Korespondenční Seminář z Programování

30. ročník **KSP** **Duben 2018**

Milí řešitelé a řešitelky!

Jaro už je v plném proudu: ptáčci zpívají, potůčky a počítače hučí, občas i trochu zasněží. A my vám přinášíme další sérii znamenitých úložek, křupavých jak houstičky čerstvě vytažené z pece (v tomto případě zapékací jednotky naší laserové tiskárny).

Na seriál jsme nezapomněli, jen nabral zpoždění, takže jsme se rozhodli vydávat seriálové úlohy zvlášť, s posunutým termínem odevzdání.

Ještě připomeneme, že každému řešiteli, který získá v tomto ročníku z každé série alespoň 5 bodů, pošleme KSP propisku, blok, placku a třeba i něco navíc.

Díky řešení KSP se také můžete vyhnout přijímacím zkouškám na MFF UK! Stačí, když získáte alespoň polovinu bodů z ročníku (tedy 150 bodů) a my vám vystavíme osvědčení úspěsného řešitele, díky kterému vás přijmou na MFF bez zkoušek. Pozor: pokud se na Matfyz hlásíte letos a chcete stihnout osvědčení za tento ročník, musíte mít 150 bodů již po této sérii. Navíc je třeba odevzdat řešení do **18. dubna** a ozvat se nám mailem.

Termín série: **14. 5. 2018 v 8:00**

Odevzdávání: Přes web na adrese *https://ksp.mff.cuni.cz/submit/*

Odměna série: **Sladkou odměnu** si vysloužíš každý, kdo odevzdá **alespoň tři** úložky **alespoň týden** před termínem a získá za ně **kladný** počet bodů.

Čtvrtá série třicátého ročníku KSP

30-4-1 Černobílé hádání — 9 bodů

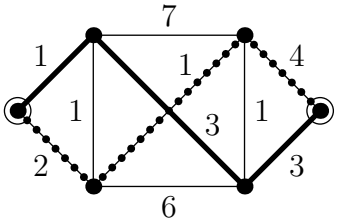
Náš kamarád má pole o n černých a bílých políčkách. Je ale poněkud stydlivý, takže nám pole nechce ukázat. Po krátkém přemlouvání nám prozradil aspoň hodnotu n a také to, že v poli je přesně m souvislých úseků stejné barvy (například v poli `###..##` jsou tři), přičemž m je řádově menší než n. Kamarád je navíc ochoten odpovídat na dotazy „Je v úseku $[a, b]$ aspoň jedno políčko bílé/černé?“ Na co nejméně dotazů zjistěte, jak vypadá celé pole.

30-4-2 Kočka na stromě — 10 bodů

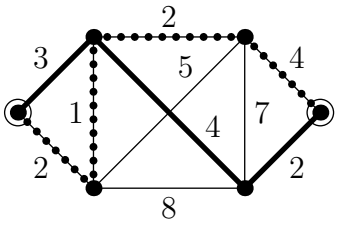
Kočka vylezla na strom a neumí slézt. Hasiči ji jedou zachránit. Potřebují dorazit co nejrychleji, takže se vydají po nejkratší cestě. Mohlo by se ale stát, že tato cesta nebude průjezdná. Chtějí tedy vyslat ještě jedno záložní auto po jiné nejkratší cestě, která nebude s trasou prvního auta mít společné nic kromě začátku a konce.

Poněkud matematičtěji řečeno, máte zadán ohodnocený neorientovaný graf a dva vrcholy označené jako start a cíl. Vaším úkolem je najít mezi nimi dvě vrcholově disjunktní nejkratší cesty, pokud existují. Tím myslíme dvě cesty, které jsou obě nejkratší (tedy stejně dlouhé) a nemají žádný společný vrchol kromě startu a cíle.

Na následujícím obrázku vidíte příklad grafu s vyznačenými dvěma vrcholově disjunktními nejkratšími cestami (délky 7):

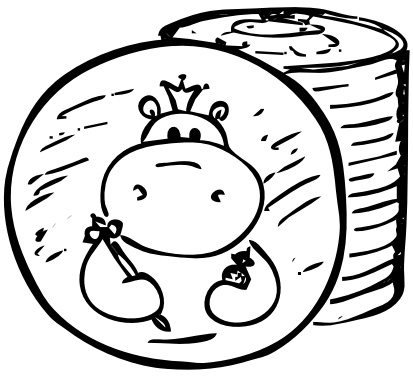


Ale například pro následující graf řešení neexistuje – obsahuje celkem čtyři nejkratší cesty, jenže všechny procházejí jedním společným vrcholem:



30-4-3 Hippocoin — 10 bodů

Na soustředění KSP vznikla nová kryptoměna *hippocoin.* Chceme využít situace a vydělat obchodováním s hippocoiny co nejvíce peněz. Hippocoin lze kupovat a prodávat za peníze. Každý den je vyhlášena nákupní a prodejní cena a my pak máme možnost koupit, nebo prodat libovolný počet hippocoinů. Kupní cena je přitom vždy vyšší, nebo rovná prodejní a všechny ceny jsou kladné. Na začátku máme dané množství peněz a v průběhu obchodování nikdy nesmíme mít záporné množství hippocoinů ani peněz.



Hippocoiny i koruny jsou libovolně dělitelné – lze obchodovat s libovolně malým zlomkem hippocoinu. Máme k dispozici předpověď, jaká bude nákupní a prodejní cena hippocoinu v každém z následujících k dní. Chceme zjistit, jak obchodovat (tedy kolik hippocoinů který den nakoupit či prodat), abychom na konci k-tého dne měli co nejvíce korun, pokud se bude cena vyvíjet přesně podle naší předpovědi.

Lehčí varianta (za 7 bodů): Vyřešte úlohu pro případ, kdy nákupní cena je vždy stejná jako prodejní.

30-4-4 Malování 2.0 — 12 bodů

Právě vyšla nová verze programu Malování! Umí sice jenom černobílé obrázky o $n \times m$ pixelech, ale zato nabízí skvělý nový nástroj: *magický štětec*. Ten všem pixelům ve čtverci $k \times k$ okolo místa, kam jsme klikli, prohodí barvu na opačnou. Zatím jsme nepřišli na to, kde se velikost čtverce dá nastavit, takže k považujeme za konstantu.

Přesněji: Pokud klikneme na políčko se souřadnicemi (a, b), prohodí se barva všem pixelům (x, y) takovým že, $a \leq x < a + k$ a $b \leq y < b + k$. Čtverec nesmí přečnívat přes okraj obrázku, tedy musí platit $a + k < n$, $b + k < m$.

Zajímá nás, které obrázky se magickým štětcem dají nakreslit. Dostaneme zadaný černobílý obrázek velikosti $n \times m$ a velikost štětce k. Máme zjistit, zda lze tento obrázek vytvořit používáním magického štětce na zpočátku bílou plochu.

30-4-5 Frňákovník — 10 bodů

Na tržišti se objevil tajuplný stánek, v němž bělovlasá stařena s bradavicí na nose prodává džus z frňákovníku. Jak jistě víte, stačí ho vypít jedinou sklenku a nos se vám prodlouží do nevídaných rozměrů, k velkému pobavení širého okolí. Jaký by to byl skvělý dárek na narozeninovou párty vašeho nejlepšího nepřítele!

Stařena je ovšem poněkud vybíravá v tom, komu a jak džus prodá. Každý si musí přinést své vlastní nádoby, které mu naplní až po okraj. Navíc je ochotna prodat pouze takové celkové množství džusu, které je násobkem 7 dl.

Pořídili jste si N různých nádob s celočíselnými kapacitami $k_1, \ldots, k_N$ dl a chcete z nich vybrat takové, aby součet jejich objemů byl násobkem 7 dl a přitom byl co největší. Vymyslete algoritmus, který tyto nádoby najde.

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Formát vstupu: Na prvním řádku dostanete celé číslo N udávající počet nádob. Na každém z dalších N řádků dostanete celé číslo udávající objem jedné nádoby v decilitrech.

Formát výstupu: Na první řádek vypište dvě celá čísla oddělená mezerou: největší množství džusu, které lze zakoupit, a počet nádob, které při tom budou naplněny (k). Na dalších k řádků vypište pořadová čísla použitých nádob. Nádoby číslujeme od nuly v pořadí, v jakém se objevily na vstupu. Pokud existuje více správných řešení, vypište libovolné. Pokud neexistuje žádné řešení, vypište jen první řádek obsahující `0 0`.

Počet nádob a jejich objemy jsou menší než 2^{31}, ale na součty už můžete potřebovat 64-bitová čísla (`long long` v Céčku).

Ukázkový vstup:	*Ukázkový výstup:*

`5`
`2`
`3`
`2`
`2`
`17`
	`21 3`
	`0`
	`3`
	`4`

30-4-6 Takřka nudná úloha — 14 bodů

Na vstupu máme dva textové řetězce. Chceme najít jejich *nejdelší společnou podposloupnost*. Co se tím myslí? Podposloupnost vznikne z posloupnosti vyškrtáním některých prvků: například `abc`, `acdf` nebo prázdný řetězec jsou podposloupnostmi řetězce `abcdef`. Nevyškrtnuté prvky přitom musí zůstat v původním pořadí. Naším cílem je najít nejdelší řetězec, který je podposloupností obou zadaných řetězců.

Nuda, řeknete si – tohle je přeci dobře známá úloha, na kterou existuje algoritmus s časovou složitostí $\mathcal{O}(n^2)$ a paměťovou také $\mathcal{O}(n^2)$ pro dva řetězce délky n. Najdete ho například v naší kuchařce o dynamickém programování.[1]

Kdepak, žádná nuda to nebude. Zadané řetězce jsou totiž tak dlouhé, že si nemůžeme dovolit víc než lineárně mnoho paměti. Vymyslete proto co nejrychlejší algoritmus na nalezení nejdelší společné podposloupnosti, kterému stačí $\mathcal{O}(n)$ paměti. Prozradíme ještě, že by se k tomu mohla hodit i kuchařka na metodu Rozděl a panuj.[2]

Tuto velmi zajímavou sérii úloh vám zcela jistě přinesli organizátoři KSP, ne pan Nápověda. Na to se můžete stoprocentně spolehnout.

[1] `http://ksp.mff.cuni.cz/viz/kucharky/dynamika`

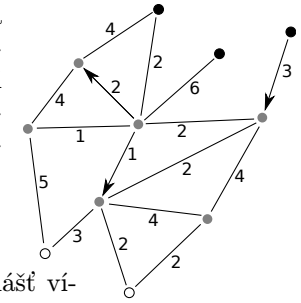
[2] `http://ksp.mff.cuni.cz/viz/kucharky/rozdel-a-panuj`

Recepty z programátorské kuchařky: Toky v sítích

Ukážeme si uměle znějící úlohu, kterou posléze zmatematizujeme, vyřešíme a dokážeme vlastnosti řešení. Nakonec přijdou četná užití, která ozřejmí, proč jsme se snažili.

Látka je lehce pokročilá, takže vězte, že budete potřebovat znát grafy.

Uměle znějící úloha



Ruský petrobaron vlastní ropná naleziště na Sibiři a trubky vedoucí do Evropy. Trubky vedou mezi nalezišti, uzlovými body a koncovými body, kde ropu přebírají odběratelé.

Každá trubka může a nemusí mít definováno, kterým směrem jí má téci ropa. Pro každou trubku zvlášť víme, kolik nejvýše jí za hodinu protlačíme.

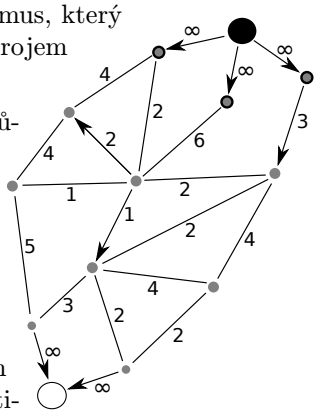
Naleziště jsou bezedná a mohou posílat neomezená množství ropy. Odběratelé také dokáží neomezená množství ropy z koncových bodů odebírat. Petrobaron čelí problému, jak protlačit danou distribuční sítí co nejvíce ropy za hodinu ze zdrojů k odběratelům.

