1. Minimální kostry

Napadl sníh a přikryl peřinou celé městečko. Po ulicích lze sotva projít pěšky, natož projet autem. Které ulice prohrneme, aby šlo dojet odkudkoliv kamkoliv, a přitom nám házení sněhu dalo co nejméně práce?

Tato otázka vede na hledání minimální kostry grafu. To je slavný problém, jeden z těch, které stály u pomyslné kolébky teorie grafů. Navíc je pro jeho řešení známo hned několik zajímavých efektivních algoritmů. Jim věnujeme tuto kapitolu.

1.1. Od městečka ke kostře

Představme si mapu zasněženého městečka z našeho úvodního příkladu jako graf. Každou hranu ohodnotíme číslem – to bude vyjadřovat množství práce potřebné na prohrnutí ulice. Hledáme tedy podgraf na všech vrcholech, který bude souvislý a použije hrany o co nejmenším součtu ohodnocení.

Takový podgraf jistě musí být strom: kdyby se v něm nacházel nějaký cyklus, smažeme libovolnou z hran cyklu. Tím neporušíme souvislost, protože konce hrany jsou nadále propojené zbytkem cyklu. Odstraněním hrany ovšem zlepšíme součet ohodnocení, takže původní podgraf nemohl být optimální. (Zde jsme použili, že odhrnutí sněhu nevyžaduje záporné množství práce, ale to snad není moc troufalé.)

Popišme nyní problém formálně.

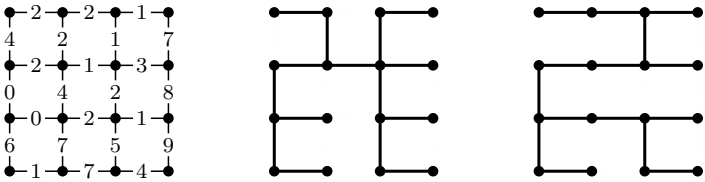
Definice:

- Nechť $G = (V, E)$ je souvislý neorientovaný graf a $w : E \to \mathbb{R}$ váhová funkce, která přiřazuje hranám čísla – jejich *váhy*.
- n a m nechť jako obvykle značí počet vrcholů a hran grafu G.
- Váhovou funkci můžeme přirozeně rozšířit na podgrafy: Váha $w(H)$ podgrafu $H \subseteq G$ je součet vah jeho hran.
- *Kostra* grafu G je podgraf, který obsahuje všechny vrcholy a je to strom. Kostra je *minimální,* pokud má mezi všemi kostrami nejmenší váhu.

Jak je vidět z obrázku 1.1, jeden graf může mít více minimálních koster. Brzy dokážeme, že jsou-li váhy všech hran navzájem různé, minimální kostra už je určena jednoznačně. To značně zjednoduší situaci, takže ve zbytku kapitoly budeme unikátnost vah předpokládat.

Cvičení

1. Rozmyslete si, že předpoklad unikátních vah není na škodu obecnosti. Ukažte, jak pomocí algoritmu, který unikátnost předpokládá, nalézt jednu z minimálních koster grafu s neunikátními vahami.
2. Upravte definici kostry, aby dávala smysl i pro nesouvislé grafy.
3. Dokažte, že mosty v grafu jsou právě ty hrany, které leží v průniku všech koster.



Obr. 1.1: Graf s vahami a dvě z jeho minimálních koster

4. Změníme-li váhu jedné hrany, jak se změní minimální kostra?

5.* Známe-li minimální kostru, jak najít druhou nejmenší?

1.2. Jarníkův algoritmus a řezy

Vůbec nejjednodušší algoritmus pro hledání minimální kostry pochází z roku 1930, kdy ho vymyslel český matematik Vojtěch Jarník. Tehdy se o algoritmy málokdo zajímal, takže myšlenka zapadla a až později byla několikrát znovuobjevena – proto se algoritmu říká též Primův nebo Dijkstrův.

Kostru budeme „pěstovat“ z jednoho vrcholu. Začneme se stromem, který obsahuje libovolný jeden vrchol a žádné hrany. Pak vybereme nejlehčí hranu incidentní s tímto vrcholem. Přidáme ji do stromu a postup opakujeme: v každém dalším kroku přidáváme nejlehčí z hran, které vedou mezi vrcholy stromu a zbytkem grafu. Takto pokračujeme, dokud nevznikne celá kostra.

Algoritmus JARNÍK
Vstup: Souvislý graf s unikátními vahami

1. $v_0 \leftarrow$ libovolný vrchol grafu
2. $T \leftarrow$ strom obsahující vrchol v_0 a žádné hrany
3. Dokud existuje hrana uv taková, že $u \in V(T)$ a $v \notin V(T)$:
4. Nejlehčí takovou hranu přidáme do T.

