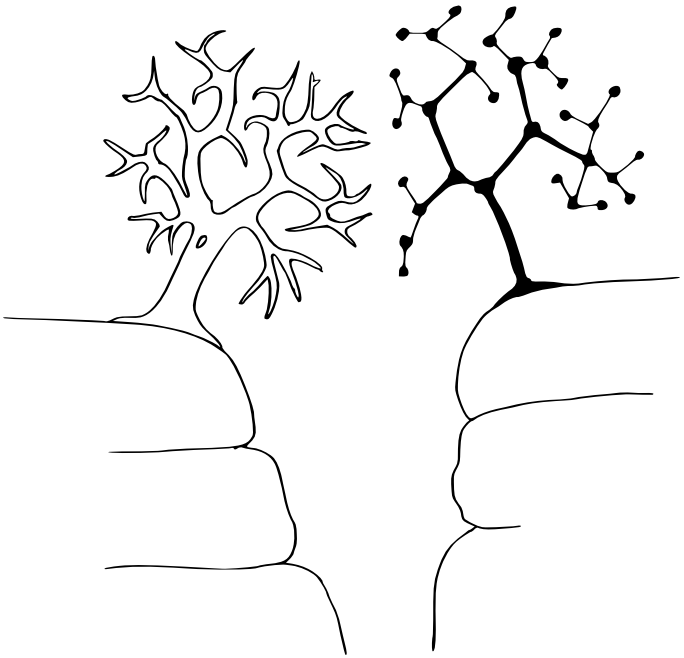
Martin Mareš

Krajinou grafových algoritmů

průvodce pro středně pokročilé



ITI 2007

ISBN 978-80-239-9049-2

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 [28], [12] a [25]. Výbornou referenční příručkou, ze které jsem častokrát čerpal i já při sestavování přednášek, je také Schrijverova monumentální monografie Combinatorial Optimization [33].

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 Ford-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 Ford-Fulkersonův algoritmus

V této kapitole nadefinujeme toky v sítích, odvodíme základní věty o nich a také Ford-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 *kapacity* jednotlivých 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 Ford-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 Kirchoffova 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 není náhoda, 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 Ford-Fulkersonův algoritmus. ♡

Ford-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. $\quad m \leftarrow \min_{e \in P} r(e)$.
4. $\quad$ Zvětšíme tok f podél P o m (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 $m \geq 1$, což může nastat pouze konečně-krát. 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 Ford-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⟩

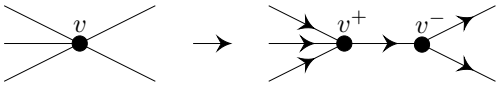
Důkaz: Z grafu sestrojíme síť tak, že s bude zdroj, t spotřebič a všem hranám nastavíme kapacitu na jednotku. Řezy v této síti odpovídají řezům v původním grafu. Podobně každý systém hranově disjunktních *st*-cest odpovídá toku stejné velikosti a naopak ke každému celočíselnému toku dovedeme najít systém 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:

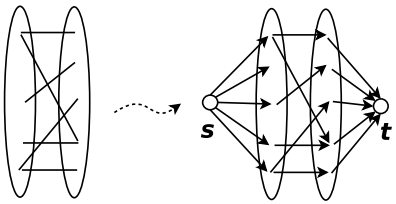
Věta (Mengerova, globální hranová neorientovaná): Neorientovaný graf G je hranově k-souvislý právě tehdy, když mezi každými dvěma vrcholy existuje alespoň k hranově disjunktních cest.

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. Ford-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 Ford-Fulkersonova algoritmu z minulé kapitoly. Nyní pojednáme pojednáme o efektivnějším Dinicově algoritmu a o různých jeho variantách pro sítě ve speciálním tvaru.

Dinicův algoritmus

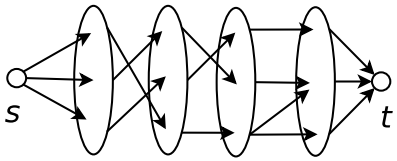
Dinicův algoritmus je založen na následující myšlence: Ve Ford-Fulkersonově algoritmu nemusíme přičítat jen zlepšující cesty, ale je možné přičítat rovnou zlepšující toky. Nejlépe toky takové, aby je nebylo obtížné najít, a přitom aby původní tok dostatečně obohatily. Vhodnými objekty k tomuto účelu 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 $\mathcal{O}(n^2 m)$.

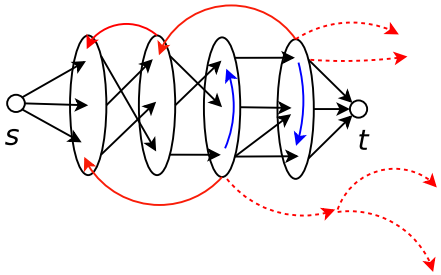
Korektnost algoritmu: Když se Dinicův algoritmus zastaví, nemůže už existovat žádná zlepšující cesta (viz krok 4) a tehdy, jak už víme z analýzy F-F algoritmu, je nalezený tok maximální.

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

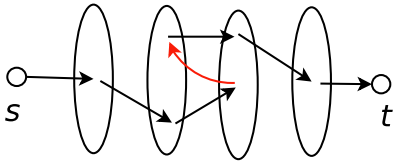
zaručeně nasytí, protože tok f_B je blokující. Musíme však dokázat, že nemohou vzniknout žádné nové cesty délky l nebo menší. V síti rezerv totiž mohou hrany nejen ubývat, ale i přibývat: pokud pošleme tok po hraně, po které ještě nic neteklo, tak v protisměru z dosud nulové rezervy vyrobíme nenulovou. 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.



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

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. ♡



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.

[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.

- 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 1 (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

⟨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 $1\times$.

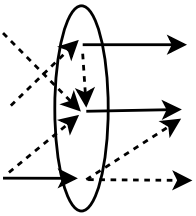
⟨7⟩ Přeci v řeznictví. Kdepak, spíše v cukrárně. Myslíte, že v cukrárně mají Dinicovy řezy? Myslím, že v cukrárně je většina řezů minimální. *(odposlechnuto na přednášce)*

⟨8⟩ Princip holubníku a nějaká ta ± 1.

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.



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 [19]. 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 najedeme 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 $\mathcal{O}(|f|n + nm)$

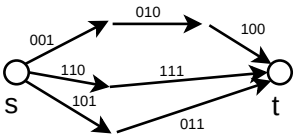
Pokud navíc budeme předpokládat, že kapacita hran je nejvýše C a G není multigraf, můžeme využít toho, že $|f| \leq Cn$ (omezeno řezem okolo s) a získat odhad $\mathcal{O}(Cn^2 + nm)$.

Scaling 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.

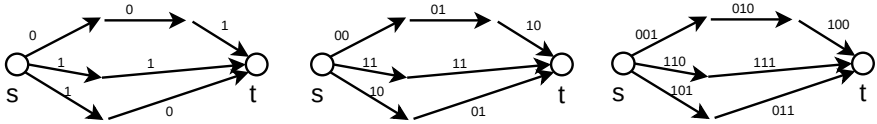
V našem případě budeme postupně budovat sítě čím dál podobnější zadané síti a v nich počítat toky, až nakonec získáme tok pro ni.