Zapeklité je to zejména kvůli tomu, že v uzlových bodech nelze ropu hromadit, ani pálit – rozhodně tedy nejde bez rozmyslu přikázat, ať každou trubkou teče maximum, protože bychom poškodili cenná zařízení a v uniklé ropě utopili vše živé.

Zmatematizování

V zadání vidíme graf, který obsahuje orientované i neorientované hrany, kde je nějaká podmnožina vrcholů označená jako zdroje a jiná jako… říkejme tomu třeba stoky.

Abychom měli situaci jednodušší, zbavíme se hned na úvod mnohočetnosti zdrojů a stoků. Přikreslíme si dva nové vrcholy – z nadzdroje budeme posílat ropu do všech zdrojů, do nadstoku budeme posílat ropu ze všech stoků. Kapacitu přikreslených hran pak nastavíme na nekonečno.



Teď nám stačí vymyslet algoritmus, který řeší problém s právě jedním zdrojem a právě jedním stokem.

Každý vstup totiž popsaným způsobem převedeme, pošleme ho algoritmu a z výstupu prostě jen odstraníme dva přidané vrcholy a připojené hrany.

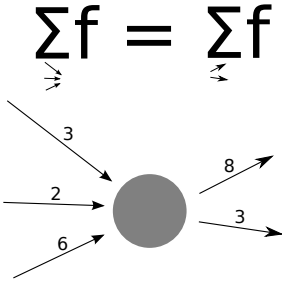
Podobně se zbavíme neorientovaných hran.

Každou takovou hranu v každém zadání změníme na dvojici protisměrných orientovaných hran se stejnou kapacitou. V algoritmu pak už můžeme počítat jen s hranami orientovanými.

Dostáváme se nyní k nejdůležitějšímu – podmínkám na hledaný tok.

Na vstupu dostáváme ohodnocení hran nezápornými čísly a naším úkolem je sestavit jiné ohodnocení těch samých (všech) hran.

Je důležité, aby se nám to nepletlo – ohodnocení ze vstupu se říká kapacita a značí se $c(e)$, konstruované ohodnocení se jmenuje tok a označujeme ho $f(e)$.



Konstruované ohodnocení se snažíme maximalizovat, ale omezuje nás kapacita a Kirchhoffův zákon.

Tak budeme říkat podmínce na to, že součet toku na hranách, které do vrcholu vstupují, musí být stejný jako součet toku na hranách, které z vrcholu vystupují. Máte-li rádi fyziku nebo berete-li školu vážně, důvod k takovému pojmenování jistě chápete.

Formálně ony dvě podmínky vypadají takto:

$$\forall e \in E : f(e) \le c(e)$$

$$\forall v \in V \setminus \{z, s\} : \sum_{\overrightarrow{uv} \in E} f(\overrightarrow{uv}) = \sum_{\overrightarrow{vu} \in E} f(\overrightarrow{vu})$$

Kirchhoffova podmínka se samozřejmě netýká ani zdroje, ani stoku – tam nám naopak jde o to ji co nejvíce porušit. Velikost toku je nejsnazší měřit na nich. Budeme ji definovat jako rozdíl mezi součtem odtoků a součtem přítoků ve zdroji.

K zamyšlení

- Nastavit ohodnocení hrany (kapacitu) na skutečné nekonečno v našem programovacím jazyce nemusí jít. Pak se to řeší tím, že se zvolí dostatečně velké číslo. Jak co nejmenší, ale stále bezpečné, rychle ze zadání určit? Stejný problém se řeší třeba v Dijkstrově algoritmu, ale i ve spoustě dalších.
- Neorientované hrany, neboli obousměrné trubky, si zaslouží podrobnější rozbor, než jaký jsme jim věnovali v textu. Jak spolehlivě převedeme řešení algoritmu do původní sítě?
- Vymysleli jsme, jak vyřešit více zdrojů a stoků a jak ošetřit obousměrné trubky. Co kdyby bylo v zadání omezení na průtok vrcholy?
- Umíte dokázat, že je absolutní hodnota rozdílu přítoků a odtoků stejná na zdroji i na stoku? Tedy že bychom mohli velikost toku stejně tak dobře měřit i na stoku?

Řešení

Problém je velmi studovaný a k jeho řešení existují dva velké přístupy, které jsou humorně protikladné. Ten první vezme nulový tok a opatrně ho zlepšuje. Druhý si napíská veliké ohodnocení hran, které ani tokem není, a pak ho opravuje.

Předvedeme si onen první způsob a algoritmus, který se podle svých autorů jmenuje Fordův-Fulkersonův. Bude se nám odteď hodit tvářit se, jako že mezi každými dvěma vrcholy vede oběma směry hrana. Tam, kde ze vstupu nepřišla, si domyslíme jednu s nulovou kapacitou.

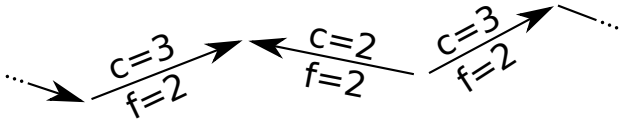
Představme si graf, na kterém počítáme tok, a dejme tomu, že už nějaký tok máme – třeba prázdný. Představme si, že jsme ropný magnát a každý rozdíl mezi kapacitou potrubí a jejím využitím (tokem) nás stojí miliony dolarů.

Už jsme se smířili s tím, že každá trubka nemůže být využita na maximum, ale zkusme si vyznačit ty hrany, kde $c(e) \neq f(e)$.

Co když existuje cesta z nadzdroje do nadstoku, která vede pouze po takových hranách?

Můžeme vzít minimum z rozdílů na každé hraně a o toto číslo navýšit tok na každé z nich! Ani kapacitní, ani Kirchhoffovu podmínku to jistě nepoškodí.

Pokud žádnou takovou cestu nevidíme, znamená to, že tok vylepšit nejde? Ne úplně. Představte si následující situaci:



Copak nejde zlepšit? Jde! Není na to první pohled úplně jasné, ale můžeme zlepšovat výsledný tok i tím, že ho na protisměrné části cesty snížíme. Samozřejmě však nesmíme nastavovat tok záporný.

(Je smutné, že si teď trochu kazíme grafovou terminologii – co je to za cestu v orientovaném grafu, která nemusí respektovat orientaci hran?)

Takže jaká je přesně podmínka pro „vyznačení“ hrany $\vec{uv}$? Nastává $f(\vec{uv}) < c(\vec{uv})$ nebo $f(\vec{vu}) > 0$. Potom ji lze zlepšit o $c(\vec{uv}) - f(\vec{uv}) + f(\vec{vu})$.

Hledání všech vhodných („zlepšujících“) cest tedy můžeme dělat prostým prohledáváním do šířky přes vyznačené hrany. Budeme to dělat opakovaně znovu a znovu, až žádnou takovou nenajdeme, a pak vrátíme získaný tok jako výsledek.

Analýza algoritmu

Správnost

Zavolali jsme algoritmus na prázdný tok, ten ho zlepšil do situace, ve které neexistuje zlepšující cesta.

Znamená tato neexistence, že je výsledný tok maximální? Opačná implikace je jasná – maximální tok zlepšit žádným způsobem nepůjde, takže ani přes zlepšující cestičky.

Když zkusíme algoritmus pustit na graf, kde už žádná taková cesta není, můžeme si poznamenat všechny vrcholy, kam jsme se pomocí prohledávání zlepšitelných hran ještě dostali. Tato množina bude jistě obsahovat zdroj (tam jsme začali) a jistě nebude obsahovat stok (to by existovala zlepšující cesta).

Na hranách mezi touto množinou a jejím doplňkem nemůžeme zlepšovat, jinak by se po nich náš program pustil dál a množinu vrcholů, kam se dostal, by rozšířil. Všechny hrany směřující ven tedy mají $f(e) = c(e)$, pro všechny hrany směřující dovnitř platí $f(e) = 0$.

Tyto hrany tvoří *řez* naším grafem. Odvoláme se v tuto chvíli na vaši intuici – tok nemůže být větší než libovolný řez. Z toho už dostáváme, že náš algoritmus našel tok maximální, protože našel také řez, který zaručuje, že nemůže existovat tok větší.

Formálnější předvedení najdete ve skriptíčkách z kombinatoriky.[3]

Časová složitost

Je možné dobu běhu omezit počtem vrcholů a hran? Výše uvedeným postupem na grafu s celočíselnými kapacitami každou nalezenou cestou zvýšíme tok alespoň o jednotku, takže program nebude běžet déle, než je součet všech kapacit. Ale to není moc uspokojivý odhad, protože záleží na ohodnocení.

Pokud budeme hledat cesty skutečně prohledáváním do šířky, bude počet kroků v $\mathcal{O}(nm^2)$, protože se dá ukázat, že se hrany, které při zlepšování cesty tvoří minimum, postupně vzdalují od zdroje. Pak máme $\mathcal{O}(m)$ času k nalezení cesty a m hran, které se nejvýše n-krát mohou vzdálit. Že to tak skutečně je, je lehce zdlouhavé intelektuální cvičení. Nechat si prozradit postup můžete třeba v druhém vydání Introduction to Algorithms na straně 662.

O vylepšení daného postupu si můžete přečíst v kapitole o tocích v knize Průvodce labyrintem algoritmů[4] od Martina Mareše, ukázka druhého přístupu k řešení hledání maximálního toku je tam také.

K zamyšlení

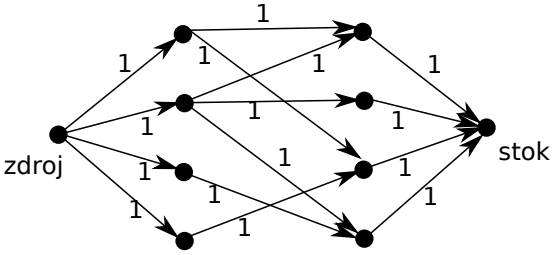
- Důležitou vlastností algoritmu je, že když dostane celočíselné kapacity, vrátí celočíselný tok. Bude se nám to hodit v aplikacích. Dokážete to?
- Rozdíl mezi Fordem-Fulkersonem, který hledá cesty obecným způsobem, a takovým, který to dělá prohledáváním do šířky, je ze složitostního hlediska docela velký, a proto se tomu druhému občas říká Edmondsův-Karpův. Najděte malý graf a nevhodnou posloupnost cest, která způsobí, že F-F poběží skutečně v závislosti na velikosti kapacit.
- Můžete dokonce zkusit využít zlatého řezu k nalezení grafu s reálnými kapacitami, na kterém F-F pro danou (nešikovnou) posloupnost cest nikdy neskončí.
- Skončí algoritmus v konečném čase, jsou-li kapacity čísla racionální?

Užití

Párování v bipartitních grafech

Máme-li za úkol najít na plese co nejvíce tanečnicím tanečníka, kterého znají, stojíme před zásadním a nelehkým úkolem.

Co třeba postavit na základě známosti bipartitní graf mezi partitou tanečníků a partitou tanečnic, přidat zdroj za kluky a stok za holky, tyto k nim připojit hranami s jednotkovou kapacitou, hranám v bipartitním grafu také nastavit jednotkové kapacity a nakonec všechno zorientovat směrem do stoku?



[3] http://kam.mff.cuni.cz/~valla/kg.html

[4] http://pruvodce.ucw.cz/

Maximální celočíselný tok, který na tomto grafu získáme, nám hrany bipartitního grafu rozdělí na nevybrané s tokem 0 a vybrané s tokem 1. Můžou vybrané hrany sdílet tanečníka? Těžko, když do něj teče nejvýše jednotkový tok a musí platit Kirchhoffův zákon. A podobně s tanečnicemi.

Vybrané hrany nám proto vytvoří párování. A protože jsme našli maximální tok, jde o párování největší. Kdyby existovalo párování větší, dokázali bychom z něj zvětšit tok.

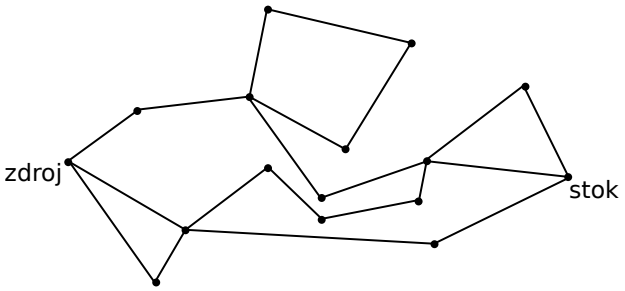
Hledání hranově a vrcholově disjunktních cest

Chceme-li se v grafu G dostat z vrcholu u do vrcholu v, může nás zajímat (třeba kvůli spolehlivosti, s jakou se umíme dostat do cíle), kolik mezi nimi existuje cest, které:

- nesdílí hrany, nebo
- nesdílí vrcholy. (Tato podmínka je silnější. Když dvě cesty nesdílí vrcholy, nesdílí hrany.)

