Korespondenční Seminář z Programování

34. ročník | **KSP** | **Prosinec 2021**

Vzorová řešení druhé série třicátého čtvrtého ročníku KSP

34-2-1 Skupinová jízdenka

Označme r cenu jednotlivého jízdného a s cenu skupinového jízdného. Potom cena, kterou Alice a Bob zaplatí, pokud se setkají v nějakém vrcholu M, je rovna $s \cdot d(M,C) + r \cdot d(A,M) + r \cdot d(B,M)$, kde $d(M,C)$ značí vzdálenost mezi vrcholy M a C, tedy délku nejkratší cesty mezi nimi. A podobně pro ostatní vrcholy. To platí, jelikož se jedná o součet cen, jak se do vrcholu dostat z A a B, a ceny, jak se společně dostat do C.

Spuštěním Dijkstrova algoritmu z města C jsme schopni pro každý vrchol M zjistit vzdálenost $d(C,M)$. Pokud tuto vzdálenost vynásobíme s, dostáváme cenu, za kterou se Alice s Bobem dostanou z M do C. Podobně můžeme pomocí Dijkstrova algoritmu puštěného z bodu A získat vzdálenost $d(A,M)$ a následným vynásobením r minimální cenu z A do M. Stejně tak spuštěním z B získáme vzdálenost $d(B,M)$ a z ní cenu $r \cdot d(B,M)$.

Nyní v každém vrcholu známe cenu, za jakou se do něj umí dostat Alice, za jakou se do něj umí dostat Bob, a kolik je bude stát se společně dopravit do Chebu. Pokud tyto hodnoty sečteme, dostaneme celkovou cenu výletu, pokud by se setkali ve vrcholu M. Jelikož se mohou setkat v kterémkoliv vrcholu a nás zajímá nejnižší cena výletu, vybereme z těchto součtů ten minimální.

Časově nejnáročnější operací je Dijkstrův algoritmus. Pokud bychom použili přímočarou implementaci, bude celková časová složitost $\mathcal{O}(n^2)$. Můžeme ale použít binární haldu a tím Dijkstrův algoritmus zrychlit na $\mathcal{O}((m+n)\log n)$. Prostorová složitost je $\mathcal{O}(n+m)$.

Program (C++):
`http://ksp.mff.cuni.cz/viz/34-2-1.cpp`

Úlohu připravili: Petr Budai, Jirka Kalvoda, Martin „Medvěd“ Mareš, Kuba Pelc, Ondra Sladký

34-2-2 Lezení

Úloha má několik různých přístupů, kterými se dá dosáhnout rychlého řešení, my si ukážeme hladový algoritmus.

Označme si celkový počet trestných bodů i-tého lezce S_i, počet trestných bodů za první dvě disciplíny A_i (to je součin pořadí v prvních dvou disciplínách, který umíme spočítat) a hypotetické pořadí v lezení na obtížnost jako B_i (tedy $S_i = A_i B_i$). Náš lezec bude ten s indexem 0.

My nyní hledáme nejhorší možné pořadí lezce 0, tedy největší $k+1$ takové, že k lezců mělo nižší počet trestných bodů než náš lezec. Nyní se podívejme, za jakých podmínek může i-tý lezec porazit lezce 0:

$$S_i < S_0$$
$$A_i B_i < A_0 B_0$$
$$B_i < \frac{A_0 B_0}{A_i}$$

Všechny hodnoty na pravé straně známe (A máme spočtená a $B_0 = 1$). Pro každého lezce tak umíme spočítat, kolikátý nejhůře může skončit, aby porazil našeho lezce. Označme toto pořadí P_i.

To již vede na hladové řešení. Pro každé pořadí v poslední disciplíně $B > 1$ najdeme ty lezce, pro které platí $B \leq P_i$. Z těchto lezců přidělíme pořadí B tomu s minimálním P_i. Přidělování pořadí umíme provést rychle, pokud si lezce setřídíme vzestupně podle P_i. Pak stačí jeden sekvenční průchod tímto polem: pokud podmínka $B \leq P_i$ platí, danému lezci můžeme rovnou přidělit pořadí B, protože lezci po něm mají vyšší P_i nebo stejné. V opačném případě víme, že lezce již nemůžeme umístit tak, aby našeho porazil, můžeme jej tedy umístit na nejhorší možnou pozici, aniž bychom cokoliv pokazili.

Tím zjistíme počet lezců, kteří toho našeho překonali. Jako výsledek vrátíme tuto hodnotu + 1.

