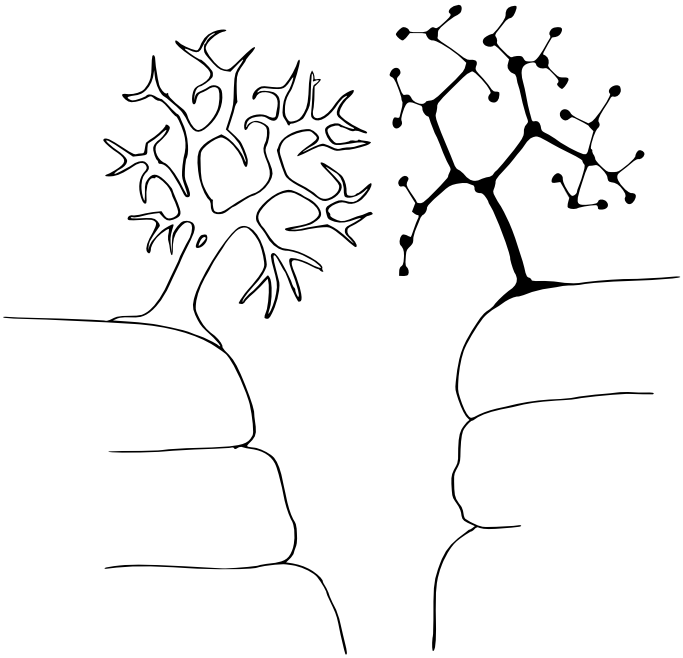
Martin Mareš

Krajinou grafových algoritmů

průvodce pro středně pokročilé



ITI 2024

Pracovní verze

0. Úvodem

Tento spisek vznikl jako učební text k přednášce z grafových algoritmů, kterou přednáším na Katedře aplikované matematiky MFF UK v Praze. Rozhodně si neklade za cíl důkladně zmapovat celé v dnešní době již značně rozkošatělé odvětví informatiky zabývající se grafy, spíše se snaží ukázat některé typické techniky a teoretické výsledky, které se při návrhu grafových algoritmů používají. Zkrátka je to takový turistický průvodce krajinou grafových algoritmů.

Jelikož přednáška se řadí mezi pokročilé kursy, dovoluji si i v tomto textu předpokládat základní znalosti teorie grafů a grafových algoritmů. V případě pochybností doporučuji obrátit se na některou z knih [41], [17] a [38]. Výbornou referenční příručkou, ze které jsem častokrát čerpal i já při sestavování přednášek, je také Schrijverova monumentální monografie Combinatorial Optimization [48].

Mé díky patří studentům Semináře z grafových algoritmů, na kterém jsem na jaře 2006 první verzi této přednášky uváděl, za výborně zpracované zápisky, jež se staly prazákladem tohoto textu. Jmenovitě:

Toky, řezy a Fordův-Fulkersonův algoritmus: Radovan Šesták
Dinicův algoritmus: Bernard Lidický
Globální souvislost a párování: Jiří Peinlich a Michal Kůrka
Gomory-Hu Trees: Milan Straka
Minimální kostry: Martin Kruliš, Petr Sušil, Petr Škoda a Tomáš Gavenčiak
Počítání na RAMu: Zdeněk Vilušínský
Q-Heapy: Cyril Strejc
Suffixové stromy: Tomáš Mikula a Jan Král
Dekompozice Union-Findu: Aleš Šnupárek

Děkuji také tvůrcům vektorového editoru Vrr, v němž jsem kreslil většinu obrázků.

V Praze v březnu 2007

Martin Mareš

Značení

V celém textu se budeme držet tohoto základního značení:

- G bude značit konečný *graf* na vstupu algoritmu (podle potřeby buďto orientovaný nebo neorientovaný; multigraf jen bude-li explicitně řečeno).
- V a E budou množiny *vrcholů* a *hran* grafu G (případně jiného grafu uvedeného v závorkách). Hranu z vrcholu u do vrcholu v budeme psát uv, ať už je orientovaná nebo ne.
- n a m bude *počet vrcholů a hran,* tedy $n := |V|$, $m := |E|$.
- Pro libovolnou množinu X vrcholů nebo hran bude $\overline{X}$ označovat doplněk této množiny; přitom z kontextu by mělo být vždy jasné, vzhledem k čemu.

Také budeme bez újmy na obecnosti předpokládat, že zpracovávaný graf je souvislý. Časovou složitost průchodu grafem do hloubky či šířky pak můžeme psát jako $\mathcal{O}(m)$, protože víme, že $n = \mathcal{O}(m)$.

1. Toky, řezy a Fordův-Fulkersonův algoritmus

V této kapitole nadefinujeme toky v sítích, odvodíme základní věty o nich a také Fordův-Fulkersonův algoritmus pro hledání maximálního toku. Také ukážeme, jak na hledání maximálního toku převést problémy týkající se řezů, separátorů a párování v bipartitních grafech. Další tokové algoritmy budou následovat v příštích kapitolách.

Toky v sítích

Intuitivní pohled: síť je systém propojených potrubí, který přepravuje tekutinu ze zdroje s (source) do spotřebiče t (target), přičemž tekutina se nikde mimo tato dvě místa neztrácí ani nevzniká.

Definice:

- *Síť* je uspořádaná pětice (V, E, s, t, c), kde:
  - (V, E) je orientovaný graf,
  - $s \in V$ *zdroj,*
  - $t \in V$ *spotřebič* neboli *stok* a
  - $c : E \to \mathbb{R}$ funkce udávající nezáporné *kapacity* hran.
- *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].
- *Tok* je ohodnocení $f : E \to \mathbb{R}$, pro které platí:
  - $\forall e : 0 \le f(e) \le c(e)$, *(dodržuje kapacity)*
  - $\forall v \neq s, t : f^\Delta(v) = 0$. *(Kirchhoffův zákon)*
- *Velikost toku:* $|f| = -f^\Delta(s)$.
- *Řez (tokový):* množina vrcholů $C \subset V$ taková, že $s \in C$, $t \notin C$. Řez můžeme také ztotožnit s množinami hran $C^- = E \cap (C \times \overline{C})$ [těm budeme říkat hrany zleva doprava] a $C^+ = E \cap (\overline{C} \times C)$ [hrany zprava doleva].
- *Kapacita řezu:* $|C| = \sum_{e \in C^-} c(e)$ (bereme v úvahu jen hrany zleva doprava).
- *Tok přes řez:* $f^+(C) = \sum_{e \in C^+} f(e)$, $f^-(C) = \sum_{e \in C^-} f(e)$, $f^\Delta(C) = f^+(C) - f^-(C)$.
- *Cirkulace* je tok nulové velikosti, čili f takové, že $f^\Delta(v) = 0$ pro všechna v.

Základním problémem, kterým se budeme zabývat, je hledání *maximálního toku* (tedy toku největší možné velikosti) a *minimálního řezu* (řezu nejmenší možné kapacity).

Větička: V každé síti existuje maximální tok a minimální řez.

Důkaz: Existence minimálního řezu je triviální, protože řezů v každé síti je konečně mnoho; pro toky v sítích s reálnými kapacitami to ovšem není samozřejmost a je k tomu potřeba trocha matematické analýzy (v prostoru všech ohodnocení hran tvoří toky kompaktní množinu, velikost toku je lineární funkce, a tedy i spojitá, pročež nabývá maxima). Pro racionální kapacity dostaneme tuto větičku jako důsledek analýzy Fordova-Fulkersonova algoritmu. ♡

Pozorování: Kdybychom velikost toku definovali podle spotřebiče, vyšlo by totéž. Platí totiž:

$$f^\Delta(s) + f^\Delta(t) = \sum_v f^\Delta(v) = \sum_e f(e) - f(e) = 0$$

(první rovnost plyne z Kirchhoffova zákona – všechna ostatní $f^\Delta(v)$ jsou nulová; druhá pak z toho, že každá hrana přispěje k jednomu $f^+(v)$ a k jednomu $f^-(v)$). Proto je $|f| = -f^\Delta(s) = f^\Delta(t)$.

Stejně tak můžeme velikost toku změřit na libovolném řezu:

Lemma: Pro každý řez C platí, že $|f| = -f^\Delta(C) \leq |C|$.

Důkaz: První část indukcí: každý řez můžeme získat postupným přidáváním vrcholů do triviálního řezu $\{s\}$ [čili přesouváním vrcholů zprava doleva], a to, jak ukáže jednoduchý rozbor případů, nezmění f^Δ. Druhá část: $-f^\Delta(C) = f^-(C) - f^+(C) \leq f^-(C) \leq |C|$. ♡

Víme tedy, že velikost každého toku lze omezit kapacitou libovolného řezu. Kdybychom našli tok a řez stejné velikosti, můžeme si proto být jisti, že tok je maximální a řez minimální. To se nám opravdu povede, platí totiž následující věta:

Věta (Ford, Fulkerson): V každé síti je velikost maximálního toku rovna velikosti minimálního řezu.

Důkaz: Jednu nerovnost jsme dokázali v předchozím lemmatu, druhá plyne z duality lineárního programování [max. tok a min. řez jsou navzájem duální úlohy], ale k pěknému kombinatorickému důkazu půjde opět použít Fordův-Fulkersonův algoritmus. ♡

Fordův-Fulkersonův 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. Trochu formálněji:

Definice:

- *Rezerva* hrany $e = (v, w)$ při toku f se definuje jako $r(e) = (c(e) - f(e)) + f(e')$, kde $e' = (w, v)$. Pro účely tohoto algoritmu budeme předpokládat, že ke každé hraně existuje hrana opačná; pokud ne, dodefinujeme si ji a dáme jí nulovou kapacitu.

- *Zlepšující cesta* je orientovaná cesta, jejíž všechny hrany mají nenulovou rezervu.

Algoritmus:

1. $f \leftarrow$ *nulový tok*.
2. Dokud existuje zlepšující cesta P z s do t:
3. $\quad \varepsilon \leftarrow \min_{e \in P} r(e)$.
4. $\quad$ Zvětšíme tok f podél P o ε (každé hraně $e \in P$ zvětšíme $f(e)$, případně zmenšíme $f(e')$, podle toho, co jde).

Analýza: Nejdříve si rozmysleme, že pro celočíselné kapacity algoritmus vždy doběhne: v každém kroku stoupne velikost toku o $\varepsilon \geq 1$, což může nastat pouze konečněkrát. Podobně pro 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. Pro iracionální kapacity obecně doběhnout nemusí, zkuste vymyslet protipříklad.

Uvažme nyní situaci po zastavení algoritmu. Funkce f je určitě tok, protože jím byla po celou dobu běhu algoritmu. Prozkoumejme teď množinu C vrcholů, do nichž po zastavení algoritmu vede zlepšující cesta ze zdroje. Jistě $s \in C$, $t \notin C$, takže tato množina je řez. Navíc pro každou hranu $e \in C^-$ musí být $f(e) = c(e)$ a pro každou $e' \in C^+$ je $f(e') = 0$, protože jinak by rezerva hrany e nebyla nulová. Takže $f^-(C) = |C|$ a $f^+(C) = 0$, čili $|f| = |C|$.

Našli jsme tedy k toku, který algoritmus vydal, řez stejné velikosti, a proto, jak už víme, je tok maximální a řez minimální. Tím jsme také dokázali Fordovu-Fulkersonovu větu[1] a existenci maximálního toku. Navíc algoritmus nikdy nevytváří z celých čísel necelá, čímž získáme:

Důsledek: Síť s celočíselnými kapacitami má maximální tok, který je celočíselný.

Časová složitost F-F algoritmu může být pro obecné sítě a nešikovnou volbu zlepšujících cest obludná, ale jak dokázali Edmonds s Karpem, pokud budeme hledat cesty prohledáváním do šířky (což je asi nejpřímočařejší implementace), poběží v čase $\mathcal{O}(m^2 n)$. Pokud budou všechny kapacity jednotkové, snadno nahlédneme, že stačí $\mathcal{O}(nm)$. Edmondsův a Karpův odhad nebudeme dokazovat, místo toho si v příští kapitole předvedeme efektivnější algoritmus.

Řezy, separátory a k-souvislost

Teorie toků nám rovněž poslouží ke zkoumání násobné souvislosti grafů.

Definice: Pro každý neorientovaný graf G a libovolné jeho vrcholy s, t zavedeme:

- *st-řez* je množina hran F taková, že v grafu $G - F$ jsou vrcholy s, t v různých komponentách souvislosti.

[1] Dokonce jsme ji dokázali i pro reálné kapacity, protože můžeme algoritmus spustit rovnou na maximální tok místo nulového a on se ihned zastaví a vydá certifikující řez.

- *st-separátor* je množina vrcholů W taková, že $s, t \notin W$ a v grafu $G - W$ jsou vrcholy s, t v různých komponentách souvislosti.
- *Řez* je množina hran, která je xy-řezem pro nějakou dvojici vrcholů x, y.
- *Separátor* je množina vrcholů, která je xy-separátorem pro nějakou dvojici vrcholů x, y.
- G je *hranově k-souvislý,* pokud $|V| > k$ a všechny řezy v G mají alespoň k hran.
- G je *vrcholově k-souvislý,* pokud $|V| > k$ a všechny separátory v G mají alespoň k vrcholů.

Všimněte si, že nesouvislý graf má řez i separátor nulové velikosti, takže vrcholová i hranová 1-souvislost splývají s obyčejnou souvislostí pro všechny grafy o alespoň dvou vrcholech. Hranově 2-souvislé jsou právě (netriviální) grafy bez *mostů,* vrcholově 2-souvislé jsou ty bez *artikulací.*

Pro orientované grafy můžeme *st*-řezy a *st*-separátory definovat analogicky (totiž, že po odstranění příslušné množiny hran či vrcholů neexistuje orientovaná cesta z s do t), globální řezy a separátory ani vícenásobná souvislost se obvykle nedefinují.

Poznámka: Minimální orientované *st*-řezy podle této definice a minimální tokové řezy podle definice ze začátku kapitoly splývají: každý tokový řez C odpovídá *st*-řezu stejné velikosti tvořenému hranami v C^-; naopak pro minimální *st*-řez musí být množina vrcholů dosažitelných z s po odebrání řezu z grafu tokovým řezem, opět stejné velikosti. [Velikost měříme součtem kapacit hran.] Dává tedy rozumný smysl říkat obojímu stejně. Podobně se chovají i neorientované grafy, pokud do „tokového" řezu počítáme hrany v obou směrech.

Analogií toků je pak existence nějakého počtu disjunktních cest (vrcholově nebo hranově) mezi vrcholy s a t. Analogií F-F věty pak budou známé Mengerovy věty:

Věta (Mengerova, lokální hranová orientovaná): Buď G orientovaný graf a s, t nějaké jeho vrcholy. Pak je velikost minimálního *st*-řezu rovna maximálnímu počtu hranově disjunktních *st*-cest.⟨2⟩

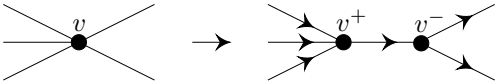
Důkaz: Z grafu sestrojíme síť tak, že s bude zdroj, t spotřebič a všem hranám nastavíme kapacitu na jednotku. Řezy v této síti odpovídají řezům v původním grafu. Podobně každý systém hranově disjunktních *st*-cest odpovídá toku stejné velikosti a naopak ke každému celočíselnému toku dovedeme najít systém disjunktních cest (hladově tok rozkládáme na cesty a průběžně odstraňujeme cirkulace, které objevíme). Pak použijeme F-F větu. ♡

Věta (Mengerova, lokální vrcholová orientovaná): Buď G orientovaný graf a s, t nějaké jeho vrcholy takové, že $st \notin E$. Pak je velikost minimálního *st*-separátoru rovna maximálnímu počtu vrcholově disjunktních *st*-cest.⟨3⟩

⟨2⟩ orientovaných cest z s do t

⟨3⟩ Tím myslíme cesty disjunktní až na krajní vrcholy.

Důkaz: Podobně jako v důkazu předchozí věty zkonstruujeme vhodnou síť. Tentokrát ovšem rozdělíme každý vrchol na vrcholy v^+ a v^-, všechny hrany, které původně vedly do v, přepojíme do v^+, hrany vedoucí z v povedou z v^- a přidáme novou hranu z v^+ do v^-. Všechny hrany budou mít jednotkové kapacity. Toky nyní odpovídají vrcholově disjunktním cestám, řezy v síti separátorům. ♡



Rozdělení vrcholu

Podobně dostaneme neorientované lokální věty (neorientovanou hranu nahradíme orientovanými v obou směrech) a z nich pak i globální varianty popisující k-souvislost grafů:

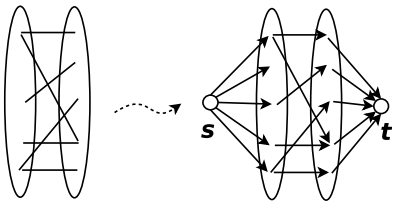
Věta (Mengerova, globální hranová neorientovaná): Neorientovaný graf G je hranově k-souvislý právě tehdy, když mezi každými dvěma vrcholy existuje alespoň k hranově disjunktních cest.

Věta (Mengerova, globální vrcholová neorientovaná): Neorientovaný graf G je vrcholově k-souvislý právě tehdy, když mezi každými dvěma vrcholy existuje alespoň k vrcholově disjunktních cest.

Maximální párování v bipartitním grafu

Jiným problémem, který lze snadno převést na hledání maximálního toku, je nalezení *maximálního párování* v bipartitním grafu, tedy množiny hran takové, že žádné dvě hrany nemají společný vrchol. Maximálním míníme vzhledem k počtu hran, nikoliv vzhledem k inkluzi.

Z bipartitního grafu $(A \cup B, E)$ sestrojíme síť obsahující všechny původní vrcholy a navíc dva nové vrcholy s a t, dále pak všechny původní hrany orientované z A do B, nové hrany z s do všech vrcholů partity A a ze všech vrcholů partity B do t. Kapacity všech hran nastavíme na jedničky:



Nyní si všimneme, že párování v původním grafu odpovídají celočíselným tokům v této síti a že velikost toku je rovna velikosti párování. Stačí tedy nalézt maximální celočíselný tok (třeba F-F algoritmem) a do párování umístit ty hrany, po kterých něco teče.

Podobně můžeme najít souvislost mezi řezy v této síti a *vrcholovými pokrytími* zadaného grafu – to jsou množiny vrcholů takové, že se dotýkají každé hrany. Tak z F-F věty získáme jinou standardní větu:

Věta (Königova): V každém bipartitním grafu je velikost maximálního párování rovna velikosti minimálního vrcholového pokrytí.

Důkaz: Pokud je W vrcholové pokrytí, musí hrany vedoucí mezi vrcholy této množiny a zdrojem a spotřebičem tvořit stejně velký řez, protože každá st-cesta obsahuje alespoň jednu hranu bipartitního grafu a ta je pokryta. Analogicky vezmeme-li libovolný st-řez (ne nutně tokový, stačí hranový), můžeme ho bez zvětšení upravit na st-řez používající pouze hrany ze s a do t, kterému přímočaře odpovídá vrcholové pokrytí stejné velikosti. ♡

Některé algoritmy na hledání maximálního párování využívají také volné střídavé cesty:

Definice: *(Volná) střídavá cesta* v grafu G s párováním M je cesta, která začíná i končí nespárovaným vrcholem a střídají se na ní hrany ležící v M s hranami mimo párování.

Všimněte si, že pro bipartitní grafy odpovídají zlepšující cesty v příslušné síti právě volným střídavým cestám a zlepšení toku podél cesty odpovídá přexorováním párování volnou střídavou cestou. Fordův-Fulkersonův algoritmus tedy lze velice snadno formulovat i v řeči střídavých cest.

2. Dinicův algoritmus a jeho varianty

Maximální tok v síti už umíme najít pomocí Fordova-Fulkersonova algoritmu z minulé kapitoly. Nyní pojednáme o efektivnějším Dinicově algoritmu a o různých jeho variantách pro sítě ve speciálním tvaru.

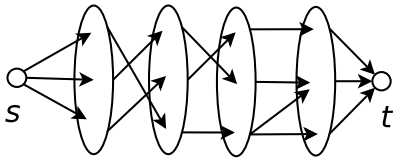
Dinicův algoritmus

Dinicův algoritmus je založen na následující myšlence: Ve Fordově-Fulkersonově algoritmu nemusíme přičítat jen zlepšující cesty, ale je možné přičítat rovnou zlepšující toky. Nejlépe toky takové, aby je nebylo obtížné najít, a přitom aby původní tok dostatečně obohatily. Vhodnými objekty k tomuto účelu jsou blokující toky:

Definice: *Blokující tok* je tok takový, že každá orientovaná *st*-cesta obsahuje alespoň jednu nasycenou hranu. [Tj. takový tok, který by našel F-F algoritmus, kdyby neuvažoval rezervy v protisměru.]

Algoritmus (Dinic):

1. Začneme s libovolným tokem f, například prázdným (všude nulovým).
2. Iterativně vylepšujeme tok pomocí zlepšujících toků: *(vnější cyklus)*
3. Sestrojíme síť rezerv: vrcholy a hrany jsou tytéž, kapacity jsou určeny rezervami v původní síti. Dále budeme pracovat s ní.
4. Najdeme nejkratší *st*-cestu. Když žádná neexistuje, skončíme.
5. Pročistíme síť, tj. ponecháme v ní pouze vrcholy a hrany na nejkratších *st*-cestách.
6. Najdeme v pročištěné síti blokující tok f_B:
7. $f_B \leftarrow$ *prázdný tok*
8. Postupně přidáváme *st*-cesty: *(vnitřní cyklus)*
9. Najdeme *st*-cestu. Např. hladově – „rovnou za nosem“.
10. Pošleme co nejvíce po nalezené cestě.
11. Smažeme nasycené hrany. (Pozor, smazáním hran mohou vzniknout slepé uličky, čímž se znečistí síť a nebude fungovat hladové hledání cest.)
12. Dočistíme síť.
13. Zlepšíme f podle f_B



Pročištěná síť rozdělená do vrstev

Složitost algoritmu: Označme l délku nejkratší *st*-cesty, n počet vrcholů sítě a m počet hran sítě.

- Jeden průchod vnitřním cyklem trvá $\mathcal{O}(l)$, což můžeme odhadnout jako $\mathcal{O}(n)$, protože vždy platí $l \leq n$.
- Vnitřní cyklus se provede maximálně m-krát, protože se vždy alespoň jedna hrana nasytí a ze sítě vypadne, takže krok 6 mimo podkroku 12 bude trvat $\mathcal{O}(mn)$.
- Čištění a dočišťování sítě dohromady provedeme takto:
  - Rozvrstvíme vrcholy podle vzdálenosti od s.
  - Zařízneme dlouhé cesty (delší, než do vrstvy obsahující t).
  - Držíme si frontu vrcholů, které mají $\deg^+ = 0$ či $\deg^- = 0$.
  - Vrcholy z fronty vybíráme a zahazujeme včetně hran, které vedou do/z nich. A případně přidáváme do fronty vrcholy, kterým při tom klesl jeden ze stupňů na 0. Vymažou se tím slepé uličky, které by vadily v podkroku 9.

  Takto kroky 5 a 12 dohromady spotřebují čas $\mathcal{O}(m)$.
- Jeden průchod vnějším cyklem tedy trvá $\mathcal{O}(mn)$.
- Jak za chvíli dokážeme, každým průchodem vnějším cyklem l vzroste, takže průchodů bude maximálně n a celý algoritmus tak poběží v čase $\mathcal{O}(n^2 m)$.

Korektnost algoritmu: Když se Dinicův algoritmus zastaví, nemůže už existovat žádná zlepšující cesta (viz krok 4) a tehdy, jak už víme z analýzy F-F algoritmu, je nalezený tok maximální.

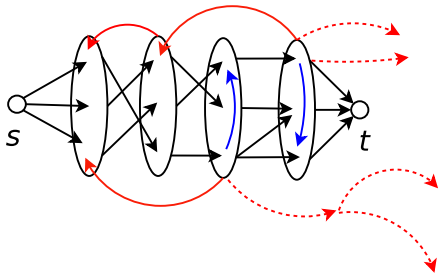
Věta: V každém průchodu Dinicova algoritmu vzroste l alespoň o 1.

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:

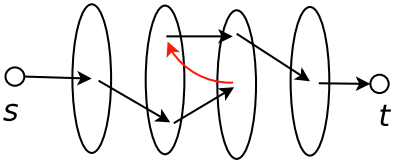
Vnější cyklus začíná s nepročištěnou sítí. Příklad takové sítě je na následujícím obrázku. Po pročištění zůstanou v síti jen černé hrany, tedy hrany vedoucí z i-té vrstvy do $(i+1)$-ní. Červené a modré[⟨4⟩] se zahodí.

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. ♡

[⟨4⟩] Modré jsou ty, které vedou v rámci jedné vrstvy, červené vedou zpět či za spotřebič či do slepých uliček. Při vytištění na papír vypadají všechny černě.



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



Cesta užívající novou zpětnou hranu

Poznámky

- Není potřeba tak puntičkářské čištění. Vrcholy se vstupním stupněm 0 nám nevadí – stejně se do nich nedostaneme. Vadí jen vrcholy s výstupním stupněm 0, kde by mohl havarovat postup v podkroku 9.
- Je možné dělat prohledávání a čištění současně. Jednoduše prohledáváním do hloubky: „Hrrr na ně!“ a když to nevyjde (dostaneme se do slepé uličky), kus ustoupíme a při ústupu čistíme síť odstraňováním slepé uličky.
- Už při prohledávání si rovnou udržujeme minimum z rezerv a při zpáteční cestě opravujeme kapacity. Snadno zkombinujeme s prohledáváním do hloubky.
- V průběhu výpočtu udržujeme jen síť rezerv a tok vypočteme až nakonec z rezerv a kapacit.
- Když budeme chtít hledat minimální řez, spustíme po Dinicovu algoritmu ještě jednu iterací F-F algoritmu.

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.

Jednotkové kapacity: 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.⟨5⟩

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 $m/l \cdot \mathcal{O}(l) = \mathcal{O}(m)$.⟨6⟩ Tedy pro jednotkové kapacity dostáváme složitost $\mathcal{O}(nm)$.

Jednotkové kapacity znovu a lépe: Vnitřní cyklus lépe udělat nepůjde. Je potřeba alespoň lineární čas pro čištění. Můžeme se ale pokusit lépe odhadnout počet iterací vnějšího cyklu.

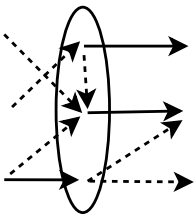
Sledujme stav sítě po k iteracích vnějšího cyklu a pokusme se odhadnout, kolik iterací ještě algoritmus udělá. Označme l délku nejkratší st-cesty. Víme, že $l > k$, protože v každé iteraci vzroste l alespoň o 1.

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?⟨7⟩ 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⟨8⟩. 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.

Jednotkové kapacity a jeden ze stupňů roven 1: 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.



⟨5⟩ Často se to implementuje tak, že protisměrné hrany vůbec nevytvoříme a když hranu nasytíme, tak v síti rezerv prostě obrátíme její orientaci.

⟨6⟩ Nebo by šlo argumentovat, tím že každou hranu použijeme jen 1×.

⟨7⟩ Přeci v řeznictví. Kdepak, spíše v cukrárně. Myslíte, že v cukrárně mají Dinicovy řezy? Myslím, že v cukrárně je většina řezů minimální. *(odposlechnuto na přednášce)*

⟨8⟩ Princip holubníku a nějaká ta ± 1.

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 $\mathcal{O}(\sqrt{n} \cdot m)$.

Mimochodem, hledání maximálního párování pomocí Dinicova algoritmu je také ekvivalentní známému Hopcroft-Karpově algoritmu [31]. Ten je založen na střídavých cestách z předchozí kapitoly a v každé iteraci nalezne množinu vrcholově disjuktních nejkratších střídavých cest, která je maximální vzhledem k inkluzi. Touto množinou pak aktuální párování přexoruje, čímž ho zvětší. Všimněte si, že tyto množiny cest odpovídají právě blokujícím tokům v pročištěné síti rezerv, takže můžeme i zde použít náš odhad na počet iterací.

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

Obecný odhad pro celočíselné kapacity: 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 $\mathcal{O}(|f|n + nm)$

Pokud navíc budeme předpokládat, že kapacita hran je nejvýše C a G není multigraf, můžeme využít toho, že $|f| \leq Cn$ (omezeno řezem okolo s) a získat odhad $\mathcal{O}(Cn^2 + nm)$.

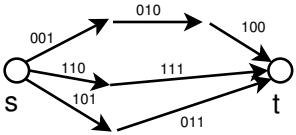
Škálování kapacit

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.

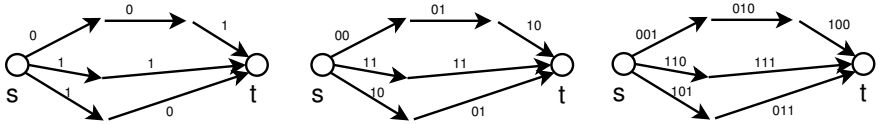
V našem případě budeme postupně budovat sítě čím dál podobnější zadané síti a v nich počítat toky, až nakonec získáme tok pro ni.

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.



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.



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

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 $\mathcal{O}(mn \log C)$.

Algoritmus tří Indů

Překvapení na konec: Dinicův algoritmus lze poměrně snadno zrychlit i ve zcela obecném případě. Malhotra, Kumar a Maheshwari vymysleli efektivnější algoritmus [39] na hledání blokujícího toku ve vrstevnaté síti, který běží v čase $\mathcal{O}(n^2)$ a použijeme-li ho v Dinicově algoritmu, zrychlíme hledání maximálního toku na $\mathcal{O}(n^3)$. Tento algoritmus vešel do dějin pod názvem Metoda tří Indů.

Mějme tedy nějakou vrstevnatou síť. Začneme s nulovým tokem a budeme ho postupně zlepšovat. Průběžně si budeme udržovat rezervy hran $r(e)$[9] a také následující rezervy vrcholů:

Definice: $r^+(v)$ je součet rezerv všech hran vstupujících do v, $r^-(v)$ součet rezerv hran vystupujících z v a konečně $r(v) := \min(r^+(v), r^-(v))$.

V každé iteraci algoritmu nalezneme vrchol s nejnižším $r(v)$ a zvětšíme tok tak, aby se tato rezerva vynulovala. Za tímto účelem nejdříve přepravíme $r(v)$ jednotek toku ze zdroje do v: u každého vrcholu w si budeme pamatovat *plán* $p(w)$, což bude množství tekutiny, které potřebujeme dostat ze zdroje do w. Nejdříve budou plány všude nulové až na $p(v) = r(v)$. Pak budeme postupovat po vrstvách směrem ke zdroji a plány všech vrcholů splníme tak, že je převedeme na plány vrcholů v následující vrstvě, až doputujeme ke zdroji, jehož plán je splněn triviálně. Nakonec analogickým způsobem protlačíme $r(v)$ jednotek z v do spotřebiče.

Během výpočtu průběžně přepočítáváme všechna r^+, r^- a r podle toho, jak se mění rezervy jednotlivých hran (při každé úpravě rezervy to zvládneme v konstantním čase) a síť čistíme stejně jako u Dinicova algoritmu.

Algoritmus: (hledání blokujícího toku ve vrstevnaté síti podle tří Indů)

1. $f_B \leftarrow$ *prázdný tok*.
2. Spočítáme rezervy všech hran a r^+, r^- a r všech vrcholů. (Tyto hodnoty budeme posléze udržovat při každé změně toku po hraně.)
3. Dokud v síti existují vrcholy s nenulovou rezervou, vezmeme vrchol v s nejmenším $r(v)$ a provedeme pro něj: *(vnější cyklus)*
4. Převedeme $r(v)$ jednotek toku z s do v následovně:
5. Položíme $p(v) \leftarrow r(v)$, $p(\cdot) = 0$.
6. Procházíme vrcholy sítě po vrstvách od v směrem k s. Pro každý vrchol w provedeme:
7. Dokud $p(w) > 0$:
8. Vezmeme libovolnou hranu uw a tok po ní zvýšíme o $\delta = \min(r(uw), p(w))$. Tím se $p(w)$ sníží o δ a $p(u)$ zvýší o δ.
9. Pokud se hrana uw nasytila, odstraníme jí ze sítě a síť dočistíme.
10. Analogicky převedeme $r(v)$ jednotek z v do t.

[9] počítáme pouze rezervu ve směru hrany, neboť nám stačí najít blokující tok, ne nutně maximální

Analýza: Nejprve si všimneme, že cyklus v kroku 8 opravdu dokáže vynulovat $p(w)$. Součet všech $p(w)$ přes každou vrstvu je totiž nejvýše roven $r(v)$, takže speciálně každé $p(w) \leq r(v)$. Jenže $r(v)$ jsme vybrali jako nejmenší, takže $p(w) \leq r(v) \leq r(w) \leq r^+(w)$, a proto je plánovaný tok kudy přivést. Proto se algoritmus zastaví a vydá blokující tok.

