Grafové algoritmy

Karel Velička

18. ledna 2025

Mgr. Martin Mareš Ph.D.

Obsah

1 Toky v sítích

1.1 Formulace problému, základní definice

Definice 1.1. (Síť) je uspořádaná pětice (V, E, s, t, c), kde:

- (V, E) je *orientovaný graf*,
- $s \in V$ je *zdroj*,
- $t \in V$ je *spotřebič*, neboli *stok* a
- $c : E \to \mathbb{R}$ funkce udávající nezáporné *kapacity* hran.

Definice 1.2. (Ohodnocení) hran je libovolná funkce $f : E \to \mathbb{R}$. Pro každé ohodnocení f můžeme definovat:

$$f^+(v) = \sum_{e=(\cdot,v)} f(e), \quad f^-(v) = \sum_{e=(v,\cdot)} f(e), \quad f^\Delta(v) = f^+(v) - f^-(v),$$

co do vrcholu *přiteče*, co *odteče* a jaký je v něm *přebytek*.

Definice 1.3. (Tok) je ohodnocení $f : E \to \mathbb{R}$, pro které platí:

- $\forall e : 0 \leq f(e) \leq c(e)$, *(*dodržuje kapacity)
- $\forall v \neq s, t : f^\Delta(v) = 0$. *(*Kirchhoffův zákon)

Definice 1.4. (Velikost toku): $|f| = -f^\Delta(s) = f^\Delta(t)$.

Definice 1.5. (Elementární *st*-řez) pro dvě vzájemně disjunktní množiny $A, B \subseteq V$, kde $s \in A, t \notin A$, je:

$$E(A, B) := \{ab \in E \mid a \in A \wedge b \in B\}.$$

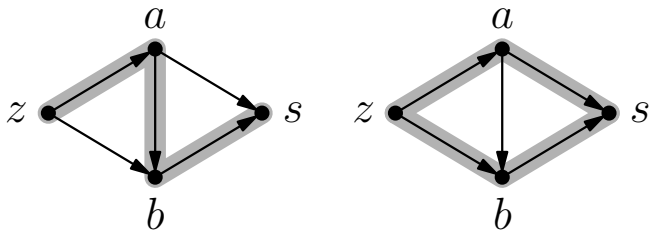
Definice 1.6. *Pro libovolné dvě množiny vrcholů A a B označíme $E(A, B)$ množinu hran vedoucích z A do B. Je-li dále f nějaká funkce přiřazující hranám čísla, označíme:*

- $f(A, B) := \sum_{e \in E(A,B)} f(e)$
- $f^\Delta(A, B) := f(A, B) - f(B, A)$

Definice 1.7. (Rezerva) $r : E \to \mathbb{R}_0^+$, že $r(uv) := (c(uv) - f(uv)) + f(vu)$.
Hrana $e \in E$ je *nasycená*, pokud $r(e) = 0$, jinak pro $r(e) > 0$ je hrana *nenasycená*.

Definice 1.8. (Zlepšující cesta) je orientovaná cesta, jejíž všechny hrany mají nenulovou rezervu.

Příklad 1.1. *Uvažujme například síť s jednotkovými kapacitami nakreslenou na obrázku. Najdeme-li nejdříve cestu zabs, zlepšíme po ní tok o 1. Tím dostaneme tok z levého obrázku, ve kterém už žádná další zlepšující cesta není. Jenže jak ukazuje pravý obrázek, maximální tok má velikost 2.*



Obrázek 1: Příklad, kdy algoritmus nefunguje.

Tuto prekérní situaci by zachránilo, kdybychom mohli poslat tok velikosti 1 proti směru hrany ab. Pak bychom tok z levého obrázku zlepšili po cestě zbas a získali bychom maximální tok z pravého obrázku. Posílat proti směru hrany ve skutečnosti nemůžeme, ale stejný efekt bude mít odečtení jedničky od toku po směru hrany.

1.2 Ford-Fulkerson algoritmus

Algoritmus 1.1. ***(Ford-Fulkerson algoritmus):*** *Nejpřímočařejší způsob, jak bychom mohli hledat toky v sítích, je začít s nějakým tokem (nulový je po ruce vždy) a postupně ho zlepšovat tak, že nalezneme nějakou nenasycenou cestu a pošleme po ní „co půjde". To bohužel nefunguje, ale můžeme tento postup trochu zobecnit a být ochotni používat nejen hrany, pro které je $f(e) < c(e)$, ale také hrany, po kterých něco teče v protisměru a my můžeme tok v našem směru simulovat odečtením od toku v protisměru.*

Algorithm 1 Ford-Fulkerson algoritmus

Input: Síť (V, E, z, s, c)
Output: Maximální tok f

1: $f \leftarrow$ *nulový tok*
2: **while** existuje zlepšující cesta P z s do t: **do**
3: $\quad\varepsilon \leftarrow \min_{e\in P} r(e)$ ▷ spočítáme rezervu celé cesty
4: $\quad$**for all** $uv \in P$ **do** ▷ zvětšíme tok f
5: $\quad\quad\delta \leftarrow \min\{f(vu), \varepsilon\}$ ▷ kolik můžeme odečíst v protisměru
6: $\quad\quad f(vu) \leftarrow f(vu) - \delta$
7: $\quad\quad f(uv) \leftarrow f(uv) + (\varepsilon - \delta)$ ▷ zbytek přičteme po směru

Analýza ukončení algoritmu:

- *Celočíselné kapacity:* Algoritmus *vždy doběhne.* V každém kroku stoupne velikost toku o $\varepsilon \geq 1$, což může nastat pouze konečně-krát.
- *Racionální kapacity:* přenásobíme-li všechny kapacity jejich společným jmenovatelem, dostaneme síť s celočíselnými kapacitami, na které se bude algoritmus chovat identicky a jak již víme, *skončí.*
- *Iracionální kapacity:* obecně *doběhnout nemusí.* Nemusí zkonvertovat k max flow.

1.3 Základní věty (min-cut/max-flow, integrality)

Lemma 1.1. *Pro každý $E(A, \bar{A})$ řez platí $f^{\Delta}(A, \bar{A}) = |f|$.*

Důkaz.

$$|f| \overset{def}{=} \sum_{v\in\bar{A}} f^{\Delta}(v) = \underbrace{f(A, \bar{A})}_{\leq c(A,\bar{A})} - \underbrace{f(\bar{A}, A)}_{\leq 0} = f^{\Delta}(A, \bar{A}).$$

■

Důsledek 1.1. *Velikost každého toku je menší nebo rovna než kapacita každého řezu: $|f| \leq c(A, \bar{A})$*

Důsledek 1.2. *Pokud $|f| = c(A, \bar{A})$, pak f je maximum a $E(A, \bar{A})$ je minimum.*

Věta 1.1. *Pokud se Fordův-Fulkersonův algoritmus zastaví, pak f je maximální.*

Důkaz. Nechť se algoritmus zastaví. Uvažme množiny vrcholů

$$A := \{v \in V \mid \text{existuje nenasycená cesta ze } z \text{ do } v\} \quad \text{a} \quad B := V \setminus A.$$

Všimneme si, že množina $E(A, B)$ je řez: Zdroj z leží v A, protože ze z do z existuje cesta nulové délky, která je tím pádem nenasycená. Spotřebič musí ležet v B, neboť jinak by existovala nenasycená cesta ze z do s, tudíž by algoritmus ještě neskončil.
Dále víme, že všechny hrany řezu mají nulovou rezervu: kdyby totiž pro nějaké $u \in A$ a $v \in B$ měla hrana uv rezervu nenulovou (nebyla nasycená), spojením nenasycené cesty ze zdroje do u s touto hranou by vznikla nenasycená cesta ze zdroje do v, takže vrchol v by také musel ležet v A, a nikoliv v B.
Proto po všech hranách řezu vedoucích z A do B teče tok rovný kapacitě hran a po hranách z B do A neteče nic. Nalezli jsme tedy řez $E(A, B)$, pro nějž $f^{\Delta}(A, B) = c(A, B)$. To znamená, že tento řez je minimální a tok f maximální. ■

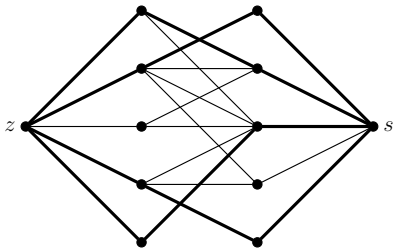
Důsledek 1.3. *Velikost maximálního toku je rovna velikosti minimálního řezu.*

Důsledek 1.4. *Fordův-Fulkersonův algoritmus nám dává celočíselné řešení.*

1.4 Hledání bipartitního párování za pomoci toků

Mějme nějaký bipartitní graf (V, E). Přetvoříme ho na síť (V', E', z, s, c) následovně:

- Nalezneme partity grafu, budeme jim říkat *levá* a *pravá*.
- Všechny hrany zorientujeme zleva doprava.
- Přidáme zdroj z a vedeme z něj hrany do všech vrcholů levé partity.
- Přidáme spotřebič s a vedeme do něj hrany ze všech vrcholů pravé partity.
- Všem hranám nastavíme jednotkovou kapacitu.



Obrázek 2: Ukázka bipartitního párování.

Nyní v této síti najdeme maximální celočíselný tok. Jelikož všechny hrany mají kapacitu 1, musí po každé hraně téci buď 0 nebo 1. Do výsledného párování vložíme právě ty hrany původního grafu, po kterých teče 1.

1.5 Symetrické formulace (průtok)

Definice 1.9. (Průtok) $f^* : E \to \mathbb{R}$ definujeme pro tok f jako: $f^*(uv) = f(uv) - f(vu)$.

Pozorování 1.1. *Průtoky mají následující vlastnosti:*

(1) $f^*(uv) = -f^*(vu)$,

(2) $f^*(uv) \leq c(uv)$,

(3) $f^*(uv) \geq -c(vu)$,

(4) pro všechny vrcholy $v \neq z, s$ *platí* $\sum_{u:uv \in E} f^*(uv) = 0$.

Lemma 1.2. ***(O průtoku):*** *Nechť funkce* $f^* : E \to \mathbb{R}$ *splňuje podmínky (1), (2) a (4). Potom existuje tok* f*, jehož průtokem je* f^*.

Důkaz. Tok f stanovíme pro každou dvojici hran uv a vu zvlášť. Předpokládejme, že $f^*(uv) \geq 0$; v opačném případě využijeme (1) a u prohodíme s v. Nyní stačí položit $f(uv) := f^*(uv)$ a $f(vu) := 0$. Díky vlastnosti (2) funkce f nepřekračuje kapacity, díky (4) pro ni platí Kirchhoffův zákon. ■

Definice 1.10. (Síť rezerv) k toku f v síti $S = (V, E, z, s, c)$ je síť $R(S, f) := (V, E, z, s, r)$, kde $r(e)$ je rezerva hrany e při toku f.

Lemma 1.3. ***(O zlepšování toků):*** *Pro libovolný tok* f *v síti* S *a libovolný tok* g *v síti* $R(S, f)$ *lze v čase* $\mathcal{O}(m)$ *nalézt tok* h *v síti* S *takový, že* $|h| = |f| + |g|$.

