§2. TEHNIKE ZA KONSTRUKCIJU ALGORITAMA

2.1. Pohlepni (greedy) algoritmi

Pohlepni algoritmi su obično jednostavnog oblika, pa zato i počinjemo s njima.

Koriste se, uglavnom, za rješavanje problema optimizacije, na pr:

- naći najkraći put u grafu
- naći najbolji redoslijed izvođenja nekog skupa poslova na računalu i sl.

U uobičajenoj situaciji nalazimo slijedeće karakteristične elemente:

- skup (niz, listu, popis) kandidata (bridovi ili grane nekog grafa, poslovi koje treba izvesti);
- skup kandidata koje smo već iskoristili;
- funkciju koja provjerava da li odredeni skup kandidata daje (predstavlja) rješenje problema, ignorirajući pitanje optimalnosti (bar za sada). [tj. odgovor je da/ne – boolean function];
- funkciju koja provjerava da li je odredeni skup kandidata dopustiv (feasible), tj. da li je moguće taj skup dopuniti tako da dobijemo bar jedno rješenje (ne nužno optimalno) za naš problem.
  Obično pretpostavljamo da problem ima bar jedno rješenje sastavljeno iz kandidata u početnom skupu;
- funkciju izbora (selekcije), koja u svakom trenutku daje najperspektivnijeg, još neiskorištenog kandidata;
- funkciju cilja (objective function) koja daje vrijednost rješenja. To je funkcija koju optimiziramo (duljina puta kojeg smo našli, vrijeme potrebno za izvođenje poslova u danom nizu i sl.)

Rješavanje problema optimizacije =
~~traženje~~ skupa kandidata koji predstavlja ~~rješenje~~ problema i optimizira (minimizira ili maksimizira) vrijednost funkcije cilja.

- Pohlepni algoritam napreduje korak po korak:
  - Na početku, skup izabranih kandidata je prazan. ($S=\emptyset$), a zadan je skup C svih raspoloživih kandidata.
  - U svakom koraku, skupu S pokušavamo dodati najboljeg preostalog kandidata. Izbor kandidata diktira funkcija izbora.
  - Ako tako povećani skup izabranih kandidata nije više dopustiv, izbacujemo upravo dodanog kandidata. Tog odbačenog kandidata nikada više ne provjeravamo.
    [Tj. svaki kandidat može biti provjeren samo jednom]
  - Ako je povećani skup kandidata dopustiv, onda taj kandidat ostaje među izabranim kandidatima.
  - Svaki put kad povećamo skup izabranih kandidata, provjeravamo da li taj skup predstavlja i rješenje našeg problema.
  - Ako da – stajemo, u protivnom – idemo na novog kandidata, ako postoji.

Algoritam možemo ovako zapisati u općem obliku:

```
procedure greedy (C: skup; var S: skup;
                             var OK: boolean);
    { C je skup svih raspoloživih kandidata }
begin
  S ← ∅; { S je skup u kom akumuliramo rješenje }
  while not rješenje (S) and (C ≠ ∅) do
    begin                                funkciju
      x ← element iz C koji maksimizira izbor(x);
      C ← C \ {x};
      if dopustiv (S ∪ {x}) then S ← S ∪ {x};
    end;
  OK ← rješenje (S);
end; { greedy }.
```

Pohlepni algoritam ne mora uopće naći rješenje, ili nađeno rješenje ne mora biti optimalno.

Tek ako dokažemo da pohlepni algoritam korektno rješava naš problem optimizacije, onda je prvo nađeno rješenje i optimalno.

"Pohlepa" – u svakom koraku, postupak bira najbolji (najveći) zalogaj koji može progutati, ne vodeći računa o budućnosti (ili prošlosti).
- Nikad ne mijenja odluku – kad je kandidat jednom uključen u rješenje, on tamo i ostaje (na dobro ili zlo). Ako je kandidat jednom odbačen, to je zauvijek (nikada ga više ne provjeravamo).
- Zbog toga su pohlepni algoritmi BRZI (svaki kandidat se provjerava najviše jednom).
  Često se koriste za rješavanje teških problema optimizacije, poput TSP, kao brza heuristika koja daje neko rješenje – iako ne optimalno. Također, mogu se koristiti za start iterativnih metoda optimizacije – koje poboljšavaju postojeća rješenja. (Pohlepa nalazi neko početno rješenje).

Napomena: funkcija izbora je najčešće bazirana na funkciji cilja – čak mogu biti identične.
  Međutim, u nekim primjenama, za isti problem – istu funkciju cilja, možemo imati nekoliko prihvatljivih i razumnih funkcija izbora.
  Treba odabrati pravu, ako želimo da algoritam radi korektno, odn. možemo dobiti različite korektne algoritme za isti problem.

Primjer 1. Kupcu treba vratiti ostatak, koristeći minimalni broj novčića (kovanaca).
  Karakteristični elementi ovog problema su:
  - kandidati: konačni skup novčića, koji na pr. odgovaraju vrijednosti od 1, 5, 10 ili 25 jedinica.
    Skup sadrži bar jedan novčić svake vrste;

- <u>rješenje</u>: ukupna vrijednost izabranog skupa novčića je <u>točno jednaka</u> iznosu kojeg treba vratiti (platiti) kupcu;
- <u>dopustiv skup</u>: ukupna vrijednost izabranog skupa novčića <u>ne prelazi</u> ($\leq$) iznos kojeg treba vratiti;
- <u>funkcija izbora</u>: izaberi novčić najveće vrijednosti u skupu preostalih kandidata;
- <u>funkcija cilja</u>: broj novčića u rješenju ∎

<u>Zadatak 1.</u> Dokaži da, uz predloženi izbor vrijednosti novčića iz primjera 1, pohlepni algoritam uvijek nalazi optimalno rješenje, ako rješenje postoji.
Ako postoji novčić vrijednosti 12, ili ako jedan od tipova novčića izbacimo iz početnog skupa, onda pohlepni algoritam ne mora dati optimalno rješenje u svakom slučaju. Nađi takve primjere.
Pokaži da se može dogoditi da pohlepni algoritam uopće ne nađe rješenje, iako ono postoji ∎

(Dodatak na pohlepu u primjeru 1:
- efikasnije je odbaciti na pr. sve novčiće vrijednosti 25 odjednom, kad preostali iznos padne ispod te vrijednosti;
- treba koristiti cjelobrojno dijeljenje umjesto ponovljenog oduzimanja).

2.1.1. <u>Minimalno razapinjuće stablo</u>

Neka je $G = (V, E)$ povezan, neusmjeren graf sa skupom vrhova V i skupom bridova (grana) E. Svaki brid ima nenegativnu <u>duljinu</u> (tj. to je mreža).

<u>Problem MST</u> (Minimal Spanning Tree) - minimalno razap. stablo
Treba naći podskup T bridova grafa G tako da svi vrhovi ostanu povezani samo bridovima iz T i da je zbroj duljina bridova u T najmanji moguć ∎

(Umjesto duljine, svakom bridu možemo pridružiti cijenu. Tada tražimo podskup T najmanje ukupne cijene)

Lako se vidi da je podgraf (V, T) grafa G stablo, tj. povezan graf bez ciklusa.
Taj graf zovemo minimalno razapinjuće stablo grafa G (min → iz cijene!).

Primjena: ako vrhovi od G predstavljaju gradove, a cijena brida {a, b} je cijena izgradnje ceste između gradova a i b, onda minimalno razapinjuće stablo grafa G kaže kako treba projektirati sistem cesta koji povezuje navedene gradove uz najmanju moguću cijenu.

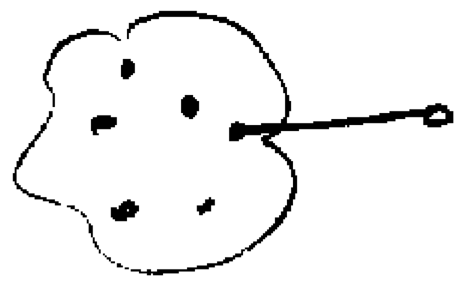
Dat ćemo 2 pohlepna algoritma za ovaj problem: Prim-ov i Kruskal-ov.

Karakteristični elementi u terminologiji pohlepnih alg. su:

- kandidati su bridovi (na početku – svi bridovi iz E)
- rješenje = skup bridova koji čini razapinjuće stablo (veže sve vrhove iz V)
- skup bridova je dopustiv, ako ne sadrži ciklus. (ne zahtjevamo da bude stablo – tj. da bude povezan – može biti i šuma – nepovezan, tj. svaka komponenta povezanosti je stablo.)
  [Tu će i biti razlika između algoritama]
- funkcija cilja je očita – suma duljina svih bridova u rješenju.

- Funkciju izbora ćemo kasnije specificirati, jer će se ona razlikovati u ta 2 alg.

- Uvodimo još 2 termina, potrebna za dokaz korektnosti ovih alg.:
  - Dopustiv skup bridova je obećavajući (ili obećava) ako se može dopuniti do optimalnog rješenja. (Posebno, prazni skup je uvijek obećavajući, jer je G povezan)
  - Brid e dira dani skup vrhova, ako je točno jedan kraj brida u tom skupu vrhova.



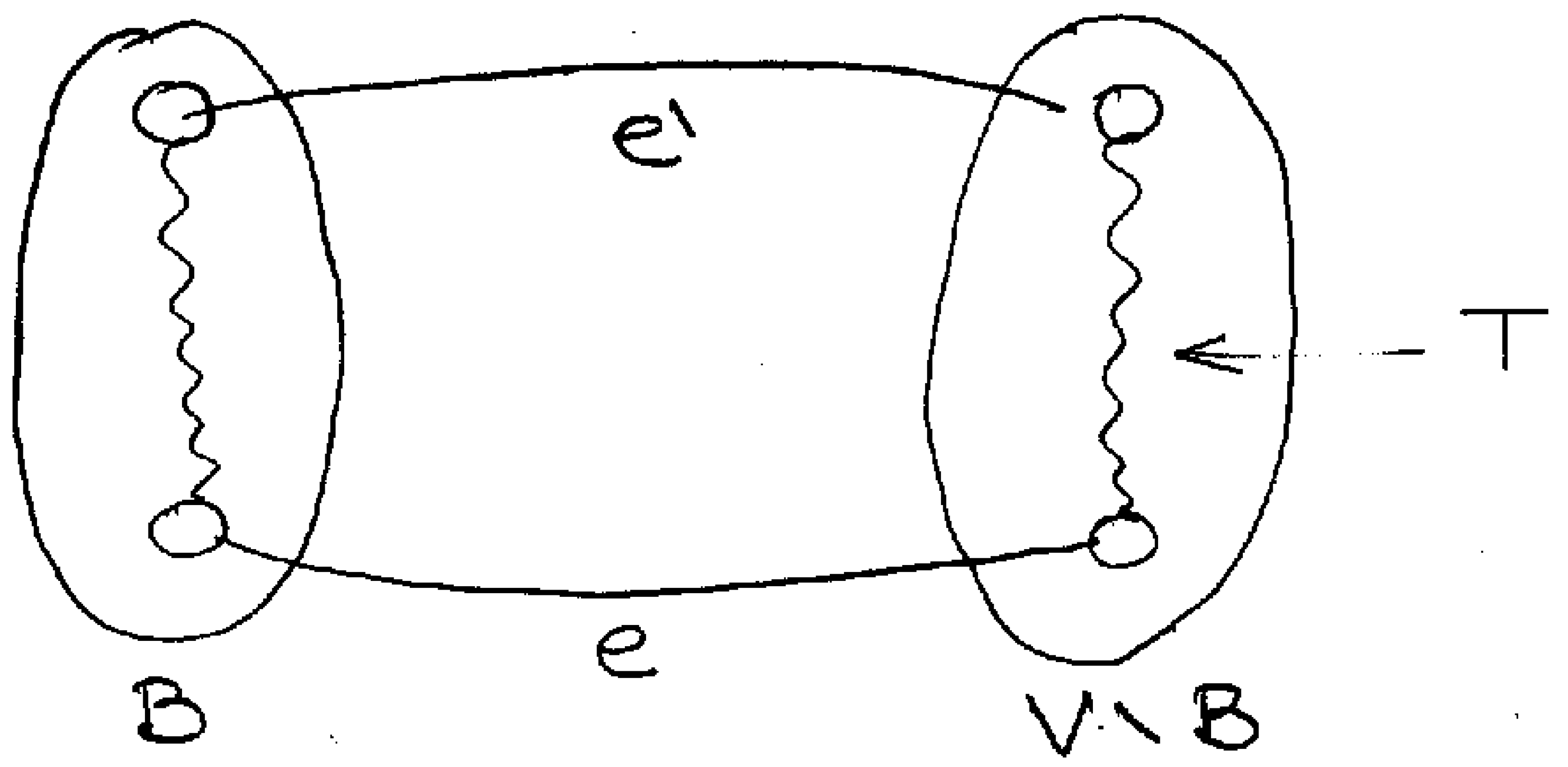
Slijedeća lema omogućava konstrukciju alg. i dokaz njihove korektnosti za problem MST.

