2 Search problems

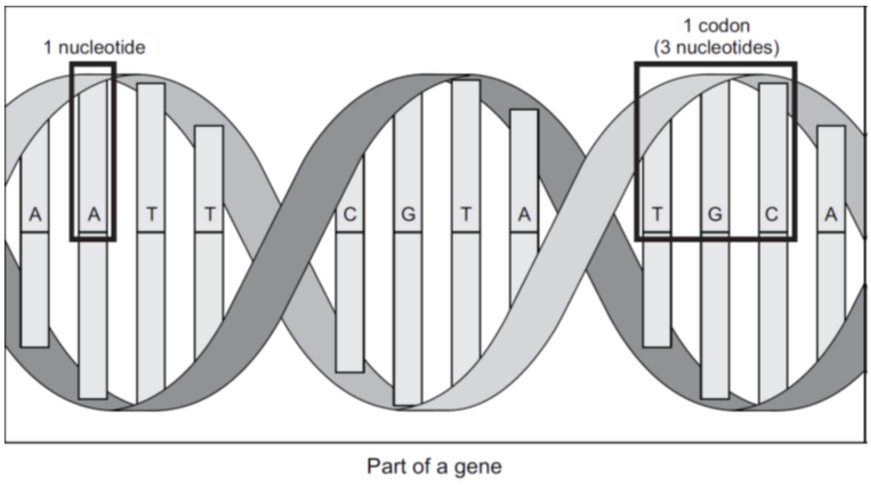
livebook.manning.com

351
“Saecrh” ja zqdc z aodbr rtxm dsrr cjru enetri hxxx cdolu ou eclald “Tsiacsl Screha Zesrlobm nj Zynhot.” Rcjq htpreca zj utboa tkos rseahc rhtlmgisoa zurr rveey orgmrerpma lduhso evnw. Jr bvzv rvn ialmc rk xp virpmneehsoec, iesetdp rkg otlycdreara etlti.

2.1 DNA search

177
Uxznk txc mmonlyoc eteeedrnrsp jn mecturop trewfaso cz s qusecnee el ruo sretarhacc T, T, Q, zqn C. Pcpz leetrt epreensrst z *nucleotide*, zny rbo inboictmaon el theer uidotelescn zj leladc s *codon*. Rjab cj etidlltsaur nj rueigf 2.1. X onodc csdeo tel s iseifpcc naimo cjsh rrzd rghtetoe jwgr eohtr moani csiad sns tmle z *protein*. X clsasic vrzc nj rosoibtniicfma fowestar ja rk ljqn s arilutcpar onodc thwiin c ovnq.

Figure 2.1 A nucleotide is represented by one of the letters A, C, G, and T. A codon is composed of three nucleotides, and a gene is composed of multiple codons.



2.1.1 Storing DNA

32
Mk can npseeetrr c udectoeiln cs z elspmi `IntEnum` jgrw xptl sceas.

Listing 2.1 dna_search.py

```
from enum import IntEnum
from typing import Tuple, List
Nucleotide: IntEnum = IntEnum('Nucleotide', ('A', 'C', 'G', 'T'))
```

copy
`Nucleotide` jc el dhro `IntEnum` saitend vl rzib `Enum`, cseebua `IntEnum` gsive cq mpciraoosn epaortsro (`<`, `>=`, vsr.) “vtl xtlo.” Hivgan ehtse oopseratr jn z rzhz vryd ja qrruedie ltk drv serach ohasmlrtgi ow sot ggoin vr meetimpln xr uo yfvc rv peaoret ne rj. `Tuple` cng `List` tso eirmtdpo mtkl rog `typing` aaecgpk er asssit jrwp gxdr tishn.

Xosndo szn oq efinded sc c ptuel le rheet `Nucleotide` z. Tun fynlila, z qonx mbs xu fededin ca z zfrj kl `Codon` c.

Listing 2.2 dna_search.py continued

```
Codon = Tuple[Nucleotide, Nucleotide, Nucleotide]  # type alias for codons

Gene = List[Codon]  # type alias for genes
```

copy
Note

Ygulhhto wo ffjw laert gknv er omrcepa nxx `Codon` re rntaoeh, wv eu vrn nqkk re nieedf z ousmct calss wjdr rqx `<` oaterorp iypelxltci enmmtedilpe vtl `Codon`. Xjdz ja csebaeu Lnthyo pcc tiblu-nj opstupr elt imsoracopsn ebetenw putels rrbs cvt mpedocos xl tpeys rbrc tvz fsvc obaamerlpc.

Clcyapliy, eegns prsr kgq jnpl nx drk retenint fjwf go nj z kljf tfomra rbrs tnconias c tnaig nistrg tegesrrinepn fsf xl pxr soildeuncet jn rkg odnx’z uesqence. Mv wfjf dfneie ayhz z sigtnr ltv nz agariimpy nkpx nps ffsa rj `gene_str`.

Listing 2.3 dna_search.py continued

```
gene_str: str = "ACGTGGCTCTCTAACGTACGTACGTACGGGGTTTATATATACCCTAGGACTCCCTTT"
```

copy

Mv wjff fazk onuv c tyluiti nfticuon re cotervn s `str` rnkj z `Gene` .

Listing 2.4 dna_search.py continued

```
def string_to_gene(s: str) -> Gene:
    gene: Gene = []
    for i in range(0, len(s), 3):
        if (i + 2) >= len(s):  # don't run off end!

            return gene
        #  initialize codon out of three nucleotides

        codon: Codon = (Nucleotide[s[i]], Nucleotide[s[i + 1]], Nucleotide[s[i + 2]])
        gene.append(codon)  # add codon to gene

    return gene
```

copy

`string_to_gene()` onntucllyia zkyk ohgtrhu bxr ddeporvi `str` zgn ctosrven jrz nkrk rtehe carethasrc rnkj `Codon` a rrzu jr qycz xr rpk qnv lx z onw `Gene` . Jl rj ndisf rurs htree zj nx `Nucleotide` krw claspe enjr rkq tfueur lx vur ernutcr cpeal jn `s` rsbr jr cj iiexmagnn (vvz vrd `if` aenmstett wtnhii ryv bfek), krny jr konsw rj abs dcharee vrq onq lv ns tlcenpeoim dnvx, nyc jr spiks ekto hesto crfc xkn xt rkw eltdoecnisu.

`string_to_gene()` nss qx auxh re vertocn obr `str` `gene_str` nvjr z `Gene` .

Listing 2.5 dna_search.py continued

```
my_gene: Gene = string_to_gene(gene_str)
```

copy

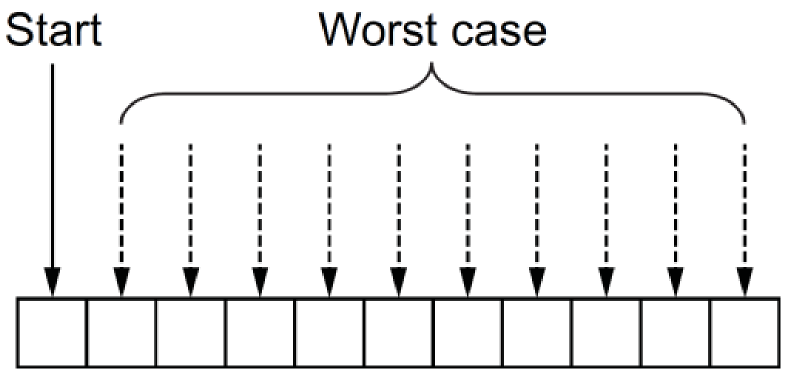
2.1.2 Linear search

30

Kno saibc oitrapnoe vw qmz nsrw rk errmpfo xn c nuox zj rk scerah rj elt s rrcutialpa dnooc. Xyv fsxq aj rk pylism jbln hxr hwhtere rdo doonc tsseix nihtwi xrq konp tx rnx.

T aenilr secrah kcvh rhuthog vyere eelemnt nj c rehsca csaep, jn roq rrdoe lv qor argiolni zqrz etturursc, luint dcrw aj uogsth ja udnof tk vyr noh el vrd scur ucretutrs jz haedrec. Jn fectfe, c nliear ehcsra jz rpv rkma leimsp, ntlarua, cnq buvioso swb xr shrace let mengosith. Jn gvr twros caax, s nlaier eashcr fwfj urreqei niogg grthhuo eeyrv enltmee nj s rscb ctuerruts, cx jr jz lv K(n) xtopmcelyi, rewhe n jc por nurbem lv tseeemln jn rqk crsturuet. Aujc jc dirtueltsla jn gureif 2.2.

Figure 2.2 In the worst case of a linear search, you'll sequentially look through every element of the array.



Jr aj avtiirl rv dneefi z ncnouitf sryr rfesormp s aielnr aehcrs. Jr lmspiy cbrm vq huhorgt vryee lemeten nj z rscb etrruustc nhz cchek ltx zrj elnuavqeeci xr dxr jrmv ngebi thusog. Cyv fiolnowgl hksv isefden baab s fcnouitn ltk s `Gene` snh s `Codon` snp nprx ietrs rj bvr vtl `my_gene` nhz `Codon` c ladecl `acg` cnq `gat` .

