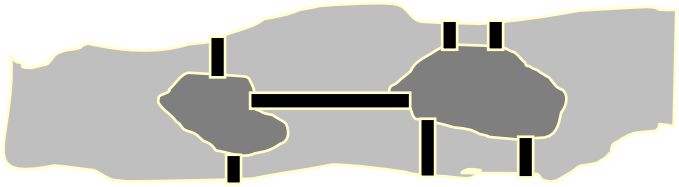
Teorija grafova i linearno programiranje

Osnovni pojmovi teorije grafova

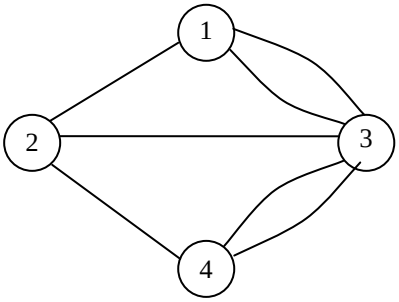
Teorija grafova je jedan od glavnih dijelova *diskretne matematike*. Primjene ove teorije mogu se naći u *informatici*, *operacionim istraživanjima*, *kombinatornoj optimizaciji*, *elektrotehnici*, *fizici*, *hemiji*, *ekonomiji*, *sociologiji* i još mnogdje drugo. Naročito su interesantne primjene u *informatici*, s obzirom da se grafovima mogu modelirati *programske strukture*, *strukture podataka*, *konačni automati*, *komunikacijske mreže*, *formalni jezici*, *planarna elektronska kola*, itd. Značaj teorije grafova u *operacionim istraživanjima* leži u činjenici da postoji veliki broj problema operacionih istraživanja koji se mogu *modelirati jezikom teorije grafova* i rješavati *tehnikama iz teorije grafova*. Isto tako, znatan broj problema teorije grafova mogu se modelirati kao problemi *matematičkog programiranja*, nerijetko i kao problemi *linearnog programiranja*.

Mada su onima kojima su ovi materijali namijenjeni osnovni elementi teorije grafova uglavnom *poznati*, prvo ćemo se, nakon kratkog historijskog uvoda, podsjetiti na *osnovne pojmove teorije grafova*, a zatim ćemo dati vezu između nekih *poznatih problema teorije grafova* i *linearnog programiranja*.

Smatra se da je teorija grafova imala svoj početak *1736. godine* kada je *Euler* dao negativno rješenje ***problema Königsberških mostova***. Naime, ruski grad starog imena Königsberg (današnji Kaljingrad) je smješten na dvije obale rijeke i dva rječna otoka. Pojedini dijelovi grada su povezani mostovima, kao na sljedećoj slici.



Problem Königsberških mostova može se iskazati u vidu pitanja *može li se preći preko svih mostova tako da se preko svakog mosta pređe samo jedanput a da se pri tome vrati na mjesto polaska*. Euler je dao *negativan odgovor* na ovo pitanje, i time riješio *prvi ikada postavljen problem koji se iskazuje jezikom teorije grafova*. Zaista, mapa Königsberga može se modelirati grafom sa sljedeće slike, a problem Königsberških mostova tada se iskazuje kao pitanje *postoji li put koji prolazi kroz sve grane grafa tačno jedanput i vraća se na mjesto polaska*.

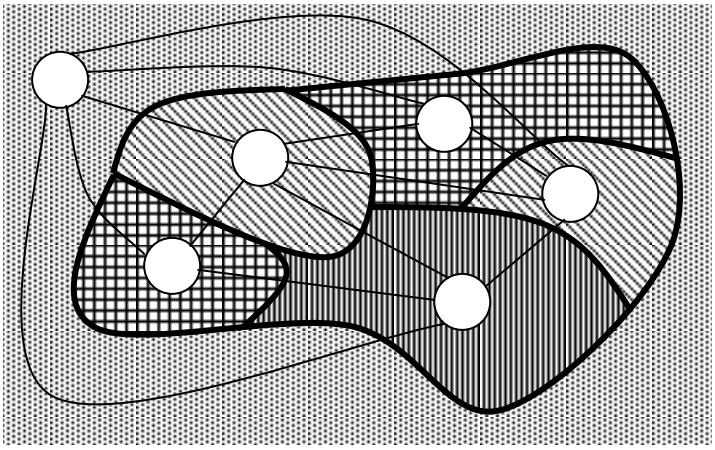


Sljedeći historijski značajan primjer je članak *G. Kirchhoffa* napisan sredinom 19. vijeka, u kojem su izloženi postupci ***analize električnih nreža***, koristeći se *pojmovima teorije grafova*. Teorija grafova omogućava da se osnovni zakoni raspodjele struja kao što su I i II *Kirchofov zakon* primjene na *formiranje sistema jednačina* koje opisuju *raspodjelu struja u električnom kolu*.

Godine 1848. u časopisu "Berliner Schachzeitung" prvi put je publikovan tzv. ***problem osam dama***, koji je bio poznat i ranije. On se može formulirati u vidu pitanja *kako i na koliko načina se može postaviti osam dama na šahovsku tablu tako da se međusobno ne napadaju*. Ovaj problem se može modelirati jezikom teorije grafova ukoliko se svakoj figuri pridruži jedan graf na sljedeći način. *Polja šahovske* table će predstavljati *čvorove grafa*. Iz čvora X ide grana ka čvoru Y ako i samo ako sa polja X figura može preći na polje Y. Na taj način je i šah *u određenoj relaciji sa teorijom grafova*.

Godine 1879. A. Cayley je članovima Londonskog društva postavio čuveni ***problem četiri boje***: *Dokazati da se svaka geografska karta može obojiti sa najviše četiri boje tako da svaka država bude obojena jednom od boja i da nikoje dvije susjedne država ne budu obojene istom bojom.* Ovdje se smatra da su *susjedne države* one koje imaju *zajedničku granicu*, ali ne i one koje imaju jednu ili više zajedničkih izoliranih tačaka. Također, smatra se da svaka država ima *teritorijalni kontinuitet,* odnosno da se cijela država sastoji iz *jednog povezanog dijela*.

Problem četiri boje se lako može prevesti na jezik teorije grafova. Naime, svakoj državi pridružimo po jedan čvor, pri čemu čvorove koji odgovaraju državama spajamo granom ako i samo ako su odgovarajuće države susjedne. Na taj način se dobija jedan graf koji je *planaran*, jer se može nacrtati *u ravni* tako da mu se *grane ne presjecaju*. Sada se problem četiri boje može formulirati na sljedeći način: *Dokazati da se čvorovi svakog planarnog grafa mogu obojiti sa najviše četiri boje tako da niti jedna grana ne spaja čvorove sa istom bojom*.



Ovaj problem riješili su *K. Appel i W. Haken* 1976. godine (dakle, skoro 100 godina nakon što je problem postavljen). Dokaz je izuzetno dug i složen, i izveden je uz intenzivnu pomoć računara (preko 1200 sati procesorskog vremena utrošeno je na razne proračune neophodne za izvođenje dokaza).

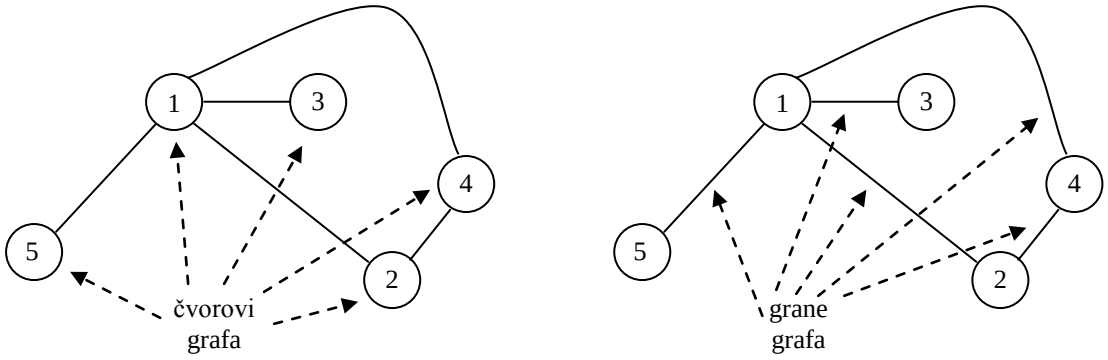
Navedena četiri problema predstavljaju historijski najpoznatije primjere, koji su doveli do nastanka nove matematičke discipline, poznate pod imenom ***teorija grafova***. Neformalno rečeno, grafovi su matematički objekti koji se sastoje od dvije vrste jedinki, koje se nazivaju *čvorovi* i *grane*. Pri tome, grane ne mogu egzistirati *samostalno*, nego one obavezno *povezuju čvorove* i, na izvjestan način, modeliraju *izvjesne veze* između čvorova.

Postoji više različitih *međusobno neekvivalentnih formalnih definicija grafova*, koje *nemaju istu izražajnu moć*, ali su *izražajnije definicije obično i komplikovanije*. Za naše potrebe, biće dovoljna *najprostija definicija* prema kojoj se ***graf*** definira kao *uređeni par* $G = (X, R)$, gdje je X neki *neprazan skup*, a R *binarna relacija* u skupu X. Elementi skupa X nazivaju se ***čvorovi grafa***, dok se elementi relacije R nazivaju ***grane grafa***. Sam skup X naziva se ***skup čvorova grafa***, dok se relacija R (koja je *skup nekih uređenih parova elemenata skupa* X) naziva i ***skup grana grafa***.

Pod ***geometrijskom predstavom*** (***reprezentacijom***) ili ***crtežom grafa*** naziva se *crtež* sastavljen od *figura* koje *modeliraju čvorove* (obično krugovi) i *orjentiranih linija* (*strelica*) koje povezuju te figure, a koje *modeliraju grane grafa*. Često se ova geometrijska reprezentacija prosto naziva grafom.

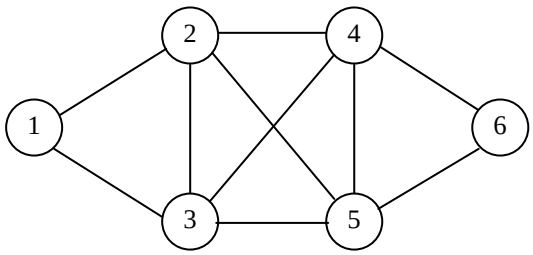
Ako *paru čvorova* x_i i x_j odgovaraju *dvije grane* (x_i, x_j) i (x_j, x_i), na crtežu grafa se ne moraju povlačiti dvije linije između čvorova x_i i x_j nego se može crtati *jedna dvostrano orjentirana linija* ili linija *bez orjentacije*. Par grana oblika (x_i, x_j) i (x_j, x_i) nazivamo i ***neusmjerena grana***.

Graf $G = (X, R)$ u kojem je R *simetrična relacija* naziva se ***simetričan*** ili ***neorjentiran graf***. Sljedeća slika prikazuje čvorove i grane jednog neorjentiranog grafa.



Graf G = (X, R) kod kojeg je R *antisimetrična relacija* naziva se ***antisimetričan*** ili ***orjentiran***. Graf koji nije ni simetričan ni antisimetričan naziva se ***hibridni graf***.

Ako se pri grafičkom predstavljanju grafa ne koristi konvencija da se *parovi grana suprotne orijentacije* zamjenjuju *neorjentiranim granama* (nego se u takvom slučaju uvijek crta par grana suprotne orjentacije), graf se tada naziva ***digraf***. Iz ovoga slijedi da na formalnom nivou nema razlike između običnog grafa i digrafa, nego je razlika u *načinu crtanja*. Ipak, postoje definicije koje omogućavaju i da se na *formalnom matematskom nivou* napravi *striktna razlika* između običnog grafa i digrafa, ali su takve definicije *složenije* i *manje praktične za rad*. Te definicije dozvoljavaju da skup grana R sadrži kako *uređene*, tako i *neuređene* parove elemenata iz X (tj. obične *dvočlane podskupove* od X). Mana takvih definicija je što se tada R više ne može posmatrati kao *relacija*, tako da se ne može koristiti ni matematski aparat razvijen za relacije. Bez obzira na to, često nije potrebno posebno voditi računa da je R relacija u skupu X, nego se može smatrati da je graf zadan kao G = (X, R) gdje je X *skup čvorova* a R *skup grana grafa*. Na primjer, neka je dat graf čija je geometrijska reprezentacija kao na sljedećoj slici:



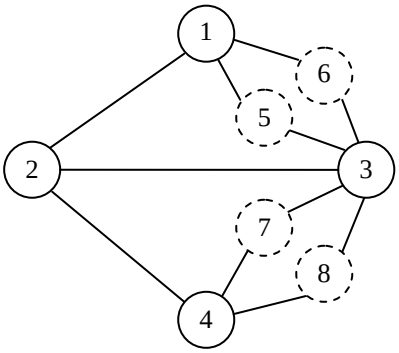
Ovaj graf opisujemo kao G = (X, R), gdje su *skup čvorova* X i *skup grana* R dati kao

$$X = \{1, 2, 3, 4, 5, 6\}$$

$$R = \{(1,2), (1,3), (2,1), (2,3), (2,4), (2,5), (3,1), (3,2), (3,4), (3,5), (4,2), (4,3), (4,5), (4,6), (5,2), (5,3), (5,4), (5,6), (6,4), (6,5)\}$$

Grafovi mogu biti ***konačni*** ili ***beskonačni*** u zavisnosti od toga da li je skup X *konačan* ili *beskonačan*. Mi ćemo se ograničiti samo na proučavanje konačnih grafova.

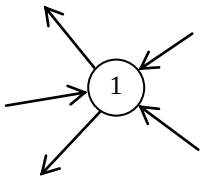
Jedan od nedostataka gore predložene definicije grafa je što ona *ne dozvoljava da dva čvora budu međusobno povezana sa više grana*, s obzirom da definicija skupa *ignorira ponavljanja elemenata*. Na primjer, model kojim se opisuje problem Königsberških mostova *ne može se adekvatno modelirati grafom* u smislu ovdje usvojene definicije jer se ne može adekvatno modelirati činjenica da su čvorovi 1 *i* 3, odnosno 3 i 4 *povezani sa po dvije grane*. Mada postoje *komplikovanije* definicije grafa koje dozvoljavaju i *višestruke grane* (to dovodi do tzv. ***multigrafova***), problem je moguće riješiti čak i uz pomoć ovdje usvojene definicije grafa uvođenjem *fiktivnih čvorova* kojom se višestruke grane između čvorova *cijepaju* na po dvije grane. Na primjer, problem Königsberških mostova se može modelirati na sljedeći način, koji dopušta prikaz pomoću ovdje usvojene definicije grafa:



Ovdje smo uveli *fiktivne čvorove* 5, 6, 7 i 8 da bismo "raskinuli" višestruke grane. Zapravo, bili su dovoljni samo fiktivni čvorovi 5 i 7, jer se jedna od višestrukih grana ne mora "raskidati", ali na ovaj način se dobija *povećana simetrija* u modelu, što svakako nije na odmet.

Grana koja spaja čvor grafa sa *samim sobom* naziva se ***petlja***. Za *dva čvora* neorjentiranog grafa bez petlji kaže se da su ***susjedni*** ako su *spojeni granom*. Dva susjedna čvora su ***krajnje tačke*** grane *koja ih spaja*. Ako je neki čvor *krajnja tačka* neke grane kaže se da se ta grana ***stiče*** u tom čvoru. U ovom slučaju se također kaže da su čvor i grana ***susjedni*** ili ***incidentni***.

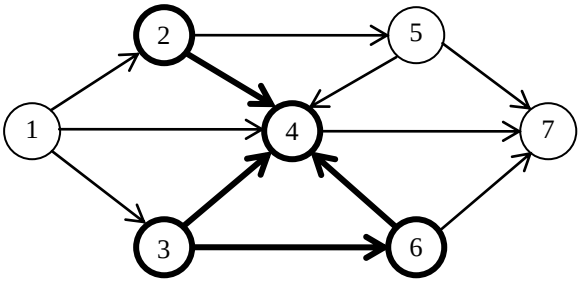
Stepen čvora je određen *brojem njemu susjednih čvorova*, odnosno, *brojem grana koje se stiču u tom čvoru*. ***Ulazni stepen čvora*** je određen *brojem grana* koje *ulaze* u taj čvor, dok je ***izlazni stepen čvora*** određen *brojem grana* koje *izlaze* iz tog čvora. Na primjer, čvor na sljedećoj slici ima ulazni stepen 3 i izlazni stepen 2, dok mu je stepen 5.



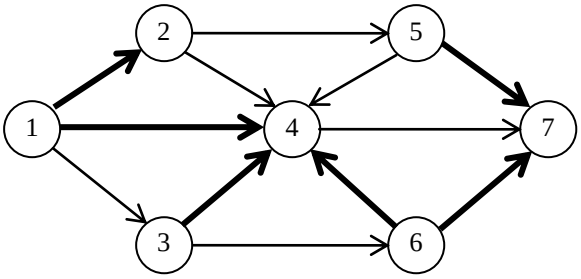
Kod neusmjernih grafova, kada govorimo o stepenu čvora; po konvenciji se neusmjerene grane *ne broje dvostruko*, odnosno *ne brojimo dvostruko činjenicu da je svaka neusmjerena grana zapravo sastavljena od dvije međusobno suprotno orijentirane grane*.

Dvije grane su ***susjedne*** ako imaju *zajednički čvor*. Ako je neka grana, koja *spaja* čvorove x_i i x_j, *orjentirana od* x_i *ka* x_j, kaže se da grana ***izlazi*** iz čvora x_i a ***ulazi*** u čvor x_j.

Za graf $H = (Y, T)$ se kaže da je ***podgraf*** grafa $G = (X, R)$ ***induciran*** (***obrazovan***) ***skupom čvorova*** **Y** ako je $Y \subset X$ i $T = R \cap (Y \times Y)$. Drugim riječima, T je podskup skupa grana R koji sadrži sve one parove iz R koji su obrazovani *samo od elemenata skupa* Y. Na taj način, H je graf koji je nastao od grafa G zadržavanjem *samo onih čvorova koji se nalaze u skupu* Y i *samo onih grana kojima su oba kraja u skupu* Y. Na primjer, sljedeća slika prikazuje jedan graf sa skupom čvorova $X = \{1, 2, 3, 4, 5, 6, 7, 8\}$ kao i jedan njegov podgraf induciran skupom $Y = \{2, 3, 4, 6\}$. Čvorovi i grane podgrafa $H = (Y, T)$ prikazane su *podebljano*:



Za graf $H = (X, T)$ se kaže da je ***djelimični*** ili ***parcijalni graf*** grafa $G = (X, R)$ ako je $T \subset R$. Dakle, graf H ima *iste čvorove* kao graf G, ali mu je skup grana *sužen* u odnosu na skup grana grafa G. Na sljedećoj slici je prikazan jedan graf i jedan njegov djelimični graf, pri čemu su grane njegovog djelimičnog grafa prikazane podebljano:

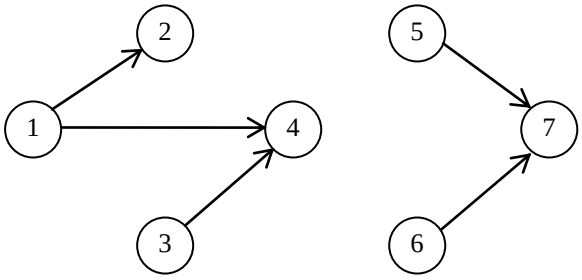


Djelimični odnosno *parcijalni* graf *nekog podgrafa datog grafa* naziva se ***djelimični podgraf*** datog grafa.

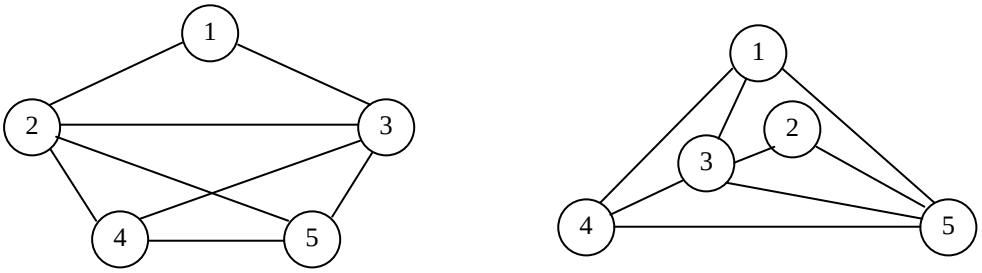
Put *dužine k* u digrafu je *svaki niz grana* $g_1, g_2, ..., g_k$ koji zadovoljava da svaka od grana u_i, $i = 2..k$ *počinje* u onom čvoru u kojem se *završava* grana u_{i-1}. Pri tome, grana u_1 može polaziti od proizvoljnog čvora digrafa. Kaže se da put ***povezuje*** čvor x_i sa čvorom x_j ako je x_i *početni čvor prve grane u putu*, a x_j *završni čvor posljednje grane u putu*.

Put može prolaziti *više puta* kroz *istu granu* ili kroz *isti čvor*. Put koji ne prolazi više puta kroz isti čvor naziva se ***elementarni put***, dok se put koji ne prozali više puta kroz istu granu naziva ***staza***. Put koji *završava u istom čvoru u kojem počinje* naziva se ***kružni put***, ***zatvoreni put*** ili ***ciklus***. Zatvorena staza naziva se ***kontura***. Niz grana $g_1, g_2, ..., g_k$ kod kojeg se kod svake od grana g_i, $i = 2..k$ *jedan od njenih krajeva* poklapa sa *jednim od krajeva grane* g_{i-1} naziva se ***lanac***. Svaki put je lanac, ali *svaki lanac nije nužno put*. Međutim, svaki lanac može postati put ukoliko *obrnemo orjentaciju nekih od grana koje čine lanac*.

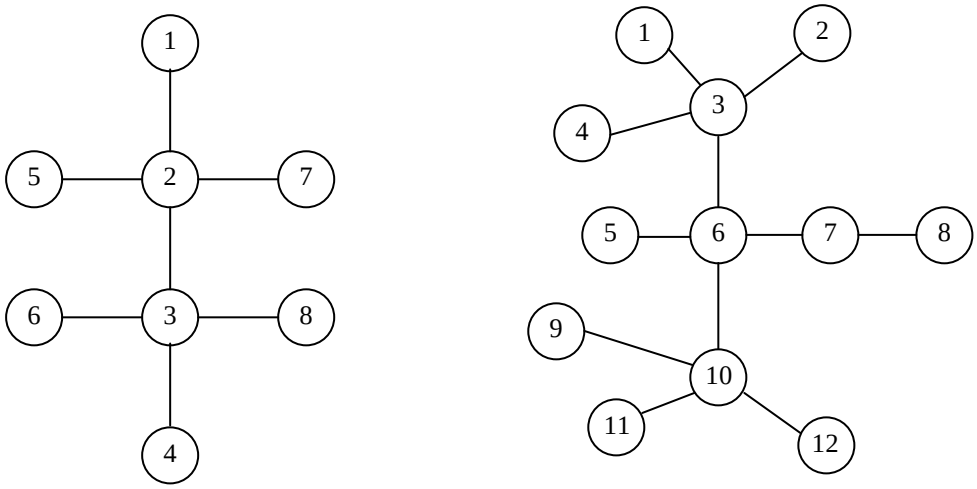
Neorjentirani graf je ***povezan*** ako se *ma koja njegova dva čvora* mogu ***povezati putem***. *Ako postoje* čvorovi koji se *ne mogu povezati putem*, graf je ***nepovezan***. Svaki nepovezani graf se satoji od *dva ili više povezanih dijelova*, ali koji *između sebe nisu povezani*. Ovi odvojeni dijelovi se nazivaju ***komponente povezanosti grafa*** (ili ponekad samo ***komponente grafa***). Na sljedećoj slici je prikazan jedan nepovezani graf sa *dvije komponetne povezanosti:*



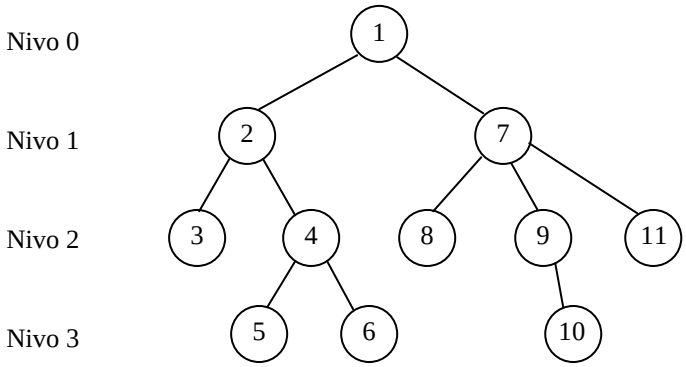
Dva grafa su ***izomorfna*** ukoliko postoji *uzajamno jdnoznačno preslikavanje* skupa njihovih *čvorova* (iz jednog na drugi) koje *održava osobinu susjednosti čvorova*. Drugim riječima, dva grafa su izomorfna ukoliko se jedan od njih može *prenumeracijom čvorova svesti na drugi*. Na primjer, dva grafa sa sljedeće slike su izomorfni, jer se graf sa lijeve strane nakon prenumeracije čvorova $1 \to 2$, $2 \to 3$, $3 \to 5$, $4 \to 4$ i $5 \to 1$ svodi se na graf sa desne strane, samo drugačije nacrtan (tačnije, nakon ove prenumeracije, graf sa lijeve i desne strane imaće isti skup čvorova i isti skup grana, tako da će postati praktično identični).



Povezan neusmjeren graf sa n čvorova i $m = n - 1$ grana naziva se ***stablo*** ili ***drvo***. Postoje i druge *međusobno ekvivalentne* definicije stabla. Recimo, prema jednoj od njih, stablo je *povezan neusmjeren graf bez ciklusa*. Svako stablo sa barem dva čvora ima *barem dva vrha stupnja* 1. Vrhovi stupnja 1 zovu se ***listovi stabla***. Sljedeća slika ilustrira primjere dva stabla.



Stablo u kome je *jedan čvor posebno označen* naziva se ***korijensko stablo***. Označeni čvor je ***korijen stabla.*** Svaki čvor *x* korjenskog stabla povezan je *jedinstvenim elementarnim putem* sa korjenom *k* stabla. *Broj grana* u ovom putu određuje *nivo čvora x*. Korijen stabla *k* ima *nivo* 0. Korijenska stabla se obično crtaju tako da se svi čvorovi nalaze *na istoj visini*, dok se čvorovi sa manjim nivoom crtaju *iznad* čvorova sa višim nivoom, kao na sljedećoj slici:



Planarni grafovi su grafovi koji se mogu *nacrtati u ravni* tako da im se *grane ne sijeku*. Planarni graf predstavljen u ravni *dijeli ravan na više konačnih zatvorenih oblasti* i *jednu beskonačnu oblast*.

Eulerov put je put koji tačno *jedanput* prolazi *kroz svaku granu grafa* (ili *multigrafa*, s obzirom da je ova definicija primjenljiva i na multigrafove). Eulerov put *može biti i zatvoren* i tada se naziva ***Eulerov ciklus*** ili ***Eulerova kontura***.

Testiranje da li u grafu postoji Eulerov put je vrlo jednostavno. Naime, ***Eulerov teorem*** tvrdi da povezan neusmjereni (multi)graf G sa barem jednom granom *posjeduje Eulerov put ako i samo ako sadrži* nijedan ili tačno dva čvora *neparnog stepena*. Ovaj teorem predstavlja *prvi teorem teorije grafova*. Njega je formulirao Euler, koji je samo skicirao osnovnu ideju dokaza. Potpuni dokaz ovog teorema dao je *Hierholzer*, tako da se on naziva i ***Euler-Hierholzerov teorem***.

Prikazaćemo sada *skicu dokaza Euler-Hierholzerovog teorema*, prvo zbog činjenice da on ilustrira neke elementarne ideje koje se koriste pri izvođenju rezultata iz teorije grafova, a drugo zbog činjenice da dokaz daje uvid u to *kako naći Eulerov put* ukoliko on postoji. Za *strogi dokaz*, potrebno je dokazati i neka pomoćna tvrđenja koja se iznose u skici dokaza, koja ćemo ovdje prihvatiti kao *intuitivno očigledna*.

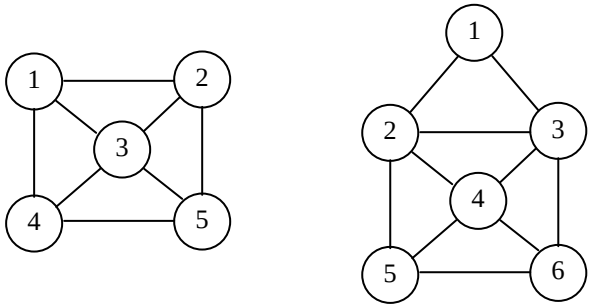
Pokažimo prvo da je uvjet *neophodan*. Pretpostavimo stoga da Eulerov put postoji. Krećući se po Eulerovom putu, kad god se dođe u neki čvor, mora se koristiti *druga grana* za napuštanje čvora. To znači da *svi čvorovi moraju biti parnog stepena*, osim možda prvog i posljednjeg. Ukoliko je Eulerov put *zatvoren*, to onda mora važiti i za *prvi čvor*, koji je tada ujedno i posljenji (jer se u prvi čvor moramo vrati *nekom drugom granom* u odnosu na onu kojom smo ga prvi put napustili). Slijedi da tada graf *nema niti jedan čvor neparnog stepena*. S druge strane, ukoliko imamo *nezatvoreni Eulerov put*, tada lako uviđamo da *prvi* i *posljednji čvor* moraju imati neparan stepen, odnosno tada postoje *tačno dva čvora neparnog stepena*. Ovim smo pokazali da je uvjet neophodan.

Sada ćemo pokazati da je uvjet *dovoljan*. Za tu svrhu, koristićemo *princip matematičke indukcije*. Za (multi)graf sa *dvije grane* teorem je tačan. Sada ćemo postaviti *induktivnu hipotezu* da je tvrđenje tačno za *sve* (*multi*)*grafove sa manje od m grana*, odnosno da *svi* (*multi*)*grafovi sa manje od m grana koji zadovoljavaju uvjete teorema posjeduju Eulerov put*, gdje je *m* neki *proizvoljan*, ali *fiksiran broj veći od* 2. Pokazaćemo da iz te pretpostavke slijedi da će tvrđenje biti tačno i *za proizvoljan* (*multi*)*graf sa m grana*. Tada će, zbog proizvoljnosti (mutli)grafa i proizvoljnosti broja *m*, tvrđenje vrijediti *za sve* (*multi*)*grafove*.

Posmatrajmo sada proizvoljan (multi)graf G sa *m* grana. Razmotrimo prvo slučaj kada *nema čvorova neparnog stepena*. Kretanje po granama započinjemo iz *proizvoljnog* čvora. Ako se svaki čvor napušta duž grane kojom se *do tada nismo kretali*, nije teško uvidjeti da ćemo se kad-tad *vratiti u početni čvor*. Put koji smo pronašli nazovimo P. Ukoliko je ovakav put prošao kroz *sve grane grafa*, on je očigledno *Eulerov put*. Sada ćemo pokazati da, ukoliko to *nije slučaj*, on *mora biti dio Eulerovog puta*. Iz grafa ćemo *ukloniti* ovaj zatvoreni put koji smo dotada našli. Preostali dio grafa *ne mora biti povezan*, ali svi njegovi čvorovi imaju paran stepen. Prema induktivnoj hipotezi, *svaka komponenta povezanosti* djelimičnog grafa koji smo dobili nakon uklanjanja nađenog zatvorenog puta *posjeduje Eulerov zatvoreni put*. Nazovimo te puteve P_1, P_2, ..., P_k gdje je *k* broj komponenti povezanosti ovog djelimičnog grafa. Pošto je G povezan, svaki od ovih Eulerovih puteva ima *barem jedan zajednički čvor* sa kružnim putem P. Traženi *Eulerov kružni put* sada možemo konstruisati tako što započnemo kretanje po putu P i kad god naiđemo na neki čvor u kojem se put P dodiruje sa nekim od puteva P_1, P_2, itd. *prvo obiđemo cijeli taj put* prije nego što *nastavimo kretanje po putu* P. Na taj način ćemo dobiti traženi Eulerov put. Puteve P_1, P_2, ..., P_k dobijamo *rekurzivnom primjenom istog postupka* na svaku od *k* komponenti povezanosti grafa koji se dobija uklanjanjem puta P iz grafa G.

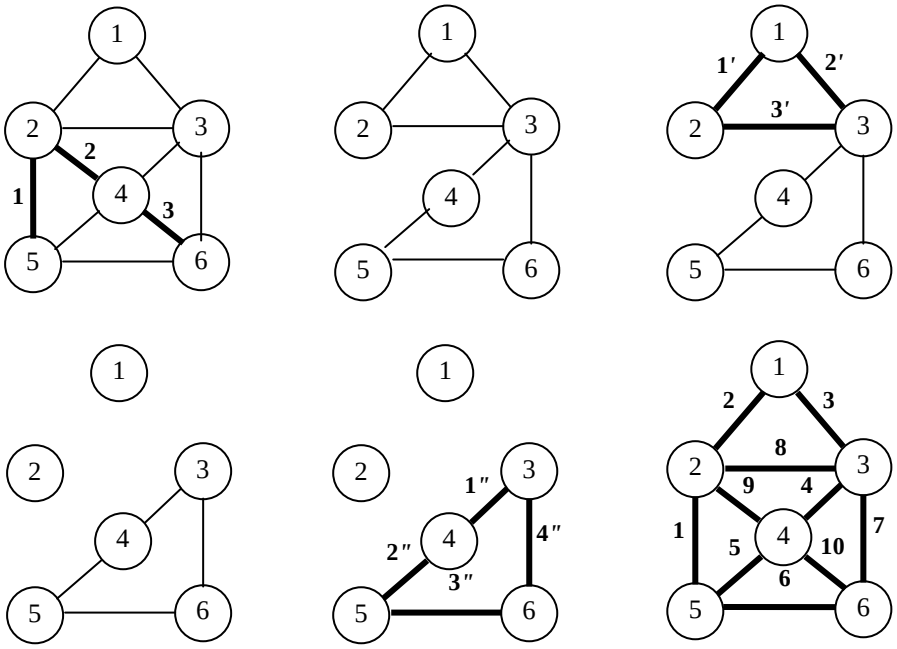
Ukoliko graf ima *tačno dva čvora sa neparnim stepenom*, obavezno treba poći *iz jednog od njih*. Dalje se krećemo po granama, koristeći samo grane kojima se *do tada nismo kretali*. Nije teško uvidjeti da ćemo na taj način uvijek *stići do drugog neparnog čvora*. Moguće je da na ovaj način već dobijemo traženi Eulerov put. Ukoliko to nije slučaj, tada se udaljavanjem grana iz tog puta dobije djelomični graf koji ima *sve čvorove parnog stepena*. Dalje postupak teče kao u predhodnom slučaju. U svakom slučaju, na konstruktivan način smo pokazali kako uz pretpostavku da su uvjeti teoreme zadovoljeni možemo formirati traženi Eulerov put. Ovim je skica dokaza kompletirana.

- ***Primjer***: Na osnovu Euler-Hierholzerove teoreme, utvrditi koji od grafova sa naredne slike posjeduje Eulerov put i naći takav put tamo gdje on postoji.



Graf sa lijeve strane ima četiri čvora sa neparnim stepenom (čvorovi 1, 2, 4 i 5) pa prema tome *ne zadovoljava* uvjete Euler-Hierholzerove teoreme (odnosno on *ne posjeduje Eulerov put*). S druge strane, graf sa desne strane ima *tačno dva čvora neparnog stepena* (čvorovi 5 i 6), tako da on *zadovoljava* uvjete Euler-Hierholzerove teoreme, tako da on *posjeduje Eulerov put*.

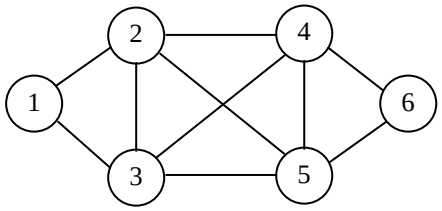
S obzirom da je u pitanju posve mali graf, za njega je lako pronaći Eulerov put prostim isprobavanjem. Međutim, ovdje ćemo demonstrirati kako se Eulerov put može naći postupkom izvedenim u skici dokaza Euler-Hierholzerove teoreme. Krenućemo na primjer od čvora 5 koji ima neparan stepen i krenuti nasumice recimo granom $(5, 2)$. Pretpostavimo da smo dalje nasumice odabrali da nastavimo kretanje duž grana $(2, 4)$ i $(4, 6)$. Na taj način, došli smo do drugog čvora neparnog stepena (čvora 6), ali *nismo formirali Eulerov put*. Uklanjanjem grana koje čine ovaj put dobijamo novi graf čiji su svi čvorovi parnog stepena, tako da u njemu *mora postojati Eulerov ciklus*. Krenimo od nekog od čvorova koji je u ovom novom grafu, a pripada do tada nađenom putu, recimo od čvora 2. S obzirom da se radi o malom grafu, vrlo je lako naći Eulerov ciklus u njemu. Međutim, pretpostavimo da ponovo nismo imali sreće, nego da smo se nakon tri koraka $(2, 1)$, $(1, 3)$ i $(3, 2)$ vratili u početni čvor. Tada ćemo ponovo iz grafa eliminirati nađeni zatvoreni put i nastaviti od nekog čvora u reduciranom grafu koji leži na tom putu. U narednom koraku nalazimo put $(3, 4)$, $(4, 5)$, $(5, 6)$ i $(6, 3)$. Dosadašnji tok postupka prikazan je na sljedećoj slici:



Na ovoj slici su sa 1, 2 i 3 obilježene grane koje su dobijene u prvoj etapi postupka, sa $1'$, $2'$ i $3'$ grane koje su dobijene nakon prvog odstranjivanja, a sa $1''$, $2''$ i $3''$ graje koje su dobijene nakon drugog odstranjivanja. Kada sve ove djelimične putanje sastavimo u jednu cjelinu, dobijamo da traženi Eulerov put glasi $1 - 1' - 2' - 1'' - 2'' - 3'' - 4'' - 3' - 2 - 3$. Nađeni Eulerov put je prikazan na kraju prethodne slike, nakon izvršene prenumeracije grana rednim brojevima od 1 do 10 u skladu sa njihovim redoslijedom duž nađenog puta.