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 [26] 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.

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

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 s.

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 $\leq C$ (scaling)	$\mathcal{O}(mn \log C)$
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 [2]

Nejprve si nadefinujme operaci *Degree Split,* která dostane jako vstup libovolný $2k$-regulární graf $G = (V, E)$ se sudým počtem hran a rozdělí ho na podgrafy $G_1 = (V, E_1)$ a $G_2 = (V, E_2)$, které budou oba k-regulární. Tuto operaci můžeme snadno provést v lineárním čase tak, že si graf rozdělíme na komponenty, v každé nalezneme eulerovský tah a jeho sudé hrany dáme do G_1 a liché do G_2.

To nám pomůže ke snadnému algoritmu pro nalezení maximálního párování ve 2^d-regulárním bipartitním grafu.⟨10⟩ Stačí provést Degree Split na dva 2^{d-1}-regulární grafy, na libovolný jeden z nich aplikovat další Degree Split atd., až se dostaneme k 1-regulárnímu grafu, který je perfektním párováním v G. To vše jsme schopni stihnout v lineárním čase, jelikož velikosti grafů, které splitujeme, exponenciálně klesají. Také bychom mohli rekurzivně zpracovávat obě komponenty a tak se v čase $\mathcal{O}(m \log n)$ dobrat ke kompletní 1-faktorizaci zadaného grafu.⟨11⟩

Pokud zadaný graf nebude 2^d-regulární, pomůžeme si tím, že ho novými hranami doplníme na 2^d-regulární a pak si při splitech 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 $n(e)$.

Degree Split 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 sudě regulární graf bez násobností, takže na něj můžeme aplikovat původní Degree Split a G'_i přiřadit ke G_i. To vše zvládneme v čase $\mathcal{O}(m)$.

Mějme nyní k-regulární bipartitní graf. 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.

Takto získáme 2^t-regulární graf, jehož reprezentace bude lineárně velká. Na tento graf budeme aplikovat operaci Degree Split a budeme si vybírat vždy tu polovinu, kde bude méně hran z F. Po t iteracích dospějeme k párování a jelikož se v každém

⟨10⟩ Všimněte si, že takové párování bude vždy perfektní (viz Hallova věta), a že takový graf má vždy sudý počet hran.

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

kroku zbavíme alespoň poloviny hran z F, nebude toto párování obsahovat žádnou takovou hranu a navíc nebude ani obsahovat násobné hrany, a tedy bude podgrafem zadaného grafu, jak potřebujeme.

Časová složitost algoritmu je $\mathcal{O}(m \log n)$, jelikož provádíme inicializaci v $\mathcal{O}(m)$ 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 $\mathcal{O}(\kappa(G)n^{3/2}m)$, kde $\kappa(G)$ je nalezený stupeň souvislosti G.

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

Globálně minimální řez (Nagamochi, Ibaraki [29])

Buď G neorientovaný graf 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 nějaký řez oddělující v_{n-1} a 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 $u_i = u_{i-1}$, a pak je nerovnost splněna jako rovnost, nebo je $u_i = v_j$, $j > i$ 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 [17].

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, $\mathcal{O}(n^2m)$ 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:

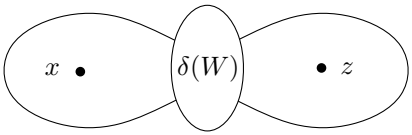
Věta (o využití GHT): Buď T libovolný GHT pro graf G a mějme dva vrcholy s a t. Dále nechť P je cesta v T mezi vrcholy s a t a e je hrana na cestě P s minimálním $r(e)$. Pak $\delta(e)$ je minimální *st*-řez v G.

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:

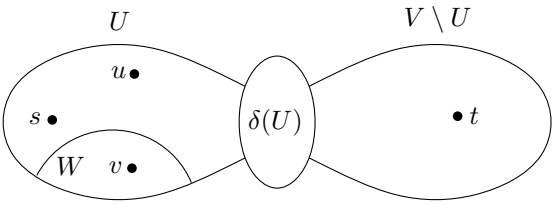
- $|\,P\,| = 1$: Hrana e je v tomto případě přímo st, takže i minimalita plyne z definice GHT.
- $|\,P\,| > 1$: Cesta P spojuje vrcholy s a t, její první hranu označme sx. Naše právě dokázané lemmátko říká, že $r(s,t) \geq \min(r(s,x), r(x,t))$. Určitě je pravda, že $r(s,x) \geq r(e)$, protože e byla 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⟩



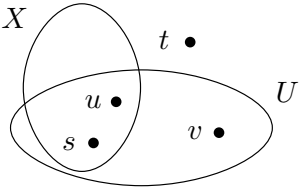
Důkaz: Nechť je $\delta(X)$ minimální uv-řez. BÚNO můžeme předpokládat, že $s \in U$ a $t \notin U$, $u \in X$ a $v \notin X$ a $s \in X$. Pokud by tomu tak nebylo, můžeme vrcholy přeznačit nebo některou z množin nahradit jejím doplňkem.

⟨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.

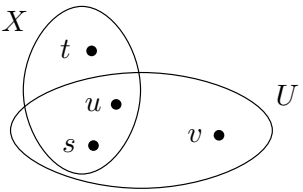
Nyní stačí nerovnosti (2) a (1) odečíst, čímž získáme:

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

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:

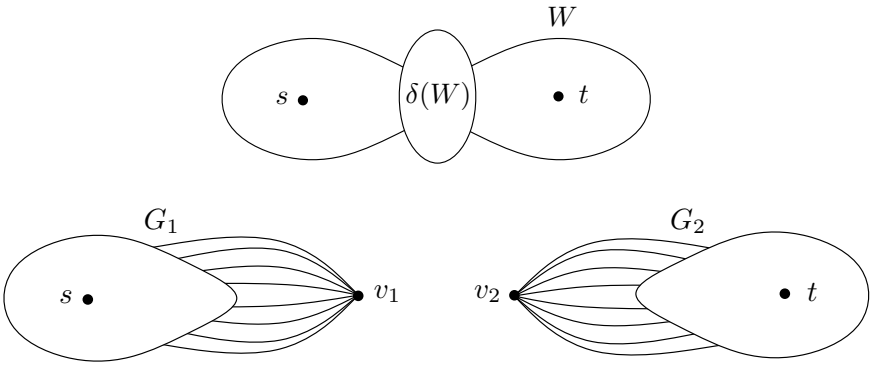
1. $\forall r : r \in C(r)$, neboli každý vrchol ČGHT je přilepen sám k sobě, a
2. $\forall st \in F : \delta\left(\bigcup_{r \in K_1} C(r)\right) = \delta\left(\bigcup_{r \in K_2} C(r)\right)$ je minimální st-řez, kde K_1 a K_2 jsou komponenty $(R, F) \setminus st$.

