Grafové algoritmy

Michal Vaner

2. ledna 2013

Obsah

1 Maximální toky v grafu

Ford-Falkonson, Dinice - viz `ADS-II`.

Maximálně $O(n)$ fází, každá maximálně $O(m \times n)$. Celková složitost je tedy $O(m \times n^2)$.

Menger's theorem:

Pro každý graf (buď orientovaný nebo neorientovaný). a $s, t \in V$, maximální počet (hranově/vrcholově) oddělených cest z s do t odpovídá minimálnímu řezu/oddělovači $s - t$. Redukce na maximální tok je možný vždy v $O(n)$.

1.1 Speciální případy

1.1.1 Všechny hrany rovny 1

Jednoduchý odhad:
Všechny hodnoty rezerv jsou buď 0 nebo 1, po pročištění mají všechny hodnotu 1. Všechny se po nalezení blokujícího toku zaplní a tudíž se odeberou. Tedy, každá fáze trvá maximálně $O(m)$ - můžu odebrat maximálně m hran. Fází může být maximálně $O(n)$.

Odhad řezem dle hran:
Zkusíme lépe odhadnout počet fází. Po skončení k-té fáze je l alespoň $k+1$. Proto je alespoň $k+1$ vrstev a tudíž musí existovat dvě sousední vrstvy, mezi kterými je maximálně $\frac{m}{k}$ hran. Toto ale můžeme vzít jako řez, tedy celkový tok nemůžeme zlepšit již o víc než $\frac{m}{k}$. Protože každá fáze zlepší tok alespoň o 1, tak zbývajících fází bude již nejvýše $\frac{m}{k}$.

Nyní, pokud celkových fází bude méně než $\sqrt{m}$, tak je odhad hotov. Pokud by jich bylo více, vezmeme si stav v době, kdy je $k = \sqrt{m}$. Z toho dostaneme, že zbývá maximálně $\frac{m}{\sqrt{m}} = \sqrt{m}$ fází, tedy celkem jich bude $2\sqrt{m}$.

Celkem tedy bude algoritmus trvat $O\left(m^{\frac{3}{2}}\right)$.

Odhad řezem hle hladin:

Existují sousední hladiny takové, že $|L_i| + |L_{i+1}| \leq \frac{2n}{k}$. Hran mezi nimi je tedy maximálně $|L_i| \times |L_{i+1}| \leq \left(\frac{n}{k}\right)^2$. Nyní stačí obdobně, jako v minulém odhadu, zvolit $k := n^{\frac{2}{3}}$. Celkově tedy dostaneme $O(n^{\frac{2}{3}} \cdot m)$.

1.1.2 Všechny hrany rovny 1 a maximální stupeň (vstupní nebo výstupní) je nejvýše 1

Použije se stejný důkaz, jako u odhadu řezem dle hran, ale můžeme použít vrstvy, ne spojení mezi nimi. Potom lze nastavit $k := \sqrt{n}$. Celkově tedy dostaneme $O(\sqrt{n} \cdot m)$.

1.1.3 Celočíselné kapacity

Kdybychom začali na nějakém libovolném toku, tak složitost je $O(nm + n\Delta f)$, kde Δf je rozdíl maximálního a původního toku. Každé hledání blokujícího toku trvá $O(n)$ a zlepší to alespoň o 1. Ten zbytek trvá v každé fázi maximálně $O(m)$ a fází je maximálně $O(n)$.

1.2 Aplikace toků v síti

1.2.1 Maximální párováni v grafu

Mějme G bipartitní. Definujeme kapacitu každé hrany na 1 a jednu polovinu grafu připojíme ke zdroji a druhou k spotřebiči.

1.2.2 Maximální párování v grafu bez toků

Mějme G bipartitní k-regulární graf. Na k-regulárním bipartitním grafu lze najít perfektního párování. Po odebrání takového párování dostáváme $(k-1)$-regulární graf a udělat znovu totéž, čímž získáváme 1-faktorizaci grafu.

Odbočka:
Máme G, které má všechny stupně sudé a má sudý počet vrcholů v každé komponentě. Chceme jeho hrany rozdělit na $E_1, E_2 \subseteq E, E_1 \cap E_2 = \emptyset, E_1 \cup E_2 = E$ tak, že každý vrchol má stejný počet hran z obou množin. Lze udělat pomocí Eulerova tahu a ten rozdělit na sudé a liché hrany.

Předpokládejme, že $k = 2^t$. Pokaždé vezmu graf, rozdělím na E_1, E_2, jedno z toho vyhodím a dostanu $\frac{k}{2}$-regulární graf. Toto opakuji se vzniklým grafem, dokud mi nezbude 1-regulární graf, tehdy mám perfektní párování. Toto lze zvládnout v $O(m)$ - počet hran klesá geometrickou řadou, dělám to t-krát.

Pokud grafy nezahazuji, ale provádím vše, dostanu 1-faktorizaci v $O(t \cdot m)$.

