Korespondenční Seminář z Programování

38. ročník | **KSP** | **Červen 2026**

Vzorová řešení páté série třicátého osmého ročníku KSP

38-5-1 Rozmanitý mrak

Pojďme si nejdřív přeložit úlohu do řeči algoritmů prohledávajících text. Úloha nám dává množinu jehel a seno a ptá se, zda je každý znak sena pokryt nějakým výskytem jehly.

Hledání více jehel v textu nám nějak zavání algoritmem *Aho-Corasickové*. Ten umí pro každou pozici v seně nahlásit všechny výskyty jehel, které na dané pozici končí. To zní použitelně! Můžeme si pro každý znak sena spočítat bit, jestli je pokrytý nějakou jehlou. Začneme se všemi bity nulovými a pak seno projdeme algoritmem Aho-Corasickové a za každý výskyt nějaké jehly označíme všechny použité znaky za pokryté.

To by sice fungovalo, ale může to být pekelně pomalé, protože každé políčko můžeme označit hodněkrát. Pojďme se zamyslet nad tím, co děláme navíc.

Zrychlujeme

Zaprvé vyhodnocujeme na každém místě konec *všech* nalezených výskytů jehel, přičemž by nám stačil pouze výskyt té nejdelší. Každá kratší jehla končící na stejném místě by pokryla již pokryté znaky. To vyřešíme drobnou modifikací algoritmu Aho-Corasickové popsanou v kuchařce[1] – při nahlašování výskytů jehel nahlásíme jenom první výskyt a nepokračujeme po zkratkových hranách dál. Tím vyhodnocujeme vždy pouze nejdelší nalezenou jehlu. Pozor, to neznamená, že se zkratkových hran můžeme zbavit: na nahlášení nejdelší nalezené jehly stále potřebujeme zkratkovou hranu z aktuálního stavu (který nemusí odpovídat jehle) do nejdelšího suffixu, který je jehlou.

Zadruhé i tak teď pro každé políčko označujeme d_i políček jako pokrytých, kde d_i je délka nejdelšího výskytu končícího na tomto políčku. Tím trávíme $\Theta(\sum_{i=1}^{D} d_i)$ času, což ale může být hodně, pokud se výskyty hodně překrývají. Představme si seno jako řetězec D áček a k němu jehlu jako řetězec $D/2$ áček. Pak na $D/2$-tém, $(D/2+1)$-tém, …, D-tém políčku najdeme vždy výskyt délky $D/2$ a dohromady označováním strávíme $\Theta(D^2)$ času. Proto náš algoritmus upravíme, aby nejprve pro každé políčko spočítal d_i pomocí algoritmu Aho-Corasickové. Potom si dopřejeme druhý průchod odzadu a zkontrolujeme, že je každý znak pokrytý. Když při průchodu zpět dojdeme na znak, kde končí nějaký výskyt jehly, zapamatujeme si kolik znaků dopředu pokrývá a tuto hodnotu postupně upravujeme.

Tenhle trik však potřebuje lineární paměť s délkou sena, pojďme se vyvarovat i toho. Uvědomme si, že si délku nalezené jehly nemusíme pamatovat pro každý znak, místo toho to zvládneme *online*. Stačí si průběžně pamatovat pozici nejlevějšího nepokrytého znaku. Pokud totiž v nějaké chvíli pokryjeme ten, pokryjeme i všechny znaky od něj napravo až po aktuální pozici. Pokud naopak daný znak za celou dobu nepokryjeme, víme že jehly nepokrývají celé seno.

Složitost

Náš algoritmus nejprve sestrojí automat Aho-Corasickové, v čase i paměti v $\mathcal{O}(\sum_{i=1}^{N} |Z_i|)$. Potom projde celý mrak a na každém znaku aktualizuje stav automatu a případně přehodnotí nejlevější nepokrytý znak při výskytu jehly. Každá z operací na každém znaku sena zabere konstantní čas, což se nasčítá na čas v $\mathcal{O}(D)$ a konstantní prostor. Celkově jsme se tedy dostali na čas v $\mathcal{O}(D + \sum_{i=1}^{N} |Z_i|)$ a paměť v $\mathcal{O}(\sum_{i=1}^{N} |Z_i|)$.

Úlohu připravil: Matúš Púll

38-5-2 Garážový výprodej

Náš první úkol je pochopit algoritmus ze zadání. Úloha prvního cyklu volání „DFS“ v G^T je seřadit vrcholy do takového pořadí $v_1, \ldots, v_n$, že jakmile $i < j$, v G smí vést cesta z v_j do v_i jedině pokud v_i a v_j sdílí komponentu silné souvislosti.

Tohoto pořadí využívá druhý cyklus volání „DFS“ (kam bychom mohli klidně dát i BFS nebo BFS se zásobníkem), kde prohledávání voláme v pořadí od v_n sestupně po v_1. Díky prvnímu odstavci máme zaručeno, že dosud neviděné vrcholy dosažitelné z aktuálního v_i jsou právě vrcholy sdílející komponentu s v_i.