Důkaz. Toky přímo sčítat nemůžeme, ale průtoky po jednotlivých hranách už ano. Pro každou hranu e položíme $h^*(e) := f^*(e) + g^*(e)$. Nahlédněme, že funkce h^* má všechny vlastnosti vyžadované lemmatem **P**.

(1) Jelikož první podmínka platí pro f^* i g^*, platí i pro jejich součet.

(2) Víme, že $g^*(uv) \leq r(uv) = c(uv) - f^*(uv)$, takže $h^*(uv) = f^*(uv) + g^*(uv) \leq c(uv)$.

(4) Když se sečtou průtoky, sečtou se i přebytky.

Zbývá dokázat, že se správně sečetly velikosti toků. K tomu si stačí uvědomit, že velikost toku je přebytkem spotřebiče a přebytky se sečetly. ■

Definice 1.11. (Blokující tok), pokud na každé orientované zs-cestě P, $\exists e \in P : f(e) = c(e)$.

Definice 1.12. (Vrstevnatá síť): Síť je *vrstevnatá (pročištěná)*, pokud všechny její vrcholy a hrany leží na nejkratších cestách ze z do s.

1.6 Dinitzův algoritmus

Algoritmus 1.2. ***(Dinitzův algoritmus)*** *začne s nulovým tokem a bude ho vylepšovat pomocí nějakých pomocných toků v síti rezerv, až se dostane k maximálnímu toku. Počet potřebných iterací přitom bude záviset na tom, jak „vydatné“ pomocné toky seženeme – na jednu stranu bychom chtěli, aby byly podobné maximálnímu toku, na druhou stranu jejich výpočtem nechceme trávit příliš mnoho času. Vhodným kompromisem jsou blokující toky:*

Algorithm 2 Dinitzův algoritmus: $\mathcal{O}(n^2m)$

Input: Síť (V, E, z, s, c)
Output: Maximální tok f

$f \leftarrow$ *nulový tok*
repeat
  $R \leftarrow$ síť rezerv, smažeme z f hrany s nulovou rezervou.
  $\ell \leftarrow$ délka nejkratší cesty ze z do s v R ▷ BFS
  if žádná taková cesta neexistuje **then** zastavíme se
  Pročistíme síť R.
  $g \leftarrow$ blokující tok v R
  Zlepšíme tok f pomocí g.
until neexistuje cesta ℓ
return tok f.

Algorithm 3 Blokující Tok: $\mathcal{O}(nm)$

Input: Vrstevnatá síť R s rezervami r
Output: Blokující tok g

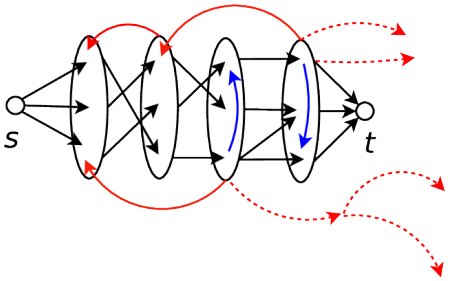
$g \leftarrow$ nulový tok
while existuje v R orientovaná cesta P ze z do s **do**
  $\varepsilon \leftarrow \min_{e \in P}(r(e) - g(e))$
  for all $e \in P$ **do**
    $g(e) \leftarrow g(e) + \varepsilon$
    if $g(e) = r(e)$ **then** smažeme e z R.
  Dočistíme síť pomocí fronty.
return tok g.

Lemma 1.4. ***(O korektnosti:)*** *Pokud se algoritmus zastaví, vydá maximální tok.*

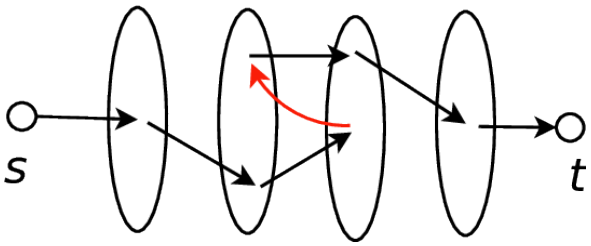
Důkaz. Z lemmatu o zlepšování toků plyne, že f je stále korektní tok. Algoritmus se zastaví tehdy, když už neexistuje cesta ze z do s po hranách s kladnou rezervou. Tehdy by se zastavil i Fordův-Fulkersonův algoritmus a ten, jak už víme, je korektní. ■

Lemma 1.5. *V každém průchodu Dinicova algoritmu vzroste l alespoň o 1.*

Důkaz. Podíváme se na průběh jednoho průchodu vnějším cyklem. Délku aktuálně nejkratší st-cesty označme l. Všechny původní cesty délky l se během průchodu zaručeně nasytí, protože tok f_B je blokující. Musíme však dokázat, že nemohou vzniknout žádné nové cesty délky l nebo menší. V síti rezerv totiž mohou hrany nejen ubývat, ale i přibývat: pokud pošleme tok po hraně, po které ještě nic neteklo, tak v protisměru z dosud nulové rezervy vyrobíme nenulovou. Rozmysleme si tedy, jaké hrany mohou přibývat:
Vnější cyklus začíná s nepročištěnou sítí. Příklad takové sítě je na následujícím obrázku. Po pročištění zůstanou v síti jen černé hrany, tedy hrany vedoucí z i-té vrstvy do $(i+1)$-ní. Červené a modré se zahodí.



Obrázek 3: Nepročištěná síť. Obsahuje zpětné hrany, hrany uvnitř vrstvy a slepé uličky.



Obrázek 4: Cesta užívající novou zpětnou hranu

Nové hrany mohou vznikat výhradně jako opačné k černým hranám (hrany ostatních barev padly za oběť pročištění). Jsou to tedy vždy zpětné hrany vedoucí z i-té vrstvy do $(i-1)$-ní. Vznikem nových hran by proto mohly vzniknout nové st-cesty, které používají zpětné hrany. Jenže st-cesta, která použije zpětnou hranu, musí alespoň jednou skočit o vrstvu zpět a nikdy nemůže skočit o více než jednu vrstvu dopředu, a proto je její délka alespoň $l+2$. Tím je věta dokázána. ■

1.7 Speciální sítě (ubíráme na obecnosti)

Při převodu různých úloh na hledání maximálního toku často dostaneme síť v nějakém speciálním tvaru – třeba s omezenými kapacitami či stupni vrcholů. Podíváme se proto podrobněji na chování Dinicova algoritmu v takových případech a ukážeme, že často pracuje překvapivě efektivně.

1.7.1 Jednotkové kapacity: $c = 1;\ \mathcal{O}(mn)$

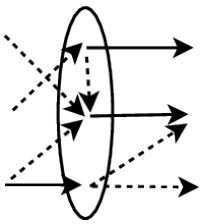
Pokud síť neobsahuje cykly délky 2 (dvojice navzájem opačných hran), všechny rezervy jsou jen 0 nebo 1. Pokud obsahuje, mohou rezervy být i dvojky, a proto síť upravíme tak, že ke každé hraně přidáme hranu opačnou s nulovou kapacitou a rezervu proti směru toku přiřkneme jí. Vzniknou tím sice paralelní hrany, ale to tokovým algoritmům nikterak nevadí.
Při hledání blokujícího toku tedy budou mít všechny hrany na nalezené st-cestě stejnou, totiž jednotkovou, rezervu, takže vždy z grafu odstraníme celou cestu. Když máme m hran, počet zlepšení po cestách délky l bude maximálně m/l. Proto složitost podkroků 9, 10 a 11 bude $m/l \cdot \mathcal{O}(l) = \mathcal{O}(m)$. Tedy pro jednotkové kapacity dostáváme složitost $\emptyset(nm)$.

1.7.2 Jednotkové kapacity znovu a lépe: $c = 1;\ \mathcal{O}(m^{3/2})$

Vnitřní cyklus lépe udělat nepůjde. Je potřeba alespoň lineární čas pro čištění. Můžeme se ale pokusit lépe odhadnout počet iterací vnějšího cyklu.
Sledujme stav sítě po k iteracích vnějšího cyklu a pokusme se odhadnout, kolik iterací ještě algoritmus udělá. Označme l délku nejkratší st-cesty. Víme, že $l > k$, protože v každé iteraci vzroste l alespoň o 1.
Máme tok f_k a chceme dostat maximální tok f. Rozdíl $f - f_k$ je tok v síti rezerv (tok v původní síti to ovšem být nemusí!), označme si ho f_R. Každá iterace velkého cyklu zlepší f_k alespoň o 1. Tedy nám zbývá ještě nejvýše $|f_R|$ iterací. Proto bychom chtěli omezit velikost toku f_R. Například řezem.
Najdeme v síti rezerv nějaký dost malý řez C. Kde ho vzít? Počítejme jen hrany zleva doprava. Těch je jistě nejvýše m a tvoří alespoň k rozhraní mezi vrstvami. Tedy existuje rozhraní vrstev s nejvýše m/k hranami Toto rozhraní je řez. Tedy existuje řez C, pro nějž $|C| \leq m/k$, a algoritmu zbývá maximálně m/k dalších kroků. Celkový počet kroků je nejvýš $k + m/k$, takže stačí zvolit $k = \sqrt{m}$ a získáme odhad na počet kroků $\mathcal{O}(\sqrt{m})$.
Tím jsme dokázali, že celková složitost Dinicova algoritmu pro jednotkové kapacity je $\mathcal{O}(m^{3/2})$. Tím jsme si pomohli pro řídké grafy.

1.7.3 Jednotkové kapacity a jeden ze stupňů roven 1: $c = 1;\ \min(\deg^+, \deg^-) = 1;\ \mathcal{O}(n^{1/2}m)$

Úlohu hledání maximálního párování v bipartitním grafu, případně hledání vrcholově disjunktních cest v obecném grafu lze převést (viz předchozí kapitola) na hledání maximálního toku v síti, v níž má každý vrchol $v \neq s, t$ buďto vstupní nebo výstupní stupeň roven jedné. Pro takovou síť můžeme předchozí odhad ještě trošku upravit. Pokusíme se nalézt v síti po k krocích nějaký malý řez. Místo rozhraní budeme hledat jednu malou vrstvu a z malé vrstvy vytvoříme malý řez tak, že pro každý vrchol z vrstvy vezmeme tu hranu, která je ve svém směru sama.



Obrázek 5

Po k krocích máme alespoň k vrstev, a proto existuje vrstva δ s nejvýše n/k vrcholy. Tedy existuje řez C o velikosti $|C| \leq n/k$ (získáme z vrstvy δ výše popsaným postupem). Algoritmu zbývá do konce $\leq n/k$ iterací vnějšího cyklu, celkem tedy udělá $k + n/k$ iterací. Nyní stačí zvolit $k = \sqrt{n}$ a složitost celého algoritmu vyjde $\mathcal{O}(\sqrt{n} \cdot m)$.

