Kapitola 1

Union-Find problém

Motivace: Po světě se toulá spousta agentů. Často se stává, že jeden agent má spoustu jmen/přezdívek, které používá například při rezervaci hotelu, restaurace, na návštěvě u zajímavých osobností. Tato jména si nevolí úplně náhodně, protože na každé jméno musí mít pravé doklady a ty je těžké sehnat. Každý agent má sadu různých jmen a dokladů na ně. Tajná služba postupně odhaluje ekvivalentní jména, tj. jména patřící stejnému agentovi. Tajná služba by ráda používala systém, do kterého si postupně bude ukládat nalezené dvojice ekvivalentních jmen a kterého by se mohla zeptat, jestli je určitá dvojice jmen ekvivalentní. Například by se chtěli zeptat: „Je agent 007 a James Bond ten samý člověk?“.

Úkol: (grafový pohled) Všechna jména agentů odpovídají vrcholům grafu $G = (V, E)$. Dvojice ekvivalentních jmen odpovídá hraně v grafu. Všechny přezdívky jednoho agenta tvoří komponentu souvislosti. Chceme se systému ptát: „Leží vrcholy u a v ve stejné komponentě souvislosti?“. Problému se také někdy říká dynamické udržování komponent souvislosti a nebo problém udržování ekvivalence.

Komponenta souvislosti určená vrcholem v je množina $C_v = \{u \in V \mid \exists \text{cesta z } u \text{ do } v\}$. V každé komponentě souvislosti vybereme jednoho reprezentanta. Pro jednoduchost budeme reprezentanta komponenty C_v značit r_v, takže pokud u a v leží ve stejné komponentě, tak $r_u = r_v$. Úkol budeme realizovat pomocí operací:

`FIND`$(v) = r_v$, operace vrátí reprezentanta komponenty souvislosti C_v.

`UNION`(u, v) provede sjednocení komponent souvislosti C_u a C_v. To odpovídá přidání hrany uv do grafu.

1.1 Triviální řešení

Předpokládejme, že vrcholy grafu jsou očíslované čísly 1 až n. Použijeme pole `R`$[1..n]$, kde `R`$[i] = r_i$, tj. číslo reprezentanta komponenty C_i. Operace `FIND` pouze vypíše `R`$[v]$ a tedy bude trvat $\mathcal{O}(1)$. K provedení `UNION`(u, v) najdeme reprezentanty $r_u =$ `FIND`(u), $r_v =$ `FIND`(v). Pokud jsou různí, tak projdeme celé pole `R`$[\cdot]$ a každý výskyt r_u přepíšeme na r_v. To nám zabere čas $\mathcal{O}(n)$.

1.2 Často dostačující řešení

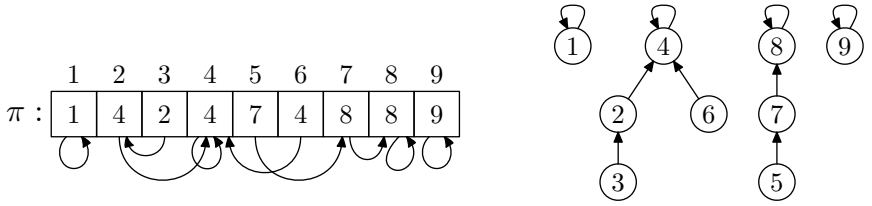
U každého reprezentanta r si pamatujeme ještě `size`$[r]$ = # prvků v komponentě C_r. A pro každou komponentu si pamatujeme spojový seznam jejich prvků. Seznam lze realizovat jako pole `next`$[\,\cdot\,]$ obsahující číslo dalšího prvku v seznamu. Seznam ukončíme nulou.

Předchozí řešení pozměníme tak, že v operaci `UNION` přepíšeme pouze prvky menší komponenty. Díky spojovému seznamu prvků menší komponenty to zvládneme v čase úměrném počtu prvků komponenty. Nakonec zřetězíme seznamy prvků obou komponent a sečteme velikosti komponent.

