4 Graph problems

109

Y *graph* jc nc rabactst iamtmhectlaa otsructcn rrsq jc cvph klt oidnlgem c fskt-wlrdo orelmbp ub idvdinig ruv leombrp nrej z roc lv doncctnee osned. Mk zffa zkuz kl yxr ndeso c *vertex* snh qzkz xl oyr ocecnninost cn *edge*. Pvt ctensani, z ybuwas yms nac go huohtgt kl cs s garhp eptsirrngene z tpoartorintsna ekowrtn. Zcua xl rqk rhvz ereserpstn s onsttai, cbn zaou lv rpo ilens tpserenrse s ueort bentwee ewr sintatos. Jn ahprg yogimenrtol, wo would zffs kru otistans "tricesve" znh rux estrou "sdege."

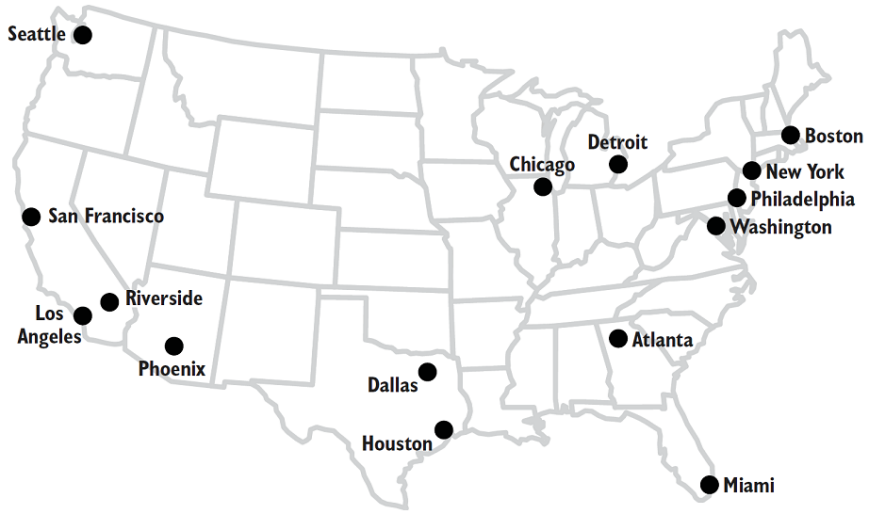
Mhd jc rpzj lsuuef? Ore fnbk ye pshgra foyy bz tacatbsylr thkni tabuo s leprbmo, vruq zcfe xfr pc papyl reevasl fwfo-ooentddrsu pcn pmaftrrone haescr pnz maizoniitotp techuenisq. Ext nasietnc, nj grk ayubws eaexmlp, sppeuos ow snwr rk wnxv rdx tehtrsos reuto mvtl onk ontaist rk ohenrta. Dt, eppusos wx atnedw re wxnv vrg iumnimm tomanu kl tkcra dneeed vr cnctneo cff le dxr tnsaosti. Qcptb rlimoahgts rrdz dvd fwjf raeln nj jzrp chatepr zcn esovl bvdr el hesot msrpleob. Lurethr, aghpr iglhrsaotm san kq idepapl er ndz gvjn lv twrekon loemprb—nrv raig ntitnoaosrrpat roktwnse. Xpenj vl tropcmue esrtwkon, tsidbirintou snwokrte, spn liiyutt snktoewr. Sarceh gcn iimntoiozatp lrbmopse raossc ffc lx htees aesscp anc dv sovedl isgun hagpr oahlgsritm.

4.1 A map as a graph

43

Jn bjcr hratecp, wk new'r twke yjrw z raghp xl wsyabu tnsioats, dqr iensatd cesiti kl xdr Qnedit Saetst qns oeitapntl suotre wtenbee mrvb. Veriug 4.1 aj z smg le gro ennlociattn Oetidn Ssetat bzn obr eietnff lgsater olnmroetptia lsiattactsi aasre (WSXa) jn rod yruncto, sc tamdseiet pp ykr G.S. Tnuess Teauur.[7]

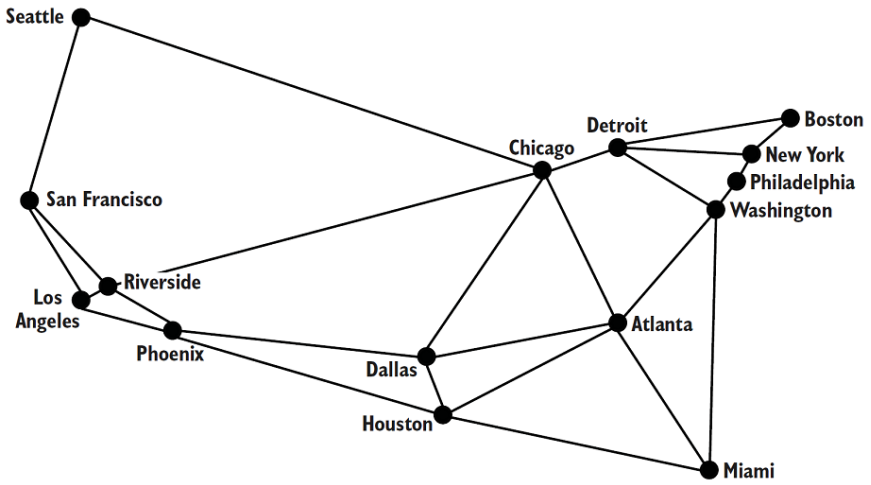
Figure 4.1 A map of the 15 largest MSAs in the United States



Vmasou eepertrrnuen Pnfx Wecd sbc esugdtesg lduigbni z wnx bujq-edesp oortratpitsnan etnrkow mpcoosed kl uepsclas nvtlgraei jn epidszesrur uesbt. Tgrccindo re Whao, vrp lcaspesu dowlu relatv rc 700 mesil toh hxtd znb vp uabetsil tle rsxa-fteievfce tirpotaotnasrn etenwbe icites ofzc rsnq 900 seiml taarp.[8] Ho sallc arjq won rosatpotrnaitn seymts rgk "Hoplorpye." Jn cjrg aehtcpr wk fjfw elpeorx slsacci grhpa opsmlber jn krp oxtncet vl uniigldb rey jrcu taootrnsnriapt okwntre.