Značaj Eulerovog puta u operacionim istraživanjima leži u tome što Eulerov put, u slučajevima kada on postoji, *daje optimalno rješenje* jednog važnog optimizacionog problema koji je poznat kao ***kineski problem poštara***. Neformalno, ovaj problem se može iskazati kao određivanje puta kojim se treba kretati poštar kroz neki grad tako da pri tome posjeti *svaku ulicu grada barem jedanput* i da se *vrati na mjesto odakle je pošao*, a da pri tome *pređe što je god moguće manji ukupni put*. Ovaj problem se, uz prikladno kodiranje, može modelirati kao *problem cjelobrojnog linearnog programiranja*. Međutim, takvi problemi su u općem slučaju *vrlo teški za rješavanje*. S druge strane, ovaj problem se lako iskazuje i jezikom teorije grafova. Zaista, neka graf modelira ulice grada pri čemu su čvorovi grafa raskrsnice i neka je svakoj grani *pridružen neki broj* (*težina*) koji predstavlja *dužinu odgovarajuće ulice*. Tada se problem može iskazati kao *nalaženje puta koji prolazi barem jednom kroz svaku granu grafa i vraća se u početni čvor, a za koji je suma težina koje su pridružene granama puta minimalna*. Jasno je da ukoliko je moguće da poštar kroz svaku ulicu prođe *tačno jednom*, tada je takav put ujedno i *optimalan*. Dakle, u ovom problemu, ako u odgovarajućem grafu postoji Eulerov ciklus, tada *taj ciklus daje optimalno rješenje problema*. S druge strane, ukoliko Eulerov ciklus ne postoji, problem se mora rješavati *na neki drugi način*. Za slučaj neusmjerenih ili čisto usmjerenih grafova *postoji efikasan algoritam za rješavanje ovog problema* (tzv. ***Edmonds-Johnsonov algoritam***), dok za slučaj *hibridnih grafova* efikasni algoritmi za rješavanje ovog prolema *nisu do danas poznati*.

Hamiltonov put je put koji tačno *jedanput* prolazi *kroz svaki čvor grafa* (ili *multigrafa*). Hamiltonov put također *može biti i zatvoren* i tada se naziva ***Hamiltonov ciklus*** ili ***Hamiltonova kontura***. Na primjer, u grafu sa sljedeće slike, put koji redom prolazi kroz čvorove 1, 2, 4, 6, 5, 3 i 1 predstavlja Hamiltonov put.



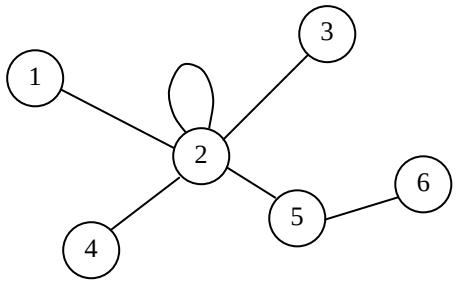
Za razliku od problema egzistencije Eulerovog puta, do danas *nisu poznati efikasno provjerljivi potrebni i dovoljni uvjeti* koje neki graf treba zadovoljavati da bi u njemu postojao Hamiltonov put, niti *efikasni postupci za njegovo nalaženje* ukoliko on postoji (istina je da se problem nalaženja Hamiltonovog ciklusa može modelirati kao problem cjelobrojnog linearnog programiranja, ali kao što je već rečeno, takvi problemi mogu biti vrlo teški za rješavanje). Ipak, poznati su brojni *dovoljni uvjeti* čije ispunjenje *garantira egzistenciju Hamiltonovog puta*, ali čije neispunjenje *nije garancija da Hamiltonov put ne postoji*. Većina ovih dovoljnih uvjeta tvrdi da graf koji u izvjesnom smislu ima *dovoljno mnogo grana* mora sadržavati Hamiltonov put. Jedan od najjednostavnijih uvjeta tog tipa dat je kroz ***Rédeiov teorem*** prema kojem u usmjerenom grafu (digrafu) u kojem između proizvoljna dva čvora x_i i x_j postoji *barem jedna od grana* (x_i, x_j) ili (x_j, x_i) *mora postojati Hamiltonov put*.

Prirodna generalizacija problema Hamiltonovog puta je ***problem trgovačkog putnika***. Neformalno iskazano, ovaj problem zahtijeva da se pronađe put kojim se treba kretati trgovački putnik tako da obiđe svaki grad tačno jedanput i da se vrati u polazni grad, a da pri tome pređe put najmanje dužine. Ovaj problem se također može izraziti u obliku (veoma teškog) problema *cjelobrojnog linearnog programiranja*. Iskazano jezikom teorije grafova, ovaj problem se može iskazati da se u grafu u kojem postoji više Hamiltonovih ciklusa, pronađe *onaj za koji je suma težina koje su pridružene granama minimalna*. Ovaj problem je toliko težak da za sada sva raspoloživa računarska snaga na svijetu iskorištena zajedno nije dovoljna da se u razumnom vremenu (kraćem od recimo jednog ljudskog vijeka) riješi problem trgovačkog putnika za proizvoljan slučaj sa 300 gradova (čvorova). Ipak, zbog svoje velike praktične važnosti, problem trgovačkog putnika je vjerovatno jedan od najviše proučavanih problema teorije grafova i operacionih istraživanja općenito. Kao rezultat velikih uloženih napora, do danas su razvijeni brojni algoritmi za rješavanje ovog problema koji mogu biti efikasni za izvjesne specijalne klase grafova kao i brojni algoritmi koji za koje se ne garantira da pronalaze baš "najbolje" rješenje, ali nalaze rješenje koje je u većini slučajeva "dovoljno dobro" za praktične primjene.

Reprezentacije grafova

Pod ***reprezentacijom grafa*** podrazumijevamo *predstavljanje* ili *zapisivanje* grafa u obliku koji je pogodan za *implementaciju na računaru*. Postoji *više različith načina* za reprezentaciju grafova. Koji od načina je *pogodniji*, često zavisi od *klase grafova* i/ili *tipova problema* koje treba riješiti. Dvije glavne klase grafova su ***rijetki*** i ***gusti*** grafovi. Tačne granice ovih klasa nisu precizno određene, a postoje i grafovi koji se ne smatraju niti rijektim niti gustim. Ukoliko graf ima n čvorova, maksimalan broj grana koji graf može imati je n^2 (slučaj kad je svaka grana spojena sa svakom granom), dok je minimalan broj grana koji graf može imati a da bude povezan reda n (tačnije, $n-1$). Rijetki su oni grafovi kod kojih je broj grana istog reda veličine kao n, dok su gusti oni grafovi kod kojih je broj grana reda veličine n^2. Precizne granice za "rjetkoću" i "gustoću" nisu date. Obično se uzima da su rijetki oni grafovi kod kojih broj grana ne prelazi iznos $n \log_2 n$, dok se uzima da je kod gustih grafova broj grana barem $n^2 / \log_2 n$.

Matrica susjedstva (***adjacencije***) je najčešći način reprezentacije grafa. Ovaj način reprezentacije podrazumijeva da su *čvorovi grafa numerirani rednim brojevima od* 1 *do n*. Prema ovom načinu, graf se predstavlja *matricom* **A** kod koje je član $a_{i,j}$ jednak 1 ako i samo ako *postoji grana* koja vezuje čvor *i* sa čvorom *j*, u suprotnom je jednak 0. Na primjer, neka je dat graf na sljedećoj slici:



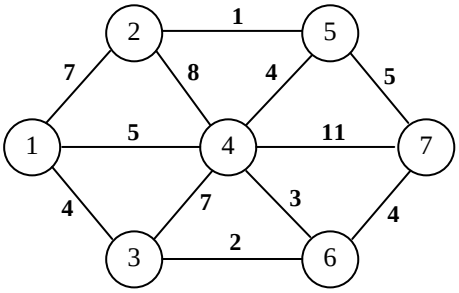
Matrica susjedstva za ovaj graf data je kao

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Matrica susjedstva je u mnogim primjenama *vrlo praktičan način za reprezentaciju grafa*. Na primjer, algoritamski korak testiranja da li su čvorovi *i* i *j* susjedni izvodi se *vrlo efikasno*, jer se svodi na prosto testiranje da li je element $a_{i,j}$ matrice **A** jednak jedinici. Pored toga, mnogi algoritmi za rješavanje raznih grafovskih problema svode se na direktnu manipulaciju sa elementima matrice **A**. Generalno, ovaj način zapisivanja je *pogodan za algoritme koji najviše operacija sprovode nad čvorovima grafa*.

Za memoriranje matrice **A** grafa sa *n* čvorova neophodno je n^2 *memorijskih lokacija* u računarskoj memoriji, *neovisno od broja grana* (uz dobro kodiranje, moguće je zauzeće memorije svesti na n^2 *bita*, jer je za pamćenje jednog elementa ove matrice dovoljan samo jedan bit). Stoga je ovaj način reprezentacije *veoma pogodan za guste*, ali *prilično nepraktičan za rijetke grafove*. Isto tako, ovaj način reprezentacije se lako uopćava za *multigrafove*. Tada se na poziciju $a_{i,j}$ stavlja *broj grana* koje spajaju čvorove *i* i *j*.

Blizak srodnik matrice susjedstva je ***težinska matrica*** **W**. Ona se definira za grafove kod kojeg je *svakoj njegovoj grani pridružena neka vrijednost* (*težina*). Za njene elemente $w_{i,j}$ vrijedi da je vrijednost $w_{i,j}$ jednaka *težini grane* koja spaja čvorove *i* i *j*. Za slučaj kada čvorovi *i* i *j* uopće nisu spojeni granom, vrijednost $w_{i,j}$ definira se *ovisno od primjene*, pri čemu je za potrebe najvećeg broja primjena u tom slučaju najpogodnije uzeti $w_{i,j} = \infty$, osim u slučaju $i = j$, kada je najpogodnije uzeti $w_{i,j} = 0$. Na primjer, neka je dat graf sa sljedeće slike, u kojem brojevi kraj grana odgovaraju težinama grana:

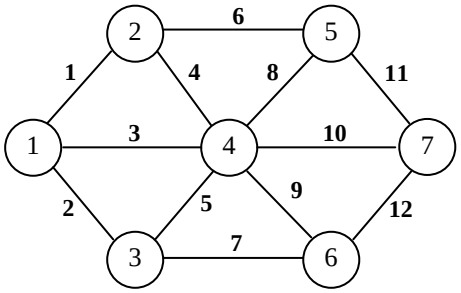


Težinska matrica za ovaj graf data je kao

$$\mathbf{W} = \begin{pmatrix} 0 & 7 & 4 & 5 & \infty & \infty & \infty \\ 7 & 0 & \infty & 8 & 1 & \infty & \infty \\ 4 & \infty & 0 & 7 & \infty & 2 & \infty \\ 5 & 8 & 7 & 0 & 4 & 3 & 11 \\ \infty & 1 & \infty & 4 & 0 & \infty & 4 \\ \infty & \infty & 2 & 3 & \infty & 0 & 4 \\ \infty & \infty & \infty & 11 & 5 & 4 & 0 \end{pmatrix}$$

U slučaju grafova kod kojih granama *nisu pridružene nikakve težine*, podrazumijeva se da *sve grane imaju težinu* 1. Ukoliko se tada odlučimo da za obilježavanje svih *nepostojećih grana* u težinskoj matrici iskoristimo vrijednost 0, tada se *težinska matrica svodi na matricu susjedstva*.

U nekim primjenama teorije grafova (naročito u primjenama kod kojih se grafovi koriste za modeliranje *električnih mreža*) pogodna je reprezentacija grafova pomoću ***matrica incidencije***. Ovaj način reprezentacije zahtijeva da izvršimo numeraciju ne samo čvorova grafa, nego i grana grafa (tako da se tačno zna koja je grana prva, koja druga, itd.). Neka imao neki graf sa *n* čvorova i *m* grana. Matrica incidencije **B** ovog grafa je matrica formata $n \times m$, pri čemu se sami elementi $b_{i,j}$ matrice **B** definiraju na različite načine za neusmjerene i usmjerene grafove. Razmotrimo prvo slučaj *neusmjerenih grafova bez petlji*. Za takve grafove, uzima se da je član $b_{i,j}$ jednak 1 ako i samo ako je čvor *i incidentan* sa granom *j*, u suprotnom je jednak 0. Na primjer, neka je dat graf kao na sljedećoj slici u kojem je, pored numeracije čvorova, izvršena i numeracija grana (broj kraj grane ovdje ne označava težinu, nego *redni broj* odgovarajuće grane):



Matrica incidencije za ovaj graf data je kao

$$\mathbf{B} = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Za slučaj *usmjerenih i hibridnih grafova*, matrica incidencije može sadržavati vrijednosti 0, 1 i –1. U tom slučaju je $b_{i,j}$ jednak 1 ukoliko grana *j izlazi* iz čvora *i*, –1 ukoliko grana *j ulazi* u čvor *i*, odnosno 0 ukoliko grana *j* i čvor *i nisu incidentni*.

Što se tiče *petlji*, one su dosta problematične, s obzirom da svaka petlja *i ulazi i izlazi* iz nekog čvora. Zbog toga se, u slučaju da je *j* petlja nad čvorom *i*, obično za slučaj neusmjerenih grafova uzima da je $b_{i,j} = 2$ (tj. 1 + 1), dok se za slučaj usmjerenih grafova uzima $b_{i,j} = 0$ (tj. 1 – 1).

Matrice incidencije su *veoma neekonomičan način* zapisivanja grafa sa aspekta *utroška memorije*, jer se u svakoj koloni ove matrice nalaze *samo dva nenulta elementa*. S druge strane, mnogi algoritmi za rad sa težinskim grafovima, koji se koriste u praktičnim primjenama, mogu se svesti na matrično množenje matrice incidencije ili nekih njenih submatrica sa težinskom matricom, ili nekim srodnim matricama. Pored toga, pojedini grafovski algoritmi, poput algoritama za nalaženje kontura unutar grafa, osjetno se pojednostavljuju u slučaju da je graf predstavljen matricom incidencije. Konačno, matrice incidencije imaju tu osobinu da su uvijek *totalno unimodularne*, što može biti od značaja prilikom modeliranja grafovskih problema kao problema matematičkog programiranja.

Postoje i razni *nematrični* načini za reprezentaciju grafova, koji su za neke primjene praktičniji od matričnih reprezentacija, pogotovo u slučaju *rijetkih grafova*. Recimo, neka je potrebno utvrditi da li je čvor *i* spojen granom *makar sa jednim drugim čvorom*. Ukoliko je graf opisan matricom susjedstva **A**, potrebno je testirati da li se u *i*-tom redu matrice **A** nalazi makar jedna jedinica, što u najgorem slučaju može zahtijevati *n* testiranja, gdje je *n* broj čvorova. Sličan problem se javlja ukoliko je potrebno pronaći *neki* (*bilo koji*) *čvor koji je spojen granom sa zadanim čvorom i* (ovaj problem se svodi na pronalaženje neke jedinice u *i*-tom redu matrice **A**). U slučajevima kada je efikasno davanje odgovora na ovakva pitanja presudno, bolje je za reprezentaciju grafova umjesto matrice susjedstva koristiti ***listu susjedstva*** (***adjacencije***). Lista susjedstva nekog grafa je lista (uređena *n*-torka) $L = (N_1, N_2, ..., N_n)$ pri čemu N_i predstavlja skup *svih čvorova koji su susjedni čvoru i* (tj. koji su povezani granom sa čvorom *i*). Na primjer, za isti graf koji smo maločas opisivali matricom susjedstva, lista susjedstva glasi

$$L = (\{2\}, \{1, 2, 3, 4, 5\}, \{2\}, \{2\}, \{2, 6\}, \{5\})$$

dok za graf koji smo opisivali matricom incidencije, lista susjedstva glasi

$$L = (\{2, 3, 4\}, \{1, 4, 5\}, \{1, 4, 6\}, \{1, 2, 3, 5, 6, 7\}, \{2, 4, 7\}, \{3, 4, 7\}, \{4, 5, 6\})$$

Jasno je da se pomoću listi susjedstva odgovor na oba prethodno postavljena pitanja može dobiti vrlo efikasno (test da li je čvor *i* spojen granom makar sa jednim čvorom svodi se na test da li je N_i prazan skup, dok za pronalaženje nekog čvora spojenog sa čvorom *i* možemo prosto uzeti bilo koji element skupa N_i). S druge strane, lista susjedstva ne omogućava naročito efikasno testiranje da li je čvor *i* spojen sa čvorom *j*, jer je za tu svrhu potrebno testirati da li je $j \in N_i$. Uz dobru implementaciju (koja uključuje čuvanje elemenata skupova N_i u sortiranom poretku i korištenje binarne pretrage), ovo testiranje može se izvesti u $\log_2 n_i$ koraka, gdje je n_i broj čvorova koji su spojeni sa čvorom *i*. U najgorem slučaju, ovaj broj koraka iznosi $\log_2 n$, gdje je *n* broj čvorova.

Treba napomenuti da se u mnogim računarskim implementacijama uzima da elementi N_i liste susjedstva L nisu skupovi, nego također *liste* (tj. uređene kolekcije elemenata), s obzirom da je efikasna implementacija skupova na računaru, što podrazumijeva efikasnu podršku pretraživanju, umetanju i brisanju elemenata, znatno komplikovanija od implementacije listi (s obzirom da efikasna implementacija skupova zahtijeva korištenje tzv. *balansiranih binarnih stabala*). Nažalost, liste se mogu pretraživati *isključivo sekvencijalno*, tako da u tom slučaju testiranje da li se element *j* nalazi u listi N_i može u najgorem slučaju zahtijevati *n* koraka, što je drastično pogoršanje u odnosu na $\log_2 n$ koraka koje imamo ukoliko zaista koristimo (propisno implementirane) skupove, a ne liste. Ipak, u slučaju *rijetkih grafova*, većina čvorova je spojena sa

relativno malo susjednih čvorova, tako da su brojevi n_i tipično *mali* u odnosu na n (tipično reda veličine ne većeg od $\log n$). U tom slučaju, reprezentacija kolekcija N_i putem listi je *sasvim prihvatljiva*.

Ranije je rečeno da matrice susjedstva nisu pogodne sa aspekta utroška memorije za rijetke grafove, jer se (uz maksimalno pogodno kodiranje) troši n^2 bita u memoriji neovisno od broja grana m. S druge strane, nije teško vidjeti da svi skupovi N_i, $i = 1, 2, ..., n$ koji su elementi liste susjedstva L imaju zajedno m elemenata. Ukoliko se za kodiranje svakog elementa skupova N_i koristi k bita, ukupan utrošak memorije je $m\,k$ bita. U slučaju *gustih grafova*, m je reda n^2, tako da u tom slučaju reprezentacija grafa pomoću listi susjedstva troši *više memorije* nego reprezentacija pomoću matrica susjedstva. Međutim, za *rijetke grafove*, m je reda n, tako da je u tom slučaju $m\,k$ mnogo manje od n^2, jer je k tipično *posve mali broj* (minimalan broj bita potreban da se kodiraju elementi skupova N_i iznosi $\log_2 n$, jer se radi o brojevima iz opsega od 1 do n). Slijedi da je, sa aspekta utroška memorije, reprezentacija rijetkih grafova pomoću listi susjedstva *mnogo isplativija od reprezentacije pomoću matrica susjedstva*, dok je u slučaju gustih grafova *obrnuto*.

U slučaju *neusmjerenih grafova*, memorijsko zauzeće prilikom njihove reprezentacije pomoću matrice susjedstva moguće je *dvostruko smanjiti* ukoliko se pamte *samo elementi matrice iznad ili ispod glavne dijagonale*, jer je tada svakako $a_{i,j} = a_{j,i}$. Za tu svrhu potrebno je kreirati i *posebnu strukturu podataka* (tzv. *grbavu matricu*). Ipak, ova memorijska ušteda dovodi do neznatnog kompliciranja algoritama koji rade sa takvom reprezentacijom. Recimo, ukoliko čuvamo samo elemente *ispod glavne dijagonale*, tada kad god algoritam treba da pristupi elementu $a_{i,j}$ kod kojeg je $j > i$, on zapravo treba pristupiti elementu $a_{j,i}$, s obzirom da takav element $a_{i,j}$ *uopće nije ni pohranjen*. Ovo zahtijeva dodatne testove u algoritmu. Načelno, dvostruko smanjenje zauzeća memorije moglo bi se izvesti i pri reprezentaciji grafova pomoću *listi susjedstva*, recimo tako što bi se u svakom od skupova N_i čuvali samo *čvorovi čiji je redni broj manji ili jednak od i*. Zaista, ukoliko je čvor i spojen granom sa čvorom j pri čemu je $j > i$, tada je i čvor j spojen granom sa čvorom i (pri čemu je $i < j$), pa će se informacija o tom spoju svakako naći u skupu N_j. Međutim, ova memorijska ušteda se *praktično nikada ne izvodi*, jer na taj način gubimo mogućnost da *efikasno saznamo koji su sve čvorovi spojeni sa čvorom i*. Zaista, skup N_i tada *ne sadrži kompletnu informaciju o tome*, već je potrebno pretražiti i *sve ostale skupove* u listi susjedstva, što je izrazito neefikasno.

Grafovi se mogu predstavljati i pomoću ***liste grana*** odnosno ***skupa grana***, što nije ništa drugo nego *lista* odnosno *skup parova* od kojih svaki par predstavlja po *jednu granu*, odnosno sadrži informaciju o njenom *početnom i krajnjem čvoru* (odnosno *krajevima* za slučaj *neusmjerenih grafova*, jer kod njih *nije bitno koji je početni a koji krajnji čvor*). Pri tome se, kod ovakve reprezentacije, u slučaju neusmjerenih grafova *nikada grane ne upisuju dvostruko* (tj. kao grana (i, j) i kao grana (j, i)). Razlika između liste i skupa grana je u tome što se kod liste grana zna koja je grana prva, koja druga, itd. dok kod skupa grana grane *nemaju nikakav redni broj* (što nam daje mogućnost da grane držimo u redoslijedu kakvom hoćemo). Recimo, za graf koji smo ranije prikazali matricom incidencije, lista grana glasila bi ovako, uvažavajući naznačeni redoslijed grana (parovi u ovoj listi mogu biti i *neuređeni*, s obzirom da svakako nije bitno koji je čvor početni a koji krajnji):

$$E = ((1, 2), (1, 3), (1, 4), (2, 4), (3, 4), (2, 5), (3, 6), (4, 5), (4, 6), (4, 7), (5, 7), (6, 7))$$

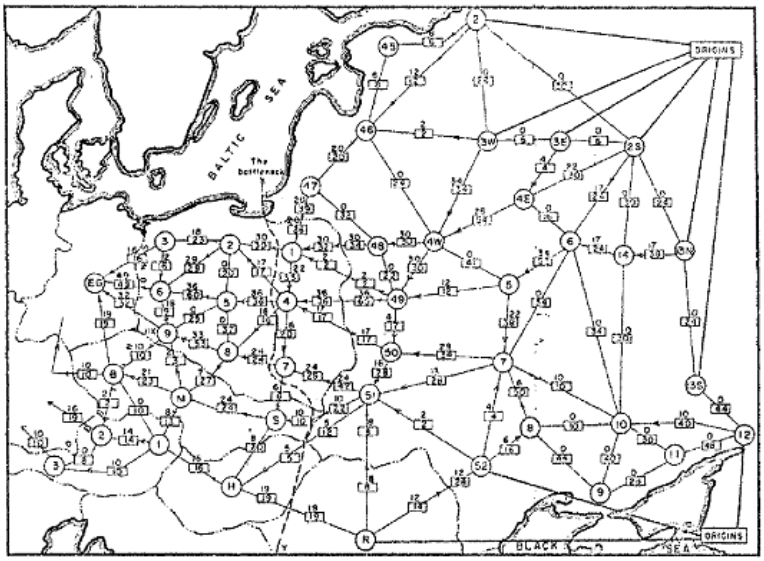
Možemo primijetiti da ukoliko graf posmatramo kao *usmjereni graf* odnosno *digraf*, tada skup grana nije ništa drugo nego *skup* R *iz definicije grafa*, uz pretpostavku da su elementi skupa X kodirani prirodnim brojevima od 1 naviše.

Mada način zapisa grafa pomoću listi ili skupova grana na prvi pogled *djeluje privlačno* zbog svoje *ekonomičnosti* (sa aspekta utroška memorije), ovaj način zapisa je *u svakom pogledu inferioran u odnosu na liste susjedstva*, čak i sa aspekta utroška memorije (jedino što je neznatno *jednostavniji za implementaciju*). Zaista, očigledno je za svaku granu potrebno zapamtiti dva čvora, što traži utrošak od $2\,m\,k$ bita u memoriji, uz pretpostavku da se troši k bita za pamćenje indeksa jednog čvora. Štaviše, ovakva reprezentacija je *vrlo nepodesna* za efikasno izvođenje algoritama nad grafovima. Na primjer, testiranje da li su čvorovi i i j susjedni zahtijeva pretragu da li se u listi odnosno skupu grana nalazi par (i, j), odnosno neki od parova (i, j) ili (j, i) za slučaj neusmjerenih grafova. Ova pretraga u najgorem slučaju zahtijeva čak m koraka za slučaj liste grana. Ukoliko se testiranje susjedstva čvorova izvodi često, algoritmi zasnovani na takvoj reprezentaciji grafa bili bi nedopustivo spori. U slučaju skupa grana, ovo se može svesti na $\log_2 m$ koraka, ukoliko grane čuvamo u nekom *sortiranom poretku* (što možemo jer nam poredak grana *nije bitan*) i pretragu vršimo postupkom *binarne pretrage*, ali je to i dalje *nepotrebno sporo*. Također, izvođenje mnogih drugih elementarnih algoritamskih koraka *ne može se efikasno izvesti sa ovakvom reprezentacijom grafa*.

Problem maksimalnog protoka i linearno programiranje

Na početku ovog poglavlja je rečeno da se veliki broj problema matematičkog programiranja može modelirati *jezikom teorije grafova*. Vrijedi i obrnuto, veliki broj problema teorije grafova mogu se prikazati kao *modeli matematičkog programiranja*, često i kao modeli *linearnog programiranja*. Jedan od prvih problema tog tipa, koji je uspješno riješen metodima teorije grafova, a koji također spada i u specijalan slučaj problema linearnog programiranja je tzv. ***problem maksimalnog protoka***. Mada u se u teoriji grafova prije razmatranja ovog problema prvo obrađuju načini rješavanja elementarnijih grafovskih problema, ovdje izlaganje započinjemo upravo sa ovim problemom, zbog njegove direktne povezanosti sa linearnim programiranjem. S druge strane, za mnoge druge vrste grafovskih problema, njihova eventualna povezanost sa linearnim programiranjem je često mnogo manje očita.

Ranih 1950-tih, istraživači ratnog vazduhoplovstva *SAD*-a *Ted Harris* i *Frank Ross* objavili su izvještaj koji proučava *željezničku mrežu* između tadašnjeg Sovjetskog saveza i zemalja istočne Evrope. Ta željeznička mreža imala je 44 stanice i 105 pruga, pri čemu je svaka pruga imala *kapacitet* koji predstavlja *maksimalnu količinu materijala* koji se može poslati tom prugom iz jedne regije u drugu regiju u određenom vremenskom razdoblju. Struktura te mreže prikazana je na sljedećoj slici.



Koristeći metodu pokušaja i grešaka, oni su odredili *maksimalnu količinu materijala* koji može da se prebaci iz Sovjeskog saveza u istočnu Evropu u posmatranom vremenskom razdoblju i *najjednostavniji način da se prekinu veze* u željezničkoj mreži, dizanjem u zrak *minimalnog* broja dijelova željezničke pruge. *Skup grana* (dijelova pruge) preko kojeg se *zaustavlja prenos* materijala, nazvali su *usko grlo*. Ovo je bila prva zabilježena aplikacija u praksi u kojoj se rješavao problem maksimalnog protoka.

Problem maksimalnog protoka se na jednostavan način može predstaviti na sljedeći način. Neka je dat *skup cijevi* sa *različitim protočnim kapacitetima*. Cijevi su spojene tako da formiraju neku *mrežu cijevi*. Potrebno je odrediti kolika je *maksimalna količina tečnosti* koja može da se kroz takvu mrežu prenese od neke *početne tačke* (*izvora*) do *krajnje tačke* (*odredišta*). Jasno je da se takva mreža može modelirati kao *usmjereni graf* kod kojeg *grane grafa* predstavljaju cijevi protočne mreže, a čvorovi predstavljaju *spojeve* tih cijevi. Pri tome, svaka grana grafa ima određeni *kapacitet* kojim je definirana *maksimalna količina* koja može proći kroz nju (tj. kroz odgovarajuću cijev).

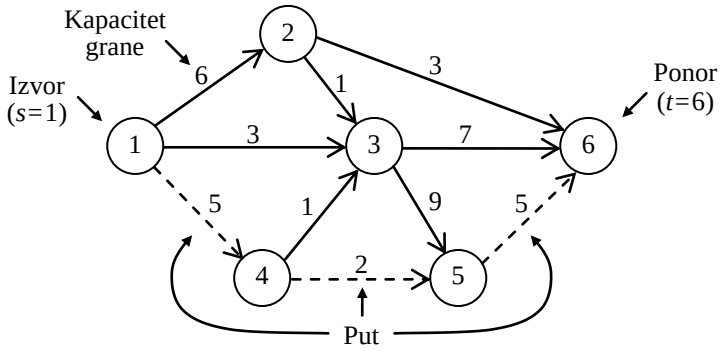
Zadatak nalaženja *maksimalnog protoka* se najčešće susreće u raznim tipovima *realnih transportnih zadataka*. Na primjer, ako je iz jednog grada (početna tačka grafa) za određeno vrijeme potrebno prebaciti *što više putnika* u drugi grad (krajnja tačka grafa) *mrežom* avionskog saobraćaja, dobija se upravo zadatak maksimalnog protoka. Podrazumijeva se da svaka relacija avionskog saobraćaja ima svoju *propusnu sposobnost*, tj. *gornju granicu* "tereta" koji se može po njoj prevesti za određeno vrijeme. Pri tome se pretpostavlja da se putnici ne mogu *ni ukrcavati ni iskrcavati u međustanicama*, inače se zadatak usložnjava. Sljedeća tablica ilustrira neke primjere šta u realnim primjenama mogu biti mreže, čvorovi, grane i protoci:

Mreža	Čvorovi	Grane	Protok
Komunikacije	– Računari – Sateliti	– Kablovi – Optička vlakna	– Audio paketi – Video paketi
Električna kola	– Registri – Sabirnice	– Žice	– Struja
Hidraulika	– Rezervoari – Pumpe	– Cijevi	– Fluidi
Finansije	– Dionice – Valute	– Transakcije	– Novac
Transport	– Aerodromi – Stanice	– Autoputevi – Pruge	– Teret – Putnici

Za formalno definiranje problema maksimalnog protoka potrebno je prvo uvesti pojam ***transportne mreže***. Pod transportnom mrežom podrazumijeva se svaki *usmjereni graf bez petlji* kod kojeg postoji neki *čvor s* nazvan ***ulaz mreže*** (ili ***izvor***) u koji *ne ulazi niti jedna grana grafa* i neki čvor *t* nazvan ***izlaz mreže*** (ili ***ponor***) iz kojeg *ne izlazi niti jedna grana grafa*, i kod kojeg je *svakoj grani* (i, j) *pridružen neki nenegativan broj* $u_{i,j}$ nazvan ***kapacitet*** ili ***propusna sposobnost*** grane. Funkcija $f(i, j)$ definirana na granama grafa (i, j) naziva se ***protok*** (ili ***fluks***) ukoliko vrijedi $0 \le f(i, j) \le u_{i,j}$ i ukoliko je za *svaki čvor k* osim čvorova *s* i *t*, *zbir vrijednosti* $f(i, k)$ za grane (i, k) koje *ulaze u čvor k* jednak *zbiru vrijednosti* $f(k, j)$ za grane (k, j) koje *izlaze iz čvora k*, odnosno

$$\sum_{(i,k)\in R} f(i,k) = \sum_{(k,j)\in R} f(k,j), \quad k \in \{1,2,...,n\} \setminus \{s,t\}$$

Ova relacija predstavlja *zakon o konzervaciji protoka* i odgovara ***prvom Kirchoffovom zakonu*** u teoriji električnih kola. ***Ukupan protok*** kroz transportnu mrežu definira se kao *zbir protoka grana koje izlaze iz čvora s*, odnosno *zbir protoka grana koje ulaze u čvor t* (lako se pokazuje da ovo dvoje mora biti jednako).



Pokazaćemo sada da se zadatak maksimalnog protoka može lako predstaviti u *obliku zadatka linarnog programiranja*. Neka je *n* broj čvorova grafa. Bez umanjenja općenitosti možemo uzeti da je izvor (početni čvor) označen sa 1, a ponor (krajnji čvor) sa *n*. Neka promjenljive $x_{i,j}$ predstavljaju *protoke* kroz grane koje vode od čvora *i* do čvora *j* i neka su $u_{i,j}$ kapaciteti odgovarajućih grana (ukoliko čvor *i* i *j* nisu spojeni granom, može se uzeti da je $u_{i,j} = 0$). Ukoliko sa *F* označimo *ukupni protok kroz mrežu*, tada se zadatak nalaženja maksimalnog protoka može zapisati u obliku

$$\arg\max F = \sum_{j=2}^{n} x_{1,j}$$

p.o.

$$\sum_{i=1}^{n} x_{i,k} - \sum_{j=1}^{n} x_{k,j} = 0, \quad k = 2..n-1$$

$$0 \le x_{i,j} \le u_{i,j}, \; i = 1..n, \; j = 1..n$$

Prva skupina ograničenja predstavljaju *jednačine očuvanja toka* za sve ostale čvorove osim izvora i ponora. One odražavaju činjenicu da u tim čvorovima *ne nastaje niti nestaje tok*, tako da ukupan tok koji *ulazi* u čvor mora biti jednak ukupnom toku koji *izlazi* iz čvora. Druga skupina ograničenja predstavljaju *ograničenja maksimalnog toka kroz grane u skladu sa njihovim kapacitetima*. Nekada se prvoj skupini ograničenja dodaje i ograničenje

$$\sum_{j=2}^{n} x_{1,j} - \sum_{i=1}^{n-1} x_{i,n} = 0$$

koje izražava činjenicu da je *ukupan protok koji prolazi kroz mrežu* jednak bilo *sumi protoka koji izlaze iz početne tačke grafa*, bilo *sumi protoka koji ulaze u krajnju tačku grafa*. Međutim, ovo ograničenje je *suvišno* (odnosno *redudantno*), jer se ono može izvesti kao *posljedica ostalih ograničenja* iz ove skupine.

Prethodni model sadrži n^2 promjenljivih, čak i za grane koje *ne postoje*. Broj promjenljivih se može znatno smanjiti ukoliko iz modela *uklonimo sve promjenljive* $x_{i,j}$ koje odgovaraju *nepostojećim granama*, odnosno uzimamo samo one promjenljive $x_{i,j}$ za koje grana (i,j) postoji u skupu grana R. Na taj način dobijamo skraćeni model koji sadrži m promjenljivih, gdje je m broj grana:

$$\arg\max F = \sum_{(1,j)\in R} x_{1,j}$$

p.o.

$$\sum_{(i,k)\in R} x_{i,k} - \sum_{(k,j)\in R} x_{k,j} = 0, \quad k = 2..n-1$$

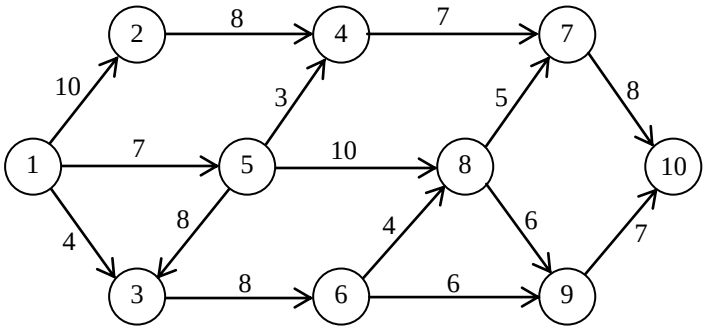
$$0 \le x_{i,j} \le u_{i,j}\ ,\ (i,j) \in R$$

Ovim ograničenjima se eventualno može dodati i (redudantno) ograničenje za jednakost ukupnog protoka iz izvora i u ponor oblika

$$\sum_{(1,j)\in R} x_{1,j} - \sum_{(i,n)\in R} x_{i,n} = 0$$

Nije teško pokazati da problem maksimalnog protoka, slično transportnom problemu, ima matricu ograničenja koja je *totalno unimodularna* (ona se efektivno sastoji od dvije submatrice od kojih je jedna *matrica incidencije* polaznog grafa, dok je druga *jedinična matrica*). Slijedi da problemi maksimalnog protoka kod kojih su kapaciteti svih grana cjelobrojni, *uvijek imaju cjelobrojno optimalno rješenje*. Drugim riječima, eventualna dodatna ograničenja na cjelobrojnost protoka kroz pojedine grane mreže *ni na kakav način ne utiču na rješavanje problema.*.