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. ♡

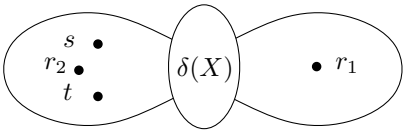
Nyní víme, že GHT existují, a také víme, jak by se daly konstruovat. Nicméně nalezení vrcholů s, t tak, aby byl minimální st-řez nejmenší možný, je časově náročné. Proto si poslední větu ještě o něco vylepšíme.

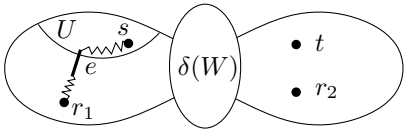
Vylepšení věty o existenci ČGHT: Na začátku důkazu není nutné hledat vrcholy s a t takové, aby byl minimální st-řez nejmenší možný. Stačí zvolit *libovolné* vrcholy $s, t \in R$ a zvolit $\delta(W)$ jako minimální st-řez.

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 [34]. 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:

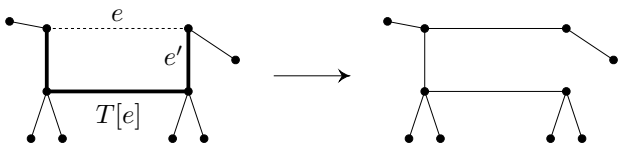
- $T[x, y]$ bude značit cestu v T, která spojuje x a y. (Cestou opět míníme množinu hran.)
- $T[e] := T[x, y]$ pro hranu $e = xy$. Této cestě budeme říkat *cesta pokrytá hranou e*.
- Hrana $e \in E \setminus T$ je *lehká vzhledem k* $T \equiv \exists e' \in T[e] : w(e) < w(e')$. Ostatním hranám neležícím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

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

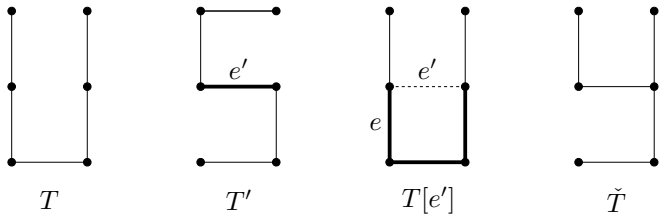
Tato věta nám dává pěknou alternativní definici MST, která místo sčítání vah váhy pouze porovnává, čili jí místo čísel stačí lineární (kvazi)uspořádání na hranách. Než se dostaneme k jejímu důkazu, prozkoumejme nejdříve, jak se dá mezi jednotlivými kostrami přecházet.

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.

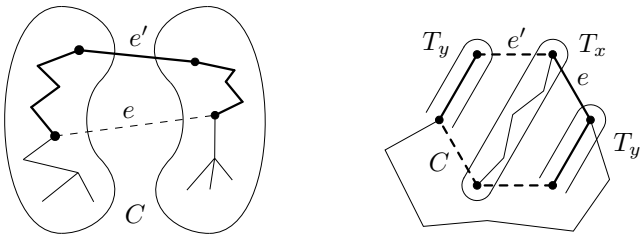
Věta: Pro Červenomodrý meta-algoritmus spuštěný na libovolném grafu s hranami lineárně uspořádanými podle vah platí:

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. ♡



Situace v důkazu Modrého a Červeného lemmatu

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

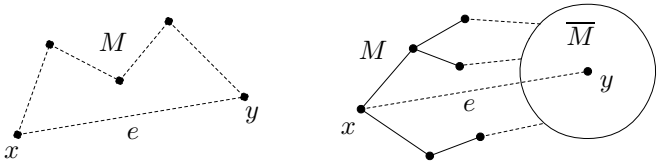
Důkaz: Opět sporem: Předpokládejme, že e byla obarvena červeně jako nejtěžší na nějaké kružnici C a že $e \in T_{min}$. Odebráním e se nám T_{min} rozpadne na dvě komponenty T_x a T_y. Některé vrcholy kružnice připadnou do komponenty T_x, ostatní do T_y. Na C ale musí existovat nějaká hrana $e' \neq e$, 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.

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

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.

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

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, takže celkově 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 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 [27]

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 $H \preceq G$) $\equiv$ H lze z G získat mazáním vrcholů či hran a kontrahováním hran (s odstraněním smyček a násobných hran).

Pozorování: $H \subseteq G \Rightarrow H \preceq G$.

Definice: Třída grafů $\mathcal{C}$ je *minorově uzavřená* $\equiv \forall G, H : G \in \mathcal{C}, H \preceq G \Rightarrow H \in \mathcal{C}$.

Věta: (Robertson, Seymour) Pokud je $\mathcal{C}$ minorově uzavřená třída grafů, existuje konečná množina grafů Z taková, že pro každý graf G platí:

$$G \notin \mathcal{C} \iff \exists H \in Z : H \preceq G.$$

Jinými slovy, každou minorově uzavřenou třídu lze charakterizovat *konečným* počtem 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. Pěkné shrnutí této teorie najdete například v Diestelove knize [13].

Pozorování: Například pro rovinné grafy jsou těmi zakázanými minory právě $K_{3,3}$ a K_5. To plyne z Kuratowského věty: jedna implikace je triviální, druhá dá trochu práce: pokud je $K_{3,3} \preceq G$, najde se v G podgraf isomorfní nějakému dělení $K_{3,3}$; pokud je $K_5 \preceq G$, podgraf isomorfní dělení K_5 se v G najít nemusí, ale pokud se nenajde, najde se tam podgraf isomorfní dělení $K_{3,3}$. (Zkuste si sami.)

Věta: (Robertson, Seymour) Pokud je třída grafů $\mathcal{C}$ minorově uzavřená a netriviální (alespoň jeden graf v ní leží a alespoň jeden neleží), pak $\exists c > 0 : \forall G \in \mathcal{C} : |E(G)| \leq c \cdot |V(G)|$.

Důsledek: Jelikož všechny grafy vygenerované předchozím algoritmem jsou minory grafu ze vstupu, můžeme pro odhad jejich hustoty použít předchozí větu a dostaneme lineární časovou složitost dokonce pro každou netriviální minorově uzavřenou třídu grafů.

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 [16])

1. Začneme libovolným vrcholem v_0, $T \leftarrow \{v_0\}$.
2. Do haldy H umístíme všechny hrany vedoucí z v_0.
3. Opakuji dokud $H \neq \emptyset$:
4. $\quad vw \leftarrow \mathit{DeleteMin}(H)$, přičemž $v \notin T, w \in T$.
5. $\quad T \leftarrow T \cup \{vw\}$
6. $\quad$ Pro všechny sousedy u vrcholu v, které 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 *DeleteMin* 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 $\mathcal{O}(m + n' \log n'/\log n) = \mathcal{O}(m)$.