Teď se jen nesmíme nechat zmást tím, kdy algoritmus prohledává v G^T, kdy v G, a jaký je přesně vliv pořadí vrcholů, které algoritmu podsuneme.

Abychom vypsali pořadí vrcholů, na kterém zlevněný algoritmus zaručeně funguje, vyřešíme první cyklus volání „DFS“ za něj. Jinými slovy, použijeme korektní algoritmus (s normálním DFS), abychom našli výše zmíněné pořadí $v_1, \ldots, v_n$, a přímo toto pořadí můžeme vypsat. První cyklus chybného algoritmu začne prohledávat z v_1 v G^T, kde nalezne jen vrcholy v komponentě s v_1 (a vloží je do zásobníku, takže budou druhým průchodem v G prohledány nakonec).

Abychom zaručili, že algoritmus selže, musíme v G najít hranu uv vedoucí z komponenty A do komponenty B, kde B má alespoň dva vrcholy. Pak můžeme chybné pořadí začít vrcholem v a všemi vrcholy komponenty A, a až po nich doplnit vrcholy komponenty B. Tím první cyklus volání BFS se zásobníkem v G^T z vrcholu v přejde rovnou do u, a z něj prohledá celou komponentu A, než se vrátí do komponenty B. Tím se stane, že druhý cyklus „DFS“ka bude zpracovávat komponentu A dříve, než komponentu B, takže z A přejde do B a chybně prohlásí A i B za součásti stejné komponenty silné souvislosti (a můžete taky promyslet, že pokud hrana uv s popsanou vlastností v G není, výprodejový algoritmus na G nikdy nemůže selhat).

Program (Python):
`http://ksp.mff.cuni.cz/viz/38-5-2.py`

Úlohu připravili: Daniel Culliver, Martin Koreček

[1] `http://ksp.mff.cuni.cz/viz/kucharky/hledani-v-textu`

38-5-3 Nespolehlivé autobusy

Máme za úkol najít (a poté vykonat) strategii, která Kevina dostane domů s nejvyšší možnou pravděpodobností. Nejprve si rozmyslíme, jak takovou strategii vůbec popsat.

Máme slíbené, že příjezdy autobusů jsou nezávislé jevy, takže to, jaké autobusy pojedou v budoucnosti, není ovlivněno tím, které autobusy jely v minulosti. Pokud tedy Kevin stojí v čase c na zastávce z, tak bude nejlepší strategie stejná, ať už se do této situace dostal jakkoliv. Speciálně pokud uvidí autobus, tak se může rozhodnout, jestli do něj nastoupí, jen podle čísla spoje. Stačí tedy předem spoje roztřídit na *užitečné* a *neužitečné*, podle toho, jestli by jimi Kevin měl jet, kdyby měl příležitost.

Úloha po nás chce, aby naše strategie měla co nejlepší pravděpodobnost úspěchu. Rozmyslíme si tedy nejprve, jak pro danou strategii tuto pravděpodobnost spočítat.

K tomu použijeme dynamické programování.[2] Jako $p_{c,z}$ si označíme pravděpodobnost toho, že se Kevin dostane do půlnoci domů, pokud v čase c bude na zastávce z. Nejprve vyřešíme okrajové podmínky. Jistě bude $p_{c,\text{Kevinův dům}} = 1$, a pro ostatní zastávky $p_{\text{půlnoc},z} = 0$. Dále pokud ze zastávky z v čase c neodjíždí žádný autobus, nebo odjíždí neužitečný autobus, tak bude $p_{c,z} = p_{c+1,z}$. Zbývá ten nejzajímavější případ: Ze zastávky z v čase c s pravděpodobností p pojede užitečný autobus, který má dorazit na zastávku z' v čase c'. Tady použijeme větu o celkové pravděpodobnosti: Pokud autobus pojede, dorazí Kevin domů s pravděpodobností $p_{c',z'}$, jinak s pravděpodobností $p_{c+1,z}$. Celkem tedy $p_{c,z} = p \cdot p_{c',z'} + (1-p) \cdot p_{c+1,z}$. Takto můžeme spočítat úspěšnost strategie s časovou a paměťovou složitostí $\mathcal{O}(\text{počet zastávek} \cdot \text{počet časů})$.

Pomocí dynamického programování můžeme optimální strategii také spočítat. Pokaždé, když budeme počítat $p_{c,z}$ pro čas odjezdu nějakého spoje, tak vyzkoušíme obě varianty – jak pokud je spoj užitečný, tak pokud je neužitečný. Vybereme si přirozeně tu nejlepší. Drobnou úpravou získáme, že užitečné jsou právě ty autobusy, kde $p_{c',z'} > p_{c+1,z}$, což dává smysl – chceme jet jedině spojem, který nás doveze do lepší situace, než v jaké jsme teď.