- ***Primjer*** : Za transportnu mrežu na slici formirati matematski model linearnog programiranja koji rješava problem maksimalnog protoka kroz ovu mrežu.



U skladu sa prethodno izvedenim modelom, za ovaj konkretan primjer dobijamo sljedeći model linearnog programiranja sa 15 promjenljivih (koliko ukupno ima grana) i 23 ograničenja:

$$\arg\max F = x_{1,2} + x_{1,3} + x_{1,5}$$

p.o.

$$x_{1,2} - x_{2,4} = 0$$
$$x_{1.3} + x_{5,3} - x_{3,6} = 0$$
$$x_{2,4} + x_{5,4} - x_{4,7} = 0$$
$$x_{1,5} - x_{5,3} - x_{5,4} - x_{5,8} = 0$$
$$x_{3,6} - x_{6,8} - x_{6,9} = 0$$
$$x_{4,7} + x_{8,7} - x_{7,10} = 0$$
$$x_{5,8} + x_{6,8} - x_{8,7} - x_{8,9} = 0$$
$$x_{6,9} + x_{8,9} - x_{9,10} = 0$$

$$0 \le x_{1,2} \le 10$$
$$0 \le x_{1,3} \le 4$$
$$0 \le x_{1,5} \le 7$$
$$0 \le x_{2,4} \le 8$$
$$0 \le x_{3,6} \le 8$$
$$0 \le x_{4,7} \le 7$$
$$0 \le x_{5,3} \le 8$$
$$0 \le x_{5,4} \le 3$$
$$0 \le x_{5,8} \le 10$$
$$0 \le x_{6,8} \le 4$$
$$0 \le x_{6,9} \le 6$$
$$0 \le x_{7,10} \le 8$$
$$0 \le x_{8,7} \le 5$$
$$0 \le x_{8,9} \le 6$$
$$0 \le x_{9,10} \le 7$$

Eventualno, ovim ograničenjima možemo dodati i redudantno ograničenje za jednakost ukupnog protoka iz izvora i u ponor:

$$x_{1,2} + x_{1,5} + x_{1,3} - x_{7,10} - x_{9,10} = 0$$

Ovako definirani modeli spadaju u *zadatke linearnog programiranja* i u načelu se mogu rješavati koristeći opće metode za rješavanje linearnog programiranja kao što je *simpleks algoritam*. Prvi problemi maksimalnog protoka su se zaista tako i rješavali, a i danas se ponekad tako radi ako imamo na raspolaganju neki univerzalni softver za rješavanje općih problema linearnog programiranja koji ne poznaje specijalističke metode za rješavanje problema maksimalnog protoka. Međutim, slično kao i kod transportnih problema, koristeći *specijalnu strukturu* zadataka nalaženja maksimalnog protoka, moguće je razviti *znatno efikasnije algoritme* za rješavanje ovih problema.

Interesantno je razmotriti i *dualni problem* problema maksimalnog protoka. Za tu svrhu, prikladno je uvesti *dvije odvojene grupe dualnih promjenljivih*. Prvu grupu čine dualne promjenljive y_i, $i = 2 \,..\, n-1$ koje su spregnute sa *ograničenjima koja predstavljaju jednačine očuvanja toka*. Drugu grupu čine dualne promjenljive $z_{i,j}$ za sve $(i, j) \in \mathrm{R}$ koje su spregnute sa *ograničenjima maksimalnog toka kroz grane u skladu sa njihovim kapacitetima*. Kada se pažljivo raspiše model problema maksimalnog protoka kao problema linearnog programiranja, nije teško vidjeti da odgovarajući dualni problem glasi

$$\arg\min C = \sum_{(i,j)\in \mathrm{R}} u_{i,j} z_{i,j}$$

p.o.

$$y_j + z_{1,j} \ge 1\,, \quad (1, j) \in \mathrm{R}, \quad j = 2 \,..\, n-1$$
$$z_{1,n} \ge 1\,, \quad \text{ako } (1, n) \in \mathrm{R}$$
$$-y_i + y_j + z_{i.j} \ge 0\,, \quad (i, j) \in \mathrm{R}, \quad i = 2 \,..\, n-1, \quad j = 2 \,..\, n-1$$
$$-y_i + z_{i,n} \ge 0\,, \quad (i, n) \in \mathrm{R}, \quad i = 2 \,..\, n-1$$
$$z_{i.j} \ge 0\,, \quad (i, j) \in \mathrm{R}$$

Recimo, za problem iz prethodnog primjera, dualni problem glasi

$$\arg\min C = 10\,z_{1,2} + 4\,z_{1,3} + 7\,z_{1,5} + 8\,z_{2,4} + 8\,z_{3,6} + 7\,z_{4,7} + 8\,z_{5,3} + 3\,z_{5,4} + \\ + 10\,z_{5,8} + 4\,z_{6,8} + 6\,z_{6,9} + 8\,z_{7,10} + 5\,z_{8,7} + 6\,z_{8,9} + 7\,z_{9,10}$$

p.o.

$$y_2 + z_{1,2} \geq 1$$
$$y_3 + z_{1,3} \geq 1$$
$$y_5 + z_{1,5} \geq 1$$
$$-y_2 + y_4 + z_{2,4} \geq 0$$
$$-y_3 + y_6 + z_{3,6} \geq 0$$
$$-y_4 + y_7 + z_{4,7} \geq 0$$
$$-y_5 + y_3 + z_{5,3} \geq 0$$
$$-y_5 + y_4 + z_{5,4} \geq 0$$
$$-y_5 + y_8 + z_{5,8} \geq 0$$
$$-y_6 + y_8 + z_{6,8} \geq 0$$
$$-y_6 + y_9 + z_{6,9} \geq 0$$
$$-y_8 + y_7 + z_{8,7} \geq 0$$
$$-y_8 + y_9 + z_{8,9} \geq 0$$
$$-y_7 + z_{7,10} \geq 0$$
$$-y_9 + z_{9,10} \geq 0$$

$$z_{1,2},\ z_{1,3},\ z_{1,5},\ z_{2,4},\ z_{3,6},\ z_{4,7},\ z_{5,3},\ z_{5,4},\ z_{5,8},\ z_{6,8},\ z_{6,9},\ z_{7,10},\ z_{8,7},\ z_{8,9},\ z_{9,10} \geq 0$$

Ukoliko po konvenciji definiramo da je $y_1 = 1$ i $y_n = 0$, dualni problem problema maksimalnog protoka može se zapisati i na sljedeći, kompaktniji i više simetričan način:

$$\arg\min C = \sum_{(i,j)\in \mathrm{R}} u_{i,j}\, z_{i,j}$$

p.o.

$$-y_i + y_j + z_{i.\,j} \geq 0\,, \quad (i,j) \in \mathrm{R}$$
$$z_{i.\,j} \geq 0\,, \quad (i,j) \in \mathrm{R}$$

Dualni problem problema maksimalnog protoka djeluje pomalo *misteriozno*, jer je na prvi pogled posve nejasno *značenje promjenljivih* koje se javljaju u ovom problemu. Da bismo ovo razjasnili, prvo treba primijetiti da je jasno da će, zbog *totalne unimodularnosti* matrice ograničenja u ovom problemu, sva bazna dozvoljena rješenja ovog problema biti cjelobrojna. Međutim, pažljivijim razmatranjem ograničenja, može se pokazati da bazna dozvoljena rješenja ne samo da će biti cjelobrojna, nego će u njima *sve promjenljive imati isključivo vrijednosti* 0 ili 1. Slijedi interesantan zaključak da su promjenljive dualnog problema za problem maksimalnog protoka u osnovi *logičke prirode*.

Dublji uvid u smisao dualnog problema za problem maksimalnog protoka dobijamo uvođenjem pojma ***reza*** ili ***presjeka*** (engl. *cut*) u transportnoj mreži. Pod *rezom* (P, Q) u transportnoj mreži $\mathrm{G} = (\mathrm{X}, \mathrm{R})$ sa *izvorom* s i *ponorom* t podrazumijevamo *svaku podjelu skupa čvorova* X na *dva disjunktna podskupa* P i Q ($\mathrm{X} = \mathrm{P} \cup \mathrm{Q}$, $\mathrm{P} \cap \mathrm{Q} = \varnothing$) takva da $s \in \mathrm{P}$ i $t \in \mathrm{Q}$. Nije teško pokazati da *svakom rezu* transportne mreže odgovara jedno *dopustivo bazno rješenje dualnog problema* kod kojeg je

$$y_i = \begin{cases} 1, \text{ako} i \in \mathrm{P} \\ 0, \text{ako} i \in \mathrm{Q} \end{cases} \qquad z_{i,j} = \begin{cases} 1, \text{ako} i \in \mathrm{P} \text{ i } j \in \mathrm{Q} \\ 0, \text{u suprotnom} \end{cases}$$

Vrijedi i obrnuto (samo što je to malo teže pokazati), odnosno *svakom dopustivom baznom rješenju dualnog problema* odgovara *neki rez u mreži* u skladu sa gore navedenim relacijama. Drugim riječima, dopustiva bazna rješenja dualnog problema zapravo predstavljaju *rezove u transportnoj mreži*.

Pod ***kapacitetom reza*** (P, Q) podrazumijeva se *suma kapaciteta svih grana čiji je početak u skupu* P *a kraj u skupu* Q. Ako pogledamo značenje dualnih promjenljivih $z_{i,j}$, odmah primijećujemo da *funkcija cilja C u dualnom problemu* nije ništa drugo nego *kapacitet reza*. Slijedi da je dualni problem problema maksimalnog protoka zapravo ***problem minimalnog reza*** (tj. problem *nalaženja reza sa minimalnim kapacitetom*). Odavde slijede neki *vrlo značajni zaključci*. Naime, prema svojstvu *slabe dualnosti*, funkcija cilja za svako dopustivo rješenje primalnog problema *uvijek je manja ili jednaka* od funkcije cilja za svako dopustivo rješenje dualnog problema. Slijedi da je *svaki dopustivi ukupni protok* uvijek manji ili jednak od *kapaciteta ma kojeg reza*, odnosno ukupni protok kroz mrežu *ograničen je odozgo kapacitetom ma kojeg reza*. Štaviše, prema svojstvu

jake dualnosti, optimalne vrijednosti funkcije cilja primala i duala su *jednake*. Drugim riječima, *maksimalni ukupni protok kroz mrežu jednak je kapacitetu reza sa minimalnim kapacitetom*. Ovo tvrđenje, poznato kao ***Ford-Fulkersonov teorem*** ili ***teorem "maksimalni protok, minimalni rez"*** (engl. *max-flow, min-cut*) i koje je *direktna posljedica opće teorije dualnosti*, predstavlja *jedan od najvažnijih teorema teorije grafova uopće*. Mnogi značajni rezultati teorije grafova (uključujući i ranije pominjani *Kőnig-Egerváryjev teorem*) mogu se, uz pogodnu interpretaciju, smatrati kao specijalan slučaj ovog teorema.

Veza između *problema maksimalnog protoka* i *problema minimalnog reza* je značajna i zbog još nekoliko razloga. Recimo, jasno je da svaki algoritam za rješavanje problema maksimalnog protoka ujedno *rješava i problem minimalnog reza*, što je značajno zbog činjenice da se problemi koji se mogu iskazati kao problem minimalnog reza *također javljaju u praktičnim primjenama*, neovisno od problema maksimalnog protoka. Pored toga, interesantna je činjenica da veza koja postoji između ova dva problema svodi problem maksimalnog protoka koji *nije problem kombinatorne optimizacije* (s obzirom da se u svakoj transportnoj mreži tipično može formirati *beskonačno mnogo različitih dopustivih protoka*) na problem minimalnog reza koji *spada u probleme kombinatorne optimizacije* (s obzirom da se u svakom grafu može formirati *samo konačno mnogo različitih rezova*).

Problem maksimalnog protoka može se *na razne načine poopćiti*. Na prvom mjestu, moguće je razmatrati *generalizacije transportnih mreža* u kojima se dopušta *više izvora* i/ili *više ponora*, pri čemu se tada pod ukupnim protokom smatra *ukupna suma protoka koji izlaze iz svih izvora* (odnosno *ukupna suma protoka koji ulaze u sve ponore*). Trivijalno je napraviti modifikaciju modela linearnog programiranja koja će obuhvatiti i ovakve slučajeve. Međutim, ovako modificirani problem može se lako svesti na obični *problem maksimalnog protoka* uvođenjem ***superizvora*** odnosno ***superponora*** koji napajaju *sve izvore* odnosno koji primaju protok iz *svih ponora* (kapaciteti fiktivnih grana koje spajaju superizvor sa izvorima odnosno ponore sa superponorom su beskonačni). Isto tako, lako je modelirati i situacije u kojima i *čvorovi imaju kapacitete*, odnosno kod kojih je ukupan protok koji prolazi kroz neki čvor *ograničen*. Ove situacije se lako svode na obične probleme maksimalnog protoka tako što se svaki čvor sa ograničenim kapacitetom *rascijepi na dva čvora* koji su povezani granom čiji je kapacitet jednak kapacitetu čvora.

Drugačiju vrstu generalizacija problema maksimalnog protoka dobijamo ukoliko pretpostavimo da protok kroz svaku granu ima svoju *cijenu* (po količinskoj jedinici protoka) $c_{i,j}$. Tada se može postaviti zadatak da se *tačno određena količina ukupnog protoka* prenese od izvora do ponora, ali *po što nižoj cijeni*. Ovakav problem naziva se ***problem najjeftinijeg protoka***. Mada se ovakav problem *ne može svesti* na obični problem protoka, trivijalno ga je izraziti u obliku *problema linearnog programiranja*. Zaista, dovoljno je u klasičnom problemu maksimalnog protoka funkciju cilja zamijeniti funkcijom cilja koja izražava ukupan trošak

$$Z = \sum_{(i,j)\in \mathrm{R}} c_{i,j} x_{i,j}$$

i dodati dopunsko ograničenje da je ukupan protok iz izvora jednak zadanoj vrijednosti (uz promjenu cilja optimizacije iz maksimizacije u minimizaciju). Moguće je i da postoje ograničenja na *minimalni dozvoljeni protok* kroz svaku od grana, kao i situacije u kojima *ne postoji niti jedan izvor niti ponor*, nego se samo zahtijeva da *određena količina tečnosti cirkulira* kroz graf, poštujući zadana ograničenja (u tom slučaju, govorimo o ***problemu cirkulacije***). Mada se svi problemi ovog tipa mogu rješavati metodima za rješavanje općih problema linearnog programiranja, njihova specijalna struktura dovela je do razvoja *posebnog metoda* za rješavanje problema ovog tipa. Ovaj metod, koji ovdje nećemo opisivati, naziva se ***mrežni simpleks***. On kombinira ideje iz klasičnog simpleks metoda sa nekim idejama teorije grafova i predstavlja *generalizaciju metoda* koji se koriste za rješavanje *transportnih problema* (koji se opet mogu posmatrati kao *specijalnan slučaj problema najjeftinijeg protoka*).

Algoritmi za rješavanje problema maksimalnog protoka

Kao što smo vidjeli, problem maksimalnog protoka je jedan od najvažnijih problema optimizacije nad grafovima koji strogo uzevši *ne spada u probleme kombinatorne optimizacije*, ali je *tijesno povezan sa njima*, s obzirom da njegov *dualni problem* (odnosno problem *minimalnog reza*) *spada u probleme kombinatorne optimizacije*. Zbog aktuelnosti problema, ovaj problem je predmet intenzivnih proučavanja i stalno se razvijaju sve bolji algoritmi za njihovo rješavanje. Neki od najvažnijih pobrojani su u sljedećoj tabeli, pri čemu rubrika "Red složenosti" označava red veličine broja operacija u *najgorem slučaju*, pri čemu se često rješenje nalazi *znatno brže* (oznaka C u tabeli označava *kapacitet mreže*, osnosno *sumu kapaciteta svih grana koje izlaze iz izvora*).

Godina	Otkrio	Metod	Red složenosti
1951	Dantzig	Simplex	$m\,n\,C$
1955	Ford, Fulkerson	Augmenting Path	$m\,F_{max}$
1970	Dinitz	Blocking Path	$n^2\,m$
1972	Edmonds, Karp	Shortest Path	$n\,m^2$
1972	Edmonds, Karp, Dinitz	Capacity Scaling	$m^2 \log_2 C$
1973	Dinitz, Gabow	Capacity Scaling	$m\,n \log_2 C$
1974	Karzanov	Preflow Push-Relabel	n^3
1983	Sleator, Tarjan	Dynamic Trees	$m\,n \log_2 n$
1986	Goldberg, Tarjan	FIFO Preflow Push-Relabel	$m\,n \log (n^2/m)$
1997	Goldberg, Rao	Length Function	$m\,n^{1+\varepsilon}$
...	...	...	...
2012	Orlin	Compact Abudance Graphs	$m\,n$

Konceptualno najjednostavniji specijalistički algoritam za rješavanje problema maksimalnog protoka je ***Ford-Fulkersonov algoritam***. Ideja ovog algoritma je da se *polazeći od nekog dopustivog protoka*, kroz iteracije *vrijednost ukupnog protoka stalno povećava* duž *određenih lanaca u grafu od izvora do ponora* sve dok se ne postigne *maksimalna moguća vrijednost ukupnog protoka*. Početni dopustivi protok se može naći ili nekom *empirijskom metodom*, ili se prosto može proći od *trivijalnog protoka* kod kojeg su protoci kroz svaku od grana *jednaki nuli* (ovakav protok je očigledno *dopustiv*).

Ford i Fulkerson *nisu tačno precizirali* kojim se tačno redoslijedom vrši povećavanje protoka, nego su to smatrali "*implementacionim detaljem*". Stoga postoji *veliki broj različitih implementacija* (verzija) ovog algoritma. Nažalost, taj "implementacioni detalj" pokazao se od presudne važnosti. Uz "loše implementacije", nađeni su čak i primjeri mreža u kojima se proces postepenog povećavanja protoka *ne završava u konačno mnogo koraka* (pa čak i *ne konvergira* ka traženom optimalnom protoku). Ovo je bilo poznato već Fordu i Fulkersonu, a najjednostavniji primjer kod kojeg se ovo može desiti konstruisao je *U. Zwick*. Ipak, ovakvi patološki primjeri zahtijevaju da kapaciteti barem nekih grana budu *iracionalni brojevi*. Ukoliko su svi kapaciteti *cjelobrojni*, pa čak i *racionalni brojevi*, svaka moguća strategija *mora dovesti do rješenja* u konačno mnogo koraka. Tačnije, ukoliko su svi kapaciteti *cijeli brojevi*, garantirano je da ukupan broj operacija tokom izvođenja algoritma ne može preći $m\,F_{max}$, gdje je F_{max} traženi optimalni protok. Na prvi pogled, izgleda da se ovim *ne sužava* praktična primjena algoritma, jer se iracionalni brojevi *po volji dobro mogu aproksimirati racionalnim brojevima*, nakon čega se *pogodnim izborom jedinice protoka* svi kapaciteti mogu svesti na cijele brojeve. Bez obzira na tu činjenicu, katastrofalno loše ponašanje nekih verzija algoritma u prisustvu iracionalnih kapaciteta govori nam da se mogu očekivati i loša ponašanja ukoliko se za računarsku implementaciju koriste *promjenljive sa pokretnim zarezom* (tj. aproksimacije realnih brojeva). Zaista, čak i sa vrlo razumnim kapacitetima, loše implementacije Ford-Fulkersonovog algoritma mogu se "zaglaviti" u beskonačnoj petlji prosto *zbog grešaka u zaokruživanju*! Pored toga, gornja granica za ukupan broj operacija $m\,F_{max}$ je *nedopustivo velika*, s obzirom da maksimalni protok F_{max} može biti veoma velik. Štaviše, sasvim je lako konstruisati primjer vrlo jednostavnih mreža (sa svega nekoliko grana) kod kojih loša implementacija Ford-Fulkersonovog algoritma zaista može tražiti upravo ovoliko operacija.

Sretna okolnost je što se opisani problemi mogu relativno lako prevazići. Obično se prva poznata verzija Ford-Fulkersonovog algoritma kod koje se opisani problemi *ne dešavaju* pripisuje *J. Edmondsu* i *R. Karpu*, iako je autorstvo ove verzije *prilično sumnjivo*. Naime, izgleda da su mnogo prije Edmondsa i Karpa za strategiju koju predlažu Edmonds i Karp znali već i Ford i Fulkerson, ali *nisu bili sigurni* da li ta strategija *uvijek rješava* opisane probleme, dok su Edmonds i Karp to uspjeli *dokazati* i pokazati da broj operacija u ovoj verziji algoritma ne može preći red $n\,m^2$. Drugi problem sa autorstvom je što je isti dokaz izveo i *E. A. Dinitz*, po nekima čak i prije Edmondsa i Karpa, samo što su Edmonds i Karp svoj dokaz *prije publicirali*. Uglavnom, danas je ova varijanta algoritma poznata kao ***Edmonds-Karpova verzija Ford-Fulkersonovog algoritma*** ili ponekad samo kao ***Edmonds-Karpov algoritam***, mada je jasno da se ne radi ni o kakvom posebnom algoritmu, nego samo o jednoj specijalnoj izvedbi Ford-Fulkersonovog algoritma.

Postoji još jedna interesantna verzija Ford-Fulkersonovog algoritma koju su također preložili Edmonds i Karp i, neovisno od njih, Dinitz. Ova verzija se naziva ***algoritam skaliranja kapaciteta*** (engl. *Capacity Scaling*). Za razliku od prethodne Edmonds-Karpove verzije u kojoj se povećanje protoka uvijek vrši duž *najkraćeg lanca* od izvora do ponora duž kojeg je povećanje protoka moguće, u ovoj verziji se povećanje protoka vrši duž *lanca sa najvećom mogućom propusnom moći* (*kapacitetom*), pri čemu je propusna moć (kapacitet) lanca jednaka *najmanjem od kapaciteta svih grana duž lanca*. Ovaj algoritam je nešto složeniji za implementaciju od prethodne verzije i za njega se garantira da je maksimalan broj operacija ograničen sa

$m^2 \log_2 C$ u osnovnoj verziji, odnosno sa $m\,n \log_2 C$ u poboljšanoj verziji, gdje je *C ukupni kapacitet grafa*. U slučajevima grafova sa mnogo čvorova i relativno malim ukupnim kapacitetom (odnosno grafova kod kojih je $\log_2 C$ znatno manje od n), ovaj algoritam može biti efikasniji od prethodnog. Međutim, sve ovo važi samo pod uvjetom da su kapaciteti grana racionalni brojevi. Ukoliko to nije slučaj, može se desiti da algoritam *ne terminira*, što je 1980. godine pokazao *M. Queyranne*.

Danas postoje i *znatno efikasniji* ali i *znatno složeniji* algoritmi za nalaženje maksimalnog protoka. Relativno je jednostavan još i ***Dinitzov algoritam*** koji se zasniva na nalaženju tzv. ***blokirajućih puteva*** (engl. *blocking paths*) odnosno ***slojevitih mreža*** (engl. *layered networks*). Poznate su i razne verzije algoritama zasnovane na korištenju tzv. ***predprotoka*** (engl. *preflow*) koji predstavljaju funkcije nad granama slične protocima, ali koje *ne zadovoljavaju zakon o konzervaciji toka*, nego dopuštaju "viškove" u čvorovima grafa. Prvi algoritam tog tipa predložio je *A. V. Karzanov*, a složenost izvedbe takvih algoritama varira od relativno jednostavne do izuzetno komplikovane. Do sada je po pitanju efikasnosti najviše odmakao *J. Orlin* čiji vrlo komplikovani algoritam zasnovan na iznimno komplikovanim strukturama podataka rješava problem maksimalnog protoka u vremenu koje ne prelazi iznos proporcionalan sa $m\,n$.

Ford-Fulkersonov algoritam (općenito)

Da bismo opisali ovaj algoritam, prvo moramo uvesti nekoliko pojmova. Neka smo kapacitet neke grane (i, j) označili sa $c_{i,j}$, a vrijednost protoka pridruženog toj grani sa $x_{i,j}$. Razlika $r_{i,j} = c_{i,j} - x_{i,j}$ naziva se ***rezerva*** (ili ***zaliha***) te grane. Rezerva grane zapravo govori *za koliko se maksimalno smije povećati protok kroz tu granu*. Grana čiji je protok jednak kapacitetu (tj. čija je rezerva jednaka nuli) naziva se ***zasićena*** ili ***potpuno iskorištena***, inače je ***nezasićena***. Grana čiji je protok jednak nuli naziva se ***neiskorištena***, inače je ***iskorištena***. Ukoliko je dat neki *lanac* od izvora do ponora (odnosno "put" od izvora do ponora koji ne mora poštovati orjentaciju grana), za neku granu duž tog lanca kažemo da je ***dobro usmjerena*** u odnosu na taj lanac ukoliko se njena orjentacija poklapa sa redoslijedom čvorova duž tog lanca. U suprotnom kažemo da je grana ***krivo usmjerena*** u odnosu na taj lanac. Lanac u kojem su *sve dobro usmjerene grane nezasićene* a *sve krivo usmjerene grane iskorištene* naziva se ***nezasićen lanac***. Za lanac koji *nije nezasićen* (tj. u kojem je *makar jedna dobro usmjerena grana zasićena* ili *makar jedna krivo usmjerena grana neiskorištena*) kažemo da je ***zasićen***.

U *zasićenim lancima* očigledno *nije moguće povećati protok kroz dobro usmjerene grane niti smanjiti protok kroz krivo usmjerene grane*, dok je to *moguće u nezasićenim lancima*, barem u određenim granicama (dok se ne dostignu kapaciteti grana ili dok se protok kroz granu ne spusti na nulu). Pretpostavimo sada da protok kroz svaku dobro usmjerenu granu *povećamo* a kroz svaku krivo usmjerenu granu *smanjimo* za isti iznos Δ. Lako je provjeriti da će novodobijeni protok i dalje zadovoljavati *zakon o konzervaciji protoka* odnosno *jednačine očuvanja toka*. Stoga će novodobijeni protok ostati legalan ukoliko δ nije veći niti od rezerve ijedne dobro usmjerene grane, niti od protoka ijedne krivo usmjerene grane. Isto tako, lako je provjeriti da će se nakon ove modifikacije, ukupan protok *povećati* za Δ. Drugim riječima, nezasićeni lanci omogućavaju da se, pomoću njih, ukupan protok poveća. Stoga se oni nazivaju i ***povećavajući lanci***.

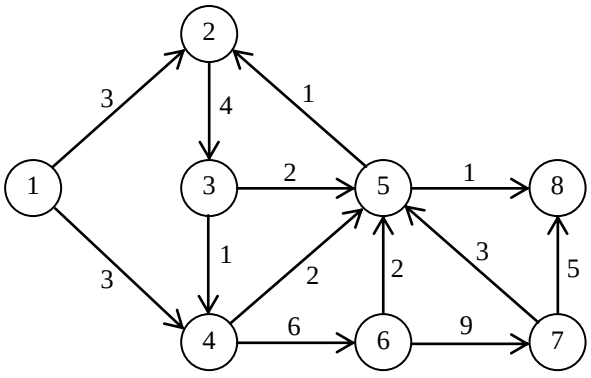
Kao što je već rečeno, Ford-Fulkersonov algoritam polazi od nekog poznatog dozvoljenog protoka (u nedostatku boljih informacija, moguće je krenuti i od trivijalnog protoka). Zatim se vrši potraga za nekim nezasićenim (povećavajućim) lancem od izvora do ponora. Ukoliko se takav lanac pronađe, vrši se povećanje protoka duž takvog lanca za maksimalno mogući iznos Δ_{max}. Očigledno je Δ_{max} jednako *minimumu od vrijednosti svih dobro usmjernih grana i vrijednosti protoka svih krivo usmjerenih grana* duž razmatranog lanca. Postupak se zatim nastavlja sve dok se više ne može pronaći *niti jedan povećavajući lanac od izvora do ponora*. U tom trenutku, *dostignut je maskimalni protok*. Naime, iz *Ford-Fulkersonove teoreme*, između ostalog slijedi da *ne postoje drugi načini da se ukupan protok kroz mrežu poveća i na kakav drugi način osim putem povećavajućih lanaca*. Treba voditi računa da lanac koji je bio povećavajući lanac neposredno nakon korekcije protoka više neće biti povećavajući lanac, ali da kasnije može ponovo postati povećavajući lanac nakon što se izvrši korekcija protoka duž nekog drugog lanca. Stoga je u svakoj iteraciji neophodno traganje za povećavajućim lancima vršiti *ispočetka*, a ne ograničiti se *samo na lance koji ranije nisu bili razmatrani*.

Osnovni problem sa gore navedenim grubim opisom Ford-Fulkersonovog algoritma je što se ništa ne govori o tome *kako* i *kojim redoslijedom* treba tražiti povećavajuće lance od izvora do ponora. U principu, za tu svrhu bi se mogao koristiti bilo koji *algoritam za traženje puteva kroz graf*, odnosno bilo koji od algoritama za *pretragu grafa*, pri čemu se u toku pretrage kroz *neiskorištene grane* smijemo kretati *samo u ispravnom smjeru*, kroz *zasićene grane samo u suprotnom smjeru* (tj. u smjeru koji je *suprotan od njihove orjentacije*), a kroz *ostale grane* (tj. *nezasićene ali iskorištene*) u *oba smjera*. Međutim, uskoro ćemo vidjeti da perfomanse algoritma mogu *drastično ovisiti* od načina kako se i kojim redoslijedom traže povećavajući

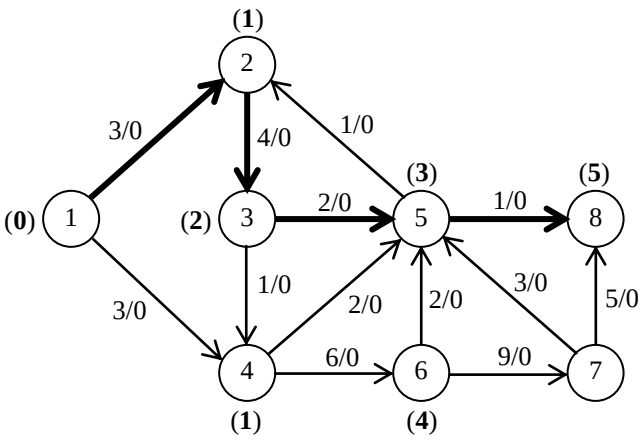
lanci. Ford i Fulkerson su izvorno za tu svrhu koristili tehniku nazvanu ***prosljeđivanje oznaka*** (***labela***). Izvoru se dodijeli oznaka 0. Zatim se u svakom narednom koraku uzima neki označeni čvor i njegov redni broj se *prosljeđuje* kao oznaka svim njegovim *neoznačenim susjedima u koje vodi nezasićena grana* i svim *neoznačenim čvorovima kojima je on susjed a u koje u njega vodi iskorištena grana* (u ovom drugom slučaju, oznaka se obično stavi sa znakom "–" kao indikacija da je oznaka proslijeđena *u suprotnom smjeru*). Postupak se nastavlja sve dok se ne *označi ponor*, ili dok *dalja prosljeđivanja više nisu moguća*. U prvom slučaju, povećavajući lanac je *nađen* i može se očitati prateći oznake *unazad* od ponora do izvora, dok u drugom slučaju povećavajući lanac *ne postoji*.

U gore opisanom Ford-Fulkersonovom postupku označavanja, još je uvijek ostalo nedorečeno *kojim redom prosljeđivati oznake čvorova*. Jedna *često korištena* (mada ne i najbolja) tehnika je da se u svakom koraku proslijedi *oznaka čvora sa najmanjim rednim brojem* od svih čvorova čije oznake još uvijek nisu proslijeđene, a koje se mogu dalje proslijediti (tj. koje se imaju kome proslijediti). Druga moguća strategija je da se u svakom koraku proba proslijediti oznaka *nekog od čvorova koji su dobili oznaku u prethodnom koraku* ukoliko je to moguće. Recimo, prvo se proba onaj od takvih čvorova sa *najmanjim rednim brojem*. Ukoliko se njegova oznaka ne može proslijediti, proba se sljedeći, itd. Ukoliko se oznaka niti jednog od čvorova označenih u prethodnom koraku ne može proslijediti, vraćamo se na neki od čvorova označenih u ranijim koracima, itd. Ova strategija označavanja svodi se zapravo na neku vrstu *pretrage grafa po dubini*, odnosno *DFS* (Depth First Search) *pretragu*. Ovo su dvije najčešće korištene strategije u naivnim izvedbama Ford-Fulkersonovog algoritma. Jasno je da su ovo samo dvije moguće od mnoštva mogućih strategija. Nažalost, kao što ćemo uskoro vidjeti, izbor dobre strategije traženja povećavajućih puteva je od *presudne važnosti* za efikasnost algoritma, pri čemu prethodno opisane strategije u općem slučaju *ne moraju voditi efikasnim realizacijama*.

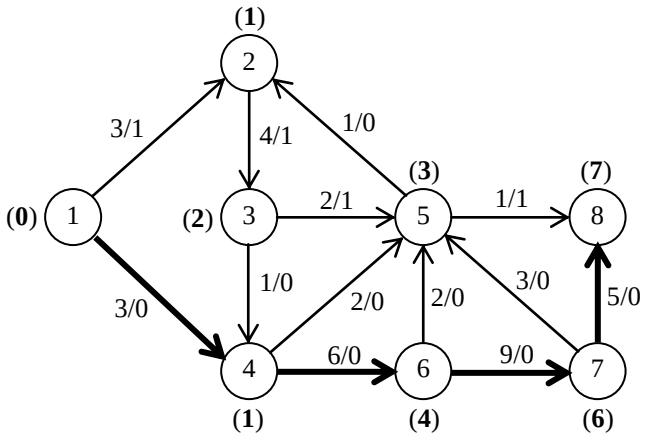
- ***Primjer***: Pomoću Ford-Fulkersonovog algoritma naći maksimalni protok u transportnoj mreži sa slike, koristeći tehniku prosljeđivanja labela za pronalaženje povećavajućih puteva:



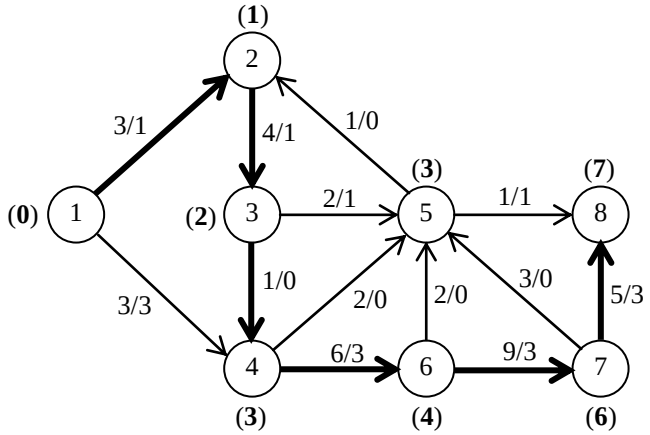
Krenućemo od trivijalnog protoka kao na sljedećoj slici (oznake kraj grana su tipa $c_{i,j} / x_{i,j}$, odnosno predstavljaju respektivno kapacitet i protok kroz granu). Ukoliko prosljeđivanje oznaka vršimo svaki put počev od čvora sa najmanjim indeksom čija se oznaka može proslijediti dalje, dobijamo situaciju kao na sljedećoj slici, koja daje povećavajući lanac $1 \to 2 \to 3 \to 5 \to 8$ (prikazan podebljano na slici):



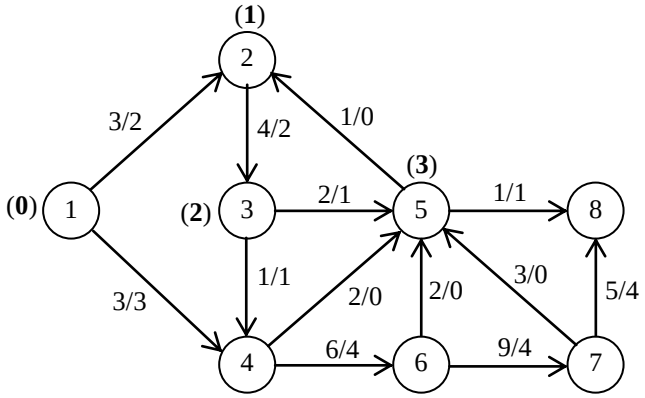
Duž ovog lanca imamo $\Delta_{max} = 1$, tako da protoke kroz sve grane lanca povećavamo za 1 (sve su dobro usmjerene), čime povećavamo ukupan protok za 1. U sljedećoj iteraciji na isti način nalazimo povećavajući lanac $1 \to 4 \to 6 \to 7 \to 8$:



Duž ovog lanca imamo $\Delta_{max} = 3$, tako da možemo ukupan protok povećati za 3, povećavajući za 3 protoke kroz sve grane lanca (ponovo nema krivo usmjerenih grana duž lanca). U sljedećoj iteraciji nalazimo povećavajući lanac $1 \to 2 \to 3 \to 4 \to 6 \to 7 \to 8$:

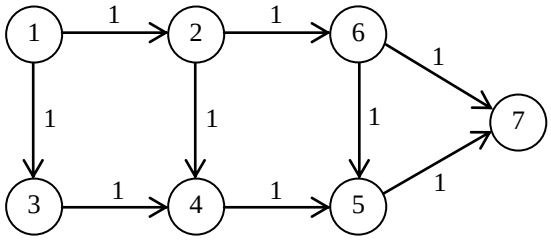


Ponovo možemo povećati ukupan protok za 1, nakon čega dolazimo do situacije prikazane na sljedećoj slici, gdje dalja prosljeđivanja više nisu moguća, a ponor nije označen. To zapravo znači da povećavajući lanac više *ne postoji*:

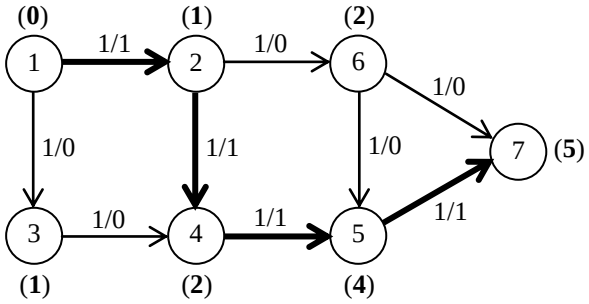


Kako više nema povećavajućih lanaca, pronađen je maksimalni ukupni protok $F_{max} = 5$. Pri tome, protoci kroz pojedine grane iznose $x_{1,2} = 2$, $x_{1,4} = 3$, $x_{2,3} = 2$, $x_{3,4} = 1$, $x_{3,5} = 1$, $x_{4,5} = 0$, $x_{4,6} = 4$, $x_{5,2} = 0$, $x_{5,8} = 1$, $x_{6,5} = 0$, $x_{6,7} = 4$, $x_{7,5} = 0$ i $x_{7,8} = 4$.

