**Minimální kostra grafu**

**Kostra grafu**

Podmnožina hran (neorientovaného) grafu, která tvoří strom propojující všechny vrcholy grafu.

Existuje jen na spojitých grafech (tj. Grafy na kterých existuje alespoň jedna cesta mezi každou dvojicí vrcholů)

Pro graf s *N* vrcholy obsahuje *N-1* hran.

**Minimální kostra grafu**

Dává smysl jen na ohodnocených grafech (každá hrana má nějakou “váhu”)

Kostra s minimálním součtem vah všech hran

**Proč minimální kostra**?

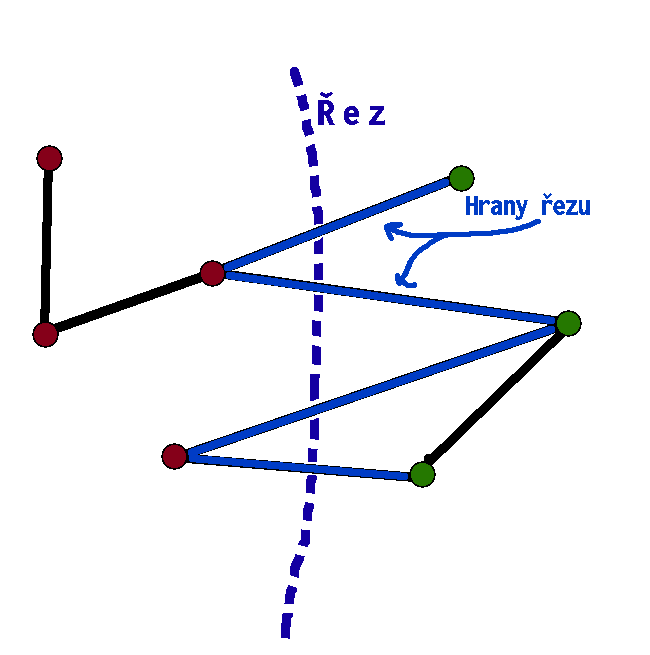
Dává nejlevnější způsob, jak propojit nějaké body, pokud známe ceny propojení různých uzlů

Například:

Máme za úkol vytvořit novou elektrickou síť, chceme aby propojila nějaká města a známe vzdálenosti různých měst.

Zřejmě u elektrické síťě nezáleží **jak** jsou města propojené, jenom že **jsou** propojené. (Myslim teda nvm nejsem fyzik)

Nejlevnější způsob propojení je tedy minimální kostra.



**Řez**

Řezem rozdělíme vrcholy grafu na dvě množiny. Jako řez potom označíme množinu všech hran, které mají oba konce v různých množinách vrcholů, tedy ty hrany, kterými “řez rozdělující graf prochází”.

**Řezové lemma**

Řezové lemma je užitečné pro důkaz správnosti algoritmů. Říká, že hrana na řezu s minimální váhou je určitě součástí minimální kostry. Pokud je na řezu více minimálních hran se stejnou váhou, pak je určitě součástí minimální kostry alespoň jedna z nich.

**Důkaz**

Uvažujme, že máme minimální kostru *T* která neobsahuje minimální hranu *m* z řezu *C*

*T* určitě obsahuje alespoň jednu jinou hranu na řezu *C*, jinak by zřejmě nebyly propojeny obě strany řezu

Nyní do *T* přidáme hranu *m*, potom *T* už není strom, obsahuje cyklus (přidáním hrany do stromu vždy vznikne cyklus s touto hranou)

Abychom se zbavily cyklu, a aby *T* bylo opět stromem, můžeme odebrat libovolnou hranu z cyklu. Zároveň alespoň jedna z hran v *T* na řezu je součástí cyklu, označme ji *u*. Jelikož je ale *m* minimální hranou na řezu, tak hrana *u* má větší váhu než *m*, odstraníme tedy z cyklu *u*

Tím jsme efektivně nahradily *u* za *m*, a snížil se tím součet vah, *T* tedy zřejmě nemohlo být minimální kostrou - spor

**Jarníkův algoritmus**

Jarníkův algoritmus z jednoho vrcholu postupně staví propojenou komponentu, ke které vždy přidá nejmenší vycházející hranu.

Dokud nejsou všechny vrcholy propojené:

Vybereme minimální hranu vycházející z aktuální komponenty a připojíme jí další vrchol

(Hranou vycházející z komponenty se myslí hrana, která má jeden konec na vrcholu v komponentě a druhý mimo)

Formálněji by se dal zapsat asi takto

1. Pro všechny vrcholy ***v*** *-* stav[***v***] ←*mimo*
2. Libovolný počáteční vrchol ***v0***
3. stav[***v0***] ←uvnitř
4. ***T*** ← prázdná množina hran (zde stavíme minimální kostru)
5. ***H*** ←minimální halda řazená podle váhy hran, obsahuje všechny hrany *v0*
6. **Dokud** ***H*** není prázdná:
7. Odeberme minimální hranu ***e*** (***u***,***v***) z haldy ***H***, kde stav[***u***]==*uvnitř* (alespoň jeden z vrcholů vždy je)
8. **Pokud**stav[***v***] ==*uvnitř*
9. Skoč na 6.
10. Přidej ***e*** do ***T***
11. stav[***v***] ←*uvnitř*
12. Přidej všechny hrany ***v*** do ***H***
13. ***T*** je minimální kostrou grafu

**Časová a prostorová složitost\***

Hlavním prvkem celého algoritmu je smyčka. Ta běží dokud jsou nějaké hrany v haldě, při každém průběhu jednu hranu odebere. Každá hrana může být do haldy přidána maximálně dvakrát (jednou každým z krajních vrcholů). Smyčka tedy proběhne maximálně *O(M)*-krát, kde *M* je počet hran.

