11. prednáška (15. 5. 2012)

Greedy algoritmy

Obsah

- **Greedy stratégia, greedy algoritmus**
- **Minimálna kostra grafu**
- **Úloha o zastávkach autobusu**
- **Problém plnenia batoha**
- **Jednoduchý rozvrhový problé**

Motivácia

Príklad 1. Veštica Teodora, ktorá sa nikdy nemýli, nám predpovedala, ako sa bude vyvíjať cena zlata v nasledujúcich dňoch. Ako s ním obchodovať, aby sme si čo najviac zarobili?

- Máme v hotovosti 300 eur. Dnešná cena zlata je 34 eur za gram. V nasledujúcich dňoch sa táto cena bude meniť nasledovne: zajtra 32 eur/g, pozajtra 30 eur/g, v ďalších dňoch to bude postupne 35, 33, 32, 38, 40 a 37 eur/g.

Motivácia

- Viete mať na konci posledného dňa 400 eur? A dá sa dosiahnuť ešte viac?
- **Ako postupovať optimálne?**
- Počkáme, kým cena klesne na 30 eur za gram. Vtedy nakúpime za všetky peniaze 10 gramov zlata.
- Na druhý deň ho zase všetko predáme, takže budeme mať 350 eur. Atď.

Pozorovanie

- V našej stratégii na riešenie úlohy sa striedajú dva kroky:
- počkáme na lacné zlato a nakúpime;
- počkáme na drahé zlato a predáme.
- Ako ale exaktne definovať, čo je „lacné zlato , a teda kedy nakupovať a kedy predávať?

Pažravá stratégia

- Cena zlata sa mení v noci.
- Teda každý večer môžeme pažravo (nenásytne) rozhodnúť, či chceme zlato alebo peniaze, a podľa toho nakúpiť alebo predať.

- Stretávame sa so situáciou, kedy sme **globálne optimálne riešenie zostrojili tak, že sme postupne urobili niekoľko lokálne optimálnych rozhodnutí.**

Greedy (pažravá) stratégia

- Príklad 2: Predpokladajme, že budeme platiť mincami. Naše mince majú hodnoty 25¢, 10¢, 5¢ a 1¢ (máme ich dosť), a predpokladajme, že je potrebné zaplatiť 63¢. Chceme pri platbe použiť čo najmenej mincí.
- Zvolíme nasledujúci algoritmus: zaplatíme najväčšou mincou nie väčšou ako 63¢, pridáme ju do zoznamu mincí, ktorými platíme a odpočítame jej hodnotu od 63¢, dostaneme 38¢. Potom vyberieme najväčšiu mincu, ktorej hodnota nie je väčšia ako 38¢, pridáme do zoznamu mincí, ktorými platíme, atď.
- Táto metóda charakterizuje *greedy stratégiu (algoritmus)*

Greedy stratégia

- Poznamenajme, že greedy stratégia v tomto prípade poskytne optimálne riešenie (vďaka vhodným hodnotám mincí).
- Ak by sme mali hodnoty mincí 1¢, 5¢, a 11¢ a mali by sme zaplatiť 15¢, podľa tejto stratégie by sme vybrali mincu 11¢ a potom štyri mince 1¢, celkom 5 mincí.
- Je toto riešenie optimálne? Nie. Tri mince 5¢ by stačili.

Greedy algoritmy

Optimalizačné problémy riešené pomocou postupností výberov, ktoré sú:

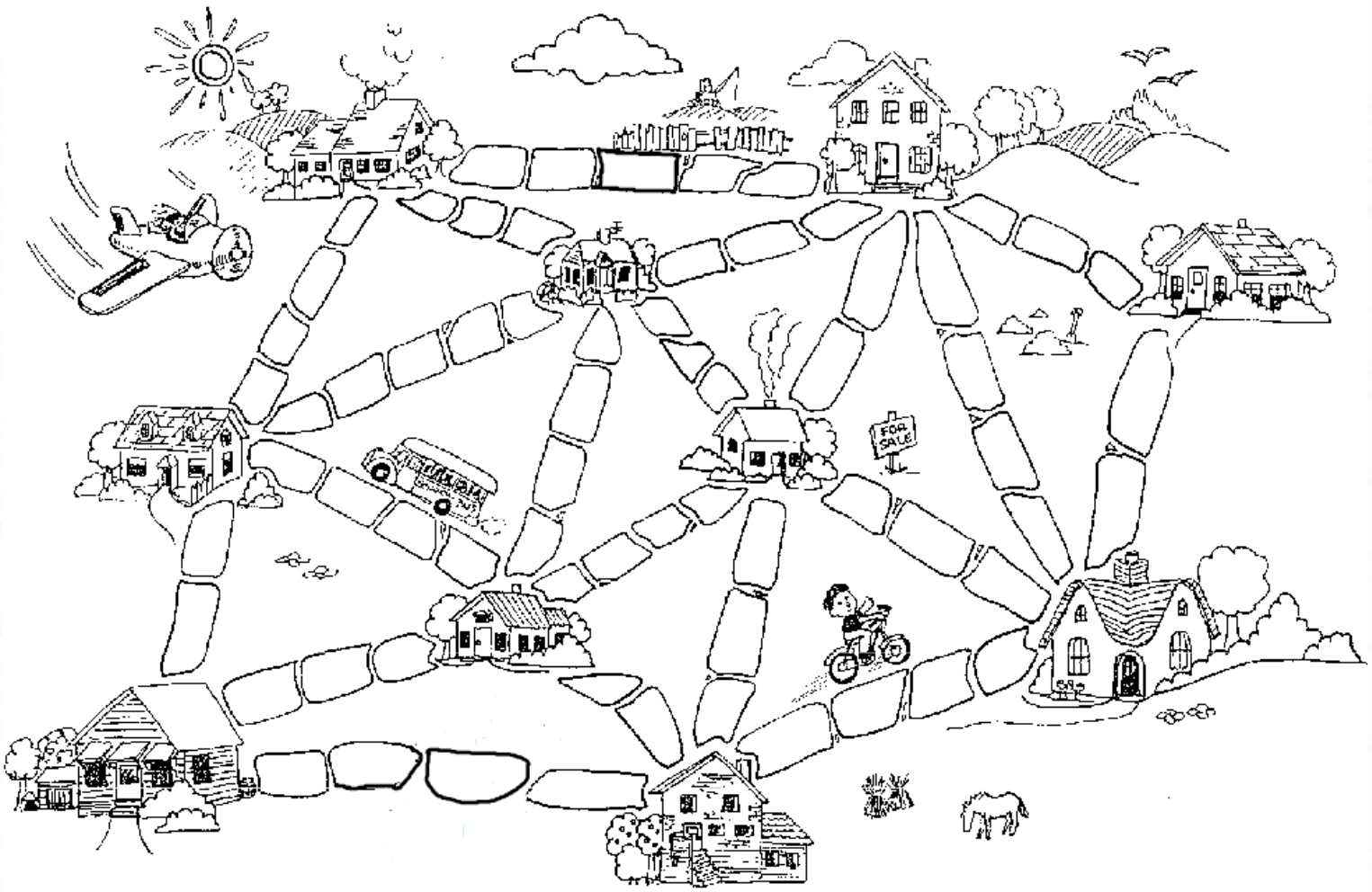
- *Realizovateľné – musia spĺňať obmedzenia problému.*
- *Lokálne optimálne – musia poskytovať najlepší lokálny výber medzi všetkými možnými výbermi v tomto kroku.*
- *Nezrušiteľné – raz urobený výber prvkov do postupnosti je nezmeniteľný.*

Nie všetky optimalizačné problémy môžu byť riešiteľné týmto prístupom!

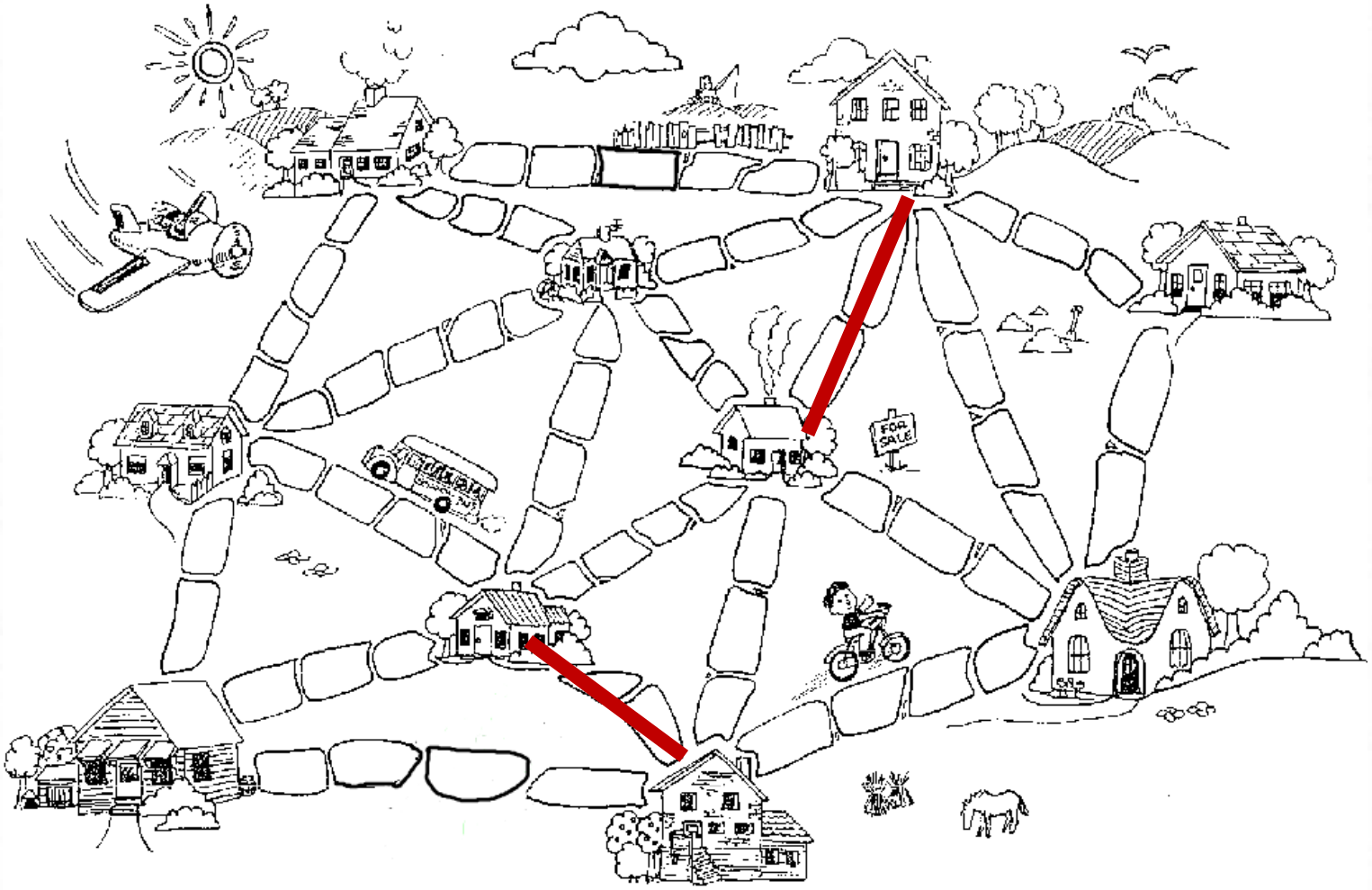
Čo sa deje

- Každá iterácia v greedy algoritme pozostáva z nasledujúcich častí:
  - Procedúra výberu - vyberá ďalšiu položku.
  - Kontrola realizovateľnosti - určuje, či rozhodnutie výberu vytvára lokálne optimálne riešenie.
  - Kontrola riešenia - určuje, či je riešenie už dosiahnuté alebo nie.

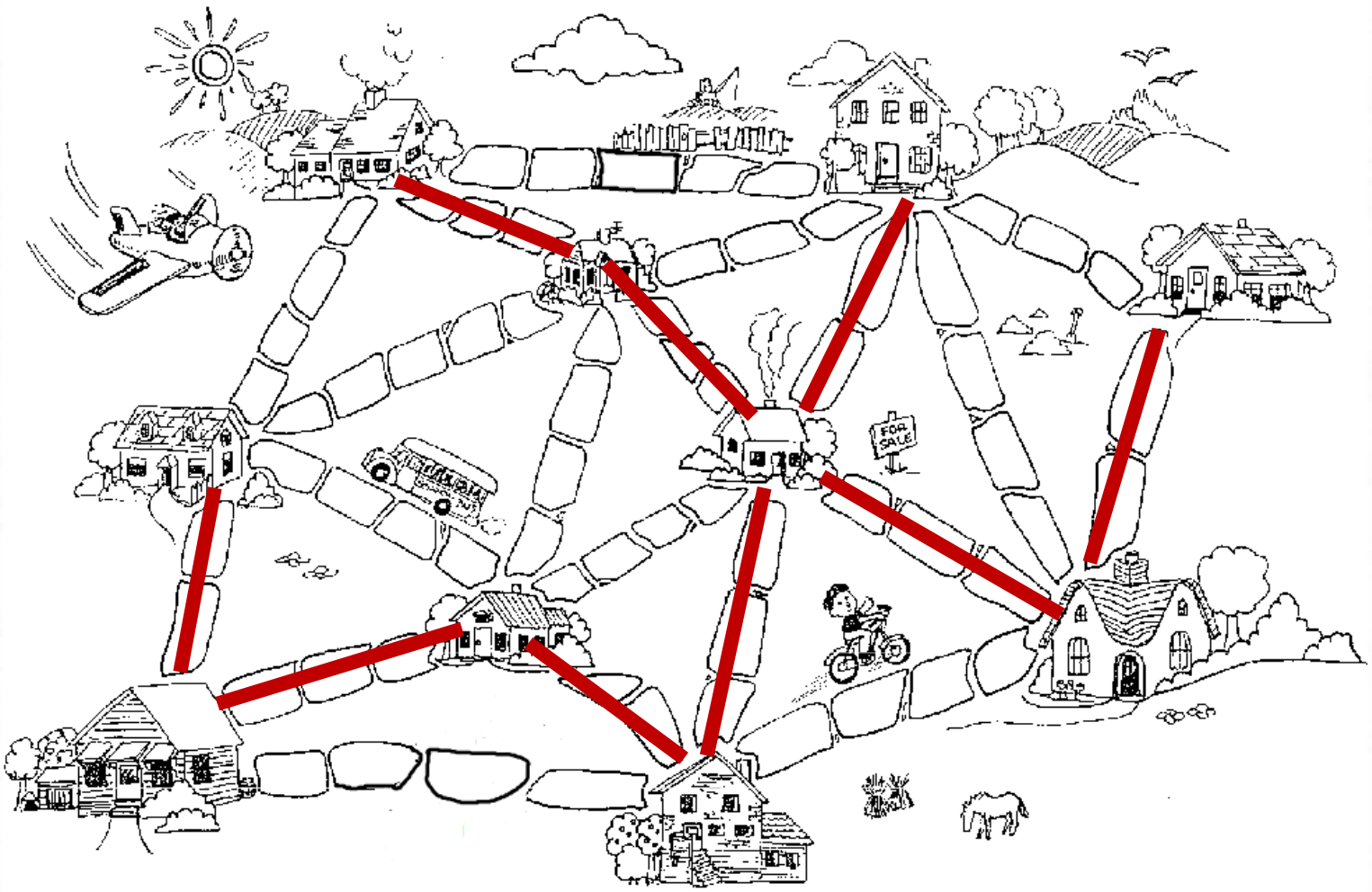
Úloha o dláždení



Úloha o dláždení



Úloha o dláždení



Problém minimálnej kostry – motivácia

- Uvažujme dopravnú sieť
- Máme nejaké peniaze z EÚ na výstavbu diaľnic
- Ktoré cesty máme prerobiť na diaľnice, aby existovalo (nie nutne priame) **spojenie medzi každými 2 mestami** výhradne po diaľniciach a pritom aby sme minuli, čo najmenej
- Vstup: pre každý úsek spájajúci 2 mestá náklady na jeho prestavbu na diaľnicu