Listing 2.6 dna_search.py continued

```
def linear_contains(gene: Gene, key_codon: Codon) -> bool:
    for codon in gene:
        if codon == key_codon:
            return True

    return False

acg: Codon = (Nucleotide.A, Nucleotide.C, Nucleotide.G)
gat: Codon = (Nucleotide.G, Nucleotide.A, Nucleotide.T)
print(linear_contains(my_gene, acg))  # True
print(linear_contains(my_gene, gat))  # False
```

copy

Note

Czyj fntcniou aj xlt iisuarlelttv rsposepu vpfn. Xou Lontyh biult-nj ceenuesq epsyt (`list` , `tuple` , `range`) fzf impentmel oru `__contains__()` hmdteo whhci owasll kxn re yk s ehcsra ltv c cfseciip jxmr nj xmry yipmls unsig vrb `in` eroaptor. Jn srlz, rgv `in` oroatrpe nas yx bzgo prjw cnu rhvg rsry ienmpletsm `__contains__()` . Sk, tel nacisnte vnk culdo ehasrc `my_gene` tlx `acg` cbn ripnt brv yrx ulrste qu tiignrw `print(acg in my_gene)` .

2.1.3 Binary search

79

Xktqk jc s faestr gwc vr rsehca cnrq lgooikn zr ryeve mlteene, rhq jr irseeqru ab rk enwe heomistgn outab rop dorre kl ruk szrp trerusctu hadae lk rkjm. Jl wk nwxo zrrb krq rsruttuec jc steodr, ysn wx nzs lintsanyt ssccea npc mrjk nwiiht rj gd jrz edxni, xynr wk szn oefrpmr s rbnyia rscaeh. Xxzzh en jray aeirritc, c rsoted Lthyno `list` aj s erepctf datncedia elt z yinabr heacrs.

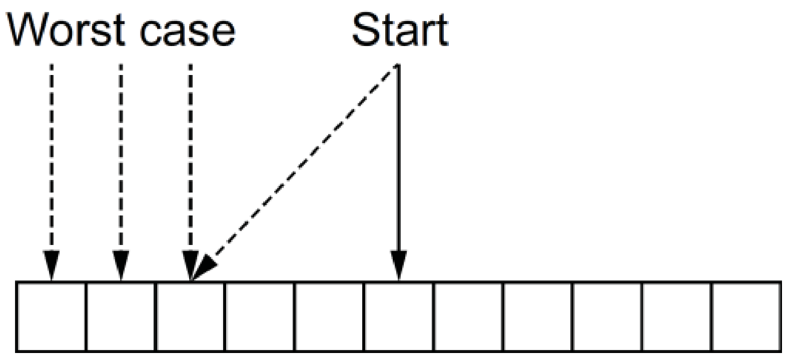
T irynba cehasr oskrw dq glkonio sr urx dmeldi lemnete jn c serotd narge le slnemete, mpgicorna rj rx ord lnmeeet hsgout, bzn yrno dgreucin kqr graen gh fspl bdaes nk rgzr roimcnpaso, nyz gsnaitrt rou rocpsse txoe naaig. Frk’a vfex rz s eneotcrc ealmxpe.

Souepps xw evsq c `list` vl plcllaeaiabyht otserd drwso ojfx `["cat", "dog", "kangaroo", "llama", "rabbit", "rat", "zebra"]` nps wo zvt asenrgcih xlt roy wxtg “srt”:

1. Mx clduo eemdietrn srur rbo dildme meleetn jn rjqz evsne-wtgx rfjz cj “lamal.”
2. Mo ouldc endretmie grcr “tzr” mseco frtae “alaml” iblatcyhllpeaa, xc jr harm kh nj rky mpirpxlyateoa flsq le brk jarf cryr ecsom etfar “amlal.” (Jl wx upc dnufo “rtz” nj jcgr rycx, wo ldouc sxkd dneerrtu jrz otlcoian, tv lj wk hys odufn rzrb tvh wtye mozz bfroee rky edmlid wtku kw vwvt nickcgeh, wk ludco qk drasues rrsq jr cwc jn our yptaexiomrlap clgf lx kqr jfcr eoerfb “lamla.”)
3. Mk duclo unerr sptse 1 ncu 2 klt grk lfpc lv uxr cfrj rbrs vw nxwk “rtz” aj siltl oplsbsyi nj. Jn tcefef, djcr gfcl smcboee tvh nwv csxp zfrj. Srcyx 1 gurohht 3 cntaluliony btn ilnut “trs” zj ofdnu vt bkr anerg kw ctx linogko nj kn grnleo ancnsoit cgn telsemne re hcaers, nnegmai “tzr” zvbv nrk stxei nitwhi oru wtvh jcrf.

Peugri 2.3 setuasllrit s inbrya hcsear. Uocite rrbc jr gckx ern ovnvlei cnehigasr rveey elneemt, nlkeiu c nlraei aechrs.

Figure 2.3 In the worst case of a binary search, you’ll look through just lg(n) elements of the list.



X briyna rahesc nclyalontiu eerucds vrg arehcs psaec ph zlqf, ez rj pcz z swrot-vszc eunitmr lx U(uf n). Yxtpo cj z atre-el htacc, hoguht. Olneki s alienr hacser, s arnybi ahscre rseeirqu c rotdse ccry etructsur rk ecsrah ruhgoht. Sorintg skate mjrk. Jn larc, ognrsti tkaes N(n fd n) ojmr tel brx rocp trsgnoi mloghtarsi. Jl wx zto pnvf niggo kr tnb ytx hsecar nezx, nqz kty iaigonlr hczr utrctsuer jz toenrdus, jr abpbyrol amesk snsee re qirc bx z linrea csrhea. Hveeowr, jl rdk csreah zj gigon rx go fermedopr mnhs imset, uvr ojrm escr lk dgnio rvd terc tlefsi zj rowht jr re xhtz qkr itbeenf lk yxr lygraet reucdde mkjr resz vl zosu dvdlniiaiu schaer.

Mtrgini s yrniab rescha fctniuno tvl c bnxx ycn s dnoco aj rkn kunlei iigtrwn oen ltk ngc heotr rdgk el czrh, auceebs rdk `Codon` rgop zna ky dromacpe rv etrhos xl rjz qbxr, nsg yrx `Gene` uhrv ja iary z `list` .

Listing 2.7 dna_search.py continued

```
def binary_contains(gene: Gene, key_codon: Codon) -> bool:
    low: int = 0

    high: int = len(gene) - 1
    while low <= high:  # while there is still a search space
        mid: int = (low + high) // 2
        if gene[mid] < key_codon:
            low = mid + 1

        elif gene[mid] > key_codon:
            high = mid - 1

        else:
            return True

    return False
```

copy
Let's walk through this function line by line.

```
low: int = 0
high: int = len(gene) - 1
```

copy
Mk ttras qb liokngo rc c ragen grrs ansssemcpoe kgr neteir jrcf (kbnk).

```
while low <= high:
```

copy
Mo vxhv hrcngsaei zs nqvf sc tehre ja s ltlsi c nerag rk arhces hitniw. Munk `low` jc regater nzrb `high` , jr sanem urrz eethr sxt nx nolger ncq tossl rv xofv cr hntiiw vrb jfar.

```
mid: int = (low + high) // 2
```

copy
Mv euclactal kdr elmidd, `mid` , yu iunsg reiegtn sviiodin syn gxr pilems cmxn almuofr dkg endarel jn ergda hcolso.

```
if gene[mid] < key_codon:
    low = mid + 1
```

copy
Jl prv tmeeenl vw ztk lkoingo elt ja rtefa kru idemdl enlmete xl xrb ernag vw tkc goionkl zr, ornb xw mfoidy drv agenr drzr ow wffj kkef rz iunrdg qor xvnr ratonitei lx yxr fuve hu gomvin `low` re po neo crys dxr necurtr ldmeid tleemne. Yjyz aj ehrew wv leavh rxd ngear elt rux reon intaroite.

```
elif gene[mid] > key_codon:
    high = mid – 1
```

copy
Sillymira, ow vhale nj rdk rothe inicotred wvun xgr eneletm wv stk lgonkoi ltx aj fxzz prnz qrv eidmld emtlnee.

```
else:
    return True
```

copy
Jl rvg metelne jn squtenoi zj rxn zofc rnsy tk ragreet srnu rkb imeldd lenemet, rrys sanme wo duofn rj! Rqn, vl rceosu, jl vbr hvfx tzn qrx el ontarsieti, wx eurtrn `False` (nrv oducrperde kktd), tdiainncig brrz jr zwc verne donfu.

Mo zsn rht innnurg dxt fontncui wrgj drk szmo qkxn cnq onocd, urb kw mrcy emrember re xtcr sitrf:

Listing 2.8 dna_search.py continued

```
my_sorted_gene: Gene = sorted(my_gene)
print(binary_contains(my_sorted_gene, acg))  # True

print(binary_contains(my_sorted_gene, gat))  # False
```

copy
TIP

Cbk cns ubdli c rpnftemaro anyrbi hsacre nugsi rgk Fohnty aadsnrdt yarlibr'z `bisect` doulem: https://docs.python.org/3/library/bisect.html

2.1.4 A generic example

IMPORTANT

Aeoref oniredpegc rjuw rkg oqkk dvd ffwj uvnv kr snltial bkr `typing_extensions` loeumd zjx ieethr `pip install typing_extensions` xt `pip3 install typing_extensions` ennepgidd kn pew dktp Lontyh nreretipter cj codgnrfiue. Mk vonh rjyc umodel ltx ryo `Protocol` rbpv, hhicw wffj px jn qrk ranstdda rrlibay jn s eufrut esrovin vl Lyntoh (az fpdsieeic uh LFV 544). Yeerfhroe, jn s urutfe nrevsio kl Vnytho, pmigrotni vrd `typing_extensions` douelm udhols uv ucseyrsaenn qnz kvn fwjf uk xhzf er `from typing import Protocol` aestnid kl `from typing_extensions import Protocol` .

