# Disjunktne množice in minimalno vpeto drevo

Luka Fürst

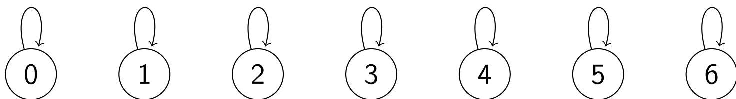
ponedeljek, 13. februarja 2023

# Disjunktne množice

- podana je univerzalna množica $U = \{0, \ldots, n-1\}$
- definiramo $A_i = \{i\}$ za vsak $i \in \{0, \ldots, n-1\}$
- podpreti želimo naslednje operacije:
  - združi množici, ki ji pripadata elementa $x$ in $y$
  - poišči množico, ki ji pripada element $x$
  - preveri, ali elementa $x$ in $y$ pripadata isti množici

# Disjunktne množice

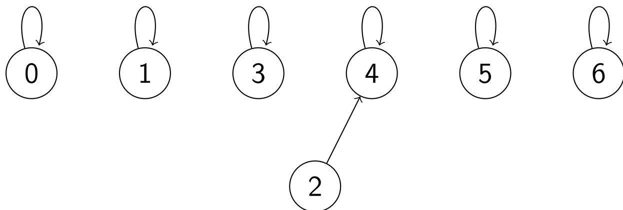
- podatkovna struktura *union-find*
- množice predstavimo z drevesi
  - po eno drevo za vsako množico
- za vsako vozlišče drevesa hranimo njegovega starša
  - koren drevesa je sam svoj starš
- koren drevesa je predstavnik množice



# Operacija združi

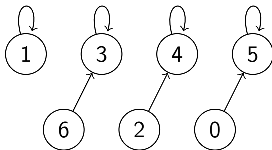
- poiščemo korena dreves, ki jima pripadata elementa
- koren enega od dreves povežemo na koren drugega drevesa

- združi(2, 4)

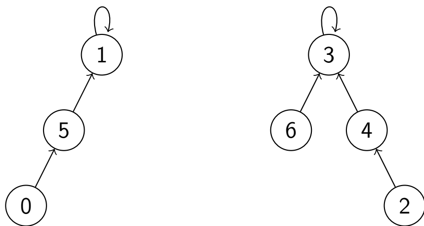


# Operacija združi

- združi(0, 5)
- združi(6, 3)

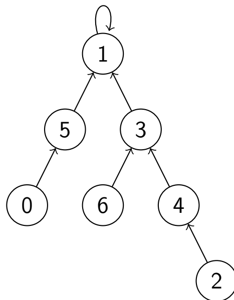



- združi(2, 6)
- združi(5, 1)



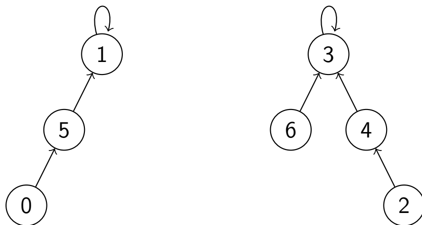
# Operacija združi

- združi(2, 5)



# Operacija poišči

- vrne koren drevesa, ki mu pripada podani element
- sprehodimo se po verigi staršev, dokler ne pridemo do korena



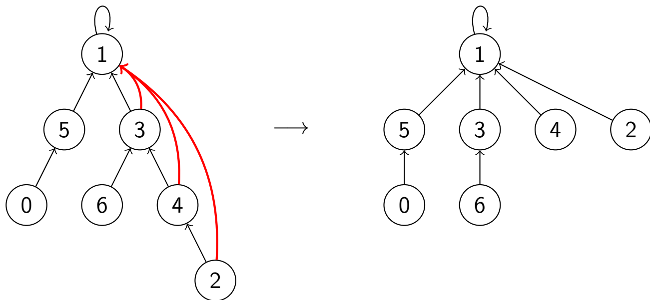

- poišči(1) = poišči(5) = poišči(0) = 1
- poišči(3) = poišči(6) = poišči(4) = poišči(2) = 3

# Operacija preveri pripadnost isti množici

- vrnemo `true` natanko v primeru, če elementa pripadata istemu drevesu (drevesu z istim korenom)
- $\text{preveri_pripadnost}(i, j) \iff \text{poišči}(i) = \text{poišči}(j)$

