Korespondenční Seminář z Programování

32. ročník **KSP** **Říjen 2019**

Milí řešitelé, řešitelky a řešitelčata!

Druhé číslo jubilejního 2^5-tého ročníku KSPčka se vám právě dostalo do ruky. A co v něm najdete? Opět 5 klasických úloh (z toho nějaké praktické) a k nim navíc pokračování datového seriálu, který přináší svůj druhý díl o OSM. A jako dezert vám tentokrát servírujeme kuchařku o hledání cest.

Všichni zájemci o studium informatiky jsou srdečně zváni na **Den otevřených dveří na Matfyzu**, který se koná **21. listopadu**. Večer předtím si nenechte ujít **Kalíšek** – setkání s organizátory a řešiteli KSPčka. Více informací najdete na našem webu.

Připomínáme, že se z každé série stále **započítává 5 nejlépe vyřešených úloh** (tedy nemusíte vyřešit úplně všechny a i tak můžete dosáhnout na plný počet bodů). Také se vám body za úlohy **přepočítávají podle vašeho služebního stáří** – na přesnou definici se podívejte do pravidel na webu.

Za úspěšné řešení KSP můžete být **přijati na MFF UK bez přijímacích zkoušek**. Úspěšným řešitelem se stává ten, kdo získá za celý ročník (této 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, placku a možná i další překvapení.

Termín série: **2. prosince 2019 v 8:00**

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

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

Odměna série: Každému, kdo **získá alespoň 3 body z každé úlohy v sérii**, pošleme **sladkou odměnu**.

Druhá série třicátého druhého ročníku KSP

Minulý díl jsme skončili po nalezení uvězněných kolonistů v jedné z budov základny a po objevení stop vedoucích ven ze základny. Kapitánka Laren se právě chystala vydat rozkaz a vy jste v hlasování rozhodli, co to bude. V době začátku psaní zadání druhého dílu to bylo nerozhodně mezi možnostmi „Vydat se se svou posádkou po stopách“ a „Uspořádat párty a počkat, co se stane“.

⋆⋆⋆

Kapitánka se zamyslela a pak vydala rozkaz: „Draku, připravte spolu s Jane nějaké vozidlo,“ zaúkolovala oba mariňáky, „vydáme se po těch stopách.“

„Zbytek posádky zůstane tady, pomůže kolonistům a kdyby bylo potřeba, bude připraven vzlétnout s Hefaistem na pomoc – udržujte loď v pohotovostním stavu.“ Rozhlédla se ještě po zbídačených kolonistech a pak spiklenecky mrkla: „A můžete zatím uspořádat párty, aby jim to trochu zvedlo náladu – u Antaraxu se mi povedlo získat mrazící kontejner plný zmrzliny, je v zadním nákladovém prostoru.“

Oba mariňáci a kapitánka se vydali připravovat jedno ze zbylých terénních vozidel do garáže základny, vyměnili v něm palivové články za zbrusu nové donesené z Hefaista, naložili zásoby na několik dní a vydali se po nalezených stopách.

Lodní lékař mezitím ošetřil všechna zranění kolonistů a byly podniknuty první opravy základny. Obzvláště zapeklitým problémem se ukázalo být seřízení fúzního reaktoru, který zatím pracoval na necelý čtvrtinový výkon.

32-2-1 Seřízení reaktoru — 11 bodů

Fúzní reaktor drží reakční oblast pod kontrolou pomocí supravodivých magnetů, ale ty nyní nejsou správně zarovnané. Supravodivé magnety použité v konstrukci tohoto konkrétního fúzního reaktoru jsou kruhové a naskládané jeden na druhém do jakéhosi válce.

Každý kruhový magnet vytváří v jednom místě na svém obvodu elektromagnetický vír a pro správné seřízení reaktoru bychom potřebovali každý z magnetů otočit tak, aby se všechny víry ocitly přesně nad sebou. Ale otáčení každého z magnetů je velmi těžká práce, takže bychom chtěli najít směr, do kterého máme všechny otočit, abychom se přitom co možná nejméně nadřeli.

Předpokládejte, že na vstupu dostanete otočení všech magnetů ve stupních (což nemusí být celé stupně, ale obecně to může být jakékoliv reálné číslo mezi 0 až 360). Dále definujme, že otočení jednoho magnetu o jeden stupeň v nějakém směru stojí jednu jednotku práce. Vaším úkolem bude určit, do jakého úhlu se mají všechny otočit, aby nás to stálo co nejméně práce.

Příklad: Mějme 3 magnety otočené do úhlů 40°, 350° a 40°. Pak nejlevnější způsob, jak všechny zarovnat, je otočit druhý z nich o +50°, čímž všechny vyrovnáme na úhlu 40° s cenou 50 jednotek práce.

