resitev

January 28, 2024

1 Nagradne točke

Oddelek za gospodarstvo in motorni promet pri Mestni občini Ljubljana je za popularizacijo kolesarjenja uvedel sistem nagradnih točk za uporabo različnih veščin. Vožnja po travi je vredna tri točke, divjanje med pešci štiri točke, vožnja po avtocesti deset točk in tako naprej. Kolesar, ki zbere določeno število nagradnih točk, dobi vozniško dovoljenje za motorno vozilo kategorije C (+ koncesijo za parkiranje na pločniku).

Točkovanje:

- črepinje: 1,
- robnik: 1,
- lonci: 1,
- gravel: 2,
- bolt: 2,
- rodeo: 2,
- trava: 3,
- pešci: 4,
- stopnice: 6,
- avtocesta: 10.

Nekaj sprememb v primerjavi z nalogo Načrtovanje poti:

- Veščine so iste kot zadnjič, le da uporabljamo samo okrajšave.
- Ključi v zemljevidu so enaki, vendar sta v zemljevidu že obe smeri, tako da se vam ni potrebno zafrkavati s funkcijo `dvosmerni_zemljevid`.
- Vrednosti v zemljevidu so množice okrajšanih veščin, na primer `{"pešci", "avtocesta"}`. (Da bi neka povezava zahtevala divjanje med pešci na avtocesti, mogoče zveni hecno, a glede na to, da so po mnenju MOL površine za pešce in kolesarje primerne tudi za parkiranje (slike na desni je posneta nedaleč od PeF), tudi preusmeritev prometa na pločnike ne bi bila preveliko presenečenje.)
- funkcije zemljevida ne dobijo kot argument, temveč uporabljajo kar globalno spremenljivko `zemljevid`.

Pri reševanju mi boste hvaležni za spodnjo sliko.

1.1 Za oceno 6-7

- Napiši funkcijo `vrednost_povezave(povezava)`, ki vrne število nagradnih točk, ki jih dobi kolesar, če prevozi `povezavo`. Povezava je podana kot terka z imeni križišč, na primer `("A", "B")`, torej v enaki obliki kot ključi zemljevida. Število točk je enako vsoti točk, ki jih dobi

za potrebne veščine. Če so potrebne veščine za neko povezavo `{"pešci", "avtocesta", "bolt"}`, mora funkcija vrniti `16` (to je, `4 + 10 + 2`).

- Napiši funkcijo `najboljsa_povezava(pot)`, ki prejme pot in vrne tisto povezavo na tej poti, ki mu prinese največ točk. Pot je podana v obliki niza, npr. `"ABVRU"`, rezultat pa naj bo v obliki terke, npr. `("V", "R")`.
- Napiši funkcijo `vrednost_poti(pot)`, ki vrne število nagradnih točk, ki jih dobi kolesar za določeno pot, to je, vsoto nagradnih točk za vse povezave na tej poti. Če mora na poti večkrat uporabiti različno veščino, dobi točke za vsako uporabo. Če pot ni možna zaradi manjkajočih povezav, mora funkcija vrniti `None`.

1.1.1 Rešitev

`vrednost_povezave` `vrednost_povezave` (skoraj) zahteva, da se spomnite učinkovitega načina za shranjevanje vrednosti posameznih veščin: uporabiti morate slovar, katerega ključi so veščine, vrednosti pa število točk, ki jih prinese njihova uporaba.

Sestavimo ga lahko tako:

[1]:
```
tocke = {"gravel": 2, "trava": 3, "lonci": 1, "bolt": 2,
         "pešci": 4, "stopnice": 6, "avtocesta": 10,
         "črepinje": 1, "robnik": 1, "rodeo": 2}
```

Če smo bolj lene sorte, ko gre za tipkanje narekovajev in vejic in dvopičij, pa tako:

[2]:
```
tocke = dict(gravel=2, trava=3, lonci=1, bolt=2, pešci=4,
             stopnice=6, avtocesta=10, črepinje=1, robnik=1,
             rodeo=2)
```

Ker tule kličemo `dict` s poimenovanimi argumenti, ta trik žal vžge le, če so ključi slovarja veljavna imena v Pythonu - začeti se morajo s črko ter vsebovati le črke, števke in podčrtaje.