Když k není 2^t, musím ho nějak zvětšit, aby byl. Pokud do udělám „normálně“, tak mi zbytečně naroste. Můžu ale přidat k hranám jejich násobnost (viz níže) a s těmi už to jde zvládnout. Jen u rozdělování si dám do každé výsledné množiny polovinu z násobnosti hrany (pokud jedna zbude, tak ji zpracuji normálně), což stále sběhne v $O(m)$.

Při nastavování násobnosti hran vyberu $\alpha := \left\lfloor \frac{2^t}{k} \right\rfloor$. $\beta := 2^t \bmod k$. Přidám do grafu $M_0 := \{v_i, w_i; i = 1, \ldots, n\}$ umělé hrany. Pak při hledání vybírám graf, který obsahuje nejméně umělých hran. Je třeba dokázat, že výsledek nebude obsahovat žádné takové hrany. Ale při každém kroku se jejich počet sníží alespoň $2\times$.

Každý krok trvá $O(m)$, t je maximálně $2 \log n$, tedy výsledek je $O(m \cdot \log n)$.

1.2.3 Měření k-souvislosti grafu

Máme graf G a chceme najít maximální k takové, že G je hranově k-souvislý. Lze zredukovat na nalezení minimálního řezu v grafu, k čemuž dokáží pomoci toky.

Nastavíme všechny kapacity na 1. Když se vyzkouší všechny dvojice zdroje a spotřebiče a z toho se vezme minimum, pak získáme složitost $O(n^{\frac{8}{3}} \cdot m)$. Lze dokázat, že stačí vzít jen jeden jako zdroj a vyzkoušet všechny spotřebiče, čímž se dostaneme na $O(n^{\frac{5}{3}} \cdot m)$.

Při hledání vrcholové k-souvislosti, chceme nejmenší oddělovač, tedy vložíme „doprostřed“ vrcholu hranu o kapacitě 1. Lze použít specializovanější dinicův algoritmus (se vstupním nebo výstupním stupněm nejvýše jedna) a vyzkoušíme všechny dvojice spotřebiče a zdroje, pak získáme složitost $O(n^{\frac{5}{2}} \cdot m)$. Trik s fixací zdroje neprojde – může být v oddělovači. Musíme vyzkoušet alespoň o jeden víc, než je velikost minimálního oddělovače, čímž se dostaneme na složitost $O(\kappa \cdot n^{\frac{3}{2}} \cdot m), \kappa$ je výsledek.

1.2.4 Minimální vážený řez

Nagamochi & Ibaraki

- $w(e)$ - váha hrany.
- $r(u, v)$ - váha minimálního řezu mezi u a v.
- $d(P, Q)$ - součet vah všech hran vedoucí z P do Q.
- $d(P) := d(P, \overline{P})$.
- $d(v) := d(\{v\})$ - vážený stupeň vrcholu.

Legální uspořádání na G je $v_1, \ldots, v_n; \forall i \forall j > i; d(\{v_i, \ldots, v_{i-1}\}, v_i) \geq d(\{v1, \ldots, v_{i-1}\}, v_j)$.

Lemma:
Když $v_1, \ldots, v_n$ je legální uspořádání na G, pak $r(v_{n-1}), v_1 = d(v_n)$.

Důkaz:
Mějme libovolný řez mezi C v_{n-1}, v_n. Chceme dokázat, že $|C| \geq d(v_n)$.

Řez dělí G na několik komponent. Definujme u_i tak, že $u_0 := v_1$ a libovolný jiný u_i je takový v_j, že $j > i$ a v_i a v_j leží v jiné komponentě.

$\forall i, d(\{v_1, \ldots, v_{i-1}\}, u_i) \leq d(\{v_1, \ldots, v_{i-1}\}, u_{i-1})$. V případě, že v_i i v_{i-1} jsou ve stejné komponentě, tak $u_i = u_{i-1}$, tedy platí rovnost. Pokud ne, pak $u_{i-1} = v_i$ a u_i je nějaké $v_{i+\epsilon}$. Pak to platí z definice legálního uspořádání.

$|C| \geq \sum_{i=1}^{n-1} d(v_i, u_i)$. Jde to z jedné komponenty do druhé, proto to musí být v řezu a ty hrany jsou různé (jdou jen doprava). To jde přepsat jako $\sum_{i=1}^{n-1} d(\{v_1, \ldots, v_i\}, u_i) - d(\{v_1, \ldots, v_{i-1}\}, u_{i-1})$. Použití minulého pozorování lze odhadnout: $\geq \sum_i d(\{v_1, \ldots, v_i\}, u_i) - d(\{v_1, \ldots, v_{i-1}\}, u_{i-1})$. Když se rozepíše a poodčítá, vyjde $d(\{v_1, \ldots, v_{n-1}\}, u_{n-1}) - d(\emptyset, u_0) = d(v_n)$.

Tedy libovolný řez musí být větší než vážený stupeň v_n v legálním uspořádání.