# Stiskanje poti

- iskanje korena je tem učinkovitejše, čim krajše so poti do korena (tj. čim nižja so drevesa)
- zato se nam med iskanjem korena splača vsa vozlišča na poti prevezati neposredno na koren
- poišči(2):



# Upoštevanje (zgornje meje) višin dreves

- da bo skupna višina drevesa čim nižja, se nam splača nižje drevo pripeti na višje
- za vsako drevo hranimo njegovo višino
  - če uporabljamo stiskanje poti, bo to samo zgornja meja višine
- če imata drevesi enako višino, se višina združenega drevesa poveča za 1, sicer pa ostane enaka

# Podatkovne strukture

- za vsako vozlišče moramo hraniti le njegovega starša
- hranimo še število množic in ocene višin posameznih dreves

```
vector<int> stars;  // stars[i]: starš elementa i
int stMnozic;
vector<int> visina; // visina[i]: ocena višine drevesa s korenom i
```

# Datoteka unionFind.h

```
class UnionFind {
    private:
        int stElementov;
        vector<int> stars;
        int stMnozic;
        vector<int> visina;

    public:
        UnionFind(int stElementov);
        int steviloMnozic();
        void zdruzi(int x, int y);
        bool istaMnozica(int x, int y);
        int poisci(int x);
};
```

# Datoteka unionFind.cpp

```
UnionFind::UnionFind(int stElementov) {
    this->stElementov = stElementov;
    this->stMnozic = stElementov;
    this->stars = vector<int>(stElementov);
    this->visina = vector<int>(stElementov);

    // napolni vektor z elementi 0, 1, ..., stElementov - 1
    // (na začetku je vsak svoj starš)
    iota(this->stars.begin(), this->stars.end(), 0);
}

int UnionFind::steviloMnozic() {
    return this->stMnozic;
}
```

# Datoteka unionFind.cpp

```
int UnionFind::poisci(int x) {
    if (this->stars[x] == x) {
        return x;
    }
    // stiskanje poti
    return this->stars[x] = this->poisci(this->stars[x]);
}

bool UnionFind::istaMnozica(int x, int y) {
    return this->poisci(x) == this->poisci(y);
}
```

# Datoteka unionFind.cpp

```
void UnionFind::zdruzi(int x, int y) {
    int x0 = this->poisci(x);
    int y0 = this->poisci(y);

    // nižje drevo povežemo na koren višjega
    if (this->visina[x0] > this->visina[y0]) {
        this->stars[y0] = x0;
    } else {
        this->stars[x0] = y0;
    }
    if (this->visina[x0] == this->visina[y0]) {
        this->visina[y0]++;
    }

    this->stMnozic--;
}
```

# Časovna zahtevnost operacij

- za praktične potrebe O(1)
- $m$ operacij združi in poišči ima časovno zahtevnost $O(m \log_2^* n)$, kjer nam $\log_2^* n$ pove, kolikokrat moramo število $n$ zaporedoma logaritmirati, da dobimo 1 ali manj
- $\log_2^* n \leq 5$ za vse $n \leq 2^{65536}$

# Problem minimalnega vpetega drevesa

- Vhod: utežen neusmerjen graf $G = (V, E)$
  - $V$: množica vozlišč
  - $E \subseteq V \times V$: množica povezav
  - $(u, v)$ in $(v, u)$ predstavljata isto povezavo
  - $c(u, v)$: cena povezave $(u, v)$
- Izhod: minimalno vpeto drevo $G' = (V, E')$
  - $E' \subseteq E$
  - $G'$ je drevo (acikličen neusmerjen graf)
  - vsota cen povezav drevesa ($\sum_{(u,v) \in E'} c(u, v)$) je minimalna

# Algoritmi

- Primov algoritem
- Kruskalov algoritem

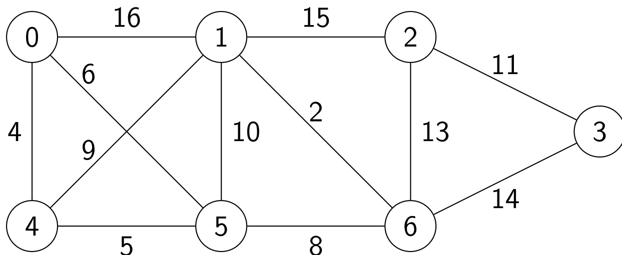
# Primov algoritem

- pričnemo s praznim drevesom
- v drevo dodamo eno od vozlišč (npr. vozlišče 0)
- dokler drevo ne vsebuje vseh vozlišč
  - med vsemi povezavami, ki niso v drevesu, poiščemo najcenejšo povezavo $(u, v)$, tako da je vozlišče $u$ v drevesu, vozlišče $v$ pa ni v drevesu
  - povezavo $(u, v)$ in vozlišče $v$ dodamo v drevo
- dobljeno drevo je minimalno vpeto drevo

