TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Aksel Õim

Push-relabel algoritmi formaalne tõestamine Coq raamistikus

Bakalaureusetöö (9 EAP)

Juhendaja: Kalmer Apinis

Tartu 2024

Push-relabel algoritmi formaalne tõestamine Coq raamistikus

Lühikokkuvõte:

Push-relabel algoritm arvutab transpordivõrgus välja maksimaalse voo, andes tippudele kõrgused ja ülejäägid. Algoritm üritab tippudes olevat ülejääki saata väljundisse, muutes vajadusel tippude kõrguseid ja samal ajal säilitades kolme erinevat invarianti. Bakalaureusetöö eesmärk on formaalselt tõestada Coq raamistikus push-relabel algoritm, andes ülevaate algoritmi implementatsioonist ja tõestatud lemmadest. Keskendutakse algoritmi tõestamisele, kuid tutvustakse Coq raamistikku, graafi mõisteid ja kirjeldatakse push-relabel algoritmi tööd.

Võtmesõnad:

Push-relabel algoritm, Coq, formaalne tõestus, algoritmid

CERCS: P175 Informaatika, süsteemiteooria

Formal proof of the push-relabel algorithm in Coq

Abstract:

The push-relabel algorithm calculates the maximum flow in a flow network by giving nodes height and excess values. The algorithm tries to send the excess from the nodes to the sink by continually changing the height of the nodes and preserving three different invariants. The main goal of this bachelor's thesis is to formally prove the push-relabel algorithm in Coq by giving an overview of the implementation of the algorithm and proven lemmas. The main focus is on the proof to the algorithm, but an overview of Coq and graph definitions will be given along with a description of the push-relabel algorithm.

Keywords:

Push-relabel algorithm, Coq, formal proof, algorithms

CERCS: P175 Informatics, systems theory

Sisukord

1. Sissejuhatus

Maksimaalse voo leidmise ülesanne on pärit 1954. aastast ja esimene algoritm selle lahendamiseks töötati välja 1955. aastal [1]. Maksimaalne voo leidmine toimub suunatud kaalutud graafis, kus tuleb leida suurim voog, mida saab ühest valitud tipust teise valitud tippu saata mööda graafi servasid [2]. Seda kasutataks, näiteks, kui soovime leida parima lahenduse vee transpordi probleemile, arvutades välja palju vett saab transportida mööda torustikku, et maksimaalne kogus vett jõuaks lõpp-punkti [2].

Alguses kasutati selle probleemi lahendamiseks algoritme, mis põhinevad iteratiivselt täiendavate teede leidmisel jääkvõrgus. Aastal 1986 leiutasid Andrew V. Goldberg ja Rober E. Tarjan push-relabel algoritmi [3]. Push-relabel'i eeliseks on see, et selle baasalgoritmi korrektsuse argument ei sisalda väiteid arvestatud teede kohta vaid baseerub tippude märgendamisel arvudega. Vaadates klassikalisi algoritme, on push-relabel algoritm üks kiiremaid maksimaalse voo leidmiseks - selle asümptootiline ajaline keerukus on $O(|V|^2|E|)$ [3, 4]. Võrreldes seda, näiteks, Edmonds-Karpi algoritmiga, mille ajaline keerukus on $O(|V||E|^2)$ ja mis kasutab iteratiivset täiendavate teede leidmist [3, 4].

Push-relabel algoritmi on juba implementeeritud erinevates versioonides aga meid huvitab, et implementatsioon oleks korrektne. Korrektsuse tagamiseks on üheks võimaluseks algoritmi implementeerimine keskkonnas, mis võimaldab formaalset verifitseerimist. Üks selline keskkond on Coq.

Coq on interaktiivne teoreemitõestaja, kus on võimalik kirjutada programme ja tõestada väiteid [5]. Coqi abil tõestati esmakordselt neljavärviteoreem – mistahes tasapinnalise kaardi värvimiseks piisab neljast värvist, nii et kaardiosa külg puutuks kokku vaid temast erinevat värvi naabriga [5]. Coqis on kirjutatud CompCert – formaalselt verifitseeritud optimeeriv C keele kompilaator [5].

Algoritmide formaalset verifitseerimist bakalaureuse tasemel Tartu Ülikoolis ei õpetata. Tuleb arvestada, et verifitseerimine lisab palju lisatööd ja ei saa eeldada, et bakalaureusetöö raames valmiks optimeeritud ja teiste implementatsioonidega konkurentsivõimeline lahendus. Saame aga implementeerida baasalgoritmi, mida tulevikus oleks võimalik optimeerida praktiliseks lahenduseks. Töö eesmärk on formaalselt tõestada Coq raamistikus push-relabel algoritm, mis leiab graafis maksimaalse voo, kasutades push ja relabel samme. Coq raamistikus pole seda veel tehtud.

Töö autor pakkus välja idee tõestada push-relabel algoritm Coq raamistikus ja koos juhendajaga implementeeriti algoritm Coqi. Juhendaja kirjutas algoritmi kohta käivad tõestatavad väited. Töö autor tõestas Coqis ära kirjutatud lemmad, lühemad ning lihtsamad lemmad tõestas autor üksinda aga raskemad ning pikemad lemmad arutati koos juhendajaga läbi ja autor kirjutas nendele tõestused.

Töö koosneb neljast osast. Esimeses osas tutvustatakse Coq raamistikku ja antakse ülevaade osadest taktikatest, mida tõestuste kirjutamisel kasutatakse. Teises osas kirjeldatakse graafidega seotuid mõisteid, mida läheb push-relabel algoritmi jaoks vaja. Kolmandas osas kirjeldatakse push-relabel algoritmi tööd. Neljas osa jaguneb kaheks: esimeses osas tutvustakse funktsioone, millega push-relabel algoritm Coqis kirjutati, teises osas formuleeritakse ning tõestatakse Coq raamistikus push-relabel algoritm ja selgitatakse tõestuse tähtsamaid osasid.

2. Coq

Selles peatükis tutvustatakse Coq raamistikku, andes lühiülevaate sellest, kuidas Coq töötab ning kirjeldatakse erinevaid käske ja taktikaid, mida töö käigus kasutati.

Coq on interaktiivne teoreemitõestaja, mis on abiks tõestuste kirjutamisel ja nende kontrollimisel [6]. Coqi ametliku dokumentatsiooni [5] järgi kasutab Coq kirjelduskeelt Gallina, mis on funktsionaalne programmeerimiskeel, millega saab matemaatilisi teoreeme sõnastada ja tõestada. Tõestuste loogika kontroll tehakse Gallina programmi tüübikontrolli abil, ehk kasutatakse Curry-Howardi vastavust, mis ütleb, et programmid vastavad tõestustele vastavas loogikas. Coqis on selleks loogikaks CIC ehk Calculus of Inductive Constructions. Kuna Coqis on kõik tõestused süsteemi poolt kontrollitud, siis on suurem tõenäosus, et tõestused on korrektsed. Coqi tõestusi võib kirjutada otse Gallina keeles või, soovitatavalt, läbi Ltac taktikate. Skriptimiskeel Ltac genereerib Gallina koodi ning selle abil saab tõestamist osaliselt automatiseerida. Taktikaid saab vaadata kui samme tõestuses, mis lõpuks tervikliku tõestuse valmistavad ja mida Coqi tuum verifitseerib [5]. Coqis on võimalik ise defineerida erinevaid andmetüüpe, mida vaja läheb ja kasutada olemasolevaid teeke [7].

Coqis on kindlad käsud ning taktikad, millega saab koostada funktsioone, luua muutujaid ja kirjutada tõestusi.

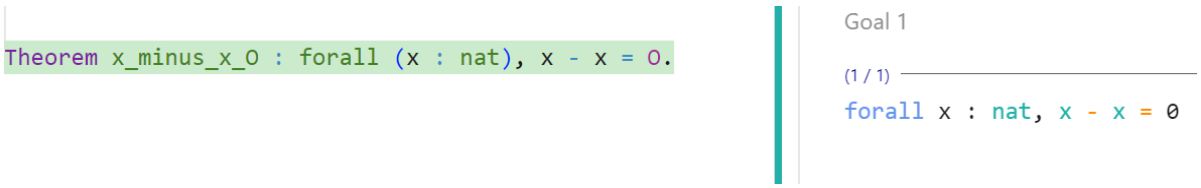
```
Definition a : nat := 5.
Definition korruta (x y : nat) : nat := x * y.

Fixpoint fact (n : nat) : nat :=
  match n with
  | O => S O
  | S n' => n * fact n'
  end.
```

Joonis 1. Erinevad defineerimise meetodid.

Joonisel 1 on näidatud kahte erinevat defineerimise meetodit *Definition ja Fixpoint*. *Definition* käsuga saab defineerida muutujaid, tüüpe, funktsioone ja terme [7]. *Fixpoint* käsuga saab koostada rekursiivseid funktsioone, mida *Definition* käsuga teha ei saa [7]. *Fixpoint* käsk vajab argumenti, mis väheneb, sest vastasel juhul ei saavutata rekursiooni baasi ja funktsioon jääkski tööle [5, 7]. Mõlema käsu korral on tegemist matemaatiliste ehk täielike puhaste funktsioonidega. Täielikult puhtaks funktsiooniks nimetatakse funktsiooni, mis samadel argumentidel tagastab alati sama väärtuse ja ei tekitada kõrvalmõjusid. Funktsioonis kasutatavatele muutujatele ja funktsiooni väljundile tuleb määrata tüüp, *Definition a : nat := 5*

ütleb, et muutuja a on naturaalarvu tüüpi ja tema väärtus on 5. Joonisel 1definitsioonis *korruta* on mõlemale muutujale x ja y omistatud naturaalarvude tüüp kasutades notatsiooni *(x y : nat)* ning väljundile on samuti omistatud naturaalarvude tüüp. Väljundi tüüp määratakse notatsiooniga *: nat* ja *:= x * y* määrab ära funktsiooni keha. *Fixpoint* definitsioonis on kasutatud Peano aritmeetikat, mis on funktsionaal programmeerimiskeeltes levinud, kus O = 0 ja S tähistab järgarvu funktsiooni (ingl *successor*) ehk S(1) = 2. *Fixpoint* definitsioonis sobitatakse muutuja n võimalustega O ja S n' kasutades süntaksit *match n with* ehk kui n = 0, siis tagastatakse väärtus 1 ja muudel juhtudel kutsub S n' uuesti välja funktsiooni *fact* argumendiga n-1.