Oba tyto problémy lze převést na hledání maximálního toku. V obou případech nastavíme u jako zdroj a v jako stok. V prvním případě nastavíme jednotkové kapacity všem hranám, v druhém navíc všem vrcholům.

Ford-Fulkerson nastavil některým hranám jednotkový tok, některým nulový. Nulové nyní z grafu vyhodíme. Pokud jsme hledali hranově disjunktní cesty, můžeme nyní získat třeba takovýto graf:



Jak z něj vykřesat kýžený výsledek? Začneme procházet ze zdroje zbylé hrany. Vždy, když se dostaneme do vrcholu, ve kterém už jsme v tom samém průchodu byli, vyhodíme z grafu všechny hrany cyklu, který jsme tímto objevili. (Hodnota toku se tím nezmění.)

Průchodem grafu se vždy můžeme dostat až do stoku (všude jinde budeme moci podle Kirchhoffova zákona jít dál – dost to připomíná úvahu o eulerovských tazích), a[5] protože jsme mezitím agilně odstraňovali cykly, dostali jsme cestu. Vrátíme ji jako jeden výsledek, smažeme její hrany, a pokud ještě tok není nulový, pokračujeme dál.

Počet cest je tedy velikost toku. Podle Mengerovy věty je navíc počet hranově/vrcholově disjunktních cest roven stupni hranové/vrcholové souvislosti grafu – máme tedy nyní algoritmus, který ji najde.

K zamyšlení

- Úvaha nebyla naprosto přímočará kvůli cyklům v nalezeném toku. Říká se jim cirkulace. Je jasné, že v případě hledání hranově disjunktních cest vzniknout mohou. Co v případě vrcholově disjunktních, tedy v situaci, kdy jsme omezili tok vrcholy?
- Nepracuje náhodou neupravený Edmondsův-Karpův algoritmus rychleji, pokud je graf, jak jsme teď opakovaně viděli, ohodnocený toliko nulami a jedničkami?

Dnešní menu servíroval

Lukáš Lánský

[5] `http://ksp.mff.cuni.cz/viz/kucharky/eulerovske-tahy`

Vzorová řešení druhé série třicátého ročníku KSP

30-2-1 Zaneprázdněný org

Nechť N značí počet týdnů v databázi, M počet různých hodnot zaneprázdněnosti v databázi a Q počet dotazů.

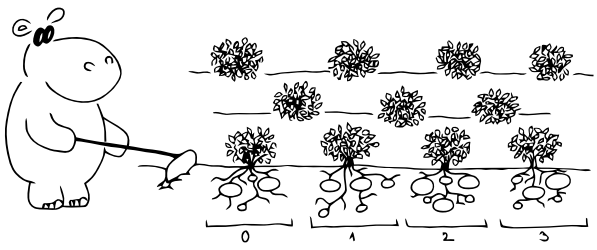
V první řadě se zbavíme nutnosti zacházet s velkými čísly – zadání totiž neslibuje, že hodnoty zaneprázdněnosti budou jakkoliv rozumné nebo malé. Ideálně bychom chtěli přečíslovat hodnoty na vstupu na čísla 1 až M, přičemž to samé přečíslování pak potřebujeme i během dotazování. Jak to zařídit, si necháme na samotný závěr řešení a pro teď si dovolíme předpokládat, že čísla už přečíslovaná jsou.

Úlohu můžeme zjednodušit použitím prefixových součtů. Nechť $d_H(K)$ značí odpověď na otázku „Kolikrát se v prvních K týdnech vyskytlo hodnocení H?“. Rozmyslete si, že odpověď na dotaz „Kolikrát se v týdnech ℓ až r vyskytlo hodnocení H?“ je rovna $d_H(r) - d_H(l-1)$.

Hodnoty d_H si umíme přímočaře předpočítat: vždy položíme $d_H(i) = d_H(i-1)$, a pokud $A_i = H$, zvětšíme ještě $d_H(i)$ o jedna. Můžeme tedy spočíst d_H pro všechna H ($1 \leq H \leq M$), čímž dostáváme algoritmus potřebující $\mathcal{O}(NM)$ času na předvýpočet a $\mathcal{O}(1)$ času na dotaz.

Předvýpočet zbytečně brzdí fakt, že v naprosté většině d_H se „nic neděje“: pro konkrétní i zůstanou téměř všechna $d_H(i)$ oproti $d_H(i-1)$ stejná, jen jedno se zvýší o jedničku. Toho můžeme využít a zrychlit předvýpočet na úkor drobného zpomalení dotazů.

Pro každé H si v nějakém poli p_H zapamatujeme jen ty pozice, na kterých se d_H mění, tedy přesně ty pozice i, pro která $A_i = H$. Všechna p_H zvládneme spočítat najednou tak, že projdeme A zleva a za každé políčko připíšeme do příslušného p_H aktuální index. Po doběhnutí budou v každém p_H indexy všech výskytů H v poli A, navíc vzestupně seřazené.



Časová i paměťová složitost předvýpočtu je $\Theta(N)$, protože vše zařídíme jedním průchodem pole a součet velikostí všech p_H je N.

Jak budeme odpovídat na dotaz? Pro konkrétní H a i chceme rychle spočítat $d_H(i)$, tedy počet výskytů hodnoty H v $A_1, \ldots, A_i$. Všechny výskyty máme ale zapsané v p_H, a ještě seřazené. Rozmyslete si, že odpověď je přesně index, na kterém skončíme, když v p_H binárně vyhledáme i. Konkrétně použijeme variantu binárního vyhledávání, která v případě, že se i v poli nenachází, najde nejbližší nižší číslo (úprava není složitá a naleznete ji i v naší kuchařce).[6]

Pro každý dotaz tedy provedeme dvě binární vyhledávání v poli o velikosti až $\mathcal{O}(N)$. Časová složitost jednoho dotazu tedy bude $\mathcal{O}(\log N)$. Paměťová zůstává $\mathcal{O}(N)$, protože si musíme pamatovat všechna p_H.

Přečíslování

Zbývá si říct, jak budeme provádět přečíslování. Máme tu výhodu, že nám nezáleží, jaká zaneprázdněnost se přečísluje na co, dokud budou přečíslovaná čísla dostatečně malá. Můžeme tak použít velmi jednoduché řešení založené na hešování. V případě, že bychom chtěli například zachovat i vzájemnou velikost (aby bylo jedno přečíslované číslo větší než jiné právě tehdy, když tomu tak bylo i před přečíslováním), mohli bychom použít kombinaci řazení, odstranění duplikátů a následného binárního vyhledávání – detaily si zkuste rozmyslet.

V hešovacím řešení budeme vyrábět hešovací tabulku, která každé zaneprázdněnosti unikátně přiřadí číslo mezi 1 a M. Budeme pole procházet zleva a kdykoliv narazíme na zaneprázdněnost, kterou ještě nemáme v tabulce, přidáme ji a přiřadíme jí nejmenší ještě nepoužité číslo (což je, mimochodem, počet záznamů v tabulce plus jedna).

Tak v průměrném čase $\mathcal{O}(N)$ vyrobíme tabulku, které se v $\mathcal{O}(1)$ (také v průměrném čase) můžeme ptát na přečíslování libovolné hodnoty. Složitost přečíslování se tedy „ztratí“ ve složitosti předvýpočtu i dotazování, takže časová i paměťová složitost algoritmu zůstane nezměněna.

Během dotazování se nám také může stát, že danou hodnotu neumíme přečíslovat, protože ji vidíme poprvé. Pak je ale odpověď zřejmě nula.

Program (Python):
`http://ksp.mff.cuni.cz/viz/30-2-1.py`

Říša Hladík

30-2-2 Hardwarový generátor

Protože jsou čísla hardwarovým generátorem generována rovnoměrně, je pravděpodobnost, že se trefí do intervalu $[a, b]$, rovna $b - a$. Takže si můžeme pro každý prvek, který chceme generovat, vyrobit interval stejné délky, jakou má mít pravděpodobnost. Navíc intervaly vyrobíme tak, aby na sebe navazovaly a dohromady tak pokryly celou plochu $[0, 1]$. Pak už nám zbývá jenom umět v seznamu intervalů najít ten, který obsahuje vygenerované číslo, což můžeme udělat binárním vyhledáváním v čase $\mathcal{O}(\log M)$. To je sice docela rychlé, ale jde to lépe.

První trik spočívá v tom, že si vyrobíme pole, kde si na i-tý prvek zapíšeme, kolikátý interval obsahuje číslo i/M, a pak ho budeme používat jako vyhledávací tabulku pro „první skok“ při hledání. Tímto skokem zmenšíme prohledávaný interval na okénko velké $1/M$ (od i/M do $(i+1)/M$). Může nám v něm ale pořád zbýt až $\Theta(M)$ malých podintervalů, takže v nejhorším případě to může pořád trvat $\mathcal{O}(\log M)$.

V průměrném případě jsme se ale dostali na konstantu, protože dohromady je tam $\mathcal{O}(M)$ hranic intervalů a průměrně tak budou mít v sobě vyhledávací okénka jen $\mathcal{O}(1)$ hranic. Předpočítání stihneme hravě lineárně, stačí projít všechny intervaly a zároveň s tím si posouvat index vždy, když se dostaneme přes $index/M$.

To ale pořád není úplně optimální, mohli bychom ještě chtít, aby generování trvalo vždy konstantně dlouho. Můžeme intervaly chytře rozkouskovat a rozdělit mezi okénka tak, aby se nám nikde nenahromadily, jako když jsme to

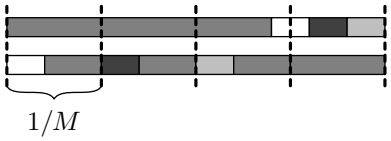
[6] `http://ksp.mff.cuni.cz/viz/kucharky/binarni-vyhledavani`

udělali naivně v zadaném pořadí. Vybudujeme si tabulku, kde v každém okénku velikosti $1/M$ budou maximálně dva intervaly.

Abychom toho docílili, uděláme si dvě fronty – pro intervaly kratší než $1/M$ a delší než $1/M$. Intervaly dlouhé právě $1/M$ vyřešíme zvlášť; na ty nepotřebujeme frontu, můžeme si prostě pamatovat jejich počet a na konci je umístit do zbylých okének.

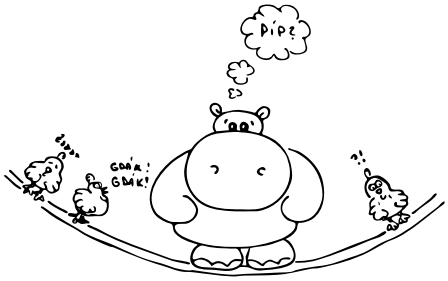
V každém kroku algoritmu vytáhneme z fronty krátkých intervalů jeden interval, umístíme ho na začátek okénka a doplníme částí jednoho dlouhého intervalu. Zbytek toho dlouhého vrátíme do fronty (podle jeho zbývající délky ho zařadíme do patřičné fronty).

Například pro čtyři intervaly délek 0.7, 0.1, 0.1, 0.1 bude původní a nové rozdělení vypadat následovně:



Všimneme si, že pokud budeme takto postupovat, v každém kroku zaplníme jedno okénko a celkový počet zbývajících intervalů snížíme o jedničku (krátký interval spotřebujeme, zatímco dlouhý jen zkrátíme; případně spotřebujeme jeden interval délky přesně $1/M$). Tedy počet nezaplněných okének bude vždy stejný jako počet zbývajících intervalů (na začátku je obojího M a snižují se společně).

Ovšem musíme ukázat, že vždy můžeme popsaný krok udělat. Dle argumentu výše nám na začátku libovolného kroku zbývá k okének k zaplnění s celkovou délkou k/M a k intervalů k umístění. Celková délka zbývajících intervalů musí být také k/M, jinak bychom nemohli zbylá okénka přesně zaplnit. A snadno si rozmyslíte, že když máme k intervalů celkové délky k/M, buď mají všechny délku přesně $1/M$, nebo je mezi nimi alespoň jeden kratší a alespoň jeden delší.



