Korespondenční Seminář z Programování

32. ročník **KSP** **Prosinec 2019**

Milí řešitelé, milé řešitelky!

Druhá série hlavní kategorie letošního ročníku KSP právě skončila, a tak vám přinášíme svá řešení zadaných úloh. Doporučujeme vám si řešení přečíst, mnohdy se tak můžete přiučit zajímavé nové postupy a triky.

Stejně jako v minulém roce plánujeme po opravení vašich řešení vydat komentáře k úlohám, kde shrneme třeba časté chyby, nebo naopak zajímavé alternativní postupy.

Pokud se vám cokoliv nezdá nebo máte nějaký dotaz, neváhejte se ozvat na našem fóru nebo emailem na známou adresu.

Vzorová řešení druhé série třicátého druhého ročníku KSP

32-2-1 Seřízení reaktoru

První zauvažujme, jaký směr nám vlastně může vyjít. Představme si jednotkovou kružnici, na níž leží magnety jako body. Výsledný úhel je pak bod, na který budeme všechny magnety přesouvat. Když si za pomoci výsledného bodu a středu rozdělíme kružnici přímkou na dvě části, dostaneme rozdělení bodů podle směru, kterým se budou pohybovat nejkratší cestou k výsledku.

Představme si nyní, že výsledným bodem pohybujeme. Jak se mění celková vzdálenost pohybu magnetů? Jednak se mění vzdálenosti k cíli v obou půlkružnicích. Druhak se nám mohou magnety překlopit z jedné půlkružnice do druhé. Druhou možnost zatím ignorujme. Pokud se budeme pohybovat v obecné pozici mezi dvěma magnety, bude se celkové množství pohybu měnit pouze lineárně. K bodům na jedné půlkružnici totiž akorát přičítáme velikost změny a na druhé půlkružnici ji zase odečítáme.

Ke zlomu v lineárních změnách může dojít přímo v místě nějakého magnetu, protože daný magnet se přesune z jedné půlkružnice na druhou. Stejně tak k němu může ale dojít přesně naproti nějakému magnetu. Protože tam k překlápění dochází také, akorát na druhé straně kružnice.

Nám tedy stačí uvažovat jako možné výsledné směry pouze orientace magnetů a jejich inverze. Sice v nějakých případech může být stejně dobrým řešením i pozice mezi magnety, ale protože je změna lineární a mění se pouze ve výše vyjmenovaných bodech, bude stejně dobrá i v nějakém bodě. A tím se konečně dostáváme k algoritmu, kterým se dá úloha vyřešit.

Myšlenkou je, že projdeme algoritmem všechny možné výsledné body a z nich vezmeme ten s minimálním nutným posunem. A tohle chceme spočítat rychle. Stačí zopakovat úvahu provedenou výše s body na kružnici a řešení se začne nabízet.

Vyrobíme si seznam všech možných výsledných bodů. Těch je nejvýše $2N$, když N je počet magnetů. Tento seznam si setřídíme a postupně ho budeme procházet.

Začneme nejmenším možným úhlem. Rozdělíme si magnety na dvě části, podle toho, na jaké půlkružnici jako leží. Počet magnetů v levé části nazveme N_ℓ a v pravé N_p. Pro každou půlkružnici spočítáme celkový vyžadovaný posun, nazvěme jej R_p, tj. celková rotace pravé půlkružnice, a R_ℓ. Neděláme to nijak složitě, sekvenčně projdeme všechny magnety. Hledáme pak bod, který minimalizuje $R_p + R_\ell$.

Máme spočítanou první možnou výslednou hodnotu, teď ji chceme aktualizovat na další. Je v seznamu nejbližší vyšší, jdeme po kružnici ve směru hodinových ručiček. Nový celkový posun spočítáme z velikosti rozdílu mezi tímto a posledním uvažovaným cílovým bodem. Nazvěme tento úhel d. Pak:

- Nové R_p spočteme ze starého $R_p - d \cdot N_p$, protože se ke všem magnetům v pravé půlkružnici přibližujeme.
- Nové R_ℓ spočteme ze starého $R_\ell + d \cdot N_\ell$, zde se naopak oddalujeme.

