UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO

Matematika – 1. stopnja

Marcel Čampa

Algoritem potisni-povišaj za iskanje maksimalnih pretokov

Delo diplomskega seminarja

Mentor: prof. dr. Sergio Cabello

Ljubljana, 2018

Kazalo

Algoritem potisni-povišaj za iskanje maksimalnih pretokov

Povzetek

Spoznali smo se s pojmi iz teorije grafov, kot so omrežje, pretok in maksimalen pretok. Predstavili smo algoritem potisni-povišaj za iskanje maksimalnega pretoka v omrežju. Algoritem smo tudi implementirali in pokazali njegovo časovno zahtevnost ter pravilnost delovanja. Pokazali smo tudi, kako se iskanje maksimalnih pretokov uporablja v vsakdanjem življenju.

Push-relabel algorithm for maximum flow problem

Abstract

We learned essential concepts of graph theory, e.g. network, flow and maximum flow. We explained the push-relabel algorithm for finding maximum flow through a network. We also showed the time complexity of push-relabel algorithm and proved its correctness. We also showed why one would want to be searching for maximum flows in real life, which we did by providing two examples.

Math. Subj. Class. (2010): 05C85, 90B10, 90C35, 90C27

Ključne besede: pretok, potisni-povišaj, graf, omrežje, maksimalen pretok, optimizacija

Keywords: flow, push-relabel, graph, network, maximum flow, optimization

1. Uvod

Tema mojega diplomskega dela je algoritem potisni-povišaj, s katerim iščemo maksimalen pretok v danem pretočnem omrežju. V angleščini rečemo temu algoritmu *push-relabel algorithm*, zato se marsikomu morda porodi vprašanje, zakaj prevod *relabel* v *povišaj*. Za ta prevod sem se odločil zato, ker takrat, ko opravimo tako imenovano operacijo *relabel*, vozlišče postavimo na večjo višino.

Cilj tega dela je spoznati se z osnovnimi koncepti teorije grafov in kombinatorične optimizacije, predvsem pa se od bližje spoznati z enim od učinkovitejših algorimov za iskanje maksimalnega pretoka. Poleg našega algoritma sodijo med bolj znane še algoritem Forda in Fulkersona, Edmonds-Karpov algoritem, Dinicev algoritem, KRT (King, Rao, Tarjan) in KRT+O (KRT + Orlin). Prva dva algoritma imata slabšo časovno zahtevnost od algoritma potisni-povišaj, Dinicev ima enako kot osnovna verzija potisni-povišaj, zadnja dva pa sta hitrejša.

Ko sem se udomačil v teoriji algoritma, sem se lotil implementacije v programskem jeziku C++, ki sem ga izbral povsem arbitrarno. Največjo vlogo pri izbiri sta sicer igrali hitrost in praktičnost, vendar sem se za C++ odločil, ker v njem že dolgo časa nisem napisal ničesar. Med pisanjem algoritma sem pravzaprav čisto sam ,,odkril" izboljšavo algoritma, ki izboljša časovno zahtevnost navadnega potisni-povišaj z $\mathcal{O}(V^2E)$ na $\mathcal{O}(V^3)$, kjer je V število vozlišč v omrežju, E pa število povezav.

Sedaj, ko je bil algoritem implementiran in stestiran, sem kolebal med tem, ali naj algoritem poženem na primerih različne velikosti in ugotovitve vključim v to delo, ali pa naj raje predstavim, kako nam lahko maksimalni pretoki pomagajo v vsakdanjem življenju. Vsekakor se mi je zdela bolj poučna slednja stvar, zato sem se zanjo tudi odločil. Navsezadnje je tudi zanimivejša kakor suhoparno nabijanje številk v tabelo, ki je sama sebi namen.

2. Osnovne definicije

V tem razdelku se bomo najprej spoznali z osnovnimi definicijami teorije grafov, brez katerih ne bomo mogli. Nato si bomo pogledali manj znane definicije in definicije specifične za algoritme tipa *potisni-povišaj*.

Definicija 2.1. Graf G je par množic $G = (V, E)$, kjer je V množica vozlišč grafa, E pa je množica povezav grafa G.

Definicija 2.2. Naj bo $G = (V, E)$ graf. **Omrežje** na grafu G je par (G, c), kjer je $c\colon V \times V \to \mathbb{R}_+ \cup \{\infty\}$ **funkcija prepustnosti**, ki vsaki povezavi (u, v) priredi njeno prepustnost $c(u, v)$. Prepustnost $c(u, v) = \infty$ natanko tedaj, ko prepustnost povezave ni omejena.

Rekli bomo še, da $c(u, v) = 0$ natanko tedaj, ko povezava ne obstaja.

Definicija 2.3. Naj bo $G = (V, E)$ graf in (G, c) omrežje na grafu G. **Pretočno omrežje** na omrežju (G, c) je četverica (G, c, s, t), kjer je $s \in V$ začetno vozlišče pretočnega omrežja, rečemo mu **izvir**, $t \in V$ pa končno vozlišče pretočnega omrežja, ki mu pravimo **ponor**.

Definicija 2.4. Psevdopretok je funkcija $f\colon V \times V \to \mathbb{R}$, ki zadošča pogojema:

(1) Za vsaki vozlišči $u, v \in V$ velja $f(u, v) = -f(v, u)$.
(2) Za vsaki vozlišči $u, v \in V$ velja $f(u, v) \leq c(u, v)$, kjer je c funkcija prepustnosti.

Definicija 2.5. Residualna prepustnost povezave glede na trenuten psevdopretok f je funkcija $c_f \colon V \times V \to \mathbb{R}_+$, definirana kot razlika prepustnosti povezave in trenutnega toka preko nje. Velja torej $c_f(u, v) = c(u, v) - f(u, v)$.

Definicija 2.6. Naj bo $G = (V, E)$ pretočno omrežje in f tok v tem omrežju. **Residualno omrežje** omrežja (G, c, s, t), inducirano s tokom f je $G_f = (V, E_f)$, kjer je

$$E_f = \{(u, v) \in V \times V \mid |c_f(u, v)| > 0\}.$$

Opomba 2.7. Opazimo dve stvari. O grafu G smo govorili kot o pretočnem omrežju. To storimo takrat, ko nas c, s in t ne zanimajo, ali pa je iz konteksta razvidno, kaj so. Druga stvar pa je, da smo uporabili besedo *tok*, brez predhodne definicije le-tega. Za sedaj si lahko mislimo, da je to psevdopretok, kar je povsem smiselno v kontekstu algoritma potisni-povišaj.

Definicija 2.8. Funkcija presežka za psevdopretok f je funkcija $e_f \colon V \to \mathbb{R}$, definirana z $e_f(u) = \sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v)$. Če je $e_f(u) > 0$, pravimo, da je u **v presežku**.

Z drugimi besedami bi lahko rekli, da funkcija e_f za vsako vozlišče pove, koliko preveč toka je vanj priteklo. To je ravno razlika med vsoto pritečenih tokov in vsoto odtečenih tokov.

Definicija 2.9. Predpretok f je tak psevdopretok, v katerem za vsak $v \in V \setminus \{s\}$ velja, da je neto tok, ki priteče v vozlišče v, nenegativen, torej da velja $e_f(v) \geq 0$.

Algoritmi tipa potisni-povišaj namreč ne ohranjajo Kirchoffovega zakona, ki velja za pretok. Zato pri algoritmih tega tipa govorimo o predpretoku. Ker ne ohranjajo Kirchoffovih zakonov, definiramo naslednjo funkcijo.

Definicija 2.10. Pretok f je tak psevdopretok, v katerem za vsak $v \in V \setminus \{s, t\}$ velja, da je neto tok, ki priteče v vozlišče v, enak nič, torej da velja $e_f(v) = 0$.

Definicija 2.11. Vrednost pretoka f je tok, ki vstopa v ponor t. Označimo ga z $|f|$. Velja torej $|f| = e_f(t)$.

Definicija 2.12. Maksimalni pretok je pretok f, za katerega velja

$$|f| = \max_{f_i} |f_i|.$$

Naslednja definicija nam bo dala nov atribut vozlišč.

Definicija 2.13. Naj bo $G = (V, E, s, t)$ pretočno omrežje. **Višinska funkcija** je funkcija $h \colon V \to \mathbb{N}_0$, za katero velja

(1) $h(s) = |V|$ in $h(t) = 0$,
(2) $h(u) \leq h(v) + 1$, za vsako povezavo $(u, v) \in E_f$, kjer je E_f množica povezav v residualnem grafu G_f.

3. Iskanje maksimalnega pretoka z algoritmom potisni-povišaj

V tem razdelku si bomo podrobneje ogledali algoritem *potisni-povišaj*. Začeli bomo s kratkim opisom delovanja algoritma in intuitivno razložili, kako algoritem deluje. Nato si bomo pogledali psevdokodo algoritma in se z njo pobližje spoznali

na zgledu. Sledila bo implementacija algoritma z rahlo izboljšavo v programskem jeziku C++. Pokazali bomo pravilnost delovanja algoritma in njene implementacije ter časovno zahtevnost algoritma.

3.1. **O algoritmu.** Algoritem potisni-povišaj deluje po preprostem principu iz narave. Predsavljajmo si, da imamo rečno omrežje, ki se začne v eni točki in konča v eni točki. Z drugimi besedami imamo eno reko, ki pa se vmes poljubno deli in združuje. Seveda je na zemlji prisotna gravitacijska sila, ki povzroči, da voda teče od višje točke proti nižji, recimo od izvira v hribih do ponora v morje, vmes pa ubira tako pot, da nikjer ne gre navzgor. V jeziku grafov lahko predstavimo omenjeni pojav na naslednji način. Tam, kjer se reka deli oziroma združi, postavimo vozlišče grafa. Del reke med dvema razvejiščema predstavlja povezavo med razvejiščema pripadajočima vozliščema. Izvir in ponor reke predstavljata vozlišči s in t. Vsakemu vozlišču pripišemo višino, na kateri se nahaja, in količino vode, ki je vanj pritekla in odtekla. Seveda se v naravi ne zgodi (razen v primeru neurij), da bi v razvejišče priteklo več vode, kot pa je je iz njega odteklo. Prav tako ne more priteči manj vode, kot je odteče.

Sedaj, ko smo se spomnili, kako deluje mati narava, in to prevedli v matematični jezik, si podrobneje poglejmo, kako deluje algoritem potisni-povišaj. Začnemo z omrežjem (od sedaj bomo rajši kot o grafih govorili o omrežjih) $G = (V, E, s, t)$ in funkcijama $c\colon V \times V \to \mathbb{N}$, ki vsaki povezavi priredi njeno kapaciteto, in $f\colon V \times V \to \mathbb{N}$, ki za vsako povezavo pove, koliko vode teče v nekem trenutku preko nje. Vozliščem $v \in V$ priredimo še funkciji $h\colon V \to \mathbb{N}_0$, ki določa višino vozlišča, in $e\colon V \to \mathbb{N}_0$, ki pove, koliko preveč vode je priteklo v neko vozlišče. Seveda velja

$$e(u) = \sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v).$$

Algoritem na začetku nastavi višino vseh vozlišč razen vozlišča s na nič in višino s na $|V|$. Tako na začetku velja $h(s) = |V|$ in $h(u) = 0$, $u \in V \setminus \{s\}$. Nato potisnemo iz s tok v sosednja vozlišča tako, da zasičimo povezave, torej da velja $f(s, v) = c(s, v)$, za vse $v \in V$, za katere velja $(s, v) \in E$. Poleg tega dodamo v residualno omrežje še obratne povezave, katerim nastavimo $f(v, s) = -f(s, v)$, da zadostimo pogoju (1) iz definicije psevdopretoka (definicija 2.4). S tem smo ustvarili tako imenovani *predpretok*. To smo lahko storili, ker je višina vozlišča s večja kot višina sosednjih vozlišč v, saj $h(s) = |V| > 0 = h(v)$.

