Korespondenční Seminář z Programování

35. ročník **KSP** **Listopad 2022**

Milí řešitelé, řešitelky a řešitelčata!

Dostává se k vám druhé číslo hlavní kategorie 35. ročníku KSP.

Letos se můžete těšit v každé z pěti sérií hlavní kategorie na **4 normální úlohy**, z toho alespoň jednu praktickou opendatovou. Také na **kuchařky** obsahující nějaká zajímavá informatická témata, hodící se k úlohám dané série. Občas se nám také objeví **bonusová X-ková úloha**, za kterou lze získat X-kové body. Kromě toho bude součástí sérií **seriál** na lineární algebru.

Autorská řešení úloh budeme vystavovat hned po skončení série. Pokud nás pak při opravování napadnou nějaké komentáře k řešením od vás, zveřejníme je dodatečně.

Odměny & na Matfyz bez přijímaček

Za úspěšné řešení KSP můžete být **přijati na MFF UK bez přijímacích zkoušek.** Úspěšným řešitelem se stává ten, kdo získá za celý ročník (hlavní kategorie) alespoň 50 % bodů. Za letošní rok půjde získat maximálně 300 bodů, takže hranice pro úspěšné řešitele je 150. Maturanti pozor, pokud chcete prominutí využít letos, musíte to stihnout do konce čtvrté série, pátá už bude moc pozdě. Také každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok, nálepku na notebook a možná i další překvapení.

Termín série: **18. prosince 2022 ve 32:00 (tedy další ráno v 8:00)**

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

Značky úloh: Lehčí úloha (či její část) vhodná pro začátečníky; Praktická open-data úloha; Úloha, u které doporučujeme začíst se do kuchařky; Seriálová úloha

Odměna série: **Sladkou odměnu** si vysloužíš ten, kdo **získá z této série alespoň 42 bodů**.

Druhá série třicátého pátého ročníku KSP

35-2-1 Kde je Dan? — 11 bodů

Na soustředění KSP se hrála seznamovací hra, při které stálo několik účastníků a organizátorů v kroužku. Ten, kdo byl na řadě, se podíval na lidi na druhé straně kroužku a řekl jejich jména. Už si bohužel nikdo nepamatuje, kde kdo z účastníků stál, ale víme, na kterých pozicích stál někdo z organizátorů. Zároveň si také pamatujeme, kolik organizátorů pojmenoval Dan, když byl na řadě. Zajímalo by nás, kde všude mohl Dan stát, když předpokládáme, že pojmenoval všechny lidi správně.

V kroužku stálo N lidí a M z nich bylo organizátorů. Pozice v kroužku jsou očíslovány po směru hodinových ručiček čísly 1 až N tak, že pozice i a $i+1$ jsou vedle sebe, a zároveň pozice 1 je vedle pozice N. Člověk na tahu vždy pojmenovával K lidí, kteří stáli přímo proti němu.

Slibujeme, že:

- $M < N$ – organizátoři tvoří výraznou menšinu ze všech lidí
- N bude vždy sudé
- K bude vždy liché
- a také $K < N$

Tedy lidé naproti Danovi jsou vždy jednoznačně určeni – je to ten přímo naproti Danovi (tedy takový člověk, že jejich pozice se liší právě o $N/2$) a dále $(K-1)/2$ lidí napravo i nalevo od tohoto člověka.

Víme, že mezi těmi K lidmi, kteří stáli naproti Danovi, bylo X organizátorů.

Na kolika možných pozicích mohl Dan stát? Už si ani nepamatujeme, jestli jsme Dana počítali mezi organizátory nebo ne, takže berte v potaz obě možnosti.

Počet lidí v kroužku (a tedy i výsledných možností) může být opravdu velký, a proto bude v některých jazycích zapotřebí použít 64-bitového datového typu (v C++ je to `long long int`). Pokud programujete v Pythonu, pak toto řešit nemusíte.

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 jsou čísla N, M, K a X oddělená mezerou. Na druhém řádku následuje M čísel popisujících, na kterých pozicích stáli organizátoři. Indexujeme od jedné.

Formát výstupu: Jediné číslo, a to počet pozic, na kterých mohl Dan stát, aby naproti němu bylo X organizátorů.

Ukázkový vstup:	*Ukázkový výstup:*
`6 3 3 2` `2 6 3`	`3`

35-2-2 Zápisky z přednášky — 12 bodů

Jirka jel přednášet na Krutou Smršť Přednášek (viz `https://ksp.mff.cuni.cz/viz/smrst`, jste také zvá-

ni!). Povídal o datové struktuře jménem binární halda (mezi přáteli občas jen halda). Ta reprezentuje nějakou množinu čísel a umožňuje nám vkládat do ní nové číslo, ptát se na nejmenší číslo v ní a následně ho odebrat.

Haldu si můžeme přestavit jako zakořeněný strom. V každém vrcholu je uložené jedno číslo, přičemž platí, že je vždy menší nebo stejně velké než čísla v jeho potomcích. Jedná se o binární vyvážený strom, tedy všechny vrcholy mají nejvýše dva potomky, a ti, co mají méně než dva, leží v nejhlubší nebo druhé nejhlubší hladině.

Haldu můžeme snadno reprezentovat pomocí jednoho pole. Její kořen leží na indexu 1 a potomci vrcholu na indexu i leží na indexech $2i$ a $2i + 1$. Pokud jste ještě Jirkovu přednášku neabsolvovali, můžete se více o haldách dozvědět v naší kuchařce.[1]

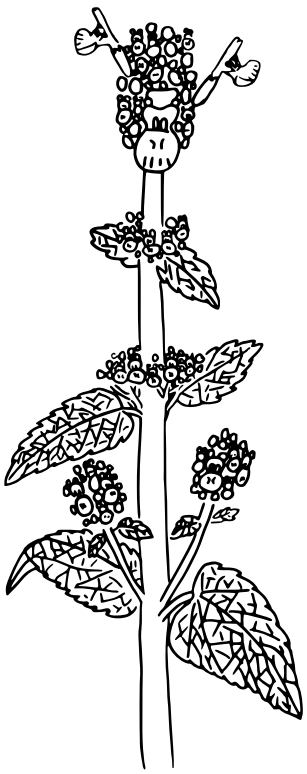
Dan Jirkovu přednášku poctivě absolvoval, a dokonce si z ní dělal poznámky toho, co bylo na tabuli. Má tam nakreslenou haldu, na které Jirka následně ukazoval operaci nalezení a odstranění nejmenšího čísla. Na přednášce Jirka opakoval tuto operaci několikrát po sobě a Dan si zapisoval, jaké číslo při každé operaci odebral. Ovšem nyní se Dan dívá na zápisky a nemůže přečíst K-té odebrané číslo.