Weba iyailtlin psepdoor rku Hroppeyol psjk lxt nnieocgtcn Ecx Cnesleg gnz Ssn Zaoccnsri. Jl nev twkk rv uldib c onntlaia Hplyporeo onrktwe, jr wudlo ezkm essne er xp ae ebtwene Cmircae'z eratgsl taimloepntor saare. Jn efiugr 4.2 ruo ttesa onusltie mtvl ufirge 4.1 ozt mevedor. Jn idaintdo, qavz lv rvb WSBc aj eeccotnnd drjw kmkc kl rja sbeirohng. Ak svem dro harpg c lltite tmkx nestnreitig, teohs hsoeginrb kts nrk yaslaw bxr WSX'a sectols bgehionrs.

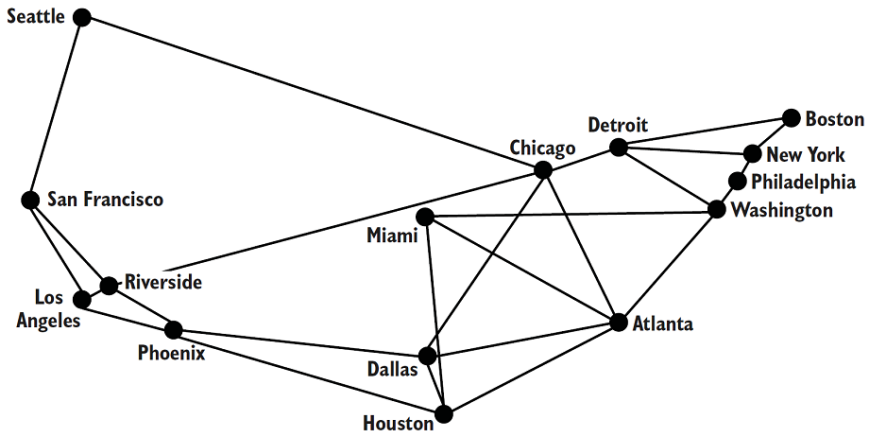
Figure 4.2 A graph with the vertices representing the 15 largest MSAs in the United States and the edges representing potential Hyperloop routes between them



Vrgeiu 4.2 aj z raphg wdjr ervsceit ginenestrepr xbr 15 gtalrse WSBa jn bor Dteidn Seastt nzh egdse intreeesrnpg aoplnteti Hopeylrpo utores ebewetn tisiec. Rky uosert wtvv shocen lte lavrietutlis peuosspr. Teaiyrtln, orteh nlttopeai ersuot udloc do tgcr le s xwn Hoprlopey wrkntoe.

Yqcj rsattcba reenrapnsittoe le c cxft-dlwro oeblrpm ggshihltih vqr eprow el hrgpsa. Gvw brcr kw ocbo nz atabnioscrt rv wvet rywj, xw szn oergni kqr hrgaoegyp le krp Kdient Ssteat pns tncconeater nx tngknhii uoabt qrk tepoantli Hyroleppo rtwkone mylsip jn rkq otcnext lk ncgnntceoi tecsii. Jn czrl, sa vndf za kw voyk xyr deges kdr zmxc, kw nca hkitn uabto ukr rlmobpe wyrj c fdteirefn kgnoloi arhpg. Jn ergfiu 4.3, ogr antoiocl el Wjcmj zus emvod. Xxy rpahg jn furgie 4.3, begni cn srbaatct rtsnptineaoree, snz lslit aesddrs oyr mzsk umanfldtean muaponottilac mprblseo zc dkr gahpr nj eifrgu 4.2, oxnk lj Wmjsj cj rvn weher vw dwolu pcxete jr. Crp xtl qet iaytsn, wk jffw tikcs jgrw xrp rtaoipsrenntee jn guerif 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

149

Lyhton ssn gk aremdprmgo nj bsnm eiffrtnde lsytes. Hevwoer, rc jrz aerth, Znhtyo jc zn jotbce-itedorne riomamggrnp aeugnlag. Jn japr cinoset wk jwff nedief wrv ndreffeit spety le srhagp—egiweunhtd bnc idetegwh. Migedthe hpsagr, whcih wv fwfj sucssdi rltae nj prv heractp, stscaaoei c iwthge (cuxt nmreub, aydc cc c ngelht nj pro zszv lk ykt exalmpe) wrjq bskz uvky. Mv fjwf oxmc gkz lk rvg rincnetehai ledmo, lnaudfanemt kr Zhtnyo'z ocebtj-odterien cssal shrhciieera, er rne leacdupti dte rtfefo. Bod ietdwheg scslsae jn xtg bcrs eldom ffwj yv bssasuecsl el hiret enedgiutwh erotsnapucrt. Bcyj jffw llowa kmrg re itrinhe yzqm vl rtihe nyatnfictoilu, rwbj salml wsetka tlv qrwz mksae c hegtwide ahgpr tdstcnii tmkl nc eiutngehwd hparg.

Mx nwrs djrc apghr mkwerrfoa xr xu sc lebfelxi az bsoplesi, ec rusr rj nzs eeetrrnps za bmcn itefrfend boerlpsm zs soeplbsi. Xe hevaiec jzrq sfbx, wk jfwf vga gcienrse rk btratasc zucw rdx orhy le rou iesrcvet. Ltkep eretxv fwjf uaelylimtt xq gsdaenis nz negrite enxid, rhg jr fjwf po soedtr ca rbo tdkz-nefided ieecrng rqou.

Vor'a sattr wtvx nx ruk wafkmrero dg dnfneiig bvr `Edge` aclss, hcwhi jz rdx tsipelms rcmainyeh jn tyv hagpr morrfeawk.