Když se místní šestatřicetihodinový den nachýlil ke konci, už byla základna opět alespoň v základu funkční – reaktor opět dodával plnou energii, všechny velké budovy byly napojeny na centrální rozvod energie a všechny trhliny v nich byly alespoň provizorně utěsněny.

Kapitánka se mezitím ohlásila s tím, že se blíží ke skalám vzdáleným asi osmdesát kilometrů od základny a že zatím stále následují stopy. Zatímco ale terénní vozidlo uhánělo mrazivou pustinou, tak se na základně postupně sešli všichni

v jídelně hlavní budovy, kde ředitel kolonie přivítal všechny se slovy: „Děkujeme posádce Hefaista," kývl směrem k šesti přítomným členům posádky, „za všechnu pomoc, kterou nám poskytli. S obnovou základny bude ještě spousta práce, ale dnes pojďme oslavovat."

Zmrzlina z Hefaista měla velký úspěch a oslava se postupně rozproudila. Jedna ze skupinek kolonistů začala dokonce plánovat, jak si vezmou dovolenou a zaletí si domů, jenom naplánovat nejrychlejší trasu.

32-2-2 Mezihvězdné jízdní řády — 12 bodů

Lidské osídlení známé části vesmíru je již tak velké, že se ustanovilo množství pravidelných dopravních linek. Cesta mezi „zastávkami" jim sice namísto pár minut trvá několik dnů, ale jinak je jejich letový plán dost podobný třeba klasickým jízdním řádům.

Na vstupu dostaneme pravidelné mezihvězdné linky jako seznam planet, které navštěvují, jak dlouho jim přelety trvají a v jaké časy létají (přesněji kdy vyráží ze své první „zastávky" a za jak dlouho se linka opakuje). Naším úkolem pak bude pro zadanou počáteční a cílovou planetu spočítat, jakým nejrychlejším způsobem se tam umíme dostat.

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 dvě čísla P a L oddělená mezerou. Ta udávají počet planet a počet mezihvězdných linek, které mezi nimi létají. Na druhém řádku se pak nachází dvě čísla S a C oddělená mezerou, která udávají index startovní a cílové planety (indexujeme od nuly, takže platí $0 \leq S, C < P$). Poté následuje L popisů mezihvězdných linek. Popis pro i-tou linku tvoří:

- Na prvním řádku čtyři čísla S_i, T_i, P_i, Z_i oddělená mezerou udávající počáteční planetu, čas, kdy vylétá ze startovní planety první loď, periodu, za jak dlouho po první lodi vyletí druhá loď (a po ní třetí, čtvrtá,...), a nakonec počet dalších zastávek.
- Na dalších Z_i řádcích je pak vždy dvojice čísel p a t oddělená mezerou, ty udávají index planety, která je další zastávkou, a čas letu od minulé planety k této.

Slibujeme, že všechny časy budou kladná celá čísla.

Formát výstupu: Na první řádek výstupu vypište dvě čísla: nejrychlejší čas, za který se umíte dostat na cílovou planetu C, pokud vyrazíte ze startovní planety S v čase 0, a počet navštívených planet na trase. Na dalších řádcích pak vždy uveďte indexy planet, přes které vaše nalezená cesta vede (první by měla být planeta S a poslední pak planeta C).

Ukázkový vstup:

```
6 3
1 5
2 1 3 2    # Linka A
1 2
3 2
1 0 1 1    # Linka B
5 15
4 1 5 2    # Linka C
3 3
5 4
```

Ukázkový výstup:

```
13 3
1
3
5
```

Popis příkladu: Existuje sice přímý spoj z planety 1 přímo na planetu 5, který vyráží v čase 0 (linka B), ale do cíle nás doveze až v čase 15. Lepší je počkat si na startu do času 3 a naskočit na loď, která nás doveze na planetu 3 v čase 5 (linka A). Tím nám sice o kousek uletěla první loď linky C, ale ta má periodu 5, takže v čase 9 se můžeme vydat dál a do cíle tak dorazíme v čase 13.

Párty byla v plném proudu, když dovnitř vrazil Zax a svým nepozemským kolébavým krokem se vydal k hloučku posádky Hefaista. Zax se uvolil k tomu, že bude držet hlídku u lodního komunikátoru, a nyní vypadal docela rozrušeně.