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 [16])

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. 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 $\log^*$. [9], [30]
- $\mathcal{O}(\mathcal{T}(m,n))$, kde $\mathcal{T}(m,n)$ je hloubka optimálního rozhodovacího stromu pro nalezení minimální kostry v grafech s patřičným počtem hran a vrcholů [31]. 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 jelikož 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) [15] – 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 $1 \ldots n$ a použít předchozí algoritmus.
- $\mathcal{O}(m)$ randomizovaně v průměrném případě [21].
- Na zjištění, zda je zadaná kostra minimální, stačí $\mathcal{O}(m)$ porovnání [24] a dokonce lze v lineárním čase zjistit, která to jsou [23]. Z toho ostatně vychází předchozí randomizovaný algoritmus.

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.

[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.

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) [5] 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) [11] 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

[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 ome-

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 [40] 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	$\mathcal{O}(n \log\log U)$	$\mathcal{O}(n \log\log n)$ [18]
MST	$\mathcal{O}(m \log\log U)$	$\mathcal{O}(m)$ [viz příští kapitola]
Dijkstra	$\mathcal{O}(m \log\log U)$	$\mathcal{O}(m + n \log\log n)$ [37], neorientovaně $\mathcal{O}(m)$ [36]

My se přidržíme ekvivalentní, ale jednodušší definice podle Erika Demaine [11].

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

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 $\mathcal{O}(k)$.

---

zením 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.

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.

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.

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

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 pří 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, **0** a **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.

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

  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 reprezentovaným toutéž bitovou maskou.

- *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 $b = 4$:

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

  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 [15], 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,

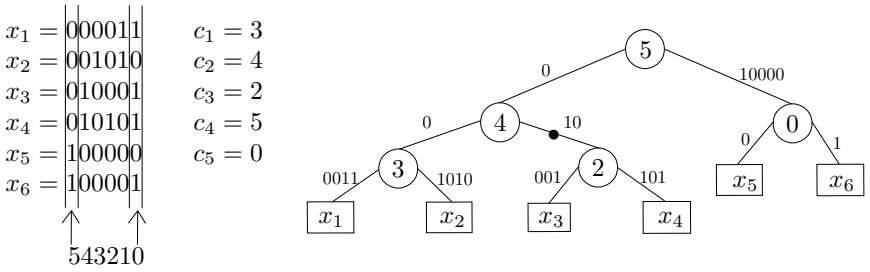
- $X = \{x_1, \ldots, x_r\}$ – uložené w-bitové prvky, očíslujeme si je tak, aby $x_1 < \ldots < x_r$,
- $c_i = \mathrm{MSB}(x_i \oplus x_{i+1})$ – nejvyšší bit, ve kterém se liší x_i a x_{i+1},
- $\mathrm{Rank}_X(x)$ – počet prvků množiny X, které jsou menší než x (přičemž x může ležet i mimo X).

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.



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 $c_1, \ldots, c_n$ (je to totiž kartézský strom nad těmito hodnotami – blíže viz kapitola o dekompozicích stromů), hodnoty v listech jsou $x_1, \ldots, x_n$ v pořadí zleva doprava.

Kdykoliv chceme indexovat tvarem stromu, můžeme tedy indexovat přímo vektorem $(c_1, \ldots, c_n)$, který má pouze $k \log w = \mathcal{O}(k^2)$ bitů. Pro zjednodušení ostatních operací ale zvolíme trochu jinou, ekvivalentní reprezentaci:

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

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 $r \cdot \log r$ 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 $r \cdot \log r$ 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. 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 [38].

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 [15].

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 by rank:* každý kořen v si pamatuje svůj rank $r(v)$. 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⟩
- *Path compression:* pokud z vrcholu vystoupíme do kořene (například během operace *Find*), přepojíme všechny vrcholy na cestě, po které jsme prošli, rovnou pod kořen.

Pozorování: Samotné pravidlo Union by rank zajistí, že strom ranku r bude mít hloubku nejvýše r a minimálně 2^r vrcholů, takže časová složitost operací bude omezena $\mathcal{O}(\log n)$ v nejhorším případě.⟨24⟩

Amortizovaně se ale popsaná struktura chová daleko lépe:

Věta: (Tarjan, van Leeuwen [35]) Kombinace Union by rank a Path compression vede k amortizované složitosti obou operací $\mathcal{O}(\alpha(n))$, kde α je inverzní Ackermannova funkce.⟨25⟩

Důkaz této věty neuvádíme, jelikož je technicky dosti náročný, ale naznačíme alespoň, že amortizovaná časová složitost je omezena iterovaným logaritmem, konkrétně že ve struktuře s n prvky trvá provedení ℓ operací Union a Find $\mathcal{O}((n + \ell) \cdot \log^* n)$.

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 (vrchol, který již není kořenem, si pamatuje svůj poslední rank z doby, kdy ještě kořenem byl): 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, využívajíce toho, že vrcholů s rankem r je nejvýše $n/2^r$:

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

Operace Union a Find potřebují nekonstantní čas pouze na vystoupání ze zadaného vrcholu v do kořene stromu a tento čas je přímo úměrný počtu hran na cestě

⟨23⟩ Stejně by fungovalo pravidlo *Union by size,* které připojuje menší strom pod větší, ale ranky máme raději, neb jsou skladnější a snáze se analyzují.

⟨24⟩ Mimochodem, Path compression samotná by také na složitost $\mathcal{O}(\log n)$ amortizovaně stačila. [35]

⟨25⟩ Existuje varianta tohoto algoritmu, která dosahuje stejné složitosti i v nejhorším případě; též je známo, že asymptoticky lepší složitosti nelze dosáhnout.

z v do kořene. Tato cesta je následně rozpojena a všechny vrcholy ležící na ní jsou přepojeny přímo pod kořen stromu. Hrany cesty, které spojují vrcholy z různých skupin (takových je jistě $\mathcal{O}(\log^* n)$), naúčtujeme právě prováděné operaci, takže jimi celkem strávíme čas $\mathcal{O}(\ell \log^* n)$. Zbylé hrany budeme počítat přes celou dobu běhu algoritmu a účtovat je vrcholům.

Uvažme vrchol v v k-té skupině, který již není kořenem stromu. Hrana z v do jeho rodiče bude účtována vrcholu v pouze tehdy, leží-li rodič také v k-té skupině. Jenže ranky vrcholů na cestě z libovolného vrcholu do kořene ostře rostou, takže při každém přepojení rank rodiče vrcholu v vzroste, a proto po $2 \uparrow k$ přepojeních bude rodič vrcholu v v $(k+1)$-ní nebo vyšší skupině. Každému vrcholu v k-té skupině tedy účtujeme 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 v [4] a [3].

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

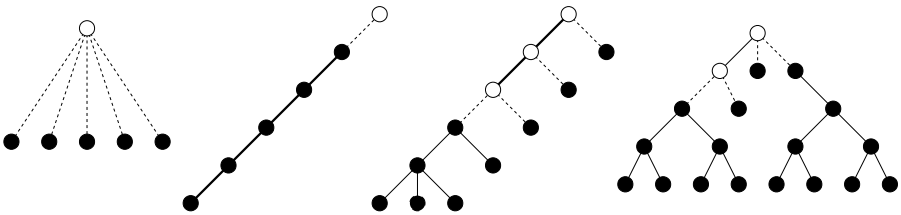
- *Kořeny mikrostromů* budou nejvyšší vrcholy v T, pod nimiž je nejvýše $\log n$ listů a které nejsou kořenem celého T.
- *Mikrostromy* leží v T od těchto kořenů níže.
- *Spojovací hrany* vedou z kořenů mikrostromů do jejich otců.
- *Makrostrom* je tvořen zbývajícími vrcholy a hranami stromu T.