Nalezení legálního uspořádání:
Hladově. Začnu s prázdnou posloupností a u všech vrcholů si spočítám, jaký mají vážený stupeň do již spočítané posloupnosti. Pak vezmu vrchol s nejvyšším tímto číslem, zařadím na konec posloupnosti a přepočítám u zbylých vrcholů jejich vážený stupeň do posloupnosti.

Pro rychlé přepočítávání můžeme použít např. fibonacciho haldu. Poté bude nalezení legálního uspořádání stát $O(m + n \cdot \log n)$.

Nalezení minimálního řezu pro celý graf:
Minimální řez buď odděluje od sebe v_n a v_{n-1} a pak ho najde tento algoritmus. Pokud ne, pak můžeme tyto vrcholy zkontrahovat a zkusit to rekurzivně na zbytek grafu. Celá časová složitost je tedy $O(m \cdot n + n^2 \cdot \log n)$.

1.3 Maximální tok pro přirozené kapacity

$$\forall e; c(e) \in \mathbb{N}, c(e) \leq \kappa$$

Myšlenka je taková, že vezmeme napřed nejvyšší bit kapacit, spočítáme maximální tok, přidáme další bit a upravíme tok, atd.

Ten první tok je jednotkový, proto ho lze spočítat rychle.

$c_i(e)$ je kapacita e v i-tém kroku. $c_{i+1}(e) = 2c_i(e) \vee 2c_i(e) + 1$

Při updatu vezmu napřed dvojnásobek původního (to určitě můžu), zlepším to během Dinicova algoritmu. Lze dokázat, že update je jen hledání jednotkového toku. Každý takový tedy proběhne v $O(m \cdot n)$, celé to zběhne v $O(m \cdot n \cdot \log \max\{c_i\})$.

2 Minimální kostra

Mějme neorientovaný ohodnocený multigraf. Chceme najít kostru, která má nejmenší možné ohodnocení.

Pokud je T kostra, pak $T_{[x,y]}$ pro nějaké $x, y \in V$ je jediná cesta mezi x a y. Pokud $e = (x, y) \in E \Rightarrow T_{[e]} := T_{[x,y]}$ (může zcela neobsahovat e a vést okolo, když e není v kostře). Pak $T_{[e]}$ nazýváme ***cesta pokrytá*** e.

$e \in E$ je T***-lehká***, pokud $\exists e' \in T_{[e]}; w(e') > w(e)$ a e je T***-těžká***, pokud není T-lehká.

Pozorování:
T je kostra a $e \notin T$ je T-lehká, pak T není minimální kostra. Lze to vyměnit a dostat menší kostru.

Podmínka minimální kostry:
T je minimální kostra $\Leftrightarrow \nexists T$-lehká hrana.

Lemma:
$\forall T, T'$ kostry $\exists$ sekvence výměn hran, která transformuje T na T'.

Důkaz:
Vyjdeme z toho, že každá kostra má stejný počet hran. Pokud $T \neq T' \Rightarrow \exists e' \in (T' - T), \exists e \in (T_{[e']} - T)$.

Provedením takto naznačené výměny se zmenší ta množina $(T' - T)$ o jednu hranu a množina musela být konečná.

Lemma (o monotónních výměnách):
T, T' kostry a T' neobsahuje žádnou T-lehkou hranu. Pak $\exists$ sekvence výměn transformující T na T' takových, že se ohodnocení v žádném kroku nesníží.

$\forall e, e'$, které vyměňujeme, $w(e') \geq w(e)$. Stačí vybrat nejmenší možnou e', čímž nikdy nevytvořím T-lehkou hranu.

Poznámka:
Pokud jsou všechny váhy různé, pak existuje jen jedna minimální kostra.

2.1 Červeno-modrý meta-algoritmus

Přiřadíme každé hraně barvy, každá může být buď bezbarvá, červená, nebo modrá. Na začátku necháme všechny nenabarvené. V každém kroku použijeme buď červené nebo modré pravidlo.

Modré pravidlo vezme libovolný řez a $e :=$ jeho nejlehčí hranu. Když není modrá, přebarvím na modro.

Červené pravidlo vezme libovolný cyklus a $e :=$ jeho nejtěžší hrana. Pokud

není červená, obarvím ji na červeno.

Opakuje, dokud to jde (tedy, již není co barvit).

2.1.1 Důkaz správnosti

Lemma (modré):
Kdykoliv je hrana e natřená na modro, pak je v minimální kostře.

Důkaz:
Sporem. Mějme hranu, která je nejlevnější a není v kostře. To, co je řezem oddělené, musí být něčím spojené, ale protože to není tato hrana, musí to být nějaká těžší. Což je ale ve sporu s minimální kostrou.

Lemma (červené):
Kdykoliv je hrana e natřená na červeno, pak není v minimální kostře.

Důkaz:
Opět sporem. Tu nejtěžší hranu můžeme vyměnit a obejít to po tom cyklu.