„Zachytil jsem nouzové volání od kapitánky, ale nepovedlo se mi navázat spojení, startujeme?" zeptal se pilota Barnabáše, který byl v nepřítomnosti kapitánky ve velení. Ten jenom rychle kývnul a celá posádka se vydala k hangáru, kde měli dýchací masky, a pak urychleně na loď. Motory byly udržovány zahřáté, takže předstartovní procedura zabrala necelé dvě minuty a pak už se devadesátimetrová loď za burácení atmosférických motorů vznesla k obloze.

Let k blízkému pohoří trval Hefaistu necelých 10 minut a pak už se vznášel nad opuštěným terénním vozidlem, které stálo vedle evidentně uměle vyrobeného tunelu do skály. Zax zkusil znovu zavolat výsadek a tentokrát se mu to povedlo, i když signál byl hodně utlumený.

„Jsem ráda, že tu jste. Někdo to tady vybudoval potají... Máme tu teď takovou situaci a docela by se nám hodila vaše pomoc," odpověděla kapitánka. Pak jim v rychlosti vylíčila, že objevili nějakou utajenou základnu, která evidentně byla opuštěná docela ve spěchu. Pravděpodobně se tu prováděly utajené experimenty s cílem uměle vyvolávat a zesilovat iontové bouře, alespoň tomu odpovídala nalezené schémata načmáraná na tabulích různě po základně.

Jedním z produktů byly i nabité prachové částice uložené v podzemní jeskyni – jenže tato jeskyně se hroutila. Kdyby se zhroutila úplně, tak hrozilo uniknutí částic do atmosféry planety, což mohlo mít nedozírné následky. Bylo potřeba najít v blízkosti místo, kam by se mohly částice bezpečně odčerpat...

32-2-3 Nádrž na částice — 8 bodů

Potřebujeme někam provizorně odčerpat nebezpečné částice. Hledáme nějakou správně velkou přirozenou nádrž v horách, do které se částice akorát vejdou, a která půjde neprodyšně zakrýt plachtou bránící úniku částic do atmosféry.

Svět si můžeme představit pro zjednodušení jako řez horami zadaný ve čtverečkové síti jako na obrázku níže. Nádrž, která se dá zakrýt plachtou, je „prohlubeň", která po stranách má políčka pevné skály přesně ve výšce hladiny (aby šla dobře ukotvit krycí plachta). Svět je na vstupu zadán výškami hor v řezu postupně po jednotlivých sloupcích.

Vaším cílem je navrhnout algoritmus, který bude umět rychle zjistit, jestli existuje nádrž dané velikosti.

```
#....###..#
##..#####.#
###########
```

Příklad: Pro vstup 3 2 1 1 2 3 3 3 2 1 3 (na obrázku výše) umíme najít nádrž velikosti 2, 3 nebo 6 (vždy mají po stranách políčka skály přesně ve výšce hladiny), ale naopak nádrž velikosti 1 podle výše zmíněných parametrů neexistuje.

Nebezpečné částice se povedlo alespoň dočasně zajistit a posádka se pustila do letmého průzkumu podivné základny. Tohle sice nespadalo do jejich úkolu, kterým byla pomoc

kolonistům a případná záchranná operace, ale zvědavost jim nedala.

Vypadalo to, že základnu opustili dost narychlo potom, co se jeden z experimentů nepovedl a způsobil masivní iontovou bouři, která zpustošila základnu. Také zde našli scházející nouzový generátor z kolonie a osobní věci se jmény několika ze zmizelých kolonistů, což nasvědčovalo tomu, že alespoň část kolonistů měla ještě jiný úkol na této základně, o které zbytek kolonistů nevěděl.

Při odchodu vymazali původní obyvatelé základny většinu dat, ale systém na spojení se sledovacími družicemi na oběžné dráze ještě zůstal funkční. Družice na oběžnou dráhu rozmístila kolonie, ale vypadalo to, že v jejich výbavě byly i další utajené komponenty a komunikační systémy schopné pracovat při různé úrovni rušení od iontových bouří. A jedna taková bouře právě vznikala...

32-2-4 Družicová komunikace — 10 bodů

Družice na oběžné dráze jsou propojeny různě odolnými komunikačními kanály. Každý z komunikačních kanálů vede mezi dvojicí družic (tedy můžeme si ho představit jako hranu v grafu) a má nějakou odolnost vůči rušení vyjádřenou celým číslem – pokud je rušení maximálně tolik co odolnost, tak spojení funguje, pokud je rušení větší, tak je spojení příliš zarušené a nedají se po něm přenášet žádná data.

Nás by s ohledem na vznikající iontovou bouři zajímala datová struktura, která bude umět rychle odpovídat na otázku, na kolik komponent se rozpadne družicová síť při rušení intenzity Q (komponentu tvoří družice, mezi nimiž existuje cesta po komunikačních kanálech odolnosti alespoň Q).