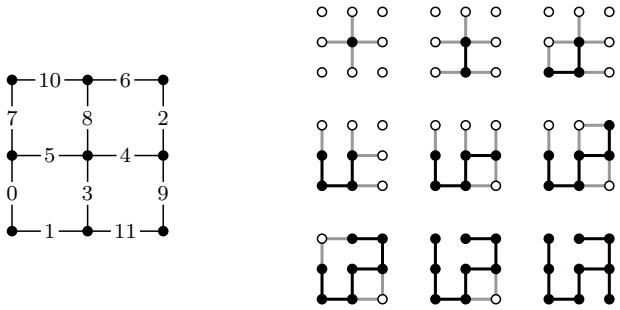
Výstup: Minimální kostra T

Tento přístup je typickým příkladem takzvaného *hladového algoritmu* – v každém okamžiku vybíráme lokálně nejlepší hranu a neohlížíme se na budoucnost.⟨1⟩ Hladové algoritmy málokdy naleznou optimální řešení, ale zrovna minimální kostra je jedním z řídkých případů, kdy tomu tak je. K důkazu se ovšem budeme muset propracovat.

Správnost

Lemma: Jarníkův algoritmus se po nejvýše n iteracích zastaví a vydá nějakou kostru zadaného grafu.

⟨1⟩ Proto je možná výstižnější anglický název *greedy algorithm,* čili algoritmus chamtivý, nebo slovenský *pažravý algoritmus.*



Obr. 1.2: Příklad výpočtu Jarníkova algoritmu.
Černé vrcholy a hrany už byly přidány do kostry,
mezi šedivými hranami hledáme tu nejlehčí.

Důkaz: Graf pěstovaný algoritmem vzniká z jednoho vrcholu postupným přidáváním listů, takže je to v každém okamžiku výpočtu strom. Po nejvýše n iteracích dojdou vrcholy a algoritmus se musí zastavit.

Kdyby nalezený strom neobsahoval všechny vrcholy, musela by díky souvislosti existovat hrana mezi stromem a zbytkem grafu. Tehdy by se ale algoritmus ještě nezastavil. (Všimněte si, že tuto úvahu jsme už potkali v rozboru algoritmů na prohledávání grafu.) □

Minimalitu kostry bychom mohli dokazovat přímo, ale raději dokážeme trochu obecnější tvrzení o řezech, které se bude hodit i pro další algoritmy.

Definice: Nechť A je nějaká podmnožina vrcholů grafu a B její doplněk. Množině hran, které leží jedním vrcholem v A a druhým v B, budeme říkat *elementární řez* určený množinami A a B.

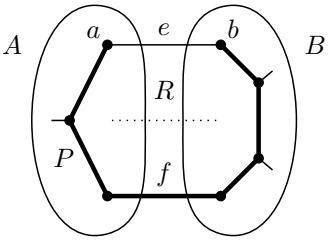
Lemma (řezové): Nechť G je graf opatřený unikátními vahami, R nějaký jeho elementární řez a e nejlehčí hrana tohoto řezu. Pak e leží v každé minimální kostře grafu G.

Důkaz: Dokážeme obměněnou implikaci: pokud nějaká kostra T neobsahuje hranu e, není minimální.

Sledujme situaci na obrázku 1.3. Označme A a B množiny vrcholů, kterými je určen řez R. Hrana e tudíž vede mezi nějakými vrcholy $a \in A$ a $b \in B$. Kostra T musí spojovat vrcholy a a b nějakou cestou P. Tato cesta začíná v množině A a končí v B, takže musí alespoň jednou překročit řez. Nechť f je libovolná hrana, kde se to stalo.

Nyní z kostry T odebereme hranu f. Tím se kostra rozpadne na dva stromy, z nichž jeden obsahuje a a druhý b. Přidáním hrany e stromy opět propojíme a tím získáme jinou kostru T'.

Spočítáme její váhu: $w(T') = w(T) - w(f) + w(e)$. Jelikož hrana e je nejlehčí v řezu, musí platit $w(f) \geq w(e)$. Nerovnost navíc musí být ostrá, neboť váhy jsou unikátní. Proto $w(T') < w(T)$ a T není minimální. □



Obr. 1.3: Situace v důkazu řezového lemmatu

Každá hrana vybraná Jarníkovým algoritmem je přitom nejlehčí hranou elementárního řezu mezi vrcholy stromu T a zbytkem grafu. Z řezového lemmatu proto plyne, že kostra nalezená Jarníkovým algoritmem je podgrafem každé minimální kostry. Jelikož všechny kostry daného grafu mají stejný počet hran, znamená to, že nalezená kostra je všem minimálním kostrám rovna. Proto platí:

Věta (o minimální kostře): Souvislý graf s unikátními vahami má právě jednu minimální kostru a Jarníkův algoritmus tuto kostru najde.

Navíc víme, že Jarníkův algoritmus váhy pouze porovnává, takže ihned dostáváme:

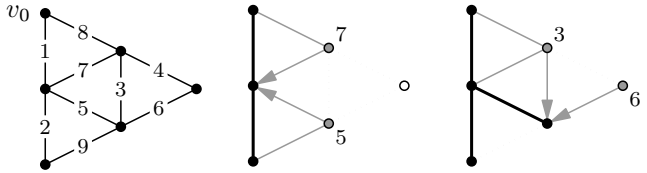
Důsledek: Minimální kostra je jednoznačně určena uspořádáním hran podle vah, na konkrétních hodnotách vah nezáleží.