Rezultat te operacije je, da se je v vozliščih, sosednjih vozlišču s, nabrala odvečna voda. Za ta vozlišča torej velja $e(v) > 0$. Sedaj lahko potisnemo vodo iz teh vozlišč naprej, saj je vode v njih preveč, želimo pa, da je odteče toliko, kot je priteče. Vendar tega ne moremo storiti, saj so višine sosednjih vozlišč prav tako enake nič. Zato si izberemo neko vozlišče u, v katerega je priteklo preveč vode, in mu povečamo višino na $\min\{h(v) : (u, v) \in E_f\} + 1$. S tem smo omogočili, da bo voda odtekla v vsaj eno izmed vozlišč. Ta postopek ponavljamo, dokler lahko potisnemo tok v omrežju ali pa povišamo neko vozlišče. Tok, ki na koncu priteče v t, je enak maksimalnemu pretoku omrežja, kar bomo pokazali kasneje.

Oglejmo si sedaj psevdokodo algoritma. Spoznali smo, da je algoritem sestavljen iz dveh osnovnih operacij, *potiskanja* in *povišanja*, zato se posvetimo tema operacijama. Začnimo s potiskanjem.

```
POTISNI (u, v)
1   // Potisnemo lahko, če je e(u) > 0,
2   // c(u,v) > 0 in h(u) = h(v) + 1.
3   delta = min{ e(u), c(u,v) - f(u,v) }
4   ČE (u,v) v E, POTEM
5       f(u,v) += delta
6   DRUGAČE f(v,u) -= delta
7   e(u) -= delta
8   e(v) += delta
```

To operacijo lahko storimo, če ima u presežek toka, torej, če velja $e(u) > 0$, če je kapaciteta povezave $c(u, v) > 0$ in če sta vozlišči u in v na primernih višinah, torej če velja $h(u) = h(v) + 1$. Najprej izračunamo, kolikšno količino δ lahko potisnemo. Ta je enaka minimumu med presežkom toka v vozlišču u in residualno kapaciteto povezave, ki je enaka $c(u, v) - f(u, v)$. To storimo v vrstici 3. V vrsticah 4–6 potisnemo tok δ po povezavi (u, v), če ta povezava obstaja. V nasprotnem primeru potisnemo $-\delta$ po obratni (residualni) povezavi. V vrsticah 7 in 8 posodobimo še presežek toka v krajiščih povezave. To storimo tako, da v začetnem vozlišču presežek zmanjšamo za tok, ki smo ga potisnili, v končnem vozlišču pa presežek povečamo.

Nadaljujmo z operacijo povišanja vozlišča.

```
POVIŠAJ (u)
1   // Vozlišče u povišamo, če je e(u) > 0 in
2   // za vsak v iz V, (u,v) v E_f, velja h(u) <= h(v).
3   h(u) = min{ h(v) : (u,v) v E_f } + 1
```

Operacija povišanja vozlišča u je precej enostavna. Storimo jo takrat, ko ima vozlišče u presežek toka, torej velja $e(u) > 0$, a hkrati ne moremo potisniti toka v sosednja vozlišča, saj so vsa na večji ali enaki višini kot u.

V opisu algoritma smo navedli še *inicializacijo predpretoka*. Zapišimo psevdokodo za to operacijo.

```
INICIALIZIRAJ_PREDPRETOK(G,s)
 1   // V grafu G si izberemo vozlišče s
 2   // in inicializiramo predpretok.
 3   ZA vsak v v V(G)
 4       h(v) = 0
 5       e(v) = 0
 6   ZA vsak (u,v) v E(G)
 7       f(u,v) = 0
 8   h(s) = |V|
 9   ZA vsak v, za katerega obstaja (s,v) v E(G)
10       f(s,v) = c(s,v)
11       e(v) = f(s,v)
```

Kot smo dejali, zgornja operacija nastavi višine vozlišč, tok po povezavah in presežek toka v vozliščih na nič. To storimo v vrsticah 3–7. V vrstici 8 nato nastavimo višino vozlišča s na število vseh vozlišč v omrežju, torej $h(s) = |V|$. V vrsticah

9 in 10 nato potisnemo tok prek vseh povezav, ki izhajajo iz s in v vrstici 11 popravimo še presežek toka v vozliščih, ki so sosednja s. Opazimo, da nismo odšteli presežka toka v vozlišču na začetku povezave, torej v vozlišču s. Tega nismo storili, ker je to nepotrebno; predstavljamo si namreč, da je v vozlišču s lahko poljubna količina vode, več kot je potrebujemo, več je lahko dobimo. Kasneje bomo videli, kaj se zgodi, če smo v inicializaciji predpretoka poslali preveč vode, kot je omrežje lahko spusti skozi.

Čas je, da navedemo še glavni del algoritma, torej ,,program", ki uporablja zgoraj navedene operacije. Psevdokoda je na videz precej preprosta.

```
POTISNI-POVIŠAJ(G,s)
1   INICIALIZIRAJ_PREDPRETOK(G,s)
2   DOKLER obstaja mogoča operacija POTISNI ali POVIŠAJ
3       izvedi mogočo operacijo
```

Na videz nedolžna, vendar skriva rahlo prepreko do povsem direktne implementacije. Vprašanje, ki se pojavi, je namreč, kako vedeti, ali lahko potisnemo oziroma povišamo. Oglejmo si zgled delovanja algoritma in sproti se nam morda porodi ideja.

3.2. **Primer delovanja algoritma.** Vzemimo preprosto omrežje na petih vozliščih. Pet vozlišč sem izbral zato, ker je to najmanjše število vozlišč, pri katerem se lahko dogajajo zanimive reči. Naj velja $G = (V, E, s, t)$, kjer je

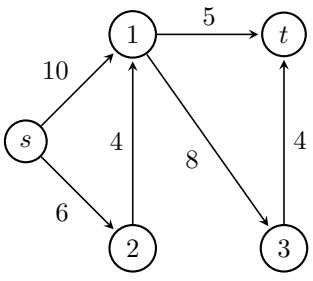
$$\begin{aligned} V &= \{0, 1, 2, 3, 4\}, \\ E &= \{(0,1), (0,2), (2,1), (1,3), (1,4), (3,4)\}, \\ s &= 0, \\ t &= 4. \end{aligned}$$

Kapacitete povezav so podane v naslednji tabeli.

Tabela 1. Kapacitete povezav omrežja G.

	$(0,1)$	$(0,2)$	$(2,1)$	$(1,3)$	$(1,4)$	$(3,4)$
$c(u,v)$	10	6	4	8	5	4

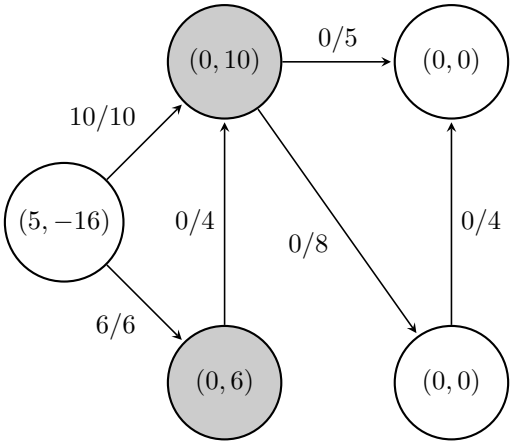
Omrežje G na začetku izgleda takole:



Slika 1. Pretočno omrežje, ki ga bomo uporabili kot zgled delovanja algoritma potisni-povišaj.

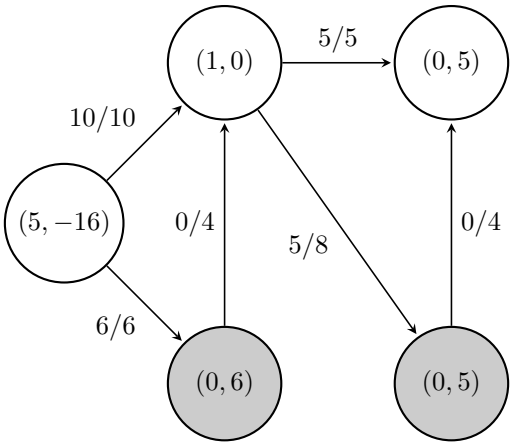
Poiščimo sedaj maksimalni pretok skozi to omrežje s pomočjo zgoraj opisanega algoritma potisni-povišaj. Zaradi večje preglednosti residualnih povezav ne bomo risali, privzemimo pa, da če na povezavi (u, v) teče tok $f(u, v)$, obstaja residualna povezava (v, u), po kateri teče tok $f(v, u) = -f(u, v)$.

Najprej inicializiramo predpretok. Višino in presežek toka v vseh vozliščih nastavimo na 0 in vsaki povezavi damo tok 0. Nato višino vozlišča s nastavimo na $|V| = 5$ in vsaki povezavi, ki se začne v s, torej povezavi $(0, 1)$ in $(0, 2)$, zasičimo. Na koncu še posodobimo presežek toka v vozliščih, sosednjim vozlišču s. Omrežje sedaj izgleda tako:



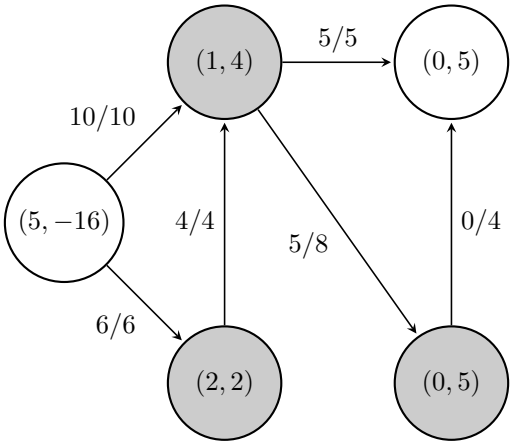
Slika 2. Pretočno omrežje po inicializaciji predpretoka. Par (h, e) v vozlišču u določa trenutno višino vozlišča $h(u)$ ter njegov presežek toka $e(u)$. Sivo obarvani sta vozlišči v presežku. Par f/c na povezavi določa, kolikšen tok f teče po tej povezavi in kolikšna je prepustnost povezave (c).

Sedaj imamo dve vozlišči v presežku, vendar iz nobenega ne moremo potisniti toka. Zato povišamo vozlišče 1 na $\min\{h(3), h(4)\} + 1 = \min\{0, 0\} + 1 = 1$. Sedaj lahko iz vozlišča 1 tok potisnemo. Po povezavi $(1, 4)$ lahko pošljemo $\delta = \min\{e(1), c(1, 4) - f(1, 4)\} = c(1, 4) - f(1, 4) = 5 - 0 = 5$ enot in s tem zasičimo povezavo. V vozlišču nam ostane še $e(1) = 5$. Sedaj pošljemo po povezavi $(1, 3)$ $\delta = \min\{e(1), c(1, 3) - f(1, 3)\} = 5$ enot. Posodobimo f in e in dobimo naslednje stanje.



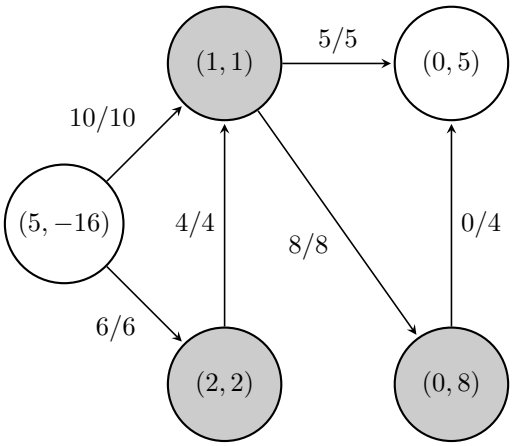
Slika 3. Opravili smo povišanje vozlišča 1 in potistnili tok iz njega.

Sedaj vozlišče 1 ni več v presežku. Naslednje vozlišče, iz katerega lahko pošljemo tok, je vozlišče 2, vendar ga moramo najprej povišati. Nastavimo $h(2) = 2$ in potisnemo iz njega tok $\delta = \min\{e(2), c(2,1) - f(2,1)\} = \min\{6, 4\} = 4$, s čimer bomo zasičili povezavo $(2,1)$.