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.

[⟨26⟩] Kdy se to hodí? Třeba v Thorupově lineárním algoritmu [36] na nejkratší cesty nebo ve Weiheho taktéž lineárním algoritmu [41] na hledání hranově disjunktních cest v rovinných grafech.

Příklad: Následující obrázek ukazuje dekompozici několika stromů za přepokladu, 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.

Cestová komprese: 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 hodí například následující myšlenka:

Definice: *(Fredericksonova clusterizace)* Nechť G je graf s vrcholy stupňů nejvýše 3 a $c \geq 1$. Pak c-clusterizací grafu G nazveme libovolný rozklad G na souvislé podgrafy *(clustery)* $C_1, C_2, \ldots, C_k$ takový, že platí:

- Každý vrchol se nachází v právě jednom clusteru (hrany mohou vést i mezi clustery).
- Každý cluster má nejvýše c vrcholů.
- Vnější stupeň každého clusteru (tj. počet hran, které vedou mezi C_i a ostatními clustery; mezi každou dvojicí clusterů počítáme jen jednu hranu) je nejvýše 3. Navíc pokud je právě 3, je cluster triviální, čili $|C_i| = 1$.
- Žádné dva sousední clustery nelze spojit.

Věta: (Frederickson [14]) Každá c-clusterizace grafu G má $\mathcal{O}(|V(G)|/c)$ clusterů. Existuje algoritmus, který jednu takovou najde v lineárním čase.

Důkaz: První část rozborem případů, druhá hladově pomocí DFS. ♡

Použití: Předchozí variantu Union-Find problému bychom také mohli vyřešit nahrazením vrcholů stupně > 3 „kruhovými objezdy bez jedné hrany“[28], 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.

[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.

[28] tzv. francouzský trik

- 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 [6] převedeme raději tento problém na jiný.

Problém: *(Range Minimum Query alias RMQ)* Chceme předzpracovat posloupnost čísel $a_1, \ldots a_n$ tak, abychom uměli rychle počítat $\min_{x \le i \le y} a_i$.⟨29⟩

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. 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 $m_{ij} = \min\{a_i, a_{i+1}, \ldots, a_{i+2^j-1}\}$, čili minima všech bloků velkých jako nějaká mocnina dvojky. Když se poté někdo zeptá na minimum bloku $a_i, a_{i+1}, \ldots, a_{j-1}$, najdeme největší k takové, že $2^k < j - i$ a vrátíme:

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

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

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 $2^{b-1} \le \sqrt{n}$ 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

⟨29⟩ Všimněte si, že pro sumu místo minima je tento problém velmi snadný.

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
$\lvert\alpha\rvert$	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:

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

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.

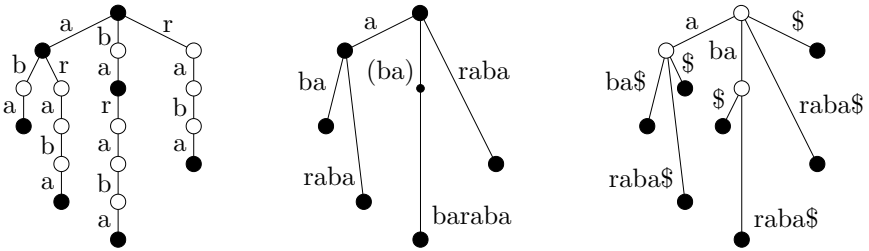
Definice: *Komprimovaná trie (Σ^+-strom)* vznikne z trie nahrazením maximálních nevětvících se cest hranami. Hrany jsou tentokrát popsané řetězci místo jednotlivými písmeny, přičemž popisky všech hran vycházejících z jednoho vrcholu se liší v prvním znaku. Vrcholům „uvnitř hran“ (které padly za oběť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

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

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 σ\$ 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)⟨30⟩ – 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 nejhlubší vnitřní vrchol, pod kterým se vyskytují listy pro α i β. Podobně můžeme sestrojit ST\$ pro libovolnou množinu slov.⟨31⟩
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* [8] – 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.

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

⟨31⟩ Jen si musíme dát pozor, abychom si moc nezvětšili abecedu, ale to bude jasné, až předvedeme konkrétní konstrukce.

Suffix Array

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 jedničky), $\sigma[i:j]$ pak podslovo složené z i-tého až j-tého znaku. Libovolnou z mezí můžeme vynechat, proto $\sigma[i:]$ bude suffix od i do konce a $\sigma[:j]$ prefix od začátku do j. Pokud $j < i$, definujeme $\sigma[i:j]$ jako prázdné slovo, takže prázdný suffix můžeme například zapsat jako $\sigma[|\sigma|+1:]$.

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

Definice: *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 $1,\ldots,n+1$, pro níž $\sigma[A[1]:] < \sigma[A[2]:] < \ldots < \sigma[A[n+1]:]$.

Definice: *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 $\sigma\$$ je lineárně ekvivalentní s dvojicí (A_σ, L_σ). [Jinými slovy, když máme jedno, můžeme z toho v lineárním čase spočítat druhé, a naopak.]

Důkaz: Když projdeme $\mathrm{ST}(\sigma\$)$ do hloubky, pořadí listů odpovídá A_σ a písmenkové hloubky vnitřních vrcholů v inorderu odpovídají L_σ. Naopak $\mathrm{ST}(\sigma)$ 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. ♡

Rekurzivní konstrukce

Tento algoritmus konstruuje pro slovo σ délky n jeho suffix array a LCP array 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 nám tedy dává lineární konstrukci $\mathrm{ST}(\sigma)$ pro libovolnou fixní abecedu.

Algoritmus: (Konstrukce A a L podle Kärkkäinena a Sanderse [22])

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}.

