resitev

January 28, 2024

Mestna občina Ljubljana je objavila video o vedenju kolesarjev v Ljubljani. Na kratko: kolesarji so po mnenju MOL znani po divjih spustih po stopnicah, divjanju med pešci in tako naprej.* Ker je osnovno prevozno sredstvo vašega profesorja kolo in ker tretjino letne kilometrine žal opravi v Ljubljani, vas prosi, da mu za lažje načrtovanje poti rešite tole nalogo.

* Seveda se med kolesarji v resnici najdejo tudi breobzirni divjaki. Vtis, ki ga želi pustiti video, pa je vseeno morda nekoliko pretiran. Sploh začetno divjanje po stopnicah; downhill vidimo na Golovcu, Klobuku in Rašici, ne ob Ljubljanici. Pa tudi tam večina kolesarjev spodobno pazi na pešce, slab vtis puščajo le posamični norci.

Zemljevid na sliki kaže 21 križišč v Ljubljani (na zahtevo mestnih strokovnjakov za varstvo osebnih podatkov smo imena lokacij zamenjali s črkami od A do V) in povezave med njimi. Povezave zahtevajo različne veščine: kdor hoče, na primer priti iz točke B do C, mora obvladati vožnjo med odvrženimi skiroji in slalom med cvetličnimi lonci.

Celoten seznam veščin, ki se pojavljajo v nalogi, je:

- **s**topnice: Spust po stopnicah
- **p**ešci: Divjanje med pešci
- **l**onci: Slalom med cvetličnimi lonci
- **b**olt: Slalom med odvrženimi skiroji
- **r**obnik: Skok na robnik pločnika
- **g**ravel: Vožnja po razsutem makadamu
- **t**rava: Oranje zelenic parkov
- **a**vtocesta: Vožnja po avtocesti
- **c**repinje: Vožnja po razbiti steklovini
- ro**d**eo: Vožnja po kolesarski poti skozi Črnuče

Zemljevid na sliki zaradi pomanjkanja prostora uporablja enočrkovne okrajšave veščin, v sami nalogi pa je zapisan takole:

```
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, R, S, T, U, V = "ABCDEFGHIJKLMNOPRSTUV"

zemljevid = {
    (A, B): "gravel trava",
    (A, V): "pešci lonci",
    (B, C): "bolt lonci",
    (B, V): "",
    (C, R): "stopnice pešci lonci",
    (D, F): "stopnice pešci",
```

```
    (D, R): "pešci",
    (E, I): "trava lonci",
    (F, G): "trava črepinje",
    (G, H): "črepinje pešci",
    (G, I): "avtocesta",
    (H, J): "robnik bolt",
    (I, M): "avtocesta",
    (I, P): "gravel",
    (I, R): "stopnice robnik",
    (J, K): "",
    (J, L): "gravel bolt",
    (K, M): "stopnice bolt",
    (L, M): "robnik pešci",
    (M, N): "rodeo",
    (N, P): "gravel",
    (O, P): "gravel",
    (P, S): "",
    (R, U): "trava pešci",
    (R, V): "pešci lonci",
    (S, T): "robnik trava",
    (T, U): "gravel trava",
    (U, V): "robnik lonci trava"
}
```

Ključi zemljevida so pari povezanih točk, pripadajoča vrednost pa je niz, ki vsebuje s presledkom ločene okrajšave veščin. Tako vidimo pod ključem `(B, C)` zapisano `"bolt lonci"`, kar je okrajšava za veščini *Slalom med odvrženimi skiroji* in *Slalom med cvetličnimi lonci*.

Vse povezave so dvosmerne (ker je kolesarjem po mnenju MOL itak vseeno, v katero smer in po kateri strani ceste vozijo). Če obstaja povezava med B in C obstaja tudi med C in B ter zahteva enake veščine.

0.1 Obvezna naloga

Napiši naslednje funkcije

1. Funkcija `dvosmerni_zemljevid(zemljevi)` prejme zemljevid (slovar, kakršen je gornji) in vrne **nov zemljevid**, ki se od podanega razlikuje po tem, da

   - se vse povezave pojavijo tudi v obrnjeni smeri (če je v podanem zemljevidu ključ `(B, C)`, je v vrnjenem zemljevidu poleg njega tudi ključ `(C, B)`);
   - vrednosti niso več nizi temveč množice veščin.