Máme tedy řešení v $\mathcal{O}(\text{počet zastávek} \cdot \text{počet časů})$, ale jde to i rychleji. Všimneme si, že se hodnoty p mění pouze v časech, kdy ze zastávky nějaký autobus odjíždí. Budeme tedy $p_{c,z}$ počítat pouze pro ně a pro půlnoc. Spočítané hodnoty p si pro každou zastávku uložíme v poli seřazeném dle času, a kdykoliv nás bude zajímat hodnota $p_{c,z}$ pro nějaký konkrétní čas, tak najdeme hodnotu pro nejbližší následující čas pomocí binárního vyhledávání. Spočítaných hodnot p bude nejvýše počet spojů + počet zastávek a užitečných zastávek je nejvýše tolik, co spojů. Pro výpočet každé hodnoty p potřebujeme znát nejvýše dvě jiné hodnoty p, a tedy provedeme nejvýše dvě binární vyhledávání. Celková složitost dynamického programování tedy bude $\mathcal{O}(N \log N)$, kde N je počet spojů. Ještě musíme předem spoje seřadit sestupně dle času odjezdu, což zabere také $\mathcal{O}(N \log N)$ času. Paměťová složitost bude $\mathcal{O}(N)$.

Zbývá už jen jeden detail: Zadání po nás nechtělo jen strategii spočítat, ale i pomoci Kevinovi ji vykonat. Z dynamického programování dostaneme seznam užitečných spojů, roztřídíme si je dle zastávek a v rámci každé je seřadíme dle času odjezdu. Kdykoliv bude Kevin potřebovat poradit, tak stačí najít nejbližší užitečný spoj odjíždějící z jeho aktuální zastávky, který můžeme najít pomocí jednoho binárního vyhledávání v čase $\mathcal{O}(\log N)$. Kevin bude jistě potřebovat poradit nejvýše $N+1$-krát, časová složitost celého algoritmu tedy bude $\mathcal{O}(N \log N)$.

Úlohu připravil: Ben Swart

38-5-4 Linuxáci a Windowsáci

Jak bylo již v zadání řečeno, je nám zadán graf na N vrcholech, kde každý vrchol reprezentuje jednoho účastníka. Tyto vrcholy jsou pak propojeny M hranami, které reprezentují kamarádské vztahy mezi jednotlivými účastníky. Označme $\mathcal{L}$ množinu vrcholů hrochů s Linuxem a $\mathcal{W}$ množinu vrcholů hrochů používajících Windows. Naším cílem je teď rozdělit zbývající vrcholy grafu do těchto množin $\mathcal{L}$ a $\mathcal{W}$ tak, abychom minimalizovali počet hran vedoucích mezi vrcholy v různých množinách. To už začíná vypadat trochu povědomě – jde přímo o definici minimálního řezu v grafu za podmínky, že nějaké předem dané vrcholy musí být v jedné nebo druhé množině vrcholů.

Předpokládejme, že máme v zadaném grafu alespoň jednoho Linuxáka a alespoň jednoho Windowsáka. V opačném případě bychom mohli ihned přiřadit všem účastníkům stejný operační systém a nikdo by si nestěžoval. K hledání minimálního řezu v tomto grafu využijeme toky v sítích – o nich si můžete přečíst něco víc v naší tokové kuchařce.[3]

Pokud vám, milí řešitelé, zatím není jasné, co mají toky společného s minimálními řezy, nezoufejte. V tomto řešení si nejdříve ukážeme, jak vůbec nějaký vhodný tok pro naši úlohu vyrobit, a následně ukážeme, že z něj dovedeme vyrobit přesně to rozdělení vrcholů na dvě části, které chceme.

Konstrukce sítě

Jak jste se mohli v kuchařce dočíst, toky umíme pouštět pouze po sítích s orientovanými hranami. Náš zadaný graf má ale všechny hrany neorientované. Proto v naší síti vytvoříme pro každou neorientovanou hranu $\{u,v\} \in E(G)$ dvě orientované hrany uv a vu. Dále je v síti potřeba mít dva speciální vrcholy – zdroj z a stok s. My si je tak do našeho grafu přidáme jako dva nové vrcholy. Následně musíme tyto dva vrcholy vhodně propojit se zbytkem sítě. Pro každý linuxácký vrchol l tak do sítě přidáme hranu zl, a pro každý windowsácký vrchol w přidáme hranu ws.

Pro tvoření toků nám zbývá jedna poslední ingredience, a to nastavit každé hraně $e \in E(S)$ kapacitu $c(e)$. Všem hranám nastavíme kapacitu na 1, kromě těch nově přidaných hran ze zdroje a stoku, kterým nastavíme kapacitu na $+\infty$ (přesnou implementaci nekonečné kapacity si ukážeme později).