# Primov algoritem

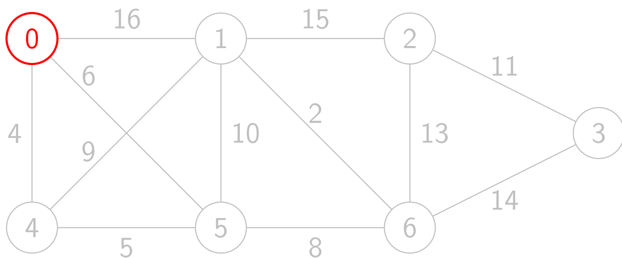
- pomagamo si s prioritetno vrsto povezav med vozlišči v drevesu in vozlišči izven drevesa
- vrsta je urejena po naraščajočih cenah povezav
- v vsaki iteraciji
  - odstranimo prvo povezavo iz vrste (naj bo to povezava $(u, v)$)
  - če vozlišča $v$ še ni v drevesu, potem povezavo $(u, v)$ in vozlišče $v$ dodamo v drevo
  - v vrsto dodamo vse povezave od vozlišča $v$ do vozlišč, ki še niso v drevesu

# Primov algoritem



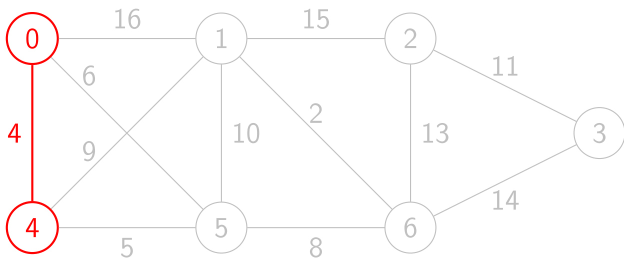

Vrsta:

# Primov algoritem



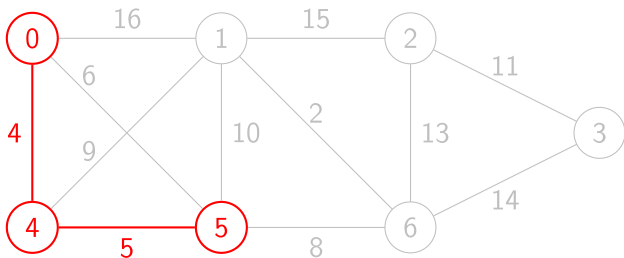

Vrsta: 0–4, 0–5, 0–1

# Primov algoritem



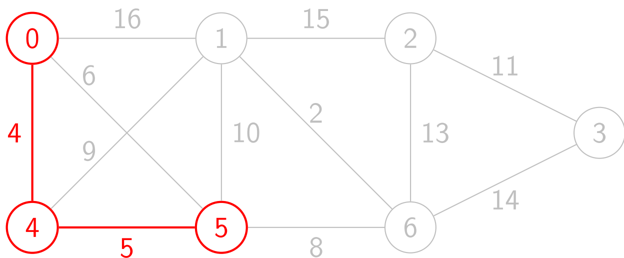

Vrsta: 4–5, 0–5, 4–1, 0–1

# Primov algoritem



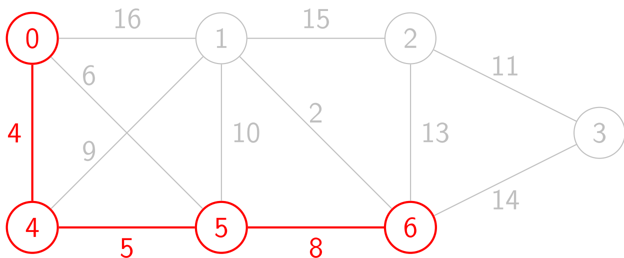

Vrsta: 0–5, 5–6, 4–1, 5–1, 0–1

# Primov algoritem



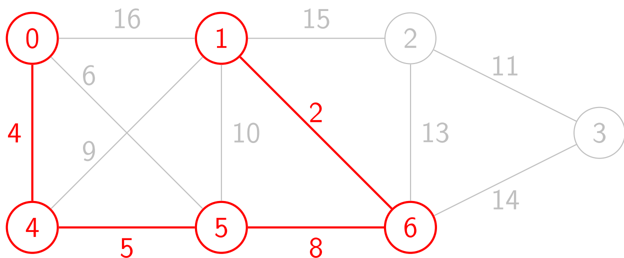

Vrsta: 5–6, 4–1, 5–1, 0–1

# Primov algoritem



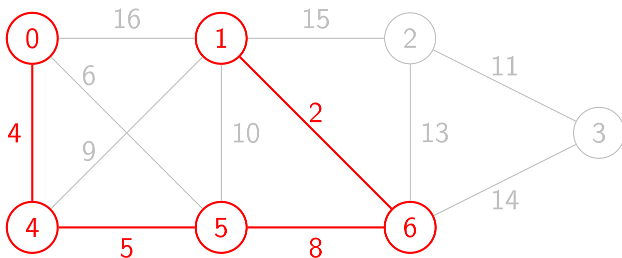

Vrsta: 6–1, 4–1, 5–1, 6–2, 6–3, 0–1

# Primov algoritem



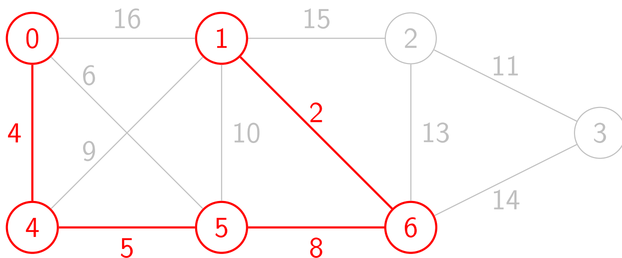

Vrsta: 4–1, 5–1, 6–2, 6–3, 1–2, 0–1

# Primov algoritem



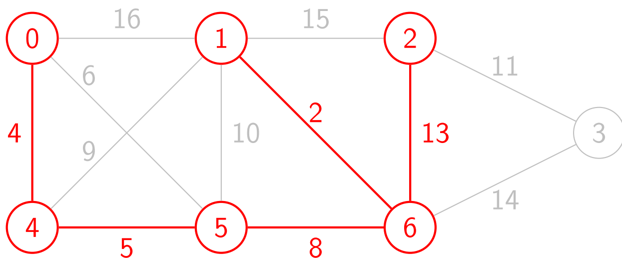

Vrsta: 5–1, 6–2, 6–3, 1–2, 0–1

# Primov algoritem



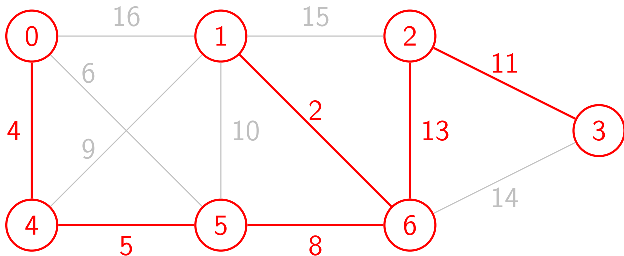

Vrsta: 6–2, 6–3, 1–2, 0–1

# Primov algoritem




Vrsta: 2–3, 6–3, 1–2, 0–1

# Primov algoritem




Vrsta: 6–3, 1–2, 0–1

# Primov algoritem — podatkovne strukture

- graf predstavimo kot seznam sosednosti
  - `vector<vector<pair<int, int>>> graf;`
  - `graf[`$u$`]` je vektor parov $(v, c)$, kjer je $v$ sosed vozlišča $u$, $c$ pa cena povezave $(u, v)$
- prioritetna vrsta trojic $(c, u, v)$
  - $u, v \in V$
  - $c = c(u, v)$
  - urejena po naraščajočih cenah
- vektor, ki za vsako vozlišče pove, ali je že del drevesa
  - `vector<bool> vDrevesu;`
  - `vDrevesu[`$u$`] == true` natanko v primeru, če je vozlišče $u$ del drevesa
- drevo predstavimo kot vektor povezav (parov $(u, v)$)