1.7.4 Třetí pokus pro jednotkové kapacity bez omezení na stupně vrcholů v síti: $c = 1;\ \mathcal{O}(n^{2/3}m)$

Hlavní myšlenkou je opět po k krocích najít nějaký malý řez. Najdeme dvě malé sousední vrstvy a všechny hrany mezi nimi budou tvořit námi hledaný malý řez. Budeme tentokrát předpokládat, že naše síť není multigraf, případně že násobnost hran je alespoň omezena konstantou.
Označme s_i počet vrcholů v i-té vrstvě. Součet počtu vrcholů ve dvou sousedních vrstvách označíme $t_i = s_i + s_{i+1}$. Bude tedy platit nerovnost:

$$\sum_i t_i \leq 2 \sum_i s_i \leq 2n.$$

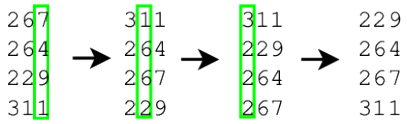
Podle holubníkového principu existuje i takové, že $t_i \leq 2n/k$, čili $s_i + s_{i+1} \leq 2n/k$. Počet hran mezi s_i a s_{i+1} je velikost řezu C, a to je shora omezeno $s_i \cdot s_{i+1}$. Nejhorší případ nastane, když $s_i = s_{i+1} = n/k$, a proto $|C| \leq (n/k)^2$. Proto počet iterací velkého cyklu je $\leq k + (n/k)^2$. Chytře zvolíme $k = n^{2/3}$. Složitost celého algoritmu pak bude $\mathcal{O}(n^{2/3}m)$.

1.7.5 Obecný odhad pro celočíselné kapacity: $c \in \mathbb{N};\ \mathcal{O}(|f|n + nm)$

Tento odhad je založen na velikosti maximálního toku f a předpokladu celočíselných kapacit. Za jednu iteraci velkého cyklu projdeme malým cyklem maximálně tolikrát, o kolik se v něm zvedl tok, protože každá zlepšující cesta ho zvedne alespoň o 1. Zlepšující cesta se tedy hledá maximálně $|f|$-krát za celou dobu běhu algoritmu. Cestu najdeme v čase $\mathcal{O}(n)$. Celkem na hledání cest spotřebujeme $\mathcal{O}(|f| \cdot n)$ za celou dobu běhu algoritmu.
Nesmíme ale zapomenout na čištění. V jedné iteraci velkého cyklu nás stojí čištění $\mathcal{O}(m)$ a velkých iterací je $\leq n$. Proto celková složitost algoritmu činí $\mathcal{O}(|f|n + nm)$

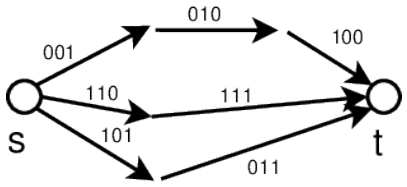
1.7.6 Škálování kapacit

Pokud jsou kapacity hran větší celá čísla omezená nějakou konstantou C, můžeme si pomoci následujícím algoritmem. Jeho základní myšlenka je podobná, jako u třídění čísel postupně po řádech pomocí radix-sortu neboli přihrádkového třídění. Pro jistotu si ho připomeňme. Algoritmus nejprve setřídí čísla podle poslední (nejméně významné) cifry, poté podle předposlední, předpředposlední a tak dále.



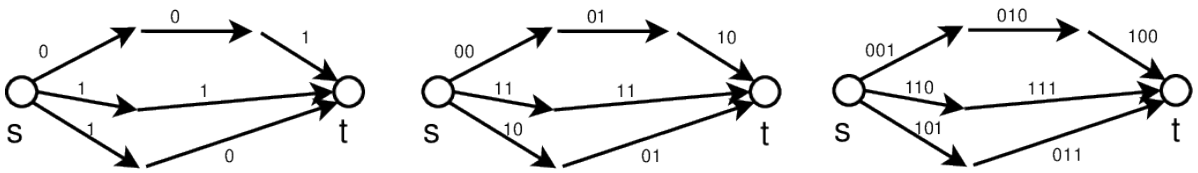
Obrázek 6: Kroky postupného třídění podle řádů.

V našem případě budeme postupně budovat sítě čím dál podobnější zadané síti a v nich počítat toky, až nakonec získáme tok pro ni.
Přesněji: Maximální tok v síti G budeme hledat tak, že hranám postupně budeme zvětšovat kapacity bit po bitu v binárním zápisu až k jejich skutečné kapacitě. Přitom po každém posunu zavoláme Dinicův algoritmus, aby dopočítal maximální tok. Pomocí předchozího odhadu ukážeme, že jeden takový krok nebude příliš drahý.



Obrázek 7: Původní síť, na hranách jsou jejich kapacity v binárním zápisu

Označme k index nejvyššího bitu v zápisu kapacit v zadané síti ($k = \lfloor \log_2 C \rfloor$). Postupně budeme budovat sítě G_i s kapacitami $c_i(e) = \lfloor c(e)/2^{k-i} \rfloor$. G_0 je nejořezanější síť, kde každá hrana má kapacitu rovnou nejvyššímu bitu v binárním zápisu její skutečné kapacity, až G_k je původní síť G.



Obrázek 8: Sítě G_0, G_1 a G_2, jak vyjdou pro síť z předchozího obrázku

Přitom pro kapacity v jednotlivých sítích platí:

$$c_{i+1}(e) = \begin{cases} 2c_i(e), & \text{pokud } (k-i-1)\text{-tý bit je 0,} \\ 2c_i(e)+1, & \text{pokud } (k-i-1)\text{-tý bit je 1.} \end{cases}.$$

Na spočtení maximálního toku f_i v síti G_i zavoláme Dinicův algoritmus, ovšem do začátku nepoužijeme nulový tok, nýbrž tok $2f_{i-1}$. Rozdíl toku z inicializace a výsledného bude malý, totiž:

Lemma 1.6. $|f_i| - |2f_{i-1}| \leq m$.

Důkaz. Vezmeme minimální řez R v G_{i-1}. Podle F-F věty víme, že $|f_{i-1}| = |R|$. Řez R obsahuje $\leq m$ hran, a tedy v G_i má tentýž řez kapacitu maximálně $2|R|+m$. Maximální tok je omezen každým řezem, tedy i řezem R, a proto tok vzroste nejvýše o m. ■

Podle předchozího odhadu pro celočíselné kapacity výpočet toku f_i trvá $\mathcal{O}(mn)$. Takový tok se bude počítat k-krát, pročež celková složitost vyjde $\mathcal{O}(mn \log C)$.

2 Pravděpodobnostní hledání řezů

2.1 Disjunktní cesty

Definice 2.1. ***(Orientovaný st-řez)*** *$C \subseteq E$, t.ž. $G \setminus E$ neobsahuje žádnou orientovanou st-cestu*

Definice 2.2. ***(Řez)*** *je množina hran, která je xy-řezem pro nějakou dvojici vrcholů x, y.*

Definice 2.3. ***(st-separátor)*** *je $W \subseteq V(G)$ taková, že $s, t \notin W$ a v $V(G) \setminus W$ není žádná s,t-cesta.*

Definice 2.4. ***(Separátor)*** *je množina vrcholů, která je xy-separátorem pro nějakou dvojici vrcholů x, y.*

Definice 2.5. ***(Hranová k-souvislost)*** *grafu G, pokud $|V| > k$ a všechny řezy v G mají alespoň k hran.*

Definice 2.6. ***(Vrcholová k-souvislost)*** *pokud $|V| > k$ a všechny separátory v G mají alespoň k vrcholů.*

*Pro nalezení min st-řezu použijeme max tok $\rightsquigarrow$ Dinitzův algoritmus. Pokud se bude jednat o **neorientovaný st-řez**, použijeme stejný algoritmus, jen budeme zdvojnásobovat hrany (2 orientace $\rightleftarrows$) $\rightsquigarrow$ Dinitz v $\mathcal{O}(n^{2/3}m)$.*

Systém hranově disjunktních *st*-cest maximální kardinality Hladový algoritmus v $\mathcal{O}(m)$.

1. Najdeme *st*-cestu, vypíšeme ji a odstraníme (velikost toku -1)
2. Může se stát, že se zacyklíme $\rightsquigarrow$ odstraníme (Kirkhoff stále platí, velikost toku se nezmění)

Hledání min řezu řešíme zvlášť pro hranové a vrcholové k-souvislosti:

- Problém zjištění *stupně hranové souvislosti* G lze převést na *hledání minimálního řezu* $\rightsquigarrow$ Dinitz $\mathcal{O}(n^{2/3}m)$.
  Pokud chceme *minimum ze všech řezů v* G, můžeme zkoušet všechny páry (s,t) $\rightsquigarrow \mathcal{O}(n^2 n^{2/3}m) = \mathcal{O}(n^{8/3}m)$.
  To ale můžeme snadno zrychlit: Zafixujeme $s \in A$ a zkoušíme všechna t. Určitě najdeme alespoň jedno $t \notin A$, takže $\mathcal{O}(n \cdot n^{2/3}m) = \mathcal{O}(n^{5/3}m)$.
- V grafu upravíme vrcholy $\rightsquigarrow$ hrany jednotkové kapacity s $\min(\deg^+, \deg^-) = 1$ $\rightsquigarrow$ Dinitz $\mathcal{O}(n^{1/2}m)$.
  Jako s postupně volíme více vrcholů, než je velikost minimálního separátoru.
  Zkoušíme, dokud počet vrcholů $<$ velikost separátoru (ten prohlásíme za minimální).
  Čas je $\mathcal{O}(\kappa \cdot n \cdot n^{1/2}m) = \mathcal{O}(\kappa(G) \cdot n^{3/2}m)$, kde κ je nalezený stupeň souvislosti.

2.2 Pravděpodobnostní hledání řezů

Algoritmus 2.1. ***(Naivní algoritmus):*** *Náhodně vybírá hrany a kontrahuje je, dokud # vrcholů neklesne na ℓ. Opakujeme, dokud se nedostaneme na velice malý graf, na který aplikujeme algoritmus* MIN-CUT.

Pozorování 2.1. *Řez v $G/e \implies$ řez v G.*

Algorithm 4 CONTRACT(G_0, ℓ):

1: $G \leftarrow G_0$
2: **while** $n > \ell$ **do**
3: Vybereme hranu $e \in E$ rovnoměrně náhodně.
4: $G \leftarrow G/e$ (kontrahujeme hranu e, smyčky odstraňujeme, paralelní hrany ponecháme).
return graf G.