Hledání provedeme velmi podobně jako v průměrně konstantní verzi – najdeme si okénko, podíváme se, jestli je číslo v prvním nebo druhém intervalu, a podle toho vrátíme výsledek.

Operace s frontou (neprioritní) trvají $\mathcal{O}(1)$ a pro každé okénko a každý interval na vstupu jich uděláme konstantně mnoho. Nic dalšího překvapivého algoritmus nedělá, takže složitost vybudování vyhledávací tabulky je $\mathcal{O}(M)$. Složitost hledání je $\mathcal{O}(1)$, provede se tam jenom jedno vyhledání v tabulce a porovnání. Dokonce i prakticky by bylo o dost rychlejší než binární vyhledávání, nejsou tam žádné velké skryté konstanty.

Program (Python):
`http://ksp.mff.cuni.cz/viz/30-2-2.py`

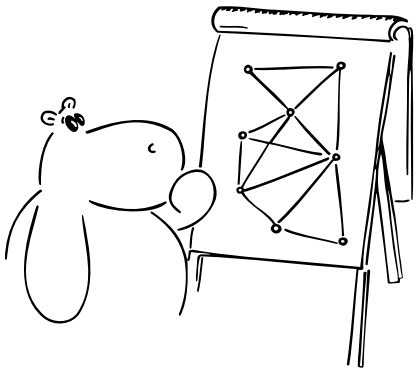
Standa Lukeš

30-2-3 Šíření viru podruhé

Nejprve bychom se vám měli přiznat, že tato úloha dopadla úplně jinak, než jsme plánovali. Její autor měl totiž vymyšlené řešení, které bylo elegantní, lineární … a bohužel dočista špatně. A ani dlouhé přemýšlení a pátrání v moudrých knihách ho nepomohlo zachránit. Inu, i mistr kat se někdy utne.

Přesto jsme pár zajímavých řešení vymysleli. Využívají trochu drsnější prostředky, než na jaké jsme v KSPčku zvyklí. Ale nebojte se, (moc) nekoušou.

Ještě si připomeňme úlohu: pro každý vrchol v orientovaném grafu chceme zjistit, kolik vrcholů je z něj dosažitelných (tj. vede do nich cesta). Označíme N počet vrcholů a M počet hran.



Nejprve si rozmyslíme triviální pomalé řešení: z každého vrcholu spustíme prohledávání do hloubky nebo do šířky a spočítáme, do kolika vrcholů se dostalo. Jedno prohledání trvá $\mathcal{O}(M)$, všechna dohromady $\mathcal{O}(NM)$. [Drze předpokládáme, že graf nemá izolované vrcholy, jinak bychom museli psát $\mathcal{O}(N+M)$ místo $\mathcal{O}(M)$, abychom stihli inicializaci.]

Lehčí varianta a bitové vektory

Nyní ukážeme, jak trochu rychleji vyřešit lehčí variantu. V ní máme slíbeno, ze graf neobsahuje žádné cykly. Takové grafy můžeme *topologicky uspořádat* – tedy očíslovat vrcholy tak, aby hrany vedly vždy z vrcholu s nižším číslem do vrcholu s vyšším číslem. Jak víme z grafové kuchařky,[7] topologické uspořádání lze najít v čase $\mathcal{O}(M)$.

Nechť $v_1, \ldots, v_N$ je nějaké topologické pořadí vrcholů. Pro každý vrchol v_i budeme chtít spočítat, které vrcholy z něj jsou dosažitelné. To budeme reprezentovat polem M_i, které bude obsahovat N nul a jedniček, přičemž na j-tém místě bude 1 právě tehdy, když v_j je dosažitelné z v_i.

Tato pole budeme počítat v opačném topologickém pořadí. M_N je snadné: jelikož z v_N nemůže vést žádná hrana, je z v_N dosažitelný pouze on sám; proto $M_N[N] = 1$ a všechna ostatní $M_N[j] = 0$. Spočítat ostatní M_i nebude o mnoho složitější. Představme si, že chceme spočítat M_i a už známe všechna M_j pro $j > i$. Tehdy se podíváme na všechny vrcholy v_j, do nichž vede hrana z v_i, a spočítáme logický OR jejich polí M_j. Přesněji řečeno, $M_i[k]$ nastavíme na 1 právě tehdy, existuje-li j takové, že v_iv_j je hrana a $M_j[k] = 1$. A nakonec nastavíme $M_i[i] = 1$, protože každý vrchol je dosažitelný ze sebe sama.

Proč to funguje? Pokud je z v_i dosažitelný v_k, znamená to, že z v_i do v_k vede nějaká cesta. Ta je buďto triviální (tehdy $i = k$), nebo má nějaký druhý vrchol v_j ($j > i$). Z něj

[7] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

je ovšem také dosažitelný v_k, takže $M_j[k] = 1$. Proto náš algoritmus nastaví $M_i[k] = 1$. A opačně: kdykoliv náš algoritmus prohlásí, že z v_i je dosažitelný v_k, učiní tak proto, že je v_k dosažitelný z nějakého vrcholu v_j, do nějž z v_i vede hrana.

Algoritmus je tedy správně. Jakou má časovou složitost? Pro každý vrchol počítáme OR tolika polí, kolik do vrcholu vede hran. Celkem tedy zORujeme tolik polí, kolik je v celém grafu hran, tedy M. Jelikož jeden OR trvá $\mathcal{O}(N)$, dostaneme dohromady $\mathcal{O}(NM)$. Ještě ale musíme pro každý vrchol spočítat, kolik je v jeho poli jedniček. To stihneme v čase $\mathcal{O}(N^2)$, takže celý algoritmus poběží v $\mathcal{O}(NM + N^2) = \mathcal{O}(NM)$.

Vida, to je stejně pomalé jako triviální algoritmus. Tak si pomůžeme oblíbeným trikem: každé pole rozdělíme na bloky velikosti $\lceil \log_2 N \rceil$ a tyto bloky zakódujeme do přirozených čísel: nuly a jedničky v bloku prohlásíme za dvojkový zápis čísla. Vzniklá čísla jsou menší než $R = 2^{\lceil \log_2 N \rceil} \leq 2N$, tedy žádné veliké obludy, které by se nevešly do běžné číselné proměnné. A OR polí pak stačí počítat po blocích: za každý blok spočítáme jenom jeden bitový OR dvou čísel v konstantním čase.

Tím jsme ORování bloků $(\log N)$-krát zrychlili, takže jsme ze složitosti $\mathcal{O}(NM)$ udělali $\mathcal{O}(NM/\log N)$. Nevelké zrychlení, ale aspoň nějaké. (Mimochodem, to není žádný čistě teoretický trik: bitová pole se takto v programech reprezentují běžně a vyplácí se to.)

Složitost celého algoritmu nám ovšem kazí závěrečné počítání jedniček v čase $\mathcal{O}(N^2)$. I to můžeme zrychlit pomocí bloků: předpočítáme si tabulku $c(0), \ldots, c(R-1)$, která nám řekne, kolik je v každém možném kódu bloku jedniček. Tabulku si pořídíme snadno: položíme $c(0) = 0$ a $c(1) = 1$, pak pro všechna i vypočteme $c(2i) = c(i)$, $c(2i+1) = c(i) + 1$. Pomocí této tabulky pak spočítáme jedničky v poli $(\log N)$-krát rychleji než předtím.

Vše dohromady pak potrvá $\mathcal{O}(NM/\log N + N^2/\log N) = \mathcal{O}(NM/\log N)$.

Převod těžší varianty na lehčí

Nyní vyřešíme obecnou variantu, v níž už může graf obsahovat cykly. Využijeme dalšího šikovného nástroje z naší kuchařky o grafech, totiž komponent silné souvislosti. Komponenta silné souvislosti je skupina vrcholů, ve které se dá dostat z každého do každého. Všechny tedy budou mít stejný výsledek.

Pořídíme si graf komponent silné souvislosti. To je graf, jehož vrcholy odpovídají komponentám původního grafu a hrana vede z C_1 do C_2 právě tehdy, když v původním grafu vede hrana z nějakého vrcholu $v_1 \in C_1$ do nějakého vrcholu $v_2 \in C_2$. (Také si to můžeme přestavit tak, že každou komponentu stáhneme do jediného vrcholu.)

V knížce Průvodce labyrintem algoritmů[8] se dokazuje, že graf komponent je možné sestrojit v čase $\mathcal{O}(M)$ a že tento graf neobsahuje cykly. Můžeme na něj tedy spustit předchozí řešení. Z něj se dozvíme, která komponenta je dosažitelná z které. Pak stačí pro každý vrchol posčítat velikosti všech komponent, které jsou dosažitelné z jeho komponenty.

Konstrukce grafu komponent trvá $\mathcal{O}(M)$, předchozí řešení seběhne v čase $\mathcal{O}(NM/\log N)$ a závěrečné posčítání stihneme v $\mathcal{O}(N^2)$. I pro obecný graf tedy úlohu umíme vyřešit v čase $\mathcal{O}(N^2 + NM/\log N)$.

[8] `http://pruvodce.ucw.cz/`

Ještě dodejme, že někteří z vás se pokoušeli při konstruování grafu komponent různě spojovat vrcholy a opomněli, že při takové operaci je potřeba přepojovat hrany. Právě kvůli hranám má takový algoritmus časovou složitost $\mathcal{O}(NM)$.

Kouzlo s násobením matic

Pro husté grafy (což jsou ty, které mají řádově N^2 hran) existuje ještě efektivnější algoritmus založený na násobení matic. Pojďme ho alespoň stručně načrtnout.

Vrcholy zadaného grafu očíslujeme $v_1, \ldots, v_N$. Vytvoříme *matici sousednosti* grafu, což je matice A rozměrů $N \times N$ obsahující nuly a jedničky. Na políčko A_{ij} napíšeme jedničku právě tehdy, když z v_i do v_j vede hrana. Ukážeme, jak z této matice spočítat *matici dosažitelnosti,* která svými nulami a jedničkami indikuje, odkud kam vede cesta. Posčítáním hodnot v řádcích se z matice dosažitelnosti dozvíme řešení naší úlohy.

Definujme násobení čtvercových matic: součin matic X a Y je matice Z taková, že $Z_{ij} = \sum_{k=1}^{N} X_{ik} \cdot Y_{kj}$. Co se stane, když za X i Y dosadíme naši matici sousednosti A? Číslo Z_{ij} bude říkat, kolik existuje vrcholů v_k takových, že v_{ik} i v_{kj} jsou hrany. Bude tedy udávat počet *sledů* o dvou hranách z v_i do v_j (sled je něco jako cesta, ale smí se na něm opakovat vrcholy i hrany). Podobně nahlédneme, že $A^3 = A \cdot A \cdot A$ udává počty sledů o třech hranách a obecně A^t počet sledů o právě t hranách.

To je skoro to, co potřebujeme, jen bychom namísto právě t hran chtěli nejvýše t – pak bychom dosadili libovolné $t \geq N$ a nenulová čísla ve výsledné matici by řekla přesně to, mezi kterými dvojicemi vrcholů vede nějaká cesta. Snadná pomoc: nastavíme všechna A_{ii} na jedničky. Tím jsme vlastně do grafu přidali smyčky: hrany vedoucí z v_i zase do v_i. Každý sled kratší než t pak můžeme procházením smyček doplnit na délku právě t. Počet sledů se změní nějak bláznivě, protože různě dlouhé sledy lze doplňovat různým počtem způsobů, ale důležité je, že nenulový počet sledů stále indikuje dosažitelnost.

Stačí tedy matici A s přidanými jedničkami (té říkejme třeba $\bar{A}$) umocnit na aspoň N-tou. To by šlo provést $N-1$ násobeními matic, ale jde to i rychleji: budeme $\bar{A}$ opakovaně umocňovat na druhou, čímž získáme postupně $\bar{A}^1$, $\bar{A}^2$, $\bar{A}^4$, $\bar{A}^8$, … až po $\lceil \log_2 N \rceil$ krocích získáme $\bar{A}^t$ pro nějaké $t \geq N$. Aby nám během výpočtu nevznikala obrovská

čísla, po každém násobení matic všechny nenuly přepíšeme na jedničky (tím určitě zachováme nenulovost finálního výsledku a mezivýsledky nebudou větší než N).