Pomocí libovolného tokového algoritmu (například pomocí Fordova-Fulkersonova algoritmu z kuchařky) pak nalezneme v S maximální tok f. Pomocí tohoto toku bychom nyní chtěli v S nalézt podmnožinu hran $C \subseteq E(S)$ takovou, že když všechny tyto hrany z S odstraníme, pak nebude exis-

[2] http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani

[3] http://ksp.mff.cuni.cz/viz/kucharky/toky-v-sitich

tovat žádná cesta ze zdroje z do stoku s a navíc součet kapacit všech hran z C je minimální. Této množině C říkáme minimální z-s řez.

Hledání minimálního z-s řezu

Abychom mohli správně popsat řez, musíme se podívat na *rezervy hran*: pro každou hranu $uv \in E(S)$ definujeme její rezervu jako $r(uv) := c(uv) - f(uv) + f(vu)$. Z definice toku platí, že $r(uv) \geq 0$ pro každou hranu uv. Místo kapacity nás tak nyní u každé hrany $uv \in E(S)$ zajímá, kolik toku podél hrany uv můžeme ještě poslat, abychom nepřekročili její kapacitu, a kolik toku z opačné hrany vu můžeme odebrat, aniž by její tok klesl pod nulu. Hranám s rezervou $r(uv) = 0$ budeme říkat *nasycené*, analogicky zbytku hran s $r(uv) > 0$ říkáme *nenasycené*. Pokud je náš nalezený tok f maximální, pak bude v síti S vždy existovat alespoň jedna nasycená hrana. Pokud by žádná taková hrana neexistovala, pak bychom mohli f o malinko zvětšit a dostat tím větší tok. To je ale ve sporu s tím, že f je maximální.

Označme D množinu všech vrcholů v takových, že vrchol v je dosažitelný ze zdroje pouze po nenasycených hranách. Nyní se podívejme na hrany, které vedou ven z této množiny D. To jsou konkrétně nasycené hrany $uv \in E(S)$ takové, že $u \in D$ a $v \in (V(S) \setminus D)$. My tvrdíme, že tyto hrany tvoří minimální $z - s$ řez C. Formální důkaz je poměrně složitý, tak jej zde neuvádíme v jeho plné kráse. Můžete si jej ale najít v Průvodci.

Pro lepší intuici alespoň naznačíme, že tato množina je z-s řez. Vezměme naši síť S a odstraňme z ní všechny hrany z C. Pokud by ve výsledné síti S' byl vrchol s dosažitelný ze zdroje z, z definice D by to znamenalo, že mezi z a s existuje cesta p, jejíž všechny hrany jsou nenasycené. Tudíž bychom opět mohli o malinko zvětšit tok f podél hran cesty p, aniž bychom překročili kapacity hran, a získat větší tok. To je ale znovu ve sporu s maximalitou toku f.

Přidělení notebooků vrcholům

Nyní tedy máme síť S vzniklou z původního grafu G přidáním vrcholů z a s a máme podmnožinu hran $C \subseteq E(S)$ s minimálním součtem kapacit, kterou když odstraníme, pak nebude existovat cesta mezi z a s. Jak to ale pomůže v rozdělení původních vrcholů grafu G do množin $\mathcal{L}$ a $\mathcal{W}$?

Prvně si uvědomme, že v množině C budou pouze „orientované“ verze původních hran z grafu G. Jelikož jsme nově přidaným hranám propojující vrchol z, s s některými dalšími vrcholy z G přiřadili nekonečné kapacity, pak jistě žádná z těchto hran nebyla vybrána do C. Zadruhé, jelikož jsme z propojili pouze s každým linuxovým vrcholem a s pouze s každým windowsovým vrcholem, pak se odstraněním hran C ze sítě S nejen zbavíme všech cest mezi z a s, ale také všech cest, které propojují některý linuxový a windowsový vrchol. Když dáme tyto dvě informace dohromady, všimneme si, že odstraněním hran C (které nyní zase bereme jako neorientované) z původního grafu G docílíme toho samého výsledku – mezi žádnými dvěma vrcholy s opačnými operačními systémy nevede cesta.

V tomto nově vzniklém grafu G' nyní všem vrcholům bez notebooku, do nichž vede nějaká cesta ze zdroje z, přiřadíme notebook s Linuxem, a zbytku přiřadíme notebook s Windows. Můžeme si všimnout, že v G' nevyvstávají žádné konflikty mezi kamarády – každý vrchol sousedí pouze s vrcholy se stejným operačním systémem. Pokud do G' přidáme zpátky hrany z C, nějaké takové konflikty vyvstanou – konkrétně všechny hrany z C budou vést mezi vrcholy s opačnými operačními systémy. Ale jelikož jsme zvolili C minimální možné, pak i těchto konfliktů bude minimální množství, čehož jsme přesně chtěli docílit.

