Minimalna vpeta drevesa

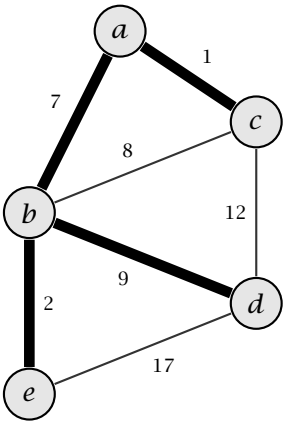
↓↓↓
Sara Veber in Tim Poštuvan

→ **V današnjem svetu so problemi postali kompleksni, zato jih človek ne more več reševati brez uporabe računalnika. Ker računalnik ni čudežna naprava, ki bi sama od sebe znala reševati probleme, mu moramo dati navodila, kako naj se problemov loti. Postopek reševanja se imenuje *algoritem*, tega pa računalniku opišemo s pomočjo različnih programskih jezikov. V članku si bomo ogledali primer iz resničnega življenja, ki ga bomo prevedli na problem iskanja *minimalnega vpetega drevesa*. Ta se navidez zdi težak, vendar se s pravilnim pristopom zelo poenostavi.**

Opis problema

Recimo, da vas najame podjetje, ki se ukvarja z infrastrukturo cestnega omrežja. Zgrajeno imajo cestno omrežje, kjer vsaka cesta povezuje dve mesti in ima določeno ceno vzdrževanja. Ker želijo prihraniti čim več denarja, bi radi nekatere ceste zaprli, a vseeno želeli, da mesta ostanejo med seboj povezana. Želijo torej minimizirati stroške vzdrževanja cest, a hkrati vseeno zagotoviti, da je iz vsakega mesta mogoče priti do vsakega drugega.

Na sliki 1 je primer majhnega cestnega omrežja. Mesta so označena z majhnimi krogi, ki so poimenovani s črkami od a do e. Ceste so črte, ki povezujejo pare mest, zraven njih pa so napisane številke, ki predstavljajo ceno vzdrževanja ceste. Tako sta mesti a in b povezani s cesto, vzdrževanje te ceste pa stane 7. Omrežje je povezano, saj lahko iz vsakega mesta pridemo do vsakega drugega. Od mesta a do mesta d lahko pridemo npr. do c, saj obstaja cesta med a in c, ter cesta med c in d.



SLIKA 1.
Primer cestnega omrežja

S takim opisom problema ni nič narobe, vendar vsebuje ogromno podatkov, ki za reševanje niso ključnega pomena. V resnici nam je čisto vseeno, ali imamo opravka z mesti ali letališči. Zaradi tega problem raje modelirajmo s pomočjo matematičnih struktur. Videli bomo, da je za naš problem najbolj primerna teorija grafov. Tako bomo ohranili samo tiste podatke, ki so res ključni za reševanje. Obstaja že veliko algoritmov, ki jih lahko uporabimo, če primerno modeliramo problem.

Matematični model problema

Pri modeliranju problema se bomo oprli na eno izmed vej matematike – teorijo grafov. Prej smo problem opisali s pomočjo mest in cest med njimi. V jeziku teorije grafov mestom rečemo vozlišča (angl. nodes), cestam pa povezave (angl. edges). Te povezave so lahko utežene ali pa ne. V našem primeru so utežene, saj ima vsaka cesta svojo ceno vzdrževanja. Strukturo, ki zajema vozlišča in povezave, imenujemo *graf*. Graf $G = (V, E)$ je množica vozlišč V

in povezav E. E je množica množic velikosti 2, kjer vsaka taka množica predstavlja eno povezavo. V našem primeru imamo uteženo graf, zato moramo imeti še dodatno funkcijo w, ki slika iz množice povezav v realna števila. Graf zgornjega omrežja bi bil videti tako

- $V = \{a, b, c, d, e\}$
  $E = \{\{a, b\}, \{a, c\}, \{b, c\}, \{b, d\}, \{d, c\}, \{e, b\}, \{d, e\}\}$

- $w : E \to \mathbb{R}$
  $\{a, b\} \mapsto 7$
  $\{a, c\} \mapsto 1$
  $\{b, c\} \mapsto 8$
  $\{b, d\} \mapsto 9$
  $\{d, c\} \mapsto 12$
  $\{e, b\} \mapsto 2$
  $\{d, e\} \mapsto 17$

Povezava $\{a, b\}$ predstavlja cesto med a in b, funkcija w pa tej povezavi priredi vrednost 7, kar je njena cena vzdrževanja.