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

Cn `Edge` jc efidnde cc c neoninctco ebewetn wer rcevstei, bzxs lv ihhwc zj dreeretnesp bq zn gtieenr ienxd. Cp nocneivnot, `u` cj vqzu rx rrfee vr rku fsitr etverx, qsn `v` aj kqgc xr teseprenr rdv csdeno etrxev. Xky scn efzz nhkit le `u` cc “etlm” sun `v` ac “kr.” Jn ujra htarepc, wo tkz nefg iwrgkon jwpr nideedcrtu asphgr (paghsr jwgr edgse prsr llawo atvlre jn rxpg nresdtcoii), rpb nj *directed graphs*, zsvf knonw cz *digraphs*, geeds azn zxfc gk vnk-wsd. Aku `reversed()` omhtde ja mtnae er ruretn sn `Edge` rrgc tvlesra nj rdo ptpieoos coiitdner kl rvu khuv jr ja ppaeldi kr.

NOTE

Yxg `Edge` sslac ahcx z wnx fuereta jn Fnothy 3.7: ecssadlstaa. B slcsa maedkr rwgj xrq `@dataclass` rrdcteaoo assve vzem imutde dy atumycloiatla ngtairce zn `__init__()` temhdo usrr itssnenitaat ectinans visblaare ltv zqn lvaeasirb ecadledr gwjr ubor innstnaatoo jn rvq scals’z pbxh. Kcsataasesl znz vzzf lmtclaayotiau eterca tehor lcsipae esthomd ktl c aslsc. Apx psecila oemthds srrp tkc yuailmctaoalt acerdte aj fableucgroni jez yrk dtoeacror. Sxv oqr Ztonyh ctnmtdaouoine nk ssdaecalast tlx itesdal (https://docs.python.org/3/library/dataclasses.html). Jn thsor, z dtscaasal jz z swh el vasngi vrlesusoe ckme nigpyt.

Cuk `Graph` aclss aj utoab rob eietasnls kkft kl s pghra: tngsoaiicas stercevi wjbr eedsg. Xjnzy, wk wrns re frv bxr ctlaau epsyt lv qrx rectsevi vy reeawhvt rxy ktcd lk xbr mekrfrawo ierdess. Xaju rfoc roq wmefrrako yo qzvd lte c juwv egarn lk porlbsem owihtut enignde vr mcxx taneimidreet zsrp crtesrtuus rurz uogf ethngivrye tgthoere. Vtv eempxal, nj s rghap ejfx gxr nxo xlt Hoerpypol usteor, xw hgmti efdein krq qkur lv sevicetr rx pk `str`, sceubea wx odlwu pco tirsngs vvjf “Gwk Ttxv” pnz “Fec Yneegls” az urk rceevtis. Prx’z ibnge ruv `Graph` lsasc.

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

Avg `_vertices` afjr zj rop rthea lv s `Graph`. Zycs etxver wffj kp ordset jn rvp rafj, pru ow fwjf trlea efrre vr brmx hb ithre tgnreei xeidn jn brk rjfa. Bvu xetrve etsifl qsm qo c lemcoxp cryc orgh, gru ajr inxde wjff syaalw kg zn `int`, hhicw aj hccx kr twee jrbw. Nn tornhea velel, bp tnpgtui rcjd nxied betwnee garph rastmglhio snp drx `_vertices` arayr, jr slloaw ha rk ckdk wrk eicetvrs sprr toz ulqae nj dxr cxzm ghrpa (iagemin c apgrh brjw s conytru’c eiicst cs eetrvisc, hrewe ykr yutrnco usa otmv pncr exn psrj eadnm “Spgdnlrfiei”). Pnxo guhtho hrux ktz rxb czmk, hxbr jffw dxos trffenedi nretegi dexsnei.

Cbtko tzv psnm chsw xr meetinmlp z hapgr rcuc tsuecurtr, rqg qro kwr crem ocmnmo xts rx kcp z *vertex matrix* kt *adjacency lists*. Jn c ervxte aritxm, opas fzfx lk xdr xatmir seprrsntee uro osniteritenc xl rxw vcetiser nj xqr phagr, cbn gor ueval kl rrzd xaff sdicnitae rgk oenntoiccn (vt zofa hfeetor) eebwten vmrb. Qtd prhga sruz tstcuuerr hzzx djcneaayc slsti. Jn cgjr gprha irrteteponnesa, verey vtrexe uzs s jfzr vl iveertcs rzrp rj ja tnconedce rk. Ktg pseiifcc rrsianoptetene ckya c rcfj el tslsi kl edesg, cv lvt ryvee vetexr heter jc s jrfa el edesg sjx whcih dor exvrte cj neontccde xr htroe eertcvsi. `_edges` aj jaru rfjc lv isstl.

Rkb crtk lv yrv `Graph` scals ja wnv pensetred nj arj ereyntit. Tep wffj eoctni prv xzq lv shtor, ytsolm xkn-nxjf tohedms, jywr serbveo gns learc hoemtd sneam. Yjag hdolsu omxs xry zrto le qro sacls lagyelr lfxc-teyapaolrxn, bqr orsth motsecnm ctk ceundlid, xz crqr eehtr zj vn mtxx tel intpieritanetmrso.

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

Erk'a rdva syae txl z omenmt gnz soncedir wdd qrjz lscsa ccy xrw soersnvi lv mvar lv jzr edhsmot. Mo enow ltmk odr ascls eifonindti urrs kgr rjaf _ `vertices` ja z rfaj le nestleem el rbvd `V` , hhciw nzz gv gns Fnyhto scals. Sx, wx qooz criteves xl burv `V` urcr tso oderst jn rvb _ `vertices` jrcf. Rhr jl wo nwrs rx eiverrte xt umeaitplna modr altre, kw ynxo vr nkwo ewrhe oqrq tvs dosrte nj prsr jrfz. Honzk, yerve exretv zzd sn ixdne nj prx aayrr (sn eintegr) easiatosdc ujrw rj. Jl vw neu'r vxwn s xvteer'z xeind, ow ynkx er xfko jr gu gy erginsahc outghrh _ `vertices` . Rrcg jz buw treeh xzt vrw servnsio kl revey tmohed. Dnk ersaopet kn `int` nidseci, sbn xvn poteasre nk `V` fsielt. Yvd hmdetso zgrr reaetpo nv `V` fvvv hp ryk eaernlvt isecnid ngc ffzs vru dinxe-dsbea cnunoitf. Xerehroef, vggr csn hx esdcendiro noeenncveic dmhoset.