   Klic

   ```
   dvosmerni_zemljevid({(A, B): "robnik bolt",
                        (A, C): "bolt rodeo pešci",
                        (C, D): ""}
   ```

   vrne

   ```
   {('A', 'B'): {'robnik', 'bolt'},
   ```

```
('B', 'A'): {'robnik', 'bolt'},
('A', 'C'): {'bolt', 'rodeo', 'pešci'},
('C', 'A'): {'bolt', 'rodeo', 'pešci'},
('C', 'D'): set(),
('D', 'C'): set()}
```

Toplo priporočam, da to funkcijo uporabite v nekaterih od naslednjih funkcij.

Rešitev To funkcijo ste pisali predvsem samo zato, da vam olajša življenje v nadaljevanju. Vseeno se je pokazalo nekaj zanimivih reči.

"Pravilna" rešitev je takšna:

[2]:
```
def dvosmerni_zemljevid(zemljevid):
    dvosmerni = {}
    for (a, b), v in zemljevid.items():
        dvosmerni[(a, b)] = dvosmerni[(b, a)] = set(v.split())
    return dvosmerni
```

Pisali bi lahko tudi `dvosmerni[a, b]` in `dvosmerni[b, a]`, brez dodatnih oklepajev okrog `a` in `b`. Python kot indeks uporabi tisto, kar je med oglatimi oklepaji in če je tam več stvari ločenih z vejico, je to zanj pač terka. Sam raje dodam oklepaje, saj v knjižnici `numpy`, ki je namenjena delu z večdimenzionalnimi tabelami in je v praksi glavni motor Pythona, z dvema indeksoma indeksiramo dvodimenzionalne tabele in tam o njima ne razmišljamo kot o terki (čeprav v resnici sta). Skratka, zaradi preglednosti tule raje eksplicitno pokažem, da gre za terko.

Precej običajna komplikacija je bila, da so študenti prirejali dvakrat.

```
dvosmerni[(a, b)] = set(v.split())
dvosmerni[(b, a)] = set(v.split())
```

Druga pogosta nerodnost je, da se študenti na zavedajo (pa naj še tolikokrat pokažem na predavanjih :), da lahko `set` podamo seznam. Množico zato sestavljajo z zanko:

```
vescine = set()
for vescina in v.split():
    vescine.add(vescina)
```

Sami, čisto sami so si krivi, če se potem še zmotijo. Ali pa če imajo preveč domače naloge.

Drugo, kar mi je padlo v oči, so težave z razpakiranjem ključev. Mnogi so namesto `for (a, b), v in zemljevid.items():` pisali `for k, v in zemljevid.items():` (opomba: `k` in `v` sta pogosto imeni, ki ju dodelimo kakim generičnim, splošnim **k**ljučem in **v**rednostim, kar posrečeno sovpada tudi z angleškim **k**ey in **v**value; pri prvi besedi gre za isti etimološki izvor, pri drugi za naključje). Zdaj `k` vsebuje oba ključa. Ustrezno prirejanje bi lahko bilo kar

```
dvosmerni[k] = dvosmerni[k[::-1] = mnozica_vescin(v)
```

vendar študenti za kaj takega večinoma niso dovolj pogumni in raje pišejo

```
dvosmerni[kljuc[0], kljuc[1]] = dvosmerni[kljuc[1], kljuc[0]] = mnozica_vescin(v)
```

Če že hočemo, bi bilo smiselno

```
    for k, v in zemljevid.items():
        a, b = k
```

in potem tako kot v prvi rešitvi.

Naloga (nadaljevanje)

- `mozna_pot(pot, zemljevid)` prejme `pot` v obliki niza z zaporedjem križišč in `zemljevid` v obliki iz uvoda naloge. Funkcija mora vrniti `True`, če je takšna pot možna (torej: če obstajajo povezave med vsemi zaporednimi križišči v nizu) in `False`, če ni.

  Klic `mozna_pot("ABCRVRIEIPNM", zemljevid)` vrne `True`, klic `mozna_pot("ABCRVREPNM", zemljevid)` pa `False`, ker ni povezave iz R v E.

Rešitev Bože, kaj se je dogajalo pri tej nalogi! :) Rešitev je preprosta: gremo čez pare, za vsakega pogledamo, ali nastopa (kot ključ) v slovarju, in čim zalotimo takšnega, ki ne, zatulimo "FALSCH!". Če preživimo do konca, pa `True`.