V nejhorším případě bude operace `UNION` trvat opět $\mathcal{O}(n)$. Ale všimněme si, že prvek i přepisujeme jenom tehdy, když komponentu C_i sjednotíme s druhou komponentou, která má alespoň tolik prvků jako C_i. Díku tomu může být každý prvek přepsán nejvýše $(\log n)$-krát. Proto bude n operací `UNION` dohromady trvat pouze $\mathcal{O}(n \log n)$. To dává amortizovaný čas pro `UNION` $\mathcal{O}(\log n)$.

1.3 Řešení s přepojováním stromečků

Další možnost, jak si pamatovat komponentu C_v je pomocí orientovaného stromečku pokrývajícího komponentu. Z každého vrcholu v povede orientovaná hrana (šipka) do jeho předchůdce $\pi(v)$ ve stromě. Z kořene r povede smyčka zpátky do kořene, tj. $\pi(r) = r$. K reprezentaci stromečků všech komponent nám tedy stačí jen pole $\pi[\cdot]$. Pro každý vrchol v si budeme navíc pamatovat délku nejdelší orientované cesty vedoucí do v a označíme ji $rank(v)$. Proto se tomuto řešení anglicky říká *union by rank*.



Začneme s grafem, který se skládá z izolovaných vrcholů a proto na začátku pro každý vrchol v nastavíme $rank(v) = 0$ a $\pi(v) = v$. V operaci `FIND`(v) najdeme reprezentanta komponenty C_v tak, že z v půjdeme po orientovaných hranách až do kořene. Kořen r_v je hledaný reprezentant komponenty C_v. Operace `UNION`(u, v) proběhne tak, že si nejprve najdeme reprezentanty r_u a r_v a pak natáhneme hranu z kořene stromu s menším rankem do kořene stromu s větším rankem. Tím vznikne nový strom reprezentující sjednocenou komponentu. Pokud budou mít oba stromy stejný rank, tak natažením hrany z r_u do r_v vznikne strom s o jedna větším rankem. Jinak se rank nezmění.

```
FIND(v):
    while v ≠ π(v) do
        v := π(v)
    return v


UNION(u, v):
    r_u := FIND(u)
    r_v := FIND(v)
    if r_u = r_v then return
    if rank(r_u) > rank(r_v) then
        π(r_v) := r_u
    else
        π(r_u) := r_v
        if rank(r_u) = rank(r_v) then
            rank(r_v) := rank(r_u) + 1
```

Časová složitost `FIND`(v) bude odpovídat výšce stromu, který procházíme, a to je $rank(r_v)$. Časová složitost `UNION` bude zhruba 2 krát tolik než časová složitost `FIND`.

Všimněme si, že pro každé v je $rank(v) < rank(\pi(v))$. Kořen s rankem k vznikl tak, že jsme spojili dva stromy s rankem $k-1$. Proto má každý strom s kořenem ranku k alespoň 2^k prvků. Graf má celkem n vrcholů, takže nejvyšší možný rank je nejvýše $\log n$. To je zároveň horním odhadem časové složitosti operací `UNION` a `FIND`.

1.4 Řešení s kompresí cestiček

Předchozí řešení můžeme vylepšit tak, že při každém zavolání `FIND`(v) a tedy při každém průchodu orientované cesty z v do kořene r_v, přepojíme všechny vrcholy na této cestě přímo do kořene r_v. Následující řešení využívá rekurze, ale můžete ho implementovat i bez ní (v prvním průchodu najdete kořen a ve druhém přesměrujete všechny vrcholy do kořene).

```
FIND(v):
  if v ≠ π(v) then
    π(v) := FIND(π(v))
  return π(v)
```

Tato heuristika zvýší čas operace `FIND` jen malinko a snadno se naprogramuje. Z dlouhodobého hlediska nám tato trocha práce navíc může hodně pomoci. Pokud znova zavoláme `FIND` na stejný vrchol, tak už kořen stromu najdeme na jeden krok. Heuristika s kompresí cestiček se vyplatí, pokud budeme operaci `FIND` volat častěji než `UNION`. To ale v grafových algoritmech nastává skoro vždy. Operaci `UNION` můžeme zavolat nejvýše n krát, protože pak už budou všechny vrcholy v jedné komponentě. Operaci `FIND` typicky voláme pro každou hranu grafu (například v algoritmech pro hledání minimální kostry). A hran může být řádově až n^2.

