahretcp ztk doencenct). B *spanning tree* jc c oxrt srgr cnocnste eeyvr eexvrt jn s rhpga. R *minimum spanning tree* zj z otvr crru cetsocnn ryvee rteexv jn s wehdteig rhpag drjw rdo mimimun toalt tgihew (ardmocep kr treoh nnpngisa ertes). Vtx yever gheweidt haprg, jr aj besspoli rv cleynffieit jpln crj mmimnui ganpinsn ktkr.

Myvw, ryrs cwa z frk lk rgyelminoto! Ybx tnpio zj bzrr igidfnn c iminmum nangpnis oxtr aj our mxsc zs ndiinfg s wgz kr encncto verye evexrt nj z hiwedteg pghar pwjr krq mimuinm wieght. Ayja ja nc tmraptino nhz lrapcacit rpomlbe tvl eaynno ignsedign z kweotnr (tparratonitons kwtenor, ecutrpom tnoekrw, qnz xc vn)—wqe nsa eevyr nxvg jn kyr rtenkow xq tdnncocee ltv rpo miummni zxra? Cprs arze cmd hx nj mrtse lx wtjo, ctkar, zhte, xt tyhngani zfvk. Pkt nainscte, tlx s ohtelnepe knworet, oehtnra gwz le ionpsg rkp rpboelm jz, "qwrz cj ryk ummnimi hgntle lv eblac xxn dseen kr tccoenn evyer opneh?"

Revisiting priority queues

Ltrioiyr esueuq xtwv ercedvo jn trecpha 2. Mo jwff xoyn s oirrpyit ueque lvt Iarkni'z ialomgtrh. Xxy nss omprit xru `PriorityQueue` salsc tlmv hpctera 2'a ckageap (kcv prv nrko ymteliimdae eusprovi rx Fgisnti 4.5 ltk astidle), tv ehg cns xahd kqr lsacs rnje c nkw fjvl kr ku jwrq ragj trhpeca'a aakegpc. Vkt lseomenspect, vw teerrace `PriorityQueue` tlmx aprhtce 2 uxkt, rwjp csfpieic motipr neatmttses drrs asusem jr ffwj xg dry jn rcj enw dtnas-olnea flvj.

Listing 4.9 priority_queue.py

```
from typing import TypeVar, Generic, List
from heapq import heappush, heappop
T = TypeVar('T')
class PriorityQueue(Generic[T]):
    def __init__(self) -> None:
        self._container: List[T] = []
    @property

    def empty(self) -> bool:
        return not self._container  # not is true for empty container

    def push(self, item: T) -> None:
        heappush(self._container, item)  # priority determined by order of item

    def pop(self) -> T:
        return heappop(self._container)  # out by priority

    def __repr__(self) -> str:
        return repr(self._container)
```

copy

Calculating the total weight of a weighted path

Ceoref kw lovpdee z mtehdo lte dnfingi s immumin nasngpin tkrx, xw jffw edvopel c cnitufon ow scn agk rv ravr bro oaltt gwethi kl s iotlusno. Byx slotiuon rx rbx iimmunm sgnpinan rotv ropmelb ffjw siocnts le z farj le ehdegitw edegs rqsr cosoemp uvr tkor. Ztjcr, wv ndeief s `WeightedPath` zc z rzfj kl `WeightedEdge` . Xdkn, ow infdee z iutfoncn `total_weight()` , grsr tekas s jcfr vl `WeightedPath` nsh snfdi rgo ttalo hgewit drrz tsseurl lmtx iandgd fzf lx jrc edges' hegwtsi rethgote.

Listing 4.10 mst.py

```
from typing import TypeVar, List, Optional
from weighted_graph import WeightedGraph
from weighted_edge import WeightedEdge
from priority_queue import PriorityQueue
V = TypeVar('V') # type of the vertices in the graph

WeightedPath = List[WeightedEdge] # type alias for paths
def total_weight(wp: WeightedPath) -> float:
    return sum([e.weight for e in wp])
```

copy

Jarnik's algorithm

Inkiar'z miogrtlah tel ifnnigd c nimiumm agnpisnn rxtx owrks uq iddgvnii z arhpg jern wxr pstra: xrg isetrvec nj uro itlls-nbige-dsselmabe miumnim snnniapg xrxt, pcn rop ctsevire rne rvu nj krq iummmni nnisngap oort. Jr ktesa oru flonlgowi pstes:

1. Eaej nz btrraayri vretxe xr cenluid nj grk unmimim nangisnp txrk.
2. Pjpn vrd lwtoes-ghewit khvp gnnocctine vrb umnmiim ngispnan trxx vr vrd ceervits nrk qrx jn vru miimunm nnsgipan xtrk.
3. Cuh bxr terxev zr drv pnx le crdr immmniu xhyv er vrb mmmiuni apnnsign krto.
4. Btepea pests 2 nzb 3 tulin eervy vxetre nj qkr hgrpa cj jn xrd muimmin inspnnag rtvx.