Takových dotazů bude přicházet mnoho (řádově tolik, jak bude velký vstup, tedy počet družic a spojení mezi nimi) a vaše datová struktura na ně musí být schopná odpovídat online (musí odpovědi vydávat průběžně, ne až na konci). Vaším cílem je dosáhnout co nejmenšího času za celou dobu běhu (součet doby předvýpočtu a doby odpovídání na jednotlivé dotazy).

Lehčí varianta (za 7 bodů): O něco méně bodů dostanete, pokud budete umět úlohu vyřešit rychle offline – tedy dotazy dostanete všechny najednou a můžete na ně odpovědět až na konci běhu algoritmu.

Kapitánka Laren spolu s hlavním inženýrem McCormackem koukali na dynamicky se vyvíjející mapu znázorňující postup iontové bouře. „Nechápu, kde se vzala tak rychle. Ale bude to teda pořádná jízda," poznamenal svým typicky klidným tónem hlavní inženýr. „Bude u kolonie asi tak do jedné hodiny. A připomínám, že z generátoru magnetického štítu kolonie zbyl škvarek. Budeme evakuovat?"

„Kolonisté to nebudou chtít vzdát, nemůžeme využít Hefaista a jeho štít?" zeptala se kapitánka. Pohled, který hlavní inženýr vrhl na kapitánku byl všeříkající – věděl, že odporovat kapitánce v tomhle nemá smysl a že ta otázka byla vlastně rozkaz. A taky věděl, že bude na něm, aby během hodiny přenastavil štít lodě tak, aby obsáhl více než stokrát větší objem prostoru, než na co byl projektovaný. Ale odporovat nešlo.

Hefaistos se urychleně přesunul zpět ke kolonii a tentokrát s ním pilot Barnabáš přistál tak, aby jeho devadesátimetrový trup dělal clonu proti postupující bouři. Hned potom se hlavní inženýr a jeho pomocník, ke kterým se přidalo několik inženýrů z kolonie, vrhli na přípravy. Hefaistos zasunul všechny antény a pro všechny případy naopak vysunul rozměrné tepelné štíty přes okna můstku a přes trysky motorů.

Kolonisté se začali přesouvat do nákladového prostoru Hefaista, protože trup odolné kosmické lodě byl bezpečnější, než již jednou zasažená kolonie. Mezitím technici z kolonie dotáhli energetické vedení od fúzního reaktoru k Hefaistovi a naopak od Hefaista natáhli několik kabelů k emitorům po obvodu kolonie. Pak už zbývalo jen vše správně seřídit a doufat.

32-2-5 Štítové emitory — 11 bodů

Štítové emitory magnetického štítu rozmístěné po obvodu kolonie je potřeba seřídit na přesný výkon, jen tak totiž vytvoří správně tvarovaný magnetický štít. Bohužel jejich nastavování není zrovna jednoduché.

Emitor je tvořen z N cívek o postupně rostoucí velikosti $1, 2, 3, \ldots, N$ a každou z nich lze zapojit v kladném nebo v záporném směru (takže cívka K k celkovému výkonu přispívá hodnotou buď K, anebo $-K$).

Vymyslete algoritmus, který pro zadaná celá čísla N a X (kde $N \geq 1$) odpoví, jestli lze dosáhnout výkonu X nějakým zapojením cívek $1, \ldots, N$ (a případně jakým).

Příklad: Pro $N = 4$ a $K = 4$ lze požadovaného výkonu dosáhnout zapojením $+1 + 2 - 3 + 4$, pro $N = 4$ a $K = 11$ to očividně nejde.

Přetlakové dveře zapadly za posledním členem posádky ve chvíli, kdy se na okraji kolonie již začaly třpytit částice narážející do magnetického štítu. Během několika minut pak bouře přišla v celé své síle. Na můstku se rozblikala diskotéka poplašných světélek po panelech technické sekce, ale naddimenzované generátory štítu zatím držely, zatím.

Po deseti minutách už to začínalo vypadat, že iontovou bouři překonají v pořádku, když tu bouře začala podle detektorů okolo základny kroužit a nápor vyvíjený na štítové emitory začal stále rychleji kolísat. Generátor na Hefaistovi kvílel, jak se pokoušel magnetické pole vyrovnávat, ale pak jeden z emitorů na okraji kolonie s efektním zábleskem selhal. Okolní emitory díru v magnetickém poli rychle vyrovnaly, ale i tak náhlý nápor zatřásl celou lodí na jejích masivních tlumičích.

„Kapitáne, tohle kolísání celý systém hrozně přetěžuje. Možná právě takhle vybuchl generátor štítu kolonie!" zvolal McCormack mezitím, co se odporoučel i druhý štítový emitor a vysokofrekvenční kvílení z generátoru rezonující celou lodí ještě zvýšilo frekvenci...