2.3 Náhodné kontrakce a jejich analýza

Zvolíme nyní pevně jeden z minimálních řezů C v zadaném grafu G_0 a označíme k jeho velikost. Pokud algoritmus ani jednou nevybere hranu ležící v tomto řezu, velikost minimálního řezu v grafu G bude rovněž rovna k. Jaká je pravděpodobnost, že se tak stane?
Označme G_i stav grafu G před i-tým průchodem cyklem a n_i a m_i počet jeho vrcholů a hran. Zřejmě $n_i = n-i+1$ (každou kontrakcí přijdeme o jeden vrchol). Navíc každý vrchol má stupeň alespoň k, jelikož jinak by triviální řez okolo tohoto vrcholu byl menší než minimální řez. Proto platí $m_i \geq kn_i/2$. Hranu ležící v řezu C tedy vybereme s pravděpodobností nejvýše $k/m_i \leq k/(kn_i/2) = 2/n_i = 2/(n-i+1)$. Všechny hrany z řezu C proto postoupí do výsledného grafu G s pravděpodobností

$$p \geq \prod_{i=1}^{n-\ell} \left(1 - \frac{2}{n-i+1}\right) = \prod_{i=1}^{n-\ell} \frac{n-i-1}{n-i+1} =$$
$$= \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \cdots \cdot \frac{\ell+1}{\ell+3} \cdot \frac{\ell}{\ell+2} \cdot \frac{\ell-1}{\ell+1} = \frac{\ell \cdot (\ell-1)}{n \cdot (n-1)}.$$

Ještě musíme ošetřit případ, kdy bychom hranu řezu smazali, protože se mezitím stala smyčkou. Ovšem smyčky vznikají pouze z hran paralelních s právě kontrahovanou hranou. Jelikož v libovolném svazku paralelních hran buďto všechny leží v C, nebo ani jedna neleží, museli jsme v takovém případě řez C rozbít už dříve. Odhad pravděpodobnosti to tedy neovlivní.
Můžeme tedy zvolit pevně ℓ, spustit na zadaný graf proceduru CONTRACT a ve vzniklém konstantně velkém grafu pak nalézt minimální řez hrubou silou (to je obzvláště snadné pro $\ell = 2$ – tehdy stačí vzít všechny zbylé hrany). Takový algoritmus nalezne minimální řez s pravděpodobností alespoň c/n^2, kde c je konstanta závislá na ℓ.

Zlepšení: Fixujeme K, výpočet zopakujeme K-krát a použijeme nejmenší z nalezených řezů. Ten už bude minimální s pravděpodobností

$$P_K \geq 1 - (1 - c/n^2)^K \geq 1 - e^{-cK/n^2}.$$

(Druhá nerovnost platí díky tomu, že $e^{-x} \geq 1-x$ pro všechna $x \geq 0$.) Pokud tedy nastavíme počet opakování K na $\Omega(n^2)$, můžeme tím pravděpodobnost chyby stlačit pod libovolnou konstantu, pro $K = \Omega(n^2 \log n)$ pod převrácenou hodnotu libovolného polynomu v n a pro $K = \Omega(n^3)$ už bude dokonce exponenciálně malá.

2.4 Karger-Steinův algoritmus

Všimněme si, že během kontrahování hran pravděpodobnost toho, že vybereme „špatnou“ hranu ležící v minimálním řezu, postupně rostla z počátečních $2/n$ až po obrovské $2/3$ v poslední iteraci (pro $\ell = 2$). Pomůže tedy zastavit kontrahování dříve a přejít na spolehlivější způsob hledání řezu.
Pokud zvolíme $\ell = \lceil n/\sqrt{2}+1 \rceil$, pak řez C přežije kontrahování s pravděpodobností alespoň

$$\frac{\ell \cdot (\ell-1)}{n \cdot (n-1)} \geq \frac{(n/\sqrt{2}+1) \cdot n/\sqrt{2}}{n \cdot (n-1)} = \frac{n/\sqrt{2}+1}{\sqrt{2} \cdot (n-1)} = \frac{n+\sqrt{2}}{2 \cdot (n-1)} \geq \frac{1}{2}.$$

Algoritmus 2.2. *Jako onen spolehlivější způsob hledání řezu následně zavoláme stejný algoritmus rekurzivně, přičemž jak kontrakci, tak rekurzi provedeme dvakrát a z obou nalezených řezů vybereme ten menší, čímž pravděpodobnost chyby snížíme.*

Algorithm 5 KARGERSTEIN(G):

1: **if** $n < 7$ **then** najdeme minimální řez hrubou silou.
2: $\ell \leftarrow \lceil n/\sqrt{2}+1 \rceil$.
3: $C_1 \leftarrow$ KARGERSTEIN(CONTRACT(G, ℓ)).
4: $C_2 \leftarrow$ KARGERSTEIN(CONTRACT(G, ℓ)).
5: **return** $\min(C_1, C_2)$.

Časová složitost za pomoci stromu rekurze: *Hloubka rekurze: v každém kroku se velikost vstupu zmenší přibližně $\sqrt{2}$-krát, takže strom rekurze bude mít hloubku $\mathcal{O}(\log n)$.*
Na i-té hladině zpracováváme 2^i podproblémů velikosti $n/2^{i/2}$.
Při výpočtu každého podproblému voláme dvakrát Contract*, která spotřebuje čas $\mathcal{O}((n/2^{i/2})^2) = \mathcal{O}(n^2/2^i)$.*
Součet přes celou hladinu je $\mathcal{O}(n^2)$ a přes všechny hladiny $\mathcal{O}(n^2 \log n)$.

Zbývá spočítat, s jakou pravděpodobností algoritmus skutečně nalezne minimální řez.
Označme p_i pravděpodobnost, že algoritmus na i-té hladině stromu rekurze (počítáno od nejhlubší, nulté hladiny) vydá správný výsledek příslušného podproblému. Jistě je $p_0 = 1$ a platí rekurence $p_i \geq 1 - (1 - 1/2 \cdot p_{i-1})^2$. Uvažujme posloupnost g_i, pro kterou jsou tyto nerovnosti splněny jako rovnosti, a všimněme si, že $p_i \geq g_i$. Víme tedy, že $g_0 = 1$ a $g_i = 1 - (1 - 1/2 \cdot g_{i-1})^2 = g_{i-1} - g_{i-1}^2/4$.
Nyní zavedeme substituci $z_i = 4/g_i - 1$, čili $g_i = 4/(z_i + 1)$, a tak získáme novou rekurenci pro z_i:

$$\frac{4}{z_i + 1} = \frac{4}{z_{i-1} + 1} - \frac{4}{(z_{i-1} + 1)^2},$$

kterou už můžeme snadno upravovat:

$$\begin{aligned} \frac{1}{z_i + 1} &= \frac{z_{i-1}}{(z_{i-1} + 1)^2}, \\ z_i + 1 &= \frac{z_{i-1}^2 + 2z_{i-1} + 1}{z_{i-1}}, \\ z_i + 1 &= z_{i-1} + 2 + \frac{1}{z_{i-1}}. \end{aligned}$$

Jelikož $z_0 = 3$, a tím pádem $z_i \geq 3$ pro všechna i, získáme z poslední rovnosti vztah $z_i \leq z_{i-1} + 2$, a tudíž $z_i \leq 2i + 3$. Zpětnou substitucí obdržíme $g_i \geq 4/(2i + 4)$, tedy $p_i \geq g_i = \Omega(1/i)$.
Nyní si stačí vzpomenout, že hloubka rekurze činí $\mathcal{O}(\log n)$, a ihned získáme odhad pro pravděpodobnost správného výsledku $\Omega(1/\log n)$. Náš algoritmus tedy stačí ziterovat $\mathcal{O}(\log^2 n)$-krát, abychom pravděpodobnost chyby stlačili pod převrácenou hodnotu polynomu. Dokázali jsme následující větu:

Věta 2.1. *Iterováním algoritmu* MinCut *nalezneme minimální řez v neohodnoceném neorientovaném grafu v čase $\mathcal{O}(n^2 \log^3 n)$ s pravděpodobností chyby $\mathcal{O}(1/n^c)$ pro libovolnou konstantu $c > 0$.*

3 Hledání nejkratších cest

3.1 Obecné vlastnosti

Obvykle se studují následující tři problémy:

- **1-1** neboli **P2PSP** (Point to Point Shortest Path) – chceme nalézt nejkratší cestu z daného vrcholu u do daného vrcholu v. (Pokud je nejkratších cest více, tak libovolnou z nich.)
- **1-n** neboli **SSSP** (Single Source Shortest Paths) – pro daný vrchol u chceme nalézt nejkratší cesty do všech ostatních vrcholů.
- **n-n** neboli **APSP** (All Pairs Shortest Paths) – zajímají nás nejkratší cesty mezi všemi dvojicemi vrcholů.

Definice 3.1. (Vzdálenost) $d(u, v) :=$ délka nejkratší cesty.

3.2 Strasti se zápornými cykly

Pro grafy s hranami záporných délek bez jakýchkoliv omezení je hledání nejkratší cesty NP-těžké Všechny známé polynomiální algoritmy totiž místo nejkratší cesty hledají nejkratší sled (nijak nekontrolují, zda cesta neprojde jedním vrcholem vícekrát).
Může nastat, že neplatí trojúhelníková nerovnost.

3.3 Prefixová vlastnost

Navíc se nám bude hodit, že každý prefix nejkratší cesty je opět nejkratší cesta. Jinými slovy pokud některá z nejkratších cest z u do v vede přes nějaký vrchol w, pak její část z u do w je jednou z nejkratších cest z u do w. (V opačném případě bychom mohli úsek $u \ldots w$ vyměnit za kratší.)
Díky této **prefixové vlastnosti** můžeme pro každý vrchol u sestrojit jeho *strom nejkratších cest* $\mathcal{T}(u)$.

3.4 Stromy nejkratších cest

Definice 3.2. (Strom nejkratších cest) v (G, e) se zdrojem v u je $\mathcal{T} \subseteq \mathcal{G}$, strom na vrcholech dosažitelných z u, orientovaný z u ven. Pro každý vrchol v dosažitelný z u je uv-cesta v $\mathcal{T}$ nejkratší uv-cesta v G.

Pozorování 3.1. *Strom nejkratších cest vždy existuje.*

Důkaz. Nechť $u = v_1, \ldots, v_n$ jsou všechny vrcholy grafu G. Indukcí budeme dokazovat, že pro každé i existuje strom $\mathcal{T}_i$, v němž se nacházejí nejkratší cesty z vrcholu u do vrcholů $v_1, \ldots, v_i$. Pro $i = 1$ stačí uvážit strom obsahující jediný vrchol u. Ze stromu $\mathcal{T}_{i-1}$ pak vyrobíme strom $\mathcal{T}_i$ takto: Nalezneme v G nejkratší cestu z u do v_i a označíme z poslední vrchol na této cestě, který se ještě vyskytuje v $\mathcal{T}_{i-1}$. Úsek nejkratší cesty od z do v_i pak přidáme do $\mathcal{T}_{i-1}$ a díky prefixové vlastnosti bude i cesta z u do v_i v novém stromu nejkratší. ■

3.5 Relaxační schéma

Vhodnou operací pro vylepšování ohodnocení je takzvaná *relaxace*. Vybereme si nějaký vrchol v a pro všechny jeho sousedy w spočítáme $h(v) + \ell(v, w)$, tedy délku sledu, který vznikne rozšířením aktuálního sledu do v o hranu (v, w). Pokud je tato hodnota menší než $h(w)$, tak jí $h(w)$ přepíšeme.
Abychom zabránili opakovaným relaxacím téhož vrcholu, které nic nezmění, budeme rozlišovat tři stavy vrcholů:

- *neviděn* ... ještě jsme ho nenavštívili,
- *otevřen* ... změnilo se ohodnocení, časem chceme relaxovat,
- *uzavřen* ... už jsme relaxovali a není potřeba znovu.