Joonis 2. Teoreemide kirjutamine Coqis.

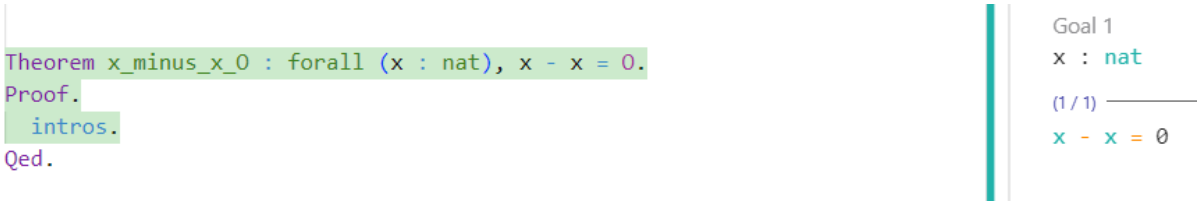
Joonisel 2 on vasakul tõestatava teoreemi nimeks pandud *x_minus_x_O*, mis väidab, et iga naturaalarvu x korral kehtib võrdus x - x = 0. Peale käsku *Theorem* siseneb Coq tõestusrežiimi (ingl p*roof mode*) ja saab kasutada taktikaid, millega teoreemi tõestada, tõestusrežiim lõppeb käsuga *Qed* [5]. Paremal on näha hetke tõestus seisundit (ingl p*roof state*), kus on välja toodud lokaalne kontekst, mis asub joonest üleval, ja joone all on eesmärk, mida tuleb tõestada [5]. Lokaalsesse konteksti tulevad hüpoteesid, lokaalsed muutujad ja definitsioonid, mida saab tõestamiseks kasutada [5]. Tõestus seisundi järg on märgitud rohelisega ehk hetkel on näha seisundit, mis on peale käsu *Theorem* rakendamist.

Peatükis tutvustati Coq raamistikku, selgitati, mis on käskude *Definition* ning *Fixpoint* vahe, näidati kuidas kirjutada teoreeme ja milline näeb välja tõestusrežiim. Järgmistes alapeatükkides tutvustatakse erinevaid taktikaid, mida töös kasutati.

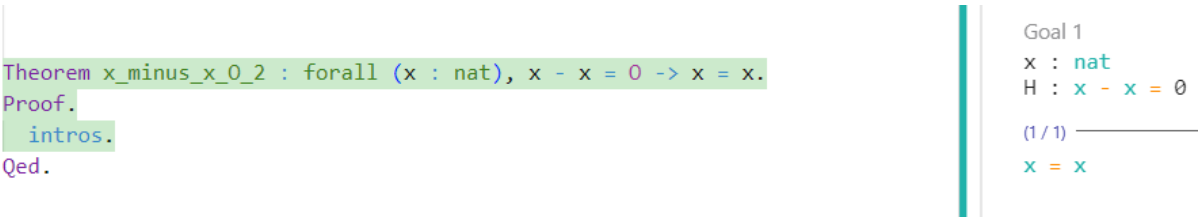
2.1 Eesmärgi koostamise taktikad

Coqis on kasutusel erinevad taktikad, millega saab eesmärke koostada ja muuta. Töös antakse ülevaade ainult väiksest osast taktikatest, Coqi dokumentatsioonist [5] saab teiste taktikate kohta rohkem lugeda. Siin alampeatükis tutvustatakse *intros, induction, destruct* ja *apply* taktikat.

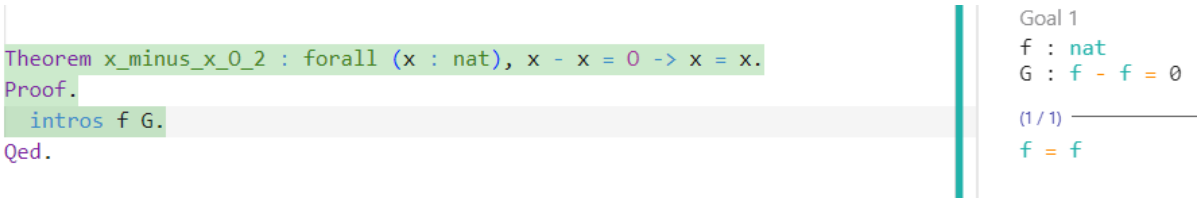
Intros taktika toob väites olevad muutujad ja implikatsioonide eeldused konteksti, kui need eksisteerivad [5, 8]. *Intros* taktikat kasutades on võimalik muutujatele ja hüpoteesidele nimed anda, et tõestust kirjutades oleks lihtsam järge pidada.



Joonis 3. *Intros* taktika kasutamine.



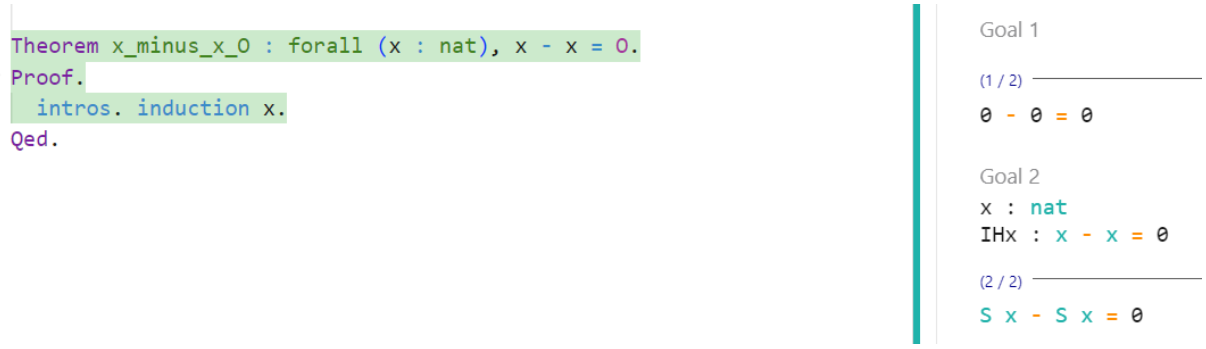
Joonis 4. *Intros* taktika kasutamine teoreemil, kus on implikatsioon.



Joonis 5. *Intros* taktika kasutamine koos nimedega.

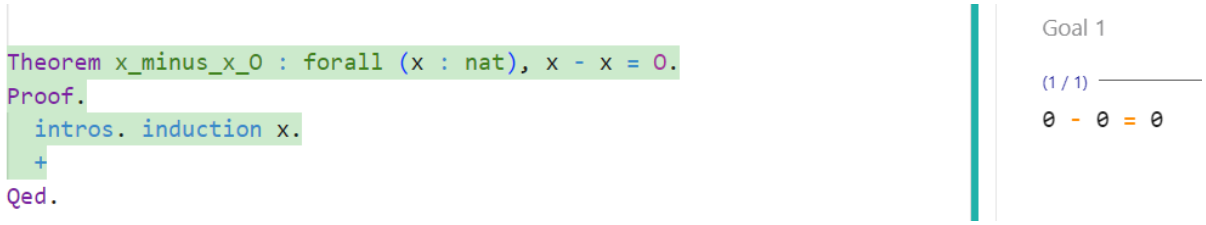
Joonisel 3 peale *intros* taktika kasutamist toodi kõik *forall* muutujad konteksti ning nüüd tuleb tõestada eesmärk x – x = 0, lokaalses kontekstis on märgitud, et x on naturaalarv kasutades notatsiooni *x : nat*. Joonisel 4 toodi lokaalsesse konteksti lisaks *x : nat* notatsioonile veel teoreemis *x_minux_x_O_2* olev implikatsiooni eeldus, milleks on hüpotees H: x – x = 0. Nimede andmist muutujatele ning hüpoteesidele näeb joonisel 5, kus muutuja x on asendatud muutuja nimega f ja hüpoteesile H on antud nimi G. Kasutatud on veel käsku *Proof*, mille võib ära jätta, aga see muudab tõestuste lugemise lihtsamaks, sest siis on teada kust algab teoreemi tõestus.

Induction taktika on induktsiooni rakendamine valitud muutujale ja see loob kaks eesmärki, mida tuleb tõestada, üheks eesmärgiks saab induktsiooni baasi tõestamine ja teiseks eesmärgiks induktsiooni sammu tõestamine [8]. Teise harusse tekib induktsiooni hüpotees, mida saab tõestamisel kasutada.



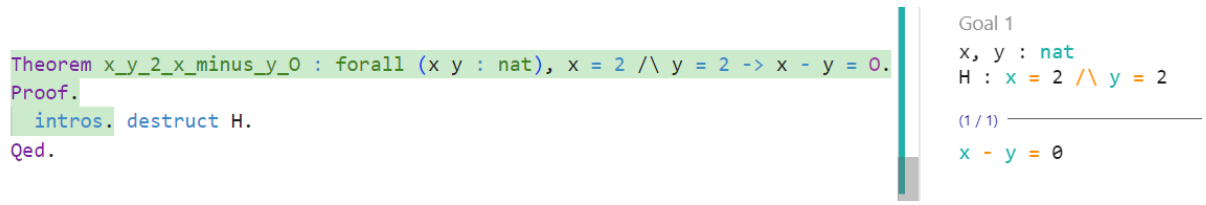
Joonis 6. *Induction* taktika kasutamine.

Joonisel 6 on näha tõestuse seisundit peale *induction x* kasutamist ja nüüd tuleb tõestada ära kaks haru, milleks on induktsiooni baas, kus x = 0 ja induktsiooni samm, kus x = S x'. Teises harus on näha, et on lisatud induktsiooni hüpotees IHx : x – x = 0. Tekkinud harusid saab eraldi tõestada kasutades + märki. Märkidega +, - ja * saab eraldada eesmärgis olevad harud, et neid ükshaaval tõestada. Joonisel 7 on võetud ette tõestamiseks esimene haru 0 – 0 = 0.