Zbývá odhadnout časovou složitost: Když oddělíme převádění plánů po hranách (kroky 7–9), zbytek jedné iterace vnějšího cyklu trvá $\mathcal{O}(n)$ a těchto iterací je nejvýše n. Všechna převedení plánu si rozdělíme na ta, kterými se nějaká hrana nasytila, a ta, která skončila vynulováním $p(w)$. Těch prvních je $\mathcal{O}(m)$, protože každou takovou hranu vzápětí odstraníme a čištění, jak už víme, trvá také lineárně dlouho. Druhý případ nastane pro každý vrchol nejvýše jednou za iteraci. Dohromady tedy trvají všechna převedení $\mathcal{O}(n^2)$, stejně jako zbytek algoritmu. ♡

Přehled variant Dinicova algoritmu

varianta	*čas*
standardní	$\mathcal{O}(n^2m)$
jednotkové kapacity	$\mathcal{O}(\sqrt{m} \cdot m) = \mathcal{O}(m^{3/2})$
jednotkové kapacity, 1 stupeň ≤ 1	$\mathcal{O}(\sqrt{n} \cdot m)$
jednotkové kapacity, prostý graf	$\mathcal{O}(n^{2/3}m)$
celočíselné kapacity	$\mathcal{O}(\lvert f\rvert \cdot n + nm)$
celočíselné kapacity $\leq C$	$\mathcal{O}(Cn^2 + mn)$
celočíselné kapacity $\leq C$ (škálování)	$\mathcal{O}(mn \log C)$
tři Indové	$\mathcal{O}(n^3)$

3. Bipartitní párování a globální k-souvislost

V předešlých kapitolách jsme se zabývali aplikacemi toků na hledání maximálního párování a minimálního *st*-řezu. Nyní si předvedeme dva algoritmy pro podobné problémy, které se obejdou bez toků.

Maximální párování v regulárním bipartitním grafu [1]

Nejprve si nadefinujme operaci *štěpení grafu,* která zadaný graf $G = (V, E)$ se všemi vrcholy sudého stupně a sudým počtem hran v každé komponentě souvislosti rozloží na disjunktní podgrafy $G_1 = (V, E_1)$ a $G_2 = (V, E_2)$, v nichž bude pro každý vrchol v platit $\deg_{G_1}(v) = \deg_{G_2}(v) = \deg_G(v)/2$. Tuto operaci můžeme snadno provést v lineárním čase tak, že si graf rozdělíme na komponenty, v každé nalezneme eulerovský tah a jeho hrany budeme přidávat střídavě do G_1 a do G_2.

Štěpení nám pomůže ke snadnému algoritmu pro nalezení maximálního párování ve 2^t-regulárním bipartitním grafu.⟨10⟩ Komponenty takového grafu mají určitě sudý počet hran, takže jej můžeme rozštěpit na dva 2^{t-1}-regulární grafy. Libovolný jeden z nich pak opět rozštěpíme atd., až dostaneme 1-regulární graf, který je perfektním párováním v G. To vše jsme schopni stihnout v lineárním čase, jelikož velikosti grafů, které štěpíme, exponenciálně klesají. Také bychom mohli rekurzivně zpracovávat obě části a tak se v čase $\mathcal{O}(m \log n)$ dobrat ke kompletní 1-faktorizaci zadaného grafu.⟨11⟩

Pokud zadaný graf nebude 2^t-regulární, pomůžeme si tím, že ho novými hranami doplníme na 2^t-regulární a pak si při štěpeních budeme vybírat ten podgraf, do kterého padlo méně nových hran, a ukážeme, že nakonec všechny zmizí. Abychom graf příliš nezvětšili, budeme se snažit místo přidávání úplně nových hran pouze zvyšovat násobnost hran existujících. Pro každou hranu e si tedy budeme pamatovat její násobnost $n(e)$.

Štěpení grafu s násobnostmi pak budeme provádět následovně: hranu e s násobností $n(e)$ umístíme do G_1 i do G_2 s násobností $\lfloor n(e)/2 \rfloor$ a pokud bylo $n(e)$ liché, přidáme hranu do pomocného grafu G'. Všimněte si, že G' bude graf bez násobností, v němž mají všechny vrcholy sudý stupeň, takže na něj můžeme aplikovat původní štěpící algoritmus a G'_i přiřadit ke G_i. To vše zvládneme v čase $\mathcal{O}(m)$.

Mějme nyní k-regulární bipartitní graf. Obě jeho partity jsou stejně velké, označme si počet vrcholů v každé z nich n. Zvolme t tak aby $2^t \geq kn$. Zvolme dále parametry $\alpha := \lfloor 2^t/k \rfloor$ a $\beta := 2^t \bmod k$. Každé původní hraně nastavíme násobnost α a přidáme triviální párování F (i-tý vrchol vlevo se spojí s i-tým vrcholem vpravo) s násobností β. Všimněte si, že $\beta < k$, a proto hran v F (včetně násobností) bude méně než 2^t.

⟨10⟩ Všimněte si, že takové párování bude vždy perfektní (viz Hallova věta).

⟨11⟩ To je rozklad hran grafu na disjunktní perfektní párování a má ho každý regulární bipartitní graf.

Takto získáme 2^t-regulární graf, jehož reprezentace bude lineárně velká. Na tento graf budeme aplikovat operaci štěpení a budeme si vybírat vždy tu polovinu, kde bude méně hran z F. Po t iteracích dospějeme k párování a jelikož se v každém kroku zbavíme alespoň poloviny hran z F, nebude toto párování obsahovat žádnou takovou hranu a navíc nebude obsahovat ani násobné hrany, takže bude podgrafem zadaného grafu, jak potřebujeme.

Časová složitost algoritmu je $\mathcal{O}(m \log n)$, jelikož provádíme inicializaci v čase $\mathcal{O}(m)$ a celkem $\log_2 kn = \mathcal{O}(\log n)$ iterací po $\mathcal{O}(m)$.

Stupeň souvislosti grafu

Problém zjištění *stupně hranové souvislosti* grafu lze převést na problém hledání minimálního řezu, který již pro zadanou dvojici vrcholů umíme řešit pomocí Dinicova algoritmu v čase $\mathcal{O}(n^{2/3}m)$. Pokud chceme najít minimum ze všech řezů v grafu, můžeme vyzkoušet všechny dvojice (s,t). To však lze snadno zrychlit, pokud si uvědomíme, že jeden z vrcholů (třeba s) můžeme zvolit pevně: vezmeme-li libovolný řez C, pak jistě najdeme alespoň jedno t, které padne do jiné komponenty než pevně zvolené s, takže minimální st-řez bude nejvýše tak velký jako C. V orientovaném grafu musíme projít jak řezy pro $s \to t$, tak i $t \to s$. Algoritmus bude mít složitost $\mathcal{O}(n^{5/3}m)$.

U *vrcholové k-souvislosti* to ovšem tak snadno nepůjde. Pokud by totiž fixovaný vrchol byl součástí nějakého minimálního separátoru, algoritmus může selhat. Přesto ale nemusíme procházet všechny dvojice vrcholů. Stačí jako s postupně zvolit více vrcholů, než je velikost minimálního separátoru. Algoritmus si tedy bude pamatovat, kolik vrcholů už prošel a nejmenší zatím nalezený st-separátor, a jakmile počet vrcholů překročí velikost separátoru, prohlásí separátor za minimální. To zvládne v čase $\mathcal{O}(\kappa(G) \cdot n^{3/2}m)$, kde $\kappa(G)$ je nalezený stupeň souvislosti G.

Pro minimální řezy v neorientovaných grafech ovšem existuje následující rychlejší algoritmus:

Globálně minimální řez (Nagamochi, Ibaraki [42])

Buď G neorientovaný multigraf s nezáporným ohodnocením hran. Označíme si:

Značení:

- $r(u,v)$ buď kapacita minimálního uv-řezu,
- $d(P,Q)$ buď kapacita hran vedoucích mezi množinami $P, Q \subseteq V$,
- $d(P) = d(P, \overline{P})$ buď kapacita hran vedoucích mezi $P \subseteq V$ a zbytkem grafu,
- $d(v) = d(\{v\})$ buď kapacita hran vedoucích z v (tedy pro neohodnocené grafy stupeň v),
- analogicky zavedeme $d(v,w)$ a $d(v,P)$.

Definice: *Legálním uspořádáním vrcholů* (LU) budeme nazývat lineární uspořádání vrcholů $v_1 \dots v_n$ takové, že platí $d(\{v_1 \dots v_{i-1}\}, v_i) \geq d(\{v_1 \dots v_{i-1}\}, v_j)$ pro každé $1 \leq i < j \leq n$.

Lemma: Je-li $v_1 \dots v_n$ LU na G, pak $r(v_{n-1}, v_n) = d(v_n)$.

Důkaz: Buď C libovolný řez oddělující v_{n-1} a v_n. Dokážeme, že jeho velikost je alespoň $d(v_n)$. Utvořme posloupnost vrcholů u_i takto:

1. $u_0 := v_1$
2. $u_i := v_j$ tak, že $j > i$, v_i a v_j jsou odděleny řezem C a j je minimální takové. [Tedy v_j je nejbližší vrchol na druhé straně řezu.]

Každé u_{i-1} je tedy buď rovno u_i, pokud jsou v_i a v_{i-1} na stejné straně řezu, nebo rovno v_i, pokud jsou v_i a v_{i-1} na stranách opačných. Z toho dostáváme, že $d(\{v_1 \dots v_{i-1}\}, u_i) \leq d(\{v_1 \dots v_{i-1}\}, u_{i-1})$, protože buďto $u_{i-1} = u_i$, a pak je nerovnost splněna jako rovnost, nebo je $u_{i-1} = v_i$ a nerovnost plyne z legálnosti uspořádání.

Chceme ukázat, že velikost našeho řezu C je alespoň taková, jako velikost řezu kolem vrcholu v_n. Všimneme si, že $|C| \geq \sum_{i=1}^{n-1} d(v_i, u_i)$. Hrany mezi v_i a u_i jsou totiž navzájem různé a každá z nich je součástí řezu C. Ukážeme, že pravá strana je alespoň $d(v_n)$:

$$\begin{aligned}
\sum_{i=1}^{n-1} d(v_i, u_i) &= \sum_{i=1}^{n-1} d(\{v_1 \dots v_i\}, u_i) - d(\{v_1 \dots v_{i-1}\}, u_i) \geq \\
&\geq \sum_{i=1}^{n-1} d(\{v_1 \dots v_i\}, u_i) - d(\{v_1 \dots v_{i-1}\}, u_{i-1}) = \\
&= d(\{v_1 \dots v_{n-1}\}, u_{n-1}) - d(\{v_1 \dots v_0\}, u_0) = \\
&= d(\{v_1 \dots v_{n-1}\}, v_n) - 0 = d(v_n).
\end{aligned}$$

♡

Dokázali jsme, že libovolný řez separující v_{n-1} a v_n je alespoň tak velký jako jednoduchý řez skládající se jen z hran kolem v_n. Když tedy sestavíme nějakou LU posloupnost vrcholů, budeme mít k dispozici jednoduchý minimální řez v_{n-1} a v_n. Následně vytvoříme graf G', v němž v_{n-1} a v_n kontrahujeme. Rekurzivně najdeme minimální řez v G' (sestrojíme nové LU atd.). Hledaný minimální řez poté buďto odděluje vrcholy v_n a v_{n-1}, a potom je řez kolem vrcholu v_n minimální, nebo vrcholy v_n a v_{n-1} neodděluje, a v takovém případě jej najdeme rekurzivně. Hledaný řez je tedy menší z rekurzivně nalezeného řezu a řezu kolem v_n.

Zbývá ukázat, jak konstruovat LU. Postačí hladově: Pamatujeme si $\forall v \neq v_1 \dots v_{i-1}$ hodnotu $d(\{v_1 \dots v_{i-1}\}, v)$, označme ji z_v. V každém kroku vybereme vrchol v s maximální hodnotou z_v, prohlásíme ho za v_i a přepočítáme z_v.

Zde se hodí datová struktura, která dokáže rychle hledat maxima a zvyšovat hodnoty prvků, například Fibonacciho halda. Ta zvládne *DeleteMax* v čase $\mathcal{O}(\log n)$ a *Increase* v $\mathcal{O}(1)$ amortizovaně. Celkem pak náš algoritmus bude mít složitost $\mathcal{O}(n(m + n \log n))$ pro obecné kapacity.

Pokud jsou kapacity malá celá čísla, můžeme využít přihrádkové struktury. Budeme si udržovat obousměrný seznam zatím použitých hodnot z_v, každý prvek

takového seznamu bude obsahovat všechny vrcholy se společnou hodnotou z_v. Když budeme mít seznam seřazený, vybrání minimálního prvku bude znamenat pouze podívat se na první prvek seznamu a z něj odebrat jeden vrchol, případně celý prvek ze seznamu odstranit. Operace *Increase* poté bude reprezentovat pouze přesunutí vrcholu o malý počet přihrádek, případně založení nové přihrádky na správném místě. *DeleteMax* proto bude mít složitost $\mathcal{O}(1)$, všechny *Increase* dohromady $\mathcal{O}(m)$, jelikož za každou hranu přeskakujeme maximálně jednu přihrádku, a celý algoritmus $\mathcal{O}(mn)$.

4. Gomory-Hu Trees

Cílem této kapitoly je popsat datovou strukturu, která velice kompaktně popisuje minimální st-řezy pro všechny dvojice vrcholů s, t v daném neorientovaném grafu. Tuto strukturu poprvé popsali Gomory a Hu v článku [27].

Zatím umíme nalézt minimální st-řez pro zadanou dvojici vrcholů v neorientovaném grafu v čase $\tau = \mathcal{O}(n^{2/3}m)$ pro jednotkové kapacity, $\mathcal{O}(n^2 m)$ pro obecné. Nalézt minimální st-řez pro každou dvojici vrcholů bychom tedy dokázali v čase $\mathcal{O}(n^2\tau)$. Tento výsledek budeme chtít zlepšit.

Značení: Máme-li graf (V, E) a $U \subseteq V$, $\delta(U)$ značí hrany vedoucí mezi U a $\overline{U}$, formálně tedy $\delta(U) = E \cap ((U \times \overline{U}) \cup (\overline{U} \times U))$. Kapacitu řezu $\delta(W)$ budeme značit $d(W)$ a $r(s, t)$ bude kapacita nejmenšího st-řezu.

Pozorování: Minimální řez rozděluje graf jen na dvě komponenty (všimněte si, že pro separátory nic takového neplatí) a každý minimální řez je tím pádem vždy možné zapsat jako $\delta(W)$ pro nějakou množinu $W \subset V$.

Gomory-Hu Tree

Definice: *Gomory-Hu Tree* (dále jen GHT) pro neorientovaný nezáporně ohodnocený graf $G = (V, E)$ je strom $T = (V, F)$ takový, že pro každou hranu $st \in F$ platí: Označíme-li K_1 a K_2 komponenty lesa $T \setminus st$, je $\delta(K_1) = \delta(K_2)$ minimální st-řez. [Pozor, F nemusí být podmnožina původních hran E.]

Další značení: Pro $e \in F$ budeme řezem $\delta(e)$ označovat řez $\delta(K_1) = \delta(K_2)$ a $r(e)$ bude jeho kapacita.

K čemu takový GHT je (existuje-li)? To nám poví následující věta:

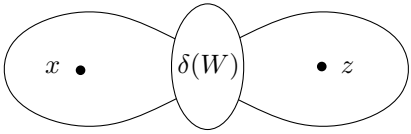
Věta (o využití GHT): Buď T libovolný GHT pro graf G a mějme dva vrcholy s a t. Dále nechť P je cesta v T mezi vrcholy s a t a e je hrana na cestě P s minimálním $r(e)$. Pak $\delta(e)$ je minimální st-řez v G.

Důkaz: Nejprve si dokážeme jedno drobné lemmátko:

Lemmátko: Pro každou trojici vrcholů x, y, z platí, že:

$$r(x, z) \geq \min(r(x, y), r(y, z)).$$

Důkaz: Buď W minimální xz-řez.



Vrchol y musí být v jedné z komponent, Pokud je v komponentě s x, pak $r(y, z) \leq d(W)$, protože $\delta(W)$ je také yz-řez. Pokud v té druhé, analogicky platí $r(x, y) \leq d(W)$. ♡

Zpět k důkazu věty: Chceme dokázat, že $\delta(e)$ je minimální st-řez. To, že je to nějaký řez, plyne z definice GHT. Minimalitu dokážeme indukcí podle délky cesty P:

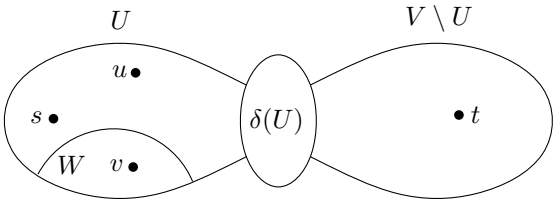
- $|\,P\,| = 1$: Hrana e je v tomto případě přímo st, takže i minimalita plyne z definice GHT.
- $|\,P\,| > 1$: Cesta P spojuje vrcholy s a t, její první hranu označme sx. Naše právě dokázané lemmátko říká, že $r(s,t) \geq \min(r(s,x), r(x,t))$. Určitě je pravda, že $r(s,x) \geq r(e)$, protože e byla hrana cesty P s nejmenším $r(e)$. To, že $r(x,t) \geq r(e)$, plyne z indukčního předpokladu, protože cesta mezi x a t je kratší než cesta P. Dostáváme tak, že $r(s,t) \geq \min(r(s,x), r(x,t)) \geq r(e)$. ♡

Pokud dokážeme GHT sestrojit, nalézt minimální st-řez pro libovolnou dvojici vrcholů dokážeme stejně rychle jako nalézt hranu s nejmenší kapacitou na cestě mezi s a t v GHT. K tomu můžeme použít například Sleator-Tarjanovy stromy, které tuto operaci dokážou provést v amortizovaném čase $\mathcal{O}(\log n)$, nebo můžeme využít toho, že máme spoustu času na předvýpočet, a minimální hrany si pro každou dvojici prostě přichystat předem. Také lze vymyslet redukci na problém nalezení společného předchůdce vrcholů ve stromě (nebude to GHT) a použít jedno z řešení tohoto problému.

Konstrukce GHT

Nyní se naučíme GHT konstruovat, čímž také rozptýlíme obavy o jejich existenci. Nejprve však budeme potřebovat jedno užitečné lemma s hnusně technickým důkazem:

Hnusně technické lemma (HTL): Buďtež s, t vrcholy grafu (V, E), $\delta(U)$ minimální st-řez a $u \neq v$ dva různé vrcholy z U. Pak existuje množina vrcholů $W \subseteq U$ taková, že $\delta(W)$ je minimální uv-řez. ⟨12⟩



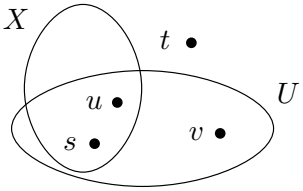
Důkaz: Nechť je $\delta(X)$ minimální uv-řez. BÚNO můžeme předpokládat, že $s \in U$ a $t \notin U$, $u \in X$ a $v \notin X$ a $s \in X$. Pokud by tomu tak nebylo, můžeme vrcholy přeznačit nebo některou z množin nahradit jejím doplňkem.

⟨12⟩ To důležité a netriviální je, že celá W leží v U.

Nyní mohou nastat následující dva případy:

a) $t \notin X$. Tehdy si všimneme, že platí:

$$d(U \cup X) \geq d(U), \tag{1}$$
$$d(U \cap X) + d(U \cup X) \leq d(U) + d(X) \tag{2}$$



První nerovnost plyne z toho, že $\delta(U \cup X)$ je nějaký st-řez, zatímco $\delta(U)$ je minimální st-řez. Druhou dokážeme rozborem případů.

Množinu vrcholů si disjunktně rozdělíme na $X \setminus U$, $X \cap U$, $U \setminus X$ a *ostatní*. Každý z řezů vystupujících v nerovnosti (2) můžeme zapsat jako sjednocení hran mezi některými z těchto skupin vrcholů. Vytvoříme tedy tabulku hran mezi čtyřmi označenými skupinami vrcholů a každému řezu z (2) označíme jemu odpovídající hrany. Protože je graf neorientovaný, stačí nám jen horní trojúhelník tabulky. Pro přehlednosti si označíme $L_1 = \delta(U \cap X), L_2 = \delta(U \cup X), P_1 = \delta(U)$ a $P_2 = \delta(X)$.

	$X \setminus U$	$X \cap U$	$U \setminus X$	*ostatní*
$X \setminus U$	—	L_1, P_1	P_1, P_2	L_2, P_2
$X \cap U$		—	L_1, P_2	L_1, L_2, P_1, P_2
$U \setminus X$			—	L_2, P_1
ostatní				—

Vidíme, že ke každé hraně řezu na levé straně nerovnosti máme vpravo její protějšek a navíc hrany mezi $U \setminus X$ a $X \setminus U$ počítáme jenom vpravo. Nerovnost (2) tedy platí.

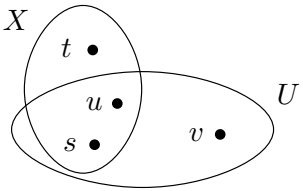
Nyní stačí nerovnosti (2) a (1) odečíst, čímž získáme:

$$d(U \cap X) \leq d(X),$$

což spolu s obrázkem dokazuje, že $\delta(U \cap X)$ je také minimální uv-řez.

b) $t \in X$. Postupovat budeme obdobně jako v předchozím případě. Tentokrát se budou hodit tyto nerovnosti:

$$d(X \setminus U) \geq d(U) \tag{3}$$
$$d(U \setminus X) + d(X \setminus U) \leq d(U) + d(X) \tag{4}$$



První platí proto, že $\delta(X \setminus U)$ je nějaký st-řez, zatímco $\delta(U)$ je minimální st-řez, druhou dokážeme opět důkladným rozborem případů.

Označme $L_1 = \delta(U \setminus X), L_2 = \delta(X \setminus U), P_1 = \delta(U)$ a $P_2 = \delta(X)$ a vytvořme tabulku:

	$X \setminus U$	$X \cap U$	$U \setminus X$	*ostatní*
$X \setminus U$	—	L_2, P_1	L_1, L_2, P_1, P_2	L_2, P_2
$X \cap U$		—	L_1, P_2	P_1, P_2
$U \setminus X$			—	L_1, P_1
ostatní				—

Stejně jako v předchozím případě nerovnost (4) platí. Odečtením (4) a (3) získáme:

$$d(U \setminus X) \leq d(X),$$

z čehož opět dostaneme, že $\delta(U \setminus X)$ je také minimální uv-řez. ♡

Nyní se konečně dostáváme ke konstrukci GHT. Abychom mohli používat indukci, zavedeme si trochu obecnější GHT.

Definice: Mějme neorientovaný graf (V, E). *Částečný Gomory-Hu Tree* (alias ČGHT) pro podmnožinu vrcholů $R \subseteq V$ je dvojice $((R, F), C)$, kde (R, F) je strom a množina $C = \{C(r) \mid r \in R\}$ je rozklad množiny vrcholů V. Tento rozklad nám říká, k jakým vrcholům ČGHT máme přilepit které vrcholy původního grafu. Navíc musí platit, že:

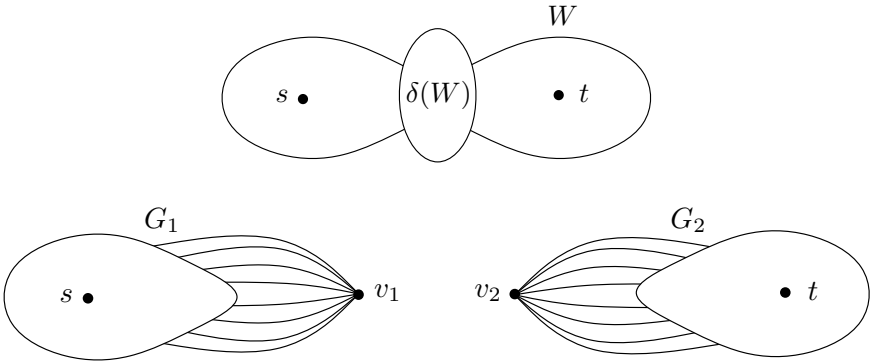
1. $\forall r : r \in C(r)$, neboli každý vrchol ČGHT je přilepen sám k sobě, a
2. $\forall st \in F : \delta\left(\bigcup_{r \in K_1} C(r)\right) = \delta\left(\bigcup_{r \in K_2} C(r)\right)$ je minimální st-řez, kde K_1 a K_2 jsou komponenty $(R, F) \setminus st$.

Věta (o existenci ČGHT): Buď (V, E) neorientovaný nezáporně ohodnocený graf. Pro každou podmnožinu vrcholů R existuje ČGHT.

Důkaz: Dokážeme indukcí podle velikosti množiny R.

- $|R| = 1$: ČGHT má jediný vrchol $r \in R$ a $C(r) = V$.
- $|R| > 1$: Najdeme dvojici vrcholů $s, t \in R$ takovou, že jejich minimální st-řez $\delta(W)$ je nejmenší možný. Nyní vytvoříme graf G_1 z grafu G kontrahováním všech vrcholů množiny W do jednoho vrcholu, který označíme v_1, a vytvoříme graf G_2 z G kontrahováním všech vrcholů z $\overline{W}$ do jednoho vrcholu v_2.⟨13⟩

⟨13⟩ Proč to děláme „tak složitě“ a přidáváme do G_1 vrchol v_1? Na první pohled to přeci vypadá zbytečně. Problém je v tom, že i když dle HTL leží všechny minimální řezy oddělující vrcholy z W v množině vrcholů W, *hrany* těchto řezů celé v podgrafu indukovaném W ležet nemusí. K těmto řezům totiž patří i hrany, které mají ve W jenom jeden konec. Proto jsme do G_1 přidali v_1 – do něj vedou všechny zajímavé hrany, které mají ve W jeden konec. Tím *zajímavé* myslíme to, že z každého vrcholu $w \in W$ vede do v_1 *nejlevnější* hrana, která z něj vedla do množiny $V \setminus W$, případně žádná, pokud do této množiny žádná hrana nevedla.



Dále vytvoříme množiny vrcholů $R_1 = R \cap \overline{W}$ a $R_2 = R \cap W$. Dle indukčního předpokladu (R_1 i R_2 jsou menší než R) existuje ČGHT $T_1 = ((R_1, F_1), C_1)$ pro R_1 na G_1 a $T_2 = ((R_2, F_2), C_2)$ pro R_2 na G_2.
Nyní vytvoříme ČGHT pro původní graf. Označme r_1 ten vrchol R_1, pro který je $v_1 \in C_1(r_1)$, a obdobně r_2. Oba ČGHT T_1 a T_2 spojíme hranou r_1r_2, takže ČGHT pro G bude $T = ((R_1 \cup R_2, F_1 \cup F_2 \cup r_1r_2), C)$. Třídy rozkladu C zvolíme tak, že pro $r \in R_1$ bude $C(r) = C_1(r) \setminus \{v_1\}$ a pro $r \in R_2$ bude $C(r) = C_2(r) \setminus \{v_2\}$ [odebrali jsme vrcholy v_1 a v_2 z rozkladu C].
Chceme ukázat, že tento T je opravdu ČGHT. C je určitě rozklad všech vrcholů a každé $r \in C(r)$ z indukčního předpokladu, takže podmínka 1 je splněna. Co se týče podmínky 2, tak:

- pro hranu r_1r_2 je $\delta(W)$ určitě minimální r_1r_2-řez, protože řez mezi s a t je současně i r_1r_2-řezem a je ze všech možných minimálních řezů na R nejmenší,
- pro hranu $e \neq r_1r_2$ je $\delta(e)$ z indukce minimální řez na jednom z grafů G_1, G_2. Tento řez také přesně odpovídá řezu v grafu G, protože v G_1 i v G_2 jsme počítali s hranami vedoucími do v_1, v_2 a protože jsme ČGHT napojili přes vrcholy, k nimž byly v_1 a v_2 přilepeny.
  HTL nám navíc říká, že existuje minimální řez, který žije pouze v příslušném z grafů G_1, G_2, takže nalezený řez je minimální pro celý graf G. ♡

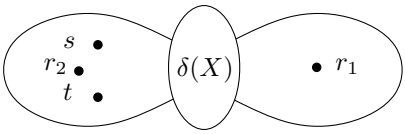
Nyní víme, že GHT existují, a také víme, jak by se daly konstruovat. Nicméně nalezení vrcholů s, t tak, aby byl minimální st-řez nejmenší možný, je časově náročné. Proto si poslední větu ještě o něco vylepšíme.

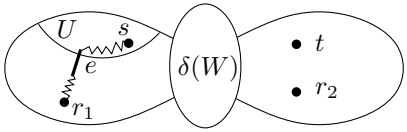
Vylepšení věty o existenci ČGHT: Na začátku důkazu není nutné hledat vrcholy s a t takové, aby byl minimální st-řez nejmenší možný. Stačí zvolit *libovolné* vrcholy $s, t \in R$ a zvolit $\delta(W)$ jako minimální st-řez.

Důkaz: Nejprve si uvědomme, proč jsme v předchozím důkazu potřebovali, aby byl $\delta(W)$ nejmenší ze všech možných st-řezů. Bylo to jenom proto, že jsme jím v ČGHT nakonec separovali vrcholy r_1 a r_2 a potřebovali jsme záruku, aby byl $\delta(W)$ opravdu minimální r_1r_2-řez. Nyní musíme ukázat, že námi nalezený st-řez $\delta(W)$ je také minimálním r_1r_2-řezem.

Pro spor tedy předpokládejme, že nějaký r_1r_2-řez $\delta(X)$ má menší kapacitu než $\delta(W)$. Navíc vezměme ten případ, kdy se to stalo „poprvé“, takže pro každé menší R je všechno v pořádku (to můžeme, protože pro $|R| = 1$ všechno v pořádku bylo).

Protože $\delta(W)$ je minimální st-řez a $\delta(X)$ má menší kapacitu, $\delta(X)$ nemůže separovat s a t. Přitom ale separuje r_1 a r_2, takže musí separovat buď s a r_1, nebo t a r_2. BÚNO nechť X separuje s a r_1.





Podívejme se nyní na ČGHT T_1 (víme, že ten je korektní) a nalezněme v něm nejlevnější hranu e na cestě spojující s a r_1. Tato hrana definuje řez $\delta(U)$, což je minimální sr_1-řez, podle HTL i v celém G. Protože $\delta(X)$ je sr_1-řez, je $d(U) \leq d(X) < d(W)$. Teď si stačí uvědomit, že $v_1 \in C(r_1)$, takže $\delta(U)$ separuje nejenom s a r_1, ale také s a v_1. Tím pádem ale separuje také s a t. To je spor, protože $d(U) < d(W)$, a přitom $\delta(W)$ měl být minimální. ♡

Teď už dokážeme GHT konstruovat efektivně – v každém kroku vybereme dva vrcholy s a t, nalezneme v čase $\mathcal{O}(\tau)$ minimální st-řez a výsledné komponenty s přidanými v_1, v_2 zpracujeme rekurzivně. Celou výstavbu tedy zvládneme v čase $\mathcal{O}(n\tau)$, čili $\mathcal{O}(n^{5/3}m)$ pro neohodnocené grafy.

5. Minimální kostry

Tato kapitola uvede problém minimální kostry, základní věty o kostrách a klasické algoritmy na hledání minimálních koster. Budeme se inspirovat Tarjanovým přístupem z knihy [51]. Všechny grafy v této kapitole budou neorientované multigrafy a jejich hrany budou ohodnoceny vahami $w : E \to \mathbb{R}$.

Minimální kostry a jejich vlastnosti

Definice:

- *Podgrafem* budeme v této kapitole mínit libovolnou podmnožinu hran, vrcholy vždy zůstanou zachovány.
- *Přidání a odebrání hrany* budeme značit $T+e := T \cup \{e\}$, $T-e := T \setminus \{e\}$.
- *Kostra* (Spanning Tree) souvislého grafu G je libovolný jeho podgraf, který je strom. Kostru nesouvislého grafu definujeme jako sjednocení koster jednotlivých komponent. [Alternativně: kostra je minimální podgraf, který má komponenty s týmiž vrcholy jako komponenty G.]
- *Váha* podgrafu $F \subseteq E$ je $w(F) := \sum_{e \in F} w(e)$.
- *Minimální kostra* (Minimum Spanning Tree, mezi přáteli též MST) budeme říkat každé kostře, jejíž váha je mezi všemi kostrami daného grafu minimální.