Musíme ještě vyřešit, že se nám jeden magnet přesouvá do druhé poloviny. To určitě nastane, protože jsme si tak nadefinovali testované úhly. Jsou akorát dvě možnosti, kde k tomu může dojít, tak vyzkoušíme obě a bod najdeme a přeřadíme ho do druhé půlkružnice. Tzn. snížíme/zvýšíme N_ℓ a N_p.

Tímto způsobem můžeme projít všechny body, u každého si spočítat R_p+R_ℓ a vybrat ten, kde je tento výraz minimální. Dostaneme tím naši výslednou orientaci magnetů.

Jak složité to celé je? V $\mathcal{O}(N \log N)$ třídíme body, pak první bod počítáme v $\mathcal{O}(N)$, ostatní v $\mathcal{O}(1)$. Tzn. celkově nám vychází $\mathcal{O}(N \log N)$. Paměti při tom použijeme pouze $\mathcal{O}(N)$.

Vašek Šraier

32-2-2 Mezihvězdné jízdní řády

Úlohu lze převést na hledání nejkratší cesty v grafu. Vrcholy budou jednotlivé planety, hrany linky mezi nimi, které budou ohodnocené podle toho, jak dlouho trvá přelet danými mezi planetami. Tento přístup má však velký nedostatek: různé hrany (linky) jdou použít v různý čas, a navíc se může stát, že na planetě musíme nějakou dobu čekat na přestup.

Jedním řešením může být, že místo aby byla planeta reprezentována jedním vrcholem, tak bude mít vrcholů mnoho: jeden pro každý časový okamžik, kdy se na planetě stane něco pro nás zajímavého, tedy když nějaký spoj přiletí nebo odletí. Vrcholy jedné planety budou spojeny hranami tak, aby šlo na planetě „čekat“, tedy posunout se do jiného vrcholu planety, který reprezentuje pozdější časový okamžik. Linky poté spojují vrcholy planet odpovídající času odletu a příletu. Hrany jsou také orientované, aby se šlo v čase pohybovat pouze dopředu.

Stále je tu ale mnohem větší problém: díky tomu, že jsou linky periodické, je tento graf nekonečný.

Obě komplikace se dají vyřešit tím, že budeme graf generovat dle potřeby za běhu programu, místo abychom ho celý vytvořili předem.

Pro každou planetu budeme mít jediný vrchol. Nebudeme si pro něj pamatovat přímo hrany, ale jen informace o linkách, které jím procházejí. Konkrétně nás pro každou linku procházející vrcholem zajímá její perioda, kdy z tohoto vrcholu poprvé odlétá, následující vrchol po současném a jak dlouho trvá dostat se na něj. S těmito informacemi si dokážeme pro libovolný čas spočítat, kdy nejdřív se pomocí této linky dostaneme na její následující vrchol.

Nyní použijeme nějaký klasický algoritmus pro nalezení nejkratší cesty, například Dijkstrův algoritmus.[1] Pokaždé, když se v těchto algoritmech se ptáme na hrany vedoucí z vrcholu, tak známe délku cesty do tohoto vrcholu, tedy kdy nejdřív se tam umíme dostat, tudíž si pro každý dotaz můžeme relevantní hrany vygenerovat. Jinak algoritmus použijeme takřka beze změny oproti své verzi pro klasické grafy.

Též byla v úloze komplikace, že počet planet a jejich čísla mohla být velmi velká, takže pole indexované číslem planety by se nevešlo do paměti. Ve skutečnosti nám stačí si ukládat „zajímavé“ planety, přes které vede nějaká linka, kterých je stále rozumné množství. Toto lze implementovat například přečíslováním planet tak, aby jejich číslo bylo indexem v poli, kde jsou pouze zajímavé planety.