Pomozte Danovi pouze za pomocí původní haldy zadané v poli zjistit, které číslo bylo odebráno jako K-té. Můžete předpokládat, že K je řádově menší než počet prvků původní haldy N. Složitost svých algoritmů tedy můžete odhadovat nejenom pomocí N, ale i K. Můžete předpokládat, že pole reprezentující haldu již máte načtené v paměti.

Toto je teoretická úloha. Není nutné ji programovat, odevzdává se pouze slovní popis algoritmu. Více informací zde: `http://ksp.mff.cuni.cz/viz/tinfo`

35-2-3 Hroší šanta — 10 bodů

Kubovi se na botanické burze podařilo sehnat vzácnou šantu hroší (Nepeta hippopotamis). Traduje se, že odvar z jejích listů pomáhá zklidnit mysl a odbourává stres.



Kuba ví, že taková hroší šanta se má správně zalévat právě K litry vody. On má ale k dispozici pouze dva džbány s kapacitami A a B litrů. Pro libovolný z těchto džbánů Kuba může v jednu chvíli provést jednu z následujících akcí:

- naplnit jej až po okraj,
- celý ho vylít,
- přelít ho do druhého džbánu, dokud druhý džbán nebude plný, nebo dokud v prvním nedojde voda.

Poraďte Kubovi, jak má vodu přelévat, aby mu zbylo K litrů vody v jedné z nádob. Kuba by se ale nerad moc nadřel, proto najděte nejkratší posloupnost akcí, která tohle splňuje. Slibujeme, že taková posloupnost existuje. 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 jsou dvě čísla A, B oddělená mezerou, představující kapacitu obou džbánů. Slibujeme, že $A \neq B$. Na druhém řádku se nachází číslo K s cílovým objemem vody v jedné z nádob.

Formát výstupu: Na první řádek vypište počet akcí, potom vypište popis jednotlivých akcí. Každou akci vypište jako jeden ze znaků `AaBb<>`: `A` znamená naplnění prvního zadaného džbánu vodou, `B` je naplnění toho druhého. Znak `a` je pokyn k vylití prvního džbánu, `b` k vylití druhého. Znak `>` přelije první džbán do druhého a `<` naopak.

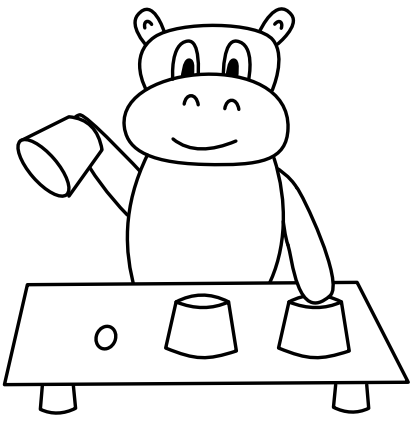
Výstup můžete jakoliv odřádkovávat a prokládat mezerami. (Avšak na prvním řádku vypište jen počet akcí.)

Ukázkový vstup:	*Ukázkový výstup:*

`4 11`
`5`
	`10`
	`A >A> A`
	`> b`
	`>A>`

35-2-4 Kamínky — 12 bodů

Tři dobrodruhové, Bojovník Khebir, zloděj Srunvor a kouzelnice Preslana, se po porážce draka vrátili do své oblíbené hospody U Hrocha, aby doplnili síly.



Khebir uviděl v rohu tajemného poutníka v kápi, kolem kterého se shromáždila kupa vesničanů. S vesničany hrál následující hru: Tajemný poutník nejprve naskládá barevné kamínky do řady. Poté vesničan může vždy zvolit dvojici stejně barevných kamínků vedle sebe, které tajemný poutník vrátí zpátky do pytlíku. Vesničan volí dvojice, dokud

[1] `http://ksp.mff.cuni.cz/viz/kucharky/haldy-a-cesty`

buď nejsou všechny kamínky v pytlíku, v takovém případě vyhraje vesničan, nebo neexistuje dvojice sousedních kamínků se stejnou barvou, v takovém případě vyhrává poutník.

Khebir se už už chtěl vsadit a pustit do hry, jenže Preslana ho zastavila, protože si myslí, že Khebir nemůže vyhrát. Pomůžete jejich spor rozseknout?

Pro zadanou posloupnost barevných kamínků najděte algoritmus, který nám řekne, zdali vyhrát lze, nebo ne, a případně jak.

Například pro řadu kamínků *červený, modrý, modrý a červený* stačí nejprve vybrat dva modré vedle sebe a poté dva červené, které se k sobě nově dostanou, a hru tak vyhrát lze. Ale naopak pro řadu *zelený, červený, modrý, modrý, zelený, červený* sice můžeme na začátku vybrat dva modré kamínky vedle sebe, ale ani po jejich odstranění nevznikne žádná další stejně barevná dvojice vedle sebe a hru tak prohrajeme.

Toto je teoretická úloha. Není nutné ji programovat, odevzdává se pouze slovní popis algoritmu. Více informací zde: `http://ksp.mff.cuni.cz/viz/tinfo`

35-2-X1 Oblázky — 10 bodů

Toto je bonusová úloha pro zkušenější řešitele, těžší než ostatní úlohy v této sérii. Nezískáte za ni klasické body, nýbrž dobrý pocit, že jste zdolali něco výjimečného. Kromě toho za správné řešení dostanete speciální odměnu a body se vám započítají do samostatných výsledků KSP-X.

Následující úloha navazuje na úlohu 35-2-4

Preslana obvinila tajemného poutníka z toho, že kamínky jsou rozdané tak, že vždy vyhraje tajemný poutník. Tajemný poutník se proto rozhodl, že změní pravidla – místo toho, aby se vybíraly dvojice stejně barevných sousedních kamínků, nyní lze vybrat dvojice nebo trojice stejně barevných sousedních kamínků. Preslanu by zajímalo, jestli jde hru vyhrát s novými pravidly. Pomůžete jí s tím?

Pro zadanou posloupnost barevných kamínků najděte algoritmus, který nám řekne, zdali vyhrát lze, nebo ne, a případně jak.

35-2-S Lineární zobrazení — 15 bodů

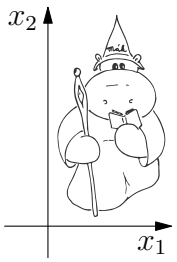
Právě čtete druhý díl seriálu. Pokud jste neřešili ten první, pro pochopení následujících odstavců je vhodné si jej přinejmenším přečíst. Úlohy z něj je stále možné odevzdávat za polovinu bodů.

V tomto dílu se podíváme na lineární zobrazení, která nám umožní matematicky popsat transformace obrázků. Úvahy o nich nás posléze zavedou k maticím.