[3]:
```
def mozna_pot(pot, zemljevid):
    zemljevid = dvosmerni_zemljevid(zemljevid)
    for a, b in zip(pot, pot[1:]):
        if (a, b) not in zemljevid:
            return False
    return True
```

Preden nadaljujemo, opazímo tole: znotraj funkcije smo naredili spremenljivko `zemljevid`, ki povozi tisto, ki smo jo dobili kot argument. Točneje, znotraj funkcije se ime `zemljevid` nanaša najprej nanaša na slovar, ki smo ga dobili kot argument, od prve vrstice naprej pa na slovar, ki ga na podlagi dobljenega slovarja sestavi `dvosmerni_zemljevid`. S tem ni nič narobe, to ni spremenilo slovarja, ki smo ga dobili kot argument. Morda bi imel nekoliko prav, kdor bi se pritoževal, da s spreminjanjem tega, na kar se nanaša `zemljevid`, ustvarjamo zmedo. Vendar gledam na to tako, kot da je funkcija `dvosmerni_zemljevid` zgolj "dopolnila" podani zemljevid s povezavami v nasprotno smer. Izgovorim se lahko tudi, da smo to naredili čisto na začetku funkcije. V splošnem pa bi imel v prejšnjem stavku omenjeni kritik prav: če se neko ime v prvi polovici funkcije nanaša na nekaj, v drugi polovici pa na nekaj drugega, bi bilo to zelo slabo, nepregledno, zavajajoče, nevarno.

Gre tudi malenkost krajše.

[4]:
```
def mozna_pot(pot, zemljevid):
    zemljevid = dvosmerni_zemljevid(zemljevid)
    for povezava in zip(pot, pot[1:]):
        if povezava not in zemljevid:
            return False
    return True
```

Če rečemo, da morajo vse povezave nastopati kot ključi v slovarju, mora biti množica povezav podmnožica množice ključev, in rešitev je potemtakem lahko kar:

[5]:
```
def mozna_pot(pot, zemljevid):
    return set(zip(pot, pot[1:])) <= set(dvosmerni_zemljevid(zemljevid))
```

Prav ta teden pa se bomo bomo učili, da se da ono različico z zanko napisati tudi učinkoviteje:

[6]:
```
def mozna_pot(pot, zemljevid):
    zemljevid = dvosmerni_zemljevid(zemljevid)
    return all(povezava in zemljevid for povezava in zip(pot, pot[1:]))
```

Najbolj tipična komplikacija tule je bila, da niste uporabili `zip`-a, temveč `range(len(pot))` in indekse. To je potem izgledalo tako:

[7]:
```
def mozna_pot(pot, zemljevid):
    zemljevid = dvosmerni_zemljevid(zemljevid)
    for i in range(len(pot) - 1):
        if (pot[i], pot[i + 1]) not in zemljevid:
            return False
    return True
```

To ni spodobno, je pa še OK. Veliko manj OK so vse rešitve, ki iz kakega razloga delajo zanko čez `zemljevid`. To je počasno in ne izkorišča tega, kar nam nudijo slovarji. Slovarjev, nimamo zato, da bi delali zanke čeznje, temveč zato, da ne bi delali zank čeznje.

Zelo pogosto sem videval tudi rešitve, ki najprej zgradijo seznam povezav in gredo nato čezenj, recimo takole (ali pa, še bolj zapleteno, z `range` ali celo kaj daljšega):

[8]:
```
def mozna_pot(pot, zemljevid):
    zemljevid = dvosmerni_zemljevid(zemljevid)

    povezave = []
    for a, b in zip(pot, pot[1:]):
        povezave.append((a, b))

    for povezava in povezave:
        if povezava not in zemljevid:
            return False
    return True
```

To je nepotrebno prelaganje podatkov med seznami. S tem ne pridobimo ničesar, razen priložnosti za napake.

Naloga (nadaljevanje)

- Funkcija`potrebne_vescine(pot, zemljevid)` prejme enake argumente kot prejšnja funkcija, s tem da bo pot, ki jo prejme, bo vedno možna. Funkcija mora vrniti množico veščin, ki jih mora kolesar obvladati, če želi prevoziti to pot.

  Klic `potrebne_vescine("RIPSTUT", zemljevid)` vrne `{'robnik', 'stopnice', 'makadam', 'trava'}`, saj so to veščine, ki jih potrebujemo za to pot (na zemljevidu označene kot *sr*, *g*, *rt* in *gt*).

Rešitev Naloga je malo podobna prejšnji, le da namesto preverjanja, ali je pot možna, sestavi množico in vanjo dodaja veščine, potrebne za vsako povezavo.

[9]:
```
def potrebne_vescine(pot, zemljevid):
    zemljevid = dvosmerni_zemljevid(zemljevid)
    vescine = set()
    for povezava in zip(pot, pot[1:]):
        vescine |= zemljevid[povezava]
    return vescine
```

Tudi to funkcijo lahko po nepotrebnem zapletemo na enake načine kot zgornjo. Pa še na enega zraven - in to je naredilo veliko študentov. Namesto

```
vescine |= zemljevid[povezava]
```

so pisali

```
for vescina in zemljevid[povezava]:
    vescine.add(povezava)
```

Če želimo v neko množico dodati vse elemente neke druge množice, je to pač unija.