Lemma (nebarevné):
Pokud existuje nebarevná hrana, pak lze aplikovat buď modré nebo červené pravidlo. Vezmu všechny modře dosažitelné vrcholy (B) z prvního vrcholu té hrany (x). Pokud druhý $y \in B$, pak lze sestavit cyklus tak, aby tato hrana byla očerveněná. Proto lze použít červené pravidlo. Pokud $y \notin B$, pak lze najít řez, který neobsahuje modrou hranu, proto lze použít modré pravidlo.

- Algoritmus se musí zastavit (nelze přebarvovat – hrana nemůže být v i mimo řez) a pokaždé můžeme jednu obarvit.
- Když se zastaví, všechny hrany jsou obarvené. Jasně plyne.
- Nakonec tvoří modré hrany minimální kostru. Plyne z modrého lemmatu.

2.1.2 Kruskalův algoritmus

Seřadí hrany vzestupně. Poté začne s prázdným modrým lesem a zkouší jednotlivé hrany.

Při každé hraně se podívá, jestli spojuje 2 různé komponenty modrého lesa, pak ji obarví modře (určitě existuje nebarevný řez, tato z nich je nejmenší). Pokud nespojuje, pak určitě lze vytvořit cyklus, kde zbytek je modrý, a obarvit na červeno.

Hodí se na to union-find struktura. Setřídíme v čase $O(f_S)$, tedy celková složitost je $O(f_S + m \cdot \alpha(n))$, α je inverzní ackermanova funkce.

2.1.3 Primův/Jarníkův algoritmus

Vezme jeden vrchol, prohlásí ho za základ modrého stromu a postupně roste. Vždy vezme nejlehčí hranu, co vede ven a tu tam přidá (určitě existuje řez, kde je nejlehčí – ten mezi modrým lesem a zbytkem). Až proroste celým grafem, ostatní hrany zbudou a lze je prohlásit za červené.

Triviální v čase $O(m \cdot n)$.

Lze si ale držet haldu hran, které mají alespoň jeden vrchol v modrém stromu. Pak při vybírání hrany buď spojuje dva modré vrcholy, pak je k ničemu a zahodíme ji. Když vede ven, použijeme ji a přidáme všechny nové hrany. Každá hrana se přidá maximálně jednou, tedy celá složitost je $O(m \cdot \log n)$ ($m = O(n^2), n^2$ se schová do 2 v logaritmu a to do O).

2.1.4 Borůvkův algoritmus

Jako Jarníkův, ale rostou paralelně. Stačí dokázat, že nevytvoří cyklus.

V každém kroku se sníží počet stromů alespoň o polovinu (každý strom se spojí alespoň s jedním). Počet kroků je tedy $O(\log n)$.

Každý lze provést na lineární čas s počtem hran, tedy celková složitost je $O(m \cdot \log n)$.

2.2 Rovinné grafy

Můžeme použít Borůvkův algoritmus, ale vrcholy kontrahovat, když se dostanou do stejné komponenty. (Smyčky se dají vyhazovat.)

Pro rovinný graf běží v lineárním čase.

2.2.1 Implementace

U každého vrcholu si pamatuji nejlevnější hranu vedoucí z něj. Spočítat toto mi bude trvat $O(m)$.

Spustíme BFS na graf se všemi vrcholy, ale jen těmito nejlevnějšími hranami. To vytvoří komponenty a ty zkontrahujeme. Stačí nadefinovat překladovou tabulku a vyházet paralelní hrany a cykly. Pomocí bucket-sortu (podle dvojic čísel vrcholů) se mi dostanou k sobě a tím je najdu.

2.2.2 Složitost

Každý krok je lineární. Každá kontrakce zachová rovinnost a sníží počet vrcholů alespoň $2\times$. Proto celková složitost je lineární.

2.2.3 Jiné grafy, kde funguje lineárně

Graf H je ***minor*** grafu G, pokud ho lze získat pomocí mazání hran či vrcholů a hranových kontrakcí.

Minorově uzavřená třída grafů je taková množina grafů, že kdykoliv vezmu libovolný graf, všechny jeho minory jsou v té třídě také.

Robersonova a Seymowrova věta:
Libovolná minorově uzavřená třída se dá charakterizovat konečnou množinou zakázaných minorů.

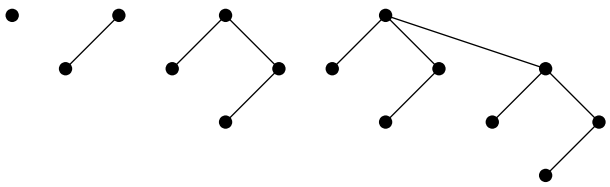
R & S věta 2:
C je minorově uzavřená třída. Potom C má omezenou hustotu.

Tedy, tento algoritmus běží lineárně pro libovolný graf z nějakého C.

3 Haldy

3.1 Binomiální

Les binomiálních stromů B_n. B_0 je jeden vrchol. B_k je strom B_{k-1} spojený s druhým B_{k-1} za kořeny.



Pozorování:

- $$|B_k| = 2^k$$
- B_k lze chápat jako vrchol, ke kterému jsou připojené $B_0, \ldots, B_{k-1}$.