Všechny úlohy z tohoto seriálu odevzdávejte dohromady v jednom zazipovaném archivu. Termín odevzdání je shodný s termínem pro druhou sérii. Poté lze odevzdávat za snížený počet bodů až do konce ročníku.

Zobrazení

Budeme zkoumat transformace následujícího obrázku:



Můžeme si představit, že se obrázek skládá z mnoha bodů. Každý z těchto bodů budeme interpretovat jako vektor. Chceme-li popsat určitou transformaci obrázku, musíme říci, kam se každý z těchto vektorů má přesunout.

Zobrazení pro nás bude funkce f, která jako argument dostane vektor $\mathbf{x}$ v původním obrázku a převede jej na vektor $f(\mathbf{x})$ v transformovaném obrázku.

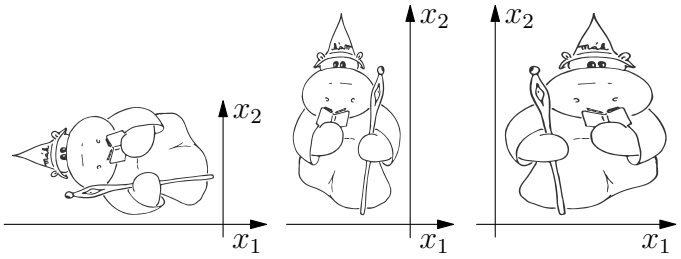
Takových zobrazení existuje fůra, většina z nich obrázek zdeformuje k nepoznání. Nás budou zajímat jen některá z nich, takzvaná *lineární zobrazení.*

V minulém dílu jsme nahlédli, že prostor $\mathbb{R}^d$ je uzavřený na součet a násobení skalárem. Od lineárního zobrazení budeme požadovat něco podobného, konkrétně budeme chtít, aby pro všechny vektory $\mathbf{x}$, $\mathbf{y}$ a skaláry a platilo:

$$f(\mathbf{x}+\mathbf{y}) = f(\mathbf{x}) + f(\mathbf{y})$$

$$f(a\mathbf{x}) = af(\mathbf{x})$$

Tyto vlastnosti splňují zobrazení, která obrázek otáčí, zrcadlově převrací nebo v některých směrech roztahují, případně kombinují více těchto operací. Podívejte se na následující příklady zobrazení a rozmyslete si, že tomu tak skutečně je.



Úkol 1– Nelineární zobrazení [2b]:

Najděte příklad zobrazení v $\mathbb{R}^2$, které není lineární. Ideálně takové, jehož chování lze popsat slovy, nikoliv pouze funkčním předpisem. Nelinearitu ukažte na konkrétních dvou vektorech, respektive vektoru a skaláru.

Báze

Nyní na chvíli odbočme a vraťme se k minulému dílu, kde jsme zkoumali generátory. Zjistili jsme, že když jsou lineárně závislé, pak lze některé z nich odebrat a jejich lineární obal se nezmění. To je občas nepraktické, proto od nich mnohdy nezávislost budeme požadovat.

Jsou-li generátory prostoru U lineárně nezávislé, řekneme, že tvoří *bázi* prostoru U. Poté lze každý vektor $\mathbf{v} \in U$ vyjádřit jako lineární kombinaci bázických vektorů díky tomu, že generují prostor U. Tato lineární kombinace je navíc jednoznačná, neboť jsou lineárně nezávislé.

Jednu možnou bázi prostoru $\mathbb{R}^d$ tvoří jednotkové vektory, které jsme již zmínili minule. Teď se nám bude hodit zavést pro ně značení: Jednotkový vektor, který má jedničku na i-té pozici, budeme značit $\mathbf{e}_i$. Dimenzi vektoru nijak neudáváme, odvodíme ji z kontextu. Například v $\mathbb{R}^3$ by byl vektor $\mathbf{e}_2$ roven $(0,1,0)^\top$.

Báze složená z jednotkových vektorů se nazývá *kanonická*. Vyjádřit vektor $\mathbf{v} = (v_1, \ldots, v_n)^\top$ pomocí této báze je jednoduché:

$$\mathbf{v} = \sum_{i=1}^{n} v_i \mathbf{e}_i$$

Popis zobrazení

Zpět k lineárním zobrazením. Naším cílem je popsat zobrazení f co nejjednodušším způsobem. Vypisovat obrazy $f(\mathbf{x})$ pro každý možný vzor $\mathbf{x}$ totiž moc zábavně nezní. Zobrazení ovšem nemůže být úplně libovolné, linearita jej dost omezuje. Mělo by tedy stačit uvést obrazy jen pro pár vektorů, zbytek pak již bude jednoznačně určen.

Asi není překvapením, že si jako vzory zvolíme generátory prostoru, který transformujeme. A protože chceme mít vzorů co nejméně, zvolíme si je tak, aby byly lineárně nezávislé. Hle, báze!

Jak jsme nahlédli výše, každý vektor $\mathbf{v}$ lze vyjádřit jako lineární kombinaci bázických vektorů. Pokud budeme znát jejich obrazy, zvládneme díky linearitě spočítat také obraz $\mathbf{v}$:

$$f(\mathbf{v}) = f\left(\sum_i a_i \mathbf{x}_i\right) = \sum_i a_i f(\mathbf{x}_i)$$

Jinými slovy, lineární zobrazení je jednoznačně určeno tím, kam se zobrazí bázické vektory.

Matice

Zatímco vektor je seznam čísel, *matice* je tabulka čísel. Lze na ni také pohlížet jako na seznam vektorů.

Matice využijeme k zápisu lineárního zobrazení. Vezmeme vektory kanonické báze, tedy jednotkové vektory, a jejich obrazy zapíšeme do sloupců matice.

$$\begin{pmatrix} | & | & & | \\ f(\mathbf{e}_1) & f(\mathbf{e}_2) & \cdots & f(\mathbf{e}_n) \\ | & | & & | \end{pmatrix}$$

Matice pro transformace obrázků by vypadaly následovně:

$$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \quad \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$$

otočení zrcadlení roztažení

Zkuste vymyslet, jak by vypadala matice, která obrázek otočí o úhel α proti směru hodinových ručiček. Řešení najdete na konci textu.

Označme $\mathbf{A}$ matici lineárního zobrazení f. Rádi bychom nadefinovali součin matice a vektoru tak, abychom mohli psát $f(\mathbf{v}) = \mathbf{A}\mathbf{v}$. Již víme, jak zapsat vektor $\mathbf{v}$ pomocí kanonické báze. Také umíme vyjádřit obraz vektoru pomocí obrazů bázických vektorů. Můžeme tedy spočítat, jak by měl vypadat obraz vektoru $\mathbf{v}$:

$$f(\mathbf{v}) = \sum_{i=1}^{n} v_i f(\mathbf{e}_i)$$

v_i je složka vektoru $\mathbf{v}$, $f(\mathbf{e}_i)$ je zase sloupec matice $\mathbf{A}$. Součin matice s vektorem spočítáme tedy tak, že vynásobíme první složku vektoru s prvním sloupcem matice, k tomu přičteme součin druhé složky s druhým sloupcem a tak dále.