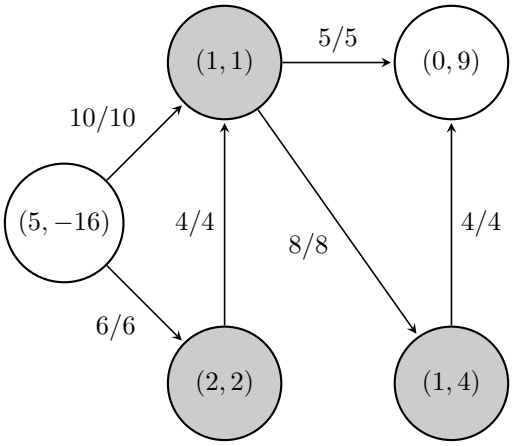
Slika 4. Opravili smo povišanje vozlišča 2 in potistnili tok iz njega.

Sedaj nam ni potrebno povišati nobenega vozlišča, saj lahko potisnemo tok iz vozlišča 1 po povezavi $(1,3)$. Potisnemo lahko $\delta = \min\{e(1), c(1,3) - f(1,3)\} = \min\{4, 8-5\} = 3$, s čimer bomo zasičili povezavo $(1,3)$, vozlišče 1 pa bo še vedno ostalo v presežku.



Slika 5. Potisnili smo tok po povezavi $(1,3)$.

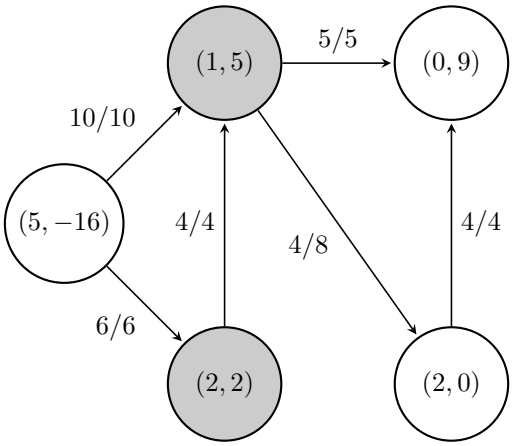
Sedaj izberemo vozlišče 3, saj ima edino še nezasičene izhodne povezave. Iz njega lahko v t pošljemo $\delta = \min\{e(3), c(3,4) - f(3,4)\} = 4$ enot, vendar ga moramo najprej povišati na višino 1.



Slika 6. Povišali smo vozlišče 3 in potisnili tok po povezavi $(3, 4)$.

Očitno smo sedaj v t dobili največ toka, kolikor ga lahko dobimo, saj je $e(t) = 9 = \sum_i (i, t)$. Vendar se na tej točki algoritem ne ustavi. Razlog za to lahko vidimo v tem, da smo iz s poslali 16 enot, v t pa prejeli le 9 enot, kar nam nekako ni všeč, saj se je $16 - 9 = 7$ enot po poti izgubilo. Prav tako imamo še vedno vozlišča v presežku, iz katerih lahko tok potisnemo. Kakor bomo videli v lemi 3.2, in kakor je morda očitno, lahko na tem vozlišču opravimo eno od operacij `POTISNI` ali `POVIŠAJ`.

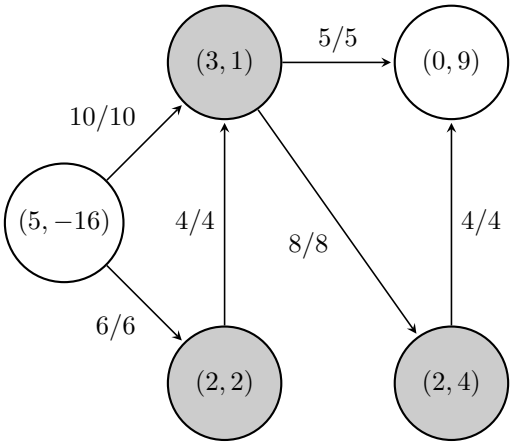
Izberemo si vozlišče 3, v katerem je trenuten presežek 4. V vozlišče t iz njega ne moremo potisniti, saj je povezava z njim zasičena. Lahko pa tok *vrnemo* v vozlišče 1. To pravzaprav pomeni, da pošljemo tok po residualni povezavi $(3, 1)$. Preden to naredimo, moramo seveda vozlišče 3 povišati.



Slika 7. Povišali smo vozlišče 3 in potisnili tok po residualni povezavi $(3, 1)$, kar je analogno temu, da smo zmanjšali tok po povezavi $(1, 3)$.

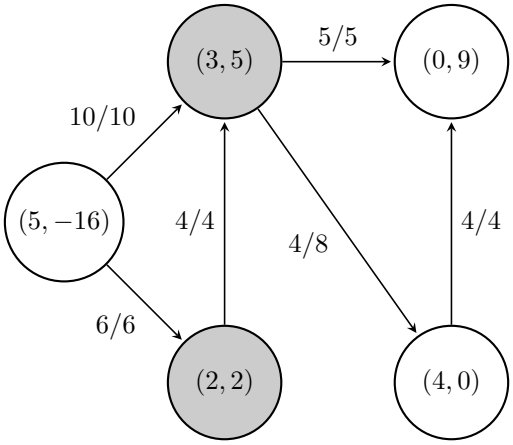
Prišli smo do najbolj zanimivega dela celega algoritma, ki morda na prvi pogled izgleda precej smešno. Izberemo si vozlišče 1, iz katerega bi radi potisnili presežek. Če to želimo storiti, moramo najprej vozlišče povišati, in sicer na $\min\{h(1), h(3), h(0)\} + 1 = 3$. Sedaj lahko potisnemo tok po residualni povezavi $(1, 2)$ ali pa po povezavi $(1, 3)$. Logično bi se nam zdelo, da toka ne bomo pošiljali nazaj v vozlišče 3, od koder smo ga ravno dobili, ampak storili bomo ravno to, saj želimo tok najprej poslati po originalni povezavi, šele nato po residualni. S tem bi potencialno omogočili, da

pride več toka v nova vozlišča in s tem ,,bližje" ponoru. Povišamo tako 1 na višino 3 in pošljemo 4 enote nazaj v vozlišče 3.



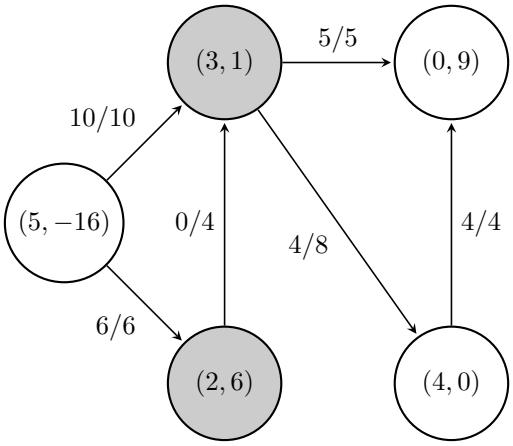
Slika 8. Povišali smo vozlišče 1 in spet poslali tok v vozlišče 3.

Kaj pa sedaj? Iz vozlišča 3 moramo spet vrniti tok nazaj v 1, pred tem pa še povišati višino $h(3)$ na 4.



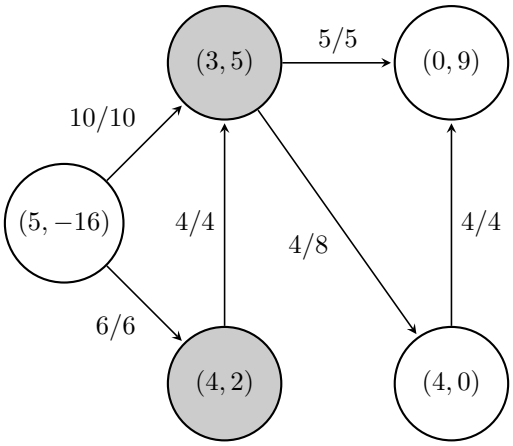
Slika 9. Povišali smo vozlišče 3 in spet vrnili tok v vozlišče 1.

V tem trenutku pa lahko končno naredimo majhen napredek, saj lahko potisnemo iz vozlišča 1 nazaj v vozlišče 2, namreč $h(1) > h(2)$ in obstaja povezava $(1, 2)$ v residualnem omrežju. Seveda lahko potisnemo le $\delta = \min\{e(1), f(u, v)\} = \min\{5, 4\} = 4$, torej bo nekaj toka še ostalo v vozlišču 1.



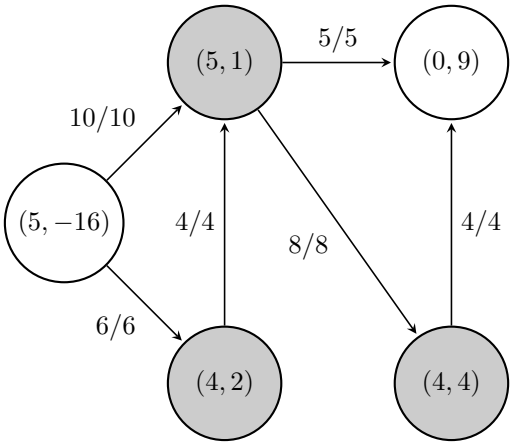
Slika 10. Vrnili smo tok iz vozlišča 1 v vozlišče 2.

Sedaj bi najraje kar vrnili 6 enot po povezavi $(0, 2)$, vendar ne moremo, saj lahko povišamo vozlišče 2 na $\min\{h(0), h(1)\} + 1 = 4 < h(0) + 1$. Tako sedaj pošljemo 4 enote nazaj v vozlišče 1 po povezavi $(2, 1)$.



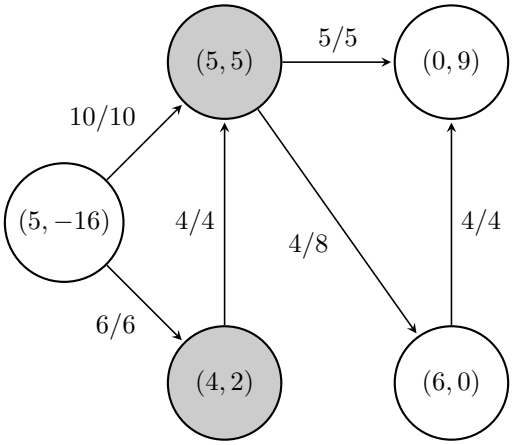
Slika 11. Povišamo vozlišče 2 in spet pošljemo tok po povezavi $(2, 1)$.

Podobno premislimo kot smo pri sliki 8. Ugotovimo, da moramo povišati vozlišče 1 na 5 in poslati tok v vozlišče 3.



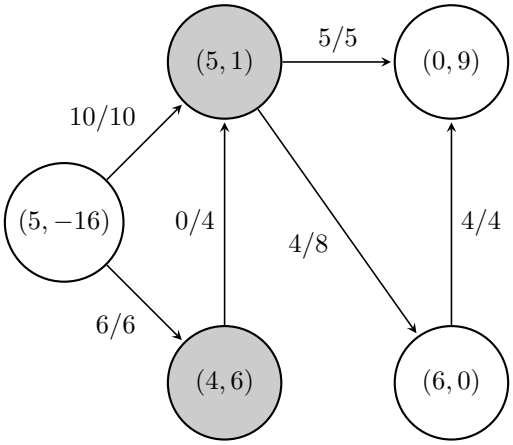
Slika 12. Povišamo vozlišče 1 in spet pošljemo tok po povezavi $(1, 3)$.

Razmislimo podobno kot pri sliki 9 – povišamo vozlišče 3 in vrnemo tok v vozlišče 1.



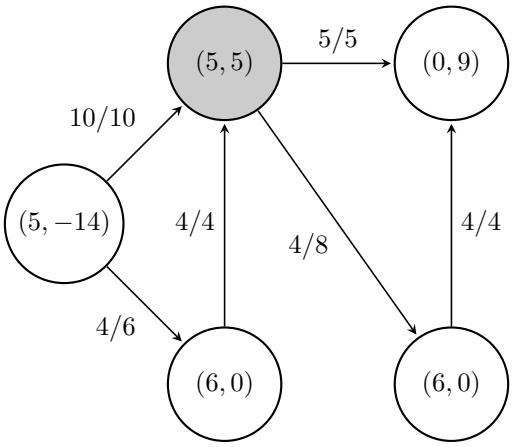
Slika 13. Povišamo vozlišče 3 in vrnemo tok v vozlišče 1.