Kuba Pelc

32-2-3 Nádrž na částice

Existuje bezpočet kvadratických a $n \log n$ řešení. Úlohu je taktéž možné řešit v průměrně lineárním čase s pomocí hešování. My si ukážeme pouze čistě lineární řešení.

Nejdříve si spočítáme prefixové součty (viz kuchařka)[2] výšek sloupců krajiny. Pořídíme si zásobník na sloupečky krajiny. Budeme v něm uchovávat kandidáty na levý okraj prohlubně jako uspořádané dvojice (výška, pořadí sloupce). Výšky na zásobníku budou vzestupné.

Začneme od levého konce krajiny a přidáme do zásobníku uspořádanou dvojici (výška prvního sloupce, pořadí sloupce tj. 0). Na každém dalším sloupci zkontrolujeme, zda nemůže tvořit pravý okraj prohlubně dané velikosti. Podíváme se na vršek zásobníku (není-li prázdný) a odstraníme postupně všechny sloupce nižší než kandidát na pravý okraj. Tyto sloupce již nemohou tvořit levý okraj pro žádnou prohlubeň končící za tímto sloupcem. Dostaneme-li na zásobník sloupec, který může tvořit levý okraj k našemu kandidátovi na pravý okraj, z prefixových součtů a výšky okrajů snadno zjistíme, jakou by tato nádrž měla kapacitu.

V případě, že jsme našli správnou kapacitu, skončíme a odpovíme souřadnice okrajů. Jinak na zásobník vložíme našeho kandidáta na pravý okraj, teď to bude naopak další kandidát na levý okraj.

Zřejmě počítání prefixových součtů je lineární, jeden výpočet kapacity je pak konstantní. Počet operací se zásobníkem je celkově lineárním s počtem sloupců, neb každý nejvýše jednou přidáme a nejvýše jednou odebereme. Dohromady tak získáváme lineární řešení.

Jiří Škrobánek

32-2-4 Družicová komunikace

Pamatujme na grafovou reprezentaci problému. Hranami myslíme komunikační kanály mezi družicemi a každá hrana má přidělenou nějakou odolnost. Komponenty, které počítáme, jsou komponenty zadaného grafu po odstranění hran s odolností nižší, než je intenzita rušení. Nechť vrcholů je N a hran M jako vždy. Pro zadanou intenzitu rušení má každá družice jednu komponentu, do které náleží, a pro efektivní ukládání takových vztahů se hodí datová struktura DFU (Disjoint-Find-Union). Můžete se s ní seznámit v naší kuchařce o minimálních kostrách,[3] pro jejichž nalezení se také dá použít.

Jak DFU použijeme my? Začneme se samostatnou komponentou pro každý vrchol a zavedeme si proměnnou k značící „momentální“ počet komponent. Pro začátek $k \leftarrow N$, což odpovídá intenzitě rušení převyšující odolnost všech hran. Hrany setřídíme sestupně podle odolnosti, spočítáme, kolika různých hodnot odolnost nabývá – nechť je to H a sestrojíme pole P velikosti H tvořené dvojicemi (odolnost, počet komponent). Nyní setříděnými hranami iterujeme následovně.

Jakmile narazíme na novou hodnotu odolnosti, podíváme se na všechny hrany, které tuto odolnost mají, a pokud vrcholy, které hrana spojuje, leží v jiné komponentě (což nám řekne *find*), provedeme na jejich komponentách *union* a snížíme naši proměnnou k o jedna. Jakmile to provedeme pro všechny hrany s příslušnou odolností, uložíme současné k na nový index v P a iterujeme dál.