Implementace

Zbývá rozmyslet, jak rychle algoritmus poběží. Už víme, že proběhne nejvýše n iterací. Pokud budeme pokaždé zkoumat všechny hrany, jedna iterace potrvá $\mathcal{O}(m)$, takže celý algoritmus poběží v čase $\mathcal{O}(nm)$.

Opakované vybírání minima navádí k použití haldy. Mohli bychom v haldě uchovávat množinu všech hran řezu (viz cvičení 1), ale existuje elegantnější a rychlejší způsob.

Budeme udržovat *sousední* vrcholy – to jsou ty, které leží mimo strom, ale jsou s ním spojené alespoň jednou hranou. Každému sousedovi s přiřadíme *ohodnocení* $h(s)$. To bude udávat, jakou nejlehčí hranou je soused připojen ke stromu.



Obr. 1.4: Jeden krok výpočtu v Jarníkově algoritmu s haldou

V každém kroku algoritmu vybereme souseda s nejnižším ohodnocením a připojíme ho ke stromu příslušnou nejlehčí hranou. To je přesně ta hrana, kterou si

vybere původní Jarníkův algoritmus. Poté potřebujeme přepočítat sousedy a jejich ohodnocení.

Sledujme obrázek 1.4. Vlevo je nakreslen zadaný graf s vahami. Uprostřed vidíme situaci v průběhu výpočtu: tučné hrany už leží ve stromu, šedivé vrcholy jsou sousední (čísla udávají jejich ohodnocení), šipky ukazují, která hrana řezu je pro daného souseda nejlehčí. V tomto kroku tedy vybereme vrchol s ohodnocením 5, čímž přejdeme do situace nakreslené vpravo.

Pozorování: Pokud ke stromu připojujeme vrchol u, musíme přepočítat sousednost a ohodnocení ostatních vrcholů. Uvažme libovolný vrchol v a rozeberme možné situace:

- Pokud byl v součástí stromu, nemůže se stát sousedním, takže se o něj nemusíme starat.
- Pokud mezi u a v nevede hrana, v okolí vrcholu v se řez nezmění, takže ohodnocení $h(v)$ zůstává stejné.
- Jinak se hrana uv stane hranou řezu. Tehdy:
  - Pakliže v nebyl sousední, stane se sousedním a jeho ohodnocení nastavíme na váhu hrany uv.
  - Pokud už sousední byl, bude se do jeho ohodnocení nově započítávat hrana uv, takže $h(v)$ může klesnout.

Stačí tedy projít všechny vrcholy v, do nichž vede z u hrana, a podle potřeby učinit v sousedem nebo snížit jeho ohodnocení.

Na této myšlence je založena následující varianta Jarníkova algoritmu. Kromě ohodnocení vrcholů si budeme pamatovat jejich *stav* (*uvnitř* stromu, *sousední,* případně úplně *mimo*) a u sousedních vrcholů příslušnou nejlehčí hranu. Při inicializaci algoritmu chvíli považujeme počáteční vrchol za souseda, což zjednoduší zápis.

Algoritmus JARNÍK2

Vstup: Souvislý graf s váhovou funkcí w

1. Pro všechny vrcholy v:
2. $\quad stav(v) \leftarrow$ *mimo*
3. $\quad h(v) \leftarrow +\infty$
4. $\quad p(v) \leftarrow$ *nedefinováno* ◁ *druhý konec nejlehčí hrany*
5. $v_0 \leftarrow$ libovolný vrchol grafu
6. $T \leftarrow$ strom obsahující vrchol v_0 a žádné hrany
7. $stav(v_0) \leftarrow$ *soused*
8. $h(v_0) \leftarrow 0$
9. Dokud existují nějaké sousední vrcholy:
10. $\quad$ Označme u sousední vrchol s nejmenším $h(u)$.
11. $\quad stav(u) \leftarrow$ *uvnitř*
12. $\quad$ Přidáme do T hranu $\{u, p(u)\}$, pokud je $p(u)$ definováno.
13. $\quad$ Pro všechny hrany uv:

14. Je-li $stav(v) \in \{soused, mimo\}$ a $h(v) > w(uv)$:
15. $stav(v) \leftarrow soused$
16. $h(v) \leftarrow w(uv)$
17. $p(v) \leftarrow u$

Výstup: Minimální kostra T

Všimněte si, že takto upravený Jarníkův algoritmus je velice podobný Dijkstrovu algoritmu na hledání nejkratší cesty. Jediný podstatný rozdíl je ve výpočtu ohodnocení vrcholů.

Platí zde tedy vše, co jsme odvodili o složitosti Dijkstrova algoritmu v oddílu **??**. Uložíme-li všechna ohodnocení do pole, algoritmus poběží v čase $\Theta(n^2)$. Pokud místo pole použijeme haldu, kostru najdeme v čase $\Theta(m \log n)$, případně s Fibonacciho haldou v $\Theta(m + n \log n)$.