Implementace a časová složitost

V našem popisu i vzorovém řešení používáme pro hledání maximálního toku v síti Fordův-Fulkersonův algoritmus. Ten najde v síti s celočíselnými (a konečnými) kapacitami maximální tok v čase $\mathcal{O}(M \cdot |f|)$, kde $|f|$ je velikost nalezeného maximálního toku (a náhodou také počet konfliktních hran). V našem algoritmu jsme ale některým hranám nastavili kapacity na $+\infty$, aby nebyly vybrány do výsledného minimálního řezu. Místo toho jim můžeme nastavit kapacitu $M + 1$, což je víc než velikost libovolného hranového řezu.

Pro efektivní rozdělení operačních systémů mezi vrcholy stačí projít síť S do hloubky od vrcholu z s podmínkou, že nikdy neprojdeme přes nasycenou hranu. Všem navštíveným vrcholům přidělíme Linux, zbytku Windows. To nám zabere $\mathcal{O}(M + N)$ času.

Výsledná časová složitost je tak $\mathcal{O}(M \cdot |f| + N)$.

Program (C++):
`http://ksp.mff.cuni.cz/viz/38-5-4.cpp`

Úlohu připravili: Kačka Doubková, Ondra Sedláček, Lukáš Veškrna

38-5-S Hešování řetězců

Úkol 1: Přidání/odebrání znaku

Srovnejme, jak vypadá hešovací funkce pro původní řetězec, jeho prodloužení a jeho zkrácení ($\equiv$ značí rovnost modulo p):

$$\begin{aligned} h_a(x_1, \ldots, x_k) &\equiv x_1 a^{k-1} + x_2 a^{k-2} + \ldots + x_k a^0, \\ h_a(x_1, \ldots, x_{k+1}) &\equiv x_1 a^k + x_2 a^{k-1} + \ldots + x_k a^1 + x_{k+1} a^0, \\ h_a(x_2, \ldots, x_k) &\equiv x_2 a^{k-2} + \ldots + x_k a^0. \end{aligned}$$

Z prvního heše můžeme ten druhý získat vynásobením číslem a a přičtením $x_{k+1} a^0 \equiv x_{k+1}$. Z prvního heše získáme ten třetí odečtením $x_1 a^{k-1}$. Pokud si mocniny čísla a předpočítáme, obě aktualizace heše zvládneme v konstantním čase.

Úkol 2: Složitost Rabina-Karpa

Připomeňme značení ze zadání: J je délka jehly, S délka sena, V počet výskytů a p prvočíslo z hešovací funkce.

Rabinův-Karpův algoritmus nejdříve stráví čas $\mathcal{O}(J)$ výpočtem heše pro počáteční polohu okénka „úplně vlevo“. Pak $(S - J)$-krát okénkem posune. Při každém posunutí přepočítá hešovací funkci, což se podle předchozího úkolu stihne v konstantním čase. Veškerým počítáním hešů tedy stráví čas $\mathcal{O}(S)$.

Algoritmus každý heš okénka porovná s hešem jehly a pokud nastane shoda, znak po znaku zkontroluje, jestli okénko opravdu odpovídá jehle. Kontrola trvá čas $\mathcal{O}(J)$. Shoda přitom může znamenat dvě věci: buďto opravdový výskyt (to nastane celkem V-krát), nebo kolizi hešovací funkce. Jelikož polynomiální hešování pro řetězce délky J je J-univerzální, kolize nastane s pravděpodobností nejvýše J/p. Z toho plyne průměrný počet kolizí nejvýše JS/p. Celkově tedy porovnáváním okénka s jehlou strávíme v průměru čas $\mathcal{O}(J(V + JS/p)) = \mathcal{O}(JV + J^2 S/p)$.

Průměrná složitost celého algoritmu tedy činí $\mathcal{O}(S + JV + J^2S/p)$. Pokud zvolíme prvočíslo p větší než J^2, složitost se vejde do $\mathcal{O}(S + JV)$. Takový algoritmus bude rychlý v případech, kdy je výskytů málo.

Úkol 3: Izomorfismus – hešovací funkce

Kódy budeme volit z tělesa $\mathbb{Z}_p$ pro vhodné prvočíslo $p > n$. Požadovaná hešovací funkce $h(k_1, k_2)$ tedy bude hešovat pár prvků $(k_1, k_2) \in \mathbb{Z}_p^2$ na prvek $\mathbb{Z}_p$.

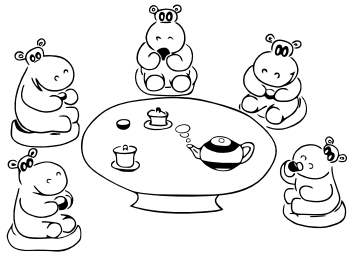
Opět se nám bude hodit polynomiální hešování. Vytvoříme systém funkcí $h_a(k_1, k_2) \equiv k_1 a + k_2$ (počítáme opět modulo p, parametrizujeme prvkem $a \in \mathbb{Z}_p$). V zadání jsme dokázali, že tento systém je 2-univerzální. To znamená, že libovolné dva páry (k_1, k_2) a (k_1', k_2') zkolidují s pravděpodobností maximálně $2/p$.