Tím jsme v P získali sestupně setříděnou posloupnost všech vyskytujících se odolností a k nim počet komponent, na které se družicová síť rozpadne při rušení intenzity rovné dané odolnosti. Při dané intenzitě rušení v grafu přetrvávají ty hrany, které mají odolnost alespoň rovnu intenzitě, což jsme přesně simulovali popsanou iterací – začali jsme s pomyslnou intenzitou přesahující odolnost všech hran a vždy jsme propojovali ty komponenty, které nám nová (nižší) intenzita umožní do grafu přidat. A získaný počet komponent odpovídá taky počtu při nižších intenzitách, které ještě neodpovídají následující nižší odolnosti hran.

To nám umožňuje pro každý dotaz Q binárním vyhledáváním v P najít takovou nejnižší odolnost, která je rovna alespoň Q a odpovědět s příslušným počtem komponent (nebo odpovědět N, pokud Q převyšuje odolnost nejodolnější hrany).

Jak dlouho to pak celé bude trvat? K setřídění hran spotřebujeme $\mathcal{O}(M \log M)$ času. Dále M hranami iterujeme a v každém kroce použijeme dvakrát *find* a potenciálně jeden *union*, asymptoticky na to potřebujeme $\mathcal{O}(M\alpha(N))$ času, kde α je inverzní Ackermannova funkce. A nakonec na každý dotaz potřebujeme $\mathcal{O}(\log M)$ času pro binární vyhledávání (různých hodnot odolnosti bude maximálně M). Zadání, říká, že dotazů bude $\Theta(M + N)$, to dává $\mathcal{O}((M + N) \log M)$, což je taky celková asymptotická složitost, protože předchozí členy jsou samy v $\mathcal{O}((M + N) \log M)$. Ještě bychom si mohli všimnout, že počet komponent klesne maximálně $(N - 1)$-krát, a nové počty do P ukládat jen při změně k, čímž se dostaneme na složitost $\mathcal{O}((M+N) \log N)$, asymptoticky jsme si ale nepomohli, protože M je $\mathcal{O}(N^2)$,

[1] `http://ksp.mff.cuni.cz/viz/kucharky/halda-a-cesty`
[2] `http://ksp.mff.cuni.cz/viz/kucharky/zakladni`
[3] `http://ksp.mff.cuni.cz/viz/kucharky/minimalni-kostry`

takže $\log M$ je $\mathcal{O}(2\log N) = \mathcal{O}(\log N)$.

Ještě se můžete podívat na příklad možné implementace s jednoduchou implementací DFU. Předpokládaný formát vstupu je popsaný před funkcí main.

Program (C++):
`http://ksp.mff.cuni.cz/viz/32-2-4.cpp`

Všimněme si, že v jednodušší verzi jsme mohli kromě hran setřídit sestupně i dotazy a místo ukládání počtu komponent pro dané odolnosti rovnou postupovat podle hodnot Q a získávat rovnou odpovědi na dotazy. Tím bychom se zbavili binárního vyhledávání, ale museli bychom setřídit dotazy, čímž bychom se dostali na $\mathcal{O}((M+N)\log(M+N))$, což je nakonec asymptoticky stejný odhad.

Martin Koreček

32-2-5 Štítové emitory

Nejprve si položme otázku, která čísla K je možné zapsat ve tvaru $\pm 1 \pm 2 \ldots \pm N$. Největší K dostaneme, pokud budou všechna znaménka kladná. Tehdy vyjde součet $1+2+\ldots+N = N\cdot(N+1)/2$, kterému budeme říkat S_N. Obdobně nejmenší součet dostaneme pro všechna znaménka záporná a bude to $-S_N$.

Také si všimneme, že pro pevné N mají všechna zapsatelná K stejnou paritu: tedy jsou buď všechna sudá, nebo všechna lichá. To proto, že překlopením znaménka u čísla x se součet změní o $2x$, což je sudé číslo. Takže K musí mít stejnou paritu jako S_N. (Tato parita je mimochodem jednoznačně určená zbytkem N mod 4.)