Kakorkoli, funkcija je potem preprosto

[3]:
```
def vrednost_povezave(povezava, zemljevid):
    vrednost = 0
    for vescina in zemljevid[povezava]:
        vrednost += tocke[vescina]
    return vrednost
```

Ali, če znamo:

[4]:
```
def vrednost_povezave(povezava, zemljevid):
    return sum(tocke[vescina] for vescina in zemljevid[povezava])
```

Oboje je spodobno, čeprav je drugo spodobnejše. Nespodobno pa je tole:

[5]:
```
def vrednost_povezave(povezava, zemljevid):
    for ključ, vrednost in zemljevid.items():
        if ključ == povezava:
```

```
            vescine = vrednost
            break

    vsota = 0
    for ključ, vrednost in tocke.items():
        for vescina in vescine:
            if ključ == vescina:
                vsota += vrednost
                break

    return vsota
```

Prva zanka gre čez vse pare ključ-vrednost v zemljevidu (tudi spremenljivke so namerno poimenovane nerodno), da poišče tistega, ki ustreza podani povezavi. Slovarje imamo prav zato, da nam tega ne bi bilo potrebno početi: celotna prva zanka naredi isto, kot `vescine = zemljevid[povezava]`, le da je bistveno daljša in še bistveneje počasnejša.

Druga je še bolj grozna: gre čez ves slovar parov veščina-točke, da za vsako veščino preveri, ali se nahaja v množici potrebnih veščin (kar spet naredi na najbolj neučinkovit možen način) in v tem primeru prišteje število točk k vsoti. Notranji `for` in `if` je možno zamenjati z `if ključ in vescine`, a vse to je tako ali tako nepotrebno, saj bi morala že zunanja zanka teči prek `vescin`. Tako, kot piše tu, za vsako veščino, ki se točkuje, preverimo, ali je potrebna; prav bi bilo za vsako veščino, ki je potrebna, ugotoviti, koliko točk je vredna. To bi bilo hitrejše, učinkovitejše, preprostejše. Boljše.

`najboljsa_povezava` Tole zahteva zanko, kakršno smo pisali nekje od drugega tedna predavanj. Klasična naloga: iščemo najboljši element glede na nek kriterij, zato si moramo znotraj zanke sproti zapomniti tako najboljši element kot vrednost kriterija.

[6]:
```
def najboljsa_povezava(zemljevid):
    naj_tock = 0
    naj_povezava = None
    for povezava in zemljevid:
        tock = vrednost_povezave(povezava, zemljevid)
        if tock > naj_tock:
            naj_tock = tock
            naj_povezava = povezava
    return naj_povezava
```

Ker je to tako pogosta reč, jo lahko izvedemo tudi s funkcijo `max`. Iščemo najboljšo povezavo, se pravi najboljši ključ iz slovarja, torej `max(zemljevid)`. Ker pa jih ne bomo primerjali ravno po abecedi, podamo dodatni argument `key`, ki vsebuje funkcijo, ki jo bo `max` poklical za vsak element. Elemente bo potem primerjal po vrednosti funkcije.

Potrebovali bi funkcijo, ki prejme povezavo in vrne njeno vrednost. Pač, naredimo.

[7]:
```
def najboljsa_povezava(zemljevid):
    def vrednost(povezava):
```

```
        return vrednost_povezave(povezava, zemljevid)

    return max(zemljevid, key=vrednost)
```

Tule bi lahko kdo pomislil, da bo šlo tudi tako.

[8]:
```
# Tole ne deluje

def najboljsa_povezava(zemljevid):
    return max(zemljevid, key=vrednost_povezave)
```

Vendar žal ne: funkcija `vrednost_povezave` zahteva tudi argument `zemljevid`, `max` pa bo poklical ključ z enim samim argumentom, povezavo.