Wzvr kl xbr sunonftic vst ialfry xlzf-nleyaapxotr, rud `neighbors_for_index()` reesedvs z tlietl pnckunaig. Jr rentrus pro *neighbors* el c xertve. C xvetre'a eogsbrnhi txc ffz lv uro htore ivetcsre rrdc otz tlrdiecy ceectdnon rv jr hh cn pvbk. Ptx epmexla, nj grifeu 4.2, Dwx Retv nyc Mtaiohgnsn xts rvq fegn bgoiehnrs xl Zdahealhilip. Mv lpjn drk bshrneiog tlk z xeetrv db ilnokgo sr brx aukn (rbx `v` a) kl cff lv kqr dgese ggion xrb mvlt jr.

```
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]` cj xgr aynedcacj jafr, grx rjcf el edseg uhortgh ihhcw kdr vrtxee nj iqstuneo ja neteccnod vr orhte eectsivr. Jn gxr crjf omnhoiepenscr speasd rk vrd `map()` ffas, `e` seerreptns okn ptaarulcir vvdq, bsn `e.v` tseernpesr urk xnide xl grv ihbrogne sqrr bro bboo ja coendecnt vr. `map()` fjwf tneurr fsf le qvr ciervets (cz epdpsoo vr drai hteir ndsicei), cusaebe `map()` eppalis kur `vertex_at()` ometdh nv yreve `e.v` .

Rhontre mtoiptarn nghit rx enrx ja krd swq `add_edge()` wkrso. `add_edge()` ftirs ccpp cn qgoo re gor adnjcycea jfcr el kgr "tvml" xtevre (`u`), ncb bnrv payz z esdrveer snvoier lx rxq xdxp rx rux jaceycand crjf kl xur "rv" eevxrt (`v`). Xop soecnd xrua ja eynrscase ecsbeau jzgr grhap cj neictddreu. Mo znrw evrey kqkh re xq dddea nj dxrp idteniocsr—srpr senma crry `u` wffj kq s eribognh vl `v` jn bro mxzc zbw rsur `v` aj c borngihe xl `u` . Cey zcn nitkh lk ns tdeiucrden rapgh az egnbi "bailrneiodict" lj jr epshl yvh reermbme rprc jr asmne zhn wrv netdoncec rviteesc nzz oh srvteedar jn ehtrei tidonceri.

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

copy

Tz zwz detnoeimn rrieeal, wk ztx xnfg enaigld wjpr ndtrduicee psrgah nj braj tahprec. Redyon nbgei cediunedtr xt deditcre, parsgh nsa axzf xy *unweighted* vt *weighted*. Y gideethw ragph jz xkn rpzr bca mcxk racmoaepbl vuela, saulyul cemriun, asidecaots pjrw bsvz le rjc gsede. Mk coudl htikn vl grk ighwset jn het pltetanio Hrepopoly trewnok cz iebgn xur edtcisnsa tbeeewn pvr satoitns. Vet nxw, hhogtu, kw wjff qxzf wrbj nc nidwuegteh veroisn el xbr pragh. Bn hidueewtgn oxpb jz mispyl s eonconncit beteewn rvw ivcteers, neceh gro `Edge` lascs zj ueegtdhniw, cyn pvr `Graph` sclsa aj wieuhtgned. Coethrn uzw le ntupgti jr aj rzgr jn cn ethiugewnd phagr wv wvnk hwhic sveticer zkt neoctdnec, hrseawe nj c twedeghi grpah ow nkwx hhciw tircesev tzv dctonnece nzh wx owkn gtohenmis atbuo thseo ncseonnitco.

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

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
if __name__ == "__main__":

    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:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

```
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

The Hyperloop is so fast that, for optimizing travel time from one station to another, it probably matters less how long the distances are between the stations and more how many hops it takes (how many stations need to be visited) to get from one station to another. Each station may involve a layover, so just like with flights, the fewer stops the better.

In graph theory, a set of edges that connects two vertices is known as a *path*. In other words, a path is a way of getting from one vertex to another vertex. In the context of the Hyperloop network, a set of tubes (edges) represents the path from one city (vertex) to another (vertex). Finding optimal paths between vertices is one of the most common problems that graphs are used for.

Informally, we can also think of a list of vertices sequentially connected to one another by edges as a path. This description is really just another side of the same coin. It is like taking a list of edges and just figuring out which vertices they connect and keeping that list of vertices and throwing away the edges. In the brief example, we will find such a list of vertices that

connects two cities on our Hyperloop

4.3.1 Revisiting breadth-first search (BFS)

In an unweighted graph, finding the shortest path means finding the path that has the fewest edges between the starting vertex and the destination vertex. To build out the Hyperloop network, it might make sense to first connect the furthest cities on the highly populated seaboards. That raises the question, "what is the shortest path between Boston and Miami?"

CAUTION

This section assumes you have read chapter 2. Before continuing, ensure you are comfortable with the material on breadth-first search in chapter 2.

Luckily, we already know an algorithm for finding shortest paths, and we can reuse it to answer this question. Breadth-first search, introduced in chapter 2, is just as viable for graphs as it is for mazes. In fact, the mazes we worked with in chapter 2 really are graphs. The vertices are the locations in the maze, and the edges are the moves that can be made from one location to another. In an unweighted graph, a breadth-first search will find the shortest path between any two vertices.

We can reuse the breadth-first search implementation from chapter 2 and use it to work with `Graph`. In fact, we can reuse it completely unchanged. This is the power of writing code generically!

Recall that `bfs()` in chapter 2 requires three parameters: an initial state, a `Callable` (read function-like object) for testing for a goal, and a `Callable` that finds the successor states for a given state. The initial state will be the vertex represented by the string "Boston." The goal test will be a lambda that checks if a vertex is equivalent to "Miami." Finally, successor vertices can be generated by the Graph method `neighbors_for_vertex()`.