Teď ukážeme, že tyto dvě nutné podmínky jsou také postačující. Tedy že množina T_N čísel zapsatelných pomocí $\pm 1 \ldots \pm N$ je $\{-S_N \mid S_N\}$, kde $\{a \mid b\}$ značí „děravý interval“ čísel od a do b s krokem 2.

Toto tvrzení dokážeme indukcí podle N. Pro $N=1$ je $S_N = 1$ a $T_N = \{-1,1\} = \{-1 \mid 1\}$. Nechť nyní tvrzení platí pro $N-1$ a chceme ukázat, že platí i pro N. Pokud se K dá zapsat pomocí $\pm 1 \ldots \pm N$, musí vzniknout přičtením $\pm N$ k nějakému číslu zapsatelnému pomocí $\pm 1 \ldots \pm (N-1)$. Proto je T_N sjednocením dvou kopií T_{N-1}: jedné posunuté o $-N$, druhé o $+N$. Podle indukčního předpokladu to tedy můžeme zapsat jako

$$\{-S_{N-1}-N \mid S_{N-1}-N\} \cup \{-S_{N-1}+N \mid S_{N-1}+N\}.$$

Ukážeme, že toto sjednocení dvou intervalů dá přesně interval $\{-S_N - N \mid S_N + N\}$. V obou jsou čísla se stejnou paritou (z levého intervalu totiž dostaneme převrácením znamének pravý interval napsaný pozpátku). Maximum pravého intervalu je rovno $S_{N-1}+N$, což je S_N; podobně minimum levého intervalu je $-S_N$.

Ještě musíme ověřit, že mezi levým a pravým intervalem není žádná díra. To by šlo upočítat upravováním výrazů, ale raději trochu přemýšlejme. Pro $N \geq 3$ je maximum levého intervalu větší nebo rovno 0 (to proto, že $S_{N-1} = 1+\ldots+(N-1) \geq N$) a minimum pravého je symetricky menší nebo rovno 0. Proto levý neskončí dřív, než pravý začne. Pro případ $N=2$ snadno ověříme, že levý interval je $\{-3,-1\}$, zatímco pravý $\{1,3\}$, takže na sebe dokonale navazují.

Víme tedy, že zapsat jdou právě čísla $\{-S_N \mid S_N\}$. Jak teď najít zápis konkrétního K? To půjde hladovým algoritmem. Úplně triviálním: pokud dostaneme číslo $K \in T_N$, podíváme se na jeho znaménko a podle toho rozhodneme, zda se v zápisu objeví $+N$ nebo $-N$. Pokud $K \geq 0$, použijeme $+N$, čímž problém převedeme na zapsání čísla $K-N$ pomocí $\pm 1 \ldots \pm (N-1)$. A symetricky pro $K<0$ napíšeme $-N$ a pokračujeme zapisováním čísla $K+N$.

Proč to funguje? Stačí si vzpomenout, jak jsme Z_N získali jako sjednocení dvou posunutých kopií Z_{N-1}. Ta pravá obsahovala všechna kladná čísla ze Z_N (a možná nějaká záporná), v té levé ležela všechna záporná čísla (a možná nějaká kladná). Takže zařazením $K>0$ do pravé kopie jsme nemohli nic pokazit a stejně ani zařazením $K<0$ do levé. V případě, že $K=0$, si můžeme vybrat libovolně, protože ze levá a pravá kopie se liší jen znaménkem a pořadím, takže pokud 0 leží v jedné z kopií, musí i v té druhé.

Zbývá rozebrat časovou a paměťovou složitost. Samotné zjištění, zda je K zapsatelné, stihneme v konstantním čase i paměti. Konstrukce zápisu trvá $\mathcal{O}(N)$ a pokud dovolíme vypisovat výsledek od největšího sčítance k nejmenšímu, stále stačí konstantní paměť.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/32-2-5.py`

Martin „Medvěd“ Mareš

32-2-6 Mapy z kOSMu

Řešení druhého dílu seriálu je (stejně jako u prvního dílu) dosti technické a tak vás opět budou zajímat asi hlavně zdrojové kódy. Ale doporučujeme vám se začíst i do textového řešení, kde se pokusíme shrnout nějaké principy a nástrahy, se kterými jste se při řešení mohli potkat.