Prav tako ne bo delovalo tole:

[9]:
```
def vrednost(povezava):
    return vrednost_povezave(povezava, zemljevid)

def najboljsa_povezava(zemljevid):
    return max(zemljevid, key=vrednost)
```

Funkcijo `vrednost` moramo definirati znotraj `najboljsa_povezava`, da vidi vrednost argumenta `zemljevid`. (Tu zadaj je nekaj znanosti; kogar zanima, naj pogleda, kaj je to closure.

Pač pa se lahko takšni poimenovani funkciji izognemo z uporabo lambda-funkcije.

[10]:
```
def najboljsa_povezava(zemljevid):
    return max(zemljevid,
               key=lambda povezava: vrednost_povezave(povezava, zemljevid))
```

Lambda-funkcijam se pri tem predmetu ne posvečamo preveč. Za programerje-začetnike morda niso tako pomembne. Če bi bile lambde v Pythonu lepe in bi jih več uporabljali, ne rečem. Žal pa so lambde v Pythonu okorne in jih ne uporabljamo veliko, zato tudi ni potrebe, da bi to grdobijo kazali študentom. Mogoče kdaj, ko se bomo programiranja raje učili v Kotlinu ali podobnem jeziku, kjer so lambde zakon.

`vrednost_poti` Gremo po poti in seštevamo. Naloga preverja, ali znamo priti do zaporednih elementov seznama - bodisi z indeksiranjem ali, boljše, z `zip`.

Slabša različica, torej, je

[11]:
```
def vrednost_poti(pot, zemljevid):
    vrednost = 0
    for i in range(len(pot) - 1):
        vrednost += vrednost_povezave((pot[i], pot[i + 1]), zemljevid)
    return vrednost
```

Boljše pa je

[12]:
```
def vrednost_poti(pot, zemljevid):
    vrednost = 0
    for a, b in zip(pot, pot[1:]):
        vrednost += vrednost_povezave((a, b), zemljevid)
    return vrednost
```

Pri tej, slednji, opazimo še, da nam para, ki ga sestavi `zip`, ni potrebno razpakirati v dve spremenljivki, saj funkcija `vrednost_povezave` tako ali tako zahteva natančno takšen par. Torej lahko pišemo kar

[13]:
```
def vrednost_poti(pot, zemljevid):
    vrednost = 0
    for povezava in zip(pot, pot[1:]):
        vrednost += vrednost_povezave(povezava, zemljevid)
    return vrednost
```

Vse tri različice se dajo spraviti v eno vrstico, saj računamo, preprosto povedano, *vsoto vrednosti, ki jo vrača* ***`vrednost_povezave`*** *za vse povezave na poti*:

[14]:
```
def vrednost_poti(pot, zemljevid):
    return sum(vrednost_povezave(povezava, zemljevid)
               for povezava in zip(pot, pot[1:]))
```

1.2 Za oceno 7-8

- Napiši funkcijo `enkratna_vrednost_poti(pot)`, ki vrne vrednost poti, pri čemer pa se vsaka uporabljena veščina šteje samo enkrat. Če kolesar, na primer, dvakrat vozi po travi, dobi vseeno le tri točke (kolikor je vredna vožnja po travi) in ne šestih. Če pot ni možna zaradi manjkajočih povezav, mora funkcija vrniti `None`.
- Napiši funkcijo `mozna_pot(pot, vescine)`, ki vrne `True`, če je s podano množico veščin možno prevoziti podano pot, pri čemer pa **smemo vsako veščino uporabiti le enkrat**. Če se na poti, na primer, dvakrat pojavijo stopnice, se pred drugimi stopnicami ustavi. (Če sploh ne zna voziti po stopnicah, pa se ustavi že pred prvimi stopnicami.) Pazite tudi na to, da nekatere povezave na poti morda sploh ne bodo obstajale; v tem primeru seveda vrnete `False`.