Jelikož třídíme hodnoty v rozsahu 1 až n^2, můžeme si je představit jako čísla o základu n, která mají nejvýše dvě cifry. Ta pak můžeme třídit pomocí radix-sortu v lineárním čase vzhledem k n. Zbytek algoritmu jsou jen sekvenční průchody, proto je celková časová složitost $\mathcal{O}(n)$ (stejně tak paměťová).

Zbývá ještě ukázat, že hladový algoritmus skutečně funguje. Pořadí lezců, které najde, určitě může nastat. Potřebujeme ale ukázat, že více lezců už toho našeho porazit nemůže.

To nahlédneme sporem. Uvažme nějaké uspořádání v poslední disciplíně takové, že $l > k$ lezců bude mít nižší skóre než náš lezec. BÚNO předpokládejme, že ti lezci, kteří jej porazí, se nachází na pozicích 2 až $l+1$. Kdyby ne, můžeme je na tyto pozice umístit a jejich skóre jen zlepšit.

Nyní se zaměřme na první místo, kde se toto uspořádání liší od toho, které generuje náš algoritmus. Uvědomme si, že lezec, kterého na toto místo přiřadil náš algoritmus, je v údajně lepší posloupnosti buď později, anebo není mezi l nejlepšími. V obou případech však můžeme dané lezce prohodit a nic se tím nepokazí.

Takto můžeme induktivně upravovat, až první místo, kde se posloupnosti budou lišit, je $(k+2)$-té místo, kde má být v údajně lepší posloupnosti lezec, který stále toho našeho porazí. To ale není možné, jelikož žádný takový nezbyl. Jelikož jsme sadou úprav, kterými jsme nic nemohli pokazit, dospěli k něčemu, co zjevně nemůže existovat, nemůže existovat ani žádné uspořádání lepší než to, které našel hladový algoritmus.

To ale znamená, že hladový přístup je korektní.

Úlohu připravil: Ondra Sladký

34-2-3 Na Hroších pláních

Na vstupu jsme dostali dva claimy ve tvaru mnohoúhelníka. Hranice každého claimu je tvořena lomenou čárou, což je posloupnost hran (úseček) napojených ve vrcholech (bodech).

Abychom si úvahy zjednodušili, předpokládejme, že žádné dva vrcholy nemají stejnou y-ovou souřadnici. Tím pádem

žádná hrana claimu není vodorovná. Později dořešíme, co si počít, když to vstup nesplňuje.

Také si všimneme, že existují dva způsoby, jak mohou mít dva claimy společný bod: buďto se protnou jejich hranice (to jsou nějaké lomené čáry), nebo leží jeden claim celý uvnitř druhého.

První případ poznáme podle průsečíků hran claimů. Druhý tak, že jeden (pevně zvolený) z vrcholů claimu leží uvnitř druhého claimu. To někdy nastane i v prvním případě, ale to nám nevadí, tak jako tak je to společný bod.

Vrchol uvnitř druhého claimu

Pro každý claim A zvolíme nějaký jeho vrchol x a otestujeme, zda leží uvnitř druhého claimu B. Vedeme z x polopřímku libovolným směrem a počítáme, kolik hran claimu B protne: pokud sudý počet, je x venku, jinak uvnitř.

Pokud se polopřímka strefí do vrcholu claimu B, podíváme se na sousední vrcholy. Pokud oba leží na různých stranách polopřímky, zjevně jsme proťali hranici claimu, takže to počítáme jako 1 průsečík. Pokud oba leží na stejných stranách, tak jsme o hranici jen „zavadili“, tudíž to počitat nechceme.

Ještě větší komplikace nastane, pokud nějaká hrana, nebo dokonce posloupnost několika po sobě jdoucích hran, leží přímo v polopřímce. Tehdy se podíváme na vrcholy těsně před touto posloupností a těsně za ní a provedeme stejnou úvahu jako při strefení se do vrcholu.

Tato část algoritmu běží v lineárním čase.

Průsečíky hran

Průsečíky hran budeme hledat zametáním roviny. Budeme posouvat vodorovnou přímku (*koště*) shora dolů a sledovat, jak se protíná se zadanými claimy (mnohoúhelníky). Budeme si udržovat *průřez* – seznam hran claimu proťatých koštětem, seřazený tak, aby průsečíky s koštětem šly zleva doprava.