Nekateri so bili še previdnejši in pred dodajanjem preverili, da ni element slučajno že v množici, `if zemljevid[povezava] not in vescine`. Tudi to je nepotrebno. Množica ne more dvakrat vsebovati istega elementa.

Nekatere je zmotilo, da njihove množice ne vsebujejo elementov v enakem vrstnem redu, v kakršnem so navedene v rešitvi. Kot sem opozoril na predavanju, vrstni red elementov v množici ni določen. Lahko se spreminja med izvajanjem. In vsakič, ko poženete program, je lahko drugačen. Množice nimajo koncepta "vrstnega reda", zato ta ni pomemben.

[10]:
```
{1, 2, 3, 4} == {2, 1, 4, 3}
```

[10]:
```
True
```

Ob tej nalogi se je dogajalo še nekaj zanimivega: nekateri so poskušali tole:

```
vescine.add(zemljevid[povezava])
```

To ne gre. To ni unija (unija doda v množico vse elemente neke druge množice), temveč dodajanje. Po tem, bi množica `vescine` vsebovala celo množico `povezava` kot *element*. Množice lahko vsebujejo samo nespremenljive stvari, množice pa so spremenljive. Zato množice ne morejo vsebovati množic. Pa četudi bi to šlo, ne bi bilo pravilno, saj mora funkcija vrniti množico veščin, ne pa množico množic veščin.

Naloga (nadaljevanje)

- Funkcija `nepotrebne_vescine(pot, zemljevid, vescine)` prejme enake argumente, poleg tega pa še množico veščin, ki jih obvlada kolesar. Vrniti mora množico veščin, ki so za to pot nepotrebne.

  Klic `nepotrebne_vescine("IPNMNPO", zemljevid, {'stopnice', 'makadam', 'bolt', 'rodeo'})` vrne `{'stopnice', 'bolt'}`, saj tidve veščini za pot `"IPNMNPO"` nista potrebni.

Rešitev Kdor je napisal kaj več kot

[11]:
```
def nepotrebne_vescine(pot, zemljevid, vescine):
    return vescine - potrebne_vescine(pot, zemljevid)
```

ima preveč časa.

Naloga (nadaljevanje)

- Funkcija `tocke_vescine(zemljevid, vescina)` vrne niz z vsemi točkami, iz katerih vodijo povezave, ki zahtevajo določeno veščino. Vsaka črka naj se pojavi le enkrat. Črke naj bodo urejene po abecedi.

  Klic `tocke_vescin(zemljevid, "avtocesta")` vrne `"GIM"`, saj se veščina ***a**vtocesta* pojavlja na povezavah, ki vodijo iz točk G, I in M.

  Klic `tocke_vescine(zemljevid, "rodeo")` vrne `"MN"`, ker imajo pravo kolesarsko stezo z rodeom samo Črnučani.

Rešitev Tu bo potrebno iti čez celoten zemljevid in spet bomo potrebovali tako ključe kot vrednosti, torej gremo čez `zemljevid.items()`. Pač pa točk ne bomo razpakirali, saj ni potrebe.

[1]:
```
def tocke_vescine(zemljevid, vescina):
    zemljevid = dvosmerni_zemljevid(zemljevid)
    tocke = set()
    for tocki, vescine in zemljevid.items():
        if vescina in vescine:
            tocke |= set(tocki)
    return "".join(sorted(tocke))
```

Omembe vredni sta zadnji vrstici. Če povezava med točkama `tocki` zahteva podano veščino, v `tocke` priunijamo tidve točki, `tocke |= set(tock)`. Seveda bi lahko dodali tudi vsako posebej, s `tocke.add`, a za to bi ju bilo potrebno razpakirati.

Kar naloga zahteva niz z urejenimi točkami, seznam točk uredimo (`sorted(tocke)`), potem pa njegove elemente, nize, zlepimo skupaj v en sam niz, kar se najprikladneje naredi kar z `"".join`. Seveda bi šlo tudi z

```
urejene = ""
for tocka in sorted(tocke):
    urejene += tocka
return urejene
```

Tak način je počasnejši (v splošnem, tu pa seveda niti ne, saj točk ni veliko), predvsem pa daljši. In trikrat daljši program ima tipično tudi trikrat več napak.