Skládání zobrazení

Nyní uvažme dvě zobrazení f, g. Zobrazení f reprezentujeme maticí $\mathbf{A}$, g zase maticí $\mathbf{B}$. Zajímalo by nás, jak se chová složené zobrazení, kde nejprve aplikujeme f a poté g. Transformací vektoru $\mathbf{v}$ získáme $g(f(\mathbf{v})) = \mathbf{B}(\mathbf{A}\mathbf{v})$. Nešlo by totéž zapsat jako $(\mathbf{B}\mathbf{A})\mathbf{v}$? Tedy nejprve spočítat jednu matici, která popíše složené zobrazení?

Pojďme vymyslet, jak by taková matice měla vypadat. Ve sloupcích by měla mít obrazy bázických vektorů. Matice $\mathbf{A}$ již obsahuje jejich obrazy podle f. Matice $\mathbf{B}$ umí jakýkoli vektor zobrazit podle g. Necháme-li tedy maticí $\mathbf{B}$ zobrazit sloupce $\mathbf{A}$, získáme hledanou matici.

Tímto jsme tedy odvodili, jak by mělo vypadat maticové násobení. Sloupec matice $\mathbf{BA}$ získáme jako lineární kombinaci sloupců z $\mathbf{B}$, kde jako koeficienty použijeme složky příslušného sloupce z $\mathbf{A}$.

Následující schéma nabízí ještě jiný pohled. Vlevo je matice $\mathbf{B}$, nahoře $\mathbf{A}$, vpravo dole jejich součin $\mathbf{BA}$. Jednu složku součinu získáme jako skalární součin příslušného řádku $\mathbf{B}$ a sloupce $\mathbf{A}$.

$$\begin{pmatrix} \cdot & \cdot & \cdot \\ 1 & 2 & 3 \\ \cdot & \cdot & \cdot \end{pmatrix} \begin{pmatrix} \cdot & \cdot & 1 \\ \cdot & \cdot & 2 \\ \cdot & \cdot & 3 \end{pmatrix} \begin{pmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & 14 \\ \cdot & \cdot & \cdot \end{pmatrix}$$

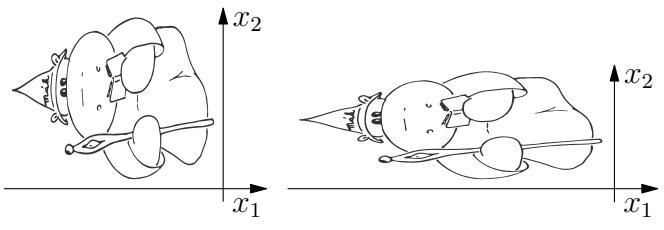
Maticí $m \times n$ myslíme matici s m řádky a n sloupci. Aby měl maticový součin $\mathbf{AB}$ smysl, musí být $\mathbf{A} \in \mathbb{R}^{m \times p}$ a $\mathbf{B} \in \mathbb{R}^{p \times n}$, jinými slovy $\mathbf{A}$ musí mít tolik sloupců, co $\mathbf{B}$ řádků. Důvod je patrný ze schématu výše, oba vektory ve skalárním součinu musí mít stejnou délku.

Ještě pojďme zapsat toto schéma matematicky. Označme a_{ij} složku matice $\mathbf{A} \in \mathbb{R}^{m \times p}$ v i-tém řádku a j-tém sloupci, obdobně b_{ij} složku $\mathbf{B} \in \mathbb{R}^{p \times n}$. Poté složku matice $\mathbf{AB}$ spočteme takto:

$$(\mathbf{AB})_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}$$

Vlastnosti maticového násobení

Pozor na to, že maticové násobení není komutativní, tedy $\mathbf{AB}$ není nutně rovno $\mathbf{BA}$. Jednak se může stát, že po prohození nebudou mít matice vhodné rozměry. Ovšem i pokud ano, může být výsledek jiný. Pokud obrázek roztáhneme ve vodorovném směru a poté otočíme, získáme jiný obrázek, než když tyto operace provedeme v opačném pořadí. Zkuste roznásobit příslušné matice a ověřit početně.



Při skládání tří a více zobrazení ovšem nezáleží na pořadí, v jakém skládáme. Maticové násobení je tedy asociativní: $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$. Jak to dokázat?

Výše jsme uvedli vzorec pro výpočet $(\mathbf{AB})_{ij}$. Zkuste podobným způsobem vyjádřit složku $((\mathbf{AB})\mathbf{C})_{ij}$ a $(\mathbf{A}(\mathbf{BC}))_{ij}$. Obě vyjádření by se měla rovnat. Nebudete-li si vědět rady, přelistujte na konec textu.

Také platí $\mathbf{A}(\mathbf{B}+\mathbf{C}) = \mathbf{AB}+\mathbf{AC}$, přičemž sčítání matic probíhá podobně jako u vektorů jednoduše po složkách. Důkaz je velmi podobný důkazu asociativity.

Zajímavou maticí je *jednotková matice*, která má ve sloupcích jednotkové vektory. Jednotkovou matici značíme $\mathbf{E}$, případně $\mathbf{E}_n$, chceme-li vyjádřit její velikost. Protože zobrazuje vektory kanonické báze na sebe samé, násobení jednotkovou maticí se chová invariantně: $\mathbf{EA} = \mathbf{A}$. Dokonce i naopak, protože jednotkové vektory se zobrazí na sloupce matice $\mathbf{A}$: $\mathbf{AE} = \mathbf{A}$.

$$\mathbf{E}_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Násobení matic $n \times n$ podle definice má časovou složitost $\Theta(n^3)$. Existují však sofistikované algoritmy, které to zvládají rychleji. Za zmínku stojí Strassenův algoritmus. Je založený na přístupu *Rozděl a panuj* a zvládne vynásobit matice pomocí 7 součinů matic velikosti $n/2 \times n/2$. To vede ke složitosti $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$. Tento algoritmus se ovšem vyplatí jen pro větší matice. Zajímá-li vás, jak přesně funguje, nahlédněte do kapitoly 10.5 v Průvodci.[2]

Využití maticového násobení

Matice se hodí i v algoritmech, které s transformacemi obrázků nikterak nesouvisí. Nejprve se podíváme na výpočet Fibonacciho čísel. To je posloupnost s prvními členy $F_0 = 0, F_1 = 1$ a rekurentním vztahem $F_{n+2} = F_n + F_{n+1}$, dostáváme tedy $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$

Naším cílem bude sestavit matici, která nám vyrobí další člen v posloupnosti. Potřebujeme tedy vektorovou reprezentaci členu F_n. Protože se vždy sčítají dva členy, budeme si pamatovat i F_{n+1}. Naše reprezentace čísla F_n tedy bude vektor:

$$\mathbf{f}_n = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}$$

Hledaná matice $\mathbf{Q}$ musí splňovat:

$$\mathbf{Q}\mathbf{f}_n = \mathbf{Q}\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \mathbf{f}_{n+1} = \begin{pmatrix} F_{n+1} \\ F_{n+2} \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_n + F_{n+1} \end{pmatrix}$$

Úkol 2– Fibonacciho čísla [5b]:

Sestavte matici $\mathbf{Q}$. Poté napište algoritmus, který dostane zadané číslo n a spočítá F_n. Napovíme, že to jde lépe než v lineárním čase. Můžete použít svůj oblíbený programovací jazyk, ale postačí i pseudokód.