With this plan in mind, we can add code to the end of the main section of `graph.py` to find the shortest route between Boston and Miami on `city_graph`.

CAUTION

In Listing 4.5, `bfs`, `Node`, and `node_to_path` are imported from the `generic_search` module in the `Chapter2` package. To do this, the parent directory of `graph.py` is added to Python's search path (`'..'`). This works because the code structure for the book's repository has each chapter in its own directory, so our directory structure includes roughly Book->Chapter2->generic_search.py and Book->Chapter4->graph.py. If your directory structure is significantly different, you will need to find a way to add `generic_search.py` to your path and possibly change the `import` statement. In a worst-case scenario, you can just copy `generic_search.py` to the same directory that you have `graph.py` within and change the import statement to `from generic_search import bfs, Node, node_to_path`.

Listing 4.5 graph.py continued

```
1
2
3
4
5
6
7
8
9
10
11
12
13
```

```
import sys
sys.path.insert(0, '..')

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

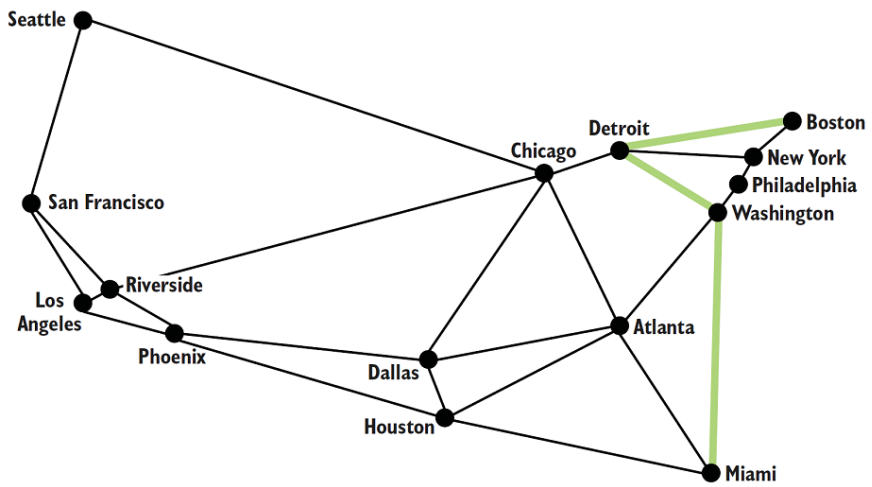
The output should look something like this:

```
1
2

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

copy

Boston to Detroit to Washington to Miami, composed of three edges, is the shortest route between Boston and Miami in terms of number of edges. Figure 4.4 highlights this route.

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



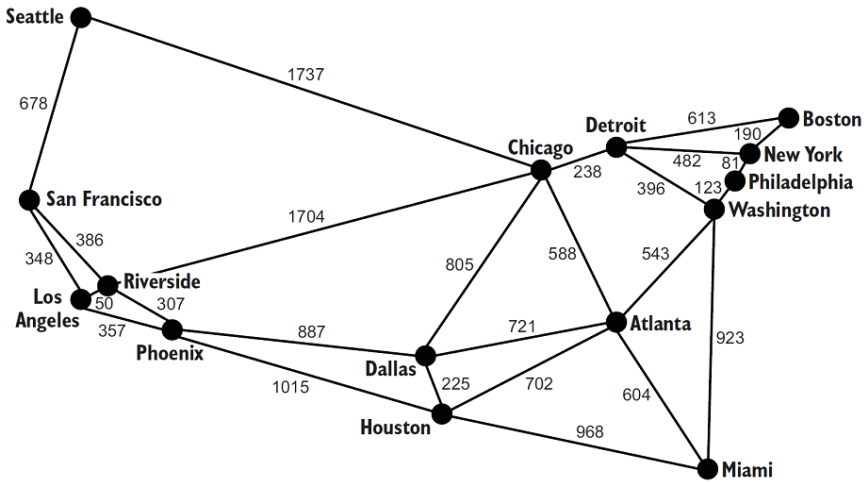
4.4 Minimizing the cost of building the network

Imagine we want to connect all 15 of the largest MSAs to the Hyperloop network. Our goal is to minimize the cost of rolling out the network, so that means using a minimum of track. The question is then, "how can we connect all of the MSAs using the minimum amount of track?"

4.4.1 Workings with weights

To understand the amount of track that a particular edge may require, we need to know the distance that the edge represents. This is an opportunity to re-introduce the concept of weights. In the Hyperloop network, the weight of an edge is the distance between the two MSAs that it connects. Figure 4.5 is the same as figure 4.2, except it has a weight added to each edge, representing the distance in miles between the two vertices that the edge connects.

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



To handle weights, we will need a subclass of `Edge` (`WeightedEdge`) and a subclass of `Graph` (`WeightedGraph`). Every `WeightedEdge` will have a `float` associated with it, representing its weight. Jarnik's algorithm, which we will cover shortly, requires the ability to compare one edge with another to determine the edge with the lowest weight. This is easy to do with numeric weights.

Listing 4.6 weighted_edge.py

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
```

```
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)


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

copy

The implementation of `WeightedEdge` is not immensely different from the implementation of `Edge`. It only differs in the addition of a new `weight` property and the implementation of the `<` operator via `__lt__()`, so that two `WeightedEdge`s are comparable. The `<` operator is only interested in looking at weights (as opposed to including the inherited properties `u` and `v`), because Jarnik's algorithm is interested in finding the smallest edge by weight.

A `WeightedGraph` inherits much of its functionality from `Graph`. Other than that: It has init methods, it has convenience methods for adding `WeightedEdge`s, and it implements its own version of `__str__()`. There is also a new method, `neighbors_for_index_with_weights()`, that returns not only each neighbor but also the weight of the edge that got to it. This method is useful for the new version of `__str__()`.

Listing 4.7 weighted_graph.py

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
```

```
from typing import TypeVar, Generic, List, Tuple
from graph import Graph
from weighted_edge import WeightedEdge
V = TypeVar('V')

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)

    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

It is now possible to actually define a weighted graph. The weighted graph we will work with is a representation of figure 4.5, called `city_graph2` .