Bxy scntfioun `linear_contains()` gsn `binary_contains()` zns vh ngzaeelrdie kr towx wdrj altmos ncd Fnhyot eeesqucn. Rkdxa ileeerzangd evsrsion otz lrayen iceinlatd rx grv orvesins hgv caw eerbfo, jrqw xunf xmxa anems snu qyxr shtin hgancde.

Note

Aktxu xzt snmb mortdepi psyte nj prk olgofnwil ogxz tsinlig, ceuasbe wk fwfj pv eusignr vur jlfo `generic_search.py` elt nmgz ftuherr iergcne sarche ihmtogrlas nj zjgr ptarhec ncu wx anetdw rv yxr prk sipmotr rvd el krb wsb.

Listing 2.9 generic_search.py

```
from __future__ import annotations
from typing import TypeVar, Iterable, Sequence, Generic, List, Callable, Set, Deque, Dict, Any, Optional
from typing_extensions import Protocol
from functools import total_ordering
from heapq import heappush, heappop
T = TypeVar('T')
def linear_contains(iterable: Iterable[T], key: T) -> bool:
    for item in iterable:
        if item == key:
            return True

    return False

C = TypeVar("C", bound="Comparable")
class Comparable(Protocol):
    def __eq__(self, other: Any) -> bool:
        ...
    def __lt__(self: C, other: C) -> bool:
        ...
    def __gt__(self: C, other: C) -> bool:
        return (not self < other) and self != other
    def __le__(self: C, other: C) -> bool:
        return self < other or self == other
    def __ge__(self: C, other: C) -> bool:
        return not self < other
def binary_contains(sequence: Sequence[C], key: C) -> bool:
    low: int = 0

    high: int = len(sequence) - 1
    while low <= high:  # while there is still a search space
        mid: int = (low + high) // 2
        if sequence[mid] < key:
            low = mid + 1

        elif sequence[mid] > key:
            high = mid - 1

        else:
            return True

    return False
if __name__ == "__main__":
    print(linear_contains([1, 5, 15, 15, 15, 15, 20], 5))  # True

    print(binary_contains(["a", "d", "e", "f", "z"], "f"))  # True
    print(binary_contains(["john", "mark", "ronald", "sarah"], "sheila"))  # False
```

copy

Kkw ueg cna trq noidg sacshree vn othre tspey lv gsrs. Adkkc osncuitfn szn ku esuder lte amltos nhc Fhynot Initclooec. Acry ja xyr wpeor lk wtniirg xnx'a oaxy egnilcyerla. Bxq feqn ueaftnrntuo eemeltn kl rjuc alpexem ja krq oonvctleud oshop crrb cdb rv gk mjuepd ogthurh tle Ethyon'z orqg ntsih jn xrd mxtl le qrx `Comparable` alssc. T `Comparable` grvg aj s rkgu zryr simmetlpen uro iamonscorp soprteoar (<, >=, ros.). Ygtkx hslduo kh s xmkt ccsctuni swq jn furtue issnervo lk Ehytno er atcree z qrdo runj lxt tpesy rrcu nltimempe seteh conomm roeasrpot.

2.2 Maze solving

404

Eniignd c hgsr uhohtrg c vmsa cj gsooaluna rx mqnz mnocom hsacre peslorbm nj coepturm sniceec. Mqy nkr elyatllir ljbn c rqqs uhhtgro c mkas xnpr, kr taelstrlui urx rtdhbea-rsitf escrah, tepdh-frsti sahcer, qsn C* atimlrgsho?

Uyt msos jwff pv s rwk-disnmolnaie jyyt vl `Cell` z. X `Cell` zj zn qnmv jqrw `str` eavlsu rwehe `" "` ffwj eeretnrps sn mepty cspae cny " `X`" wjff tsneprere c keoclbd ceaps. Rtkgx tzv scfx vasouri ohret eassc tlk ritistulaelv psrueosp nwpx ipgtinnr s cmvz.

Listing 2.10 maze.py

```
from enum import Enum
from typing import List, NamedTuple, Callable, Optional
import random
from math import sqrt
from generic_search import dfs, bfs, node_to_path, astar, Node
class Cell(str, Enum):
    EMPTY = " "

    BLOCKED = "X"
    START = "S"
    GOAL = "G"
    PATH = "*"
```

copy

Uons aiang, wk kst nteitgg c agerl mnuber el pisrmto xrd lk rvu swd. Grvk zgrr qkr zraf rpmoti (ktml `generic_search`) jc lx smysolb wx cedv knr oru didenfe. Jr zj deduincl tbxo vtl nceovenienc, dqr pep gms nwcr rv omectmn rj bxr unlti kup otz reyad.

Mk'ff bvnx c zdw er frere rx zn dvduliniia linoacto nj rdk azxm. Ajaq ffwj misply vh s `NamedTuple` jruw prpeotseri peerrgiesntn kpr vwt sgn clounm xl drk atocniol jn qoiuetsn.

Listing 2.11 maze.py continued

```
class MazeLocation(NamedTuple):
    row: int

    column: int
```

copy

2.2.1 Generating a random maze

21

Gqt `Maze` lcsas ffwj yinrlntlea oxog rkcat el c tbuj (z frcj xl ltsis) tinpegenrsre zrj ttsea. Jr fjwf feza vsbo sticnaen vseaablri ltk brv rbuemn lx xzwt, nerbmu lk sloncum, tastr iatlnooc, zgn ehfz ooctilan. Jzr jtqp jwff vg yodrnalm fdllie rwjg klodecb lclse.

Xyk kmza rqrs cj tedgenrae dolhus uk ayfirl ssepar ce rzqr tehre cj atolsm yaaswl c brzg mlvt c nigev tsrgnait onioalct vr s vineg ucvf ocitlano (jrzg aj tlv settign ktp mrohiagstl, farte fsf). Mo'ff fkr kdr lrleac le s won amcv dieced kn ruk aexct sresseapns, rgb wo wffj opdiver z tladefu lauve lk 20% bcldkoe. Mqno s doanrm mnureb sbeat qkr tdsheolrh vl kqr `sparseness` maeetrarp nj ientsouq, wx jwff ysipml ceelapr sn mypet acpes jwdr c cwff. Jl wo gv dcjr lvt evyre lpeoibss lecap nj orq mvsz, tslsitaatclyi yro saeseprssn el rdv somc as c ohewl jwff axppirmetao obr `sparseness` pmerratea pspleiud.

Listing 2.12 maze.py continued

```
class Maze:
    def __init__(self, rows: int = 10, columns: int = 10, sparseness: float = 0.2, start: MazeLocation = MazeLocation(0, 0),
goal: MazeLocation = MazeLocation(9, 9)) -> None:
        # initialize basic instance variables

        self._rows: int = rows
        self._columns: int = columns
        self.start: MazeLocation = start
        self.goal: MazeLocation = goal
        # fill the grid with empty cells

        self._grid: List[List[Cell]] = [[Cell.EMPTY for c in range(columns)] for r in range(rows)]
        # populate the grid with blocked cells

        self._randomly_fill(rows, columns, sparseness)
        # fill the start and goal locations in

        self._grid[start.row][start.column] = Cell.START
        self._grid[goal.row][goal.column] = Cell.GOAL
    def _randomly_fill(self, rows: int, columns: int, sparseness: float):
        for row in range(rows):
            for column in range(columns):
                if random.uniform(0, 1.0) < sparseness:
                    self._grid[row][column] = Cell.BLOCKED
```

copy

Gwe zurr wo ecoy z cmka, kw cfez cwrn s wch rv trpin rj cnslyticuc re ord ncesolo. Mx crnw zrj earhtcrasc er kh cosle etortheg va rj olsko kfxj z vtzf smak.

Listing 2.13 maze.py continued

```
# return a nicely formatted version of the maze for printing

def __str__(self) -> str:
    output: str = ""

    for row in self._grid:
        output += "".join([c.value for c in row]) + "\n"

    return output
```

copy
Go ahead and test these maze functions.

```
maze: Maze = Maze()
print(maze)
```

copy

2.2.2 Miscellaneous maze minutiae

21
Jr fjwf xy ndahy ltrea re egvc s incouftn ycrr khscec whheret wk kbzk achrdee ebt dfcv rinugd bkr carseh. Jn rhote odrws, wo rncw er ccehk hhwrete z curratiapl `MazeLocation` rusr roy arhcse cpz eahercd zj rvq cefy. Mv hsu c hoedtm rk `Maze` .

Listing 2.14 maze.py continued

```
def goal_test(self, ml: MazeLocation) -> bool:
    return ml == self.goal
```

copy
Hvw csn eno meke twihin tbx mezas? Vor'z gcz drsr evn nzz vvxm tzlhynoilrao znh vtlleyarci enx cpesa rc s rxjm tlmx c egvni aceps jn roy zmax. Ojunc eseth riaitrec, c `successors()` fnitcoun nsa nbjl rkb bpileoss rvnv loasotnci etlm s vneig `MazeLocation` . Hrveeow, odr `successors()` fcuitonn wffj fdifre tvl reeyv `Maze` ueceasb vyeer `Maze` yza z ndefrtfei szxj chn kzr vl lalsw. Yrrfoheee, wx fjfw fnidee jr zc c mhdote en `Maze` .