Predstavitev omrežja smo si že pogledali, da pa ugotovimo, kaj je rešitev problema v matematičnem jeziku, moramo spoznati še pojme *vpetega podgrafa*, *drevesa* in *minimalnega vpetega drevesa*. Vpet podgraf $H = (V', E')$ grafa G dobimo tako, da odstranimo nekaj povezav grafa G, a vseeno ohranimo vsa vozlišča. Torej $V' = V$, saj ohranimo vsa vozlišča, vendar $E' \subseteq E$, ker odstranimo nekaj povezav. Primer vpetega podgrafa našega zgornjega grafa:

- $V = \{a, b, c, d, e\}$
  $E = \{\{b, c\}, \{b, d\}, \{d, c\}, \{e, b\}, \{d, e\}\}$.

Vidimo, da ima vpet podgraf enako množico mest, vendar drugačno množico cest. V tem vpetem podgrafu ni ceste med a in b ter ceste med a in c, torej je množica cest podmnožica cest osnovnega grafa.

Naslednja izredno pomembna struktura v teoriji grafov je drevo. Drevo je definirano kot povezani graf brez ciklov. Povezanost grafa pomeni, da lahko preko povezav pridemo iz vsakega vozlišča do vsakega drugega. Cikel pa je podmnožica vozlišč, ki so povezana v krog. Torej, če se začnemo sprehajati iz poljubnega vozlišča na ciklu in obiskujemo samo vozlišča na tem ciklu, a ne uporabimo iste povezave večkrat, se lahko vrnemo v začetno vozlišče. Primer drevesa je na sliki 1 označen odebeljeno.

- $V = \{a, b, c, d, e\}$
  $E = \{\{a, b\}, \{a, c\}, \{b, d\}, \{e, b\}\}$.

Vozlišča so kljub temu, da nimamo več vseh povezav, še vedno povezana. Zelo pomembna lastnost dreves je, da je med vsakim parom vozlišč natanko ena pot. Poleg tega ima drevo z n vozlišči natanko $n - 1$ povezav. Obe lastnosti sta razvidni iz primera.

Iz prejšnjih dveh definicij lahko sedaj sestavimo definicijo vpetega drevesa. To je vpet graf, ki je hkrati drevo. Nas pa bo zanimala prav posebna vrsta vpetih dreves – minimalna vpeta drevesa. Minimalna vpeta drevesa so taka vpeta drevesa, ki imajo najmanjši možen seštevek uteži. Na zgornji sliki je odebeljeno minimalno vpeto drevo, ki ima seštevek povezav enak 19. Z nekaj poskušanja se lahko prepričate, da v tem grafu ne obstaja vpeto drevo z manjšim seštevkom uteži.

Z uporabo teorije grafov smo problem prevedli na iskanje minimalnega vpetega drevesa v uteženem grafu. Hkrati smo dobili tudi trivialni algoritem za iskanje minimalnih vpetih dreves. Poiščemo vsa vpeta drevesa, seštejemo uteži povezav in vzamemo tistega, ki ima najmanjši seštevek uteži.

V grafu je lahko tudi eksponentno mnogo vpetih dreves glede na število vozlišč, zato je trenutni algoritem izredno neučinkovit. K sreči obstaja Kruskalov algoritem, ki hitro najde minimalno vpeto drevo. V nadaljevanju si ga bomo podrobneje ogledali, vendar moramo prej spoznati še pomembno *podatkovno strukturo*, ki jo uporablja. Za nas bo ključna podatkovna struktura *disjunktnih množic*.

Disjunktne množice

Učinkovitost algoritma je velikokrat odvisna od učinkovite podatkovne strukture. Dober primer je ravno Kruskalov algoritem, čigar učinkovitost je odvisna predvsem od podatkovne strukture disjunktnih množic. Disjunktne množice (angl. disjoint-set data structure) so podatkovna struktura, ki hrani *povezane komponente grafa*. Povezana komponenta grafa je podmnožica vozlišč, kjer je vsak par vozlišč iz te množice med seboj dostopen preko povezav. Podatkovna struktura se imenuje disjunktne množice,

ker so povezane komponente te strukture med seboj disjunktne. To pomeni, da je vsako vozlišče v največ eni povezani komponenti. S pomočjo disjunktnih množic znamo izredno hitro povezovati komponente grafa in ugotoviti, ali je par vozlišč povezan. V nadaljevanju si bomo podrobneje pogledali ti dve operaciji.