Lema 1. Neka je $G = (V, E)$ povezan neusmjeren graf sa zadanim duljinama svih bridova. Neka je $B \subset V$ pravi podskup skupa vrhova grafa G. Neka je $T \subseteq E$ obećavajući skup bridova, takav da niti jedan brid iz T ne dira B. Neka je e najkraći brid koji dira B (ili bilo koji takav, ako ima više najkraćih). Tada je i skup $T \cup \{e\}$ obećavajući.

Dz: T je obećavajući, po pretp. Stoga postoji minimalno razapinjuće stablo U grafa G takvo da je $T \subseteq U$.

Ako je $e \in U$, nemamo što dokazivati.

U suprotnom, $e \notin U$, kada brid e dodamo skupu U, onda nastaje točno jedan ciklus (jer je U stablo!).



U tom ciklusu, jer e dira B, mora postojati bar još jedan brid e', koji također dira B (inače se ciklus ne zatvara!)

Ako izbacimo brid e', ciklus nestaje i dobivamo novo stablo U' koje razapinje graf G.

No, po pretpostavci, duljina brida e ne prelazi duljinu brida e', pa ukupna duljina bridova u U' ne prelazi ukupnu duljinu bridova u U.

Tada je i U' minimalno razapinjuće stablo za graf G i sadrži brid e.

Na kraju, mora biti $T \subseteq U'$, jer e' ne može biti u T. Naime, e' dira B, a niti jedan brid iz T, po pretp., ne dira B.

Dakle, i $T \cup \{e\}$ je obećavajući (u U').

Q.E.D.

Početni skup kandidata je skup svih bridova. Pohlepni alg. bira bridove u nekom poretku. Svaki brid se ili dodaje u buduće rješenje ili izbacuje iz daljnjeg razmatranja.

Osnovna razlika između raznih pohlepnih alg. za MST je u poretku izbora bridova.

Primov algoritam (1957-Prim, 1959-Dijkstra, original: 1930-JARNÍK)

Starta s bilo kojim vrhom - "korijenom" stabla, tj. na početku je B jednočlan, s bilo kojim vrhom. Skup T bridova je inicijalno prazan.

U svakom koraku dodajemo novi brid - granu već konstruiranom stablu i algoritam staje kad povežemo sve vrhove (stignemo do njih).

Tj. stalno imamo stablo koje "raste".

- U jednom koraku, Primov algoritam nalazi najkraći mogući brid $\{u, v\}$ takav da je

$$u \in V \setminus B \quad \text{i} \quad v \in B.$$

Tada dodaje u u skup B i brid $\{u, v\}$ u skup T.

Na taj način, bridovi u skupu T, u svakom trenutku predstavljaju minimalno razapinjuće stablo za vrhove iz B.

Postupak se nastavlja sve dok je $B \neq V$.

Neformalni zapis algoritma je:

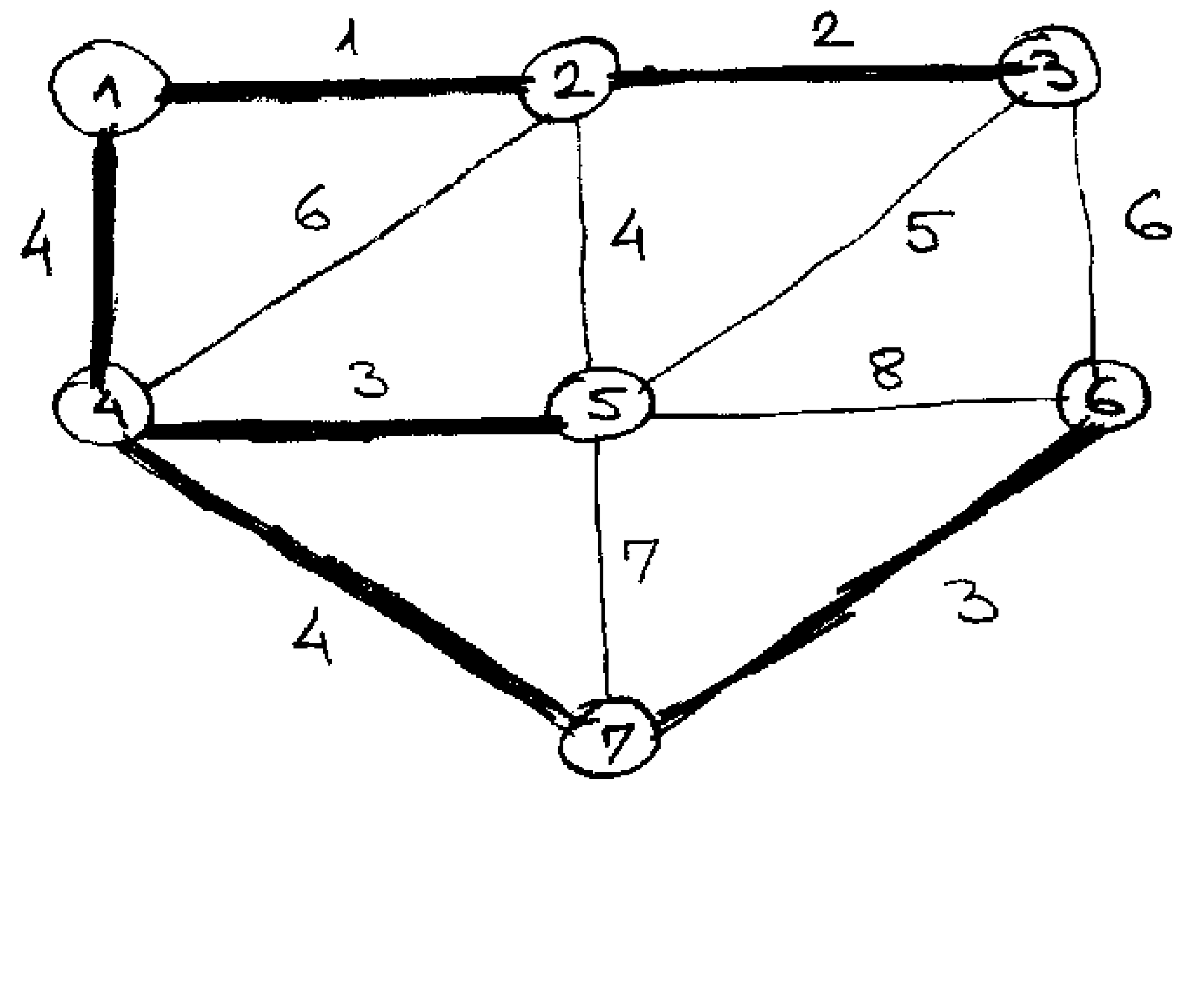
```
procedure Prim ( G=(V,E) : graf;
                 l : E → R_0^+ : funkcija;
                 var T : skup_bridova );

begin
  {inicijalizacija}
  T ← ∅;   {sadržavat će bridove min. raz. stabla}
  B ← bilo koji element iz V;
  while B ≠ V do
    begin
    nađi {u,v} najmanje duljine l({u,v}), takav
        da je u ∈ V\B i v ∈ B.;
    T ← T ∪ {{u,v}};
    B ← B ∪ {u};
    end;
  end; {Prim}
```

Teorem 1. Primov algoritam radi korektno, tj. za povezan neusmjeren graf G, on vraća minimalno razapinjuće stablo T.

Dokaz: Direktno iz Leme 1, indukcijom po broju vrhova u skupu B. Q.E.D.

Uoči da se while izvodi točno $|V|-1$ puta.

Primjer 2. Zadan je graf G, sa 7 vrhova i bridovima s duljinama kao na slici:



Ilustrirajmo rad Primovog algoritma na tom grafu.
Čvor 1 uzimamo (proizvoljno) kao polazni.

Korak	Izabrani brid $\{u,v\}$	Skup povezanih čvorova B
Inicijalizacija	—	$\{1\}$
1	$\{2,1\}$ (mogući izbori su $\{2,1\}$, $\{4,1\}$) $\underset{1}{\{2,1\}}$, $\underset{4}{\{4,1\}}$	$\{1,2\}$
2	$\{3,2\}$ (mogući izbori su: $\{4,1\}$, $\{3,2\}$, $\{4,2\}$, $\{5,2\}$) dulj: 4, 2, 6, 4	$\{1,2,3\}$
3	$\{4,1\}$ (mogući izbori su: $\{4,1\}$, $\{4,2\}$, $\{5,2\}$, $\{5,3\}$, $\{6,3\}$) dulj: 4, 6, 4, 5, 6 na por	$\{1,2,3,4\}$
4	$\{5,4\}$ (mogući izbori su: $\{5,2\}$, $\{5,3\}$, $\{5,4\}$, $\{6,3\}$, $\{7,4\}$) dulj. 4, 5, 3, 6, 4	$\{1,2,3,4,5\}$
5	$\{7,4\}$ (mogući izbori su: $\{6,3\}$, $\{6,5\}$, $\{7,4\}$, $\{7,5\}$) dulj: 6, 8, 4, 7	$\{1,2,3,4,5,7\}$
6	$\{6,7\}$ (mogući izbori su: $\{6,3\}$, $\{6,5\}$, $\{6,7\}$) dulj: 6, 8, 3	$\{1,2,3,4,5,6,7\} = G$

Stablo T sadrži bridove:

$$\underset{1}{\{2,1\}},\ \underset{2}{\{3,2\}},\ \underset{4}{\{4,1\}},\ \underset{3}{\{5,4\}},\ \underset{4}{\{7,4\}},\ \underset{3}{\{6,7\}}$$

Ukupna duljina je 17 ■

Napomena: U ovoj formi, izabrani brid uvijek prihvaćamo, tj. nema odbacivanja. (*while* se izvodi točno $|V|-1$ puta – tj. *for*)

Kad bi funkcija izbora (kao što i treba) vratila samo brid najmanje duljine, među preostalim bridovima, imali bismo mogućnost odbacivanja.

Funkcija dopustivog rješenja treba prihvatiti samo one bridove $\{u,v\}$ za koje je $u \in V \setminus B$ i $v \in B$ (prema Lemi 1) tako da dobijemo minimalno razapinjuće stablo za $B \cup \{u\}$.

- Graf može imati više minimalnih razapinjućih stabala. U algoritmu se ta mogućnost ogleda tako da imamo nekoliko bridova iste najmanje duljine $\ell(\{u,v\})$ koji zadovoljavaju i uvjet $u \in V \setminus B$, $v \in B$.

- Za potpunu specifikaciju algoritma, treba odabrati strukture podataka za prikaz objekata koje omogućavaju efikasno izvođenje izbora.

- Jednu jednostavnu implementaciju dobivamo ovako: Pretpostavimo da su vrhovi grafa G numerirani brojevima od 1 do n, $V = \{1, 2, \ldots, n\}$. Bridove i pripadne duljine zadajemo simetričnom matricom L.

$$L(u,v) = \begin{cases} \ell(\{u,v\}) & , \text{ ako je } \{u,v\} \in E \\ \infty & , \text{ ako } \{u,v\} \notin E. \end{cases}$$

Smatramo da su duljine nenegativne.
(Možemo smatrati da su duljine pozitivne (najčešće u praksi), a tada koristiti $\emptyset$, umjesto ∞, uz testiranje na $\emptyset$. Tada L odgovara matrici susjedstva, s tim da, umjesto 1, piše $\ell(\{u,v\})$. Ovo se lako dodaje u sljedeći algoritam).

[U praksi, $\ell(\{u,v\}) = 0$ može imati smisla. Na pr. kod projektiranja novog sistema cesta, to znači da već postoji cesta između gradova u, v, tj. cijena <u>nove</u> ceste je $\emptyset$.]

Koristimo još 2 polja: nearest (najbliži) i mindist (najmanja udaljenost).

Za vrh $i \in V \setminus B$,

nearest[i] = vrh iz B, najbliži vrhu i,

mindist[i] = udaljenost vrhova i, nearest[i].

Za vrh $i \in B$ stavljamo mindist[i] = −1 (ovdje može i ∅, ako su sve udaljenosti baš pozitivne).

- Skup B ne treba posebno reprezentirati, jer se on može rekonstruirati iz mindist.

  Proizvoljno inicijaliziramo $B = \{1\}$, a nearest[1] i mindist[1] uopće ne koristimo.

- Ako je $|V| = n$, onda znamo da razapinjuće stablo T mora imati točno $n-1$ bridova (zbog povezanosti grafa G). U prethodnom, T je bio skup bridova. Zbog poznate duljine ($|T| = n-1$), možemo koristiti i polje bridova. Svaki brid je dvočlani skup – pa možemo koristiti ili polje duljine 2, ili pravi skup, ili record s 2 komponente. Možemo, za T, koristiti i 2 obična polja. (prvo za prvi član – vrh iz brida, a drugo za drugi vrh).
- Točna reprezentacija za T ovisi o konkretnoj primjeni, pa ju nećemo detaljno specificirati.
- Također, nećemo vratiti ukupnu duljinu za T. Dodatak toga u algoritam je trivijalan.