Dá se ukázat, že v implementaci s kompresí cestiček bude m operací `UNION` nebo `FIND` trvat $\mathcal{O}(m\alpha(n))$, kde $\alpha(n)$ je inverzní Ackermannova funkce[1] (pro prakticky použitelná čísla je to konstanta menší rovná 4, ale jinak pro hodně velká n roste až do nekonečna). Amortizovaný čas obou operací tedy je $\mathcal{O}(\alpha(n))$, což je prakticky konstantní čas $\mathcal{O}(1)$.

Poznámka: Prakticky se dobře chová i následující řešení, které při zavolání `FIND`(v) přepojí do kořene pouze vrchol v. Také se o něm dá ukázat, že časová složitost m operací `UNION` nebo `FIND` trvá $\mathcal{O}(m\alpha(n))$.

```
FIND(v):
  while π(v) ≠ π(π(v)) do
    π(v) := π(π(v))
  return π(v)
```

[1] Inverzní Ackermannova funkce $\alpha(n)$ je inverzní funkce k Ackermannově funkci $A(n)$. Ackermannova funkce je definovaná jako diagonála Ackermannovy hierarchie, tedy $A(n) := A(n,n)$. Funkce $A(n)$ roste mnohem rychleji než libovolný polynom, exponenciála či $n!$. Ackermannova hierarchie se počítá rekurzivně.

$$A(m,n) = \begin{cases} n+1 & \text{pro } m = 0, \\ A(m-1,1) & \text{pro } m > 0 \text{ a } n = 0, \\ A(m-1, A(m,n-1)) & \text{pro } m > 0 \text{ a } n > 0. \end{cases}$$

Funkce v jednotlivých řádcích jsou $A(1,n) = n+2$, $A(2,n) = 2n+3$, $A(3,n) = 2^{n+3} - 3$, $A(4,n) = \underbrace{2^{2^{2^{\cdot^{\cdot^{2}}}}}}_{n+3} - 3$.

Hodnoty $A(n)$ pro $n = 0, 1, 2, \ldots$ jsou 1, 3, 8, 61, $2^{2^{2^{65533}}} - 3$, ...

1.4.1 Upočítání amortizovaného času $\mathcal{O}(\log^* n)$

V předchozích sekcích jsme si dokázali, že časová složitost operace `UNION` nebo `FIND` je v nejhorším případě $\mathcal{O}(\log n)$ a zmínili jsme se, že se dá pro kompresi cestiček ukázat amortizovaný čas jedné operace $\mathcal{O}(\alpha(n))$. Teď si ukážeme malinko slabší, ale prakticky dostačující, odhad amortizované časové složitosti.

Číslo $\log^* n$ je definováno jako počet po sobě aplikovaných operací log takový, abychom z čísla n dostali něco menšího nebo rovno 1. Například $\log^* 1000 = 4$ protože $\log\log\log\log 1000 \leq 1$. Pro všechna prakticky použitelná čísla x je $\log^* x \leq 5$. Aby byl iterovaný logaritmus $\log^* x > 5$, tak bychom potřebovali $x > 2^{65536}$. S tak velkým číslem se ale prakticky nikdy nesetkáte.

Věta 1 *Pokud začneme s prázdnou datovou strukturou obsahující n jednoprvkových komponent a provádíme posloupnost m operací* `UNION` *nebo* `FIND`*, tak je celkový čas provedení všech m operací $\mathcal{O}((m+n)\log^* n)$.*

Operace `UNION`(u,v) najde reprezentanty komponent (kořeny stromů) zavoláním `FIND`(u), `FIND`(v) a pak natáhne šipku mezi kořeny spojovaných komponent. Komprese cestiček na ní nemá žádný vliv. Natažení šipky mezi kořeny zabere jen konstantní čas. Proto při odhadu celkové časové složitosti budeme počítat jen čas operací `FIND` (místo času pro `UNION`(u,v) budeme počítat čas `FIND`(u), `FIND`(v)) a k nim přičteme $\mathcal{O}(\#\text{operací } \texttt{UNION})$.

