9. Dekompozice stromů

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

Union-Find Problem

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

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

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

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

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

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

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

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

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

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

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

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

K tomu nám ovšem stačilo samotné pravidlo Union podle ranku, o kompresi cest jsme zatím dokázali pouze to, že složitost v nejhorším případě nezhoršuje.⟨2⟩ Kombinace obou metod se ve skutečnosti chová mnohem lépe:

Věta: (Tarjan, van Leeuwen [7]) Posloupnost m operací *Union* a *Find* provedená na prázdné struktuře s n vrcholy trvá $\mathcal{O}(n + m\alpha(m, n))$, kde α je inverzní Ackermannova funkce.⟨3⟩

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

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

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

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

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

⟨2⟩ Mimochodem, Komprese cest samotná by také na složitost $\mathcal{O}(\log n)$ amortizovaně stačila [7].

⟨3⟩ Je známo [6], že asymptoticky lepší složitosti nelze dosáhnout, a to ani v modelu silnějším než RAM. Námi uváděný algoritmus si téměř vystačí s Pointer Machine, jen porovnávání ranků z tohoto modelu vybočuje. Složitost operací v nejhorším případě je obecně horší, je znám dolní odhad $\Omega(\log n / \log \log n)$; více viz Alstrup [1].

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

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

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

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

♡

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

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

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

Union-Find s předem známými Uniony

Dále nás bude zajímat speciální varianta Union-Find problému, v níž dopředu známe posloupnost Unionů, čili strom, který spojováním komponent vznikne.[4] 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 [3, 2].

[4] Kdy se to hodí? Třeba v Thorupově lineárním algoritmu [8] na nejkratší cesty nebo ve Weiheho taktéž lineárním algoritmu [9] na hledání hranově disjunktních cest v rovinných grafech.

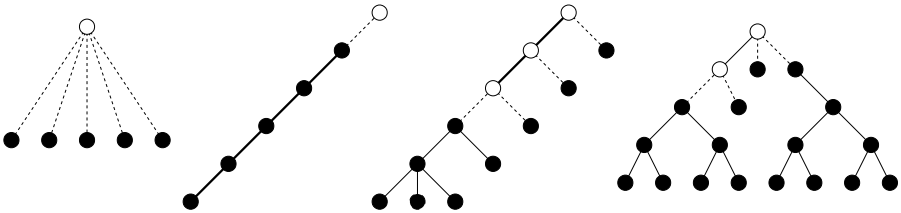
Definice: *(Microtree/Macrotree dekompozice)* Pro zakořeněný strom T o n vrcholech definujeme:

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

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

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

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



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

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

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

Find(x, y) :

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

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

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

Union(x, y):

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

Find(x, y):

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

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

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

Union(x, y) :

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

Find(x, y) :

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

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

Union(x, y):

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

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

Find(x, y):

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

Analýza: Operace *Find* trvá konstantní čas, protože se rozloží na $\mathcal{O}(1)$ *Find*ů v dílčích strukturách a každý z nich trvá konstantně dlouho. Všech n operací *Union* trvá $\mathcal{O}(n)$, jelikož způsobí $\mathcal{O}(n)$ amortizovaně konstantních operací s mikrostromy, spojovacími hranami a cestami a $\mathcal{O}(n/\log n)$ operací s makrostromem, které trvají $\mathcal{O}(\log n)$ amortizovaně každá.⟨5⟩

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

Fredericksonova clusterizace

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

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

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

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

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

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

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

Věta: (Frederickson [5]) Každá k-clusterizace stromu T na n vrcholech obsahuje $\mathcal{O}(n/k)$ clusterů.

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

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

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

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

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

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

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

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

Věta: (Frederickson [5]) Pro každé k lze k-clusterizaci stromu o n vrcholech najít v čase $\mathcal{O}(n)$.

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

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

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

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

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

Stromoví předchůdci

Problém: *(Least Common Ancestor alias LCA)* Chceme si předzpracovat zakořeněný strom T tak, abychom dokázali pro libovolné dva vrcholy x, y najít co nejrychleji jejich nejbližšího společného předchůdce.

Triviální řešení LCA:

- Vystoupáme z x i y do kořene, označíme vrcholy na cestách a kde se poprvé potkají, tam je hledaný předchůdce. To je lineární s hloubkou a nepotřebuje předzpracování.
- Vylepšení: Budeme stoupat z x a y střídavě. Tak potřebujeme jen lineárně mnoho kroků vzhledem ke vzdálenosti společného předchůdce.
- Předpočítáme všechny možnosti: předzpracování $\mathcal{O}(n^2)$, dotaz $\mathcal{O}(1)$.
- … co dál?

Věrni vtipům o matfyzácích a článku [4] 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 $\min_{x \le i \le y} a_i$.⟨6⟩

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

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

Triviální řešení RMQ:

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

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

poté někdo zeptá na minimum bloku $a_i, a_{i+1}, \ldots, a_{j-1}$, najdeme největší k takové, že $2^k < j - i$ a vrátíme:

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

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

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

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

Všimneme si, že ačkoliv bloků je mnoho, jejich možných typů (tj. posloupností klesání a stoupání) je pouze $2^{b-1} \leq \sqrt{n}$ a bloky téhož typu se liší pouze posunutím o konstantu. Vybudujeme proto kvadratickou strukturu pro jednotlivé typy a pro každý blok si zapamatujeme, jakého je typu a jaké má posunutí. Celkem strávíme čas $\mathcal{O}(n + \sqrt{n} \cdot \log^2 n) = \mathcal{O}(n)$ předzpracováním a $\mathcal{O}(1)$ dotazem.

Mimo to ještě vytvoříme komprimovanou posloupnost, v níž každý blok nahradíme jeho minimem. Tuto posloupnost délky n/b budeme používat pro části dotazů týkající se celých bloků a připravíme si pro ni „logaritmickou“ variantu triviální struktury. To nás bude stát $\mathcal{O}(n/b \cdot \log(n/b)) = \mathcal{O}(n/\log n \cdot \log n) = \mathcal{O}(n)$ na předzpracování a $\mathcal{O}(1)$ na dotaz.

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

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

Lemma: Kartézský strom je možné zkonstruovat v lineárním čase.

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

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

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

Výsledky této podkapitoly můžeme shrnout do následující věty:

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

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

Literatura

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

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

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

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

[5] 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.

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

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

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

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