,

Algoritmus 3.1. *Náš algoritmus bude fungovat následovně:*

Algorithm 6 RELAXACE:

```
1: h(∗) ← ∞, h(u) ← 0.
2: stav(∗) ← neviděn, stav(u) ← otevřen.
3: while existují otevřené vrcholy do
4:     v ← libovolný otevřený vrchol.
5:     stav(v) ← uzavřen.
6:     for all hrany vw ∈ E do                          ▷ Relaxujeme v
7:         if h(w) > h(v) + ℓ(v, w) then
8:             h(w) ← h(v) + ℓ(v, w).
9:             stav(w) ← otevřen.
   return d(u, v) = h(v) pro všechna v.
```

Věta 3.1. *Spustíme-li meta-algoritmus na graf bez záporných cyklů, pak:*

1. *Ohodnocení $h(v)$ vždy odpovídá délce nějakého sledu z u do v.*
2. *$h(v)$ dokonce odpovídá délce nějaké cesty z u do v.*
3. *Algoritmus se vždy zastaví.*
4. *Po zastavení jsou označeny jako uzavřené právě ty vrcholy, které jsou dosažitelné z u.*
5. *Po zastavení mají konečné $h(v)$ právě všechny uzavřené vrcholy.*
6. *Pro každý dosažitelný vrchol je na konci $h(v)$ rovno $d(u, v)$.*

Důkaz. Dokážeme jednotlivě:

1. Dokážeme indukcí podle počtu kroků algoritmu.
2. Stačí rozmyslet, v jaké situaci by vytvořený sled mohl obsahovat cyklus.
3. Cest, a tím pádem i možných hodnot $h(v)$ pro každý v, je konečně mnoho.
4. Implikace $\Rightarrow$ je triviální, pro $\Leftarrow$ stačí uvážit neuzavřený vrchol, který je dosažitelný z u cestou o co nejmenším počtu hran.
5. $h(v)$ nastavujeme na konečnou hodnotu právě v okamžicích, kdy se vrchol stává otevřeným. Každý otevřený vrchol je časem uzavřen.

6. Kdyby tomu tak nebylo, vyberme si ze „špatných“ vrcholů v takový, pro nějž obsahuje nejkratší cesta z u do v nejmenší možný počet hran. Vrchol v je zajisté různý od u, takže má na této cestě nějakého předchůdce w. Přitom w už musí být ohodnocen správně a relaxace, která mu toto ohodnocení nastavila, ho musela prohlásit za otevřený. Jenže každý otevřený vrchol je později uzavřen, takže w poté musel být ještě alespoň jednou relaxován, což muselo snížit $h(v)$ na správnou vzdálenost.

■

3.6 Bellman-Ford-Moore algoritmus

Otevřené vrcholy udržujeme ve frontě (relaxujeme vrchol na počátku fronty, nově otevírané zařazujeme na konec).
- Fáze i uzavře všechny vrcholy fáze $i-1$.

Pozorování 3.2. *Jedna fáze algoritmu je v čase $O(m)$. Celkem v $O(nm)$.*

Lemma 3.1. *Algoritmus se zastaví po n fázích.*

Důkaz. Maximální délka nejdelšího sledu je $n-1 \implies$ na konci $n-1$ fáze se všechny $h(v)$ nezvětší $\implies +1$ fáze na uzavření otevřených vrcholů. ■

Pozorování 3.3. *Pokud běží déle než n fází, obsahuje záporné hrany.*

Lemma 3.2. *Na konci i-té fáze platí $\forall v : h(v) \leq$ délka nejkratšího uv-sledu s maximálně i hranami.*

Důkaz. Indukcí podle i.

- Pro $i = 0$ to triviálně platí.
- Pro $i \mapsto i+1$: Uvažujme nyní vrchol v na konci $(i+1)$-ní fáze a nějaký nejkratší uv sled P o $i+1$ hranách. Označme wv poslední hranu tohoto sledu a P' sled bez této hrany, který tedy má délku i.

  Podle indukčního předpokladu je na konci i-té fáze $h(w) \leq \ell(P')$. Tuto hodnotu získalo $h(w)$ nejpozději v i-té fázi, při tom jsme vrchol w otevřeli, takže jsme ho nejpozději v $(i+1)$-ní fázi zavřeli a relaxovali. Po této relaxaci je ovšem $h(v) \leq h(w) + \ell(w,v) \leq \ell(P') + \ell(w,v) = \ell(P)$.

■

3.7 Dijkstrův algoritmus

Pokud jsou všechny délky hran nezáporné, můžeme použít efektivnější pravidlo pro výběr vrcholu navržené Dijkstrou. To říká, že vždy relaxujeme ten z otevřených vrcholů, jehož ohodnocení je nejmenší.

Věta 3.2. *Dijkstrův algoritmus uzavírá vrcholy v pořadí podle neklesající vzdálenosti od u a každý dosažitelný vrchol uzavře právě jednou.*

Důkaz. Indukcí dokážeme, že v každém okamžiku mají všechny uzavřené vrcholy ohodnocení menší nebo rovné ohodnocením všech otevřených vrcholů. Na počátku to jistě platí. Nechť nyní uzavíráme vrchol v s minimálním $h(v)$ mezi otevřenými. Během jeho relaxace nemůžeme žádnou hodnotu snížit pod $h(v)$, jelikož v grafu s nezápornými hranami je $h(v) + \ell(v,w) \geq h(v)$. Hodnota zbývajících otevřených vrcholů tedy neklesne pod hodnotu tohoto nově uzavřeného. Hodnoty dříve uzavřených vrcholů se nemohou nijak změnit. ■

Důsledek 3.1. *Počet relací $\leq n$.*

Časová složitost $\mathcal{O}(nT_I + nT_X + mT_D)$ *(Insert, ExtractMin, Decrease)*

3.7.1 Haldy

Datová struktura pro otevření vrcholů s $h(\_)$ s operacemi Insert, ExtractMin, Decrease.
V relaxačních algoritmech běží Insert, ExtractMin v $\leq n$ operacích a Decrease v $\leq m$.

3.8 Datové struktury pro Dijkstrův algoritmus

Všechny délky hran jsou nezáporná celá čísla omezená nějakou konstantou L. Všechny vzdálenosti jsou tedy omezeny číslem nL, takže nám stačí datová struktura schopná uchovávat takto omezená celá čísla.

Datová struktura	T_I	T_X	T_D	Celkem
Pole	1	n	1	n^2
Binání halda	$\log n$	$\log n$	$\log n$	$m \log n$
d-ární halda	$\log_d n$	$d \log_d n$	$\log_d n$	$m \frac{\log n}{\log m/k}$
Fibonacciho halda	1	$\log n$	1	$m + n \log n$
Pole přihrádek	1	nL	1	$m + nL$
Strom nad přihrádkami	$\log L$	$\log L$	$\log L$	$m \log L$
Multiple přihrádky	$\frac{\log L}{\log \log L}$	1	1	$m + n \frac{\log n}{\log \log L}$

3.8.1 Pole přihrádek

Pole indexované hodnotami $0 \dots nL$, kde i-tý prvek obsahuje seznam vrcholů, jejichž ohodnocení $= i$.

- Insert, Decrease v $\mathcal{O}(1)$, budeme-li si u každého prvku pamatovat jeho polohu v seznamu.
- ExtractMin potřebuje najít první neprázdnou přihrádku, ale jelikož víme, že posloupnost odebíraných minim je monotónní, stačí hledat od místa, kde se hledání zastavilo minule.

Všechna hledání přihrádek tedy zaberou dohromady $\mathcal{O}(nL)$ a celý Dijkstrův algoritmus bude trvat $\mathcal{O}(m + nL)$. Prostor je stejný, což není moc dobré. Můžeme zlepšit.
Všechny neprázdné přihrádky se nacházejí v úseku pole dlouhém $L+1$, takže stačí indexovat $\pmod{L+1}$. Pouze si musíme dávat pozor, abychom správně poznali, kdy se struktura vyprázdnila, což zjistíme například pomocí počítadla otevřených vrcholů. Prostor klesne na $\mathcal{O}(L + m)$.

3.8.2 Strom nad přihrádkami

- Insert, ExtractMin, Decrease v $\mathcal{O}(\log L)$,

Celková složitost Dijkstrova algoritmu vyjde , přičemž čas L spotřebujeme na inicializaci struktury (té se lze za jistých podmínek vyhnout, viz zmíněná kapitola).

3.8.3 Multi-level přihrádky

Podobně jako u třídění čísel, i zde se vyplácí stavět přihrádkové struktury víceúrovňově. Jednotlivé hodnoty budeme zapisovat v soustavě o základu B, který zvolíme jako nějakou mocninu dvojky, abychom mohli s číslicemi tohoto zápisu snadno zacházet pomocí bitových operací. Každé číslo tedy zabere nejvýše $d = 1 + \lfloor \log_B L \rfloor$ číslic; pokud bude kratší, doplníme ho zleva nulami.
Nejvyšší patro struktury bude tvořeno polem B přihrádek, v i-té z nich budou uložena ta čísla, jejichž číslice nejvyššího řádu je rovna i. Za *aktivní* prohlásíme tu přihrádku, která obsahuje aktuální minimum. Přihrádky s menšími indexy jsou prázdné a zůstanou takové až do konce výpočtu, protože halda je monotónní. Pokud v přihrádce obsahující minimum bude více prvků, budeme ji rozkládat podle druhého nejvyššího řádu na dalších B přihrádek atd. Celkem tak vznikne až d úrovní.
Struktura bude obsahovat následující data:

- Parametry L, B a d.
- Pole úrovní (nejvyšší má číslo $d-1$, nejnižší 0), každá úroveň je buďto prázdná (a pak jsou prázdné i všechny nižší), nebo obsahuje pole U_i o B přihrádkách a v každé z nich seznam prvků. Pole úrovní používáme jako zásobník, udržujeme si číslo nejnižší neprázdné úrovně.
- Hodnotu μ předchozího odebraného minima.

Operace Insert vloží hodnotu do nejhlubší možné přihrádky. Podívá se tedy na nejvyšší úroveň: pokud hodnota patří do přihrádky, která není aktivní, vloží ji přímo. Jinak přejde o úroveň níže a zopakuje stejný postup. To vše lze provést v konstantním čase: stačí se podívat, jaký je nejvyšší jedničkový bit ve xoru nové hodnoty s číslem μ (opět viz kapitola o výpočetních modelech), a tím zjistit číslo úrovně, kam hodnota patří.
Pokud chceme provést Decrease, odstraníme hodnotu z přihrádky, ve které se právě nachází (polohu si můžeme u každé hodnoty pamatovat zvlášť), a znovu ji vložíme.
Většinu práce samozřejmě přenecháme funkci ExtractMin.Ta začne prohledávat nejnižší obsazenou úroveň od aktivní přihrádky dál (to, která přihrádka je na které úrovni aktivní, poznáme z číslic hodnoty μ). Pokud přihrádky na této úrovni dojdou, prázdnou úroveň zrušíme a pokračujeme o patro výše.