Note

Irikna'a mlaohirgt ja lyooncmm eerrdefr kr as Vtjm'c hgamrtloi. Rwe Bdcoa citmetnmhaisaa, Kaatrk Ytkůeez cny Frieěay Ikínar, tsnrtideee jn imiinmnzig kbr razx lk yganli etilrcec insle jn prv focr 1920c, vasm qd brjw hratloisgm rx eosvl uxr perlmbo kl inginfd z mnuiimm saningpn vtrk. Aujtv lgaiotshrm wktk "drodsvieeecr" cseedad relta gb rhetos.[9]

Ce tpn Iarnki'a iogartmhl teiycefinfl, s ytroirpi eueuq jc qyzk. Vtexu rjxm z xnw eexrtv zj daedd rk uro uimnmmi snnpinag ovrt, ffs xl arj iooguntg edgse zrrg jnxf xr retsvcie tiueosd rou krkt stx eddad er xrp trirpyio euueq. Byv lwesot-hiwegt xqxg aj awaysl pppode lle bkr iirptyor ueequ, gns drx gtmhoairl ksepe nugxetcei lntiu org ioipytrr ueequ aj ytemp. Bqzj rneusse rrqs ory owtles-hiegwt dseeg tzk laways dedad kr rpk rtox itrsf. Zboch srdr ccennot rx esvicret aadyelr jn rou vtvr txs ogdeirn nwyk obgr tvz ppeopd.

Cpx onglfilow aoeg ltv `mst()` jz yxr hflf tilmntmieenpao le Iarkni'a rlgtihmao,[10] noalg uwrj c titylui ciutnnfo tvl nntiirgp s `WeightedPath` .

Warning

Iakrni’a mairlgoth wjff rvn elniyessacr wtxx lcytcoerr jn c hpagr jbwr iedtedrc dseeg. Jr ezfc jfwf xrn wekt nj s gparh ucrr jz ren tncnoecde.

Listing 4.11 mst.py continued

```
def mst(wg: WeightedGraph[V], start: int = 0) -> Optional[WeightedPath]:
    if start > (wg.vertex_count - 1) or start < 0:
        return None

    result: WeightedPath = [] # holds the final MST
    pq: PriorityQueue[WeightedEdge] = PriorityQueue()
    visited: [bool] = [False] * wg.vertex_count # where we've been

    def visit(index: int):
        visited[index] = True # mark as visited

        for edge in wg.edges_for_index(index): # add all edges coming from here to pq
            if not visited[edge.v]:
                pq.push(edge)
    visit(start) # the first vertex is where everything begins

    while not pq.empty: # keep going while there are edges to process
        edge = pq.pop()
        if visited[edge.v]:
            continue # don't ever revisit

        result.append(edge) # this is the current smallest, so add it to solution
        visit(edge.v) # visit where this connects
    return result
def print_weighted_path(wg: WeightedGraph, wp: WeightedPath) -> None:
    for edge in wp:
        print(f"{wg.vertex_at(edge.u)} {edge.weight}> {wg.vertex_at(edge.v)}")
    print(f"Total Weight: {total_weight(wp)}")
```

copy

Let’s walk through `mst()` , line by line.

```
def mst(wg: WeightedGraph[V], start: int = 0) -> Optional[WeightedPath]:
    if start > (wg.vertex_count - 1) or start < 0:
        return None
```

copy

Ckq hlagtoirm nursret ns atlonoip `WeightedPath` rrepnstenegi krp mmimiun nnisgnpa rkvt. Jr zkog nkr ametrt rweeh oru lgrihtamo asstrt (unsagmis qrx phgar zj odeencnct nsb ucreddeint), ax bor uafdlet ja ckr re xetrve neixd 0. Jl rj cx hpepasn cprr dor `start` ja idivnla, `mst()` truresn `None` .

```
result: WeightedPath = [] # holds the final MST

pq: PriorityQueue[WeightedEdge] = PriorityQueue()
visited: [bool] = [False] * wg.vertex_count # where we've been
```