<u>Algoritam 1</u> (Primov algoritam za MST)

```
procedure Prim ( n : integer;
                 L : matrix ; { L[1..n, 1..n] koristimo }
                 var T : skup-bridora );

  var i, j, k : 1.. n;

begin
   { inicijalizacija B = [1] , T = [ ] }
  T ← ∅ ; { T će akumulirati bridove MST }
  for i ← 2 to n do    { samo vrh 1 je u B }
     begin
     nearest [i] ← 1;
     mindist [i] ← L[i,1];
     end;
   { pohlepna petlja }
  for i ← 1 to n-1 do   { T sadrži n-1 bridora }
     begin
     min ← ∞;
     for j ← 2 to n do
       if (mindist[j] >= ∅) and (mindist[j] < min) then
          begin
          min ← mindist[j];
          k ← j
          end;
       { k je def. ako je G povezan, inače ne !! }
    T ← T ∪ { k, nearest[k] }
    mindist [k] ← -1 ; { dodaj vrh k skupu B }
       { popravi polja nearest, mindist za novi k u B }
    for j ← 2 to n do
       if L[k, j] < mindist [j] then   { j ∉ B }
ovo       begin
ne treba  mindist [j] ← L[k, j]
za i=n-1  nearest [j] ← k
          end;
     end; { for i = pohlepna petlja }
end; { Prim }
```

Zadatak 2. Pretpostavka za korektnost algoritma je da je G povezan graf. Ovaj algoritam to ne provjerava. Kako ga treba popraviti da uredno javlja da je G nepovezan? Što tada vraća algoritam za T?

- Analiziramo kompleksnost Primovog algoritma.

Za graf G s n vrhova, pohlepna (vanjska) petlja (po i) se izvodi točno $n-1$ puta.

U našoj implementaciji – jedan korak sadrži još dvije petlje s po $n-1$ prolaza, a sve ostale operacije su elementarne i ne ovise o n.

Dakle, jedan korak ima trajanje $O(n)$ (kod nas čak $\Theta(n)$).

Očito je da za cijeli algoritam vrijedi

$$\boxed{T(n) = O(n^2)}$$

a kod nas čak $T(n) = \Theta(n^2)$.

Za empirijski pristup, pogodan je kvadratni polinom za $T(n)$.

- Preciznije trajanje, naravno, ovisi o tome koliko bridova ima graf G (najviše ih je $\frac{n(n-1)}{2} \sim \frac{n^2}{2}$), tj. koliko "gust" je G.

To utječe bitno na broj prolaza kroz then blokove u oba if-a.

Ako je graf "rijedak" (blizu stablu – min broj bridova je $n-1$, zbog povezanosti), onda je tih prolaza vrlo malo.

Zbog toga je pogodnije polja nearest i mindist reprezentirati dinamički. Posebno to vrijedi za mindist, tako da ne pamtimo vrijednosti -1 (već povezane čvorove).

Ova modifikacija bitno skraćuje broj prolaza kroz obje unutarnje petlje.

- Detaljnija analiza bitno ovisi o detaljima strukture grafa G (broju bridova, valenciji – stupnju i sl.).

Kruskalov algoritam (1956 - Kruskal)

Ovaj algoritam, također, u skupu bridova T, akumulira minimalno razapinjuće stablo grafa G.

Na početku je $T = \emptyset$. U svakom koraku skupu T pokušavamo dodati neki brid.

U svakom trenutku, podgraf od G oblika (V, T), sastavljen od svih vrhova grafa G i bridova iz T se sastoji od nekog broja povezanih komponenti (ali sam ne mora biti povezan).

Bridovi iz T, sadržani u nekoj komponenti povezanosti tog grafa (V, T), čine minimalno razapinjuće stablo za tu komponentu (tj. vežu sve vrhove u toj komponenti).

Na početku, kada je T prazan, svaki vrh grafa G čini zasebnu, trivijalnu povezanu komponentu. Takvih komponenti ima $n = |V|$.

Kruskalov algoritam spaja disjunktne povezane komponente u jednu, veću povezanu komponentu, dodavanjem brida između njih.

Stajemo u trenutku kad dobijemo samo jednu komponentu povezanosti i tada je T minimalno razapinjuće stablo za G, jer je G povezan.

– Kako dodajemo bridove pri gradnji sve većih komponenti povezanosti?

Pohlepno! Pretražujemo sve bridove grafa G, uzlazno sortirano po duljini tih bridova. Tj. počinjemo s najkraćim bridovima.

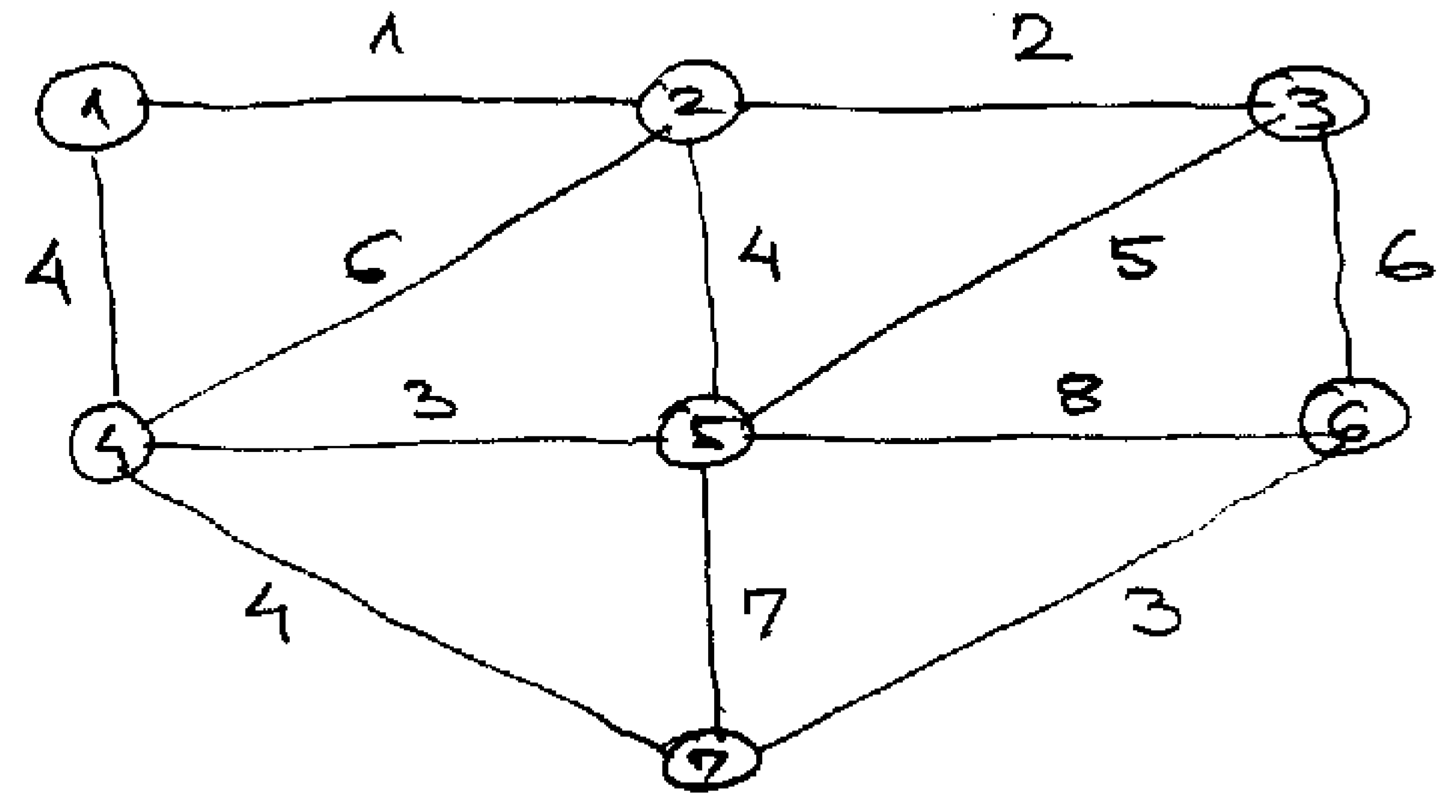
Ako najkraći preostali brid spaja dva vrha u raznim (disjunktnim!) komponentama povezanosti, onda ga dodajemo u T. Posljedica = dvije komponente povezanosti se spajaju u jednu, novim bridom u T. Bridovi u T i dalje čine MST za tu novu komponentu.

U protivnom, brid se odbacuje, jer spaja 2 vrha iz iste komponente povezanosti. Pošto bridovi u T već tvore MST za tu komponentu, dodavanje tog novog brida u skup T bi zatvorilo ciklus i narušilo željeno svojstvo skupa T.

Teorem 2. Za povezan, neusmjeren graf G, Kruskalov algoritam radi korektno.

Dokaz: smo upravo proveli!
Formalno, to je indukcija po broju izabranih bridova do tog časa, pozivanjem na lemu 1 u svakom koraku.
Q.E.D.

Primjer 3. Ilustrirajmo rad algoritma na istom grafu kao i za Primov algoritam u primjeru 2.



Bridovi ovog grafa, sortirani uzlazno po duljini, su redom:

	{1,2},	{2,3},	{4,5},	{6,7},	{1,4},	{2,5},	{4,7},	{3,5},
dulj:	1	2	3	3	4	4	4	5

	{2,4},	{3,6},	{5,7},	{5,6}
dulj:	6	6	7	8.

korak	promatrani brid	povezane komponente u (V,T)
Inicijalizacija	—	{1} {2} {3} {4} {5} {6} {7}
1	najmanji: {1,2}	{1,2} {3} {4} {5} {6} {7}
2	slijedeći: {2,3}	{1,2,3} {4} {5} {6} {7}
3	slijedeći: {4,5}	{1,2,3} {4,5} {6} {7}
4	slijedeći: {6,7}	{1,2,3} {4,5} {6,7}
5	slijedeći: {1,4}	{1,2,3,4,5} {6,7}
6	slijedeći: {2,5}	odbacuje se (oba vrha u istoj komponenti → ciklus)
7	slijedeći: {4,7}	{1,2,3,4,5,6,7} = V → STOP

T sadrži bridove: {1,2}, {2,3}, {4,5}, {6,7}, {1,4}, {4,7} ukupne duljine 17. Rezultat je isti kao kod Primovog alg. ■

Zadatak 3 Graf može imati više minimalnih razapinjućih stabala. Kako se to odražava u Kruskalovom algoritmu?

Razlika od Primovog algoritma: biramo obećavajuće bridove, ne vodeći računa o tome da li su vezani s prethodno odabranim bridovima. Pazimo samo na to da nikad ne zatvorimo ciklus.
Tj. tokom Kruskalovog algoritma, graf (V,T) je šuma, a stajemo kad postane povezan, tj. stablo.

Zadatak 4. Što se događa u Kruskalovom algoritmu, ako graf G nije povezan?

Implementacija algoritma se sastoji iz operacija s određenim brojem skupova. Svaki skup čine svi vrhovi u istoj povezanoj komponenti, u danom trenutku. Preciznije, trebamo što efikasnije izvesti slijedeće dvije operacije:

find (x) – "nađi x" = nađi skup – komponentu povezanosti u ~~kojoj~~ kojoj se nalazi vrh x;

merge (A,B) – "spoji A i B" – koja spaja dvije disjunktne komponente – skupa u jedan skup. (unija disjunktnih skupova).

Potrebna apstraktna struktura (ili tip) podataka su tzv. disjunktni skupovi, ili, preciznije, particije nad određenim skupom (u ovom slučaju, to je skup svih vrhova, a komponente povezanosti su particija tog skupa).

- Ovu strukturu i efikasnu implementaciju operacija find i merge, opisujemo nakon algoritma. Algoritam ćemo sastaviti tako da koristi ove 2 operacije kao potprograme, koje kasnije specificiramo.
- U Kruskalovom algoritmu moramo sortirati bridove po duljini. Zbog toga je graf pogodnije prikazati vektorom (ili poljem, listom) bridova s pridruženim duljinama, a ne matricom udaljenosti.