Úkol 1 – Délka kolejí [3b]:

První úloha byla jednoduchá a byla o tom hlavně si vyzkoušet přepočet vzdálenosti s použitím haversinové formule. Tato formule není úplně přesná, protože považuje Zemi za velkou kouli, ale pro naše použití bohatě dostačovala.

Řešení bylo opět potřeba udělat dvouprůchodově, ale tento „trik“ jsme se naučili již v minulém dílu seriálu. Při prvním průchodu jsme si nalezli všechny cesty s tagem `railway=rail` (což jsou cesty značící jednotlivé koleje, podívejte se třeba na nějaké velké nádraží) a z těchto cest jsme si zapamatovali ID bodů. Ve druhém průchodu jsme pak k bodům nalezli odpovídající souřadnice.

Délku všech kolejí jsme pak již spočítali jednoduše, kdy jsme počítali (pomocí výše zmíněné formule) součet vzdáleností mezi sousedními dvojicemi bodů cesty.

Doplníme, že kdyby nějaká kolej byla zmapována špatně (v zatáčce namísto oblouku jenom lomená čára se zlomem v jednom bodě), tak by námi spočítaná vzdálenost samozřejmě neodpovídala skutečnosti, ale v mapách bývají oblouky tratí vyznačené rozumně, takže nám tyto nepřesnosti nevadí.

Součet délek všech kolejí v Brně nám vyšel 331,178 km, v celé Evropě pak 614 641,426 km (spočítat nám to jednovláknově zabralo 4 hodiny a 36 minut).

Program (Go):
`http://ksp.mff.cuni.cz/viz/32-2-6-1-koleje.go`

Úkol 2 – Klesající cesta [6b]:

Největší záludnosti tohoto úkolu byly dvě – správné napasování SRTM dat na souřadnice a pak sestavení správného grafu, abychom v něm mohli najít nejdelší klesající cestu (pro naší cyklistickou vyjížďku jenom z kopce). Pojďme záludnosti vyřešit postupně.

Data ze SRTM mapování, která jsme vám dali k dispozici, měla celkem jednoduchý binární formát a na technické stránce seriálu na webu jsme dokonce dali ukázku jak tento jednoduchý binární formát načíst (v podstatě je to čtverec 3601×3601 dvoubajtových výšek). Jak ale z toho nějak elegantně vyrobit něco (například nějaký objekt), kterého se půjde ptát na výšku libovolného bodu v něm?

My jsme na to šli tak, že jsme si nejprve zjistili „obálku“ aneb jaké všechny čtverce budeme potřebovat načíst – tu jsme zjistili podle nejvýchodnějšího, nejzápadnějšího, nejsevernějšího a nejjižnějšího bodu a vzali jsme všechny čtverce z tohoto obdélníku. Dost možná tím načteme i nějaké SRTM čtverce, ve kterých žádný ze zkoumaných bodů nebude, ale data jsou to relativně malá a můžeme si je v případě celého obdélníku jednoduše uložit do dvourozměrného pole a v konstantním čase se na ně dotazovat. Dvě věci, na které musíme dát při načítání pozor, je správná orientace čtverců a pak také to, že se dva sousední čtverce o jeden bod překrývají.