1.2.1 Rešitev

`enkratna_vrednost_poti` Napisati moramo nekaj takšnega. V množico `porabljeno` bomo beležili veščine, ki jih je kolesaril že uporabil. (Zakaj množico? Ker vanjo dodajamo in ker preverjamo, ali vsebuje neko vrednost. To delo se najbolj poda množicam.) Potem seštevamo vrednosti, podobno kot v prejšnji nalogi, vendar pred seštevanje damo še pogoj: veščino upoštevamo le, če še ni bila uporabljena, obenem pa zabeležimo, da je uporabljena.

[15]:
```
def enkratna_vrednost_poti(pot):
    porabljeno = set()
    vrednost = 0
    for povezava in zip(pot, pot[1:]):
```

```
        for vescina in zemljevid[povezava]:
            if vescina not in porabljeno:
                vrednost += tocke[vescina]
                porabljeno.add(vescina)
    return vrednost
```

Naloga zahteva še, da vrnemo `None`, če pot ni možna. Lahko bi šli po odsekih poti in vsakega preverili, ali se nahaja med ključi zemljevida. Recimo tako, da bi na začetek funkcije dodali nekaj v slogu

```
    for povezava in zip(pot, pot[1:]):
        if povezava not in zemljevid:
            return None
```

Gre pa še preprosteje: vse odseke poti vržemo v množico (`set(zip(pot, pot[1:]))`). Ključe zemljevida damo v drugo množico (`set(zemljevid)`). Prva množica mora biti podmnožica prve, sicer pot ni možna.

[16]:
```
def enkratna_vrednost_poti(pot):
    if not set(zip(pot, pot[1:])) <= set(zemljevid):
        return None

    porabljeno = set()
    vrednost = 0
    for povezava in zip(pot, pot[1:]):
        for vescina in zemljevid[povezava]:
            if vescina not in porabljeno:
                vrednost += tocke[vescina]
                porabljeno.add(vescina)
    return vrednost
```

Ljubitelji jedrnatosti pa napišejo kar

[17]:
```
def enkratna_vrednost_poti(pot):
    if not set(zip(pot, pot[1:])) <= set(zemljevid):
        return None

    mozne = tocke.copy()
    return sum(mozne.pop(vescina, 0)
               for povezava in zip(pot, pot[1:])
               for vescina in zemljevid[povezava])
```

`mozna_pot` En pristop je, da v množico beležimo vse veščine, ki smo jih uporabili. Pri vsaki povezavi, ki jo kanimo ubrati, preverimo, ali posedujemo vse potrebne veščine in ali nobene od veščin, ki jih zahteva povezava, še nismo uporabili.

[18]:
```
def mozna_pot(pot, zemljevid, vescine):
    uporabljene = set()
```

```
    for povezava in zip(pot, pot[1:]):
        if povezava not in zemljevid \
                or not zemljevid[povezava] <= vescine \
                or zemljevid[povezava] & uporabljene:
            return False
        uporabljene |= zemljevid[povezava]
    return True
```

Čeprav `zemljevid[povezava]` ne vzame nič posebno veliko časa, gre človeku vseeno na živce, da mora to stalno ponavljati. Temu se lahko izognemo z dodatno spremenljivko.

[19]:
```
def mozna_pot(pot, zemljevid, vescine):
    uporabljene = set()
    for povezava in zip(pot, pot[1:]):
        if povezava not in zemljevid:
            return False
        potrebne = zemljevid[povezava]
        if not potrebne <= vescine or potrebne & uporabljene:
            return False
        uporabljene |= zemljevid[povezava]
    return True
```

Vendar smo si s tem nakopali dodatni `if`. Znebimo se ga z Pythonovim kontroverznim mrožem. Prav zato obstaja, prav zaradi takih situacij.

[20]:
```
def mozna_pot(pot, zemljevid, vescine):
    uporabljene = set()
    for povezava in zip(pot, pot[1:]):
        if povezava not in zemljevid \
                or not (potrebne := zemljevid[povezava]) <= vescine \
                or potrebne & uporabljene:
            return False
        uporabljene |= potrebne
    return True
```

Lahko pa se znajdemo drugače: če povezava ne obstaja, se delamo, da zahteva veščino, ki je sploh ni in je zato gotovo nimamo.