⋆⋆⋆

Opět máte možnost ovlivnit, jak se bude příběh vyvíjet dál. A nyní může váš hlas rozhodnout ještě víc o osudu posádky Hefaista a všech kolonistů. Co by měli v této situaci udělat? Zahlasujte do ***11. listopadu*** *v anketě*[1] *(tentokrát jsou možnosti plně na vašich návrzích).*

Druhý díl příběhu pro vás sepsal

Jirka Setnička

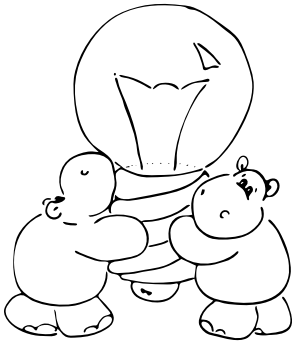
[1] `https://poll.ly/#/293gZ8WW`

32-2-6 Mapy z kOSMu — 17 bodů

Vítejte u druhého dílu datového seriálu. Jak jsme slíbili v minulém dílu, tak druhý díl ještě věnujeme datům z OSM, ale zkusíme je obohatit o něco navíc – ať už to bude správné počítání vzdáleností na mapě, nebo třeba výšková data získaná při radarovém mapování z raketoplánu.

Jak počítat vzdálenost

Při kreslení heatmapy v prvním dílu seriálu jste si mohli všimnout, že je oproti klasických mapám taková splácnutá a lehce deformovaná (oproti mapám, které klasicky vídáme). Bylo to tím, že jsme brali zeměpisné souřadnice přímo jako osy x a y bez jakéhokoliv přepočtu. To by fungovalo, kdybychom uznávali teorii o placaté Zemi, ale na Zemi tvaru koule (či lépe řečeno geoidu) to bohužel nefunguje – lehce si můžeme uvědomit, že jeden stupeň zeměpisné šířky měří na rovníku určitě větší vzdálenost, než někde u tučňáků na Antarktidě poblíž jižního pólu.



S problémem, jak správně nakreslit mapu do roviny tak, aby byla co nejméně deformovaná, se kartografové potýkají od té doby, co začali věřit na kulatou Zemi. Nikdy to nelze udělat úplně správně – některé části mapy budou přesněji zachovávat své proporce, zatímco jiné se deformují víc (například na většině evropocentrických map, se kterými jste se potkali ve škole, je abnormálně roztažená Antarktida). Existuje mnoho různých promítání – ze známých zmiňme třeba Mercatorovo – a více se o nich můžete dozvědět třeba na stránka na Wikipedii.[2]

My však teď nebudeme chtít kreslit mapu, ale počítat vzdálenosti na ní. Ani měření vzdáleností nefunguje tak, že bychom spočítali vzdálenost mezi zadanými body pomocí Pythagorovo věty, ale potřebujeme brát v úvahu vzdálenosti na kouli (v základním přiblížení), případně rovnou na geoidu (pokud chceme mít výsledky přesnější, protože Země je vlivem rotace zploštělá koule). Anglicky se nejkratší vzdálenosti dvou bodů na povrchu koule říká *great-circle distance*, česky pro to máme kratší slovo *ortodroma*. Na stránce na Wikipedii[3] můžete najít vzoreček, ale my pro naše počítání zvolíme o něco lépe použitelnější metodu.

Onou lépe použitelnou metodou je *haversinová formule*,[4] která počítá s úhly a stranami sférického trojúhelníku (trojúhelníku, který nakreslíme na povrch koule). Stále nebereme v úvahu geoid, ale nám to pro tuto úlohu bude stačit. Vzorce jsme pro vás přechroustali (s inspirací z projektu *Rosetta Code*) a zde vám přinášíme funkci na počítání vzdálenosti v Pythonu vracející výsledek v metrech:

```
from math import radians, sin, cos, atan2, sqrt

def haversineDistance(point1, point2):
  R = 6378137  # Poloměr Země (v~metrech)
  (lat1, lon1) = point1
  (lat2, lon2) = point2
  phi1    = radians(lat1)
  phi2    = radians(lat2)
  dphi    = radians(lat2 - lat1)
  dlambda = radians(lon2 - lon1)

  a = sin(dphi/2)**2 + \
    cos(phi1) * cos(phi2) * sin(dlambda/2)**2

  return 2*R*atan2(sqrt(a), sqrt(1 - a))
```

Nyní již máme dost věcí k tomu, abychom vám zadali první podúlohu – podobně jako v minulém dílu by to mělo být jenom jednoduché cvičení na rozjezd:

Úkol 1 [3b]:

Zajímala by nás délka všech vlakových kolejí na mapě (a to včetně všech kolejí na nádražích, všech zmapovaných vleček a podobně). Koleje jsou naštěstí v OSM docela jasně označené – jsou to cesty s tagem `railway=rail`.