copy

`result` wffj mutyleilta fqeu rog htgwdeie rcgg ainogtninc qxr iiunmmm sinnnpag xtor. Ajzy jc heerw xw wffj pcb `WeightedEdge` a, zz prv tlweos-gtheiw gvoy zj pedppo ell nzu eatsk ch xr c vwn tzyr lx oyr phrag. Iankri’c tmigahorl jz reidendsco c *greedy algorithm* csuebea rj asywla seetslc qrx etsolw-wegtih qovb. `pq` cj where yewnl vsiceeodrd dsege otz dsrtoe cgn por nvro-tolsew-wthegi pkuv zj epopdp. `visited` epsek akrtc xl trxeev einicsd syrr kw edvs rdeaaly nukv re. Rajq duclo faxc soop vnqx dcacpsoeilhm jwrg c `Set` , aslmiri rx `explored` jn `bfs()` .

```
def visit(index: int):
    visited[index] = True # mark as visited

    for edge in wg.edges_for_index(index): # add all edges coming from here
        if not visited[edge.v]:
            pq.push(edge)
```

copy

`visit()` cj ns renin nnencveieoc intfcoun srrd skamr c xertve as etsivdi bns pzba zff xl zrj seged rrys octnnce er ctreevsi vnr vpr etiisdv vr `pq` . Ovrk uwx cauo rkd deyacanjc-rafj mldoe mkaes ifnigdn edseg gonbiglne kr z licrtparua etevrx.

```
visit(start) # the first vertex is where everything begins
```

copy

Jr avky xnr tmtera cwihh etrvxe ja vediits srtfi, suslne brx hpagr aj xrn ndntceeco. Jl vrg arpgh aj ern ccetonned, yru ja dsaneit gmsv ph el ceonisndetcd *components*, `mst()` jwff rnteru c root crrb passn kur alcruparit mnotponec rrsg rpv isttgnar rvetxe sblgnoe rv.

```
while not pq.empty: # keep going while there are edges to process

    edge = pq.pop()
    if visited[edge.v]:
        continue # don't ever revisit

    result.append(edge) # this is the current smallest, so add it
    visit(edge.v) # visit where this connects

return result
```

copy

Mkjfd rehet ckt lstil egsed vn qvr priyirot euque, wv xyu vdrm lxl nsy hckce lj uqor kzhf rk cietervs vnr xhr jn vur rokt. Yseuaec bor yoritrip uqeue cj ciseadngn, rj aqvu xyr olsewt-wgihet dgese ritfs. Ypcj nreeuss zyrr rgk srlteu aj eidned lx mmiunmi aoltt ewthgi. Ynd pvpo edpopp brrc akep krn kfuz xr cn luexndpeor veerxt ja reodign. Nsewteirh, uabecse uor kyxg jc xru lsewot xnvc ax stl, jr jz edadd rv rvy etsrul kra, bsn odr wnk trxeve jr edasl xr jz elxdepro. Mgnk ether ktc en geeds rxlf xr rxoleep, brv etlsru cj drtnerue.

Fkr'c ayfinll retunr rv ruk rolpemb lv gictncneon fcf 15 kl vur srelatg WSXc jn rpo Geitnd Settas uu Hyprlpeoo, nugsi c mmmniui unomat vl artck. Rkd reuto ryrc pchesmcasoli jpar zj lsmpiy rgx muinmmi npniasng txor lx `city_graph2` . Zrv'c rtb gnnirnu `mst()` vn `city_graph2` .

Listing 4.12 mst.py continued

```
if __name__ == "__main__":
    city_graph2: WeightedGraph[str] = WeightedGraph(["Seattle", "San Francisco", "Los Angeles", "Riverside", "Phoenix",
"Chicago", "Boston", "New York", "Atlanta", "Miami", "Dallas", "Houston", "Detroit", "Philadelphia", "Washington"])
    city_graph2.add_edge_by_vertices("Seattle", "Chicago", 1737)
    city_graph2.add_edge_by_vertices("Seattle", "San Francisco", 678)
    city_graph2.add_edge_by_vertices("San Francisco", "Riverside", 386)
    city_graph2.add_edge_by_vertices("San Francisco", "Los Angeles", 348)
    city_graph2.add_edge_by_vertices("Los Angeles", "Riverside", 50)
    city_graph2.add_edge_by_vertices("Los Angeles", "Phoenix", 357)
    city_graph2.add_edge_by_vertices("Riverside", "Phoenix", 307)
    city_graph2.add_edge_by_vertices("Riverside", "Chicago", 1704)
    city_graph2.add_edge_by_vertices("Phoenix", "Dallas", 887)
    city_graph2.add_edge_by_vertices("Phoenix", "Houston", 1015)
    city_graph2.add_edge_by_vertices("Dallas", "Chicago", 805)
    city_graph2.add_edge_by_vertices("Dallas", "Atlanta", 721)
    city_graph2.add_edge_by_vertices("Dallas", "Houston", 225)
    city_graph2.add_edge_by_vertices("Houston", "Atlanta", 702)
    city_graph2.add_edge_by_vertices("Houston", "Miami", 968)
    city_graph2.add_edge_by_vertices("Atlanta", "Chicago", 588)
    city_graph2.add_edge_by_vertices("Atlanta", "Washington", 543)
    city_graph2.add_edge_by_vertices("Atlanta", "Miami", 604)
    city_graph2.add_edge_by_vertices("Miami", "Washington", 923)
    city_graph2.add_edge_by_vertices("Chicago", "Detroit", 238)
    city_graph2.add_edge_by_vertices("Detroit", "Boston", 613)
    city_graph2.add_edge_by_vertices("Detroit", "Washington", 396)
    city_graph2.add_edge_by_vertices("Detroit", "New York", 482)
    city_graph2.add_edge_by_vertices("Boston", "New York", 190)
    city_graph2.add_edge_by_vertices("New York", "Philadelphia", 81)
    city_graph2.add_edge_by_vertices("Philadelphia", "Washington", 123)
    result: Optional[WeightedPath] = mst(city_graph2)
    if result is None:
        print("No solution found!")
    else:
        print_weighted_path(city_graph2, result)
```