Toto je sice standardní definice MST, ale jinak je dosti nešikovná, protože vyžaduje, aby bylo váhy možné sčítat. Ukážeme, že to není potřeba.

Definice: Buď $T \subseteq G$ nějaká kostra grafu G. Pak:

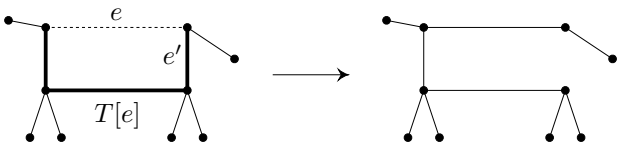
- $T[x, y]$ bude značit cestu v T, která spojuje x a y. (Cestou opět míníme množinu hran.)
- $T[e] := T[x, y]$ pro hranu $e = xy$. Této cestě budeme říkat *cesta pokrytá hranou e.*
- Hrana $e \in E \setminus T$ je *lehká vzhledem k T* $\equiv \exists e' \in T[e] : w(e) < w(e')$. Ostatním hranám neležícím v kostře budeme říkat *těžké.*

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

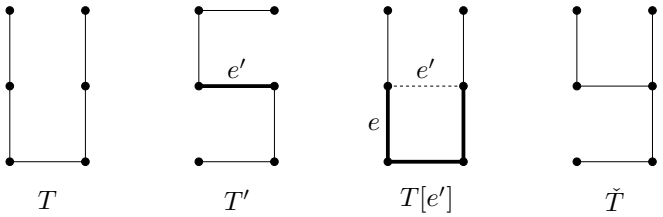
Tato věta nám dává pěknou alternativní definici MST, která místo sčítání vah váhy pouze porovnává, čili jí místo čísel stačí lineární (kvazi)uspořádání na hranách. Než se dostaneme k jejímu důkazu, prozkoumejme nejdříve, jak se dá mezi jednotlivými kostrami přecházet.

Definice: Pro kostru T a hrany e, e' zaveďme $swap(T, e, e') := T - e + e'$.

Pozorování: Pokud $e' \notin T$ a $e \in T[e']$, je $swap(T, e, e')$ opět kostra. Stačí si uvědomit, že přidáním e' do T vznikne kružnice (konkrétně $T[e'] + e'$) a vynecháním libovolné hrany z této kružnice získáme opět kostru.



Kostra T, cesta $T[e]$ a výsledek operace $swap(T, e', e)$



Jeden krok důkazu swapovacího lemmatu

Lemma o swapování: Máme-li libovolné kostry T a T', pak lze z T dostat T' konečným počtem operací *swap*.

Důkaz: Pokud $T \neq T'$, musí existovat hrana $e' \in T' \setminus T$, protože $|T| = |T'|$. Kružnice $T[e'] + e'$ nemůže být celá obsažena v T', takže existuje hrana $e \in T[e'] \setminus T'$ a $\check{T} := swap(T, e, e')$ je kostra, pro kterou $|\check{T} \,\Delta\, T'| = |T \,\Delta\, T'| - 2$. Po konečném počtu těchto kroků tedy musíme dojít k T'. ♡

Monotónní lemma o swapování: Je-li T kostra, k níž neexistují žádné lehké hrany, a T' libovolná kostra, pak lze od T k T' přejít posloupností swapů, při které váha kostry neklesá.

Důkaz: Podobně jako u předchozího lemmatu budeme postupovat indukcí podle $|T \,\Delta\, T'|$. Pokud zvolíme libovolně hranu $e' \in T' \setminus T$ a k ní $e \in T[e'] \setminus T'$, musí $\check{T} := swap(T, e, e')$ být kostra bližší k T' a $w(\check{T}) \geq w(T)$, jelikož e' nemůže být lehká vzhledem k T, takže speciálně $w(e') \geq w(e)$.

Aby mohla indukce pokračovat, potřebujeme ještě dokázat, že ani k nové kostře neexistují lehké hrany v $T' \setminus \check{T}$. K tomu nám pomůže zvolit si ze všech možných hran e' tu s nejmenší vahou. Uvažme nyní hranu $f \in T' \setminus \check{T}$. Cesta $\check{T}[f]$ pokrytá touto hranou v nové kostře je buďto původní $T[f]$ (to pokud $e \notin T[f]$) nebo $T[f] \,\Delta\, C$, kde C je kružnice $T[e'] + e'$. První případ je triviální, ve druhém si stačí uvědomit, že $w(f) \geq w(e')$ a ostatní hrany na C jsou lehčí než e'. ♡

Důkaz věty:

- $\exists$ lehká hrana $\Rightarrow$ T není minimální.
  Nechť $\exists e$ lehká. Najdeme $e' \in T[e] : w(e) < w(e')$ (ta musí existovat z definice lehké hrany). Kostra $T' := swap(T, e, e')$ je lehčí než T.

- K T neexistuje lehká hrana $\Rightarrow$ T je minimální.

Uvažme nějakou minimální kostru T_{min} a použijme monotónní swapovací lemma na T a T_{min}. Z něj plyne $w(T) \leq w(T_{min})$, a tedy $w(T) = w(T_{min})$. ♡

Věta: Jsou-li všechny váhy hran navzájem různé, je MST určena jednoznačně.

Důkaz: Máme-li dvě MST T_1 a T_2, neobsahují podle předchozí věty lehké hrany, takže podle monotónního lemmatu mezi nimi lze přeswapovat bez poklesu váhy. Pokud jsou ale váhy různé, musí každé swapnutí ostře zvýšit váhu, a proto k žádnému nemohlo dojít. ♡

Poznámka: Často se nám bude hodit, aby kostra, kterou hledáme, byla určena jednoznačně. Tehdy můžeme využít předchozí věty a váhy změnit o vhodné epsilony, respektive kvaziuspořádání rozšířit na lineární uspořádání.

Červenomodrý meta-algoritmus

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.

Meta-algoritmus:

1. Na počátku jsou všechny hrany bezbarvé.
2. Dokud to lze, použijeme jedno z následujících pravidel:
3. *Modré pravidlo:* Vyber řez takový, že jeho nejlehčí hrana není modrá,[14] a obarvi ji na modro.
4. *Červené pravidlo:* Vyber cyklus takový, že jeho nejtěžší hrana není červená, a obarvi ji na červeno.

Věta: Pro Červenomodrý meta-algoritmus spuštěný na libovolném grafu s hranami lineárně uspořádanými podle vah platí:

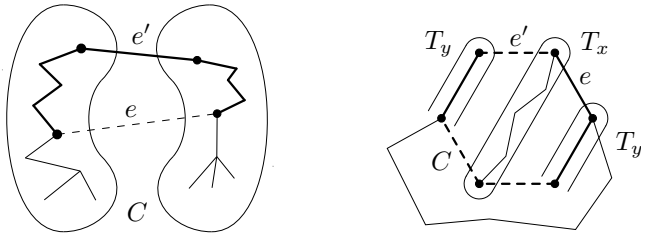
1. Vždy se zastaví.
2. Po zastavení jsou všechny hrany obarvené.
3. Modře obarvené hrany tvoří minimální kostru.

Důkaz: Nejdříve si dokážeme několik lemmat. Jelikož hrany mají navzájem různé váhy, můžeme předpokládat, že algoritmus má sestrojit jednu konkrétní minimální kostru T_{min}.

Modré lemma: Je-li libovolná hrana e algoritmem kdykoliv obarvena na modro, pak $e \in T_{min}$.

Důkaz: Sporem: Hrana e byla omodřena jako nejlehčí hrana nějakého řezu C. Pokud $e \notin T_{min}$, musí cesta $T_{min}[e]$ obsahovat nějakou jinou hranu e' řezu C. Jenže e' je těžší než e, takže operací $swap(T_{min}, e', e)$ získáme ještě lehčí kostru, což není možné. ♡

[14] Za touto podmínkou nehledejte žádná kouzla, je tu pouze proto, aby se algoritmus nemohl zacyklit neustálým prováděním pravidel, která nic nezmění.



Situace v důkazu Modrého a Červeného lemmatu

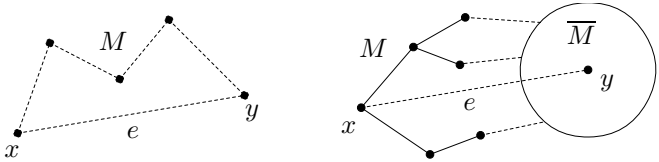
Červené lemma: Je-li libovolná hrana e algoritmem kdykoliv obarvena na červeno, pak $e \notin T_{min}$.

Důkaz: Opět sporem: Předpokládejme, že e byla obarvena červeně jako nejtěžší na nějaké kružnici C a že $e \in T_{min}$. Odebráním e se nám T_{min} rozpadne na dvě komponenty T_x a T_y. Některé vrcholy kružnice připadnou do komponenty T_x, ostatní do T_y. Na C ale musí existovat nějaká hrana $e' \neq e$, jejíž krajní vrcholy leží v různých komponentách, a jelikož hrana e byla na kružnici nejtěžší, je $w(e') < w(e)$. Pomocí $swap(T_{min}, e, e')$ proto získáme lehčí kostru, a to je spor. ♡

Bezbarvé lemma: Pokud existuje nějaká neobarvená hrana, lze ještě použít některé z pravidel.

Důkaz: Nechť existuje hrana $e = xy$, která je stále bezbarvá. Označíme si M množinu vrcholů, do nichž se lze z x dostat po modrých hranách. Nyní mohou nastat dvě možnosti:

- $y \in M$ (tj. existuje modrá cesta z x do y): Modrá cesta je v minimální kostře a k minimální kostře neexistují žádné lehké hrany, takže hrana e je nejdražší na cyklu tvořeném modrou cestou a touto hranou a mohu na ni použít červené pravidlo.



Situace v důkazu Bezbarvého lemmatu

- $y \notin M$: Tehdy řez $\delta(M)$ neobsahuje žádné modré hrany, takže na tento řez můžeme použít modré pravidlo. ♡

Důkaz věty:

- *Zastaví se:* Z červeného a modrého lemmatu plyne, že žádnou hranu nikdy nepřebarvíme. Každým krokem přibude alespoň jedna obarvená hrana, takže se algoritmus po nejvýše m krocích zastaví.

- *Obarví vše:* Pokud existuje alespoň jedna neobarvená hrana, pak podle bezbarvého lemmatu algoritmus pokračuje.
- *Najde modrou MST:* Podle červeného a modrého lemmatu leží v T_{min} právě modré hrany. ♡

Poznámka: Červené a modré pravidlo jsou v jistém smyslu duální. Pro rovinné grafy je na sebe převede obyčejná rovinná dualita (stačí si uvědomit, že kostra duálního grafu je komplement duálu kostry primárního grafu), obecněji je to dualita mezi matroidy, která prohazuje řezy a cykly.

Klasické algoritmy na hledání MST

Kruskalův neboli Hladový:[15]

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. Pro každou hranu e se podíváme, zda spojuje dvě různé komponenty – pokud ano, přidáme ji do kostry, jinak ji zahodíme.

Červenomodrý pohled: pěstujeme modrý les. Pokud hrana spojuje dva stromečky, je určitě minimální v řezu mezi jedním ze stromečků a zbytkem grafu (ostatní hrany téhož řezu jsme ještě nezpracovali). Pokud nespojuje, je maximální na nějakém cyklu tvořeném touto hranou a nějakými dříve přidanými.

Potřebujeme čas $\mathcal{O}(m \log n)$ na setřídění hran a dále datovou strukturu pro udržování komponent souvislosti (Union-Find Problem), se kterou provedeme m operací *Find* a n operací *Union*. Nejlepší známá implementace této struktury dává složitost obou operací $\mathcal{O}(\alpha(n))$ amortizovaně, kde α je inverzní Ackermannova funkce. Celkově tedy hladový algoritmus doběhne v čase $\mathcal{O}(m \log n + m\alpha(n))$.

Borůvkův:

Opět si budeme pěstovat modrý les, avšak tentokrát jej budeme rozšiřovat ve fázích. V jedné fázi nalezneme ke každému stromečku nejlevnější incidentní hranu a všechny tyto nalezené hrany naráz přidáme (aplikujeme několik modrých pravidel najednou). Pokud jsou všechny váhy různé, cyklus tím nevznikne.

Počet stromečků klesá exponenciálně $\Rightarrow$ fází je celkem $\log n$. Pokud každou fázi implementujeme lineárním průchodem celého grafu, dostaneme složitost $\mathcal{O}(m \log n)$. Mimo to lze každou fázi výtečně paralelizovat.

Jarníkův:

Jarníkův algoritmus je podobný Borůvkovi, ale s tím rozdílem, že nenecháme růst celý les, ale jen jeden modrý strom. V každém okamžiku nalezneme nejlevnější hranu vedoucí mezi stromem a zbytkem grafu a přidáme ji ke stromu (modré pravidlo); hrany vedoucí uvnitř stromu průběžně zahazujeme (červené pravidlo). Kroky

[15] Možná hladový s malým ‘h’, ale tento algoritmus je pradědečkem všech ostatních hladových algoritmů, tak mu tu čest přejme.

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

Cvičení: Nalezněte jednoduchý algoritmus pro výpočet MST v grafech ohodnocených vahami $\{1, \ldots k\}$ se složitostí $\mathcal{O}(mk)$ nebo dokonce $\mathcal{O}(m + nk)$.

6. Rychlejší algoritmy na minimální kostry

V této kapitole popíšeme několik pokročilejších algoritmů pro problém minimální kostry. Vesměs to budou různá vylepšení klasických algoritmů z minulé kapitoly.

Upravená verze Borůvkova algoritmu pro rovinné grafy

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. To funguje obecně, ale ukážeme, že pro rovinné grafy tak dosáhneme lineární časové složitosti.

Pozorování: Pokud $F \subseteq \mathrm{MST}(G)$ (kde $\mathrm{MST}(G)$ je minimální kostra grafu G), G' je graf vzniklý z G kontrakcí podél hran z F, pak kostra grafu G, která vznikne z $\mathrm{MST}(G')$ zpětným expandováním kontrahovaných vrcholů, je $\mathrm{MST}(G)$. Pokud kontrakcí vzniknou smyčky, můžeme je ihned odstraňovat; pokud paralelní hrany, ponecháme z nich vždy tu nejlehčí. To nás vede k následujícímu algoritmu:

Algoritmus: MST v rovinných grafech [40]

1. Ke každému vrcholu najdeme nejlevnější incidentní hranu – dostaneme množinu hran $F \subseteq E$.
2. Graf kontrahujeme podle F následovně:
3. Prohledáme do šířky graf (V, F) a přiřadíme každému vrcholu číslo komponenty, v níž se nachází.
4. Přečíslujeme hrany v G podle čísel komponent.
5. Odstraníme násobné hrany:
6. Setřídíme hrany lexikograficky přihrádkovým tříděním (násobné hrany jsou nyní pospolu).
7. Projdeme posloupnost hran a z každého úseku multihran odstraníme všechny až na nejlevnější hranu. Také odstraníme smyčky.
8. Pokud stále máme netriviální graf, opakujeme předchozí kroky.
9. Vrátíme jako MST všechny hrany, které se v průběhu algoritmu dostaly do F.

Časová složitost: Označme si n_i a m_i počet vrcholů a hran na počátku i-té iterace. Každý z kroků 1–7 trvá $\mathcal{O}(m_i)$, proto i celý cyklus algoritmu trvá $\mathcal{O}(m_i)$. Počet vrcholů grafu klesá s každým cyklem exponenciálně: $n_i \leq n/2^i$. Na začátku každého cyklu je graf rovinný (kontrakcí hrany v rovinném grafu se rovinnost zachovává) a není to multigraf, takže počet jeho hran je lineární v počtu vrcholů: $m_i < 3n_i$. Celkovou časovou složitost dostaneme jako součet dob trvání všech cyklů: $\mathcal{O}(\sum_i m_i) = \mathcal{O}(\sum_i n_i) = \mathcal{O}(n)$.

Minorově uzavřené třídy

Předchozí algoritmus ve skutečnosti funguje v lineárním čase i pro obecnější třídy grafů, než jsou grafy rovinné. Tím správným universem jsou minorově uzavřené třídy:

Definice: 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).

Pozorování: Relace $\preceq$ je částečné uspořádání na třídě všech grafů (reflexivita a tranzitivita plynou přímo z definice, pro antisymetrii si stačí všimnout, že jak mazáním, tak kontrakcemi klesá součet počtu vrcholů s počtem hran).

Pozorování: Pokud je graf H podgrafem grafu G, pak je také jeho minorem. Opačně to neplatí: například C_3 je minorem libovolné delší kružnice, ale určitě ne jejím podgrafem.

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

Příklady: Třída všech grafů a prázdná třída jsou jistě minorově uzavřené. Těm říkáme *triviální* třídy. Mezi ty netriviální patří třeba lesy, rovinné grafy, grafy nakreslitelné na další plochy, grafy nakreslitelné do $\mathbb{R}^3$ tak, že žádná kružnice netvoří uzel.

Definice: 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 $\mathrm{Forb}(\{G_1, \ldots, G_k\})$.

Příklady: $\mathrm{Forb}(K_2)$ je třída všech grafů, které neobsahují žádnou hranu. $\mathrm{Forb}(C_3)$ je třída všech lesů. $\mathrm{Forb}(K_5, K_{3,3})$ jsou všechny rovinné grafy – to je minorová analogie Kuratowského věty (pokud graf G obsahuje dělení grafu H, pak je $H \preceq G$; opačně to obecně není pravda, ale zrovna pro dvojici K_5 a $K_{3,3}$ to funguje; zkuste si sami).

Lze každou minorově uzavřenou třídu $\mathcal{C}$ zapsat jako $\mathrm{Forb}(\mathcal{F})$ pro nějakou třídu $\mathcal{F}$ zakázaných minorů? Zajisté ano, stačí například zvolit $\mathcal{F}$ jako doplněk třídy $\mathcal{C}$. Slavná Robertsonova-Seymourova věta [46] dokonce zaručuje existenci *konečné* třídy zakázaných minorů. To není samo sebou, dokazuje se to dosti obtížně (a je to jedna z nejslavnějších kombinatorických vět za posledních mnoho let), ale plyne z toho spousta zajímavých důsledků. My si vystačíme s daleko jednoduššími prostředky a zájemce o hlubší studium minorové teorie odkážeme na Diestelovu knihu [18].

Ve zbytku této sekce dokážeme následující větu, která je zobecněním klasického tvrzení o hustotě rovinných grafů na minorově uzavřené třídy:

Definice: *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 (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.

Důsledek: 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.

Důkaz věty: 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 $G \notin \mathrm{Forb}(K_k)$. 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 $\mathcal{C} \subseteq \mathrm{Forb}(K_k)$. Proto musí platit $\varrho(\mathcal{C}) \leq \varrho(\mathrm{Forb}(K_k))$. Takže postačuje omezit hustotu tříd s jedním zakázaným minorem, který je úplným grafem, a to plyne z následující Maderovy věty. ♡

Věta (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.

Důkaz: Viz Diestel [18].

Jarníkův algoritmus s Fibonacciho haldou

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.

Algoritmus: Jarníkův algoritmus #2 (Fredman, Tarjan [25])

1. Začneme libovolným vrcholem v_0, $T \leftarrow \{v_0\}$.
2. Do haldy H umístíme všechny hrany vedoucí z v_0.
3. Opakujeme, dokud $H \neq \emptyset$:
4. $\quad vw \leftarrow \mathit{ExtractMin}(H)$, přičemž $v \notin T, w \in T$.
5. $\quad T \leftarrow T \cup \{vw\}$
6. $\quad$ Pro všechny sousedy u vrcholu v, kteří dosud nejsou v T, upravíme haldu:
7. $\quad\quad$ Pokud ještě v H není hrana incidentní s u, přidáme hranu uv.
8. $\quad\quad$ Pokud už tam nějaká taková hrana je a je-li těžší než uv, nahradíme ji hranou uv a provedeme *DecreaseKey*.

Správnost algoritmu přímo plyne ze správnosti Jarníkova algoritmu.

Časová složitost: Složitost tohoto algoritmu bude $\mathcal{O}(m+n \log n)$, neboť vnější cyklus se provede nanejvýš n-krát, za *ExtractMin* v něm tedy zaplatíme celkem $\mathcal{O}(n \log n)$, za přidávání vrcholů do H a nalézání nejlevnějších hran zaplatíme celkem $\mathcal{O}(m)$ (na každou hranu takto sáhneme nanejvýš dvakrát), za snižování vah vrcholů v haldě rovněž pouze $\mathcal{O}(m)$ (nanejvýš m-krát provedu porovnání vah a *DecreaseKey* v kroku 8 za $\mathcal{O}(1)$).

Toto zlepšení je důležitější, než by se mohlo zdát, protože nám pro grafy s mnoha hranami (konkrétně pro grafy s $m = \Omega(n \log n)$) dává lineární algoritmus.

Kombinace Jarníkova a Borůvkova algoritmu

K dalšímu zlepšení dojde, když před předchozím algoritmem spustíme $\log\log n$ cyklů Borůvkova algoritmu s kontrahováním vrcholů, čímž graf zahustíme.

Algoritmus: Jarníkův algoritmus #3 (původ neznámý)

1. Provedeme $\log\log n$ cyklů upraveného Borůvkova algoritmu s kontrahováním hran popsaného výše.
2. Pokračujeme Jarníkovým algoritmem #2.

Časová složitost: Složitost první části je $\mathcal{O}(m \log\log n)$. Počet vrcholů se po první části algoritmu sníží na $n' \leq n/\log n$ a složitost druhé části bude tedy nanejvýš $\mathcal{O}(m + n' \log n') = \mathcal{O}(m)$.

Jarníkův algoritmus s omezením velikosti haldy

Ještě většího zrychlení dosáhneme, omezíme-li Jarníkovu algoritmu #2 vhodně velikost haldy, takže nám nalezne jednotlivé podkostřičky zastavené v růstu přetečením haldy. Podle těchto podkostřiček graf následně skontrahujeme a budeme pokračovat s mnohem menším grafem.

Algoritmus: Jarníkův algoritmus #4 (Fredman, Tarjan [25])

1. Opakujeme, dokud máme netriviální G (s alespoň jednou hranou):
2. $t \leftarrow |V(G)|$.
3. Zvolíme $k \leftarrow 2^{2m/t}$ (velikost haldy).
4. $T \leftarrow \emptyset$.
5. Opakujeme, dokud existují vrcholy mimo T:
6. Najdeme vrchol v_0 mimo T.
7. Spustíme Jarníkův alg. #2 pro celý graf od v_0. Zastavíme ho, pokud:
8. $|H| \geq k$ (byla překročena velikost haldy) nebo
9. $H = \emptyset$ (došli sousedé) nebo
10. do T jsme přidali hranu oboustranně incidentní s hranami v T (připojili jsme novou podkostru k nějaké už nalezené).
11. Kontrahujeme G podle podkoster nalezených v T.

Pozorování: Pokud algoritmus ještě neskončil, je každá z nalezených podkoster v T incidentní s alespoň k hranami (do toho počítáme i vnitřní hrany vedoucí mezi vrcholy podkostry). Jak to vypadá pro jednotlivá ukončení:

8. $|H| \geq k$ – všechny hrany v haldě jsou incidentní s T a navzájem různé, takže incidentních je dost.
9. $H = \emptyset$ – nemůže nastat, algoritmus by skončil.
10. Připojím se k už existující podkostře – jen ji zvětším.

Časová složitost: Důsledkem předchozího pozorování je, že počet podkoster v jednom průchodu je nanejvýš $2m/k$. Pro t' a k' v následujícím kroku potom platí $t' \leq 2m/k$

a $k' = 2^{2m/t'} \geq 2^k$. Průchodů bude tedy nanejvýš $\log^* n$[16], protože průchod s $k > n$ bude už určitě poslední. Přitom jeden vnější průchod trvá $\mathcal{O}(m + t \log k)$, což je pro $k = 2^{2m/t}$ rovno $\mathcal{O}(m)$. Celkově tedy algoritmus poběží v čase $\mathcal{O}(m \log^* n)$.

I odhad $\log^* n$ je ale příliš hrubý, protože nezačínáme s haldou velikosti 1, nýbrž $2^{2m/n}$. Můžeme tedy počet průchodů přesněji omezit funkcí $\beta(m,n) = \min\{i : \log^{(i)} n < m/n\}$ a časovou složitost odhadnout jako $\mathcal{O}(m\beta(m,n))$. To nám dává lineární algoritmus pro grafy s $m \geq n \log^{(k)} n$ pro libovolnou konstantu k, jelikož $\beta(m,n)$ tehdy vyjde omezená konstantou.

Další výsledky

- $\mathcal{O}(m\alpha(m,n))$, kde $\alpha(m,n)$ je obdoba inverzní Ackermannovy funkce definovaná podobně, jako je $\beta(m,n)$ obdobou $\log^*$. [11], [43]
- $\mathcal{O}(\mathcal{T}(m,n))$, kde $\mathcal{T}(m,n)$ je hloubka optimálního rozhodovacího stromu pro nalezení minimální kostry v grafech s patřičným počtem hran a vrcholů [44]. Jelikož každý deterministický algoritmus založený na porovnávání vah lze popsat rozhodovacím stromem, je tento algoritmus zaručeně optimální. Jen bohužel nevíme, jak optimální stromy vypadají, takže je stále otevřeno, zda lze MST nalézt v lineárním čase. Nicméně tento algoritmus pracuje i na Pointer Machine, pročež víme, že pokud je lineární složitosti možné dosáhnout, není k tomu potřeba výpočetní síla RAMu.[17]
- $\mathcal{O}(m)$ pro grafy s celočíselnými vahami (na RAMu) [24] – ukážeme v jedné z následujících kapitol.
- $\mathcal{O}(m)$, pokud už máme hrany setříděné podle vah: jelikož víme, že záleží jen na uspořádání, můžeme váhy přečíslovat na $1 \ldots m$ a použít předchozí algoritmus.
- $\mathcal{O}(m)$ průměrně: randomizovaný algoritmus, který pro libovolný vstupní graf doběhne v očekávaném lineárním čase [33].
- Na zjištění, zda je zadaná kostra minimální, stačí $\mathcal{O}(m)$ porovnání [37] a dokonce lze v lineárním čase zjistit, která to jsou [36]. Z toho ostatně vychází předchozí randomizovaný algoritmus.

[16] $\log^* n$ je inverzní funkce k „věži z mocnin“, čili $\min\{i : \log^{(i)} n < 1\}$, kde $\log^{(i)} n$ je i-krát iterovaný logaritmus.

[17] O výpočetních modelech viz příští kapitola.

7. Výpočetní modely

Když jsme v předešlých kapitolách studovali algoritmy, nezabývali jsme se tím, v jakém přesně výpočetním modelu pracujeme. Konstrukce, které jsme používali, totiž fungovaly ve všech obvyklých modelech a měly tam stejnou časovou i prostorovu složitost. Ne vždy tomu tak je, takže se výpočetním modelům podívejme na zoubek trochu blíže.

Druhy výpočetních modelů

Obvykle se používají následující dva modely, které se liší zejména v tom, zda je možné paměť indexovat v konstantním čase či nikoliv.

Pointer Machine (PM) [6] pracuje se dvěma typy dat: *čísly* v pevně omezeném rozsahu a *pointery,* které slouží k odkazování na data uložená v paměti. Paměť tohoto modelu je složená z pevného počtu registrů na čísla a na pointery a z neomezeného počtu *krabiček.* Každá krabička má pevný počet položek na čísla a pointery. Na krabičku se lze odkázat pouze pointerem.

Aritmetika v tomto modelu (až na triviální případy) nefunguje v konstantním čase, datové struktury popsatelné pomocí pointerů (seznamy, stromy ...) fungují přímočaře, ovšem pole musíme reprezentovat stromem, takže indexování stojí $\Theta(\log n)$.

Random Access Machine (RAM) [16] je rodinka modelů, které mají společné to, že pracují výhradně s (přirozenými) čísly a ukládají je do paměti indexované opět čísly. Instrukce v programu (podobné assembleru) pracují s operandy, které jsou buď konstanty nebo buňky paměti adresované přímo (číslem buňky), případně nepřímo (index je uložen v nějaké buňce adresované přímo). Je vidět, že tento model je alespoň tak silný jako PM, protože odkazy pomocí pointerů lze vždy nahradit indexováním.

Pokud ovšem povolíme počítat s libovolně velkými čísly v konstantním čase, dostaneme velice silný paralelní počítač, na němž spočítáme téměř vše v konstantním čase (modulo kódování vstupu). Proto musíme model nějak omezit, aby byl realistický, a to lze udělat více způsoby:

- *Zavést logaritmickou cenu instrukcí* – operace trvá tak dlouho, kolik je počet bitů čísel, s nimiž pracuje, a to včetně adres v paměti. Elegantně odstraní absurdity, ale je dost těžké odhadovat časové složitosti; u většiny normálních algoritmů nakonec po dlouhém počítání vyjde, že mají složitost $\Theta(\log n)$-krát větší než v neomezeném RAMu.
- *Omezit velikost čísel* na nějaký pevný počet bitů (budeme mu říkat *šířka slova* a značit w) a operace ponechat v čase $\mathcal{O}(1)$. Abychom mohli alespoň adresovat vstup, musí být $w \geq \log N$, kde N je celková velikost vstupu. Jelikož aritmetiku s $\mathcal{O}(1)$-násobnou přesností lze simulovat s konstantním zpomalením, můžeme předpokládat, že $w = \Omega(\log N)$, tedy že lze přímo pracovat s čísly polynomiálně velkými vzhledem k N. Ještě bychom si měli ujasnit, jakou množinu operací povolíme:
  - *Word-RAM* – „céčkové“ operace: $+$, $-$, $*$, $/$, mod (aritmetika); $<<$, $>>$ (bitové posuvy); $\wedge$, $\vee$, $\oplus$, $\neg$ (bitový and, or, xor

a negace).

- AC^0-*RAM* – libovolné funkce vyčíslitelné hradlovou sítí polynomiální velikosti a konstantní hloubky s hradly o libovolně mnoha vstupech.⟨18⟩ To je teoreticky čistší, patří sem vše z předchozí skupiny mimo násobení, dělení a modula, a také spousta dalších operací.
- *Kombinace předchozího* – tj. pouze operace Word-RAMu, které jsou v AC^0.

Ve zbytku této kapitoly ukážeme, že na RAMu lze počítat mnohé věci efektivněji než na PM. Zaměříme se na Word-RAM, ale podobné konstrukce jdou provést i na AC^0-RAMu. (Kombinace obou omezení vede ke slabšímu modelu.)

Van Emde-Boas Trees

Van Emde-Boas Trees neboli VEBT [59] jsou RAMová struktura, která si pamatuje množinu prvků X z nějakého omezeného universa $X \subseteq \{0, \ldots, U-1\}$, a umí s ní provádět „stromové operace“ (vkládání, mazání, nalezení následníka apod.) v čase $\mathcal{O}(\log\log U)$. Pomocí takové struktury pak například dokážeme:

	pomocí VEBT	nejlepší známé pro celá čísla
třídění	$\mathcal{O}(n \log\log U)$	$\mathcal{O}(n \log\log n)$ [29]
MST	$\mathcal{O}(m \log\log U)$	$\mathcal{O}(m)$ [viz příští kapitola]
Dijkstra	$\mathcal{O}(m \log\log U)$	$\mathcal{O}(m + n \log\log n)$ [54], neorientovaně $\mathcal{O}(m)$ [53]

My se přidržíme ekvivalentní, ale jednodušší definice podle Erika Demaine [16].

Definice: VEBT(U) pro universum velikosti U (BÚNO $U = 2^k = 2^{2^\ell}$) obsahuje:

- *min*, *max* reprezentované množiny (mohou být i nedefinovaná, pokud je množina moc malá),
- *přihrádky* $P_0, \ldots, P_{\sqrt{U}}$ obsahující zbývající hodnoty.⟨19⟩ Hodnota x padne do $P_{\lfloor x/\sqrt{U} \rfloor}$. Každá přihrádka je uložena jako VEBT($\sqrt{U}$), který obsahuje příslušná čísla mod $\sqrt{U}$. [Bity každého čísla jsme tedy rozdělili na vyšších $k/2$, které indexují přihrádku, a nižších $k/2$ uvnitř přihrádky.]
- Navíc ještě *„sumární“* VEBT($\sqrt{U}$) obsahující čísla neprázdných přihrádek.

⟨18⟩ Pro zvídavé: AC^k je třída všech funkcí spočítetelných polynomiálně velkou hradlovou sítí hloubky $\mathcal{O}(\log^k n)$ s libovolně-vstupovými hradly a NC^k totéž s omezením na hradla se dvěma vstupy. Všimněte si, že $NC^0 \subseteq AC^0 \subseteq NC^1 \subseteq AC^1 \subseteq NC^2 \subseteq \ldots$.