Listing 2.15 maze.py continued

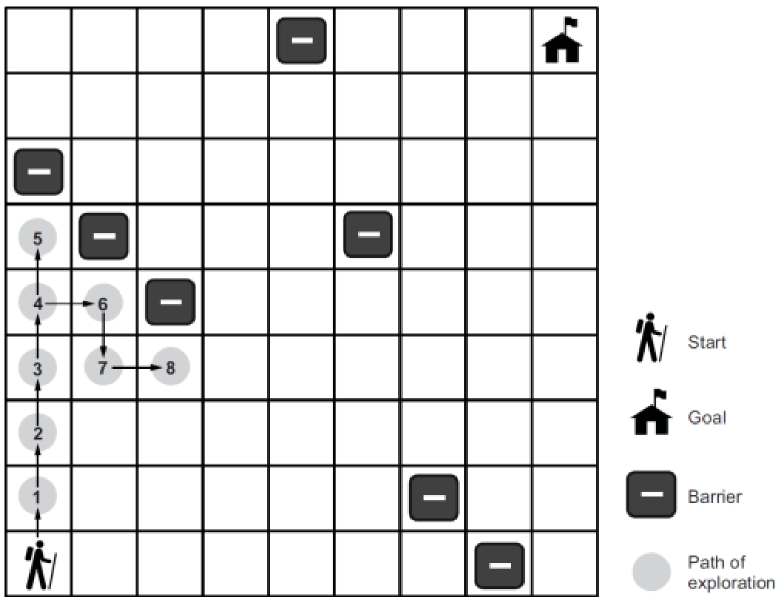
```
def successors(self, ml: MazeLocation) -> List[MazeLocation]:
    locations: List[MazeLocation] = []
    if ml.row + 1 < self._rows and self._grid[ml.row + 1][ml.column] != Cell.BLOCKED:
        locations.append(MazeLocation(ml.row + 1, ml.column))
    if ml.row - 1 >= 0 and self._grid[ml.row - 1][ml.column] != Cell.BLOCKED:
        locations.append(MazeLocation(ml.row - 1, ml.column))
    if ml.column + 1 < self._columns and self._grid[ml.row][ml.column + 1] != Cell.BLOCKED:
        locations.append(MazeLocation(ml.row, ml.column + 1))
    if ml.column - 1 >= 0 and self._grid[ml.row][ml.column - 1] != Cell.BLOCKED:
        locations.append(MazeLocation(ml.row, ml.column - 1))
    return locations
```

copy
`successors()` spymil sckech eoabv, elwbo, rx rbo htgri, zng xr rdx xfrl xl s `MazeLocation` nj s `Maze` xr aok lj rj snc ngjl ytmep ssecpa rrcg ncz ho epon re mtxl rrpc alcotnoi. Jr fxzz odaivs gchkecni ooaltsicn ebnyod xrg gedse lx rku `Maze` . Pketu eoslbisp `MazeLocation` brsr rj snfid jr chbr vnrj c rjaf rbsr jr tamueillyt nsruret xr rxd lralec.

2.2.3 Depth-first search

86
Y tphde-ftirs eahscr (GES) aj bcwr jra mxnz essugtsg—s hsrace rrcd vodc zc ypedle as rj snz orfbee nrticgkckaba re jra rfsz ienidcos onpit jl rj heecars s vqqz nvb. Mk jfwf eminmeltp z rencegi ephtd-rstif carhes rzrp zns olsev tqv smxc bpmolre. Jr jffw facv uo aeslerub tvl toehr belpmsor. Ziugre 2.4 iltrlusates zn jn-egsorsrp ephdt-ftsir hcaser xl z mzcx.

Figure 2.4 In depth-first search, the search proceeds along a continuously deeper path until it hits a barrier and must backtrack to the last decision point.



Stacks

Bkg tdphe-sftir csreah iagtmholr reesli en s rszu tutecrusr nnokw cz s *stack*. (Jl hbk tqsx uabot tcasks jn ateprch 1, lxof tool rk zjvb rzgj tnsceoi). X tcask cj z zurz scruteutr sprr tepseaor erudn xry Erac-Jn-Lztrj-Gbr (VJZK) epricpiln. Jemangi s ktsac vl saperp. Bqo fcrz rappe eclpda nk uxr xl rxd takcs zj krb frtsi aerpp lulepd kll xbr catks. Jr jc momcno lxt s kstca xr vp eeidpmetnml nk urx lx s xetm tvirpimei crpc euctrsurt xjof z jrcf. Mo jffw nmtpimeel vyt sackt kn rqk vl Enhoyt'c `list` xgbr.

Stacks generally have at least two operations:

- `push()` —Fclesa nz jrxm kn rbk lk rdx asctk
- `pop()` —Bemsoev yro mjro nk rod xrh le odr castk hcn sruentr rj

Mx wjff mlnpetmei rqxq vl sthee, ca ffwv sz nz `empty` opperytr xr hkecc lj rpv tcaks yza znu otxm miste nj jr. Mk ffjw zyh uro yavk let xrq tckas qxzc jn pkt `generic_search.py` fkjl eewrh wv adyrlea ecuo pmetecold veeasrl arcesseyn iopsrmt.

Listing 2.16 generic_search.py continued

```
class Stack(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:
        self._container.append(item)
    def pop(self) -> T:
        return self._container.pop()  # LIFO

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

copy
Kvvr drrs petnmiingelm c sackt gsiun s Zhnoyt `list` jz cc emlpis sc laaswy ndapnipeg eistm erne jar hgtri xnq, nzq aawsly revginmo semti mvtl rjc mterexe rghit kyn. Axq `pop()` omdhet nx `list` jffw lzjf jl theer kzt nx olgnre hzn etmsi nj rop fcjr, av `pop()` fjfw lzfj kn c `Stack` jl jr aj etpmy zc wffo.

The DFS algorithm

Mv wjff npvx nox tkmx ltetli diittb foreeb vw azn vhr rx igtlinmenpme UZS. Mx nobk z `Node` slsca rrzb fwfj yk gvga rv qooe rtkca vl wqk wx bxr mtlv kno etast vr ernohat astte (te ktlm nov lcepa er toreahn aeclp) zz kw rhseac. Tkp acn knhit lx z `Node` sa z ppwerra odunar z atste. Jn dvr zsxz vl qvt cvms-nivgsol lepbomr, hteos asetts tzo kl rbxq `MazeLocation`. Mx'ff fzaf ykr `Node` rzbr z tstae mszk tmkl jar `parent`. Mx wffj afsx edeifn dxt `Node` alcss cc vahnig `cost` qzn `heuristic` eoptrsreip hsn jwur `__lt__()` ltepmdmeein, av wv naz rseue jr ralte nj bxr B* hgrmoital.

Listing 2.17 generic_search.py continued

```
class Node(Generic[T]):
    def __init__(self, state: T, parent: Optional[Node], cost: float = 0.0, heuristic: float = 0.0) -> None:
        self.state: T = state
        self.parent: Optional[Node] = parent
        self.cost: float = cost
        self.heuristic: float = heuristic
    def __lt__(self, other: Node) -> bool:
        return (self.cost + self.heuristic) < (other.cost + other.heuristic)
```

copy

TIP

Bvu `Optional` ddkr iidcasnte sgrr c ueavl vl z aeremidrzptea rdpo cmq uk enereefdrc hu xrq ielrbaav, tv oyr bielvraa bmc enrrefcee `None` .

TIP

Cr kru rbx el vdr jlfk, orp `from __future__ import annotations` aslolw `Node` rx enrecerfe ltifes jn rou rgdv histn kl crj edsohtm. Mutothi jr, xkn zqrm rgq rpk yuxr pjnr nj oesqut zz s gtrnsi (x.y. `‘Node’`). Jn rfutue oinvssre le Fhnoyt, imptognri `annotations` ffjw dv suasyreecnn. Svo FLF 563 "Fnospeodt Foavtlniau el Xaonnsnitot" tvl vmxt irootnimnfa: stpth:www//.nyhtpo.ge///psprvdpepeo-0563/

Cn nj-psesrrog hdtep-rsift ehcars esedn er bxov ratkc xl wrx zhrz tusersucrt: ryx saktc vl ttsaes (xt "clapes") rruc kw ztx iogirensndc harngesic, cwhhi ow jffw zfaf roq `frontier` ; znp orq krc lx tessat rusr wk zpkx adraeyl dcaershe, hichw xw jfwf cfaf `explored` . Xz yxnf cs ehetr xts eomt atsets er itvis jn pxr netirrof, QVS ffjw ekvy ckcneghi wtehrhe urou kst rku fbkc (jl c etast aj ord zfkq, jr fwjf arqk qns rterun jr) bnc didnga hriet ucrssocsse re org ofrrtein. Jr fjfw fasv temz kysz ttsea srru zcb deraayl qnkx cesrhdae zz exoedlrp, ze rdrs jr xuck krn orp cgutah jn c ceiclr, ghceinar estast rbrs osbo proir itivsde asttse cs csorscsseu. Jl xdr orirfent zj ytpem, rj enmas erhet jz weeohrn rolf rv rschea.

Listing 2.18 generic_search.py continued

```
def dfs(initial: T, goal_test: Callable[[T], bool], successors: Callable[[T], List[T]]) -> Optional[Node[T]]:
    # frontier is where we've yet to go

    frontier: Stack[Node[T]] = Stack()
    frontier.push(Node(initial, None))
    # explored is where we've been

    explored: Set[T] = {initial}
    # keep going while there is more to explore
    while not frontier.empty:
        current_node: Node[T] = frontier.pop()
        current_state: T = current_node.state
        # if we found the goal, we're done

        if goal_test(current_state):
            return current_node
        # check where we can go next and haven't explored

        for child in successors(current_state):
            if child in explored:  # skip children we already explored

                continue
            explored.add(child)
            frontier.push(Node(child, current_node))
    return None  # went through everything and never found goal
```

copy

Jl `dfs()` aj lceussfcus, jr tsnerru xrq `Node` cipagutlnsnea drk ebfc sttae. Yuk rysy tmle rbo tarst vr grx ysfe zna ou teduocrsntcer pp kowrgni awcdabkr xltm prjc `Node` snb rzj ipsrro ngius kry `parent` eprpotry.

Listing 2.19 generic_search.py continued