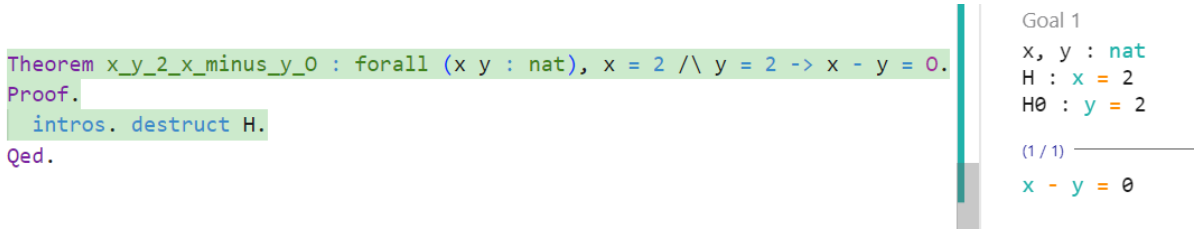


Joonis 7. Harude eraldamine.

Destruct taktikaga on võimalik eraldada kaks erinevat tõestuse poolt, näiteks saab eraldada konjunktsiooni, disjunktsiooni, implikatsiooni ja ekvivalentsi [8]. Veel saab *destruct* taktikat kasutada ka siis, kui on vaja kaht erinevat haru, kus esimeses harus on kaks muutujat omavahel võrdsed ja teises harus mittevõrdsed, see võimaldab tõestada tingimuslause *if* harusid. Joonisel 8 on näha, et hüpoteesis H on konjunktsioon x = 2 /\ y = 2, peale *destruct* taktika kasutamist on joonisel 9 näha, et konjunktsioon tehti kaheks, kus hüpoteesis H on konjunktsiooni esimene pool ja hüpoteesis H0 konjunktsiooni teine pool. Erinevalt *induction* taktikast *destruct* ei loo uut hüpoteesi vaid lihtsalt teeb valitud hüpoteesi mitmeks osaks.

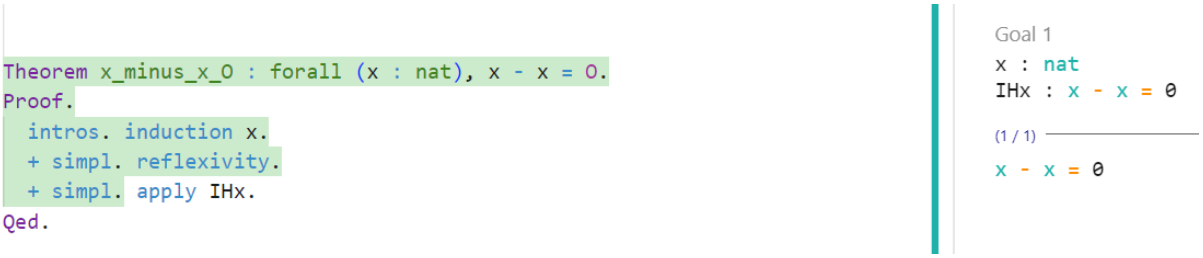


Joonis 8. H enne *destruct* taktika kasutamist.

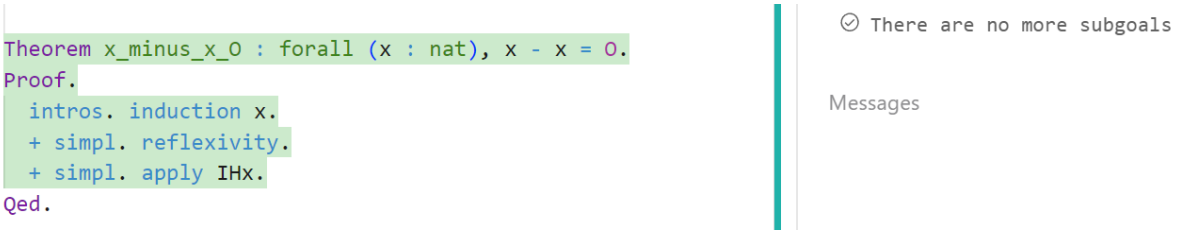


Joonis 9. H ja H0 peale *destruct* taktika kasutamist.

Apply taktika kasutab implikatsiooni ning hüpoteesi, et muuta eesmärke ja hüpoteese [8]. Rakendades hüpoteesi *apply* taktikaga eesmärgile on võimalik eesmärk ära tõestada või kui seda rakendada teisele hüpoteesile, siis saab uusi hüpoteese luua. Joonisel 11 on näha, et peale hüpoteesi IHx : x – x = 0 rakendamist eesmärgile x – x = 0, mis on näidatud joonisel 10, sai haru tõestatud.



Joonis 10. Tõestuse seisund enne *apply* kasutamist eesmärgil.



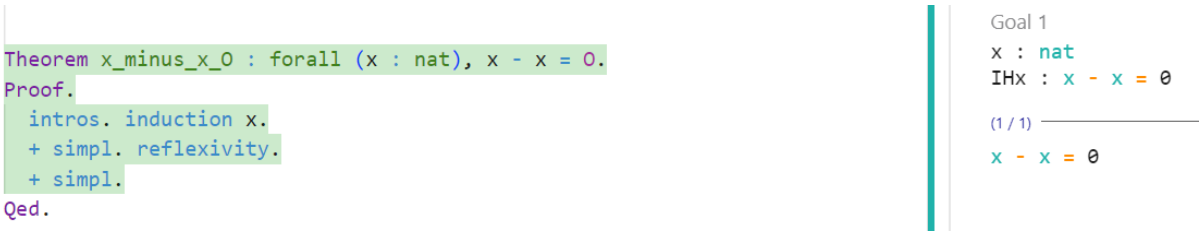
Joonis 11. Tõestuse seisund peale *apply* kasutamist eesmärgil.

Alapeatükis tutvustati Coqi taktikaid, millega saab eesmärke koostada ja tõestada, nendeks olid *intros*, *induction*, *destruct* ja *apply*. Järgmises alapeatükis tutvustatakse taktikaid, millega saab eesmärke ning hüpoteese lihtsustada ja muuta.

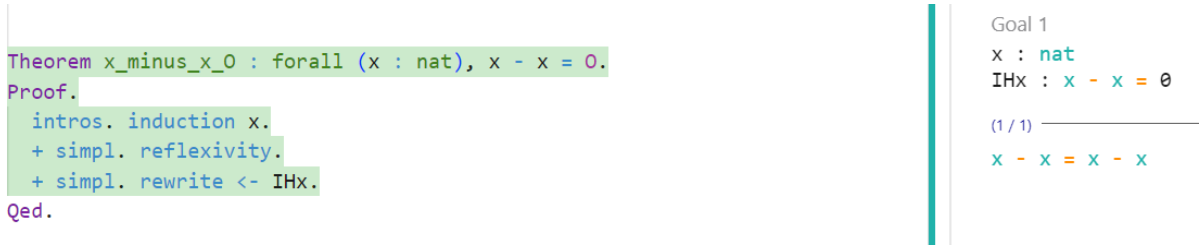
2.2 Eesmärgi muutmise taktikad

Simpl, *rewrite*, *unfold* taktikad muudavad hetke tõestuse olekut, muutes eesmärke või hüpoteese ning lihtsustades väljendeid või muutes tõestuse seisundi konteksti [8]. Taktika *simpl* lihtsustab etteantud eesmärki, hüpoteesi või definitsiooni [8]. Joonisel 12 lihtsustati *simpl* taktikat kasutades eesmärk S x – S x = 0 eesmärgiks x – x = 0. Taktika *simpl* rakendas lahutamise definitsiooni kasutades Peano aritmeetika reegleid, kus järgarvude funktsioonide

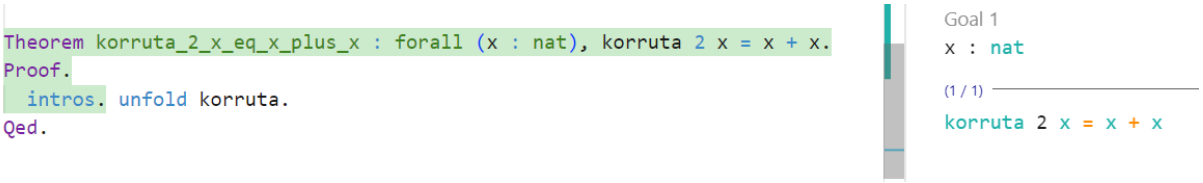
kutsed taandatakse paarikaupa. Taktika *rewrite* laseb kirjutada võrduse vasaku või parema poole üle eesmärgis või hüpoteesis [8], joonisel 13 kirjutati eesmärgis oleva võrduse parem pool üle kasutades rewrite ← IHx. Selleks, et võrduse vasakut poolt üle kirjutada tuleb kasutada rewrite → IHx, ehk nool määrab ära suuna, kumb võrduse pool üle kirjutatakse. Taktika *unfold* vahetab termi definitsiooniga välja [8]. Joonistel 14 ja 15 on näha, mida teeb taktika *unfold*, joonisel 14 on eesmärk kujul *korruta* 2 x = x + x, aga peale taktikat *unfold* vahetatakse joonisel 15 eesmärgis *korruta* term välja tema definitsiooniga, milleks on 2 * x.



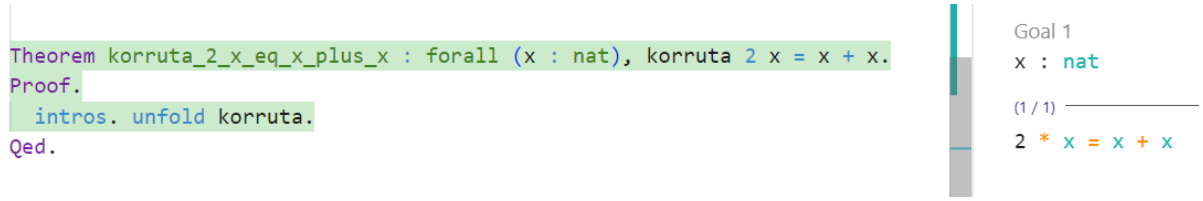
Joonis 12. Tõestuse seisund peale *simpl* taktika kasutamist.



Joonis 13. Tõestuse seisund peale *rewrite* taktikat, kirjutades parema poole üle



Joonis 14. Eesmärk enne *unfold* taktika kasutamist.



Joonis 15. Eesmärk peale *unfold* taktika kasutamist

Peatükis tutvustati taktikaid, mis muudavad tõestuste eesmärki, nendeks olid *simpl*, mis lihtsustas eesmärki, hüpoteesi või definitsiooni, *rewrite*, mis kirjutas eesmärgi või hüpoteesis oleva võrduse üle ja *unfold*, mis vahetas termi definitsiooni vastu välja.