V každém průběhu cyklu odebíráme jednu hranu z haldy, potom nějaké hrany možná přidáme, za každou hranu tedy uděláme kostantní počet operací na haldě. Všechny ostatní kroky ve smyčce trvají konstantní čas. Celková časová složitost je tedy *O(MlogM)*, protože operace na haldě trvá logaritmický čas oproti počtu prvků.

Potřebujeme si pamatovat stav pro každý vrchol, vybrané hrany a haldu, celkem tedy *O(N+M)* paměti.

\* Jelikož *N<=M<N2* tak se zanedbáním konstant platí *O(MlogM)<=O(MlogN2)=O(MlogN)*

Můžeme taky zjednodušit *O(N+M)=O(M)*

**Důkaz správnosti**

Jarníkův algoritmus v každém kroku k již propojené komponentě přidá nejkratší hranu vedoucí ven. Uvažujme řez rozdělující aktuálně propojenou komponentu od zbytku grafu v libovolný moment průběhu algoritmu. Hrany vycházející z této komponenty jsou hrany uvažovaného řezu, a ta přidaná hrana je z nich minimální, podle řezového lemmatu tedy je součástí minimální kostry.

Jelikož toto platí pro každou přidanou hranu, je výsledná množina hran nutně minimální kostrou.

**Kruskalův algoritmus**

Tento algoritmus zkouší přidávat hrany od té nejkratší, pokud hrana spojí dosud nespojené komponenty, přidá ji.

Postup je tedy jednoduchý, nejdříve si seřadíme všechny hrany podle váhy, potom je postupně procházíme od nejlehčí, a ty hrany, které spojí zatím nespojené komponenty přidáme.

1. Seřadit hrany podle váhy
2. ***T*** - Graf s *N* vrcholy, každý vrchol je ve vlastní komponentě (graf je tedy bez hran, budeme zde stavět minimální kostru)
3. Pro každou hranu ***e****(****u****,****v****)* od nejlehčí opakujme:
4. **Pokud** ***u****,****v*** jsou v různých komponentách
5. Přidejme hranu ***e*** do ***T***
6. ***T*** je minimální kostra

Zbývá vyřešit, jak zjistit, jestli vrcholy patří do stejné komponenty. Šlo by to pomocí nějakého prohledávání v lineárním čase, to je ale moc pomalé. Použijeme k tomu hezkou datovou strukturu zvanou Union-Find (nebo taky Disjoint Set Union). Ta je popsaná na konci dokumentu, teď nám stačí vědět, že dokáže zjistit, jestli prvky patří do stejné množiny v logaritmickém čase, stejně tak i spojení dvou množin.

**Časová složitost**

Nejdříve seřadí hrany za *O(MlogM).* Cyklus proběhne maximálně *M*-krát, uvnitř udělá nějaké operace na Union-Findu. Ceklová časová složitost je *O(MlogM)*

Union-Find potřebuje *O(N)* paměti, seřazený seznam hran *O(M)*, celkem je potřeba *O(N+M)* paměti. (Stejně jako u jarníka to jde zredukovat)

**Důkaz správnosti**

Uvažujme, že jsme právě narazili na hranu, která spojuje dosud nepropojené komponenty. Můžeme si vybrat jednu z komponent, které spojuje, a udělat řez oddělující tuto komponentu od zbytku grafu. Tento řez tedy prochází jen hranami, které ještě nebyly přidané, a zároveň vedou mezi dosud nespojenými komponentami. To znamená, že žádna z hran na řezu ještě nebyla vyhodnocená, a jelikož hrany vyhodnocujeme v pořadí podle váhy, tak je právě řešená hrana nejlehčí ze všech dosud nevyhodnocených hran a tedy nejlehčí na řezu. Tím pádem musí být součástí minimální kostry.

**Borůvkův algoritmus**

Borůvkův algoritmus funguje v krocích. V každém kroku pro každou komponentu najde minimální vycházející hranu, potom všechny tyto hrany přidá, čímž se komponenty spojí do větších. Takové kroky se opakují, dokud nezbývá jen jedna komponenta.