Teď počítejme pravděpodobnost selhání algoritmu pro nějaké dva stromy na n vrcholech. Algoritmus nejvýše $(2n)$-krát hešuje páry kódů. Selže, pokud pro nějaké dva různé páry kódů vyjde stejný heš.

Pravděpodobnost selhání se dá spočítat dvojím způsobem. Buď můžeme využít toho, že pro libovolné jevy A a B platí $P(A \cup B) \leq P(A) + P(B)$. Proto pravděpodobnost, že *nějaká* dvojice párů zkoliduje, můžeme shora omezit součtem pravděpodobností kolizí přes všechny dvojice párů. Takových dvojic je $\binom{2n}{2} = 2n(2n-1)/2 \leq 2n^2$. Každá zkoliduje s pravděpodobností nejvýše $2/p$. Pravděpodobnost selhání tedy nepřekročí $2n^2 \cdot 2/p = 4n^2/p$.

Druhá možnost je využít Markovovu nerovnost. Pro každou dvojici párů bychom si pořídili indikátor toho, že dvojice zkolidovala. Počet kolizí bychom vyjádřili jako součet indikátorů. Z linearity střední hodnoty bychom odvodili střední hodnotu počtu kolizí. A pak bychom Markovovou nerovností odhadli pravděpodobnost, že počet kolizí překročí $1/2$. I tímto způsobem dojdeme k odhadu cn^2/p pro nějakou konstantu c.

Chceme-li pravděpodobnost kolize stlačit pod dané $\varepsilon > 0$, postačí zvolit $p > cn^2/\varepsilon$.



Úkol 4: Izomorfismus – kontrola správnosti

Budeme předpokládat, že jsme si zapamatovali nejen heše kořenů, ale i všech vnitřních vrcholů.

Mějme stromy S a T, kterým vyšel stejný heš kořene. Musí tedy být $h(s_1, s_2) = h(t_1, t_2)$, kde s_1 a s_2 jsou kódy dětí kořene stromu S a t_1 a t_2 kódy dětí kořene T. Navíc jsme děti prohodili, aby platilo $s_1 \leq s_2$ a $t_1 \leq t_2$.

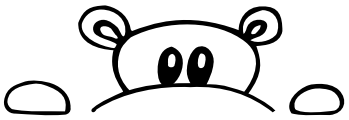
Kdyby se páry (s_1, s_2) a (t_1, t_2) nerovnaly, znamenalo by to, že jsme našli kolizi hešovací funkce. Tím pádem by stromy určitě nebyly izomorfní.

Nebo je $s_1 = t_1$ a $s_2 = t_2$. Pak by měl být levý podstrom stromu S izomorfní levému podstromu stromu T a analogicky pro pravé podstromy. Rekurzivně se tedy zavoláme na oba levé podstromy a pak na oba pravé, abychom ověřili, jestli izomorfismus opravdu nastal, nebo zda je rovnost kódů podstromů způsobena kolizí někde hlouběji.

Dodejme, že v případě kolize se nám může stát, že se rekurzivně zavoláme na porovnání prázdného stromu s neprázdným. V takovém případě okamžitě vyhlásíme, že stromy izomorfní nejsou.

Zbývá rozmyslet časovou složitost. Nechť mají stromy S i T oba po n vrcholech. Během rekurze potkáme každý podstrom stromu S nejvýše jednou a strávíme jím konstantní čas. Podstrom je přitom jednoznačně určen svým kořenem, takže podstromů je právě n. Algoritmus tedy běží v čase $\mathcal{O}(n)$.

Jaký vliv na složitost bude mít opakování algoritmu v případě kolize? Nastavme p tak, aby pravděpodobnost kolize byla menší než $1/2$. Podle lemmatu o džbánu bude tedy průměrný počet spuštění, než uspějeme, menší nebo roven 2. Každý pokus nás stojí $\mathcal{O}(n)$ na výpočet hešů a $\mathcal{O}(n)$ na ověření izomorfismu. Průměrná složitost celého algoritmu tedy činí $\mathcal{O}(2 \cdot n) = \mathcal{O}(n)$.



Úkol 5: Izomorfismus – více dětí

Uvažme na chvíli stromy, v nichž každý vnitřní vrchol má právě d dětí. Mohli bychom opět použít polynomiální hešování, tentokrát na d-tice. Jenže by nás příliš brzdilo, že každou d-tici musíme před zahešováním uspořádat. Jak se tomu vyhnout?