3. Graaf

Selles peatükis tutvustatakse graafidega seotuid mõisteid ja erinevaid graafi liike, sest järgmistes peatükkides hakatakse neid mõisteid kasutama, et tutvustada push-relabel algoritmi ning kirjeldada selle kirja panemiseks kasutatud funktsioone ja tõestusi.

3.1 Graafi mõisted

Reimo Palmi [9] järgi nimetatakse **suunatud graafiks** paari $G = (V, E)$, kus V on mittetühi tippude hulk ja E on järjestatud tipupaaride hulga $V \times V$ alamhulk. **Ahel** on tippude järjend v_0, v_1, ..., v_k, kus iga kaks järjestikust tippu on servaga ühendatud. Graaf $G = (V, E)$ on **sidus**, kui iga tipu $v, u \in V$ korral leidub ahel, mis neid tippe ühendab. Suunatud graafis nimetatakse **sisendiks** tippu, kust servad ainult väljuvad ja **väljundiks** tippu, kuhu servad ainult sisenevad. Graafi $G = (V, E)$ **alamgraafiks** nimetatakse graafi $G' = (V', E')$, kus $V' \subseteq V$ ja $E' \subseteq E$ ehk graafil G' on samad tipud ning servad, mis graafil G, aga osad tipud ja servad on kustutatud [9].

Alapeatükis anti definitsioonid mõistetele, mida hakatakse järgmistes peatükkides kasutama, edasi tutvustatakse erinevaid graafi liike ja nendega seonduvaid mõisteid.

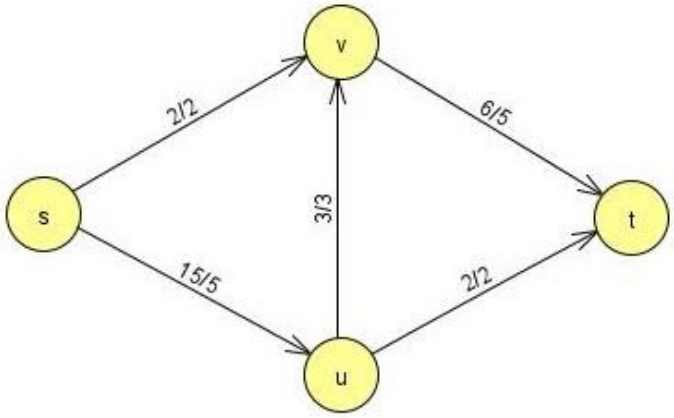
3.2 Transpordivõrk

Transpordivõrguks (ingl *flow network*) nimetatakse nelikut $N = (G, c, s, t)$, kus G on sidus suunatud graaf $G = (V, E)$, milles V on graafi tippude hulk ja E servade hulk, c on serva läbilaskevõime (ingl *capacity)* funktsioon $c: E \rightarrow R^+$, kus R^+ tähistab positiivseid reaalarve, sest servadel saab olla ainult positiivne läbilaskevõime ning tipud $s, t \in V$ on vastavalt sisend ja väljund [2, 10]. Kuigi raamatute definitsioonides on kasutatud reaalarve, siis paljude ülesannete jaoks on piisav, kui kasutada täisarve. Selle töö implementatsioonis on kasutatud ratsionaalarve, sest need on Coqis paremini toetatud. Igal serval $e \in E$ graafis G on voog (ingl *flow)*, mis on defineeritud funktsiooniga $f: E \rightarrow R^+ \cup 0$ ehk $f(e)$ näitab serva e voogu ja funktsioon f rahuldab tingimusi [2, 4, 10]:

- Läbilaskevõime piirang: $\forall e \in E: 0 \leq f(e) \leq c(e)$ [2, 4, 10] ehk serva voo väärtus ei saa olla negatiivne ega suurem kui serva läbilaskevõime.
- Voo säilivus: $\forall v, u \in V \backslash \{s, t\}: \sum_{(v,u)\in E} f(v, u) = \sum_{(u,v)\in E} f(u, v)$ [2, 10] ehk tippu sissetuleva voo väärtus on võrdne tipust väljamineva voo väärtusega tippude v ja u vahel, välja arvatud tippudes s ja t, mis on vastavalt sisend ja väljund [4].

Voog näitab palju serva *e* läbilaskevõimest kasutatakse. Graafi koguvoo (ingl *total flow)* saab leida valemiga $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$, ehk kui sisendist väljuvate voogude summast maha lahutada sisendisse sisenevate voogude summa, siis saame graafi koguvoo [2].

Joonisel 16 on transpordivõrk, kus servadel on läbilaskevõime ja voog. Näiteks serva (s, u) läbilaskevõime on 15 ja voog 5 ning serva (u, t) läbilaskevõime on 3 ja voog 3.



Joonis 16. Transpordivõrk.

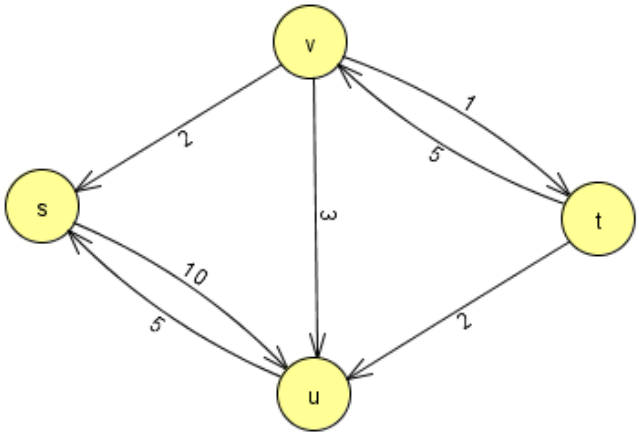
Alapeatükis anti ülevaade graafi liigist, milleks on transpordivõrk, kus servadel on läbilaskevõime ja voog. Selleks, et graafis olev voog oleks korrektne pidi see rahuldama kahte erinevat tingimust, milleks oli, et serval olev voog ei saa olla negatiivne ega suurem kui serva läbilaskevõime ja sissetuleva voo väärtus peab olema võrdne väljamineva voo väärtusega.

3.3 Jääkvõrk

Jääkvõrguks (ingl *residual network*) nimetatakse graafi $G_f = (V, E_f)$, kus E_f hulgas on servad, mis on saadud transpordivõrgu $G = (V, E)$ hulgast E [2, 4]. Serv lisatakse hulka E_f, kui antud serval on jääk positiivse väärtusega [2, 4].

Jääkvõrgul G_f on järgmised omadused:

- G_f tippude hulk on sama, mis graafil G [4].
- Iga serva $e = (u, v)$ jäägi (ingl *residual capacity*) lisame jääkvõrku G_f, kui $f(e) < c(e)$, seega jäägi saame valemiga $c_f(e) = c(e) - f(e)$ [2, 4]. Jääkvõrku G_f saame lisada servad ainult siis, kui $c_f(e) > 0$ [4].
- Iga serva $e = (u, v)$, kui $e \in G$ ja $f(e) > 0$ korral lisame serva $e' = (v, u)$ jääkvõrku G_f läbilaskevõimega $f(e)$ [4].

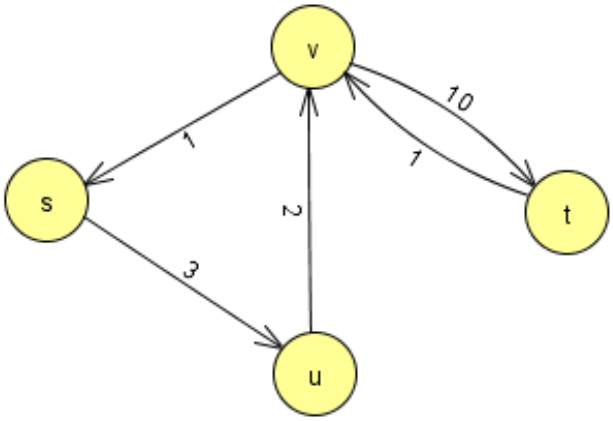


Joonis 17. Jääkvõrk.

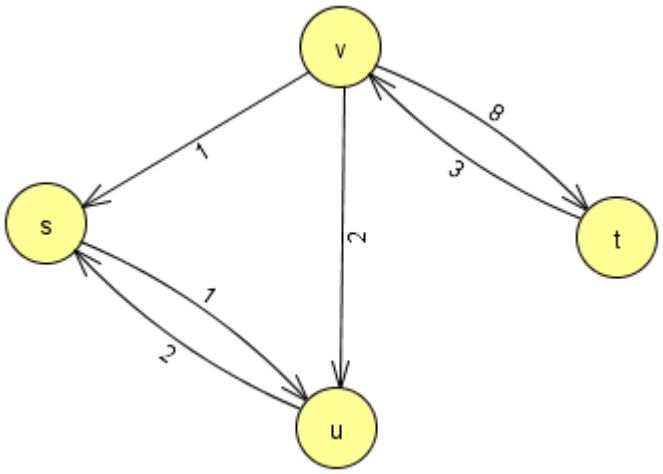
Joonisel 17 on tehtud jääkvõrk transpordivõrgust, mis oli toodud joonisel 16. Jääkvõrgus tähistab serv (s, u) serva jääki ning serv (u, s) voogu ehk serva (s, u) jääk on 10 ja voog 5. Kuna transpordivõrgus oli serva (u, v) läbilaskevõime ära kasutatud, siis jääkvõrgus on ainult serv (v, u) voo väärtusega 3.

3.4 Täiendav tee

Täiendav tee (ingl *augmenting path*) jääkvõrgus $G_f = (V, E_f)$ on ahel sisendist väljundisse mööda servasid, millel on veel jääk [2]. Täiendavas tees p saame p jäägi $c_f(p) = min\{c_f(u,v): (u,v)\}$ ehk valime täiendaval teel kõige väiksema jäägiga serva [2]. Kui täiendavat teed jääkvõrgus G_f ei leidu ehk kui sisendi ja väljundi vahel puudub ahel, siis antud voog on maksimaalne [10].



Joonis 18. Täiendav tee jääkgraafis.



Joonis 19. Jääkvõrk peale täiendava tee leidmist.