Neformalni zapis algoritma je:

```
procedure Kruskal ( G=(V,E) : graf;
                    l : E → R₀⁺ : funkcija;
                    var T : skup_bridova );

begin
   { inicijalizacija }
   Sortiraj E uzlazno po duljini l bridova;
   n ← card(V);  { ili |V| }              (bolje N umjesto n (v. find merge))
   T ← ∅;  { akumulira bridove MST }
   inicijaliziraj n (disjunktnih) skupova tako da svaki
      sadrži točno po jedan različit vrh iz V;

   { pohlepna petlja }
   repeat
      {u,v} ← najkraći preostali brid; { do tada neobrađen }
      ucomp ← find(u);   } { nađi komponente povezanosti u
      vcomp ← find(v);   }   kojima su vrhovi u i v }
      if ucomp ≠ vcomp then { prihvati brid {u,v} }
         begin
         merge(ucomp, vcomp);  { spoji dvije različite
                                 komponente povezanosti u jednu }
         T ← T ∪ {{u,v}};
         end
      else
         odbaci brid {u,v};  { jer bi zatvorio ciklus u
                               istoj komponenti }

   until card(T) = n-1;
end; { Kruskal }
```

- U ovom algoritmu može doći i do odbacivanja brida, pa repeat ne možemo pretvoriti u for.
- Uvjet $card(T) = n-1$ odgovara tome da znamo da razapinjuće stablo mora imati $n-1$ bridova u povezanom grafu.
  Ekvivalentan uvjet je da je preostala samo jedna komponenta povezanosti.
- Ako graf nije povezan, ovaj uvjet treba modificirati, na pr. tako da stajemo kad smo obradili sve bridove grafa. (Onda T daje minimalnu razapinjuću šumu. Dokaži to!)

Prije analize kompleksnosti Kruskalovog algoritma, moramo specificirati operacije (funkcije) find i merge.

Struktura disjunktnih skupova

Pretpostavimo, općenitije, da imamo N objekata numeriranih brojevima od 1 do N. Te objekte želimo grupirati u disjunktne skupove, tako da se svaki objekt nalazi u točno jednom od skupova, u svakom trenutku.
Tj. imamo particiju skupa $\{1, \ldots, N\}$, koja reprezentira neku relaciju ekvivalencije.

Skupove u particiji treba nekako označiti (imenovati, nazvati) da bismo ih mogli razlikovati.
U svakom skupu izabiremo tzv. kanonski objekt (element) koji će služiti kao oznaka (labela, ime) tog skupa.

Na početku, tih N objekata se nalazi u N različitih skupova, od kojih svaki sadrži po točno jedan objekt. Taj objekt je, nužno, i oznaka za taj skup. (To reprezentira trivijalnu, najveću, relaciju ekvivalencije - svaki element je ekvivalentan samo sebi).

Nakon toga izvodimo grupiranje tih objekata u nizu od n koraka: Svaki korak je operacija jedne od ovih dviju vrsta:

- za dani objekt, nađi (find) skup (klasu ekvivalencije) kojem on pripada i vrati oznaku tog skupa;
- za dvije zadane, različite oznake, spoji (merge) dva pripadna skupa u jedan (unija klasa u jednu = grupiranje ⟺ dvije razne grupe više ne razlikujemo i grupiramo ih u istu grupu, smatramo ih istim!)

Ovaj problem ima niz primjena. Na pr. u statistici, ili u teoriji grafova (Kruskal) ili u jezicima i prevođenju (razrješavanje EQUIVALENCE naredbi u FORTRANu, vodi upravo na ovaj problem).

- Traži se efikasna reprezentacija za ovaj tip podataka.

- Jedna od mogućih reprezentacija je očita - i, usput, "pohlepna".
Zamislimo da smo odlučili da najmanji element svakog skupa koristimo kao oznaku za skup.
Na pr. skup $\{7, 3, 16, 9\}$ (poredak elemenata nije bitan!} zovemo "skup 3". [Ovo je "pohlepno" označavanje!]

Ako deklariramo polje (vektor)

skup [1..N]

dovoljno je za svaki objekt zapamtiti oznaku skupa kojem on pripada, na odgovarajuće mjesto u polju. Tj. za svaki element pamtimo pointer na pripadni skup. (Očito se bitno koristi disjunktnost skupova – svaki element se nalazi u točno jednom skupu).

$$skup[i] = j \iff i \in \text{skup s oznakom } j.$$

Primijetimo da ova reprezentacija vrijedi za <u>bilo koje</u> označavanje skupa nekim njegovim elementom (a ne samo za označavanje najmanjim elementom).

– Inicijalizacija je:

```
procedure init;
  {inicijaliziraj skupove na jednočlane, svaki element
   u svom skupu}
begin
  for i ← 1 to N do
    skup [i] ← i;
end; {init}
```

Smatramo da je polje "skup" globalno.
Ova procedura također ne ovisi o označavanju elemenata, jer su na početku, svi skupovi jednočlani ⇒ svaki skup ima oznaku svojim jedinim elementom.

– Tražene operacije find i merge su:

```
function find1 (x);
  {vraća oznaku skupa u kom se nalazi objekt x}
                {oznaka je istog tipa kao i objekt}
begin
  find1 ← skup [x];
end; {find1}
```

Kasnije ćemo dati efikasniju implementaciju, pa zato naziv find1 = 1. verzija operacije find.
Ova implementacija, također, ne ovisi o označavanju.

- Spajanje ovisi o označavanju, jer treba odabrati oznaku novog skupa (koji nastaje spajanjem 2 skupa).

```
procedure merge1 (a, b)
  {Spaja skupove s oznakama a i b}
  {Tj. treba propisno postaviti oznaku pripadnog
   skupa za sve elemente u uniji}
begin
  i ← a; j ← b;
  if i > j then zamijeni i, j;
   {Tj. i ← min{a,b}, j ← max{a,b} }
  for k ← 1 to N do
    if skup[k] = j then
      skup[k] = i;   {sa većeg na manji}
end; {merge1}
```

- Kako procjenjujemo efikasnost?

Ovisno o primjeni, operacije find i merge se mogu javljati u bilo kom redoslijedu.

Zbog toga, treba naći vrijeme potrebno za izvođenje proizvoljnog niza od n operacija tipa "find" ili "merge", s tim da startamo iz početne situacije u "init".

- Što su elementarne operacije?

Smatramo da je jedno dohvaćanje, provjera ili modifikacija elementa u polju "skup" elementarna operacija (i to brojimo!)

Očito find1 troši konstantno vrijeme (1 takva operacija), dok merge1 zahtijeva vrijeme reda veličine N (zbog petlje).

Niz od n takvih operacija, u najgorem slučaju, troši vrijeme reda veličine $n \cdot N$

$$T_W(n \times \{\text{find1 ili merge1}\}) \in O(nN).$$

- Međutim, lako se može postići znatno bolje od toga.

Ideja - pravilnije rasporediti vrijeme između ove 2 operacije

Ovdje - find1 - trivijalan $O(1)$
merge1 - dugotrajan $O(N)$

- Svaki skup možemo prikazati kao "naopako" korijensko stablo, s idejom da korijen služi kao oznaka skupa. Za to je još uvijek dovoljno samo jedno polje "skup".
- Prikaz realiziramo na slijedeći način:

  ako je skup[i] = i, onda je i istovremeno oznaka skupa i korijen pripadnog stabla;

  ako je skup[i] = j ≠ i, onda je j otac od i u nekom stablu

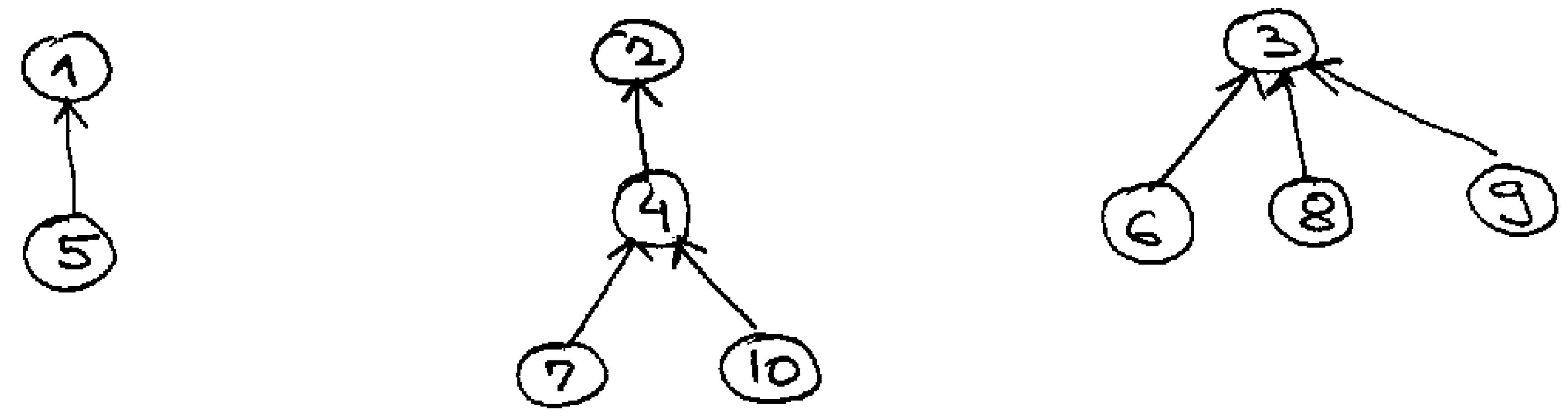
  (tj. ako element nije korijen, onda pamtimo pointer na oca u nekom stablu).

Na pr. polje "skup" oblika:

objekt =	1	2	3	4	5	6	7	8	9	10
skup[objekt] =	1	2	3	2	1	3	4	3	3	4

$\underbrace{\uparrow \quad \uparrow \quad \uparrow}_{\text{korijeni}}$

prikazuje šumu od 3 stabla oblika:

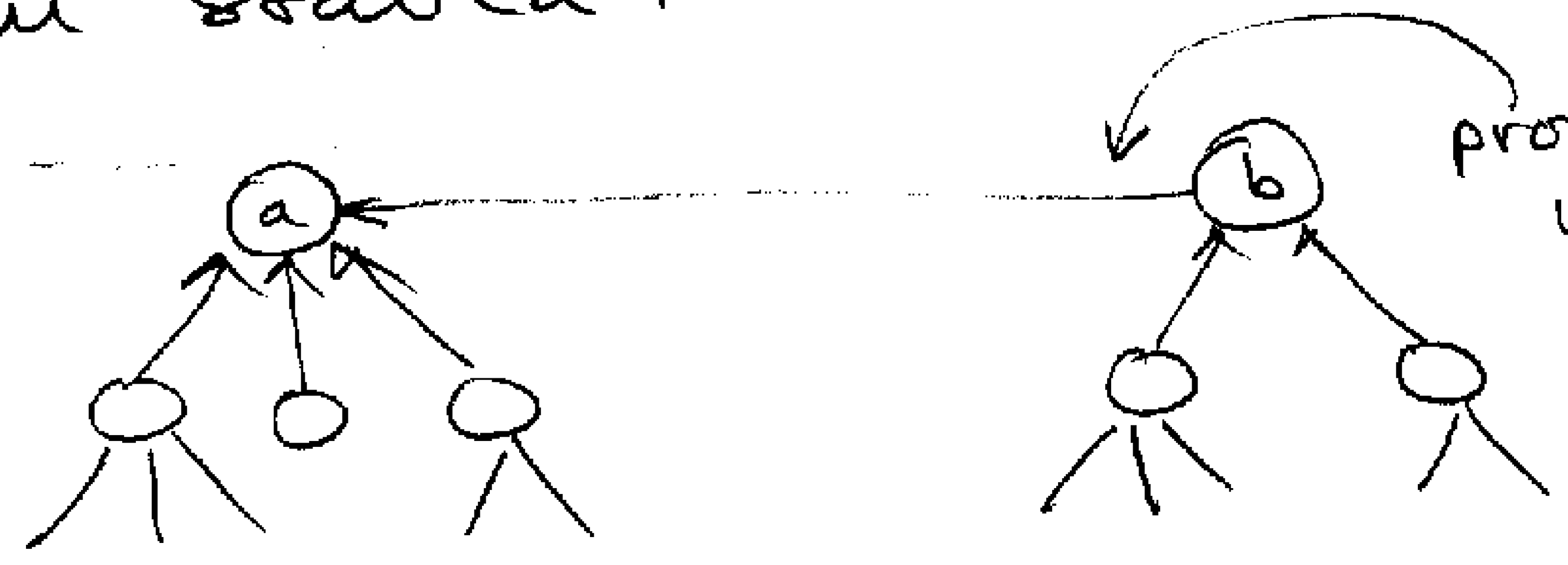


(strelice prikazuju smjer pointera u polju "skup", a ne smjerove bridova u stablu kao grafu)

Ova 3 stabla, očito, prikazuju skupove:

{1, 5} {2, 4, 7, 10} {3, 6, 8, 9}.