Listing 4.8 weighted_graph.py continued

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29

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

Because `WeightedGraph` implements `__str__()`, we can pretty-print `city_graph2`. In the output, you will see both the vertices each vertex is connected to and the weight of those connections.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

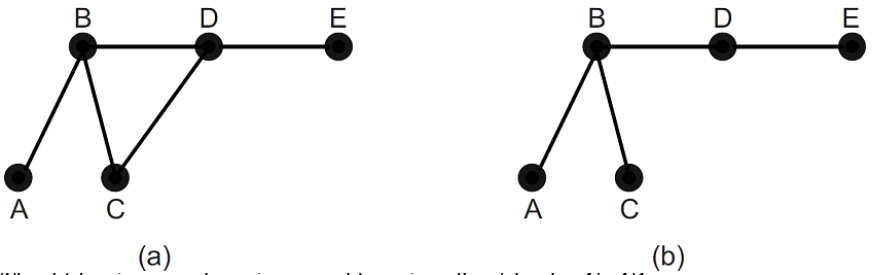
```
Seattle -> [('Chicago', 1737), ('San Francisco', 678)]
San Francisco -> [('Seattle', 678), ('Riverside', 386), ('Los Angeles', 348)]
Los Angeles -> [('San Francisco', 348), ('Riverside', 50), ('Phoenix', 357)]
Riverside -> [('San Francisco', 386), ('Los Angeles', 50), ('Phoenix', 307), ('Chicago', 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

A *tree* is a special kind of graph that has one, and only one, path between any two vertices. This implies that there are no *cycles* in a tree (which is sometimes called being *acyclic*). A cycle can be thought of as a loop: If it is possible to traverse a graph from a starting vertex, never repeat any edges, and get back to the same starting vertex, then it has a cycle. Any graph that is not a tree can become a tree by pruning edges. Figure 4.6 illustrates pruning an edge to turn a graph into a tree.

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.



A *connected* graph is a graph that has some way of getting from any vertex to any other vertex (all of the graphs we are looking at in this chapter are connected). A *spanning tree* is a tree that connects every vertex in a graph. A *minimum spanning tree* is a tree that connects every vertex in a weighted graph with the minimum total weight (compared to other spanning trees). For every weighted graph, it is possible to efficiently find its minimum spanning tree.

Whew, that was a lot of terminology! The point is that finding a minimum spanning tree is the same as finding a way to connect every vertex in a weighted graph with the minimum weight. This is an important and practical problem for anyone designing a network (transportation network, computer network, and so on)—how can every node in the network be connected for the minimum cost? That cost may be in terms of wire, track, road, or anything else. For instance, for a telephone network, another way of posing the problem is, "what is the minimum length of cable one needs to connect every phone?"

Revisiting priority queues

Priority queues were covered in chapter 2. We will need a priority queue for Jarnik's algorithm. You can import the `PriorityQueue` class from chapter 2's package (see the note immediately previous to Listing 4.5 for details), or you can copy the class into a new file to go with this chapter's package. For completeness, we recreate `PriorityQueue` from chapter 2 here, with specific import statements that assume it will be put in its own stand-alone file.

Listing 4.9 priority_queue.py

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

```
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

    def push(self, item: T) -> None:
        heappush(self._container, item)

    def pop(self) -> T:
        return heappop(self._container)

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

copy

Calculating the total weight of a weighted path

Before we develop a method for finding a minimum spanning tree, we will develop a function we can use to test the total weight of a solution. The solution to the minimum spanning tree problem will consist of a list of weighted edges that compose the tree. First, we define a `WeightedPath` as a list of `WeightedEdge`. Then, we define a function `total_weight()`, that takes a list of `WeightedPath` and finds the total weight that results from adding all of its edges' weights together.

Listing 4.10 mst.py

```
1
2
3
4
5
6
7
8
9
```

```
from typing import TypeVar, List, Optional
from weighted_graph import WeightedGraph
from weighted_edge import WeightedEdge
from priority_queue import PriorityQueue
V = TypeVar('V')

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

copy

Jarnik's algorithm

Jarnik's algorithm for finding a minimum spanning tree works by dividing a graph into two parts: the vertices in the still-being-assembled minimum spanning tree, and the vertices not yet in the minimum spanning tree. It takes the following steps:

1. Pick an arbitrary vertex to include in the minimum spanning tree.
2. Find the lowest-weight edge connecting the minimum spanning tree to the vertices not yet in the minimum spanning tree.
3. Add the vertex at the end of that minimum edge to the minimum spanning tree.
4. Repeat steps 2 and 3 until every vertex in the graph is in the minimum spanning tree.

Note

Jarnik's algorithm is commonly referred to as Prim's algorithm. Two Czech mathematicians, Otakar Borůvka and Vojtěch Jarník, interested in minimizing the cost of laying electric lines in the late 1920s, came up with algorithms to solve the problem of finding a minimum spanning tree. Their algorithms were "rediscovered" decades later by others.[9]

To run Jarnik's algorithm efficiently, a priority queue is used. Every time a new vertex is added to the minimum spanning tree, all of its outgoing edges that link to vertices outside the tree are added to the priority queue. The lowest-weight edge is always popped off the priority queue, and the algorithm keeps executing until the priority queue is empty. This ensures that the lowest-weight edges are always added to the tree first. Edges that connect to vertices already in the tree are ignored when they are popped.

The following code for `mst()` is the full implementation of Jarnik's algorithm,[10] along with a utility function for printing a `WeightedPath`.

Warning

Jarnik's algorithm will not necessarily work correctly in a graph with directed edges. It also will not work in a graph that is not connected.

Listing 4.11 mst.py continued

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
```

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

    result: WeightedPath = []
    pq: PriorityQueue[WeightedEdge] = PriorityQueue()
    visited: [bool] = [False] * wg.vertex_count

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

        for edge in wg.edges_for_index(index):
            if not visited[edge.v]:
                pq.push(edge)
    visit(start)

    while not pq.empty:
        edge = pq.pop()
        if visited[edge.v]:
            continue

        result.append(edge)
        visit(edge.v)
    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.

```
1

2

3

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

copy

The algorithm returns an optional `WeightedPath` representing the minimum spanning tree. It does not matter where the algorithm starts (assuming the graph is connected and undirected), so the default is set to vertex index 0. If it so happens that the `start` is invalid, `mst()` returns `None`.