Joonisel 18 on täiendavaks teeks servad (s, u), (u, v) ja (v, t), täiendava tee jäägiks saab serva (u, v) läbilaskevõime, milleks on väärtus 2, sest see on leitud teel miinimum väärtus. Joonisel 19 on näha jääkvõrgu seisundit peale täiendava tee leidmist, serv (u, v) läbilaskevõime on ära kasutatud ning nüüd on nende kahe tipu vahel ainult serv (v, u), mis näitab voogu. Serva (s, u) jääk on 1 ning voog 2 ja serva (v, t) jääk on 8 ning voog 3.

3.5 Eelvoog

Eelvoog on funktsioon f, mis seab igale servale vastavusse positiivse reaalarvu ehk $f: E \rightarrow R^+$, kus E on graafi $G = (V, E)$ servade hulk [4]. Eelvoog peab rahuldama järgmisi tingimusi:

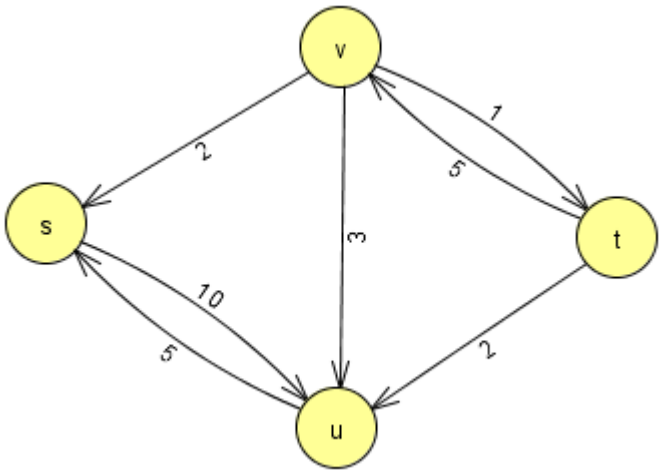
- $\forall e \in E: 0 \leq f(e) \leq c(e)$ ehk serva voo väärtus ei saa olla negatiivne ega suurem kui serva läbilaskevõime [4].
- $\forall v, u \in V \backslash \{s\}: \sum_{(v,u)\in E} f(v,u) \geq \sum_{(u,v)\in E} f(u,v)$ ehk tippu sissetuleva voo väärtus on suurem või võrdne tipust väljamineva voo väärtusega tippude v ja u vahel, välja arvatud tipus s [4].

Eelvoog pidi täitma kahte tingimust, need on, et serva voo väärtus ei saa olla negatiivne ega suurem kui serva läbilaskevõime ja tippu siseneva voo väärtus võis olla suurem või võrdne väljamineva voo väärtusega. Eelvoog on voog, mille tingimused on leevemad

3.6 Maksimaalne voog

Maksimaalse voo leidmine on selleks, et leida parim lahendus näiteks vee transpordi probleemile, arvutades välja palju vett saab transportida mööda torustikku, et maksimaalne kogus vett

jõuaks lõpp-punkti [2]. Selleks koostatakse alguses transpordivõrk, kus sisendtipuks on alg-punkt, kust vett hakatakse mööda torusid saatma ning väljundiks lõpp-punkt kuhu vett saadetakse ja tipud punktid, kus vesi erinevatesse torudesse edasi saadetakse. Transpordivõrgus olevad servad saavad läbilaskevõimeks torude läbilaskevõime. Peale transpordivõrgu koostamist on teada palju on iga serv võimeline vett transportima, siis saab koostada jääkvõrgu, millel saab leida täiendavaid teid ehk hakkame vett mööda torusid saatma kuni enam ei leidu teed sisendist väljundisse. Transpordivõrgu maksimaalseks vooks nimetatakse võrgu servade maksimaalset läbilaskevõimet sisendist väljundisse [10, 11].



Joonis 20. Maksimaalne voog graafis.

Joonisel 20 oleva jääkvõrgu maksimaalne voog on 7, mis saadakse kui liita servade (t, u) ja (t, v) vood, milleks on vastavalt 2 ja 5. Tegemist on maksimaalse vooga, sest jääkvõrgus enam ei leidu täiendavat teed sisendist s väljundisse t.

4. Push-relabel algoritm

Kogu järgnev peatükk toetub Jon Kleinberg ja Éva Tardos [4] teosele, mis kirjeldab push-relabel algoritmi. Push-relabel algoritmi vältel on igal tipul kõrgus $h(u), \forall u \in V$ ja ülejääk $x_f(u) = \forall u, v \in V: \sum_{(u,v)\in E} f(u,v) - \sum_{(v,u)\in E} f(v,u)$. Ülejääk on tippu saadetud voog, mida üritatakse teistesse tippudesse edasi saata kuni jõutakse väljundisse. Alguses toimub initsialiseerimine, peale mida kehtivad kogu algoritmi vältel kaks järgnevat invarianti:

1. $h(s) = |V|$, ehk sisendtipu kõrgus on alati võrdne tippude arvuga $|V|$.
2. $h(t) = 0$, ehk väljundtipu kõrgus on alati 0.

Seejärel hakatakse tippude kõrguseid muutma nii, et kehtiks kolmas invariant:

3. $\forall (u,v) \in G_f, h(u) \leq h(v) + 1$, ehk lõpptipu kõrgus saab olla maksimaalselt üks vähem kui algtipu kõrgus.

Kolmanda invariandi kohaselt saab ülejääki edasi saata ainult tippudesse, mis asuvad kõrguse poolest üks aste madalamal. Algoritmi vältel säilitatakse eelvoo tingimused ning alles lõpus, kui üheski tipus ei ole enam ülejääki saab eelvoost voog ja algoritm lõpetab töö. Tippudes olev ülejääk, mida ei õnnestunud väljundisse saata, saadetakse töö käigus tagasi sisendisse.

Kui invariandid ja voo tingimused on rahuldatud, siis on leitud graafi maksimaalne voog. Selle saab tõestada vastuväitelise tõestusega: eeldame, et jääkvõrgus G_f leidub täiendav tee sisendist väljundisse, siis selle tee pikkus saab maksimaalselt olla $|V|$ - 1. Esimesest invariandist saame, et sisendi kõrgus on $|V|$, teisest invariandist, et väljundi kõrgus on 0 ja kolmandast invariandist saame, et tippude kõrgus saab väheneda maksimaalselt ühe võrra. Kuna me alustame sisendist, mille kõrgus on $|V|$ ning liigume alla korraga ainult ühe sammu võrra ja peame jõudma väljundisse, mille kõrgus 0, siis tekib vastuolu, sest sisendist väljundisse minnes peab tee pikkus olema $|V|$, aga maksimaalne tee pikkus saab olla $|V|$ - 1.

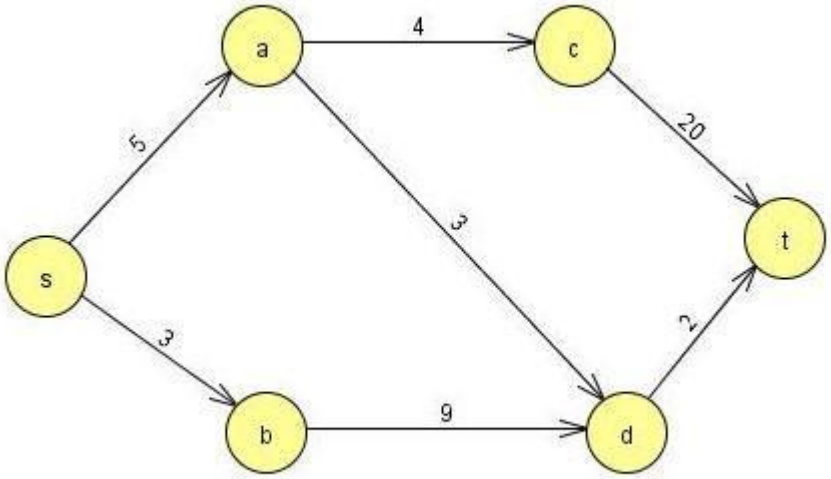
4.1 Initsialiseerimine

Algoritmi alguses toimub algväärtustamine, mille sammud on järgnevad:

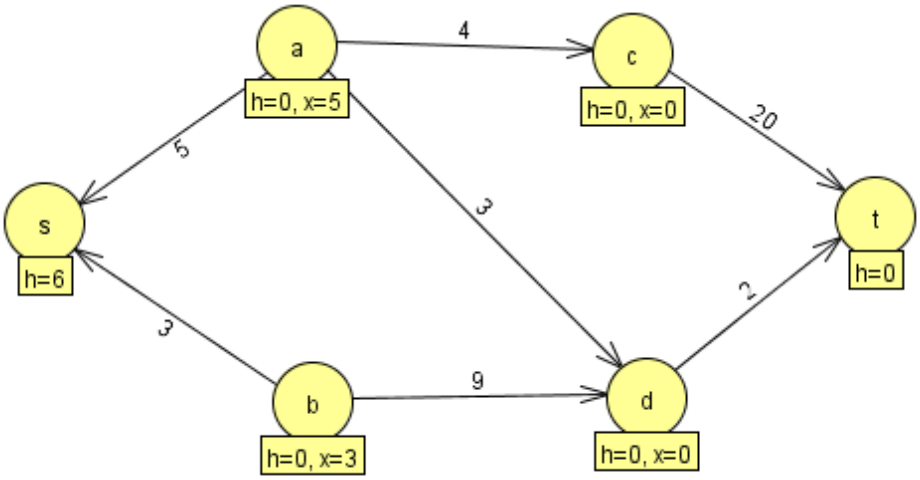
- $h(u) = \begin{cases} n & \text{, kui } u = s \\ 0 & \text{, muidu} \end{cases}$, siin $n = |V|$, ehk sisend saab kõrguseks $|V|$ ja kõik teised tipud saavad kõrguseks 0.

- $f(e) = \begin{cases} c(e) & ,\forall e = (s,v) \\ 0 & ,\text{muidu} \end{cases}$, ehk iga serv, mis väljub sisendist saab vooks antud serva maksimaalse läbilaskevõime väärtuse ja ülejäänud servad saavad voo väärtuseks 0.

Peale initsialiseerimist on rahuldatud kõik kolm invarianti, milleks on, et sisendi kõrgus on võrdne tippude arvuga, väljundi kõrgus on 0 ja lõpptippude kõrgused on maksimaalselt üks vähem kui algtippude kõrgused. Kuna sisendist väljuvate servade läbilaskevõime kasutatakse ära, siis need servad enam ei kuulu jääkvõrku ehk $(s,v) \notin G_f$ ja teiste servade $(u,v) \in G_f$ puhul $h(u) \leq h(v) + 1$ kuna $0 \leq 0 + 1$, seega mõlemal juhul ei rikuta kolmandat invarianti.