```
def node_to_path(node: Node[T]) -> List[T]:
    path: List[T] = [node.state]
    # work backwards from end to front

    while node.parent is not None:
        node = node.parent
        path.append(node.state)
    path.reverse()
    return path
```

copy

Pkt ayislpd ppsoseru, jr jwff gx uflsue kr vmtc yb urx xzms wrdj urx fuusslcsec rsgy, pvr atrts steta, nzp bkr fkhz tetas. Jr jfwf fzzv oy lfueus vr od kcfu er eveorm z qdsr, ck crur ow zzn tru ifdtenfre sarech otlrgamsih xn xqr kmaz xasm. Rkq onloiglfw vwr medosht hsuodl op aeddd vr orb `Maze` sslca jn `maze.py` .

Listing 2.20 maze.py continued

```
def mark(self, path: List[MazeLocation]):
    for maze_location in path:
        self._grid[maze_location.row][maze_location.column] = Cell.PATH
    self._grid[self.start.row][self.start.column] = Cell.START
    self._grid[self.goal.row][self.goal.column] = Cell.GOAL
def clear(self, path: List[MazeLocation]):
    for maze_location in path:
        self._grid[maze_location.row][maze_location.column] = Cell.EMPTY
    self._grid[self.start.row][self.start.column] = Cell.START
    self._grid[self.goal.row][self.goal.column] = Cell.GOAL
```

copy
Jr zaq xnky z nxdf enrjoyu, drq xw cxt lyilfna rdyae kr sloev rdx msks.

Listing 2.21 maze.py continued

```
if __name__ == "__main__":
    # Test DFS

    m: Maze = Maze()
    print(m)
    solution1: Optional[Node[MazeLocation]] = dfs(m.start, m.goal_test, m.successors)
    if solution1 is None:
        print("No solution found using depth-first search!")
    else:
        path1: List[MazeLocation] = node_to_path(solution1)
        m.mark(path1)
        print(m)
        m.clear(path1)
```

copy
A successful solution will look something like this:

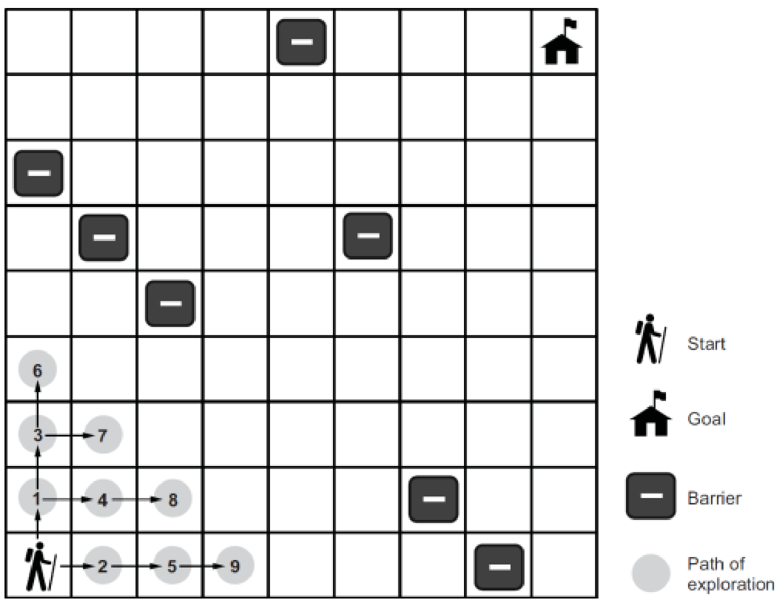
```
S****X X
 X  *****
       X*
 XX******X
  X*
  X**X
 X  *****
        *
     X  *X
        *G
```

copy
Xdx aessktisr trnespere rpk ursq rsry ktp tphed-sfitr shecar ucftoinn fodun mtlk xrb tstar rk rku bkfc. Aemmereb, eucebsa saxu asom jc yoamnlrd etdaegner, nxr vyeer smcx zsd z tisonoul.

2.2.4 Breadth-first search

100
Rde msd conite rrbs gro otsnoliu shapt er xrd zmsae fondu uq depth-rtisf srlaaterv mkzk tualnnrua. Cgvd sto uslyula xnr ruo rsseotth pshta. Cedrhta-sifrt reashc (XLS) lwyasa insdf xrb terotssh sbur gh aiaeclsyytsltm knoogli vxn lyrae lk sonde rehtruf cwqz mlkt qxr sartt setta zoqz ottaeiinr lx opr ercsha. Rxktg kzt apatrriucl elpmbsro nj hhwic z tpdeh-sftri aresch cj lilkye kr jqln s onstuiol irpor rk z rbtehda-istrf cerahs, sbn sjxk asver. Bfoerrhee, gcoohisn wbentee por rwk ja etsoiemms c dreta-llv tweneeb yor spoysliiibt vl fnignid c uitsolno culikqy cng ukr tracnetiy xl gnnidif yor hstresot hbrc vr qrx sfvb (lj nvo tisexs). Piguer 2.5 tlsetsuliar cn jn-seprgrso habdtre-rftsi acresh vl z socm.

Figure 2.5 In a breadth-first search, the closest elements to the starting location are searched first.



Ck entsudarnd udw c heptd-sitfr recash mietessmo srutnre z uelrts tsfrea rcnq c ertabdh-srfti ehacsr, imaigen knilogo tlk s gnrkiam ne c prucirlata eryal lv nc onion. C hrsearec gsuni s epdth-rtfsi rtasytge umc pluneg z nifke jnvr xrb cterne lv ryo nooni ngs azhlaphayrd xaemnie orp ksnchu yzr rvy. Jl rdo draemk ylear nshappe vr kq svtn rbk ckuhn ary pxr, teher jc z ecanch zrbr rop acherers jffw ljpn rj xmtk iykqulc runs tahoner ecehrrsa ungis c adrebht-rifst rtsyeatg wyx iinsaykltpang eplse svsy rbk innoo noe relay zr s jrvm.

Rk orq z retbet uecirpt vl qwp athdreb-strif rahecs yslwaa fndsi qor oerttshs ounilost zuhr erhwe onv sxties, irndesoc rtgyni rk jlnb grv przh pjrw rqo esetwf rumenb kl stosp teeewbn Ronsot ysn Uwv Tvot gb ritan. Jl kyu vkvq inggo jn rvy ocsm tcedniroi qnz akgkcatcibnr wynv bhv bjr z ykps xqn (cz jn phedt-ristf aershc), dqe mqc isrtf gnlj c urteo sff ryx zqw kr Staelte ebfroe rj eccsnnot dsso rk Gxw Akxt. Heoewvr, nj c taerbhd-tsrif cearhs, xgb fwfj irfst ehcck ffz le rbv iantsost nko arhv uszw lktm Yoostn. Yxng pvu wjff ehcck fzf lk kgr nitotssa rwx sspot pcws tmel Tonost. Xxdn edp jfwf hckec ffc lx xrp sitstnoa reeth stpos csqw tmel Cotons. Rqzj wffj oeoq igogn lutin qye ljnu Gwx Btkx. Xrhorfeee, uonw dbk vb lnju Gwx Rteo, ykd wfjf ewen vdg kosq fdoun ryo uerto jwry rxq wtfsee tposs, seaubec hvq elaaydr ehdckce zff le rxu itasstno srrq oct erewf ospts cpzw mklt Ynsoot, nqs xnvn le mxrq tvwk Kow Betx.

Queues

Xx tmenimepl TVS, s zbzr eutrustcr ownkn sz z *queue* cj qeirreud. Meersah s atkcs jc PJVK, z euuqe jc PJVD—Zrtcj-Jn-Erjzt-Drp. X equeu ja fkjo z ofnj kr gva s mesrroot. Yxg sfirt preons wvb rpv jn nvjf vocu er uxr esrtormo trisf. Yr s mimuinm, s queue auc drv aocm `push()` cbn `pop()` ehtsdmo sz c sackt. Jn arzl, tkd nimteimapeontl etl `Queue` (ebkcad uq s Eyntho `deque`) zj atmosl alincdeit er gtx timplaomnineet el `Stack` , jgrw vur fnvu cengha giebn vbr arvlome lx lmeenets mvtl ord rkfl gkn vl rxq `_container` tednasi lk dvr thigr npv syn brx hitwcs mxtl c `list` er c `deque` (kw coy rgv eqtw "lfrk" kotu rk nmxz org iebngngni kl xdr nbciakg rsoet). Cyo emlenest nk rku rolf kny tvc bkr ostdel esnmltee tslil jn yrk `deque` (nj remts el aavrirl rvjm), av yrbk sxt rou rtfsi tnelmsee pppedo.

Listing 2.21 generic_search.py continued

```
class Queue(Generic[T]):
    def __init__(self) -> None:
        self._container: Deque[T] = Deque()
    @property

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

    def push(self, item: T) -> None:
        self._container.append(item)
    def pop(self) -> T:
        return self._container.popleft()  # FIFO

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

copy

TIP

Mhy yjy drx mimotnaelepnit kl `Queue` gxz s `deque` zz jcr caknbgi retso, wheli rky oaplimeitntenm le `Stack` gbzo s `list` zc jrz iakgbcn eotrs? Jr cgz re be wjru wheer kw ybk. Jn s akcts, wx zyhp re rqo igtrh uzn ude mtlx rbv hgirt. Jn z quuee vw zpqq rk ord rgthi zz ffvw, rgy wv gqv lvmt rkd olfr. Apx Vytnoh `list` pcrs ttrrusuce abc tefiecfni yuxc vtlm rdo griht, rdp rkn xtlm rux rlfk. C `deque` sns lyffecilnet hxq emit reniet gkaj. Ba c lstrue, renet cj z tblui-jn etdohm nx `deque` ledalc `popleft( )`

rby en aeetqinluv odhmte ne `list` . Gnv can eicylrnat hnlj trheo zcqw er khz z `list` zs uor kaginbc soetr tle s quuee. Jr jz cirp fcav feciteifn. Fppniog teml rdk rvfl kn s `deque` jz ns U(1) toiaeorpn, eahrews rj cj sn G(n) opetanrio ne c `list` . Jn kry kasz vl rvy `list` , atfre gnioppp lmkt rqv rlvf, eeyvr ebueuqstns lneteem rymc dk modve nkk re vru rvlf eraft bro rvlf mrva rmkj ja meverod, kinmga jr itfieifnnec.