Cvičení

1. V rozboru implementace jsme navrhovali uložit všechny hrany řezu do haldy. Rozmyslete si všechny detaily tak, aby váš algoritmus běžel v čase $\mathcal{O}(m \log n)$.
2. Dokažte správnost Jarníkova algoritmu přímo, bez použití řezového lemmatu.
3. Dokažte, že Jarníkův algoritmus funguje i pro grafy, jejichž váhy nejsou unikátní. Jak by pro takové grafy vypadalo řezové lemma?

1.3. Borůvkův algoritmus

Inspirací Jarníkova algoritmu byl algoritmus ještě starší, objevený v roce 1926 Otakarem Borůvkou, pozdějším profesorem matematiky v Brně. Můžeme se na něj dívat jako na paralelní verzi Jarníkova algoritmu: namísto jednoho stromu jich pěstujeme více a v každé iteraci se každý strom sloučí s tím ze svých sousedů, do kterého vede nejlehčí hrana.

Algoritmus Borůvka

Vstup: Souvislý graf s unikátními vahami

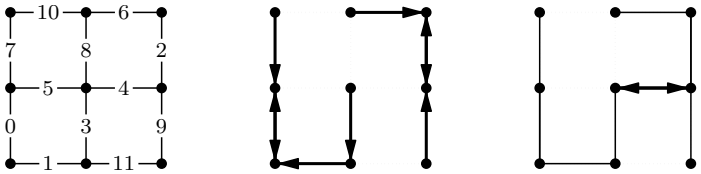
1. $T \leftarrow (V, \emptyset)$ ◁ *začneme triviálním lesem izolovaných vrcholů*
2. Dokud T není souvislý:
3. Rozložíme T na komponenty souvislosti $T_1, \ldots, T_k$.
4. Pro každý strom T_i najdeme nejlehčí z hran mezi T_i a zbytkem grafu a označíme ji e_i.
5. Přidáme do T hrany $\{e_1, \ldots, e_k\}$.

Výstup: Minimální kostra T

Správnost dokážeme podobně jako u Jarníkova algoritmu.

Věta: Borůvkův algoritmus se zastaví po nejvýše $\lfloor \log_2 n \rfloor$ iteracích a vydá minimální kostru.

Důkaz: Nejprve si všimneme, že po k iteracích má každý strom lesa T alespoň 2^k vrcholů. To dokážeme indukcí podle k: na počátku ($k = 0$) jsou všechny stromy



Obr. 1.5: Příklad výpočtu Borůvkova algoritmu. Směr šipek ukazuje, který vrchol si vybral kterou hranu.

jednovrcholové. V každé další iteraci se stromy slučují do větších, každý s alespoň jedním sousedním. Proto se velikosti stromů pokaždé minimálně zdvojnásobí.

Nejpozději po $\lfloor \log_2 n \rfloor$ iteracích už velikost stromů dosáhne počtu všech vrcholů, takže může existovat jen jediný strom a algoritmus se zastaví. (Zde jsme opět použili souvislost grafu, rozmyslete si, jak přesně.)

Zbývá nahlédnout, že nalezená kostra je minimální. Opět použijeme řezové lemma: každá hrana e_i, kterou jsme vybrali, je nejlehčí hranou elementárního řezu mezi stromem T_i a zbytkem grafu. Všechny vybrané hrany tedy leží v jednoznačně určené minimální kostře a je jich správný počet. (Zde jsme potřebovali unikátnost vah, viz cvičení 1.) □

Ještě si rozmyslíme implementaci. Ukážeme, že každou iteraci lze zvládnout v lineárním čase s velikostí grafu. Rozklad na komponenty provedeme například prohledáním do šířky. Poté projdeme všechny hrany, pro každou se podíváme, které komponenty spojuje, a započítáme ji do průběžného minima obou komponent. Nakonec vybrané hrany přidáme do kostry.

Důsledek: Borůvkův algoritmus nalezne minimální kostru v čase $\mathcal{O}(m \log n)$.

Cvičení

1. Unikátnost vah je u Borůvkova algoritmu důležitá, protože jinak by v kostře mohl vzniknout cyklus. Najděte příklad grafu, kde se to stane. Jak přesně pro takové grafy selže náš důkaz správnosti? Jak algoritmus opravit?
2. Borůvkův algoritmus můžeme upravit, aby každý strom lesa udržoval zkontrahovaný do jednoho vrcholu. Iterace pak vypadá tak, že si každý vrchol vybere nejlehčí incidentní hranu, tyto hrany zkontrahujeme a zapamatujeme si, že patří do minimální kostry. Ukažte, jak tento algoritmus implementovat tak, aby běžel v čase $\mathcal{O}(m \log n)$. Jak si poradit s násobnými hranami a smyčkami, které vznikají při kontrakci?
3. Sestrojte příklad grafu, na kterém algoritmus z předchozího cvičení potřebuje čas $\Omega(m \log n)$.
4.* Ukažte, že pokud algoritmus z cvičení 2 používáme pro rovinné grafy, běží v čase $\Theta(n)$. Opět je potřeba správně ošetřit násobné hrany.