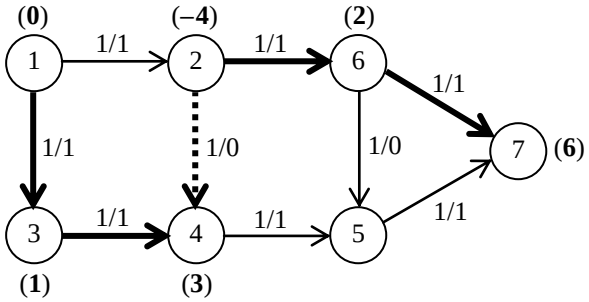
- ***Primjer***: Pomoću Ford-Fulkersonovog algoritma naći maksimalni protok u transportnoj mreži sa slike, koristeći tehniku prosljeđivanja labela za pronalaženje povećavajućih puteva:



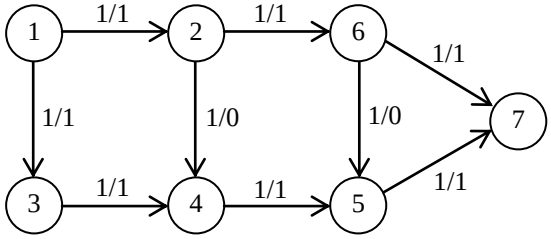
U prethodnom primjeru, svi povećavajući lanci sadržavali su samo dobro usmjerene grane. Ovaj primjer ilustrira slučaj u kojem će se javiti i krivo usmjerene grane u povećavajućem lancu, budemo li se slijepo držali opisanog postupka prosljeđivanja oznaka. Krenućemo ponovo od trivijalnog protoka. Mada se u ovom primjeru odmah "golim okom" uočavaju dva povećavajuća lanca $1 \to 2 \to 6 \to 7$ i $1 \to 3 \to 4 \to 5 \to 7$, nakon čijeg bismo iskorištavanja odmah došli do maksimalnog protoka $F_{max} = 2$, striktna primjena prethodno opisanog postupka prosljeđivanja oznaka pronaći će povećavajući lanac $1 \to 2 \to 4 \to 5 \to 7$:



Ovaj lanac možemo iskoristiti da povećamo ukupan protok za $\Delta_{max} = 1$ povećavajući protok duž svih njegovih grana za 1. Međutim, u narednoj iteraciji, jedini povećavajući lanac koji se može pronaći je lanac $1 \to 3 \to 4 \to 2 \to 6 \to 7$ koji sadrži i jednu *krivo usmjerenu granu* (prikazanu crtkano na slici):



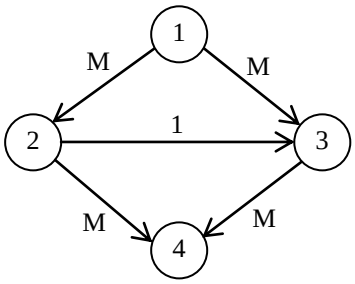
Za ovaj lanac je također $\Delta_{max} = 1$, pa možemo povećati ukupni protok za 1 povećavajući za 1 protok kroz dobro usmjerene grane $(1,3)$, $(3,4)$, $(2,6)$ i $(6,7)$, te *smanjujući protok* za isti iznos kroz krivo usmjerenu granu $(2,4)$. Time dobijamo situaciju prikazanu na sljedećoj slici u kojoj očigledno *nema povećavajućih puteva*, tako da je rješenje pronađeno na ovoj slici ujedno i optimalno:



Ford-Fulkersonov algoritam se lako može izvoditi i *direktno nad težinskom matricom grafa*. Kako to najbolje izvesti, objasnićemo nešto kasnije, kada razmotrimo Edmonds-Karpovu verziju Ford-Fulkersonovog

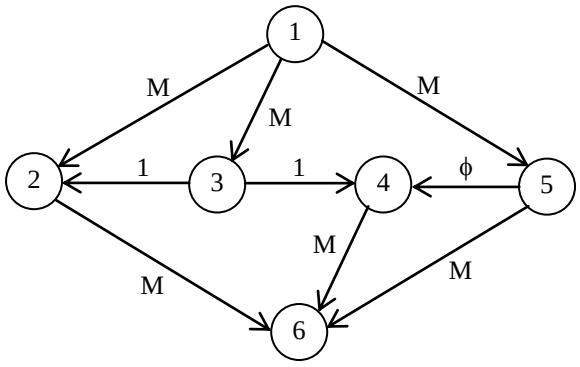
algortitma, s obzirom da se radi o jedinoj verziji Ford-Fulkersonovog algoritma koju se isplati praktično koristiti i programski realizirati.

Očigledno je da u prikazanoj izvedbi Ford-Fulkersonovog algoritma redoslijed traženja povećavajućih puteva može bitno ovisiti i od toga *kako su numerirani čvorovi*. Slijedeća dva primjera pokazuju u kojoj mjeri loš izbor povećavajućih puteva može loše djelovati na perfomanse algoritma. Neka je data transportna mreža na sljedećoj slici, gdje je M neki veliki prirodan broj:



Krenemo li od trivijalnog protoka, očigledno je da u ovoj transportnoj mreži postoje dva povećavajuća puta $1 \to 2 \to 4$ i $1 \to 3 \to 4$ koji, nakon što se oba iskoriste, daju maksimalni protok 2M. Međutim, pretpostavimo da neka loša varijanta Ford-Fulkersonovog algoritma prvo pronađe povećavajući lanac $1 \to 2 \to 3 \to 4$, duž kojeg se protok može povećati za 1, a nakon toga povećavajući lanac $1 \to 3 \to 2 \to 4$ (sa jednom *krivo usmjerenom* granom), duž kojeg se protok ponovo može povećati za 1. Nakon toga, lanac $1 \to 2 \to 3 \to 4$ ponovo postaje povećavajući lanac i algoritam bi sada mogao upasti u ciklus u kojem bi se u svakoj iteraciji protok uvećavao samo za 1, tako da bi bilo potrebno 2M iteracija da se dostigne maksimalni protok. Pošto M može biti po volji veliki broj, slijedi da i maksimalan broj iteracija može biti po volji veliki. Mada je vrlo malo vjerovatno da bi ijedna praktična implementacija Ford-Fulkersonovog algoritma na računaru za ovako jednostavan graf dovela do tako lošeg redoslijeda traženja povećavajućih puteva, ovaj primjer ilustrira barem teoretski kakvi se problemi mogu javiti (i zaista se javljaju u složenijim mrežama) pri naivnim implementacijama Ford-Fulkersonovog algoritma.

Izloženi primjer nije i najgora stvar koja se može desiti pri primjeni Ford-Fulkersonovog algoritma. Ako su svi kapaciteti grana cijeli brojevi, bar smo sigurni da algoritam *mora jednom završiti*. Naime, tada se u svakoj iteraciji ukupni protok mora povećati makar za 1. Kako je tada i maksimalni ukupni protok F_{max} također cijeli broj, slijedi da se algoritam sigurno mora završiti u najviše F_{max} iteracija. Slično se može pokazati da algoritam mora terminirati i kada su svi kapaciteti *racionalni brojevi*. Međutim, ukoliko među kapacitetima ima *iracionalnih brojeva*, čak *nije garantirano ni da algoritam mora terminirati*. Vjerovatno najgori primjer i najjednostavniji primjer šta se sve može desiti dao je *U. Zwick*. Razmotrimo graf na sljedećoj slici, gdje je M neki veliki broj, a $\phi = (\sqrt{5}-1)/2 \approx 0.618$ (ova konstanta je izabrana tako da vrijedi $1 - \phi = \phi^2$):



Nije teško vidjeti da je maksimalni protok u ovom grafu $2M + 1$, a može se postići krenemo li od trivijalnog protoka i primijenimo redom povećavajuće lance $1 \to 2 \to 6$, $1 \to 5 \to 6$ i $1 \to 3 \to 4 \to 6$. Međutim, pretpostavimo da neka verzija Ford-Fulkersonovog algoritma prvo pronađe povećavajući lanac $1 \to 3 \to 4 \to 6$, a nakon toga respektivno pronađe povećavajući lanac $1 \to 5 \to 4 \to 3 \to 2 \to 6$, zatim lanac $1 \to 3 \to 4 \to 5 \to 6$, zatim ponovo lanac $1 \to 5 \to 4 \to 3 \to 2 \to 6$, te nakon toga lanac $1 \to 2 \to 3 \to 4 \to 6$. Po završetku ovih 5 iteracija vrijednosti protoka kroz grane $(3,2)$, $(3,4)$ i $(5,4)$ te ukupan protok iznosiće

$$x_{3,2} = 1 - \phi^2, \qquad x_{3,4} = 1, \qquad x_{5,4} = \phi\,(1 - \phi^2), \qquad F = 1 + 2\,\phi + 2\,\phi^2$$

Nakon ovoga, posljednja četiri povećavajuća lanca mogu ponovo biti povećavajući lanci istim redom, te ćemo nakon naredne 4 iteracije imati

$$x_{3,2} = 1 - \phi^4, \qquad x_{3,4} = 1, \qquad x_{5,4} = \phi\,(1 - \phi^4), \qquad F = 1 + 2\,\phi + 2\,\phi^2 + 2\,\phi^3 + 2\,\phi^4$$

Četiri posljednja povećavajuća lanca ponovo mogu biti povećavajući lanci istim redom. Matematičkom indukcijom se može pokazati da ćemo nakon ukupno $4k + 1$ iteracija imati

$$x_{3,2} = 1 - \phi^{2k}, \qquad x_{3,4} = 1, \qquad x_{5,4} = \phi\,(1 - \phi^{2k}), \qquad F = 1 + 2\,\phi + 2\,\phi^2 + 2\,\phi^3 + \ldots + 2\,\phi^{2k}$$

a nakon toga posljednja četiri povećavajuća lanca ponovo mogu biti povećavajući lanci istim redom. Ovaj postupak se očigledno može nastaviti *unedogled*, što znači da pri ovakvom izboru povećavajućih lanaca postupak *nikad ne završava*. Što je najgore, vrijednost ukupnog protoka koja se na taj način postiže *uopće ne teži optimalnom protoku*. Zaista, kako iteracije odmiču, ukupna vrijednost protoka koja se postiže teži ka vrijednosti

$$F = 1 + 2\,\phi + 2\,\phi^2 + 2\,\phi^3 + \ldots = 1 + 2\sum_{k=1}^{\infty} \phi^k = 1 + \frac{2}{1 - \phi} = 4 + \sqrt{5} \approx 6.237$$

što, u slučaju da je M velik broj, nije *ni blizu* optimalnom protoku $F_{max} = 2\mathrm{M} + 1$. Dakle, pri ovakvom nesretnom izboru povećavajućih lanaca, Ford-Fulkersonov algoritam ne samo da *ne terminira*, nego čak i *ne konvergira* ka optimalnom rješenju!

Gore prikazani drastični primjeri lošeg ponašanja Ford-Fulkersonovog algoritma ipak nastaju samo pri *vrlo nesretnim odabirima* redoslijeda biranja povećavajućih lanaca, koji su malo vjerovatni u stvarnim situacijama. Recimo, dvije strategije za nalaženje povećavajućih lanaca koje smo opisivali kada smo govorili o prosljeđivanju oznaka, ako ništa drugo barem garantiraju da *algoritam uvijek terminira*, iako se i za ove strategije mogu formirati primjeri grafova u kojima će broj iteracija potreban da se dostigne optimalno rješenje biti *eksponencijalna funkcija* broja čvorova, što je neprihvatljivo loše. Srećom, postoji *jednostavna strategija za izbor povećavajućih lanaca* koja garantira da maksimalni broj iteracija potreban da se dostigne optimalno rješenje neće preći $m\,n\,/\,2$. O ovome će biti govora u sljedećem odjeljku.

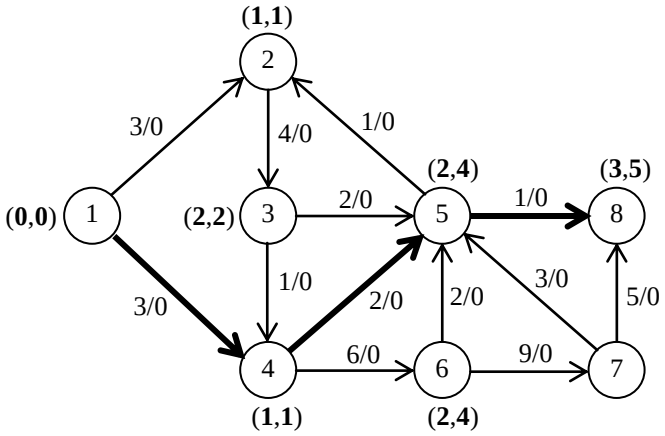
Edmonds-Karpova verzija Ford-Fulkersonovog algoritma

Varijanta Ford-Fulkersonovog algoritma koja se danas najviše koristi u praksi poznata je pod nazivom *Edmonds-Karpova verzija Ford-Fulkersonovog algoritma* ili ponekad samo *Edmonds-Karpov algoritam*, mada je jasno da se radi samo o specijalnoj verziji Ford-Fulkersonovog algoritma. U ovoj varijanti, u svakoj iteraciji se povećanje protoka vrši isključivo putem *najkraćeg od svih mogućih povećavajućih lanaca* od izvora do ponora (ili nekim od njih ukoliko takvih lanaca ima više), pri čemu se pod dužinom lanca prosto smatra *broj grana u lancu* (a ne zbir njihovih težina). Nije previše teško pokazati da ova strategija garantira da optimalno rješenje mora biti pronađeno u najviše $m\,n\,/\,2$ iteracija, a tipično se pronalazi nakon *mnogo manjeg broja iteracija*.

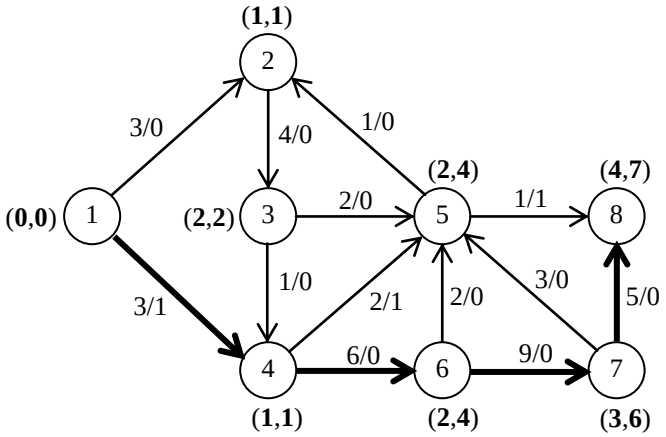
Ostaje još samo problem *kako naći najkraći povećavajući lanac*. U principu, to se radi *BFS pretragom* kroz graf, pri čemu se, kao i inače, u toku pretrage kroz neiskorištene grane smijemo kretati samo u ispravnom smjeru, kroz zasićene grane samo u suprotnom smjeru (tj. u smjeru koji je suprotan od njihove orjentacije), a kroz ostale grane (tj. nezasićene ali iskorištene) u oba smjera. Ovu pretragu isto možemo obaviti tehnikom prosljeđivanja oznaka čvorova, ali ćemo ovaj put staviti da se oznaka čvora sastoji od *dvije komponente*. Prva komponenta je *najkraća udaljenost čvora* od izvora (posmatrano u smjeru pretrage), a druga je, kao i do sada, redni broj čvora koji je izvršio prosljeđivanje. Svaki put kada proslijedimo redni broj nekog čvora nekom drugom čvoru, on će dobiti najkraću udaljenost za 1 veću od čvora koji je izvršio prosljeđivanje. Pri tome, prosljeđivanje vršimo prvo počev od čvora na najkraćoj udaljenosti 0 (tj. od izvora), zatim od svih čvorova na najkraćoj udaljenosti 1, zatim od svih čvorova na najkraćoj udaljenosti 2, itd. Pri implementaciji na računaru, da bi se ovo izvodilo efikasno, potrebno je u nekom spisku čuvati indekse svih čvorova koji su trenutno na istoj najkraćoj udaljenosti, da bismo uštedili vrijeme za pronalazak takvih čvorova. Ukoliko se sve ovo izvede kako treba, najkraći povećavajući put može se naći u vremenu proporcionalnom sa m, tako da je trajanje kompletnog algoritma u najgorem slučaju reda $m^2\,n$.

- ***Primjer***: Pomoću Edmonds-Karpove verzije Ford-Fulkersonovog algoritma naći maksimalni protok u transportnoj mreži iz prvog primjera u prethodnom odjeljku.

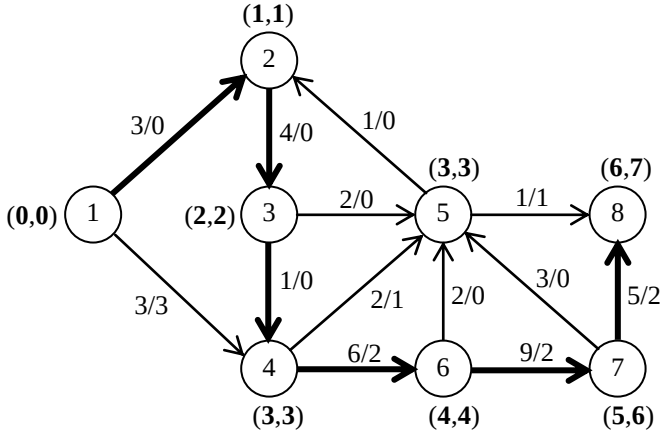
Polazeći od trivijalnog protoka, prethodno opisani postupak prosljeđivanja oznaka zasnovan na *BFS pretrazi* daje najkraći povećavajući lanac $1 \to 4 \to 5 \to 8$:



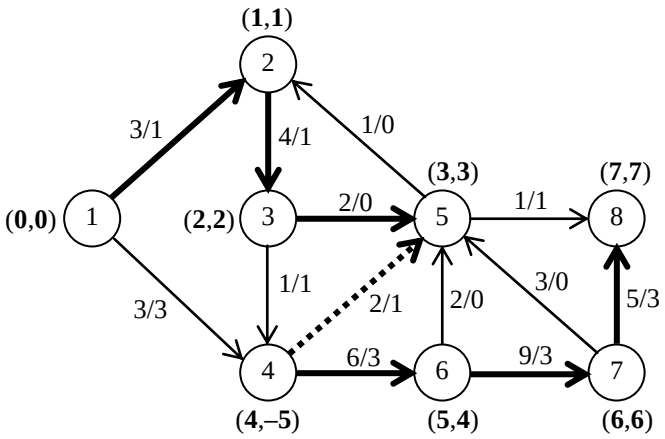
Duž ovog lanca sve grane su dobro usmjerene i imamo $\Delta_{max} = 1$, pa povećavamo ukupni protok za 1 povećavajući protok za 1 kroz sve grane duž ovog lanca. Sljedeća iteracija daje nam povećavajući lanac $1 \to 4 \to 6 \to 7 \to 8$:



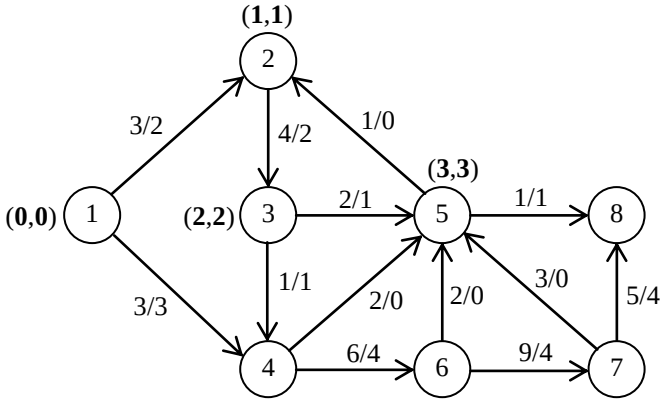
Ovaj lanac se ponovo sastoji samo od dobro usmjerenih grana, pri čemu je $\Delta_{max} = 2$. Stoga ćemo povećati za 2 protok kroz sve grane duž ovog lanca, čime će se i ukupni protok povećati za 2. U sljedećoj iteraciji nalazimo povećavajući lanac $1 \to 2 \to 3 \to 4 \to 6 \to 7 \to 8$:



Za ovaj lanac imamo $\Delta_{max} = 1$, pri čemu su ponovo sve grane dobro usmjerene. Nakon izvršene korekcije protoka, u sljedećoj iteraciji nalazimo povećavajući lanac $1 \to 2 \to 3 \to 5 \to 4 \to 6 \to 7 \to 8$, koji sadrži *krivo usmjerenu granu* $(4, 5)$:



Kako je za ovaj lanac $\Delta_{max} = 1$, povećaćemo za 1 protok kroz sve grane duž ovog lanca osim grane $(4, 5)$, dok ćemo protok kroz granu $(4, 5)$ *smanjiti* za 1. Ovim se ukupan protok povećava za 1. Nakon ovoga, postupak obilježavanja ne daje više niti jedan novi povećavajući lanac, tako da je optimalno rješenje *pronađeno*:



Vidimo da smo za dostizanje optimalnog rješenja utrošili *jednu iteraciju više* nego kod naivne verzije Ford-Fulkersonovog algoritma, što ukazuje na to da Edmonds-Karpova verzija *ne garantira* pronalaženje sekvence povećavajućih lanaca koja vodi do rješenja u najmanjem broju iteracija. S druge strane, ono što Edmonds-Karpova iteracija garantira je da *broj iteracija nikada ne može biti prevelik*, odnosno ograničen je odozgo sa $m\,n\,/\,2$.

Edmonds-Karpova verzija Ford-Fulkersonovog algoritma se (kao, uostalom, i bilo koja druga verzija Ford-Fulkersonovog algoritma) *lako izvodi direktno na težinskom matricom grafa* (tj. *matricom kapaciteta* **C**), što je pogodno u slučajevima kada je graf *nepregledan* za grafički prikaz ili kada je potrebno *isprogramirati* algoritam na računaru. Zapravo, praktičnije je vršiti manipulacije sa tzv. ***rezidualnom matricom*** **R** kod koje svakoj grani (i, j) odgovara polje na poziciji (i, j) čija je vrijednost $r_{i,j} = c_{i,j} - x_{i,j}$, ali i polje na poziciji (j, i) čija je vrijednost prosto $x_{i,j}$ (dakle, uzimamo da je $r_{j,i} = x_{i,j}$). U matričnom obliku, možemo pisati $\mathbf{R} = \mathbf{C} - \mathbf{X} + \mathbf{X}^{\mathrm{T}}$ gdje je **X** *matrica protoka kroz grane grafa*. Jasno je da je, u slučaju da smo krenuli od *trivijalnog protoka*, na početku prosto $\mathbf{R} = \mathbf{C}$. Ova matrica je interesantna zbog činjenice da njeni *nenulti elementi* odgovaraju *smjeru kretanja* kuda se smijemo kretati kroz graf pri traženju povećavajućih lanaca. Tačnije, ukoliko je u matrici **R** element na poziciji (i, j) različit od 0, tada i samo tada smijemo proslijediti redni broj čvora i kao oznaku čvoru j (naravno, pod uvjetom da čvor j već nije dobio ranije oznaku). Dakle, ne moramo voditi računa o smjerovima kojima se smijemo kretati kroz grane (da li smijemo ići u dobrom ili krivom smjeru, ili u oba), jer su te informacije *već kodirane u strukturi matrice* **R**. Nakon što pronađemo povećavajući lanac, dovoljno je sve elemente u matrici **R** koji odgovaraju povećavajućem lancu prosto *umanjiti* za Δ_{max}, a njima simetrične elemente u odnosu na dijagonalu matrice *uvećati* za Δ_{max} (ovo je očigledno zbog načina kako je formirana matrica **R**). Drugim riječima, ne trebamo se brinuti šta treba povećati a šta smanjiti u ovisnosti da li smo se kretali u pravom ili krivom smjeru. Po

završetku algoritma, odnosno kad više nema povećavajućih lanaca, optimalno rješenje možemo očitati na *dva načina*. Naime, za svaku granu (i, j) optimalnu vrijednost $x_{i,j}$ možemo očitati bilo iz polja $r_{i,j}$ na poziciji (i, j) kao $x_{i.j} = c_{i,j} - r_{i,j}$, bilo *direktno* iz polja na poziciji (j, i) kao $x_{i.j} = r_{j,i}$.

- ***Primjer***: Naći ponovo maksimalni protok u mreži iz prethodnog primjera, ali bez korištenja grafičke reprezentacije grafa, nego vršeći direktne manipulacije nad rezidualnom matricom grafa.

Na početku algoritma, s obzirom da krećemo od *trivijalnog protoka*, rezidualna matrica jednaka je težinskoj matrici (na mjestu nepostojećih grana svugdje imamo *nule* jer su nepostojeće grane sa aspekta protoka praktično ekvivalentne granama nultog kapaciteta). Stoga imamo

$$\mathbf{R} = \mathbf{C} = \begin{pmatrix} 0 & 3 & 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 6 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 2 & 0 & 9 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Radi preglednosti, predstavićemo ovu matricu u formi tablice i krećemo sa postupkom prosljeđivanja. Iz prvog reda vidimo da redni broj prvog čvora možemo proslijediti drugom i četvrtom čvoru (dakle, u drugi i četvrti red), itd. Oznake kraj strelica desno od *i*-tog reda odgovaraju oznakama koje su dodijeljene *i*-tom čvoru prilikom prosljeđivanja. Kada smo označili red 8 koji odgovara ponoru, povećavajući lanac je pronađen, i lako ga pronalazimo prateći oznake *unazad* (oznaka u redu 8 upućuje na red 5, oznaka u redu 5 na red 4, itd.). Ukoliko oznaka u redu k upućuje na red p_k, u povećavajućem putu se nalazi grana koja odgovara polju (p_k, k) u matrici. To ne mora nužno biti zaista grana (p_k, k), nego može biti i grana (k, p_k) ukoliko lanac kroz nju ide u *krivom smjeru*, ali ukoliko manipuliramo sa matricom **R**, to je za algoritam *potpuno nebitno*. Uglavnom, nakon prve iteracije dolazimo do situacije u sljedećoj tablici, pri čemu osjenčena polja odgovaraju povećavajućem lancu:

	1	2	3	4	5	6	7	8	
1	0	3	0	3	0	0	0	0	← (0, 0)
2	0	0	4	0	0	0	0	0	← (1, 1)
3	0	0	0	1	2	0	0	0	← (2, 2)
4	0	0	0	0	2	6	0	0	← (1, 1)
5	0	1	0	0	0	0	0	1	← (2, 4)
6	0	0	0	0	2	0	9	0	← (2, 4)
7	0	0	0	0	3	0	0	5	
8	0	0	0	0	0	0	0	0	← (3, 5)

Vrijednost $\Delta_{max} = 1$ nalazimo u *najmanjem osjenčenom polju*. Sada oduzimamo tu vrijednost od svih osjenčenih polja, a dodajemo na njima simetrična polja u odnosu na glavnu dijagonalu. Tako dobijamo novu rezidualnu matricu, za koju ponovo primjenjujemo isti postupak. Novodobijena situacija je kao u sljedećoj tablici:

	1	2	3	4	5	6	7	8	
1	0	3	0	2	0	0	0	0	← (0, 0)
2	0	0	4	0	0	0	0	0	← (1, 1)
3	0	0	0	1	2	0	0	0	← (2, 2)
4	1	0	0	0	1	6	0	0	← (1, 1)
5	0	1	0	1	0	0	0	0	← (2, 4)
6	0	0	0	0	2	0	9	0	← (2, 4)
7	0	0	0	0	3	0	0	5	← (3, 6)
8	0	0	0	0	1	0	0	0	← (4, 7)

Ovim je gotova i druga iteracija. Nakon umanjivanja svih osjenčenih polja za $\Delta_{max} = 2$ i uvećavanja njima simetričnh polja za isti iznos, te provođenja postupka prosljeđivanja, dolazimo do situacije prikazane u sljedećoj tablici:

	1	2	3	4	5	6	7	8	
1	0	3	0	0	0	0	0	0	← (0, 0)
2	0	0	4	0	0	0	0	0	← (1, 1)
3	0	0	0	1	2	0	0	0	← (2, 2)
4	3	0	0	0	1	4	0	0	← (3, 3)
5	0	1	0	1	0	0	0	0	← (3, 3)
6	0	0	0	2	2	0	7	0	← (4, 4)
7	0	0	0	0	3	2	0	3	← (5, 6)
8	0	0	0	0	1	0	2	0	← (6, 7)

Sada je $\Delta_{max} = 1$. Nakon nove korekcije matrice **R** i još jednog postupka prosljeđivanja, dobijamo situaciju kao u sljedećoj tablici:

	1	2	3	4	5	6	7	8	
1	0	2	0	0	0	0	0	0	← (0, 0)
2	1	0	3	0	0	0	0	0	← (1, 1)
3	0	1	0	0	2	0	0	0	← (2, 2)
4	3	0	1	0	1	3	0	0	← (4, 5)
5	0	1	0	1	0	0	0	0	← (3, 3)
6	0	0	0	3	2	0	6	0	← (5, 4)
7	0	0	0	0	3	3	0	2	← (6, 6)
8	0	0	0	0	1	0	3	0	← (7, 7)

Ovdje ponovo imamo $\Delta_{max} = 1$. Umanjićemo sve osjenčene elemente za 1 i ponoviti isti postupak, nakon čega dolazimo do situacije u kojoj prosljeđivanjem *ne možemo stići* do reda 8 koji odgovara ponoru, što znači da je *nađeno optimalno rješenje*:

	1	2	3	4	5	6	7	8	
1	0	1	0	0	0	0	0	0	← (0, 0)
2	2	0	2	0	0	0	0	0	← (1, 1)
3	0	2	0	0	1	0	0	0	← (2, 2)
4	3	0	1	0	2	2	0	0	
5	0	1	1	0	0	0	0	0	← (3, 3)
6	0	0	0	4	2	0	5	0	
7	0	0	0	0	3	4	0	1	
8	0	0	0	0	1	0	4	0	

Traženo optimalno rješenje $x_{1,2} = 2$, $x_{1,4} = 3$, $x_{2,3} = 2$, $x_{3,4} = 1$, $x_{3,5} = 1$, $x_{4,5} = 0$, $x_{4,6} = 4$, $x_{5,2} = 0$, $x_{5,8} = 1$, $x_{6,5} = 0$, $x_{6,7} = 4$, $x_{7,5} = 0$ i $x_{7,8} = 4$ možemo sada lako očitati na bilo koji od gore dva opisana načina (za tu svrhu, neophodno je pogledati i matricu **C** da bismo utvrtili koje su grane stvarno postojale u polaznom grafu, jer je ta informacija *izgubljena* u matrici **R**).

Problem najkraćeg puta i linearno programiranje

Problem najkraćeg puta spada u još jedan značajan ekstremalni zadatak teorije grafova koji spada u oblast *kombinatorne optimizacije*, a koji se može *lako modelirati kao problem linearnog programiranja*. S druge strane, ovaj problem se *gotovo nikada ne rješava kao problem linearnog programiranja* (osim možda ukoliko pri ruci nemamo nikakav *namjenski softver* za tu svrhu, a imamo na raspolaganju *univerzalni softver* za rješavanje problema linearnog programiranja). Za to postoje najmanje dva razloga. Prvo, za rješavanje problema najkraćeg puta postoje *brojni mnogo efikasniji algoritmi* u odnosu na opće algoritme za rješavanje problema linearnog programiranja. Drugo, slično kao kod problema raspoređivanja, dopustiva bazna rješenja ovog problema predstavljenog modelom linearnog programiranja su u većini slučajeva *degenerirana sa visokim stepenom degeneracije*.

Zadatak *traženja najkraćeg puta* javlja se u mnogim praktičnim problemima u različitim oblastima, od kojih vrijedi istaći sljedeće:

- Nalaženje *najkraće relacije* između bilo koja dva mjesta na karti sa ucrtanom *mrežom saobraćajnica* i *naznačenim dužinama svih dionica* (to se može koristiti za potrebe *kopnenog, željezničkog, cestovnog* i raznih drugih vrsta saobraćaja);
- Izrada *daljinara na autokartama*;
- Primjene u *računarskim naukama*, koje podrazumijevaju sve vrste *umrežavanja, rješavanja upita, transporta podataka,* i slično;
- Primjene u *komunikacionim tehnologijama*;
- Primjene u *mrežnim sistemima općenito* (električne mreže, internet, itd.).

Da bismo preciznije iskazali šta se podrazumijeva pod *najkraćim putem*, podsjetimo se da se *put* između neka dva čvora u grafu definira kao *niz lûkova* (*orjentiranih grana*) kod kojeg se *kraj jednog lûka* poklapa sa *početkom sljedećeg lûka* u nizu. Prema tome, put je *sekvenca* ili *niz lûkova* a ne prosto skup lûkova, odnosno redoslijed lûkova je jako bitan. Pretpostavićemo da je svakom lûku pridružena *neka brojčana vrijednost*, za koju ćemo uzeti da predstavlja *njegovu dužinu* (nekada se umjesto *dužine* govori i o *cijeni* grane). Tada se pod ***dužinom puta*** između dva čvora podrazumijeva *suma dužina pripadajućih lûkova,* odnosno lûkova koji obrazuju taj put. ***Rastojanje između dva čvora*** definira se kao *dužina najkraćeg puta* između ta dva čvora, pri čemu se uzima da je *rastojanje čvora od samog sebe jednako nuli*. Isto tako, uzima se da je *rastojanje između čvorova koji pripadaju različitim komponentama povezanosti* jednako $+\infty$.

U skladu sa usvojenim definicijama, *zadatak nalaženja najkraćeg puta* može se formalno definirati na sljedeći način. Neka je dat (orjentirani) graf $G = (X, R)$ i neka je određen *polazni čvor* $s \in X$, pri čemu je svakom lûku (i, j) iz R pridružena određena vrijednost $l_{i,j}$. Potrebno je odrediti *najkraći put od čvora s do nekog drugog zadanog čvora* $t \in X$. Dužine $l_{i,j}$ su u većini primjena *nenegativne*, mada postoje i primjene u kojima su moguće *negativne dužine grana* (to se recimo dešava u nekim ekonomskim modelima u kojima grane modeliraju *finansijske transakcije*).

Ovako definirani zadatak nalaženja najkraćeg puta naziva se i ***problem najkraćeg puta tipa "jedan do jednog"*** (engl. *one-to-one*) odnosno ***problem najkraćeg puta između jednog para čvorova*** (engl. *single-pair shortest path*), s obzirom da se traži najkraći put između *fiksna dva čvora*. Međutim, većina algoritama za rješavanje problema najkraćeg puta zapravo rješava *nešto općenitiji zadatak* u kojem se određuju najkraći putevi od čvora *s* do *svih drugih čvorova i iz* X, odnosno do *čvorova iz nekog podskupa skupa* X, koji između ostalog sadrži i ciljni čvor *t*. U ovom slučaju govorimo od ***problemu najkraćeg puta tipa "jedan do svih"*** (engl. *one-to-all*) odnosno ***tipa "jedan do nekih"*** (engl. *one-to-some*). Problemi ovog tipa nazivaju se i ***problemi najkraćeg puta sa jedinstvenim polazištem*** (engl. *single-source shortest path*). Analogno se mogu definirati i ***problemi najkraćeg puta sa jedinstvenim odredištem***.