Vaším úkolem tak bude spočítat délku všech kolejí na mapě. Opět vše počítejte na mapě Brna a na mapě Evropy.

Výstup odevzdejte (pro každý ze vstupů) jako jedno číslo (i s jednotkou, tedy jestli je vzdálenost v metrech, kilometrech nebo třeba v mílích). Pro počítání vzdáleností používejte výše zmíněnou haversinovou formuli.

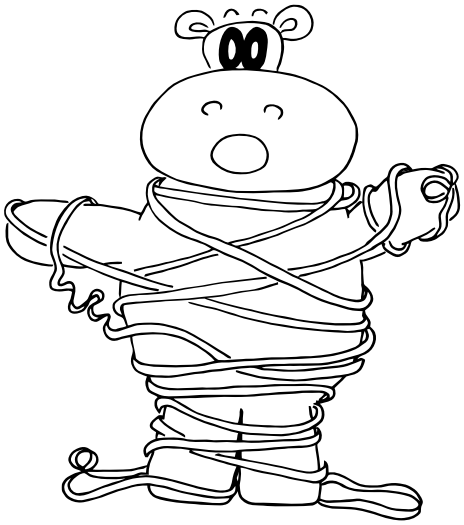
Pro Hrochův Týnec, který používáme jako testovací vstup, nám naše řešení spočítalo délku kolejí na $8,931\,km$.

Mapy jsou dostupné ke stažení z webové stránky seriálu;[5] jsou stejné, jako v první sérii.

Nadmořské výšky v mapě

Samotná data z OSM neobsahují nadmořské výšky – body (nodes) mají jenom atributy `lat` a `lon`. Ale přitom v mapách generovaných z OSM (například Mapy.cz za našimi a slovenskými hranicemi) běžně vídáme vrstevnice a jsou i docela přesné, odkud se tam berou?

Tou správnou odpovědí je, že je to vždy kombinace OSM a nějakého zdroje výškových dat. Pro výšková data se nejčastěji používají volně dostupná data z mise SRTM (neboli *Shuttle Radar Topography Mission*). To byla jedenáctidenní mise amerického raketoplánu Atlantis v únoru roku 2000, během níž pomocí dvojice radarů nasnímkovali většinu povrchu Země s rozlišením na jednu úhlovou vteřinu (asi 30 metrů na rovníku).



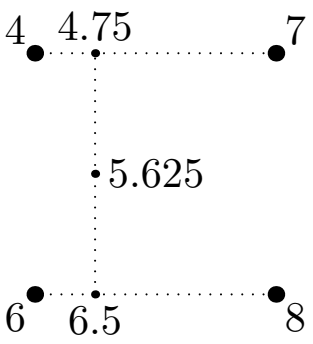
[2] https://cs.wikipedia.org/wiki/Mapov%C3%A9_zobrazen%C3%AD
[3] https://cs.wikipedia.org/wiki/Ortodroma
[4] https://en.wikipedia.org/wiki/Haversine_formula
[5] http://ksp.mff.cuni.cz/viz/serial-osm

Mise to byla nanejvýš zajímavá, včetně výzvy schovat šedesátimetrový stožár do čtyřmetrové krabice v nákladovém prostoru raketoplánu (radary potřebovaly být při snímkování v přesné pozici šedesát metrů od sebe a výsuvný stožár musel být kvůli jinému rozměrnému vybavení umístěn v nákladovém prostoru raketoplánu napříč – doporučujeme si vyhledat na internetu obrázky pod heslem „SRTM mast“). Pro nás (a pro všechny další uživatele OSM) je teď ale důležitější, že získaná data tvoří nejkompletnější volně dostupnou topografickou mapu Země.

SRTM data se dají získat z mnoha zdrojů a stále ještě dochází k jejich drobným opravám (například dopočítávání odrazů v úzkých údolích). Abychom měli stejná vstupní data, tak jsme opět vystavili kopie těchto dat na webu na stránce seriálu. Data jsou v binárním formátu, ale tento formát je velmi jednoduchý a navíc jsme na stránce seriálu na webu přichystali ukázky kódu na jeho parsování.

Po načtení SRTM dat dostaneme mřížku nadmořských výšek v metrech pro každou celou úhlovou vteřinu (takže na rovníku by jednotlivé datapointy byly od sebe vzdálené 30 metrů). Jak z toho ale dopočítat nadmořskou výšku pro libovolný bod, který nemusí ležet na jednom ze SRTM datapointů?

