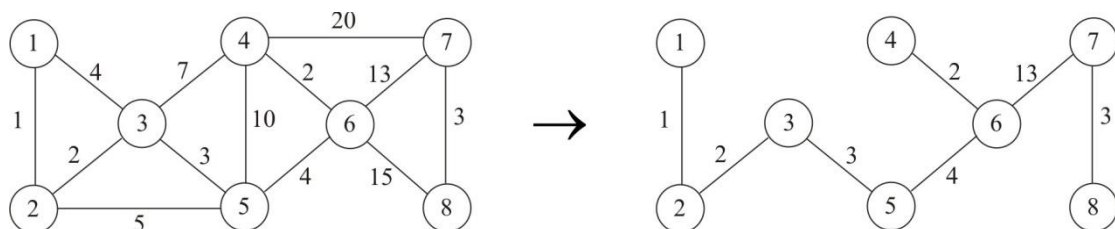


Vaja 3 – Minimalno vpeto drevo (Kruskalov algoritem)

1. Splošna predstavitev problema

Skozi svojo zgodovino se je človek pogosto srečeval s problemom, kako najučinkoviteje med seboj povezati določene točke, v začetku vasi s cestami, ki so morale biti vedno boljše in širše, ko so vasi preraščale v mesta, nato mesta s telefonskimi in električnimi vodi, vodovodnimi cevmi in podobno. Danes se tovrstni problemi kažejo pri povezovanju računalnikov v omrežja, pa tudi iskanju optimalnih letalskih poti med posameznimi mesti ali državami. Pri vseh teh problemih gre za enak problem. Povezati je potrebno vse točke, tako da lahko pridemo iz vsake točke v katerikoli drugo, povezave pa morajo biti take, da je potovanje pri tem najcenejše. Ker gre torej za pogoste probleme, ki so ponavadi povezani s precejšnjimi stroški ne preseneča, da so skozi zgodovino nastajale različne rešitve, s tem problemom pa so se ukvarjali tudi razni matematiki, ki so rešitev problema poiskali v okviru teorije grafov. Posamezne točke so predstavili z vozlišči, povezave med njimi pa z množico povezav v matematičnem smislu, kjer je vsaka povezava določena s krajiščema. Da bi poiskali najcenejšo rešitev pa so povezave opremili tudi s ceno. Na tak način so dobili obtežen graf. Če so vsa vozlišča povezana, temu grafu pravimo povezan obtežen graf. Zgoraj opisan problem lahko predstavimo v teoriji grafov kot iskanje minimalnega vpetega drevesa v danem grafu. Minimalno vpeto drevo je aciklični povezan graf, katerega vsota povezav je minimalna. Minimalno vpeto drevo dobimo z odstranjevanjem odvečnih povezav iz izhodiščnega grafa (glej sliko 1).



Slika 1: Izhodiščni graf in iskano minimalno vpeto drevo

Poznamo več algoritmov za iskanje minimalnega vpetega drevesa. Najbolj znana sta Primov in Kruskalov algoritem, ki sta dobila imena po svojih avtorjih.

V naši vaji se bomo osredotočili na Kruskalov algoritem, ki temelji na strategiji požrešna metoda, kar pomeni, da podatke najprej primerno uredi, nato jih pa jemlje drugega za drugim in se glede na kriterijsko funkcijo odloči, ali bo podatek vključil v rešitev ali pa ga bo zavrnil. Kruskalov algoritem v skladu s to strategijo uredi povezave glede na njihovo ceno v naraščajočem vrstnem redu. Nato jemlje povezavo za povezavo iz predhodno urejenega seznama in jo sprejme, v kolikor nova povezava ne tvori cikla s katero od sprejetih povezav (spomnimo se namreč, da je drevo graf brez ciklov). Kruskalov algoritem gradi rešitev s pomočjo vpetega gozda, to je več vpetih dreves, ki jih med seboj povezuje, dokler ne ostane eno samo drevo. Od tega trenutka dalje so vse

ostale povezave zavrnjene, saj bi tvorile cikel. Da postopek pohitrimo, upoštevamo, da ima drevo $n/2$ povezav, pri čemer je n število vozlišč v grafu. Če želimo pokazati delovanje Kruskalovega algoritma na primeru, se moramo dogovoriti, za oznake, ki jih bomo uporabljali. Vsak graf je določen z dvema množicama: množico vozlišč in množico povezav, predstavimo pa ga lahko tudi z matriko sosednosti, ki je za izhodiščni graf G na sliki 1 naslednja:

$$C = \begin{bmatrix} \infty & 1 & 4 & \infty & \infty & \infty & \infty & \infty \\ & \infty & 2 & \infty & 5 & \infty & \infty & \infty \\ & & \infty & 7 & 3 & \infty & \infty & \infty \\ & & & \infty & 10 & 2 & 20 & \infty \\ & & & & \infty & 4 & \infty & \infty \\ & & & & & \infty & 13 & 15 \\ & & & & & & \infty & 3 \\ & & & & & & & \infty \end{bmatrix}.$$

Matrika C je zgornje trikotna, saj so povezave v grafu G neusmerjene.