Problem najkraćeg puta sa *jedinstvenim polazištem* (dakle tipa "jedan do svih" ili "jedan do nekih") može se iskazati i na sljedeći način. U datom grafu, potrebno je naći puteve sa *najmanjom dužinom* (odnosno *najmanjom ukupnom cijenom*) od *polaznog* (*korijenskog*) *čvora* do *svih ostalih čvorova u grafu* ili do *svih čvorova unutar nekog podskupa čvorova u grafu*. Ovi najkraći putevi mogu se predstaviti (usmjerenim) *stablom* T *sa korijenom s* koje ima osobinu da u njemu *postoji jedinstven put* od korijena *s* do bilo kojeg drugog čvora *i* u grafu (ili nekom podskupu čvorova u grafu) pri čemu taj put upravo predstavlja *najkraći put*. Ovo (usmjereno) stablo se naziva ***stablo najkraćih puteva***. Većina algoritama za rješavanje problema najkraćeg puta upravo formira ovakvo stablo.

Moguće je zahtijevati da se pronađu *svi najkraći putevi od ma kojeg čvora u grafu do ma kojeg drugog čvora u grafu*. U tom slučaju govorimo o ***problemu najkraćeg puta tipa "svi do svih"*** (engl. *all-to-all*) odnosno ***problemu najkraćeg puta između svih parova čvorova*** (engl. *all-pairs shortest path*). Jasno je da je u grafu sa *n* čvorova moguće konstruisati *n odvojenih stabala najkraćih puteva*, uzimajući svaki put drugi čvor kao početni čvor. Na taj način, problem najkraćeg puta tipa "svi do svih" može se svesti na rješavanje *n* problema najkraćeg puta tipa "jedan do svih". Međutim, *to nije i najbolji način za rješavanje ovakvih problema,* s obzirom da za probleme najkraćeg puta tipa "svi do svih" postoje specijalistički algoritmi koji ovaj problem rješavaju *brže* nego što traje *n* izvršavanja algoritama za rješavanje problema najkraćeg puta tipa "jedan do svih".

Sada ćemo pokazati na koji način se *problem najkraćeg puta može modelirati kao problem linearnog programiranja*. Pri tome ćemo se ograničiti na problem najkraćeg puta između *jednog fiksnog para čvorova* (tj. tipa "jedan do jednog"), jer se jedino ovaj tip problema najkraćeg puta može lako modelirati kao problem linearnog programiranja. Za graf ćemo pretpostaviti da je *orjentiran*, a eventualno prisustvo neorjentiranih grana modeliraćemo kao *par suprotno orjentiranih grana* (tj. koristićemo *model digrafa*). Isto tako, radi

pojednostavljenja razmatranja, bez umanjenja općenitosti ćemo smatrati da je polazni čvor numeriran sa 1 a ciljni čvor sa n, jer se ovo uvijek može postići pogodnom numeracijom čvorova.

Ukoliko se zadatak *nalaženja najkraćeg puta* posmatra kao *zadatak optimizacije*, onda traženi najkraći put predstavlja *ekstrem* (*minimum*) funkcije koja definira *ukupnu cijenu* (mjeru) prelaska *iz početne u krajnu tačku* grafa. Ta je funkcija zadana kao *zbir* "*dužina*" *grana od početne do krajnje tačke*. U ovako definiranom zadatku, potrebno je odrediti *koje grane povezuju početnu tačku sa krajnjom* tako da *zbir njihovih dužina bude minimalan*.

Teoretski, *bilo koja grana grafa može da se nađe u najkraćem putu*, a da li će se neka konkretna grana zaista u tom putu i naći, ovisi od raspodjele dužina svih grana u grafu. Zbog toga će u funkciji cilja *figurirati sve grane grafa*. Kao promjenljive ćemo, slično kao kod problema raspoređivanja, uvesti *logičke promjenljive* $x_{i,j}$ koje mogu imati *samo vrijednosti* 0 ili 1, pri čemu $x_{i,j} = 1$ znači da se grana (i, j) *nalazi* u traženom najkraćem putu, dok $x_{i,j} = 0$ znači da se grana (i, j) *ne nalazi* u traženom najkraćem putu. Naravno, nije potrebno uvoditi promjenljive za grane koje se *ne nalaze u grafu*. Drugim riječima, promjenljive $x_{i,j}$ će postojati samo za one indekse i i j za koje je $(i, j) \in \mathrm{R}$. Slijedi da se funkcija cilja može prikazati u obliku

$$S = \sum_{(i,j)\in \mathrm{R}} l_{i,j} x_{i,j}$$

gdje su $l_{i,j}$ dužine grana grafa a $x_{i,j}$ predstavljaju *indikatorske* 0–1 promjenljive koje govore da li se odgovarajuća grana nalazi ili ne nalazi u traženom najkraćem putu. Zadatak nalaženja nakraćeg puta sad se svodi na ustanovljavanje *koje su od promjenljivih* $x_{i,j}$ *jednake jedinici, a koje nuli*.

Da bismo *smanjili broj promjenljivih*, možemo primijetiti da ukoliko su dužine svih grana *nenegativne, možemo eliminirati* sve grane koje *ulaze* u početni čvor, kao i grane koje *izlaze* iz završnog čvora, s obzirom da takve *sigurno ne mogu pripadati najkraćem putu*. Zaista, kada bi pripadale, takav put bi imao *ciklus* koji prolazi kroz početni odnosno kroz završni čvor, a najkraći put *ne može sadržavati cikluse* (jer bi eliminacijom grana koje obrazuju ciklus dobili put koji je kraći). Jedina iznimka mogla bi nastupiti ukoliko bismo u grafu imali *cikluse negativne ukupne težine*, što je nemoguće ukoliko su sve dužine grana nenegativne.

Razmotrimo sada *kakva ograničenja* moraju zadovoljavati promjenljive $x_{i,j}$. S obzirom da traženi najkraći put prema pretpostavci *počinje* u čvoru 1, onda iz tog čvora izlazi *jedna i samo jedna grana koja pripada najkraćem putu*. Odavde slijedi ograničenje

$$\sum_{(1,j)\in \mathrm{R}} x_{1,j} = 1$$

Isto tako, kako po pretpostavci najkraći put *završava* u čvoru n, onda u taj čvor ulazi *jedna i samo jedna grana koja pripada najkraćem putu*. Zato se za *krajnji čvor* može napisati ograničenje

$$\sum_{(i,n)\in \mathrm{R}} x_{i,n} = 1$$

Što se tiče *ostalih čvorova grafa* (koji nisu niti polazni niti završni) vrijedi da u razmatrani čvor ili *tačno jedna grana u nju ulazi i tačno jedna grana iz nje izlazi*, ili *nijedna grana niti ulazi niti izlazi*. Zbog toga se za sve ostale čvorove može napisati ograničenje

$$\sum_{(i,k)\in \mathrm{R}} x_{i,k} = \sum_{(k,j)\in \mathrm{R}} x_{k,j} \ , \ k = 2 \mathinner{..} n-1$$

Na prvi pogled, izgleda da su potrebna ograničenja i da je lijeva ili desna strana u prethodnim jednakostima *manja ili jednaka od* 1. Zaista, ovako kako su ova ograničenja napisana, izgleda da nas ništa ne sprečava da sume koje se javljaju u ovim ograničenjima budu jednake recimo 2, što bi zapravo značilo da u čvor i *ulaze i izlaze tačno dvije grane koje pripadaju najkraćem putu*, što očigledno *nije moguće*. Međutim, takva dodatna ograničenja bila bi *suvišna*. Naime, uz pomoć elementarnih algebarskih manipulacija, nije teško uvidjeti da ta ograničenja automatski slijede kao *posljedica* ostalih prisutnih ograničenja, tako da ih nije potrebno eksplicitno zadavati.

Konačno, kako sve promjenljive zbog njihove logičke prirode mogu imati samo vrijednosti 0 ili 1, javljaju se i ograničenja $x_{i,j} \in \{0, 1\}$. Sada se zadatak nalaženja najkraćeg puta može zapisati u obliku

$$\arg\min S = \sum_{(i,j)\in R} l_{i,j}\, x_{i,j}$$

p.o.

$$-\sum_{(1,j)\in R} x_{1,j} = -1$$

$$\sum_{(i,k)\in R} x_{i,k} - \sum_{(k,j)\in R} x_{k,j} = 0,\ k = 2\,..\,n-1$$

$$\sum_{(i,n)\in R} x_{i,n} = 1$$

$$x_{i,j} \in \{0,1\},\quad (i,j) \in R$$

Prvo ograničenje određuje da *iz početne tačke grafa* mora izaći *samo jedna grana* koja će se nalaziti na najkraćem putu. Ovo ograničenje je pomnoženo sa –1 u odnosu na oblik ovog ograničenja koji smo nešto ranije naveli, zbog bolje simetrije dobijenih jednačina. Naime, uz takvu konvenciju, sve promjenljive koje odgovaraju granama koje *izlaze* iz nekog čvora imaće predznak "–" u odgovarajućoj jednačini za taj čvor, dok će promjenljive koje odgovaraju granama koje *ulaze* u čvor u odgovarajućoj jednačini imati predznak "+". Naredna skupina ograničenja analogne su *jednačinama očuvanja toka* kod problema minimalnog protoka. Naime, najkraći put može se posmatrati kao *jedinični tok* koji je potrebno "protjerati" od početnog do krajnjeg čvora *po minimalnoj cijeni* i taj tok ne može *niti nastati niti nestati* u bilo kojem čvoru koji nije prvi niti posljednji. U ovom konkretnom slučaju, to znači da za *sve ostale čvorove* (osim prvog i posljednjeg) *koji se nalaze na najkraćem putu jedna grana najkraćeg puta u njih ulazi i jedna grana najkraćeg puta iz njih izlazi*, odnosno da *u čvorove koji se ne nalaze na najkraćem putu nijedna grana najkraćeg puta u njih niti ulazi niti izlazi*. Konačno, posljednje ograničenje (ne računajući ograničenja na 0–1 prirodu promjenljivih) govori da u *krajnju tačku grafa* mora ući *tačno jedna grana koja pripada najkraćem putu*. Funkcija cilja kao rezultat daje *dužinu najkraćeg puta*, a promjenljive $x_{i,j}$ koje u optimalnom rješenju dobiju vrijednost 1, *određuju koje grane se nalaze na najkraćem* (*kritičnom*) *putu*.

U prethodnom modelu, problem predstavljaju ograničenja $x_{i,j} \in \{0,1\}$ zbog kojih ovaj model *nije model linearnog programiranja*, već *cjelobrojnog linearnog programiranja* (odnosno njegove podvrste poznate kao *logičko* ili 0–1 *programiranje*). Međutim, slično kao kod problema raspoređivanja, ukoliko ova ograničenja zamijenimo sa ograničenjima $x_{i,j} \geq 0$, dobijamo problem *linearnog programiranja* koji je, u nešto slobodnijem smislu, praktično *ekvivalentan* polaznom problemu. Tačnije, *bazna optimalna rješenja* novodobijenog problema poklapaju se sa *optimalnim rješenjima* polaznog problema tj. traženim *najkraćim putevima* (govorimo *u množini* jer najkraći put *ne mora uvijek biti jedinstven*). Zaista, nije teško pokazati da ograničenja $x_{i,j} \geq 0$ zajedno sa *ostalim ograničenjima* povlače da mora biti i $x_{i,j} \leq 1$, tako da sva dopustiva rješenja novodobijenog problema linearnog programiranja moraju biti između 0 i 1 uključivo. Međutim, kako je matrica ograničenja ovog problema *totalno unimodularna* (ona zapravo nije ništa drugo nego *matrica incidencije za pripadni graf*), sva *bazna optimalna rješenja* ovog problema su *cjelobrojna*. Ova cjelobrojnost, zajedno sa ograničenjima $0 \leq x_{i,j} \leq 1$, ima za posljedicu da *sva bazna rješenja novodobijenog problema linearnog programiranja automatski zadovoljavaju i uvjet* $x_{i,j} \in \{0,1\}$. Primijetimo da je u ovom slučaju, totalna unimodularnost matrice incidencije bila *ključni faktor* koji je omogućio da se u konačnici problem najkraćeg puta svede na problem linearnog programiranja.

Model linearnog programiranja koji odgovara problemu nalaženja najkraćeg puta može se u nešto kompaktnijem obliku zapisati i na sljedeći način:

$$\arg\min S = \sum_{(i,j)\in R} l_{i,j}\, x_{i,j}$$

p.o.

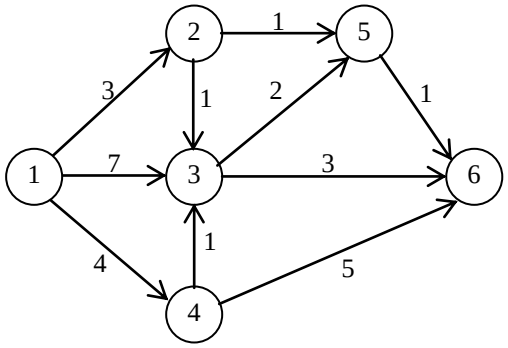
$$\sum_{(i,k)\in R} x_{i,k} - \sum_{(k,j)\in R} x_{k,j} = \begin{cases} -1, & \text{za } k = 1 \\ 0, & \text{za } k = 2\,..\,n-1 \\ 1, & \text{za } k = n \end{cases}$$

$$x_{i,j} \geq 0,\quad (i,j) \in R$$

Zaista, za $k = 1$, *prva suma* u prethodnim ograničenjima *jednaka je nuli* (jer nema grana koje ulaze u čvor 1), dok je za $k = n$ *druga suma jednaka nuli* (jer nema grana koje izlaze iz čvora n).

Ovaj model ostaje na snazi čak i ukoliko *ima grana koje ulaze u čvor* 1 *odnosno koje izlaze iz čvora n*. Zaista, takve grane *svakako neće ležati na najkraćem putu*, pa će prva suma u prethodnim ograničenjima za $k = 1$ odnosno druga suma za $k = n$ i tada biti jednaka nuli.

- ***Primjer*** : Za usmjereni graf na slici formirati matematski model linearnog programiranja koji rješava problem nalaženja najkraćeg puta od početnog čvora 1 do krajnjeg čvora 6.



U skladu sa prethodno provedenim razmatranjima, dobijamo model linearnog programiranja u sljedećem obliku:

$$\arg\min S = 3x_{1,2} + 7x_{1,3} + 4x_{1,4} + x_{2,3} + x_{2,5} + 2x_{3,5} + 3x_{3,6} + x_{4,3} + 5x_{4,6} + x_{5,6}$$

p.o.

$$-x_{1,2} - x_{1,3} - x_{1,4} = -1$$
$$x_{1,2} - x_{2,3} - x_{2,5} = 0$$
$$x_{1,3} + x_{2,3} + x_{4,3} - x_{3,5} - x_{3,6} = 0$$
$$x_{1,4} - x_{4,3} - x_{4,6} = 0$$
$$x_{2,5} + x_{3,5} - x_{5,6} = 0$$
$$x_{3,6} + x_{4,6} + x_{5,6} = 1$$
$$x_{1,2}, x_{1,3}, x_{1,4}, x_{2,3}, x_{2,5}, x_{3,5}, x_{3,6}, x_{4,3}, x_{4,6}, x_{5,6} \geq 0$$

Mada postoje mnogo efikasniji načini za rješavanje problema najkraćeg puta, ovako dobijeni model se može riješiti i koristeći opće metode za rješavanje linearnog programiranja kao što je *simpleks algoritam*, što može imati smisla ako imamo na raspolaganju neki univerzalni softver za rješavanje općih problema linearnog programiranja koji ne poznaje specijalističke metode za rješavanje problema najkraćeg puta. Međutim, ukoliko želimo riješiti problem najkraćeg puta na taj način, moramo biti sigurni da je softver koji koristimo *zasnovan na simpleks algoritmu* ili nekom drugom algoritmu koji *pretražuje isključivo bazna dopustiva rješenja*. Naime, u slučaju da najkraći put *nije jedinstven*, tada ekvivalentni model linearnog programiranja može imati i nebaznih optimalnih rješenja koja *nisu cjelobrojna*, a koja očito *ne predstavljaju* rješenje polaznog problema najkraćeg puta. Nažalost, algoritmi za rješavanje linearnog programiranja koji se zasnivaju na *kretanje kroz unutrašnjost dopustive oblasti* (metodi *unutrašnje tačke*) a ne na kretanju kroz *vršne tačke dopustive oblasti*, imaju tendenciju da u takvim slučajevima upravo pronađu neko *nebazno optimalno rješenje* koje *nije cjelobrojno*, Jasno je da takvo rješenje *nije ni od kakve koristi* za razmatrani problem najkraćeg puta.

Razmotrimo još i *dualni problem* za problem nalaženja najkraćeg puta. Dualni problem za ekvivalentni model linearnog programiranja koji se dobije iz problema najkraćeg puta je *vrlo jednostavne strukture* i glasi

$$\arg\max W = y_n - y_1$$

p.o.

$$y_j - y_i \leq l_{i,j}, \quad (i, j) \in \mathrm{R}$$

Jednostavna struktura dualnog problema omogućava razvoj *vrlo efikasnih dualnih algoritama* (tipično *primalno-dualnog tipa*) za rješavanje problema najkraćeg puta. Optimalne vrijednosti dualnih promjenljivih *nikada nisu jednoznačno određene*, s obzirom da funkcija cilja *od većine njih uopće i ne zavisi* (zavisi samo od y_1 i y_n, tačnije *samo od njihove razlike*). Ukoliko uzmemo da je $y_1 = 0$, vrijednosti y_i za čvorove *duž najkraćeg puta* predstavljaju *najkraće udaljenosti* tih čvorova od početnog čvora, dok za čvorove *koji ne leže duž najkraćeg puta* vrijednosti odgovarajućih dualnih promjenljivih nemaju u općem slučaju neko

smisleno značenje (mada ćemo kasnije vidjeti da se ovim promjenljivim može dati značenje ukoliko se one posmatraju u kontekstu problema tzv. *mrežnog planiranja*).

Na kraju, istaknimo još da se problem najkraćeg puta može posmatrati i kao *specijalan slučaj problema raspoređivanja*. Zaista, uzmimo $n-1$ *izvršilaca* I_1, I_2, ..., I_{n-1} koje ćemo postaviti respektivno u čvorove 1, 2, ..., $n-1$ i $n-1$ *mjesta izvršavanja* M_1, M_2, ..., M_{n-1} koje ćemo postaviti respektivno u čvorove 2, 3, ..., n (dakle, mjesto M_j u čvor $j+1$). "Cijene" $c_{i,j}$ raspoređivanja i-tog izvršioca na j-to mjesto definiraćemo kao dužinu grane koja "spaja" i-tog izvršioca i j-to mjesto, odnosno uzećemo da je $c_{i,j} = l_{i,j+1}$. Nije teško pokazati da optimalno rješenje ovog problema raspoređivanja ujedno daje i rješenje problema najkraćeg puta, odnosno grana $(i, j+1)$ pripada najkraćem putu ako i samo ako čvor i leži na najkraćem putu i ako je u optimalnom rješenju odgovarajućeg problema raspoređivanja $x_{i,j} = 1$. Slijedi da se traženi najkraći put lako nalazi prateći nenulte vrijednosti $x_{i,j}$ počev od $i = 1$.

Općenito o algoritmima za nalaženje nakraćeg puta u grafu

Kao što je već rečeno, *problem najkraćeg puta* je vjerovatno jedan od *najfundamentalnijih zadataka optimizacije na grafovima*, ne samo zbog svoje *velike praktične primjene*, nego i zbog činjenice da se on često javlja kao *podzadatak* u mnogim drugim problemima grafovske optimizacije. Stoga ne čudi da je za rješavanje ovog problema razvijeno mnogo različitih specijalističkih algoritama.

Postoji više različitih metoda za rješavanje zadataka najkraćeg puta koji se koriste u praksi. Pri tome je karakteristično da perfomanse pojedinih metoda *zavise od strukture grafa* (prvenstveno od toga da li je graf *gust ili ne*, ali i od nekih drugih činjenica), tako da *ne postoji metod koji bi bio bolji od svih drugih metoda za sve vrste grafova*. Drugim riječima, izbor metoda kojim će se rješavati problem najkraćeg puta treba *prilagoditi grafu* na kojem se problem rješava.

Jedan od najstarijih i najjednostavnijih metoda za nalaženje najkraćeg puta u grafu je ***Bellmanov algoritam***. Ovaj algoritam je nažalost *veoma ograničen*, jer se može primijeniti *isključivo na usmjerene grafove koji ne sadrže cikluse*. Međutim, u određenim oblastima primjene *javljaju se upravo takvi grafovi* i u tom slučaju Bellmanov algoritam je vjerovatno najbolji izbor. Naime, ovaj algoritam je *vrlo jednostavan za implementaciju* i *prilično brz*, s obzirom da mu je vrijeme izvršavanja proporcionalno broju grana m. Za slučaj *rijetkih grafova*, kod kojih je m istog reda veličine kao i broj čvorova n, ovo je vjerovatno *ubjedljiivo najbrži algoritam* za nalaženje najkraćeg puta (naravno, pod uvjetom da je graf takav da se na njega uopće može primijeniti ovaj algoritam), dok za slučaj *gustih grafova* kod kojih je m reda veličine kao n^2 postoje i drugi algoritmi *uporedive brzine*, koji su ponekad čak i *brži* od njega. Treba još istaći da se vrijeme izvršavanja proporcionalno sa m postiže pod uvjetom da je graf *predstavljen listom susjedstva*. U slučaju reprezentacije pomoću *matrice susjedstva* vrijeme izvršavanja biće proporcionalno sa n^2 kao i kod mnogih drugih algoritama za nalaženje najkraćeg puta, tako da tada prednosti ovog algoritma *ne dolaze do izražaja*. Međutim, već smo vidjeli da matrice susjedstva *svakako nisu dobar izbor za reprezentaciju rijetkih grafova*.

Bellmanov algoritam je kasnije upotpunio *L. R. Ford* i tako je nastao ***Bellman-Fordov algoritam*** koji uklanja ograničenje na usmjerene grafove bez ciklusa. Drugim riječima, ovaj algoritam je primjenljiv *na sve vrste grafova*, ali je u najgorem slučaju i do *n puta sporiji* od Bellmanovog algoritma. Drugim riječima, vrijeme izvršavanja mu je u najgorem slučaju reda mn ukoliko je graf predstavljen listom susjedstva, odnosno reda n^3 ukoliko je graf predstavljen matricom susjedstva. Ovaj algoritam se može izraziti *u dvije verzije*, približno *istih perfomansi*. ***Bellmanova verzija*** neposrednije prati ideje izvornog Bellmanovog algoritma, dok je ***Fordova verzija*** nešto praktičnija za izvedbu (pogotovo pri ručnom radu).

Bellman-Fordov algoritam je vjerovatno najviše korišteni algoritam za nalaženje najkraćeg puta u grafovima kod kojih težine grana mogu biti i *negativni brojevi*. Naime, svi drugi poznati algoritmi za nalaženje najkraćeg puta koji dopuštaju i negativne težine grana imaju manje-više *slične perfomanse* kao i Bellman-Fordov algoritam, a komplikovaniji su od njega. Međutim, u praksi ipak dominiraju primjene u kojima *težine grana mogu biti samo nenegativni brojevi*. U takvim slučajevima, algoritam poznat kao ***Dijkstrin algoritam*** postiže *mnogo bolje perfomanse*. Ovaj algoritam nastao je kombiniranjem ideja iz Bellmanovog algoritma sa danas zaboravljenim ***Dantzigovim algoritmom***, koji se više ne koristi, jer je *u svim slučajevima znatno sporiji od Dijkstrinog algoritma* (ponekad i do n puta), a posjeduje *ista ograničenja* na nenegativnost težina grana kao i Dijkstrin algoritam.

Dijkstrin algoritam dolazi u *nekoliko verzija* koje se zasnivaju na *istom principu*, ali se razlikuju u *pojedinim implementacionim detaljima*. Osnovna izvedba ima vrijeme izvršavanja približno proporcionalno sa n^2, neovisno od toga da li je graf reprezentiran pomoću matrice ili liste susjedstva (tačnije, za *rijetke*

grafove moguće je korištenjem listi susjedstva skratiti trajanje izvjesnih operacija, ali u izrazu za ukupan broj izvršenih operacija i dalje ostaje član proporcionalan sa n^2). Za guste grafove, ovo je *odlično vrijeme izvršavanja* za koje nije poznato da se može više skratiti. Međutim, u slučaju *rijetkih grafova*, moguće je postići i *bolja vremena izvršavanja*. Ove poboljšane izvedbe zahtijevaju upotrebu *specijalnih struktura podataka* koje omogućavaju *ubrzanu izvedbu nekih algoritamskih koraka* (ali, nažalost, *usporavaju neke druge korake*, tako da je neophodno *naći neki kompromis*). Te strukture podataka su razne izvedbe tzv. *redova sa prioritetom* koji omogućavaju *efikasno pronalaženje najmanjeg podatka* u skupini podataka i *efikasno ažuriranje* nakon njegovog uklanjanja. Ukoliko se kao red sa prioritetom upotrijebi struktura podataka poznata pod nazivom ***hrpa*** ili ***gomila*** (engl. *heap*) i iskoristi reprezentacija grafa pomoću liste susjedstva, vrijeme izvršavanja je moguće svesti na iznos proporcionalan sa $(m+n)\log_2 n$ u najgorem slučaju. Za guste grafove ovo je *lošije* u odnosu na osnovnu izvedbu (s obzirom da je kod njih m reda n^2), ali za slučaj *rijetkih grafova* (kod kojih je m reda n) ovo je tipično *mnogo bolje* od n^2 kod osnovne izvedbe. Upotrebom još naprednijih varijanti redova sa prioritetom, kao što su tzv. ***Fibonaccijeve hrpe*** (engl. *Fibonacci heaps*) vrijeme izvršavanja moguće je još spustiti na iznos proporcionalan sa $m + n\log_2 n$. U skladu s tim, implementacije Dijkstrinog algoritma obično se dijele na *elementarne*, koje ne koriste redove sa prioritetom, i *napredne* koje se zasnivaju na raznim vrstama redova sa prioritetom.

Danas je *kompleksnost mnogih vrsta mreža* koje se susreću u praksi (cestovnih, električnih ili raznih tipova mreža u oblasti informacijskih i komunikacionih tehnologija) izuzetno porasla i zahtijeva sve *bolje metode za nalaženje najkraćih puteva* u njima. S tim ciljem koristi se sve veći broj *različitih algoritama* i njihovih *poboljšanja*. S obzirom da postoji zaista veliki broj različitih algoritama za rješavanje problema najkraćeg puta, čije perfomanse variraju od slučaja do slučaja, nije čudno da empirijska istraživanja performansi ovih algoritama *nisu dala jednoznačan odgovor* koji algoritam najbrže rješava zadatak nalaženja najkraćeg puta za zadatke koji odgovaraju *stvarnim situacijama* u praksi. Kako praktični problemi u kojima se javlja zadatak najkraćeg puta obično daju *rijetke grafove*, velika pažnja se posvećuje upravo razvoju efikasnih algoritama na rijetkim grafovima. Posebnu pažnju su pobudile implementacije Dijkstrinog algoritma koje kao red sa prioritetom koriste strukturu podataka za čije ime u literaturi na južnoslovenskim jezicima ne postoji općeprihvaćen prevod, a na engleskom jeziku se naziva ***bucket*** (u bukvalnom prevodu, ovo je *kanta za vodu*). Ova struktura podataka je, u suštini, neka vrsta *dvostruko povezane kružne liste*. Tako se u posljednje vrijeme smatra da sljedeća tri algoritma najbrže rješavaju zadatke koji odgovaraju stvarnim situacijama u praksi:

- Algoritam ***rasta grafa*** (engl. *graph growth*) implementiran pomoću dva klasična *reda čekanja* (engl. *queue*), tj. pomoću dvije FIFO (First In First Out) liste;
- Dijkstrin algoritam implementiran koristeći *bucket*-e kao redove sa prioritetom;
- Dijkstrin algoritam koji koristi *dvostruke bucket-e* (engl. *double buckets*) kao redove sa prioritetom (tj. *bucket*-e čiji su elementi ponovo *bucket*-i).

Najbolje performanse za rješavanje problema najkraćeg puta tipa "jedan do svih" postižu se uglavnom korištenjem algoritma rasta grafa implementiranog sa dvije liste. S druge strane, ukoliko je cilj rješavanje problema najkraćeg puta tipa "jedan do jednog" ili "jedan do nekih", tada se obično bolje perfomanse mogu postići nekom od implementacija Dijkstrinog algoritma, s obzirom da se izvođenje ovog algoritma može prekinuti čim se otkrije najkraća udaljenost do odredišnog čvora (ili zadane skupine čvorova), što se može desiti i prije prirodnog završetka algoritma. Kod nekih implementacija perfomanse mogu zavisiti ne samo od broja grana nego i od *maksimalne dužine grana*. Tako se za slučaj grafova čije težine grana *nisu prevelike* (po *Zhanu* i *Noonu* granica je oko 1500) za dobijanje najboljih perfomansi preporučuje upotreba Dijkstrinog algoritma koji koristi *bucket*-e, dok se za slučaj većih težina grana preporučuje implementacija koja koristi *dvostruke bucket-e*.

Treba istaći i da algoritmi za nalaženje najkraćeg puta predstavljaju *ključni dio protokola usmjeravanja* (*rutiranja*) u računarskim mrežama. Svi ovi protokoli računaju *najkraći raspoloživi put* do odredišta da bi utvrdili kojim putem treba usmjeravati pakete podataka. Tako je npr. *Dijkstra algoritam* posebno pogodan za realizaciju tzv. ***OSPF*** (Open Shortest Path First) ***protokola*** unutar IP mreža. Nedostatak Dijkstrinog algoritma u distribuiranim sistemima kakve su računarske mreže je u činjenici da *svako čvorište mreže u kojem se implementira protokl rutiranja* (npr. svaki ruter u mreži) *mora znati stanje kompletne mreže* (odnosno, informacije o *svim drugim čvorištima i granama mreže*). S druge strane, kod Bellman-Fordovog algoritma dovoljno je da *svako čvorište mreže u kojem se implementira protokol rutiranja poznaje samo informacije o njemu incidentnim granama i njemu susjednim čvorištima*. Ovo nekada može biti prednost koja ide u prilog Bellman-Fordovom algoritmu, bez obzira na činjenicu da on može biti i do n puta *sporiji* od Dijkstrinog algoritma. Stoga se jednostavniji protokoli rutiranja kao što je ***RIP*** (Routing Information

Protocol) ***protokol***, koji se koriste u manjim mrežama i sa jednostavnijim ruterima, zasnivaju upravo na Bellman-Fordovom algoritmu.

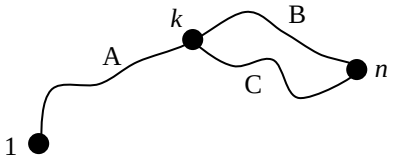
Bellmanov algoritam

Opis algoritama za nalaženje najkraćeg puta u grafovima započećemo opisom *Bellmanovog algoritma*, čije su osnovne ideje iskorištene u mnogim drugim algoritmima (uključujući *Bellman-Fordov i Dijkstrin algoritam*). Kao što je već rečeno, ovaj algoritam se može primijeniti *samo na usmjerene grafove bez ciklusa*. Pored toga, ovaj algoritam zahtijeva da graf bude ***pravilno numeriran*** (kaže se još i ***topološki sortiran***). Za usmjereni graf koji ne sadrži cikluse kaže se da je *pravilno numeriran* ukoliko *za svaku granu grafa vrijedi da je redni broj njenog početnog čvora uvijek manji od rednog broja njenog završnog čvora*, tj. ukoliko za svaku granu $(i, j) \in \mathrm{R}$ vrijedi $i < j$. Lako se uviđa da je pravilna numeracija grafa izvodljiva *ako i samo ako je graf usmjeren i ne sadrži cikluse*. U slučaju da imamo usmjereni graf bez ciklusa koji *nije pravilno numeriran*, za primjenu Bellmanovog algoritma potrebno je prethodno izvršiti ***renumeraciju*** čvorova grafa, o čemu ćemo govoriti nešto kasnije. Sretna okolnost je da se u praksi Bellmanov algoritam uglavnom primjenjuje na grafove koji su takvi po prirodi da su već *sami po sebi pravilno numerirani*.

Bellmanov algoritam se zasniva na primjeni jednog poznatog principa koji je formulirao također Bellman, a poznat je kao ***Bellmanov princip optimalnosti***. Ovaj princip je *veoma općenit* i leži u osnovi jedne vrlo široke porodice metoda za rješavanje raznih optimizacionih problema, koje se zajedničkim imenom nazivaju ***dinamičko programiranje***, a o kojima ćemo kasnije detaljnije govoriti. U suštini, Bellmanov algoritam nije ništa drugo nego *primjena Bellmanovog principa optimalnosti na problem najkraćeg puta*. Bellmanov princip optimalnosti, u svojoj najopćenitijoj formi, može se iskazati ovako:

Neka je potrebno izvršiti optimizaciju nekog postupka koji se odvija u n etapa, polazeći od nekog početnog stanja S_1 do nekog krajnjeg stanja S_n. Dalje, neka je poznato da optimalno stanje nakon k koraka glasi S_k. Tada optimalna strategija koja vodi iz stanja S_k u stanje S_n nije ništa drugo nego dio optimalne strategije koja vodi iz stanja S_1 u stanje S_n.

Primijenjeno na problem najkraćeg puta, Bellmanov princip možemo iskazati ovako. Neka je potrebno naći najkraći put iz čvora 1 do čvora n i neka nam je na neki način poznato da taj najkraći put vodi kroz neki čvor k. Tada *najkraći put od čvora k do čvora n mora biti dio najkraćeg puta od čvora* 1 *do čvora n*. Recimo, na sljedećoj slici pretpostavimo da najkraći put od čvora 1 do čvora n vodi prvo od čvora 1 do nekog čvora k putanjom A a zatim od čvora k do čvora n putanjom B. Tada putanja B mora ujedno biti i najkraći put od čvora k do čvora n. Zaista, ukoliko bi postojala neka *kraća putanja* od čvora k do čvora n, recimo putanja C, tada bi unija putanja A i C bila kraća od unije putanja A i B, tako da unija putanja A i B ne bi bila najkraći put od čvora 1 do čvora n, što protivrječi pretpostavci.



Pretpostavimo sada da imamo neki graf sa n čvorova, pri čemu smo sa $l_{i,j}$ označili dužinu grane (i, j). Neka je f_i dužina "*lokalnog*" *najkraćeg puta od čvora i do čvora n*. Jasno je da je tada $f_n = 0$, dok za vrijednosti $i < n$ na osnovu principa optimalnosti možemo zaključiti da mora vrijediti jednačina, koja se naziva ***Bellmanova jednačina za najkraći put unazad***:

$$f_i = \min \{ f_j + l_{i,j} \mid i < j \le n \wedge (i, j) \in \mathrm{R} \}$$

Zaista, najkraći put iz čvora i u čvor n mora voditi kroz *neki njemu susjedni čvor j*, pri čemu zbog pravilne numeracije grafa vrijedi $i < j$. Kada bismo *tačno znali* koji je to čvor j, dužina najkraćeg puta od čvora i do čvora n bila bi, prema principu optimalnosti, tačno $f_j + l_{i,j}$ (dužina najkraćeg puta od j do n uvećana za dužinu puta od i do j, koji je samo jedna grana). Međutim, kako mi *ne znamo* koji je to čvor j, možemo ispitati *sve susjede* čvora i, i onaj koji daje najkraći ukupni put *mora biti onaj pravi*.

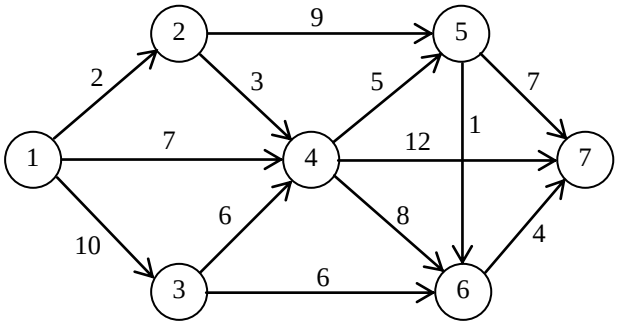
Činjenica da nam za računanje vrijednosti f_i trebaju samo vrijednosti f_j za $j > i$ (što je posljedica pravilne numeracije grafa) omogućava nam da postupno izračunamo sve vrijednosti f_i *unazad* (dakle, za i od n do 1 naniže) koristeći relacije

$$f_n = 0$$

$$f_i = \min \{ f_j + l_{i,j} \mid i < j \le n \wedge (i,j) \in \mathrm{R} \}, \quad i = n-1, n-2, ..., 1$$

Drugim riječima, lako se izračunavaju *dužine najkraćih puteva svih čvorova grafa do čvora n*, odnosno na ovaj način se rješava *problem najkraćeg puta sa zadanim odredištem* (problem tipa "svi do jednog"). Sami najkraći putevi se lako očitavaju ako pratimo za koju vrijednost *j* je postignut minimum pri računanju vrijednosti f_i.