Všimneme si, že než se nějaké dvě hrany protnou, musí se nutně v průřezu stát sousedními. Stačí tedy pokaždé, když upravíme průřez, zjistit, jaké dvojice hran se staly sousedními, a pro ně otestovat, zda se někde pod koštětem protnou.

Koštětem není potřeba pohybovat spojitě – zajímavé věci se dějí jenom tehdy, projde-li koště vrcholem claimu. Proto si seřadíme vrcholy shora dolů a budeme koštětem „skákat“ po jejich y-ových souřadnicích. U každého vrcholu si také zapamatujeme, ke kterému claimu patří a které dvě hrany se v něm potkávají.

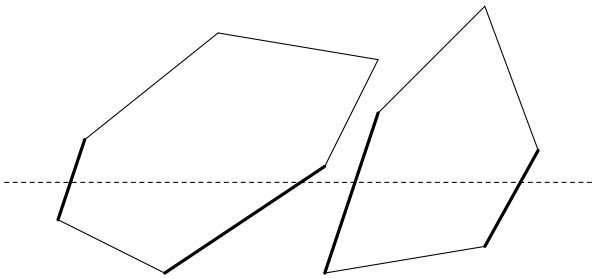
Konvexní případ

Pro konvexní claimy je zametání jednoduché.

Představme si jeden konvexní claim s vrcholy v obecné poloze. Claim má jednoznačně určený nejvyšší a nejnižší bod. V těchto bodech můžeme okraj claimu rozdělit na levou a pravou část.

Při zametání claimu narazí koště nejprve na nejvyšší bod, pak bude nějakou dobu průřez obsahovat jednu hranu z levého okraje a jednu z pravého, až nakonec koště v nejnižším bodě claim opustí. Údržba průřezu tedy bude jednoduchá: v nejvyšším bodě přidáme nejvyšší dvě hrany. Pak v každém vrcholu jedna hrana skončí a druhá začne, takže z průřezu jednu odstraníme a druhou přidáme. Nakonec v nejnižším bodě zrušíme obě hrany.

Nyní si představíme, že zametáme dva claimy současně a udržujeme společný průřez. Vkládání a mazání hran provádíme pro každý claim zvlášť, jak jsme popsali. Ale kdykoliv se stanou sousedními dvě hrany z různých claimů, podíváme se, zda se neprotínají.



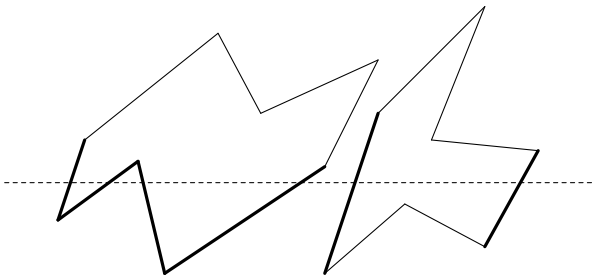
Celkem navštívíme $\mathcal{O}(n)$ vrcholů. Jelikož průřez vždy obsahuje nejvýše 4 hrany, zvládneme každý vrchol obsloužit v konstantním čase. Nesmíme ovšem zapomenout na počáteční seřazení vrcholů shora dolů, takže celý algoritmus trvá $\mathcal{O}(n \log n)$.

Implementaci nám může zjednodušit, že v průřezu musí obě hrany jednoho claimu vždy sousedit. V opačném případě jsme už dříve museli objevit průsečík.

Nekonvexní případ

Zametání nekonvexních claimů je trochu složitější. Průřez může obsahovat libovolný sudý počet hran jednoho claimu. Podíváme se, co se může s průřezem stát, narazíme-li na vrchol claimu, kde se potkávají nějaké dvě hrany:

- Obě hrany pokračují dolů – tehdy jsme na „špičce“ claimu, takže přidáme obě hrany do průřezu.
- Obě hrany pokračují nahoru – obě hrany skončily a z průřezu je smažeme.
- Jedna hrana vede nahoru, druhá dolů – horní hranu odstraníme, dolní přidáme.



Pokud přidáme hranu, musíme najít její sousedy a ověřit, jestli se s nimi neprotíná. Podobně pokud hranu smažeme, stala se předchozí hrana sousedem následující, takže ověřujeme jejich průsečík.