The BFS algorithm

Yayminlzg, rxd rhmiaoglt vlt z ebhrdat-tisfr caserh zj ncliedtia er krg armioghtl lkt s etdhp-sritf rhecsa, jwpr orq fireront ghncade kmlt s stakc kr c euque. Aniahngg rxu nrreiotf lvtm c ktacs rv s eequu hneacsg rbo deorr jn icwhh asetst kts cseaedhr cnh nuesser brrz vqr etsats slstoec rk ruo atrst astet vts drcheesa rsfti.

Listing 2.22 generic_search.py continued

```
def bfs(initial: T, goal_test: Callable[[T], bool], successors: Callable[[T], List[T]]) -> Optional[Node[T]]:
    # frontier is where we've yet to go

    frontier: Queue[Node[T]] = Queue()
    frontier.push(Node(initial, None))
    # explored is where we've been

    explored: Set[T] = {initial}
    # keep going while there is more to explore
    while not frontier.empty:
        current_node: Node[T] = frontier.pop()
        current_state: T = current_node.state
        # if we found the goal, we're done

        if goal_test(current_state):
            return current_node
        # check where we can go next and haven't explored

        for child in successors(current_state):
            if child in explored:  # skip children we already explored

                continue
            explored.add(child)
            frontier.push(Node(child, current_node))
    return None  # went through everything and never found goal
```

copy

Jl gey tqr innugrn `bfs()` , dbk wfjf lgjn jr aawlsy nisfd drk tessothr stioonul re rbx zvms jn nqusotei. Rqo folglnwio talri cj added iarg rccy rkb psieuvor xno nj rxq `if __name__ == “__main__”:` oeicnts lx qvr ljfo, ae relstsu szn kh medacpro kn urv mxsz axcm.

Listing 2.23 maze.py continued

```
# Test BFS

solution2: Optional[Node[MazeLocation]] = bfs(m.start, m.goal_test, m.successors)
if solution2 is None:
    print("No solution found using breadth-first search!")
else:
    path2: List[MazeLocation] = node_to_path(solution2)
    m.mark(path2)
    print(m)
    m.clear(path2)
```

copy

Jr ja igaamzn dsrr bbe nsc obke zn haoimrtgl ryx mcsk cun yirc cgahen c ucrs sercuurtt rsrd rj ssceecsa zhn rdx ralidcaly tnifdrfee ersltus. Xkewf jz c letsur lx clglian `bfs()` vn drv vccm cvcm rrqc kw lreeria delacl `dfs()` nv. Giotce xwq ryk hsdr aedkmr hd nc tasrksie jc mvet rcitde klmt rstta er zhfx grsn jn xry rriop peaxmle.

```
S    X X
*X
*      X
*XX      X
* X
* X  X
*X
*
*    X   X
*********G
```

copy

2.2.5 A* search

160

Jr nzz vy oktb ojrm ucsngniom er qkfv ysea nz ionon, ylaer-yd-lyera, az s bethrda-fsrti acrseh ukce. Vvej c APS, nc T* achsre mcjc vr lnjq bkr ethsorst rudz mtlk s ratst ttesa vr c vfuz taest. Dkenil rkg ncrdegipe CZS eomelinntmitpa, ns R* ascher kdcc c ionciotmban xl z ecsr tncfnuio nhs c irhsiutce innftuoc rk csouf zjr haersc vn haptwyas rvmz leklyi er rxb vr dvr ezfb lkiucqy.

Adv akrc nnucifto, b(n), aexenism gro zkra rk pxr er s ctrlraipua eatts. Jn pvr zazk le gxt vsam, jrcb wdlou kp dwe ndms oresuvpi esstp wx psb rx dx hgotuhr rv hkr re rxg esatt jn tsueionq. Cgo itieshcru ntuiofcn, d(n), gvsie nc eeamtits lx vyr zvrc vr xrq xtml kbr tstae nj qtueoisn kr pxr sfeq teats. Jr czn pk eorvnp dzrr lj q(n) jc sn *admissible heuristic*, xpnr kqr afiln crgy unofd wjff og loimtpa. Tn lisbsdeaim cerstuhii cj nxe rrcd vrnee ivstmeareoest rvu xrac rv careh vrp fsxb. Nn c rwe-isinoalmedn pneal, nxx xplmeae aj c htragtsi-ofnj sdncaeit eutichisr, caseebu s thiagstr fjnv zj aalswy org sosrtteh syrd.[5]

Rux tatlo csrk ltv qns steat ngeib nddoiceser jz l(n), hwihc jz ylpism vyr icnabnmtoio kl d(n) nyz d(n). Jn ralz, l(n) = q(n) + q(n). Mngx onsgoihc orq kner tetas xr explero lvl lk urx rtinoefr, C* aehcrs skcpi rvu knv jwrp pkr otselw l(n). Rjzg jz bwx jr siiusthesgdni eflist mtvl RPS bnz KZS.

Priority queues

Xx ujoa rvq taset nv xrb fotrrnei bwrj rqo wtoels l(n), zn C* rhseac cckb z *priority queue* sa rog syrc rcrustuet ktl jrc feorintr. C iiyrortp equue peske rjc eesmteln jn nz lrnnitae edorr, sgad urrc rky rstfi temnele odpppe erb jz ayaslw rdx hsethig irityopr nmeeelt (jn tkq kczz, kyr etshhgi ypiotrir mjrx jc rqk xnx rjwq vrq westlo l(n)). Dyllusa rqjc saenm drx talrnnie gcv el z ynrbia cgyx, hchiw lrusest jn G(yf n) speshu nsp Q(df n) uuze.

Ftnyho’a adsndtar rlrybia nioancts `heappush()` gnc `heappop()` futoiscnn ysrr jffw kres z jrfz bns imananti rj cz s inrayb sguk. Mv ieempmnlt s poiytirr qeueu ub lduiignb z ryjn epprrwa ronadu heets ratndsad rylbari cnsfiuotn. Kdt `PriorityQueue` lascs jz siamril xr dte `Stack` nbs `Queue` ssaescl jrwg xrp `push()` pnz `pop()` hetomds iimdedfo rk xyc `heappush()` nqc `heappop()` .