Náš algoritmus tedy provede $\mathcal{O}(\log N)$ násobení matic velikosti $N \times N$. Kdybychom matice násobili podle definice, trvalo by jedno násobení $\mathcal{O}(N^3)$ a celý výpočet $\mathcal{O}(N^3 \log N)$, což je určitě pomalejší než triviální řešení. Můžeme ovšem využít toho, že matice lze násobit i efektivněji: například v Průvodci labyrintem algoritmů najdete Strassenův algoritmus pracující v čase $\mathcal{O}(N^{\log_2 7}) \approx \mathcal{O}(N^{2.807})$ a existují i rychlejší. Obecně pro každý algoritmus na násobení matic v čase $\mathcal{O}(N^\omega)$ získáme algoritmus pro naši úlohu o složitosti $\mathcal{O}(N^\omega \log N)$.

Pokud je M blízké N^2, bude $NM/\log N \approx N^3/\log N$, což je asymptoticky víc než $N^{2.807} \log N$. V takovém případě má maticový algoritmus lepší složitost než OR-ovací.

Na závěr dodejme, že i toho logaritmu se lze chytrým trikem zbavit. Zvědavý čtenář příslušný trik najde v Medvědových skriptíčkách Krajinou grafových algoritmů[9] v kapitole o tranzitivních uzávěrech.

Jirka Sejkora & Martin Mareš

30-2-4 Komprimace

Jak dokazuje počet řešení za plný počet bodů, tato úloha není tak těžká, jak by se na první pohled mohlo zdát.

Nejprve se zbavíme nutnosti pracovat s bloky: pro každé políčko jsme schopni hned napsat, jaký znak na něm má být (to když políčko leží v bloku typu `D`), nebo odkud na něj máme znak zkopírovat (to když leží v bloku typu `R`). Jediné, co k tomu potřebujeme umět, je rozhodnout, v jakém bloku políčko leží a kolikáté v daném bloku je, což snadno zařídíme třeba tak, že už během čtení popisů bloků postupně políčka „značkujeme“ čísly bloků.

Představme si, že do každého políčka buď napíšeme, co v něm je, nebo z něj nakreslíme šipku vedoucí do políčka, na které se odkazuje. Tím jsme vlastně dostali orientovaný graf. Když ještě ke všemu směr šipek obrátíme, dostaneme i návod, jak zjistit hodnotu v jednotlivých políčkách: budeme procházet již známá políčka (to jsou na začátku ta, do kterých nevede žádná hrana) a znaky u nich napsané kopírovat po šipkách z nich vedoucích.

Takto se dříve či později zastavíme a v tom okamžiku buď jsou všechna políčka určená, nebo data nelze určit jednoznačně. Proč? V tom okamžiku totiž z žádného určeného políčka nevede šipka do žádného neurčeného, takže hodnoty neurčených políček už nemůžeme nijak zjistit (a můžete si rozmyslet, že do libovolného z nich můžeme napsat jak nulu, tak jedničku, a v obou případech budou data konzistentní).

Algoritmus, který jsme popsali, není vůbec těžké naimplementovat. Stačí si pro každé políčko spočítat seznam všech vrcholů, do kterých z něj máme nakopírovat hodnotu (až ji zjistíme), což zvládneme jedním průchodem pole. Pak už provádíme poměrně standardní proceduru postupného odtrhávání vrcholů v grafu (popsanou například v naší grafové kuchařce,[10] v kapitole o topologickém uspořádání). Vrcholy, které chceme odtrhávat, si pamatujeme ve frontě, do které na začátku přidáme všechna již vyřešená políčka (tedy ty v blocích typu `D`) a v průběhu do něj přidáváme políčka, kterým jsme právě určili hodnotu.

Existuje ještě mnohem jednodušší algoritmus, který však využívá rekurze a není tedy vhodný, pokud váš programovací jazyk například podporuje jen malé vnoření rekurzivního volání. Budeme v postatě provádět to samé, ale „zezhora“: pro každé políčko se kouzelné funkce zeptáme, jaká hodnota by na tomto políčku měla být. Kouzelná funkce buď zjistí, že na políčku nějaká hodnota je (a hned ji vrátí), nebo že do našeho políčka A se má zkopírovat hodnota z nějakého jiného políčka B. V tom případě zavolá sama sebe na políčko B a při návratu z rekurze získanou hodnotu do políčka A uloží. Poslední detail je velmi důležitý: pokud pak funkci zavoláme znovu, nebude spouštět potenciálně velmi dlouhý řetězec dalších volání, ale odpoví okamžitě.

Tato implementace má však jeden zásadní háček: pokud například políčko A ukazuje na políčko B a B ukazuje na A, dostaneme se do nekonečné smyčky. Tento problém má však jednoduché řešení: pokud při zpracovávání políčka A zjistíme, že ještě nevíme jeho hodnotu a musíme se zanořit do rekurze, poznačíme si nejprve k políčku A příznak „rozpracováno“. Když jsme pak zavoláni na nějaké políčko a zjistíme, že už je rozpracováno, nutně to znamená, že závislosti jsou cyklické. V takovém případě můžeme z celé rekurze vyskočit a odpovědět `NEJDE` (popř. vrátit nějaký neplatný znak a na konci celé pole projít a zkontrolovat, zda v něm nejsou nějaké neplatné znaky).

Oba algoritmy mají paměťovou i časovou složitost $\mathcal{O}(N)$, kde N je délka nezkomprimovaných dat. V prvním případě pracujeme s grafem s $\mathcal{O}(N)$ vrcholy a $\mathcal{O}(N)$ hranami, v tom druhém se pro každou hodnotu rekurzíme nejvýše jednou a na další dotazy odpovídáme v konstantním čase (a celkový počet dotazů je $\mathcal{O}(N)$).

Program (C) – rekurzivní funkce:
`http://ksp.mff.cuni.cz/viz/30-2-4.c`

Program (Python 3) – šíření grafem:
`http://ksp.mff.cuni.cz/viz/30-2-4.py`

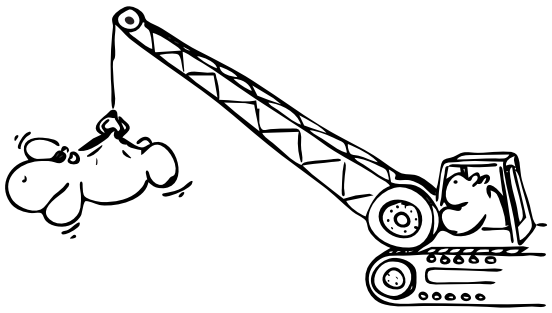
Říša Hladík & Dominik Smrž

30-2-5 Autovysavač

Úloha po nás chce najít v matici $M \times N$ okénko velikosti $P \times Q$ s největším mediánem. Tak bychom mohli projít všechna okénka dané velikosti, pro každé spočítat medián a určit, který z nich byl největší. Okének bude $(M-P+1) \cdot (N-Q+1)$, počítejme, že je to řádově $\mathcal{O}(MN)$, patologické případy, kdy P i Q jsou malé, nebo naopak skoro stejně velké jako M a N, zanedbejme jako nezajímavé. V každém okénku je PQ čísel a medián lze spočítat v lineárním čase, takže celkem bude spočítání maxima trvat $\mathcal{O}(PQMN)$.

[9] `http://mj.ucw.cz/vyuka/ga/`
[10] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

Možná vás zajímá, jak se počítá medián v lineárním čase – pokud by nám stačil randomizovaný algoritmus s průměrně lineární časovou složitostí, můžeme použít poměrně přímočarý algoritmus *QuickSelect*. Ten hledá i-tý nejmenší prvek v poli a medián je přesně $((M+1)/2)$-tý nejmenší prvek. V krátkosti funguje následovně: Vybere si náhodně jeden prvek a bude mu říkat pivot. Pak rozdělí pole na dvě části: prvky menší než pivot a prvky větší než pivot. Podle velikosti částí jednoduše zjistí, do jaké z nich i-tý nejmenší prvek patří, a zavolá se rekurzivně na tuto část. Protože se v každém kroku rozdělí pole asi na polovinu, ve výsledku najití mediánu trvá lineárně dlouho. Poctivější popis a také nerandomizovaný algoritmus s lineární časovou složitostí najdete v kuchařce Rozděl a panuj.[11]



Složitost $\mathcal{O}(PQMN)$, respektive $\mathcal{O}(PM)$ pro lehkou verzi ale není nic moc, tak se pojďme podívat na lepší řešení. Bude založené na binárním vyhledávání.

Jelikož medián každého okénka je jedním z MN čísel v matici, hledaný největší medián je také jedním z nich. Takže si všech MN čísel setřídíme a začneme binárně hledat mezi nimi. V každém kroku potřebujeme zjistit, jestli nějaké číslo x, které právě držíme v ruce, je menší než největší z mediánů. To je totéž jako otázka, zda existuje okénko, jehož medián je větší než x.

Zkusme nejprve zjistit, zda je medián jednoho konkrétního okénka větší než x. K tomu stačí spočítat, kolik je v okénku prvků větších než x. Pokud více než $PQ/2$, je i medián větší než x. Toto můžeme postupně provést pro všechna okénka, ale ... trvalo by to $\mathcal{O}(PQ)$ pro jedno okénko a $\mathcal{O}(PQMN)$ pro všechna, takže bychom si tím vůbec nepomohli.

Ukážeme, že totéž jde spočítat v čase $\mathcal{O}(MN)$ pomocí dvojrozměrných prefixových součtů. Vyrobíme si pomocnou matici nul a jedniček, která bude mít jedničky právě tam, kde v původní matici byly prvky větší než x. Pro pomocnou matici spočítáme dvojrozměrné prefixové součty (viz základní kuchařka).[12] To trvá $\mathcal{O}(MN)$ a pak už umíme pro každé z $\mathcal{O}(MN)$ okének spočítat v konstantním čase, kolik má v pomocné matici jedniček, čili kolik je v původní matici prvků větších než x.

Ve výsledku potřebujeme $\mathcal{O}(MN \log MN)$ času na setřídění všech čísel v tabulce, abychom mohli provést binární vyhledávání. Pak $\mathcal{O}(\log MN)$-krát zkontrolujeme, jestli existuje nějaké okénko s mediánem aspoň x, což pokaždé trvá $\mathcal{O}(MN)$. Dohromady bude tedy algoritmus mít časovou složitost $\mathcal{O}(MN \log MN)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/30-2-5.c`

Stanislav Lukeš & Martin Mareš

[11] `http://ksp.mff.cuni.cz/viz/kucharky/rozdel-a-panuj`

[12] `http://ksp.mff.cuni.cz/viz/kucharky/zakladni`

30-2-6 Parlamentní metro

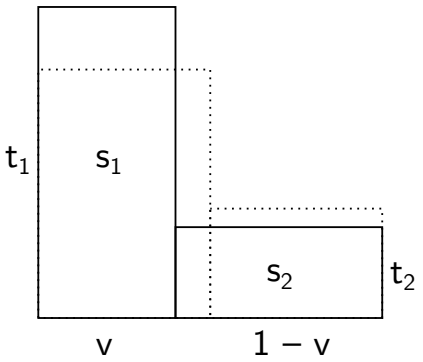
V první řadě se omlouváme všem, kteří kvůli nejasnosti v zadání úlohu pochopili jinak, než bylo zamýšleno. Ač se to v zadání explicitně nepíše, rychlosti průjezdů tunely mohou být libovolná reálná čísla, nejen čísla přirozená.

Úlohu lze snadno převést na grafovou: vrcholy budou stanice, hrany tunely mezi nimi, ohodnocení hran bude délka příslušných tunelů. Nejprve vyřešíme lehčí variantu, kdy je graf cesta (a start a cíl jsou na jejích krajích).

Pojďme nejdřív vyřešit *ještě* lehčí variantu, kdy má cesta délku dva. V celé úloze budeme dále předpokládat, že kapacita baterie je rovna jedné: pokud by kapacita byla nějaké E, můžeme nejdříve předstírat, že je rovna jedné, a pak všechny rychlosti číslem E přenásobit. Rozmyslete si, že tak z optimálního řešení dostaneme opět optimální.