Joonis 21. Transpordivõrk G, mille peal algoritmi rakendatakse.



Joonis 22. Jääkvõrgu G_f seisund peale initsialiseerimist.

Joonisel 21 on kujutatud transpordivõrk G, millele hakkab push-relabel algoritm maksimaalset voogu leidma. Alguses tehakse algoritmi poolt initsialiseerimine ning graafist G tehakse jääkvõrk G_f ja tippudele lisatakse kõrgus h ning ülejääk x ja sisendist väljuvate servade läbilaskevõimed kasutatakse ära. Seisundit peale initsialiseerimist on näha jooniselt 22, kus tipul a on ülejääk 5 ning tipul b ülejääk 3 ja graafis pole enam servasid (s, a) ja (s, b) vaid servad (a, s) ja (b, s).

4.2 Push ja Relabel operatsioonid

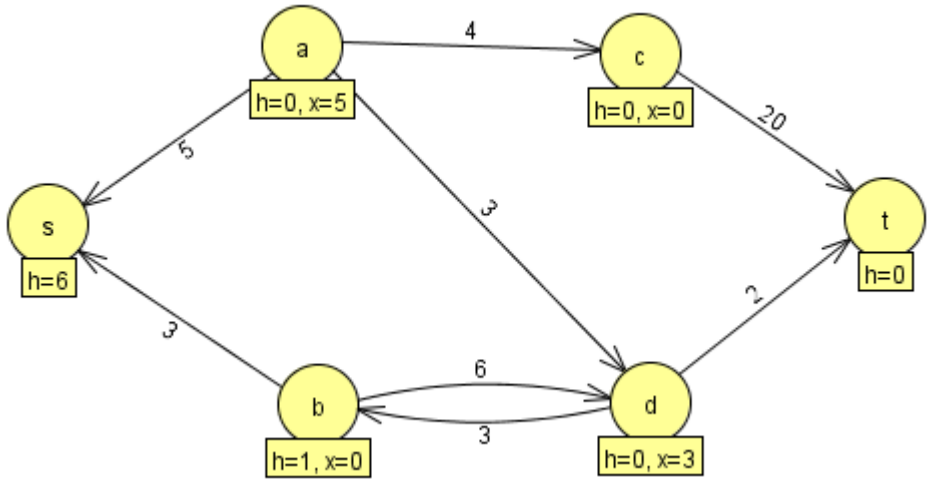
Algoritmi push samm on järgnev:

- Vali $e = (u, v) \in G_f$, $u \neq t$, kus $x_f(u) > 0$ nii, et $h(u) = h(v) + 1$.
- Olgu $\Delta = min\{x_f(u),\ c(e)\}$, ehk tipu ülejäägist ja serva läbilaskevõimest valitakse miinimum.
- Suurenda $f(e)$ väärtust Δ võrra.

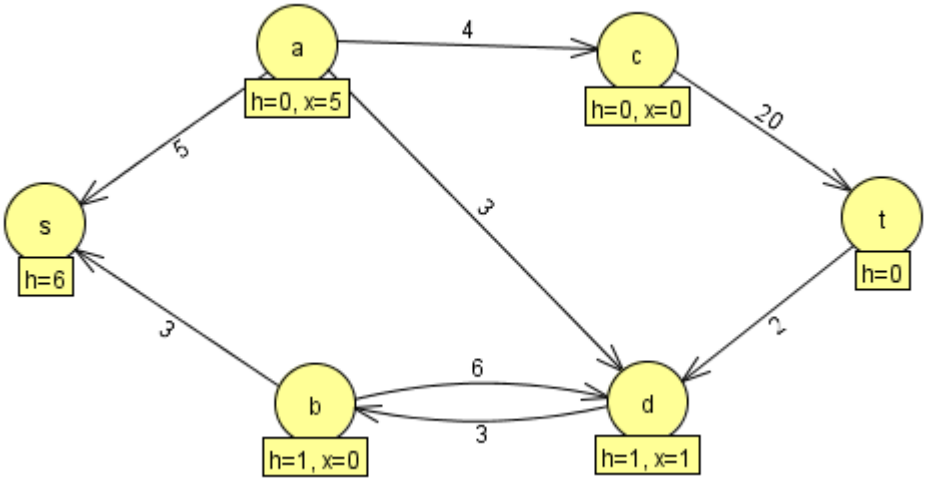
Algoritmi relabel samm tehakse, siis kui $u \neq t$ ja $x_f(u) > 0$, aga push sammu pole võimalik teha:

- Leia vähima kõrgusega tipp v, kus $(u, v) \in G_f$.
- Sea $h(u)$ uueks kõrguseks $h(v) + 1$.

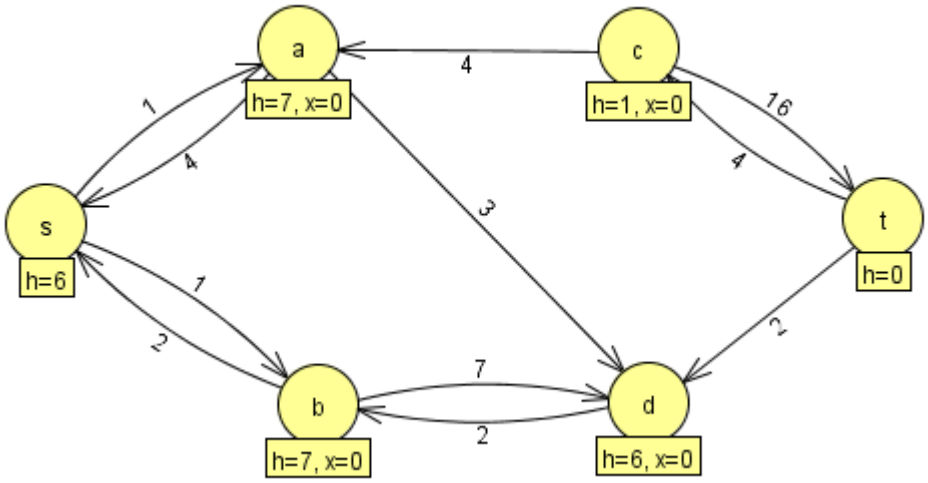
Neid kahte sammu tehakse nii kaua kuni veel leidub tipp $u \neq t$, millel on ülejääk $x_f(u) > 0$. Nüüd kui initsialiseerimine on joonisel 22 tehtud, siis leitakse tipp, mis omab ülejääki ja suurimat kõrgust. Näites on valikus tipud a ja b. Joonisel 23 on näha seisundit peale seda, kui valiti tipp b ning tehti relabel samm tipul b ja push samm tippu d. Tipu b kõrgust suurendati ühe võrra ning tipus olev ülejääk, mis oli 3, saadeti tippu d. Nüüd on tipus b ülejääk 0 ja tipus d ülejääk 3. Peale seda on valikus tipud a ja d, joonisel 24 valiti järgmiseks push sammu algustipuks tipp d. Neid samme korratakse nii kaua kuni enam ei leidu ühtegi tippu, mis omaks ülejääki. Jääkvõrgu lõppseisu algoritmi lõpus saab näha joonisel 25.



Joonis 23. Jääkvõrgu seisund peale sammu relabel(b) ja push(b,d).



Joonis 24. Jääkvõrgu seisund peale sammu relabel(d) ja push(d,t).



Joonis 25. Jääkvõrgu seisund algoritmi lõpus.

Algoritmi lõpus on säilitatud kaks esimest invarianti $h(s) = |V|$ ja $h(t) = 0$. Igas tipus on täidetud ka voo tingimused, milleks on läbilaskevõime piirang, mis nõudis, et ühegi serva voo väärtus ei ole suurem kui läbilaskevõime ja voo säilivus, igas tipus sissetulev voog on võrdne väljamineva vooga. Kuna kõik invarinadid on säilitatud ning voo tingimused täidetud, siis ongi leitud graafi maksimaalne voog.

Järgmises peatükis tõestatakse algoritmi osaline korrektsus näidates, et algoritmi töö lõpus saadud voog on korrektne ja invariandid on säilitatud kogu algoritmi töö vältel.

5. Push-relabel algoritmi tõestus

Järgnev peatükk tutvustab ainult formaliseeringut, kõiki tõestusi saab näha GitHubis https://github.com/Aksel2/push-relabel või programmi failis *PR.v*, mis on esitatud lisas 1.
Push-relabel algoritmi kirjutamisel ja tõestamisel on kasutatud Coq versiooni 8.19.0.
Coqis on kasutatud push-relabel algoritmi realiseerimiseks järgnevaid parameetreid:

- vs - tippude hulk
- es - servade hulk
- ac - aktiivsete tippude hulk (tipud, kus ülejääk on suurem kui 0)
- f - voog
- c - läbilaskevõime funktsioon
- l - tipu kõrguse funktsioon
- s - sisendtipp
- t - väljundtipp
- fn – transpordivõrk

5.1 Funktsioonid

Järgnevalt tuuakse välja ja selgitatakse põhilised funktsioonid, mida kasutati push-relabel algoritmi koostamiseks. Osades funktsioonides on sees parameeter tr, mis on loodud silumise (ingl *debugging*) eesmärgiks, algoritmi töös see ei oma tähtsust.

```
Definition excess (fn:FlowNet) (f: @EMap.t R 0) : V -> R :=
    let '((vs, es),c,s,t) := fn in
    fun u =>
    QSumList (map (fun v => f[(v,u)]) vs) -
    QSumList (map (fun v => f[(u,v)]) vs) .
```

Definitsioon *excess fn f x* arvutab transpordivõrgu fn, millel on eelvoog f, tipu x ülejäägi, lahutades väljaminevast voost maha sissetuleva voo. *QSumList (map (fun v => f[(v,u)]) vs)* vastab summale $\sum_{(v,u)\in vs} f(v,u)$ ja *QSumList (map (fun v => f[(u,v)]) vs)* summale $\sum_{(u,v)\in vs} f(u,v)$.

```
Definition has_excess_not_sink fn f v  :=
  let '((vs, es),c,s,t) := fn in
  if T.eqb v t || T.eqb v s then
     false
  else if 0 <? excess fn f v then
```

```
    true
  else
    false
```