- ***Primjer***: Koristeći Bellmanov algoritam, naći najkraći put između čvorova 1 i 7 u grafu prikazanom na sljedećoj slici:



Lako se vidi da je graf *pravilno numeriran*, tako da su *ispunjeni uvjeti* za primjenu Bellmanovog algoritma. Težinska matrica za ovaj graf glasi:

$$\mathbf{W} = \begin{pmatrix} 0 & 2 & 10 & 7 & \infty & \infty & \infty \\ \infty & 0 & \infty & 3 & 9 & \infty & \infty \\ \infty & \infty & 0 & 6 & \infty & 6 & \infty \\ \infty & \infty & \infty & 0 & 5 & 8 & 12 \\ \infty & \infty & \infty & \infty & 0 & 1 & 7 \\ \infty & \infty & \infty & \infty & \infty & 0 & 4 \\ \infty & \infty & \infty & \infty & \infty & \infty & 0 \end{pmatrix}$$

Na dijagonalu su upisane nule, zbog toga što je korisno smatrati da je svaki čvor *na udaljenosti nula od sebe* samog. Također, zbog pravilne numeracije grafa, ova matrica ima elemente različite od ∞ *samo iznad glavne dijagonale*. Koristeći Bellmanove jednačine za najkraći put unazad, dobijamo:

$$f_7 = 0$$
$$f_6 = \min \{ f_7 + l_{6,7} \} = 0 + 4 = 4 \qquad (\text{za } j = 7)$$
$$f_5 = \min \{ f_6 + l_{5,6}, f_7 + l_{5,7} \} = \min \{ 4 + 1, 0 + 7 \} = 5 \qquad (\text{za } j = 6)$$
$$f_4 = \min \{ f_5 + l_{4,5}, f_6 + l_{4,6}, f_7 + l_{4,7} \} = \min \{ 5 + 5, 4 + 8, 0 + 12 \} = 10 \qquad (\text{za } j = 5)$$
$$f_3 = \min \{ f_4 + l_{3,4}, f_6 + l_{3,6} \} = \min \{ 10 + 6, 4 + 6 \} = 10 \qquad (\text{za } j = 6)$$
$$f_2 = \min \{ f_4 + l_{2,4}, f_5 + l_{2,5} \} = \min \{ 10 + 3, 5 + 9 \} = 13 \qquad (\text{za } j = 4)$$
$$f_1 = \min \{ f_2 + l_{1,2}, f_3 + l_{1,3}, f_4 + l_{1,4} \} = \min \{ 13 + 2, 10 + 10, 10 + 7 \} = 15 \qquad (\text{za } j = 2)$$

Odavde direktno vidimo da dužina traženog najkraćeg puta iznosi 15, dok sam najkraći put glasi $1 \to 2 \to 4 \to 5 \to 6 \to 7$ (minimum pri računanju f_1 je postignut za $j = 2$, minimum pri računanju f_2 postignut je za $j = 4$, itd.).

Pri ručnom radu se, radi preglednosti, često formira tablica pomoćnih vrijednosti $f_{i,j} = f_j + l_{i,j}$, tako da se onda f_i određuje kao najmanja od svih izračunatih vrijednosti $f_{i,j}$ za sve vrijednosti $j > i$. Ovakav primjer računanja prikazan je u nastavku (polja u kojima se nalaze vrijednosti f_i prikazana su *osjenčeno*). U *dnu tablice* upisuju se izračunate vrijednosti f_j dobijene u postupku računanja, što čini preglednijim kasnija izračunavanja veličina $f_{i,j}$. Primijetimo da se ova tablica popunjava *odozdo nagore* (tj. od većih ka manjim vrijednostima *i*):

	1	2	3	4	5	6	7
1		15	20	17			
2				13	14		
3				16		10	
4					10	12	12
5						5	7
6							4
f_j	15	13	10	10	5	4	0

Princip optimalnosti se može primijeniti i *u suprotnom smjeru*, gledajući *od početnog čvora prema kraju*. Ukoliko sa f_j označimo *dužinu najkraćeg puta od čvora* 1 *do čvora j*, primjena principa optimalnosti dovodi do sljedeće jednačine, koja se naziva ***Bellmanova jednačina za najkraći put unaprijed***:

$$f_j = \min \{ f_i + l_{i,j} \mid 1 \le i < j \wedge (i, j) \in \mathrm{R} \}$$

Zaista, najkraći put iz čvora 1 u čvor *j* mora voditi kroz *neki čvor i kojem je čvor j susjedni čvor*, pri čemu zbog pravilne numeracije grafa vrijedi $i < j$. Kada bismo *tačno znali* koji je to čvor *i*, dužina najkraćeg puta od čvora 1 do čvora *j* bila bi, prema principu optimalnosti, tačno $f_i + l_{i,j}$ (dužina najkraćeg puta od 1 do *i* uvećana za dužinu puta od *i* do *j*, koji je samo jedna grana). Međutim, kako mi *ne znamo* koji je to čvor *i*, možemo ispitati *sve čvorove kojima je čvor j susjed*, i onaj koji daje najkraći ukupni put *mora biti onaj pravi*.

Zahvaljujući pravilnoj numeraciji grafa, za računanje vrijednosti f_j trebaju samo vrijednosti f_i za $i > j$, što nam omogućava da postupno izračunamo sve vrijednosti f_j *unaprijed* koristeći relacije

$$f_1 = 0$$

$$f_j = \min \{ f_i + l_{i,j} \mid 1 \le i < j \wedge (i, j) \in \mathrm{R} \}, \quad j = 2 \,..\, n$$

Na primjer, za graf iz prethodnog primjera, ovo računanje bi teklo ovako:

$f_1 = 0$
$f_2 = \min \{ f_1 + l_{1,2} \} = 0 + 2 = 2$ (za $i = 1$)
$f_3 = \min \{ f_1 + l_{1,3} \} = 0 + 10 = 10$ (za $i = 1$)
$f_4 = \min \{ f_1 + l_{1,4}, f_2 + l_{2,4}, f_3 + l_{3,4} \} = \min \{0 + 7, 2 + 3, 10 + 6\} = 5$ (za $i = 2$)
$f_5 = \min \{ f_2 + l_{2,5}, f_4 + l_{4,5} \} = \min \{2 + 9, 5 + 5\} = 10$ (za $i = 4$)
$f_6 = \min \{ f_3 + l_{3,6}, f_4 + l_{4,6}, f_5 + l_{5,6} \} = \min \{10 + 6, 5 + 8, 10 + 1\} = 11$ (za $i = 5$)
$f_7 = \min \{ f_5 + l_{5,7}, f_6 + l_{6,7} \} = \min \{10 + 7, 11 + 4\} = 15$ (za $i = 6$)

Sada traženi najkraći put dobijamo razmotavanjem *unazad* (u čvor 7 smo stigli iz čvora 6, u čvor 6 iz čvora 5, u čvor 5 iz čvora 4, u čvor 4 iz čvora 2, te u čvor 2 iz čvora 1).

Određivanje najkraćeg puta *unaprijed* (tj. od početnog čvora) pomoću Bellmanovog algoritma također je moguće preglednije predstaviti *u formi tabele*, samo što se preračunavanja sada vrše *po kolonama* umjesto po redovima, *slijeva nadesno*:

	2	3	4	5	6	7	f_i
1	2	10	7				0
2			5	11			2
3			16		16		10
4				10	13		5
5					11	17	10
6						15	11
7							15

Određivanje najkraćeg puta pomoću Bellmanovog algoritma *unaprijed* očigledno rješava *problem najkraćeg puta sa jedinstvenim polazištem* (odnosno problem tipa "jedan do svih"). Mada određivanje najkraćeg puta *unaprijed* djeluje prirodnije, Bellmanov algoritam se u praksi *gotovo uvijek izvodi unazad*. Razlog za to leži u činjenici da reprezentacija grafa pomoću liste susjedstva (koja se uglavnom koristi kod grafova na koje se primjenjuje ovaj algoritam) *nije pogodna za izvođenje Bellmanovog algoritma unaprijed*.

Naime, za potrebe izvođenja Bellmanovog algoritma unaprijed u svakom koraku nam je potreban *spisak svih čvorova kojima je tekući čvor susjed*, a ta se informacija *ne može dobiti efikasno* iz liste susjedstva (za efikasno dobijanje te informacije trebala bi nam neka vrsta *inverzne liste susjedstva* kod koje bi se za svaki čvor čuvao ne skup njegovih susjeda, nego skup svih čvorova kojima je on susjed, što se efektivno svodi na listu susjedstva grafa čije su sve grane *suprotno usmjerene* u odnosu na grane razmatranog grafa).

Ukoliko nam je pri izvođenju algoritma unazad potreban najkraći put do nekog drugog čvora a ne do čvora n, na prvi pogled se čini da imamo problem, jer zbog zahtjeva na pravilnu numeraciju grafa ne možemo proizvoljno izabrati koji ćemo čvor označiti sa n. Međutim, nije nikakav problem da se algoritam starta *od proizvoljnog čvora do kojeg želimo naći najkraći put*, a ne nužno od čvora n. Zaista, zbog pravilne numeracije grafa, *niti jedan put* (pa *samim tim ni najkraći*) od čvora 1 do ma kojeg čvora i sigurno *ne može prolaziti* kroz čvorove sa indeksima većim od i, tako da ti čvorovi jednostavno *neće biti* "*u igri*". Slično vrijedi i ukoliko nam pri izvršavanju algoritma unaprijed treba najkraći put od nekog drugog čvora, a ne čvora označenog sa 1.

Kao što je već rečeno, osnovna prednost Bellmanovog algoritma u odnosu na druge je njegova *jednostavnost i brzina*. Zaista, očigledno je da je vrijeme izvođenja ovog algoritma *proporcionalno broju grana m*, jer se *svaka grana grafa obrađuje tačno jedanput* (ovo naravno vrijedi jedino pod uvjetom da su informacije o granama grafa *brzo dostupne*, što je slučaj ako je graf predstavljen listom susjedstva). Međutim, ključni nedostaci algoritma su ograničenje isključivo na usmjerene grafove bez ciklusa i potreba za renumeracijom čvorova u slučaju da graf nije pravilno numeriran.

Bellman-Fordov algoritam

Bellman-Fordov algoritam predstavlja *prirodno proširenje* Bellmanovog algoritma na *sve vrste grafova*, tj. na grafove koji nisu nužno niti usmjereni, niti bez ciklusa, niti pravilno numerirani. Ključna ideja ovog algoritma je činjenica da Bellmanov princip optimalnosti primijenjen na problem najkraćeg puta *mora vrijediti za sve grafove*, samo što u slučaju kada nemamo pravilnu numeraciju grafa (koja nije ni moguća ukoliko graf nije usmjereni graf bez ciklusa) nemamo više olakšavajuću okolnost da mora vrijediti $i < j$. Drugim riječima, za proizvoljne grafove moraju vrijediti *Bellmanove jednačine za najkraći put unazad* oblika

$$f_i = \min \{ f_j + l_{i,j} \mid 1 \le j \le n \wedge (i,j) \in \mathrm{R} \}, \;\; i = 1 \,..\, n$$

uz polaznu pretpostavku $f_n = 0$, odnosno *Bellmanove jednačine za najkraći put unaprijed* oblika

$$f_j = \min \{ f_i + l_{i,j} \mid 1 \le i \le n, (i,j) \in \mathrm{R} \}, \;\; j = 1 \,..\, n$$

uz polaznu pretpostavku $f_1 = 0$. Međutim, problem je što se zbog nepostojanja ograničenja $i < j$ ove jednačine *ne mogu poredati u takav poredak da je za izračunavanje ma koje od vrijednosti* f_i (ili f_j) potrebno poznavati *samo prethodno izračunate vrijednosti*, kao što smo mogli kod Bellmanovog algoritma. Stoga je zapravo osnovni problem *kako riješiti ove jednačine*.

Bellman-Fordov algoritam za rješavanje ovih jednačina koristi jednu poznatu tehniku iz opće teorije jednačina poznatu pod nazivom ***prosta iteracija***. Naime, neka imamo neku opću jednačinu oblika $\mathbf{x} = \mathbf{g}(\mathbf{x})$ gdje je $\mathbf{x}$ neka nepoznata veličina, koja u općem slučaju može biti *vektorska* (tj. višedimenzionalna) veličina, a $\mathbf{g}$ neka zadana (vektorska) funkcija argumenta $\mathbf{x}$. Neka smo, počev od neke fiksne vrijednosti $\mathbf{x}^{(0)}$ generirali niz vrijednosti $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$, $\mathbf{x}^{(3)}$ itd. tako da je $\mathbf{x}^{(1)} = \mathbf{g}(\mathbf{x}^{(0)})$, $\mathbf{x}^{(2)} = \mathbf{g}(\mathbf{x}^{(1)})$ i, općenito, $\mathbf{x}^{(k)} = \mathbf{g}(\mathbf{x}^{(k-1)})$ za $k = 1, 2, 3, \ldots$ Kaže se da je vrijednost $\mathbf{x}^{(k)}$ dobijena u *k-toj iteraciji* polazne jednačine. Ukoliko se za neko p dogodi da imamo da je $\mathbf{x}^{(p)} = \mathbf{x}^{(p-1)}$, jasno je da je onda takvo $\mathbf{x}^{(p)}$ rješenje jednačine $\mathbf{x} = \mathbf{g}(\mathbf{x})$. Naravno, u općem slučaju nemamo *nikakve garancije* da će se *tako nešto ikada dogoditi*, međutim *ukoliko se dogodi, našli smo jedno rješenje postavljene jednačine*. U općoj teoriji jednačina, daju se razni potrebni i dovoljni uvjeti pod kojim ovaj postupak vodi ka cilju, u šta se ovdje nećemo upuštati.

Kao što je gore rečeno, Bellman-Fordov algoritam zapravo rješava Bellmanove jednačine koristeći prostu iteraciju. Pokazaćemo prvo *Bellmaovu verziju* ovog algoritma, jer se ona neposrednije oslanja na prethodno opisane ideje. Neka $f_i^{(k)}$ predstavlja pretpostavljenu vrijednost za f_i dobijenu u k-toj iteraciji. Bellman-Fordov algoritam se također može izvoditi *unazad* ili *unaprijed*. Pri izvođenju *unazad*, inicijalno se postavlja $f_n^{(0)} = 0$ i $f_i^{(0)} = \infty$ za $i = 1 \,..\, n-1$ nakon čega se kroz iteracije (dakle, redom za $k = 1, 2, \ldots$) računaju vrijednosti

$$f_i^{(k)} = \min \{ f_j^{(k-1)} + l_{i,j} \mid 1 \le j \le n \wedge (i,j) \in \mathrm{R} \}, \;\; i = n, n-1, \ldots, 1$$

Pri tome se uzima da je $l_{i,i} = 0$, odnosno svaki čvor je *na udaljenosti nula od sebe samog*. Iteracije se ponavljaju sve dok u nekoj iteraciji sve izračunate vrijednosti $f_i^{(k)}$ ne ostanu iste kao u prethodnoj iteraciji, tj. sve dok ne bude $f_i^{(k)} = f_i^{(k-1)}$ za sve $i = 1 \,..\, n-1$. Bellman i Ford su dokazali da ovaj algoritam sigurno završava u *konačno mnogo iteracija* u slučaju da u grafu ne postoje *ciklusi negativne ukupne težine*. S druge strane, u slučajevima kada postoje ciklusi negativne ukupne težine, najkraći put zapravo i *ne postoji*, jer se tada obilaskom takvih ciklusa proizvoljan broj puta *dužina puta može proizvoljno smanjiti*.

Broj neophodnih iteracija se tipično može *smanjiti* tako što se u svakoj *k*-toj iteraciji za računanje vrijednosti $f_i^{(k)}$ kad god je to moguće uzimaju vrijednosti $f_j^{(k)}$ a ne $f_j^{(k-1)}$, dakle vrijednosti koje su prethodno izračunate u *istoj* a ne u prethodnoj iteraciji (to je, zbog redoslijeda izračunavanja, očito moguće za $j > i$). Drugim riječima, vrijednosti $f_i^{(k)}$ treba računati po formuli

$$f_i^{(k)} = \min\left(\{\, f_j^{(k-1)} + l_{i,j} \mid 1 \le j \le i \wedge (i,j) \in \mathrm{R} \,\} \cup \{\, f_j^{(k)} + l_{i,j} \mid i < j \le n \wedge (i,j) \in \mathrm{R} \,\}\right),\ \ i = n, n-1, \ldots, 1$$

Na ovaj način ne samo da se smanjuje broj iteracija, nego se i metod čini *lakšim za programiranje*. Zaista, ukoliko bi vrijednosti $f_i^{(k)}$ računali samo na osnovu vrijednosti $f_i^{(k-1)}$ iz prethodne iteracije, morali bismo u svakom trenutku u memoriji čuvati kako vrijednosti $f_i^{(k)}$ iz tekuće iteracije, tako i vrijednosti $f_i^{(k-1)}$ iz prethodne iteracije, jer bismo u suprotnom upisujući $f_i^{(k)}$ u istu lokaciju u kojoj se nalazila i vrijednost $f_i^{(k-1)}$ *prebrisali ovu vrijednost*, a koja nam kasnije može zatrebati. Međutim, ukoliko uvijek koristimo "*najsvježiju*" izračunatu vrijednost za f_i, možemo prosto čim izračunamo $f_i^{(k)}$ upisati izračunatu vrijednost u *istu memorijsku lokaciju* gdje smo čuvali vrijednost $f_i^{(k-1)}$. Drugim riječima, dovoljno je čuvati (recimo, u nekom vektoru) samo *najaktuelnije* vrijednosti za f_i, a nakon toga se gornje računanje prosto izvodi algoritamskim korakom

$$f_i \leftarrow \min\{\, f_j + l_{i,j} \mid 1 \le i \le n, (i,j) \in \mathrm{R} \,\},\ \ i = n, n-1, \ldots, 1$$

gdje simbol "$\leftarrow$" predstavlja *zamjenu* tekuće vrijednosti promjenljive sa lijeve strane izračunatom vrijednošću sa desne strane, u duhu *imperativnih programskih jezika* (poput uloge operatora "=" u programskom jeziku *C* i drugim jezicima izvedenim iz njega).

Može se dokazati da uz ovakvu izvedbu algoritma, broj iteracija ne može premašiti *n*. Slijedi da ukoliko se algoritam *ne završi nakon n iteracija*, možemo sigurno znati da u grafu *postoji ciklus negativne ukupne težine*, tako da najkraći put zapravo i *ne postoji*. Broj operacija koje se izvrše u svakoj iteraciji očigledno je proporcionalan broju grana *m* (kao kod Bellmanovog algoritma), tako da je ukupno trajanje algoritma u najgorem slučaju proporcionalno sa $m\,n$.

Ukoliko se Bellmanova verzija Bellman-Fordovog algoritma izvodi *unaprijed*, tada se na početku postavlja $f_1^{(0)} = 0$ i $f_j^{(0)} = \infty$ za $j = 2 \,..\, n$ nakon čega se kroz iteracije računaju vrijednosti

$$f_j^{(k)} = \min\{\, f_i^{(k-1)} + l_{i,j} \mid 1 \le i \le n, (i,j) \in \mathrm{R} \,\},\ \ j = 1 \,..\, n$$

odnosno

$$f_j^{(k)} = \min\left(\{\, f_i^{(k)} + l_{i,j} \mid 1 \le i < j, (i,j) \in \mathrm{R} \,\} \cup \{\, f_i^{(k-1)} + l_{i,j} \mid j \le i \le n, (i,j) \in \mathrm{R} \,\}\right),\ \ j = 1 \,..\, n$$

ukoliko u svakom koraku uzimamo u obzir *najaktuelniju* vrijednost za f_j. Ovo računanje se u suštini izvodi jednostavnim algoritamskim korakom

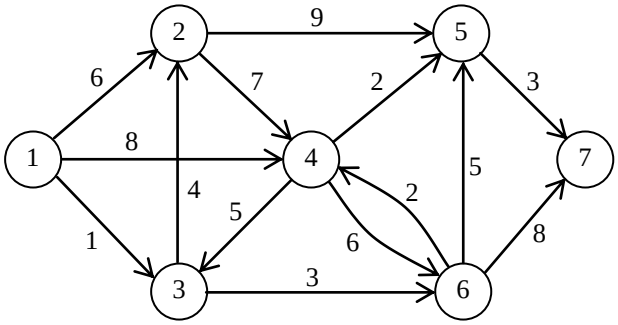
$$f_j \leftarrow \min\{\, f_i + l_{i,j} \mid 1 \le i \le n, (i,j) \in \mathrm{R} \,\}$$

s obzirom da se najaktuelnije vrijednosti za f_j mogu prosto čuvati u *jednom vektoru* (uz *prepisivanje* novoizračunatih vrijednosti preko starih vrijednosti).

Pri ručnom radu moguće je uštediti nešto računanja, jer se u svakoj *k* toj iteraciji ne moraju ponovo računati one veličine f_i (odnosno f_j) koje zavise samo od onih veličina koje se u prethodnoj (k–1-voj) iteraciji nisu promijenile u odnosu na njihovu raniju vrijednost. Međutim, pri implementaciji na računaru, ove uštede nije moguće ostvariti, jer sama provjera da li je potrebno računati veličinu f_i (odnosno f_j) ili ne troši praktično isto vrijeme kao i njeno računanje, tako da od nje *nema svrhe*.

Vrijedi još napomenuti da se obrada čvorova u toku jedne iteracije *ne mora vršiti striktno po navedenom redoslijedu*, jedino je bitno da se u toku iteracije *obrade svi čvorovi*. Broj neophodnih iteracija može ovisiti od redoslijeda obrade čvorova, tako da postoje neke *empirijske preporuke* za redoslijed obrade čvorova koje često mogu smanjiti broj neophodnih iteracija, u šta se ovdje nećemo upuštati.

- ***Primjer***: Koristeći Bellmanovu verziju Bellman-Fordovog algoritma, naći najkraći put između čvorova 1 i 7 u grafu sa sljedeće slike.



Težinska matrica za ovaj graf glasi

$$\mathbf{L} = \begin{pmatrix} 0 & 6 & 1 & 8 & \infty & \infty & \infty \\ \infty & 0 & \infty & 7 & 9 & \infty & \infty \\ \infty & 4 & 0 & \infty & \infty & 3 & \infty \\ \infty & \infty & 5 & 0 & 2 & 6 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & 3 \\ \infty & \infty & \infty & 2 & 5 & 0 & 8 \\ \infty & \infty & \infty & \infty & \infty & \infty & 0 \end{pmatrix}$$

Na ovaj graf očigledno nije moguće primijeniti Bellmanov algoritam, jer u njemu *postoje ciklusi* (takvi su recimo ciklusi $2 \to 4 \to 3 \to 2$ i $4 \to 3 \to 6 \to 4$). Mada se Bellmanova verzija Bellman-Fordovog algoritma češće izvodi *unazad* (jer je tako povoljnije s obzirom na tipične načine reprezentacije grafova), ovdje ćemo postupak obaviti *unaprijed*, da lakše uočimo sličnosti i razlike u odnosu na Fordovu verziju algoritma (koja se tipično obavlja upravo unaprijed), a koju ćemo uskoro demonstrirati. Dakle, inicijalno ćemo postaviti $f_1^{(0)} = 0$ i $f_j^{(0)} = \infty$ za $j = 2 \,..\, 7$ nakon čega u prvoj iteraciji imamo

$f_1^{(1)} = \min \{ f_1^{(0)} \} = 0$
$f_2^{(1)} = \min \{ f_1^{(1)} + l_{1,2}, f_2^{(0)}, f_3^{(0)} + l_{3,2} \} = \min \{0 + 6, \infty, \infty\} = 6$ (za $i = 1$)
$f_3^{(1)} = \min \{ f_1^{(1)} + l_{1,3}, f_3^{(0)}, f_4^{(0)} + l_{4,3} \} = \min \{0 + 1, \infty, \infty\} = 1$ (za $i = 1$)
$f_4^{(1)} = \min \{ f_1^{(1)} + l_{1,4}, f_2^{(1)} + l_{2,4}, f_4^{(0)}, f_6^{(0)} + l_{6,4} \} = \min \{0 + 8, 6 + 7, \infty, \infty\} = 8$ (za $i = 1$)
$f_5^{(1)} = \min \{ f_2^{(1)} + l_{2,5}, f_4^{(1)} + l_{4,5}, f_5^{(0)}, f_6^{(0)} + l_{6,5} \} = \min \{6 + 9, 8 + 2, \infty, \infty\} = 10$ (za $i = 4$)
$f_6^{(1)} = \min \{ f_3^{(1)} + l_{3,6}, f_4^{(1)} + l_{4,6}, f_6^{(0)} \} = \min \{1 + 3, 8 + 4, \infty\} = 4$ (za $i = 3$)
$f_7^{(1)} = \min \{ f_5^{(1)} + l_{5,7}, f_6^{(1)} + l_{6,7}, f_7^{(0)} \} = \min \{10 + 3, 4 + 8, \infty\} = 12$ (za $i = 6$)

Naravno, računanja koja se ovdje javljaju poput $f_2^{(1)} = \min \{ f_1^{(1)} + l_{1,2}, f_2^{(0)}, f_3^{(0)} + l_{3,2} \}$ efektivno se izvode algoritamskim korakom poput $f_2 \leftarrow \min \{ f_1 + l_{1,2}, f_2, f_3 + l_{3,2} \}$. U drugoj iteraciji imamo:

$f_1^{(2)} = \min \{ f_1^{(1)} \} = 0$
$f_2^{(2)} = \min \{ f_1^{(2)} + l_{1,2}, f_2^{(1)}, f_3^{(1)} + l_{3,2} \} = \min \{0 + 6, 6, 1 + 4\} = 5$ (za $i = 3$)
$f_3^{(2)} = \min \{ f_1^{(2)} + l_{1,3}, f_3^{(1)}, f_4^{(1)} + l_{4,3} \} = \min \{0 + 1, 1, 8 + 5\} = 1$ (za $i = 1$)
$f_4^{(2)} = \min \{ f_1^{(2)} + l_{1,4}, f_2^{(2)} + l_{2,4}, f_4^{(1)}, f_6^{(1)} + l_{6,4} \} = \min \{0 + 8, 5 + 7, 8, 4 + 2\} = 6$ (za $i = 6$)
$f_5^{(2)} = \min \{ f_2^{(2)} + l_{2,5}, f_4^{(2)} + l_{4,5}, f_5^{(1)}, f_6^{(1)} + l_{6,5} \} = \min \{5 + 9, 6 + 2, 10, 4 + 5\} = 8$ (za $i = 4$)
$f_6^{(2)} = \min \{ f_3^{(2)} + l_{3,6}, f_4^{(2)} + l_{4,6}, f_6^{(1)} \} = \min \{1 + 3, 8 + 4, 4\} = 4$ (za $i = 3$)
$f_7^{(2)} = \min \{ f_5^{(2)} + l_{5,7}, f_6^{(2)} + l_{6,7}, f_7^{(1)} \} = \min \{8 + 3, 4 + 8, 12\} = 11$ (za $i = 5$)

U trećoj iteraciji imamo:

$f_1^{(3)} = \min \{ f_1^{(2)} \} = 0$
$f_2^{(3)} = \min \{ f_1^{(3)} + l_{1,2}, f_2^{(2)}, f_3^{(2)} + l_{3,2} \} = \min \{0 + 6, 5, 1 + 4\} = 5$ (za $i = 3$)
$f_3^{(3)} = \min \{ f_1^{(3)} + l_{1,3}, f_3^{(2)}, f_4^{(2)} + l_{4,3} \} = \min \{0 + 1, 1, 6 + 5\} = 1$ (za $i = 1$)
$f_4^{(3)} = \min \{ f_1^{(3)} + l_{1,4}, f_2^{(3)} + l_{2,4}, f_4^{(2)}, f_6^{(2)} + l_{6,4} \} = \min \{0 + 8, 5 + 7, 6, 4 + 2\} = 6$ (za $i = 6$)

$f_5^{(3)} = \min \{ f_2^{(3)} + l_{2,5}, f_4^{(3)} + l_{4,5}, f_5^{(2)}, f_6^{(2)} + l_{6,5}\} = \min \{5+9, 6+2, 8, 4+5\} = 8$ (za $i = 4$)

$f_6^{(3)} = \min \{ f_3^{(3)} + l_{3,6}, f_4^{(3)} + l_{4,6}, f_6^{(2)}\} = \min \{1+3, 8+4, 4\} = 4$ (za $i = 3$)

$f_7^{(3)} = \min \{ f_5^{(3)} + l_{5,7}, f_6^{(3)} + l_{6,7}, f_7^{(2)}\} = \min \{8+3, 4+8, 11\} = 11$ (za $i = 5$)

Kako su u ovoj iteraciji sve vrijednosti f_j jednake vrijednostima u *prethodnoj iteraciji*, algoritam se ovdje završava (u načelu, veličine $f_1^{(3)}$, $f_2^{(3)}$, $f_4^{(3)}$, $f_5^{(3)}$, $f_6^{(3)}$ i $f_7^{(3)}$ nije ni trebalo računati, s obzirom da zavise *samo od veličina koje se nisu promijenile u odnosu na prethodnu iteraciju*). Dakle, dužina traženog najkraćeg puta iznosi $f_7^{(3)} = 11$. Sam najkraći put lako očitavamo "razmotavanjem" unazad, i on glasi $1 \to 3 \to 6 \to 4 \to 5 \to 7$.

Slično kao kod Bellmanovog algoritma, i za Bellmanovu verziju Bellman-Fordovog algoritma moguće je formirati razne *tabelarne sheme* koje omogućavaju *pregledniji rad*. Međutim, ovdje nećemo prikazivati takve sheme, s obzirom na činjenicu da je Fordova verzija ovog algoritma svakako preglednija i praktičnija.

Razmotrimo sada *Fordovu verziju* Bellman-Fordovog algoritma. Mada je ukupna količina računanja kod ove verzije algoritma *otprilike ista* kao u Bellmanovoj verziji, ova verzija je mnogo preglednija i praktičnija, kako za *ručni rad*, tako i za *izvedbu na računaru*. Osnovna ideja ove verzije sastoji se u sljedećem. Bellmanova verzija u svakoj iteraciji *popravlja* vrijednosti f_j za sve čvorove računajući najmanju od svih vrijednosti $f_i + l_{i,j}$ za sve grane (i, j) koje *ulaze* u čvor j. Kod Fordove verzije, u toku jedne iteracije svaki put kad se razmatra neki čvor i, umjesto da se popravlja vrijednost f_i za taj čvor, testiraju se vrijednosti f_j za *sve njegove susjede*. Ispostavi li se da je vrijednost $f_i + l_{i,j}$ *manja* od trenutne vrijednosti f_j, vrijednost f_j se *zamjenjuje* sa vrijednošću $f_i + l_{i,j}$. U suprotnom, vrijednost f_j se ostavlja *netaknutom*. Drugim riječima, za svaki čvor i, za sve grane (i, j) koje iz njega izlaze izvršava se algoritamski korak

$$f_j \leftarrow \min \{ f_j, f_i + l_{i,j}\}$$

odnosno, jedna iteracija algoritma može se opisati slijedom algoritamskih koraka

$$f_j \leftarrow \min \{ f_j, f_i + l_{i,j}\} \text{ za sve } (i, j) \in \mathrm{R},\ i = 1 \,..\, n$$

Ovi algoritamski koraci se očigledno jednostavnije izvode nego algoritamski koraci kod Bellmanove verzije algoritma. Kaže se da Bellmanova verzija ***povlači*** (engl. *pull*) u čvor i vrijednosti f_j svih njegovih susjeda, dok Fordova verzija ***potiskuje*** (engl. *push*) vrijednost f_i svim svojim susjedima. Samih koraka ima *više*, ali se u svakom koraku vrši *samo jedno uspoređivanje*, tako da ukupan broj operacija po iteraciji *ostaje isti*. Valjanost ovog postupka slijedi iz činjenice da će sve vrijednosti f_j dobiti svoje ispravne vrijednosti kakve zahtijeva Bellmanov princip optimalnosti (tj. jednake *minimumu od svih vrijednosti* $f_i + l_{i,j}$) čim se obrade svi čvorovi koji za svog susjeda imaju čvor j.

Pored činjenice da je Fordova verzija *praktičnija* za rad, ona ima i dodatnu prednost što je u svakoj iteraciji dovoljno razmatrati *samo one čvorove* za koje se vrijednost f_i u prethodnoj iteraciji *promijenila* u odnosu na ranije iteracije. Za razliku od eventualnih testova za skraćivanje računanja u Bellmanovoj verziji algoritma, ovo testiranje veoma se jednostavno izvodi i treba ga vršiti, s obzirom da se na taj način može znatno skratiti računanje. Ovo je ujedno i razlog zbog čega je Fordova verzija Bellman-Fordovog algoritma tipično *efikasnija u praksi*. Stoga se, u praktičnim primjenama Bellman-Fordovog algoritma, koristi gotovo isključivo Fordova verzija.

- ***Primjer***: Koristeći Fordovu verziju Bellman-Fordovog algoritma, naći najkraći put između čvorova 1 i 7 u grafu iz prethodnog primjera.

Slično kao kod Bellmanove verzije, inicijalno postavljamo $f_1 = 0$ i $f_j = \infty$ za $j = 2 \,..\, 7$. Nakon toga, u prvoj iteraciji imamo:

Čvor 1: $f_2 \leftarrow \min \{ f_2, f_1 + l_{1,2}\} = \min \{\infty, 0+6\} = 6$ ($i = 1$)

$f_3 \leftarrow \min \{ f_3, f_1 + l_{1,3}\} = \min \{\infty, 0+1\} = 1$ ($i = 1$)

$f_4 \leftarrow \min \{ f_4, f_1 + l_{1,4}\} = \min \{\infty, 0+8\} = 8$ ($i = 1$)

Čvor 2: $f_4 \leftarrow \min \{ f_4, f_2 + l_{2,4}\} = \min \{8, 6+7\} = 8$

$f_5 \leftarrow \min \{ f_5, f_2 + l_{2,5}\} = \min \{\infty, 6+9\} = 15$ ($i = 2$)

Čvor 3: $f_2 \leftarrow \min \{ f_2, f_3 + l_{3,2}\} = \min \{6, 1+4\} = 5$ ($i = 3$)

$f_6 \leftarrow \min \{ f_6, f_3 + l_{3,6}\} = \min \{\infty, 1+3\} = 4$ ($i = 3$)

Čvor 4: $f_3 \leftarrow \min\{ f_3, f_4 + l_{4,3}\} = \min\{1, 8+5\} = 1$
$f_5 \leftarrow \min\{ f_5, f_4 + l_{4,5}\} = \min\{15, 8+2\} = 10$ ($i = 4$)
$f_6 \leftarrow \min\{ f_6, f_4 + l_{4,6}\} = \min\{4, 8+6\} = 4$

Čvor 5: $f_7 \leftarrow \min\{ f_7, f_5 + l_{5,7}\} = \min\{\infty, 10+3\} = 13$ ($i = 5$)

Čvor 6: $f_4 \leftarrow \min\{ f_4, f_6 + l_{6,4}\} = \min\{8, 4+2\} = 6$ ($i = 6$)
$f_5 \leftarrow \min\{ f_5, f_6 + l_{6,5}\} = \min\{10, 4+5\} = 9$ ($i = 6$)
$f_7 \leftarrow \min\{ f_7, f_6 + l_{6,7}\} = \min\{13, 4+8\} = 12$ ($i = 6$)

Čvor 7: –

Sa strane smo pribilježili redni broj tekućeg čvora i u slučaju da je došlo *do korekcije* odgovarajuće vrijednosti f_j za neki od njegovih susjeda, što će nam biti kasnije od koristi kada bude trebalo očitati traženi najkraći put. Za čvor 7 nismo imali nikakva računanja, zbog činjenice da *iz ovog čvora ne izlazi niti jedna grana*. U drugoj iteraciji sada imamo:

Čvor 1: $f_2 \leftarrow \min\{ f_2, f_1 + l_{1,2}\} = \min\{5, 0+6\} = 5$
$f_3 \leftarrow \min\{ f_3, f_1 + l_{1,3}\} = \min\{1, 0+1\} = 1$
$f_4 \leftarrow \min\{ f_4, f_1 + l_{1,4}\} = \min\{6, 0+8\} = 6$

Čvor 2: $f_4 \leftarrow \min\{ f_4, f_2 + l_{2,4}\} = \min\{6, 5+7\} = 6$
$f_5 \leftarrow \min\{ f_5, f_2 + l_{2,5}\} = \min\{9, 5+9\} = 9$

Čvor 3: $f_2 \leftarrow \min\{ f_2, f_3 + l_{3,2}\} = \min\{5, 1+4\} = 5$
$f_6 \leftarrow \min\{ f_6, f_3 + l_{3,6}\} = \min\{4, 1+3\} = 4$

Čvor 4: $f_3 \leftarrow \min\{ f_3, f_4 + l_{4,3}\} = \min\{1, 6+5\} = 1$
$f_5 \leftarrow \min\{ f_5, f_4 + l_{4,5}\} = \min\{9, 6+2\} = 8$ ($i = 4$)
$f_6 \leftarrow \min\{ f_6, f_4 + l_{4,6}\} = \min\{4, 6+6\} = 4$

Čvor 5: $f_7 \leftarrow \min\{ f_7, f_5 + l_{5,7}\} = \min\{12, 8+3\} = 11$ ($i = 5$)

Čvor 6: $f_4 \leftarrow \min\{ f_4, f_6 + l_{6,4}\} = \min\{6, 4+2\} = 6$
$f_5 \leftarrow \min\{ f_5, f_6 + l_{6,5}\} = \min\{8, 4+5\} = 8$
$f_7 \leftarrow \min\{ f_7, f_6 + l_{6,7}\} = \min\{11, 4+8\} = 11$

Čvor 7: –

Kako su se na kraju druge iteracije promijenile jedino vrijednosti za f_5 i f_7, u trećoj iteraciji treba razmotriti samo čvorove 5 i 7:

Čvor 5: $f_7 \leftarrow \min\{ f_7, f_5 + l_{5,7}\} = \min\{11, 8+3\} = 11$

Čvor 7: –

Kako u trećoj iteraciji nije došlo *ni do kakve promjene* u izračunatim vrijednostima f_j, algoritam se završava. Traženi najkraći put očitava se *razmotavanjem unazad*, prateći koje su vrijednosti i bile aktuelne u trenutku kada je došlo do promjene odgovarajućih vrijednosti f_j.