- U ovoj reprezentaciji, merge je trivijalan – dovoljno je promijeniti korijen jednog od dva skupa (stabla) (većeg na manjeg!). Dakle, treba promijeniti jedan element u polju i njega odmah nalazimo – jer je to korijen stabla:



promijeni u a, tj. zaveži na a.

```
procedure merge2 (a, b);
  { Spaja skupove s oznakama a i b }
begin
  if a < b then
    skup[b] ← a
  else
    skup[a] ← b;
end; { merge2 }
```

Oznaka je i korijen (tj. ulaz: skup[a]=a, skup[b]=b). Postavi oca veće oznake na manji od ulaznih korijena. Manji korijen je korijen unije.

Međutim, operacija find postaje bitno teža. Za dani element, treba se popeti po stablu sve do korijena, da saznamo oznaku skupa kojem taj element pripada.

```
function find2 (x)
  { Nalazi oznaku skupa koji sadrži objekt x }
  begin
    i ← x;        { start na tom elementu }
      { sve dok ne stignemo do korijena,
        popni se za jedan nivo u visinu }
    while skup[i] ≠ i do
      i ← skup[i];          { pređi na svog oca }
    find2 ← i;
  end; { find2 }
```

Jedna find operacija zahtjeva vrijeme proporcionalno visini stabla u kom se nalazi objekt.

Na početku su sva stabla trivijalna, jer sadrže samo korijen, tj. visine $\emptyset$.

Samo operacija merge mijenja visinu stabla i to za najviše 1. U jednom stablu se visina ne mijenja, a u drugom je novi korijen direktni otac starog korijena.

Dakle, nakon niza od n operacija find ili merge, visina stabla je najviše n, za jednu find operaciju:

$$T_W(\text{find2}) \in O(n).$$

Ukupno vrijeme za niz od n operacija find2 ili merge2 je onda, u najgorem slučaju (svi find2 $\Rightarrow \sum_{k=1}^{n} c \cdot k = O(n^2)$)

$$T_W(n \times \{\text{find2 ili merge2}\}) \in O(n^2)$$

– Ako je $n \ll N$, to je bitan napredak u odnosu na find1, merge1. Međutim, ako je $n \approx N$ (a upravo to je u Kruskalovom algoritmu, gdje je $n \geq N-1$), onda nismo ništa dobili, već možda izgubili.

– Uočimo da je ovo suprotni ekstrem od find1, merge1:

Ovdje: find2 – dugotrajan $O(n)$
merge2 – trivijalan $O(1)$.

– Gdje je problem?
U visini stabala! Nakon k poziva merge2, možemo dobiti stablo visine k, tako da svaki sljedeći poziv find2 troši vrijeme reda veličine k.

Pokušajmo, stoga, ograničiti (kontrolirati) visinu dobivenih stabala.
Dobro bi bilo da, kod spajanja, kao korijen ostavimo onaj korijen koji ima stablo veće visine, tj. da povećavamo visinu "nižeg" stabla.
Tj. korijen stabla manje visine postaje dijete onog drugog korijena.
Ovo je pohlepno kontroliranje visine!

– No, tada treba promijeniti način označavanja skupova. (Do sada smo, proizvoljno, koristili najmanji element kao oznaku).
I dalje je korijen stabla ujedno i oznaka skupa. Međutim, sada kontrola visine određuje oznaku skupa nakon spajanja.
Ako spajamo dva stabla, visina h_1 i h_2, visina h stabla dobivenog spajanjem je:

$$h = \begin{cases} \max\{h_1, h_2\}, & \text{ako je } h_1 \neq h_2 \\ h_1 + 1, & \text{ako je } h_1 = h_2. \end{cases}$$

Ovom tehnikom spajanja, visina stabala ne raste više tako brzo.

Zadatak 5. Dokaži matematičkom indukcijom da ovom tehnikom spajanja vrijedi:
Startajući iz početne situacije (init), nakon proizvoljnog niza (broja) merge operacija, stablo koje sadrži k ~~vrhova~~ objekata ima visinu h
$$h \leq \lfloor \lg k \rfloor$$ ■

U implementaciji ove tehnike, moramo pamtiti visine stabala. Zgodan način za to je korištenje globalnog polja

visina [1.. N],

tako da je:

visina [i] = visina objekta (vrha) i u njegovom trenutnom stablu.

Pri tome korijen ima najveću visinu (zato se ovo i zove "obratno" korijensko stablo), a najdublji čvorovi – listovi imaju visinu Ø. Čvorovi koji više nisu korijeni imaju istu visinu, bez obzira na daljnja spajanja!
Na taj način, ako je a oznaka nekog skupa, visina [a] daje direktno visinu pripadnog stabla.
Inicijalno, treba postaviti visina [i] = 0, ∀i = 1, .., N u proceduri init.

Proceduru find2 ne treba mijenjati, a nova operacija merge3 ima oblik:

```
procedure merge3 (a, b)
   { Spaja skupove s oznakama a i b }
begin
   { Pretpostavljamo da je a ≠ b, v. napomenu iza! }
  if visina [a] = visina [b] then
    begin
    visina [a] ← visina [a] + 1;
    skup [b] ← a
    end
  else
    if visina [a] < visina [b] then
      skup [a] ← b
    else
      skup [b] ← a;
end; { merge3 }
```

<u>Napomena</u>: Procedure merge1 i merge2 su radile korektno i ako je na ulazu bilo $a=b$ (Provjeri sam!).
U merge3 to <u>ne</u> vrijedi, jer bi visina stabla s korijenom a narasla za 1, a da se stablo nije promijenilo. Zbog toga pretpostavljamo da je $a \neq b$ na ulazu u merge3, ili treba dodati pripadni if oko cijelog posla ■

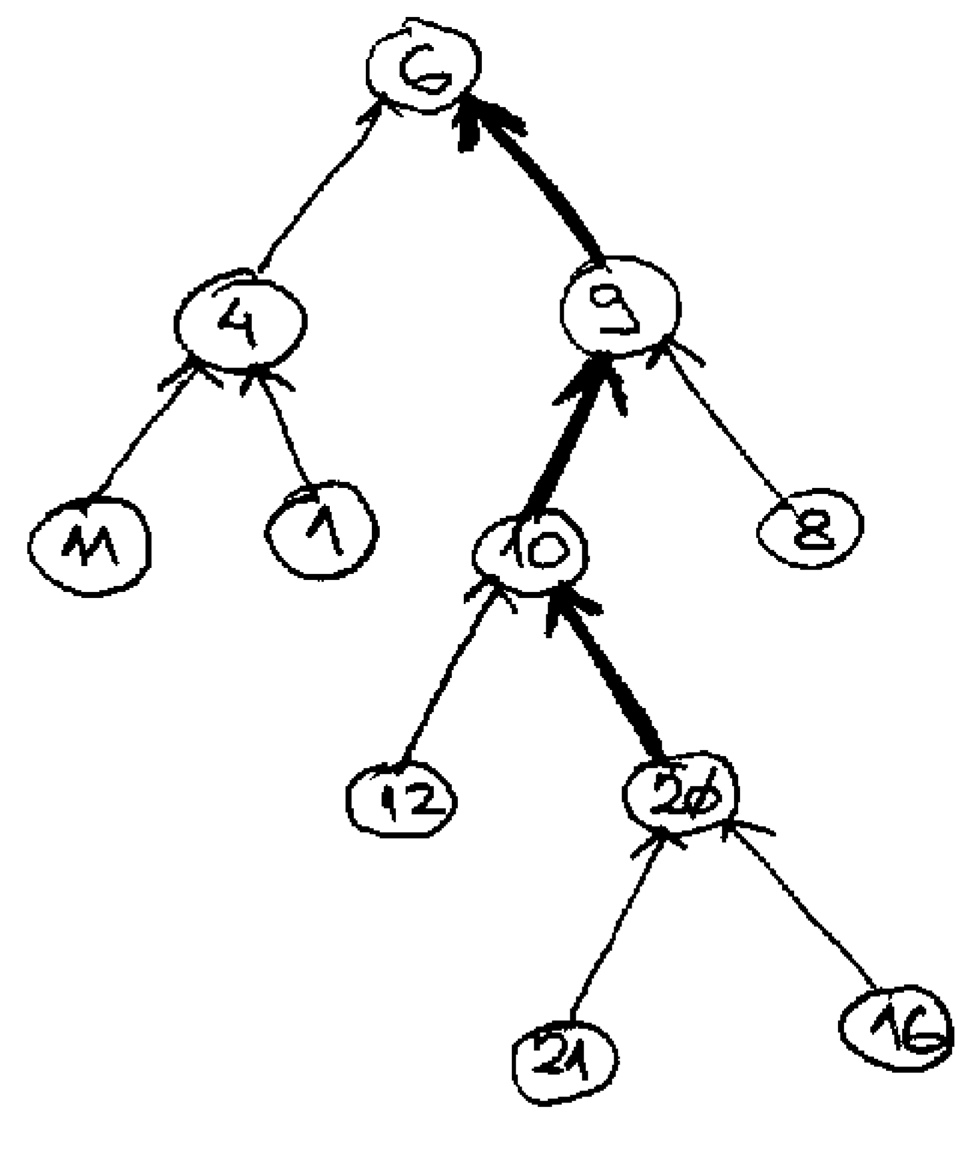
<u>Zadatak 6</u>. Dokaži da za ukupno vrijeme potrebno za izvršavanje niza od n operacija tipa find2 ili merge3, u najgorem slučaju vrijedi

$$T_W(n \times \{\text{find2}, \text{merge3}\}) = O(n \log n).$$

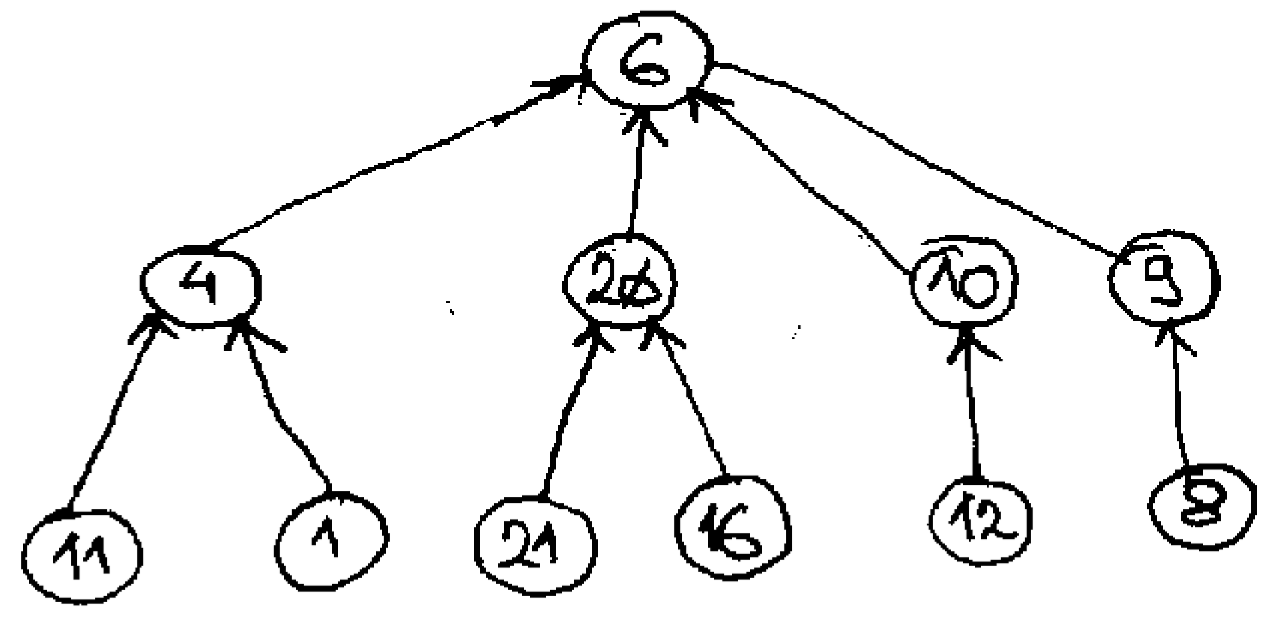
Uputa: koristi zadatak 5 i činjenicu da se visina čvorova koji nisu korijeni više ne mijenja ■

- Modifikacijom procedure find2, možemo još ubrzati ove operacije.
Kada odredujemo kojem skupu pripada neki objekt x, prvo obilazimo bridove stabla putem prema korijenu (koji je i oznaka stabla).
U trenutku kad saznamo korijen, možemo još jednom obići te iste bridove, i svakom čvoru (objektu), na kojeg naiđemo na tom putu, postaviti njegov pointer direktno na korijen.
Ova tehnika se zove <u>skraćivanje</u> (kompresija, sabijanje) <u>puta</u> (path compression).

- Na pr. izvršavanjem operacije find(20) na stablu:



dobivamo stablo:



Visina stabla pada s 4 na 2, a vrhovi 20, 10, 9 na putu od 20 do korijena sada pokazuju direktno na korijen.

Skraćivanje putova očito ne mijenja preostale vrhove i vrhove pointere.

Osim toga, očito teži smanjivanju visine stabla i stoga ubrzava kasnije find operacije.