⟨19⟩ Alespoň jedno z *min*, *max* musí být uloženo zvlášť, aby strom obsahující pouze jednu hodnotu neměl žádné podstromy. My jsme pro eleganci struktury zvolili uložit zvlášť obojí.

Operace se strukturou budeme provádět následovně. Budeme si přitom představovat, že v přihrádkách jsou uložena přímo čísla reprezentované množiny, nikoliv jen části jejich bitů – z čísla přihrádky a hodnoty uvnitř přihrádky ostatně dokážeme celou hodnotu rekonstruovat a naopak hodnotu kdykoliv rozložit na tyto části.

FindMin – minimum nalezneme v kořeni v čase $\mathcal{O}(1)$.

Find(x) – přímočaře rekurzí přes přihrádky v čase $\mathcal{O}(k)$.

Insert(x):

1. Ošetříme triviální stromy (prázdný a jednoprvkový)
2. Je-li třeba, prohodíme x s min, max.
3. Prvek x padne do přihrádky P_i, která je buď:
4. prázdná ⇒ *Insert* hodnoty i do sumárního stromu a založení triviálního stromu pro přihrádku; nebo
5. neprázdná ⇒ převedeme na *Insert* v podstromu.

V každém případě buď rovnou skončíme nebo převedeme na *Insert* v jednom stromu nižšího řádu a k tomu vykonáme konstantní práci. Celkem tedy $\mathcal{O}(k)$.

Delete(x) – smazání prvku bude opět pracovat v čase $\mathcal{O}(k)$.

1. Ošetříme triviální stromy (jednoprvkový a dvouprvkový).
2. Pokud mažeme min (analogicky max), nahradíme ho minimem z první neprázdné přihrádky (tu najdeme podle sumárního stromu v konstantním čase) a převedeme na *Delete* v této přihrádce.
3. Prvek x padne do přihrádky P_i, která je buď:
4. jednoprvková ⇒ zrušení přihrádky a *Delete* ze sumárního stromu; nebo
5. větší ⇒ *Delete* ve stromu přihrádky.

Succ(x) – nejmenší prvek větší než x, opět v čase $\mathcal{O}(k)$:

1. Triviální případy: pokud $x < min$, vrátíme min; pokud $x \geq max$, vrátíme, že následník neexistuje.
2. Prvek x padne do přihrádky P_i a buďto:
3. P_i je prázdná nebo $x = max(P_i)$ ⇒ pomocí *Succ* v sumárním stromu najdeme nejbližší další neprázdnou přihrádku P_j:
4. existuje-li ⇒ vrátíme $min(P_j)$,
5. neexistuje-li ⇒ vrátíme max.
6. nebo $x < max(P_i)$ ⇒ převedeme na *Succ* v P_i.

Složitosti operací jsou pěkné, ale nesmíme zapomenout, že strukturu je na počátku nutné inicializovat, což trvá $\Omega(\sqrt{U})$.[20] Z následujících úvah ovšem vyplyne, že si inicializaci můžeme odpustit.

[20] Svádí to k nápadu ponechat přihrádky neinicializované a nejdříve se vždy zeptat sumárního stromu, ale tím bychom si pokazili časovou složitost.

Modely inicializace

Jak může být definován obsah paměti na počátku výpočtu:

„Při odchodu zhasni“: Zavedeme, že paměť RAMu je na počátku inicializována nulami a program ji po sobě musí uklidit (to je nutné, aby programy šlo iterovat). To u VEBT není problém zařídit.

Neinicializovano: Na žádné konkrétní hodnoty se nemůžeme spolehnout, ale je definováno, že neinicializovanou buňku můžeme přečíst a dostaneme nějakou korektní, i když libovolnou, hodnotu. Tehdy nám pomůže:

Věta: Buď P program pro Word-RAM s nulami inicializovanou pamětí, běžící v čase $T(n)$. Pak existuje program P' pro Word-RAM s neinicializovanou pamětí počítající totéž v čase $\mathcal{O}(T(n))$.

Důkaz: Během výpočtu si budeme pamatovat, do kterých paměťových buněk už bylo něco zapsáno, a které tedy mají definovanou hodnotu. Prokládaně uložíme do paměti dvě pole: M, což bude paměť původního stroje, a L – seznam čísel buněk v M, do kterých už program zapsal. Přitom $L[0]$ bude udávat délku seznamu L.

Program nyní začne tím, že vynuluje $L[0]$ a bude simulovat původní program, přičemž kdykoliv ten bude chtít přečíst nějakou buňku z M, podíváme se do L, zda už je inicializovaná, a případně vrátíme nulu a buňku připíšeme do L.

To je funkční, ale pomalé. Redukci tedy vylepšíme tak, že založíme další proložené pole R, jehož hodnota $R[i]$ bude říkat, kde v L se vyskytuje číslo i-té buňky, nebo bude neinicializována, pokud takové místo dosud neexistuje.

Před čtením $M[i]$ se teď podíváme na $R[i]$ a ověříme, zda $R[i]$ neleží mimo seznam L a zda je $L[R[i]] = i$. Tím v konstantním čase zjistíme, jestli je $M[i]$ již inicializovaná, a jsme také schopni tuto informaci v témže čase udržovat. ♡

„Minové pole“: Neinicializované buňky není ani dovoleno číst. V tomto případě nejsme schopni deterministické redukce, ale alespoň mužeme použít randomizovanou – ukládat si obsah paměti do hashovací tabulky, což při použití universálního hashování dá složitost $\mathcal{O}(1)$ na operaci v průměrném případě.

Technické triky

VEBT nedosahují zdaleka nejlepších možných parametrů – lze sestrojit i struktury pracující v konstantním čase. To v následující kapitole také uděláme, ale nejdříve v této poněkud technické stati vybudujeme repertoár základních operací proveditelných na Word-RAMu v konstantním čase.

Rozcvička: *nejpravější jednička* ve dvojkovém čísle (hodnota, nikoliv pozice):

$$\begin{aligned} x &= \mathbf{01}\cdots\mathbf{011000000} \\ x-1 &= \mathbf{01}\cdots\mathbf{010111111} \\ x\wedge(x-1) &= \mathbf{01}\cdots\mathbf{010000000} \\ x\oplus(x\wedge(x-1)) &= \mathbf{00}\cdots\mathbf{001000000} \end{aligned}$$

Nyní ukážeme, jak RAM používat jako vektorový počítač, který umí paralelně počítat se všemi prvky vektoru, pokud se dají zakódovat do jediného slova. Libovolný

n-složkový vektor, jehož složky jsou b-bitová čísla ($n(b+1) \leq w$), zakódujeme poskládáním jednotlivých složek vektoru za sebe, proloženě nulovými bity:

$$\mathbf{0}x_{n-1}\mathbf{0}x_{n-2}\cdots\mathbf{0}x_1\mathbf{0}x_0$$

S vektory budeme provádět následující operace: (latinkou značíme vektory, alfabetou čísla, $\mathbf{0}$ a $\mathbf{1}$ jsou jednotlivé bity, $(\ldots)^k$ je k-násobné opakování binárního zápisu)

- *Replicate*(α) – vytvoří vektor $(\alpha, \alpha, \ldots, \alpha)$:

  $$\alpha * (\mathbf{0}^b\mathbf{1})^n$$

- *Sum*(x) – sečte všechny složky vektoru (předpokládáme, že se součet vejde do b bitů):

  a) vymodulením číslem $\mathbf{1}^{b+1}$ (protože $\mathbf{10}^{b+1} \bmod \mathbf{1}^{b+1} = 1$), či

  b) násobením vhodnou konstantou:

$$\begin{array}{cccccccccc}
 & & & & & x_{n-1} & \cdots & x_2 & x_1 & x_0 \\
 & & & & * & \mathbf{0}^b\mathbf{1} & \cdots & \mathbf{0}^b\mathbf{1} & \mathbf{0}^b\mathbf{1} & \mathbf{0}^b\mathbf{1} \\
\hline
 & & & & & x_{n-1} & \cdots & x_2 & x_1 & x_0 \\
 & & & & x_{n-1} & x_{n-2} & \cdots & x_1 & x_0 & \\
 & & & x_{n-1} & x_{n-2} & x_{n-3} & \cdots & x_0 & & \\
 & & & \vdots & \vdots & \vdots & \vdots & & & \\
x_{n-1} & \cdots & x_2 & x_1 & x_0 & & & & & \\
\hline
r_{n-1} & \cdots & r_2 & r_1 & s_n & \cdots & s_3 & s_2 & s_1 &
\end{array}$$

  Zde je výsledkem dokonce vektor všech částečných součtů: $s_k = \sum_{i=0}^{k-1} x_i$, $r_k = \sum_{i=k}^{n-1} x_i$.

- *Cmp*(x, y) – paralelní porovnání dvou vektorů: i-tá složka výsledku je 1, pokud $x_i < y_i$, jinak 0.

$$\begin{array}{cccccccccc}
 & \mathbf{1} & x_{n-1} & \mathbf{1} & x_{n-2} & \cdots & \mathbf{1} & x_1 & \mathbf{1} & x_0 \\
- & \mathbf{0} & y_{n-1} & \mathbf{0} & y_{n-2} & \cdots & \mathbf{0} & y_1 & \mathbf{0} & y_0
\end{array}$$

  Ve vektoru x nahradíme prokládací nuly jedničkami a odečteme vektor y. Ve výsledku se tyto jedničky změní na nuly právě u těch složek, kde $x_i < y_i$. Pak je již stačí posunout na správné místo a okolní bity vynulovat a znegovat.

- *Rank*(α, x) – spočítá, kolik složek vektoru x je menších než α:

  $$\mathit{Rank}(\alpha, x) = \mathit{Sum}(\mathit{Cmp}(\mathit{Replicate}(\alpha), x)).$$

- *Insert*(α, x) – zatřídí hodnotu α do setříděného vektoru x:
  Zde stačí pomocí operace *Rank* zjistit, na jaké místo novou hodnotu zatřídit, a pak to pomocí bitových operací provést („rozšoupnout“ existující hodnoty).

- *Unpack*(α) – vytvoří vektor, jehož složky jsou bity zadaného čísla (jinými slovy proloží bity bloky b nul).
  Nejdříve číslo α replikujeme, pak andujeme vhodnou bitovou maskou, aby v i-té složce zůstal pouze i-tý bit a ostatní se vynulovaly, a pak provedeme *Cmp* s vektorem samých nul.
- *Unpack*$_\varphi(\alpha)$ – podobně jako předchozí operace, ale bity ještě prohází podle nějaké pevné funkce φ:
  Stačí zvolit bitovou masku, která v i-té složce ponechá právě $\varphi(i)$-tý bit.
- *Pack*(x) – dostane vektor nul a jedniček a vytvoří číslo, jehož bity jsou právě složky vektoru (jinými slovy škrtne nuly mezi bity):
  Představíme si, že složky čísla jsou o jeden bit kratší a provedeme *Sum*. Například pro $n = 4$ a $b = 4$:

$$\left|\mathbf{0}\ \mathbf{0}\ \mathbf{0}\ \mathbf{0}\ x_3\right|\mathbf{0}\ \mathbf{0}\ \mathbf{0}\ \mathbf{0}\ x_2\left|\mathbf{0}\ \mathbf{0}\ \mathbf{0}\ \mathbf{0}\ x_1\right|\mathbf{0}\ \mathbf{0}\ \mathbf{0}\ \mathbf{0}\ x_0\right|$$
$$\left|\mathbf{0}\ \mathbf{0}\ \mathbf{0}\ \mathbf{0}\left|x_3\ \mathbf{0}\ \mathbf{0}\ \mathbf{0}\right|\mathbf{0}\ x_2\ \mathbf{0}\ \mathbf{0}\right|\mathbf{0}\ \mathbf{0}\ x_1\ \mathbf{0}\left|\mathbf{0}\ \mathbf{0}\ \mathbf{0}\ x_0\right|$$

  Jen si musíme dát pozor na to, že vytvořený vektor s kratšími složkami není korektně prostrkán nulami. Konstrukce *Sum* pomocí modula proto nebude správně fungovat a místo $\mathbf{1}^b$ vygeneruje $\mathbf{0}^b$. To můžeme buď ošetřit zvlášť, nebo použít konstrukci přes násobení, které to nevadí.

Nyní ještě několik operací s normálními čísly. Chvíli předpokládejme, že pro b-bitová čísla na vstupu budeme mít k dispozici b^2-bitový pracovní prostor, takže budeme moci používat vektory s b složkami po b bitech.

- $\#1(\alpha)$ – spočítá jedničkové bity v zadaném čísle.
  Stačí provést *Unpack* a následně *Sum*.
- *Permute*$_\pi(\alpha)$ – přehází bity podle zadané fixní permutace.
  Provedeme *Unpack*$_\pi$ a *Pack* zpět.
- *LSB*(α) – Least Significant Bit (pozice nejnižší jedničky):
  Podobně jako v rozcvičce nejdříve vytvoříme číslo, které obsahuje nejnižší jedničku a vpravo od ní další jedničky, a pak tyto jedničky posčítáme pomocí #1:

$$\begin{aligned}\alpha &= \cdots\cdots\mathbf{10000}\\ \alpha - 1 &= \cdots\cdots\mathbf{01111}\\ \alpha \oplus (\alpha - 1) &= \mathbf{0}\cdots\mathbf{011111}\end{aligned}$$

- *MSB*(α) – Most Significant Bit (pozice nejvyšší jedničky):
  Z *LSB* pomocí zrcadlení (operací *Permute*).

Poslední dvě operace dokážeme spočítat i v lineárním prostoru, například pro *MSB* takto: Rozdělíme číslo na bloky velikosti $\lfloor\sqrt{w}\rfloor$. Pak pro každý blok zjistíme, zda v něm je aspoň jedna jednička, zavoláním *Cmp*$(0, x)$. Pomocí *Pack* z toho dostaneme slovo y odmocninové délky, jehož bity indikují neprázdné bloky. Na toto číslo

zavoláme předchozí kvadratické *MSB* a zjistíme index nejvyššího neprázdného bloku. Ten pak izolujeme a druhým voláním kvadratického algoritmu najdeme nejlevější jedničku uvnitř něj.⟨21⟩

⟨21⟩ Dopouštíme se drobného podvůdku – vektorové operace předpokládaly prostrkané nuly a ty tu nemáme. Můžeme si je ale snadno pořídit a bity, které jsme nulami přepsali, prostě zpracovat zvlášť.

8. Q-Heaps

V minulé kapitole jsme zavedli výpočetní model RAM a nahlédli jsme, že na něm můžeme snadno simulovat vektorový počítač s vektorovými operacemi pracujícími v konstantním čase. Když už máme takový počítač, pojďme si ukázat, jaké datové struktury na něm můžeme vytvářet.

Svým snažením budeme směřovat ke strukturám, které zvládnou operace *Insert* a *Delete* v konstantním čase, přičemž bude omezena buďto velikost čísel nebo maximální velikost struktury nebo obojí. Bez újmy na obecnosti budeme předpokládat, že hodnoty, které do struktur ukládáme, jsou navzájem různé.

Značení: w bude vždy značit šířku slova RAMu a n velikost vstupu algoritmu, v němž datovou strukturu využíváme (speciálně tedy víme, že $w \geq \log n$).

Word-Encoded B-Tree

První strukturou, kterou popíšeme, bude vektorová varianta B-stromu. Nemá ještě tak zajímavé parametry, ale odvozuje se snadno a jsou na ní dobře vidět mnohé myšlenky používané ve strukturách složitějších.

Půjde o obyčejný B-strom s daty v listech, ovšem kódovaný vektorově. Do listů stromu budeme ukládat k-bitové hodnoty, vnitřní vrcholy budou obsahovat pouze pomocné klíče a budou mít nejvýše B synů. Strom bude mít hloubku h. Hodnoty všech klíčů ve vrcholu si budeme ukládat jako vektor, ukazatele na jednotlivé syny jakbysmet.

Se stromem zacházíme jako s klasickým B-stromem, přitom operace s vrcholy provádíme vektorově: vyhledání pozice prvku ve vektoru pomocí operace *Rank*, rozdělení a slučování vrcholů pomocí bitových posuvů a maskování, to vše v čase $\mathcal{O}(1)$. Stromové operace (*Find*, *FindNext*, *Insert*, *Delete*, ...) tedy stihneme v čase $\mathcal{O}(h)$.

Zbývá si rozmyslet, co musí splňovat parametry struktury, aby se všechny vektory vešly do konstantního počtu slov. Kvůli vektorům klíčů musí platit $Bk = \mathcal{O}(w)$. Jelikož strom má až B^h listů a nejvýše tolik vnitřních vrcholů, ukazatele zabírají $\mathcal{O}(h \log B)$ bitů, takže pro vektory ukazatelů potřebujeme, aby bylo $Bh \log B = \mathcal{O}(w)$. Dobrá volba je například $B = k = \sqrt{w}$, $h = \mathcal{O}(1)$, čímž získáme strukturu obsahující $w^{\mathcal{O}(1)}$ prvků o $\sqrt{w}$ bitech, která pracuje v konstantním čase.

Q-Heap

Předchozí struktura má zajímavé vlastnosti, ale často je její použití znemožněno omezením na velikost čísel. Popíšeme tedy o něco složitější konstrukci od Fredmana a Willarda [24], která dokáže totéž, ale s až w-bitovými čísly. Tato struktura má spíše teoretický význam (konstrukce je značně komplikovaná a skryté konstanty nemalé), ale překvapivě mnoho myšlenek je použitelných i prakticky.

Značení:

- $k = \mathcal{O}(w^{1/4})$ – omezení na velikost haldy,
- $r \leq k$ – aktuální počet prvků v haldě,

- $X = \{x_1, \ldots, x_r\}$ – uložené w-bitové prvky, očíslujeme si je tak, aby $x_1 < \ldots < x_r$,
- $c_i = \mathrm{MSB}(x_i \oplus x_{i+1})$ – nejvyšší bit, ve kterém se liší x_i a x_{i+1},
- $\mathrm{Rank}_X(x)$ – počet prvků množiny X, které jsou menší než x (přičemž x může ležet i mimo X).

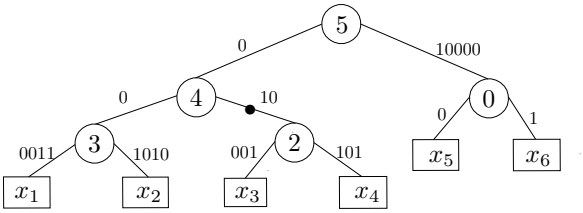
Předvýpočet: Budeme ochotni obětovat čas $\mathcal{O}(2^{k^4})$ na předvýpočet. To může znít hrozivě, ale ve většině aplikací bude $k = \log^{1/4} n$, takže předvýpočet stihneme v čase $\mathcal{O}(n)$. V takovém čase mimo jiné stihneme předpočítat tabulku pro libovolnou funkci, která má vstup dlouhý $\mathcal{O}(k^3)$ bitů a kterou pro každý vstup dovedeme vyhodnotit v polynomiálním čase. Nadále tedy můžeme bezpečně předpokládat, že všechny takové funkce umíme spočítat v konstantním čase.

Iterování: Všimněte si, že jakmile dokážeme sestrojit haldu s k prvky pracující v konstantním čase, můžeme s konstantním zpomalením sestrojit i haldu s $k^{\mathcal{O}(1)}$ prvky. Stačí si hodnoty uložit do listů stromu s větvením k a konstantním počtem hladin a v každém vnitřním vrcholu si pamatovat minimum podstromu a Q-Heap s hodnotami jeho synů. Tak dokážeme každé vložení i odebrání prvku převést na konstantně mnoho operací s Q-Heapy.

Náčrt fungování Q-Heapu: Nad prvky $x_1, \ldots, x_r$ sestrojíme trii T a nevětvící se cesty zkomprimujeme (nahradíme hranami). Listy trie budou jednotlivá x_i, vnitřní vrchol, který leží mezi x_i a x_{i+1}, bude testovat c_i-tý bit čísla. Pokud budeme hledat některé z x_i, tyto vnitřní vrcholy (budeme jim říkat *značky*⟨22⟩) nás správně dovedou do příslušného listu. Pokud ale budeme hledat nějaké jiné x, zavedou nás do nějakého na první pohled nesouvisejícího listu a teprve tam zjistíme, že jsme zabloudili. K našemu překvapení však to, kam jsme se dostali, bude stačit ke spočítání ranku prvku a z ranků už odvodíme i ostatní operace.

Příklad: Trie pro zadanou množinu čísel. Ohodnocení hran je pouze pro názornost, není součástí struktury.

$x_1 = 000011 \quad c_1 = 3$
$x_2 = 001010 \quad c_2 = 4$
$x_3 = 010001 \quad c_3 = 2$
$x_4 = 010101 \quad c_4 = 5$
$x_5 = 100000 \quad c_5 = 0$
$x_6 = 100001$
543210



Lemma R: $\mathrm{Rank}_X(x)$ je určen jednoznačně kombinací:

(i) tvaru stromu T,
(ii) indexu i listu x_i, do kterého nás zavede hledání hodnoty x ve stromu,

⟨22⟩ třeba turistické pro orientaci v lese

(iii) vztahu mezi x a x_i ($x < x_i$, $x > x_i$ nebo $x = x_i$) a
(iv) pozice $b = \mathrm{MSB}(x \oplus x_i)$.

Důkaz: Pokud $x = x_i$, je zjevně $\mathrm{Rank}_X(x) = i$. Předpokládejme tedy $x \neq x_i$. Hodnoty značek klesají ve směru od kořene k listům a na cestě od kořene k x_i se všechny bity v x_i na pozicích určených značkami shodují s bity v x. Přitom až do pozice b se shodují i bity značkami netestované. Sledujme tuto cestu od kořene až po b: pokud cesta odbočuje doprava, jsou všechny hodnoty v levém podstromu menší než x, a tedy se do ranku započítají. Pokud odbočuje doleva, jsou hodnoty v pravém podstromu zaručeně větší a nezapočítají se. Pokud nastala neshoda a $x < x_i$ (tedy b-tý bit v x je nula, zatímco v x_i je jedničkový), jsou všechny hodnoty pod touto hranou větší; při opačné nerovnosti jsou menší. ♡

Příklad: Vezměme množinu $X = \{x_1, x_2, \ldots, x_6\}$ z předchozího příkladu a počítejme $\mathrm{Rank}_X(011001)$. Místo první neshody je označeno puntíkem. Platí $x > x_i$, tedy celý podstrom je menší než x, a tak je $\mathrm{Rank}_X(011001) = 4$.

Rádi bychom předchozí lemma využili k sestrojení tabulek, které podle uvedených hodnot vrátí rank prvku x. K tomu potřebujeme především umět indexovat tvarem stromu.

Pozorování: Tvar trie je jednoznačně určen hodnotami $c_1, \ldots, c_{r-1}$ (je to totiž kartézský strom nad těmito hodnotami – blíže viz kapitola o dekompozicích stromů), hodnoty v listech jsou $x_1, \ldots, x_r$ v pořadí zleva doprava.

Kdykoliv chceme indexovat tvarem stromu, můžeme místo toho použít přímo vektor $(c_1, \ldots, c_r - 1)$, který má $k \log w$ bitů. To se sice už vejde do vektoru, ale pro indexování tabulek je to stále příliš (všimněte si, že $\log w$ smí být vůči k libovolně vysoký – pro w známe pouze dolní mez). Proto reprezentaci ještě rozdělíme na dvě části:

- $B := \{c_1, \ldots, c_r\}$ (množina všech pozic bitů, které trie testuje, uložená ve vektoru setříděně),
- $C : \{1, \ldots, r\} \to B$ taková, že $B[C(i)] = c_i$.

Lemma R': $\mathrm{Rank}_X(x)$ lze spočítat v konstantním čase z:

(i') funkce C,
(ii') hodnot $x_1, \ldots, x_r$,
(iii') $x[B]$ – hodnot bitů na „zajímavých“ pozicích v čísle x.

Důkaz: Z předchozího lemmatu:

(i) Tvar stromu závisí jen na nerovnostech mezi polohami značek, takže je jednoznačně určený funkcí C.
(ii) Z tvaru stromu a $x[B]$ jednoznačně plyne list x_i a tyto vstupy jsou dostatečně krátké na to, abychom mohli předpočítat tabulku pro průchod stromem.
(iii) Relaci zjistíme prostým porovnáním, jakmile známe x_i.
(iv) MSB umíme na RAMu počítat v konstantním čase.

Mezivýsledky (i)–(iv) jsou opět dost krátké na to, abychom jimi mohli indexovat tabulku. ♡

Počítání ranků je téměř vše, co potřebujeme k implementaci operací *Find*, *Insert* a *Delete*. Jedinou další překážku tvoří zatřiďování do seznamu $x_1, \ldots, x_r$, který je moc velký na to, aby se vešel do $\mathcal{O}(1)$ slov. Proto si budeme pamatovat zvlášť hodnoty v libovolném pořadí a zvlášť permutaci, která je setřídí – ta se již do vektoru vejde. Řekněme tedy pořádně, co vše si bude struktura pamatovat:

Stav struktury:

- k, r – kapacita haldy a aktuální počet prvků (čísla),
- $X = \{x_1, \ldots, x_r\}$ – hodnoty prvků v libovolném pořadí (pole čísel),
- ϱ – permutace na $\{1, \ldots, r\}$ taková, že $x_i = X[\varrho(i)]$ a $x_1 < x_2 < \ldots < x_r$ (vektor o $r \cdot \log k$ bitech),
- B – množina „zajímavých“ bitových pozic (setříděný vektor o $r \cdot \log w$ bitech),
- C – funkce popisující značky: $c_i = B[C(i)]$ (vektor o $r \cdot \log k$ bitech),
- předpočítané tabulky pro různé funkce.

Nyní již ukážeme, jak provádět jednotlivé operace:

Find(x) :

1. $i \leftarrow \mathrm{Rank}_X(x)$.
2. Pokud $x_i = x$, odpovíme ANO, jinak NE.

Insert(x) :

1. $i \leftarrow \mathrm{Rank}_X(x)$.
2. Pokud $x = x_i$, hodnota už je přítomna.
3. Uložíme x do $X[{+}{+}\,r]$ a vložíme r na i-té místo v permutaci ϱ.
4. Přepočítáme c_{i-1} a c_i. Pro každou změnu c_j:
5. Pokud ještě nová hodnota není v B, přidáme ji tam.
6. Upravíme $C(j)$, aby ukazovalo na tuto hodnotu.
7. Upravíme ostatní prvky C, ukazující na hodnoty v B, které se vložením posunuly.
8. Pokud se na starou hodnotu neodkazuje žádné jiné $C(\cdot)$, smažeme ji z B.

Delete(x) :

1. $i \leftarrow \mathrm{Rank}_X(x)$ (víme, že $x_i = x$).
2. Smažeme x_i z pole X (například prohozením s posledním prvkem) a příslušně upravíme ϱ.
3. Přepočítáme c_{i-1} a c_i a upravíme B a C jako při Insertu.

Časová složitost: Všechny kroky operací po výpočtu ranku trvají konstantní čas, rank samotný zvládneme spočítat v $\mathcal{O}(1)$ pomocí tabulek, pokud známe $x[B]$. Zde

je ovšem nalíčen háček – tuto operaci nelze na Word-RAMu konstantním počtem instrukcí spočítat. Pomoci si můžeme dvěma způsoby:

a) Využijeme toho, že operace $x[B]$ je v AC^0, a vystačíme si se strukturou pro AC^0-RAM. Zde dokonce můžeme vytvářet haldy velikosti až $w \log w$. Také při praktické implementaci můžeme využít toho, že současné procesory mají instrukce na spoustu zajímavých AC^0-operací, viz např. pěkný rozbor v [56].
b) Jelikož B se při jedné Q-Heapové operaci mění pouze o konstantní počet prvků, můžeme si udržovat pomocné struktury, které budeme umět při lokální změně B v lineárním čase přepočítat a pak pomocí nich indexovat. To pomocí Word-RAMu lze zařídit, ale je to technicky dosti náročné, takže čtenáře zvědavého na detaily odkazujeme na článek [24].

Aplikace Q-Heapů

Jedním velice pěkným důsledkem existence Q-Heapů je lineární algoritmus na nalezení minimální kostry grafu ohodnoceného celými čísly. Získáme ho z Fredmanovy a Tarjanovy varianty Jarníkova algoritmu (viz kapitoly o kostrách) tak, že v první iteraci použijeme jako haldu Q-Heap velikosti $\log^{1/4} n$ a pak budeme pokračovat s původní Fibonacciho haldou. Tak provedeme tolik průchodů, kolikrát je potřeba zlogaritmovat n, aby výsledek klesl pod $\log^{1/4} n$, a to je konstanta. Všimněte si, že by nám dokonce stačila halda velikosti $\Omega(\log^{(k)} n)$ s operacemi v konstantním čase pro nějaké libovolné k.

9. Dekompozice stromů

V této kapitole ukážeme několik datových struktur založených na myšlence dekompozice problému na dostatečně malé podproblémy, které už umíme (obvykle vhodným kódováním čísly) řešit v konstantním čase.

Union-Find Problem

Problém: Udržování tříd ekvivalence: na počátku máme N jednoprvkových ekvivalenčních tříd, provádíme operace *Find* (zjištění, zda dva prvky jsou ekvivalentní) a *Union* (sloučení dvou tříd do jedné). Také na to lze pohlížet jako na inkrementální udržování komponent souvislosti neorientovaného grafu: *Union* je přidání hrany, *Find* test, zda dva vrcholy leží v téže komponentě. To se hodí v mnoha algoritmech, kupříkladu v Kruskalově algoritmu pro hledání minimální kostry.

Triviální řešení: Prvky každé třídy obarvíme unikátní barvou (identifikátorem třídy). Operace *Find* porovnává barvy, *Union* prvky jedné ze sjednocovaných tříd přebarvuje.

Operace *Find* tak pracuje v konstantním čase, *Union* může zabrat až lineární čas. Můžeme si pomoci tím, že vždy přebarvíme *menší* ze slučovaných ekvivalenčních tříd (budeme si pro každou třídu pamatovat seznam jejích prvků a velikost). Tehdy může být každý prvek přebarven jen $\mathcal{O}(\log n)$-krát, jelikož každým přebarvením se alespoň zdvojnásobí velikost třídy, ve které prvek leží. Posloupnost operací *Union*, kterou vznikla třída velikosti k, tak trvá $\mathcal{O}(k \log k)$, takže můžeme bezpečně prohlásit, že amortizovaná složitost operace *Union* je $\mathcal{O}(\log n)$.

Chytřejší řešení: Každou třídu budeme reprezentovat zakořeněným stromem s hranami orientovanými směrem ke kořeni (jinými slovy pro každý prvek si pamatujeme jeho otce nebo že je to kořen). *Find* nalezne kořeny stromů a porovná je, *Union* připojí kořen jedné třídy pod kořen druhé. Aby stromy nedegenerovaly, přidáme dvě pravidla:

- *Union dle ranku:* každý kořen v si bude pamatuje svůj rank $r(v)$, což je nějaké přirozené číslo. Na počátku jsou všechny ranky nulové. Pokud spojujeme dva stromy s kořeny v, w a $r(v) < r(w)$, připojíme v pod w a rank zachováme. Pokud $r(v) = r(w)$, připojíme libovolně a nový kořen bude mít rank $r(v) + 1$.[23]
- *Komprese cest:* pokud z vrcholu vystoupíme do kořene (například během operace *Find*), přepojíme všechny vrcholy na cestě, po které jsme prošli, rovnou pod kořen.

Pro účely analýzy struktury budeme uvažovat také ranky ostatních vrcholů – každý vrchol si ponese svůj rank z doby, kdy byl naposledy kořenem. Struktura se ovšem podle ranků vnitřních vrcholů nijak neřídí a nemusí si je ani pamatovat. Stromu s kořenem ranku r budeme zkráceně říkat *strom ranku r.*

[23] Stejně by fungovalo pravidlo *Union dle velikosti,* které připojuje menší strom pod větší, ale ranky máme raději, neb jsou skladnější a snáze se analyzují.

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

Důkaz: Na počátku (pro jednovrcholové stromy) tvrzení jistě platí. Nechť provedeme Union, který připojí vrchol v pod w. Cesty do kořene z vrcholů, které ležely pod w, zůstanou zachovány, pouze se vrcholu w případně zvýší rank. Cesty z vrcholů pod v se rozšíří o hranu vw, na které rank v každém případě roste. Komprese cest nahrazuje otce vrcholu jeho vzdálenějším předkem, takže se rank otce může jedině zvýšit. ♡

Invariant R: Strom ranku r obsahuje alespoň 2^r vrcholů.