4. Z A_{01} a L_{01} vydělíme $A_0 = A_{\sigma_0}$, A_1, L_0 a L_1. Také si pro každý prvek A_i zapamatujeme, kde se v A_{01} vyskytoval.
5. Dopočítáme A_2: Jelikož $\sigma_2[i:] = \sigma[3i+2:] = \sigma[3i+2]\sigma[3i+3:] = \sigma[3i+2]\sigma_0[i+1:]$ a všechna $\sigma_0[i:]$ už máme setříděná, můžeme všechna $\sigma_2[i:]$ setřídit dvěma průchody přihrádkového třídění.
6. Dopočítáme L_2: Stejným trikem jako A_2 – pokud jsou první písmena různá, je společný prefix prázdný, jinak má délku $1 + \mathrm{LCP}(\sigma_0[i+1:], \sigma_0[j+1:]) = 1 + \min_{i+1 \le k < j+1} L_0[k]$. Minimum zvládneme pro každou dvojici i, j spočítat v konstantním čase pomocí datové struktury pro intervalová minima.
7. $A_0, A_1, A_2 \xrightarrow{merge} A$ – sléváme tři setříděné posloupnosti, takže stačí umět prvky libovolných dvou posloupností v konstantním čase porovnat:

$$\begin{aligned}
\sigma_0[i:] < \sigma_1[j:] &\text{ podle zapamatovaných pozic v } A_{01}\\
\sigma_0[i:] < \sigma_2[k:] &\equiv \sigma[3i]\,\sigma_1[i:] < \sigma[3k+2]\,\sigma_0[k+1:]\\
&\Leftrightarrow (\sigma[3i] < \sigma[3k+2]) \vee\\
&\quad (\sigma[3i] = \sigma[3k+2] \wedge \sigma_1[i:] < \sigma_0[k+1:])\\
\sigma_1[j:] < \sigma_2[k:] &\equiv \sigma[3j+1]\,\sigma[3j+2]\,\sigma_0[j+1:] <\\
&\quad \sigma[3k+2]\,\sigma[3k+3]\,\sigma_1[k+1:]
\end{aligned}$$

8. Dopočítáme L – pokud sousedí suffix ze $\sigma_{0,1}$ se suffixem ze $\sigma_{0,1}$, vyčteme výsledek přímo z L_{01}. Pokud sousedí σ_2 se σ_2, stačí použít už spočítané L_2. Pokud sousedí $\sigma_{0,1}$ se σ_2, odebereme první jeden nebo dva znaky, ty porovnáme samostatně a v případě shody zbude suffix ze σ_0 a suffix ze σ_1 (stejně jako při slévání) a pro ty dokážeme L dopočítat pomocí struktury pro intervalová minima v L_{01}.

Analýza časové složitosti: Třídění v prvním volání trvá $\mathrm{Sort}(n, \Sigma)$, ve všech ostatních voláních 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

Ukkonenův algoritmus [39] konstruuje suffixový strom bez dolarů inkrementálně: začne se stromem pro prázdné slovo (ten má jediný vrchol, a to kořen) a postupně přidává další znaky na konec slova. To zvládne v čase $\mathcal{O}(1)$ amortizovaně na přidání jednoho znaku. 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 hashováním.

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

1. Pokud β byl nevnořený suffix slova σ, je i βa nevnořený suffix σa. Z toho víme, že listy zůstanou listy, pouze jim potřebujeme prodloužit nálepky. Pomůžeme si snadno: zavedeme *otevřené hrany,* jejichž nálepka je „od pozice i do konce“. Listy se tak o sebe postarají samy.
2. Pokud β bylo větvící slovo, zůstane nadále větvící – tedy vnitřní vrcholy ve stromu zůstanou.
3. Pokud β byl vnořený suffix (tj. vnitřní či skrytý vrchol), pak se βa buďto vyskytuje v σ, a tím pádem je to vnořený suffix nového slova a strom není nutné upravovat, nebo se v σ nevyskytuje a tehdy pro něj musíme založit novou odbočku a nový list s otevřenou hranou.

Víme tedy, co všechno musí algoritmus ve stromu pří rozšíření slova upravit, zbývá vyřešit, jak to udělat efektivně. K tomu se hodí pár definic a lemmat:

Definice: *Aktivní suffix* $\alpha(\sigma)$ říkáme nejdelšímu vnořenému suffixu slova σ.

Lemma: Suffix β slova σ je vnořený $\Leftrightarrow |\beta| \leq |\alpha(\sigma)|$.

Důkaz: Každý suffix vnořeného suffixu je opět vnořený. ♡

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 $|\alpha(\sigma a)| = |\beta a| \leq |\alpha(\sigma)a|$. ♡

Definice: Suffix βa je *zralý* $\equiv \beta$ je vnořený suffix σ, ale βa není podslovem σ (tedy musíme pro něj při přidávání znaku a k aktuálnímu slovu σ zakládat nový vrchol).

Lemma: Suffix β je zralý $\Leftrightarrow |\alpha(\sigma)a| \geq |\beta a| > |\alpha(\sigma a)|$.

Důkaz: Jelikož β je vnořeným suffixem σ, musí platit první nerovnost. Aby byl zralý, musí také nebýt vnořeným suffixem σa, a tomu odpovídá druhá nerovnost. ♡

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 vnitřní vrchol, α zkrátíme zleva o znak a testujeme dál.

Analýza: Úprav stromu provedeme $\mathcal{O}(1)$ amortizovaně (každá úprava slovo α zkrátí, každé přidání znaku ho prodlouží o znak, takže všech zkrácení je $\mathcal{O}(|\sigma|)$). Stačí tedy ukázat, jak provést úpravu v (amortizovaně) konstantním čase, k čemuž potřebujeme α reprezentovat šikovně a také si udržovat pomocné informace (zpětné hrany), abychom uměli rychle zkracovat.

Definice: *Referenční pár* je dvojice (π, τ), v níž π je vrchol stromu a τ libovolné slovo. Tento pár popisuje slovo $\pi\tau$. Referenční pár je *kanonický,* pokud neexistuje hrana vedoucí z vrcholu π s nálepkou, která by byla prefixem slova τ.

Pozorování: Ke každému slovu existuje právě jeden kanonický referenční pár, který ho popisuje. Všimněte si, že je to 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 ze všech vrcholů nejdelším vlastním suffixem slova π.

Pozorování: Zpětné hrany jsme sice zavedli stejně obecně, jako se to dělá při konstrukci vyhledávacích automatů podle Aha a Corasickové [1], ale v našem případě se *back* pro vnitřní vrcholy chová daleko jednodušeji (a na žádné jiné ho potřebovat nebudeme): pokud je π vnitřní vrchol, musí to být větvící podslovo, a tím pádem každé jeho zkrácení zleva musí být také větvící podslovo. Tedy $back(\pi)$ dá π bez prvního znaku, a to se nám bude hodit při zkracování suffixů.

Algoritmus podrobněji: (Doplnili jsme detaily do předchozího algoritmu.)

1. Vstup: $\alpha = \alpha(\sigma)$ reprezentovaný jako kanonický referenční pár (π, τ), T suffixový strom pro σ a jeho funkce *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 byl přítomen, tak α zkrátíme a test opakujeme:
11. Je-li $\pi \neq \varepsilon$, nastavíme $\pi := back(\pi)$. V opačném případě (jsme v kořeni) zkrátíme τ o znak zleva.
12. Pár (π, τ) už popisuje zkrácené slovo, ale nemusí být kanonický, takže to ještě napravíme:
13. Dokud existuje hrana vedoucí z π, jejíž popiska je prefixem slova τ, tak se po této hraně posuneme, čili prodloužíme π o tuto popisku a zkrátíme o ni τ.
14. Zpět na krok 2.
15. Pokud αa už je přítomen, zbývá přidat a k α a zastavit se:
16. $\tau := \tau a$.
17. Kanonikalizace stejně jako v bodech 12–13.[32]
18. Dopočítáme zpětné hrany (viz níže).
19. Výstup: $\alpha = \alpha(\sigma a)$ coby kanonický referenční pár (π, τ), T suffixový strom pro σa a jeho funkce *back*.