Rank vrcholů mění pouze operace `UNION`. Rank kořene se ještě může zvýšit, ale jakmile vrchol přestane být kořenem, tak už se jeho rank nemění. Kompresí cestiček se $rank(v)$ nezmění. Na druhou stranu už ho nebudeme moci interpretovat jako délku nejdelší orientované cesty vedoucí do vrcholu v.

Pozorování 1

- *Pro každý vrchol v je $rank(v) < rank(\pi(v))$.*
- *Každý kořen s rankem k má alespoň 2^k potomků.*
- *Pokud máme celkem n vrcholů, tak vrcholů s rankem k je nejvýše $n/2^k$.*

Důkaz: První vlastnost platí, protože vrchol ranku k vznikl povýšením při spojování dvou kořenů ranku $(k-1)$ (a jejich podstromů). Z toho indukcí dokážeme i druhou vlastnost (zkuste to).

Druhou vlastnost můžeme rozšířit i na vrcholy, které už nejsou kořenem. Každý takový vrchol musel být jednou kořenem a tehdy pro něj vlastnost platila. Od té doby se jeho rank, ani jeho potomci nezměnili.

Protože jsou všechny podstromy určené vrcholy ranku k vzájemně disjunktní, a každý má 2^k vrcholů, tak je vrcholů ranku k nejvýše $n/2^k$. ∎

Pracujeme s n vrcholy a proto jejich rank může nabývat pouze hodnot od 0 do $\log n$ (dle předchozího pozorování). Tyto hodnoty si rozdělíme do následujících pečlivě vybraných intervalů (důvod vyplyne později)

$$\{1\},\ \{2\},\ \{3,4\},\ \{5,6,\ldots,16\},\ \{17,18,\ldots,2^{16}=65536\},\ \{65537,\ldots,2^{65536}\},\ldots$$

Každý interval je tvaru $\{k+1,k+2,\ldots,2^k\}$, kde k je mocnina dvojky. Počet těchto skupin je $\log^* n$. Prakticky si vystačíme s prvními pěti intervaly, protože jinak bychom museli mít více než 2^{65536} vrcholů. To se prakticky nikdy nestane.

V posloupnosti operací trvá každá operace `FIND` jinak dlouho. Pro výpočet amortizovaného času operace `FIND` použijeme účetní metodu (viz zavedení amortizované

časové složitosti na straně **??**). Za každý krok práce budeme muset zaplatit jednou korunu.

Každý vrchol dostane na účet nějaké peníze a to tak, aby všechny vrcholy dohromady dostaly nejvýše $\mathcal{O}(n \log^* n)$ Kč. Každá operace `FIND`(v) dostane $\mathcal{O}(\log^* n)$ Kč. Z těchto peněz musí zaplatit všechnu svoji práci (průchod z v do kořene a kontrakce této cesty). Pokud by jí peníze nestačili, tak si může na platbu půjčit od vrcholů, které prochází. Celkový čas m operací `FIND` bude nejvýše počet peněz a to je $\mathcal{O}(m \log^* n) + \mathcal{O}(n \log^* n)$.

Nyní zbývá ukázat: (a) jak rozdělit $\mathcal{O}(n \log^* n)$ Kč mezi vrcholy a (b) jak budou probíhat platby za prováděnou práci, abychom na žádném účtu neklesly do mínusu.