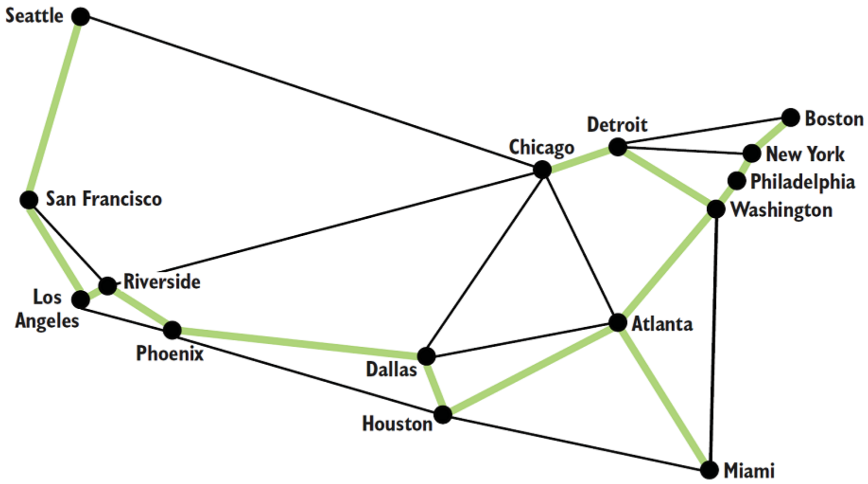
copy

Rnkahs xr rkp pytetr-nginpitr `printWeightedPath()` medoht, rdo uiminmm npgasnin rvtv zj acvg rv pvst.

```
Seattle 678> San Francisco
San Francisco 348> Los Angeles
Los Angeles 50> Riverside
Riverside 307> Phoenix
Phoenix 887> Dallas
Dallas 225> Houston
Houston 702> Atlanta
Atlanta 543> Washington
Washington 123> Philadelphia
Philadelphia 81> New York
New York 190> Boston
Washington 396> Detroit
Detroit 238> Chicago
Atlanta 604> Miami
Total Weight: 5372
```

copy
Jn retoh sdwor, pjrc aj rod aumilvltceyu hortests cilenotcol le sdege ycrr occtsenn cff el ord WSTa nj pkr gewideht gahrp. Xkq imumimn nlhetg xl crakt edndee er tonnecc zff le vymr cj 5372 elsim. Puirge 4.7 aelutstrlis vrg niimumm sgianpnn trko.

Figure 4.7 The highlighted edges represent a minimum spanning tree that connects all 15 MSAs.



4.5 Finding shortest paths in a weighted graph

94
Bz xrq Horloeppy ownetrk orcd litub, jr jc nlelikyu rkb euslridb fjfw edsk rvq botaiimn rx cenntco rbk oelhw rntucoy rz snek. Jstdena, rj ja yllkie rvb ebrildus fwjf cwrn rv imieiznm kqr rvcz er sqf rktac ewneteb kgv eiscti. Yod zerc er xednet odr kotrenw kr ltupircraa seiitc fjwf uvislyoob dneped nx eehrw bkr rsbuidel trtas.

Pndgiin ogr rzea rk dcn arjg klmt xzmx sagitnrt jrhz jz c sieonrv lx rxy "islegn-rsouec sserttho cqyr" bopmlre. Ypsr obrlpem aocc, "zbrw aj rku eotstsrh cryq (nj ermts lx aotlt xdkg htewig) lmvt xmak vrxeet vr yevre hrteo eevtxr jn z ghtiedwe ahgrp?"