Dále, žádná úroveň stromu není použitá dvakrát. To zaručuje, že halda velikosti n má pevně danou velikost.

Každá hrana zachovává uspořádání, že nahoře jsou menší.

3.1.1 Slití hald

Podobně jako binární sčítání. Když se potkají 2 stromy se stejnou úrovní, tak se spojí (tak, aby ten s lehčím kořenem byl nahoře) a vznikne jeden větší strom, který pokračuje jako přenos do další úrovně.

Toto lze stihnout $O(r)$, kde r je maximální úroveň stromu. r lze shora odhadnout logaritmem.

3.1.2 Vkládání

Vytvoříme haldu o 1 prvku, poté ji slijeme s původní.

3.1.3 Nalezení minima

Mohli bychom projít všechny kořeny najít ho, ale budeme si udržovat ukazatel na strom s nejmenším kořenem.

3.1.4 Odebrání minima

Odebráním kořene z některého stromu se rozpadne na menší podstromečky. Ty tvoří také binomiální haldu, tak ji slijeme.

3.1.5 Snížení

Obvyklým bubláním.

3.1.6 Odebrání libovolného

Stejné, jako snížení „až do kořene“ a odebrání.

3.2 Líná binomiální halda

Již nebudeme požadovat, aby každé úrovně stromu byl maximálně jeden exemplář.

3.2.1 Merge

Můžeme seznamy pouze spojit, což lze udělat konstantně.

3.2.2 Odebrání minima

Napřed odebereme, poté haldy slijeme a nakonec haldu vyčistíme a uděláme z toho normální binomiální haldu.

1. Setřídíme stromy podle úrovně (stále platí, že $r \leq \log n$, takže to můžeme udělat přihrádkově).
2. Projdeme to od nejmenších a spojujeme po dvojicích, přestrkujeme o 1 úroveň výš a ten max. jeden co zbude tu necháme.

Nechť t je počet stromů. Setřídění trvá $O(t + \log n)$, slívání trvá $O(t + \log n)$ (každé spojení jeden ubere, máme jich t).

Definujeme potenciál $\Phi := \#\text{stromů}$. Slití sečte oba Φ dohromady, přidání jednoho jeden vytvoří. Vyčištění bude odebírat. Odebrání bude trvat $O(\log n)$ a vytvoří $O(\log n)$ nových stromečků, tedy přidá $O(\log n)$ do Φ. Slitím dvou stromečků se spotřebuje 1 z Φ, celkem „Neslití“ může být jen $O(\log n)$, protože maximální r je stále $O(\log n)$. Tedy, na každý slitý stromeček si předem „našetří“

3.3 Fibonacciho halda

Je to les zakořeněných stromů, jeho tvar se ukáže dále. Každý vrchol obsahuje:

- Element
- Seznam synů
- Stupeň (počet synů)
- Barvu (buď černá nebo bílá)
- Otce

Uspořádání na vrcholech je obvyklé. Každý kořen je bílý.

Většina operací je stejná jako u líné binomiální haldy.

3.3.1 Slití, přidání

Jako u binomiální haldy – jednoduše se seznamy spojí k sobě.

3.3.2 Řez

Pokud vrchol V není kořen, tak ho odpojí od otce a uděláme ho kořenem (s přebarvením na bílo). Nesmíme řezat 2 syny stejného vrcholu, proto si budeme pomocí barvy pamatovat otce, kteří již ztratili syna, jako černé.

Pokud sebereme syna černému vrcholu, tak ho uřízneme také, takto postupujeme směrem nahoru ke kořeni.

Kořen nikdy nepřebarvíme načerno, i kdyby mu byl brán syn (ten lze virtuálně uříznout a zařadit mezi kořeny kdykoliv bez práce).

3.3.3 Odebrání minima

Halda obsahuje ukazatel na strom s nejmenším prvkem. Tomuto stromu se vezme vrchol, čímž se rozpadne na $O(d)$ stromů ($d := \max^\circ(v) | \forall v \in V$). Kořeny těchto stromů se přebarví na bílo a stanou se z nich kořeny.

Pak se pustí uklizení. Vrcholy se stejným stupněm se vždy spojí dohromady, podobně jako u binomiální haldy, poté budu mít maximálně d stromů.

3.3.4 Snížení

Pokud je v kořen nebo není po zvýšení dost malý, neděje se nic. Pokud by se snížil pod otce, tak ho řízneme.

3.3.5 Časová složitost

$$\Phi := \#\text{stromů} + 2 \cdot \#\text{černých vrcholů}$$

Úroveň vrcholu v u kořene bereme jako jeho stupeň, pokud přestane být kořenem, pak se mu již nemění.

Pozorování:
Stupeň vrcholu může být maximálně o 1 menší než jeho úroveň.

Uklízení probíhá stejně, jako u líných binomiálních hald, tedy trvá amortizovaně $O(d)$.