Potřebujeme najít datovou strukturu, ve které si budeme průřez pamatovat, abychom všechny tyto operace uměli provádět rychle. Potřebujeme umět vložit novou hranu na správné místo, smazat hranu a zjistit předchůdce a následníka hrany. To zajistí vyhledávací strom v čase $\mathcal{O}(\log n)$ na operaci. Jenže ve vrcholech vyhledávacího stromu si nemůžeme pamatovat x-ové souřadnice průsečíků hran s koštětem – ty se každým posunutím koštěte změní. Pořadí hran se nicméně nemění (zatím se žádné dvě hrany neproťaly), takže pro každé dvě hrany je jednoznačně určeno, která z nich je nalevo a která napravo. Do vrcholů stromu tedy uložíme přímo hrany.

Algoritmus celkem zpracuje n vrcholů, v každém z nich provede konstantně mnoho přidání/smazání hran v průřezu.

To pokaždé vyvolá $\mathcal{O}(1)$ operací se stromem, z nichž každá stojí $\mathcal{O}(\log n)$. Celkem tedy algoritmus pracuje v čase $\mathcal{O}(n \log n)$.

Obecná poloha

Zbývá dořešit, co si počít, pokud dva body mají stejnou y-ovou souřadnici. (Rozmyslete si, kde jsme tento předpoklad vlastně potřebovali.)

Pomůže obvyklý trik: představíme si, že celou rovinu otočíme okolo libovolného bodu o velmi malý úhel po směru hodinových ručiček. Tím se v každé dvojici se stejnou y-ovou souřadnicí posune levý bod nad pravý. A pokud byl předtím nějaký bod nad jiným, tento vztah bude při dostatečně malém otočení zachován.

Takové otočení je ovšem ekvivalentní s tím, že body nesetřídíme jenom podle y-ové souřadnice, nýbrž lexikograficky primárně podle y, sekundárně podle x. Zbytek algoritmu není potřeba měnit.

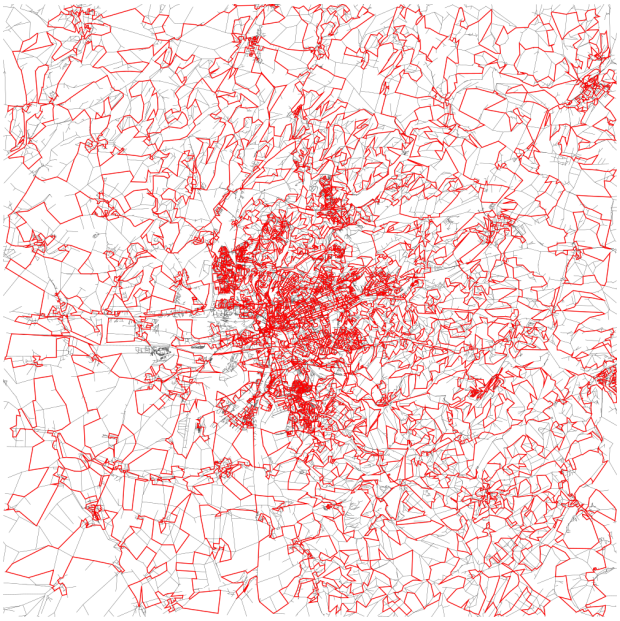
Úlohu připravil: Martin „Medvěd“ Mareš

34-2-4 Brněnská procházka

Zadaná úloha byla hledáním nejdelší vážené indukované cesty na zadaném grafu. Nalezení nejdelší indukované cesty je NP-těžký problém pro obecné neohodnocené grafy. Náš graf ale není neohodnocený, a už vůbec není obecný – jde o skoro rovinný graf, ostatně vzniknul z cest v Brně a okolí.

Bohužel vám ale efektivní algoritmus pro nalezení optimální cesty nedáme, protože ho neznáme, pokud vůbec existuje. Místo toho vám představíme vybraná řešení organizátorů, kteří úlohu řešili spolu s vámi.

Řešení Jirky Sejkory



Mé řešení pracuje ve dvou fázích. V první fázi najde nějakou počáteční cestu, které je alespoň trochu dobrá, ale ne moc dobrá. V druhé fázi pak tuto cestu vylepšuje.

První cestu nalézám opakovaným náhodným výběrem počátečního bodu a poté náhodným prodlužováním cesty, dokud to jde. Jakmile dojdeme do slepé uličky, tak skončíme a máme naši cestu. Tento postup velmi rychle končí ve slepých uličkách, takže toto několikrát opakuji (tisíce pokusů) a pamatuji si nejdelší cestu, kterou jsem našel. Pro drobné vylepšení vybírám následující vrchol vážený počtem vrcholů, které jsou z něj ještě dostupné. Tohle by šlo výrazně zlepšit (všimněte si, že nikdy nebacktrackuji), ale mým cílem je zde najít nějakou cestu, ne nutně příliš dobrou.