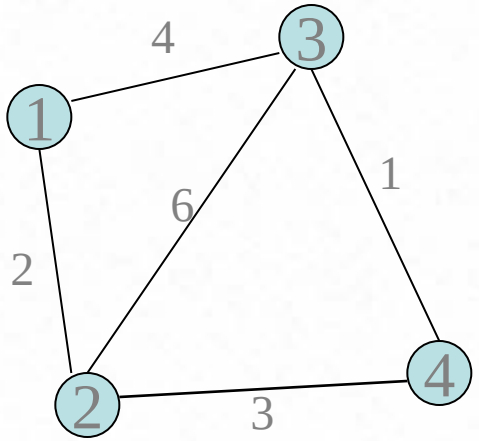
Iné praktické motivácie

- Telefónna sieť: ktoré linky prebudovať na optické, aby bolo možné presmerovať všetky hovory po optických linkách a chceme pritom minúť čo najmenej ?
- Algoritmus na riešenie ako prvý navrhol v roku 1926 Otakar Borůvka, keď riešil problém efektívnej konštrukcie elektrickej siete na Morave

Minimálna kostra grafu

- *Kostra grafu (spanning tree)* súvislého grafu *G*: súvislý acyclický podgraf grafu *G, ktorý obsahuje všetky vrcholy grafu G*.
- *Minimálna kostra grafu (minimal spanning tree - MST)* hranovo ohodnoteného súvislého grafu *G*: je to kostra grafu *G* s minimálnym ohodnotením.
- Príklad:
- Do kostry patria hrany

  s ohodnotením 1, 2, 3

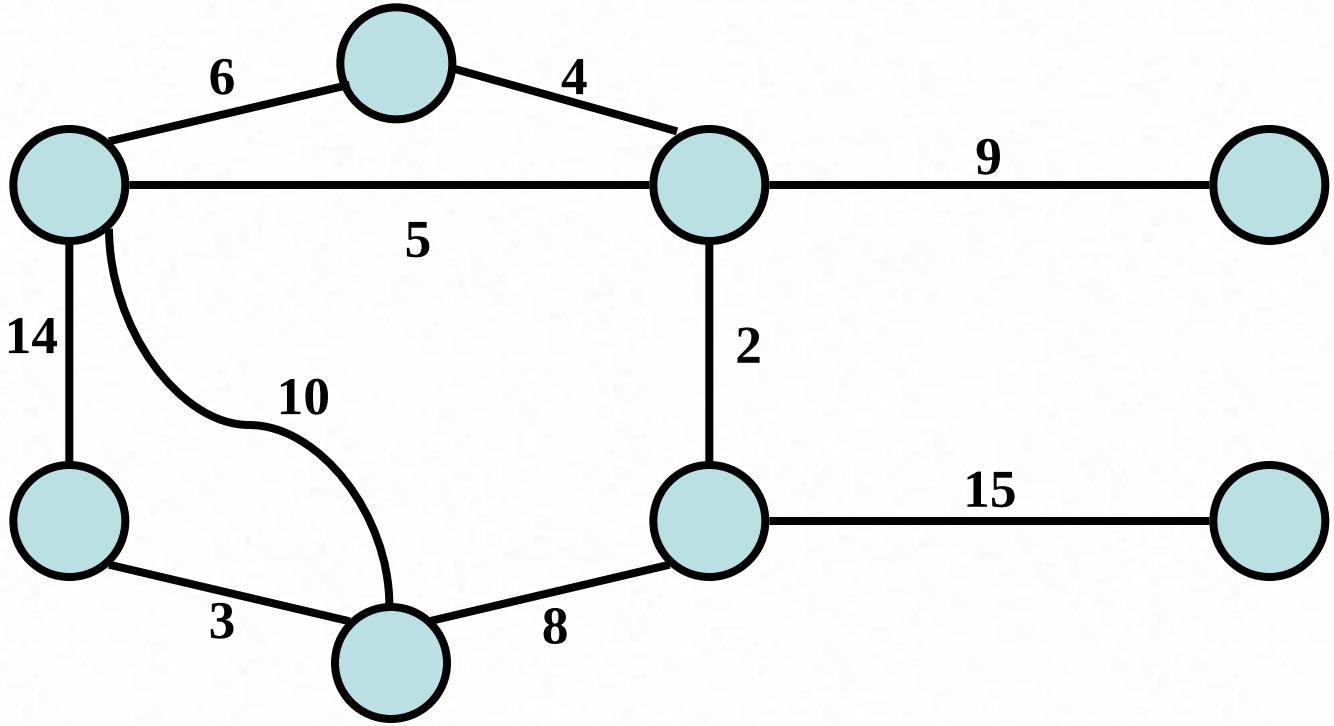


Vlastnosti kostier (bez dôkazu)

- Graf môže mať veľa kostier
- Každá kostra grafu s *n* vrcholmi má práve *n-1* hrán
- Hrany kostry nevytvárajú cyklus
- Pridanie ľubovoľnej nekostrovej hrany ku kostre vytvorí cyklus
- Medzi každými 2 vrcholmi grafu existuje jediná cesta využívajúca len kostrové hrany
- Minimálna kostra: taká kostra, v ktorej súčet ohodnotení (váh) hrán je minimálny spomedzi všetkých možných kostier grafu

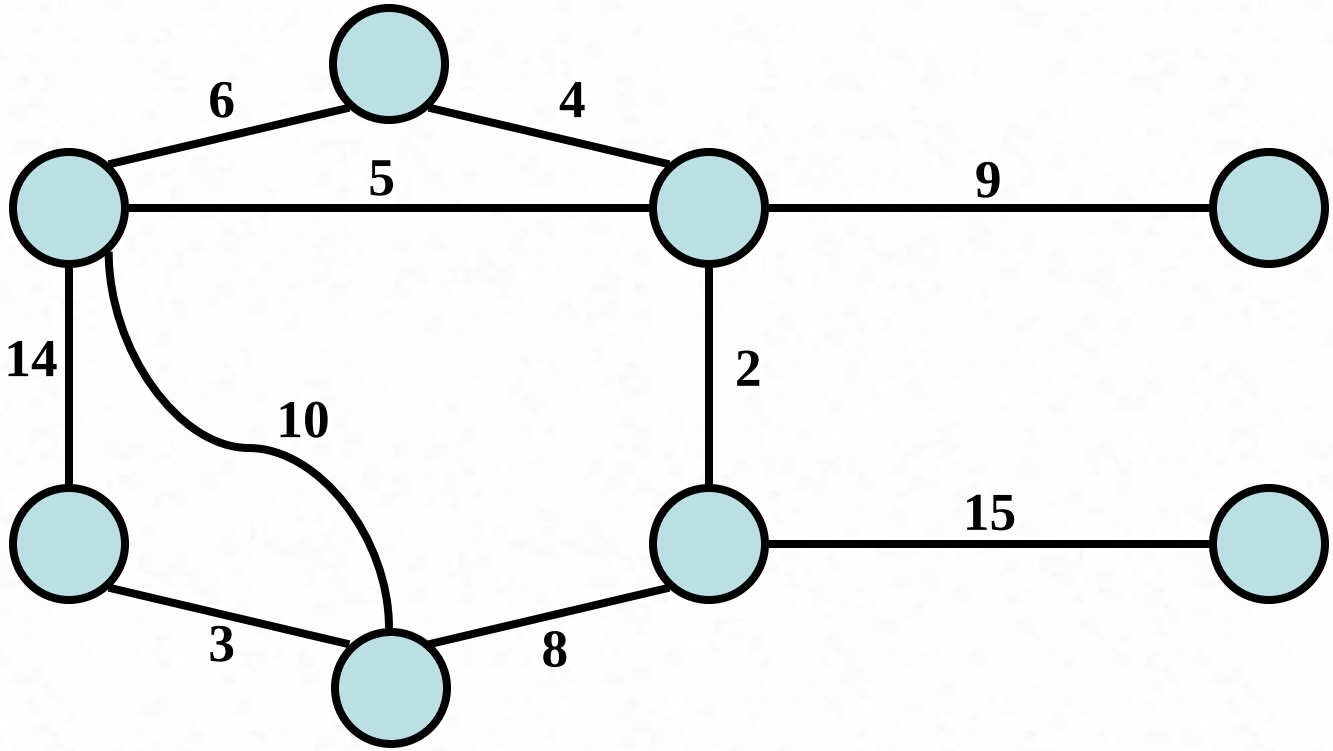
Minimálna kostra grafu

- Problém: daný je súvislý, neorientovaný, hranovo ohodnotený graf:



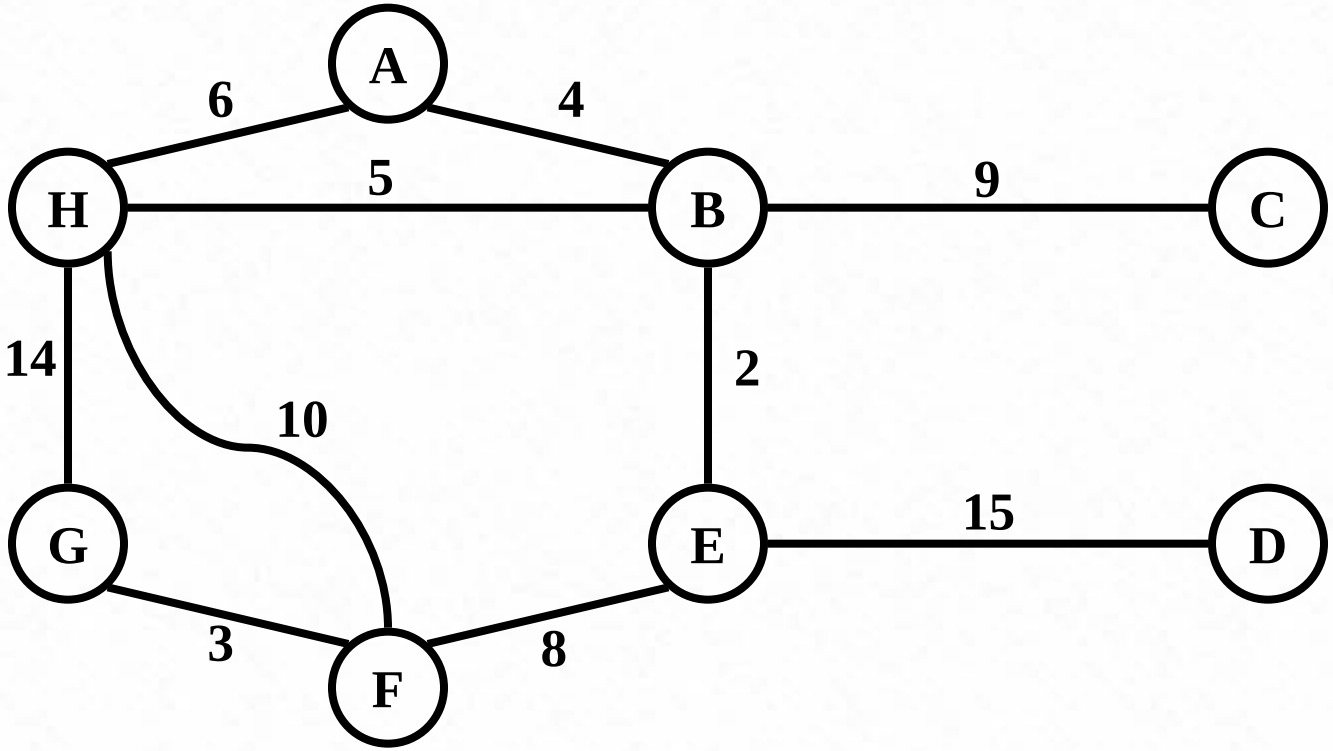
Minimálna kostra grafu

- nájsť kostru použitím hrán, ktoré minimalizujú celkové ohodnotenie.



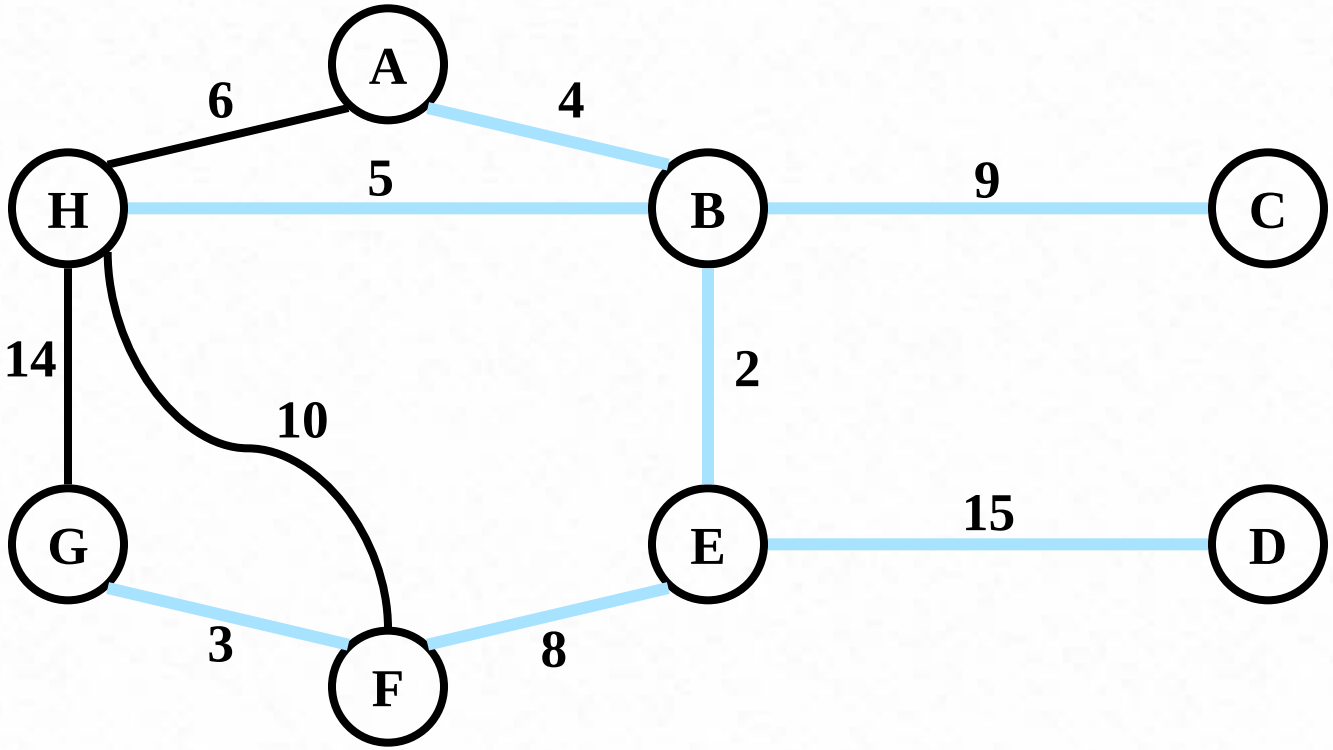
Minimálna kostra grafu

- *Ktoré hrany tvoria minimálnu kostru grafu na tomto obrázku?*



Minimálna kostra grafu

- Odpoveď:

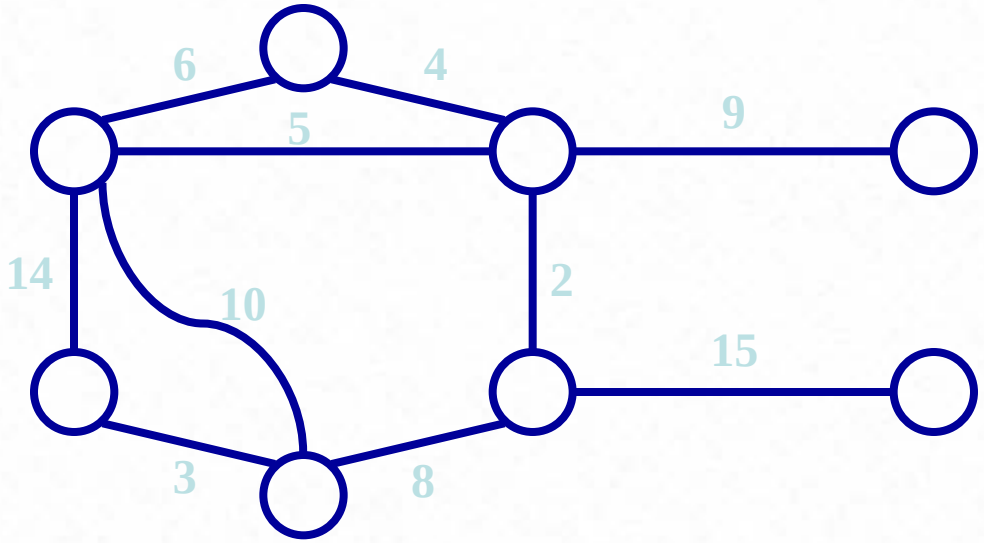


Dijkstrov-Primov algoritmus

- Tvorbu kostry začneme jedným vrcholom, dostaneme strom T_1.
- V každom kroku zabezpečíme "zväčšovanie" stromu o jeden vrchol / jednu hranu
  - Skonštruujeme postupnosť expandujúcich stromov T_1, T_2, …
- V každom kroku skonštruujeme T_{i+1} z T_i: pridáme hranu s minimálnym ohodnotením vychádzajúcu z vrcholu v strome (T_i) a vedúcu do vrcholu, ktorý ešte nie je v strome
  - výber z "okrajových" hrán (toto je "greedy" krok!)
- Algoritmus sa zastaví, keď sú do kostry pridané všetky vrcholy.

Primov algoritmus

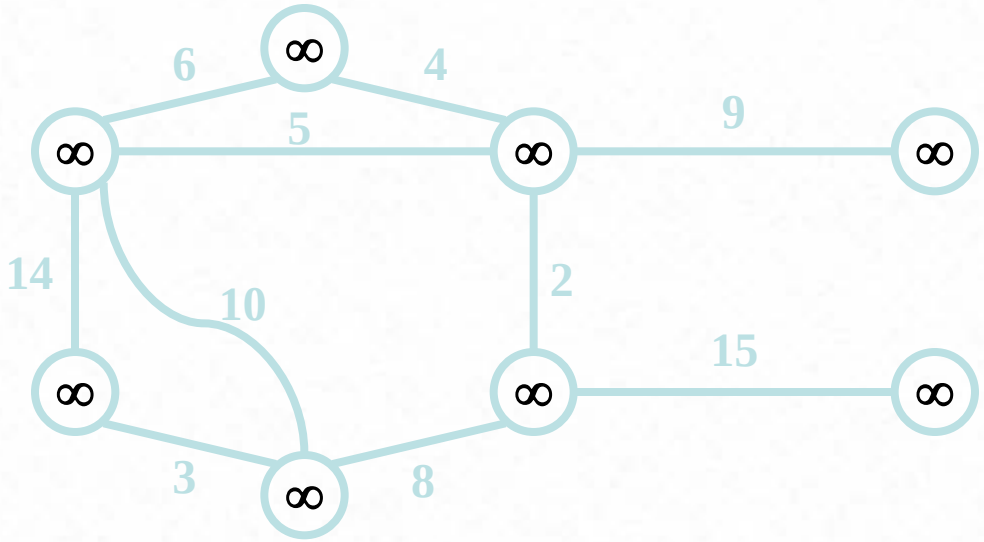
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Príklad

Primov algoritmus

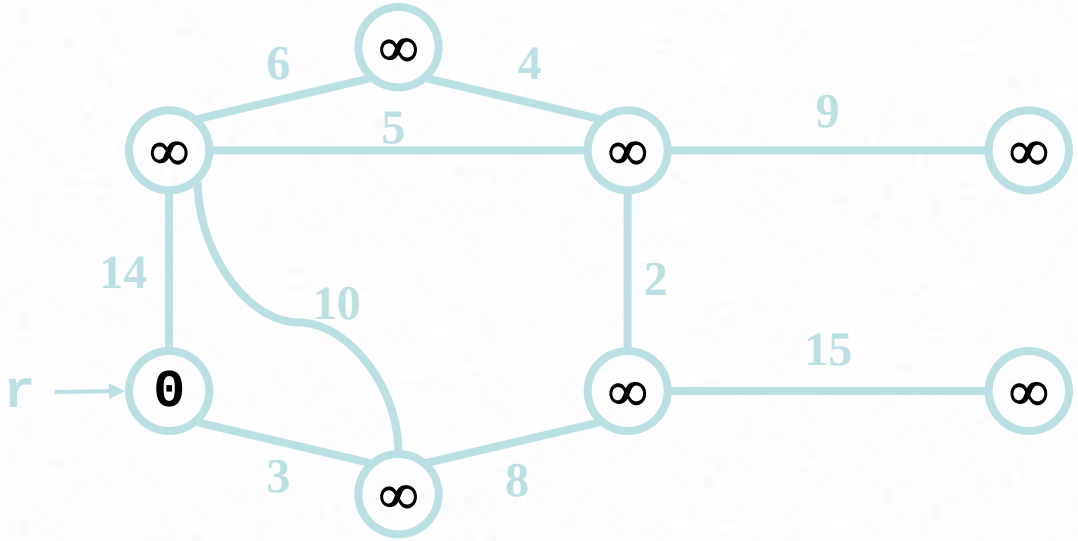
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Beh algoritmu na príklade

Primov algoritmus

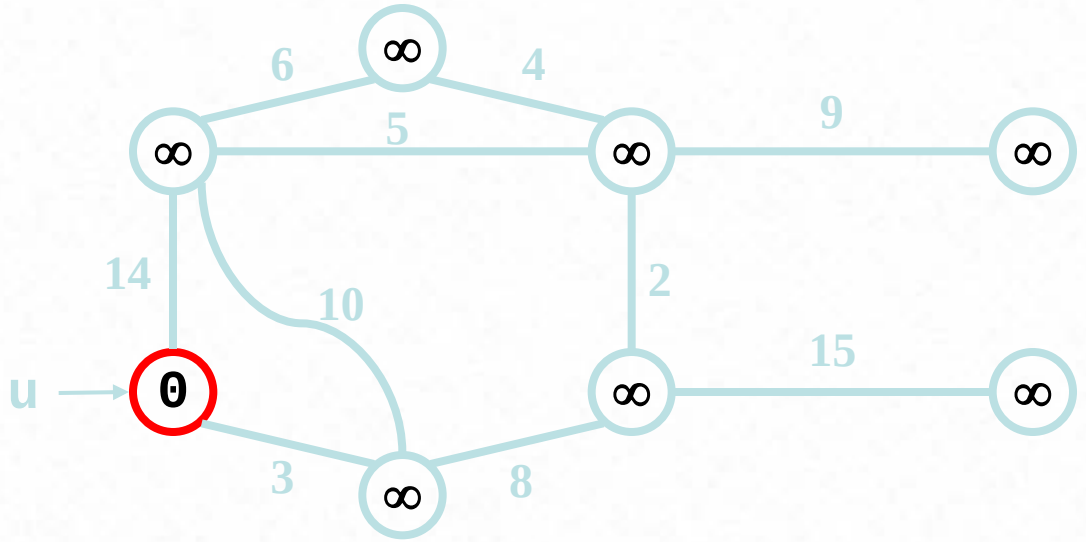
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Výber štartovacieho vrcholu r

Primov algoritmus

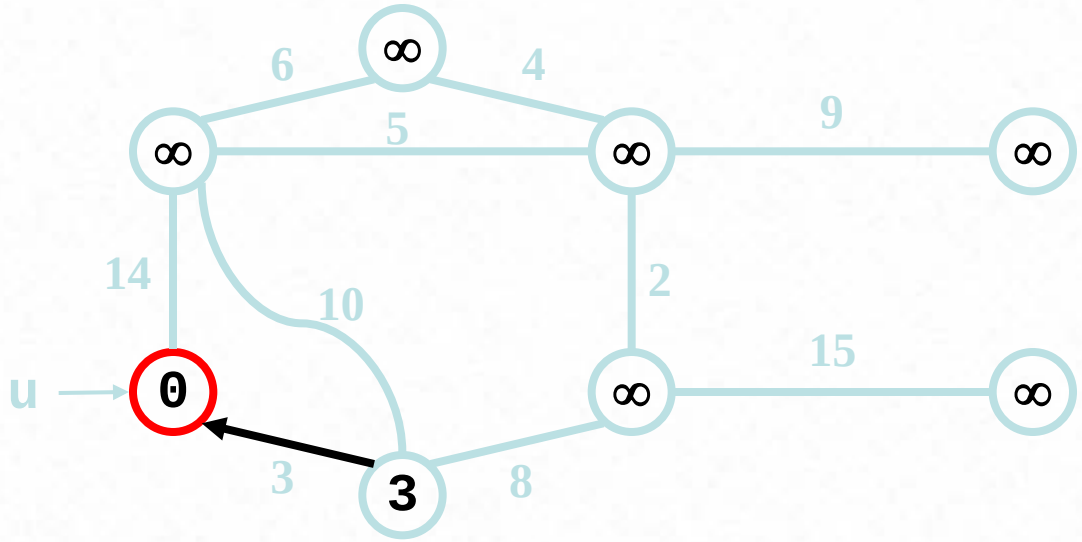
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Červené vrcholy boli odstránené z Q

Primov algoritmus

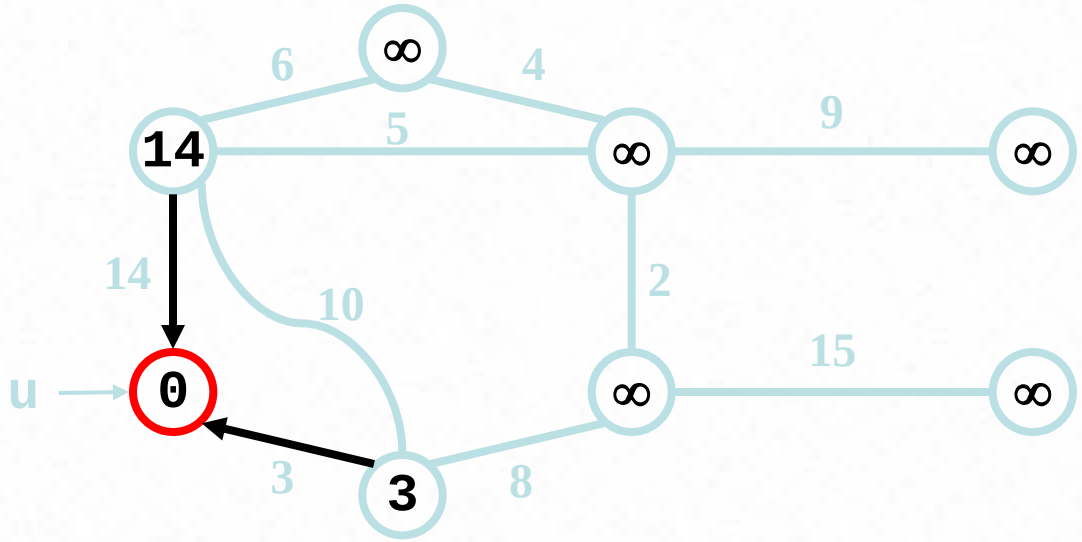
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Čierne šípky určujú smerníky k rodičom