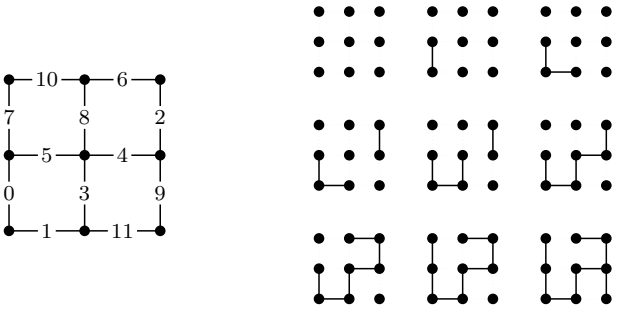
1.4. Kruskalův algoritmus a Union-Find

Třetí algoritmus na hledání minimální kostry popsal v roce 1956 Joseph Kruskal. Opět je založen na hladovém přístupu: zkouší přidávat hrany od nejlehčí po nejtěžší a zahazuje ty, které by vytvořily cyklus.

Algoritmus KRUSKAL
Vstup: Souvislý graf s unikátními vahami

1. Uspořádáme hrany podle vah: $w(e_1) < \ldots < w(e_m)$.
2. $T \leftarrow (V, \emptyset)$ ◁ *začneme lesem samých izolovaných vrcholů*
3. Pro $i = 1, \ldots, m$ opakujeme:
4. $\quad u, v \leftarrow$ krajní vrcholy hrany e_i
5. $\quad$ Pokud u a v leží v různých komponentách lesa T:
6. $\quad\quad T \leftarrow T + e_i$

Výstup: Minimální kostra T



Obr. 1.6: Příklad výpočtu Kruskalova algoritmu.

Lemma: Kruskalův algoritmus se zastaví a vydá minimální kostru.

Důkaz: Konečnost je zřejmá z omezeného počtu průchodů hlavním cyklem. Nyní ukážeme, že hranu $e = uv$ algoritmus přidá do T právě tehdy, když e leží v minimální kostře.

Pokud algoritmus hranu přidá, stane se tak v okamžiku, kdy se vrcholy u a v nacházejí v nějakých dvou rozdílných stromech T_u a T_v lesa T. Hrana e přitom leží v elementárním řezu oddělujícím strom T_u od zbytku grafu. Navíc mezi hranami tohoto řezu musí být nejlehčí, neboť případnou lehčí hranu by algoritmus potkal dříve a už by stromy spojil. Nyní stačí použít řezové lemma.

Jestliže naopak algoritmus hranu e nepřidá, činí tak proto, že hrana uzavírá cyklus. Ostatní hrany tohoto cyklu algoritmus přidal, takže podle minulého odstavce tyto hrany leží v minimální kostře. Kostra ovšem žádné cykly neobsahuje, takže v ní e určitě neleží. □

Nyní se zamysleme nad implementací. Třídění hran potrvá $\mathcal{O}(m \log m) \subseteq \mathcal{O}(m \log n^2) = \mathcal{O}(m \log n)$. Zbytek algoritmu potřebuje opakovaně testovat, zda hrana spojuje dva různé stromy. Jistě bychom mohli pokaždé prohledat les do šířky, ale to by trvalo $\mathcal{O}(n)$ na jeden test, celkově tedy $\mathcal{O}(nm)$.

Opakované prohledávání znamená spoustu zbytečné práce. Les se totiž mezi jednotlivými kroky algoritmu mění pouze nepatrně – buď zůstává stejný, nebo do něj přibude jedna hrana. Neuměli bychom komponenty průběžně přepočítávat? Na to by se hodila následující datová struktura:

Definice: *Struktura Union-Find* reprezentuje komponenty souvislosti grafu a umí na nich provádět následující operace:

- FIND(u, v) zjistí, zda vrcholy u a v leží v téže komponentě.
- UNION(u, v) přidá hranu uv, čili dvě komponenty spojí do jedné.

V kroku 5 Kruskalova algoritmu tedy provádíme operaci FIND a v kroku 6 UNION. Složitost celého algoritmu proto můžeme vyjádřit následovně:

Věta: Kruskalův algoritmus najde minimální kostru v čase $\mathcal{O}(m \log n + m \cdot T_f(n) + n \cdot T_u(n))$, kde $T_f(n)$ a $T_u(n)$ jsou časové složitosti operací FIND a UNION na grafech s n vrcholy.

Union-Find s polem

Hledejme nyní rychlou implementaci struktury Union-Find. Nejprve zkusíme, kam nás zavede triviální přístup: pořídíme si pole K, které každému vrcholu přiřadí číslo komponenty. Můžeme si ji představovat jako barvu vrcholu.

FIND se podívá na barvy vrcholů a v konstantním čase je porovná. Veškerou práci oddře UNION: při slučování komponent projde všechny vrcholy jedné komponenty a přebarví je.