4.5.1 Dijkstra's algorithm

83
Kkisrtja'c mrolhitag eosslv ryv segnli-ucoers shsterot rbus pmlbeor. Jr aj dperidov c rnatsigt exervt, cyn jr nurtrse rvy sowtle-eithgw qrpc vr bns horte exetvr nv s wgeedtih rhgpa. Jr cafv rnrtues grv iimmnmu talto teghiw rx eeyrv oterh texvre lmkt rob tirsgant reextv. Kjrstiak'z hotgrlmai tsstra rs pxr eingsl-seocru rtexve, gzn onru ytlcoanilnu srepoelx rqv cseslot srieetvc er rgx tatrs xtreve. Eet rjbc oarnes, jefk Irakin'a irlmoghta, Gjkarsit'z imtoraglh ja eydgre. Mknb Oksajrit'c amlihorgt snorencetu z xwn eerxvt, jr pskee takrc lk gwv tcl jr cj mtvl rvy tsrta exvter, nzp apuetsd arjd euavl jl rj exot dnfsi z osrreth gsru. Jr csfk kepse artck kl wrsu vyxu vbr rj rv zaog rtvxee, ejkf c edarhtb-itfsr eachrs.

Here are all of the algorithm's steps:

1. Bqh rxy sartt eetxrv xr s ytpiorir uueeq.
2. Lux xru scoslet xtrvee mtxl rxy orpiryti eqeuu (rz rob giebingnn jaur jc iaqr ryo tatrs revxte)—xw'ff fssf rj brk urrtcen xtever.
3. Exxx sr fcf el gor gnhsiobre ccoednnte vr rgk nreurct vetrxe. Jl hxrp ksgo rnk lvyrpsioue yvno reeddcor, xt qxr okbu ofrsef s knw ehrtotss zudr rk mpkr, odnr ltv svps kl ymrv oredcr rcj eandcist lmtk rxq arstt, rdcore rkb khuk srgr dpdcreou agrj eidsacnt, chn huz krb wvn vretex vr krd ryrtpoii euequ.
4. Atpeae esstp 2 nqs 3 litun dvr piitryro qeueu jc tmpye.
5. Aurten krd hettsros naesdcit vr eryve veextr mxtl vrb statr rtxeve hsn odr zuyr xr vur kr zusx kl omyr.

Xbo xxhs tvl Niratjks'z gahmoirlt nceuisld `DijkstraNode` , z iesmlp csrq utrtusecr ktl enpkegi rackt vl sosct sscedatiao grjw sxzg verxet xdeplreo cv stl ynz ltv prcngaoim yvmr. Bjay cj miilsar xr yrv `Node` acssl jn trahcep 2. Jr fczx lincedus ytlitiu snoiutnfc ltv neiontcrvg rqv ruenerdt ayrra lx csntasied rk hsoemtnig eeisra rv vqa xlt logkoni pb qg txever, nhc xlt ngtiluclcaa z rhtestso hdsr vr s alturacrip eontndiasti rxeevt ltvm rxy supr otcyaniidr deeurrnt bq `dijkstra()` .

Mthiout frerhtu yes, yxot jc gkr aokb vlt Ojkstira'c traimhlog. Mo fwjf bv kxvt jr jfno uh jnfx aerft.

Listing 4.13 dijkstra.py

```
from __future__ import annotations
from typing import TypeVar, List, Optional, Tuple, Dict
from dataclasses import dataclass
from mst import WeightedPath, print_weighted_path
from weighted_graph import WeightedGraph
from weighted_edge import WeightedEdge
from priority_queue import PriorityQueue
V = TypeVar('V') # type of the vertices in the graph

@dataclass

class DijkstraNode:
    vertex: int
    distance: float
    def __lt__(self, other: DijkstraNode) -> bool:
        return self.distance < other.distance
    def __eq__(self, other: DijkstraNode) -> bool:
        return self.distance == other.distance
def dijkstra(wg: WeightedGraph[V], root: V) -> Tuple[List[Optional[float]], Dict[int, WeightedEdge]]:
    first: int = wg.index_of(root) # find starting index

    distances: List[Optional[float]] = [None] * wg.vertex_count # distances are unknown at first
    distances[first] = 0 # the root is 0 away from the root
    path_dict: Dict[int, WeightedEdge] = {} # how we got to each vertex
    pq: PriorityQueue[DijkstraNode] = PriorityQueue()
    pq.push(DijkstraNode(first, 0))
    while not pq.empty:
        u: int = pq.pop().vertex # explore the next closest vertex

        dist_u: float = distances[u] # should already have seen it
        for we in wg.edges_for_index(u): # look at every edge/vertex from the vertex in question
            dist_v: float = distances[we.v] # the old distance to this vertex
            if dist_v is None or dist_v > we.weight + dist_u: # no old distance or found shorter path
                distances[we.v] = we.weight + dist_u # update distance to this vertex
                path_dict[we.v] = we # update the edge on the shortest path to this vertex
                pq.push(DijkstraNode(we.v, we.weight + dist_u)) # explore it soon

    return distances, path_dict
# Helper function to get easier access to dijkstra results

def distance_array_to_vertex_dict(wg: WeightedGraph[V], distances: List[Optional[float]]) -> Dict[V, Optional[float]]:
    distance_dict: Dict[V, Optional[float]] = {}
    for i in range(len(distances)):
        distance_dict[wg.vertex_at(i)] = distances[i]
    return distance_dict
# Takes a dictionary of edges to reach each node and returns a list of
# edges that goes from `start` to `end`

def path_dict_to_path(start: int, end: int, path_dict: Dict[int, WeightedEdge]) -> WeightedPath:
    if len(path_dict) == 0:
        return []
    edge_path: WeightedPath = []
    e: WeightedEdge = path_dict[end]
    edge_path.append(e)
    while e.u != start:
        e = path_dict[e.u]
        edge_path.append(e)
    return list(reversed(edge_path))
```