Ještě zmiňme jedno využití v grafových algoritmech. Mějme graf s n vrcholy zadaný maticí sousednosti. To je matice $\mathbf{A} \in \mathbb{R}^{n\times n}$, která má na pozici $\mathbf{A}_{ij}$ jedničku, pokud vede hrana z vrcholu i do vrcholu j, jinde jsou nuly. Jaký význam má matice $\mathbf{A}^2 = \mathbf{AA}$? Dle definice:

$$(\mathbf{A}^2)_{ij} = \sum_{k=1}^{n} a_{ik}a_{kj}$$

Jak tuto sumu interpretovat? Započítám jedničku za každou dvojici hran, kde první vede z i do k a druhá z k do j. Výraz tedy určuje počet sledů délky 2 vedoucích z i do j. Pro připomenutí, *sled* označuje jakoukoli procházku po grafu, ve které se na rozdíl od cesty mohou opakovat vrcholy nebo dokonce hrany. Třeba výraz $(\mathbf{A}^2)_{ii}$ rozhodně nemusí být nulový.

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

Obdobně $(\mathbf{A}^\ell)_{ij}$ by byl počet sledů délky ℓ.

Úkol 3– Dosažitelnost [5b]:

Popište, jak upravit předchozí postup, aby spočítal matici dosažitelnosti. Tato matice na pozici i, j obsahuje jedničku právě tehdy, pokud existuje orientovaná cesta z i do j. Vyhněte se počítání s obrovskými čísly.

Matice a soustavy rovnic

Pomocí maticového násobení můžeme elegantně zapsat soustavu rovnic:

$$\mathbf{Ax} = \mathbf{b}$$

Hledáme vektor neznámých $\mathbf{x}$. Koeficienty neznámých v jedné rovnici jsou zapsány v řádku matice $\mathbf{A}$. Pravé strany rovnic tvoří vektor $\mathbf{b}$.

Také Gaussovu eliminaci můžeme zapsat maticově, elementární úpravy budou pracovat s řádky matice. Při ručním počítání nám maticový zápis ušetří psaní, nemusíme opisovat plusy a neznámé.

Minule jsme narazili na soustavy, které neměly jednoznačné řešení nebo řešení neměly vůbec. K této situaci došlo, pokud při Gaussově eliminaci vznikl řádek se samými nulami. Vzhledem k tomu, že při eliminaci k němu pouze přičítáme násobky jiných řádků, znamená to, že musel být jejich lineární kombinací. Jinými slovy řádky v matici $\mathbf{A}$ byly lineárně závislé.

Naopak, jsou-li řádky matice $\mathbf{A}$ lineárně nezávislé, pak má rovnice $\mathbf{Ax} = \mathbf{b}$ vždy jednoznačné řešení. Lineární nezávislost řádků je důležitá vlastnost, na kterou ještě mnohokrát narazíme. Má proto svůj název, o matici $\mathbf{A}$ řekneme, že je *regulární*.

Pozor na to, že úvahy výše platí pouze pro soustavy se čtvercovou maticí, tedy takové, kde je počet rovnic stejný jako počet neznámých. Je-li neznámých více než rovnic, pak nám ani lineární nezávislost řádků nezajistí jednoznačnost. Naopak, je-li rovnic více, pak může řešení být jednoznačné navzdory závislým řádkům. Pojem regulární matice se také vztahuje pouze ke čtvercovým maticím.

Inverzní matice

Zobrazení nám dávají nový pohled na soustavu rovnic: Hledáme vektor $\mathbf{x}$, který je maticí $\mathbf{A}$ zobrazen na $\mathbf{b}$. Při řešení soustavy rovnic se vlastně snažíme přehrát zobrazení pozpátku. Neexistuje tedy matice, která by toto opačné zobrazení popisovala?

Matice, kterou hledáme, se nazývá *inverzní* a budeme ji značit $\mathbf{A}^{-1}$. Chtěli bychom, aby její součin s $\mathbf{A}$ byl roven jednotkové matici. Pak bychom mohli vynásobit obě strany soustavy rovnic zleva inverzní maticí a získat explicitní vyjádření vektoru neznámých $\mathbf{x}$:

$$\mathbf{A}^{-1}\mathbf{Ax} = \mathbf{Ex} = \mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

Jak inverzní matici spočítat? Tato matice by byla neznámou $\mathbf{X}$ v rovnici $\mathbf{AX} = \mathbf{E}$. Maticové násobení můžeme rozepsat po sloupcích. Pokud označíme $\mathbf{x}_i$ sloupec matice $\mathbf{X}$, dostaneme pro každé i rovnici $\mathbf{Ax}_i = \mathbf{e}_i$. A tu můžeme interpretovat jako soustavu rovnic, kterou umíme vyřešit.

Jak jsme nahlédli výše, jednoznačné řešení mají pouze soustavy rovnic, ve kterých je matice $\mathbf{A}$ regulární. To také znamená, že inverzní matice existuje pouze k regulárním maticím.

Doteď jsme tiše předpokládali, že platí $\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1}$. Pro úplnost tedy předkládáme důkaz. Dejme tomu, že máme matici $\mathbf{X}$, pro kterou platí $\mathbf{A}\mathbf{X} = \mathbf{E}$. Předpokládejme, že existuje $\mathbf{A}^{-1}$ taková, že $\mathbf{A}^{-1}\mathbf{A} = \mathbf{E}$. Chceme ukázat, že $\mathbf{X}$ a $\mathbf{A}^{-1}$ jsou stejné.

$$\mathbf{X} = \mathbf{E}\mathbf{X} = (\mathbf{A}^{-1}\mathbf{A})\mathbf{X} = \mathbf{A}^{-1}(\mathbf{A}\mathbf{X}) = \mathbf{A}^{-1}\mathbf{E} = \mathbf{A}^{-1}$$

Úkol 4– Obecná inverze [3b]:

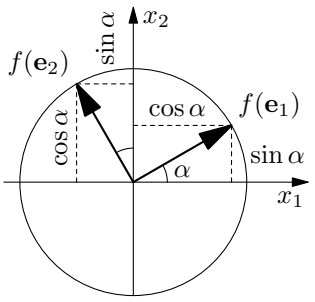
Najděte inverzní matici k

$$\mathbf{A} = \begin{pmatrix} c & -c \\ c & c \end{pmatrix}$$

pro obecnou konstantu c. Popište svůj postup. Možných přístupů je více, co třeba si představit zobrazení, které tato matice popisuje?