Znašli smo se v identični situaciji kot pri sliki 9, le višine vozlišč so drugačne. Spet lahko vrnemo tok brez povišanja iz 1 v 2.



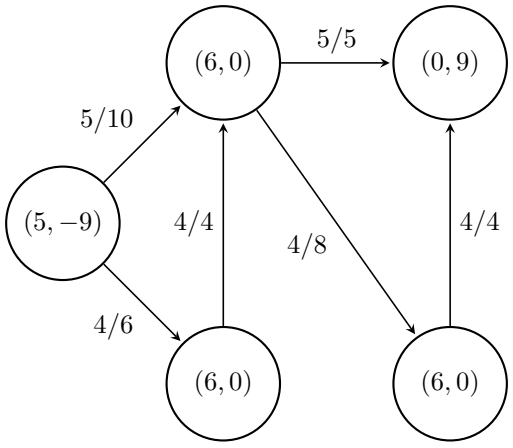
Slika 14. Vrnemo tok iz vozlišča 1 v vozlišče 2.

Povišati moramo vozlišče 2 na $\min\{h(0), h(1)\} + 1 = 6$, vendar še ne smemo vrniti toka v s, saj je prosta povezava $(2, 1)$, po kateri lahko pošljemo 4 enote. V vozlišču 2 nam bosta potem ostali še 2 enoti, ki pa ju končno lahko vrnemo v s. Znajdemo se v sledeči situaciji.



Slika 15. Povišamo vozlišče 2 na 6. Najprej pošljemo tok 4 po originalni povezavi $(2, 1)$, nato pa vrnemo tok 2 v s.

Tokrat nam ne bo treba ponovno pošiljati toka iz 1 v 3, saj bomo vozlišče 1 dvignili na $\min\{h(0), h(2), h(3)\} + 1 = \min\{5, 6, 6\} + 1 = 6$, kar pa pomeni, da ne moremo potisniti v vozlišče 3. Vrnemo torej ves tok v presežku v vozlišču 1 v s in s tem se algoritem zaključi, saj ne moremo opraviti nobene operacije več.



Slika 16. Povišamo vozlišče 1 na 6 in vrnemo tok 5 v s. Tako izgleda naše omrežje po končanem algoritmu.

Vidimo sedaj, da po poti nismo izgubili nobene enote, saj smo jih iz s poslali 9, 9 pa smo jih tudi prejeli v t. Števila $f(u, v)$, kjer je $(u, v) \in E$, nam povedo, kolikšno količino moramo poslati po povezavi (u, v), da bo pretok maksimalen.

3.3. **Implementacija algoritma v programskem jeziku C++.** Najprej si poglejmo idejo implementacije. Prvo vprašanje, ki se nam postavi, je, kako bomo predstavili graf. Za vsako vozlišče si moramo zapomniti njegovo višino in presežek toka, ki se v njem nahaja. Odločimo se, da bomo vozlišče predstavili s strukturo, saj bo tako koda bolj berljiva. Lahko bi sicer uporabili razred, vendar ni potrebe,

lahko pa bi naredili tudi dva vektorja, enega, ki bi predstavljal višinsko funkcijo in drugega, ki bi povedal, koliko presežka toka je v vsakem izmed vozlišč. Za predstavitev grafa bomo tako uporabili vektor vozlišč in pa matriko $p \in \{0,1\}^{V \times V}$ boolovih vrednosti. Element p_{ij} bo povedal, ali je povezava (i,j) v grafu ali ne. Prav tako si bomo definirali dve matriki $c, f \in \mathbb{N}_0^{V \times V}$. Matrika c bo imela na ij-tem mestu kapaciteto povezave (i,j), matrika f pa pretok preko povezave (i,j).

Definirali si bomo še vrsto, v katero bomo spravljali vozlišča, ki imajo presežek toka. S tem bomo lahko do elementov, na katerih moramo opraviti operacijo potiska ali povišanja, dostopali v konstantnem času, algoritem pa bo deloval pravilno. Vzeli bomo namreč prvo vozlišče v vrsti, na njem naredili ali `POTISNI` ali `POVIŠAJ`, in ga odstranili iz vrste, če bo po opravljeni operaciji njegov presežek ničeln. Prav tako bomo na konec vrste dodali vozlišče, v katerega smo potisnili tok, če je seveda šlo za operacijo potiska. Na tej točki bomo potrebovali še vektor boolovih vrednosti, ki bo za vsako vozlišče povedal, ali smo ga že dodali med presežke. S tem se izognemo podvajanju elementov v vrsti presežkov in tako pripomoremo k hitrejšemu delovanju algoritma; in hkrati tudi pravilnejšemu, saj se lahko zgodi, da bi algoritem želel opravljati potisk ali povišanje na vozlišču, ki smo ga spraznili, ko se je prejšnjič pojavil v vrsti.

Algoritem bomo razbili na več funkcij. Glavne bodo seveda `potisni_povisaj`, `potisni` in `povisaj`. Poglejmo si opis, kaj počne katera izmed funkcij.

- `potisni_povisaj()`: Izračuna maksimalni pretok v grafu. Najprej inicializira predpretok, nato pa izvaja zanko, dokler obstaja vozlišče s presežkom. Vsakič se odloči, ali bo naredila potisk ali povišanje in potisk. Za povišanjem naredimo takoj potisk zato, da zmanjšamo število vrtljajev zanke in nam tako ni potrebno še enkrat računati najnižjega soseda, saj bo ostal isti.
- `inicializiraj_predpretok()`: Inicializira predpretok. Nastavi višino vozlišča s na moč množice vozlišč. Nato potisne tok po vseh sosednjih povezavah in posodobi presežke v vozliščih. Vmes tudi doda residualno (obratno) povezavo, katere kapaciteta je 0, tok po njej pa je enak negativno predznačenemu toku, ki smo ga poslali po originalni povezavi. Če vozlišče, v katerega smo potisnili tok, ni t, ga dodamo med vozlišča s presežki in označimo, da je bil tja dodan.
- `potisni(u,v)`: Potisne tok iz vozlišča u v vozlišče v. Tok, ki ga lahko potisne, je enak minimumu med presežkom v vozlišču u in residualno kapaciteto povezave (u,v). Če smo vozlišče u izpraznili, ga odstranimo iz vrste presežkov in označimo, da ga ni več notri. Če vozlišče v ni t ali s in še ni bilo dodano med presežke, ga dodamo.
- `povisaj(u,h)`: Poveča višino vozlišča u na $h+1$.
- `najnizji_sosed(u)`: Poišče tisto vozlišče v med sosedi u, ki ima najmanjšo višino in povezava (u,v) še ni zasičena.
- `napolni_graf()`: Inicializira vse potrebne spremenljivke. V vektor vozlišč doda vozlišča z višino in presežkom nič. V matriko kapacitet vpiše pripadajoče kapacitete ter v matriki p označi, katere povezave obstajajo.

Oglejmo si sedaj implementacijo.

```
// Implementacija algoritma potisni-povisaj v C++

//================================================================
```

```
//
//        KNJIZNICE IN DEFINICIJE
//
//=================================================================

#include<iostream>
#include<queue>
#include<cstdio>
#include<cstdlib>
#include<climits>

using namespace std;

// Vozlisce predstavimo s strukturo. Za vsako vozlisce si zapomnimo
// njegovo visino in presezek toka v njem.
struct Vozlisce
{
    int h, e;

    Vozlisce(int h, int e)
    {
        this->h = h;
        this->e = e;
    }
};

// Prototipi funkcij.
int potisni_povisaj();
void inicializiraj_predpretok();
void potisni(int u, int v);
void povisaj(int u, int h);
int najnizji_sosed(int u);
void napolni_graf();

// Globalne spremenljivke.

// Graf predstavimo z vektorjem vozlisc.
vector<Vozlisce> vozlisca;

// V vrsto dajemo vozlisca, ki so v presezku. S tem lahko v
// O(1) dostopamo do naslednjega vozlisca, na katerem je potrebno
// opraviti operacijo POTISNI ali POVISAJ.
queue<int> presezki;

// V vektor shranjujemo vozlisca, ki smo jih ze dodali med
// presezke. S tem dosezemo, da se vsako vozlisce v presezkih
// pojavi kvecjemu enkrat.
vector<bool> viden;
```

```
// Matrika povezav grafa. Ce je p[i][j] true, to pomeni, da
// povezava obstaja. S tem se izognemo temu, da bi se v sosedih
// vozlisc zaceli sosedi ponavljati. Prej smo namrec za vsako
// vozlisce imeli vektor sosedov in s tem ponazorili povezave.
// V obeh primerih s tem zmanjsamo casovno zahtevnost iz O(E)
// na O(V).
bool** p;

int** c; // Matrika kapacitet povezav.
int** f; // Matrika toka.

//=====================================================================
//
//        MAIN
//
//=====================================================================

// Main funkcija, uporabljena za testiranje programa.
int main()
{
    // S standardnega vhoda preberi podatke o vozliscih
    // in povezavah in napolni vektorja vozlisca in povezave.
    napolni_graf();

    // Izvedi algoritem POTISNI-POVISAJ in izpisi rezultat.
    cout << "Maksimalni pretok je " << potisni_povisaj() << endl;
}

//=====================================================================
//
//        ALGORITEM
//
//=====================================================================

// Glavna funkcija algoritma potisni-povisaj. Najprej
// inicializira predpretok, potem pa izvaja operacije potiska
// in povisanja, kakor je pac potrebno. To pocne, dokler
// obstaja vozlisce s presezkom (to ne moreta biti s in t).
int potisni_povisaj()
{
    inicializiraj_predpretok();

    // Dokler ima katero izmed vozlisc presezek toka,
    // moramo opraviti ali POTISNI ali POVISAJ. Ce opravimo
    // POVISAJ, lahko takoj potem tudi potisnemo na tisto
    // vozlisce. S tem si zmanjsamo stevilo zank.
    while (presezki.size() > 0)
    {
        int u = presezki.front();
```

```
        int v = najnizji_sosed(u);

        if (vozlisca[u].h == 1 + vozlisca[v].h)
            potisni(u, v);
        else
        {
            povisaj(u, vozlisca[v].h);
            potisni(u,v);
        }
    }

    // Vrnemo presezek v vozliscu t. Lahko bi vrnili
    // tudi -presezek v vozliscu s.
    return vozlisca.back().e;
}

// Inicializira predpretok. Visino vozlisca s nastavi na
// |V|, zasici povezave iz s, doda residualne povezave
// in ce sosed ni vozlisce t, ga doda v presezke.
void inicializiraj_predpretok()
{
    vozlisca[0].h = vozlisca.size();

    for (int v = 0; v < vozlisca.size(); v++)
    {
        if (p[0][v])
        {
            // Zasici povezavo.
            f[0][v] = c[0][v];

            // Nastavi presezek v sosedu.
            vozlisca[v].e = f[0][v];

            // Doda residualno povezavo.
            f[v][0] -= f[0][v];
            p[v][0] = true;

            // V vozliscu s posodobi oddani tok.
            vozlisca[0].e -= f[0][v];

            // Ce vozlisce ni t, ga doda v presezke.
            if (v != vozlisca.size()-1)
            {
                presezki.push(v);
                viden[v] = true;
            }
        }
    }
}
```

```
// Vrne indeks najnizjega soseda.
int najnizji_sosed(int u)
{
    int sosed;
    int min_visina = INT_MAX;

    for (int v = 0; v < vozlisca.size(); v++)
    {
        if (p[u][v])
        {
            // Ce je visina soseda manjsa od trenutne najmanjse
            // visine in ce povezava ni zasicena, potem je to
            // kandidat za najnizjega soseda.
            if (vozlisca[v].h < min_visina && c[u][v] - f[u][v] > 0)
            {
                min_visina = vozlisca[v].h;
                sosed = v;
            }
        }
    }

    return sosed;
}

// Operacija POTISNI. Potisne lahko minimum med presezkom
// v vozliscu in residualno kapaciteto povezave.
void potisni(int u, int v)
{
    int delta = min(vozlisca[u].e, c[u][v] - f[u][v]);

    // Posodobi presezek v krajiscih povezave.
    vozlisca[u].e -= delta;
    vozlisca[v].e += delta;

    // Posodobi tok prek povezave in residualne povezave.
    f[u][v] += delta;
    f[v][u] -= delta;

    // Doda obratno povezavo.
    p[v][u] = true;

    // Ce smo z vozliscem opravili, ga odstranimo iz presezkov.
    if (vozlisca[u].e == 0)
    {
        presezki.pop();
        viden[u] = false;
    }
```

```
    // Ce konec povezave ni t ali s in ce vozlisce se ni v
    // presezkih, ga doda v presezke.
    if (v != vozlisca.size()-1 && v != 0 && !viden[v])
    {
        presezki.push(v);
        viden[v] = true;
    }
}

// Operacija POVISAJ. Vozliscu nastavi visino na prej
// izracunano minimalno visino "dobrih" sosedov h + 1.
void povisaj(int u, int h)
{
    vozlisca[u].h = h + 1;
}

// Iz datoteke prebere podatke o stevilu vozlisc in
// povezavah. V vektor vozlisc doda vozlisca, ki jim
// nastavi visino in presezek na 0. Vsakemu vozliscu
// nato doda sosede in nastavi kapaciteto povezav.
void napolni_graf()
{
    int V;
    scanf("%d\n", &V);

    c = (int**) malloc(V*sizeof(int*));
    f = (int**) malloc(V*sizeof(int*));
    p = (bool**) malloc(V*sizeof(int*));

    for (int i = 0; i < V; i++)
    {
        vozlisca.push_back(Vozlisce(0, 0));
        viden.push_back(false);
        c[i] = (int*) malloc(V*sizeof(int));
        f[i] = (int*) malloc(V*sizeof(int));
        p[i] = (bool*) malloc(V*sizeof(bool));
    }

    int u, v, kapaciteta;
    while (scanf("%d %d %d\n", &u, &v, &kapaciteta) != EOF)
    {
        c[u][v] += kapaciteta;
        p[u][v] = 1;
    }
}
```