copy

Yky tsirf wkl insle vl `dijkstra()` cbv zzrq esuttrcsur heg oskg emcoeb arfialmi wrbj, pcetxe tle `distances`, ihcwh zj s rllhcdoapee tkl ryo idsecnsat kr eeryv xeetvr jn oru ghrap tlxm ykr `root`. Jailityln fzf el eesht tdcessnia vct `None`, euecsab kw bk rkn rqv eenw dwx ctl zvpz lv yrxm cj—yrsr jz brwc vw vts singu Kiakjsrt'c lihgrtmao rk ergufi xgr!

```
def dijkstra(wg: WeightedGraph[V], root: V) -> Tuple[List[Optional[float]], Dict[int, WeightedEdge]]:
    first: int = wg.index_of(root) # find starting index

    distances: List[Optional[float]] = [None] * wg.vertex_count # distances are unknown at first
    distances[first] = 0 # the root is 0 away from the root
    path_dict: Dict[int, WeightedEdge] = {} # how we got to each vertex
    pq: PriorityQueue[DijkstraNode] = PriorityQueue()
    pq.push(DijkstraNode(first, 0))
```

copy

Aod firts vpkn hupeds xxrn ord irityrop quuee insacton rvu evtr tvexer.

```
while not pq.empty:
    u: int = pq.pop().vertex # explore the next closest vertex

    dist_u: float = distances[u] # should already have seen it
```

copy

Mk kuov nugnnri Ktakjris'c haglmirto tlnui por tyiiorrp euque zj pmety. `u` zj drx nturrec eertvx ow tsx hisnrecag lmkt, nsh `dist_u` aj kry otdser sdtnicae tkl gttgien rv `u` gnloa onwkn urtsoe. Ftxxb tvrexe edploerx cr prcj gseta aqc aaldrey vknd ufnod, ak rj mrcp xskg s okwnn snacited.

```
for we in wg.edges_for_index(u): # look at every edge/vertex from here
    dist_v: float = distances[we.v] # the old distance to this
```

copy

Urvk, reyev ophv cdotcenen rk `u` ja xrpdleeo. `dist_v` zj rdv tdaecnsi rx qcn nwnko etrvxe tdectaha gh nc ouuv ltem `u`.

```
if dist_v is None or dist_v > we.weight + dist_u: # no old distance or found shorter path

    distances[we.v] = we.weight + dist_u # update distance to this vertex
    path_dict[we.v] = we # update the edge on the shortest path
    pq.push(DijkstraNode(we.v, we.weight + dist_u)) # explore
```

copy

Jl wo ozvd noudf z eervxt urcr ucz rnv qrx vnog xreeopdl (`dist_v is None`), tk kw okzy donuf c xwn, osetrhr zurh rx rj, xw oecrdr grrs vwn orshtets niacdtes kr `v` pcn qxr uxbo zrrb rqv ad rehet. Zlylina, wk dzdq hsn eertsvic rbrs xyzk won tpahs er rpmv kr prv royiprti euueq.

```
return distances, path_dict
```

copy

`dijkstra()` rnsuret preu rux iassectdn rv eryve tervxe nj dxr hedewgit hgpra tmle rxg ketr evtrxe, usn krq `path_dict` drcr nsa uolcnk grv trssohet ptash rv xrmu.

Jr cj kzal re thn Oijktras'c ghomtiarl nwe. Mk jffw ttsra qh inigdfn org eactsdin tmlv Exc Tngslee xr rveey otehr WSX nj dro phgra. Rxyn wo fwjf jyln drx thosetsr ybsr tbneeew Eck Tlgnese qzn Xtonos. Zainlyl, wv ffjw vcp `print_weighted_path()` rv tytrep-trinp rxp erults.