[21]:
```
def mozna_pot(pot, zemljevid, vescine):
    uporabljene = set()
    for povezava in zip(pot, pot[1:]):
        potrebne = zemljevid.get(povezava, {"ne bo šlo"})
        if not potrebne <= vescine or potrebne & uporabljene:
            return False
        uporabljene |= potrebne
    return True
```

Naloge pa se lahko lotimo tudi z druge strani: namesto, da v dodatno množico beležimo, kar smo

že uporabili, lahko iz množice `vescine` preprosto odstranjujemo uporabljene veščine.

[22]:
```
def mozna_pot(pot, zemljevid, vescine):
    for povezava in zip(pot, pot[1:]):
        potrebne = zemljevid.get(povezava, {"ne bo šlo"})
        if not potrebne <= vescine:
            return povezava[0]
        vescine = vescine - potrebne
    return pot[-1]
```

Tule je pomembno, da pišemo vescine = vescine - potrebne in ne vescine -= potrebne. Prvo naredi novo množico (ki je razlika trenutnih veščin in uporabljenih, potrebnih veščin), drugo pa spremeni obstoječo množico vescine. Ker smo množico vescine prejeli kot argument funkcije, je ne smemo spreminjati, saj bi dejansko spremenili množico tistemu, ki je poklical funkcijo.

1.3 Za oceno 8-9

- Napiši funkcijo `do_nagrade(pot, meja)`, ki vrne točko na poti, do katerega lahko pelje kolesar, tako da skupno število njegovih nagradnih točk *še ne preseže* podane meje `meja` (po čemer mu MOL za izkazano objestnost podeli častno vozniško dovoljenje). Pri tem štetju se vsaka uporabljena veščina lahko šteje večkrat (kot pri nalogi `vrednost_poti`, ne kot pri `enkratna_vrednost_poti`).

  Če pot zaradi kake manjkajoče povezave ni možna, funkcija vrne točko, kjer se je kolesar prisiljen ustaviti.

  Lahko pa se zgodi tudi, da pride do konca poti. V tem primeru funkcija seveda vrne zadnjo točko.

1.3.1 Rešitev

Tule pa ni posebne umetnosti. `meja` bo število točk, ki jih imamo še na voljo. Ko pade pod 0, je konec veselja.

[23]:
```
def do_nagrade(pot, zemljevid, meja):
    for povezava in zip(pot, pot[1:]):
        if povezava not in zemljevid:
            return povezava[0]
        meja -= vrednost_povezave(povezava, zemljevid)
        if meja < 0:
            return povezava[0]
    return pot[-1]
```

1.4 Za oceno 9-10

V teh dveh funkcijah bomo kolesarja spet omejili tako, da sme vsako veščino uporabiti samo enkrat.

- Napiši funkcijo `sosede(tocka, vescine)`, ki prejme neko točko in množico veščin, ki so mu še na voljo. Funkcija naj vrne seznam parov. Prvi elementi parov so sosednje točke, do

katerih ima kolesar s temi veščinami dostop. Drugi element so veščine, ki mu bodo preostale, če gre v to točko.

Ker se kolesar rad pokaže, naj funkcija ignorira povezave, ki ne zahtevajo nobene veščine. Pot iz P v S ni možna, kot da je ne bi bilo.

Vrstni red elementov v seznamu je lahko poljuben. (Testi ga bodo potem za lažje preverjanje uredili, a to ni tvoja skrb.)

Klic `sosede("I", {"avtocesta", "trava", "lonci", "stopnice", "bolt"})`

vrne seznam

```
[("G", {"trava", "lonci", "stopnice", "bolt"}),
 ("M", {"trava", "lonci", "stopnice", "bolt"}),
 ("E", {"avtocesta", "stopnice", "bolt"}),
]
```

S podanimi veščinami gre lahko v G in M (pri čemer v pripadajoči množici ni več avtoceste) in v E (pri čemer potem ne more več voziti po travi in med lonci. Med sosedi pa ni R, ker ne obvlada (več?) robnikov in P (gravel).