Jakmile najdeme neprázdnou přihrádku, nalezneme v ní minimum a to se stane novým μ. Pokud v přihrádce nebyly žádné další prvky, skončíme. V opačném případě zbývající prvky rozprostřeme do přihrádek na bezprostředně nižší úrovni, kterou tím založíme.
Čas strávený hledáním minima můžeme rozdělit na několik částí:

- $\mathcal{O}(B)$ na inicializaci nové úrovně – to naúčtujeme prvku, který jsme právě mazali;
- hledání neprázdných přihrádek – prozkoumání každé prázdné přihrádky naúčtujeme jejímu vytvoření, což se rozpustí v $\mathcal{O}(B)$ na inicializaci úrovně;
- zrušení úrovně – opět naúčtujeme jejímu vzniku;
- rozhazování prvků do přihrádek – jelikož prvky v hierarchii přihrádek putují během operací pouze doleva a dolů (jejich hodnoty se nikdy nezvětšují), klesne každý prvek nejvýše d-krát, takže stačí, když na všechna rozhazování přispěje časem $\mathcal{O}(d)$;
- hledání minima – minimum naúčtujeme smazanému prvku, ostatní prvky, které jsme museli projít, naúčtujeme jejich rozhazování.

Stačí tedy, aby každý prvek při Insert u zaplatil čas $\mathcal{O}(B+d)$ a jak Decrease, tak ExtractMin budou mít konstantní amortizovanou složitost. Dijkstrův algoritmus pak poběží v $\mathcal{O}(m+n(B+d))$.
Zbývá nastavit parametry tak, abychom minimalizovali výraz $B+d = B+\log L/\log B$. Vhodná volba je $B = \log L/\log\log L$. Při ní platí

$$\frac{\log L}{\log B} = \frac{\log L}{\log\left(\log L/\log\log L\right)} = \frac{\log L}{\log\log L - \log\log\log L} = \Theta(B).$$

Tehdy Dijkstra vydá výsledek v čase $\mathcal{O}(m + n\cdot\frac{\log L}{\log\log L})$.

3.8.4 Dinitzův trik pro hrany reálné délky

4 Potenciály

4.1 Potenciály a eliminace záporných hran.

Definice 4.1. (Potenciál) budeme říkat libovolné funkci $\psi : V \to \mathbb{R}$. Pro každý potenciál zavedeme *redukované délky hran* $\ell_\psi(u,v) := \ell(u,v) + \psi(u) - \psi(v)$. Potenciál nazveme *přípustný*, pokud žádná hrana nemá zápornou redukovanou délku.

Pozorování 4.1. *Pro redukovanou délku libovolné cesty P z u do v platí: $\ell_\psi(P) = \ell(P) + \psi(u) - \psi(v)$.*

Důkaz. Nechť cesta P prochází přes vrcholy $u = w_1, \ldots, w_k = v$. Potom:

$$\ell_\psi(P) = \sum_i \ell_\psi(w_i, w_{i+1}) = \sum_i \left(\ell(w_i, w_{i+1}) + \psi(w_i) - \psi(w_{i+1})\right).$$

Tato suma je ovšem teleskopická, takže z ní zbude

$$\sum_i \ell(w_i, w_{i+1}) + \psi(w_1) - \psi(w_k) = \ell(P) + \psi(u) - \psi(v).$$

■

Důsledek 4.1. *Potenciálovou redukcí se délky všech cest mezi u a v změní o tutéž konstantu, takže struktura nejkratších cest zůstane nezměněna.*

4.2 Heuristické 1-1 nejkratší cesty a obousměrný Dijkstra

Můžeme spustit prohledávání z obou konců zároveň, tedy zkombinovat hledání od s v původním grafu s hledáním od t v grafu s obrácenou orientací hran.
Zastavíme se v okamžiku, kdy jsme jeden vrchol uzavřeli v obou směrech. Pozor ovšem na to, že součet obou ohodnocení tohoto vrcholu nemusí být roven $d(v,u)$.
Nejkratší cesta ještě může vypadat tak, že přechází po nějaké hraně mezi vrcholem uzavřeným v jednom směru a vrcholem uzavřeným ve směru druhém (ponechme bez důkazu). Stačí tedy během relaxace zjistit, zda je konec hrany uzavřený v opačném směru prohledávání, a pokud ano, započítat cestu do průběžného minima.
Obousměrný Dijkstrův algoritmus projde sjednocení nějaké koule okolo s s nějakou koulí okolo t, které obsahuje nejkratší cestu. Průměry koulí přitom závisí na tom, jak budeme během výpočtu střídat oba směry prohledávání. V nejhorším případě samozřejmě můžeme prohledat celý graf.

4.3 A* algoritmus

Jedná se o modifikaci Dijkstrova algoritmu, která využívá heuristickou funkci pro dolní odhad vzdálenosti do cíle; označme si ji $\psi(v)$. V každém kroku pak uzavírá vrchol v s nejmenším možným součtem $h(v) + \psi(v)$ aktuálního ohodnocení s heuristikou.
Intuice za tímto algoritmem je jasná: pokud víme, že nějaký vrchol je blízko od počátačního vrcholu s, ale bude z něj určitě daleko do cíle t, zatím ho odložíme a budeme zkoumat nadějnější varianty.
Heuristika se přitom volí podle konkrétního problému – např. hledáme-li cestu v mapě, můžeme použít vzdálenost do cíle vzdušnou čarou.
Je u tohoto algoritmu zaručeno, že vždy najde nejkratší cestu? Na to nám dá odpověď teorie potenciálů:

Věta 4.1. *Běh algoritmu A* odpovídá průběhu Dijkstrova algoritmu na grafu redukovaném potenciálem $-\psi$. Přesněji, pokud označíme h^* aktuální ohodnocení v A* a h aktuální ohodnocení v synchronně běžícím Dijkstrovi, bude vždy platit $h(v) = h^*(v) - \psi(s) + \psi(v)$.*

Důkaz. Indukcí podle doby běhu obou algoritmů. Na počátku je $h^*(s)$ i $h(s)$ nulové a všechna ostatní h^* a h jsou nekonečná, takže tvrzení platí. V každém dalším kroku A* vybere vrchol v s nejmenším $h^*(v)+\psi(v)$, což je tentýž vrchol, který vybere Dijkstra ($\psi(s)$ je stále stejné).
Uvažujme, co se stane během relaxace hrany vw: Dijkstra se pokusí snížit ohodnocení $h(w)$ o $\delta = h(w) - h(v) - \ell_{-\psi}(v,w)$ a provede to, pokud $\delta > 0$. Ukážeme, že A* udělá totéž:

$$\begin{aligned}\delta &= (h^*(w) - \psi(s) + \psi(w)) - (h^*(v) - \psi(s) + \psi(v)) - (\ell(v,w) - \psi(v) + \psi(w)) \\ &= h^*(w) - \psi(s) + \psi(w) - h^*(v) + \psi(s) - \psi(v) - \ell(v,w) + \psi(v) - \psi(w) \\ &= h^*(w) - h^*(v) - \ell(v,w).\end{aligned}$$

Oba algoritmy tedy až na posunutí dané potenciálem počítají totéž. ■

Důsledek 4.2. *Algoritmus A* funguje jen tehdy, je-li potenciál $-\psi$ přípustný.*

Příklad: Pro rovinnou mapu to heuristika daná euklidovskou vzdáleností ϱ, tedy $\psi(v) := \varrho(v,t)$, splňuje: Přípustnost požaduje pro každou hranu uv nerovnost $\ell(u,v) - \psi(v) + \psi(u) \geq 0$, tedy $\ell(u,v) - \varrho(v,t) + \varrho(u,t) \geq 0$. Jelikož $\ell(u,v) \geq \varrho(u,v)$, stačí dokázat, že $\varrho(u,v) - \varrho(v,t) + \varrho(u,t) \geq 0$, což je ovšem trojúhelníková nerovnost pro metriku ϱ.

5 APSP algoritmy a transitivní uzávěr

Dosažitelnost (transitivní uzávěr): matice matice sousedů A vyprodukuje matici A^* vzdáleností (délek stran).
Můžeme vyřešit spuštěním $n\times$BFS, což nám dá složitost $\Theta(nm)$.
Počítání vzdáleností. $L \to D$, kde L je matice aktuálních délek (obsahuje $L_{ij} = l(i,j)$ pokud $ij \in E$, jinak $+\infty$) a D je matice vzdáleností. Můžeme vyřešit spuštěním $n\times$Dijkstra (s Fib. haldou), což nám dá složitost $\Theta(n^2 \log n + nm)$. Pokud je graf hustý, tak je obojí n^3. Definujeme si tedy lepší algoritmus.

5.1 Floyd-Warshall algoritmus a jeho generalizace

Funguje pro libovolný orientovaný graf bez záporných cyklů.
Označme D^k_{ij} délku nejkratší cesty z vrcholu i do vrcholu j přes vrcholy 1 až k (tím myslíme, že všechny vnitřní vrcholy cesty leží v množině $\{1,\dots,k\}$). Jako obvykle položíme $D^k_{ij} = +\infty$, pokud žádná taková cesta neexistuje. Pak platí:

$$\begin{aligned}D^0_{ij} &= \text{délka hrany } ij, \\ D^n_{ij} &= \text{hledaná vzdálenost z } i \text{ do } j, \\ D^k_{ij} &= \min(D^{k-1}_{ij}, D^{k-1}_{ik} + D^{k-1}_{kj}).\end{aligned}$$

První dvě rovnosti plynou přímo z definice. Třetí rovnost dostaneme rozdělením cest z i do j přes 1 až k na ty, které se vrcholu k vyhnou (a jsou tedy cestami přes 1 až $k-1$), a ty, které ho použijí – každou takovou můžeme složit z cesty z i do k a cesty z k do j, obojí přes 1 až $k-1$.
Zbývá vyřešit jednu maličkost: složením cesty z i do k s cestou z k do j nemusí nutně vzniknout cesta, protože se nějaký vrchol může opakovat. V grafech bez záporných cyklů nicméně takový sled nemůže být kratší než nejkratší cesta, takže tím falešné řešení nevyrobíme. (Přesněji: ze sledu $i\alpha v\beta k\gamma v\delta j$, kde $v \notin \beta,\gamma$, můžeme vypuštěním části $v\beta k\gamma v$ nezáporné délky získat sled z i do j přes 1 až $k-1$, jehož délka nemůže být menší než D^{k-1}_{ij}.)
Samotný algoritmus postupně počítá matice $D^0, D^1, \dots, D^n$ podle uvedeného předpisu:

Algoritmus 5.1. ***(Floyd-Warshall):*** *Náhodně vybírá hrany a kontrahuje je, dokud # vrcholů neklesne na ℓ.*

Algorithm 7 FloydWarshall(G_0, ℓ):

1: $D^0 \leftarrow$matice délek hran.
2: **for** $k = 1, \ldots, n$ **do**
3: **for** $i, j = 1, \ldots, n$ **do**
4: $D^k_{ij} \leftarrow \min(D^{k-1}_{ij}, D^{k-1}_{ik} + D^{k-1}_{kj})$.
5: Matice vzdáleností $\leftarrow D^n$.