Prilikom *ručne izvedbe* Fordovog algoritma, svi provedeni koraci se mogu pregledno predstaviti u *tabelarnoj formi* (alternativno, ukoliko sam graf nije previše gust i nepregledan, prateće informacije se mogu bilježiti *kraj čvorova na crtežu grafa*). Takva tabelarna forma za prvu iteraciju za graf iz prethodnog primjera mogla bi izgledati recimo ovako (nije teško uočiti šta predstavljaju odgovarajuća polja u ovoj tabeli):

i \ j	1	2	3	4	5	6	7	f_i
	0	∞	∞	∞	∞	∞	∞	
1		6	1	8				0
2				13	15			∞, 6, 5
3		5				4		∞, 1
4			13		10	14		∞, 8, 6
5							13	∞, 15, 10, 9
6				6	9		12	∞, 4
7								∞, 12

Slično, drugoj iteraciji mogla bi odgovarati sljedeća tabela:

i \ *j*	1	2	3	4	5	6	7	f_i
	0	5	1	6	9	4	12	
1		6	1	8				0
2				12	14			5
3		5				4		1
4			11		8	12		6
5							11	9, 8
6				6	9		12	4
7								12, 11

Konačno, trećoj iteraciji odgovara sljedeća tabela:

i \ *j*	1	2	3	4	5	6	7	f_i
	0	5	1	6	8	4	11	
5							11	8
7								11

Slično kao kod Bellmanove verzije algoritma, ni kod Fordove verzije redoslijed kojim se obrađuju čvorovi u suštini *nije bitan*. Jedino što je bitno je da se u toku jedne iteracije obrade *svi čvorovi*. Zapravo, čak nije neophodno ni da svi čvorovi budu obrađeni u jednoj iteraciji, ali se algoritam završava *onog trenutka kada se ne može više promijeniti vrijednost f_j niti za jedan čvor*.

Prethodno prikazana Fordova verzija algoritma izvodila se *unaprijed*, tj. polazeći od početne vrijednosti $f_1 = 0$. Radi kompletnosti, treba navesti i da se Fordova verzija također može izvoditi i *unazad*, tj, polazeći od početne vrijednosti $f_n = 0$ i izvršavajući za svaki čvor j algoritamski korak $f_i \leftarrow \min \{ f_i, f_j + l_{i,j} \}$ za sve čvorove *i kojima je čvor j susjed*, tj. za sve $(i, j) \in R$. Međutim, Fordova verzija Bellman-Fordovog algoritma se u praksi *gotovo uvijek izvodi unaprijed*, iz istih razloga zbog kojih se Bellmanova verzija gotovo uvijek izvodi unazad. Naime, pri reprezentaciji grafa pomoću listi susjedstva, mnogo je lakše saznati koji su čvorovi susjedi razmatranog čvora nego saznati kojim je čvorovima razmatrani čvor susjed.

Dijkstrin algoritam

Sada ćemo preći na opis *Dijkstrinog algoritma*. Kao što je već rečeno, ovaj algoritam je *mnogo brži* od Bellman-Fordovog algoritma, ali ima ograničenje da *dužine* (*težine*) *svih grana moraju biti nenegativne*. Sretna je okolnost da je *većina grafova koji se susreću u praktičnim primjenama takva* (ali ne i svi). Dijkstrin algoritam kombinira ideje iz Bellman-Fordovog algoritma sa jednim principom koji je preuzet iz jednog nešto starijeg ali mnogo neefikasnijeg algoritma za nalaženje najkraćeg puta, koji se naziva *Dantzigov algoritam* i koji se danas više ne koristi. Prema ovom principu, ukoliko su dužine svih grana nenegativne, tada ukoliko je (i, j) *najkraća od svih grana koje vode iz čvora i*, ta grana ujedno *mora biti i najkraći put iz čvora i u čvor j*. Zaista, uz nenegativne dužine grana, svaki drugi put koji bi vodio iz čvora i u čvor j morao bi biti duži od dužine grane (i, j), ako ni zbog čega drugog, onda zbog činjenice da su sve druge grane koje izlaze iz čvora i duže od grane (i, j).

Izloženi princip omogućava dvije stvari. Prvo, zahvaljujući ovom principu, možemo biti sigurni da će postojati čvorovi za koje se vrijednosti f_i u daljim iteracijama *neće mijenjati*, odnosno za koje možemo biti sigurni *da smo već odredili dužinu najkraćeg puta koji vodi do njih*. Inače, Dijkstra je za veličine f_i koristio naziv ***potencijali čvorova***, tako da ćemo ovdje i mi prihvatiti tu terminologiju. Recimo, konačna vrijednost potencijala $f_1 = 0$ je utvrđena *već na samom početku*. Dalje, ukoliko je $(1, j)$ *najkraća grana* koja vodi iz početnog čvora 1, konačna vrijednost potencijala čvora j će biti utvrđena *već nakon prve iteracije*, tako da sada već imamo dva čvora čija je konačna vrijednost potencijala utvrđena. Analogno možemo zaključiti da će nakon svake od iteracija, *najmanji od potencijala za sve čvorove koji do tada nisu dobili utvrđenu konačnu vrijednost postati utvrđena konačna vrijednost tog potencijala*. Drugo, ovaj princip nam omogućava da u svakoj iteraciji ne moramo prolaziti *kroz sve čvorove* da bismo "popravljali" vrijednosti potencijala za njihove susjede (kao u Fordovoj verziji Bellman-Fordovog algoritma), nego je dovoljno popravke vršiti *samo od čvora i koji je u prethodnoj iteraciji dobio utvrđenu konačnu vrijednost potencijala* (taj čvor ćemo zvati ***referentni čvor***). Zaista, u odnosu na čvorove koji su u ranijim iteracijama (prije prethodne) već dobili utvrđene konačne vrijednosti potencijala, ništa *se više ne može promijeniti*, dok u odnosu na čvorove čije konačne vrijednosti potencijala još nisu utvrđeni, *ne vrijedi vršiti popravke* sve dok postoji mogućnost da

se vrijednost njihovog potencijala izmijeni. Sve ove ideje zajedno tvore *Dijkstrin algoritam*, koji se u vidu niza koraka može iskazati ovako (uz pretpostavku da se traži najkraći put od čvora 1 do čvora n).

1. Postaviti $f_1 = 0$ i $f_j = \infty$ za $j = 2 \,..\, n$.
2. Proglasiti čvor 1 za referentni čvor ($r = 1$).
3. Proglasiti potencijal f_r referentnog čvora konačnim. Ukoliko je $r = n$, preći na korak 6.
4. Za sve susjede j referentnog čvora r čiji potencijali *nisu konačni* obaviti algoritamski korak $f_j \leftarrow \min \{ f_j, f_r + l_{r,j} \}$. Drugim riječima, ukoliko je vrijednost $f_r + l_{r,j}$ manja od tekuće vrijednosti f_j, prepraviti f_j na ovu novu vrijednost. Također, svaki put kada se ova prepravka izvrši za neki čvor j, *zapamtiti* (recimo u nekoj promjenljivoj p_j) *koji je bio referentni čvor* kada je ta popravka vršena, tj. staviti $p_j \leftarrow r$). Ove informacije će nam biti potrebne da na kraju očitamo traženi najkraći put.
5. Od svih čvorova čiji potencijal još nije konačan, pronaći čvor k sa *najmanjim potencijalom*. Proglasiti čvor k za novi referentni čvor ($r \leftarrow k$) i vratiti se na korak 3.
6. Traženi najkraći put je pronađen. Njegova dužina iznosi f_n, pri čemu sam put možemo očitati razmotavanjem "unazad" prateći vrijednosti p_j počev od $j = n$.

Ukoliko najkraći put od čvora 1 do čvora n *ne postoji* (što se, uz nenegativne težine grana, može desiti *jedino ukoliko uopće ne postoji nikakav put* od čvora 1 do čvora n), algoritam će se završiti sa dužinom najkraćeg puta $f_n = \infty$. Ova situacija se može prepoznati i prije prirodnog završetka algoritma ukoliko u koraku 5. svi razmatrani čvorovi imaju isključivo potencijal ∞. U tom slučaju, algoritam se može prekinuti prije vremena.

Može se primijetiti da, za razliku od Bellman-Fordovog algoritma koji rješava problem najkraćeg puta tipa "jedan do svih" (odnosno "svi do jednog"), Dijkstrin algoritam rješava problem tipa "jedan do nekih", s obzirom da se u koraku 3. algoritam prekida čim potencijal čvora n postane konačan (u tom trenutku, poznati su najkraći putevi do onih i samo onih čvorova čiji je potencijal postao konačan). Ukoliko se umjesto toga algoritam izvodi *sve dok potencijali svih čvorova ne postanu konačni*, dobija se verzija Dijsktrinog algoritma koja rješava probleme tipa "jedan do svih".

- ***Primjer***: Koristeći Dijkstrin algoritam, naći najkraći put između čvorova 1 i 7 u grafu iz prethodnog primjera.

Tok algoritma najbolje je prikazati *u tabelarnoj formi* (u slučaju kada graf nije previše gust i nepregledan, odgovarajuće vrijednosti f_i i p_i obično se bilježe *direktno na crtežu*, kraj čvorova grafa:

Ref. čvor (r)	f_r	1	2	3	4	5	6	7
		0	∞	∞	∞	∞	∞	∞
1	0		6 (1)	**1 (1)**	8 (1)	∞	∞	∞
3	1		5 (3)		8 (1)	∞	**4 (3)**	∞
6	4		**5 (3)**		6 (6)	9 (6)		12 (6)
2	5				**6 (6)**	9 (6)		12 (6)
4	6					**8 (4)**		12 (6)
5	8							**11 (5)**

Pri svakom novom referentnom čvoru r, vrijednosti $f_r + l_{r,j}$ porede se sa aktuelnom vrijednošću f_j. Ukoliko je vrijednost $f_r + l_{r,j}$ manja od aktuelne vrijednosti f_j, u polje se upisuje *nova vrijednost* $f_r + l_{r,j}$, zajedno sa odgovarajućim $p_j = r$ (prikazano u zagradi), inače se aktuelne vrijednosti f_j i p_j prosto *prepisuju*. Svijetlo osjenčeno su prikazana polja koja odgovaraju izvršenim *promjenama*, dok su tamno osjenčeno i podebljano prikazana polja u kojima su potencijali dobili svoju *konačnu vrijednost*. Postupak se *završava* kada potencijal *ciljnog čvora* dobije *konačnu vrijednost* $f_7 = 11$.S obzirom da smo dobili $p_7 = 5$, $p_5 = 4$, $p_4 = 6$, $p_6 = 3$ i $p_3 = 1$, traženi najkraći put glasi $1 \rightarrow 3 \rightarrow 6 \rightarrow 4 \rightarrow 5 \rightarrow 7$.

Razmotrimo sada *vremensku složenost* Dijkstrinog algoritma. U svakoj iteraciji neki novi čvor postaje referentan, tako da broj iteracija *ne može preći n*. U svakoj iteraciji, u koraku 4 vrši se u najgorem slučaju onoliko popravki koliko ima grana koje izlaze iz referentnog čvora (dakle, u najgorem slučaju ne više od n). Također, izbor novog referentnog čvora u koraku 5 traži najviše n upoređivanja. Dakle, broj operacija u svakoj iteraciji ne prelazi iznos proprorcionalan sa n, tako da je ukupno trajanje algoritma u najgorem slučaju proporcionalno sa n^2.

U slučaju da je graf predstavljen težinskom matricom, u koraku 4. se moraju testirati *svi elementi* reda matrice koji odgovara referentnom čvoru, što je tačno n koraka. Izgleda da bi se u slučaju rijetkih grafova mogla ostvariti ušteda ukoliko bi graf bio predstavljen pomoću *listi susjedstva*, jer tada ukupan broj testova u svim iteracijama zajedno *ne bi mogao biti veći od ukupnog broja grana m*, što je za rijetke grafove bolje od n^2. Mada ovo zaista daje izvjesno ubrzanje kod rijetkih grafova, sama ova izmjena *nije dovoljna* da smanji red vremenske složenosti algoritma ispod n^2. Naime, kritičan je korak 5, koji u najgorem slučaju zahtijeva do n upoređivanja, *neovisno od toga kako je predstavljen graf*. Međutim, ukoliko bismo potencijale čuvali u nekoj strukturi podataka koja omogućava da se korak 5 izvede *brže*, mogli bismo znatno smanjiti vrijeme izvršavanja. Tu upravo svoju priliku dobijaju *redovi sa prioritetom*. Mada takve strukture podataka omogućavaju da se korak 5 izvede mnogo brže, problem nastaje u činjenici da se pohranjeni potencijali moraju povremeno i *mijenjati* (u koraku 4), a redovi sa prioritetom ne omogućavaju tako jednostavnu i efikasnu izmjenu podataka koji su u njima pohranjeni, tako da je očito potrebno vršiti razne kompromise. Ove ideje leže u osnovi *naprednih implementacija* Dijkstrinog algoritma, u koje se nećemo detaljnije upuštati. Nije na odmet još jednom istaći da sve te napredne implementacije pokazuju svoje prednosti u odnosu na elementarne implementacije *samo u slučaju rijetkih grafova*.

Može se primijetiti da su krajnje dobijene vrijednosti za f_i iste kod *sva tri opisana algoritma* (Bellmanov, Bellman-Fordov i Dijkstrin), pod uvjetom da se Dijsktrin algoritam provodi do trenutka kada *svi potencijali dobiju konačnu vrijednost*. Pri tome je interesantno da konačne vrijednosti f_i predstavljaju *jedno od optimalnih rješenja dualnog problema za problem najkraćeg puta*. Zahvaljujući ovoj činjenici, može se pokazati da se Dijkstrin algoritam zapravo spada u porodicu *primalno-dualnih metoda*, slično kao i mađarski algoritam (bez obzira što ovaj algoritam nije ni dualni ni primalni metod, s obzirom da sve do kraja algoritma djelimična rješenja koja se njim dobijaju nisu niti primalno niti dualno dopustiva). Ova činjenica može se iskoristiti i za razna *ubrzanja* Dijkstrinog algoritma. Na primjer, ukoliko se *nekako procijeni* neko "dobro" dualno dopustivo rješenje problema najkraćeg puta, Dijkstrin algoritam se može startati polazeći od takvog rješenja a ne "od nule", što može skratiti trajanje algoritma. Na ovoj ideji se zasniva popularni ***A* algoritam***, koji ovdje nećemo opisivati.

Renumeracija čvorova

Već je rečeno da se Bellmanov algoritam može koristiti samo za *usmjerene grafove bez ciklusa* koji su pri tome i *pravilno numerirani*. Ukoliko imamo graf koji je usmjeren i bez ciklusa, ali koji *nije pravilno numeriran*, potrebno je izvršiti *neki od algoritama za renumeraciju čvorova*, koji će *obezbijediti pravilnu numeraciju*. U teoriji grafova i teoriji algoritama poznati su brojni algoritmi za tu svrhu (algoritmi za *topološko sortiranje*), koji su *prilično efikasni* (vrijeme izvršavanja je tipično proporcionalno broju grana m) i *relativno jednostavni*. Ovi algoritmi ujedno i *otkrivaju* da li pravilna numeracija za graf *uopće postoji*. U te algoritme se ovdje nećemo upuštati (mada ćemo kasnije demonstrirati jedan takav algoritam kada budemo razmatrali problem tzv. *mrežnog planiranja*). Međutim, ukoliko nam je jedini cilj *pronaći pravilnu numeraciju* pri čemu nam *vrijeme izvršavanja nije previše bitno*, moguće je pravilnu numeraciju obaviti jednim jednostavnim postupkom, koji je u suštini *specijalan slučaj Bellman-Fordovog algoritma*, samo primijenjen na nalaženje *najdužeg* (a ne najkraćeg) *puta* u grafu *kod kojeg sve grane imaju dužinu* 1.

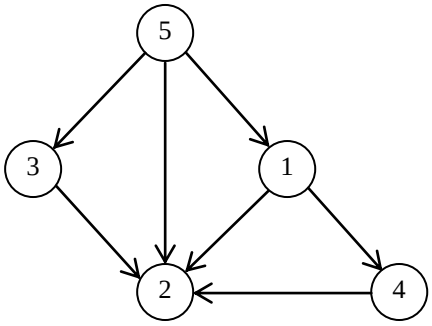
Prema ovom postupku, svim čvorovima j se dodjeljuje ***rang*** r_j koji se na početku za sve čvorove postavlja na vrijednost 0. Proces dalje teče kroz niz iteracija. U svakoj narednoj iteraciji se za svaki čvor j, $j = 1 \,..\, n$ izvršava algoritamski korak

$$r_j \leftarrow \max\left(\{r_j\} \cup \{r_i + 1 \mid 1 \le i \le n, (i, j) \in \mathrm{R}\}\right)$$

Iteracije se obavljaju sve dok se u nekoj iteraciji ne promijeni niti jedan rang u odnosu na vrijednosti iz prethodne iteracije. To se mora dogoditi nakon *najviše n iteracija*. U suprotnom, to ukazuje da pravilna numeracija razmatranog grafa *ne postoji*. Nije teško vidjeti da je u najgorem slučaju broj izvršenih operacija proporcionalan sa $n\,m$.

Po završetku iteracija, čvor sa rangom 0 je *početni čvor* (tj. čvor u koji *ne ulazi niti jedna grana*), dok rangovi ostalih čvorova predstavljaju *maksimalnu dužinu puta* (mjerenu *brojem grana*) koje vode od početnog čvora do njih. Samo renumeriranje se nakon toga vrši na sljedeći način. Čvor sa najmanjim rangom dobija najmanji redni broj (tj. 1). Dalje se u čvorovi numeriraju u rastućem redoslijedu u skladu sa njihovim rangom (tj. sljedeći se numeriraju čvorovi sa prvim sljedećim rangom po veličini, itd.). Redoslijed kojim se numeriraju čvorovi sa istim rangom *nije bitan*.

- ***Primjer*** : Pronaći pravilnu numeraciju za graf sa sljedeće slike.



Očigledno je da ovaj graf *nije pravilno numeriran*. Inicijaliziraćemo rangove svih čvorova na 0, nakon čega u prvoj iteraciji imamo:

$r_1 \leftarrow \max\{r_1, r_5+1\} = \max\{0, 0+1\} = 1$
$r_2 \leftarrow \max\{r_2, r_1+1, r_3+1, r_4+1, r_5+1\} = \max\{0, 1+1, 0+1, 0+1, 0+1\} = 2$
$r_3 \leftarrow \max\{r_3, r_5+1\} = \max\{0, 0+1\} = 1$
$r_4 \leftarrow \max\{r_4, r_1+1\} = \max\{0, 1+1\} = 2$
$r_5 \leftarrow \max\{r_5\} = \max\{0\} = 0$

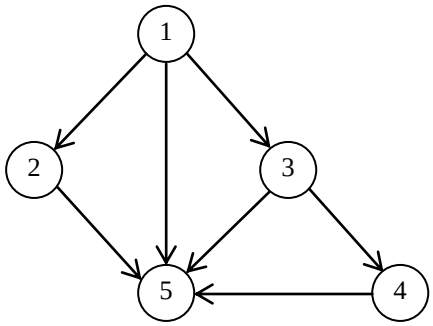
U drugoj iteraciji imamo:

$r_1 \leftarrow \max\{r_1, r_5+1\} = \max\{1, 0+1\} = 1$
$r_2 \leftarrow \max\{r_2, r_1+1, r_3+1, r_4+1, r_5+1\} = \max\{2, 1+1, 1+1, 2+1, 0+1\} = 3$
$r_3 \leftarrow \max\{r_3, r_5+1\} = \max\{1, 0+1\} = 1$
$r_4 \leftarrow \max\{r_4, r_1+1\} = \max\{2, 1+1\} = 2$
$r_5 \leftarrow \max\{r_5\} = \max\{0\} = 0$

U trećoj iteraciji imamo:

$r_1 \leftarrow \max\{r_1, r_5+1\} = \max\{1, 0+1\} = 1$
$r_2 \leftarrow \max\{r_2, r_1+1, r_3+1, r_4+1, r_5+1\} = \max\{2, 1+1, 1+1, 2+1, 0+1\} = 3$
$r_3 \leftarrow \max\{r_3, r_5+1\} = \max\{1, 0+1\} = 1$
$r_4 \leftarrow \max\{r_4, r_1+1\} = \max\{2, 1+1\} = 2$
$r_5 \leftarrow \max\{r_5\} = \max\{0\} = 0$

Kako u trećoj iteraciji *nije bilo nikakvih promjena* u rangovima u odnosu na prethodnu iteraciju, svi rangovi su određeni. Čvor 5 sa rangom 0 dobija indeks 1, čvorovi 1 i 3 sa rangom 1 dobijaju indekse 2 i 3 (svejedno koji će dobiti indeks 2 a koji indeks 3), čvor 4 sa rangom 2 dobija indeks 4 i, konačno, čvor 2 sa rangom 3 dobija indeks 5:



Kao što je već ranije nagoviješteno, kada budemo razmatrali *tehnike mrežnog planiranja*, prikazaćemo još jedan jednostavan algoritam za renumeraciju čvorova koji je *posebno jednostavan za ručni rad*, a uz *kvalitetnu implementaciju* (koja nije posve očigledna iz samog opisa algoritma) može biti vrlo efikasan i za realizaciju na računaru.

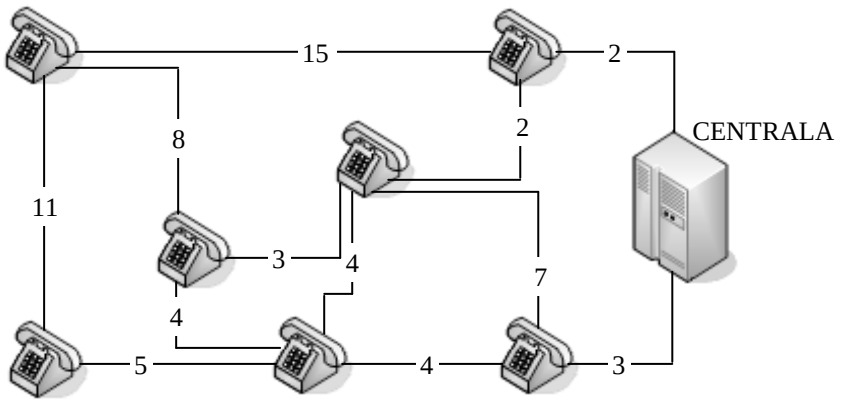
Problem minimalnog povezujućeg stabla i linearno programiranje

Problem minimalnog povezujućeg stabla spada u još jedan interesantan *ekstremalni problem teorije grafova* koji spada u oblast *kombinatorne optimizacije*. Ovaj problem je karakterističan po tome što se *također može izraziti modelom linearnog programiranja* (na ne baš očigledan način), samo što se to *nikada ne vrši u praksi*, ne samo zbog toga što za rješavanje problema minimalnog povezujućeg stabla postoje vrlo efikasni i jednostavni algoritmi, nego i zbog činjenice da je dobijeni model linearnog programiranja *neprihvatljivo glomazan* (do te mjere da je praktično *neupotrebljiv*) čak i u slučaju *posve jednostavnih grafova*. Bez obzira na to, ovdje će ipak biti prikazano kako se i problemi ovog tipa mogu svesti na probleme linearnog programiranja, čisto sa ciljem da se *demonstrira u kojoj mjeri su modeli linearnog programiranja obuhvatni*, kao i da se *ilustriraju neke ideje kako se neki koncepti teorije grafova mogu iskazati jezikom matematičkog programiranja*. Naime, ideje koje će biti prikazane ovdje mogu se iskoristiti za modeliranje nekih *znatno težih* problema teorije grafova.

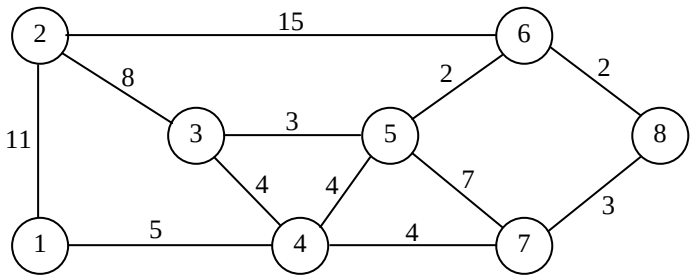
Istaknimo prvo *u čemu se sastoji problem minimalnog povezujućeg stabla* i *u čemu je njegov* značaj. U praksi je često potrebno izvršiti *povezivanje određeniog broja tačaka*, recimo pri izgradnji *telefonskih, računarskih, elektro-energetskih, vodovodnih, kanalizacionih, gasovodnih* i *raznih drugih tipova mreža*. U ovim tipovima mreža, *sve tačke moraju biti povezane*, tako da se one opisuju *povezanim grafovima* (podsjetimo se da je graf *povezan* ukoliko postoji *barem jedan put* između *bilo koja dva čvora u grafu*). Pretpostavimo sada da je cilj da se pri tom povezivanju utroši *najmanje resursa*. Ako *težine grana* $c_{i,j}$ predstavljaju *utrošak odgovarajućeg resursa* za povezivanje dvaju tačaka u grafu, onda će *optimalno povezivanje*, tj. *minimalni utrošak resursa* odgovarati nalaženju *djelimičnog grafa* datog grafa koji je *povezan* i koji je *minimalan* u smislu da mu je *suma težina grana manja ili jednaka od sume težine grana svih drugih djelimičnih grafova* datog grafa.

Lako je pokazati da *optimalno rješenje* (*najkraće povezivanje*) *ne smije sadržavati konture* (*cikluse*), tako da *optimalno rješenje mora biti povezani graf bez kontura*, odnosno *stablo* (*drvo*). Zbog toga se ovdje i govori o *minimalnom povezujućem stablu*. Da bismo to pokazali, pretpostavimo *suprotno* da optimalno rješenje *sadrži ciklus*. Izaberimo bilo koju granu ciklusa (*i*, *j*) i *izbrišimo je*. Pošto su čvorovi *i* i *j* ležali na ciklusu, oni će i *dalje biti povezani*. Slijedi da *izbacivanje ma koje grane koja je u ciklusu neće* "*razvezati*" *graf*, odnosno graf će *ostati povezan*, a njegova *ukupna težina* (tj. suma težina njegovih grana) će biti *smanjena za težinu izbačene grane*. Dakle, *graf koji posjeduje ciklus ne može biti optimalno rješenje problema minimalnog povezivanja*.

Kao primjer problema nalaženja minimalnog povezujućeg stabla se može posmatrati zadatak *razvoda telefonske mreže* u nekom naseljenom mjestu. Neka pojedine kuće predstavljaju *čvorove grafa*, dok *grane grafa* modeliraju telekomunikacione kablove. Svaka od grana ima svoju "*težinu*", tj. *dužinu* ili *cijenu* utrošenog kabla. Povezivanje treba obaviti tako da se *povežu sve kuće* a da utrošak *kablova bude što je god moguće manji*. Sljedeća slika prikazuje jedan primjer sa datom dužinom veza između pojedinih kuća.



Ovaj problem se može modelirati kao problem nalaženja minimalnog povezujućeg stabla u grafu (neusmjerenom) sa sljedeće slike:



Razmotrimo sada kako se problem minimalnog povezujućeg stabla može izraziti kao model linearnog programiranja. Kako usmjerenost ne igra nikakvu ulogu u problemima minimalnog povezujućeg stabla, bez umanjenja općenitosti smatraćemo da grafovi koje razmatramo *nisu usmjereni* i da se (neusmjerena) grana koja povezuje dva čvora modelira parom oblika (i, j) pri čemu je $i < j$. Promjenljive modela će ponovo biti *logičkog* (0–1) *tipa*, pri čemu $x_{i,j} = 1$ označava da se grana (i, j) *nalazi* u traženom minimalnom povezujućem stablu, dok $x_{i,j} = 0$ označava da se odgovarajuća grana u razmatranom stablu *ne nalazi*.

Prije nego što pređemo na razmatranje općeg slučaja, probajmo prvo konstruisati odgovarajući model linearnog programiranja za gore navedeni primjer. Funkcija cilja, koju treba minimizirati, očigledno se može prikazati u obliku

$$Z = 11\,x_{1,2} + 5\,x_{1,4} + 8\,x_{2,3} + 15\,x_{2,6} + 4\,x_{3,4} + 3\,x_{3,5} + 4\,x_{4,5} + 4\,x_{4,7} + 2\,x_{5,6} + 7\,x_{5,7} + 2\,x_{6,8} + 3\,x_{7,8}$$

Svako stablo koje ima n čvorova mora imati tačno $n-1$ grana. Kako je u posmatranom primjeru $n = 8$, slijedi da je za formiranje stabla neophodan sljedeći uvjet:

$$x_{1,2} + x_{1,4} + x_{2,3} + x_{2,6} + x_{3,4} + x_{3,5} + x_{4,5} + x_{4,7} + x_{5,6} + x_{5,7} + x_{6,8} + x_{7,8} = 7$$

Međutim, ovaj uvjet *nije dovoljan*, jer nema garancije da će bilo kojih $n-1$ grana formirati stablo (kad bismo imali samo ovaj uvjet, optimalno rješenje bi se sastojalo prosto od *skupa* 7 *najlakših grana*). Pored ovog uvjeta, neophodno je i da *izabrane grane razapinju čitav graf*, odnosno da je *svaki čvor incidentan sa makar jednom od izabranih grana*. To omogućava da napišemo sljedećih 8 ograničenja (zvaćemo ih *ograničenja za čvorove*), po jedno za svaki od čvorova:

$$x_{1,2} + x_{1,4} \geq 1$$
$$x_{1,2} + x_{2,3} + x_{2,6} \geq 1$$
$$x_{2,3} + x_{3,4} + x_{3,5} \geq 1$$
$$x_{1,4} + x_{3,4} + x_{4,5} + x_{4,7} \geq 1$$
$$x_{3,5} + x_{4,5} + x_{5,6} + x_{5,7} \geq 1$$
$$x_{2,6} + x_{5,6} + x_{6,8} \geq 1$$
$$x_{4,7} + x_{5,7} + x_{7,8} \geq 1$$
$$x_{6,8} + x_{7,8} \geq 1$$

Ni ovi uvjeti nisu dovoljni, jer *nema garancije da neke od izabranih grana neće formirati ciklus*. Stoga je potrebno dodati uvjete koji *sprečavaju formiranje ciklusa*. Nažalost, ovakvih uvjeta može biti jako mnogo. Naime, *niti u jednom podgrafu koji sadrži* $k < n$ *čvorova ne smije biti više od* $k-1$ *izabranih grana*, inače će se sigurno formirati ciklus (svaki graf sa k čvorova i više od $k-1$ grana *mora imati ciklus*). Pri tome, iz razmatranja očigledno možemo *isključiti* podgrafove sa k čvorova u kojima se *ne nalazi barem k grana*.

Krenimo sa $k = 3$. U razmatranom grafu postoje *samo dva podgrafa* sa 3 čvora, a koji sadrže barem tri grane. To su podgrafovi obrazovani od čvorova 3, 4 i 5, odnosno 4, 5 i 7. Za ova dva podgrafa uvjeti sprečavanja formiranja ciklusa glase

$$x_{3,4} + x_{3,5} + x_{4,5} \leq 2$$
$$x_{4,5} + x_{4,7} + x_{5,7} \leq 2$$

Za $k = 4$ već postoji 10 podgrafova koji sadrže barem četiri grane, pa samim tim i toliko novih ograničenja za sprećavanje formiranja ciklusa. Recimo, za podgrafove obrazovane od čvorova 1, 2, 3 i 4, zatim od čvorova 1, 3, 4 i 5, te od čvorova 3, 4, 5 i 7, odgovarajuća ograničenja glase

$$x_{1,2} + x_{1,4} + x_{2,3} + x_{3,4} \le 3$$
$$x_{1,4} + x_{3,4} + x_{3,5} + x_{4,5} \le 3$$
$$x_{3,4} + x_{3,5} + x_{4,5} + x_{4,7} + x_{5,7} \le 3$$

Možemo primijetiti da su neka od ovih ograničenja *suvišna*. Na primjer, u podgrafu obrazovanom od čvorova 1, 3, 4 i 5, nije moguće formirati *ni jedan ciklus* osim onaj u podgrafu obrazovanom od čvorova 3, 4 i 5, a taj ciklus smo *već isključili*. Međutim, ovakva suvišna ograničenja nije lako odmah uočiti bez obavljanja dublje analize strukture grafa i ovih ograničenja.

Sada bismo trebali nastaviti dalje za $k = 5$, $k = 6$ i, na kraju, za $k = 7$. Međutim, lako se vidi da su ograničenja za $k = 7$ zapravo *posljedica ograničenja za čvorove* i *početnog ograničenja na ukupan broj grana*, te se mogu izostaviti (alternativno, ograničenja za čvorove se mogu izostaviti ukoliko se uvrste ograničenja za $k = 7$). Konačno, treba još dodati i ograničenja $x_{i,j} \in \{0, 1\}$ za koje se pokazuje da se mogu zamijeniti *linearnim* ograničenjima $0 \le x_{i,j} \le 1$.

Sve u svemu, za razmotreni primjer dobijamo model linearnog programiranja koji *sadrži više desetina ograničenja*, iako se radi o *sasvim malom grafu* (sa 8 čvorova i 12 grana). Odavde se vidi *zbog čega su modeli linearnog programiranja loše sredstvo za modeliranje problema minimalnog povezujućeg stabla*. Da bismo vidjeli *koliko je situacija zapravo loša*, razmotrićemo i opći slučaj. Generaliziranjem gore provedenog postupka, lako možemo uočiti da se problem minimalnog povezujućeg stabla može opisati sljedećim modelom linearnog programiranja:

$$\arg\min Z = \sum_{(i,j)\in R} c_{i,j} x_{i,j}$$

p.o.

$$\sum_{(i,j)\in R} x_{i,j} = n - 1$$

$$\sum_{\substack{i\in S,\, j\in S \\ (i,j)\in R}} x_{i,j} \le k - 1 \quad \text{za sve } k\text{-člane podskupove S skupa } \{1, 2, ..., n\} \text{ za sve } k = 3 .. n-1$$

$$0 \le x_{i,j} \le 1, \quad (i,j) \in R$$

Iz skupine prethodnih ograničenja se očigledno mogu izbaciti ograničenja u kojima sume sa lijeve strane jednakosti nemaju barem k članova, jer su takve sume sigurno manje ili jednake od $k-1$ (zbog uvjeta $0 \le x_{i,j} \le 1$). Međutim, u slučaju da *niti jedno ograničenje ne možemo odbaciti* (što je tipičan slučaj u *gustim grafovima*), broj ograničenja raste *eksponencijalno* sa porastom n. Zaista, kako skup od n elemenata ima ukupno 2^n podskupova, nije teško prebrojati da ukupan broj ograničenja ukoliko se niti jedno ograničenje ne izbaci iznosi $2^n - n + 1$. Očigledno, zbog *velike brzine* kojom raste eksponencijalna funkcija, problemi sa ovolikim brojem ograničenja postaju *nesavladivi* za iole veće vrijednosti n. Zbog toga se problem minimalnog povezujućeg stabla *nikada* ne rješava kao problem linearnog programiranja, čak i ukoliko nam na raspolaganju nije ništa drugo osim nekog univerzalnog softvera za rješavanje općih problema linearnog programiranja. Međutim, dobijeni model je ipak interesantan zbog činjenice da se neki *teški problemi* teorije grafova, za koje *efikasni metodi rješavanja nisu poznati* (poput problema nalaženja *Hamiltonovog ciklusa* ili *problema trgovačkog putnika*), mogu modelirati koristeći slične ideje. Treba još istaći da je u slučaju *planarnih grafova*, koji *nikada nisu gusti* (s obzirom da se može pokazati da nijedan planaran graf sa n čvorova ne može imati više od $3n-6$ grana) moguće korištenjem specijalnih tehnika izvršiti *drastičnu redukciju* broja ograničenja i postići da broj ograničenja bude *linearna funkcija* broja čvorova n, što je već posve prihvatljivo.

Na kraju, treba napomenuti da mogućnost zamjene (nelinearnih) ograničenja $x_{i,j} \in \{0, 1\}$ (linearnim) ograničenjima $0 \le x_{i,j} \le 1$ slijedi iz *totalne unimodularnosti* matrice ograničenja koju formiraju ostala ograničenja. Ta totalna unimodularnost nije u ovom slučaju *nimalo očigledna*, a slijedi iz jednog rezultata teorije grafova poznatog kao ***Johnsonov teorem***. Nažalost, takva mogućnost zamjene *nije izvodljiva* u većini problema grafova koji spadaju u "teške" probleme teorije grafova.

Općenito o algoritmima za nalaženje minimalnog povezujućeg stabla

S obzirom da problem nalaženja minimalnog povezujućeg stabla predstavlja izuzetno važan problem kombinatorne optimizacije na grafovima, sretna je okolnost da i za ovaj problem postoje brojni efikasni algoritmi. Interesantno je da bez obzira što na fundamentalnom nivou ovaj problem *nema ništa zajedničko* sa

problemom najkraćeg puta, neki od algoritama za rješavanje problema minimalnog povezujućeg stabla su *zapanjujuće slični po formi* algoritmima za rješavanje problema najkraćeg puta. Međutim, bez obzira na upadljivu sličnost, ti algoritmi se oslanjaju na *posve drugačijim principima*, tako da je sličnost samo *formalna*.