Při každém řezu (kde se něco děje, zanedbáme triviální řezy) se spotřebuje konstantní čas na zpracování řezaného vrcholu a vznikne jeden strom, tedy můžeme přidat do potenciálu. Pokud bubláme nahoru, spotřebováváme „černý“ potenciál, jeden za nový strom a jeden za zpracování vrcholu.

Invariant (1):
Pokud máme vrchol se stupněm r a k synů se stupni $r_0 \leq r_1 \leq \ldots \leq r_{k-1}$, potom $r - 1 \leq k \leq r$ a $r_0 \geq 0$.

Důkaz:
Při vytvoření nového stromu, potom vznikne spojením dvou $r-1$ stromů, proto tento dostane úroveň r. V horním stromu indukce platí. Přidáním nového syna pod ten kořen se to tím nezhorší, protože má úroveň $r-1$.

Nezničí to ani řez, pokud je to kořen, tak se úroveň správně upraví, jinak může ztratit maximálně jednoho syna.

Invariant (2):
Pokud v má úroveň k, potom $|T_v| \geq h_k$ (T_v je strom zakořeněný v v).

$$\begin{aligned} h_0 &= 1 \\ h_1 &= 1 \\ h_2 &= 2 \\ &\vdots \\ h_k &= 1 + h_0 + h_0 + h_1 + h_2 + \ldots + h_{k-3} \end{aligned}$$

Dostaneme sečtením podstromů a v samotného. Tato funkce roste podobně jako fibonacciho čísla, jen jsou o jedničku větší. Toto lze dokázat indukcí a dosazením.

TODO: Tadyten důkaz není kompletní, doplnit a pochopit

3.3.6 Použití

- Zrychlení Dijkstry.
- Jarníkův algoritmus (Fredman & Tarjan). Pamatujeme si, jak nejlevněji se dostaneme do některého vnějšího bodu. To děláme v této haldě.
- Lze ještě zrychlit tak, že napřed pustíme $\log \log n$ kroků Borůvkova algoritmu. Tím se sníží počet vrcholů alespoň log-krát. Celé to tedy běží v $O(m \log \log n)$.
- Celé to jde zrychlit ještě tak, že omezíme velikost haldy (aby to běhalo rychleji) a když halda dojde, tak začneme nový strom, ty se dají spojovat.

4 Výpočetní modely

4.1 Ukazatelový stroj

Existují dva datové typy:

- Malé celočíselné typy (existuje horní hranice na počet hodnot, které může nabývat)
- Ukazatele

Při výpočtu je k dispozici omezené množství registrů pro ukazatele a integery a neomezené množství paměti, kam se dá ukazovat.

Jediné operace běžící v konstantním čase jsou operace na konstantně velkých datech a dereference ukazatelů.

4.2 RAM

„Random Access Machine“.

Umí pracovat s libovolně velkými integery, má paměť indexovanou těmito integery a jsou v ní zase tyto integery.

Z ukazatelového stroje na RAM lze přecházet přímo. Opačně se musí ukládat paměť někam do stromu, takže je $\log n$ pomalejší (při přímém převodu).

5 Datové struktury pro integery

5.1 Van Emde-Boas Tree

Vyhledávací strom, pamatuje si integery z $X \subseteq \{0, 1, 2, \ldots, u-1\}$ a pracuje v $O(\log \log u)$.

$VEBT(u)$:

- Min.
- Max.
- Přihrádky $B_0, \ldots, B_{\sqrt{u}-1}$ uložené také jako $VEBT(\sqrt{u})$.
- shrnující strom S jako $VEBT(\sqrt{u})$ udržující seznam neprázdných přihrádek.

Minimum a maximum není uložené v žádné přihrádce.

Nechť $u = 2^{2^k}$ (jinak lze konec vynechat).

5.1.1 Vložení

Pokud je tam jeden nebo žádný prvek (min je max a nebo je zcela prázdný), pak jen upravuji minimum a maximum.

Jinak, zkontroluji, jestli nenahrazuji minimum nebo maximum a případně pokračuji s tím, co bylo minimum/maximum. Pokud je přihrádka, kam strkám, prázdná, vytvořím novou, zapíšu si ji do S. V každém případě nakonec vložím do té přihrádky rekurzivně. (Jedno z toho bude triviální případ, proto se to nerozdělí na 2 větve.)

5.1.2 Získání

Obdobně.

5.1.3 Následník

(Vstupem může být i číslo, které tam není, prostě vrátí něco většího.)

- Napřed zkontrolujeme triviální věci.
- $< min \longrightarrow$ vrátí minimum.
- $\geq max \longrightarrow$ není tu.
- Zkusím se podívat do příslušné přihrádky, když se to nepovede, pomocí shrnujícího stromu najdu následující neprázdný strom a najdu v něm minimum.
- Pokud nenajdeme následující přihrádku, vrátí lokální maximum.

5.1.4 Mazání

Obdobně, křížené přidání a hledání následníka (pro nahrazování minima apod.).

5.2 Užitečné funkce

5.2.1 Zjištění, že x je mocnina 2

Porovnám x a $x - 1$. Když na tom udělám bitové $\wedge$, pak dostanu 0 pro mocninu 2 (s výjimkou $x = 0$).