# Primov algoritem — implementacija (1/2)

```
using ii = pair<int, int>;
using iii = tuple<int, int, int>;
using pq = priority_queue<iii, vector<iii>, greater<iii>>;

vector<ii> minimalnoVpetoDrevo(const vector<vector<ii>>& graf) {
    int stVozlisc = graf.size();
    vector<bool> vDrevesu(stVozlisc);
    pq vrsta;                   // prioritetna vrsta trojic (c, u, v)
    vector<ii> rezultat;        // minimalno vpeto drevo

    // vozlišče 0 dodamo v drevo, njegove povezave pa v PV
    vDrevesu[0] = true;
    for (auto [sosed, cena]: graf[0]) {
        vrsta.push(iii(cena, 0, sosed));
    }
    ...
}
```

# Primov algoritem — implementacija (2/2)

```
vector<ii> minimalnoVpetoDrevo(const vector<vector<ii>>& graf) {
    ...
    int stDodanihPovezav = 0;
    while (stDodanihPovezav < stVozlisc - 1) {
        auto [cena, izvor, cilj] = vrsta.top();
        vrsta.pop();    // vzamemo najcenejšo povezavo iz PV
        if (!vDrevesu[cilj]) {
            vDrevesu[cilj] = true;   // posodobi drevo
            rezultat.push_back(ii(izvor, cilj));
            for (auto [sosed, cena]: graf[cilj]) { // posodobi PV
                if (!vDrevesu[sosed]) {
                    vrsta.push(iii(cena, cilj, sosed));
                }
            }
            stDodanihPovezav++;
        }
    }
    return rezultat;
}
```

# Primov algoritem — časovna zahtevnost

- naj bo $n = |V|$ in $m = |E|$
- dodamo $m$ povezav v PV $\rightarrow O(m \log m)$
- največ $m$-krat odvzamemo povezavo iz PV $\rightarrow O(m \log m)$
- $O(m \log m) = O(m \log n^2) = O(2m \log n) = O(m \log n)$

# Kruskalov algoritem

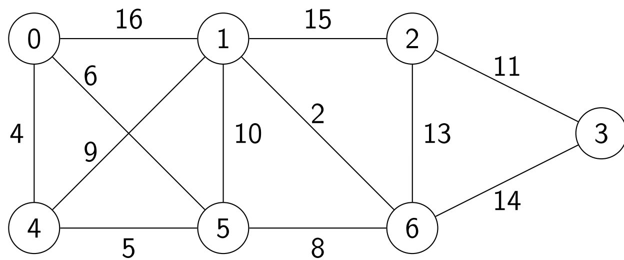
- na začetku je vsako vozlišče svoje drevo
- dokler nimamo enega samega drevesa
  - poiščemo vozlišči $u$ in $v$, tako da pripadata različnima drevesoma in da je $c(u, v)$ minimalna
  - združimo drevesi, ki jima pripadata vozlišči $u$ in $v$
- dobljeno drevo je minimalno vpeto drevo

# Kruskalov algoritem

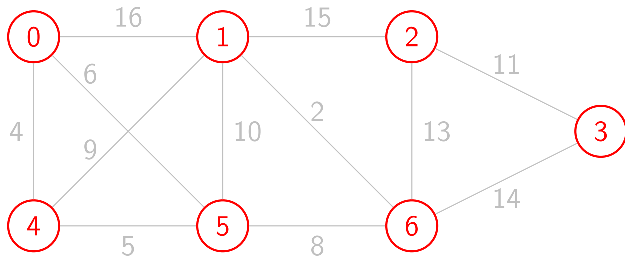
- vzdržujemo prioritetno vrsto povezav, urejeno po naraščajočih cenah
- v vsaki iteraciji iz vrste odstranimo najcenejšo povezavo
- če odvzeta povezava povezuje vozlišči, ki pripadata različnima drevesoma, jo uporabimo za združitev dreves

# Kruskalov algoritem

Vrsta:



# Kruskalov algoritem




Vrsta:

1–6

0–4

4–5

0–5

5–6

1–4

1–5

2–3

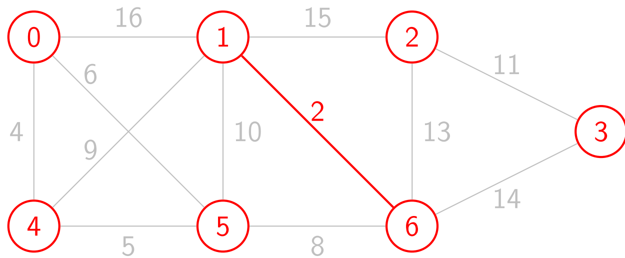
2–6

3–4

1–5

0–1

# Kruskalov algoritem




Vrsta:

~~1–6~~

0–4

4–5

0–5

5–6

1–4

1–5

2–3

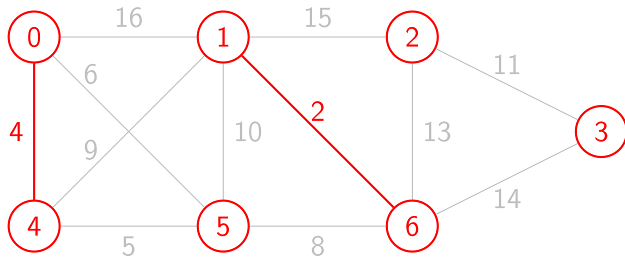
2–6

3–4

1–5

0–1

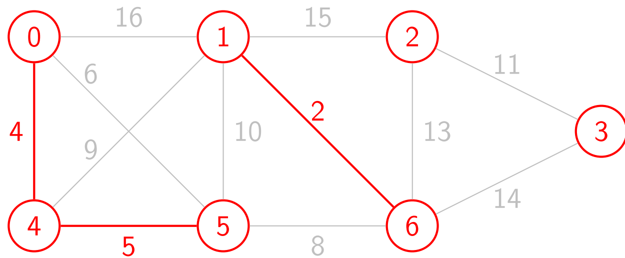
# Kruskalov algoritem




Vrsta:

~~1–6~~
~~0–4~~
4–5
0–5
5–6
1–4
1–5
2–3
2–6
3–4
1–5
0–1

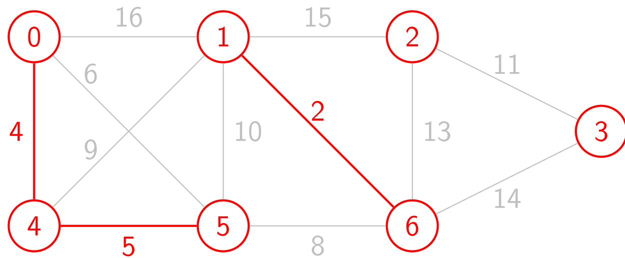
# Kruskalov algoritem




Vrsta:

~~1–6~~
~~0–4~~
~~4–5~~
0–5
5–6
1–4
1–5
2–3
2–6
3–4
1–5
0–1

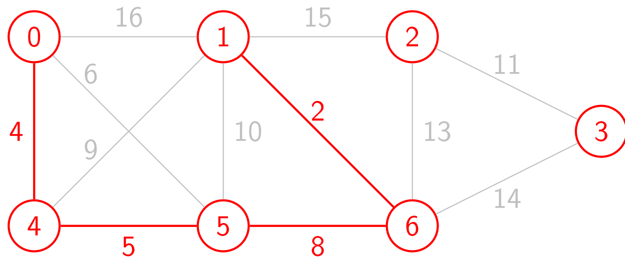
# Kruskalov algoritem




Vrsta:

~~1–6~~
~~0–4~~
~~4–5~~
~~0–5~~
5–6
1–4
1–5
2–3
2–6
3–4
1–5
0–1

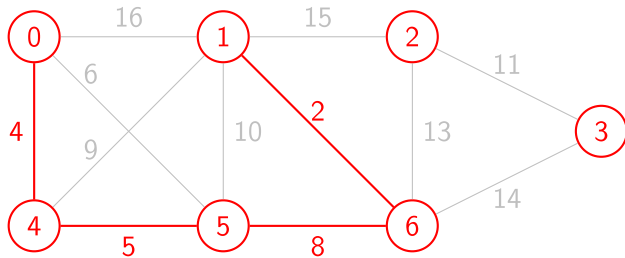
# Kruskalov algoritem




Vrsta:

~~1–6~~
~~0–4~~
~~4–5~~
~~0–5~~
~~5–6~~
1–4
1–5
2–3
2–6
3–4
1–5
0–1

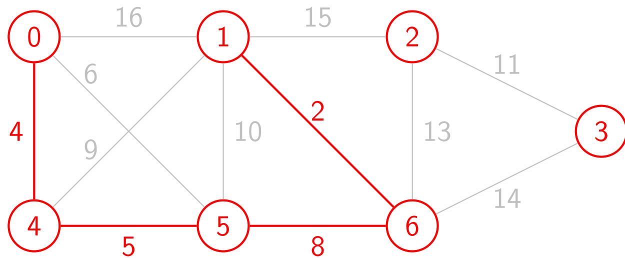
# Kruskalov algoritem




Vrsta:

~~1–6~~
~~0–4~~
~~4–5~~
~~0–5~~
~~5–6~~
~~1–4~~
1–5
2–3
2–6
3–4
1–5
0–1

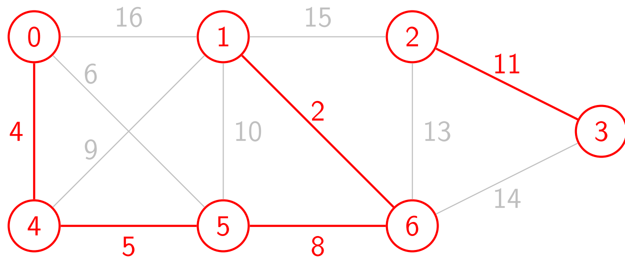
# Kruskalov algoritem




Vrsta:

~~1–6~~
~~0–4~~
~~4–5~~
~~0–5~~
~~5–6~~
~~1–4~~
~~1–5~~
2–3
2–6
3–4
1–5
0–1

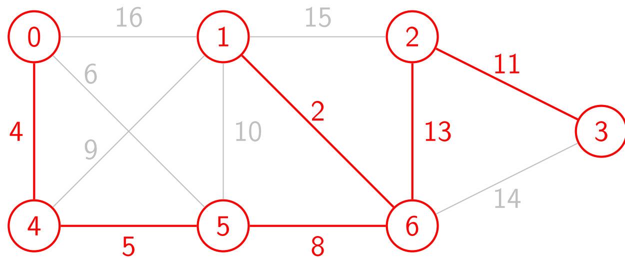
# Kruskalov algoritem




Vrsta:

~~1–6~~
~~0–4~~
~~4–5~~
~~0–5~~
~~5–6~~
~~1–4~~
~~1–5~~
~~2–3~~
2–6
3–4
1–5
0–1

# Kruskalov algoritem




Vrsta:

~~1–6~~

~~0–4~~

~~4–5~~

~~0–5~~

~~5–6~~

~~1–4~~

~~1–5~~

~~2–3~~

~~2–6~~

3–4

1–5

0–1

# Kruskalov algoritem — podatkovne strukture

- prioritetna vrsta trojic $(c, u, v)$
  - $u, v \in V$
  - $c = c(u, v)$
  - urejena po naraščajočih cenah
- disjunktna množica dreves (gozd)
  - podatkovna struktura *union-find*
- drevo predstavimo kot vektor povezav (parov $(u, v)$)

# Kruskalov algoritem — implementacija

- funkcija, ki na podlagi vhodnega grafa izdela PV

```
pq graf2pv(int& stVozlisc) {
    int stPovezav;
    cin >> stVozlisc >> stPovezav;
    pq vrsta;
    for (int i = 0;  i < stPovezav;  i++) {
        int u, v, teza;
        cin >> u >> v >> teza;
        vrsta.push(iii(teza, u, v));
    }
    return vrsta;
}
```

# Kruskalov algoritem — implementacija

```
vector<ii> minimalnoVpetoDrevo(int stVozlisc, pq& vrsta) {
    UnionFind gozd(stVozlisc);
    vector<ii> rezultat;

    while (gozd.steviloMnozic() > 1) {
        auto [teza, u, v] = vrsta.top();
        vrsta.pop();
        if (!gozd.istaMnozica(u, v)) {
            gozd.zdruzi(u, v);
            rezultat.push_back(ii(u, v));
        }
    }
    return rezultat;
}
```

# Kruskalov algoritem — časovna zahtevnost

- dodamo $m$ povezav v PV $\rightarrow O(m \log m)$
- odvzamemo $m$ povezav iz PV $\rightarrow O(m \log m)$
- izvedemo $m$ operacij združi in poišči $\rightarrow O(m \log^* n)$
- skupaj $O(m \log m) = O(m \log n)$