Nejprve k rozdělení peněz. Každý vrchol dostane peníze v momentě, kdy přestane být kořenem. Od této chvíle už se jeho rank nemění. Pokud jeho rank leží v intervalu $\{k+1, k+2, \ldots, 2^k\}$, tak dostane 2^k Kč. Z pozorování 1 víme, že počet vrcholů s rankem větším než k je nejvýše

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \frac{n}{2^{k+3}} + \cdots \leq \frac{n}{2^k}$$

Proto vrcholům s rankem v jednom konkrétním intervalu zaplatíme nejvýše n korun. Různých intervalů je $\log^* n$ a tudíž celkem vrcholům rozdělíme $\leq n \log^* n$ Kč.

Teď se podíváme na to, jak budou probíhat platby. Čas jedné operace `FIND`(v) je počet skoků po šipkách z vrcholu v do kořene, v každém vrcholu na cestě musíme vykonat jeden krok práce. Rank vrcholů na této cestě roste. Každý vrchol x na této cestě padne jedné ze dvou kategorií: buď je $rank(\pi(x))$ ve vyšším intervalu než $rank(x)$, a nebo jsou oba ve stejném.

Vrcholů prvního typu je nejvýše $\mathcal{O}(\log^* n)$ a proto práce v nich zabere nejvýše čas $\mathcal{O}(\log^* n)$. Tuto práci platí operace `FIND`(v) ze svého účtu.

Vrcholy druhého typu musí práci zaplatit ze svého účtu. Za odměnu budou „převěšeni“ a budou si ukazovat na rodiče s vyšším rankem než byl rank jejich původního rodiče.

Pokaždé, když vrchol druhého typu platí ze svého účtu, tak je povýšen. Tedy po nejvýše tolika povýšeních, kolik dostal peněz (to je nejvýše počet různých hodnot v daném intervalu), dosáhne nirvány a bude si ukazovat na rodiče z vyššího intervalu (už nebude muset nikdy platit).

1.5 Přehled všech řešení

reprezentace	`FIND`	`UNION`
pole	$\mathcal{O}(1)$	$\mathcal{O}(n)$
pole (union by size)	$\mathcal{O}(1)$	$\mathcal{O}(n)$ amortiz $\mathcal{O}(\log n)$
stromečky (union by rank)	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
stromečky s kompresí	$\mathcal{O}(\log n)$ amortiz $\mathcal{O}(\alpha(n))$	$\mathcal{O}(\log n)$ amortiz $\mathcal{O}(\alpha(n))$

Co se týká implementace, tak jsou všechna řešení natolik jednoduchá, že můžeme vždy naprogramovat nejlepší řešení pomocí stromečků s kompresí cestiček. Dostaneme tak "téměř" konstantní časovou složitost každé operace.

Upočítání odhadů časové složitosti $\mathcal{O}(\alpha(n))$ pochází od Tarjana [**?**].

1.6 Příklady

1. (Tranzitivní uzávěr grafu) Dostanete graf $G = (V, E)$. Hrana uv vyjadřuje, že u a v jsou v relaci R (graf je v podstatě jen zápis relace R obrázkem). Relace R je určitě symetrická, protože máme neorientovaný graf. Relace R ale nemusí být tranzitivní (pokud aRb a bRc tak potom i aRc). Chtěli bychom ji rozšířit na relaci, která už tranzitivní je.

   Tranzitivní uzávěr grafu je nejmenší nadgraf G (doplnění hran), který už je tranzitivní. Jinými slovy, *tranzitivní uzávěr grafu* G je graf $G_T = (V, E_T)$ s původními vrcholy a vw je hrana tranzitivního uzávěru právě tehdy když v G existuje cesta mezi v a w.

   (a) Navrhněte efektivní algoritmus, který najde tranzitivní uzávěr neorientovaného grafu G.

   (b) Tranzitivní uzávěr můžeme definovat i pro orientované grafy (tj. pro relace, které nemusí být symetrické). *Tranzitivní uzávěr orientovaného grafu* G je orientovaný graf $G_T = (V, E_T)$ s původními vrcholy a vw je hrana tranzitivního uzávěru právě tehdy když v G existuje orientovaná cesta z v do w. Navrhněte efektivní algoritmus, který najde tranzitivní uzávěr orientovaného grafu G.