Opazimo naslednjo stvar. V funkciji `potisni_povisaj` smo zahtevali, da se program izvaja dokler obstaja vozlišče s presežkom. Spomnimo se, da smo v psevdokodi namreč zapisali, da se mora program izvajati dokler obstaja mogoča operacija potiska ali povišanja. Da sta stvari ekvivalentni, si bomo pogledali v dokazu pravilnosti algoritma (lema 3.2).

3.4. **Pravilnost delovanja algoritma.** V tem podrazdelku bomo s pomočjo lem pokazali, da tako algoritem `POTISNI-POVIŠAJ` kot tudi njegova implementacija delujeta pravilno. S tem mislimo na to, da se algoritem konča in ob tem vrne pravilen rezultat, torej res pretok, ki je maksimalen.

Če si pogledamo, kako deluje funkcija `potisni`, vidimo, da nikjer v kodi ne uporabimo dejstva, da je razlika višine med vozliščema nujno ena. Vendar to še vseeno zahtevamo.

Lema 3.1. *Naj bo $(G = (V, E), c, s, t)$ pretočno omrežje, $f\colon V \times V \to \mathbb{N}_0$ predpretok v G in $h\colon V \to \mathbb{N}_0$ višinska funkcija. Potem za vsaki vozlišči $u, v \in V$ velja, da če je $h(u) > h(v) + 1$, potem povezava (u, v) ni v residualnem omrežju.*

Ta lema nam pove, da ne obstaja residualna povezava med u in v, če je $h(u) > h(v) + 1$. To pomeni, da če potisnemo tok v vozlišče za več kot ena nižje, ne bomo naredili nič konkretnega, kar se je preprosto prepričati.

Sedaj si poglejmo obljubljeno lemo, ki nam zagotavlja pravilnost delovanja implementacije in postopka, ki smo ga uporabili v zgledu. Spomnimo se namreč, da smo v implementaciji algoritem opravljali, dokler je bilo kakšno vozlišče s presežkom toka, čeprav smo v psevdokodi zapisali, da moramo algoritem opravljati, dokler je mogoča katera izmed operacij `POTISNI` in `POVIŠAJ`.

Lema 3.2 (na vozlišču s presežkom lahko opravimo ali potisk ali povišanje)**.** *Naj bo $G = (V, E, s, t)$ pretočno omrežje, f predpretok, h višinska funkcija in $e\colon V \to \mathbb{N}_0$ funkcija, ki za vsako vozlišče pove, kolikšen je v njem presežek toka. Če ima vozlišče $u \in V$ presežek toka, torej $e(u) > 0$, potem lahko na tem vozlišču opravimo ali operacijo `POTISNI` ali operacijo `POVIŠAJ`.*

Dokaz. Naj ima u presežek toka. Za vsako residualno povezavo (u, v) velja $h(u) \leq h(v)+1$, ker je h višinska funkcija. Če ne moremo opraviti operacije `POTISNI`, potem za vse residualne povezave (u, v) velja $h(u) < h(v) + 1$, oziroma $h(u) \leq h(v)$. Torej lahko opravimo operacijo `POVIŠAJ`. □

Oglejmo si tri leme o višinski funkciji.

Lema 3.3 (višine vozlišč se nikoli ne zmanjšajo)**.** *Med izvajanjem programa `POTISNI -POVIŠAJ` velja za vsako vozlišče $u \in V$, da se $h(u)$ nikoli ne zmanjša. Še več, vsakič, ko na u opravimo povišanje, se njegova višina poveča za vsaj ena.*

Dokaz. Ker se višine vozlišč spreminjajo le med povišanji, je za dokaz celotne leme zadosti pokazati drugi del leme. Naj bo sedaj u vozlišče, na katerem opravljamo povišanje. Torej za vse $v \in V$, za katere je $(u, v) \in E_f$, velja $h(u) \leq h(v)$. Ker to velja za vsak v, velja tudi

$$u \leq \min_{(u,v)\in E_f} h(v)$$

kar pa je ekvivalentno

$$u < 1 + \min_{(u,v)\in E_f} h(v).$$

□

Lema 3.4. *Med izvajanjem programa* `POTISNI-POVIŠAJ(G,s)` *h vedno zadrži lastnosti višinske funkcije, opisane v definiciji 2.13.*

Dokaz. Dokaz bomo naredili s pomočjo indukcije na število osnovnih operacij.

Po inicializaciji predpretoka je h očitno višinska funkcija.

Poglejmo si najprej, kaj se zgodi med operacijo `POVIŠAJ(u)`. `POVIŠAJ(u)` zagotovi, da za vsako residualno povezavo $(u, v) \in E_f$ po opravljeni operaciji velja $h(u) \leq h(v) + 1$. Vzemimo sedaj residualno povezavo, ki vstopa v u, recimo $(w, u) \in E_f$. Po lemi 3.3 iz $h(w) \leq h(u) + 1$ pred operacijo sledi $h(w) < h(u) + 1$ po operaciji. Torej operacija `POVIŠAJ(u)` očitno ohranja h kot višinsko funkcijo.

Ostane nam pokazati še, da če je h višinska funkcija pred operacijo `POTISNI(u,v)`, potem je tudi po operaciji. Med to operacijo se zgodi natanko ena izmed naslednjih stvari:

(1) *Dodamo residualno povezavo* (v, u) *v* E_f. V tem primeru imamo $h(v) = h(u) - 1 < h(u) + 1$, torej h ostane višinska funkcija.
(2) *Odstranimo residualno povezavo* (u, v) *iz* E_f. Z odstranitvijo residualne povezave (u, v) pravzaprav izgubimo zahtevo iz definicije višinske funkcije (definicija 2.13), tako da h na prazno ostane višinska funkcija.

Ker vse operacije ohranjajo lastnosti višinske funkcije funkcije h, po principu indukcije sledi lema. □

Sledi lema, ki nam da pomembno posledico definicije višinske funkcije.

Lema 3.5. *Naj bo* $G = (V, E, s, t)$ *pretočno omrežje,* f *predpretok v* G *in* h *višinska funkcija na* V. *Potem ne obstaja pot od* s *do* t *v residualnem omrežju* G_f.

Dokaz. Pokažimo s pomočjo protislovja. Recimo torej, da v G_f obstaja pot p od s do t, kjer je $p = \langle v_0 = s, v_1, v_2, \ldots v_{n-1}, v_n = t\rangle$. Brez škode za splošnost lahko predpostavimo, da je p enostavna pot, torej pot brez ciklov. Potem velja $n < |V|$. Ker je p pot v residualnem omrežju G_f, za vsak $i = 0, 1, \ldots, n-1$ velja, da je $(v_i, v_{i+1}) \in E_f$. Ker pa je h višinska funkcija, za vsak $i = 0, 1, \ldots, n-1$ velja $h(v_i) \leq h(v_{i+1}) + 1$. Od tod sledi, da velja $h(s) \leq h(t) + n$. Ampak, ker je po definiciji višinske funkcije $h(t) = 0$, dobimo $h(s) \leq n < |V|$, kar pa je v protislovju s tem, da je h višinska funkcija. Veljati bi namreč moralo $h(s) = |V|$. Zaključimo, da v residualnem omrežju G_f torej ne obstaja pot med s in t. □

Manjka nam še klasični izrek iskanja maksimalnih pretokov, ki mu v angleščini pravimo *max-flow min-cut theorem*. Nismo se sicer spoznali s pojmom *prerez omrežja*, vendar bomo izrek vseeno navedli v celoti, saj se to edino spodobi. Dokaz izreka bomo spustili, saj je preobširen in zahteva določena znanja, ki za algoritme tipa potisni-povišaj nimajo vrednosti.

Izrek 3.6 (izrek max-pretok min-prerez)**.** *Naj bo* $G = (V, E)$ *in* f *pretok v omrežju* (G, c, s, t). *Naslednje trditve so ekvivalentne.*

(1) f *je maksimalni pretok v* G.
(2) *Residualno omrežje* G_f *ne vsebuje nobene povečujoče poti.*
(3) $|f| = c(S, T)$ *za nek prerez* (S, T) *omrežja* G.

Sedaj smo pridobili vso potrebno znanje, da dokažemo, da ČE se algoritem POTISNI-POVIŠAJ zaključi, je potem predpretok, ki ga algoritem vrne, enak maksimalnemu pretoku skozi omrežje.

Izrek 3.7. *Naj bo $G = (V, E, s, t)$ pretočno omrežje. Če poženemo algoritem POTISNI-POVIŠAJ na pretočnem omrežju G in se ustavi, potem je predpretok f, ki ga algoritem vrne, enak maksimalnemu toku skozi pretočno omrežje G.*

Dokaz. Najprej pokažimo, da je predpretok f na vsaki iteraciji zanke DOKLER v vrstici 2 algoritma POTISNI-POVIŠAJ res predpretok. Očitno je pred prvo iteracijo f predpretok, saj za to poskrbi operacija INICIALIZIRAJ_PREDPRETOK. Znotraj zanke DOKLER se lahko zgodita le dve operaciji, ali POTISNI ali POVIŠAJ. Operacija POVIŠAJ vpliva le na višine vozlišč in ne na vrednosti $f(u, v)$. Posledično f ostane predpretok.

Med operacijo POTISNI pride do sprememb vrednosti $f(u, v)$. Poskrbeti moramo, da se ohranita lastnosti predpretoka:

(1) Za vsako povezavo $(u, v) \in E$ velja $f(u, v) \leq c(u, v)$.
(2) Za vsako vozlišče $u \in V$ velja $e(u) = \sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v) \geq 0$.