Tule nam v resnici ne bi bilo potrebno uporabiti obrnjenega zemljevida: pomaga nam le v toliko, da so njegove vrednosti množice veščin in ne nizi z (s presledki) ločenimi oznakami veščin. Zato bi bila pravilna rešitev tudi

[2]:
```
def tocke_vescine(zemljevid, vescina):
    tocke = set()
    for tocki, vescine in zemljevid.items():
```

```
        if vescina in vescine.split():
            tocke |= set(tocki)
    return "".join(sorted(tocke))
```

Poleg tega, da smo izpustili `zemljevid = dvosmerni_zemljevid(zemljevid)`, smo le še spremenili pogoj: namesto if `if vescina in vescine`, moramo, ker so `vescine` zdaj niz, pisati `if vescina in vescine.split()`. V množico jih ni potrebno pretvarjati; iskanje elementa v seznamu ne bo vzelo nič več (točneje: vzelo bo manj) časa kot pretvarjanje seznama v množico. Pač pa ne smemo izpustiti `split`: pogoj `if vescina in vescine` bi sicer deloval, vendar le zato, ker nobena veščina ne nastopa kot podniz druge veščine. Če bi imeli poleg veščine *stopnice* tudi veščino *top* (na primer, ko se MOL odloči, da kolesarskih stez ne bo zasneževal le s snegom, napluženim s ceste, temveč bo kupil še snežne topove), pa bi `if vescina in vescine` takrat, ko bi bila `vescina` enaka `"top"`, pobral tudi povezave, kjer sploh ne bi bilo topov, temveč zgolj stopnice.

0.2 Dodatna naloga

Napiši funkcijo `koncna_tocka(pot, zemljevid, vescine)`, ki prejme enake argumente kot `nepotrebne_vescine`. Vrniti mora dve stvari: točko, do katere lahko kolesar s temi veščinami prevozi to pot, in množico veščin, ki mu manjkajo, da bi šel naprej iz te točke. Ta množica naj ne vključuje morebitnih drugih veščin, ki bi ga čakale na nadaljnji poti, temveč le manjkajoče veščine za prvo povezavo, ki je ne uspe prevoziti.

Klic `koncna_tocka("ABCRVB", zemljevid, {"makadam", "trava"})` vrne `("B", {'lonci', 'bolt'})`. Kolesar, ki bi se namenil iti po poti `"ABCRVB"` bi se zataknil v točki B, ker ne zna voziti slaloma med cvetličnimi lonci in odvrženimi skiroji. Nadaljnja pot bi sicer zahtevala še druge veščine (recimo spust po stopnicah med C in R), vendar funkcija ne gleda naprej.

0.2.1 Rešitev

Gremo po poti. Če naletimo na povezavo, ki zahteva veščine, ki jih kolesar ne premore, vrne trenutno točko in množico veščin, potrebnih za to povezavo, ki ji odštejemo množico veščin, ki jih kolesar obvlada. Če pridemo do konca, pa vrnemo zadnjo točko in prazno množico.

[12]:
```
def koncna_tocka(pot, zemljevid, vescine):
    zemljevid = dvosmerni_zemljevid(zemljevid)
    for a, b in zip(pot, pot[1:]):
        if not zemljevid[(a, b)] <= vescine:
            return a, zemljevid[(a, b)] - vescine
    return pot[-1], set()
```

Zanimiv je pogoj, `not zemljevid[(a, b)] <= vescine`. To ni isto kot `zemljevid[(a, b)] > vescine`. Pogoj `not zemljevid[(a, b)] <= vescine` pravi, da *ni res, da množica* `vescine` *vsebuje vse veščine, ki so potrebne na povezavi (in morda še kakšno zraven).* Pogoj `zemljevid[(a, b)] > vescine` pa pravi, da *povezava zahteva vse veščine, ki jih premore kolesar in še vsaj eno zraven.*

Recimo, da imamo

[13]:
```
vescine = {"a", "b", "c"}
na_povezavi = {"c", "d"}
```

Pogoj, ki preveri, ali se kolesar tu ustavi, se glasi

[14]:
```
not na_povezavi <= vescine
```

[14]:
```
True
```

`True`, saj se kolesar dejansko ustavi, ker ne obvlada veščine d.

Pogoj, ki je navidez le poenostavljena oblika gornjega, pa ne vrne istega, pravega rezultata.

[15]:
```
na_povezavi > vescine
```

[15]:
```
False
```

Če sta `x` in `y` dve števili, velja, da je `not x <= y` isto kot`x > y`. Če sta `x` in `y` množici, pa pač ni tako.