Podatkovna struktura si za vsako vozlišče hrani svojega starša – prvo vozlišče nad njim. Če vozlišče nima starša, je koren. Ko implementiramo podatkovno strukturo, si za starša takega vozlišča shranimo kar to isto vozliče. Funkcija `initialize` ustvari začetno stanje komponent, kjer je vsako vozlišče svoja povezana komponenta. V skladu s prej povedanim je starš vsakega vozlišča kar vozlišče samo.

```
def initialize(n):
    parent = [0 for i in range(n)]
    for i in range(n):
        parent[i] = i
    return parent
```

Sedaj si poglejmo, kako ugotoviti, če sta dve vozlišči v isti povezani komponenti. Opazimo lahko, da ima vsaka povezana komponenta natanko en koren. Ta pa je za vsa vozlišča znotraj iste povezane komponente enak. Torej sta dve vozlišči v isti povezani komponenti, če imata enak koren. Funkcija `find` najde koren vozlišča x, pri čemer mora imeti podano tabelo, ki smo jo dobili pri `initialize`. To naredi tako, da rekurzivno išče starša svojega starša, dokler ne pride do korena. V implementaciji uporabimo še kompresijo poti (angl. path compression), ki pospeši iskanje korena. Med iskanjem vsa vozlišča na poti do korena prevezuje direktno na koren. To pospeši iskanje, saj je hitrost odvisna od števila vozlišč na poti do korena.

```
def find(parent, x):
    # vozlišče je koren
    if parent[x] == x:
        return x
    # starša nastavimo na koren komponente
    parent[x] = find(parent[x])
    return parent[x]
```

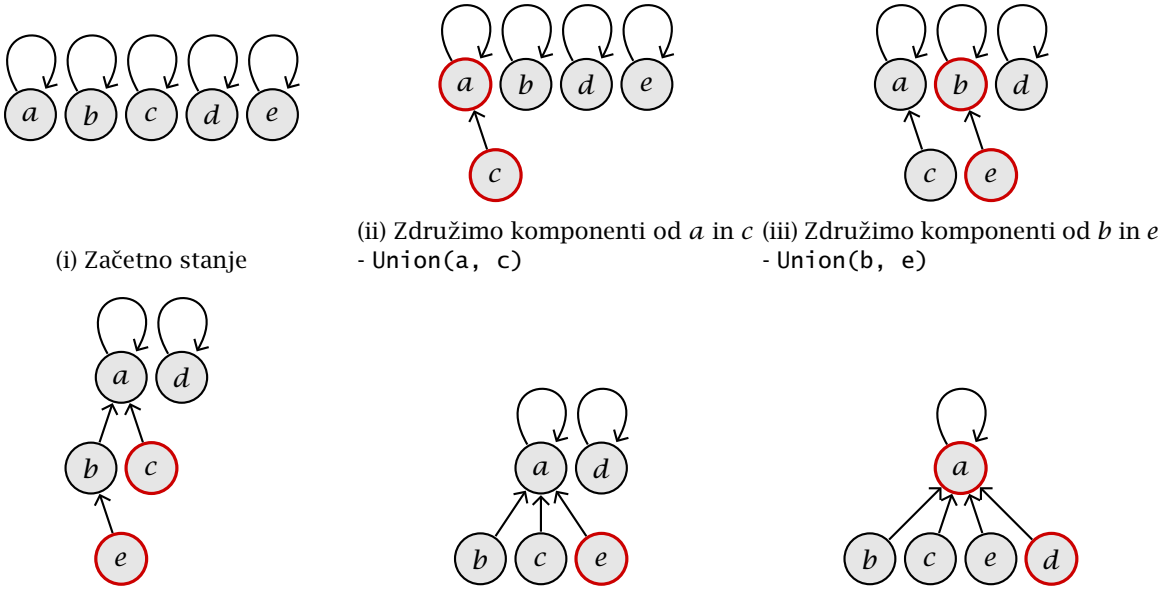
Zadnja operacija, ki jo podatkovna struktura premore, je združevanje povezanih komponent. To naredi tako, da starša korena ene izmed obeh komponent nastavi na koren druge komponente. To posledično spremeni tudi koren vseh vozlišč v tisti komponenti. Funkcija `union` naredi točno to, samo da pred združevanjem preveri, če sta vozlišči slučajno že v isti povezani komponenti. Če je temu tako, združevanje nima smisla, zato funkcija ne stori ničesar.

```
def union(parent, x, y):
    # korena komponent od x in y
    root_x = find(parent, x)
    root_y = find(parent, y)
    if root_x != root_y:
        parent[root_x] = root_y
```