Ker smo v vrstici 3 operacije POTISNI vzeli $\delta = \min\{e(u), c(u, v) - f(u, v)\}$, vidimo, da bomo ne glede na to, kaj bo minimum, uspeli zadostiti pogoju (1). Namreč, če vzamemo $c(u, v) - f(u, v)$, bomo povezavo (u, v) ravno nasičili in bo po opravljeni operaciji veljalo $f(u, v) = c(u, v)$. Če pa velja $e(u) < c(u, v) - f(u, v)$, pa povezave ne bomo zasičili in tako niti ne bomo presegli njene kapacitete.

Podobno premislimo, da se ohrani lastnost (2). Če v vrstici 3 operacije POTISNI vzamemo $\delta = e(u)$, bomo potem v vrstici 7 zmanjšali presežek v vozlišču u na nič, torej bo veljalo $e(u) = 0$. Če pa bomo vzeli $\delta = c(u, v) - f(u, v) < e(u)$, bo po opravljeni operaciji veljalo $\tilde{e}(u) = e(u) - \big(c(u, v) - f(u, v)\big) > 0$, kjer je $\tilde{e}(u)$ presežek po opravljeni operaciji. Ker $e(v)$ prištejemo pozitivno vrednost, bo po opravljeni operaciji še vedno veljalo $e(v) > 0$.

Ob zaključku izvajanja algoritma velja $e(u) = 0$ za vsak $u \in V \setminus \{s, t\}$. Namreč iz leme 3.2 in dejstva, da je f po vsaki operaciji še vedno predpretok, sledi, da po zaključku ne morejo obstajati vozlišča s presežki (s in t namreč nikoli nista v presežku). To pomeni, da je predpretok f pravzaprav tok. Lema 3.4 nam pove, da je ob zaključku algoritma h še vedno višinska funkcija, od koder po lemi 3.5 sledi, da ob zaključku algoritma ni poti med s in t v residualnem omrežju G_f. Po izreku 3.6 (izrek o maksimalnem pretoku in minimalnem prerezu) je tako f maksimalni pretok. □

S tem smo pokazali, da če se algoritem konča, dobimo pravilen rezultat. Ostane nam pokazati še, da se algoritem sploh konča. To bomo naredili v naslednjem podrazdelku, v katerem se bomo ukvarjali s časovno zahtevnostjo. Omejili bomo število operacij, ki se lahko zgodijo, in s tem pokazali, da se algoritem res konča.

3.5. **Časovna zahtevnost algoritma.** V tem podrazdelku bomo kot obljubljeno dokazali še, da se algoritem POTISNI-POVIŠAJ konča. To bomo naredili s pomočjo omejevanja števila operacij, ki se lahko zgodijo. Namesto osnovnih dveh operacij, POTISNI in POVIŠAJ, se bomo tukaj ukvarjali s tremi operacijami – operacijo

`POTISNI` bomo namreč razdelili na dve, na tisto, ki povezavo zasiči, in tisto, ki je ne.

Preden se lotimo analize časovne zahtevnosti pa si poglejmo in dokažimo še eno lemo. Spomnimo se, da namreč dovolimo povezave v izvir s v residualnem omrežju.

Lema 3.8. *Naj bo $G = (V, E, s, t)$ pretočno omrežje in f predpretok v G. Potem za vsako vozlišče $x \in V$, ki je v presežku, obstaja enostavna pot od x do s v residualnem omrežju G_f.*

Dokaz. Za vozlišče v presežku x definiramo

$$U = \{v : \text{obstaja enostavna pot od } x \text{ do } v \text{ v } G_f\}.$$

Predpostavimo, da $s \notin U$, in pokažimo protislovje. Definirajmo še $U' = V \setminus U$. Velja torej $V = U \cup U'$. Spomnimo se še, da je $e(u) = \sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v)$. Tako velja

$$\begin{aligned}
\sum_{u \in U} e(u) &= \sum_{u \in U} \left(\sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v) \right) \\
&= \sum_{u \in U} \left(\left(\sum_{v \in U} f(v, u) + \sum_{v \in U'} f(v, u) \right) - \left(\sum_{v \in U} f(u, v) + \sum_{v \in U'} f(u, v) \right) \right) \\
&= \sum_{u \in U} \sum_{v \in U} f(v, u) + \sum_{u \in U} \sum_{v \in U'} f(v, u) - \sum_{u \in U} \sum_{v \in U} f(u, v) - \sum_{u \in U} \sum_{v \in U'} f(u, v) \\
&= \sum_{u \in U} \sum_{v \in U'} f(v, u) - \sum_{u \in U} \sum_{v \in U'} f(u, v).
\end{aligned}$$

Zadnja enakost sledi iz dejstva, da seštevamo po povezavah, ki imajo krajišča v isti množici. Torej dobimo v vsoti tako $f(u, v)$ kot tudi $f(v, u)$. Ker velja $f(u, v) = -f(v, u)$, velja

$$\sum_{u \in U} \sum_{v \in U} f(u, v) = \sum_{u \in U} \sum_{v \in U} f(v, u) = 0.$$

Vemo, da je $\sum_{u \in U} e(u) > 0$, saj $x \in U$ in $e(x) > 0$ ter $e(u) \geq 0$ za vse $y \in V$, razen s, za katerega smo predpostavili $s \notin U$. Torej

$$\sum_{u \in U} \sum_{v \in U'} f(v, u) - \sum_{u \in U} \sum_{v \in U'} f(u, v) > 0.$$

Ker je tok po vseh povezavah nenegativen, mora veljati $\sum_{u \in U} \sum_{v \in U'} f(v, u) > 0$. To pomeni, da obstajata vozlišči $u' \in U$ in $v' \in U'$, za kateri velja $f(v', u') > 0$. Ampak to pomeni, da obstaja residualna povezava (u', v'). To pa pomeni, da obstaja enostavna pot od x do v', namreč $x \rightsquigarrow u' \to v'$. Pot $x \rightsquigarrow u'$ namreč obstaja po definiciji U. To pa je v protislovju z definicijo U. □

Z naslednjo lemo bomo omejili največjo možno višino vozlišč, njena posledica pa bo omejila skupno število opravljanj operacije `POVIŠAJ`.

Lema 3.9. *Naj bo $G = (V, E, s, t)$ pretočno omrežje. Na koncu izvajanja algoritma `POTISNI-POVIŠAJ` na G, za vsak $u \in V$ velja $h(u) \leq 2|V| - 1$.*

Opomba 3.10. Vemo, da višina vozlišč med izvajanjem algoritma POTISNI-POVIŠAJ ne pada. To pomeni, da je v vsakem trenutku izvajanja algoritma $h(u) \leq 2|V| - 1$ za vsak $u \in V$.

Dokaz. Višini vozlišč s in t se ne spreminjata. Velja $h(s) = |V| \leq 2|V| - 1$ in $h(t) = 0 \leq 2|V| - 1$.

Osredotočimo se torej na vozlišča $u \in V \setminus \{s, t\}$. Na začetku velja $h(u) = 0 \leq 2|V| - 1$. Pokažimo, da po vsaki operaciji POVIŠAJ še vedno velja $h(u) \leq 2|V| - 1$. Vsakič, ko povišamo vozlišče u, ima u presežek toka. Lema 3.8 nam pove, da obstaja enostavna pot p od u do s v residualnem grafu G_f. Naj bo $p = \langle v_0 = u, v_1, \ldots, v_{n-1}, v_n = s \rangle$. Velja $n \leq |V| - 1$, saj je p enostavna pot. Velja tudi, da je za vsak $i = 0, 1, \ldots, n-1$ povezava $(v_i, v_{i+1}) \in E_f$. Po lemi 3.4 velja $h(v_i) = h(v_{i+1}) + 1$. Če združimo te enakosti po celi poti p, dobimo

$$h(u) = h(v_0) \leq h(v_n) + n = h(s) + n \leq h(s) + |V| - 1 = 2|V| - 1.$$

□

Posledica 3.11 (omejenost števila operacij POVIŠAJ)**.** *Naj bo $G = (V, E, s, t)$ pretočno omrežje. Potem je število operacij POVIŠAJ med izvajanjem algoritma POTISNI-POVIŠAJ manjše od $2|V|^2$.*

Dokaz. Povišamo lahko samo $|V| - 2$ vozlišč; vozlišči s in t imata namreč ves čas isto višino. Za vozlišče $u \in V \setminus \{s, t\}$ na začetku velja $h(u) = 0$. Lema 3.9 nam pove, da vozlišče u ne preseže višine $2|V| - 1$ in ker višina ne pada, ga lahko povišamo največ $(2|V| - 1)$-krat. Ker se to lahko zgodi za $|V| - 2$ vozlišč, je število operacij POVIŠAJ tako navzgor omejeno z

$$(|V| - 2)(2|V| - 1) = 2|V|^2 - 5|V| + 2 < 2|V|^2.$$

□

Ostane nam pokazati še, da je tudi število operacij POTISNI omejeno. Kot že rečeno, bomo dokaz razbili na dva dela. Najprej bomo pokazali, da je omejeno število potiskov, ki povezavo zasičijo, potem pa bomo pokazali še, da je omejeno tudi število potiskov, ki povezave ne zasičijo.

Lema 3.12 (omejenost števila operacij POTISNI, ki zasičijo povezavo)**.** *Naj bo $G = (V, E, s, t)$ pretočno omrežje. Potem je število operacij POTISNI, ki povezavo zasičijo, manjše od $2|V||E|$.*

Dokaz. Za vsak par vozlišč $u, v \in V$ bomo prešteli, koliko potiskov, ki zasičijo povezavo, je iz u v v in obratno. Če kakšen tak potisk obstaja, je vsaj ena od povezav (u, v) in (v, u) v E.

Recimo, da se je zgodil potisk iz u v v. Na tej točki je veljalo $h(v) = h(u) - 1$. Če želimo še kdaj potisniti iz u v v, moramo najprej potisniti iz v v u nekaj toka, saj je povezava sedaj zasičena, in veljati mora $h(v) = h(u) + 1$. Ker $h(u)$ ne pada, se mora $h(v)$ povečati za vsaj 2. Podobno premislimo, da se mora $h(u)$ povečati vsaj za 2 med potiski iz v v u, ki zasičijo povezavo. Spet uporabimo lemo 3.9, ki nam pove, da so višine vozlišč ves čas med 0 in $2|V| - 1$. Iz tega sledi, da se lahko $h(u)$ poveča za 2 manj kot $|V|$-krat med delovanjem algoritma. Ker se mora med dvema potiskoma, ki zasičita povezavo, vsaj enkrat eden od $h(u)$ in $h(v)$ povečati za 2, je tako možnih največ $2|V|$ potiskov med u in v, ki zasičijo povezavo (lahko jih je tudi manj, če se $h(u)$ oziroma $h(v)$ povečata večkrat). Ker je povezav $|E|$, tako dobimo, da je potiskov, ki zasičijo povezavo, največ $2|V||E|$. □

Lema 3.13 (omejenost števila operacij `POTISNI`, ki ne zasičijo povezave)**.** *Naj bo $G = (V, E, s, t)$ pretočno omrežje. Potem je število operacij `POTISNI`, ki ne zasičijo povezave, manjše od $4|V|^2(|V| + |E|)$.*

Že pogled na zgornjo mejo nas prepriča, da bo dokaz zgornje leme težji oziroma kompleksnejši, kot dokaz prejšnjih. Pa se ga lotimo.

Dokaz. Definirajmo najprej potencial Φ kot

$$\Phi = \sum_{v:e(v)>0} h(v),$$

torej kot vsoto višin vozlišč v presežku. Na začetku je $\Phi = 0$. Opazimo, da se vrednost Φ lahko spremeni po povišanju, potisku, ki zasiči povezavo, in potisku, ki povezave ne zasiči. Najprej bomo omejili, za koliko se lahko Φ poviša po potisku, ki zasiči povezavo, in povišanju. Potem bomo pokazali, da se po vsakem potisku, ki povezave ne zasiči, vrednost potenciala Φ zmanjša vsaj za 1. To znanje bomo potem uporabili za izračun zgornje meje števila potiskov, ki povezave ne zasičijo.