Kot smo povedali, Kruskalov algoritem išče končni rezultat preko vpetega gozda, ki se pa na koncu združi v eno samo drevo. Zaradi tega bomo povezave vseh dreves hranili v skupnem seznamu, medtem, ko bomo vozlišča dreves hranili v množicah, ki jih bomo označili s črko S in indeksom, ki bo povedal za katero drevo gre. Tako bo oznaka S_1 pomenila, da gre za vozlišča prvega drevesa, S_2 za vozlišča drugega drevesa itd. Vsako povezavo bomo označili s trojico števil, p , q in $cena$, pri čemer indeksa p in q označujeta krajiščni vozlišči, $cena$ pa je cena povezave.

Imejmo graf G s slike 1, podan z matriko C , iz katere generiramo seznam povezav tako, da pri vsakem končnem členu matrike generiramo povezavo. Krajiščni vozlišči sta določeni z vrstico in stolpcem matrike, cena pa je končna vrednost. Tako dobimo seznam povezav, ki ga uredimo v naraščajočem vrstnem redu (glej tabelo 1).

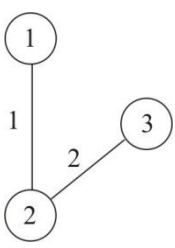
Tabela 1: Urejen seznam povezav izhodiščnega grafa s slike 1 z osmimi vozlišči

<u>p</u>	<u>q</u>	<u>cena</u>
1	2	1
2	3	2
4	6	2
3	5	3
7	8	3
1	3	4
5	6	4
2	5	5
3	4	7
4	5	10
6	7	13
6	8	15
4	7	20

Omenili smo že, da jemljemo povezave iz tabele 1 drugo za drugo in s kriterijsko funkcijo preverjamo, ali povezavo lahko sprejmemo ali ne. Pri tem imamo naslednje možnosti:

1. $p \in S_i$ in $q \in S_i$; vozlišči p in q sta že vključeni v istem drevesu, zato bi sprejetje take povezave nujno vodilo k tvorbi cikla, kar bi uničilo drevo, zato tako povezavo zavrnamo.
2. $p \in S_i$ in $q \in S_j$; povezava $p-q$ povezuje dve vozlišči, ki se nahajata v različnih drevesih. Tako povezavo sprejmemo, saj se v tem primeru drevesi združita v večje drevo, to pa pomeni, da se združita tudi obe množici v eno večjo množico:
 $S_i = S_i \cup S_j$.
3. $p \in S_i$, $q \notin nobeni$ množici oziroma kar je enak primer $q \in S_i$, $p \notin nobeni$ množici; to pomeni, da povezava $p-q$ povezuje vozlišče, ki se nahaja v drevesu z vozliščem, ki do sedaj ni bilo vključeno še nikamor. Tako povezavo sprejmemo, saj to pomeni, da se drevo za eno vozlišče razširi. To pomeni, da moramo povečati tudi množico S_i :
 $S_i = S_i \cup \{q\}$ oziroma $S_i = S_i \cup \{p\}$.
4. p in q nista v nobeni množici, kar pomeni, da smo s povezavo $p-q$ povezali dve nepovezani točki, to pa pomeni, da smo dobili novo drevo. Tako povezavo sprejmemo in ustvarimo novo množico S_k :
 $S_k = \{p, q\}$.

V skladu z definiranimi pravili se sprehodimo skozi seznam v tabeli 1. Prva povezava je povezava 1–2.