S druge strane, novi find traje dvostruko duže od prethodnog (dva puta prolazi put do korijena).

- Nova verzija find3 ima oblik:

```
function find3 (x);
  { Nalazi oznaku skupa koji sadrži objekt x }
begin
  r ← x;
  while skup[r] ≠ r do          { ovo je isto kao
    r ← skup[r];                  u find2 }
  find3 ← r;
    { r je korijen stabla, smanji put }
  i ← x;
  while i ≠ r do
    begin
    j ← skup[i];
    skup[i] ← r;
    i ← j
    end;
end; { find3 }
```

- Ako koristimo skraćivanje putova, bez modifikacije polja "visina", onda više ne vrijedi da je visina stabla s korijenom a dana s visina[a]. Vrijednost visina[a] ostaje gornja ograda za pravu visinu.

Tu vrijednost onda zovemo rang stabla i mijenjamo i ime globalnog polja "visina" u "rang". Tu izmjenu onda treba napraviti i u merge3.

(Razlog za promjenu naziva: da naziv ne sugerira pogrešno značenje odn. interpretaciju).

Zadatak 7. Može li se u skraćivanju puteva efikasno pratiti i korektno postaviti prava visina novodobivenog stabla?
[Kako to utiče na kompleksnost – u find4 (koji tako prati visinu) i paketu od n operacija find4 i merge3].

Modificirane procedure init i merge3 u terminu polja "rang" glase:

```
procedure init;
   {inicijaliziraj stabla za skupove na jednočlane
    i pripadne rangove}
begin
   for i ← 1 to N do
      begin
      skup [i] ← i;
      rang [i] ← ∅;
      end;
 end; {init}

procedure merge3 (a, b);
   {Spaja skupove s oznakama a i b, uz pretpostavku
    da je a ≠ b}
begin
   if rang[a] = rang[b] then
      begin
      rang[a] ← rang[a] + 1;
      skup[b] ← a;
      end
   else
     if rang[a] < rang[b] then
       skup[a] ← b
     else
       skup [b] ← a;
end; {merge3}
```

Analiza kompleksnosti niza od n operacija tipa find3 ili merge3 je vrlo težak posao i nećemo ga detaljno provesti.
[v. Algorithmics, Example 2.2.10, p.60 – 63].

Može se pokazati da, u najgorem slučaju, vrijedi:

$$T_W(n \times \{find3, merge3\}) = O(n \cdot lg^* N)$$

uz uvjet da je $n \geq N$ (što je i najčešći slučaj u praksi).
Funkcija $lg^* : \mathbb{N} \to \mathbb{N}$ je definirana sa:

$$lg^* N = \min\{k \mid \underbrace{lg\, lg \ldots lg}_{k \text{ puta logaritmirano}} N \leq 0\}.$$

Funkcija lg^* izuzetno sporo raste. Vrijedi:

$$\begin{aligned}
&lg^* 1 = 1 = 2^0 \\
&lg^* N \leq 2 \quad \text{za } N \leq 2^1 = 2 \quad (= 2^{2^0}) \\
&lg^* N \leq 3 \quad \text{za } N \leq 2^2 = 4 \quad (= \underbrace{2^{2^{2^0}}}_{3 \text{ dvojke}}) \\
&lg^* N \leq 4 \quad \text{za } N \leq 2^4 = 16 \quad (= 2^{2^{2^{2^0}}} \text{ 4 –"–}) \\
&lg^* N \leq 5 \quad \text{za } N \leq 2^{16} = 65536 \quad (= 5 \ldots) \\
&lg^* N \leq 6 \quad \text{za } N \leq 2^{65536} \quad (= 6 \text{ dvojki})
\end{aligned}$$

Dakle, za sve praktične potrebe, $lg^* N$ možemo smatrati najviše konstantnim (≤ 6), tj. vrijeme za n operacija find3 ili merge3 je praktički linearno.

(Nap: precizna analiza koristi Ackermannovu funkciju i također ne daje linearno vrijeme, u najgorem slučaju.
Također, do danas, nije poznata linearna realizacija operacija find i merge).

[v. Algorithmics, Prob. 1.9.7, →, 1.9.9., p. 34]

Vratimo se analizi kompletnosti Kruskalovog algoritma i to samo u najgorem slučaju.

Neka je $G=(V,E)$ povezan, neusmjeren graf s $n=|V|$ vhora i $a=|E|$ bridova.

U Kruskalovom algoritmu imamo sljedeće operacije:

(a) Uzlazno sortiranje bridova. Može se pokazati da je za sortiranje a objekata, u najgorem slučaju, potrebno vrijeme

$$t_{sort}(a) = O(a \log a)$$

Na pr. to vrijedi za heapsort algoritam. Quicksort je u prosjeku brži, ali u najgorem slučaju zahtijeva vrijeme $O(a^2)$.

U povezanom grafu vrijedi:

$$n-1 \leq a \leq \frac{n(n-1)}{2}$$

pa je

$$\log a = \Theta(\log n)$$

tj.

$$\underline{t_{sort} = O(a \log n)}$$

(b) Inicijalizacija n disjunktnih skupova procedurom init troši vrijeme

$$t_{init} = O(n)$$

(c) pohlepna repeat petlja radi na strukturi disjunktnih skupa nad skupom od n vhora.
U najgorem slučaju, kad pretražujemo baš svih a bridova imamo:

$2a$ find operacija (za svaki brid po dvije)

i točno $n-1$ merge operacija (svaka dodaje po jedan brid u T).

Ukupan broj tih operacija je $2a+n-1$, pa je vrijeme za njih, u najgorem slučaju, dano s

$$t_{find,\,merge} = O((2a+n-1)\lg^* n)$$

jer je broj operacija $2a+n-1$ sigurno veći od broja objekata n, zbog $a \geq n-1$. To ujedno kaže da je

$$t_{find,\,merge} = O(a \lg^* n)$$

(d) sve preostale operacije pojedinačno troše najviše konstantno vrijeme. Repeat petlja se izvršava najviše a puta (prolaz kroz sve bridove), pa je preostalo potrebno vrijeme najviše

$$t_{ostalo} = O(a).$$

- Ukupno potrebno vrijeme je zbroj ova 4 vremena. Očito je

$$\lg^* n = O(\log n)$$

pa je vanjski faktor najviše a, a pod logaritmom je najviše n.

Dakle:

$$\boxed{T_W(n, a) = O(a \log n)}.$$

- Usporedimo to s Primovim algoritmom.
  - Ako je graf G gust (s mnogo bridova), onda je

$$a \approx \frac{n(n-1)}{2}$$

i Kruskalov algoritam troši vrijeme

$$T_W(\text{Kruskal}) = O(n^2 \log n).$$

Dapače, tada je najgori slučaj realističniji, jer ima mnogo bridova za pretragu, pogotovo ako su približno istih duljina.

Primov algoritam zahtijeva vrijeme

$$T_W(\text{Prim}) = O(n^2)$$

i on je, očito, bolji.

  - Ako je graf G rijedak, onda je

$$a \approx n - 1$$

pa Kruskalov algoritam troši samo

$$T_W(\text{Kruskal}) = O(n \log n)$$

i on je, očito, bitno brži od Primovog.

Napomena: Kruskalov algoritam se može još ponešto ubrzati (u prosjeku, iako ne i u najgorem slučaju).

Bridove treba držati u obliku invertiranog heapa (stoga). Inverzija znači da pravilo heapa treba okrenuti, tako da svaki unutrašnji čvor ima vrijednost manju ili jednaku od svoje djece. (zbog uzlaznog sorta)

Bridove ne sortiramo odmah, pa inicijalizacija traje $O(a)$. U repeat petlji modificiramo heap pri traženju najkraćeg preostalog brida. Ta podraga traje $O(\log a) = O(\log n)$ u svakom prolazu.

Ova modifikacija je posebno efikasna kada MST nalazimo relativno brzo, u trenutku dok ostaje još mnogo neobrađenih bridova.

Originalni algoritam tada bespotrebno troši vrijeme na sortiranje tih bridova.

- Za neke grafove postoje i brži algoritmi od Kruskalovog. (Takvi grafovi su dosta česti - jer su mape bliske planarnim grafovima - s relativno malo bridova).

[2.1.2. Najkraći putevi
3. Raspoređivanje (scheduling)

].

2.1.2. Najkraći putevi

Zadan je usmjereni graf $G=(V,E)$ i svaki usmjereni brid $e=(u,v)$ od vrha u do vrha v ima zadanu duljinu (ili cijenu):

$$\ell(e) = \ell((u,v)) \quad \text{(skraćena oznaka je } \ell(u,v)\text{)}$$

Pretpostavljamo da su duljine bridova nenegativne:

$$\ell(e) \geq 0, \quad \forall e \in E.$$

Zadan je još jedan istaknuti vrh $v_0 \in V$, kojeg zovemo izvor ili polazni vrh.

Problem SSSP (Single Source Shortest Paths) – najkraći putevi iz jednog izvora:

Treba naći duljine najkraćih puteva (i same puteve) od izvora do svih preostalih vrhova grafa, koji su dohvatljivi iz izvora.

Naime, neki od preostalih vrhova grafa ne moraju biti dohvatljivi iz izvora, pa algoritam to mora otkriti.

Drugim riječima, algoritam mora vratiti podskup $R \subseteq V$ vrhova koji su dohvatljivi (engl. reachable) iz izvora i vrijednosti

$$d(v) = \text{najkraća udaljenost – duljina najkraćeg puta od } v_0 \text{ do } v, \text{ za } \forall v \in R.$$

Dogovorno, smatramo da je $v_0 \in R$ i $d(v_0)=0$, jer je to najprirodnije za realizaciju algoritma (iako se algoritam može realizirati i bez tog dogovora).

Ovaj problem možemo riješiti pohlepnim algoritmom koji se često naziva Dijkstrin algoritam. Osnovnu formulaciju algoritma predložio je E. W. Dijkstra (1959. g.), a kasnije su predložene još mnoge varijante i ubrzanja.

U općoj terminologiji pohlepnih algoritama:
skup C = skup svih raspoloživih kandidata - vrhova
skup S = skup već izabranih vrhova.

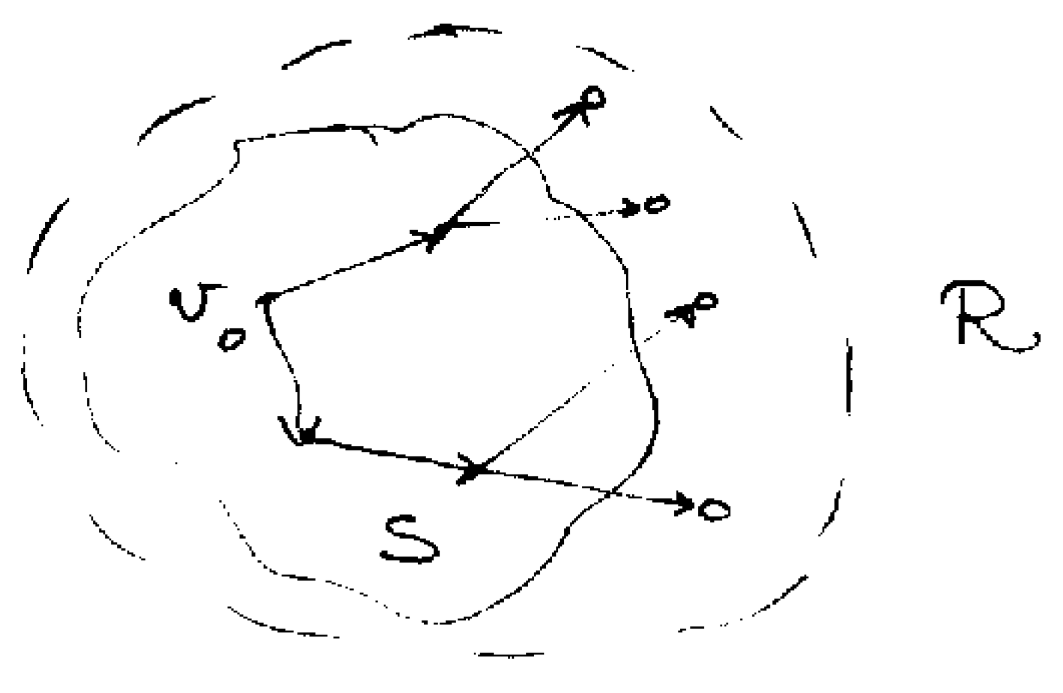
- Na početku algoritma, izabiremo vrh v_0 u S, tj. skup S sadrži samo izvor v_0, a skup C sve ostale vrhove $C = V \setminus \{v_0\}$.

- U prvom koraku, promatramo sve vrhove iz C koji su dohvatljivi iz v_0 i biramo onaj koji je najbliži vrhu v_0. Njega prebacujemo iz C u S.