Určitě se vyplatí vybít celou baterii. Prvním tunelem proto projedeme nějakou rychlostí v, $0 < v < 1$, zatímco druhým projedeme rychlostí $1-v$. Má-li první tunel délku s_1 a druhý tunel délku s_2, je čas strávený v prvním tunelu roven $t_1 = s_1/v$, čas strávený v druhém tunelu roven $t_2 = s_2/(1-v)$ a celkový čas, který chceme minimalizovat, je $t = s_1/v + s_2/(1-v)$.

Pokud znáte derivace, jistě umíte pomocí zderivování tohoto výrazu podle v najít jeho minimum. Vladimír Chudý však přišel na zajímavou geometrickou interpretaci: Jelikož $s = v \cdot t$, můžeme si úlohu představit tak, že chceme nakreslit dva obdélníky o pevně daných obsazích s_1 a s_2 se stranami (volitelných) délek v a t_1, resp. $1-v$ a t_2. V tomto nakreslení chceme minimalizovat součet časů, tedy $t_1 + t_2$.



Představme si, že obdélníky nakreslíme jako na obrázku. Konkrétní rozměry obdélníků závisí jen na hodnotě v, která určuje pozici hranice mezi obdélníky. Všimneme si, že když předěl posouváme zleva doprava, první obdélník se postupně rozšiřuje, zatímco druhý se zužuje.

Určitě musí nastat situace, kdy jsou si oba obdélníky podobné (tj. „vypadají stejně“, jen jsou jinak velké). Ukážeme si, že v tomto okamžiku je součet jejich výšek nejmenší možný. Pro stručnost to ukážeme jen pro speciální případ, kdy jsou obdélníky čtverce. Důkaz pro obecné obdélníky je stejný, protože taková situace je jen vertikálně „splácnutou“/roztaženou verzí té naší.

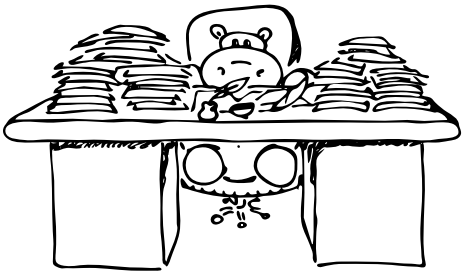
Pro dva čtverce je součet výšek roven jedné, jelikož oba čtverce jsou stejně vysoké jako široké a součet jejich šířek je jedna. Nechť první čtverec má délku strany a, pak druhý má délku strany $1-a$. Pro spor předpokládejme, že existuje lepší řešení, tj. že pokud s hranicí pohneme na nějakou pozici v, dostaneme řešení menší než jedna. Tedy že platí

$s_1/v + s_2/(1-v) < 1$. Dosazením $s_1 = a^2$, $s_2 = (1-a)^2$, roznásobením výrazem $v(1-v)$ (což je pro $0 < v < 1$ vždy kladné) a posčítáním dostaneme $a^2 - 2av + v^2 < 0$, tedy $(a-v)^2 < 0$, což nemůže nastat, protože druhá mocnina čehokoliv je nezáporná.

V optimálním řešení jsou si tedy obdélníky podobné, platí tedy $v/t_1 = (1-v)/t_2$. Dosazením $t_1 = s_1/v$, $t_2 = s_2/(1-v)$ dostaneme $v^2/s_1 = (1-v)^2/s_2$, neboli $v/\sqrt{s_1} = (1-v)/\sqrt{s_2}$, z čehož už dokážeme v snadno spočítat (protože s_1 a s_2 jsou nějaké konstanty). Vyjádřeno v řeči původní úlohy, poměr rychlosti průjezdu tunelem ku odmocnině z délky tunelu je pro oba tunely stejný.

K čemu nám všechna ta námaha byla, když pořád umíme vyřešit jen případ $n = 2$? Výsledek o rovnosti poměrů se totiž dá zobecnit i pro delší cesty: V optimálním řešení lehké varianty platí, že pro všechny úseky je číslo $v_i/\sqrt{s_i}$ stejné. Jakto? Kdyby tvrzení neplatilo, nutně by nějaké sousední tunely i a $i+1$ musely mít jiné poměry. Podíváme se na úsek cesty (v_i, v_{i+1}), jako by to byla cesta délky dva. V aktuálním řešení na ni máme z celkové kapacity baterie přiděleno $v_i + v_{i+1}$ energie. Když ale v_i a v_{i+1} změníme tak, aby se poměry $v/\sqrt{s}$ vyrovnaly (a přitom součet rychlostí zůstal nezměněný), určitě nezměníme množství spotřebované energie pro celou cestu a celkový čas přitom snížíme. Tím pádem jsme z údajně optimálního řešení získali „ještě optimálnější“, což samozřejmě nejde.

Máme tedy elegantní způsob, jak vyřešit lehčí variantu: Na začátku všechny délky tunelů odmocníme a pak energii tunelům rozdělíme v příslušných poměrech. Konkrétně spočteme $S = \sqrt{s_1} + \cdots + \sqrt{s_n}$ a i-tému tunelu přidělíme $\sqrt{s_i}/S$ energie.



Těžší varianta

Těžší varianta se vlastně bude řešit velmi podobně. Už totiž víme, že libovolnou cestu ze startu do cíle zvládneme nejrychleji projet v čase $\frac{s_1}{v_1} + \cdots + \frac{s_k}{v_k} = \frac{s_1}{\sqrt{s_1}/S} + \cdots + \frac{s_k}{\sqrt{s_k}/S} = S(\sqrt{s_1} + \cdots + \sqrt{s_k}) = S^2$. My chceme najít nejrychlejší cestu do cíle, tedy cestu s nejmenším S^2. Jelikož čím je S větší, tím je i S^2 větší,[13] stačí hledat cestu s nejmenším $S = \sqrt{s_1} + \cdots + \sqrt{s_k}$.

Hledáme tedy cestu v grafu, pro kterou je součet nějakých hodnot na hranách co nejmenší. Na to použijeme starý dobrý Dijkstrův algoritmus. Ještě předtím však hodnoty na hranách odmocníme, protože chceme co nejmenší součet $\sqrt{a_1} + \cdots + \sqrt{a_k}$, nikoliv $a_1 + \cdots + a_k$.

Časová složitost složitost je rovna $\mathcal{O}((N+M)\log N)$, kde N je počet stanic a M počet tunelů, paměťová je $\mathcal{O}(N)$.

Říša Hladík

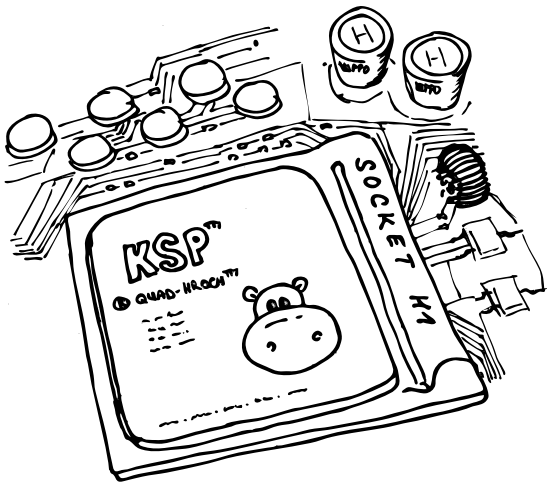
Program (C++):
`http://ksp.mff.cuni.cz/viz/30-2-6.cpp`

30-2-7 Paměť očima assembleru

Úkol 1: znaménkovost load/store instrukcí

V prvním úkolu jsme se zamýšleli, proč některé load/store instrukce mají znaménkové a bezznaménkové varianty, zatímco jiné nikoli.

První věc, kterou je třeba si uvědomit, je, že v registrech nejsou uložená čísla, nýbrž posloupnosti 32 bitů. Pokud registr obsahuje hodnotu 0xFFFFFFFE (11111111 11111111 11111111 11111110), procesor „neví“, jestli tato hodnota představuje znaménkové číslo -2, nebo bezznaménkové číslo 4 294 967 294. Je na nás a našem programu, jakým způsobem obsah registru interpretujeme. Proto i náš simulátor ukazuje u každého registru znaménkovou i bezznaménkovou interpretaci obsahu – nemůže nijak vybrat „tu správnou“.



Zpět k naší otázce. Nejjednodušším případem jsou 32-bitové instrukce `LDR` a `STR`. Ty prostě překopírují 4 bajty z/do paměti bit po bitu a je úplně jedno, co tyto bity znamenají. Hodnota výše se do paměti uloží jako posloupnost bajtů 0xFE, 0xFF, 0xFF, 0xFF (little endian), bez ohledu na znaménkovost.

Teď si představme, že načítáme z paměti např. 8-bitové číslo. Pokud máme v paměti bajt 0xFE, může představovat jak bezznaménkové číslo 254, tak znaménkové číslo -2. V prvním případě ho chceme do registru zapsat jako 0x000000FE, v druhém jako 0xFFFFFFFE (což je 32-bitová reprezentace čísla -2).

Vlastně řešíme následující problém: máme k dispozici např. 8-bitovou reprezentaci nějakého čísla a chceme ji prodloužit na např. 32-bitovou reprezentaci téhož čísla. Ale jak ukazuje příklad výše, toto se dělá rozdílně pro znaménková a bezznaménková čísla. V bezznaménkovém případě stačí prostě hodnotu zleva doplnit nulami.

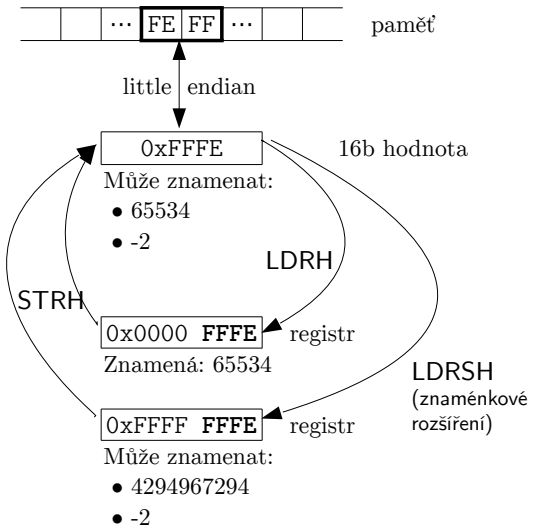
Ve znaménkovém případě musíme provést takzvané *znaménkové rozšíření*. Vezmeme nejvyšší (nejlevější) bit původní reprezentace a jeho hodnotou zleva doplníme reprezentaci na požadovanou délku. A protože nejvyšší bit funguje jako znaménkový, dá se zrovna tak říct, že kladná čísla doplňujeme zleva nulami, záporná jedničkami.

Potřebujeme tedy load instrukcím říct, jaký druh rozšíření na 32 bitů mají použít: proto existují znaménkové a bezznaménkové verze.

[13] Jinými slovy, druhá mocnina je na kladných číslech rostoucí funkce.

Naopak při ukládání tohle není potřeba. Třeba pokud chceme obsah registru uložit jako 8-bitové číslo, prostě vezmeme nejnižších (nejpravějších) 8 bitů a zapíšeme je jako jeden bajt do paměti. Rozmyslete si, že tohle dá správný výsledek pro znaménková i bezznaménková čísla (pokud jsou dost malá, aby se vešla do 8 bitů).

Pro 16-bitové load/store je situace velmi analogická, jen nesmíme zapomenout, že výsledné dva bajty se uloží v pořadí little endian:



Úkol 2: obrácení pole

Postupujeme přímočaře: nejdříve prohodíme první prvek s posledním, potom druhý s předposledním atd., až skončíme uprostřed. Prohození dvou prvků provedeme tak, že je načteme do dvou registrů a potom zapíšeme na opačná místa.