Důkaz: Indukcí podle času. Pro jednovrcholové stromy o nulovém ranku tvrzení platí. Nechť připojíme vrchol v pod vrchol w. Je-li $r(v) < r(w)$, rank stromu zůstane zachován a strom se ještě zvětší. Je-li $r(v) = r(w)$, rank stromu se zvětší o 1, ale z indukce víme, že oba spojované stromy měly alespoň $2^{r(v)}$ vrcholů, takže jejich spojením vznikne strom o alespoň $2^{r(v)+1}$ vrcholech. Komprese cest zasahuje pouze do vnitřní struktury stromu, ranky ani velikosti stromů nemění. ♡

Důsledek: Rank každého stromu je $\mathcal{O}(\log n)$, takže rank každého vnitřního vrcholu taktéž. Díky invariantu C strávíme výstupem z každého vrcholu do kořene také čas $\mathcal{O}(\log n)$, takže logaritmická je i složitost operací *Union* a *Find*.

K tomu nám ovšem stačilo samotné pravidlo Union podle ranku, o kompresi cest jsme zatím dokázali pouze to, že složitost v nejhorším případě nezhoršuje.⟨24⟩ Kombinace obou metod se ve skutečnosti chová mnohem lépe:

Věta: (Tarjan, van Leeuwen [52]) Posloupnost m operací *Union* a *Find* provedená na prázdné struktuře s n vrcholy trvá $\mathcal{O}(n + m\alpha(m, n))$, kde α je inverzní Ackermannova funkce.⟨25⟩

Důkaz této věty neuvádíme, jelikož je technicky dosti náročný. Místo toho podobnou metodou ukážeme trochu slabší výsledek s iterovaným logaritmem:

Věta': Ve struktuře s n prvky trvá provedení posloupnosti m operací *Union* a *Find* $\mathcal{O}((n + m) \cdot \log^* n)$.

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

Funkce $2 \uparrow k$ je tedy k-krát iterovaná mocnina dvojky a $\log^*$ je funkce k této funkci inverzní.

Vrcholy ve struktuře si nyní rozdělíme podle jejich 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. Odhadněme nyní shora počet vrcholů v k-té skupině.

⟨24⟩ Mimochodem, Komprese cest samotná by také na složitost $\mathcal{O}(\log n)$ amortizovaně stačila [52].

⟨25⟩ Je známo [23], že asymptoticky lepší složitosti nelze dosáhnout, a to ani v modelu silnějším než RAM. Námi uváděný algoritmus si téměř vystačí s Pointer Machine, jen porovnávání ranků z tohoto modelu vybočuje. Složitost operací v nejhorším případě je obecně horší, je znám dolní odhad $\Omega(\log n / \log \log n)$; více viz Alstrup [2].

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

Důkaz: Nejprve ukážeme, že vrcholů s rankem r je nejvýše $n/2^r$. Kdybychom nekomprimovali cesty, snadno to plyne z invariantů C a R: každému vrcholu ranku r přiřadíme všech jeho alespoň 2^r potomků. Jelikož ranky na cestách směrem ke kořeni rostou, žádného potomka jsme nemohli přiřadit více vrcholům. Komprese cest ovšem nemůže invariant porušit, protože nemění ranky ani rozhodnutí, jak proběhne který *Union*.

Teď už stačí odhad $n/2^r$ sečíst přes všechny ranky ve skupině:

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

♡

Důkaz věty: Operace *Union* a *Find* potřebují nekonstantní čas pouze na vystoupání po cestě ze zadaného vrcholu v do kořene stromu. 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 stromu.

Hrany cesty, které spojují vrcholy z různých skupin (takových je $\mathcal{O}(\log^* n)$), naúčtujeme právě prováděné operaci. Celkem jimi tedy strávíme čas $\mathcal{O}(m \log^* n)$. 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 $\mathcal{O}(n \log^* n)$. ♡

Union-Find s předem známými Uniony

Dále nás bude zajímat speciální varianta Union-Find problému, v níž dopředu známe posloupnost Unionů, čili strom, který spojováním komponent vznikne.⟨26⟩ Jiná interpretace téhož (jen pozpátku) je dekrementální udržování komponent souvislosti lesa: na počátku je dán les, umíme smazat hranu a otestovat, zda jsou dva vrcholy v témže stromu.

Popíšeme algoritmus, který po počátečním předzpracování v čase $\mathcal{O}(n)$ zvládne *Union* i *Find* v amortizovaně konstantním čase. Tento algoritmus je kombinací dekompozic popsaných Alstrupem [4, 3].

⟨26⟩ Kdy se to hodí? Třeba v Thorupově lineárním algoritmu [53] na nejkratší cesty nebo ve Weiheho taktéž lineárním algoritmu [60] na hledání hranově disjunktních cest v rovinných grafech.

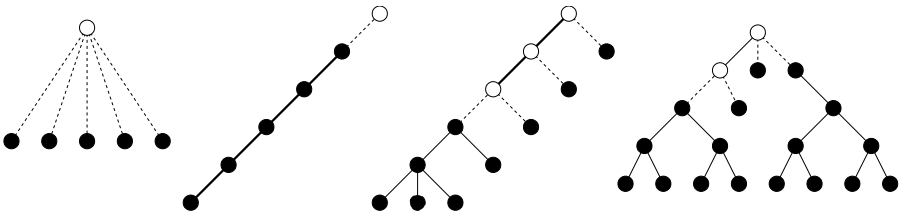
Definice: *(Microtree/Macrotree dekompozice)* Pro zakořeněný strom T o n vrcholech definujeme:

- *Kořeny mikrostromů* budou nejvyšší vrcholy v T, pod nimiž je nejvýše $\log n$ listů a které nejsou kořenem celého T.
- *Mikrostromy* leží v T od těchto kořenů níže.
- *Spojovací hrany* vedou z kořenů mikrostromů do jejich otců.
- *Makrostrom* je tvořen zbývajícími vrcholy a hranami stromu T.

Pozorování: Každý mikrostrom má nejvýše $\log n$ listů. Pod každým listem makrostromu leží alespoň jeden mikrostrom (může jich být i více, viz dekompozice hvězdy na obrázku), takže listů makrostromu je nejvýše $n/\log n$.

Vnitřních vrcholů makro- i mikrostromů ale může být nešikovně mnoho, protože se ve stromech mohou vyskytovat dlouhé cesty. Pomůžeme si snadno: každou cestu si budeme pamatovat zvlášť a ve stromu ji nahradíme hranou, která bude vložena právě tehdy, když budou přítomny všechny hrany cesty.

Příklad: Následující obrázek ukazuje dekompozici několika stromů za předpokladu, že $\log n = 4$. Vrcholy mikrostromů jsou černé, makrostromu bílé. Spojovací hrany kreslíme tečkovaně, hrany komprimovaných cest tučně.



Algoritmus pro cesty: Cestu délky l rozdělíme na úseky délky $\log n$, pro něž si uložíme množiny již přítomných hran (po bitech jako čísla). Pak si ještě pamatujeme zkomprimovanou cestu (hrany odpovídají úsekům a jsou přítomny právě tehdy, jsou-li přítomny všechny hrany příslušného úseku) délky $l/\log n$ a pro ni „přebarvovací“ strukturu pro Union-Find.

Union(x, y) (přidání hrany $e = xy$ do cesty):

1. Přidáme e do množiny hran přítomných v příslušném úseku.
2. Pokud se tím úsek naplnil, přidáme odpovídající hranu do zkomprimované cesty.

Find(x, y) :

1. Pokud x a y jsou v témže úseku, otestujeme bitovými operacemi, zda jsou všechny hrany mezi x a y přítomny.
2. Pokud jsou v různých úsecích, rozdělíme cestu z x do y na posloupnost celých úseků, na které nám odpoví zkomprimovaná cesta, a dva dotazy v krajních částečných úsecích.

Operace uvnitř úseků pracují v čase $\mathcal{O}(1)$, operace na zkomprimované cestě v $\mathcal{O}(\log l)$ amortizovaně, ale za dobu života struktury je jich $\mathcal{O}(l/\log n) = \mathcal{O}(l/\log l)$, takže celkově zaberou lineární čas.

Komprese cest: Operace na mikro/makro-stromech budeme následujícím způsobem převádět na operace s jejich cestově komprimovanými podobami a na operace s cestovými strukturami:

Union(x, y):

1. Pokud $e = xy$ leží uvnitř nějaké cesty, přidáme ji do cesty, což buďto způsobí přidávání jiné hrany, a nebo už jsme hotovi.
2. Provedeme *Union* v komprimovaném stromu.

Find(x, y):

1. Pokud x a y leží uvnitř jedné cesty, zeptáme se cestové struktury a končíme.
2. Pokud x leží uvnitř nějaké cesty, zjistíme dotazem na cestovou strukturu, ke kterému krajnímu vrcholu cesty je připojen, a x nahradíme tímto vrcholem. Není-li připojen k žádnému, je evidentně odpověď na celý *Find* negativní; pokud k oběma, vybereme si libovolný, protože jsou stejně v cestově komprimovaném stromu spojeny hranou. Analogicky pro y.
3. Zeptáme se struktury pro komprimovaný strom.

Algoritmus pro mikrostromy: Po kompresi cest má každý mikrostrom nejvýše $2 \log n$ vrcholů, čili také nejvýše tolik hran. Hrany si očíslujeme přirozenými čísly, každou množinu hran pak můžeme reprezentovat $(2 \log n)$-bitovým číslem a množinové operace provádět pomocí bitových v konstantním čase.

Pro každý mikrostrom si předpočítáme pro všechny jeho vrcholy v množiny P_v hran ležících na cestě z kořene mikrostromu do v. Navíc si budeme pro celý mikrostrom pamatovat množinu přítomných hran F.

Union(x, y) :

1. Najdeme pořadové číslo i hrany xy (máme předpočítané).
2. $F \leftarrow F \cup \{i\}$.

Find(x, y) :

1. $P \leftarrow P_x \,\Delta\, P_y$ (množina hran ležících na cestě z x do y).
2. Pokud $P \setminus F = \emptyset$, leží x a y ve stejně komponentě, jinak ne.

Algoritmus pro celý problém: Strom rozložíme na mikrostromy, makrostrom a spojovací hrany. V mikrostromech i makrostromu zkomprimujeme cesty. Pro cesty a mikrostromy použijeme výše popsané struktury, pro každou spojovací hranu si budeme pamatovat jen značku, zda je přítomna, a pro makrostrom přebarvovací strukturu.

Union(x, y):

1. Pokud $e = xy$ je spojovací, poznamenáme si, že je přítomna, a končíme.

2. Nyní víme, že e leží uvnitř mikrostromu nebo makrostromu, a tak provedeme *Union* na příslušné struktuře.

Find(x, y):

1. Leží-li x a y v jednom mikrostromu, zeptáme se struktury pro mikrostrom.
2. Je-li x uvnitř mikrostromu, zeptáme se mikrostruktury na spojení s kořenem mikrostromu. Není-li, odpovíme NE, stejně jako když není přítomna příslušná spojovací hrana. Jinak x nahradíme listem makrostromu, do kterého spojovací hrana vede. Podobně pro y.
3. Odpovíme podle struktury pro makrostrom.

Analýza: Operace *Find* trvá konstantní čas, protože se rozloží na $\mathcal{O}(1)$ *Find*ů v dílčích strukturách a každý z nich trvá konstantně dlouho. Všech n operací *Union* trvá $\mathcal{O}(n)$, jelikož způsobí $\mathcal{O}(n)$ amortizovaně konstantních operací s mikrostromy, spojovacími hranami a cestami a $\mathcal{O}(n/\log n)$ operací s makrostromem, které trvají $\mathcal{O}(\log n)$ amortizovaně každá.⟨27⟩

Cvičení: Zkuste pomocí dekompozice vyřešit následující problém: je dán strom, jehož každý vrchol může být označený. Navrhněte datovou strukturu, která bude umět v čase $\mathcal{O}(\log \log n)$ označit nebo odznačit vrchol a v čase $\mathcal{O}(\log n/\log \log n)$ najít nejbližšího označeného předchůdce.

Fredericksonova clusterizace

Mikro/makro-stromová dekompozice není jediný způsob, jak stromy rozkládat. Někdy se více hodí následující myšlenka:

Definice: *(Fredericksonova clusterizace)* Nechť $k \geq 1$ je přirozené číslo a T strom s maximálním stupněm nejvýše 3. Pak k-clusterizací stromu T nazveme libovolný rozklad $V_1 \cup \ldots \cup V_t$ množiny $V(T)$, pro který platí:

- Podgrafy stromu T indukované jednotlivými V_i jsou souvislé. Těmto podgrafům budeme říkat *clustery* a značit je C_i.
- Z každého clusteru vedou nejvýše 3 hrany do sousedních clusterů. Takovým hranám říkáme *vnější,* jejich počet je *vnější stupeň* clusteru $ed(C_i)$. Hrany uvnitř clusterů nazveme *vnitřní.*
- Nechť $|C_i|$ značí počet vrcholů clusteru C_i. Pak pro všechny clustery platí $|C_i| \leq k$ a pro clustery vnějšího stupně 3 dokonce $|C_i| = 1$.
- Žádné dva sousední clustery není možné sloučit.

Úmluva: Clustery vnějšího stupně 0 se nazývají *izolované,* stupně 1 *listové,* stupně 2 *cestové* a stupně 3 *větvicí.*

⟨27⟩ To je v průměru $\mathcal{O}(1)$ na operaci a dokonce i amortizovaně, pokud necháme inicializaci struktury, která je lineární, naspořit potenciál $\mathcal{O}(n)$, ze kterého budeme průběžně platit slučování v makrostromu.

Pozorování: Nechť C a D jsou sousední clustery (bez újmy na obecnosti $ed(C) \geq ed(D)$). Kdy je lze sloučit? Především $|C|+|D|$ musí být nejvýše rovno k. Pak mohou nastat následující případy:

- Pokud C i D jsou listové, lze je sloučit do jednoho izolovaného clusteru.
- Pokud C je cestový a D listový, lze je sloučit do listového clusteru.
- Pokud C i D jsou cestové, lze je sloučit do cestového clusteru.
- Pokud C je větvicí a D listový, lze je sloučit do cestového clusteru.
- V ostatních případech slučovat nelze, neboť by vznikl cluster vnějšího stupně 4 nebo stupně 3 s více než jedním vrcholem.

Věta: (Frederickson [22]) Každá k-clusterizace stromu T na n vrcholech obsahuje $\mathcal{O}(n/k)$ clusterů.

Důkaz: Pokud clusterizace obsahuje pouze izolované nebo listové clustery, pak je konstantně velká a tvrzení triviálně platí.

Pokud navíc obsahuje cestové clustery, musí být clustery propojeny do jedné cesty, která začíná a končí listovými clustery a ostatní clustery jsou cestové. Cestové clustery rozdělíme na velké (alespoň $k/2$ vrcholů) a malé (ostatní). Všimneme si, že malé spolu nemohou sousedit. Velkých je přitom nejvýše n/k, takže malých nejvýše $n/k + 1$.

Zbývá obecný případ, v němž jsou i větvicí clustery. Uvažme clusterizační strom S: jeho vrcholy odpovídají clusterům, hrany externím hranám mezi nimi. Tento strom zakořeňme v libovolném větvicím clusteru.

Pokud ve stromu S nahradíme každou nevětvící se cestu hranou, vznikne nějaký komprimovaný strom S'. V něm už jsou pouze listové clustery (coby listy) a větvicí clustery (jako vnitřní vrcholy).

Nyní si všimneme, že pro každý list ℓ stromu S platí, že tento cluster spolu s cestovými clustery nad ním (které se schovaly do hrany mezi ℓ a jeho otcem) musí mít velikost alespoň k. V opačném případě by totiž bylo možné tyto clustery společně s větvicím clusterem nad nimi sloučit do jediného clusteru, což by porušilo poslední podmínku z definice clusterizace.

Proto strom S' obsahuje nejvýše n/k listů. A jelikož všechny jeho vnitřní vrcholy mají alespoň 2 syny, musí být vnitřních vrcholů také nanejvýš n/k.

Zbývá započítat cestové clustery. Uvažme hranu e stromu S' a cestové clustery, které se do ní zkomprimovaly. Už víme, že je-li celková velikost těchto clusterů r, může jich být nanejvýš $2r/k + 1$. A jelikož clustery jsou disjunktní, v součtu přes všechny hrany e dostaneme $2n/k +$ počet hran stromu $S' = \mathcal{O}(n/k)$.

Clusterů všech typů je tedy dohromady $\mathcal{O}(n/k)$. ♡

Věta: (Frederickson [22]) Pro každé k lze k-clusterizaci stromu o n vrcholech najít v čase $\mathcal{O}(n)$.

Důkaz: Clusterizaci lze najít upraveným hledáním do hloubky, ale při tom je nutné řešit mnoho různých případů slučování clusterů. Místo toho použijeme následující hladový algoritmus.

Nejprve vytvoříme z každého vrcholu triviální cluster. Taková clusterizace splňuje všechny podmínky kromě poslední. Budeme tedy clustery hladově slučovat.

Pořídíme si frontu clusterů, u nichž jsme ještě nezkontrolovali slučitelnost. Na počátku do ní umístíme všechny clustery. Pak vždy odebereme cluster, prozkoumáme jeho sousedy a pokud mezi nimi je nějaký, s nímž lze slučovat, tak to provedeme. Nový cluster uložíme do fronty, oba staré z fronty odstraníme.

Všimneme si, že při každé kontrole poklesne velikost fronty o 1 – buďto jsme neslučovali a zmizel pouze kontrolovaný cluster, anebo slučovali, ale pak zmizely dva clustery a přibyl jeden. Jelikož kontrolu i sloučení zvládneme v konstantním čase, celý algoritmus doběhne v čase $\mathcal{O}(n)$. ♡

Použití: Předchozí variantu Union-Find problému bychom také mohli vyřešit nahrazením vrcholů stupně většího než 3 „kruhovými objezdy bez jedné hrany“, nalezením $(\log n)$-clusterizace, použitím bitové reprezentace množin uvnitř clusterů a přebarvovací struktury na hrany mezi clustery.

Stromoví předchůdci

Problém: *(Least Common Ancestor alias LCA)* 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.
- Vylepšení: Budeme stoupat z x a y střídavě. Tak potřebujeme jen lineárně mnoho kroků vzhledem ke vzdálenosti společného předchůdce.
- Předpočítáme všechny možnosti: předzpracování $\mathcal{O}(n^2)$, dotaz $\mathcal{O}(1)$.
- … co dál?

Věrni vtipům o matfyzácích a článku [7] převedeme raději tento problém na jiný.

Problém: *(Range Minimum Query alias RMQ)* Chceme předzpracovat posloupnost čísel $a_1, \dots a_n$ tak, abychom uměli rychle počítat $\min_{x \le i \le y} a_i$.[28]

Lemma: LCA lze převést na RMQ s lineárním časem na předzpracování a konstantním časem 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. $\mathrm{LCA}(x, y)$ pak bude nejvyšší vrchol mezi libovolnou návštěvou x a libovolnou návštěvou y. ♡

Triviální řešení RMQ:

- Předpočítáme všechny možné dotazy: předzpracování $\mathcal{O}(n^2)$, dotaz $\mathcal{O}(1)$.
- Pro každé i a $j \le \log n$ předpočítáme $m_{ij} = \min\{a_i, a_{i+1}, \dots, a_{i+2^j-1}\}$, čili minima všech bloků velkých jako nějaká mocnina dvojky. Když se

[28] Všimněte si, že pro sumu místo minima je tento problém velmi snadný.

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 $\mathcal{O}(n \log n)$.

My si ovšem všimneme, že náš převod z LCA vytváří dosti speciální instance problému RMQ, totiž takové, v nichž je $|a_i - a_{i+1}| = 1$. Takovým instancím budeme říkat RMQ±1 a budeme je umět řešit šikovnou dekompozicí.

Dekompozice 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 $\mathcal{O}(n/b \cdot \log(n/b)) = \mathcal{O}(n/\log n \cdot \log n) = \mathcal{O}(n)$ na předzpracování a $\mathcal{O}(1)$ na dotaz.

Tak jsme získali algoritmus pro RMQ±1 s konstantním časem na dotaz po lineárním předzpracování a výše zmíněným převodem i algoritmus na LCA se stejnými parametry. Ještě ukážeme, že převod může fungovat i v opačném směru, a tak můžeme získat i konstantní/lineární algoritmus pro obecné RMQ.

Definice: *Kartézský strom* pro posloupnost $a_1, \ldots, a_n$ je strom, jehož kořenem je minimum posloupnosti, tj. nějaké $a_j = \min_i a_i$, jeho levý podstrom je kartézský strom pro $a_1, \ldots, a_{j-1}$ a pravý podstrom kartézský strom pro $a_{j+1}, \ldots, a_n$.

Lemma: Kartézský strom je možné zkonstruovat v lineárním čase.

Důkaz: Použijeme inkrementální algoritmus. Vždy si budeme pamatovat kartézský strom pro již zpracované prvky a pozici posledního zpracovaného prvku v tomto stromu. Když přidáváme další prvek, hledáme místo, kam ho připojit, od tohoto označeného prvku nahoru. Povšimněme si, že vzhledem k potenciálu rovnému hloubce označeného prvku je časová složitost přidání prvku amortizovaně konstantní. ♡

Lemma: RMQ lze převést na LCA s lineárním časem na předzpracování a konstantním časem na převod dotazu.

Důkaz: Sestrojíme kartézský strom a RMQ převedeme na LCA v tomto stromu. ♡

Výsledky této podkapitoly můžeme shrnout do následující věty:

Věta: Problémy LCA i RMQ je možné řešit v konstantním čase na dotaz po předzpracování v lineárním čase.

Cvičení: Vymyslete jednodušší strukturu pro RMQ, víte-li, že všechny dotazy budou na intervaly stejné délky.

10. Suffixové stromy

V této kapitole popíšeme jednu pozoruhodnou datovou strukturu, pomocí níž dokážeme problémy týkající se řetězců převádět na grafové problémy a řešit je tak v lineárním čase.

Řetězce, trie a suffixové stromy

Definice:

Σ	konečná abeceda – množina znaků (znaky budeme značit latinskými písmeny)
Σ^*	množina všech slov nad Σ (slova budeme značit řeckými písmeny)
ε	prázdné slovo
$\|\alpha\|$	délka slova α
$\alpha\beta$	zřetězení slov α a β ($\alpha\varepsilon = \varepsilon\alpha = \alpha$)
α^R	slovo α napsané pozpátku
α je *prefixem* β	$\exists\gamma : \beta = \alpha\gamma$ (β začíná na α)
α je *suffixem* β	$\exists\gamma : \beta = \gamma\alpha$ (β končí na α)
α je *podslovem* β	$\exists\gamma, \delta : \beta = \gamma\alpha\delta$ (značíme $\alpha \subset \beta$)
α je *vlastním prefixem* β	je prefixem a $\alpha \neq \beta$ (analogicky vlastní suffix a podslovo)

Pozorování: Prázdné slovo je prefixem, suffixem i podslovem každého slova včetně sebe sama. Podslova jsou právě prefixy suffixů a také suffixy prefixů.

Definice: *Trie (Σ-strom)* pro konečnou množinu slov $X \subset \Sigma^*$ je orientovaný graf $G = (V, E)$, kde:

$V = \{\alpha : \alpha$ je prefixem nějakého $\beta \in X\}$,
$(\alpha, \beta) \in E \equiv \exists x \in \Sigma : \beta = \alpha x$.

Pozorování: Trie je strom s kořenem ε. Jeho listy jsou slova z X, která nejsou vlastními prefixy jiných slov z X. Hrany si můžeme představit popsané písmeny, o něž prefix rozšiřují, popisky hran na cestě z kořene do vrcholu α dávají právě slovo α.

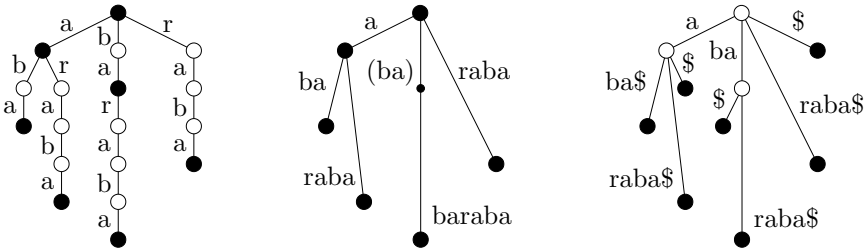
Definice: *Komprimovaná trie (Σ^+-strom)* vznikne z trie nahrazením maximálních nevětvících se cest hranami. Hrany jsou tentokrát popsané řetězci místo jednotlivými písmeny, přičemž popisky všech hran vycházejících z jednoho vrcholu se liší v prvním znaku. Vrcholům „uvnitř hran“ (které padly za oběť kompresi) budeme říkat *skryté vrcholy.*

Definice: *Suffixový strom (ST)* pro slovo $\sigma \in \Sigma^*$ je komprimovaná trie pro $X = \{\alpha :$ α je suffixem $\sigma\}$.

Pozorování: Vrcholy suffixového stromu (včetně skrytých) odpovídají prefixům suffixů slova σ, tedy všem jeho podslovům. Listy stromu jsou suffixy, které se v σ již nikde jinde nevyskytují (takovým suffixům budeme říkat *nevnořené*). Vnitřní vrcholy odpovídají *větvícím podslovům,* tedy podslovům $\alpha \subset \sigma$ takovým, že $\alpha a \subset \sigma$ i $\alpha b \subset \sigma$ pro nějaké dva různé znaky a, b.

Někdy může být nepraktické, že některé suffixy neodpovídají listům (protože jsou vnořené), ale s tím se můžeme snadno vypořádat: přidáme na konec slova σ nějaký znak \$, který se nikde jinde nevyskytuje. Neprázdné suffixy slova $\sigma\$$ odpovídají suffixům slova σ a žádný z nich nemůže být vnořený. Takový suffixový strom budeme značit ST\$.

Příklad:



Suffixy slova „baraba“: trie, suffixový strom, ST s dolarem

Nyní jak je to s konstrukcí suffixových stromů:

Lemma: Suffixový strom pro slovo σ délky n je reprezentovatelný v prostoru $\mathcal{O}(n)$.

Důkaz: Strom má $\mathcal{O}(n)$ listů a každý vnitřní vrchol má alespoň 2 syny, takže vnitřních vrcholů je také $\mathcal{O}(n)$. Hran je rovněž lineárně. Nálepky na hranách stačí popsat počáteční a koncovou pozicí v σ. ♡

Věta: Suffixový strom pro slovo σ délky n lze sestrojit v čase $\mathcal{O}(n)$.

Důkaz: Ve zbytku této kapitoly předvedeme dvě různé konstrukce v lineárním čase. ♡

Aplikace – co vše dokážeme v lineárním čase, když umíme lineárně konstruovat ST:

1. *Inverzní vyhledávání* (tj. předzpracujeme si v lineárním čase text a pak umíme pro libovolné slovo α v čase $\mathcal{O}(|\alpha|)$ rozhodnout, zda se v textu vyskytuje)⟨29⟩ – stačí sestrojit ST a pak jej procházet od kořene. Také umíme najít všechny výskyty (odpovídají suffixům, které mají jako prefix hledané slovo, takže stačí vytvořit ST\$ a vypsat všechny listy pod nalezeným vrcholem) nebo přímo vrátit jejich počet (předpočítáme si pomocí DFS pro každý vrchol, kolik pod ním leží listů).
2. *Nejdelší opakující se podslovo* – takové podslovo je v ST\$ nutně větvící, takže stačí najít vnitřní vrchol s největší *písmenkovou hloubkou* (tj. hloubkou měřenou ve znacích místo ve hranách).
3. *Histogram četností podslov délky k* – rozřízneme ST\$ v písmenkové hloubce k a spočítáme, kolik původních listů je pod každým novým.
4. *Nejdelší společné podslovo* slov α a β – postavíme ST pro slovo $\alpha\$_1\beta\$_2$, jeho listy odpovídají suffixům slov α a β. Takže stačí pomocí DFS najít

⟨29⟩ Čili přesný opak toho, co umí vyhledávací automat – ten si předzpracovává dotaz.

nejhlubší vnitřní vrchol, pod kterým se vyskytují listy pro α i β. Podobně můžeme sestrojit ST\$ pro libovolnou množinu slov.⟨30⟩

5. *Nejdelší palindromické podslovo* (tj. takové $\beta \subset \alpha$, pro něž je $\beta^R = \beta$) – postavíme společný ST\$ pro slova α a α^R. Postupně procházíme přes všechny možné středy palindromického podslova a všimneme si, že takové slovo je pro každý střed nejdelším společným prefixem podslova od tohoto bodu do konce a podslova od tohoto bodu pozpátku k začátku, čili nějakého suffixu α a nějakého suffixu α^R. Tyto suffixy ovšem odpovídají listům sestrojeného ST a jejich nejdelší společný prefix je nejbližším společným předchůdcem ve stromu, takže stačí pro strom vybudovat datovou strukturu pro společné předchůdce a s její pomocí dokážeme jeden střed prozkoumat v konstantním čase.
6. *Burrows-Wheelerova Transformace* [9] – jejím základem je lexikografické setřídění všech rotací slova σ, což zvládneme sestrojením ST pro slovo $\sigma\sigma$, jeho uříznutím v písmenkové hloubce $|\sigma|$ a vypsáním nově vzniklých listů v pořadí „zleva doprava“.

Cvičení: Zkuste vymyslet co nejlepší algoritmy pro tyto problémy bez použití ST.

Suffixová pole

V některých případech se hodí místo suffixového stromu používat kompaktnější datové struktury.

Notace: Pro slovo σ bude $\sigma[i]$ značit jeho i-tý znak (číslujeme od nuly), $\sigma[i : j]$ pak podslovo $\sigma[i]\sigma[i+1]\ldots\sigma[j-1]$. Libovolnou z mezí můžeme vynechat, proto $\sigma[i :]$ bude suffix od i do konce a $\sigma[: j]$ prefix od začátku do $j-1$. Pokud $j \leq i$, definujeme $\sigma[i : j]$ jako prázdné slovo, takže prázdný suffix můžeme například zapsat jako $\sigma[|\sigma| :]$.

$\mathrm{LCP}(\alpha, \beta)$ bude značit délku nejdelšího společného prefixu slov α a β, čili největší $i \leq |\alpha|, |\beta|$ takové, že $\alpha[: i] = \beta[: i]$.

Definice: *Suffixové pole (Suffix Array)* A_σ pro slovo σ délky n je posloupnost všech suffixů slova σ v lexikografickém pořadí. Můžeme ho reprezentovat například jako permutaci A čísel $0, \ldots, n$, pro níž $\sigma[A[0] :] < \sigma[A[1] :] < \ldots < \sigma[A[n] :]$.

Definice: *Pole nejdelších společných prefixů (Longest Common Prefix Array)* L_σ pro slovo σ je posloupnost, v níž $L_\sigma[i] := \mathrm{LCP}(A_\sigma[i], A_\sigma[i+1])$.

Věta: Suffixový strom pro slovo $\sigma\$$ a dvojici (A_σ, L_σ) na sebe lze v lineárním čase převádět.

Důkaz: Když projdeme ST($\sigma\$$) do hloubky, pořadí listů odpovídá posloupnosti A_σ a písmenkové hloubky vnitřních vrcholů v inorderu odpovídají L_σ. Naopak ST($\sigma\$$) získáme tak, že sestrojíme kartézský strom pro L_σ (získáme vnitřní vrcholy ST), doplníme do něj listy, přiřadíme jim suffixy podle A_σ a nakonec podle listů rekonstruujeme nálepky hran. ♡

⟨30⟩ Jen si musíme dát pozor, abychom si moc nezvětšili abecedu; jak moc si ji můžeme dovolit zvětšit, vyplyne z konkrétních konstrukcí.

Rekurzivní konstrukce

Ukážeme algoritmus, který pro slovo $\sigma \in \Sigma^*$ délky n sestrojí jeho suffixové pole a LCP pole v čase $\mathcal{O}(n + \mathrm{Sort}(n, \Sigma))$, kde $\mathrm{Sort}(\ldots)$ je čas potřebný pro setřídění n symbolů z abecedy Σ. V kombinaci s předchozími výsledky tedy dostaneme lineární konstrukci $\mathrm{ST}(\sigma)$ pro libovolnou fixní abecedu.