Definitsioon *has_excess_not_sink fn f v* kontrollib, et antud tipp v ei oleks väljund ega sisend ja ülejääk oleks suurem kui 0. *T.eqb* tagastab tõeväärtuse true, siis kui argumentideks antud tipud on võrdsed ning *0 <? Excess fn f v* tagastab true väärtuse, siis kui tipu v ülejääk on suurem kui 0.

```
Definition res_cap (fn: FlowNet) (f: @EMap.t R 0) u v : R :=
  let '((vs, es),c,s,t) := fn in
  if (u,v) ∈e es then
    c u v -  f[(u,v)]
  else
    f[(v,u)]
```

Definitsioon *res_cap f u v* arvutab välja serva (u, v) alles oleva läbilaskevõime ja tagastab selle. Siin tähistab *c u v* serva läbilaskevõimet ja *f[(u,v)]* serva voogu. Tingimus (u,v) ∈e es tagastab tõeväärtuse true, siis kui serv (u, v) kuulub servade hulka es. Kui serv (u, v) ei kuulu servade hulka, siis tagastatakse voog, mis läheb tagurpidi ehk serva (v, u) voog.

```
Definition push fn f u v : @EMap.t R 0 :=
    let '((vs, es),c,s,t) := fn in
    let delta := Qmin (excess fn f u) (res_cap fn f u v) in
    if (u,v) ∈e es then
        (EMap.update (u,v) (fun x=>x+delta) f)
    else
        (EMap.update (v,u) (fun x=>x-delta) f)
```

Definitsioon *push fn f u v* valib tipu u ülejäägist ning läbilaskevõimest Qmin abil miinimumi ja saadab selle voona edasi järgmisesse tippu v. Kui (u,v) ∈e es on tõene ehk serv (u, v) kuulub servade hulka es, siis *(EMap.update (u,v) (fun x=>x+delta) f)* suurendab serva (u, v) voogu delta võrra. Vastasel juhul kui (u,v) ∈e es tagastas tõeväärtuse false, siis vähendatakse serva (v, u) voogu delta võrra. Viimasel juhul on tegemist olukorraga kus algselt mööda serva (v, u) saadeti voogu aga selle ülejääk tuleb nüüd maha arvestada.

```
Definition relabel_find fn f (l:@NMap.t nat O) u vs :=
    let fvs := VSet.filter (fun v => 0 <? res_cap fn f u v) vs in
```

```
VSet.fold_left (fun r v =>
match r with
        | None => Some v
        | Some r => if (l[r] <=? l[v])%nat then Some r else Some v
        end) fvs None .
```

Definitsioon *relabel_find fn f l u vs* filtreerib välja tipud, mille vahel on läbilaskevõime ära kasutatud ja jätab alles tipud, mille vahel on läbilaskevõime olemas. Peale seda otsib, kas leiab tipu r, mille kõrgus on väiksem või võrdne tipu v kõrgusega. Kui tipu r kõrgus on väiksem või võrdne tipu v kõrgusega siis tagastatakse tipp r, vastasel juhul tagastatakse tipp v.

```
Definition relabel fn f (l:@NMap.t nat O) u : option (@NMap.t nat O):=
        let '((vs, es),c,s,t) := fn in
        match relabel_find fn f l u vs with
        | None => None
        | Some n => Some (NMap.replace u (1+l[n]) l)
        end.
```

Definitsioon *relabel fn f l u* suurendab tipu u kõrgust 1 võrra. Selleks kasutatakse definitsiooni *relabel_find*, et leida tippude hulgast naabertipp, millel on kõige väikseim kõrgus. Kui leitakse kõige väiksema kõrgusega tipp, siis *(NMap.replace u (1+l[n]) l)* asendab tipu u kõrguse leitud kõrgusest 1 võrra suuremaga. Kui sobivat tippu ei leidu, siis relabel nurjub – see juhtum aga algoritmi töö käigus kunagi ei realiseeru.

```
Fixpoint find_push_node fn f (l:@NMap.t nat O) u vs' :=
        let '((vs, es),c,s,t) := fn in
        match vs' with
        | nil => None
        | v::vs' =>
                if (l[u]=? 1+l[v])%nat &&
                        (0 <? Res_cap fn f u v)
                then
                        Some v
                else
                        find_push_node fn f l u vs'
        end.
```

Rekursiivse funktsiooni *find_push_node* parameetrid on fn f l u vs'. Funktsioon otsib tippude vs' hulgast tippu v, kuhu saaks voogu saata ning mis oleks tipu u kõrgusest 1 võrra kõrgemal ja servade (u, v) vahel oleks veel läbilaskevõimet. Kui mõlemad tingimused on täidetud, siis tagastatakse tipp v. Kui vähemalt üks tingimustest pole täidetud, siis otsitakse edasi ja tipp v sisuliselt eemaldatakse hulgast vs'. Rekursiivne funktsiooni *find_push_node* töötab kuni tagastatakse mingi tipp v või kui hulk vs' saab tühjaks, mille korral tagastatakse väärtus None.

```
Fixpoint initial_push fn f ac es: (@EMap.t Q 0*list V) :=
    let '((_, _),c,s,t) := fn in
    match es with
    | nil => (f,ac)
    | (u,v)::es =>
        if T.eqb s u then
            let f' := EMap.replace (u,v) (c u v) f in
            let ac :=
                if has_excess_not_sink fn f' v then
                    (VSet.add v ac)
                else
                    ac
            in
            initial_push fn f' ac es
        else
            initial_push fn f ac es
    end.
```

Rekursiivse funktsiooni *initial_push* parameetrid on fn, f, ac ja es. Funktsioon täidab push-relabel algoritmi initsialiseerimise ühe sammu, milleks on sisendtipust väljuvatele servadele voo saatmine, kasutades ära serva kogu läbilaskevõime. Funktsiooni töö käigus võetakse hulgast es servasid ja kontrollitakse, kas leitud serv on algtipuga s, kui *T.eqb s u* tagastab tõeväärtuse true, siis saadetakse mööda leitud serva nii palju voogu kui võimalik, ehk serva kogu läbilaskevõime kasutatakse ära. Siis kasutatakse definitsiooni *has_excess_not_sink*, et kontrollida kas tipu v saab lisada hulka ac või ei saa ning *initial_push* kutsutakse uuesti välja uuendatud vooga f' ja uuendatud servade hulgaga es, kus on eemaldatud serv (u, v). *T.eqb s u* tõeväärtuse false korral kutsutakse funktsioon *initial_push* uuesti välja vana vooga f ja uuendatud hulgaga es, kus pole enam serva (u, v). Kui hulgas es pole enam servasid, siis funktsioon lõpetab töö ja tagastab voo f ja aktiivsete tippude hulga ac.

```
Fixpoint gpr_helper_trace fn f l ac g tr : (option (@EMap.t Q 0)*list Tr) :=
    let '((vs, es),c,s,t) := fn in
    match g with
    | O => (None, OutOfGas::tr)
    | S g' =>
        match VSet.choice ac with
        | None => (Some f,tr)
        | Some (u,ac') =>
        match find_push_node fn f l u vs with
        | Some v =>
            let f' := push fn f u v in
            let ac' := if 0 <? (excess fn f' u) then ac else ac' in
            if has_excess_not_sink fn f' v  then
                let ac'' := VSet.add v ac' in
                gpr_helper_trace fn f' l ac'' g' (Push u v f' ac''::tr)
            else
                let ac'' := VSet.remove v ac' in
                gpr_helper_trace fn f' l ac'' g' (Push u v f' ac'::tr)
        | None =>
            match relabel fn f l u with
            | None => (None, RelabelFailed::tr)
            | Some l' =>
                gpr_helper_trace fn f l' ac g' (Relabel u (l'[u]) l'::tr)
            end
        end
        end
    end.
```

Rekursiivse funktsiooni *gpr_helper_trace* parameetrid on fn f l ac g ja tr. Naturaalarvuline parameeter g on „kütus“, ehk piiraja, mitu rekursiivset sammu saab teha enne kui algoritm alla annab ja töö lõpetab. Tegemist on standardse võttega Coqis programmeerides, kus antakse ette parameeter, mis selgelt rekursiivselt väheneb, sest osadel juhtudel Coq ei suuda ise tuvastada rekursiooni jaoks vajalikku parameetrit. Kui kütus saab otsa, siis algoritm tagastab väärtuse None. Algoritmi termineerumise tõestamiseks, mida me selles töös ei ürita, ongi vaja näidata,

et iga transpordivõrgu fn jaoks leidub mingi kütuse kogus, mis garanteerib selle, et algoritm ei tagasta väärtust None.

Funktsioon leiab graafis maksimaalse voo ja tagastab selle, juhul kui graafis pole tippe või servasid, siis tagastab väärtuse None. Peale kütuse kontrollimist kontrollitakse, kas aktiivseid tippe veel leidub. Kui aktiivseid tippe, mis pole sisend ega väljund, ei leidu, siis tagastatakse voog. Kui leidus aktiivseid tippe, siis *match find_push_node fn f l u vs with* otsib, kas leidub selline tipp v, millele saaks push sammu rakendada. Kui leidub tipp v, siis serva (u, v) peal rakendatakse push samm. Peale push sammu on vaja rekursiivseks kutseks paika sättida parameeter ac – selleks tuleb kontrollida, kas tipud u ja v on peale push sammu veel aktiivsed. Esmalt *if 0 <? (excess fn f' u) then ac else ac'* kontrollib, kas tipul u on veel ülejääki, kui on, siis jäetakse alles algne aktiivsete tippude hulk ac, kus tipp u on sees, vastasel juhul jäetakse alles hulk ac', kust tipp u on eemaldatud. Vastavalt sellele, kas *has_excess_not_sink fn f v* tagastab tõeväärtuse true või false, siis rekursiivses kutses lisatakse või eemaldatakse tipp v aktiivsete tippude hulgast ac.