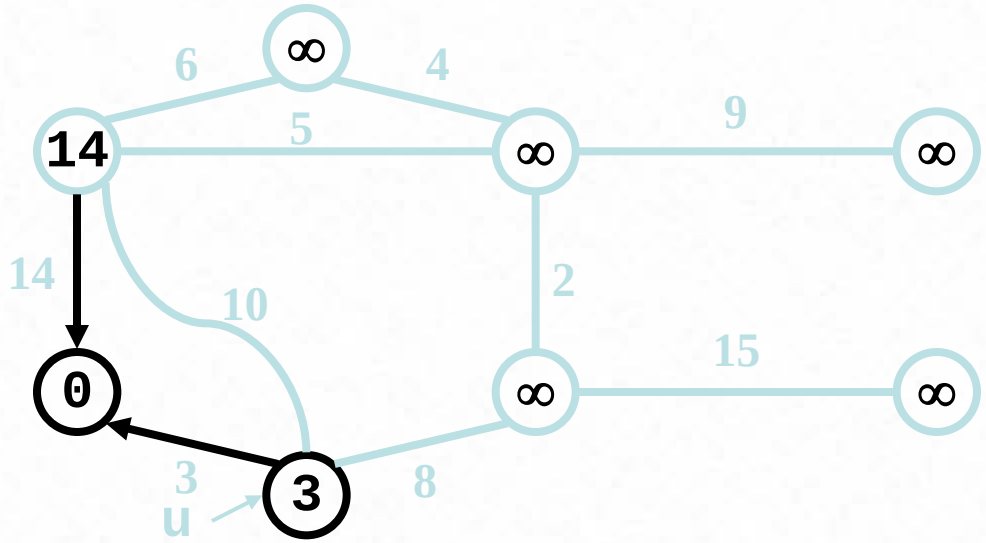
Primov algoritmus

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



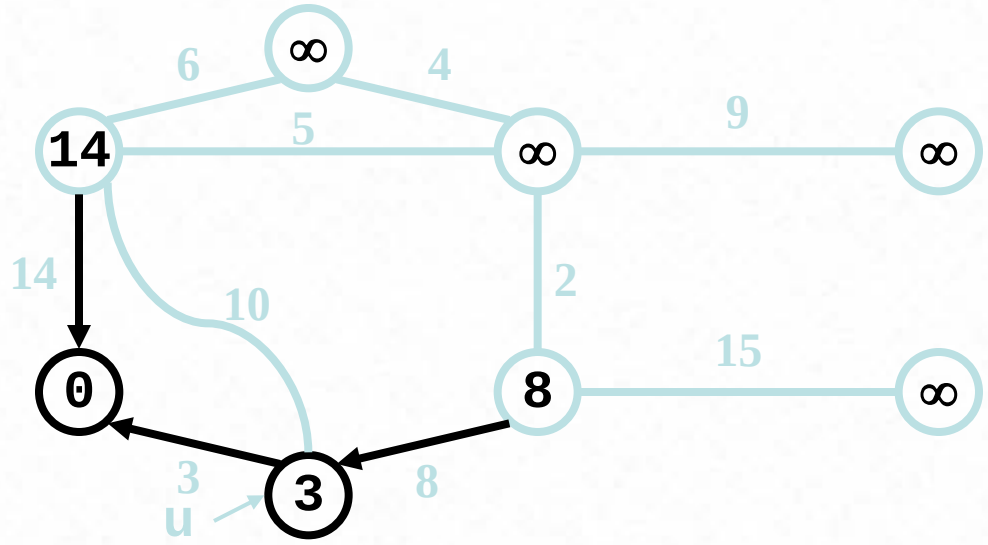
Primov algoritmus

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



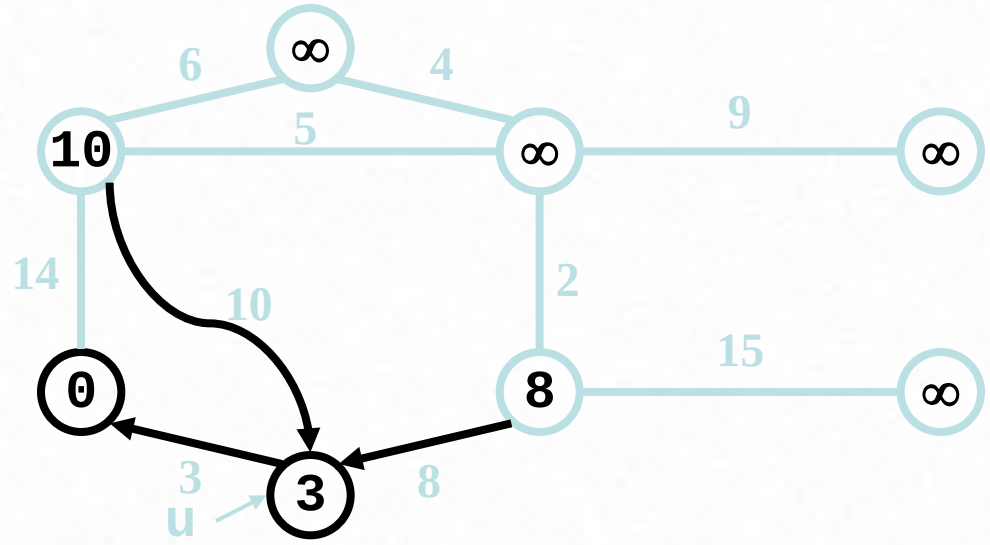
Primov algoritmus

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



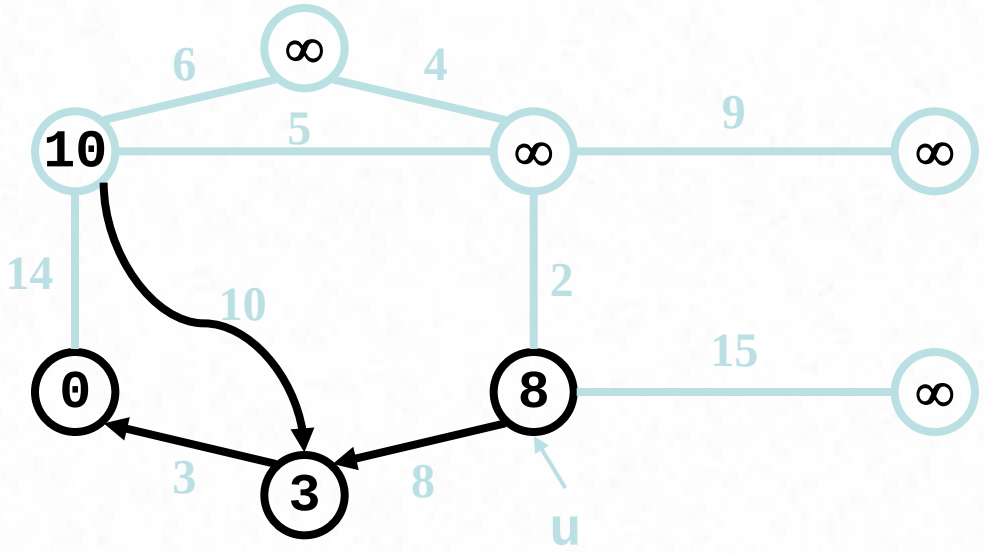
Primov algoritmus

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



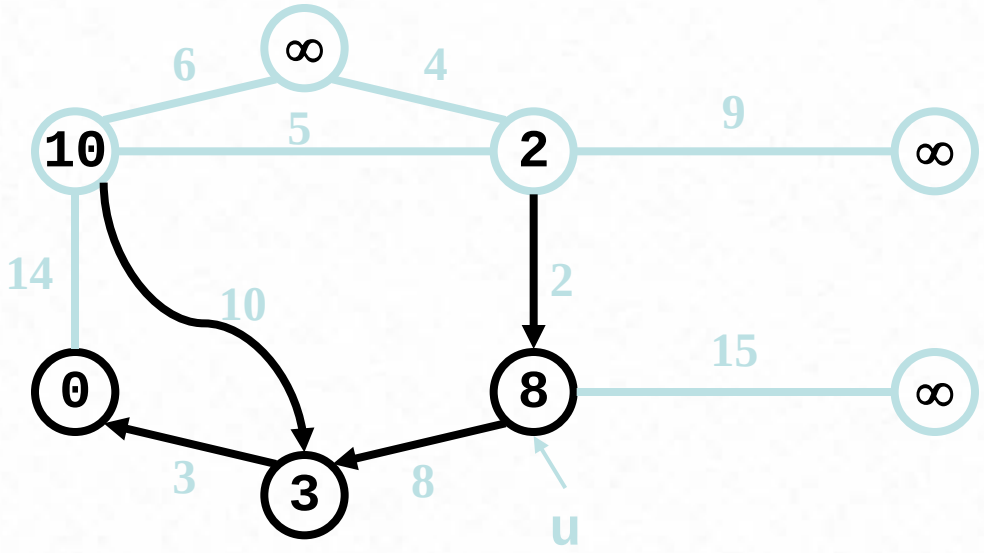
Primov algoritmus

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



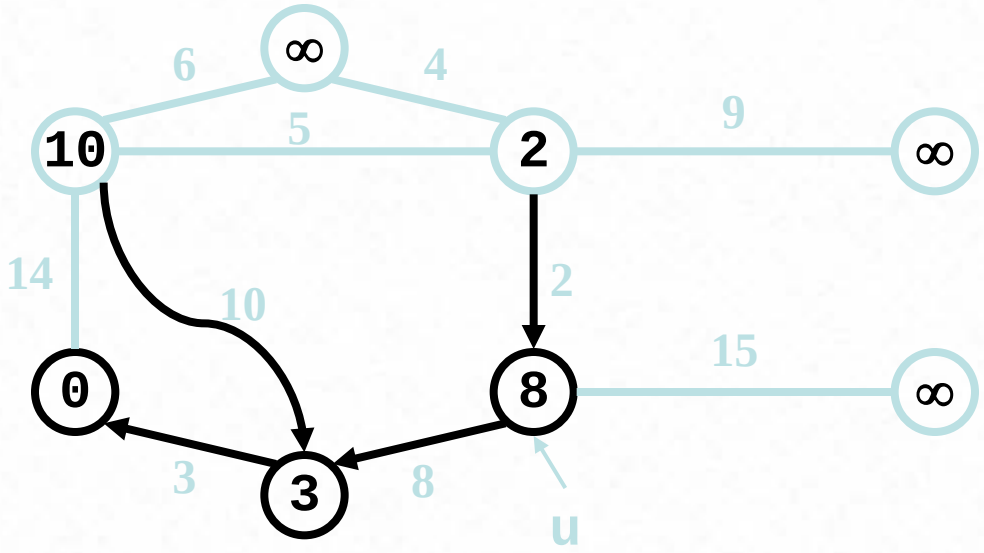
Primov algoritmus

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



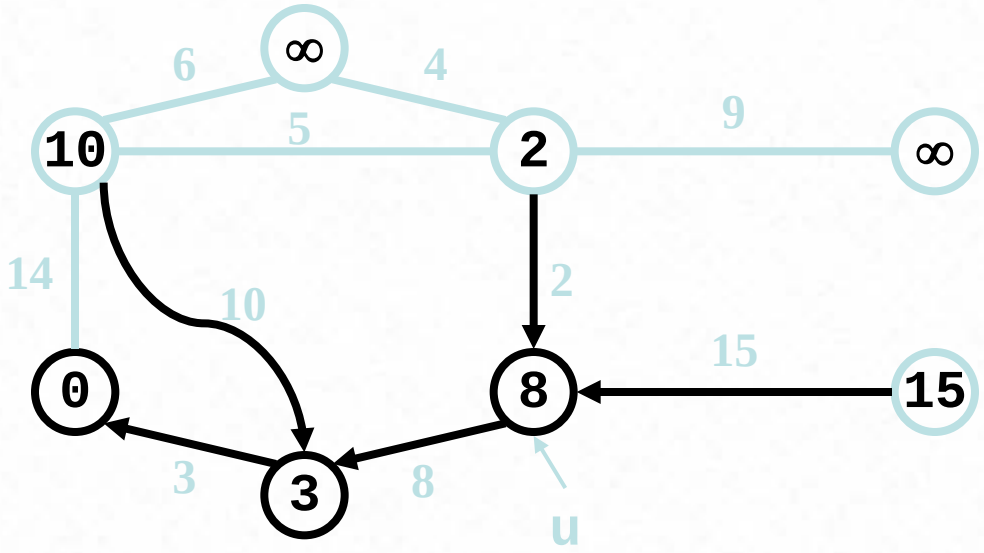
Primov algoritmus

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



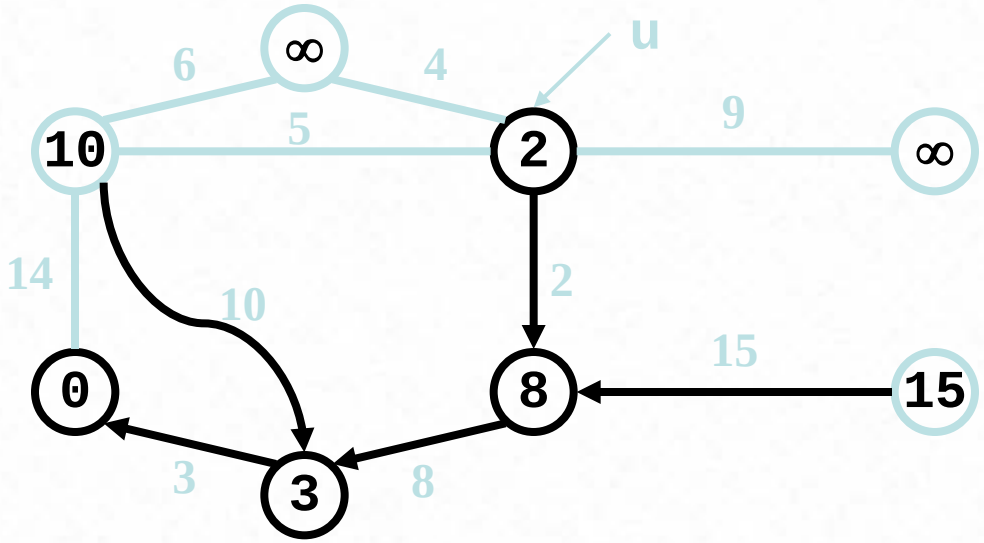
Primov algoritmus

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



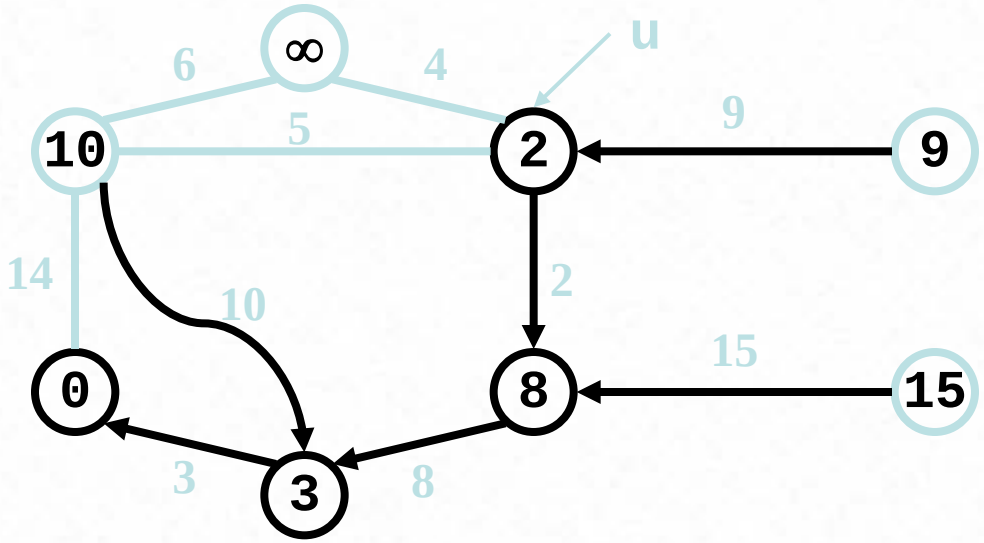
Primov algoritmus

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



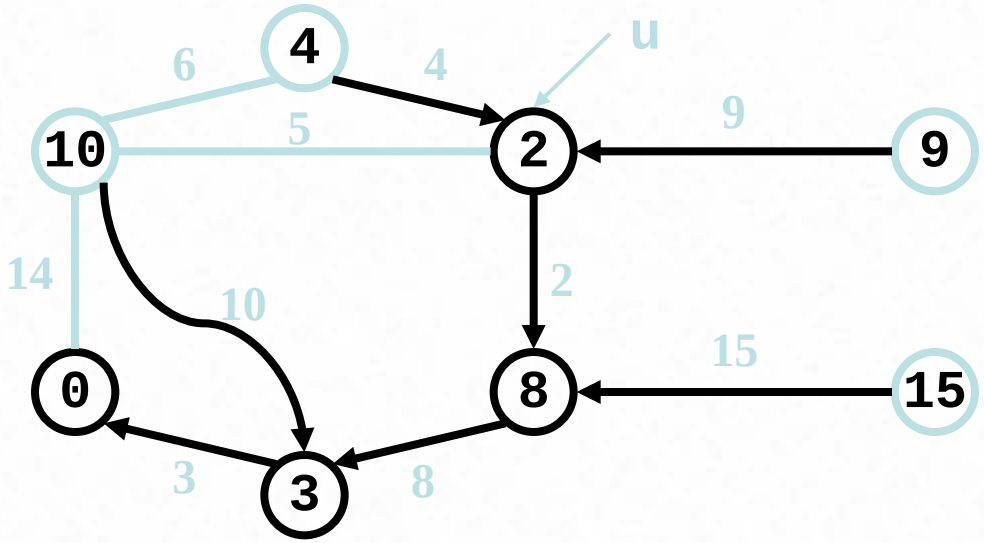
Primov algoritmus

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



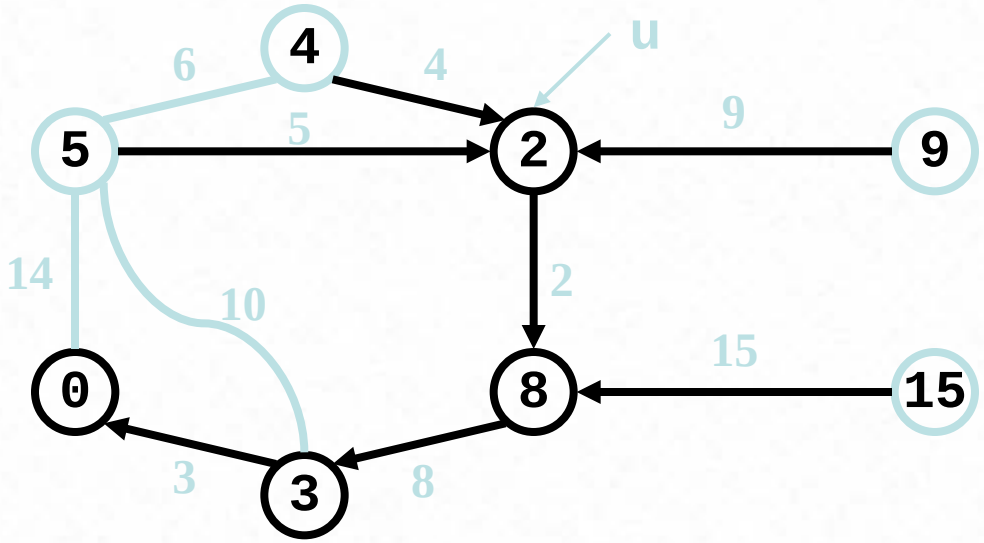
Primov algoritmus

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

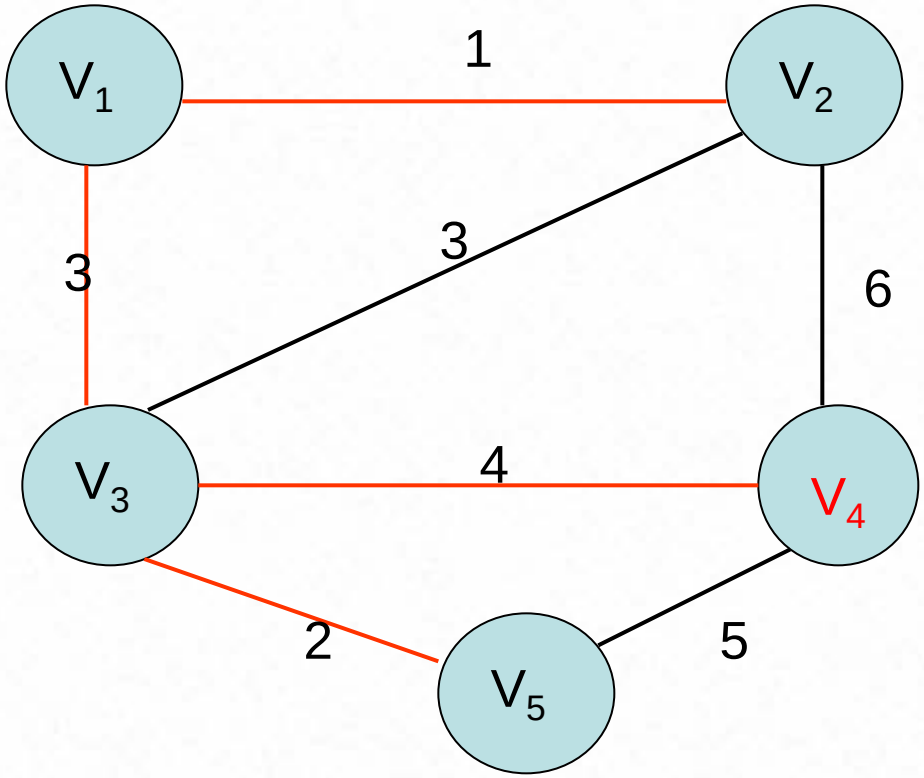


Primov algoritmus

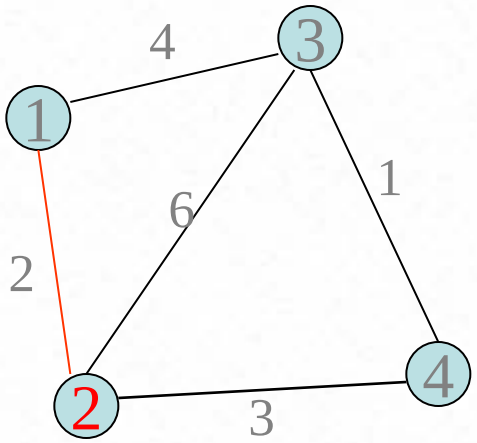
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

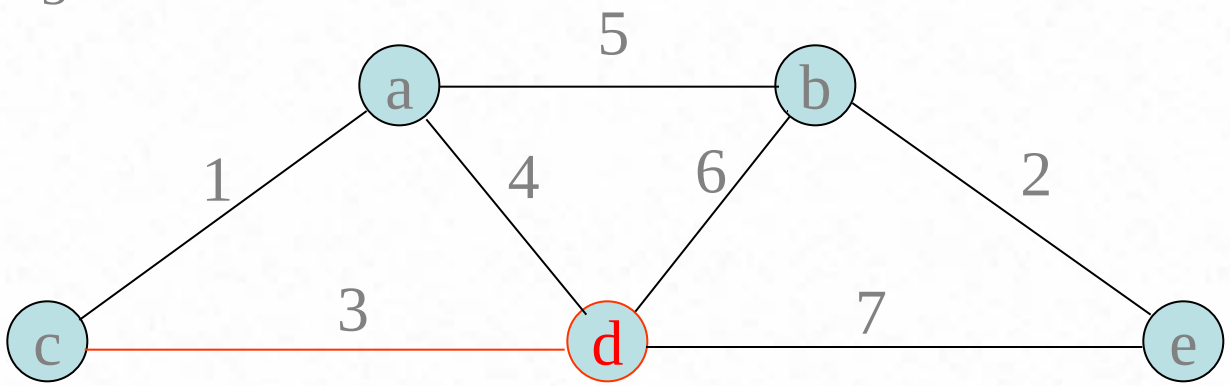


Príklad 1



Ďalšie príklady





Poznámky o Primovom algoritme

- Potrebujeme dokázať, že tento algoritmus skutočne počíta minimálnu kostru grafu.
- Potrebujeme použiť prioritný rad na nájdenie okrajovej hrany s najnižším ohodnotením: použijeme *min-heap*
- *Efektívnosť:* pre graf s n vrcholmi a m hranami:

$$(n - 1 + m) \log n$$

Počet krokov (min-heap vymazania)

vloženie/vymazanie z min-heap

Počet uvažovaných hrán (min-heap vloženia)

$$\Theta(m \log n)$$

Dôkaz korektnosti Primovho algoritmu

- Nech G je súvislý hranovo ohodnotený graf.
- V každej iterácii Primovho algoritmu je pridaná hrana, ktorá má jeden vrchol v podgrafe vytvárajúcom kostru a vrchol mimo tohto podgrafu.