- Napiši funkcijo `dosegljive(tocka, vescine)`, ki vrne množico točk, ki so kolesarju dosegljive iz podane točke s podanimi veščinami, če sme vsako veščino uporabiti samo enkrat, poleg tega pa noče voziti po povezavah, ki ne zahtevajo nobene veščine. Množica vključuje tudi trenutno točko.

  Klic `dosegljive(I, {"avtocesta", "trava", "lonci", "stopnice", "bolt", "gravel", "rodeo"})` vrne `{"I", "G", "E", "P", "M", "N", "K"}`.

  - I: tu začne;
  - G: sem lahko pride ker oblada avtocesto, naprej pa ne more, ker ne obvlada črepinj;
  - E: pozna travo in lonce, iz E pa ne more več nikamor;
  - V točko R ne more, ker ne obvlada robnikov
  - P: sem pride, ker obvlada gravel. V S ne gre, ker mu je pod častjo, v O pa ne more, ker je gravel že uporabil. (Tudi v N ne more iz P, pač pa bo tja prišel iz M);
  - M: pri tem porabi avtocesto, pot pa lahko nadaljuje v
  - N: kjer uporabi rodeo (in iz N še v P, ki pa ga itak doseže že po drugi poti);
  - K: prav tako dosegljiv iz M.

  Namig: iz vsake točke je dosegljiva ta točka in vse točke, ki so dosegljive iz njenih sosed, vendar pri zmanjšanem naboru veščin.

  Ne boj se, nič se ne bo zaciklalo, saj pri vsakem klicu uporabimo kakšno veščino.

1.4.1 Rešitev

`sosede(tocka, vescine)` Tokrat bo potrebno narediti zanko čez pare (ključ, vrednost). Ključi bodo (potencialne) sosede, vrednosti pa potrebujemo zato, da preverimo, ali je soseda dejansko dosegljiva z veščinami, ki so nam na voljo. Zanka bo torej `for (t, soseda), potrebne in zemljevid.items()`.

Kaj moramo zložiti v seznam, ki ga bomo vrnili? Pare (soseda, preostale veščine). Soseda je pač `soseda`, preostale veščine pa so te, ki jih imamo (`vescine`) minus te, ki jih bomo porabili za

sprehod do te sosede (`potrebne`). Ker gre za množice, jih lahko dejansko odštevamo. V seznam torej zlagamo pare `(soseda, vescine - potrebne)`.

Katere "sosede" pa nas zanimajo? Pogoji so trije.

- Zanimajo nas samo tiste povezave v slovarju, ki vodijo iz trenutne točke. Začetna točka povezave (`t`) mora biti torej enaka trenutni točki (`tocka`), torej `t == tocka`.
- Poleg tega naloga pravi, da kolesar ne bo vozil po povezavah, ki ne zahtevajo nobene veščine. Torej mora biti `potrebne` neprazna množica. Prazne množice so neresnične, neprazne pa resnične; pogoje se lahko glasi kar `potrebne`.
- Končno, `potrebne` veščine morajo biti podmnožica veščin, ki so nam na voljo, `potrebne <= vecine`.

Pa imamo funkcijo.

[24]:
```
def sosede(tocka, vescine):
    return [(soseda, vescine - potrebne)
            for (t, soseda), potrebne in zemljevid.items()
            if t == tocka and potrebne and potrebne <= vescine
            ]
```

dosegljive(tocka, vescine) Tole je pa čisto običajna rekurzija. Dosegljiva je trenutna točka, poleg tega pa so dosegljive tudi vse točke, ki so dosegljive iz sosed.

[25]:
```
def dosegljive(tocka, vescine):
    kam = {tocka}
    for soseda, preostale in sosede(tocka, vescine):
        kam |= dosegljive(soseda, preostale)
    return kam
```