Procedura FIND(u, v)

1. Odpovíme ANO právě tehdy, když $K(u) = K(v)$.

Procedura UNION(u, v)

1. Pro všechny vrcholy x:
2. Pokud $K(x) = K(u)$:
3. $K(x) \leftarrow K(v)$

FIND proběhne v čase $\mathcal{O}(1)$ a UNION v $\mathcal{O}(n)$, takže celý Kruskalův algoritmus potrvá $\mathcal{O}(m \log n + m + n^2) = \mathcal{O}(m \log n + n^2)$.

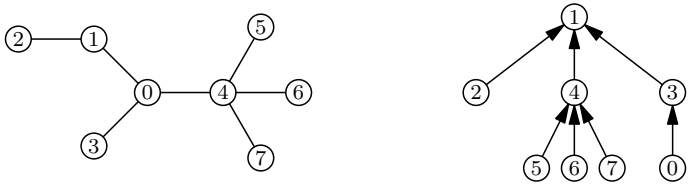
Kvadratická složitost nás sotva uspokojí. Můžeme se pokusit přečíslovávání komponent zrychlit (viz cvičení 3), ale místo toho raději změníme reprezentaci struktury.

Union-Find s keříky

Každou komponentu budeme reprezentovat stromem orientovaným směrem do kořene. Těmto stromům budeme říkat *keříky,* abychom je odlišili od stromů, s nimiž pracuje Kruskalův algoritmus.

Vrcholy každého keříku budou odpovídat vrcholům příslušné komponenty. Hrany nemusí odpovídat hranám původního grafu, jejich podoba záleží na historii operací s naší datovou strukturou.

Do paměti můžeme keříky ukládat přímočaře: každý vrchol v si bude pamatovat svého otce $P(v)$, případně nějakou speciální hodnotu $\emptyset$, pokud je kořenem.



Obr. 1.7: Komponenta a její reprezentace keříkem

Operace Find vystoupá z každého vrcholu do kořene keříku a porovná kořeny:

Procedura Kořen(x)

1. Dokud $P(x) \neq \emptyset$:
2. $\quad x \leftarrow P(x)$
3. Vrátíme kořen x.

Procedura Find(u, v)

1. Vrátíme ano právě tehdy, když Kořen(u) = Kořen(v).

Hledání kořene, a tím pádem i operace Find trvají lineárně s hloubkou keříku.

Operace Union sloučí komponenty tak, že mezi kořeny keříků natáhne novou hranu. Může si přitom vybrat, který kořen připojí pod který – obojí bude správně. Ukážeme, že vhodnou volbou udržíme keříky mělké a Find rychlý.

Uděláme to takto: Do kořene každého keříku uložíme číslo $H(v)$, jež bude říkat, jak je tento keřík hluboký. Na počátku mají všechny keříky hloubku 0. Při slučování keříků připojíme mělčí keřík pod kořen toho hlubšího a hloubka se nezmění. Jsou-li oba stejně hluboké, rozhodneme se libovolně a keřík se prohloubí. Union bude vypadat takto:

Procedura Union(u, v):

1. $a \leftarrow$ Kořen(u), $b \leftarrow$ Kořen(v)
2. Je-li $a = b$, ihned skončíme.
3. Pokud $H(a) < H(b)$:
4. $\quad P(a) \leftarrow b$
5. Pokud $H(a) > H(b)$:
6. $\quad P(b) \leftarrow a$
7. Jinak:
8. $\quad P(b) \leftarrow a$

9. $H(a) \leftarrow H(a) + 1$

Teď ukážeme, že naše slučovací pravidlo zaručí, že keříky jsou vždy mělké (a zaslouží si svůj název).

Invariant: Keřík hloubky h obsahuje alespoň 2^h vrcholů.

Důkaz: Budeme postupovat indukcí podle počtu operací UNION. Na počátku algoritmu mají všechny keříky hloubku 0 a $2^0 = 1$ vrchol.

Nechť nyní provádíme UNION(u, v) a hloubky obou keříků jsou různé. Připojením mělčího keříku pod kořen toho hlubšího se hloubka nezmění a počet vrcholů neklesne, takže nerovnost stále platí.

Pokud mají oba keříky tutéž hloubku h, víme z indukčního předpokladu, že každý z nich obsahuje minimálně 2^h vrcholů. Jejich sloučením tudíž vznikne keřík hloubky $h + 1$ o alespoň $2 \cdot 2^h = 2^{h+1}$ vrcholech. Nerovnost je tedy opět splněna. □

Důsledek: Hloubky keříků nepřekročí $\log n$.

Důkaz: Strom větší hloubky by podle invariantu obsahoval více než n vrcholů. □

Věta: Časová složitost operací UNION a FIND v keříkové reprezentaci je $\mathcal{O}(\log n)$.

Důkaz: Hledání kořene keříku zabere čas lineární s jeho hloubkou, tedy $\mathcal{O}(\log n)$. Obě operace datové struktury provedou dvě hledání kořene a $\mathcal{O}(1)$ dalších operací. □

Důsledek: Kruskalův algoritmus s keříkovou strukturou pro Union-Find najde minimální kostru v čase $\mathcal{O}(m \log n)$.