Pořídíme si *komutativní* hešovací funkci, tedy takovou, jejíž výsledek se nezmění, když libovolně proházíme vstupy. Nabízí se vytvořit ji kombinací úplně náhodné hešovací funkce (slíbené v zadání) s operací XOR. Kódy budou prvky množiny B všech b-bitových řetězců (tedy čísel od 0 do $2^b - 1$). Pořídíme si náhodnou funkci f z B do B (ta nám bude kódy „náhodně promíchávat“). A d-tici kódů zahešujeme předpisem $h(x_1, \ldots, x_d) = f(x_1 \oplus \ldots \oplus x_d)$. Přitom počet dětí d se může mezi vrcholy stromu lišit.

To vypadá funkčně … do okamžiku, kdy si uvědomíme, že nějaké dva podstromy mohou být izomorfní (tedy $k_i = k_j$ pro nějaké i a j). Pak bude $k_i \oplus k_j = 0$, takže kód vyjde stejně, jako kdyby tam tyto dva podstromy vůbec nebyly.

Problém se ale dá docela snadno napravit: místo v B budeme počítat v $\mathbb{Z}_p$ pro dost velké p a místo XORování budeme sčítat modulo p. Pořídíme si tedy náhodnou funkci f ze $\mathbb{Z}_p$ do $\mathbb{Z}_p$ a hešovací funkci definujeme předpisem $h(x_1, \ldots, x_d) \equiv f(x_1 + \ldots + x_d)$.

Izomorfní stromy určitě dostanou stejné kódy. Potřebujeme ale ukázat, že je málo pravděpodobné, že by neizomorfním stromům kvůli kolizím vyšel stejný kód.

Nechť testujeme izomorfismus dvou stromů na n vrcholech. Uvážíme všech jejich $2n$ podstromů. Sloučíme ty, které jsou izomorfní, a zůstanou nám neizomorfní podstromy $T_1, \ldots, T_m$ pro $m \leq 2n$. Očíslujeme si je v pořadí, ve které přidělujeme kódy, což znamená od nejmělčího podstromu až po ten nejhlubší. Přidělené kódy označíme $k_1, \ldots, k_m$, přičemž $k_i = f(c_i)$ pro nějaká čísla $c_1, \ldots, c_m$.

Tvrzení: Pro každé $t \leq m$ platí, že s pravděpodobností aspoň $p_t > 0$ (stanovíme později) jsou čísla $c_1, \ldots, c_t$ navzájem různá, příslušná k_i navzájem různá a navíc všechna k_i tvoří rovnoměrně náhodně vybranou t-tici různých prvků ze $\mathbb{Z}_p$.

Důkaz: Provedeme indukcí podle t. Pro $t = 1$ uvažujeme pouze podstrom T_1, což musí být jednovrcholový podstrom. Tím pádem $c_1 = 0$, $h_1 = f(0)$ je rovnoměrně náhodný prvek $\mathbb{Z}_p$, takže vyjde $p_1 = 1$.

Nyní provedeme indukční krok. Už jsme zakódovali podstromy $T_1, \ldots, T_{t-1}$ a právě kódujeme T_t. Budeme předpokládat, že zatím nedošlo k žádné kolizi ani mezi c_i, ani mezi k_i. To nastalo s pravděpodobností aspoň p_{t-1}.

Kořen stromu T_t má nějaké podstromy, které jsou mělčí, takže už dostaly kód. Některé z nich mohou být izomorfní, takže je opět sloučíme. Zbudou neizomorfní stromy T_{i_1} až T_{i_s}, přičemž T_{i_j} se vyskytl a_j-krát. Tyto stromy už dostaly nějaké kódy $k_{i_1}, \ldots, k_{i_s}$. Stromu T_t tedy přiřadíme kód $k_t = h(c_t)$, kde $c_t \equiv a_{i_1}k_{i_1} + \ldots + a_{i_s}k_{i_s}$.

Jaká je pravděpodobnost, že jsme tím vytvořili kolizi? Kolize může vzniknout dvěma způsoby: buďto se nové c_t rovná některému z předchozích c_i (a tím pádem je jistě i $k_t = k_i$), nebo kolize vznikne až zahešováním funkcí f.

První možnost: podle indukčního předpokladu je k_{i_1} rovnoměrně náhodný prvek $\mathbb{Z}_p$. Člen $a_{i_1}k_{i_1}$ je také v $\mathbb{Z}_p$ rovnoměrně náhodný, protože násobení náhodného prvku tělesa pevným prvkem dá zase náhodný prvek. A pokud ho přičteme k čemukoliv, dostaneme zase rovnoměrně náhodný prvek. Navíc bude nezávislý na všech c_i (protože funkce f si „přináší svou vlastní náhodnost“). Takže do některého z předchozích c_i se strefíme s pravděpodobností $(i-1)/p$.