[32] Dokonce jednodušší, protože projdeme nejvýše jednu hranu.

Časová složitost:

Kanonikalizace pracuje v amortizovaně konstantním čase, protože každá její iterace zkrátí τ a za každé spuštění algoritmu se τ prodlouží jen jednou, a to o jeden znak.

Průchodů hlavním cyklem je, jak už víme, amortizovaně konstantní počet a každý průchod zvládneme v konstantním čase.

Zbývá dodat, jak nastavovat novým vrcholům jejich *back*. To potřebujeme jen pro vnitřní vrcholy (na zpětné hrany z listů se algoritmus nikdy neodkazuje) a 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 určitě ještě nebudeme tuto hranu potřebovat, protože π vždy jen zkracujeme, a tak můžeme vznik zpětné hrany o iteraci zpozdit a zvládnout to tak také v čase $\mathcal{O}(1)$.

Celkově je tedy časová složitost inkrementálního udržování suffixového stromu amortizovaně konstantní. ♡

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 [20], 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 [7], a ten zde ukážeme.

Taktéž jakmile už nějaké rovinné nakreslení máme, lze z něj celkem snadno vytvářet rovinná nakreslení s různými speciálními vlastnostmi. Za zmínku stojí například Schnyderův algoritmus [32] generující v lineárním čase nakreslení, v němž všechny hrany jsou úsečky a vrcholy leží v mřížových bodech mřížky $(n-2)\times(n-2)$, a o něco jednodušší algoritmus [10] kreslící do mřížky $(2n-4)\times(n-2)$.

Tak s chutí do toho ...

DFS a bloky

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

Definice: Prohledávání grafu (orientovaného nebo neorientovaného) do hloubky rozdělí E na čtyři druhy hran: *stromové* (po nichž DFS prošlo a rekurzivně se zavolalo; tyto hrany vytvářejí *DFS strom* orientovaný z kořene), *zpětné* (vedou do vrcholu na cestě mezi prohledávaným vrcholem a kořenem, čili do takového, který se právě

nachází na zásobníku, a v tomto směru si je zorientujeme), *dopředné* (vedou do již zpracovaného vrcholu ležícího v DFS stromu pod aktuálním vrcholem) a zbývající *příčné* (z tohoto vrcholu do jiného podstromu).

Lemma: Prohledáváme-li do hloubky neorientovaný graf, nevzniknou žádné dopředné ani příčné hrany. V orientovaném grafu mohou existovat dopředné a také příčné vedoucí „zprava doleva“, tedy do dříve navštíveného podstromu.

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). 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:

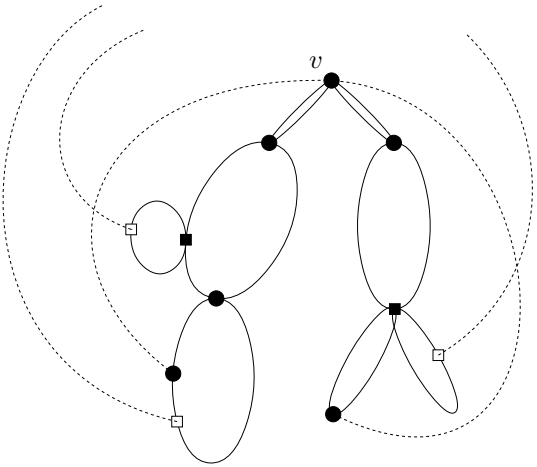
Lemma: Pokud v je vrchol grafu, u jeho otec a w jeho syn v DFS stromu, pak stromové hrany uv a vw leží v tomtéž bloku ($uv \sim vw$) právě tehdy, když *LowPoint*$(w) <$ *Enter*(v), a v je artikulace právě když některý z jeho synů w má *LowPoint*$(w) \geq$ *Enter*(v). Kořen DFS stromu je artikulace právě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

Postup kreslení

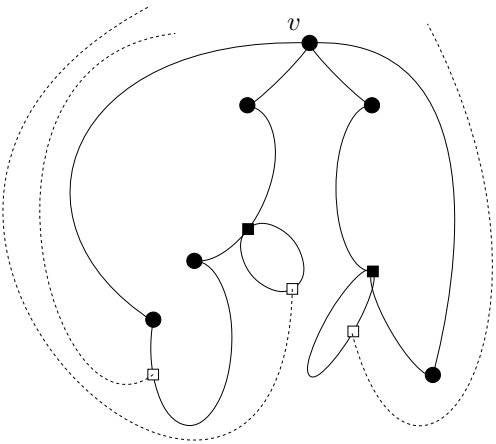
Graf budeme kreslit v opačném pořadí oproti DFS, tj. od největších *Enter*ů k nejmenším, a vždy si budeme udržovat blokovou strukturu již nakreslené části

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, můžeme artikulace naklonovat a každý blok pak 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 externě aktivní 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 *externě aktivní* 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 externě aktivní, 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 externě aktivní vrchol. Tehdy musíme některé bloky překlopit tak, aby externě aktivní vrcholy zůstaly venku. Potřebujeme tedy datové struktury, pomocí nichž bude možné překlápět efektivně a co víc, také rychle poznávat, kdy je překlápění potřebné.

Externí aktivita

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 externí aktivitu nadefinujeme tak, aby pokrývala i tyto případy:

Definice: Vrchol w je *externě aktivní,* pokud buďto z w vede zpětná hrana do ještě nenakresleného vrcholu, nebo je pod w připojen externě aktivní blok, čili blok obsahující alespoň jeden externě aktivní vrchol.

Jinými slovy vrchol w je externě aktivní 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 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 externě aktivní 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 externě aktivní, 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 externě aktivní, 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 post-processing, 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 podobně jako externí aktivitu nadefinujeme ještě živost vrcholu a ta bude odpovídat zpětným hranám vedoucím do v.

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. Není-li živý vrchol či blok externě aktivní, budeme mu říkat *interně aktivní.* Pakliže není vrchol/blok ani živý, ani externě aktivní, budeme ho nazývat *neaktivní.*

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.[33]

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

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

Důkaz: Alespoň polovina vrcholů, po nichž jsme v libovolném bloku prošli, zmizí z vnější stěny, takže hledání kořenů bloků trvá $\mathcal{O}(l)$. Pro každou zpětnou hranu označíme jeden vrchol jako živý a pak pokračujeme hledáním kořenů. ♡

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 neaktivní 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 externě aktivní vrchol (případně poté, co jsme ho ošetřili jako živý), procházení zastavíme, protože za externě aktivní vrchol již nemůžeme po této straně hranice nic připojit, aniž by se externě aktivní vrchol dostal dovnitř nakreslení.