Cvičení

1. Dokažte správnost Kruskalova algoritmu přímo, bez použití řezového lemmatu.
2. Fungoval by Kruskalův algoritmus pro neunikátní váhy hran?
3. Datová struktura pro Union-Find s polem by se dala zrychlit tím, že bychom pokaždé přečíslovávali tu menší z komponent. Dokažte, že pak je během života struktury každý vrchol přečíslován nejvýše $(\log n)$-krát. Co z toho plyne pro složitost operací? Nezapomeňte, že je potřeba efektivně zjistit, která z komponent je menší, a vyjmenovat její vrcholy.
4. Jaká posloupnost UNIONů odpovídá obrázku 1.7?

1.5.* Komprese cest

Keříkovou datovou strukturu můžeme dále zrychlovat. Kruskalův algoritmus tím sice nezrychlíme, protože nás stejně brzdí třídění hran. Ale co kdybychom hrany dostali už setříděné, nebo jejich váhy byly celočíselné a šlo je třídit přihrádkově? Tehdy můžeme operace s keříky zrychlit ještě jedním trikem: *kompresí cest.*

Kdykoliv hledáme kořen nějakého keříku, trávíme tím čas lineární v délce cesty do kořene. Když už to děláme, zkusme při tom strukturu trochu vylepšit. Všechny vrcholy, přes které jsme prošli, převěsíme rovnou pod kořen. Tím si ušetříme práci v budoucnosti.

Procedura KořenSKompresí(x)

1. $r \leftarrow$ Kořen(x)
2. Dokud $P(x) \neq r$:
3. $\quad t \leftarrow P(x)$
4. $\quad P(x) \leftarrow r$
5. $\quad x \leftarrow t$
6. Vrátíme kořen r.

Pozor na to, že převěšením vrcholů mohla klesnout hloubka keříku. Uložené hloubky, které používáme v UNIONech, tím pádem přestanou souhlasit se skutečností. Místo abychom je přepočítávali, necháme je být a přejmenujeme je. Budeme jim říkat *ranky* a budeme s nimi zacházet úplně stejně, jako jsme předtím zacházeli s hloubkami.[2]

Podobně jako u původní struktury bude platit následující invariant:

Invariant R: Keřík s kořenem ranku r má hloubku nejvýše r a obsahuje alespoň 2^r vrcholů.

Důkaz: Indukcí podle počtu operací UNION. Komprese cest nemění ani rank kořene, ani počet vrcholů, takže se jí nemusíme zabývat. □

Ranky jsou tedy stejně jako hloubky nejvýše logaritmické, takže složitost operací v nejhorším případě zůstává $\mathcal{O}(\log n)$. Ukážeme, že průměrná složitost se výrazně snížila. (To je typický příklad takzvané amortizované složitosti, s níž se blíže setkáme v kapitole **??**.)

Definice: *Věžovou funkci* $2 \uparrow k$ definujeme následovně: $2 \uparrow 0 = 1$, $2 \uparrow (k+1) = 2^{2 \uparrow k}$.

Definice: *Iterovaný logaritmus* $\log^* x$ je inverzí věžové funkce. Udává nejmenší k takové, že $2 \uparrow k \geq x$.

Příklad: Funkce $2 \uparrow k$ roste přímo závratně:

$$\begin{aligned} 2 \uparrow 1 &= 2, \\ 2 \uparrow 2 &= 2^2 = 4 \\ 2 \uparrow 3 &= 2^4 = 16 \\ 2 \uparrow 4 &= 2^{16} = 65\,536 \\ 2 \uparrow 5 &= 2^{65\,536} \approx 10^{19\,728} \end{aligned}$$

Iterovaný logaritmus libovolného „rozumného“ čísla je tedy nejvýše 5.

Věta: Ve struktuře s kompresí cest na n vrcholech trvá provedení $n-1$ operací UNION a m operací FIND celkově $\mathcal{O}((n+m) \cdot \log^* n)$.

Ve zbytku tohoto oddílu větu dokážeme.

[2] Anglický *rank* by se dal do češtiny přeložit jako *hodnost*. V lineární algebře se pojem hodnosti používá, ale při studiu datových struktur bývá zvykem používat původní anglický termín.

Pro potřeby důkazu budeme uvažovat ranky všech vrcholů, nejen kořenů – každý vrchol si ponese svůj rank z doby, kdy byl naposledy kořenem. Struktura sama se ovšem podle ranků vnitřních vrcholů neřídí a nemusí si je ani pamatovat. Dokažme dva invarianty o rankách vrcholů.

Invariant C: Na každé cestě z vrcholu do kořene příslušného keříku ranky ostře rostou. Jinými slovy rank vrcholu, který není kořen, je menší, než je rank jeho otce.

Důkaz: Pro jednovrcholové keříky tvrzení jistě platí. Dále se keříky mění dvojím způsobem:

Přidání hrany v operaci UNION: Nechť připojíme vrchol b pod a. Cesty do kořene z vrcholů, které původně ležely pod a, zůstanou zachovány, pouze se vrcholu a mohl zvýšit rank. Cesty z vrcholů pod b se rozšíří o hranu ba, na které rank v každém případě roste.