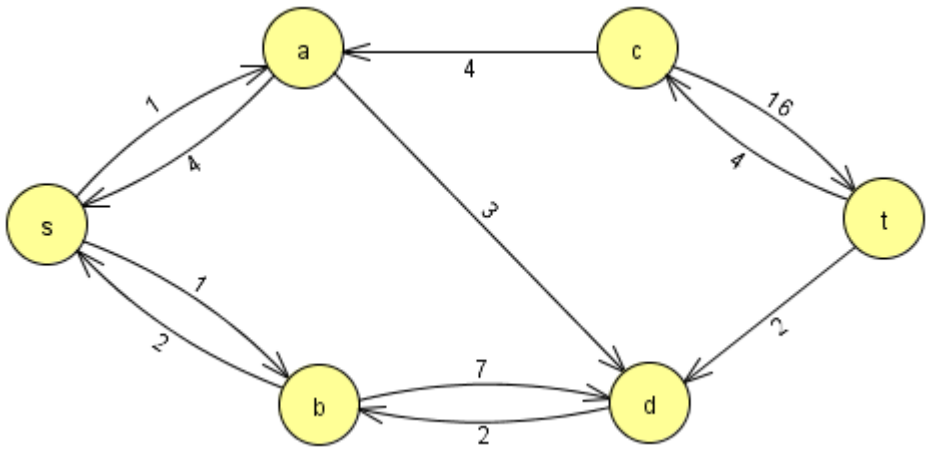
Kui *match find_push_node fn f l u vs with* andis väärtuseks None, siis *match relabel fn f l u with* üritab tipul u teha relabel sammu ja õnnestumise korral kutsub funktsiooni *gpr_helper_trace* välja uue kõrgusega.

Nüüd jääb üle ainult kogu algoritm kokku panna:

```
Definition gpr_trace (fn:FlowNet) : (option (@EMap.t Q 0)*list Tr) :=
    let '((vs, es),c,s,t) := fn in
    let labels := NMap.replace s (length vs) (NMap.empty O) in
    let bound := (length es * length vs * length vs)%nat in
    let '(f, active) := initial_push fn (EMap.empty 0) nil es in
    gpr_helper_trace fn f labels active bound (Init f labels active :: nil).
```

Definitsioon *gpr_trace* algväärtustab graafi, muutes tippude kõrgused nii, et tipp s on kõrgusega *length vs* ja kõik teised tipud kõrgusega 0. Seejärel toestab algse push sammu tipust s väljuvate servade peal. Lõpus kutsutakse välja rekursiivne funktsioon *gpr_helper_trace*, mis leiab maksimaalse voo ja tagastab leitud väärtuse funktsioonile *gpr_trace*.

Push-relabel algoritmi rakendatakse joonisel 21 näidatud transpordivõrgule. Algoritm tagastab väljundina ([(0, 1, 4); (0, 2, 2); (2, 4, 2); (4, 5, 2); (1, 3, 4); (3, 5, 4)], kus (0, 1, 4) tähistab, et tipust 0 tippu 1 saadetakse voog väärtusega 4. Täpset lõppseisu, mis vastab tagastatud järjendile näeb jooniselt 26.



Joonis 26. Graaf push-relabel algoritmi töö lõpus.

Alampeatükis tutvustati ja kirjeldati funktsioone, millega realiseeriti push-relabel algoritm Coqis. Põhifunktsiooniks oli *gpr_trace*, mis tagastas järjendi, kus oli iga serva jaoks välja arvutatud voog, arvutused tegi rekursiivne funktsioon *gpr_helper_trace*.

5.2 Lemma

Siin peatükis tutvustatakse peamise lemma tõestust ja selles kasutatud definitsioone.

```
Definition ActiveNode (fn:FlowNet) (f:@EMap.t Q 0)v :=
    let ’((vs, es),c,s,t) := fn in
    (v ∈v vs) = true  /\ excess fn f v > 0.
```

Definitsioon ActiveNode tagastab tõeväärtuse true, kui ette antud tipp v kuulub tippude hulka ja tipu v ülejääk on suurem kui 0.

```
Definition FlowConservationConstraint (fn:FlowNet) (f:@EMap.t Q 0) :=
    let ’((vs, es),c,s,t) := fn in
    forall v, (v ∈v vs) = true −> v<>s −> v<>t −> excess fn f v == 0.
```

Definitsioon *FlowConservationConstraint* tagastab tõeväärtuse true, kui iga tipu v korral, mis kuulub tippude hulka ja mis pole sisend ega väljund, on tipu ülejääk võrdne väärtusega 0. Kui see funktsioon tagastab tõeväärtuse true, siis transpordivõrgus on täidetud voo säilivus nõue ja eelvoog on võrdne vooga.

Edasi tutvustatakse tõestuse põhilist lemmat, milleks on *FlowConservationGprMain*.

```
1. Lemma FlowConservationGprMain fn (l:@NMap.t nat O):
2.     let '((vs, es),c,s,t) := fn in
3.     (forall u v, ((u, v) ∈e es = true) -> (u ∈v vs) = true /\ (v ∈v vs) = true) ->
```

```
4.   VSet.IsSet vs ->
5.  (forall u v, c u v >= 0) ->
6.  s<>t ->
7.  forall f' l' tr',
8.  gpr_trace fn = (Some (f',l'),tr') ->
9.   (forall n, ActiveNode fn f' n -> n=t \/ n=s) /\
10.   FlowConservationConstraint fn f'/\
11.   length vs = l'[s] /\ O=l'[t].
```

Lemma *FlowConservationGprMain* eeldab, et kui on olemas serv (u, v), siis u ja v on tipud (rida 3) ning servade läbilaskevõime on mittenegatiivne (rida 5) ja sisend ei ole väljund (rida 6) ning funktsioon *gpr_trace* tagastab voo f’ ja kõrgused l’ (rida 8). Siis lemmast järeldub, et aktiivne tipp võib olla vaid sisend või väljund (rida 9) ehk üheski tipus välja arvatud sisendis või väljundis ei ole ülejääk suurem kui 0. Veel on täidetud voo säilivuse tingimus (rida 10) ehk igas tipus välja arvatud sisendis ja väljundis on ülejääk võrdne väärtusega 0. Viimaseks, sisendi kõrgus on võrdne tippude arvuga ja väljundi kõrgus on 0 (rida 11), seega on säilitatud neljandas peatükis tutvustatud kaks invariandi tingimust, milleks on, et sisendi s kõrgus on võrdne tippude arvuga ja väljundi t kõrgus on võrdne väärtusega 0. Seega kuna kõik kolm tingimust on algoritmi lõpus täidetud, siis eelvoog on võrdne vooga, mis omakorda on võrdne maksimaalse vooga. Selle lemmaga ongi tõestatud algoritmi korrektsus, näidates, et algoritmi töö lõpus saadud voog on korrektne ja et invariandid säilitati kogu algoritmi töö vältel.

6. Kokkuvõte

Bakalaureusetöö eesmärgiks oli formaalselt tõestada Coq raamistikus push-relabel algoritmi korrektsus. Töö suurimaks saavutuseks oli Coqis push-relabel algoritmi korrektsuse formaalne tõestamine, kuid hetkel on tõestatud mitte optimeeritud push-relabel algoritmi lahendus.

Töös veel tutvustati Coq raamistikku, erinvaid graafi mõisteid ja liike, kirjeldati push-relabel algoritmi tööd ning tutvustati algoritmi kirjutamisel kasutatud funktsioone ja viimasena kirjeldati põhilemmat, mis tõestas algoritmi korrektsust.

Edasiarendusena saaks praktilises osas optimeerida koodi, et algoritmi efektiivsemaks muuta, kirjutada Ltaci skripte, et vältida kordusi ja tõestusi lühemaks muuta. Veel saaks tõestada ära programmi termineeruvuse, näidates, et iga transpordivõrgu korral leidub selline piiraja, mis garanteerib selle, et algoritm lõpetab töö tagastades maksimaalse voo.

Viidatud kirjandus

[1] Schrijver A. On the history of the transportation and maximum flow problems. Math Program. 91. Springer, 2002, lk 437–445. https://doi.org/10.1007/s101070100259 (15.05.2024)

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest ja Clifford Stein. Introduction To Algorithms. 3. Väljaanne. Massachusetts Institute of Technology, 2009. https://pd.daffodilvarsity.edu.bd/course/material/book-430/pdf_content (15.05.2024)

[3] Goldberg V. Andrew ja Tarjan E. Robert. A new approach to the maximum-flow problem. *Journal of the ACM*, 1988, nr. 4, lk 921-940. https://dl.acm.org/doi/pdf/10.1145/48014.61051 (15.05.2024)

[4] Jon Kleinberg ja Éva Tardos. Algorithm Design. Pearson Education, Inc, 2006. https://edisciplinas.usp.br/pluginfile.php/7933913/course/section/6549987/Algorithm%20Design.pdf (15.05.2023)

[5] The Coq Reference Manual. https://coq.inria.fr/doc/V8.19.0/refman/ (15.05.2024)

[6] Coq, A short introduction to Coq. https://coq.inria.fr/a-short-introduction-to-coq (15.05.2024)

[7] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg ja Brent Yorgey. Logical Foundations. Electronic textbook, 2023. https://softwarefoundations.cis.upenn.edu/lf-current/index.html (15.05.2024)

[8] 3110 Coq Tactics Cheatsheet. https://www.cs.cornell.edu/courses/cs3110/2018sp/a5/coq-tactics-cheatsheet.html (15.05.2024)

[9] Reimo Palm. Diskreetse matemaatika elemendid. Tartu Ülikooli Kirjastus, 2003. https://kodu.ut.ee/~reimo_p/teosed/dme/dme.pdf (15.05.2024)

[10] Yefim Dinitz. Dinitz' Algorithm: The Original Version and Even's Version. Theoretical Computer Science: Essays in Memory of Shimon Even. Springer, 2006, 218-240. https://link.springer.com/chapter/10.1007/11685654_10 (15.05.2024)

[11] Ravindra Ahuja, Thomas Magnanti ja James Orlin. Network Flows: Theory, Algorithms, and Applications. Prentice-Hall, Inc, 1993.

http://www.dl.behinehyab.com/Ebooks/NETWORK/NET005_354338_www.behinehyab.com.pdf (15.05.2023)

Lisad

I. Push-relabel algoritmi tõestus Coqis

Tõestused asuvad tööga kaasas olevas failis *PR.v* ja GitHubis https://github.com/Aksel2/push-relabel

Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, Aksel Õim,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose
   Push-relabel algoritmi formaalne tõestamine Coq raamistikus,
   mille juhendaja on Kalmer Apinis,
   reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.
2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 4.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.
3. Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.
4. Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Aksel Õim

15.05.2024