- Pretože G je súvislý, existuje vždy cesta do každého vrcholu.
- Výstup *Y* Primovho algoritmu je strom, pretože vrchol a hrana, ktoré sú pridané do Y sú spojené.

Dôkaz korektnosti Primovho algoritmu

- Nech *Y1* je minimálna kostra G. Ak *Y1=Y, tak Y* je minimálna kostra. Inak, nech *e* je prvá hrana pridaná počas konštrukcie *Y, ktorá nie je v Y1*, a *V* nech je množina vrcholov spojených hranami pridanými pred pridaním *e*.
- Potom jeden koncový bod *e* je vo *V* a druhý nie je. Pretože *Y1* je kostra G, existuje cesta v *Y1 spájajúca tieto dva koncové vrcholy*. Keď prechádzame pozdĺž tejto cesty, musíme naraziť na hranu *f* spájajúcu vrchol vo *V* s vrcholom, ktorý nie je vo V.

Dôkaz korektnosti Primovho algoritmu

- **Teraz, v iterácii, keď je pridávaná hrana e do *Y*, *f* by mohla byť pridaná tiež a mohla by byť pridaná namiesto *e*, *ak* by jej ohodnotenie bolo menšie ako *e*. Pretože *f* nebola pridaná, ohodnotenie f nie je menšie ako ohodnotenie e.**

- **Nech *Y2* je graf získaný z Y1 odstránením *f* a pridaním *e*.**
- **Je ľahké ukázať, že *Y2* je súvislý, má ten istý počet hrán ako Y1 a celková váha jeho hrán nie je väčšia ako v *Y1*, preto je to tiež minimálna kostra G a obsahuje *e* a všetky hrany pridané pred e počas konštrukcie *V*.**
- **Opakovaním vyššie uvedených krokov vieme získať minimálnu kostru grafu G, ktorá je identická s *Y*. Toto dokazuje, že *Y* je minimálna kostra.**

Iný greedy algoritmus pre MST: Kruskalov

- Začneme s prázdnym lesom stromov.
- V každom kroku "porastie" MST o jednu hranu.
  - Medzikroky majú obvykle les stromov (nesúvislý graf).
- V každom kroku pridáme hranu s minimálnym ohodnotením medzi už použité hrany, ak nevznikne cyklus.
  - Hrany sú na začiatku utriedené podľa rastúceho ohodnotenia.

Iný greedy algoritmus pre MST: Kruskalov

V každom kroku hrana môže:

- Zväčšiť nejaký existujúci strom.
- Spojiť dva existujúce stromy do jedného stromu.
- Vytvoriť nový strom.

- Je potrebný efektívny spôsob pre určovanie /vyhnutie sa cyklom

- Algoritmus skončí, keď všetky vrcholy sú v jednom strome.