Pro dotazy na výšku konkrétního bodu si musíme pamatovat ještě dvě další věcí – jaký je offset levého spodního rohu naší tabulky (neboli na které zeměpisné šířce a délce začíná) a jaký je přepočet zeměpisných souřadnic na indexy v naší tabulce. V našem případě byl přepočet na stupně $1/3600$, neboli posun o jeden index v tabulce dál je posun o $1/3600$ stupně v dané ose. Výpočet tak může vypadat třeba takto:

```
latIndex = (lat - latStart) / stepsize
lonIndex = (lon - lonStart) / stepsize
```

Většinou nám ale indexy v tabulce nevyjdou jako celá čísla, ale náš dotaz se trefí „někam mezi“. I na toto jsme pamatovali, a proto jsme vám v zadání dali k dispozici krátkou funkci, která spočítá hodnotu podle okolních mřížkových bodů, pro připomenutí se když tak podívejte do zadání.

Nyní už bychom měli mít připravený jednoduchý objekt, do kterého načteme SRTM data a kterého se pak můžeme ptát na výšky libovolných bodů.

Jak tedy zjistíme výšky cest? Náš program bude mít klasické dvojprůchodového schéma – v prvním průchodu najdeme žádané cesty a zapamatujeme si jejich body. Tady jsme vám nechali trochu volnost v tom, jaké cesty s tagem `highway` budete uvažovat, my jsme ve vzorovém řešení vzali všechny s výjimkou `highway=motorway` a `highway=trunk` (dálnice a víceproudé silnice). Ve druhém průchodu pak nalezneme souřadnice všech bodů a uložíme si je.

A teď přichází druhá těžší část – sestavení si správného grafu. Naším cílem bude sestavit si DAG neboli orientovaný acyklický graf – vrcholy budou křižovatky a hrany budou klesající cesty (orientované od vyššího k nižšímu bodu). Vrcholy si navíc budou pamatovat, jaká nejdelší klesající cesta v nich končí.

Pak můžeme projít cestu po cestě a filtrovat z nich takové, které splňují naše požadavky – jeden jejich konec je níž než druhý a současně jsou monotónní (tedy v klesajícím směru je každý další node cesty nejvýše tak vysoko jako předcházející). Pokud cesta projde tímto filtrem, tak spočítáme její délku a přidáme ji jako hranu do konstruovaného grafu. My jsme navíc od cest vyžadovali v ukázkovém řešení i nějaké minimální klesání na 100 metrech, přesněji rozdíl jednoho metru mezi počátečním a koncovým bodem za každých 100 metrů délky.

Po zkonstruování celého grafu si do fronty ke zpracování vložíme všechny vrcholy, do kterých nevede žádná hrana, a z nich zahájíme prohledávání: Odebereme vrchol v z fronty, podíváme se na všechny jeho sousedy po klesajících hranách a pokud v některém z nich umíme dosáhnout delší klesající cesty, než u něj máme poznamenáno (neboli že nejdelší klesající cesta končící ve v plus délka aktuálně zpracovávané hrany je delší, než doposud známá nejdelší klesající cesta končící v sousedovi), tak cestu aktualizujeme na delší. Současně také tomuto sousedovi snížíme vstupní stupeň na nulu a pokud dosáhne nuly, tak ho přidáme do fronty ke zpracování.

Tímto projdeme všechny vrcholy (zamyslete se proč) a každý zpracujeme právě jednou. Když pak najdeme maximum, máme nejdelší cestu. Tuto cestu můžeme vypsat jako index na sebe navazujících cest v OSM, nebo třeba vyrobit GPX soubor, který se dá dobře prohlížet například v Mapách.cz.

Námi nalezené nejdelší klesající cesty (s parametry popsanými výše) jsou:

- Brno: `vyjizdka_brno.gpx` 1692,9 metrů
- Česká republika: `vyjizdka_CR.gpx` 5670,4 metrů

Program (Go):
`http://ksp.mff.cuni.cz/viz/32-2-6-2-vyjizdka.go`

Úkol 3 – Párování adres [8b]:

Párování adres byla už o poznání složitější úloha. Možných řešení bylo několik, jsme zvědaví, na co jste přišli :)

Kdyby nám nevadilo, že to bude pomalé, můžeme použít znalosti z předchozího dílu a využít funkci na určování, jestli je bod v polygonu. Budovy jsou totiž vlastně polygony a adresy jsou body, které k nim máme přiřadit. Můžeme projít všechny adresy a porovnat se všemi budovami. Asi vás nepřekvapí, že to bude mít kvadratickou složitost, a bude to tak dost pomalé. Nicméně, když si všechno načteme do paměti, tak pro Brno by toto řešení mělo doběhnout.