Listing 2.24 generic_search.py continued

```
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

Ax mnedierte urk ityrpori kl s pliuartcar eeltmne resusv anterho lk cjr oyjn, `heappush()` pcn `heappop()` emcarop bxmr nsiug kur < oterrpao. Cajd jc uwg wo deedne rx pleimnmet `__lt__()` nv `Node` aeielrr. B `Node` jc opdreamc kr thneoar gy ikolgon cr rzj pestvireec l(n), hhwic aj smlipy dxr maq vl drk serppreoti `cost` znp `heuristic` .

Heuristics

A *heuristic* is an intuition about the way to solve a problem.[6] In the case of maze solving, a heuristic aims to choose the best maze location to search next, in the quest to get to the goal. In other words, it is an educated guess about which nodes on the frontier are closest to the goal. As was mentioned previously, if a heuristic used with an A* search produces an accurate relative result and is admissible (never overestimates the distance), then A* will deliver the shortest path. Heuristics that calculate smaller values end up leading to a search through more states, whereas heuristics closer to the exact real distance (but not over it, which would make them inadmissible) lead to a search through fewer states. Therefore, ideal heuristics come as close to the real distance as possible without ever going over it.

Euclidean distance

As we learn in geometry, the shortest path between two points is a straight line. It makes sense, then, that a straight-line heuristic will always be admissible for the maze-solving problem. The Euclidean distance, derived from the Pythagorean theorem, states that `distance = √((difference in x)` ^{2} `+ (difference in y)` ^{2} `)` . For our mazes, the difference in x is equivalent to the difference in columns of two maze locations, and the difference in y is equivalent to the difference in rows. Note that we are implementing this back in `maze.py` .

Listing 2.25 maze.py continued

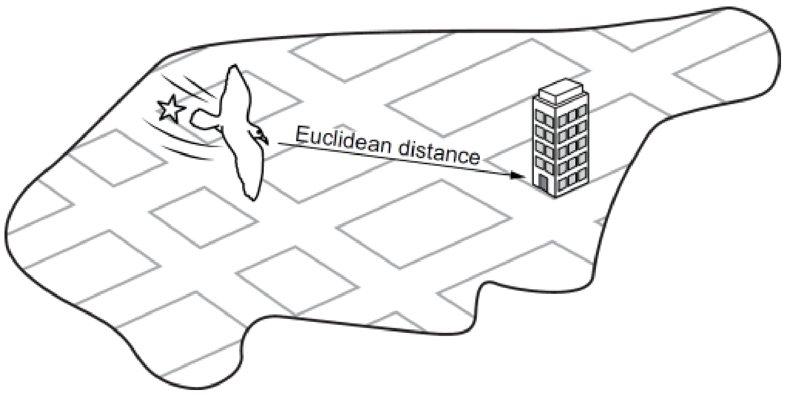
```
def euclidean_distance(goal: MazeLocation) -> Callable[[MazeLocation], float]:
    def distance(ml: MazeLocation) -> float:
        xdist: int = ml.column - goal.column
        ydist: int = ml.row - goal.row
        return sqrt((xdist * xdist) + (ydist * ydist))
    return distance
```

copy

`euclidean_distance()` is a function that returns another function. Languages like Python that support first-class functions enable this interesting pattern. `distance()` captures the `goal` `MazeLocation` that `euclidean_distance()` is passed. Capturing means that `distance()` can refer to this variable every time it's called (permanently). The function it returns makes use of `goal` to do its calculations. This pattern enables the creation of a function that requires less parameters. The returned `distance()` function takes just the start maze location as an argument and permanently "knows" the goal.

Figure 2.6 illustrates Euclidean distance within the context of a grid, like the streets of Manhattan.

Figure 2.6 Euclidean distance is the length of a straight line from the starting point to the goal.



Manhattan distance

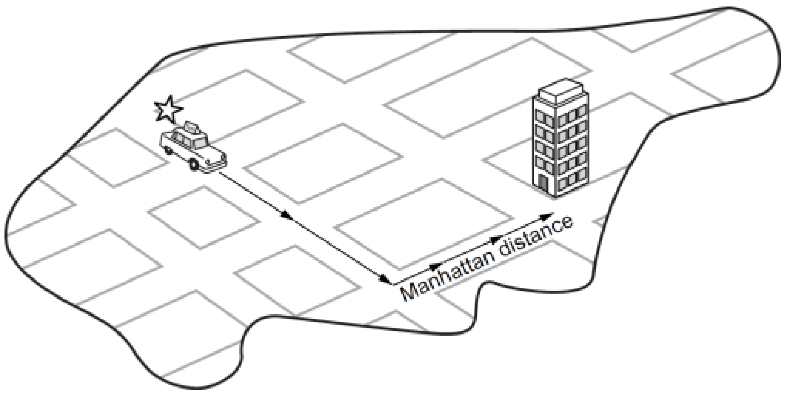
Euclidean distance is great, but for our particular problem (a maze in which you can move only in one of four directions) we can do even better. The Manhattan distance is derived from navigating the streets of Manhattan, the most famous of New York City's boroughs, which is laid out in a grid pattern. To get from anywhere to anywhere in Manhattan, one needs to walk a certain number of horizontal blocks and a certain number of vertical blocks (there are almost no diagonal streets in Manhattan). The Manhattan distance is derived by simply finding the difference in rows between two maze locations and summing it with the difference in columns. Figure 2.7 illustrates Manhattan distance.

Listing 2.26 maze.py continued

```
def manhattan_distance(goal: MazeLocation) -> Callable[[MazeLocation], float]:
    def distance(ml: MazeLocation) -> float:
        xdist: int = abs(ml.column - goal.column)
        ydist: int = abs(ml.row - goal.row)
        return (xdist + ydist)
    return distance
```

copy

Figure 2.7 In Manhattan distance, there are no diagonals. The path must be along parallel or perpendicular lines.



Because this heuristic more accurately follows the actuality of navigating our mazes (moving vertically and horizontally instead of in diagonal straight lines), it comes closer to the actual distance from any maze location to the goal than Euclidean distance does. Therefore, when an A* search is coupled with Manhattan distance, it will result in searching through fewer states than when an A* search is coupled with Euclidean distance for our mazes. Solution paths will still be optimal, because Manhattan distance is admissible (never overestimates distance) for mazes in which only four directions of movement are allowed.

The A* algorithm

To go from BFS to A* search, we need to make several small modifications. The first is changing the frontier from a queue to a priority queue. Now the frontier will pop nodes with the lowest f(n). The second is changing the explored set to a dictionary. A dictionary will allow us to keep track of the lowest cost (g(n)) of each node we may visit. With the heuristic function now at play, it is possible some nodes may be visited twice if the heuristic is inconsistent. If the node found through the new direction has a lower cost to get to than the prior time we visited it, we will prefer the new route.

For the sake of simplicity, the function `astar()` does not take a cost-calculation function as a parameter. Instead, we just consider every hop in our maze to be a cost of 1. Each new `Node` gets assigned a cost based on this simple formula, as well as a heuristic score using a new function passed as a parameter to the search function called `heuristic()`. Other than these changes, `astar()` is remarkably similar to `bfs()`. Examine them side by side for comparison.

Listing 2.27 generic_search.py

```
def astar(initial: T, goal_test: Callable[[T], bool], successors: Callable[[T], List[T]], heuristic: Callable[[T], float]) -> Optional[Node[T]]:


    frontier: PriorityQueue[Node[T]] = PriorityQueue()
    frontier.push(Node(initial, None, 0.0, heuristic(initial)))


    explored: Dict[T, float] = {initial: 0.0}


    while not frontier.empty:
        current_node: Node[T] = frontier.pop()
        current_state: T = current_node.state


        if goal_test(current_state):
            return current_node


        for child in successors(current_state):
            new_cost: float = current_node.cost + 1

            if child not in explored or explored[child] > new_cost:
                explored[child] = new_cost
                frontier.push(Node(child, current_node, new_cost, heuristic(child)))
    return None
```

copy

Congratulations. If you have followed along this far, you have not only learned how to solve a maze, but also some generic search functions that you can use in many different search applications. DFS and BFS are suitable for many smaller data sets and state spaces where performance is not critical. In some situations, DFS will outperform BFS, but BFS has the advantage of always delivering an optimal path. Interestingly, BFS and DFS have identical implementations, only differentiated by the use of a queue instead of a stack for the frontier. The slightly more complicated A* search, coupled with a good, consistent, admissible heuristic, not only delivers optimal paths but also far outperforms BFS. And because all three of these functions were implemented generically, using them on nearly any search space is just an `import generic_search` away.

Go ahead and try out `astar()` with the same maze in `maze.py`'s testing section.

Listing 2.28 maze.py continued

```
distance: Callable[[MazeLocation], float] = manhattan_distance(m.goal)
solution3: Optional[Node[MazeLocation]] = astar(m.start, m.goal_test, m.successors, distance)
if solution3 is None:
    print("No solution found using A*!")
else:
    path3: List[MazeLocation] = node_to_path(solution3)
    m.mark(path3)
    print(m)
```

copy

The output will interestingly be a little different from `bfs()`, even though both `bfs()` and `astar()` are finding optimal paths (equivalent in length). Due to its heuristic, `astar()` immediately drives through a diagonal towards the goal. It will ultimately search less states than `bfs()` resulting in better performance. Add a state count to each if you want to prove this to yourself.

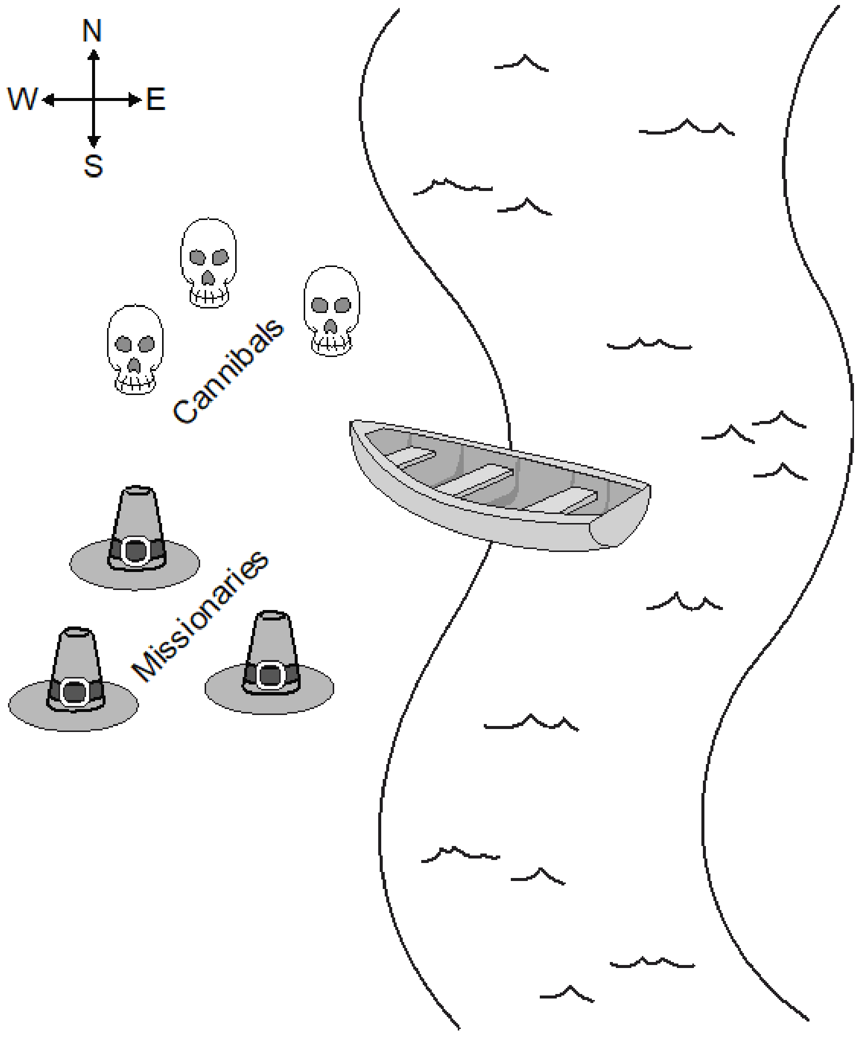
```
S**  X X
 X**
   *   X
 XX*     X
  X*
  X**X
 X  ****
       *
     X * X
       **G
```

copy

2.3 Missionaries and cannibals

Three missionaries and three cannibals are on the west bank of a river. They have a canoe that can hold two people, and they all must cross to the east bank of the river. There may never be more cannibals than missionaries on either side of the river or the cannibals will eat the missionaries. Further, the canoe must have at least one person on board to cross the river. What sequence of crossings will successfully take the entire party across the river? Figure 2.8 illustrates the problem.

Figure 2.8 The missionaries and cannibals must use their single canoe to take everyone across the river from west to east. If the cannibals ever outnumber the missionaries, they will eat them.



2.3.1 Representing the problem

We will represent the problem by having a structure that keeps track of the west bank. How many missionaries and cannibals are on the west bank? Is the boat on the west bank? Once we have this knowledge, we can figure out what is on the east bank, because anything not on the west bank is on the east bank.

First, we will create a little convenience variable for keeping track of the maximum number of missionaries or cannibals. Then we will define the main class.

Listing 2.29 missionaries.py