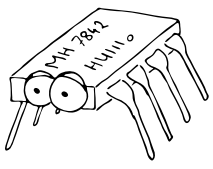
Procházení pole můžeme řešit třeba tak, že si ve dvou registrech budeme udržovat adresu aktuálního levého a pravého prohazovaného prvku. Po prohození levou adresu zvětšíme a pravou zmenšíme. Skončíme, když je pravá adresa menší nebo rovna levé (narazili jsme na prostředek nebo jej překročili).

```
MOV r0, #0x10000
// načti délku do r1 a posuň r0 na začátek pole
LDR r1, [r0], #4
SUB r1, #1 // poslední prvek má index délka-1
ADD r1, r1, r0, lsl #4 // adresa posled. prvku

//odteď ukazují r0,r1 na pár prohazovaných prvků

smycka:
CMP r0, r1
BHS konec
// načti pár k prohození
LDR r2, [r0]
LDR r3, [r1]
// ulož opačně a posuň ukazatele
STR r3, [r0], #4
STR r2, [r1], #-4
B smycka

konec:
```



Úkol 3: nulování paměti

Chceme vynulovat N bajtů paměti pomocí $0.3 \cdot N + \mathcal{O}(1)$ vykonaných instrukcí. Máme k dispozici méně než jednu instrukci na bajt, takže určitě musíme jednou instrukcí vynulovat více bajtů. Nabízí se použít instrukci `STR` pro nulování bajtů po čtveřicích.

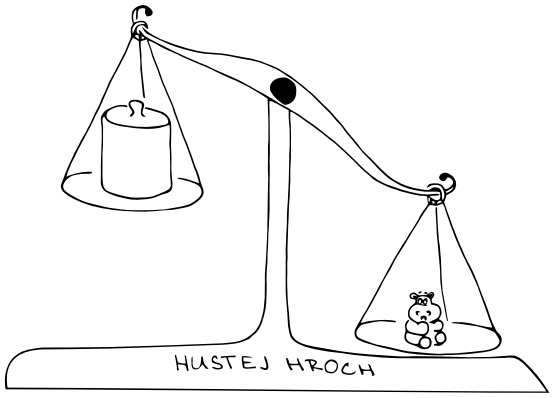
To má dva háčky. V prvé řadě N nemusí být násobkem čtyř, ale o zbylé 1–3 bajty se případně postaráme na konci, to se schová do aditivní konstanty.

```
MOV r1, #0x10000
MOV r2, #0
smycka:
SUBS r0, #4
STRHS r2, [r1], #4
BHI smycka

//Pokud r0 nebylo násobkem 4, v poslední iteraci
//jsme odečetli moc a dostali se do záporu
ADDLO r2, #4

// Teď mohly zbýt nejvýš tři bajty k vynulování
zbytek:
SUBS r0, #1
STRBHS r2, [r1], #1
BHI zbytek
```

V implementaci smyčky jsme použili několik užitečných triků. Odčítání používáme zároveň jako porovnání, čímž ušetříme jednu CMP instrukci. Díky tomu se může stát, že v poslední iteraci odečteme moc a skončíme v záporu, to ale snadno opravíme po skončení smyčky.



Existují dva obyklé způsoby, jak zapisovat smyčky. S podmínkou na začátku:

```
smycka:
<podmíněný skok na 'konec', pokud
  se má smyčka ukončit>
<tělo smyčky>
B smycka
konec:
```

nebo s podmínkou na konci:

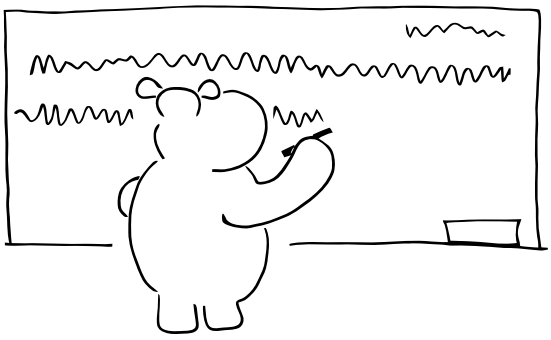
```
smycka:
<tělo smyčky>
<podmíněný skok na 'smycka', pokud
  má smyčka pokračovat>
```

Smyčka s podmínkou na konci se provede vždy alespoň jednou, což se nám tady nehodí, protože `r0` může být méně než 4 a nechceme vynulovat víc paměti než máme. Ale smyčka s podmínkou na začátku obvykle potřebuje dvě instrukce skoku, to je hrozné plýtvání. My jsme využili toho,

že na ARMu jde podmínku připojit k libovolné instrukci a vytvořili tak hybridní smyčku s podmínkou na začátku, která si vystačí s jednou instrukcí skoku.

Ovšem i se všemi těmito optimalizacemi potřebuje naše smyčka tři instrukce na iteraci (při které vynuluje čtyři bajty). Celkem tedy provedeme $\frac{3}{4}N + \mathcal{O}(1)$ instrukcí.

Trávíme více času režijí smyčky než užitečnou prací. Protože režie smyčky je na jednu iteraci konstantní, nabízí se udělat více práce v jedné iteraci. Třeba tak, že použijeme více instrukcí `STR` za sebou.



Rozmysleme si, jestli to pomůže. Pokud v těle smyčky bude k instrukcí `STR`, vynulujeme na jednu iteraci $4k$ bajtů a provedeme $k + 2$ instrukcí. Provedeme tedy $(k + 2)/4k$ instrukcí na vynulování jednoho bajtu. Chceme, aby tento výraz byl nejvýše 0.3. To je jednoduchá nerovnice, vyřešením dostáváme $k \geq 10$.

Výsledný kód bude vypadat následovně:

```
MOV r1, #0x10000
MOV r2, #0
smycka:
SUBS r0, #40
STRHS r2, [r1], #4
STRHS r2, [r1], #4
STRHS r2, [r1], #4
STRHS r2, [r1], #4
STRHS r2, [r1], #4
STRHS r2, [r1], #4
STRHS r2, [r1], #4
STRHS r2, [r1], #4
STRHS r2, [r1], #4
STRHS r2, [r1], #4
STRHS r2, [r1], #4
BHI smycka

// Pokud r0 nebylo násobkem 40, v poslední
// iteraci jsme odečetli moc a dostali se
// do záporu
ADDLO r2, #40

// Teď mohlo zbýt nejvýš 39 bajtů k vynulování
zbytek:
SUBS r0, #1
STRBHS r2, [r1], #1
BHI zbytek
```

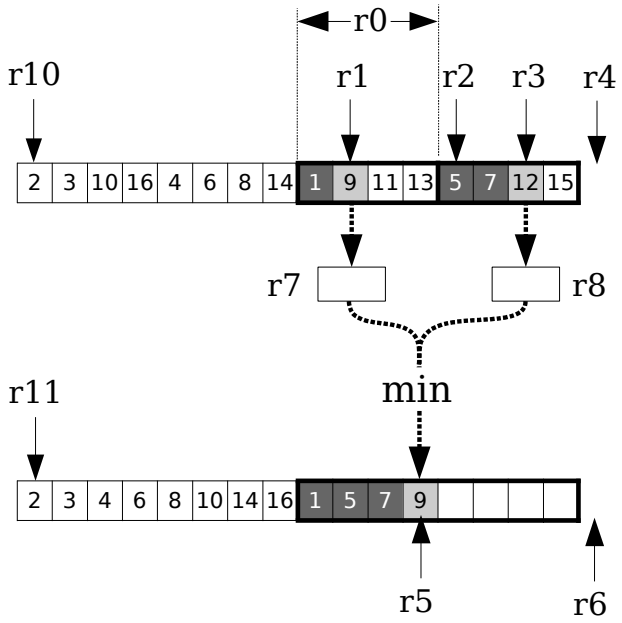
Tato verze pomocí 12 instrukcí vynuluje 40 bajtů, provede tedy $12/40n + \mathcal{O}(1) = 0.3n + \mathcal{O}(1)$ instrukcí, jak jsme chtěli. Použité technice se říká rozbalování smyček (loop unrolling) a běžně se používá ke zrychlení smyček s krátkým tělem. Třeba céčkové překladače takovouto úpravu dělají automaticky v rámci optimalizací.

Úkol 4: třídění

V souladu s radou v zadání budeme implementovat nerekurzivní verzi MergeSortu z třídící kuchařky.[14] Pro jednoduchost budeme předpokládat, že délka posloupnosti je mocninou dvojky ($n = 2^k$). Algoritmus se skládá z k fází. Na začátku i-té fáze máme posloupnost tvořenou řadou bloků délky 2^{i-1}, každý z kterýchž je už z minulých fází setříděný. V i-té fázi slijeme dvojice sousedních bloků do jednoho bloku dvojnásobné délky (2^i). Na konci poslední fáze tvoří celou posloupnost jeden velký setříděný blok délky 2^k, čímž máme hotovo.

Protože slévání nejde jednoduše provádět na místě, potřebujeme mít oddělený prostor pro výsledek, který nám také zadání slibuje. V kuchařce se po každé fázi obsah pomocného výstupního pole zkopíruje zpátky do původního, aby mohl posloužit jako vstup pro další fázi. To je ale zbytečné plýtvání. Místo toho stačí prostě prohodit význam těchto dvou polí. Tedy liché fáze budou používat hlavní pole jako vstup a pomocné pole jako výstup, sudé naopak.

Zbývá to celé přepsat do assembleru. Možná polovinu práce tvoří rozvrhnout si, co si pamatovat ve kterém registru. Zkusme to třeba takto:



- `r0`: Velikost aktuálně slévaného bloku (v bajtech, mocnina dvojky). Dva bloky velikosti `r0` slévame do jednoho velikosti $2 \cdot$ `r0`.
- `r1`: Adresa aktuálního prvku v levém zdrojovém bloku.
- `r2`: Adresa konce levého zdrojového bloku (ukazuje na adresu těsně za koncem, tak je obvyklé konce reprezentovat).
- `r3`: Adresa aktuálního prvku v pravém zdrojovém bloku.
- `r4`: Adresa konce pravého zdrojového bloku.
- `r5`: Adresa aktuálního prvku v cílovém bloku (kam se umístí příští prvek).
- `r6`: Adresa konce cílového bloku.
- `r7`, `r8`: Pomocné registry pro dočasné hodnoty, například prvky načtené z paměti pro porovnání.
- `r9`: Celková velikost pole (v bajtech).
- `r10`: Adresa začátku zdrojového pole (to obsahuje setříděné bloky velikosti `r0`).

[14] `http://ksp.mff.cuni.cz/viz/kucharky/trideni`

- `r11`: Adresa začátku cílového pole (v něm vytváříme setříděné bloky velikosti $2 \cdot$ `r0`). Na konci každé fáze prohodíme hodnoty r10 a r11.

Různá místa v poli si pamatujeme jako paměťové adresy namísto indexů, což trochu zjednodušuje přístup do paměti. Všimněte si, že si u každého bloku pamatujeme aktuální pozici a konec, ale nikoli začátek. Protože začátek ve skutečnosti k ničemu nepotřebujeme. Pro přístup do paměti nám stačí aktuální pozice, při řízení smyčky porovnáváme aktuální pozici s adresou konce.

Základní strukturu programu lze zapsat následujícím pseudokódem. Doporučujeme současně sledovat vzorový program odkazovaný níže – tohle je spíš návod, jak se v něm vyznat, než samostatné čtení. Notace: `=` je přiřazení, `neco:` jsou assemblerové labely, `->` skoky (dle kontextu možná podmíněné), `P:` je *precondition* – něco, co slibujeme, že před vstupem do tohoto místa bude platit.

```
r0 = 4 // slučujeme 1prvkové (4bajtové) bloky
faze:
  P: r0 obsahuje velikost bloků, které chceme
     v této fázi slučovat
  P: bloky velikosti r0 už jsou setříděné
  Nastavíme r1 a r5 na začátek prvního vstupního
    a výstupního bloku.
    r1 = r10
    r5 = r11
  blok:
    // začínáme slévat dva bloky
    P: r1 ukazuje na začátek levého vstup. bloku
       ke zpracování, r5 na začátek příslušného
       výstupního bloku
    // dopočteme pravý vstup. blok a konce bloků
    r2 = r1 + r0
    r3 = r1 + r0
    r4 = r1 + 2*r0
    r6 = r5 + 2*r0
    prvek:
      Pokud jsme na konci levého bloku (r1==r2):
        -> doberpravy
      Analogicky pro pravý: -> doberlevy
      P: r1 < r2, r3 < r4, r5 < r6
         (v žádném bloku nejsme na konci)
      Přidáme další prvek na výstup: na adresu
       [r5] uložíme menší z [r1], [r3] a správně
       posuneme ukazatele (viz níže)
      -> prvek
    doberlevy:
      P: r3 == r4 // (pravý blok vyprázdněn)
      Zkopíruj zbytek levého bloku (od adresy
        r1 do r2 - 4) na výstup (r5 až r6 - 4)
        -> konecmerge
      doberpravy:
        (analogicky)
      konecmerge:
        P: ve všech blocích jsme na konci
           (r1 == r2, r3 == r4, r5 == r6)
        Pokud jsme zpracovali poslední blok
        (r5 >= r11+r9):
          -> konecfaze
        Jinak se posuneme na následující blok:
          // Nový levý blok bude začínat za koncem
          // aktuálního pravého.
          r1 = r4
          // r5 netřeba měnit, už ukazuje na
          // začátek sousedního bloku
          -> blok
    konecfaze:
      r0 = r0 * 2
      prohoď r10, r11
      -> faze
```

Jeden krok slévání (label `prvek:`) je jednoduchý. Načteme čísla z aktuálních pozic v obou vstupních blocích, porovnáme je a uložíme na výstup. Zároveň musíme posunout výstupní ukazatel a jeden ze vstupních (podle toho, které číslo jsme vybrali).

```
prvek:
// Pokud jsme v některém bloku na konci...
CMP r1, r2
BHS doberpravy // (viz vzorový program)
CMP r3, r4
BHS doberlevy

LDR r7, [r1]
LDR r8, [r3]

CMP r7, r8
// Pokud akt. prvek z levého bloku je menší,
// přidáme ho na výstup a posuneme levý pointer
STRLE r7, [r5], #4
ADDLE r1, #4
// Analogicky pro pravý
STRGT r8, [r5], #4
ADDGT r3, #4

B prvek
```

Zbytek programu najdete na našem webu.