Aby nám to doběhlo pro celou Evropu, tak budeme potřebovat trochu sofistikovanější řešení. Víme, s jakými daty počítáme, a tak můžeme bez problému předpokládat, že většina budov jsou malé polygony, velmi vzácně se překrývají a jsou docela rovnoměrně rozmístěné po celé mapě (přesně to jsme počítali v minulém díle :)). Můžeme si tak mapu rozdělit na čtverce a budovy si postupně přiřadíme ke všem čtvercům, které je obsahují. Když budeme procházet adresy, tak stačí se dívat jen do toho čtverce, kam spadá její bod, nemusíme procházet všechno. Jakou to ale bude mít složitost záleží na velikosti čtverců – čím více čtverců, tím více sežereme paměti a tím méně budeme muset pro každý bod porovnat polygonů.

Jak poznat, do jakých čtverců spadá jedna budova? Abstraktněji řešíme problém, jestli má čtverec a mnohoúhelník nějaký průnik. Je jasné, že když čtverec obsahuje alespoň jeden z bodů polygonu, tak určitě nějaký průnik mají. Co když ale čtvercem jenom prochází hrana mnohoúhelníka nebo je dokonce celý uvnitř? Pak zase mnohoúhelník obsahuje alespoň jeden z krajních bodů čtverce.

My ale potřebujeme najít všechny čtverce ve kterých je daná budova a rozhodně kvůli tomu nechceme iterovat přes celou síť. Tady nám pomůže starý dobrý princip „rozděl a panuj“ a rekurze. Rozdělíme si celou plochu na 2×2 čtverce a zjistíme, se kterými má budova nějaký průnik. Na tyto se zavoláme rekurzivně a budeme to dělat, dokud nám nebude připadat, že jsou čtverce dostatečně malé. Sice se v nejhorším případě můžeme rekurzit do všech čtverců najednou a

vlastně si tak nepomoci oproti kompletnímu průchodu, ale můžeme si dovolit předpokládat, že budovy budou rozumně velké, a tak nebudou často zabírat víc čtverců najednou.

Když už hledáme čtverce rekurzí, můžeme rekurzivní udělat celou datovou strukturu. Tím si ušetříme hledání optimální velikosti čtverce a nebudeme tolik závislí na rovnoměrnosti rozložení budov po mapě (se kterou taky není tak slavné :)). Vyrobíme si tak strukturu, které se říká kvadrantový strom neboli quadtree,[4] akorát budeme občas přiřazovat budovy do více listů najednou.

Takže nejdřív načteme všechny budovy, vyrobíme z nich stromek a pak budeme jenom vyhledávat adresy. Když najdeme, vypíšeme řádek na výstup… až na to, že nám dojde paměť. Budov je totiž v Evropě moc (jak víte z předchozího dílu), a tak budeme muset na něčem ušetříme. Vezmeme znovu stejné zbraně a rozdělíme si vstup na menší. Problém je jenom s budovami a tak vždy načteme jenom jejich část, aby se nám vešly do paměti. Pro každý fragment budov musíme samozřejmě projít všechny adresy, takže s klesající paměťovou náročností se zvedne náročnost časová.

Když si rozdělíme vstup na K částí, tak paměťová náročnost bude $O(N/K)$ a časová zhruba $O(NK)$.

Jirka Setnička & Standa Lukeš

[4] `https://en.wikipedia.org/wiki/Quadtree`