5.2.2 Práce s vektory

Máme nějakou posloupnost vektorů, každý má b bitů a každý má zleva 1 bit zarážku.

- Replikace (dostat nějakou hodnotu do všech hodnot) – vynásobením x vektorem samých jedniček.
- Součet prvků – např. $x \bmod 2^{b+1} - 1$, nebo vynásobením vektorem se samými jedničkami a použitím správného kusu (viz násobení pod sebe).
- Vektorové porovnání (jednička vyjde, pokud složka prvního je menší než složka druhého). Paddingy nastavím v jednom na jedničky, odečtu druhý, zbude nebo nezbude tato jednička (pak stačí jen správně posunout a vy-andit).
- Počet prvků menších než α – zreplikujeme α, vektorově porovnáme a sečteme je.
- Rozbalení α – vypsat bity každý zvlášť – replikace α, vy-andovat to šikovnou konstantou, porovnáme s nulovým vektorem.
- Zabalení – opačně – představíme si, že to má $b - 1$ bitové složky a sečteme (problém je jen ten, že tam nejsou zarážky).

5.3 Stromy

(a, b)-strom. Každý vrchol kromě kořenu a listů má alespoň a a nejvýše b synů, kořen má alespoň 2 a maximálně b synů. Obsahuje $k(v)$ – vektor klíčů, seřazené a $p(v)$ – seznam ukazatelů na data. Operaci na jedné úrovni dokážeme dělat konstantně.

Celkově tedy můžeme operaci udělat logaritmicky s počtem vrcholů.

Hodilo by se mít výšku konstantní, pak můžeme upravovat jiné parametry, ale máme omezený maximální počet hodnot uložených.

Pokud velikost slova je alespoň $O(\log m)$ – aby šel přečíst vstup – tak to na kostry bude stačit.

5.4 Q-Heap

TODO: Tady to moc nedává smysl, celej QHeap Zadefinujme parametry:

w Velikost slova

n Velikost vstupu

k Velikost haldy, $k := w^{\frac{1}{4}}$

r Aktuální počet prvků, $r \leq k$

$X = \{x_1, \ldots, x_r\}$ Prvky

c_i Nejvyšší bit rozdílu x_i a x_{i+1}

Inicializace bude trvat $O(2^{x^4})$, ale operace pak bude konstantní. Při malé velikosti se stihne v $O(n)$.

Můžeme si předpočítat tabulky výsledků pro funkci se vstupem $O(x^3)$ a polynomiálním čase pro vyhodnocení tak, že vyhledání trvá konstantně dlouho.

5.4.1 První nástřel

Kdybychom chtěli spočítat $rank_X(y)$, tak budeme mít radix strom na X a zkomprimujeme jej (přeskočíme „cestičky“). Můžeme tedy testovat jen v bitech, které se liší.

Pak, když podle toho najdeme list (a ten ani nemusí být stejný), pak rank je specifikován:

- Tímto strome
- Indexem listu
- Porovnání listu s hodnotou
- $MSB(y \oplus x_i)$

Můžeme si je předpočítat, indexování stromem se bude provádět pomocí reprezentace stromu pomocí $c_1, \ldots, c_{r-1}$.

Celé rank tedy půjde (po preprocesingu) vyhodnotit v $O(1)$.

Problém je s úpravami – nemohu si udržovat hodnoty seřazené, tak je budeme mít neseřazené a vektor, který obsahuje permutaci a říká, jak jsou správně seřazené.

5.4.2 Insert

Spočítáme rank, přidáme do hodnot a přepočítáme permutaci. Nakonec musíme změnit hodnoty v C.

6 Sufixové stromy

6.1 Značení

- Σ – konečná abeceda

- Σ^* množina konečných řetězců nad Σ
- $|\alpha|$ je délka α
- ϵ je prázdný řetězec ($|\epsilon| = 0$)
- $\alpha\beta$ je zřetězení α a β
- $\alpha[i]$ je i-tý znak (začínáme od nuly)
- $\alpha[i:j]$ je $\alpha[i]\alpha[i+1]\ldots\alpha[j-1]$, $j \leq i \Rightarrow \alpha[i:j] = \epsilon$
- $\alpha[i:]$ je sufix začínající v i
- $\alpha[:j]$ je prefix končící před j
- $\alpha[:] = \alpha$

α je ***podřetězec*** β ($\alpha \subset \beta$) $\Leftrightarrow \exists\gamma, \delta; \beta = \gamma\alpha\delta$. Podřetězec se nazývá ***prefix***, pokud $\gamma = \epsilon$ a ***sufix***, pokud $\delta = \epsilon$.

Každý podřetězec je prefix nějakého sufixu (nebo naopak).

Mějme $X \subset \Sigma^*$, X konečné. ***Trie*** je graf, kde každý vrchol je nějaký prefix nějakého řetězce z X a hrana mezi α a β vede $\Leftrightarrow \beta = \alpha x, x \in \Sigma$.