Program (assembler):
`http://ksp.mff.cuni.cz/viz/30-2-7-mergesort.asm`

Úkol 5: zatřídění do spojového seznamu

Na adrese 0x100000 je uložen pointer na začátek seznamu, v r0 pointer na nově přidávaný prvek. Předpokládáme, že struktura pro jeden prvek obsahuje 32-bitovou hodnotu a 32-bitový ukazatel na následníka.

Úkol je celkem jednoduchý, jen je třeba rozmyslet si několik okrajových případů:

- Zatřiďujeme na začátek seznamu (pak je třeba přepsat ukazatel na začátek).
- Zatřiďujeme na konec seznamu (je třeba dát si pozor, ať nezkoušíme dereferencovat null pointer).
- Seznam je prázdný (ukazatel na začátek je null).

Opět si rozvrhneme registry:

- r0 – adresa vkládaného prvku
- r1 – hodnota vkládaného prvku
- r2 – adresa aktuálního prvku seznamu, se kterým porovnáváme
- r3 – hodnota tohoto prvku
- r4 – adresa prvku předcházejícího r2
- r10 – konstanta 0x100000 (adresa pointeru na první prvek)

Samotný kód už je potom velmi přímočarý a asi nepotřebuje další vysvětlení:

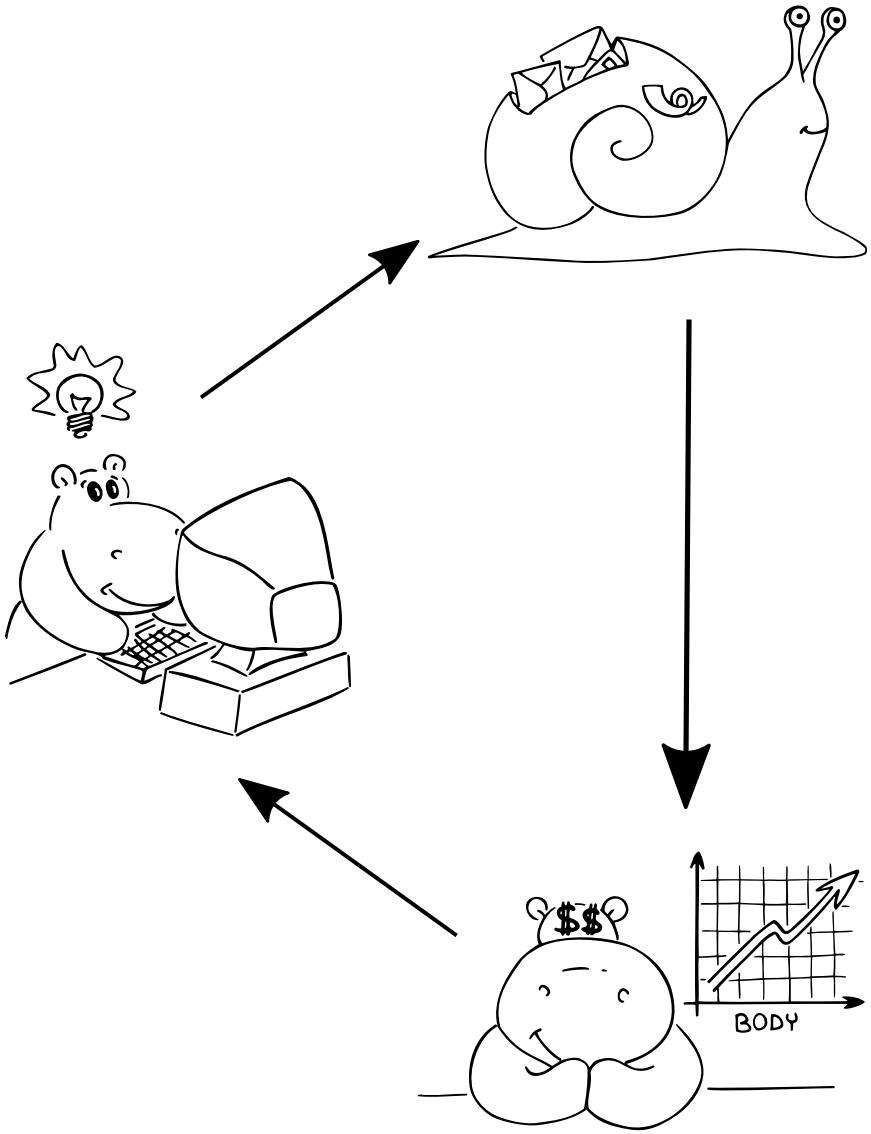
```
LDR r1, [r0]
MOV r10, #0x100000
LDR r2, [r10]
MOV r4, #0

smycka:
CMP r2, #0
BEQ vloz
LDR r3, [r2]
CMP r1, r3
BLE vloz
MOV r4, r2
LDR r2, [r2, #4] // r2 = [r2].dalsi
B smycka

vloz:
// Vlož nový prvek mezi prvky na adr. r4 a r2
// Pokud vkládáme na začátek seznamu, r4 == 0.
// Pokud vkládáme na konec seznamu, r2 == 0.

STR r2, [r0, #4] // [r0].dalsi
CMP r4, #0
STREQ r0, [r10] // r0 je nový první prvek
STRNE r0, [r4, #4]  // [r4].dalsi = r0
```

Filip Štědronský



Výsledková listina druhé série třicátého ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*2-1*	*2-2*	*2-3*	*2-4*	*2-5*	*2-6*	*2-7*	*série*	*celkem*
0.					11	13	10	10	12	12	15	63,0	125,0
1.	Josef Minařík	GJarošeBO	3	2	11	13	2,5	10	11	12	11	61,4	121,3
2.	Veronika Nechaieva	KyivLyceum	4	2	5	8	2	10	3,5	10,5	13,5	55,4	111,8
3.	Jiří Škrobánek	G Wicht	4	2	6,5	12	2	10	8	4,5	13,5	57,1	109,4
4.	Michal Kodad	SPŠSmíchov	2	10	6	8		10	3		12	40,4	92,7
5.	Jan Černý	BiGy Žďár	2	2	6,5	8	2	10	3,5	1,5	7	48,2	91,9
6.	Klára Tauchmanová	GOhradníPH	4	3	10	8	2,5	10	3,5		4,5	46,0	91,2
7.	Petr Zahradník	GaSOŠ ÚL	3	2		8		10	8	3,5	8,5	50,7	85,6
8.	Jan Kaifer	GKepleraPH	2	8	10	3	2	10				27,0	82,0
9.	Matěj Kripner	GEBenešeKL	3	2	11		9	10				30,7	77,5
10.	Daniel Skýpala	GTomkovaOL	0	5	10,5	3	1	10	3		3,5	36,8	72,5
11.	Jáchym Mierva	BiGy Žďár	1	2	6	8	3	10	3,5			42,4	70,6
12.	Ondřej Bleha	GBNěmcovHK	3	2	11	2		10				25,5	68,8
13.	Jakub Komárek	GUHradiště	3	2			5,5	10			14,5	33,0	67,5
14.	Martin Kurečka	GJarošeBO	4	4				10				10,0	66,5
15.	Vladimír Chudý	G Chrudim	1	2	4		1	10	3	11		37,2	65,3
16.	Václav Pavlíček	SPSEPard	2	9	10		9	7	4		3	35,5	64,6
17.	Pavel Hudec	G JGJ PH	4	1	11	11		10	12	12		57,5	57,5
18.	Adam Dejl	G JGJ PH	4	2	6			10		3,5	8	37,6	54,2
19.	Michal Zaslavský	GKepleraPH	3	2	4		0	10	3			23,1	52,3
20.	Tomáš Černý	GArabskáPH	2	1	4	8	6	7	3,5		10	49,8	49,8
21.	Jiří Kvapil	GTomkovaOL	0	2	1	7	0,5	10	2			28,5	43,5
22.	František Kmječ	G Brandýs	2	7	11	8		7				28,6	42,0
23.	Filip Geib	G MMH LM	4	6								0,0	41,3
24.	Jakub Pelc	G UherBrod	4	12				10			14	23,9	38,9
25.	Martin Zimen	GJMasarJI	3	1								0,0	33,9
26.	Jiří Löffelmann	GLitoměřPH	4	9				10				10,0	31,5
27.	Vojtěch Michal	GNVPláníPH	3	2							12,5	14,3	29,3
28.	Ondřej Gonzor	G Brandýs	1	6	6		6	4				21,0	27,1
29.	Vít Skalický	GPísnickáPH	0	3				10				10,0	25,1
30.	Tomáš Strnad	GŽamberk	4	1								0,0	23,0
31.	Tomáš Sládek	GJHroncaBA	2	2				7				9,0	19,9
32.	Lucia Krajčoviechová	GJHroncaBA	2	1								0,0	19,8
33.	Miroslav Hrabal	GTomkovaOL	4	6								0,0	18,5
34.–35.	Zuzana Urbanová	GFXŠaldyLI	4	2								0,0	15,0
	Ondřej Wrzecionko	GTěš	3	1								0,0	15,0
36.	Filip Masár	PiarGNitra	4	3								0,0	14,7
37.	Eliška Vlčinská	GHladnov	3	3								0,0	14,6
38.	Jakub Růžička	GNymburk	3	1								0,0	13,4
39.	Jakub Jirkal	GJungmanLT	3	2								0,0	13,2
40.	Michal Tomek	GHumpolec	4	1								0,0	12,3
41.	Vojtěch Zabořil	GTurnov	1	1	1			4				9,7	9,7
42.	Andrej Ohrablo	GJHroncaBA	2	1				7				9,1	9,1
43.	Karel Balej	GRokycany	3	3								0,0	9,0
44.	Václav Zvoníček	GJarošeBO	2	1								0,0	7,5
45.	Prokop Randáček	GFXŠaldyLI	−1	1	4		0					7,4	7,4
46.	Viktor Fukala	GKepleraPH	1	4								0,0	6,8
47.	Lenka Vincenová	GTomkovaOL	4	1								0,0	4,0
48.	Lukáš Caha	GZborovPH	4	5								0,0	3,8
49.	Jakub Ucháč	ŠMaVVzt	2	1								0,0	2,7
50.–51.	Vojtěch Březina	GCoubTábor	1	1								0,0	1,3
	Vojtěch Káně	G Brandýs	2	1								0,0	1,3

matfyz

KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.

Webové stránky: *https://ksp.mff.cuni.cz/*

E-mail: *ksp@mff.cuni.cz*

Diskusní fórum: *https://ksp.mff.cuni.cz/forum/*

Chcete-li s námi komunikovat bezpečně, můžete si ověřit náš HTTPS certifikát – jeho SHA1 fingerprint je: `E9:DB:EE:C6:62:BC:14:DE:09:E4:E8:97:DC:36:0E:87:B3:50:B0:01`.