Jedna z možností je vzít nejbližší SRTM datapoint (neboli zaokrouhlit souřadnice na celé vteřiny), ale to nám vyrobí něco jako „schody“, což moc neodpovídá reálnému tvaru terénu. Lepší je tedy spočítat výšku podle poměru vzdáleností k nejbližším datapointům – to se dá udělat třeba tak, že nejdříve spočteme výšku podle poměru v jedné ose (čímž 2D problém splácneme do 1D) a pak spočítáme finální výšku pomocí této úsečky a poměru ve druhé ose. Nejlépe je to asi vidět na následujícím obrázku a na ukázce kódu:




```
def altitude(x, y):
  # Spočítáme si okolní mřížkové body
  # (nevadí nám, když některé budou stejné)
  xA = math.floor(x)
  xB = math.ceil(x)
  yA = math.floor(y)
  yB = math.ceil(y)
  topL = SRTM[xA][yA]
  topR = SRTM[xB][yA]
  bottomL = SRTM[xA][yB]
  bottomR = SRTM[xB][yB]

  # Spočítáme si průměr left-right podle x
  # (tím čtverec splácneme do úsečky top-bottom)
  pomer = x - Xa
  top = topL + (topR - topL) * pomer
  bottom = bottomL + (bottomR - bottomL) * pomer

  # Spočítáme si průměr top-bottom podle y
  return top + (bottom - top) * (y - yA)
```

Dvourozměrné pole `SRTM` obsahuje nadmořské výšky v metrech, funkce `altitute` očekává už přepočtené „indexy“ do tohoto pole jako čísla s desetinnou čárkou.

Pojďme získaná výšková data využít pro nalezení nějaké zajímavé cesty.

Úkol 2 [6b]:

Chceme si naplánovat vyjížďku na kole takovou, abychom při ní vůbec nemuseli šlapat. Chceme tedy najít nejdelší trasu vedoucí jenom z kopce.

Najděte nejdelší (počítáno v metrech) posloupnost cest (`way` v řeči OSM) s tagem `highway` (a libovolnou hodnotou tohoto tagu) takovou, že všechny cesty obsažené v této posloupnosti budou jenom klesat a jednotlivé cesty na sebe budou navazovat v koncových bodech (napojování cest jinde než v koncových bodech pro jednoduchost neuvažujme, úlohu by to o něco zkomplikovalo a není to potřeba). Výšková data berte podle SRTM a dopočítávejte výše zmíněným způsobem.

Cestu považujeme za klesající, pokud jeden její konec je ostře níž, než druhý, a zároveň nikde na ní není „údolí“ ani „kopec“ (tedy nenásleduje vyšší bod za nižším, rovinky se ale uvnitř jedné cesty vyskytovat mohou). Na příkladech: cesty $4, 3, 2, 2, 1$ nebo $4, 4, 4, 4, 3$ jsou klesající, ale cesty $4, 3, 2, 1, 2, 1$ nebo $4, 4, 4, 4, 4$ už podle naší definice klesající nejsou (první obsahuje „údolí“ a druhá má oba koncové body ve stejné výšce).

Řešení odevzdávejte jako seznam cest včetně jejich bodů s uvedenými souřadnicemi a nadmořskými výškami v metrech (abychom z toho kdyžtak mohli vykreslit výškový profil a trasu do GPSky). A taky napište, jak dlouhá vaše cesta je (počítejte stejnou formulí, jako v minulé podúloze).

Protože celá Evropa nám na tento úkol přijde příliš velká, tak tuto úlohu spočítejte na malých datech pro Brno a na velkých datech pro ČR (opět použijte mapy vystavené na stránce seriálu na webu, ať máme stejná data).

Bonus: Pokud byste chtěli ještě bonusové body (a třeba nějakou speciální odměnu, kterou autoři seriálu vymyslí), můžete zkusit najít nejlepší takovou trasu v Praze, po které se dá jít pěšky (vhodně si vyfiltrujte hodnoty tagu `highway`). Pokud se nám nějaká bude líbit, tak slibujeme uspořádání KSP procházky po této trase :-).

Rozdělení vstupu

Na poslední úlohu se vám bude hodit technika, kterou jsme zmiňovali už ve vzorovém řešení první série – občas je potřeba si vstupní data (v tomto případě mapu) rozdělit na více částí.