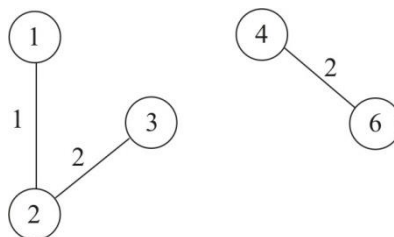
povezava	množica	drevesa	sprejete povezave
a)	$S_1 = \{1, 2\}$		$[1-2]$
1–2			
$1 \notin nobeni$ množici			
$2 \notin nobeni$ množici			
sprejmemo			
b)	$S_1 = \{1, 2, 3\}$		$\begin{bmatrix} 1-2 \\ 2-3 \end{bmatrix}$
2–3			
$2 \in S_1$			
$3 \notin nobeni$ množici			
sprejmemo			

c)

4-6
4 $\notin$ nobeni množici
6 $\notin$ nobeni množici

$S_1 = \{1, 2, 3\}$
 $S_2 = \{4, 6\}$

sprejmemo



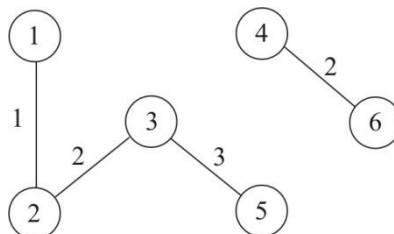
$\begin{bmatrix} 1-2 \\ 2-3 \\ 4-6 \end{bmatrix}$

d)

3-5
 $3 \in S_1$
5 $\notin$ nobeni množici

$S_1 = \{1, 2, 3, 5\}$
 $S_2 = \{4, 6\}$

sprejmemo



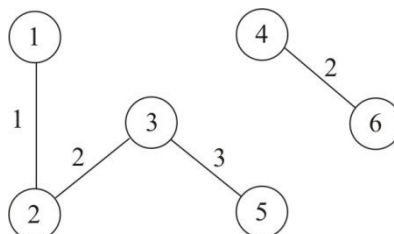
$\begin{bmatrix} 1-2 \\ 2-3 \\ 4-6 \\ 3-5 \end{bmatrix}$

e)

7-8
7 $\notin$ nobeni množici
8 $\notin$ nobeni množici

$S_1 = \{1, 2, 3, 5\}$
 $S_2 = \{4, 6\}$
 $S_3 = \{7, 8\}$

sprejmemo

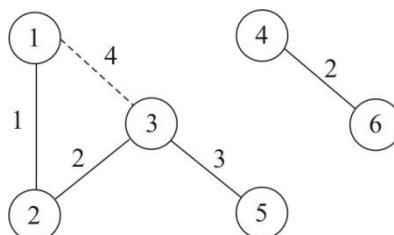


$\begin{bmatrix} 1-2 \\ 2-3 \\ 4-6 \\ 3-5 \\ 7-8 \end{bmatrix}$

f)

1-3
 $1 \in S_1$
 $3 \in S_1$

zavrնemo

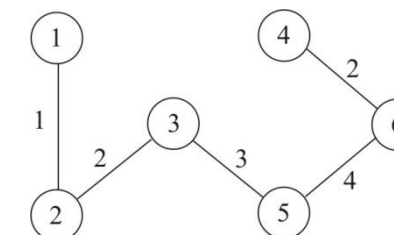


g)

5-6
 $5 \in S_1$
 $6 \in S_2$

$S_1 = \{1, 2, 3, 4, 5, 6\}$
 $S_3 = \{7, 8\}$

sprejmemo

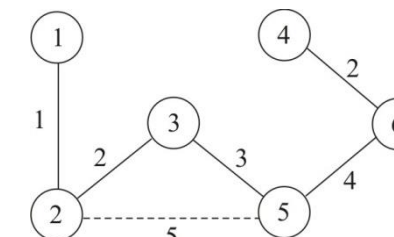


$\begin{bmatrix} 1-2 \\ 2-3 \\ 4-6 \\ 3-5 \\ 7-8 \\ 5-6 \end{bmatrix}$

h)

2-5
 $2 \in S_1$
 $5 \in S_1$

zavrնemo



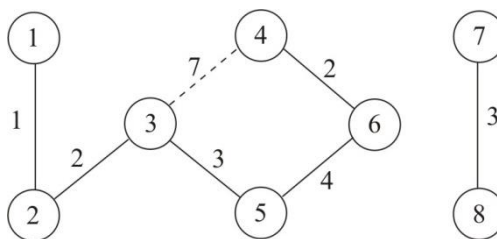
i)

3-4

$3 \in S_1$

$4 \in S_1$

zavrնemo



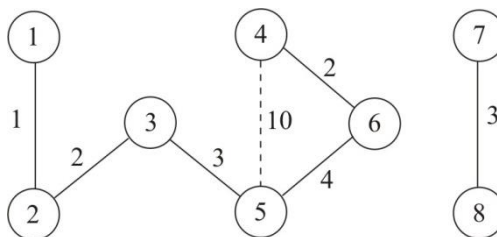
j)

4-5

$4 \in S_1$

$5 \in S_1$

zavrնemo



k)