Řešení na konci textu

Matice rotace. Jaký vektor vznikne, pokud otočíme vektor $(1,0)^\top$ o úhel α? Výsledný vektor bude ležet na jednotkové kružnici. Jeho souřadnice lze tedy popsat goniometrickými funkcemi, konkrétně $(\cos\alpha, \sin\alpha)^\top$. Ještě potřebujeme totéž pro vektor $(0,1)^\top$, jeho otočením vznikne $(-\sin\alpha, \cos\alpha)^\top$.



Stačí tedy umístit tyto vektory do matice:

$$\begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}$$

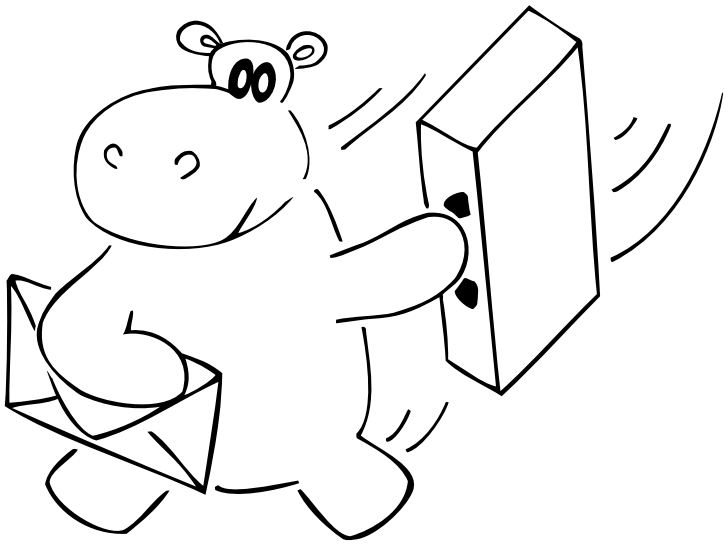
Asociativita maticového násobení. Mějme matice o rozměrech $\mathbf{A} \in \mathbb{R}^{m\times p}, \mathbf{B} \in \mathbb{R}^{p\times q}, \mathbf{C} \in \mathbb{R}^{q\times n}$.

$$((\mathbf{A}\mathbf{B})\mathbf{C})_{ij} = \sum_{k=1}^{q} (\mathbf{A}\mathbf{B})_{ik} c_{kj} = \sum_{k=1}^{q}\sum_{l=1}^{p} a_{il} b_{lk} c_{kj}$$

$$(\mathbf{A}(\mathbf{B}\mathbf{C}))_{ij} = \sum_{k=1}^{q} a_{ik} (\mathbf{B}\mathbf{C})_{kj} = \sum_{k=1}^{q}\sum_{l=1}^{p} a_{ik} b_{lk} c_{lj}$$

Dostali jsme sumu stejných výrazů, jen pořadí sum je opačné. To ovšem nevadí, protože sčítání reálných čísel je komutativní.

David Klement



Recepty z programátorské kuchařky: Haldy, heapsort a Dijkstrův algoritmus

Dnešní povídání o algoritmech a datových strukturách se bude zabývat jedním z nejznámějších algoritmů: Dijkstrovým algoritmem pro hledání nejkratších cest v grafech. K tomu (a nejenom k tomu) se nám bude hodit šikovná datová strukturka zvaná halda, tak si předvedeme nejdříve ji.

Halda

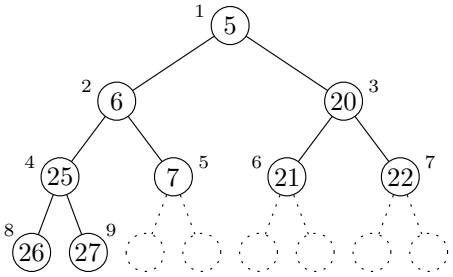
Halda je datová struktura pro uchovávání množiny čísel (či jakýchkoliv jiných prvků, na kterých máme definováno uspořádání, tj. umíme pro každou dvojici prvků říci, který z nich je menší). Tato datová struktura obvykle podporuje následující operace: přidání nového prvku, nalezení nejmenšího prvku a odebrání nejmenšího prvku. My si ukážeme jednoduchou implementaci haldy, která bude při uložení N prvků potřebovat čas $\mathcal{O}(\log N)$ na přidání či odebrání jednoho prvku a $\mathcal{O}(1)$ (tj. konstantní) na zjištění hodnoty nejmenšího prvku.

Naše implementace bude vypadat následovně: Pokud halda obsahuje N prvků, uložíme její prvky do pole na pozice 1 až N. Prvek na pozici k bude mít dva *následníky,* a to prvky na pozicích $2k$ a $2k+1$; samozřejmě, pokud je k velké, a tedy např. $2k+1 > N$, má takový prvek jen jednoho či dokonce žádného následníka. Naopak prvek na pozici $\lfloor k/2 \rfloor$ nazveme *předchůdcem* prvku na pozici k. Ti z vás, kteří znají binární stromy, v tomto jistě rozpoznali způsob, jak v poli uchovávat úplné binární stromy (následníci jsou synové a předchůdci otcové v obvyklé stromové terminologii, prvek č. 1 je kořen stromu).

Prvky haldy však v poli neuchováváme v úplně libovolném pořadí. Chceme, aby platilo, že každý prvek je menší nebo roven všem svým následníkům. Naše halda tedy může vypadat např. takto:

1	2	3	4	5	6	7	8	9
5	6	20	25	7	21	22	26	27

Tomu odpovídá tento strom:



Z toho, co jsme si právě popsali, je jasné, že nejmenší prvek je uložen na pozici s indexem 1, a tedy můžeme snadno v konstantním čase zjistit jeho hodnotu. Ještě prozradíme, jak lze prvky do haldy rychle přidávat a odebírat:

Jestliže halda obsahuje N prvků, pak nový prvek, říkejme mu třeba x, přidáme na konec pole, tj. na pozici s indexem $N+1$. Nyní x porovnáme s jeho předchůdcem. Pokud je jeho předchůdce menší, je vše v pořádku a jsme hotovi. V opačném případě x s jeho předchůdcem prohodíme. Tím jsme problém napravili, ale nyní může být x menší než jeho nový předchůdce. To lze napravit dalším prohozením a tak budeme pokračovat dále, než se buďto dostaneme do situace, kdy už je x větší nebo rovno svému předchůdci, nebo „vybublá“ až do kořene haldy, kde už žádného předchůdce nemá. Protože se v každém kroku pozice, na níž se prvek x právě nachází, zmenší alespoň na polovinu, provedeme dohromady nejvýše $\mathcal{O}(\log N)$ výměn, a tedy spotřebujeme čas $\mathcal{O}(\log N)$.