Jedním z důvodů může být urychlení pomocí paralelizace, čímž využijeme více výpočetních jader procesoru, ale úlohu bychom dokázali vyřešit i bez toho, jen nám to ušetří čas. To je dozajista užitečné, ale občas musíme vstupní data rozdělit do více částí už jenom proto, abychom úlohu vůbec zvládli vyřešit. Například když potřebujeme mít data v operační paměti, ale dat je příliš mnoho. Pak je potřeba spočítat, kolik elementů se nám do paměti vejde (například 64-bitové IDčko bodu zabere 8 bytů paměti, takže 128 IDček již zabere 1KB) a pak si vstupní data šikovně rozdělit.

V případě mapy se nabízí dvě možnosti, jak data rozdělovat:

- Podle počtu – rozdělíme si data třeba po jednom milionu bodů. To můžeme udělat, pokud pro spočítání řešení

(alespoň v tomto kroku) nepotřebujeme mít ve společné části elementy, které leží na mapě blízko sebe. Výhoda tohoto rozdělení je, že si můžeme dopředu dobře určit, kolik elementů budou jednotlivé části obsahovat.

- Podle souřadnic – určíme si několik oblastí (třeba „čtverce“ po jednom stupni zeměpisné šířky/délky) a do nich si body rozhazujeme. To má výhodu v tom, že blízké věci zůstanou ve stejné části (musím si jenom dát pozor na věci blízko hranic), ale špatně se dopředu odhaduje, jak budou jednotlivé části velké. To se ale dá vyřešit tím, že si příliš velkou část znovu rozdělím na několik menších, dokud už nemají vhodnou velikost (třeba takovou, aby se vše z nich vešlo do paměti).

Problematické to v případě OSM může být třeba s rozdělováním cest, které u sebe souřadnice přímo uvedené nemají. Můžeme si v takovém případě zkusit načíst všechny cesty do paměti a druhým průchodem k nim dohledávat jejich souřadnice, ale co když se nám do paměti nevejdou? V takovém případě můžeme zkusit zpracovat první milion cest a najít pro ně souřadnice, pak to samé pro druhý milion cest a tak dále, dokud nezpracuji (a nerozdělím) všechny cesty.

V poslední úloze se vám nějaké podobné triky budou velmi pravděpodobně hodit. Tato úloha je taková perlička na konec a je asi nejtěžší ze všech úloh o OSM, které jsme si zatím připravili. Ale všechny potřebné ingredience byste měli již mít k dispozici – a pokud byste třeba něčím chtěli do projektu OSM přispět, tak je to úloha i velmi užitečná.

Úkol 3 [8b]:

Vaším úkolem je spárovat adresní body (`node` s nějakým z tagů `addr:...` – tedy například `addr:city`) s budovami (`way` s tagem `building`).

Řekneme, že budova patří k nějakému adresnímu bodu právě tehdy, pokud se adresní bod nachází uvnitř polygonu vytvořeného cestou znázorňující tvar budovy. Určování, jestli se bod nachází uvnitř polygonu, jsme dělali už ve třetí úloze prvního dílu seriálu, takže se můžete inspirovat třeba ve vzorovém řešení. Větším problémem ale bude množství adresních bodů a množství budov – testovat všechny navzájem asi stačit nebude.

Tuto úlohu spočítejte na malých datech pro Brno a na velkých datech pro celou Evropu (opět použijte mapy, které jsou ke stažení ze stránky seriálu na webu). Zvláště pro Evropu bude výpočet trvat asi celkem dlouho, takže doporučujeme nenechávat úlohu na poslední večer.

Výstup odevzdávejte ve formátu ID adresního bodu a ID budovy oddělené mezerou (každou dvojici na samostatný řádek, ať se nám to dobře kontroluje).

Pro Hrochův Týnec najdete spočítaná data na webu.

A to je z OSM vše. V příštím díle se už pravděpodobně podíváme na nějaká jiná zajímavá data… takže se určitě máte na co těšit :)

Těšíme se na vaše řešení!

Jirka Setnička & Standa Lukeš

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

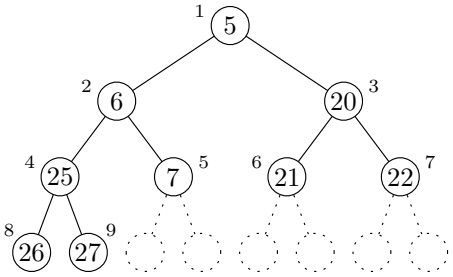
Halda je datová struktura pro uchovávání množiny čísel (či jakýchkoliv jiných prvků, na kterých máme definováno uspořádání, tj. umíme pro každou dvojici prvků říci, který z nich je menší). Tato datová struktura obvykle podporuje následující operace: přidání nového prvku, nalezení nejmenšího prvku a odebrání 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)[6] 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

[6] `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
```

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`.