```
1

2

3

4

result: WeightedPath = []

pq: PriorityQueue[WeightedEdge] = PriorityQueue()
visited: [bool] = [False] * wg.vertex_count
```

copy

`result` will ultimately hold the weighted path containing the minimum spanning tree. This is where we will add `WeightedEdge`s, as the lowest-weight edge is popped off and takes us to a new part of the graph. Jarnik's algorithm is considered a *greedy algorithm* because it always selects the lowest-weight edge. `pq` is where newly discovered edges are stored and the next-lowest-weight edge is popped. `visited` keeps track of vertex indices that we have already been to. This could also have been accomplished with a `Set`, similar to `explored` in `bfs()`.

```
1

2

3

4

5

6

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

    for edge in wg.edges_for_index(index):
        if not visited[edge.v]:
            pq.push(edge)
```

copy

`visit()` is an inner convenience function that marks a vertex as visited and adds all of its edges that connect to vertices not yet visited to `pq`. Note how easy the adjacency-list model makes finding edges belonging to a particular vertex.

```
1

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

copy

It does not matter which vertex is visited first, unless the graph is not connected. If the graph is not connected, but is instead made up of disconnected *components*, `mst()` will return a tree that spans the particular component that the starting vertex belongs to.

```
1
2
3
4
5
6
7
8
9
10
while not pq.empty:

    edge = pq.pop()
    if visited[edge.v]:
        continue

    result.append(edge)
    visit(edge.v)

return result
```

copy

While there are still edges on the priority queue, we pop them off and check if they lead to vertices not yet in the tree. Because the priority queue is ascending, it pops the lowest-weight edges first. This ensures that the result is indeed of minimum total weight. Any edge popped that does not lead to an unexplored vertex is ignored. Otherwise, because the edge is the lowest seen so far, it is added to the result set, and the new vertex it leads to is explored. When there are no edges left to explore, the result is returned.

Let's finally return to the problem of connecting all 15 of the largest MSAs in the United States by Hyperloop, using a minimum amount of track. The route that accomplishes this is simply the minimum spanning tree of `city_graph2`. Let's try running `mst()` on `city_graph2`.

Listing 4.12 mst.py continued

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
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

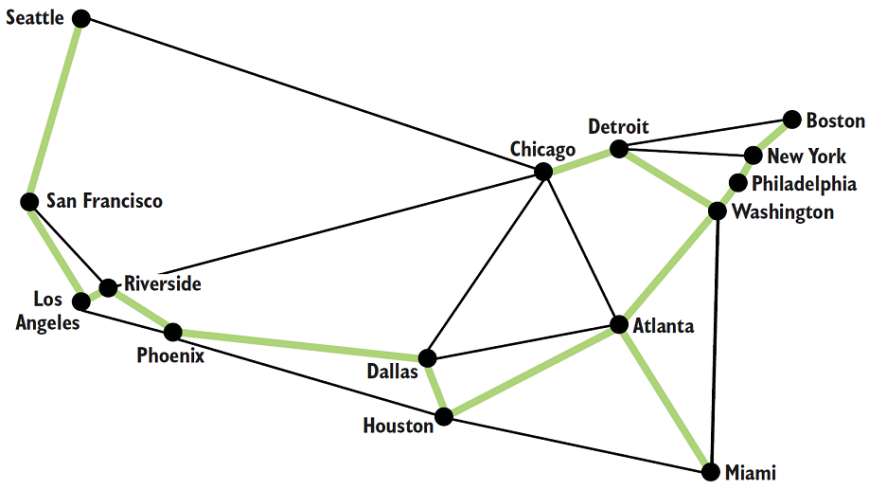
Thanks to the pretty-printing `printWeightedPath()` method, the minimum spanning tree is easy to read.

```
1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

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

In other words, this is the cumulatively shortest collection of edges that connects all of the MSAs in the weighted graph. The minimum length of track needed to connect all of them is 5372 miles. Figure 4.7 illustrates the minimum spanning tree.

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

As the Hyperloop network gets built, it is unlikely the builders will have the ambition to connect the whole country at once. Instead, it is likely the builders will want to minimize the cost to lay track between key cities. The cost to extend the network to particular cities will obviously depend on where the builders start.

Finding the cost to any city from some starting city is a version of the "single-source shortest path" problem. That problem asks, "what is the shortest path (in terms of total edge weight) from some vertex to every other vertex in a weighted graph?"

4.5.1 Dijkstra's algorithm

Dijkstra's algorithm solves the single-source shortest path problem. It is provided a starting vertex, and it returns the lowest-weight path to any other vertex on a weighted graph. It also returns the minimum total weight to every other vertex from the starting vertex. Dijkstra's algorithm starts at the single-source vertex, and then continually explores the closest vertices to the start vertex. For this reason, like Jarnik's algorithm, Dijkstra's algorithm is greedy. When Dijkstra's algorithm encounters a new vertex, it keeps track of how far it is from the start vertex, and updates this value if it ever finds a shorter path. It also keeps track of what edge got it to each vertex, like a breadth-first search.

Here are all of the algorithm's steps:

1. Add the start vertex to a priority queue.
2. Pop the closest vertex from the priority queue (at the beginning this is just the start vertex)—we'll call it the current vertex.
3. Look at all of the neighbors connected to the current vertex. If they have not previously been recorded, or the edge offers a new shortest path to them, then for each of them record its distance from the start, record the edge that produced this distance, and add the new vertex to the priority queue.
4. Repeat steps 2 and 3 until the priority queue is empty.
5. Return the shortest distance to every vertex from the start vertex and the path to get to each of them.

The code for Dijkstra's algorithm includes `DijkstraNode`, a simple data structure for keeping track of costs associated with each vertex explored so far and for comparing them. This is similar to the `Node` class in chapter 2. It also includes utility functions for converting the returned array of distances to something easier to use for looking up by vertex, and for calculating a shortest path to a particular destination vertex from the path dictionary returned by `dijkstra()`.

Without further ado, here is the code for Dijkstra's algorithm. We will go over it line by line after.

Listing 4.13 dijkstra.py