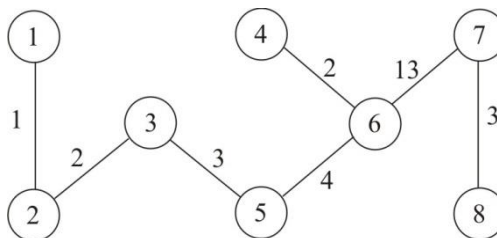
6-7

$6 \in S_1$

$7 \in S_3$

sprejmemo

$S_1 = \{1, 2, 3, 4, 5, 6, 7, 8\}$



$\begin{bmatrix} 1-2 \\ 2-3 \\ 4-6 \\ 3-5 \\ 7-8 \\ 5-6 \\ 6-7 \end{bmatrix}$

S sprejetjem povezave 6-7 smo sprejeli sedem povezav in če upoštevamo, da ima izhodiščni graf G osem vozlišč, smo vpeto drevo našli in z iskanjem lahko zaključimo, saj bodo vse povezave od tod naprej zavrnjene.

2. Pomoč pri implementaciji

Pri sami implementaciji imamo dve možnosti. Problema se lahko lotimo s statičnimi ali dinamičnimi podatkovnimi strukturami. Čeprav imajo dinamične podatkovne strukture svoje prednosti, se bomo raje osredotočili na statične podatkovne strukture, kar pomeni, da bomo povezave vpisovali v statično polje. Pri tem je potrebno najprej določiti maksimalno velikost polja. Pri tem se moramo vprašati, koliko povezav pa lahko pri izhodiščnem grafu sploh pričakujemo. V najslabšem primeru je naš izhodiščni graf poln, to pa pomeni, da ima $n(n-1)/2$ povezav, pri čemer je n število vozlišč. Iz tega sledi, da mora biti velikost polja določena s to vrednostjo in je odvisna od maksimalnega števila vozlišč, ki jih ima lahko vhodni graf. Podatkovna struktura za povezavo je v jeziku C++ naslednja:

```
struct povezava {  
    int p;  
    int q;  
    int cena;  
};
```

Iz opisanega vemo, da je potrebno seznam povezav predhodno urediti. Ob upoštevanju, da smo za hrambo seznama uporabili statično polje, bomo za urejanje uporabili algoritem Hitro uredi, ki ga bomo ustrezno prilagodili, da bomo lahko urejali polje tipa *povezava*.

Obravnava množic vozlišč dreves

Preden zapišemo psevdokod algoritma, moramo določiti na kak način bomo obravnavali množice S . Ker ne vemo kakšno bo njihovo število, niti koliko vozlišč bodo vsebovale, je prva misel podatkovna struktura seznam seznamov. Ker pa je vzdrževanje in iskanje po tej podatkovni strukturi časovno zahtevno, se bomo raje lotili hitrejših variante, ki je obenem tudi enostavnejša za implementacijo. Ideja temelji na dejstvu, da je vsako vozlišče lahko natanko v eni množici ali pa ni povezano. To pomeni, da lahko množice rešimo s pomočjo polja, pri katerem indeks predstavlja vozlišče, vrednost v polju pa številko množice v kateri se vozlišče nahaja. V kolikor želimo v takem polju združiti množici i in j , se je potrebno sprehoditi skozi polje in povsod kjer se pojavi j le – tega zamenjamo z vrednostjo i . Delovanje tega algoritma si oglejmo na že znanem primeru iskanja minimalnega vpetega drevesa iz slike 1.

Na začetku je potrebno vse vrednosti v polju postaviti na vrednost 0, kar pomeni, da vozlišča niso vključena še v nobeno drevo. Začetno stanje polja množice lahko vidimo na sliki 2.

1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0

Slika 2: Začetna inicializacija polja *Množice*

Po sprejetju povezave 1–2 se v elementa 1 in 2 v polju množice vpiše vrednost 1. Polje Množica je sedaj naslednje (glej slika 3):

1	2	3	4	5	6	7	8
1	1	0	0	0	0	0	0

Slika 3: Polje *Množice* po sprejetju povezave 1–2

Stanje v polju *Množice* po sprejetju povezav 2–3 in 4–6 je prikazano na sliki 4.

1	2	3	4	5	6	7	8
1	1	1	2	0	2	0	0

Slika 4: Polje *Množice* po sprejetju povezav 2–3 in 4–6

Po sprejetju povezav 3–5 in 7–8 se število dreves poveča, kar se odraža tudi v polju *Množice*, ki je prikazano na sliki 5.