Algoritmus: (Konstrukce A a L podle Kärkkäinena a Sanderse [35])

1. Redukujeme abecedu na $1 \ldots n$: ve vstupním slovu je nejvýše n různých znaků, takže je stačí setřídit a přečíslovat.
2. Definujeme slova σ_0, σ_1, σ_2 následovně:

$$\begin{aligned}\sigma_0[i] &:= \langle \sigma[3i], \sigma[3i+1], \sigma[3i+2] \rangle \\ \sigma_1[i] &:= \langle \sigma[3i+1], \sigma[3i+2], \sigma[3i+3] \rangle \\ \sigma_2[i] &:= \langle \sigma[3i+2], \sigma[3i+3], \sigma[3i+4] \rangle\end{aligned}$$

   Všechna σ_k jsou slova délky $\approx n/3$ nad abecedou velikosti n^3. Dovolíme si mírně zneužívat notaci a používat symbol σ_k i jejich přepis do abecedy původní.
3. Zavoláme algoritmus rekurzivně na slovo $\sigma_0\sigma_1$, čímž získáme A_{01} a L_{01}. (Suffixy slova $\sigma_0\sigma_1$ odpovídají suffixům slov σ_0 a σ_1.)
4. Spočítáme pole P_0 a P_1, která nám budou říkat, kde se v A_{01} vyskytuje který suffix slov σ_0 a σ_1. Tedy $A_{01}[P_0[i]] = i$, $A_{01}[P_1[i]] = i + |\sigma_0|$. Jinými slovy, P_0 a P_1 budou části inverzní permutace k A_{01}. Všimněte si, že platí $P_i[x] < P_j[y]$ právě tehdy, když $\sigma_i[x :] < \sigma_j[y :]$, takže suffixy slov σ_0 a σ_1 od této chvíle umíme porovnávat v čase $\mathcal{O}(1)$.
5. Vytvoříme A_2 (suffixové pole pro σ_2): Jelikož $\sigma_2[i :] = \sigma[3i+2 :] = \sigma[3i+2]\sigma[3i+3 :] = \sigma[3i+2]\sigma_0[i+1 :]$, odpovídá lexikografické pořadí suffixů $\sigma_2[i :]$ pořadí dvojic $(\sigma[3i+2], P_0[i+1])$. Tyto dvojice ovšem můžeme setřídit dvěma průchody přihrádkového třídění.
6. Slijeme A_{01} a A_2 do A: sléváme dvě setříděné posloupnosti, takže stačí umět jejich prvky v konstantním čase porovnat:

$$\begin{aligned}\sigma_0[i :] < \sigma_2[j :] &\Leftrightarrow \sigma[3i :] < \sigma[3j+2 :] \\ &\Leftrightarrow \sigma[3i]\,\sigma_1[i :] < \sigma[3j+2]\,\sigma_0[j+1 :], \\ \sigma_1[i :] < \sigma_2[j :] &\Leftrightarrow \sigma[3i+1 :] < \sigma[3j+2 :] \\ &\Leftrightarrow \sigma[3i+1]\,\sigma[3i+2]\,\sigma_0[i+1 :] < \\ &\quad \sigma[3j+2]\,\sigma[3j+3]\,\sigma_1[j+1 :].\end{aligned}$$

   Pokaždé tedy porovnáme nejvýše dvě dvojice znaků a pak dvojici suffixů slov σ_0 a σ_1, k čemuž nám pomohou pole P_0 a P_1.
7. Dopočítáme L:
8. Pokud v A sousedí suffix slova $\sigma_{0,1}$ se suffixem slova $\sigma_{0,1}$, sousedí tyto dva suffixy i v A_{01}, takže jejich LCP najdeme přímo v L.

9. Setkají-li se dva suffixy slova σ_2, všimneme si, že $\sigma_2[i:] = \sigma[3i+2:] = \sigma[3i+2]\,\sigma_0[i+1:]$. $\mathrm{LCP}(\sigma_2[i:], \sigma_2[j:])$ je tedy buďto 0 (pokud $\sigma[3i+2] \neq \sigma[3j+2]$), nebo $1+3\cdot\mathrm{LCP}(\sigma_0[i+1:], \sigma_0[j+1:])$, případně totéž zvýšené o 1 nebo 2, pokud se trojznaky v σ_0 následující po LCP zčásti shodují. Přitom $\mathrm{LCP}(\sigma_0[p:], \sigma_0[q:])$ spočítáme pomocí L. Je to totiž minimum intervalu v L mezi indexy $P_0[p]$ a $P_0[q]$. To zjistíme v konstantním čase pomocí struktury pro intervalová minima.
10. Pokud se setká suffix slova $\sigma_{0,1}$ se suffixem slova σ_2, stačí tyto suffixy přepsat podobně jako v 6. kroku a problém tím opět převést na výpočet LCP dvou suffixů slov $\sigma_{0,1}$.

Analýza časové složitosti: Třídění napoprvé trvá $\mathrm{Sort}(n, \Sigma)$, ve všech rekurzivních voláních už je lineární (trojice čísel velikosti $\mathcal{O}(n)$ můžeme třídit tříprůchodovým přihrádkovým tříděním s $\mathcal{O}(n)$ přihrádkami). Z toho dostáváme:

$$T(n) = T(2/3 \cdot n) + \mathcal{O}(n), \text{ a tedy } T(n) = \mathcal{O}(n).$$

♡

Ukkonenova inkrementální konstrukce

Ukkonen popsal algoritmus [58] pro konstrukci suffixového stromu bez dolarů, pracující inkrementálně: Začne se stromem pro prázdné slovo a postupně na konec slova přidává další znaky a přepočítává strom. Každý znak přitom přidá v amortizovaně konstantním čase. Pro slovo σ tedy dokáže sestrojit ST v čase $\mathcal{O}(|\sigma|)$.

Budeme předpokládat, že hrany vedoucí z jednoho vrcholu je možné indexovat jejich prvními písmeny – to bezpečně platí, pokud je abeceda pevná; není-li, můžeme si pomoci hešováním.

Pozorování: Když slovo σ rozšíříme na σa, ST se změní následovně:

1. Všechny stávající vrcholy stromu (včetně skrytých) odpovídají podslovům slova σ. Ta jsou i podslovy σa, takže se budou nacházet i v novém stromu.
2. Pokud β bylo větvící slovo, zůstane nadále větvící – tedy vnitřní vrcholy ve stromu zůstanou.
3. Každý nový suffix βa vznikne prodloužením nějakého původního suffixu β. Přitom:
   - Pokud byl β nevnořený suffix (čili byl reprezentovaný listem), ani βa nebude vnořený. Z toho víme, že listy zůstanou listy, pouze jim potřebujeme prodloužit nálepky. Aby to netrvalo příliš dlouho, zavedeme *otevřené hrany,* jejichž nálepka říká „od pozice i do konce“. Listy se tak o sebe postarají samy.
   - Pokud β byl vnořený suffix (tj. vnitřní či skrytý vrchol):
     - Buď se βa vyskytuje v σ, a tím pádem je to vnořený suffix nového slova a strom není nutné upravovat;

- nebo se βa v σ nevyskytuje – tehdy pro něj musíme založit nový list s otevřenou hranou a případně i nový vnitřní vrchol, pod nímž bude tento list připojen.

Víme tedy, co všechno je při rozšíření slova potřeba ve stromu upravit. Zbývá vyřešit, jak to udělat efektivně.

Vnořené suffixy: Především potřebujeme umět rozpoznat, které suffixy jsou vnořené a které nikoliv. K tomu se hodí všimnout si, že vnořené suffixy tvoří souvislý úsek:

Lemma: Je-li α vnořený suffix slova σ a β je suffix slova α, pak β je v σ také vnořený.

Důkaz: Ve slově sigma se vyskytuje αx a αy pro nějaké dva různé znaky x a y. Každý z těchto výskytů přitom končí výskytem slova β, jednou následovaným x, podruhé y. ♡

Stačí si tedy zapamatovat nejdelší vnořený suffix slova σ. Tomu budeme říkat *aktivní suffix* a budeme ho značit $\alpha(\sigma)$. Libovolný suffix $\beta \subseteq \sigma$ pak bude vnořený právě tehdy, když $|\beta| \leq |\alpha(\sigma)|$.

Aktivní suffix tedy tvoří hranici mezi nevnořenými a vnořenými suffixy. Jak se tato hranice posune, když slovo σ rozšíříme? Na to je odpověď snadná:

Lemma: Pro každé σ, a platí: $\alpha(\sigma a)$ je suffixem $\alpha(\sigma)a$.

Důkaz: $\alpha(\sigma a)$ i $\alpha(\sigma)a$ jsou suffixy slova σa, a proto stačí porovnat jejich délky. Slovo $\beta :=$ „$\alpha(\sigma a)$ bez koncového a“ je vnořeným suffixem v σ, takže $|\beta| \leq |\alpha(\sigma)|$, a tedy také $|\alpha(\sigma a)| = |\beta a| \leq |\alpha(\sigma)a|$. ♡

Hranice se tedy může posouvat pouze doprava, případně zůstat na místě. Toho lze snadno využít.

Idea algoritmu: Udržujeme si $\alpha = \alpha(\sigma)$ a při přidání znaku a zkontrolujeme, zda αa je stále vnořený suffix. Pokud ano, nic se nemění, pokud ne, přidáme nový list a případně také vnitřní vrchol, α zkrátíme zleva o znak a testujeme dál.

Analýza: Po přidání jednoho znaku na konec slova σ provedeme amortizovaně konstantní počet úprav stromu (každá úprava slovo α zkrátí, po všech úpravách přidáme k α jediný znak). Tudíž stačí ukázat, jak provést každou úpravu v (amortizovaně) konstantním čase. K tomu potřebujeme šikovnou reprezentaci slova α, která bude umět efektivně prodlužovat zprava, zkracovat zleva a testovat existenci vrcholu ve stromu.

Definice: *Referenční pár* pro slovo $\alpha \subseteq \sigma$ je dvojice (π, τ), v níž π je vrchol stromu, τ libovolné slovo a $\pi\tau = \alpha$. Navíc víme, že $\tau \subseteq \sigma$, takže si τ stačí pamatovat jako dvojici indexů ve slově σ.

Referenční pár je *kanonický,* pokud neexistuje hrana vedoucí z vrcholu π s nálepkou, která by byla prefixem slova τ. (Všimněte si, že taková hrana se pozná podle toho, že první znak nálepky se shoduje s prvním znakem slova τ a nálepka není delší než slovo τ. Shodu ostatních znaků není nutné kontrolovat.)

Pozorování: Ke každému slovu $\alpha \subseteq \sigma$ existuje právě jeden kanonický referenční pár, který ho popisuje. To je ze všech referenčních párů pro toto slovo ten s nejdelším π (nejhlubším vrcholem).

Definice: Zpětná hrana $back(\pi)$ vede z vrcholu π do vrcholu, který je zkrácením slova π o jeden znak zleva. (Nahlédneme, že takový vrchol musí existovat: pokud je π vnitřní vrchol, pak je slovo π větvící, takže každý jeho suffix musí také být větvící, a tím pádem musí odpovídat nějakého vrcholu.)

Operace s referenčními páry: S referenčním párem (π, τ) popisujícím slovo α potřebujeme provádět následujicí operace:

- *Přidání znaku a na konec:* Připíšeme a na konec slova τ. To je jistě referenční pár pro αa, ale nemusí být kanonický. Přitom můžeme snadno ověřit, zda se αa ve stromu nachází, a případně operaci odmítnout.
- *Odebrání znaku ze začátku:* Pokud π není kořen stromu, položíme $\pi \leftarrow back(\pi)$ a zachováme τ. Pokud naopak je π prázdný řetězec, odebereme z τ jeho první znak (to lze udělat v konstantním čase, protože τ je reprezentované dvojicí indexů do σ).
- *Převedení na kanonický tvar:* Obě předchozí operace mohou vytvořit referenční pár, který není kanonický. Pokaždé proto kanonicitu zkontrolujeme a případně pár upravíme. Ověříme, zda hrana z π indexovaná písmenem a není dost krátká na to, aby byla prefixem slova τ. Pokud je, tak se po této hraně přesuneme dolů, čímž π prodloužíme a τ zkrátíme, a proces opakujeme. Jelikož tím pokaždé τ zkrátíme a kdykoliv jindy se τ prodlouží nejvýše o 1, mají všechny převody na kanonický tvar amortizovaně konstantní složitost.

Nyní již můžeme doplnit detaily, získat celý algoritmus a nahlédnout, že pracuje v amortizovaně konstantním čase.

Algoritmus podrobněji:

1. *Vstup:* $\alpha = \alpha(\sigma)$ reprezentovaný jako kanonický referenční pár (π, τ), T suffixový strom pro σ spolu s hranami *back*, nový znak a.
2. Zjistíme, jestli αa je přítomen ve stromu, a případně ho založíme:
3. Pokud $\tau = \varepsilon$: ($\alpha = \pi$ je vnitřní vrchol)
4. Vede-li z vrcholu π hrana s nálepkou začínající znakem a, pak je přítomen.
5. Nevede-li, není přítomen, a tak přidáme novou otevřenou hranu vedoucí z π do nového listu.
6. Pokud $\tau \neq \varepsilon$: (α je skrytý vrchol)
7. Najdeme hranu, po níž z π pokračuje slovo τ (která to je, poznáme podle prvního znaku slova τ).
8. Pokud v popisce této hrany po τ následuje znak a, pak je αa přítomen.
9. Pokud nenásleduje, tak nebyl přítomen, čili tuto hranu rozdělíme: přidáme na ni nový vnitřní vrchol, do nějž povede

hrana s popiskou τ a z něj zbytek původní hrany a otevřená hrana do nového listu.

10. Pokud αa nebyl přítomen, tak α zkrátíme a vrátíme se na krok 2.
11. Nyní víme, že αa již byl přítomen, takže upravíme referenční pár, aby popisoval αa.
12. Dopočítáme zpětné hrany (viz níže).
13. *Výstup:* $\alpha = \alpha(\sigma a)$ jako kanonický referenční pár (π, τ), T suffixový strom pro σa a jeho zpětné hrany *back*.

Zpětné hrany: Zbývá dodat, jak nastavovat novým vrcholům jejich zpětné hrany. To potřebujeme jen pro vnitřní vrcholy (na zpětné hrany z listů se algoritmus nikdy neodkazuje). Všimneme si, že pokud jsme založili vrchol, odpovídá tento vrchol vždy současnému α a zpětná hrana z něj povede do zkrácení slova α o znak zleva, což je přesně vrchol, který založíme (nebo zjistíme, že už existuje) v příští iteraci hlavního cyklu. V další iteraci ještě určitě nebudeme tuto hranu potřebovat, protože π vždy jen zkracujeme, a tak můžeme vznik zpětné hrany o iteraci zpozdit. Výroba zpětné hrany tedy bude také trvat jen konstantně dlouho.

11. Kreslení grafů do roviny

Rovinné grafy se objevují v nejrůznějších praktických aplikacích teorie grafů, a tak okolo nich vyrostlo značné množství algoritmů. I když existují výjimky, jako například již zmíněné hledání kostry rovinného grafu, většina takových algoritmů pracuje s konkrétním vnořením grafu do roviny (rovinným nakreslením).

Proto se zaměříme na algoritmus, který zadaný graf buďto vnoří do roviny, nebo se zastaví s tím, že graf není rovinný. Tarjan již v roce 1974 ukázal [32], že je to možné provést v lineárním čase, ale jeho algoritmus je poněkud komplikovaný. Od té doby se objevilo mnoho zjednodušení, prozatím vrcholících algoritmem Boyera a Myrvoldové [8], a ten zde ukážeme.

Zmiňme ještě, že existují i algoritmy, které vytvářejí rovinná nakreslení s různými speciálními vlastnostmi. Je to například Schnyderův algoritmus [47] generující v lineárním čase nakreslení, v němž jsou hrany úsečkami a vrcholy leží v mřížových bodech mřížky $(n-2) \times (n-2)$, a o něco jednodušší algoritmus Chrobaka a Payneho [13] kreslící do mřížky $(2n-4) \times (n-2)$. Oba ovšem pracují tak, že vyjdou z obyčejného rovinného nakreslení a upravují ho, aby splňovalo dodatečné podmínky.

DFS a bloky

Připomeňme si nejprve některé vlastnosti prohledávání do hloubky (DFS) a jeho použití k hledání komponent vrcholové 2-souvislosti *(bloků)*.

Definice: Prohledávání orientovaného grafu do hloubky rozdělí hrany do čtyř druhů:

- *stromové* – po nich DFS prošlo a rekurzivně se zavolalo; tyto hrany vytvářejí *DFS strom* orientovaný z kořene;
- *zpětné* – vedou do vrcholu, který leží na cestě z kořene DFS stromu do právě prohledávaného vrcholu; jinými slovy vedou do vrcholu, který se právě nachází na zásobníku;
- *dopředné* – vedou do již zpracovaného vrcholu ležícího v DFS stromu pod aktuálním vrcholem;
- *příčné* – zbývající hrany, které vedou do vrcholu již zpracovaného vrcholu v jiném podstromu.

Pozorování: Pokud DFS spustíme na neorientovaný graf a hranu, po níž jsme už jednou prošli, v opačném směru ignorujeme, existují pouze stromové a zpětné hrany. DFS strom tvoří kostru grafu.

Nyní už se zaměříme pouze na neorientované grafy ...

Lemma: Relace „Hrany e a f leží na společné kružnici“ (značíme $e \sim f$) je ekvivalence. Její třídy tvoří maximální 2-souvislé podgrafy (bloky); ekvivalenční třídy s jedinou hranou (mosty) považujeme za triviální bloky. Vrchol v je artikulace právě tehdy, sousedí-li s ním hrany z více bloků.

Pokud spustíme na graf DFS, je přirozené testovat, do jakých bloků patří stromové hrany sousedící s právě prohledávaným vrcholem v: stromová hrana uv, po které jsme do v přišli, a hrany vw_1 až vw_k vedoucí do podstromů T_1 až T_k (zpětné hrany

jsou vždy ekvivalentní s hranou uv). Pokud je $uv \sim vw_i$, musí existovat cesta z podstromu T_i do vrcholu u, která nepoužije právě testované hrany. Taková cesta ale může podstrom opustit pouze zpětnou hranou (stromová je zakázaná a dopředné ani příčné neexistují). Jinými slovy $uv \sim vw_i$ právě tehdy, když existuje zpětná hrana z podstromu T_i do vrcholu u nebo blíže ke kořeni.

Pokud některá dvojice vw_i, vw_j není ekvivalentní přes hranu uv (nebo pokud hrana uv ani neexistuje, což se nám v kořeni DFS stromu může stát), leží tyto hrany v různých blocích, protože T_i a T_j mohou být spojeny jen přes své kořeny (příčné hrany neexistují). Ze zpětných hran tedy získáme kompletní strukturu bloků.

Nyní si stačí rozmyslet, jak existenci zpětných hran testovat rychle. K tomu se bude hodit:

Definice: Je-li v vrchol grafu, pak:

- *Enter*(v) udává pořadí, v němž DFS do vrcholu v vstoupilo.
- *Ancestor*(v) je nejmenší z *Enter*ů vrcholů, do nichž vede z v zpětná hrana, nebo $+\infty$, pokud z v žádná zpětná hrana nevede.
- *LowPoint*(v) je minimum z *Ancestor*ů vrcholů ležících v podstromu pod v, včetně v samého.

Pozorování: *Enter*, *Ancestor* i *LowPoint* všech vrcholů lze spočítat během prohledávání, tedy také v lineárním čase.

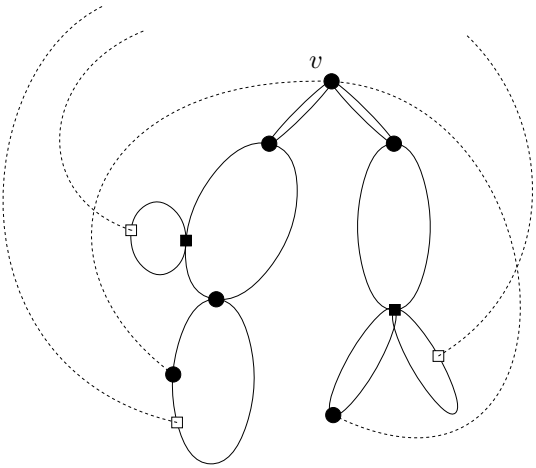
Rozpoznávání bloků a artikulací můžeme shrnout do následujícího lemmatu:

Lemma: Pokud v je vrchol grafu, u jeho otec a w jeho syn v DFS stromu, pak stromové hrany uv a vw leží v tomtéž bloku ($uv \sim vw$) právě tehdy, když *LowPoint*$(w) <$ *Enter*(v), a v je artikulace právě když některý z jeho synů w má *LowPoint*$(w) \geq$ *Enter*(v). Kořen DFS stromu je artikulace, právě když má více než jednoho syna.

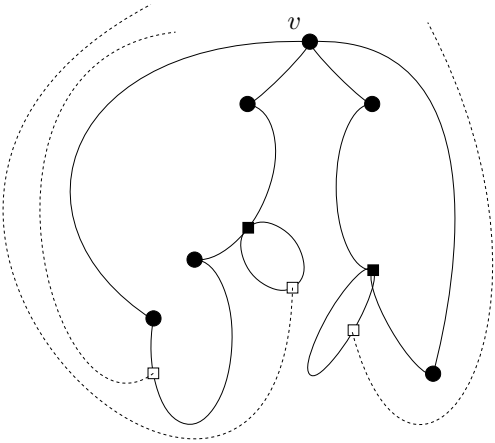
Postup kreslení

Graf budeme kreslit v opačném pořadí oproti DFS, tj. od největších *Enter*ů k nejmenším, a vždy si budeme udržovat blokovou strukturu již nakreslené části grafu, uspořádanou podle DFS stromu – každý blok bude mít svůj kořen, s výjimkou nejvyššího bloku je tento kořen současně artikulací v nadřazeném bloku. Aby se nám tato situace snadno reprezentovala, artikulace naklonujeme a každý blok dostane svou vlastní kopii artikulace.

Také budeme využívat toho, že nakreslení každého bloku, který není most, je ohraničeno kružnicí, a mosty zdvojíme, aby to pro ně platilo také. Navíc v libovolném nakreslení můžeme kterýkoliv blok spolu se všemi bloky ležícími pod ním překlopit podle kořenové artikulace, aniž bychom porušili rovinnost.



Před nakreslením zpětných hran ...



... po něm (čtverečky jsou externí vrcholy)

Všimněme si, že pokud vede z nějakého už nakresleného vrcholu ještě nenakreslená hrana, lze pokračovat po nenakreslených hranách až do kořene DFS stromu. Všechny vrcholy, ke kterým ještě bude potřeba něco připojit (takovým budeme říkat *externí* a hranám rovněž; za chvíli to nadefinujeme formálně), proto musí ležet v téže stěně dosud nakresleného podgrafu a bez újmy na obecnosti si vybereme, že to bude vnější stěna.

Základním krokem algoritmu tedy bude rozšířit nakreslení o nový vrchol v a o všechny hrany vedoucí z něj do jeho (již nakreslených) DFS-následníků. Stromové hrany půjdou nakreslit vždy, přidáme je jako triviální bloky (2-cykly) a nejsou-li to mosty, brzy se nějakou zpětnou hranou spojí s jinými bloky. Zpětné hrany byly až

donedávna externí, takže přidání jedné zpětné hrany nahradí cestu po okraji bloku touto hranou (tím vytvoří novou stěnu) a také může sloučit několik bloků do jednoho, jak je vidět z obrázků.

Bude se nám hodit, že čas potřebný na tuto operaci je přímo úměrný počtu hran, které ubyly z vnější stěny, což je amortizovaně konstanta.

Může se nám ale také stát, že zpětná hrana zakryje nějaký externí vrchol. Tehdy musíme některé bloky překlopit tak, aby externí vrcholy zůstaly venku. Potřebujeme tedy datové struktury, pomocí nichž bude možné překlápět efektivně a co víc, také rychle poznávat, kdy je překlápění potřebné.

Externí vrcholy

Jestliže z nějakého vrcholu v bloku B vede dosud nenakreslená hrana, musí být tento vrchol na vnější stěně, takže musí také zůstat na vnější stěně i vrchol, přes který je B připojen ke zbytku grafu. Proto externost nadefinujeme tak, aby pokrývala i tyto případy:

Definice: Vrchol w je *externí,* pokud buďto z w vede zpětná hrana do ještě nenakresleného vrcholu, nebo je pod w připojen externí blok, čili blok obsahující alespoň jeden externí vrchol. Ostatním vrcholům budeme říkat *interní.*

Jinými slovy platí, že vrchol w je externí při zpracování vrcholu v, pakliže $\mathit{Ancestor}(w) < \mathit{Enter}(v)$, nebo pokud pro některého ze synů x ležícího v jiném bloku je $\mathit{LowPoint}(x) < \mathit{Enter}(v)$. Druhá podmínka funguje díky tomu, že kořen bloku má v tomto bloku právě jednoho syna (jinak by existovala příčná hrana, což víme, že není pravda), takže minimum z *Ancestor*ů všech vrcholů ležících uvnitř bloku a v podřízených blocích je přesně *LowPoint* tohoto syna. Ve statickém grafu by se všechny testy redukovaly na $\mathit{LowPoint}(w)$, nám se ovšem bloková struktura průběžně mění, takže musíme uvažovat bloky v současném okamžiku. Proto si zavedeme:

Definice: $\mathit{BlockList}(w)$ je seznam všech bloků připojených v daném okamžiku pod vrcholem w, reprezentovaných jejich kořeny (klony vrcholu w) a jedinými syny kořenů. Tento seznam udržujeme setříděný vzestupně podle *LowPoint*ů synů.

Lemma: Vrchol w je externí při zpracování vrcholu v, pokud je buďto $\mathit{Ancestor}(w) < \mathit{Enter}(v)$, nebo první prvek seznamu $\mathit{BlockList}(w)$ má $\mathit{LowPoint} < \mathit{Enter}(v)$. Navíc seznamy *BlockList* lze udržovat v amortizovaně konstantním čase.

Důkaz: První část plyne přímo z definic. Všechny seznamy na začátku běhu algoritmu sestrojíme v lineárním čase přihrádkovým tříděním a kdykoliv sloučíme blok s nadřazeným blokem, odstraníme ho ze seznamu v příslušné artikulaci. ♡

Reprezentace bloků a překlápění

Pro každý blok si potřebujeme pamatovat vrcholy, které leží na hranici (některé z nich jsou externí, ale to už umíme poznat) a bloky, které jsou pod nimi připojené. Dále ještě vnitřní strukturu bloku včetně uvnitř připojených dalších bloků, ale jelikož žádné vnitřní vrcholy nejsou externí, vnitřek už neovlivní další výpočet a potřebujeme jej pouze pro vypsání výstupu.

Pro naše účely bude stačit zapamatovat si u každého bloku, jestli je oproti nadřazenému bloku překlopen. Tuto informaci zapíšeme do kořene bloku. Každý vrchol na hranici bloku pak bude obsahovat dva ukazatele na sousední vrcholy. Neumíme sice lokálně poznat, který ukazatel odpovídá kterému směru, ale když se nějakým směrem vydáme, dokážeme ho dodržet – stačí si vždy vybrat ten ukazatel, který nás nezavede do právě opuštěného vrcholu.

Každý vrchol si také bude pamatovat seznam svých sousedů, podle orientace bloku buďto v hodinkovém nebo opačném pořadí. Chceme-li přidat hranu, potřebujeme tedy znát absolutní orientaci, ale to půjde snadno, jelikož hrany přidáváme jen k vrcholům na hranici, poté co k nim po hranici dojdeme z kořene.

K překlopení bloku včetně všech podřízených bloků nám stačí invertovat bit v kořeni, pokud chceme překlopit jen tento blok, invertujeme bity i v kořenech všech podřízených bloků, jež najdeme obcházením hranice.

Na konci algoritmu spustíme dodatečný průchod, který všechny překlápěcí bity přenese ve směru od kořene k potomkům a určí tak absolutní orientaci všech seznamů sousedů i hranic.

Živý podgraf

Když nakreslíme nový vrchol v a z něj vedoucí stromové hrany, musíme obejít každý podstrom, ve vhodném pořadí nakreslit zpětné hrany do v a podle potřeby překlopit bloky. V podstromu ovšem může být mnoho bloků, které žádnou pozornost nevyžadují a běh algoritmu by zbytečně brzdily. Proto si před samotným kreslením označíme *živou* část grafu – to je část, kterou potřebujeme během kreslení navštívit; zbytku grafu se budeme snažit vyhýbat, aby nás nezdržoval.

Definice: Vrchol w je *živý,* pokud z něj buďto vede zpětná hrana do právě zpracovávaného vrcholu v, nebo pokud pod ním je připojen živý blok, tj. blok obsahující živý vrchol.

Živé vrcholy přitom mohou být i externí (pokud z nich vedou zpětné hrany jak do vrcholu v, tak do ještě nenakreslených vrcholů). Pokud nějaký vrchol není ani živý, ani externí, budeme ho nazývat *pasivní.*

Před procházením podstromů tedy nejprve probereme všechny zpětné hrany vedoucí do v a označíme živé vrcholy. Pro každou zpětnou hranu potřebujeme oživit vrchol, z nějž hrana vede, dále artikulaci, pod níž je tento blok připojen, a další artikulace na cestě do v. Tedy pokaždé, když vstoupíme do bloku (nějakým vrcholem na vnější stěně), potřebujeme nalézt kořen bloku. To uděláme tak, že začneme obcházet vnější stěnu oběma směry současně, až dojdeme v některém směru do kořene. Navíc si všechny vrcholy, přes něž jsme prošli, označkujeme a přiřadíme k nim rovnou ukazatel na kořen, tudíž po žádné části hranice neprojdeme vícekrát.⟨31⟩

⟨31⟩ Značky ani nebude potřeba mazat, když si u nich poznamenáme, který vrchol byl kořenem v okamžiku, kdy jsme značku vytvořili, a značky patřící ke starým kořenům budeme ignorovat, resp. přepisovat.

Výstupem této části algoritmu budou značky u živých vrcholů a u artikulací také seznamy podřízených živých bloků. Tyto seznamy budeme udržovat uspořádané tak, aby externí bloky následovaly po všech interních. To nám usnadní práci v hlavní části algoritmu.

Lemma: Pro každý kořen trvá značení živých vrcholů čas $\mathcal{O}(k + \ell)$, kde k je počet kreslených zpětných hran a ℓ počet hran, které zmizely z vnější stěny, čili amortizovaně konstanta.

Důkaz: Po žádné hraně neprojdeme více než jednou. Navíc alespoň polovina z těch, po nichž jsme prošli, zmizí z vnější stěny, takže hledání kořenů bloků trvá $\mathcal{O}(\ell)$. Pro každou zpětnou hranu označíme jeden vrchol jako živý a pak pokračujeme hledáním kořenů, které jsme již započítali. ♡

Kreslení zpětných hran

Nyní již máme vše připraveno – datové struktury, detekci externích vrcholů a označování živého podgrafu – a zbývá doplnit, jak algoritmus kreslí zpětné hrany. Jelikož zpětné hrany vedoucí do v nemohou způsobit sloučení bloků ležících pod v (na to jsou potřeba zpětné hrany vedoucí někam nad v a ty ještě nekreslíme), zpracováváme každý podstrom zvlášť. Vždy přidáme triviální blok pro stromovou hranu, pod něj připojíme blokovou strukturu zatím nakreslené části podstromu a vydáme se po hranici této struktury nejdříve jedním a pak druhým směrem.

Oba průchody vypadají následovně: Procházíme seznam vrcholů na hranici a pasivní vrcholy přeskakujeme. Pokud objevíme živý vrchol, nakreslíme vše, co z něj vede, případně se zanoříme do živých bloků, které jsou připojeny pod tímto vrcholem. Pokud objevíme externí vrchol (poté, co jsme ho případně ošetřili jako živý), procházení zastavíme, protože za externí vrchol již nemůžeme po této straně hranice nic připojit, aniž by se externí vrchol dostal dovnitř nakreslení.

Přitom se řídíme dvěma jednoduchými pravidly:

Pravidlo #1: V každém živém vrcholu zpracováváme nejdříve zpětné hrany do v, pak podřízené živé interní bloky a konečně podřízené živé externí bloky. (K tomu se nám hodí, že máme seznamy živých podřízených bloků setříděné.)

Pravidlo #2: Pokud vstoupíme do dalšího bloku, vybereme si směr, ve kterém budeme pokračovat, následovně: preferujeme směr k živému internímu vrcholu, pokud takový neexistuje, pak k živému externímu vrcholu. Pokud se tento směr liší od směru, ve kterém jsme zatím hranici obcházeli, blok překlopíme.

Časová složitost této části algoritmu je lineární ve velikosti živého podgrafu až na dvě výjimky. Jednou je konec prohledávání od posledního živého vrcholu k bodu zastavení, druhou pak vybírání strany hranice při vstupu do bloku. V obou můžeme procházet až lineárně mnoho pasivních vrcholů. Pomůžeme si ovšem snadno: kdykoliv projdeme souvislý úsek hranice tvořený pasivními vrcholy, přidáme pomocnou hranu, která tento úsek překlene. Můžeme ji dokonce přidat do nakreslení a podrozdělit si tak vnější stěnu.