Přitom se řídíme dvěma jednoduchými pravidly:

Pravidlo #1: V každém živém vrcholu zpracováváme nejdříve zpětné hrany do v, pak podřízené interně aktivní bloky a konečně podřízené externě aktivní 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 interně aktivnímu vrcholu, pokud takový neexistuje, pak k živému externě aktivnímu 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 neaktivních vrcholů. Pomůžeme si ovšem snadno:

[33] 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.

kdykoliv projdeme souvislý úsek hranice tvořený neaktivními vrcholy, přidáme pomocnou hranu, která tento úsek překlene. Můžeme ji dokonce přidat do nakreslení a podrozdělit si tak vnější stěnu.

Hotový algoritmus

Celý algoritmus bude vypadat takto:

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 končíme.
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.

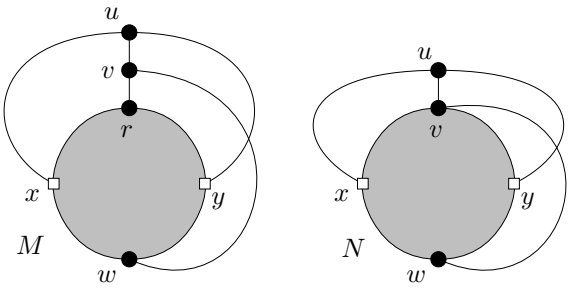
Důkaz: První krok je korektní, jelikož pro všechny rovinné grafy je $m \leq 3n-6$; nadále tedy můžeme předpokládat, že $m = \mathcal{O}(n)$. Lineární časovou složitost kroků 4–6 a 9 jsme již diskutovali, kroky 7–8 jsou lineární ve velikosti živého podgrafu, a tedy také $\mathcal{O}(n)$. Nakreslení vydané 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

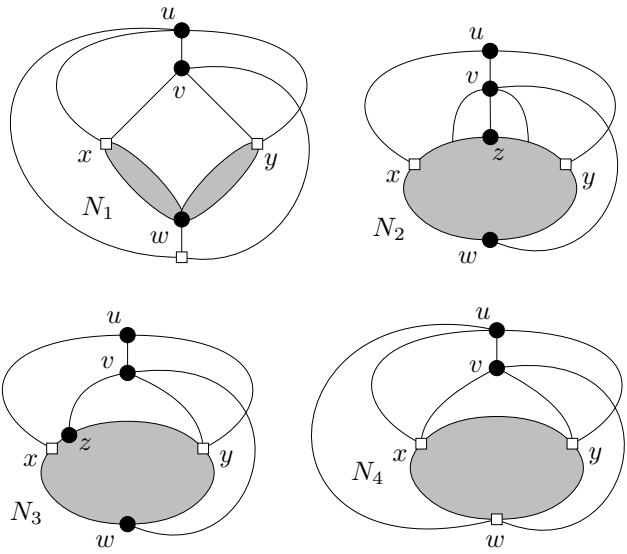
Lemma: Pokud existuje zpětná hrana, kterou algoritmus nenakreslil, graf na vstupu není rovinný.

Důkaz: Pro spor předpokládejme, že po zpracování vrcholu v existuje nějaká zpětná hrana wv, kterou algoritmus nenakreslil, čili že přístup z v k w je v obou směrech blokován externě aktivními vrcholy. Rozborem případů ukážeme, že tato situace vede ke sporu buďto s pravidly #1 a #2 nebo s rovinností grafu.

Označme B blok, ve kterém leží na obou stranách hranice nějaké externě aktivní vrcholy x a y a pod nimi je připojen nějakou cestou vrchol w. Takový blok musí určitě existovat, protože jinak by algoritmus všechny bloky na cestě z v do w popřeklápěl tak, aby se hrana wv vešla. V grafu se tedy musí vyskytovat jeden z následujících minorů (do vrcholu u jsme kontrahovali celou dosud nenakreslenou část grafu; vybarvená část odpovídá vnitřku bloku; hranaté vrcholy jsou externě aktivní):



Minor M přitom odpovídá situaci, kdy v neleží v bloku B. Tento případ snadno vyloučíme, protože M je isomorfní s grafem $K_{3,3}$. V grafu se proto musí vyskytovat N. Tento minor je ale rovinný, takže musíme ukázat, že vnitřek bloku brání nakreslení hrany vw dovnitř. Vždy pak dojdeme k některému z následujících nerovinných minorů (N_1 až N_3 jsou isomorfní s $K_{3,3}$ a N_4 s K_5):



Uvažme, jak bude B vypadat po odebrání vrcholu v a hran z něj vedoucích:

a) *přestane být 2-souvislý* – tehdy se zaměříme na bloky ležící na cestě xy:

   1) w *je artikulace* na této cestě – BÚNO je taková artikulace v DFS prohledána po bloku obsahujícím x, ale před y. Tehdy nám jistě x nezabránilo v tom, abychom do w došli (může blokovat jenom jednu stranu hranice), takže jsme se ve w museli rozhodnout, že přednostně zpracujeme pokračování cesty do y před hranou vw, a to je spor s pravidlem #1.

2) *w je v bloku připojeném pod takovou artikulací* – aby se pravidlo #1 vydalo do y místo podřízených bloků, musí být alespoň jeden z nich externě aktivní, takže v G je minor N_1.
3) *w je v bloku na cestě nebo připojen pod takový blok* – opět si všimneme, že do bloku jsme vstoupili mezi x a y. Abychom se podle pravidla #2 rozhodli pro stranu, z níž nevede hrana vw, musela na druhé straně být také hrana do v, a proto se v grafu vyskytuje minor M.

b) *zůstane 2-souvislý* a vznikne z něj nějaký blok B' – tehdy rozebereme, jaké hrany vedou mezi v a B':

1) *více než dvě hrany* – minor N_2.
2) *alespoň jedna hrana na „horní“ cestu* (to jest na tu, na niž neleží w) – minor N_3.
3) *dvě hrany do x, y nebo na „dolní“ cestu* – ať už jsme vstoupili na hranici bloku B' kteroukoliv hranou, pravidlo #2 nám řeklo, že máme pokračovat vrchem, což je možné jedině tehdy, je-li na spodní cestě ještě jeden externě aktivní vrchol, a to dává minor N_4. ♡

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.

L. Literatura

[1] A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

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

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

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

[7] 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.

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

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

[10] 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.

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

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

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

[14] 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.

[15] 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.

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

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

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

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

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

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

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

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

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

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

[26] 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.

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

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

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

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

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

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

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

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

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

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

[37] 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.

[38] 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.

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

[40] 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.

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

O. Obsah

Mgr. Martin Mareš

Krajinou grafových algoritmů

Vydal Institut Teoretické Informatiky
Univerzita Karlova v Praze
Matematicko-Fyzikální Fakulta
Malostranské nám. 25
118 00 Praha 1
jako 330. publikaci v ITI Series.

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

Vytisklo Reprostředisko UK MFF.

Vydání první
72 stran
Náklad 100 výtisků
Praha 2007

ISBN 978-80-239-9049-2