Za boljše razumevanje si poglejmo še primer na sliki 2. Na (i) vidimo začetno stanje, kjer je vsako vozlišče svoja povezana komponenta. Pri (ii) povežemo komponenti od a in c. Koren komponente od a je na začetku kar a, koren komponente od c pa c. Ker želimo komponenti povezati, nastavimo starša od c na a. Pri (iii) naredimo isto, vendar z b in e. Korak (iv) je podoben (ii). Koren komponente od c je a, koren komponente od e pa b. Nato nastavimo starša od b na a. Slika (v) ilustrira kompresijo poti. Ko iščemo koren komponente od e, starše vozlišč, ki jih sreča na poti do korena, nastavimo na koren. V tem primeru nastavi starša od b in e na koren. Korak (vi) je spet podoben (ii), le da povezujemo komponenti od a in d.

Kruskalov algoritem

Sedaj, ko smo spoznali vsa potrebna orodja, se lahko osredotočimo na iskanje minimalnih vpetih dreves. Za iskanje minimalnih vpetih dreves bomo uporabili Kruskalov algoritem. Algoritem je leta 1956 razvil Joseph Kruskal. Spada med požrešne algoritme, saj na vsakem koraku izbere najboljšo izmed možnosti, a je kljub temu končna rešitev optimalna. Algoritem poteka v naslednjih korakih. Na začetku uredi povezave glede na uteži, tako da so povezave z manjšo utežjo pred tistimi z večjo. V tem vrstnem redu potem obravnava povezave. Recimo, da ima trenutna povezava krajišči v x in y. Če sta x in y v različnih povezanih komponentah, povezavo doda. Tako x in y pristaneta v isti povezani komponenti. Če sta x in y v isti komponenti, potem bi dodajanje te povezave povzročilo cikel. To se seveda ne sme zgoditi,



(i) Začetno stanje

(ii) Združimo komponenti od a in c - `Union(a, c)`

(iii) Združimo komponenti od b in e - `Union(b, e)`

(iv) Združimo komponenti od c in e - `Union(c, e)`

(v) Iščemo koren komponente od e - `Find(e)`

(vi) Združimo komponenti od a in d - `Union(a, d)`

SLIKA 2.
Primer zaporedja ukazov na disjunktnih množicah

ker drevo ne sme imeti ciklov. To je razumljivo tudi s stališča osnovnega problema, saj to pomeni, da bi ohranili še eno cesto, čeprav je ne potrebujemo. S tem bi se po nepotrebnem povečali stroški vzdrževanja. Povezav torej ne smemo dodajati tako, da bi delali cikle. Vendar to še vedno ne pomeni, da bomo s takim postopkom prišli do optimalne rešitve. Zakaj pa ne bi raje vzeli trenutne povezave in odstranili eno izmed ostalih na ciklu. S tem bi se prav tako izognili ciklu? Odgovor je, da so vrednosti uteži na vseh prej dodanih povezavah manjše od trenutne, zato bi z dodajanjem te povezave samo povečali vrednost končnega vpetega drevesa. To je intuitivni dokaz pravilnosti algoritma. Z dodajanjem povezav nadaljujemo, dokler nam ne ostane samo ena povezana komponenta. Ta komponenta je drevo in ima minimalen seštevek uteži, zato je minimalno vpeto drevo. Za združevanje in preverjanje pripadnosti komponent uporabimo prej omenjene disjunktne množice.

Desno je koda Kruskalovega algoritma v Pythonu. Vozlišča (a, b, ...) bomo označili s številom (0, 1, ...), saj je bolj priročno za implementacijo.

```
a, b, c, d, e = (0, 1, 2, 3, 4)
# Povezave oblike (cena, prvo vozlišče, drugo
   vozlišče)
edges = [(7, a, b), (1, a, c), (8, c, b), (9, b, d),
   (12, c, d), (2, b, e), (17, e, d)]

# Uredimo povezave naraščajoče, sortira se najprej
    po prvi komponenti
edges.sort()
parents = initialize(5)
minimum_spanning_tree = 0.0
for edge in edges:
   # Razpakiramo podatke o povezavi: vozlišči x in y
    s ceno cost
   cost, x, y = edge
   # Korena komponent od x in y
   root_x = find(parents, x)
   root_y = find(parents, y)
   # Če sta korena različna, dodamo povezavo
   if root_x != root_y:
      union(parents, x, y)
      minimum_spanning_tree += cost
```