Oglejmo si najprej oba načina, na katera se potencial Φ lahko poveča. Povišanje vozlišča očitno lahko poveča Φ za največ $2|V| - 1 < 2|V|$, saj po tej operaciji ostane množica, po kateri seštevamo, enaka, vozlišče pa ima po lemi 3.9 lahko največjo višino $2|V| - 1$, za kolikor se mu lahko tudi največ poviša. Potisk, ki zasiči povezavo (u, v), lahko poveča Φ za največ $2|V| - 2 < 2|V|$. Po tej operaciji se namreč množica, po kateri seštevamo, kvečjemu poveča, saj vanjo vstopi v, u pa v njej lahko ostane ali pa ne. Če ostane, se tako lahko Φ poveča za višino vozlišča v, ki pa je lahko največ $2|V| - 2$ (največja možna višina vozlišč je po lemi 3.9 enaka $2|V| - 1$, vendar je vsaj vozlišče u višje od v, drugače potisk ne bi bil mogoč). Če pa u ne ostane v množici, torej ni več v presežku (zgodilo se je, da je veljalo $e(u) = c(u, v) - f(u, v)$), pa se Φ celo pomanjša, saj smo iz množice, po kateri seštevamo, odstranili vozlišče z večjo višino kot je višina vozlišča, ki smo ga v množico dodali.

Preostane nam obravnavati potisk, ki povezave ne zasiči. Pred potiskom je bil u v presežku, vozlišče v pa je lahko bilo v presežku, lahko pa tudi ne. Po operaciji u ne bo več v presežku, saj pri potisku, ki povezave ne zasiči, velja $\delta = e(u)$, torej je po potisku $e(u) = 0$. Vozlišče v pa je po operaciji v presežku, razen če $v = s$. To pomeni, da se je Φ zaradi operacije zmanjšal za $h(u)$, povečal pa za ali 0 ali $h(v)$. Ker velja $h(u) - h(v) = 1$, se je Φ skupno zmanjšal za vsaj 1.

Tako lahko iz zgornjih ugotovitev glede povečanja potenciala Φ, posledice 3.11 in leme 3.12 zaključimo, da se lahko Φ poveča za največ

$$(2|V|)(2|V|^2) + (2|V|)(2|V||E|) = 4|V|^2(|V| + |E|).$$

Ker je $h(u) \geq 0$ za vsak $u \in V \setminus \{s, t\}$ (vozlišči s in t ne moreta biti v presežku), je tudi $\Phi \geq 0$. To pomeni, da se Φ lahko zmanjša za največ $4|V|^2(|V| + |E|)$, kar pa ravno pomeni, da je število potiskov, ki povezave ne zasičijo, enako $4|V|^2(|V| + |E|)$. □

Sedaj pa lahko zgornje ugotovitve združimo v zaključni izrek.

Izrek 3.14 (časovna zahtevnost)**.** *Algoritem `POTISNI-POVIŠAJ` med izvajanjem naredi $\mathcal{O}(V^2E)$ osnovnih operacij.*

Dokaz. Sledi neposredno iz posledice 3.11 ter lem 3.12 in 3.13. □

4. Primer uporabe

Oglejmo si sedaj dva primera uporabe iskanja maksimalnega pretoka, s katerima se ljudje srečujejo (oziroma se srečujemo) v vsakdanjem življenju. Prvi primer bo povsem optimizacijske narave, pri katerem bomo dobesedno iskali maksimalni pretok skozi omrežje cest, pri drugem primeru pa bomo pravzaprav nek povsem drug primer, ki na prvi pogled izgleda čisto nepovezan s temo tega diplomskega dela, prevedli na iskanje maksimalnih pretokov v omrežju.

4.1. **Problem ponudbe in povpraševanja.**

Prevod imena problema iz angleščine (*circulation-demand problem*) je na prvi pogled morda ne najbolj posrečen, saj v slovenščini uporabljamo izraz ponudba in povpraševanje v kontekstu na tak način, kakor bi se v angleščini uporabljal *suply and demand*, vendar se mi zdi povsem na mestu in utemeljen.

Ta problem namreč govori o tem, da imamo p podjetij, ki proizvajajo neko dobrino, ter n naselij, ki to dobrino potrebujejo. Podjetja in naselja so med seboj povezana s cestnim omrežjem, po katerem lahko prepeljemo našo dobrino. Vsaka od cest ima kapaciteto c_i, ki pove, koliko enot dobrine lahko prepeljemo po njej (v neki časovni enoti). Naloga problema je najti pretok skozi omrežje pri dani funkciji prepustnosti $c = (c_1, c_2, \ldots, c_r)$, kjer je r število vseh cest, ki zadovolji povpraševanju.

Recimo torej, da imamo k podjetij in pišimo $\mathcal{S} = \{s_1, s_2, \ldots, s_p\}$. Imejmo n naselij in pišimo $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$. Označimo z $|s_i|$ količino dobrine, ki jo lahko proizvede podjetje s_i. Podobno označimo še $|t_i|$ količino dobrine, ki jo potrebuje naselje t_i. Recimo še, da imamo dano cestno omrežje $\mathcal{C} = \{(x, y) \mid (x \in \mathcal{S} \vee x \in \mathcal{T}) \wedge y \in \mathcal{T}\}$, kjer (x, y) pomeni, da imamo cesto od x do y. Cestnemu omrežju $\mathcal{C}$ priredimo pozitivno funkcijo prepustnosti $c = (c_1, c_2, \ldots, c_{|\mathcal{C}|})$ za katero torej velja $c_i > 0$ za vsak $i \in \{1, 2, \ldots, |\mathcal{C}|\}$.

Hitro ugotovimo, da gre za problem, pri katerem imamo več izvorov in več ponorov. Ker smo do sedaj obravnavali le omrežja z enim izvorom in enim ponorom, si oglejmo naslednjo lemo.

Lema 4.1. *Pretočno omrežje $(G, c, \mathcal{S}, \mathcal{T})$, kjer je $G = (V, E)$, $\mathcal{S} = \{s_1, s_2, \ldots, s_m\}$ množica izvirov in $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$ množica ponorov, je ekvivalentno pretočnemu omrežju (G', c', s, t), ki ga dobimo na naslednji način:*

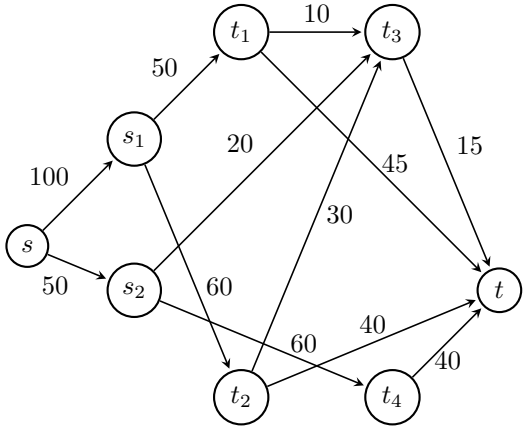
$$
\begin{aligned}
G' &= (V', E'),\\
V' &= V \cup \{s, t\},\\
E' &= E \cup \{(s, s_i) \mid s_i \in \mathcal{S}\} \cup \{(t_i, t) \mid t_i \in \mathcal{T}\},\\
c'(e) &= \begin{cases} c(e) & ; \quad e \in E \\ |s_i| & ; \quad e = (s, s_i) \\ |t_i| & ; \quad e = (t_i, t) \end{cases},
\end{aligned}
$$

torej na način, da dodamo vozlišče s, ki predstavlja skupni izvir, in vozlišče t, ki predstavlja skupni ponor. Funkcijo prepustnosti posodobimo tako, da je prepustnost povezave (s, s_i) enaka količini dobrine, ki jo podjetje s_i lahko proizvede, in da je prepustnost povezave (t_i, t) enaka povpraševanju v naselju t_i.

To pomeni, da v našem primeru dodamo vozlišči s in t, ki bosta služili kot skupen izvor in ponor. Očitno sedaj velja, da bomo zadovoljili povpraševanje v vseh naseljih natanko tedaj, ko bo maksimalen pretok skozi dobljeno pretočno omrežje enak povpraševanju vseh naselij, torej

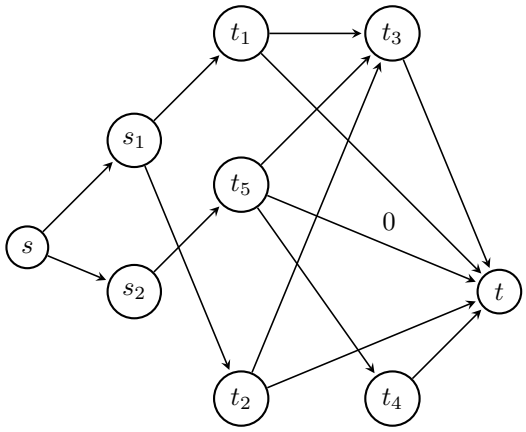
$$|f| = \sum_{t_i} c(t_i, t).$$

Za lažjo predstavo recimo sedaj, da imamo dve podjetji, s_1 in s_2, ter štiri naselja, t_1, t_2, t_3 in t_4, s takim cestnim omrežjem in funkcijo prepustnosti, kakor kaže naslednja slika.



Slika 17. Pretočno omrežje, ki ga bomo uporabili kot zgled uporabe algoritma potisni-povišaj za problem ponudbe in povpraševanja.

Gre za preprost primer. Lahko bi se zgodilo, da bi recimo na povezavi med t_1 in t_3 bilo še neko drugo naselje, ki pa ne potrebuje te dobrine. To pomeni, da ga iz obravnave lahko v splošnem kar izključimo. Recimo tako, da bi imeli naslednje pretočno omrežje:

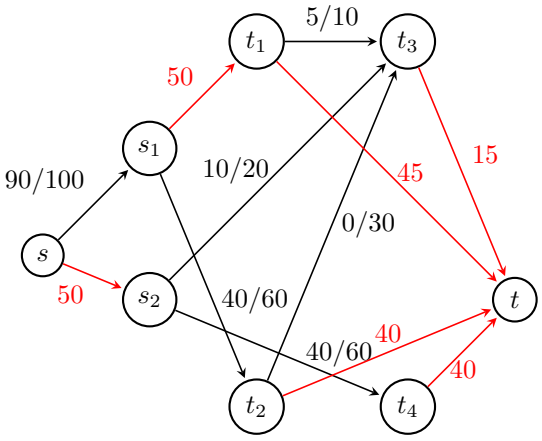


Slika 18. Pretočno omrežje, katerega bi se dalo poenostaviti.

To pretočno omrežje bi brez problema poenostavili na prejšnje, saj vozlišče t_5 očitno ne vpliva na maksimalni pretok. Vendar se takim poenostavitvam v določenih primerih rajši izognemo, saj nam vozlišča, kot so t_5, natančneje povedo, kje dejansko

dobrina potuje. V resničnem svetu bi tako to lahko recimo pomenilo, da gre tovor iz Ljubljane v Maribor preko Celja.

Vrnimo se sedaj nazaj k našemu primeru. Očitno je maksimalni pretok skozi to omrežje enak $15 + 45 + 40 + 40 = 140 = \sum_{t_i} |t_i|$, torej lahko iz podjetij s_1 in s_2 zadovoljimo potrebam vseh naselij t_1, t_2, t_3 in t_4.



Slika 19. Primer maksimalnega pretoka v danem pretočnem omrežju. Povezave, obarvane rdeče, so zasičene.

4.2. **Eliminacija ekip v baseballu.** V tem podrazdelku si bomo pogledali primer, kako nam iskanje maksimalnih pretokov pride prav v vsakdanjem življenju – in to v primeru, ki zgleda morda povsem druge narave kakor maksimalni pretoki. Ogledali si bomo, kako lahko modeliramo pretočno omrežje, s katerim lahko določimo, ali ima ekipa v prvenstvu še možnosti za končno zmago ali izenačenje na prvem mestu.