Odebírání nejmenšího prvku probíhá podobně: Prvek z poslední pozice (tj. z pozice N) přesuneme na pozici 1, tedy místo minima. Místo s předchůdci jej však porovnáme s jeho následníky, a v případě, že je větší než některý z jeho následníků, opět je prohodíme (pokud je větší než oba následníci, prohodíme ho s menším z nich). A protože se nám v každém kroku index „bublajícího“ prvku v poli alespoň zdvojnásobí, opět spotřebujeme čas $\mathcal{O}(\log N)$.

Jako cvičení si rozmyslete, že v čase $\mathcal{O}(\log N)$ lze z haldy smazat dokonce libovolný prvek, pokud si ovšem pamatujeme, kde se v haldě nachází. Také můžeme prvek ponechat a jen změnit jeho hodnotu.

Ještě si předvedeme program:

```
halda = []

def nejmensi():
  return halda[0]

def swap(a,b):
  (halda[a], halda[b]) = (halda[b], halda[a])

def vloz(prvek):
  # Vložíme prvek na konec
  halda.append(prvek)
  i = len(halda) - 1
  while (i > 0) and (halda[i//2] > halda[i]):
    swap(i, i//2)
    i = i//2

def smaz_nejmensi():
  N = len(halda)
  nejmensi = halda[0]
  halda[0] = halda[N - 1]
  # Odstraníme prvek z konce
  halda.pop()
  N -= 1
  i = 0
  while 2*i < N:
    j = i
    if halda[j] > halda[2*i]:
      j = 2*i
    if (2*i+1 < N) and (halda[j]>halda[2*i+1]):
      j = 2*i+1
    if i == j:
      break
    swap(i, j)
    i = j
  return nejmensi
```

HeapSort

Když už máme k dispozici haldu, můžeme pomocí ní například snadno třídit čísla. Máme-li N čísel, která chceme setřídit, vytvoříme si z nich nejprve haldu o N prvcích (například postupným vkládáním do prázdné haldy), načež z ní budeme postupně N-krát odebírat nejmenší prvek. Tím získáme prvky původního pole v rostoucím pořadí. Celkově provedeme N vložení, N nalezení minima a N smazání. To vše dohromady stihneme v čase $\mathcal{O}(N \log N)$.

Než si ukážeme program, přidáme ještě dva triky, které nám implementaci značně usnadní. Předně si vše uložíme do jed-

noho pole – to bude při plnění haldy obsahovat na svém začátku haldu a na konci zbytek vstupního pole, přitom zbytek pole se bude postupně zmenšovat a uvolňovat tak místo haldě; naopak v druhé polovině algoritmu budeme zmenšovat haldu a do volného prostoru ukládat setříděné prvky. K tomu se nám bude hodit získávat prvky v opačném pořadí, proto si upravíme haldu tak, aby udržovala nikoliv minimum, nýbrž maximum.

Druhý trik spočívá v tom, že nebudeme haldu vytvářet postupným vkládáním, nýbrž naopak zabubláváním prvků (podobným, jako děláme při mazání minima) od konce. Všimněte si, že takto také získáme správné nerovnosti mezi prvky a jejich následníky, a dokonce tak zvládneme celou haldu vytvořit v lineárním čase (proč to tak je, si zkuste dokázat sami, stačí si uvědomit, kolikrát zabubláváme které prvky). Zbytek třídění bohužel nadále zůstává $\mathcal{O}(N \log N)$.

Tomuto algoritmu se obvykle říká *HeapSort* (čili třídění haldou) a je jedním z mála známých rychlých třídících algoritmů, které nepotřebují pomocnou paměť.

```
# Zabublání prvku dolů:
#   N určuje část pole vyhrazenou haldě
#   i je index zabublávaného prvku
def bublej(pole, N, i):
  while 2*i < N:
    j = 2*i
    if (j+1 < N) and (pole[j+1] > pole[j]):
      j = j+1
    if pole[i] >= pole[j]:
      break
    (pole[i], pole[j]) = (pole[j], pole[i])
    i = j

def heapsort(pole):
  N = len(pole)
  for i in range(N//2, -1, -1):
    bublej(pole, N, i)
  # Výběr maxima a jeho přesun nakonec
  for i in range(N - 1, 0, -1):
    (pole[0], pole[i]) = (pole[i], pole[0])
    # Dál už bubláme jen na zbytku pole
    bublej(pole, i, 0)
  return pole
```

Dijkstrův algoritmus

Nyní již konečně k slíbenému *Dijkstrovu algoritmu.* Tento algoritmus dostane orientovaný graf s hranami ohodnocenými nezápornými čísly (viz kuchařka o grafech)[3] a nalezne v něm nejkratší cestu mezi dvěma zadanými vrcholy. Ve skutečnosti tento algoritmus dělá o malinko více: Najde totiž nejkratší cesty z jednoho zadaného vrcholu do všech ostatních.

Nechť v_0 je vrchol grafu, ze kterého chceme určit délky nejkratších cest. Budeme si udržovat pole délek zatím nalezených cest z vrcholu v_0 do všech ostatních vrcholů grafu. Navíc u některých vrcholů budeme mít poznamenáno, že cesta nalezená do nich je už ta nejkratší možná. Takovým vrcholům budeme říkat *definitivní.* Na začátku inicializujeme v poli všechny hodnoty na ∞ kromě hodnoty odpovídající vrcholu v_0, kterou inicializujeme na 0 (délka nejkratší cesty z v_0 do v_0 je 0). V každém *kroku* algoritmu pak provedeme následující: Vybereme vrchol w, který ještě není definitivní, a mezi všemi takovými vrcholy je délka zatím nalezené cesty do něj nejkratší možná. Vrchol w prohlásíme za definitivní. Dále otestujeme, zda pro nějaký vrchol v cesta z vrcholu v_0 do w a pak po hraně z w do v není kratší, než zatím nalezená cesta z v_0 do v, a je-li tomu tak, upravíme délku zatím nalezené cesty do v. Toto provedeme pro všechny takové vrcholy v. Celý algoritmus skončí, pokud jsou už všechny vrcholy definitivní nebo všechny vrcholy, které nejsou definitivní, mají délku cesty rovnou ∞ (v takovém případě se graf skládá z více nesouvislých částí).