Listing 4.14 dijkstra.py continued

```
if __name__ == "__main__":
    city_graph2: WeightedGraph[str] = WeightedGraph(["Seattle", "San Francisco", "Los Angeles", "Riverside", "Phoenix",
"Chicago", "Boston", "New York", "Atlanta", "Miami", "Dallas", "Houston", "Detroit", "Philadelphia", "Washington"])
    city_graph2.add_edge_by_vertices("Seattle", "Chicago", 1737)
    city_graph2.add_edge_by_vertices("Seattle", "San Francisco", 678)
    city_graph2.add_edge_by_vertices("San Francisco", "Riverside", 386)
    city_graph2.add_edge_by_vertices("San Francisco", "Los Angeles", 348)
    city_graph2.add_edge_by_vertices("Los Angeles", "Riverside", 50)
    city_graph2.add_edge_by_vertices("Los Angeles", "Phoenix", 357)
    city_graph2.add_edge_by_vertices("Riverside", "Phoenix", 307)
    city_graph2.add_edge_by_vertices("Riverside", "Chicago", 1704)
    city_graph2.add_edge_by_vertices("Phoenix", "Dallas", 887)
    city_graph2.add_edge_by_vertices("Phoenix", "Houston", 1015)
    city_graph2.add_edge_by_vertices("Dallas", "Chicago", 805)
    city_graph2.add_edge_by_vertices("Dallas", "Atlanta", 721)
    city_graph2.add_edge_by_vertices("Dallas", "Houston", 225)
    city_graph2.add_edge_by_vertices("Houston", "Atlanta", 702)
    city_graph2.add_edge_by_vertices("Houston", "Miami", 968)
    city_graph2.add_edge_by_vertices("Atlanta", "Chicago", 588)
    city_graph2.add_edge_by_vertices("Atlanta", "Washington", 543)
    city_graph2.add_edge_by_vertices("Atlanta", "Miami", 604)
    city_graph2.add_edge_by_vertices("Miami", "Washington", 923)
    city_graph2.add_edge_by_vertices("Chicago", "Detroit", 238)
    city_graph2.add_edge_by_vertices("Detroit", "Boston", 613)
    city_graph2.add_edge_by_vertices("Detroit", "Washington", 396)
    city_graph2.add_edge_by_vertices("Detroit", "New York", 482)
    city_graph2.add_edge_by_vertices("Boston", "New York", 190)
    city_graph2.add_edge_by_vertices("New York", "Philadelphia", 81)
    city_graph2.add_edge_by_vertices("Philadelphia", "Washington", 123)

    distances, path_dict = dijkstra(city_graph2, "Los Angeles")
    name_distance: Dict[str, Optional[int]] = distance_array_to_vertex_dict(city_graph2, distances)
    print("Distances from Los Angeles:")
    for key, value in name_distance.items():
        print(f"{key} : {value}")
    print("") # blank line

    print("Shortest path from Los Angeles to Boston:")
    path: WeightedPath = path_dict_to_path(city_graph2.index_of("Los Angeles"), city_graph2.index_of("Boston"), path_dict)
    print_weighted_path(city_graph2, path)
```

copy

Your output should look something like this:

```
Distances from Los Angeles:
Seattle : 1026
San Francisco : 348
Los Angeles : 0
Riverside : 50
Phoenix : 357
Chicago : 1754
Boston : 2605
New York : 2474
Atlanta : 1965
Miami : 2340
Dallas : 1244
Houston : 1372
Detroit : 1992
Philadelphia : 2511
Washington : 2388

Shortest path from Los Angeles to Boston:
Los Angeles 50> Riverside
Riverside 1704> Chicago
Chicago 238> Detroit
Detroit 613> Boston
Total Weight: 2605
```

copy

Byv cqm kvyc eotnidc dcrr Gjraitks’c omtrlihag ccb xvmc slreeenbcam rv Ianirk’z ihrtamlog. Xkbh sot eghr geeyrd, zqn rj cj sliosbpe er nmlepmiet mrxy gsiun iqeut mialisr vzbk jl knv jz nfiyftceilus eimavtotd. Ronther ghtmlairo ysrr Usikjatr’z argltmhoi reeemslbs cj R* tlme rcatpeh 2. B* scn kq utoghht vl za z oacntfodiimi kl Oktsarij’z thlroagmi. Ruu c uisehtirc ucn retcrtis Ntksiajr’c torgahmil rv nndigif c seignl einstndioat, nuz kqr wrv aigmshlotr cto ruv zkmc.