Recimo, da imamo ekipe $i \in \mathcal{T} = \{1, 2, 3, \ldots, n\}$, za nek $n \in \mathbb{N}$. Ekipa i ima v nekem določenem trenutku tekom prvenstva w_i zmag, do konca pa mora odigrati še r_i iger. Definirajmo še r_{ij} kot število iger, ki jih še mora odigrati ekipa i proti ekipi j. Očitno velja

$$r_i = \sum_{j \in \mathcal{T} \setminus \{i\}} r_{ij}.$$

Od sedaj naprej se bomo osredotočili na ekipo x. Poglejmo si sedaj, kako bi lahko hitro sklepali napačno. Naj bo q število zmag, ki jih ima v tem trenutku vodilna ekipa. Tako bi lahko naivno sklepali, da ima naša ekipa še možnosti za zmago, če velja $w_x + r_x \geq q$. Pojasnimo, zakaj je to sklepanje napačno. Recimo, kar bomo od sedaj naprej vedno predpostavljali, da ekipa x zmaga vse igre do konca prvenstva. Potem bo imela $w_x + r_x$ zmag. Recimo, da vodilna ekipa, ki ima sedaj q zmag, do konca prvenstva vse igre izgubi. Naj velja $w_x + r_x \geq q$. Vendar, ker je vodilna ekipa vse igre izgubila, se lahko zgodi, da je katera izmed ostalih ekip tako morala nabrati več kot $w_x + r_x$. Oglejmo si naslednjo tabelo.

Tabela 2. Trenutno stanje na lestvici, ki ga opisuje naš primer.

ekipa	w_i	r_i	r_{i1}	r_{i2}	r_{ix}
1	70	6		3	3
2	68	5	3		2
x	65	5	3	2	

Recimo, da sedaj naša ekipa x zmaga vse igre. Potem ima na koncu $w_x + r_x = 65 + 5 = 70 \geq 70 = q$ točk. Vendar morata med seboj igre odigrati tudi ekipa 1 in 2. Ekipa 1 ne sme zmagati nobene igre, drugače je pred našo ekipo x. Torej mora proti ekipi 2 izgubiti čisto vse 3 igre, kar pomeni, da bo ekipa 2 na koncu imela $68 + 3 = 71 > 70$ zmag. To pomeni, da kljub temu, da je $w_x + r_x \geq q$, ekipa x nima več možnosti za zmago, torej je eliminirana.

Sedaj, ko smo se prepričali, da problem ni čisto trivialen, se vrnimo nazaj k splošnemu primeru, kjer imamo n ekip. Recimo, da nas zanima, ali ima ekipa n še možnosti za zmago. Modelirajmo naslednje pretočno omrežje:

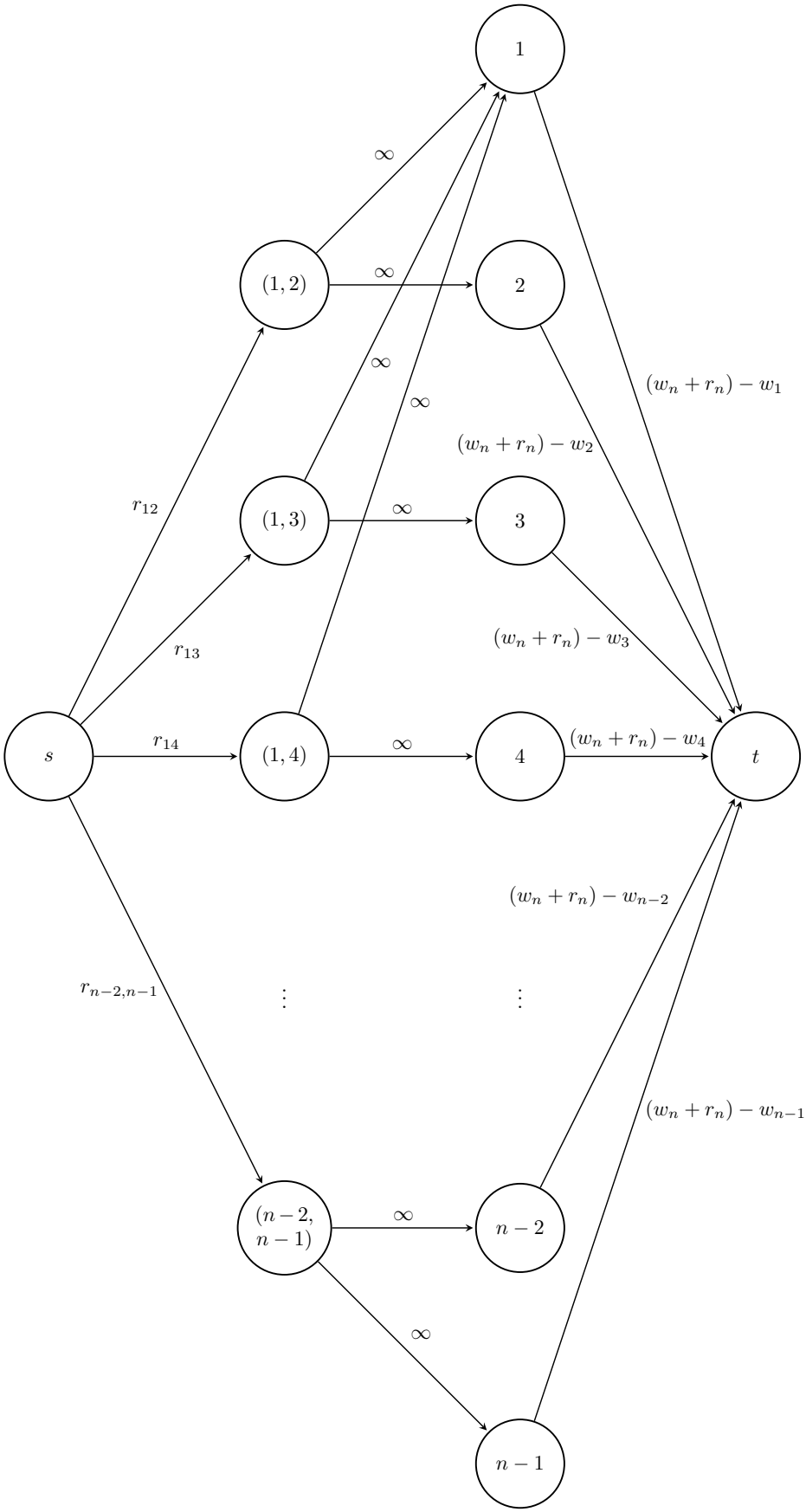
$$
\begin{aligned}
V &= s \cup \binom{\mathcal{T} \setminus \{n\}}{2} \cup \mathcal{T} \setminus \{n\} \cup t, \\
E &= \left\{ (s, (i,j)) : (i,j) \in \binom{\mathcal{T} \setminus \{n\}}{2} \right\} \cup \\
&\quad \cup \left\{ ((i,j), \alpha) : (i,j) \in \binom{\mathcal{T} \setminus \{n\}}{2} \wedge \alpha \in \{i,j\} \right\} \cup \\
&\quad \cup \{ (\alpha, t) : \alpha \in \mathcal{T} \setminus \{n\} \}, \\
c(e) &= \begin{cases} r_{ij} & ; \quad e \in \left\{ (s, (i,j)) : (i,j) \in \binom{\mathcal{T}\setminus\{n\}}{2} \right\} \\ \infty & ; \quad e \in \left\{ ((i,j), \alpha) : (i,j) \in \binom{\mathcal{T}\setminus\{n\}}{2} \wedge \alpha \in \{i,j\} \right\} \\ (w_n + r_n) - w_\alpha & ; \quad e \in \{ (\alpha, t) : \alpha \in \mathcal{T} \setminus \{n\} \} \end{cases}, \\
s &= \{\text{število tekem, ki jih morajo odigrati ekipe } 1, 2, \ldots, n-1\}, \\
t &= \{\text{število točk, ki jih bodo na koncu imele ekipe } 1, 2, \ldots, n-1.\}
\end{aligned}
$$

To izgleda izjemno grdo in nepregledno, zato si poglejmo še shemo ravnokar definiranega pretočnega omrežja, ki je ponazorjeno na sliki 20.

Opazimo, da nikjer v omrežju nimamo ekipe n, ravno za katero nas zanima, ali ima še možnosti za zmago. Ekipe n nismo pozabili, marveč smo jo nalašč izpustili. Predpostavimo namreč lahko, da če ima ekipa n še možnosti za zmago (in ne rabi zmagati vseh preostalih iger), lahko kar rečemo, da bo zmagala vse preostale igre. S tem lahko vse, kar je povezano z ekipo n, izbrišemo iz omrežja.

Razložimo, kaj to omrežje pomeni. Vozlišče s predstavlja vse igre, ki še morajo biti odigrane in ki ne vključujejo ekipe n. V naslednjem nivoju imamo vozlišča oblike (i,j), ki predstavljajo tekme med ekipama i in j, ki še morajo biti odigrane. Kot nam je že znano, r_{ij} predstavlja število takih tekem. Utež na teh povezavah je torej očitno smiselna.

Na naslednjem nivoju imamo vozlišča oblike i, kjer vsako vozlišče predstavlja ekipo i. Zopet je naša ekipa n izključena iz že obrazloženega razloga. Povezave, ki



Slika 20. Omrežje, ki modelira problem eliminacije ekip v baseballu.

vstopajo v ta vozlišča, imajo omejitev neskončno, saj pravzaprav več kot r_{ij} tako ali tako ne bodo mogla dobiti.

Na koncu vsa vozlišča, ki predstavljajo ekipe, povežemo v skupen ponor t. Uteži na povezavah, ki vstopajo v t, predstavljajo, koliko tekem lahko i-ta ekipa še zmaga, da bo naša ekipa n na koncu zmagala prvenstvo. To je ravno razlika med končnim številom zmag ekipe n, to je $w_n + r_n$ in številom zmag, ki jih je ekipa i že dosegla, torej w_i.

Kako pa si lahko pomagamo s tem omrežjem pri določanju, ali ima naša ekipa še vedno možnost za končno zmago? Označimo

$$r = \sum_{i=1}^{n-1} r_i = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n-1} r_{ij}.$$

Opazimo, da je to teoretično največji možen pretok v našem pretočnem omrežju, saj več kolikor r ne moremo spraviti iz izvira s, lahko pa se seveda zgodi, da je maksimalen pretok skozi omrežje manjši. Tako se nam porodi ideja, ki je po zgornjem razmisleku pravilna, da ima naša ekipa n še možnosti za zmago, če je vrednost maksimalnega pretoka f, torej $|f|$, enaka r.

Opomba 4.2. Če bi nas v splošnem namesto za zadnjo ekipo, torej ekipo n, zanimalo, ali lahko ekipa $k \in \{1, 2, \ldots, n\}$ zmaga prvenstvo, se stvari lotimo na povsem enak način. Ekipo k izključimo iz omrežja, formula za r pa se prepiše v

$$r = \sum_{\substack{i \in \{1,\ldots n\} \\ i \neq k}} r_i = \sum_{\substack{i,j \in \{1,\ldots,n\} \\ i<j \\ i,j \neq k}} r_{ij}.$$

Slovar strokovnih izrazov

potisni-povišaj push-relabel
pretok, tok flow
psevdopretok pseudoflow
predpretok preflow
zasičiti saturate
izvir source
ponor sink, terminal

Literatura

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest in C. Stein, *Introduction to Algorithms*, MIT Press, Massachusetts, 2009.
[2] R.J. Wilson in J.J. Watkins, *Uvod v teorijo grafov*, DMFA založništvo, Ljubljana, 1997.
[3] S. Meyers, *Effective Modern C++*, O'Reilly Media, Kalifornija, 2014.
[4] J. Kleinberg in E. Tardos, *Algorithm design*, Pearson Education Limited, London, 2013.
[5] R. Sedgewick, *Algorithms in C++*, Addison-Wesley Professional, Boston, 1998.
[6] Wikipedia, *Maximum flow problem*, dostopno na `https://en.wikipedia.org/wiki/Maximum_flow_problem`, zadnji dostop 14. marec 2018.