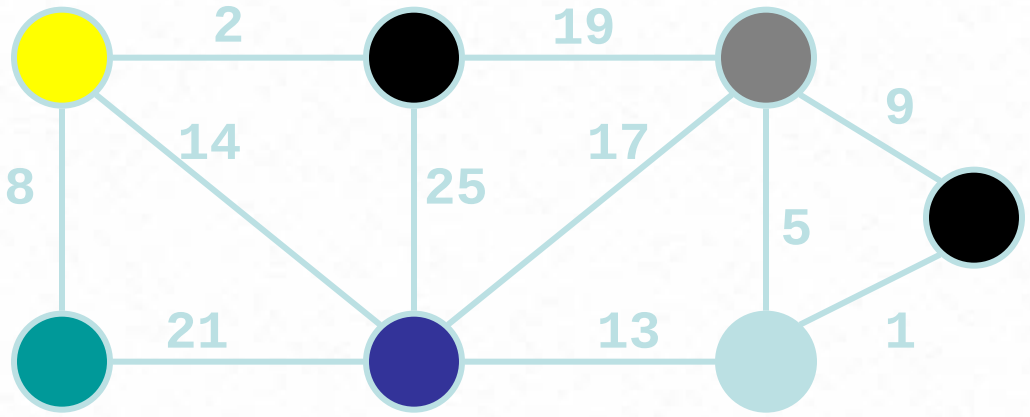
Kruskalov algoritmus

Beh algoritmu:


```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

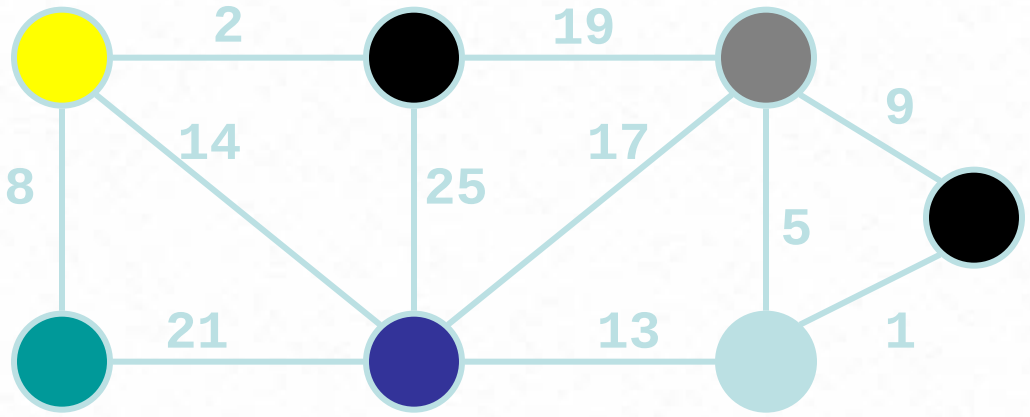
Kruskalov algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



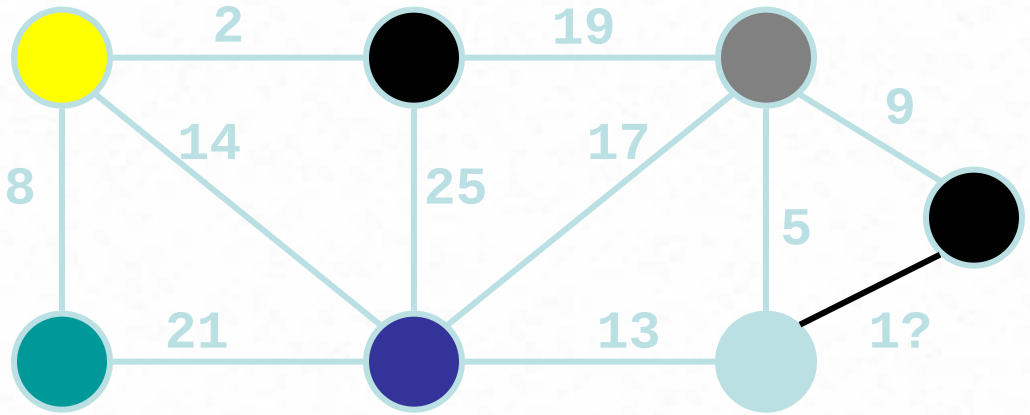
Kruskalov algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



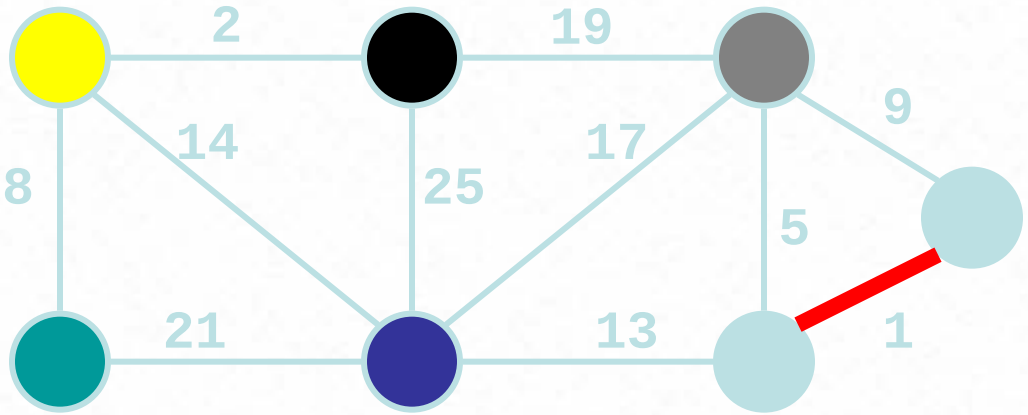
Kruskalov algoritmus

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```



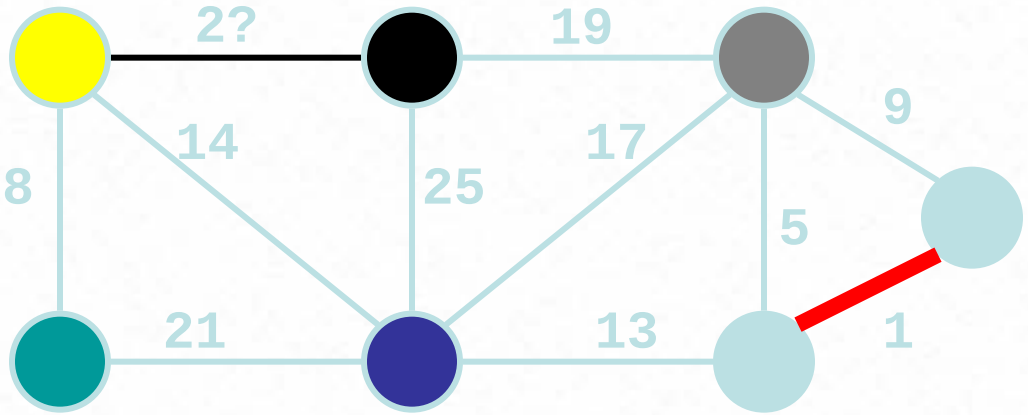
Kruskalov algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



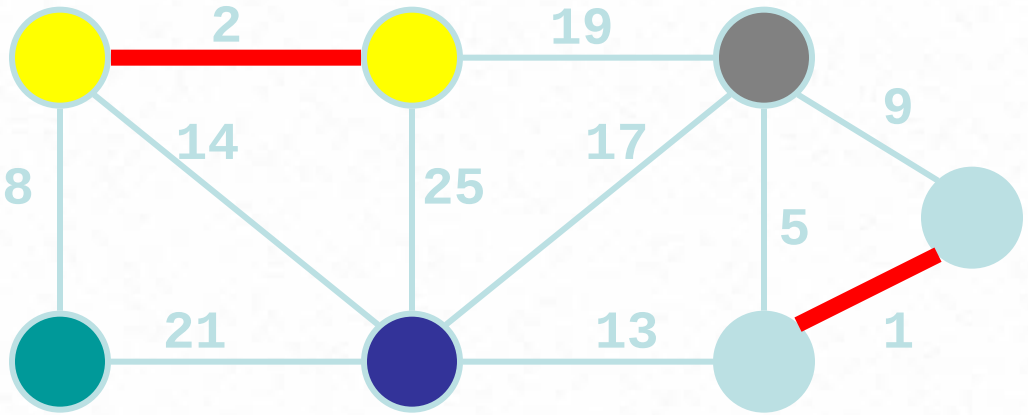
Kruskalov algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



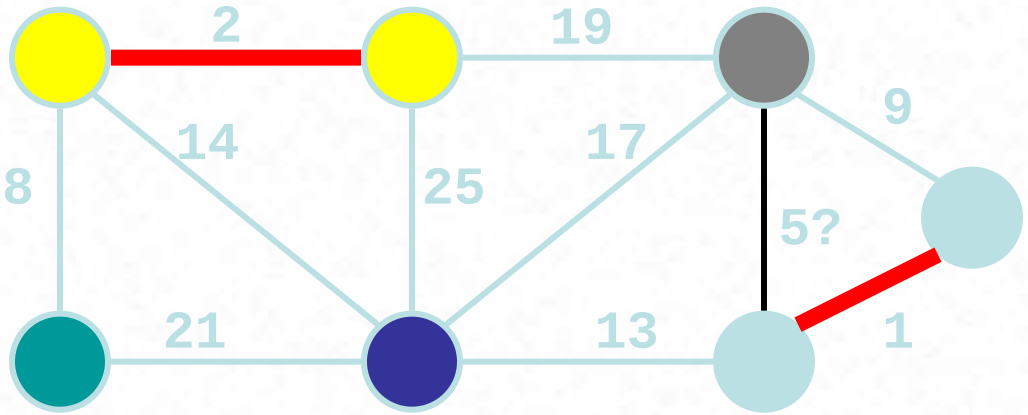
Kruskalov algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



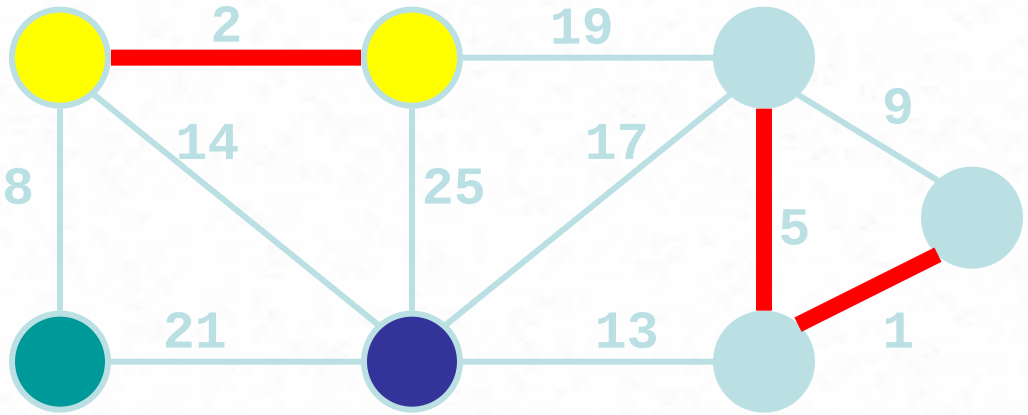
Kruskalov algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



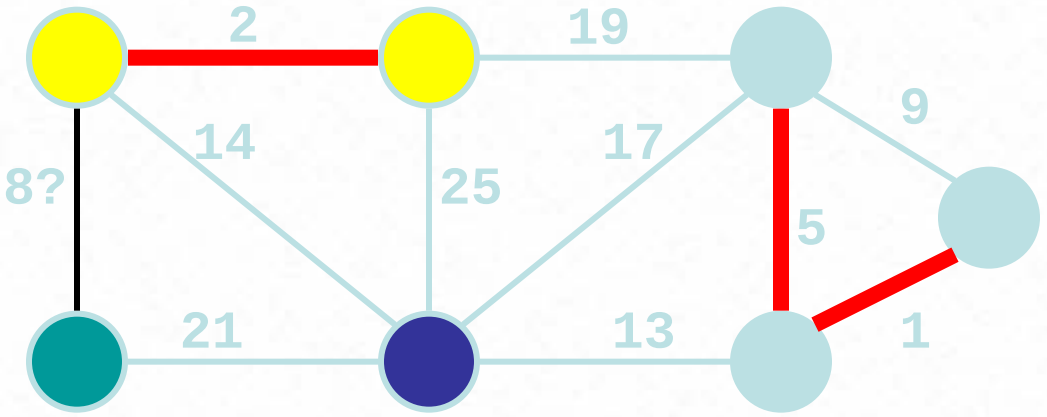
Kruskalov algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



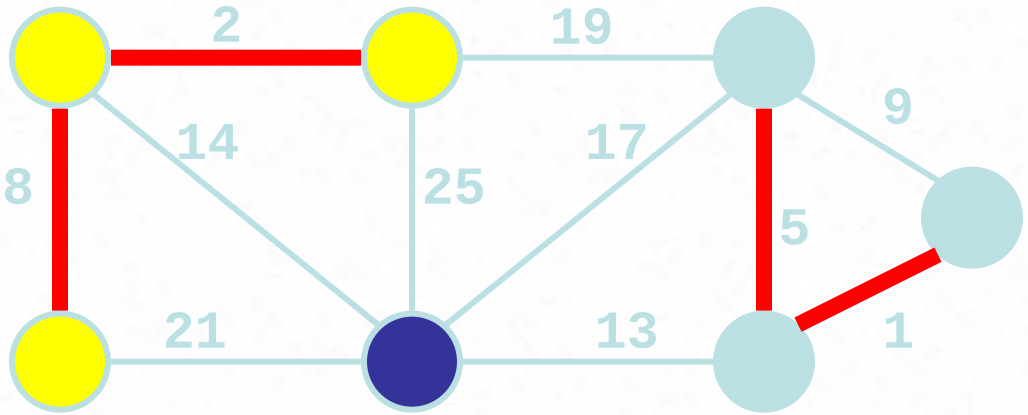
Kruskalov algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



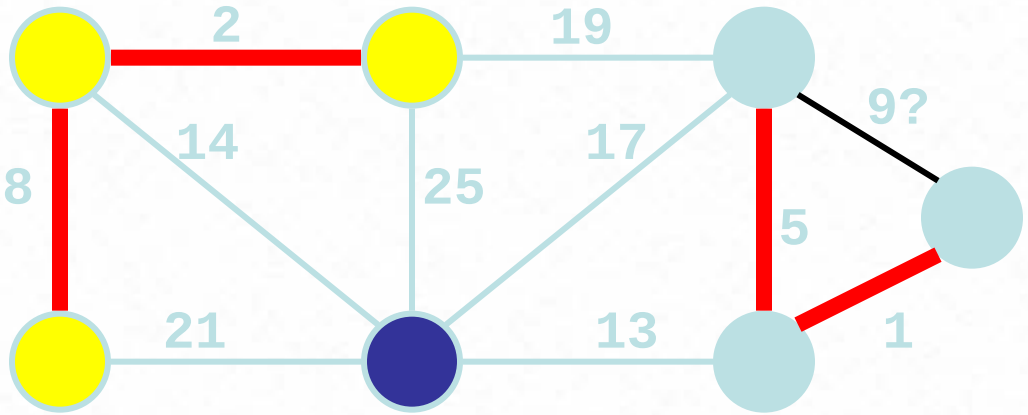
Kruskalov algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



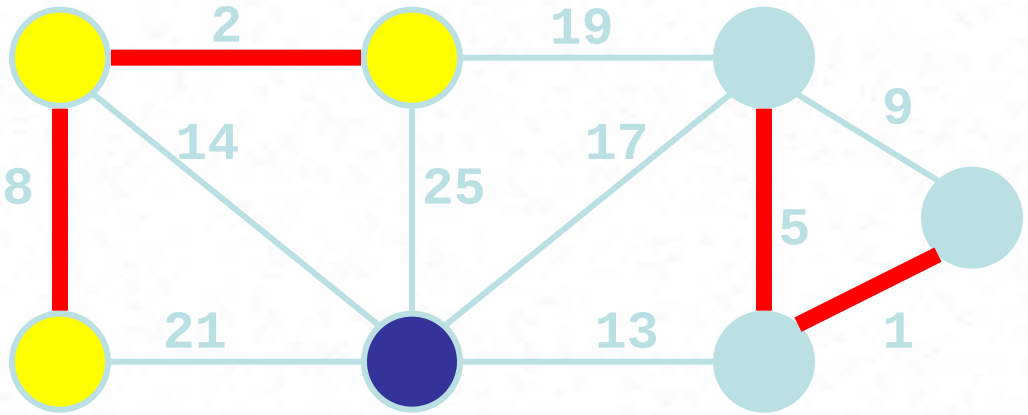
Kruskalov algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



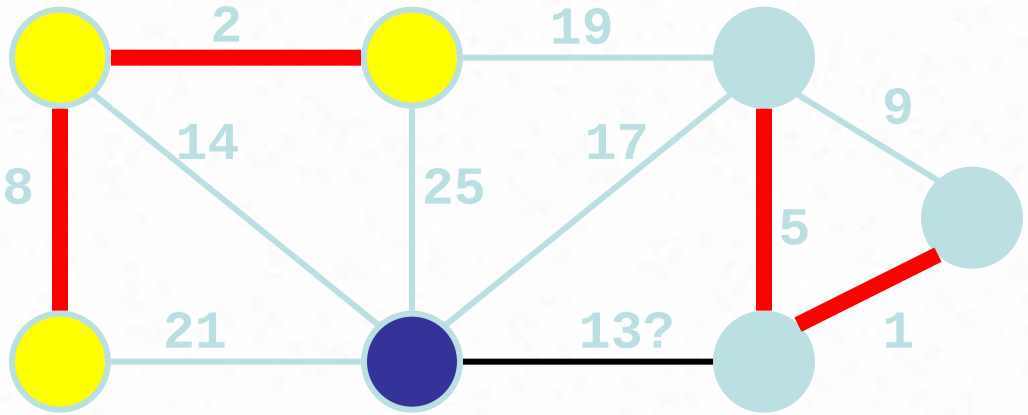
Kruskalov algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



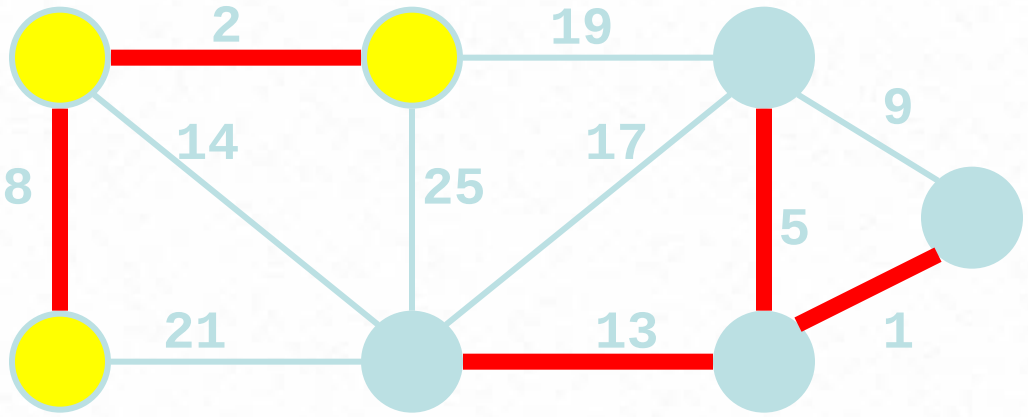
Kruskalov algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



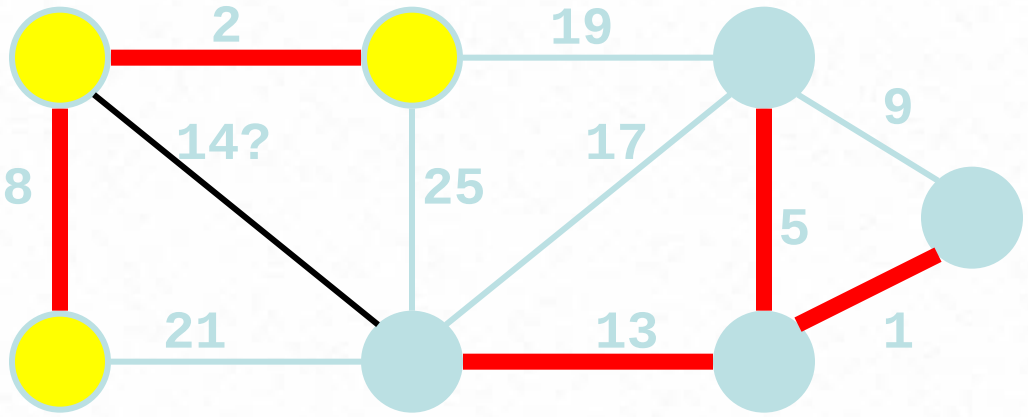
Kruskalov algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



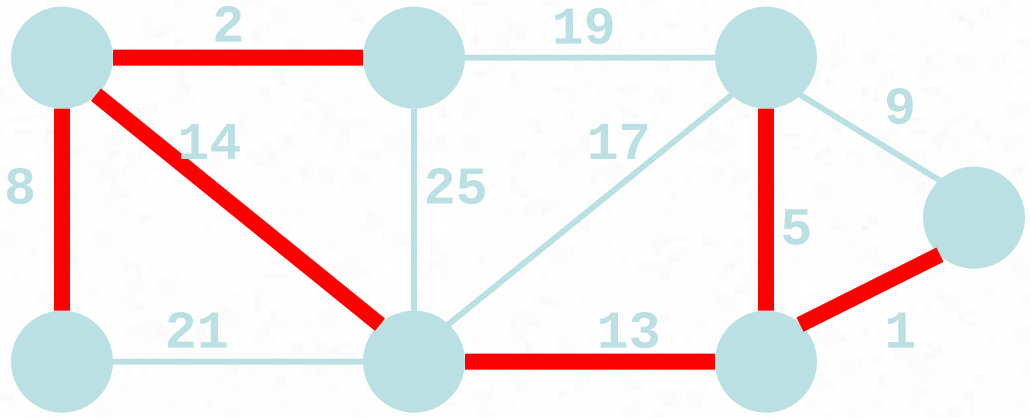
Kruskalov algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



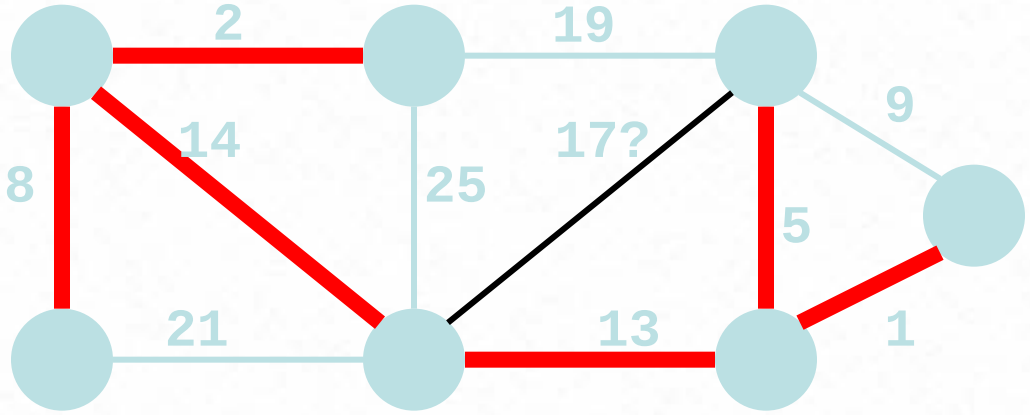
Kruskalov algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



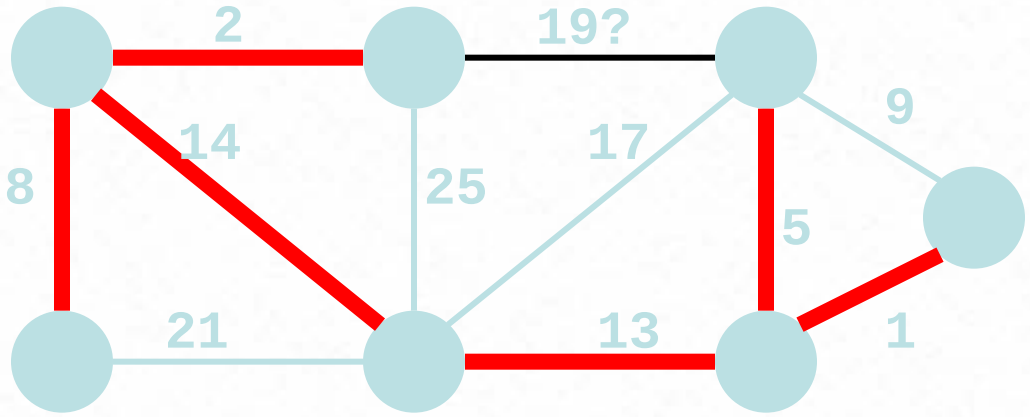
Kruskalov algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



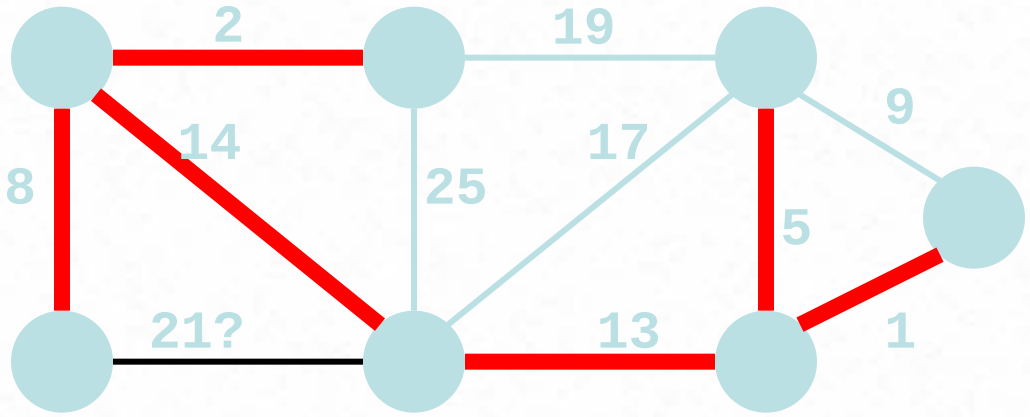
Kruskalov algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



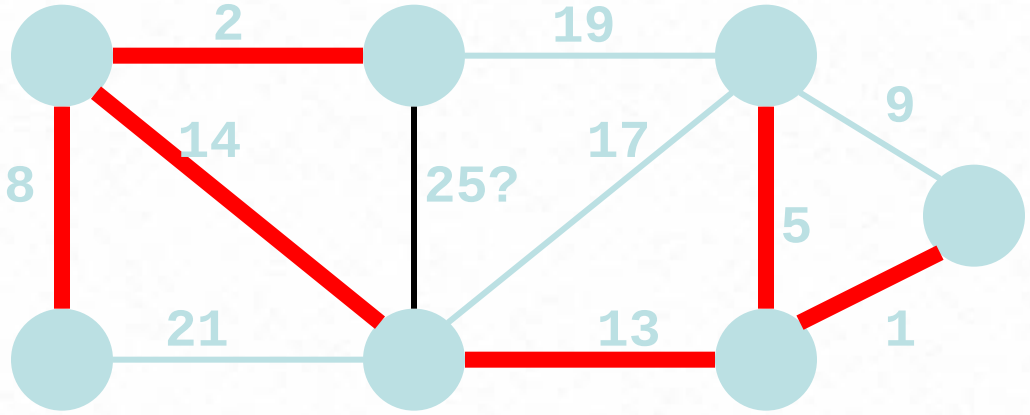
Kruskalov algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



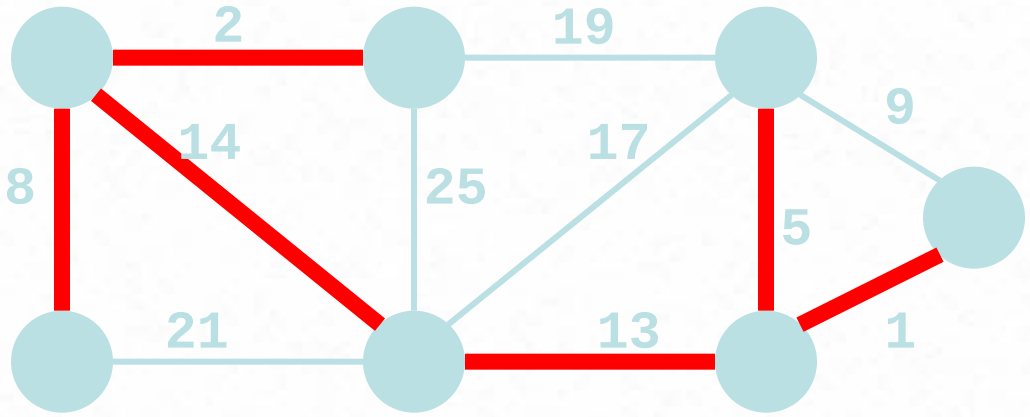
Kruskalov algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



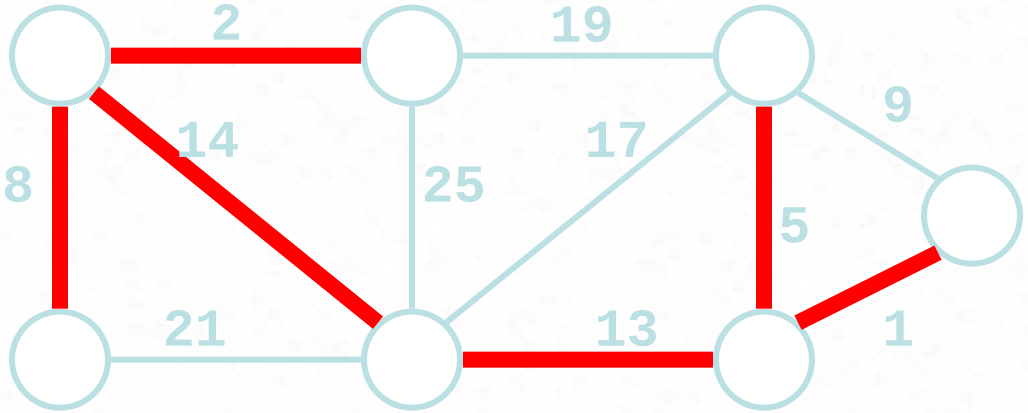
Kruskalov algoritmus

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

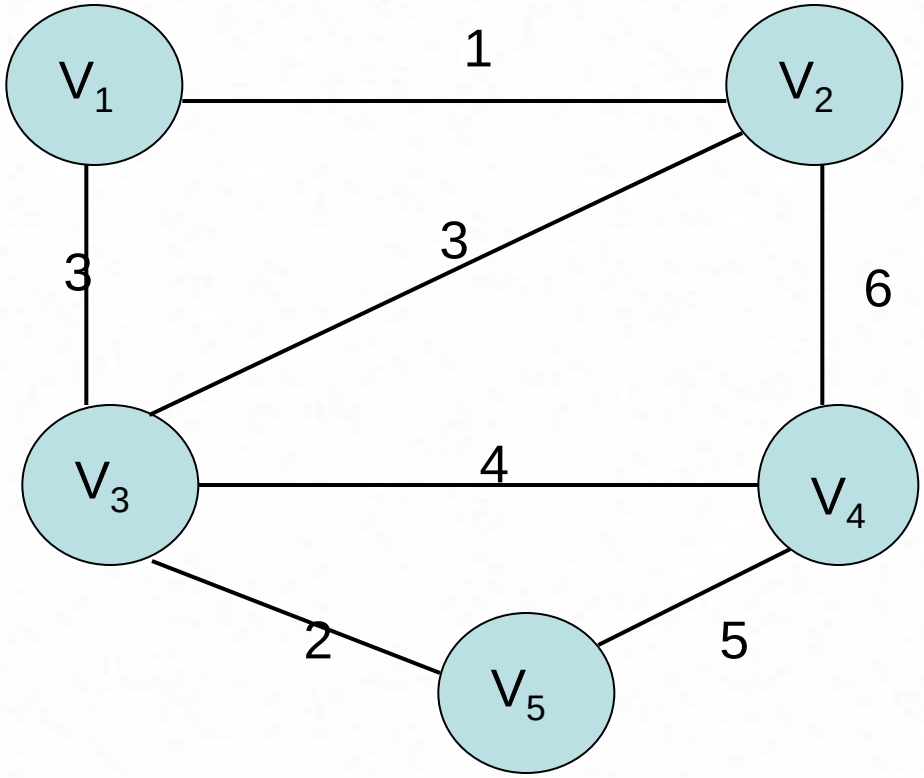


Kruskalov algoritmus

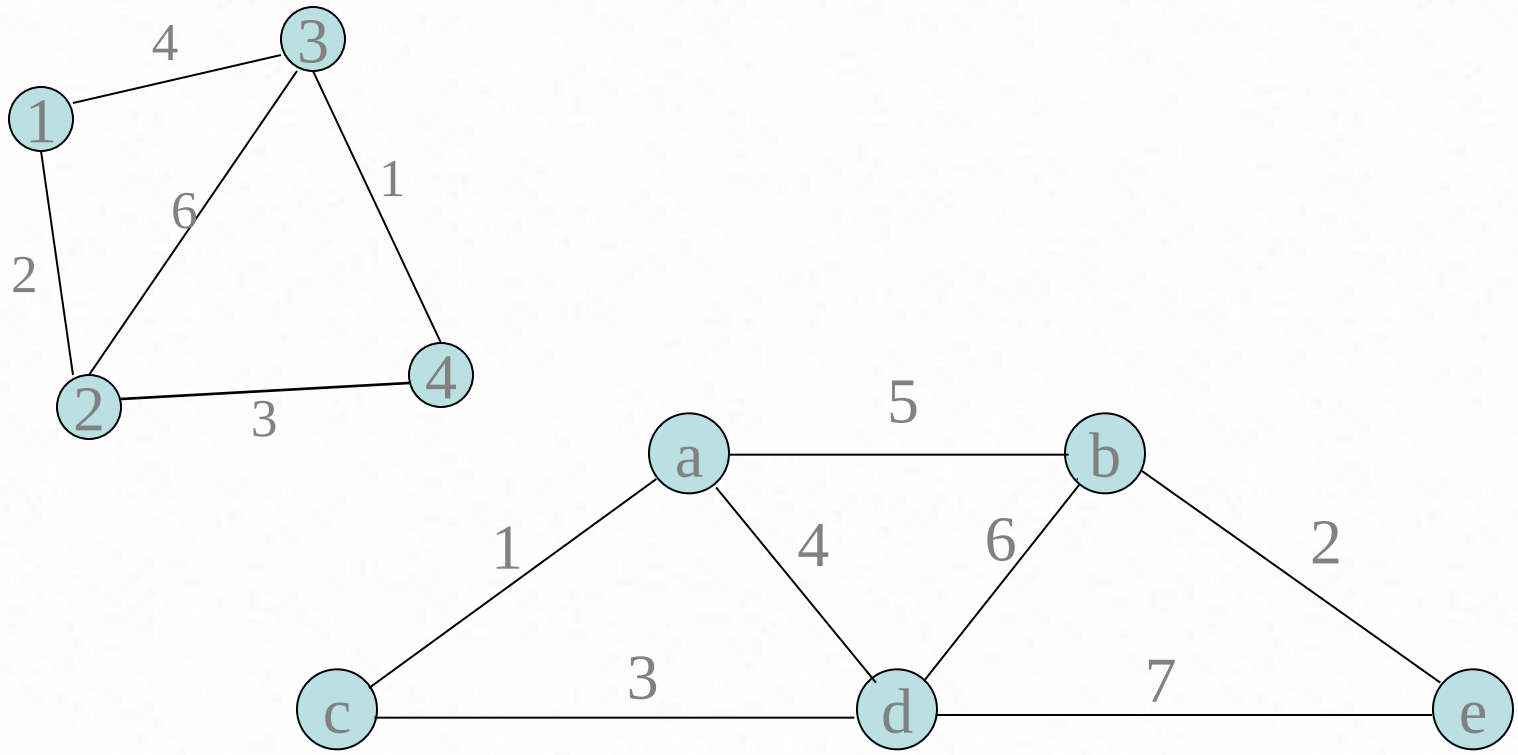
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



Príklad 1



Príklady



Poznámky o Kruskalovom algoritme

- Algoritmus vyzerá jednoduchší ako Dijkstrov-Primov, ale je
  - Ťažšie implementovateľný (kontrola cyklov v grafe!)
  - Menej efektívny $\Theta(m \log m)$
- *Kontrola cyklov:* cyklus existuje akk hrana spája vrcholy v tom istom komponente.
- *Union-find* algoritmy

Porovnanie Prim & Kruskal

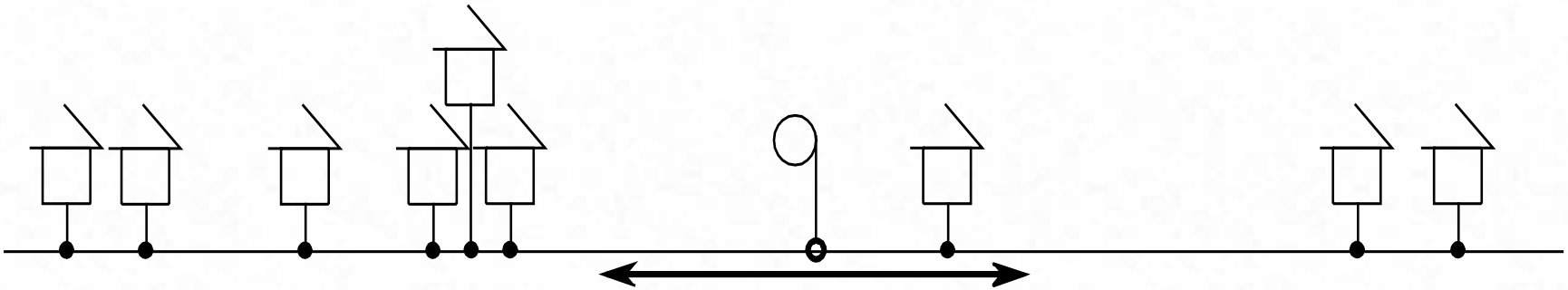
- Primov algoritmus: $T(n) \in \Theta(n^2)$
- Kruskalov algoritmus: $W(m, n) \in \Theta(m \lg m)$ a $W(m, n) \in \Theta(n^2 \lg n)$
- Ak má graf veľký počet vrcholov, tak Primov algoritmus je lepší.
- Ak má graf malý počet hrán, tak čas behu Kruskalovho algoritmu je lepší.
- Časová zložitosť závisí od použitých dátových štruktúr v implementácii.