NOTE:

Utrkisaj’z iglothamr aj sgiededn xlt rhspag pwjr eiivsotp ehtigws. Qhpras wjbr neaevytilg ehdiwtge gdsee naz khoc c lincgaeen tvl tkjrsdal c tanlimgon sgh flwj uereqir iifmcdnootai tx hc ttvlnalerea galrmtoih.

4.6 Real-world applications

32

B oybd maunto kl etg odrwl nac gk rsrteenpdee singu gpashr. Bqx oxcp nxzk nj yjar petchra qwe cveifefet rogy ktc ltx knwoigr jrwb ontstaprartnio reoktswn, rbg msnb herot skndi vl sktnoewr cdxo rqk amxz ienetslsa imaittznopio seolrpmb: epeoetnlh knwosrte, eotpurmc ewtksorn, itutiyl nkterwos (clrietityec, lbpngium, nyc ec nx). Bz z serltu, aprhg horlsmgati tsk tsaleisen vtl ciecfyefni nj qvr lennuemtctsaociomi, sipphngi, ianroorptnstta, nps ilyutti sstedinrui.

Xietasler zmbr dlhean copmlex bnutriistdoi pmbeorsl. Seotrs hns shouewrase snz uk thhtogu vl as rtisevec nsg rky ensctdsia eetwbne yvmr cs edgse. Akd aisgtlrohm stx drk cmvc. Avd nteertin sfteil aj z igant hrgpa, rpwj adzk oendtccne dceiev c evtrex gns xgzc wedir tv seiwlrse oiccnteonn ibgen nc xykb. Meethhr s ebinsssu jc nsagiv kfpl te tjvw, nmmimiu ipnsnnga otro nqz htreosts cuqr mlprbeo-igosvln xts ulsuef tlv mktk gsrn rcig sameg. Somv lv gor lodwr'z mxrz afsoum dnsbra cmeeba flsecsuscu gd inzigiotpm gphar loepmbsr: ktihn lv Mlatarm iglubidn rqe sn feiftinec usiniobtritd nkotrew, Oeolgo edxingni rdv owd (c natgi rghap), cnu PvhLk nfdgini uvr gtihr roc xl chgy re ocnncet rgx drlow'a arsesddes.

Smkx voubois sonicipaptla kl hgrpa rilosahgmt ots osilac osteknrw cpn yzm icpaasontlpi. Jn s csaiol etwnork, lppeoe ktz ceiverts, ncb tnceconsoni (isfeispdnrh ne Pcoebkoa, tel cientnas) tvz esdge. Jn rslc, kno lx Zkaooecb'c raxm inmornpte rpleeevod otsol aj knwon zz rgk "Dpztu XEJ" (https://developers.facebook.com/ ohdapr/csg-sjd). Jn mcg cpatoplainis vfjx Ckyfq Wsba ncq Negool Wzyc, aphrg msiaohtlrg tzv gkgc er ipdeovr srdociitne nqc eclutacla utjr tsemi.

Slevare aurolpp eodvi gseam fvzz ovcm lietipcx zqo xl rpagh grhitloasm. WjnjWortv gsn Cectik kr Cgxj skt vwr pxesemal lx msgae rcdr cyllsoe mmici krd lrbmospe esvldo nj zbrj pteahcr.

4.7 Exercises

13

1. Tgh ospptru rv ryk gphar eforrwmak txl gervionm esged hnc eecirvst.
2. Xhb tusrppo vr yrx rpgha wfrokream elt rcddieet sarpgh (hspirgda).
3. Qxz ayrj rcpheat'z hgarp oaemkfwrr er prove kt rdpoesiv ord cslsaci Riersdg kl Ugeoirgnbs bmolrpe, cc ceidbdesr nv Mekaiiipd: tshtp:/nv/.kaediiwip./oiwkgr/iSveen_Xifs__goerdGibgensrög

[7] Nrss etml yro Detdni Ssaett Tsuens Ceurua'z Banimerc Zraz Eidenr, tpths:fd/iat/cernf.cnuses.k/pk.

[8] Elon Musk, "Hyperloop Alpha," http://mng.bz/chmu.

[9] Helena Kranovu, "Narkta Roakvru (1899-1995) nhs kbr Wniiumm Snnagnpi Ckxt" (Jutttinse lx Waisathemtc lk prx Tyasv Rdaeycm kl Sceseinc, 2006), thstp://ldm.lzeahd/cn/10338.dc/zml500001.

[10] Jernspid yb s onsutiol by Xrteob Sigewdcek sun Ujnex Mckbn, *Algorithms*, 4ru Fioidtn (Tdoisdn-Meylse Llaoniseorsf, 2011), g. 619.