Hotový algoritmus

Algoritmus (kreslení do roviny):

1. Pokud má graf více než $3n - 6$ hran, odmítneme ho rovnou jako nerovinný.
2. Prohledáme graf G do hloubky, spočteme *Enter, Ancestor* a *LowPoint* všech vrcholů.
3. Vytvoříme *BlockList* všech vrcholů přihrádkovým tříděním.
4. Procházíme vrcholy v pořadí klesajících *Enter*ů, pro každý vrchol v:
5. Nakreslíme všechny stromové hrany z v jako triviální bloky (2-cykly).
6. Označíme živý podgraf.
7. Pro každého syna vrcholu v obcházíme živý podgraf náležící k tomuto vrcholu v obou směrech a kreslíme zpětné hrany do v.
8. Zkontrolujeme, zda všechny zpětné hrany vedoucí do v byly nakresleny, a pokud ne, prohlásíme graf za nerovinný a skončíme.
9. Projdeme hotové nakreslení do hloubky a zorientujeme seznamy sousedů.

Věta: Tento algoritmus pro každý graf doběhne v čase $\mathcal{O}(n)$ a pokud byl graf rovinný, vydá jeho nakreslení, v opačném případě ohlásí nerovinnost.

Důkaz: První krok je korektní, jelikož pro všechny rovinné grafy je $m \leq 3n-6$; nadále tedy můžeme předpokládat, že $m = \mathcal{O}(n)$. Lineární časovou složitost kroků 4–6 a 9 jsme již diskutovali, kroky 7–8 jsou lineární ve velikosti živého podgrafu, a tedy také $\mathcal{O}(n)$. Nakreslení vydané algoritmem je vždy rovinné a všechny stromové hrany jsou vždy nakresleny, zbývá tedy ukázat, že zpětnou hranu můžeme nenakreslit, jen pokud graf nebyl rovinný. Tomu věnujeme zbytek kapitoly. ♡

Důkaz korektnosti

Lemma: Pokud existuje zpětná hrana, kterou algoritmus nenakreslil, graf na vstupu není rovinný.

Důkaz: Pro spor předpokládejme, že po zpracování vrcholu v existuje nějaká zpětná hrana wv, kterou algoritmus nenakreslil. Tedy přístup z v k w je v obou směrech blokován externími vrcholy. Rozborem případů ukážeme, že pokud jsme se důsledně řídili pravidly #1 a #2, může se to stát pouze v nerovinném grafu.

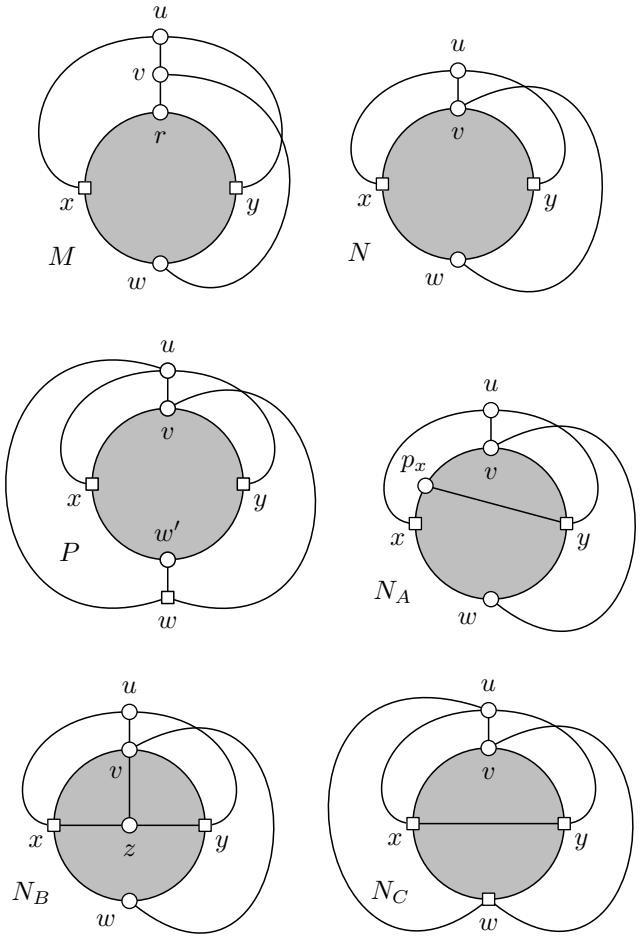
Překážející blok. Uvažme posloupnost bloků vedoucí od vrcholu v k w. V této posloupnosti se musí vyskytovat nějaký blok B, který má na obou stranách hranice aspoň jeden externí vrchol – jinak bychom totiž bloky popřeklápěli tak, abychom se dostali až do w. Označme x externí vrchol na levé straně hranice bloku B, y ten na pravé. Vrcholy x, y, v a w jsou zjevně navzájem různé.

Nadřazené bloky. Nejprve ukážeme, že kořenem bloku B musí být přímo vrchol v. Kdyby totiž mezi v a B ležely nějaké další bloky, našli bychom v grafu minor M z následujícího obrázku, který je isomorfní s grafem $K_{3,3}$. Do vrcholu u

jsme zkontrahovali ještě nenakreslenou část grafu, do v všechny bloky ležící nad B, do x a y případné podřízené externí bloky (díky nimž se x a y staly externími) a do w všechny bloky ležící mezi B a w.

Blok B je tedy nejvyšší a přímo obsahuje vrchol v. Této situaci odpovídá minor N, který je ovšem rovinný, takže na prokázání nerovinnosti grafu budeme muset zkoumat vnitřek bloku.

Podřízené bloky. Předtím ale rozebereme případ, kdy vrchol w neleží přímo v bloku B, nýbrž v nějakém podřízeném bloku, který je připojen pod nějakou artikulací $w' \in B$. Tento podřízený blok přitom nemůže být externí: kdyby byl, najdeme v grafu minor P obsahující $K_{3,3}$. Tedy není externí, takže vstoupíme-li během obcházení do w', pravidlo #2 zaručí, že dojdeme do w a díky pravidlu #1 vzápětí nakreslíme hranu wv. Tu jsme nenakreslili, takže jsme nedošli ani do w'. Můžeme proto bez újmy na obecnosti předpokládat, že hrana wv vede přímo z bloku B.



Existence plotu. Stačí se tedy omezit na situaci s jediným blokem B, v němž leží vrcholy v, w, x i y. Dokážeme nyní, že uvnitř bloku existuje *plot* – cesta mezi x a y, jejíž zbývající vrcholy neleží na hranici bloku.

Předpokládejme pro spor, že plot neexistuje. Před nakreslením zpětných hran vedoucích do v ještě blok B neexistoval a jeho hrany patřily do několika menších bloků. Speciálně víme, že w byla artikulace oddělující x od y, takže každá cesta mezi x a y musela procházet přes w. Proto v pořadí podle DFS musí ležet w před aspoň jedním z vrcholů x a y. Buď x nebo y tedy předtím musel ležet v nějakém podřízeném bloku pod w. A z nějakého takového bloku také musela vést zpětná hrana (tehdy ještě externí) do v, která později uzavřela blok B. K této hraně jsme se museli během obcházení dostat, a to je možné pouze přes w. Vrchol w tedy musel být navštíven a hrana wv nakreslena, což je spor.

Nejvyšší plot. Plot tedy existuje. Zvolíme mezi všemi ploty ten nejvyšší, čili nejbližší k v (rozmyslete si, že to je v rovinném nakreslení dobře definované). Označme p_x vrchol, v němž se plot napojuje na levou cestu z v do w, a obdobně p_y na pravé cestě. Rozmyslíme si, jak může situace vypadat.

A. Předně žádný z vrcholů p_x a p_y nemůže být blíž k v než x a y. Pokud by bez újmy na obecnosti p_x ležel mezi v a x, našli bychom v grafu minor N_A obsahující $K_{3,3}$: cestu mezi y a p_y jsme zkontrahovali do y, celý plot jsme zkontrahovali do hrany p_xy, ostatní vrcholy jsou utvořeny stejně jako v předchozích minorech.

B. Dále ukážeme, že plot může být připojen k v pouze přes p_x a p_y. V opačném případě by se v grafu vyskytoval minor N_B obsahující $K_{3,3}$: do x jsme zkontrahovali cestu mezi x a p_x, podobně do y cestu mezi y a p_y.

C. Nakonec se přesvědčíme, že na dolní cestě z x do y přes w nemůže ležet žádný externí vrchol. To by totiž způsobovalo minor N_C isomorfní s K_5: do w jsme zkontrahovali cestu mezi externím vrcholem a w a také všechny případné podřízené bloky až k externí hraně.

Vnitřek bloku B. Nyní uvažujme, jak graf vypadal před nakreslením vrcholu v. I tehdy musel plot společně s dolní cestou ležet v jednom bloku. Tento blok musel být z jedné strany připojený nenakreslenými hranami k v přes p_x, z druhé přes p_y (a díky vlastnosti **B** nikudy jinudy). Říkejme tomuto bloku B' a označme $r_1 \in \{p_x, p_y\}$ jeho kořen a r_2 druhý z vrcholů p_x, p_y.

Jelikož jsme nakonec nakreslili zpětnou hranu uzavírající blok B, museli jsme někdy dojít do r_1. V tomto okamžiku:

- Díky **A:** r_2 je předkem x nebo y (případně je takovému vrcholu roven), takže je nyní externí.
- Díky **B:** půjdeme-li z r_1 po plotu, nejbližší „zajímavý“ vrchol bude r_2.
- Díky **C:** žádný vrchol na dolní cestě není externí.

Při vstupu do r_1 tedy plot vede k externími vrcholu, zatímco dolní cesta k internímu. Podle pravidla #2 si algoritmus musí vybrat dolní cestu, kde ho nic nezastaví, takže dojde až do w a nakreslí zpětnou hranu wv. To je opět spor. ♡

Poznámka: Podle tohoto důkazu bychom také mohli v lineárním čase v každém nerovinném grafu nalézt Kuratowského podgraf, dokonce také v $O(n)$, jelikož když je $m > 3n - 6$, můžeme se omezit na libovolných $3n - 5$ hran, které určitě tvoří nerovinný podgraf.

12. Pravděpodobnostní algoritmus na řezy

Nahlédněme alespoň jednou kapitolou do světa pravděpodobnostních algoritmů. Není totiž výjimkou, že s pomocí generátoru náhodných čísel vyřešíme některé grafové problémy daleko snáze a často také efektivněji, než to dovedeme deterministicky. Pravděpodobnostní přístup si předvedeme na Kargerově-Steinově algoritmu [34] pro hledání minimálního řezu v neohodnoceném neorientovaném grafu. Připomeňme, že s deterministickými algoritmy jsme zatím dosáhli časové složitosti $\mathcal{O}(n^{5/3}m)$ pomocí toků nebo $\mathcal{O}(nm)$ Nagamochiho-Ibarakiho algoritmem.

Náhodné kontrakce

Uvažujme nejdříve následující algoritmus, který náhodně vybírá hrany a kontrahuje je, dokud počet vrcholů neklesne na ℓ. (Konkrétní hodnotu ℓ zvolíme později.)

Algoritmus CONTRACT(G_0, ℓ):

1. $G \leftarrow G_0$.
2. Dokud $n > \ell$:
3. Vybereme hranu $e \in E$ rovnoměrně náhodně.
4. $G \leftarrow G/e$ (kontrahujeme hranu e, smyčky odstraňujeme, paralelní hrany ponecháme).
5. Vrátíme jako výsledek graf G.

Jaká je pravděpodobnost, že výsledný graf G má stejně velký minimální řez jako zadaný graf G_0? Všimněme si nejprve, že každý řez v grafu G/e je i řezem v grafu G (až na přeznačení hran při kontrakci, ale předpokládejme, že hrany mají nějaké identifikátory, které kontrakce zachovává). Podobně je-li v grafu G řez neobsahující hranu e, odpovídá mu stejně velký řez v G/e. Velikost minimálního řezu tedy kontrakcí nikdy neklesne – může pouze stoupnout, pokud skontrahujeme hranu ležící ve všech minimálních řezech,

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.

Nabízí se otázka, k čemu je dobrý algoritmus, který vydá správný výsledek s pravděpodobností na řádu $1/n^2$. To opravdu není mnoho, ale stejně jako mnoho jiných randomizovaných algoritmů i tento můžeme iterovat: 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.

Implementace

Odbočme na chvíli k implementačním záležitostem. Jak reprezentovat graf, abychom stihli rychle provádět kontrakce? Bude nám stačit obyčejná matice sousednosti, v níž pro každou dvojici vrcholů budeme udržovat, kolik paralelních hran mezi těmito vrcholy vede. Pokud chceme kontrahovat hranu uv, stačí projít všechny hrany vedoucí (řekněme) z vrcholu u a zařadit je k vrcholu v. To zvládneme v čase $\mathcal{O}(n)$ na jednu kontrakci.

Pro náhodný výběr hrany budeme udržovat pole stupňů vrcholů, vybereme náhodně vrchol v s pravděpodobností úměrnou stupni a poté projitím příslušného řádku matice sousednosti druhý vrchol v s pravděpodobností úměrnou počtu hran mezi u a v. To opět trvá čas $\mathcal{O}(n)$.

Procedura Contract tedy pracuje v čase $\mathcal{O}((n-\ell)\cdot n)$ a celý K-krát ziterovaný algoritmus v $\mathcal{O}(Kn^2)$. Pokud se spokojíme s převráceně polynomiální pravděpodobností chyby, nalezneme minimální řez v čase $\mathcal{O}(n^4 \log n)$.

Kargerův-Steinův algoritmus

Předchozí prostinký algoritmus můžeme ještě podstatně vylepšit. 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

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

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.

Hotový algoritmus bude vypadat následovně:

Algoritmus MinCut(G):

1. Pokud $n < 7$, najdeme minimální řez hrubou silou.
2. $\ell \leftarrow \lceil n/\sqrt{2}+1 \rceil$.⟨32⟩
3. $C_1 \leftarrow$ MinCut(Contract(G, ℓ)).
4. $C_2 \leftarrow$ MinCut(Contract(G, ℓ)).
5. Vrátíme menší z řezů C_1, C_2.

Jakou bude mít tento algoritmus časovou složitost? Nejprve odhadněme hloubku 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 proceduru Contract, která spotřebuje čas $\mathcal{O}((n/2^{i/2})^2) = \mathcal{O}(n^2/2^i)$. Součet přes celou hladinu tedy činí $\mathcal{O}(n^2)$ a přes všechny hladiny $\mathcal{O}(n^2 \log n)$. Oproti původnímu kontrakčnímu algoritmu jsme si tedy moc nepohoršili.

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:

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

⟨32⟩ To je méně než n, kdykoliv $n \geq 7$.

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: 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 $c > 0$.

Cvičení

1. Zvolte lepší reprezentaci grafu, abyste prostorovou složitost snížili z $\Theta(n^2)$ na $\mathcal{O}(m)$. Může se tím zmenšit časová složitost?
2. Dokažte, že z našeho rozboru pravděpodobnosti chyby plyne, že v každém grafu je nejvýše $\mathcal{O}(n^2)$ minimálních řezů.
3. Ukažte, jak algoritmus upravit pro grafy s ohodnocenými hranami.

13. Nejkratší cesty

Problém hledání nejkratší cesty v (obvykle ohodnoceném orientovaném) grafu provází teorii grafových algoritmů od samých počátků. Základní algoritmy pro hledání cest jsou nedílnou součástí základních kursů programování a algoritmů, my se budeme věnovat zejména různým jejich vylepšením.

Uvažujme tedy nějaký orientovaný graf, jehož každá hrana e je opatřena *délkou* $\ell(e) \in \mathbb{R}$. Množině hran (třeba sledu nebo cestě) pak přiřadíme délku rovnou součtu délek jednotlivých hran.

Pro vrcholy u, v definujeme jejich *vzdálenost* $d(u, v)$ jako nejmenší možnou délku cesty z u do v (jelikož cest je v grafu konečně mnoho, minimum vždy existuje). Pokud z u do v žádná cesta nevede, položíme $d(u, v) := \infty$.

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.

Překvapivě, tak obecně, jak jsme si uvedené problémy definovali, je neumíme řešit v polynomiálním čase: pro grafy, které mohou obsahovat hrany záporných délek bez jakýchkoliv omezení, je totiž hledání nejkratší cesty NP-těžké (lze na něj snadno převést existenci hamiltonovské cesty). Všechny známé polynomiální algoritmy totiž místo nejkratší cesty hledají nejkratší sled – nijak nekontrolují, zda cesta neprojde jedním vrcholem vícekrát.

Naštěstí pro nás je to v grafech bez cyklů záporné délky totéž: pokud se v nalezeném sledu vyskytne cyklus, můžeme jej „vystřihnout“ a tím získat sled, který není delší a který má méně hran. Každý nejkratší sled tak můžeme upravit na stejně dlouhou cestu. V grafech bez záporných cyklů je tedy jedno, zda hledáme sled nebo cestu; naopak vyskytne-li se záporný cyklus dosažitelný z počátečního vrcholu, nejkratší sled ani neexistuje.

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 $u \dots w$ vyměnit za kratší.)

Díky této *prefixové vlastnosti* můžeme pro každý vrchol u sestrojit jeho *strom nejkratších cest* $\mathcal{T}(u)$. To je nějaký podgraf grafu G, který má tvar stromu zakořeněného v u a orientovaného směrem od kořene, a platí pro něj, že pro každý vrchol v je (jediná) cesta z u do v v tomto stromu jednou z nejkratších cest z u do v v původním grafu.

Pozorování: 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. ♡

Zbývá se dohodnout, v jakém tvaru mají naše algoritmy vydávat výsledek. U problémů typu 1-1 je nejjednodušší vypsat celou cestu, u 1-n můžeme jako výstup vydat strom nejkratších cest z daného počátku (všimněte si, že stačí uvést předchůdce každého vrcholu), u n-n vydáme strom nejkratších cest pro každý ze zdrojových vrcholů.

Často se ovšem ukáže, že podstatná část problému se skrývá v samotném výpočtu vzdáleností a sestrojení předchůdců je triviálním rozšířením algoritmu. Budeme tedy obvykle jen počítat vzdálenosti a samotnou rekonstrukci cest ponecháme čtenáři jako snadné cvičení.

Relaxační algoritmus

Začněme problémem 1-n a označme u výchozí vrchol. Většina známých algoritmů funguje tak, že pro každý vrchol v udržují ohodnocení $h(v)$, které v každém okamžiku odpovídá délce nějakého sledu z u do v. Postupně toto ohodnocení upravují, až se z něj stane vzdálenost $d(u, v)$ a algoritmus se může zastavit.

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) a *uzavřen* (už jsme relaxovali a není potřeba znovu).

Náš algoritmus bude fungovat následovně:

1. $h(*) \leftarrow \infty$, $h(u) \leftarrow 0$.
2. $stav(*) \leftarrow$ *neviděn*, $stav(u) \leftarrow$ *otevřen*.
3. Dokud existují otevřené vrcholy, opakujeme:
4. $\quad v \leftarrow$ libovolný otevřený vrchol.
5. $\quad stav(v) \leftarrow$ *uzavřen*.
6. $\quad$ Relaxujeme v:
7. $\quad\quad$ Pro všechny hrany vw opakujeme:
8. $\quad\quad\quad$ Je-li $h(w) > h(v) + \ell(v, w)$:
9. $\quad\quad\quad\quad h(w) \leftarrow h(v) + \ell(v, w)$.
10. $\quad\quad\quad\quad stav(w) \leftarrow$ *otevřen*.

11. Vrátíme výsledek $d(u, v) = h(v)$ pro všechna v.

Podobně jako u minimálních koster, i zde se jedná o meta-algoritmus, protože v kroku 4 nespecifikuje, který z otevřených vrcholů vybírá. Přesto ale můžeme dokázat několik zajímavých tvrzení, která na konkrétním způsobu výběru nezávisejí.

Věta: 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:

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. ♡

Dokázali jsme tedy, že meta-algoritmus pro libovolnou implementaci kroku 4 spočítá správné vzdálenosti.

Cvičení:

- Nechť do algoritmu doplníme udržování předchůdců tak, že v kroku 9 přenastavíme předchůdce vrcholu w na vrchol v. Dokažte, že předchůdci dosažitelných vrcholů budou tvořit strom a že tento strom bude stromem nejkratších cest z vrcholu u.
- Dokažte, že pro graf, v němž je alespoň jeden záporný cyklus dosažitelný z počátečního vrcholu, se algoritmus nezastaví a ohodnocení všech vrcholů na cyklu postupně klesnou libovolně hluboko. Nedosažitelné záporné cykly chod algoritmu samozřejmě nijak neovlivní.

Bellmanův-Fordův-Mooreův algoritmus

Bellman [5], Ford [21] a Moore objevili nezávisle na sobě algoritmus (říkejme mu BFM), který lze v řeči našeho meta-algoritmu formulovat takto: Otevřené vrcholy udržujeme ve frontě (vždy relaxujeme vrchol na počátku fronty, nově otevírané zařazujeme na konec). Co toto pravidlo způsobí?

Věta: Časová složitost algoritmu BFM činí $\mathcal{O}(nm)$.

Důkaz: Běh algoritmu rozdělíme na fáze. Nultá fáze sestává z vložení vrcholu u do fronty. V $(i+1)$-ní fázi relaxujeme ty vrcholy, které byly do fronty uloženy během i-té fáze.

Jelikož relaxace vrcholu trvá lineárně se stupněm vrcholu a každý vrchol se dané fáze účastní nejvýše jednou, trvá jedna fáze $\mathcal{O}(m)$. Zbývá ukázat, že fází provedeme nejvýše n.

Indukcí dokážeme, že na konci i-té fáze je každé ohodnocení $h(v)$ shora omezeno délkou nejkratšího z uv-sledů o nejvýše i hranách. Pro $i = 0$ to triviálně platí. 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 $h(v) \leq h(w) + \ell(w,v) \leq \ell(P') + \ell(w,v) = \ell(P)$. ♡

Cvičení:

- Ukažte, že asymptoticky stejné časové složitosti by dosáhl algoritmus, který by vrcholy očísloval $v_1, \ldots, v_n$ a opakovaně by je v tomto pořadí relaxoval tak dlouho, dokud by se ohodnocení měnila.
- Jak algoritmus upravit, aby v i-té fázi spočítal minimální délky sledů o právě i hranách?
- Jak lze algoritmus BFM využít k nalezení záporného cyklu?

Dijkstrův algoritmus

Pokud jsou všechny délky hran nezáporné, můžeme použít efektivnější pravidlo pro výběr vrcholu navržené Dijkstrou [19]. To říká, že vždy relaxujeme ten z otevřených vrcholů, jehož ohodnocení je nejmenší.

Věta: 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. ♡

Přímočará implementace Dijkstrova algoritmu by tedy pokaždé v čase $\mathcal{O}(n)$ vybrala otevřený vrchol s nejmenším ohodnocením, v čase $\mathcal{O}(n)$ ho relaxovala a toto by se opakovalo nejvýše n-krát. Algoritmus by tudíž doběhl v čase $\mathcal{O}(n^2)$, což je pro husté grafy zajisté optimální. Zkusíme nyní zrychlit výpočet na řídkých grafech.

Všechny relaxace trvají dohromady $\mathcal{O}(\sum_v \deg(v)) = \mathcal{O}(m)$, takže úzkým hrdlem je vybírání minima. Použijeme pro něj vhodnou datovou strukturu, v níž budeme udržovat množinu všech otevřených vrcholů spolu s jejich ohodnoceními. Od datové struktury potřebujeme, aby uměla operace *Insert* (vložení vrcholu), *ExtractMin* (nalezení a smazání minima) a *Decrease* (snížení hodnoty vrcholu). První dvě operace přitom voláme nejvýše n-krát a operaci *Decrease* nejvýše m-krát. Celý algoritmus tedy doběhne v čase

$$\mathcal{O}(nT_I(n) + nT_E(n) + mT_D(n)),$$

kde $T_I(n)$, $T_E(n)$ a $T_D(n)$ jsou časové složitosti jednotlivých operací na struktuře o nejvýše n prvcích (stačí amortizovaně).

Jaké možnosti máme pro volbu struktury?

- *pole* – *Insert* a *Decrease* stojí konstantu, *ExtractMin* trvá $\mathcal{O}(n)$, celkem tedy $\mathcal{O}(n^2)$.
- *(binární) halda* – všechny tři operace umíme provést v čase $\mathcal{O}(\log n)$, takže celkem $\mathcal{O}(m \log n)$. To je pro husté grafy horší, pro řídké lepší.
- *k-regulární halda* – pokud haldu upravíme tak, že každý prvek bude mít až k synů, hloubka haldy klesne na $\mathcal{O}(\log_k n)$. Operace „vybublávající“ prvky směrem nahoru, což je *Insert* a *Decrease*, se zrychlí na $\mathcal{O}(\log_k n)$. Ovšem *ExtractMin* potřebuje zkoumat všechny syny každého navštíveného prvku, takže se zpomalí na $\mathcal{O}(k \log_k n)$.

  Celková složitost tedy vyjde $\mathcal{O}(nk \log_k n + m \log_k n)$. Oba členy se vyrovnají pro $k = m/n$, čímž získáme $\mathcal{O}(m \log_{m/n} n)$. Člen $\log_{m/n} n$ je přitom $\mathcal{O}(1)$, kdykoliv je $m \geq n^{1+\varepsilon}$ pro nějaké $\varepsilon > 0$, takže pro dostatečně husté grafy jsme získali lineární algoritmus.

  (Všimněte si, že pro $m \approx n^2$ algoritmus zvolí $k \approx n$, takže halda degeneruje na jediné patro, tedy na pole, které se opravdu ukázalo být optimální volbou pro husté grafy.)
- *Fibonacciho halda* [25] – *Insert* a *Decrease* stojí $\mathcal{O}(1)$, *ExtractMin* má složitost $\mathcal{O}(\log n)$ [vše amortizovaně]. Dijkstrův algoritmus proto doběhne v čase $\mathcal{O}(m + n \log n)$. To je lineární pro grafy s hustotou $\Omega(\log n)$. Téže složitosti operací dosahují i jiné, méně známé haldy [28, 20], které mohou být v praxi výrazně rychlejší.
- *Monotónní haldy* – můžeme použít nějakou jinou haldu, která využívá toho, že posloupnost odebíraných prvků je neklesající. Pro celá čísla na RAMu to může být například Thorupova halda [55] se složitostí $\mathcal{O}(\log \log n)$ u operace *ExtractMin* a $\mathcal{O}(1)$ u ostatních operací. Dijkstra tedy běží v $\mathcal{O}(m + n \log \log n)$.
- *Datové struktury pro omezené universum* – prozkoumáme vzápětí.

Cvičení:

- Najděte příklad grafu se zápornými hranami (ale bez záporných cyklů), na kterém Dijkstrův algoritmus selže.
- Rozmyslete si, že pokud nevyužijeme nějaké speciální vlastnosti čísel (celočíselnost, omezený rozsah), je mez $\mathcal{O}(m + n \log n)$ nejlepší možná, protože Dijkstrovým algoritmem můžeme třídit n-tici čísel. Thorup dokonce dokázal [57], že z každého třídícího algoritmu se složitostí $\mathcal{O}(nT(n))$ můžeme odvodit haldu se složitostí mazání $\mathcal{O}(T(n))$.
- Jsou-li délky hran celočíselné, můžeme se na Dijkstrův algoritmus dívat i takto: Představme si, že každou hranu nahradíme cestou tvořenou příslušným počtem hran jednotkové délky a na vzniklý neohodnocený graf spustíme prohledávání do šířky. To samozřejmě vydá správný výsledek, ale poměrně pomalu, protože bude většinu času trávit posouváním vlny „uvnitř“ původních hran. Můžeme si tedy pro každou původní hranu nařídit „budík“, který nám řekne, za kolik posunutí vlny dospějeme na její konec. Dokažte, že tento algoritmus je ekvivalentní s Dijkstrovým.

Celočíselné délky

Uvažujme nyní grafy, v nichž jsou všechny délky hran 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.

Použijeme jednoduchou přihrádkovou strukturu: pole indexované hodnotami od 0 do nL, jeho i-tý prvek bude obsahovat seznam vrcholů, jejichž ohodnocení je rovno i. Operace *Insert* a *Decrease* zvládneme v konstantním čase, budeme-li si u každého prvku pamatovat jeho polohu v seznamu. Operace *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 $\mathcal{O}(nL + m)$.

Prostorová složitost $\mathcal{O}(nL + m)$ je nevalná, ale můžeme ji jednoduchým trikem snížit: Všimneme si, že všechna konečná ohodnocení vrcholů nemohou být větší než aktuální minimum zvětšené o L. Jinými slovy všechny neprázdné přihrádky se nacházejí v úseku pole dlouhém $L + 1$, takže stačí indexovat modulo $L + 1$. Pouze si musíme dávat pozor, abychom správně poznali, kdy se struktura vyprázdnila, což zjistíme například pomocí počítadla otevřených vrcholů. Čas si asymptoticky nezhoršíme, prostor klesne na $\mathcal{O}(L + m)$.

Podobný trik můžeme použít i pro libovolnou jinou datovou strukturu: rozsah čísel rozdělíme na „okna“ velikosti L (v i-tém okně jsou čísla $iL, iL + 1, \ldots, (i + 1)L - 1$). V libovolné chvíli mohou tedy být neprázdná nejvýše dvě okna. Stačí nám proto pořídit si dvě struktury pro ukládání čísel v rozsahu $0, \ldots, L - 1$ – jedna z nich bude reprezentovat aktuální okno (to, v němž leželo minulé minimum), druhá okno bezprostředně následující. Minimum mažeme z první struktury; pokud už je prázdná, obě struktury prohodíme. Operace *Insert* podle hodnoty určí, do které struktury se

má vkládat. S operací *Decrease* je to složitější – hodnota z vyšší struktury může přeskočit do té nižší, ale v takovém případě ji ve vyšší struktuře vymažeme (to je *Decrease* na $-\infty$ následovaný *ExtractMin*em) a do té nižší vložíme. To se každému prvku může přihodit nejvýše jednou, takže stále platí, že se každý prvek účastní $\mathcal{O}(1)$ vložení a $\mathcal{O}(1)$ extrakcí minima.

Ukázali jsme tedy, že pro naše potřeby postačuje struktura schopná uchovávání čísel menších nebo rovných L.

Nabízí se použít van Emde-Boasův strom z kapitoly o výpočetních modelech. Ten dosahuje složitosti $\mathcal{O}(\log \log L)$ pro operace *Insert* a *ExtractMin*, operaci *Decrease* musíme překládat na *Delete* a *Insert*. Celková složitost Dijkstrova algoritmu vyjde $\mathcal{O}(L + m \log \log L)$, přičemž čas L spotřebujeme na inicializaci struktury (té se lze za jistých podmínek vyhnout, viz zmíněná kapitola).

Vraťme se ale ještě k využití přihrádek...

Víceúrovňové přihrádky

Podobně jako u třídění čísel, i zde se vyplácí stavět přihrádkové struktury víceúrovňově (původně popsáno Goldbergem a Silversteinem [26]). 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 $\mathcal{O}(m + n \cdot \frac{\log L}{\log\log L})$.

HOT Queue – kombinace přihrádek s haldou

Cherkassky, Goldberg a Silverstein [12] si všimli, že ve víceúrovňových přihrádkových strukturách platíme příliš mnoho za úrovně, ve kterých se za dobu jejich existence objeví jen malé množství prvků. Navrhli tedy ukládat prvky z „malých“ úrovní do společné haldy. Výsledné struktuře se říká HOT (Heap-on-Top) Queue. My si předvedeme její upravenou variantu (v té původní se skrývá několik chyb).

Pořídíme si haldu, řekněme Fibonacciho, a navíc ke každé úrovni počítadlo udávající, kolik prvků z této úrovně jsme uložili do haldy. Dokud toto počítadlo nepřeroste nějaký parametr H, přihrádky nebudeme zakládat a prvky poputují do haldy. Čas $\mathcal{O}(B)$ na založení úrovně a její procházení proto můžeme rozpočítat mezi H prvků, které se musely na dané úrovni objevit, než byla rozpřihrádkována. (Povšimněte

si, že počítadlo nikdy nesnižujeme, pouze ho vynulujeme, když je úroveň zrušena. Proto vůbec nemusí odpovídat skutečnému počtu prvků z příslušné úrovně, které jsou právě uloženy v haldě. To ovšem vůbec nevadí – počítadlo pouze hlídá, aby se úroveň nevytvořila příliš brzy, tedy abychom měli dost prvků k rozúčtování času.)

Proměnnou μ necháme ukazovat na místo, kde jsme se při hledání minima v přihrádkách zastavili. Současné globální minimum struktury může být nižší, nachází-li se minimum zrovna v haldě. Stále je však zaručeno, že před μ se nenachází žádná neprázdná přihrádka.