Úloha o zastávkach autobusu

- V obci stojí niekoľko domov. Všetky stoja pri jedinej rovnej ceste, ktorá cez obec prechádza. O každom dome vieme, na koľkom metri cesty od začiatku obce má bránu.
- Našou úlohou je rozmiestniť na ceste čo najmenej autobusových zastávok tak, aby sme nimi pokryli celú obec. Presnejšie, zastávky musia byť umiestnené tak, aby to nik nemal z domu na zastávku ďalej ako 500 metrov.

Úloha o zastávkach autobusu

V Kocúrkove stoja domy na nasledujúcich súradniciach: 100, 250, 600, 1000, 1100, 1200, 2300, 3300 a 3450 metrov od začiatku obce. Nájdite ručne čo najlepšie rozmiestnenie zastávok. Situáciu si môžeme znázorniť nasledovne:



Úloha o zastávkach autobusu

- Optimálne riešenie je postaviť štyri zastávky. Existuje veľa spôsobov, ako to spraviť: napríklad budeme mať zastávky na súradniciach 300 (sem budú chodiť z prvých troch domov), 1100 (štvrtý až šiesty dom), 2800 (siedmy a ôsmy dom) a 3350 (posledný dom).
- Iné, rovnako dobré riešenie, je postaviť zastávky na súradniciach 600, 1200, 2300 a 3333.

Úloha o zastávkach autobusu

- Ako ale dokázať, že nám na túto obec nestačia tri zastávky?
- Ako sformulovať všeobecný algoritmus, ktorý bude túto úlohu riešiť a vždy nájde optimálne riešenie?

Optimálne riešenie teda vieme zostrojiť tak, že stále opakujeme nasledujúce kroky:

1. **Nájdeme prvý dom v obci, ktorý ešte pri sebe nemá zastávku.**
2. **Postavíme zastávku 500 metrov zaň.**

Problém plnenia batohu

Knapsack problem

- Zlodej sa vlámal do klenotníctva a chce si naplniť batoh pokladmi (chce ukradnúť čo najviac).
- Daných je n položiek S = {položka$_1$, položka$_2$, …, položka$_n$}, z ktorých každá má svoju váhu w_i a profit p_i.
- Ktoré položky by mal zlodej vložiť do svojho batoha, ktorý má váhovú kapacitu W, aby dosiahol maximálny profit?

Problém plnenia batohu

- Najjednoduchší prístup by bol vygenerovať všetky možné podmnožiny šperkov a pre každú vypočítať profit. Potom zobrať podmnožinu s najväčším profitom.
- Toto vyžaduje ? času a nazýva sa ? algoritmus.

Greedy stratégia

- Predpokladajme, že máme 3 položky:

Položka	Váha	Profit
W1	25 kg	10 EUR
W2	10 kg	9 EUR
W3	10 kg	9 EUR

W=30

Prístup 1

- Ukradne položku s najväčším profitom:
  - S = 3 w = [25, 10, 10], p = [10, 9, 9], W = 30
- Profit je 10, hoci optimálny profit by mohol byť 18

Prístup 2

- Ukradne položku s najväčším profitom vzhľadom na jednotku váhy
  - S = 3 , w = [5, 10, 20], p = [50, 60, 140], W = 30
- Vyberá v poradí [1, 3, 2].
- Profit je 190, hoci optimálny profit by bol 200

Zlomkový problém plnenia batohu

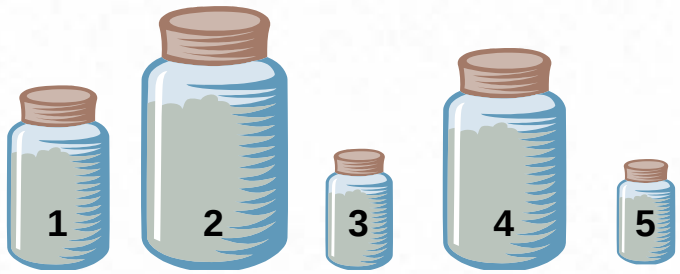
- Fractional Knapsack Problem
- Zlodej nemusí ukradnúť celú položku, ale môže vziať ľubovoľný zlomok položky.
- Poskytneme optimálne riešenie:
  - 50 + 140 + (5/10) * 60 = 220
  - Zlodej vezme 5/10 z položky 2
  - 5 je zostávajúca kapacita batohu

Zlomkový problém plnenia batoha

- Dané je: Množina S s *n* položkami, i-tá položka má
  - b_i – kladný benefit
  - w_i – kladnú váhu
- Cieľ: vybrať položky s maximálnym celkovým benefitom, ale váhou najviac *W*.
- Ak povolíme zlomkové množstvá, tak tento problém sa nazýva zlomkový problém plnenia batoha.
  - V tomto prípade, ak označíme x_i množstvo, ktoré vezmeme z i-tej položky
    - Maximalizujeme $\sum_{i \in S} b_i (x_i / w_i)$
    - Obmedzenie: $\sum_{i \in S} x_i \leq W$

Príklad

- Dané: Množina S obsahujúca n položiek, i-tá položka má