Komprese cest nahrazuje otce vrcholu jeho vzdálenějším předkem, takže se rank otce může jedině zvýšit. □

Invariant P: Počet vrcholů ranku r nepřesáhne $n/2^r$.

Důkaz: Kdybychom nekomprimovali cesty, bylo by to snadné: vrchol ranku r by měl alespoň 2^r potomků (dokud je kořenem, plyne to z invariantu **R**; jakmile přestane být, potomci se už nikdy nezmění). Navíc díky invariantu **C** nemá žádný vrchol více předků ranku r, takže v keříku najdeme tolik disjunktních podkeříků velikosti alespoň 2^r, kolik je vrcholů ranku r.

Komprese cest nemůže invariant porušit, jelikož nemění ani ranky, ani rozhodnutí, jak proběhne který UNION. □

Nyní vrcholy ve struktuře rozdělíme do *skupin* podle ranků: k-tá skupina bude tvořena těmi vrcholy, jejichž rank je od $2\uparrow(k-1)+1$ do $2\uparrow k$. Vrcholy jsou tedy rozděleny do $1+\log^*\log n$ skupin (nezapomeňte, že ranky nepřesahují $\log n$). Odhadněme nyní shora počet vrcholů v k-té skupině.

Invariant S: V k-té skupině leží nejvýše $n/(2\uparrow k)$ vrcholů.

Důkaz: Sečteme odhad $n/2^r$ z invariantu **P** přes všechny ranky ve skupině:

$$\frac{n}{2^{2\uparrow(k-1)+1}}+\frac{n}{2^{2\uparrow(k-1)+2}}+\cdots+\frac{n}{2^{2\uparrow k}}\le\frac{n}{2^{2\uparrow(k-1)}}\cdot\sum_{i=1}^{\infty}\frac{1}{2^i}=\frac{n}{2^{2\uparrow(k-1)}}\cdot 1=\frac{n}{2\uparrow k}.$$

□

Důkaz věty: Operace UNION a FIND potřebují nekonstantní čas pouze na vystoupání po cestě ze zadaného vrcholu do kořene keříku. Čas strávený na této cestě je přímo úměrný počtu hran cesty. Celá cesta je přitom rozpojena a všechny vrcholy ležící na ní jsou přepojeny přímo pod kořen keříku.

Hrany cesty, které spojují vrcholy z různých skupin (takových je $\mathcal{O}(\log^* n)$), naúčtujeme právě prováděné operaci. Celkem jimi tedy strávíme čas $\mathcal{O}((n+m)\cdot\log^* n)$. Zbylé hrany budeme počítat přes celou dobu běhu algoritmu a účtovat je vrcholům.

Uvažme vrchol v v k-té skupině, jehož rodič leží také v k-té skupině. Jelikož hrany na cestách do kořene ostře rostou, každým přepojením vrcholu v rank jeho rodiče vzroste. Proto po nejvýše $2\uparrow k$ přepojeních se bude rodič vrcholu v nacházet v některé z vyšších skupin. Jelikož rank vrcholu v se už nikdy nezmění, bude hrana z v do jeho otce již navždy hranou mezi skupinami. Každému vrcholu v k-té skupině tedy naúčtujeme nejvýše $2\uparrow k$ přepojení a jelikož, jak už víme, jeho skupina obsahuje nejvýše $n/(2\uparrow k)$ vrcholů, naúčtujeme celé skupině čas $\mathcal{O}(n)$ a všem skupinám dohromady $\mathcal{O}(n \log^* n)$. □

Dodejme, že komprese cest se ve skutečnosti chová ještě lépe, než jsme dokázali. Správnou funkcí, která popisuje rychlost operací, není iterovaný logaritmus, ale ještě mnohem pomaleji rostoucí *inverzní Ackermannova funkce.* Rozdíl se nicméně projeví až pro nerealisticky velké vstupy a důkaz příslušné věty je zcela mimo možnosti našeho úvodního textu.

1.6. Další cvičení

1. Vymyslete algoritmus na hledání kostry grafu, v němž jsou váhy hran přirozená čísla od 1 do 5.

2.* Rozmyslete si, jak v případě, kdy váhy nejsou unikátní, najít *všechny* minimální kostry. Jelikož koster může být mnoho (pro úplný graf s jednotkovými vahami jich je n^{n-2}), snažte se o co nejlepší složitost v závislosti na velikosti grafu a počtu minimálních koster.

3. Jak hledat minimální kostru za předpokladu, že se určené vrcholy musí stát jejími listy? Jako další listy můžete využívat i neoznačené vrcholy.

4.* *Rekonstrukce metriky:* Mějme strom na množině $\{1, \ldots, n\}$ s ohodnocenými hranami. Metrika stromu je matice, která na pozici i, j udává vzdálenost mezi vrcholy i a j. Vymyslete algoritmus, jenž sestrojí strom se zadanou metrikou, případně odpoví, že takový strom neexistuje.