Operace budou fungovat takto:

Insert:

1. Spočítáme, do které úrovně i má prvek padnout (bitovými operacemi).
2. Pokud je počítadlo této úrovně menší než H, zvýšíme ho, vložíme prvek do haldy a skočíme.
3. Nebyly-li ještě pro tuto úroveň založeny přihrádky, vyrobíme prázdné.
4. Vložíme prvek do příslušné přihrádky.

Decrease:

1. Pokud se prvek nachází v haldě (to si u každého prvku pamatujeme), provedeme *Decrease* v haldě a skončíme.
2. Smažeme prvek z jeho přihrádky a provedeme *Insert* s novou hodnotou.

ExtractMin:

1. Dokud je μ menší než aktuální minimum haldy, opakujeme:
2. Najdeme přihrádku odpovídající hodnotě μ.
3. Je-li tato přihrádka prázdná, přejdeme na další (upravíme μ). Jsme-li na konci úrovně, zrušíme ji, vynulujeme její počítadlo a pokračujeme o úroveň výš.
4. Není-li prázdná, rozprostřeme ji o úroveň níž (stejným způsobem jako při *Insert*u, takže prvních H prvků vložíme do haldy).
5. Smažeme minimum z haldy a vrátíme ho jako výsledek.

Pustíme se do analýzy složitosti. Jako parametry si zvolíme počet hladin d (takže počet přihrádek B na jedné úrovni je roven $L^{1/d}$) a „haldovou konstantu“ H.

Nejprve si všimneme, že než počítadlo nějaké úrovně vynulujeme, jsou bezpečně z haldy pryč všechny prvky patřící do této úrovně. Celkem se tedy v haldě vyskytuje nejvýše $\mathcal{O}(dH)$ prvků. *ExtractMin* v haldě proto trvá $\mathcal{O}(\log(dH))$, ostatní haldové operace $\mathcal{O}(1)$.

Nyní rozúčtujeme čas operací mezi jednotlivé prvky (opět si vše předplatíme v *Insert*u a ostatní operace poběží v amortizovaně konstantním čase):

- Každý prvek může být nejvýše jednou za dobu svého života vložen do haldy a nejvýše jednou z ní vyjmut. Na obojí dohromady přispěje $\mathcal{O}(\log dH)$.

- Prvek se účastní nejvýše d přehození do nižší úrovně. Pokud byl přihozen do haldy, už tam setrvá, jinak pokaždé zaplatí $\mathcal{O}(1)$ za zařazení do přihrádky, celkem tedy $\mathcal{O}(d)$ na prvek.
- Vytvoření, projití a smazání přihrádek na jedné úrovni nastane až tehdy, co bylo H prvků patřících do této úrovně vloženo do haldy. Stačí tedy, aby každý prvek přispěl časem $\mathcal{O}(B/H) = \mathcal{O}(L^{1/d}/H)$.

Každý prvek tedy platí $\mathcal{O}(d + \log dH + L^{1/d}/H) = \mathcal{O}(d + \log H + L^{1/d}/H)$. Pojďme najít nastavení parametrů, které tento výraz minimalizuje. Nejprve zvolme H tak, aby se d vyrovnalo s $L^{1/d}/H$. Tedy $H = L^{1/d}/d$. Celý výraz tím zjednodušíme na $\mathcal{O}(d + \log(L^{1/d}/d)) = \mathcal{O}(d + 1/d \cdot \log L - \log d) = \mathcal{O}(d + 1/d \cdot \log L)$. Oba sčítance volbou $d = \sqrt{\log L}$ vyrovnáme.

HOT Queue tedy zvládne *Insert* s amortizovanou časovou složitostí $\mathcal{O}(\sqrt{\log L})$ a ostatní operace v čase amortizovaně konstantním. Použijeme-li tuto strukturu v Dijkstrově algoritmu, spočte vzdálenosti v čase $\mathcal{O}(m + n\sqrt{\log L})$.

Dinicův algoritmus

Zajímavé vylepšení Dijkstrova algoritmu navrhl Dinic. Všiml si, že pokud je každá hrana dlouhá alespoň δ, můžeme uzavírat nejen vrchol s minimálním ohodnocením μ, ale i všechny s ohodnocením menším než $\mu + \delta$.

Pro takto upravený Dijkstrův algoritmus bude stále platit, že při uzavírání vrcholu odpovídá ohodnocení skutečné vzdálenosti, takže uzavřené vrcholy již nejsou znovu otevírány.

O monotonii vzdáleností jsme sice přišli, ale v přihrádkové struktuře nebo haldě můžeme klíče nahradit hodnotami $h'(v) := \lfloor h(v)/\delta \rfloor$. Tyto hodnoty se totiž chovají monotónně a vrcholy se stejným $h'(v)$ můžeme libovolně zaměňovat.

Kteroukoli z popsaných přihrádkových struktur můžeme tedy použít, pouze v rozboru časové složitosti nahradíme L výrazem L/δ. Tento přístup přitom funguje i pro neceločíselné délky hran, pouze potřebujeme mít předem k dispozici netriviální dolní odhad na všechny délky.

Potenciály

Viděli jsme, že v grafech s nezápornými délkami hran se nejkratší cesty hledají snáze. Nabízí se nalézt nějakou transformaci, která by dovedla délky hran upravit tak, aby byly nejkratší cesty zachovány (samozřejmě ne jejich délky, ale alespoň to, které cesty jsou nejkratší). Nabízí se následující fyzikální inspirace:

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

Zbývá přijít na to, kde si obstarat nějaký přípustný potenciál. Přidejme do grafu nový vrchol z, přiveďme z něj hrany nulové délky do všech ostatních vrcholů a označme $\psi(v)$ vzdálenost ze z do v v tomto grafu. Aby byl tento potenciál přípustný, musí pro každou hranu uv platit $\ell_\psi(u,v) = \ell(u,v) + \psi(u) - \psi(v) \geq 0$. Tuto nerovnost můžeme upravit na $\ell(u,v) + d(z,u) - d(z,v) \geq 0$, čili $d(z,u) + \ell(u,v) \geq d(z,v)$, což je ale obyčejná trojúhelníková nerovnost pro vzdálenost v grafu, která jistě platí.

Jedním výpočtem nejkratších cest v grafu se zápornými hranami (třeba algoritmem BFM) tedy dokážeme spočítat potenciál, který nám poslouží k redukci všech hran na nezáporné délky. To nám samozřejmě nepomůže, chceme-li jednorázově hledat nejkratší cestu, ale může nám to výrazně zjednodušit další, složitější práci s grafem. Jak uvidíme v příští kapitole, můžeme tak například nalézt vzdálenosti mezi všemi dvojicemi vrcholů v čase $\mathcal{O}(nm + n^2 \log n)$.

Na závěr tohoto oddílu dokážeme jedno pomocné tvrzení o potenciálech, které nám pomůže v konstrukci algoritmů typu 1-1:

Lemma: Pokud f a g jsou přípustné potenciály, pak jsou jimi i:

1. konvexní kombinace f a g, tedy $\alpha f + \beta g$ pro libovolné $\alpha, \beta \geq 0, \alpha + \beta = 1$;
2. $\min(f, g)$;
3. $\max(f, g)$.

Důkaz: Buď uv hrana. Potom z definice přípustnosti platí $\ell(u,v) \geq f(v) - f(u)$ a $\ell(u,v) \geq g(v) - g(u)$. Jednotlivé části tvrzení dokážeme takto:

1. Pokud obě strany nerovnosti pro f vynásobíme konstantou α, u nerovnosti pro g konstantou β a obě nerovnosti sečteme, dostaneme:

   $$(\alpha + \beta) \cdot \ell(u,v) \geq (\alpha f(v) + \beta g(v)) - (\alpha f(u) + \beta g(u)),$$

   což je přesně požadovaná nerovnost pro přípustnost funkce $\alpha f + \beta g$.
2. Označme $h := \min(f, g)$. Nechť bez újmy na obecnosti $f(u) \leq g(u)$. Pokud také platí $f(v) \leq g(v)$, shoduje se minimum s funkcí f a není co dokazovat. V opačném případě je $h(u) = f(u)$, $h(v) = g(v)$. Tehdy si stačí všimnout, že $h(v) - h(u) = g(v) - f(u) < f(v) - f(u) \leq \ell(u,v)$, takže potenciál h je přípustný.
3. Dokážeme analogicky. ♡

Algoritmy pro problém typu 1-1

Zaměřme se nyní na případ, kdy chceme hledat nejkratší cesty mezi zadanou dvojicí vrcholů s, t. Obvykle se i v této situaci používají algoritmy 1-n (SSSP) a

v obecném případě ani není efektivnější řešení známo. Existuje ovšem velké množství heuristik, kterými lze obvykle výpočet zrychlit. Některé z nich si předvedeme na Dijkstrově algoritmu.

Dijkstrův algoritmus ve své klasické podobě neví vůbec nic o cílovém vrcholu a prohledá celý graf. Hned se nabízí využít toho, že od okamžiku uzavření kteréhokoliv vrcholu se již jeho ohodnocení nezmění. Pokud tedy uzavřeme cílový vrchol, můžeme se zastavit.

Jakou část grafu prohledáváme teď? V metrice dané vzdálenostmi v grafu je to nejmenší koule se středem ve vrcholu u, která obsahuje nejkratší cestu (dobře se to představuje na hledání v silniční síti na rovinné mapě).

Také 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. Oba postupy můžeme libovolně střídat a 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 $d(v,u)$ – zkuste vymyslet protipříklad. 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.

Algoritmus A*

V umělé inteligenci se často pro hledání nejkratší cesty v rozsáhlých grafech (obvykle stavových prostorech úloh) používá algoritmus nazývaný A* [30]. 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 uzavíra 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: 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: Algoritmus A* funguje jen tehdy, je-li potenciál $-\psi$ přípustný.

Například 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.

14. Transitivní uzávěry

V předchozí kapitole jsme se zabývali algoritmy pro hledání nejkratších cest z daného počátečního vrcholu. Nyní se zaměříme na případy, kdy nás zajímají vzdálenosti, případně pouhá dosažitelnost, mezi všemi dvojicemi vrcholů.

Výstupem takového algoritmu bude *matice vzdáleností* (případně *matice dosažitelnosti*). Na ni se také můžeme dívat jako na *transitivní uzávěr* zadaného grafu – to je graf na téže množině vrcholů, jehož hrany odpovídají nejkratším cestám v grafu původním.

Jeden způsob, jak transitivní uzávěr spočítat, se ihned nabízí: postupně spustit Dijkstrův algoritmus pro všechny možné volby počátečního vrcholu, případně se před tím ještě zbavit záporných hran pomocí potenciálů. Tak dosáhneme časové složitosti $\mathcal{O}(mn + n^2 \log n)$. To je pro řídké grafy nejlepší známý výsledek – jen $\mathcal{O}(\log n)$-krát pomalejší, než je velikost výstupu.

Je-li graf hustý, pracují obvykle rychleji algoritmy založené na maticích, zejména na jejich násobení. Několik algoritmů tohoto druhu si nyní předvedeme, a to jak pro výpočet vzdáleností, tak pro dosažitelnost.

Graf na vstupu bude vždy zadán maticí délek hran – to je matice $n \times n$, jejíž řádky i sloupce jsou indexované vrcholy a na pozici (i, j) se nachází délka hrany ij; případné chybějící hrany doplníme s délkou $+\infty$.

Floydův-Warshallův algoritmus

Začněme algoritmem, který nezávisle na sobě objevili Floyd a Warshall. 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 $\{1, \ldots, k\}$). 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 $D^0, D^1, \ldots, D^n$ podle uvedeného předpisu:

1. $D^0 \leftarrow$ matice délek hran.
2. Pro $k = 1, \ldots, n$:
3. Pro $i, j = 1, \ldots, n$:
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.

Regulární výrazy

Předchozí algoritmus lze zajímavě zobecnit, totiž tak, aby pro každou dvojici vrcholů sestrojil vhodnou reprezentaci množiny všech sledů vedoucích mezi nimi. Tato reprezentace bude velice podobná regulárním výrazům známým z UNIXu a z teorie automatů. Budeme připouštět orientované multigrafy se smyčkami a násobnými hranami.

Definice: *Svazek* je množina sledů, které mají společný počáteční a koncový vrchol. *Typem* svazku nazveme uspořádanou dvojici těchto vrcholů. Místo „svazek typu (u, v)“ budeme obvykle říkat prostě *uv-svazek.*

Triviálními případy svazků jsou prázdná množina $\emptyset$, samotná hrana uv a pro $u = v$ také sled ε_u nulové délky. Svazky můžeme kombinovat těmito operacemi:

- $A \cup B$ – *sjednocení* dvou svazků téhož typu,
- AB nebo $A \cdot B$ – *zřetězení* uv-svazku A s vw-svazkem B: výsledkem je uw-svazek obsahující všechna spojení sledu z A se sledem z B,
- A^* – *iterace* uu-svazku: výsledkem je uu-svazek $\varepsilon_u \cup A \cup AA \cup AAA \cup \ldots$ (tedy všechna možná spojení konečně mnoha sledů z A).

Ve výrazu definujícím A^* jsme využili, že operace sjednocení a zřetězení jsou asociativní, takže je nemusíme závorkovat. Navíc sjednocení má ve výrazech nižší prioritu než zřetězení.

Svazky budeme obvykle reprezentovat *sledovými výrazy,* což jsou výrazy konečné délky sestávající z triviálních svazků a výše uvedených operací.

Pozorování: Sledy můžeme reprezentovat řetězci nad abecedou, jejíž symboly jsou identifikátory hran. Sledové výrazy pak odpovídají (typovaným) regulárním výrazům nad touto abecedou.

Ukážeme, jak pro všechny dvojice vrcholů i, j sestrojit sledový výraz R_{ij} popisující svazek všech sledů z i do j. Podobně jako u Floydova-Warshallova algoritmu

zavedeme R_{ij}^k coby výraz popisující sledy z i do j přes 1 až k a nahlédneme, že platí:

$$R_{ij}^0 = \begin{cases} \text{množina všech hran z } i \text{ do } j & \text{pokud } i \neq j \\ \varepsilon_i \cup \text{všechny smyčky v } i & \text{pokud } i = j \end{cases}$$
$$R_{ij}^n = \text{hledané } R_{ij},$$
$$R_{ij}^k = R_{ij}^{k-1} \cup R_{ik}^{k-1}(R_{kk}^{k-1})^* R_{kj}^{k-1}.$$

První dvě rovnosti opět plynou přímo z definice (množinu všech hran zapíšeme buď jako prázdnou nebo ji vytvoříme sjednocováním z jednoprvkových množin). Třetí rovnost vychází z toho, že každý sled z i do j buď neobsahuje k, nebo ho můžeme rozdělit na části mezi jednotlivými návštěvami vrcholu k.

Algoritmus tedy bude postupně budovat matice $R^0, R^1, \ldots, R^n$ podle těchto pravidel. Provedeme při tom $\Theta(n^3)$ operací, ovšem s výrazy, jejichž délka může být až řádově 4^n. Raději než jako řetězce je proto budeme ukládat v podobě acyklických orientovaných grafů: vrcholy budou operace, hrany je budou připojovat k operandům.

K přímému použití se takový exponenciálně dlouhý výraz hodí málokdy, ale může nám pomoci odpovídat na různé otázky týkající se sledů s danými koncovými vrcholy. Máme-li nějakou funkci f ohodnocující množiny sledů kódované sledovými výrazy, stačí umět vyhodnotit:

- výsledek pro triviální výrazy $f(\emptyset)$, $f(\varepsilon)$ a $f(e)$ pro hranu e,
- hodnoty $f(\alpha \cup \beta)$, $f(\alpha\beta)$ a $f(\alpha^*)$, známe-li již $f(\alpha)$ a $f(\beta)$.

Pro výpočet všech $f(R_{ij})$ nám pak stačí $\Theta(n^3)$ vyhodnocení funkce f.

Poznámka pro ctitele algebry: Výše uvedená konstrukce není nic jiného, než popis homomorfismu f z algebry $(\mathcal{S}, \emptyset, \varepsilon_1, \ldots, \varepsilon_n, e_1, \ldots, e_m, \cup, \cdot, ^*)$ nad množinou $\mathcal{S}$ všech svazků do nějaké algebry $(X, \mathbf{0}, c_1, \ldots, c_n, h_1, \ldots, h_m, \oplus, \otimes, \circledast)$, kde X je množina všech ohodnocení sledů, $\mathbf{0}$, $c_1, \ldots, c_n$ a $h_1, \ldots, h_m$ jsou konstanty, $\oplus$ a $\otimes$ binární operace a $\circledast$ unární operace. Průběh výpočtu upraveného algoritmu je pak homomorfním obrazem průběhu výpočtu původního algoritmu pracujícího přímo se svazky.

Příklady:

Nejkratší sled:

$$f(\emptyset) = +\infty$$
$$f(\varepsilon) = 0$$
$$f(e) = \ell(e) \quad (\text{délka hrany } e)$$
$$f(\alpha \cup \beta) = \min(f(\alpha), f(\beta))$$
$$f(\alpha\beta) = f(\alpha) + f(\beta)$$
$$f(\alpha^*) = \begin{cases} 0 & \text{pro } f(\alpha) \geq 0 \\ -\infty & \text{pro } f(\alpha) < 0 \end{cases}$$

(Pokud předpokládáme, že v grafu nejsou záporné cykly, je $f(\alpha^*)$ vždy nulové a dostaneme přesně Floydův-Warshallův algoritmus.)

Nejširší cesta (hranám jsou přiřazeny šířky, šířka cesty je minimum z šířek hran):

$$\begin{aligned} f(\emptyset) &= 0 \\ f(\varepsilon) &= \infty \\ f(e) &= w(e) \quad \text{(šířka hrany } e\text{)} \\ f(\alpha \cup \beta) &= \max(f(\alpha), f(\beta)) \\ f(\alpha\beta) &= \min(f(\alpha), f(\beta)) \\ f(\alpha^*) &= \infty \end{aligned}$$

Převod konečného automatu na regulární výraz: Vrcholy multigrafu budou odpovídat stavům automatu, hrany možným přechodům mezi stavy. Každou hranu ohodnotíme symbolem abecedy, po jehož přečtení automat přechod provede. Funkce f pak přiřadí uv-svazku ψ regulární výraz popisující množinu všech řetězců, po jejichž přečtení automat přejde ze stavu u do stavu v po přechodech z množiny ψ. Vyhodnocování funkce f odpovídá přímočarým operacím s řetězci.

Násobení matic

Řešíme-li grafové problémy pomocí matic, nabízí se použít známé subkubické algoritmy pro lineárně algebraické úlohy. Ty jsou obvykle založeny na efektivním násobení matic – dvě matice $n \times n$ můžeme vynásobit v čase:

- $\mathcal{O}(n^3)$ podle definice,
- $\mathcal{O}(n^{2.808})$ Strassenovým algoritmem [50],
- $\mathcal{O}(n^{2.376})$ algoritmem Coppersmithe a Winograda [15],
- $\mathcal{O}(n^{2.373})$ algoritmem Williamsové [61].

Obecně přijímaná hypotéza říká, že pro každé $\omega > 2$ existuje algoritmus násobící matice se složitostí $\mathcal{O}(n^\omega)$. Jediný známý dolní odhad je přitom $\Omega(n^2 \log n)$ pro aritmetické obvody s omezenou velikostí konstant [45]. Blíže se o tomto fascinujícím tématu můžete dočíst v Szegedyho článku [14].

Předpokládejme tedy, že umíme násobit matice v čase $\mathcal{O}(n^\omega)$ pro nějaké $\omega < 3$.

Co se stane, když mocníme matici sousednosti A grafu? V matici A^k se na pozici i, j nachází počet sledů délky k, které vedou z vrcholu i do vrcholu j. (Snadný důkaz indukcí vynecháváme.) Pro libovolné $k \geq n-1$ jsou tedy v $(A+E)^k$ (kde E značí jednotkovou matici; doplnili jsme tedy do grafu smyčky) nenuly přesně tam, kde z i do j vede cesta.

To nám dává jednoduchý algoritmus pro výpočet matice dosažitelnosti R (R_{ij} je 1 nebo 0 podle toho, zda z i do j vede cesta). Do matice A doplníme na diagonálu jedničky a poté ji $\lceil \log_2 n \rceil$-krát umocníme na druhou. Abychom se vyhnuli velkým číslům, nahradíme po každém umocnění nenuly jedničkami. Celkem tedy provedeme $\mathcal{O}(\log n)$ násobení matic, což trvá $\mathcal{O}(n^\omega \log n)$.

Na tento postup se můžeme dívat i obecněji:

Definice: $(\oplus, \otimes)$-*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.

Pozorování: Je-li operace $\oplus$ je asociativní a komutativní, operace $\otimes$ asociativní a $\otimes$ distributivní přes $\oplus$, pak $(\oplus, \otimes)$-součin matic je asociativní a distributivní přes $\oplus$ matic.

Pozorování: Pokud A a B jsou matice svazků (A_{ij} a B_{ij} jsou ij-svazky) a C jejich $(\cup, \cdot)$-součin, pak C_{ij} je svazek všech sledů vzniklých spojením nějakého sledu z A začínajícího v i a nějakého sledu z B končícího v j.

Podobně jako v předchozí sekci si tedy můžeme pořídit funkci f přiřazující svazkům hodnoty z nějaké množiny X a operace $\oplus$ a $\otimes$ na X, pro něž platí $f(\alpha \cup \beta) = f(\alpha) \oplus f(\beta)$ a $f(\alpha\beta) = f(\alpha) \otimes f(\beta)$. Pak stačí vzít matici popisující ohodnocení všech sledů délky 1 (to je obdoba matice sousednosti), a provést $\mathcal{O}(\log n)$ $(\oplus, \otimes)$-součinů k tomu, abychom znali ohodnocení svazků sledů délky právě k pro nějaké $k \geq n$.

S $(\vee, \wedge)$-součiny a maticí sousednosti s jedničkami na diagonále získáme algoritmus pro výpočet dosažitelnosti. Každý součin přitom můžeme provést jako obyčejné násobení matic následované přepsáním nenul na jedničky, takže celý výpočet běží v čase $\mathcal{O}(n^\omega \log n)$.

Podobně můžeme počítat i matici vzdáleností: začneme s maticí délek hran doplněnou o nuly na diagonále a použijeme $(\min, +)$-součiny. Tyto součiny ale bohužel neumíme převést na klasické násobení matic. Přesto je známo několik algoritmů efektivnějších než $\Theta(n^3)$, byť pouze o málo: například Zwickův [63] v čase $\mathcal{O}(n^3 \sqrt{\log \log n} / \log n)$ (založený na dekompozici a předpočítání malých bloků) nebo Chanův [10] v $\mathcal{O}(n^3 / \log n)$ (používající geometrické techniky). Abychom porazili Floydův-Warshallův algoritmus, potřebovali bychom ovšem větší než logaritmické zrychlení, protože součinů potřebujeme vypočítat logaritmicky mnoho.

Dodejme ještě, že pro grafy ohodnocené malými celými čísly je možné využít celou řadu dalších triků. Zájemce o tento druh algoritmů odkazujeme na Zwickův článek [62].

Rozděl a panuj

Předchozí převod je ovšem trochu marnotratný. Šikovným použitím metody Rozděl a panuj můžeme časovou složitost ještě snížit. Postup předvedeme pro dosažitelnost: na vstupu tedy 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: 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. ♡

Algoritmus: Výpočet matice A^* provedeme podle předchozí věty. Spočítáme 2 tranzitivní uzávěry matic poloviční velikosti rekurzivním voláním, dále pak $\mathcal{O}(1)$ $(\vee, \wedge)$-součinů a $\mathcal{O}(1)$ součtů matic.

Časová složitost $t(n)$ tohoto algoritmu bude splňovat následující rekurenci:

$$t(1) = \Theta(1), \quad t(n) = 2t(n/2) + \Theta(1) \cdot \mu(n/2) + \Theta(n^2),$$

kde $\mu(k)$ značí čas potřebný na jeden $(\vee, \wedge)$-součin matic $k \times k$. Jelikož jistě platí $\mu(n/2) = \Omega(n^2)$, má tato rekurence podle kuchařkové věty řešení $t(n) \in \mathcal{O}(\mu(n))$.

Ukázali jsme tedy, že výpočet matice dosažitelnosti je nejvýše stejně náročný jako $(\vee, \wedge)$-násobení matic – můžeme ho proto provést v čase $\mathcal{O}(n^\omega)$. Dokonce existuje přímočarý převod v opačném směru, takže oba problémy jsou asymptoticky stejně těžké.

Podobně nalézt matici vzdáleností je stejně těžké jako $(\min, +)$-součin, takže na to stačí čas $\mathcal{O}(n^3/\log n)$.

Seidelův algoritmus

Pro neorientované neohodnocené grafy můžeme dosáhnout ještě lepších výsledků. Matici vzdáleností lze spočítat v čase $\mathcal{O}(n^\omega \log n)$ Seidelovým algoritmem [49]. Ten funguje následovně:

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

Seidelův algoritmus bude postupovat rekurzivně: Sestrojí graf G^2, rekurzí spočítá jeho matici vzdáleností D' a z ní pak rekonstruuje 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 $\mathcal{O}(n^\omega \log n)$.

L. Literatura

[1] N. Alon. A simple algorithm for edge-coloring bipartite multigraphs. *Inf. Process. Lett.*, 85(6):301–302, 2003.

[2] S. Alstrup, A. Ben-Amram, and T. Rauhe. Worst-case and amortised optimality in union-find. In *Proceedings of the 31st annual ACM symposium on Theory of computing*, pages 499–506. ACM, 1999.

[3] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *IEEE Symposium on Foundations of Computer Science*, pages 534–544, 1998.

[4] S. Alstrup, J. P. Secher, and M. Spork. Optimal on-line decremental connectivity in trees. *Information Processing Letters*, 64(4):161–164, 1997.

[5] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[6] Ben-Amram. What is a "pointer machine"? *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 26, 1995.

[7] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Theoretical INformatics*, pages 88–94, 2000.

[8] J. Boyer and W. Myrvold. On the cutting edge: Simplified $\mathcal{O}(n)$ planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004.

[9] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Systems Research Center, 1994.

[10] T. Chan. All-Pairs Shortest Paths with Real Weights in $O(n^3/\log n)$ Time. *Algorithmica*, 50(2):236–243, February 2008.

[11] B. Chazelle. A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity. *J. ACM*, 47:1028–1047, 2000.

[12] B. Cherkassky, A. Goldberg, and C. Silverstein. Buckets, heaps, lists, and monotone priority queues. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 83–92. Society for Industrial and Applied Mathematics, 1997.

[13] M. Chrobak and T. H. Payne. A linear-time algorithm for drawing a planar graph on a grid. *Information Processing Letters*, 54(4):241–246, 1995.

[14] H. Cohn, R. Kleinberg, B. Szegedy, and C. Umans. Group-theoretic Algorithms for Matrix Multiplication. *Foundations of Computer Science, Annual IEEE Symposium on*, pages 379–388, 2005.

[15] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of symbolic computation*, 9(3):251–280, 1990.

[16] E. Demaine. Advanced Data Structures. MIT Lecture Notes, 2005.

[17] J. Demel. *Grafy a jejich aplikace.* Academia, Praha, 2002.

[18] R. Diestel. *Graph Theory.* Springer-Verlag Berlin and Heidelberg GmbH & Co. K, 2005.

[19] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[20] A. Elmasry. The violation heap: a relaxed Fibonacci-like heap. *Computing and Combinatorics*, pages 479–488, 2010.

[21] L. R. Ford Jr. Network flow theory. Paper P-923, The RAND Corperation, Santa Moncia, California, August 1956.

[22] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. In *IEEE Symposium on Foundations of Computer Science*, pages 632–641, 1991.

[23] M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *STOC '89: Proceedings of the 21st annual ACM Symposium on Theory of Computing*, pages 345–354, 1989.

[24] M. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. In *Proceedings of FOCS'90*, pages 719–725, 1990.

[25] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[26] A. Goldberg and C. Silverstein. Implementations of Dijkstra's algorithm based on multi-level buckets. *Network optimization*, pages 292–327, 1997.

[27] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *Journal of SIAM*, 9(4):551–570, 1961.

[28] B. Haeupler, S. Sen, and R. Tarjan. Rank-pairing heaps. *Algorithms-ESA 2009*, pages 659–670, 2009.

[29] Y. Han. Deterministic sorting in O (nloglogn) time and linear space. *Journal of Algorithms*, 50(1):96–105, 2004.

[30] P. Hart, N. Nilsson, and B. Raphael. Correction to a formal basis for the heuristic determination of minimum cost paths. *ACM SIGART Bulletin*, (37):28–29, 1972.

[31] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[32] J. Hopcroft and R. Tarjan. Efficient Planarity Testing. *Journal of the ACM (JACM)*, 21(4):549–568, 1974.

[33] D. R. Karger, P. N. Klein, and R. E. Tarjan. Linear expected-time algorithms for connectivity problems. *J. ACM*, 42:321–328, 1995.

[34] D. R. Karger and C. Stein. A new approach to the minimum cut problem. *J. ACM*, 43(4):601–640, 1996.

[35] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 13th International Conference on Automata, Languages and Programming*. Springer Verlag, 2003.

[36] V. King. A simpler minimum spanning tree verification algorithm. In *Workshop on Algorithms and Data Structures*, pages 440–448, 1995.

[37] J. Komlós. Linear verification for spanning trees. *Combinatorica*, 5(1):57–65, 1985.

[38] L. Kučera. *Kombinatorické algoritmy*. SNTL, Praha, 1989.

[39] V. Malhotra, M. Kumar, and S. Maheshwari. An $\mathcal{O}(|V|^3)$ algorithm for finding maximum flows in networks. *Information Processing Letters*, 7(6):277–278, 1978.

[40] M. Mareš. Two linear time algorithms for MST on minor closed graph classes. *Archivum Mathematicum*, 40:315–320, 2004.

[41] J. Matoušek and J. Nešetřil. *Kapitoly z diskrétní matematiky*. Karolinum, Praha, 2002.

[42] H. Nagamochi and T. Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM J. Discret. Math.*, 5(1):54–66, 1992.

[43] S. Pettie. Finding minimum spanning trees in $O(m\alpha(m, n))$ time. Tech Report TR99-23, Univ. of Texas at Austin, 1999.

[44] S. Pettie and V. Ramachandran. An Optimal Minimum Spanning Tree Algorithm. In *Proceedings of ICALP'2000*, pages 49–60. Springer Verlag, 2000.

[45] Raz, R. On the complexity of matrix product. In *STOC '02: Proceedings of the 34th annual ACM Symposium on Theory of Computing*, pages 144–151, 2002.

[46] N. Robertson and P. D. Seymour. Graph minors: XX. Wagner's Conjecture. *Journal of Combinatorial Theory Series B*, 92(2):325–357, 2004.

[47] W. Schnyder. Embedding planar graphs on the grid. *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 138–148, 1990.

[48] A. Schrijver. *Combinatorial Optimization — Polyhedra and Efficiency*, volume 24 of *Algorithms and Combinatorics*. Springer Verlag, 2003.

[49] R. Seidel. On the all-pairs-shortest-path problem. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 745–749. ACM, 1992.

[50] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[51] R. E. Tarjan. *Data structures and network algorithms*, volume 44 of *CMBS-NSF Regional Conf. Series in Appl. Math.* SIAM, 1983.

[52] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984.

[53] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM (JACM)*, 46(3):362–394, 1999.

[54] M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *Proceedings of the thirty-fifth ACM symposium on Theory of computing*, pages 149–158, 2003.

[55] M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 149–158, 2003.

[56] M. Thorup. On AC^0 Implementations of Fusion Trees and Atomic Heaps. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 699–707, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[57] M. Thorup. Equivalence between priority queues and sorting. *Journal of the ACM*, 54(6):28, 2007.

[58] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[59] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977.

[60] K. Weihe. Edge-Disjoint (s, t)-Paths in Undirected Planar Graphs in Linear Time. *Journal of Algorithms*, 23(1):121–138, 1997.

[61] Williams, V. V. Multiplying matrices faster than Coppersmith-Winograd. In *STOC '12: Proceedings of the 44th annual ACM Symposium on Theory of Computing*, pages 887–898, 2012.

[62] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM (JACM)*, 49(3):289–317, 2002.

[63] U. Zwick. A Slightly Improved Sub-Cubic Algorithm for the All Pairs Shortest Paths Problem with Real Edge Lengths. *Algorithmica*, 46(2):181–192, October 2006.

O. Obsah

Mgr. Martin Mareš, Ph.D.

Krajinou grafových algoritmů

Vydal Institut Teoretické Informatiky
Univerzita Karlova v Praze
Matematicko-Fyzikální Fakulta
Malostranské nám. 25
118 00 Praha 1
jako ZZZ. publikaci v ITI Series.

Sazbu písmem Computer Modern v programu TEX provedl autor.
Obrázek na titulní straně nakreslil Jakub Černý.

Vydání jedenapůlté (pracovní verze)
Praha 2024