Předtím, než dokážeme, že právě představený algoritmus opravdu nalezne délky nejkratších cest z vrcholu v_0, se zamysleme nad jeho časovou složitostí.

Pro každý z N vrcholů si délku dosud nalezené cesty uchováváme v poli. Celý algoritmus provede nejvýše N kroků, protože v každém kroku nám přibyde jeden definitivní vrchol. Ten vybíráme jako minimum z délky aktuální cesty přes všechny dosud nedefinitivní vrcholy, kterých je $\mathcal{O}(N)$. V každém kroku musíme zkontrolovat tolik vrcholů v, kolik hran vede z vrcholu w. Počet takových změn pro všechny kroky dohromady je pak nejvýše $\mathcal{O}(M)$, kde M je počet hran vstupního grafu. Z toho vyjde časová složitost $\mathcal{O}(N^2 + M)$, čili $\mathcal{O}(N^2)$, jelikož M je nejvýše N^2. Tuto implementaci Dijkstrova algoritmu najdete na konci naší kuchařky.

K uchovávání délek dosud nalezených nejkratších cest můžeme ovšem použít haldu. Ta bude na začátku obsahovat N prvků a v každém kroku se počet jejích prvků sníží o jeden: Nalezneme a smažeme nejmenší prvek, to zvládneme v čase $\mathcal{O}(\log N)$, a případně upravíme délky nejkratších cest do sousedů právě zpracovávaného vrcholu. To pro každou hranu trvá rovněž $\mathcal{O}(\log N)$, celkově za všechny hrany tedy $\mathcal{O}(M \log N)$. Z toho vyjde celková časová složitost algoritmu $\mathcal{O}((N+M) \log N)$, a to je pro „řídké“ grafy (tedy grafy s $M \ll N^2$) výrazně lepší.

Vraťme se nyní k důkazu správnosti Dijkstrova algoritmu. Ukážeme, že po každém kroku algoritmu platí následující tvrzení: Nechť A je množina definitivních vrcholů. Pak délka dosud nalezené cesty z v_0 do v (v je libovolný vrchol grafu) je délka nejkratší cesty $v_0 v_1 \ldots v_k v$ takové, že všechny vrcholy $v_0, v_1, \ldots, v_k$ jsou v množině A. Tvrzení dokážeme indukcí dle počtu kroků algoritmu, které již proběhly. Tvrzení zřejmě platí před a po prvním kroku algoritmu. Nechť w je vrchol, který byl v předchozím kroku prohlášen za definitivní. Uvažme nejprve nějaký vrchol v, který je definitivní. Pokud $v = w$, tvrzení je triviální. V opačném případě ukážeme, že existuje nejkratší cesta z v_0 do v přes vrcholy z A, která nepoužívá vrchol w. Označme D délku cesty z v_0 do v přes vrcholy A bez vrcholu w. Protože v každém kroku vybíráme vrchol s nejmenším ohodnocením a ohodnocení vybraných vrcholů v jednotlivých krocích tvoří neklesající posloupnost (váhy hran jsou nezáporné!), tak délka cesty z v_0 do w přes vrcholy z A je alespoň D. Ale potom délka libovolné cesty z v_0 do v přes w používající vrcholy z A je alespoň D. Z volby D pak víme, že existuje nejkratší cesta z v_0 do v přes vrcholy z A, která nepoužívá vrchol w.

Nyní uvažme takový vrchol v, který není definitivní. Nechť $v_0 v_1 \ldots v_k v$ je nejkratší cesta z v_0 do v taková, že všechny vrcholy $v_0, v_1, \ldots, v_k$ jsou v množině A. Pokud $v_k = w$, pak jsme ohodnocení v změnili na délku této cesty v právě proběhlém kroku. Pokud $v_k \neq w$, pak $v_0 v_1, \ldots, v_k$ je nejkratší

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

cesta z v_0 do v_k přes vrcholy z množiny A, a tedy můžeme předpokládat, že žádný z vrcholů $v_1, \ldots, v_k$ není w (podle toho, co jsme si rozmysleli na konci minulého odstavce). Potom se ale délka cesty do v rovnala správné hodnotě už před právě proběhlým krokem.

Vzhledem k tomu, že po posledním kroku množina A obsahuje právě ty vrcholy, do kterých existuje cesta z vrcholu v_0, dokázali jsme, že náš algoritmus funguje správně.

Na závěr ještě poznamenejme, že Dijkstrův algoritmus je možné snadno upravit tak, aby nám kromě určení délky nejkratší cesty i takovou cestu našel: U každého vrcholu si v okamžiku, kdy mu měníme ohodnocení, poznamenáme, ze kterého vrcholu do něj přicházíme. Nejkratší cestu do nějakého vrcholu pak zrekonstruujeme tak, že u posledního vrcholu této cesty zjistíme, který vrchol je předposlední, u předposledního, který je předpředposlední, atd.

Poznámka pro zvídavé: existují i jiné druhy hald, například k-regulární haldy, v nichž má každý prvek k následníků (rozmyslete si, jaká je v takové haldě časová složitost operací a jak nastavit k v závislosti na M a N, aby byl Dijkstrův algoritmus co nejrychlejší), nebo tzv. Fibonacciho halda, která dokáže upravit hodnotu prvku v konstantním čase. S tou pak umíme hledat nejkratší cesty v čase $\mathcal{O}(M + N \log N)$.

Dnešní menu Vám servírovali

Dan Kráľ, Martin Mareš a Petr Škoda

Implementace Dijkstrova algoritmu s haldou

```
# Struktura pro vrchol jako dvojici (index, vzdálenost)
# (definuje vlastní porovnávání, aby šlo použít haldu výše)
class vrchol():
  def __init__(self, index, vzdalenost):
    self.index = index
    self.vzdalenost = vzdalenost
  # Předefinování operátoru >
  def __gt__(self, obj):
    return self.vzdalenost.__gt__(obj.vzdalenost)

halda = []
def dijkstra(N, cesty, start):
  final = [False] * N
  vzdalenost = [None] * N
  # Inicializace startu:
  vloz(vrchol(start, 0))
  vzdalenost[start] = 0
  while halda:
    # Vytáhneme z haldy nejmenší nezpracované místo
    v = smaz_nejmensi()
    if final[v.index]:
      continue
    # Označíme místo za použité
    final[v.index] = True
    # Projdeme všechny sousedy
    for (i, delka) in cesty[v.index]:
      if vzdalenost[i] is None or v.vzdalenost + delka < vzdalenost[i]:
        vzdalenost[i] = v.vzdalenost + delka
        vloz(vrchol(i, vzdalenost[i]))
  return vzdalenost
```

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/*