Oglejmo si še potek algoritma na grafu iz slike 1. Na sliki 3 so prikazani posamezni koraki. Z rdečo je pobarvana trenutna povezava, z odebeljeno črno pa že dodane povezave. Najmanjšo utež ima povezava a–c. Ker sta a in c v različnih povezanih komponentah, jo dodamo v graf. Trenutna cena vpetega podgrafa je 1. Pri dodajanju povezave v korakih (ii) in (iii) ne ustvarimo cikla, zato obe povezavi dodamo v minimalno vpeto drevo. Korak (iv) je prvi, pri katerem povezave ne vzamemo. Lahko vidimo, da povezave $a - -b$, $a - -c$ in $b - -c$ tvorijo cikel, zato $b - -c$ ne smemo dodati. Na sliki se vidi tudi, da sta uteži obeh povezav na ciklu manjši, zato trenutne povezave ni vredno zamenjati z že obstoječo (sklep zgoraj). V koraku (v) povezavo spet vzamemo in dobimo minimalno vpeto drevo s ceno 19, prikazano na sliki (vi).

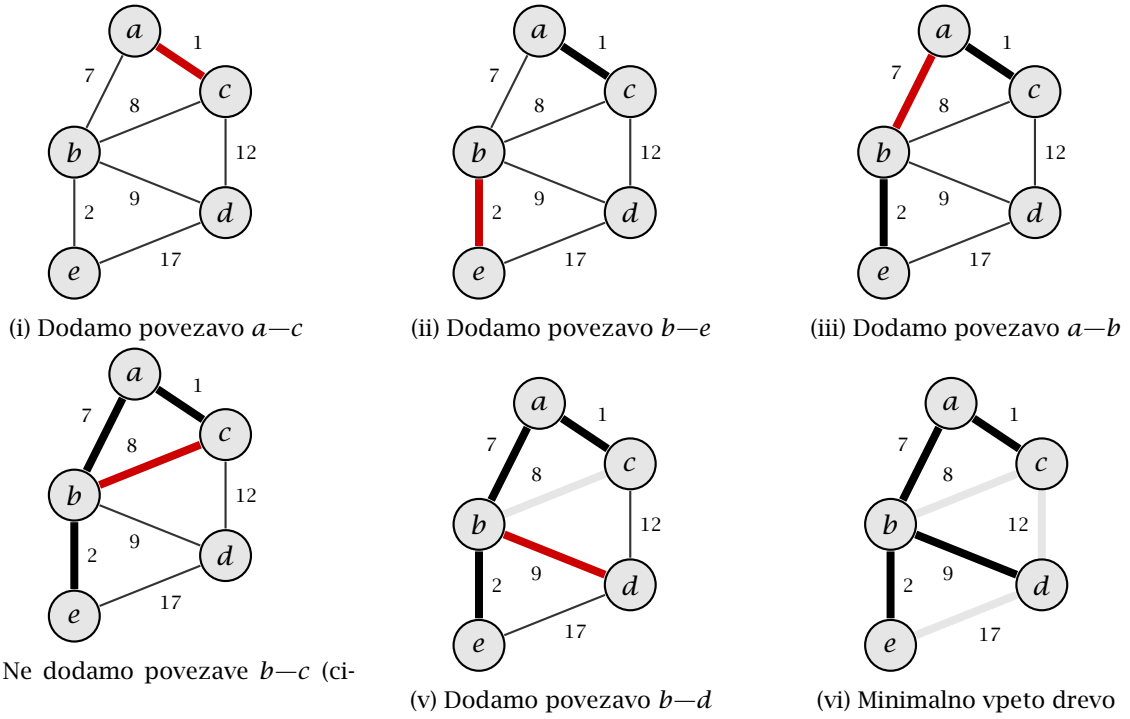
Literatura

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest in C. Stein, *Introduction to Algorithms*, The MIT Press, 3rd Ed., 2009.

[2] S. Halim *Competitive Programming 3*, 3rd Ed., 2013.

[3] *Kruskal's algorithm*, dostopno na `en.wikipedia.org/wiki/Kruskal\%27s_algorithm`, ogled 8. 4. 2020.

[4] *Disjoint-set data structure*, dostopno na `en.wikipedia.org/wiki/Disjoint-set_data_structure`, ogled 8. 4. 2020.



(i) Dodamo povezavo a—c

(ii) Dodamo povezavo b—e

(iii) Dodamo povezavo a—b

(iv) Ne dodamo povezave b—c (cikel)

(v) Dodamo povezavo b—d

(vi) Minimalno vpeto drevo

SLIKA 3.
Iskanje minimalnega vpetega drevesa za začetni graf