Mada postoji veliki broj algoritama za rješavanje problema minimalnog povezujućeg stabla, danas se uglavnom koriste dva algoritma, poznati kao ***Primov algoritam*** i ***Kruskalov algoritam***. Izbor koji od ova dva algoritma koristiti zavisi od *gustoće grafa*, tako da je za *guste grafove bolje koristiti Primov algoritam*, a za *rijetke grafove Kruskalov algoritam*. Inače, oba ova algoritma spadaju u porodicu tzv. ***pohlepnih*** ili ***proždrljivih*** (engl. *greedy*) ***algoritama***, koji u svakom koraku vrše izbor one odluke koja *u tom trenutku najviše doprinosi krajnjem cilju*, ne vodeći računa kako će se taj izbor odraziti na *eventualne odluke koje treba donijeti u budućnosti*.

Pohlepni algoritmi u većini praktičnih problema *rijetko daju optimalno rješenje*, ali se za slučaj problema minimalnog povezujućeg stabla prilično jednostavno može dokazati da je kod njih *optimalna strategija upravo pohlepna strategija*. Rijetka je i "divna" situacija da se neki kombinatorni problem može riješiti jednostavnim algoritmom pohlepnog tipa. Situacija je još značajnija zbog činjenice da problem minimalnog povezujućeg stabla, posmatran kao kombinatorni problem, ima *izuzetno mnogo dopustivih rješenja*, koja ne *moraju biti optimalna*. Zaista, čak i mali grafovi mogu imati *izuzetno mnogo povezujućih stabala*. Konkretnije, prema ***Cayleyevoj teoremi*** iz teorije grafova, graf sa n čvorova može imati čak do n^{n-2} povezujućih stabala (recimo, 100000000 za $n = 10$). Ovo naravno u potpunosti isključuje kao moguće bilo kakve ideje rješavanja zasnovane na *ispitivanju svih mogućih povezujućih stabala*.

Primov algoritam se često navodi u brojnim udžbenicima, ali se u njima nažalost formulira na prilično nespretan način koji, ukoliko se direktno prevede u računarsku implementaciju, daje vrijeme izvršavanja reda n^3 gdje je n broj čvorova grafa. Međutim, činjenica je da se Primov algoritam može relativno lako preurediti u oblik koji *nevjerovatno podsjeća na Dijkstrin algoritam*. U tom slučaju, za njegovo vrijeme izvršavanja vrijedi sve što vrijedi i za Dijkstrin algoritam. Dakle, njegovo vrijeme izvršavanja je reda n^2 uz elementarnu izvedbu koja ne koristi napredne strukture podataka (poput redova sa prioritetom), dok se uz upotrebu sofisticiranih redova sa prioritetom (napredne izvedbe) vrijeme izvršavanja može u najboljem slučaju svesti na iznos približno proporcionalan sa $m + n \log_2 n$ (m je, kao i inače, broj grana grafa).

Kruskalov algoritam se smatra pogodnijim od Primovog algoritma za slučaj rijetkih grafova. Da bi bio efikasan, ovaj algoritam zahtijeva da *sve grane grafa* budu *sortirane u rastući redoslijed po težini*, ili *pohranjene u nekoj strukturi podataka koja omogućava jednostavan pristup granama u rastućem redoslijedu po težini* (npr. u nekoj strukturi podataka kao što je *hrpa* odnosno *gomila*). Ukoliko grane nisu sortirane po težini, potrebno ih je prethodno *sortirati*. Ovo sortiranje je, uz upotrebu *dobrih algoritama za sortiranje* (*Quick Sort*, *Merge Sort*, *Heap Sort*) moguće obaviti u vremenu proporcionalnom sa $m \log_2 m$ u najgorem slučaju. Za sortiranje se nipošto *ne smiju koristiti jednostavni algoritmi za sortiranje* (poput *Bubble sort*, *Select Sort*, itd.) zbog *neprihvatljivo dugog trajanja sortiranja* (reda m^2). Zapravo, ukoliko se koriste loši postupci sortiranja, perfomanse algoritma postaju *lošije* nego da se *sortiranje uopće nije koristilo*, jer se *previše vremena izgubi na sortiranje*.

Iz izloženog slijedi da je sortiranje *osnovna predradnja* za potrebe Kruskalovog algoritma. Ova predradnja, kao što je već rečeno, izvršava se u vremenu reda $m \log_2 m$. Što se tiče *ostatka algoritma*, u literaturi se često navodi da se i ostatak algoritma može izvesti u vremenu reda $m \log_2 m$, što znači da se čitav algoritam također izvodi u vremenu proporcionalnom sa $m \log_2 m$ u najgorem slučaju. Ovo je istina, ali se u literaturi *rijetko opisuje* kako postići da se i ostatak Kruskalovog algoritma nakon sortiranja zaista obavi u ovom vremenu. Većina opisa Kruskalovog algoritma koje se mogu pronaći u literaturi zahtijevaju nakon sortiranja vrijeme koje je u najgorem slučaju proporcionalno sa n^2, tako da uz takve izvedbe Kruskalov algoritam *ne može nikada biti efikasniji od Primovog algoritma* (misli se na razumne izvedbe Primovog algoritma čije vrijeme izvršavanja ne prelazi red n^2). Dakle, usprkos rasprostranjenom mišljenju, *naivne izvedbe Kruskalovog algoritma nisu brže od Primovog algoritma*, čak ni za rijetke grafove. Drugim riječima, potrebna je izvjesna domišljatost u implementaciji Kruskalovog algoritma da bi on postao brži od Primovog algoritma za rijetke grafove. Također, odmah se vidi da je Kruskalov algoritam *nepodesan za guste grafove*. Naime, kod njih je m reda n^2, pa bi se za samo sortiranje potrošilo više vremena nego što bi trajao čitav Primov algoritam.

Treba još istaći da se problem minimalnog povezujućeg stabla tipično definira samo za neusmjerene grafove. Ukoliko se govori o minimalnom povezujućem stablu za slučaj usmjerenih grafova, usmjerenja grana se *ignoriraju*, odnosno razmatra se minimalno povezujuće stablo za odgovarajući *neusmjereni graf*. Stoga ćemo u nastavku podrazumijevati da radimo sa *neusmjerenim grafovima*, odnosno grafovima čija su matrica susjedstva i težinska matrica *simetrične matrice*.

Primov algoritam

Primov algoritam počinje od proizvoljnog čvora grafa i postepeno gradi stablo dodajući u svakoj iteraciji *jednu po jednu granu* (sa pripadajućim čvorovima), pri čemu nakon svake iteracije izabrane grane formiraju *stablo koje predstavlja dio traženog minimalnog povezujućeg stabla*. Ova stabla se tipično nazivaju ***fragmenti***. Pri tome se grane biraju na *pohlepan način*, odnosno izborom grane *najmanje težine* od svih grana koje se mogu uzeti da se proširi fragment. U mnogim udžbenicima, Primov algoritam se (nažalost) opisuje kao sljedeći niz koraka:

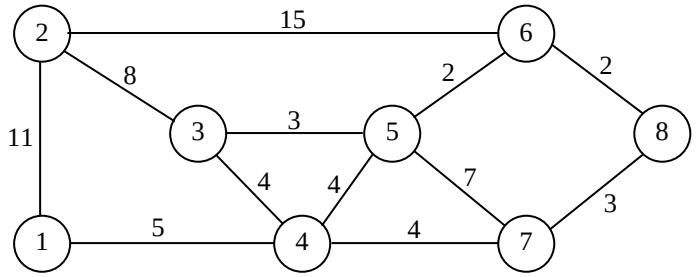
1. Izabrati proizvoljan početni čvor (recimo, čvor 1). Na početku se fragment sastoji samo od tog čvora.
2. Od svih grana koje spajaju neki čvor u fragmentu sa nekim čvorom izvan fragmenta izabrati granu *najmanje težine* i dodati je u fragment (zajedno sa pripadnim čvorovima).
3. Ukoliko još ima čvorova koji nisu ušli u fragment, vratiti se na korak 2. U suprotnom, formirani fragment je minimalno povezujuće stablo.

Ovaj algoritam se vrlo jednostavno može izraziti i u formi u kojoj se obrada izvodi *direktno nad težinskom matricom grafa* **C**:

1. Izabrati proizvoljan red matrice (recimo, red 1). Obilježiti na neki način (recimo strelicom) taj red i odgovarajuću kolonu matrice **C** (tj. prvi red i prvu kolonu).
2. Od svih elemenata matrice **C** koji se nalaze u presjeku *obilježenih redova* i *neobilježenih kolona* pronaći *najmanji element*. Neka je taj element u koloni *k*. Obilježiti taj element, te obilježiti (strelicom) *k*-ti red i *k*-tu kolonu.
3. Ukoliko još ima neobilježenih redova, vratiti se na korak 2. U suprotnom, obilježeni elementi matrice **C** odgovaraju granama minimalnog povezujućeg stabla.

Ovaj algoritam je očigledno *vrlo jednostavan i veoma se lako izvodi*. Problem sa ovakvom verzijom Primovog algoritma (zvaćemo je *naivna verzija*) je što korak 2 očito traži vrijeme reda n^2 u najgorem slučaju. Kako je dalje očigledno da u ovom algoritmu ima n iteracija, to daje ukupno vrijeme izvršavanja reda n^3, što je neprihvatljivo mnogo. Mnogi udžbenici nažalost prikazuju *samo ovu verziju algoritma* i, što je još gore, uz napomenu da Primov algoritam ima vrijeme izvršavanja reda n^2 (što u principu jeste tačno, ali *nije tačno za ovakvu verziju algoritma*). Stoga se ova verzija algoritma (zbog jednostavnosti) koristi uglavnom *za posve male grafove*, i to uglavnom *pri ručnom radu*. Prije nego što vidimo kako se Primov algoritam može reorganizirati da radi *brže*, demonstriraćemo ovaj algoritam na jednom primjeru.

- ***Primjer***: Koristeći naivnu verziju Primovog algoritma, pronaći minimalno povezujuće stablo za graf sa sljedeće slike (ovo je zapravo graf koji odgovara telefonskoj mreži koju smo ranije razmatrali):



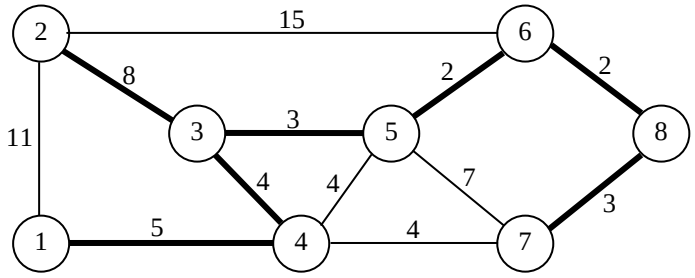
Težinska matrica za ovaj graf glasi:

$$\mathbf{C} = \begin{pmatrix} 0 & 11 & \infty & 5 & \infty & \infty & \infty & \infty \\ 11 & 0 & 8 & \infty & \infty & 15 & \infty & \infty \\ \infty & 8 & 0 & 4 & 3 & \infty & \infty & \infty \\ 5 & \infty & 4 & 0 & 4 & \infty & 4 & \infty \\ \infty & \infty & 3 & 4 & 0 & 2 & 7 & \infty \\ \infty & 15 & \infty & \infty & 2 & 0 & \infty & 2 \\ \infty & \infty & \infty & 4 & 7 & \infty & 0 & 3 \\ \infty & \infty & \infty & \infty & \infty & 2 & 3 & 0 \end{pmatrix}$$

Tok izvršavanja algoritma prikazaćemo u sljedećoj tablici (koja odgovara matrici **C**), pri čemu je radi lakšeg praćenja kraj strelica naznačeno u kojoj su iteraciji obilježen odgovarajući red i kolona:

	1	2	3	4	5	6	7	8	
1	0	11	∞	**5**	∞	∞	∞	∞	← 1
2	11	0	8	∞	∞	15	∞	∞	← 8
3	∞	**8**	0	4	**3**	∞	∞	∞	← 3
4	5	∞	**4**	0	4	∞	4	∞	← 2
5	∞	∞	3	4	0	**2**	7	∞	← 4
6	∞	15	∞	∞	2	0	∞	**2**	← 5
7	∞	∞	∞	4	7	∞	0	3	← 7
8	∞	∞	∞	∞	∞	2	**3**	0	← 6
	↑	↑	↑	↑	↑	↑	↑	↑	
	1	8	3	2	4	5	7	6	

Traženo minimalno povezujuće stablo odgovara *osjenčenim poljima u tabeli*, odnosno čine ga grane $(1,4)$, $(3,2)$, $(3,5)$, $(4,3)$, $(5,6)$, $(6,8)$ i $(8,7)$, kao što je prikazano na sljedećoj slici. Traženo minimalno povezujuće stablo ima ukupnu težinu $5+8+3+4+2+2+3=27$.



Sad ćemo vidjeti kako se, uz pomoć jednostavnih dosjetki, Primov algoritam može *ubrzati skoro sa faktorom n*. Uvedimo veličine f_j za čvorove *izvan fragmenta* koji predstavljaju dužinu *najkraće grane* koja ih spaja sa *nekim od čvorova unutar fragmenta* i pripadne oznake p_j koje govore *koji je to čvor unutar fragmenta* do kojeg vodi ta najkraća grana. Tada se grana sa najmanjom težinom od svih grana koje spajaju neki čvor u fragmentu sa nekim čvorom izvan fragmenta može odrediti prosto kao (p_k, k) gdje je k čvor izvan fragmenta kojem odgovara *najmanja vrijednost* f_j (ovo se može obaviti u vremenu u najgorem slučaju proporcionalnom sa n). Dalje, svaki put kada u fragment uđe novi čvor r (zvaćemo taj čvor *referentnim čvorom*), treba eventualno korigirati vrijednosti f_j za *sve njegove susjede* (ukoliko je vrijednost $c_{r,j}$ manja od f_j, nova vrijednost za f_j treba postati upravo $c_{r,j}$, u suprotnom se f_j ne mijenja), dok se vrijednosti f_j za ostale čvorove *neće promijeniti* (za ove korekcije isto u najgorem slučaju treba vrijeme proporcionalno sa n). Ostaje još samo pitanje *početnih vrijednosti* za f_j. Sasvim je moguće na početku postaviti $f_j = \infty$, pa će nakon što prvi čvor postane referentni, postupak korekcije f_j za njegove susjede dati ispravne početne vrijednosti za sve njegove susjede. Slijedi da se modificirana verzija Primovog algoritma može predstaviti sljedećim nizom koraka:

1. Postaviti $f_1 = 0$ i $f_j = \infty$ za $j = 2 \,..\, n$.
2. Proglasiti čvor 1 za referentni čvor ($r = 1$).
3. Proglasiti čvor r za element fragmenta. Ukoliko su svi čvorovi postali elementi fragmenta, preći na korak 6.
4. Za sve susjede j referentnog čvora r koji još uvijek *nisu u fragmentu* obaviti algoritamski korak $f_j \leftarrow \min\{f_j, c_{r,j}\}$. Drugim riječima, ukoliko je vrijednost $c_{r,j}$ manja od tekuće vrijednosti f_j, prepraviti f_j na ovu novu vrijednost. Također, svaki put kada se ova prepravka izvrši za neki čvor j, *zapamtiti* (recimo u nekoj promjenljivoj p_j) *koji je bio referentni čvor* kada je ta popravka vršena, tj. staviti $p_j \leftarrow r$). Ove informacije će nam biti potrebne da na kraju očitamo traženo minimalno povezujuće stablo.
5. Od svih čvorova koji još nisu ušli u fragment, pronaći čvor k sa *najmanjom vrijednošću* f_j. Proglasiti čvor k za novi referentni čvor ($r \leftarrow k$) i vratiti se na korak 3.
6. Traženo minimalno povezujuće stablo je konstruisano. Ovo stablo sastoji se od grana (p_j, j) za sve $j = 2 \,..\, n$.

Već na prvi pogled može se primijetiti *upadljiva sličnost* sa *Dijkstrinim algoritmom*. Ako zanemarimo razlike koje su prvenstveno *terminološke prirode* (recimo, čvorovima u Dijkstrinom algoritmu čiji je "potencijal dobio konačnu vrijednost" u Primovom algoritmu odgovaraju čvorovi "koji su postali elementi

fragmenta"), jedina bitna razlika je u načinu kako se izvodi korak 4. Naime, u Dijkstrinom algoritmu javlja se algoritamski korak $f_j \leftarrow \min\{f_j, f_r + l_{r,j}\}$, dok se u Primovom algoritmu umjesto njega javlja neznatno prostiji korak $f_j \leftarrow \min\{f_j, c_{r,j}\}$. Osim ovog sitnog detalja, ova dva algoritma su *praktično identična*. Stoga ne treba da čudi da se implementacije Primovog i Dijkstrinog algoritma tipično razlikuju *samo u jednoj liniji koda*. Treba ipak naglasiti da veličine f_j po završetku Dijkstrinog algoritma itekako imaju svoje značenje (dužine najkraćih puteva od čvora 1 do čvora j), dok u Primovom algoritmu veličina f_j za neki čvor j *gubi svaki smisao* onog trenutka kada čvor j postane element fragmenta.

- ***Primjer***: Koristeći poboljšanu verziju Primovog algoritma, pronaći minimalno povezujuće stablo za graf iz prethodnog primjera.

Slično kao kod Dijkstrinog algoritma, tok algoritma najbolje je prikazati *u tabelarnoj formi* (i ovdje vrijedi da se u slučaju kada graf nije previše gust i nepregledan, odgovarajuće vrijednosti f_i i p_i obično bilježe *direktno na crtežu*, kraj čvorova grafa):

Ref. čvor (r)	1	2	3	4	5	6	7	8
	0	∞	∞	∞	∞	∞	∞	∞
1		11 (1)	∞	**5 (1)**	∞	∞	∞	∞
4		11 (1)	**4 (4)**		4 (4)	∞	4 (4)	∞
3		8 (3)			**3 (3)**	∞	4 (4)	∞
5		8 (3)				**2 (5)**	4 (4)	∞
6		8 (3)					4 (4)	**2 (6)**
8		8 (3)					**3 (8)**	
7		**8 (3)**						

Pri svakom novom referentnom čvoru r, vrijednosti $c_{r,j}$ porede se sa aktuelnom vrijednošću f_j. Ukoliko je vrijednost $c_{r,j}$ manja od aktuelne vrijednosti f_j, u polje se upisuje *nova vrijednost* $c_{r,j}$, zajedno sa odgovarajućim $p_j = r$ (prikazano u zagradi), inače se aktuelne vrijednosti f_j i p_j prosto *prepisuju*. Svijetlo osjenčeno su prikazana polja koja odgovaraju izvršenim *promjenama*, dok su tamno osjenčeno i podebljano prikazana polja u kojima su *čvorovi postali dio fragmenta*. Postupak se *završava* kada *svi čvorovi postaju dio fragmenta*.Traženo minimalno povezujuće stablo očitavamo iz osjenčenih polja kao grane oblika (p_j, j) za svako tamno osjenčeno polje, odnosno minimalno povezujuće stablo čine grane $(1,4)$, $(3,2)$, $(3,5)$, $(4,3)$, $(5,6)$, $(6,8)$ i $(8,7)$.

Na kraju, treba naglasiti da bez obzira na izuzetno veliku *izvedbenu sličnost* Primovog i Dijkstrinog algoritma, na suštinskom nivou ova dva algoritma *gotovo da nemaju ništa zajedničko*. Recimo, Primov algoritam je klasični princip *pohlepnog metoda*, dok Dijkstrin algoritam to *nije*. Dalje, Primov algoritam se *ne zasniva* na Bellmanovom principu optimalnosti, pa samim tim *nema nikakvih dodirnih tačaka ni sa dinamičkim programiranjem*. Pored toga, veličine f_j kod Primovog algoritma služe samo za *ubrzavanje algoritma* i gube svaki smisao onog trenutka kada odgovarajući čvor uđe u fragment, dok su ove veličine *od fundamentalnog značaja* za sam rad Dijkstrinog algoritma i imaju svoje značenje i po završetku algoritma. Na kraju, za razliku od Dijsktinog algoritma, Primov algoritam *ne spada u primalno-dualne metode*, niti veličine f_j imaju *ikakve veze sa promjenljivim dualnog problema za problem minimalnog povezujućeg stabla*.

Kruskalov algoritam

Kruskalov algoritam je još jedan algoritam *pohlepnog tipa* za pronalaženje minimalnog povezujućeg stabla koji je konceptualno *jednostavniji* od Primovog, i uz dobru implementaciju (koja baš nije očigledna iz samog opisa algoritma) može biti *brži od njega* za slučaj *rijetkih grafova*. Za razliku od Primovog algoritma koji u svakoj iteraciji uzima *najlakšu granu koja se može povezati sa do tada uzetim granama* (tako da u svakoj iteraciji do tada uzete grane obrazuju stablo koje smo nazivali fragmentom), Kruskalov algoritam prosto u svakoj iteraciji bira *najlakšu granu u cijelom grafu koja do tada nije uzeta*, bez obzira da li će ona biti povezana sa do tada uzetim granama ili ne. Jedini uvjet koji se postavlja je da grana koju uzimamo *ne smije obrazovati konturu sa do tada uzetim granama*. Algoritam završava nakon što uzmemo $n-1$ grana, koliko je neophodno da se formira povezujuće stablo.

Mada je Kruskalov algoritam principijelno vrlo jednostavan, prilikom njegove praktične implementacije postoje dva problema koja treba razmotriti. Prvi problem je *kako efikasno birati granu sa najmanjom težinom* u svakoj od $n-1$ iteracija. Razmotrimo prvo prvi problem. Ukoliko koristimo naivni pristup, granu sa najmanjom težinom možemo naći u m koraka, gdje je m broj grana. Sljedeću granu sa najmanjom težinom

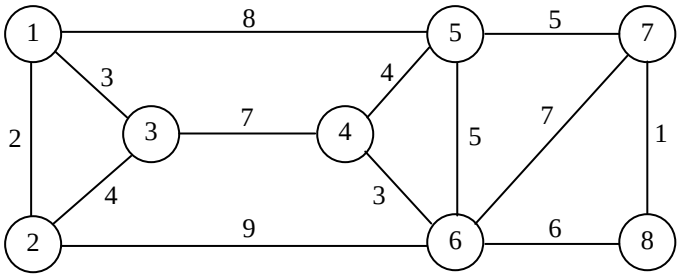
možemo naći u $m-1$ koraka (s obzirom da smo jednu granu već iscrpili), sljedeću u $m-2$ koraka, itd. Međutim, treba voditi računa da nećemo u svakoj iteraciji imati sreće da odmah možemo uzeti prvu pronađenu granu, s obzirom da ona možda formira konturu sa dotada izabranim granama. Drugim riječima, moramo biti spremni da ćemo u pojedinim iteracijama morati razmatrati *čitavo mnoštvo grana*. Nije preteško zaključiti da ćemo u najgorem slučaju do završetka algoritma morati razmotriti sve grane grafa. Naime, može se desiti da će u posljednjoj iteraciji u stablo ući *grana sa najvećom težinom* u čitavom grafu (recimo, to će se sigurno desiti ukoliko je ta grana takva da bi njeno izbacivanje *razdvojilo graf na dva nepovezana dijela*), što znači da smo prije nje sigurno morali razmotriti *sve ostale grane*. Dakle, samo za potrebe pronalaženja grana sa najmanjom težinom koje smijemo uzeti u svakoj od iteracija potrebno je u najgorem slučaju utrošiti ukupno $m+(m-1)+(m-2)+\ldots+1=m(m+1)/2$ koraka (tj. približno $m^2/2$ koraka). Situacija je još gora ukoliko bismo algoritam *direktno izvodili nad težinskom matricom*, jer bi nam tada za traženje svake grane trebalo m koraka, s obzirom da ne možemo lako prosto "preskočiti" polja koja odgovaraju do tada uzetim granama. Slijedi da bi nam, u najgorem slučaju, samo na pronalaženja najlakših grana u toku kompletnog algoritma trebalo ukupno vrijeme reda m^2, što je *nedopustivo mnogo*. Generalno, težinska matrica *nije pogodna reprezentacija grafa* za potrebe Kruskalovog algoritma, tako da *ne treba ni pokušavati implementirati Kruskalov algoritam oslanjajući se na operacije nad težinskom matricom*.

Da bismo riješili ovaj problem, grane grafa je potrebno izdvojiti u *poseban spisak*, a zatim izvršiti sortiranje ovog spiska grana *po težinama*. Naime, primjenom naprednih tehnika sortiranja sortiranje m grana može obaviti u približno $m\log_2 m$ koraka, što je za veliko m znatno manje od $m^2/2$, a nakon sortiranja izbor grane sa najmanjom težinom u svakoj iteraciji nalazimo praktično u jednom koraku. Bitno je naglasiti da se *ništa ne postiže* ukoliko se sortiranje izvede nekim *neefikasnim algoritmom*, kod kojeg je broj koraka za sortiranje proporcionalan sa m^2. Naprotiv, u tom slučaju dobijaju se *lošije perfomanse* nego da je primijenjen naivni pristup. Kao *alternativa sortiranju*, moguće je sve grane ubaciti u strukturu podataka poznatu kao *hrpa* odnosno *gomila*. Zaista, ova struktura podataka omogućava da se u *jednom koraku pronađe najlakša grana*, nakon čega se ta grana može *ukloniti* iz hrpe u vremenu proporcionalnom sa $\log_2 m$. Ukoliko smo testirali ukupno k grana do završetka algoritma, ukupno vrijeme koje ćemo za to potrošiti je proporcionalno sa $k\log_2 m$. Ukoliko je k mnogo manje od m, ovo može biti manje od vremena koje bi se potrošilo na sortiranje. Zbog toga ovaj pristup vrijedi koristiti ukoliko imamo razloga da vjerujemo da će se algoritam završiti *znatno prije nego što se testiraju sve grane* (što opet ovisi od *prirode grafa* koji se razmatra).

Drugi, znatno delikatniji problem koji trebamo riješiti je *kako efikasno utvrditi da li izabrana grana formira konturu sa do tada izabranim granama ili ne*, s obzirom da problem utvrđivanja postojanja konture u grafu sam po sebi nije posve elementaran. Za testiranje prisustva kontura postoje brojni algoritmi zasnovani na *pretragama grafa* (poput *BFS pretrage*), ali obavljanje *kompletnih testova* ovog tipa svaki put kada uzmemo novu granu je *isuviše neefikasno*. Stoga je osnovna ideja da problem testiranja da li grana formira konturu ili ne u svakoj iteraciji ne treba rješavati *nezavisno od onoga što znamo od ranije*, već da trebamo *maksimalno iskoristiti informacije koje se mogu prikupiti u prethodnim iteracijama*. Postoji mnogo načina da se ovaj postupak izvede efikasno. Najpoznatiji ali ne i najefikasniji načini zasnivaju se na ideji nazvanoj ***bojenje čvorova***. Postoje također i mnogi načini da se ova ideja izvede, a vjerovatno najefikasniji od njih je sljedeći. Svakom čvoru se pridruži jedna brojčana vrijednost nazvana ***boja*** ili ***klasa čvora***. Na početku se svim čvorovima pridruži vrijednost 0, koja označava "odsustvo boje" (tj. svi čvorovi su *bezbojni* na početku). Ideja je da u svakom trenutku čvorovi koji su do tada međusobno povezani do tada uzetim granama budu *iste boje*. Stoga, granu (i, j) smijemo uzeti ili ako je makar jedan od čvorova bezbojan (što bi značilo da on trenutno nije povezan uzetim granama niti sa jednim drugim čvorom), ili ukoliko su im boje različite. Nakon uzimanja grane koja spaja neka dva čvora, vršimo korekciju njihovih boja na sljedeći način. Ukoliko su *oba čvora bezbojna*, dodijelimo im *boju koja je za* 1 *veća od najveće do tada korištene boje*. Ukoliko je samo jedan od ta dva čvor bezbojan, dodijelimo mu *istu boju kao što je boja onog drugog čvora*. Ukoliko su oba čvora obojena, tada *svim čvorovima grafa koji imaju istu boju kao jedan od njih* dodijelimo *istu boju kao boja onog drugog čvora*.

Opisani postupak je elegantan i jednostavan, ali u njemu ipak postoji jedan problem. Naime, ukoliko su prilikom uzimanja grane (i, j) oba čvora i i j obojena, moguće je da će se morati korigirati boje dosta čvorova, nekada skoro i n čvorova. Kako ukupno uzimamo $n-1$ grana, to može dovesti da vrijeme izvršavanja ovog dijela algoritma bude reda n^2, pa je nemoguće postići ukupno vrijeme izvršavanja reda $m\log_2 m$. Međutim, prije nego što vidimo kako se Kruskalov algoritam može još poboljšati, demonstrirajmo prvo ovu verziju algoritma na jednom primjeru.

- ***Primjer***: Pomoću Kruskalovog algoritma uz bojenje čvorova, naći minimalno povezujuće stablo u grafu sa slike:



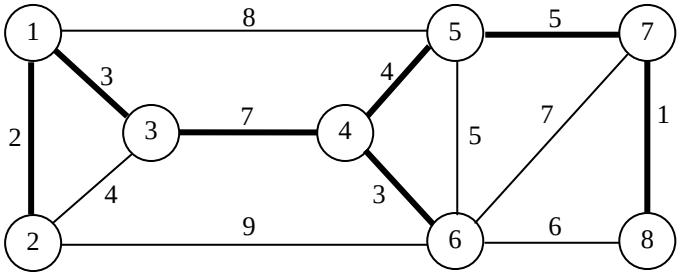
Za potrebe Kruskalovog algoritma ne vrijedi ni sastavljati težinsku matricu, nego treba krenuti odmah od *spiska grana*. Na početku, ovaj spisak grana treba *sortirati u rastući redoslijed po težinama grana*, čime se dobija sljedeći spisak:

$$(7,8),\ (1,2),\ (1,3),\ (4,6),\ (2,3),\ (4,5),\ (5,6),\ (5,7),\ (6,8),\ (3,4),\ (6,7),\ (1,5),\ (2,6)$$

Dalji tok postupka prikazan je u sljedećoj tabeli, pri čemu je za svaku granu prikazano da li se uzima ili ne, kao i boje čvorova nakon uzimanja svake od grana:

Grana	Uzeti	1	2	3	4	5	6	7	8
(7, 8)	da	0	0	0	0	0	0	1	1
(1, 2)	da	2	2	0	0	0	0	1	1
(1, 3)	da	2	2	2	0	0	0	1	1
(4, 6)	da	2	2	2	3	0	3	1	1
(2, 3)	ne	2	2	2	3	0	3	1	1
(4, 5)	da	2	2	2	3	3	3	1	1
(5, 6)	ne	2	2	2	3	3	3	1	1
(5, 7)	da	2	2	2	3	3	3	3	3
(6, 8)	ne	2	2	2	3	3	3	3	3
(3, 4)	da	3	3	3	3	3	3	3	3

Slijedi da minimalno povezujuće stablo obrazuju grane $(7,8)$, $(1,2)$, $(1,3)$, $(4,6)$, $(4,5)$, $(5,7)$ i $(3,4)$, kao što je prikazano na sljedećoj slici. Nađeno minimalno povezujuće stablo ima ukupnu težinu $1+2+3+3+4+5+7=25$.



Za ubrzanje Kruskalovog algoritma je ključno primijetiti da se često *mnogo vremena gubi na korigiranje boja čvorova*. Stoga ćemo umjesto boja čvorova čvorovima pridružiti *neke druge dopunske informacije*, koje su takve da je njihovo ažuriranje praktično *trivijalno*, a koje i dalje omogućavaju *relativno jednostavnu detekciju* da li grana obrazuje konturu sa dotada uzetim granama. Konkretnije, svakom čvoru i ćemo pridružiti dvije dopunske informacije: oznaku njemu ***referentnog čvora*** r_i i ***težinu čvora*** w_i. Na početku označavamo da *nijedan čvor nema referentnog čvora* i da mu je težina 1 (tj. postavljamo $r_i = 0$ i $w_i = 1$ za sve $i = 1 \,..\, n$. Da bismo testirali da li smijemo uzeti granu (i, j) ili ne, moramo prvo naći tzv. ***korijenske čvorove*** za čvorove i i j, koje ćemo označiti sa p i q. Da bismo našli korijenski čvor za neki čvor, prvo gledamo *njegov referentni čvor* (ako takvog ima), zatim *referentni čvor njegovog referentnog čvora* i tako sve dok ne dođemo do čvora koji *nema referentnog čvora*, koji predstavlja *traženi korijenski čvor*. Sada, granu koja spaja čvorove i i j smijemo uzeti *ako i samo ako su njihovi korijenski čvorovi različiti* (tj.

ukoliko je $p \neq q$). Nakon što eventualno uzmemo granu, korekcija se vrši samo nad čvorovima p i q prema sljedećem postupku. Onom od ta dva čvora koji ima *manju težinu* postavljamo *onaj drugi čvor za referentni*, dok tom drugom *uvećavamo težinu* za iznos *težine onog čvora sa manjom težinom* (ukoliko oba imaju istu težinu, svejedno je koji ćemo od njih posmatrati kao da ima manju težinu). Drugim riječima, ukoliko je $w_p < w_q$, vršimo korekcije $r_p \leftarrow q$ i $w_q \leftarrow w_q + w_p$, a u suprotnom vršimo korekcije $r_q \leftarrow p$ i $w_p \leftarrow w_p + w_q$.

Sasvim je lako uvjeriti se u korektnost opisanog postupka, jer je suština svega da će *svi čvorovi koji su do tada povezani uzetim granama imati isti korijenski čvor*. Pri tome, smisao težina čvorova je u tome da one predstavljaju ukupan broj čvorova koji direktno ili indirektno upućuju na posmatrani čvor (računajući i njega samog). Činjenica da uvijek korijenskom čvoru manje težine postavljamo drugi čvor veće ili jednake težine za referentni *garantira da broj koraka potreban da se pronađe korijenski čvor nekog čvora ne može biti veći od* $\log_2 n$. Naime, matematskom indukcijom se može dokazati da ukoliko je za neki čvor potrebno h koraka da se pronađe referentni čvor, taj referentni čvor mora imati težinu barem 2^h. Kako je jasno da težina niti jednog čvora ne može biti veća od n, slijedi da h ne može biti veći od $\log_2 n$. Stoga, čak i u najgorem slučaju kada je potrebno obraditi svih m grana prije nego što se formira kompletno povezujuće stablo, ukupan broj operacija koje je potrebno obaviti nakon sortiranja ne prelazi iznos proporcionalan sa $m \log_2 n$. Kako je broj operacija za sortiranje reda $m \log_2 m$ i $m > n$, to ukupno vrijeme izvršavanja čitavog algoritma neće preći iznos proporcionalan sa $m \log_2 m$.

- ***Primjer***: Za graf iz prethodnog primjera, naći minimalno povezujuće stablo u grafu sa slike koristeći poboljšanu verziju Kruskalovog algoritma.

Krenućemo od istog sortiranog spiska grana kao u prethodnom primjeru. Dalji tok postupka prikazan je u sljedećoj tabeli, pri čemu je ponovo za svaku granu prikazano da li se uzima ili ne, kao i oznaka referentnog čvora i težina svakog čvora nakon uzimanja svake od grana (vrijednosti koje stoje u zagradama su težine čvorova):

Grana	Uzeti	1	2	3	4	5	6	7	8
(7, 8)	da	0 (1)	0 (1)	0 (1)	0 (1)	0 (1)	0 (1)	0 (2)	7 (1)
(1, 2)	da	0 (2)	1 (1)	0 (1)	0 (1)	0 (1)	0 (1)	0 (2)	7 (1)
(1, 3)	da	0 (3)	1 (1)	1 (1)	0 (1)	0 (1)	0 (1)	0 (2)	7 (1)
(4, 6)	da	0 (3)	1 (1)	1 (1)	0 (2)	0 (1)	4 (1)	0 (2)	7 (1)
(2, 3)	ne	0 (3)	1 (1)	1 (1)	0 (2)	0 (1)	4 (1)	0 (2)	7 (1)
(4, 5)	da	0 (3)	1 (1)	1 (1)	0 (3)	4 (1)	4 (1)	0 (2)	7 (1)
(5, 6)	ne	0 (3)	1 (1)	1 (1)	0 (3)	4 (1)	4 (1)	0 (2)	7 (1)
(5, 7)	da	0 (3)	1 (1)	1 (1)	0 (5)	4 (1)	4 (1)	4 (2)	7 (1)
(6, 8)	ne	0 (3)	1 (1)	1 (1)	0 (5)	4 (1)	4 (1)	4 (2)	7 (1)
(3, 4)	da	4 (3)	1 (1)	1 (1)	0 (8)	4 (1)	4 (1)	4 (2)	7 (1)

Vidimo da smo došli do istih zaključaka kao i korištenjem tehnike zasnovane na bojenju čvorova, što je uostalom bilo i očekivano.

Na kraju, treba još napomenuti da ukoliko želimo da Kruskalov algoritam implementiramo u vremenu reda $m \log_2 m$, ulaz u algoritam *ni u kom slučaju ne smije biti težinska matrica*. Naime, ukoliko bi ulaz u algoritam bio težinska matrica, tada bi se *samo za izdvajanje informacija o svim granama* u spisak grana koji treba sortirati *potrošilo vrijeme koje je reda* n^2.