1. komponenta[***x***] ← ***x*** pro všechny ***x***
2. ***T*** ← graf obsahující ***N*** vrcholů a žádné hrany
3. **Dokud** ***T*** není spojitý:
4. ***k*** ← počet komponent
5. délka[***k***]; délka[***x***] ← ∞, pro všechny ***x*** od **1** do ***k***
6. hrana[***k***]
7. Pro všechny hrany ***e***(***u***,***v***):
8. **Pokud** komponenta[***u***]!=komponenta[***v***]:
9. **Pokud** váha ***e*** < délka[***u***]:
10. délka[***u***] ← váha ***e***
11. hrana[***u***] ← ***e***
12. **Pokud** váha ***e*** < délka[***v***]:
13. délka[***v***] ← váha ***e***
14. hrana[***v***] ← ***e***
15. Přidej hrana[***x***] do ***T*** pro všechny ***x***
16. Přepočítej komponenty
17. ***k*** ← nový počet komponent
18. **Pokud** ***k*** = 1, ukonči program
19. ***T*** je minimální kostra

Přepočítat komponenty můžeme pomocí prohledávání do hloubky v lineárním čase. Alternativně můžeme komponenty řešit pomocí Union-Findu.

Při každém kroku se každá komponenta spojí alespoň s jednou jinou, tudíž počet komponent vždy klesne alespoň na polovinu. Provede se tedy maximálně *logN* kroků.

**Časová a Prostorová složitost**

V každém kroku procházíme všechny hrany, potom přepočítáme komponenty, obojí za *O(M)*. Časová složitost je *O(MlogN)*

Potřebujeme si pamatovat všechny hrany, pro každý vrchol do jaké komponenty patři, a pro každou komponentu nejlepší hranu, celkem tedy *O(N+M)*

**Důkaz správnosti**

Pro každou komponentu si vybereme minimální vycházející hranu. Pokud si tedy představíme řez oddělující komponentu od zbytku grafu, je tato hrana na tomto řezu minimální. Podle řezového lemmatu je tedy součástí minimální kostry.

**Union-Find**

Tato datová struktura udržuje rozdělení prvků do oddělených množin. Jmenuje se podle svýd dvou základních operací:

Union(*a*,*b*): spojí množiny obsahující *a* a *b*

Find(*a*): Zjistí, do jaké množiny patří *a*

Asi první způsob, co nás napadne, je uložit si do pole pro každý prvek, do jaké množiny patří. Tím dosáhneme konstantního času na operaci Find, ale Union potrvá lineární čas vzhledem k velikosti množin, tím si tedy moc nepomůžeme.

Klíčovou myšlekou je vybrat si v každé množině jeden z prvků jako zástupnce pro každou množinu. Find potom taky bude vracet zástupce dané množiny.

Pro každý prvek si budeme pamatovat jeho rodiče, pro prvek *x* jej označme *P(x)*. Pokud *x=P(x)*, pak je *x* zástupcem své množiny, jinak platí že *x* a *P(x)* patří do stejné množiny.

Každá množina je tedy efektivně strom, kde zástupce je kořen. Operace tedy můžeme implementovat následovně

**Find**(*a*):

**Pokud** *a=P(a)*:

**return** *a*

**return** **Find**(*P(a)*)

**Union**(*a,b*):

*u* ← **Find**(*a*)

*v* ← **Find**(*b*)

*P(u)*← *v*

Find tedy jde nahoru, než narazí na zástupce, toho potom vrátí. Union najde zástupce spojovaných množin, potom jednoho z nich napojí pod druhého. Časová složitost obou operací odpovídá hloubce.

**Optimalizace**

Vzhledem k tomu, že čas obou operací závisí na hloubce, bylo by dobré mít nad ní větší kontrolu. Pro každý prvek si budeme pamatovat, do jaké hloubky pod ním strom jde. Pokud spojujeme dva prvky, napojíme množinu s menší hloubkou pod tu s větší hloubkou. Pokud mají hloubku stejnou, napojíme je v libovolném pořadí, a novému zástupci spojené množiny zvýšíme hloubku o 1. Tímto způsobem zaručíme, že hloubka každé množiny je maximálně *log(n)*, kde n je počet prvků v množině. S tím už máme zaručený logaritmický čas obou operací. Pokud označíme hloubku, kterou si pamatujeme k danému prvku *x* jako *H(x)*, tak optimalizovaný union byde vypadat asi takto:

**Union**(*a,b*):

*u* ← **Find**(*a*)

*v* ← **Find**(*b*)

**Pokud** *H(u)>H(v)*:

*P(v)* ← *u*

**Pokud** *H(u)<H(v)*:

*P(u)* ← *v*

**Pokud** *H(u)=H(v)*:

*P(u)* ← *v*

*H(u)=H(u)+*1

*P(u)*← *v*

Druhou optimalizací je komprese cest. Když už ve funkci Find procházíme celou cestu od prvku do kořene, můžeme všechny prvky na cestě přesměrovat rovnou do kořene a zrychlit tak příští hledání. Find by tedy vypadalo asi takto

**Find**(*a*):

**Pokud** *a=P(a)*:

**return** *a*

*P(a)* = **Find**(*P(a)*)

**return** *P(a)*

To složitost operací sníží (amortizovaně) na inverzní ackermannovu funkci, což je prakticky konstanta. O tom je víc informací tady <https://pruvodce.ucw.cz/static/pruvodce.pdf#s7.5>