Časová složitost tohoto algoritmu činí $\Theta(n^3)$. Kubickou prostorovou složitost můžeme snadno snížit na kvadratickou: Buď si uvědomíme, že v každém okamžiku potřebujeme jen aktuální matici D^k a předchozí D^{k-1}. Anebo nahlédneme, že můžeme D^{k-1} na D^k přepisovat na místě. U hodnot D_{ik} a D_{kj} je totiž podle definice stará i nová hodnota stejná. Algoritmu tedy stačí jediné pole velikosti $n \times n$, které na počátku výpočtu obsahuje vstup a na konci výstup.

5.2 Násobení matic

Definice 5.1. $(\oplus, \otimes)$**-součin** matic $A, B \in X^{n \times n}$, kde $\oplus$ a $\otimes$ jsou dvě asociativní binární operace na množině X, je matice C taková, že

$$C_{ij} = \bigoplus_{k=1}^{n} A_{ik} \otimes B_{kj}.$$

Klasické násobení matic je tedy $(+, \cdot)$-součin.

5.2.1 Algebraický pohled na násobení matic

5.2.2 Divide and conquer algoritmus

Na vstupu dostaneme matici sousednosti A, výstupem má být její transitivní uzávěr A^* (matice dosažitelnosti). Všechny součiny matic v tomto oddílu budou typu $(\vee, \wedge)$.
Vrcholy grafu rozdělíme na dvě množiny X a Y přibližně stejné velikosti, bez újmy na obecnosti tak, aby matice A měla následující blokový tvar:

$$A = \begin{pmatrix} P & Q \\ R & S \end{pmatrix},$$

kde podmatice P popisuje hrany z X do X, podmatice Q hrany z X do Y, atd.

Věta 5.1. *Pokud matici A^* zapíšeme rovněž v blokovém tvaru:*

$$A^* = \begin{pmatrix} I & J \\ K & L \end{pmatrix},$$

bude platit:

$$\begin{aligned} I &= (P \vee QS^*R)^*, \\ J &= IQS^*, \\ K &= S^*RI, \\ L &= S^* \vee S^*RIQS^*. \end{aligned}$$

Důkaz. Jednotlivé rovnosti můžeme číst takto:

I: Sled z X do X vznikne opakováním částí, z nichž každá je buďto hrana uvnitř X nebo přechod po hraně z X do Y následovaný sledem uvnitř Y a přechodem zpět do X.

J: Sled z X do Y můžeme rozdělit v místě, kdy naposledy přechází po hraně z X do Y. První část přitom bude sled z X do X, druhá sled uvnitř Y.

K: Se sledem z Y do X naložíme symetricky.

L: Sled z Y do Y vede buďto celý uvnitř Y, nebo ho můžeme rozdělit na prvním přechodu z Y do X a posledním přechodu z X do Y. Část před prvním přechodem povede celá uvnitř Y, část mezi přechody bude tvořit sled z X do X a konečně část za posledním přechodem zůstane opět uvnitř Y.

■

5.2.3 Seidelův algoritmus

Pro G neorientovaný jednotkové délky můžeme dosáhnout ještě lepších výsledků.
Matici vzdáleností lze spočítat v čase $\mathcal{O}(n^\omega \log n)$ Seidelovým algoritmem.

Definice 5.2. (Druhá mocnina grafu G) je graf G^2 na téže množině vrcholů, v němž jsou vrcholy i a j spojeny hranou právě tehdy, existuje-li v G sled délky nejvýše 2 vedoucí z i do j.

Pozorování 5.1. *Matici sousednosti grafu G^2 získáme z matice sousednosti grafu G jedním $(\vee, \wedge)$-součinem, tedy v čase $\mathcal{O}(n^\omega)$.*

Algoritmus 5.2. (Seidlův) Rekurzivně: Sestrojíme graf G^2, rekurzí spočítáme jeho matici vzdáleností D' a z ní pak rekonstruujeme matici vzdáleností D zadaného grafu. Rekurze končí, pokud $G^2 = G$ – tehdy je každá komponenta souvislosti zahuštěna na úplný graf, takže matice vzdáleností je rovna matici sousednosti.

Zbývá ukázat, jak z matice D' spočítat matici D. Zvolme pevně i a zaměřme se na funkce $d(v) = D_{iv}$ a $d'(v) = D'_{iv}$. Jistě platí $d'(v) = \lceil d(v)/2 \rceil$, pročež $d(v)$ je buď rovno $2d'(v)$ nebo o 1 nižší. Naučíme se rozpoznat, jestli $d(v)$ má být sudé nebo liché, a z toho vždy poznáme, jestli je potřeba jedničku odečíst.

Jak vypadá funkce d na sousedech vrcholu $v \neq i$? Pro alespoň jednoho souseda u je $d(u) = d(v) - 1$ (to platí pro sousedy, kteří leží na některé z nejkratších cest z v do i). Pro všechny ostatní sousedy je $d(u) = d(v)$ nebo $d(u) = d(v) + 1$.
Pokud je $d(v)$ sudé, vyjde pro sousedy ležící na nejkratších cestách $d'(u) = d'(v)$ a pro ostatní sousedy $d'(u) \geq d'(v)$, takže průměr z $d'(u)$ přes sousedy je alespoň $d'(v)$. Je-li naopak $d(v)$ liché, musí být pro sousedy na nejkratších cestách $d'(u) < d(v)$ a pro všechny ostatní $d'(u) = d(v)$, takže průměr klesne pod $d'(v)$.
Průměry přes sousedy přitom můžeme spočítat násobením matic: vynásobíme matici vzdáleností D' maticí sousednosti grafu G. Na pozici i, j se objeví součet hodnot D'_{ik} přes všechny sousedy k vrcholu j. Ten stačí vydělit stupněm vrcholu j a hledaný průměr je na světě.
Po provedení jednoho násobení matic tedy dovedeme pro každou dvojici vrcholů v konstantním čase spočítat D_{ij} z D'_{ij}. Jedna úroveň rekurze proto trvá $\mathcal{O}(n^\omega)$ a jelikož průměr grafu pokaždé klesne alespoň dvakrát, je úrovní $\mathcal{O}(\log n)$ a celý algoritmus doběhne ve slíbeném čase $\mathcal{O}(n^\omega \log n)$.

6 Minimální kostry

6.1 Úvod

Věta 6.1. *Kostra T je minimální $\Leftrightarrow$ neexistuje hrana lehká vzhledem k T.*

6.2 Červeno-černý algoritmus a speciální použití

Všechny tradiční algoritmy na hledání MST lze popsat jako speciální případy následujícího meta-algoritmu. Rozeberme si tedy rovnou ten. Formulujeme ho pro případ, kdy jsou všechny váhy hran navzájem různé.

Algoritmus 6.1. ***(Červeno-modrý):*** *Náhodně vybírá hrany a kontrahuje je, dokud # vrcholů neklesne na ℓ.*

Algorithm 8 RedBlue:

1: Na počátku jsou všechny hrany bezbarvé.
2: **while** lze uplatnit alespoň jedno z pravidel **do**
3: *Modré:* $\exists e \in C$, že e je nejlehčí z C a nastav $e \leftarrow$ modrá
4: *Červené:* $\exists e \in K$ cyklus, že e je nejtěžší z K a nastav $e \leftarrow$ červená

6.2.1 Jarníkův algoritmus

Necháváme růst jen jeden modrý strom. MST je na začátku prázdná, přidáváme vždy nejlehčí hranu mezi T a $\bar{T}$.

- *Červené pravidlo:* zahodit všechny hrany v rámci stromu.
- *Modré pravidlo:* přidat nejlehčí hranu spojující T a $\bar{T}$.

Kroky opakujeme, dokud se strom nerozroste přes všechny vrcholy.
Při šikovné implementaci pomocí haldy dosáhneme časové složitosti $\mathcal{O}(m \log n)$.

6.2.2 Borůvkův algoritmus

Pěstujeme modrý les, rozšiřujeme ho ve fázích. V každé fázi nalezneme nejlevnější incidentní hranu a všechny tyto nalezené hrany naráz přidáme (modré pravidlo).
Časová složitost je $\mathcal{O}(m \log n)$: Počet stromů klesá exponenciálně, takže fází $\log n$ a navíc každou fázi implementujeme lineárním průchodem celého grafu.

6.2.3 Kruskalův algoritmus

Algoritmus 6.2. ***(Kruskalův):*** *Hrany setřídíme vzestupně podle vah, pro kadždou se podíváme, jestli spojuje 2 komponenty, pokud ano, tak přidáme.*

Algorithm 9 KRUSKAL:

1: Setřídíme hrany podle vah vzestupně.
2: Začneme s prázdnou kostrou (každý vrchol je v samostatné komponentě souvislosti).
3: Bereme hrany ve vzestupném pořadí.
4: **for all** $e \in E$ **do**
5: Podíváme se, zda e spojuje dvě různé komponenty.
6: **if** ano **then** e přidáme ji do kostry.
7: **else** zahodíme e.

Setřídění je $m \log m$, Union-Find struktura pro komponenty je $m\times$find, $n\times$unioon $\implies \mathcal{O}(\log n) \implies m \log n$.

6.3 Borůvkuv algoritmus s kontrakcemi a filtrováním

Algoritmus 6.3. ***(Borůvka s kontrakcemi):*** *Vyjdeme z myšlenky, že můžeme po každém kroku původního Borůvkova algoritmu vzniklé komponenty souvislosti grafu kontrahovat do jednoho vrcholu a tím získat menší graf, který můžeme znovu rekurzivně zmenšovat. Pro rovinné grafy tak dosáhneme lineární časové složitosti.*

Algorithm 10 CONTRACTIVEBORŮVKA:

$T \leftarrow \emptyset$
while $n > 1$ **do**
$S \leftarrow \{$*nejdelší incidentní hrana pro každý vrhol*$\}$
Kontrakce S
Odstranění smyček
Filtrace paralelních hran
$T \leftarrow T \cup S$
return T

6.4 MST v rovinných grafech a Minorově uzavřené třídy

Definice 6.1. (Minor): Graf H je *minorem* grafu G (značíme $H \preceq G$), pokud lze H získat z G mazáním vrcholů či hran a kontrahováním hran (s odstraněním smyček a násobných hran).

Definice 6.2. (Minorová uzavřenost): Třída grafů $\mathcal{C}$ je *minorově uzavřená*, pokud kdykoliv $G \in \mathcal{C}$ a $H \preceq G$, platí také $H \in \mathcal{C}$.

Definice 6.3. (Forb): Pro třídu grafů $\mathcal{C}$ definujeme $\mathrm{Forb}(\mathcal{C})$ jako třídu všech grafů, jejichž minorem není žádný graf z $\mathcal{C}$. Pro zjednodušení značení budeme pro konečné třídy psát $\mathrm{Forb}(G_1, \ldots, G_k)$ namísto $\mathrm{Forb}(\{G_1, \ldots, G_k\})$.

6.5 Hustota minorově uzavřených tříd

Definice 6.4. (Hustota): *Hustotou* neprázdného grafu G nazveme $\varrho(G) = |E(G)|/|V(G)|$. Hustotou třídy $\varrho(\mathcal{C})$ pak nazveme supremum z hustot všech neprázdných grafů ležících v této třídě.