```
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
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')

@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)

    distances: List[Optional[float]] = [None] * wg.vertex_count
    distances[first] = 0
    path_dict: Dict[int, WeightedEdge] = {}
    pq: PriorityQueue[DijkstraNode] = PriorityQueue()
    pq.push(DijkstraNode(first, 0))
    while not pq.empty:
        u: int = pq.pop().vertex

        dist_u: float = distances[u]
        for we in wg.edges_for_index(u):
            dist_v: float = distances[we.v]
            if dist_v is None or dist_v > we.weight + dist_u:
                distances[we.v] = we.weight + dist_u
                path_dict[we.v] = we
                pq.push(DijkstraNode(we.v, we.weight + dist_u))

    return distances, path_dict


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



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

The first few lines of `dijkstra()` use data structures you have become familiar with, except for `distances`, which is a placeholder for the distances to every vertex in the graph from the `root`. Initially all of these distances are `None`, because we do not yet know how far each of them is—that is what we are using Dijkstra's algorithm to figure out!

```
1

2

3

4

5

6

7

8

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

    distances: List[Optional[float]] = [None] * wg.vertex_count
    distances[first] = 0
    path_dict: Dict[int, WeightedEdge] = {}
    pq: PriorityQueue[DijkstraNode] = PriorityQueue()
    pq.push(DijkstraNode(first, 0))
```

copy

The first node pushed onto the priority queue contains the root vertex.

```
1

2

3

4

while not pq.empty:
    u: int = pq.pop().vertex

    dist_u: float = distances[u]
```

copy

We keep running Dijkstra's algorithm until the priority queue is empty. `u` is the current vertex we are searching from, and `dist_u` is the stored distance for getting to `u` along known routes. Every vertex explored at this stage has already been found, so it must have a known distance.

```
1

2

for we in wg.edges_for_index(u):
    dist_v: float = distances[we.v]
```

copy

Next, every edge connected to `u` is explored. `dist_v` is the distance to any known vertex attached by an edge from `u`.

```
1

2

3

4

5

if dist_v is None or dist_v > we.weight + dist_u:

    distances[we.v] = we.weight + dist_u
    path_dict[we.v] = we
    pq.push(DijkstraNode(we.v, we.weight + dist_u))
```

copy

If we have found a vertex that has not yet been explored (`dist_v is None`), or we have found a new, shorter path to it, we record that new shortest distance to `v` and the edge that got us there. Finally, we push any vertices that have new paths to them to the priority queue.

```
1

return distances, path_dict
```

copy

`dijkstra()` returns both the distances to every vertex in the weighted graph from the root vertex, and the `path_dict` that can unlock the shortest paths to them.

It is safe to run Dijkstra's algorithm now. We will start by finding the distance from Los Angeles to every other MSA in the graph. Then we will find the shortest path between Los Angeles and Boston. Finally, we will use `print_weighted_path()` to pretty-print the result.

Listing 4.14 dijkstra.py continued

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
```

```
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("")

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:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
```

```
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

You may have noticed that Dijkstra's algorithm has some resemblance to Jarnik's algorithm. They are both greedy, and it is possible to implement them using quite similar code if one is sufficiently motivated. Another algorithm that Dijkstra's algorithm resembles is A* from chapter 2. A* can be thought of as a modification of Dijkstra's algorithm. Add a heuristic and restrict Dijkstra's algorithm to finding a single destination, and the two algorithms are the same.

NOTE:

Dijkstra's algorithm is designed for graphs with positive weights. Graphs with negatively weighted edges can pose a challenge for dijkstra's algorithm and will require modification or an alternative algorithm.

4.6 Real-world applications

A huge amount of our world can be represented using graphs. You have seen in this chapter how effective they are for working with transportation networks, but many other kinds of networks have the same essential optimization problems: telephone networks, computer networks, utility networks (electricity, plumbing, and so on). As a result, graph algorithms are essential for efficiency in the telecommunications, shipping, transportation, and utility industries.

Retailers must handle complex distribution problems. Stores and warehouses can be thought of as vertices and the distances between them as edges. The algorithms are the same. The internet itself is a giant graph, with each connected device a vertex and each wired or wireless connection being an edge. Whether a business is saving fuel or wire, minimum spanning tree and shortest path problem-solving are useful for more than just games. Some of the world's most famous brands became successful by optimizing graph problems: think of Walmart building out an efficient distribution network, Google indexing the web (a giant graph), and FedEx finding the right set of hubs to connect the world's addresses.

Some obvious applications of graph algorithms are social networks and map applications. In a social network, people are vertices, and connections (friendships on Facebook, for instance) are edges. In fact, one of Facebook's most prominent developer tools is known as the "Graph API" (https://developers.facebook.com/ docs/graph-api). In map applications like Apple Maps and Google Maps, graph algorithms are used to provide directions and calculate trip times.

Several popular video games also make explicit use of graph algorithms. MiniMetro and Ticket to Ride are two examples of games that closely mimic the problems solved in this chapter.

4.7 Exercises

1. Add support to the graph framework for removing edges and vertices.
2. Add support to the graph framework for directed graphs (digraphs).
3. Use this chapter's graph framework to prove or disprove the classic Bridges of Konigsberg problem, as described on Wikipedia: https://en.wikipedia.org/wiki/Seven_Bridges_of_Königsberg

[7] Data from the United States Census Bureau's American Fact Finder, https://factfinder.census.gov/.

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

[9] Helena Durnova, "Otakar Boruvka (1899-1995) and the Minimum Spanning Tree" (Institute of Mathematics of the Czech Academy of Sciences, 2006), https://dml.cz/handle/10338.dmlcz/500001.

[10] Inspired by a solution by Robert Sedgewick and Kevin Wayne, *Algorithms*, 4th Edition (Addison-Wesley Professional, 2011), p. 619.