  - b_i – kladný benefit
  - w_i – kladnú váhu
  - Cieľ: Vybrať položky s maximálnym celkovým benefitom, ale s váhou najviac W.



Položky:	1	2	3	4	5
Váha:	4 ml	8 ml	2 ml	6 ml	1 ml
Benefit EUR:	12	32	40	30	50
Hodnota: (EUR za ml)	3	4	20	5	50

10 ml

"Batoh"

Riešenie:

- 1 ml z 5
- 2 ml z 3
- 6 ml zo 4
- 1 ml z 2

Zlomkový problém plnenia batoha

- Greedy výber: V každom kroku vyberať položku s najvyššou **hodnotou** (pomer benefit k váhe)
  - Na ukladanie položiek použijeme prioritný rad (*heap-based priority queue*), potom časová zložitosť je O(n log n).

Algorithm *fractionalKnapsack*(*S*, *W*)

Input: set S of items w/ benefit b_i and weight w_i; max. weight W

Output: amount x_i of each item i to maximize benefit with weight at most W

for ***each item i in S***

$x_i \leftarrow 0$

$v_i \leftarrow b_i / w_i$ {value}

$w \leftarrow 0$ {current total weight}

while $w < W$

remove item i with highest v_i

$x_i \leftarrow \min\{w_i, W - w\}$

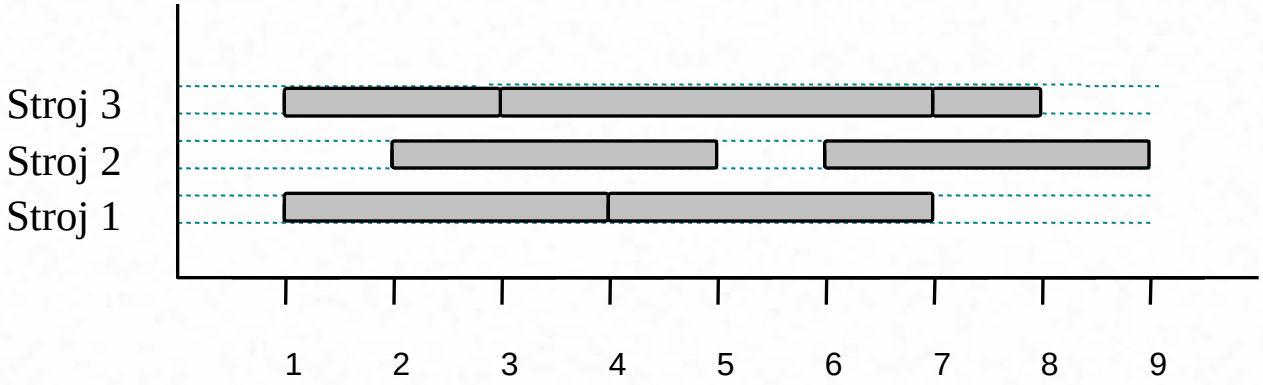
$w \leftarrow w + \min\{w_i, W - w\}$

Zlomkový problém plnenia batoha

- Korektnosť: Predpokladajme, že existuje lepšie riešenie
  - Teda existuje položka *i* s vyššou hodnotou než vybratá položka *j* (t.j., $v_j<v_i$) , ak nahradíme položku j položkou i, dostaneme lepšie riešenie.
  - Teda neexistuje žiadne lepšie riešenie než to, ktoré dáva greedy algoritmus.

Rozvrhovanie úloh

- Dané: množina T obsahujúca n úloh, z ktorých každá má:
  - Štartovací čas, s_i
  - Čas dokončenia, f_i (kde $s_i < f_i$)
- Cieľ: Vykonať všetky úlohy s minimálnym počtom strojov.”



Rozvrhovanie úloh

- **Greedy výber:** Berieme úlohy podľa ich štartovacích časov a použijeme pritom čo najmenší počet strojov pre toto poradie.
  - Čas behu: O(n log n).

Algorithm *taskSchedule*(T)

Input: set T of tasks w/ start time s_i and finish time f_i

Output: non-conflicting schedule with minimum number of machines

$m \leftarrow 0$ {no. of machines}

while *T is not empty*

remove task i w/ smallest s_i

if *there's a machine j for i* **then**

schedule i on machine j

else

$m \leftarrow m + 1$

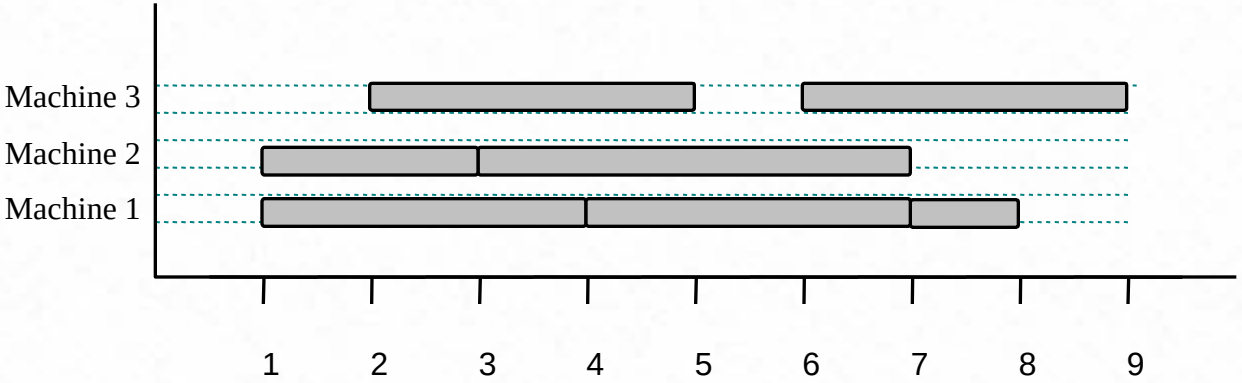
schedule i on machine m

Rozvrhovanie úloh

- **Korektnosť:** Predpokladajme, že existuje lepší rozvrh.
  - Môžeme použiť k-1 strojov
  - Algoritmus používa k strojov
  - Nech i je prvá úloha na stroji k
  - Úloha i musí byť v konflikte s inými k-1 úlohami
  - Teda je k navzájom konfliktných strojov
  - Ale to znamená, že sa nedá urobiť nekonfliktný rozvrh na k-1 strojoch.

Príklad

- Dané: množina T obsahujúca n úloh, z ktorých každá má:
  - Štartovací čas, s_i
  - Koncový čas, f_i (kde $s_i < f_i$)
  - [1,4], [1,3], [2,5], [3,7], [4,7], [6,9], [7,8] (usporiadané podľa začiatku)
- Cieľ: Vykonať všetky úlohy na minimálnom počte strojov



Záver

- Greedy (pažravý) algoritmus je každý algoritmus, ktorý rieši daný problém na základe metaheuristického prístupu, nájdením najlepšej lokálnej voľby, pričom dúfa, že sa takto dopracuje ku globálnemu optimálnemu riešeniu.
- Existujú problémy, pre ktoré tento prístup dáva skutočne optimálne riešenie (vždy to treba dokázať).
- Existujú problémy, pre ktoré tento prístup nedá globálny optimálny výsledok (ale niekedy to stačí).

Ďakujem za pozornosť

Použitá literatúra:

Briana B. Morrison: Greedy algorithms, dostupné na webe

Motivačný príklad o veštici a obrázok k dláždeniu zo študijných materiálov ĎVUI.