Věta 6.2. ***(o hustotě minorově uzavřených tříd):*** *Pokud je třída grafů $\mathcal{C}$ minorově uzavřená a netriviální (alespoň jeden graf v ní leží a alespoň jeden neleží), pak má konečnou hustotu.*

Důkaz. Ukážeme nejprve, že pro každou třídu $\mathcal{C}$ existuje nějaké k takové, že $\mathcal{C} \subseteq \mathrm{Forb}(K_k)$.
Už víme, že $\mathcal{C}$ lze zapsat jako $\mathrm{Forb}(\mathcal{F})$ pro nějakou třídu $\mathcal{F}$. Označme F graf z $\mathcal{F}$ s nejmenším počtem vrcholů; pokud existuje více takových, vybereme libovolný. Hledané k zvolíme jako počet vrcholů tohoto grafu.

Inkluze tvaru $\mathcal{A} \subseteq \mathcal{B}$ je ekvivalentní tomu, že kdykoliv nějaký graf G neleží v $\mathcal{B}$, pak neleží ani v $\mathcal{A}$. Mějme tedy nějaký graf $G \notin \text{Forb}(K_k)$. Proto $K_k \preceq G$. Ovšem triviálně platí $F \preceq K_k$ a relace „být minorem“ je tranzitivní, takže $F \preceq G$, a proto $G \notin \mathcal{C}$.
Víme tedy, že $\mathcal{C} \subseteq \text{Forb}(K_k)$. Proto musí platit $\varrho(\mathcal{C}) \leq \varrho(\text{Forb}(K_k))$. Takže postačuje omezit hustotu tříd s jedním zakázaným minorem, který je úplným grafem, a to plyne z následující Maderovy věty. ■

Důsledek 6.1. *Pokud používáme kontrahující Borůvkův algoritmus na grafy ležící v nějaké netriviální minorově uzavřené třídě, pak všechny grafy, které algoritmus v průběhu výpočtu sestrojí, leží také v této třídě, takže na odhad jejich hustoty můžeme použít předchozí větu. Opět vyjde, že časová složitost algoritmu je lineární.*

Věta 6.3. ***(Maderova):*** *Pro každé $k \geq 2$ existuje $c(k)$ takové, že kdykoliv má graf hustotu alespoň $c(k)$, obsahuje jako podgraf nějaké dělení grafu K_k.*

6.6 Jarníkův/Dijkstrův algoritmus s Fibonacciho haldou

Algoritmus 6.4. ***(Jarník-Dijkstra):*** *Původní Jarníkův algoritmus s haldou má díky ní složitost $\mathcal{O}(m \log n)$, to zlepšíme použitím Fibonacciho haldy H, do které si pro každý vrchol sousedící se zatím vybudovaným stromem T uložíme nejlevnější z hran vedoucích mezi tímto vrcholem a stromem T. Tyto hrany bude halda udržovat uspořádané podle vah.*

Algorithm 11 JarníkDijkstra:

```
1:  T ← ∅
2:  stav(v_0) ← otevřený;   stav(∗) ← neviděný
3:  h(v_0) ← −∞;   h(∗) ← nedefinováno;   ae(∗) ← nedefinováno        ▷ 'ae' značí aktivní hranu
4:  while ∃v : stav(v) = otevřený do
5:      Vezmeme takové v s nejmenším h(v)
6:      stav(v) ← zavřený
7:      T ← T ∪ {ae(v)}
8:      for all vw ∈ E do
9:          if stav(w) = neviděný then
10:             stav(w) ← otevřený
11:             h(w) ← W(vw)                                        ▷ 'W' značí váhu
12:             ae(w) ← vw
13:         if stav(w) = otevřený   &   h(w) > W(vw) then
14:             h(w) ← W(vw)
        return T
```

Dostaneme tak časovou složitost $\mathcal{O}(m + n \log n)$. Pokud je ovšem $m \geq n \log n$, tak je to pouze $\mathcal{O}(m)$.

6.7 Fredman-Tarjan algoritmus

Algoritmus 6.5. ***(Fredman-Tarjan):*** *Původní Jarníkův algoritmus s haldou má díky ní složitost $\mathcal{O}(m \log n)$, to zlepšíme použitím Fibonacciho haldy H, do které si pro každý vrchol sousedící se zatím vybudovaným stromem T uložíme nejlevnější z hran vedoucích mezi tímto vrcholem a stromem T. Tyto hrany bude halda udržovat uspořádané podle vah.*

Algorithm 12 FredmanTarjan:

1: $T \leftarrow \emptyset$.
2: **while** $n > 1$ **do**
3: $F \leftarrow \emptyset$. ▷ les
4: $t \leftarrow 2^{\lceil 2m_0/n \rceil}$.
5: **while** $\exists v \in V \setminus V(F)$ **do**
6: Spustíme JarníkDijkstra omezený na t položek z vrcholu v.
7: Zastavíme, když:
(1) halda je prázdná,
(2) velikost haldy $= t$,
(3) připojíme vrchol, který už je v F.
8: Připojíme výsledný strom do F.
9: $T \leftarrow T \cup F$.
10: Kontrahujeme F.

Lemma 6.1. *Jedna fáze běží v čase $\mathcal{O}(m)$.*

Důkaz. Máme $\mathcal{O}(m_i + n_i \log t + m_i)$. Platí $m_i \leq m_0$, máme tady $\mathcal{O}(2m_0 + n_i \log t)$.
Za t do logaritmu substituujeme $2^{\lceil 2m_0/n \rceil}$ a dostaneme tak $n_i \log 2^{\lceil 2m_0/n \rceil} \in \cancel{n_i} \cdot \mathcal{O}(\frac{m_0}{\cancel{n_i}})$.
Dostaneme tak celkem $\mathcal{O}(3m_0) = \mathcal{O}(m_0)$. ■

Definice 6.5. (Tower function) je zjednodušení značení pro $2 \uparrow k = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \Big\}k\times$. Inverzní funkci značíme $\log^* k$.

Pozorování 6.1. *Počet stromů v $F_i \leq \frac{2m_i}{t} \implies n_{i+1} \leq \frac{2m_i}{t}$.*

Věta 6.4. *Fredman-Tarjan běží v čase $\mathcal{O}(m \cdot \log^* n)$.*

Důkaz.

$$t_{i+1} = 2^{\lceil 2m_0/n_{i+1} \rceil} \geq 2^{\frac{2m_0}{n_{i+1}}} \overset{(Poz.\ 6.1.)}{\geq} 2^{\frac{\cancel{2m_0} \cdot t_i}{\cancel{2m_i}}} \geq 2^{t_i}$$

Dostali jsme tedy $t_{i+1} \geq 2^{t_i}$. Využijeme tower function a: $t_i \geq 2 \uparrow i \implies$ chceme $\log^* i$.
Jedna fáze je $\mathcal{O}(m)$, takže celkem dostaneme $\mathcal{O}(m \cdot \log^* n)$. ■

7 LCA a RMQ

7.1 LCA - Lowest Common Ancestor

Chceme si předzpracovat zakořeněný strom T tak, abychom dokázali pro libovolné dva vrcholy x, y najít co nejrychleji jejich nejbližšího společného předchůdce.

Triviální řešení LCA:

- Vystoupáme z x i y do kořene, označíme vrcholy na cestách a kde se poprvé potkají, tam je hledaný předchůdce. To je lineární s hloubkou a nepotřebuje předzpracování.
- Lze vylepšit: Budeme stoupat z x a y střídavě. Tak potřebujeme jen lineárně mnoho kroků vzhledem ke vzdálenosti společného předchůdce.

7.2 RMQ - Range Minimum Query

Chceme předzpracovat posloupnost čísel $a_1, \ldots a_n$ tak, abychom uměli rychle počítat $\min_{x \leq i \leq y} a_i$.

Triviální řešení RMQ:

- Předpočítáme všechny možné dotazy: předzpracování $\mathcal{O}(n^2)$, dotaz $\mathcal{O}(1)$.
- Pro každé i a $j \leq \log n$ předpočítáme $m_{ij} = \min\{a_i, a_{i+1}, \ldots, a_{i+2^j-1}\}$, čili minima všech bloků velkých jako nějaká mocnina dvojky. Když se poté někdo zeptá na minimum bloku $a_i, a_{i+1}, \ldots, a_{j-1}$, najdeme největší k takové, že $2^k < j - i$ a vrátíme:

  $$\min(\min\{a_i, \ldots, a_{i+2^k-1}\}, \min\{a_{j-2^k}, \ldots, a_{j-1}\}).$$

  Tak zvládneme dotazy v čase $\mathcal{O}(1)$ po předzpracování v čase $\mathcal{O}(n \log n)$.

7.3 Redukce z LCA na RMQ

Lemma 7.1. *LCA lze převést na RMQ s lineárním časem na předzpracování a $\mathcal{O}(1)$ na převod dotazu.*

Důkaz. Strom projdeme do hloubky a pokaždé, když vstoupíme do vrcholu (ať již poprvé nebo se do něj vrátíme), zapíšeme jeho hloubku. LCA(x, y) pak bude nejvyšší vrchol mezi libovolnou návštěvou x a libovolnou návštěvou y. ■

Převod z LCA vytváří dosti speciální instance problému RMQ. Takové, v nichž je $|a_i - a_{i+1}| = 1$. Takovým instancím budeme říkat RMQ±1 a budeme je umět řešit šikovnou dekompozicí.

7.4 Dekompozice RMQ ± 1

Pro RMQ±1: Vstupní posloupnost rozdělíme na bloky velikosti $b = 1/2 \cdot \log n$, každý dotaz umíme rozdělit na část týkající se celých bloků a maximálně dva dotazy na části bloků.

—

Všimneme si, že ačkoliv bloků je mnoho, jejich možných typů (tj. posloupností klesání a stoupání) je pouze $2^{b-1} \leq \sqrt{n}$ a bloky téhož typu se liší pouze posunutím o konstantu. Vybudujeme proto kvadratickou strukturu pro jednotlivé typy a pro každý blok si zapamatujeme, jakého je typu a jaké má posunutí. Celkem strávíme čas $\mathcal{O}(n + \sqrt{n} \cdot \log^2 n) = \mathcal{O}(n)$ předzpracováním a $\mathcal{O}(1)$ dotazem.
Mimo to ještě vytvoříme komprimovanou posloupnost, v níž každý blok nahradíme jeho minimem. Tuto posloupnost délky n/b budeme používat pro části dotazů týkající se celých bloků a připravíme si pro ni „logaritmickou“ variantu triviální struktury. To nás bude stát $\mathcal{O}(\frac{n}{b} \cdot \log(\frac{n}{b})) = \mathcal{O}(\frac{n}{\log n} \cdot \log n) = \mathcal{O}(n)$ na předzpracování a $\mathcal{O}(1)$ na dotaz.

Co jsem nestihl sepsat:

Union find.
Union-Find with unions known in advance via Frederickson's decomposition and binary coding.

Zdroje

Čerpal především z přednášek Martina Mareše a jeho skript:

- Krajinou grafových algoritmů - skripta Martina Mareše
- Průvodce labyrintem algoritmů - skripta Martina Mareše
- Přednášky za rok 2024/25 Martina Mareše
- Poznámky z hodin Aničky Kmentové (2023/24)