1	2	3	4	5	6	7	8
1	1	1	2	1	2	3	3

Slika 5: Polje *Množice* po sprejetju povezav 3–5 in 7–8

S sprejetjem povezave 5–6 združimo množici S_1 in S_2 . Stanje v polju *Množice* po združitvi obeh množic je prikazano na sliki 6.

1	2	3	4	5	6	7	8
1	1	1	1	1	1	3	3

Slika 6: Polje *Množice* po sprejetju povezave 5–6

Končno s sprejetjem povezave 6–7 združimo še množici S_1 in S_3 , kar se kaže v polju *Množice* na sliki 7.

1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1

Slika 7: Polje *Množice* po sprejetju povezave 6–7

Iz slike 7 vidimo, da so v polju *Množice* same enice in da bo vsaka povezave povežala vozlišči znotraj istega drevesa, zaradi česar so zavrnjene.

Psevdokod Kruskalovega algoritma

Z razrešitvijo obravnave množic lahko zapišemo Kruskalov algoritem s psevdokodom. Tega najdemo v izpisu 1.

```
function KRUSKAL(P, R)
begin
  HITRO_UREDI(P, 0, št_povezav);
  R := 0;
  koncano = false;
  i := 1;
  št_sprej_pov := 0;
  while not koncano do
    begin
      if not pogoj 1 then
        begin
          if P[i] ustreza pogoju 2 then
            begin
              R := R  $\cup$  {P[i]};
              št_sprej_pov := št_sprej_pov + 1;
              Združi množici;
            end
          else if P[i] ustreza pogoju 3 then
            begin
              R := R  $\cup$  {P[i]};
              št_sprej_pov := št_sprej_pov + 1;
              Pridruži množici vozlišče p ali q;
            end
          else if P[i] ustreza pogoju 4 then
            begin
              R := R  $\cup$  {P[i]};
              št_sprej_pov := št_sprej_pov + 1;
              Ustvari novo množico {p, q};
            end
          end
        end
      end

      if št_sprej_pov = n-1 then
        koncano := true;
      else
        i := i+1;
      end
    end
  end
```

Izpis 1: Psevdokod Kruskalovega algoritma

Pri tem je vhodni parameter P seznam povezav v grafu, parameter R pa je izhodni parameter, v katerega shranjujemo sprejete povezave.

Pred samim iskanjem je potrebno seznam povezav urediti, za ker uporabimo algoritem Hitro uredi. Število sprejetih povezav štejemo s spremenljivko št_sprej_pov . Skozi seznam urejenih povezav se sprehajamo v zanki **while**, dokler ne sprejmemo vseh $n-1$ povezav. Ko je iskanje povezav končano, se procedura konča in vrne seznam sprejetih povezav.

3. Zahteve naloge

Implementirati je potrebno Kruskalov algoritem, ki bo lahko poiskal minimalno vpeto drevo v vhodnem grafu z do 1.500 vozlišč. Za urejanje seznama povezav modificirajte algoritem za hitro urejanje. Za testiranje pravilnosti uporabite rutino za nalaganje zaporedja povezav grafa iz datoteke, ki jo imate priloženo na portalu, omogočita pa tudi generiranje poljubnega polnega grafa, kjer cene povezav generirajte s pomočjo funkcije `rand()`.

Ob zagonu programa se mora zagnati meni, ki je prikazan na sliki 8.

Kruskalov algoritem - izbira: 1 Preberi graf iz datoteke 2 Generiraj naključni graf 3 Poženi 4 Izpis sprejetih povezav 5 Konec Vaša izbira:

Slika 8: Začetni meni, ob zagonu aplikacije

Ob izbiri menijske postavke *Preberi graf iz datoteke* poženite priloženo proceduro, za branje podatkov iz datoteke, ki naj prebere datoteko z imenom *graf.txt*. Pri izbiri menijske postavke *Generiraj naključni graf* mora aplikacija od uporabnika zahtevati število vozlišč grafa, ki pa ne sme biti večje od 1.500, nato pa zgenerira povezave polnega grafa, pri čemer uporabi generator naključnih števil za generiranje cen teh povezav. Ob izbiri postavke *Poženi* se zažene Kruskalov algoritem, uporabniku pa program izpiše koliko povezav in vozlišč je bilo v izhodiščnem grafu in koliko povezav je bilo sprejetih s strani algoritma ter koliko časa je algoritem za to potreboval. Pri izbiri postavke *Izpis sprejetih povezav* je potrebno izpisati povezave, ki so bile sprejete, pri čemer za vsako povezavo izpiše trojko p, q in *ceno*.

Program se konča, ko uporabnik izbere menijsko postavko *Konec*.