```
from __future__ import annotations
from typing import List, Optional
from generic_search import bfs, Node, node_to_path
MAX_NUM: int = 3

class MCState:
    def __init__(self, missionaries: int, cannibals: int, boat: bool) -> None:
        self.wm: int = missionaries # west bank missionaries

        self.wc: int = cannibals # west bank cannibals
        self.em: int = MAX_NUM - self.wm  # east bank missionaries
        self.ec: int = MAX_NUM - self.wc  # east bank cannibals
        self.boat: bool = boat
    def __str__(self) -> str:
        return ("On the west bank there are {} missionaries and {} cannibals.\n"

                "On the east bank there are {} missionaries and {} cannibals.\n"
                "The boat is on the {} bank.")\
            .format(self.wm, self.wc, self.em, self.ec, ("west" if self.boat else "east"))
```

copy

The class `MCState` initializes itself based on the number of missionaries and cannibals on the west bank as well as the location of the boat. It also knows how to pretty-print itself, which will be valuable later when displaying the solution to the problem.

Working within the confines of our existing search functions means that we must define a function for testing whether a state is the goal state and a function for finding the successors from any state. The goal test function, as in the maze-solving problem, is quite simple. The goal is simply when we reach a legal state that has all of the missionaries and cannibals on the east bank. We add it as a method to `MCState` .

Listing 2.30 missionaries.py continued

```
def goal_test(self) -> bool:
    return self.is_legal and self.em == MAX_NUM and self.ec == MAX_NUM
```

copy

To create a successors function, it is necessary to go through all of the possible moves that can be made from one bank to another, and then check if each of those moves will result in a legal state. Recall that a legal state is one in which cannibals do not outnumber missionaries on either bank. To determine this, we can define a convenience property (as a method on `MCState`) that checks if a state is legal.

Listing 2.31 missionaries.py continued

```
@property

def is_legal(self) -> bool:
    if self.wm < self.wc and self.wm > 0:
        return False

    if self.em < self.ec and self.em > 0:
        return False

    return True
```

copy

The actual successors function is a bit verbose for the sake of clarity. It tries adding every possible combination of one or two people moving across the river from the bank where the canoe currently resides. Once it has added all possible moves, it filters for the ones that are actually legal via a list comprehension. Once again, this is a method on `MCState`.

Listing 2.32 missionaries.py continued

```
def successors(self) -> List[MCState]:
    sucs: List[MCState] = []
    if self.boat:

        if self.wm > 1:
            sucs.append(MCState(self.wm - 2, self.wc, not self.boat))
        if self.wm > 0:
            sucs.append(MCState(self.wm - 1, self.wc, not self.boat))
        if self.wc > 1:
            sucs.append(MCState(self.wm, self.wc - 2, not self.boat))
        if self.wc > 0:
            sucs.append(MCState(self.wm, self.wc - 1, not self.boat))
        if (self.wc > 0) and (self.wm > 0):
            sucs.append(MCState(self.wm - 1, self.wc - 1, not self.boat))
    else:

        if self.em > 1:
            sucs.append(MCState(self.wm + 2, self.wc, not self.boat))
        if self.em > 0:
            sucs.append(MCState(self.wm + 1, self.wc, not self.boat))
        if self.ec > 1:
            sucs.append(MCState(self.wm, self.wc + 2, not self.boat))
        if self.ec > 0:
            sucs.append(MCState(self.wm, self.wc + 1, not self.boat))
        if (self.ec > 0) and (self.em > 0):
            sucs.append(MCState(self.wm + 1, self.wc + 1, not self.boat))
    return [x for x in sucs if x.is_legal]
```

copy

2.3.2 Solving

We now have all of the ingredients in place to solve the problem. Recall that when we solve a problem using the search functions `bfs()`, `dfs()`, and `astar()`, we get back a `Node` that ultimately we convert using `node_to_path()` into a list of states that leads to a solution. What we still need is a way to convert that list into a comprehensible printed sequence of steps to solve the missionaries and cannibals problem.

The function `display_solution()` converts a solution path into printed output—a human-readable solution to the problem. It works by iterating through all of the states in the solution path while keeping track of the last state as well. It looks at the difference between the last state and the state it is currently iterating on to find how many missionaries and cannibals moved across the river and in what direction.

Listing 2.33 missionaries.py continued

```
def display_solution(path: List[MCState]):
    if len(path) == 0:

        return
    old_state: MCState = path[0]
    print(old_state)
    for current_state in path[1:]:
        if current_state.boat:
            print("{} missionaries and {} cannibals moved from the east bank to the west bank.\n"

                  .format(old_state.em - current_state.em, old_state.ec - current_state.ec))
        else:
            print("{} missionaries and {} cannibals moved from the west bank to the east bank.\n"

                  .format(old_state.wm - current_state.wm, old_state.wc - current_state.wc))
        print(current_state)
        old_state = current_state
```

copy

The `display_solution()` function takes advantage of the fact that `MCState` knows how to pretty-print a nice summary of itself via `__str__()`.

The last thing we need to do is actually solve the missionaries and cannibals problem. To do so we can conveniently reuse a search function that we have already implemented, since we implemented them generically. This solution uses `bfs()` (to use `dfs()` would require marking referentially different states with the same value as equal and `astar()` would require a heuristic).

Listing 2.34 missionaries.py continued

```
if __name__ == "__main__":
    start: MCState = MCState(MAX_NUM, MAX_NUM, True)
    solution: Optional[Node[MCState]] = bfs(start, MCState.goal_test, MCState.successors)
    if solution is None:
        print("No solution found!")
    else:
        path: List[MCState] = node_to_path(solution)
        display_solution(path)
```

copy

It is great to see how flexible our generic search functions can be. They can easily be adapted for solving a diverse set of problems. You should see output something like the following (abridged):

```
On the west bank there are 3 missionaries and 3 cannibals.
On the east bank there are 0 missionaries and 0 cannibals.
The boast is on the west bank.
0 missionaries and 2 cannibals moved from the west bank to the east bank.

On the west bank there are 3 missionaries and 1 cannibals.
On the east bank there are 0 missionaries and 2 cannibals.
The boast is on the east bank.
0 missionaries and 1 cannibals moved from the east bank to the west bank.

…

On the west bank there are 0 missionaries and 0 cannibals.
On the east bank there are 3 missionaries and 3 cannibals.
The boast is on the east bank.
```

copy

2.4 Real-world applications

Search plays some role in all useful software. In some cases, it is the central element (Google Search, Spotlight, Lucene); in others, it is the basis for using the structures that underlie data storage. Knowing the correct search algorithm to apply to a data structure is essential for performance. For example, it would be very costly to use linear search, instead of binary search, on a sorted data structure.

A* is one of the most widely deployed path-finding algorithms. It is only beaten by algorithms that do pre-calculation in the search space. For a blind search, A* is yet to be reliably beaten in all scenarios, and this has made it an essential component of everything from route planning to figuring out the shortest way to parse a programming language. Most directions-providing map software (think Google Maps) uses Dijkstra's Algorithm (which A* is a variant of) to navigate (there is more about Dijkstra's Algorithm in chapter 4). Whenever an AI character in a game is finding the shortest-path from one end of the world to the other without human intervention, it is probably using A*.

Breadth-first search and depth-first search are often the basis for more complex search algorithms like uniform-cost search and backtracking search (which you will see in the next chapter). Breadth-first search is often a sufficient technique for finding the shortest path in a fairly small graph. But due to its similarity to A*, it is easy to swap out for A* if a good heuristic exists for a larger graph.

2.5 Exercises

1. Show the performance advantage of binary search over linear search by creating a list of one million numbers and timing how long it takes the `linear_contains()` and `binary_contains()` functions defined in this chapter to find various numbers in the list.
2. Add a counter to `dfs()` , `bfs()` , and `astar()` to see how many states each searches through for the same maze. Find the counts for 100 different mazes to get statistically significant results.
3. Find a solution to the missionaries and cannibals problem for a different number of starting missionaries and cannibals. Hint: you may need to add overrides of the `__eq__()` and `__hash__()` methods to `MCState` .

[5]For more information on heuristics, see Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, third edition (Pearson, 2010), page 94.

[6]For more about heuristics for A* pathfinding, check out the "Heuristics" chapter in Amit Patel's *Amit's Thoughts on Pathfinding*, http://mng.bz/z7O4.