Můžeme je zkomprimovat a cesty, které se nevětví, zkrátit na jednu hranu. Pak se to nazývá ***komprimovaná trie***.

Sufixový strom je komprimovaná trie pro všechny sufixy nějakého σ.

Pokud přidáme nějaký „ukončovací“ znak (značme ho \$, nějaký, který se nikde ve slovech nevyskytuje), pak nejsou žádné skryté sufixy v hranách.

Velikost:
Lze reprezentovat v $O(|\sigma|)$.

Důkaz:
Listů je nejvýše $|\sigma| + 1$ a protože každý vnitřní vrchol má výstupní stupeň alespoň 2, pak je jich také lineárně mnoho. Cedulky hran jsou podřetězce, můžeme si pamatovat jen konce.

Rychlost:
Lze vytvořit v $O(|\sigma|)$. (Pro nějaké pevné Σ)

6.2 Aplikace

- Obrácené podřetězcové dotazy – seno si připravíme a odpovídáme, kde všude se daná jehla vyskytuje.
- Nejdelší opakující se podřetězec – nejnižší dělící se vrchol.

- Histogramy řetězců délky k – uříznu si vršek, na listech vidím, kolik toho tam původně viselo.
- Nejdelší společný podřetězec – sestavím to na obou řetězcích spojených nějakým oddělovačem.

Permutace na číslech $1, 2, \ldots, |\sigma| + 1$, které setřídí sufixy lexikograficky, nazveme ***sufixové pole***. Budeme jej značit SA_X

Dále, ***pole nejdelších společných prefixů*** je pole, kde každý prvek je nejdelší společný prefix dvou po sobě jdoucích sufixů v SA_X. Značíme LCP_X.

Lemma:
Sufixové pole a pole nejdelších společných prefixů je lineárně ekvivalentní se sufixovým stromem. Ze stromu na ně je to jednoduché.

Opačně přes výběry minim v poli nejdelších společných prefixů. (Pokaždé, když potkám ϵ, dostanu se do kořene atp.)

6.2.1 Problémy LCA a RMQ

(Least Common Prefix, Range Minimum Request.)

RMQ lze převést na LCA. Vytvoří se ***kartézský strom*** – strom, kde v kořeni je minimum a levý a pravý podstrom jsou kartézské stromy levé a pravé půlky. To jde vytvořit lineárně. Vyhledávání poté funguje přímo.

TODO: Jak vytvořit

LCA je možné převést na RMQ. Postavím RMQ tak, že projdu do hloubky a pamatuji si hloubku každého vrcholu, pak dotaz je nalezení minima mezi vrcholy. Jediný problém je, že vrchol tam může být víckrát, ale je jedno, který z nich použiji. Lze dokázat, že celková velikost bude $2n$.

Tento RMQ, který vyjde, je takový, že se sousední liší pouze o 1. To lze vyřešit buď hloupě, předpočítáním v $O(n^2)$ všech možností (dvourozměrná tabulka), nebo předpočítáním všech intervalů délek 2^k, předpočítání v $O(n \cdot \log n)$ a dotaz v $O(1)$ (najdou se dva menší intervaly, které to dohromady přesně pokrývají, vybere se to menší z výsledků).

Můžeme rozdělit vstup na bloky délky b, dostaneme n/b bloků. Pak můžu nahradit každý blok jeho minimem, dotaz provést tak, že se podívám do okrajových bloků a minimum z toho vnitřku přes bloky. Pokud b je dostatečně malé, pak je spoustu bloků stejných, dají se popsat klikaticí a posunem, klikatic je jen 2^b.

Pro každý z bloků můžu použít hloupou verzi, na předpočítání je potřeba $O(n + 2^b b^2)$.

Na zbytek použijeme to s logaritmem, tedy $O(\frac{n}{b} \cdot \log \frac{n}{b})$. Když zvolím b jako $\frac{1}{2} \cdot \log_2 n$, tak celý preprocesing bude trvat $O(n)$.

6.2.2 Algoritmus na tvorbu těchto dvou polí

Algoritmus je rekurzivní.

1. Zmenšit abecedu na $1, \ldots, n$ – vyhází ty znaky, které nejsou použité.
2. Definujeme tři pomocné řetězce: Každý z nich bude sestaven ze znaků sloučených vždy z trojice znaků původních, na pozicích dělitelných třemi, o 1 doprava a o 2 doprava.
3. Zrekurzit na první a druhý řetězec spojené za sebe.
4. Oddělit jednotlivá pole suffixů.
5. Spočítá se pole suffixů pro třetí řetězec. Porovnávat lze podle prvního znaku a pole pro první řetězec. Tohle jde udělat přihrádkovým tříděním.
6. Spojit tyto pole dohromady pomocí jednoho kroku mergesortu, podobný trik na porovnávání.
7. Podobně spočítat pole nejdelších společných prefixů, pomocí RMQ (každý prvek je prefix dvou sousedních sufixů) pro výsledek rekurzivního volání.

TODO: Doplnit