- Općenito, u svakom koraku, skup S povećavamo pohlepno za po jedan vrh, sve dok je to moguće.
  Ideja: - U svakom koraku, S sadrži sve one vrhove grafa, čija najmanja udaljenost od izvora je već poznata.
  - Iz skupa C, biramo onaj vrh koji je <u>najbliži</u> izvoru (od preostalih) i prebacujemo ga u S. (tj. nema odbacivanja)

  Algoritam staje, kad u C nema više vrhova dohvatljivih iz izvora (tj. funkcija "rješenje", odgovara pojmu "dohvatljiv iz izvora").
  Tj. na kraju je $R = S$.

- Pošto S povećavamo u svakom koraku, isto ćemo napraviti i sa skupom R.
  U svakom koraku, skup R sadrži sve vrhove koji su u S (tj. već dohvaćeni najkraćim putem) ili su direktno - granom dohvatljivi iz nekog vrha iz S.



Zbog toga uvodimo slijedeću definiciju:

<u>Def.</u> Za neki put od izvora v_0 do nekog vrha $v \in V$, kažemo da je <u>S-put</u> (<u>specijalan</u> ili <u>poseban put</u>), ako svi vrhovi na tom putu, osim eventualno zadnjeg vrha v, pripadaju skupu S.

Tj. samo zadnja grana S-puta može izaći iz S, a sve prijašnje su u S.

Dakle, skup R, na svakom koraku algoritma, možemo opisati ovako:

$$R = \{ v \in V \mid \exists \text{ S-put od } v_0 \text{ do } v \}.$$

- Za praćenje duljina najkraćih putova uvodimo polje D, indeksirano vrhovima iz V, s vrijednostima u $\mathbb{R}_0^+$, u oznaci

$$D : \underline{\text{array}}\ V\ \underline{\text{of}}\ \mathbb{R}_0^+ .$$

Njegovo značenje u svakom koraku je:

$D[v]$ = duljina najkraćeg S-puta od v_0 do v, za $\forall v \in R$. (najkraća S-udaljenost).

(za ostale vrhove $v \in C \setminus R$, $D[v]$ nije definiran).

- Sada možemo precizno opisati jedan korak algoritma:
  - Naći $v \in R \setminus S$ s <u>najmanjim</u> $D[v]$ na $R \setminus S$ i dodaj ga skupu S.
  - Ako takav ne postoji, tj. $R = S \Rightarrow$ iz skupa S se ne može van S-putevima $\Rightarrow$ preostali vrhovi iz C su nedohvatljivi iz v_0
  - Ako takav postoji, nakon prebacivanja vrha v iz $R \setminus S$ u S, treba popraviti skup R i polje D.
    (a) nakon što je v u S, duljine nekih S-puteva do vrhova u $R \setminus S$ se mogu skratiti
    (dokazat ćemo da ne treba gledati i vrhove iz S – do njih su putevi već najkraći)
    (b) sve vrhove iz $C \setminus R$ koji su direktno vezani s v, treba prebaciti u R i postaviti pripadne D-ove.

Neformalni zapis ovog algoritma je:

```
procedure Dijkstra ( G=(V,E): graf;
                     l: E -> R0+ : funkcija;  v0 : vrh;
                     var R : skup_vrhova {⊆ V};
                     var D : polje {indeksi iz V,
                                   vrijednosti u R0+ } );
begin
   {inicijalizacija}
  S <- {v0};  D[v0] <- ∅;  { D[v0] nije bitan, ne koristi se }
  R <- {v0};  { stalno držimo S ⊆ R }
  C <- V \ {v0};  { stalno držimo C = V \ S }
  for svi susjedi w ∈ C vrha v0 do
    { to su svi w ∈ C, za koje je (v0,w) ∈ E }
    begin
    R <- R ∪ {w};  { w dohvatljiv iz v0 }
    D[w] <- l(v0,w);
    end;
     {pohlepna petlja}
  while S ≠ R do
    begin
    naći v ∈ R \ S, takav da je D[v] = min_{w ∈ R\S} D[w];
    S <- S ∪ {v};
    C <- C \ {v};      { C = V \ S }
    for svi susjedi w ∈ C vrha v do
       { to su svi w ∈ C, za koje je (v0,w) ∈ E }
      if w ∈ R then
        {već postoji S-put do w, provjeri da li se
         može naći kraći S-put do w, preko v }
        begin
        if D[w] > D[v] + l(v,w) then
          D[w] <- D[v] + l(v,w);
        end
      else  { w ∈ C \ R, prvi puta dohvatljiv kroz v }
        begin
        R <- R ∪ {w};
        D[w] <- D[v] + l(v,w);
        end;
    end; {while}
end; {Dijkstra}
```

Preostaje dokazati da ova pohlepna strategija korektno rješava problem.
Naime, algoritam stvarno nalazi najkraće S-putove do vrhova i to samo prije ubacivanja vrha u skup S. (kad je vrh jednom u S, ne mijenjamo više D)
Treba vidjeti da:
(1) $D[v]$ stalno sadrži duljinu najkraćeg S-puta do v (čim je $v \in R$)
(2) u trenutku kad v dodajemo u S, najkraći S-put do v je i najkraći put od v_0 do v, tj. za $v \in S$, $D[v]$ = duljina najkraćeg puta od v_0 do v.

Dokaz: provodimo ga matematičkom indukcijom po koracima algoritma.

- Baza indukcije - inicijalizacija:
Skup S sadrži samo v_0, pa $D[v_0]$ možemo proizvoljno definirati, jer nema puta od v_0 do v_0.
Skup R sadrži samo vrhove koji su direktno vezani s v_0. Do njih postoji jedan jedini S-put (od samo jedne grane) i taj je, očito, i najkraći.
Dakle, baza vrijedi.

- Korak indukcije:
Pretpostavimo da na početku novog koraka (u točki while) vrijedi:
(a) $\forall u \in S$, $D[u]$ = duljina najkraćeg puta od v_0 do u
(b) $\forall u \in R \setminus S$, $D[u]$ = duljina najkraćeg S-puta od v_0 do u.

Ako je $R = S$, onda smo po (a), za sve vrhove iz $S (=R)$, našli najkraće udaljenosti od v_0. Nadalje, iz S nema niti jedne grane koja završava izvan S, pa su svi vrhovi iz $V \setminus S$ nedohvatljivi.
Dakle, algoritam korektno staje s $R = S$.
(S je tada komponenta povezanosti iz vrha v_0).

Ako je $R \neq S$, onda se izvršava slijedeći korak algoritma, koji dodaje vrh v skupu S.

(a) Novi S je za v veći od ~~starog~~, pa za sve ostale vrhove $u \in S$, $u \neq v$, tvrdnja vrijedi iz pretpostavke indukcije.

Ostaje provjeriti da tvrdnja vrijedi i za v.

→ Po pretpostavci (b), $D[v]$ sigurno daje duljinu najkraćeg S-puta do v. Treba pokazati da je to i najkraća udaljenost od v_0 do v, tj. da najkraći put od v_0 do v ne može prolaziti vrhom izvan S.

Pretpostavimo suprotno, da najkraći put prolazi nekim vrhom izvan S. Neka je $x \notin S$ prvi vrh na koji naiđemo tim putem od v_0 do v. Tada je sigurno $x \in R$, jer postoji S-put od v_0 do x, pa $D[x]$ daje najkraću S-udaljenost od v_0 do x (po pretpostavci (b) indukcije).

No, onda vrijedi:

najkraća udaljenost do v (preko x) $\geq$ udaljenost od v_0 do x duž tog puta $\geq$
(jer bridovi imaju nenegativnu duljinu)

$\geq D[x]$ (jer taj dio puta do x je S-put, a $D[x]$ je najkraća S-udaljenost do x)

$\geq D[v]$ (jer vrijedi $x \in R \setminus S$, a v se bira tako da je
$$D[v] = \min_{w \in R \setminus S} D[w]$$
tj. $D[v] \leq D[x]$ u tom trenu).

Imamo ili kontradikciju (ako je put kroz x kraći) ili put kroz x mora imati istu duljinu kao i ovaj (i još udaljenost od x do v mora biti $\emptyset$).

Time je dokazano da je nađeni S-put do v ujedno i najkraći (ili barem jedan od najkraćih) put od v_0 do v.

(b) Promatrajmo vrh $w \in R \setminus S$. Kad vrh v dodamo skupu S, imamo dvije mogućnosti za najkraći S-put od izvora do w:

- ili se nije promijenio, pa $D[w]$, po pretpostavci (b), već sadrži najkraću S-udaljenost od izvora do w,
- ili se skratio i sad prolazi kroz v.

No, tada v mora biti zadnji vrh iz S na tom putu. Da je neki drugi vrh $x \in S$ zadnji, jer je x već ranije bio u S, mora biti

$$D[x] \leq D[v]$$

što je kontradikcija s pretpostavkom da se najkraći S-put skratio.

Jer je v zadnji vrh iz S na tom putu do w, onda je njegova ukupna duljina $D[v] + \ell(v,w)$ i vrijedi

$$D[v] + \ell(v,w) < D[w]$$

v i w moraju biti vezani, jer je ovo S-put, v zadnji iz S, $w \in R \setminus S$

jer je novi najkraći S-put kraći od starog. Zbog toga popravljamo $D[w]$, ako vrijedi ova nejednakost.

Ovo vrijedi za vrhove w koji su i ranije bili u $R \setminus S$.

- Preostaje popraviti skup $R \setminus S$ za slijedeći korak. U njega treba dodati sve one vrhove $w \in C$, koji ranije nisu bili dohvatljivi iz S, a dodavanjem vrha v u S, to postaju. Tj. u R treba dodati sve vrhove $w \in C \setminus R$ koji su direktno vezani s v (samo jedna grana i to (v,w) može van iz S).

  Najkraći (i jedini) S-put do takvog w ima duljinu

$$D[v] + \ell(v,w)$$

što spremamo u $D[w]$. Q.E.D.

Dakle, na izlazu iz Dijkstrinog algoritma vrijedi

<u>Teorem 3.</u>
$$D[v] = d(v), \quad \forall v \in R,$$

tj. algoritam radi korektno.

<u>Zadatak 8.</u> Provjeri da algoritam radi korektno, ako inicijalizacija glasi samo:

$\{$ inicijalizacija $\}$
$S \leftarrow \emptyset;$
$R \leftarrow \{v_0\};\ D[v_0] \leftarrow 0;$
$C \leftarrow V;\quad \{C = V \setminus S\}$

bez prve <u>for</u> petlje. Tada se <u>while</u> $S \neq R$ <u>do</u>... petlja može pretvoriti u

<u>repeat</u>
...
<u>until</u> $S = R$;

jer u prvom koraku sigurno prolazi kroz petlju, birajući $v = v_0$.

Algoritam tada bitno koristi $D[v_0] = 0$ $\blacksquare$

Također, pokaži da algoritam u zadnjem prolazu kroz while (ili repeat) ne obavlja ništa u <u>for</u> petlji (nema ispravke D i R) $\blacksquare$

– Ostaje riješiti problem nalaženja i vraćanja samih najkraćih puteva. No, iz dokaza korektnosti algoritma, vidi se da je za najkraće S-puteve dovoljno pamtiti <u>zadnji vrh iz S</u> na tom putu. Po tvrdnji (a) dokaza, u trenutku kad v dodamo u S, imamo zapamćenog njegovog neposrednog prethodnika na najkraćem (ne samo S-) putu.

Dakle, čim do vrha w postoji neki S-put (tj. čim je $w \in R$), treba zapamtiti zadnji vrh iz S na tom putu. Popravak tog prethodnika treba vršiti, kad se mijenja najkraći S-put do w. Sve potrebne informacije već postoje u algoritmu.

Za pamćenje puteva uvodimo polje P

P : <u>array</u> V <u>of</u> V

indeksirano vrhovima iz V, s vrijednostima u V.

Značenje:

$P[w] = v \iff$ v je zadnji vrh iz S, na najkraćem S-putu od v_0 do w

i to vrijedi za $\forall w \in R$.

Jedino polazni vrh v_0 nema svog prethodnika, što je najjednostavnije označiti s

$$P[v_0] = v_0 .$$

Ovaj način označavanja je vrlo sličan onom u strukturi disjunktnih skupova – pointer na oca-prethodnika, a korijen je sam sebi oznaka.