Druhá možnost: jelikož c_t je různé od všech předchozích c_i, bude $k_t = f(c_t)$ rovnoměrné náhodné a nezávislé na všech předchozích k_i. Tím pádem se do některého z předchozích k_i strefíme s pravděpodobností opět $(i-1)/p$.

Pravděpodobnost, že kolize nevznikla ani v tomto kroku, ani v žádném z předchozích, tedy bude rovna

$$p_t = (1 - (i-1)/p)^2 \cdot p_{t-1}.$$

Důsledky: Spokojeni s důkazem tvrzení, odhadneme ještě pravděpodobnost úspěchu p_m. „Rozbalením“ rekurentní formule z důkazu dostaneme:

$$p_m = \left(1 - \frac{0}{p}\right)^2 \cdot \left(1 - \frac{1}{p}\right)^2 \cdot \ldots \cdot \left(1 - \frac{m-1}{p}\right)^2$$

To odhadneme zdola:

$$p_m \geq \left(1 - \frac{m}{p}\right)^{2m}$$

Teď si vzpomeneme na důkaz narozeninového paradoxu a odhad $1 - \varepsilon \approx \mathrm{e}^{-\varepsilon}$. Proto:

$$p_m \approx \mathrm{e}^{-\frac{m}{p}\cdot 2m} = \mathrm{e}^{-\frac{2m^2}{p}}$$

Tento výraz jsme volbou prvočísla $p \gg 2m^2$ schopni přitlačit libovolně blízko k 1.

Stále tedy funguje myšlenka drze hešovat, a teprve dodatečně zkontrolovat, zda rovnost kódů opravdu znamená izomorfismus, a pokud ne, algoritmus spustit znovu. Průměrná časová složitost se nezměnila, jen je potřeba v implementaci úkolu 2 domyslet, jak párovat děti obou kořenů, abychom nemuseli třídit. Na to stačí použít hešovaci tabulku.

Úkol 6: Mezigalaktická kandidátka

Především si všimneme, že za každý znak vstupu se vektor frekvencí změní jenom v jedné ze svých k složek (kde k je počet barev). Tato složka přispívá v kódu na jedné pozici kladně, na jiné možná záporně. Proto se kód změní v nejvýše dvou složkách. Chceme tedy hešovací funkci, kterou půjde při změně na jedné pozici v konstantním čase přepočítat. To opět splňuje polynomiální hešování, pokud si předpočítáme tabulku mocnin $a^i \bmod p$.

Do jak velkého prostoru potřebujeme hešovat, aby nám moc nevadily kolize? Připomeneme si úvahy o perfektním hešování ze 4. dílu seriálu a fakt, že polynomiální hešování k-složkových vektorů je k-univerzální. Z toho plyne, že zvolíme-li $p > 2kn^2$, bude hešovací funkce s pravděpodobností aspoň $1/2$ perfektní.

Projdeme tedy vstup, průběžně budeme udržovat vektor četností, jeho kód a heš kódu. Budeme si zaznamenávat všechny heše. To stihneme v čase $\mathcal{O}(n)$.

Pak přihrádkovým tříděním (viz řešení 3. úkolu ze 4. dílu) v čase $\mathcal{O}(n)$ setřídíme všechny heše a pro každou hodnotu heše najdeme jeho první a poslední výskyt. Podle toho identifikujeme nejdelší úsek, který má na začátku i na konci stejný heš. To je buďto opravdový nejdelší jednobarevný úsek, nebo nějaký vícebarevný, který se tam vloudil zrádnou kolizí.

Naštěstí umíme v čase $\mathcal{O}(n)$ pomocí vektorů četností snadno ověřit, že nalezený úsek je jednobarevný. A pokud není, spustíme celý algoritmus znovu. To nastane s pravděpodobností nejvýše $1/2$, takže za průměrně dva pokusy algoritmus úspěšně doběhne. Dosahuje tedy průměrně lineární časové složitosti.

Závěrem

Tím náš seriál o náhodě v algoritmech pravděpodobně dospěl ke konci. Ukázali jsme si několik snadných aplikací pravděpodobnosti, naučili se uvažovat o hešování a objevili, že pravděpodobnostní techniky často pomohou k sestrojení překvapivě jednoduchého algoritmu. Za to ovšem zaplatíme tím, že algoritmus běží rychle jenom v průměru, nebo někdy dokonce nemusí vydat správný výsledek.

Aplikací pravděpodobnosti v informatice je zajisté mnohem víc, ale to už je vyprávění na někdy jindy. Na viděnou se těší autoři seriálu

Martin „Medvěd“ Mareš, Dan Skýpala a Katia Kočická

matfyz

KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.
Realizace projektu byla podpořena Ministerstvem školství, mládeže a tělovýchovy.

Webové stránky: *https://ksp.mff.cuni.cz/*
E-mail: *ksp@mff.cuni.cz*
Organizátoři a kontakty: *https://ksp.mff.cuni.cz/kontakty/*