U algoritam treba dodati:

- izlazni parametar

  <u>var</u> P : polje-vrhova {indeksi i vrijednosti iz V}

- svagdje gdje se postavlja D za neki vrh, treba postaviti i P:

  - u inicijalizaciji, iza $D[v_0] \leftarrow \emptyset$:

    $$P[v_0] \leftarrow v_0$$

  - u <u>for</u> petlji, u inicijalizaciji, iza $D[w] \leftarrow \dots$,

    $$P[w] \leftarrow v_0$$

  - u <u>while</u> petlji, na dva mjesta postavljamo $D[w]$, a iza tih naredbi treba dodati

    $$P[w] \leftarrow v$$

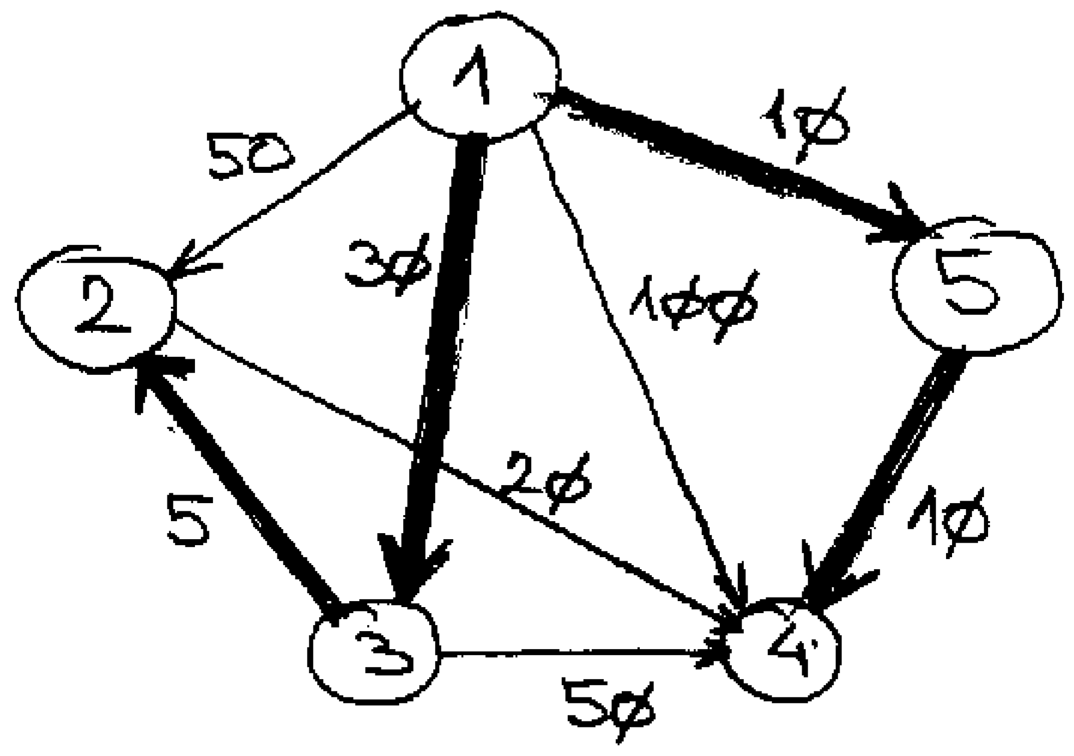
    (na prvom mjestu treba postavljanje D i P zatvoriti u <u>begin</u>, <u>end</u> zagrade).

- Po završetku algoritma, u polju P dobivamo naopako korijensko stablo, s korijenom v_0. Ono reprezentira tzv. <u>stablo najkraćih puteva</u>.

<u>Zadatak 9.</u> Dokaži da najkraći putevi u Dijkstrinom algoritmu (spojeni zajedno) tvore stablo.
Polje P reprezentira upravo to stablo ∎

Zadatak 10. Dokaži da Dijkstrin algoritam nalazi redom najbliže vrhove polaznom vrhu v_0, rastuće po njihovoj najmanjoj udaljenosti od v_0.
Drugim riječima, nakon k koraka ($|S|=k+1$), u skupu S se nalaze v_0 i njemu najbližih k vrhova grafa G.
Tj. Dijkstrin algoritam možemo koristiti i za nalaženje zadanog broja (k) najbližih vrhova polaznom vrhu v_0 ■

Primjer 4. Pokažimo rad Dijkstrinog algoritma za usmjereni graf $G=(V,E)$ s 5 vrhova, i zadanim duljinama kao na slici:



Polazni vrh je $v_0 = 1$.

Korak	Izabrani vrh v u S	skup R skup C	polje D	polje P
inic.	1	$\{1,2,3,4,5\}$ $C=\{2,3,4,5\}$	$[0,50,30,100,10]$ najbliži je 5	$[1,1,1,1,1]$
1	5	—"— $C=\{2,3,4\}$	iz 5 može samo u 4 $[0,50,30,20,10]$ najbliži preostali je 4	$[1,1,1,5,1]$
2	4	—"— $C=\{2,3\}$	nema promjene, jer iz 4 nikamo $[0,50,30,20,10]$ najbliži preostali je 3	$[1,1,1,5,1]$
3	3	—"— $C=\{2\}$	iz 3 može u preostali vrh 2 i to se isplati $[0,35,30,20,10]$	$[1,3,1,5,1]$
4	2	—"— $C=\emptyset$	ovdje nema promjene jer je $C=\emptyset$ $[0,35,30,20,10]$	$[1,3,1,5,1]$
STOP	$S=R=\{1,2,3,4,5\}$			

Stablo najkraćih puteva prikazano je podebljanim granama na slici.

— Prije analize složenosti, pokažimo jednu vrlo jednostavnu implementaciju Dijkstrinog algoritma.

Pretpostavljamo da graf ima $n = |V|$ vrhova i da su vrhovi numerirani brojevima od 1 do n, tj.

$$V = \{1, 2, \dots, n\}.$$

Bridove i pripadne duljine zadajemo (slično kao u Primovom algoritmu) - matricom susjedstva L s duljinama bridova:

$$L(u,v) = \begin{cases} \ell(u,v) & , \text{ako je } (u,v) \in E \quad (\underset{u}{\circ} \longrightarrow \underset{v}{\circ}) \\ \infty & , \text{inače}. \end{cases}$$

Za usmjereni graf, matrica L ne mora biti simetrična. Dijagonalni elementi $L(u,u)$ ove matrice mogu biti bilo koji nenegativni brojevi, bez utjecaja na rad algoritma. U praksi je logično staviti

$$L(u,u) = 0 \text{ ili } \infty.$$

Reprezentacija nepovezanosti duljinom ∞, omogućava da skup R uopće ne moramo pamtiti. Dogovorno smatramo da je stalno $R = V$, tj. svi vrhovi su "dohvatljivi", ali je inicijalno

$$D[w] = \infty, \quad \forall w \neq v_0 .$$

tj. S-putevi mogu imati i beskonačnu duljinu.

<u>Zadatak 11.</u> Vrijednost ∞ u praksi možemo zamijeniti nekim vrlo velikim brojem (maxreal ili čak maxreal / n). Bitno je da korektno možemo simulirati operaciju

$$\infty + \infty = \infty$$

Koliko velika mora biti ta vrijednost, obzirom na zadane postojeće duljine $\ell(u,v)$, pa da algoritam radi korektno?

(Odgovor: $> (n-1) \cdot \max\limits_{(u,v) \in E} \ell(u,v)$ je dobra ocjena, jer je najdulji mogući put najviše ovoliko dug.

Od preostala 2 skupa $C, S \subseteq V$, dovoljno je pamtiti jednog, jer je $C = V \setminus S$. Bolje je pamtiti S, jer njega treba vratiti kao rezultat (tada je $R = S$, pa je i skup S korektan izlaz)!

Algoritam 3. (Dijkstrin algoritam za SSSP)

```
procedure Dijkstra ( n : integer ;  {broj vrhova}
                     v0 : integer ;  {1..n , polazni vrh}
                     L : matrix ;  {array [1..n, 1..n] of real}
                     var S : skup_vrhova;  {set of 1..n}
                     var D : polje_udalj;  {array [1..n] of real}
                     var P : polje_vrhova  {array [1..n] of 1..n}
                   );

var  v, w : integer ;  {1..n}
     mind : real ;  {zajedno s ∞}

begin
   S := {v0};
   for w := 1 to n do
      begin
      D[w] := L[v0, w];
      P[w] := v0 ;        {precizuje : if D[w] < ∞ then P[w] := v0;}
      end;
   D[v0] := 0.0;  {nije bitno}
   repeat  {pohlepna petlja}
      mind := ∞;
      for w := 1 to n do
         if not (w in S) then
            if D[w] < mind then
               begin
               v := w;
               mind := D[v];   {ili D[w]}
               end;
      if mind < ∞ then   {v inače nije definiran, tj. R=S}
         begin
         S := S ∪ {v};
         for w := 1 to n do
            if not (w in S) then
               if mind + L[v, w] < D[w] then
                  begin
                  D[w] := mind + L[v, w];
                  P[w] := v;
                  end;
         end;
   until mind = ∞  {or S = [1..n] se može dodati};
end; {Dijkstra}
```

Složenost algoritma 3:

Inicijalizacija očito troši vrijeme $O(n)$, čak $\Theta(n)$, jer imamo petlju koja se izvršava n puta.

Svaki korak repeat petlje troši, također, vrijeme reda veličine $\Theta(n)$, jer imamo dvije petlje koje se izvršavaju n puta.

Dakle, za cijeli Dijkstrin algoritam vrijedi

$$T(n) = \Theta(n^2)$$

za graf s n vrhova.

U reprezentaciji grafa matricom udaljenosti, bolja ocjena i nije moguća. Matrica L ima n^2 elemenata, i potrebno je barem vrijeme $\Theta(n^2)$ samo za provjeru koji bridovi jesu u E, a koji ne (tj. za obilazak matrice L).

Drugim riječima, za tu reprezentaciju, naša implementacija je optimalna, do na konstantni faktor.

Ta reprezentacija je pogodna i dobra, ako je graf G gust, tj. ima blizu n^2 grana (ili $\Theta(n^2)$ grana).

– Međutim, ako je graf G rijedak, pa za broj grana $e = |E|$ vrijedi
$$e \ll n^2,$$
matrica L sadrži vrlo mnogo vrijednosti ∞, koje nepotrebno provjeravamo.

Mnogo bolje je za svaki vrh pamtiti skup svih vrhova dohvatljivih iz tog vrha, skupa s udaljenostima. Obično se to realizira tako da se pamti lista susjeda s udaljenostima u polju po vrhovima.

Nažalost, niti to nije dovoljno da složenost padne ispod $\Theta(n^2)$.

Traženje susjeda je ubrzano, ali problem ostaje u traženju najbližeg vrha skupu S, tj. operaciji:

nađi $v \in R \setminus S$, takav da je $D[v] = \min_{w \in R \setminus S} D[w]$.

Skup R može vrlo brzo postati bogat s $\Theta(n)$ elemenata, dok je S vrlo mali na početku algoritma.

Klasični algoritam traženja minimuma pretragom cijelog $R \setminus S$, diže složenost na $\Theta(n^2)$.

— Ubrzanje se može postići tako da vrhove grafa iz $R \setminus S$, držimo u obliku invertiranog heapa, uređenog uzlazno po $D[v]$, pa je minimum uvijek na vrhu.

Operacija $S \leftarrow S \cup \{v\}$, izbacuje element s vrha heapa. Dovođenje heapa u uređaj traje $O(\log n)$, jer heap ima najviše n elemenata (čak samo $n-1$, jer v_0 ne treba!)

Dodavanje novog elementa u heap, kad povećavamo skup R, traje također $O(\log n)$.

Takvih operacija ima najviše $2(n-1)$ - za svaki vrh po jedno dodavanje i izbacivanje, pa je potrebno vrijeme

$$T_1 = O(n \log n)$$

Na kraju, svaka promjena vrijednosti $D[w]$, za neki $w \in R \setminus S$, također traži dovođenje heapa u red i traje $O(\log n)$.

No, takva operacija se obavlja najviše jednom za svaki brid (v, w) grafa (ako smo v upravo ubacili u S, a grana (v, w) daje novi najkraći S-put do $w \in R \setminus S$).

Ukupno potrebno vrijeme je

$$T_2 = O(e \log n).$$

Ukupno potrebno vrijeme je onda

$$T(n,e) = O((e+n) \log n)$$

jer sve ostale operacije su elementarne, tj. troše $\Theta(n)$ zbog while petlje.

Za $e \ll n$, ovo je značajno ubrzanje!

— Korištenje Fibonaccijevog heapa daje $T(n,e) = O(e + n \log n)$ (Fredman, Tarjan, 1984. g.)

<u>Zadatak 12.</u> Kako reagira Algoritam 3 na negativne dijagonalne elemente u matrici L?
Može li se Dijkstrin algoritam primijeniti i na graf s negativnim duljinama bridova $\ell(e) < 0$?
Može li se algoritam spasiti tako da se svim bridovima doda neka konstantna duljina, pa da sve duljine postanu nenegativne?

<u>Zadatak 13.</u> Može li se sličan pohlepni algoritam primijeniti za problem traženja <u>najduljih</u> puteva iz jednog izvora?
Da li tada pozitivni ciklusi predstavljaju problem, iako pojam puta ne dozvoljava cikluse (tj. ponavljanje vrhova na putu)?