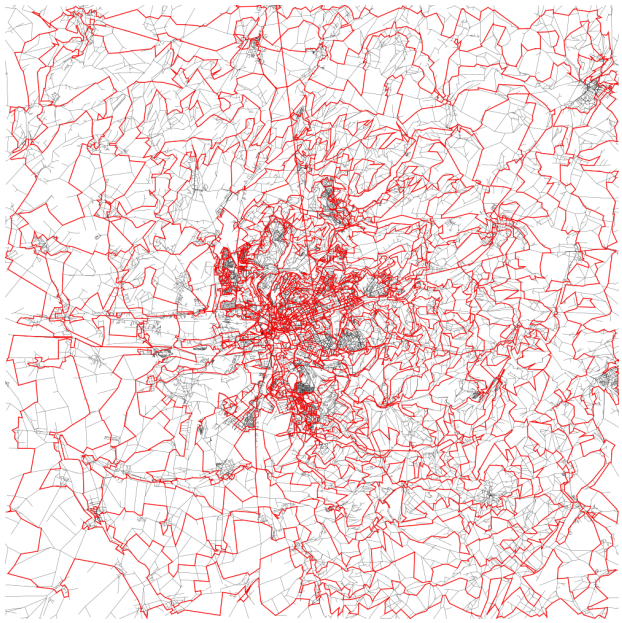
Vylepšování cesty je ta mnohem zajímavější část. Nalezenou cestu vylepšíme tak, že někde odstraníme souvislý úsek cesty délky N, a najdeme pomocí DFS co nejdelší validní smyčku, která zase cestu spojí. Tím jsme problém převedli zase na hledání nejdelší cesty v nějakém obřím grafu (jde o strom všech validních cest, exponenciálně větší než zadaný graf). Přidáme proto omezení na hloubku prohledávání, a díky tomu najdeme nejdelší cestu, která používá maximálně K hran. Při prohledávání jsem používal K v rozsahu od 16 (první vylepšení) do 26 (když už byl graf skoro plný).

Délky vyříznutých úseků N jsem používal 2, 4, 6, 10, 20, 50. Ty nejdelší úseky byly jen zřídka k něčemu užitečné, jelikož je muselo nahradit málo dlouhých hran, a to je v grafu vzácné. Vždy jsem v náhodném pořadí vyzkoušel všechny vyříznuté úseky (těch je lineárně s aktuální délkou cesty).

Nevýhodou tohoto přístupu je, že není možné posunout počáteční a koncový bod cesty. Jsme proto do jisté míry vázáni na počáteční řešení.

Tento algoritmus dobíhá do jednotek minut. Je ovšem možné vícekrát opakovat různé délky a také hledat vylepšení, které používá více hran, což běh algoritmu výrazně prodlužuje. Nalezená řešení byla dlouhá kolem 3300 kilometrů, i při opakovaném spouštění se délka výrazně nelišila.

Řešení Kuby Pelce



Mé řešení je založené na myšlence vést cestu spirálovitě od krajů mapy do středu. Nejdelší cestu budu hledat prohledáváním do hloubky (dále DFS). Začnu v nejzápadnějším vrcholu mapy a v každém vrcholu se pokusím jít tou nejlevější možnou cestou vzhledem ke směru, ze kterého jsem do vrcholu přišel. K určení směrů využívám GPS souřadnice ve vstupu. Jelikož se jedná o DFS, pokud se někdy dostanu do vrcholu, ze kterého v cestě nemůžu validně pokračovat, prostě se vrátím o vrchol zpět a půjdu druhou nejlevější cestou, a tak dále.

Pokud bych použil neupravené DFS, tak by tento algoritmus vlastně zkusil najít všechny možné cesty začínající v nejzápadnějším vrcholu, a to by trvalo extrémně dlouho. Proto používám několik heuristik.

Ještě před spuštěním algoritmu v grafu najdu všechny mosty (takové hrany, jejichž odstraněním se graf rozpadne na dvě jinak nepropojené části – komponenty), odstraním je z grafu, najdu největší vzniklou komponentu a algorimus spustím jen na ní. Při hledání nejdelší cesty je poměrně zbytečné chodit přes mosty, protože cesta se poté už nemůže vrátit zpět. V reálném městě se slepé cesty příliš nevyskytují, a pokud ano, tak jsou tak krátké, že jejich vynecháním si moc neuškodíme.

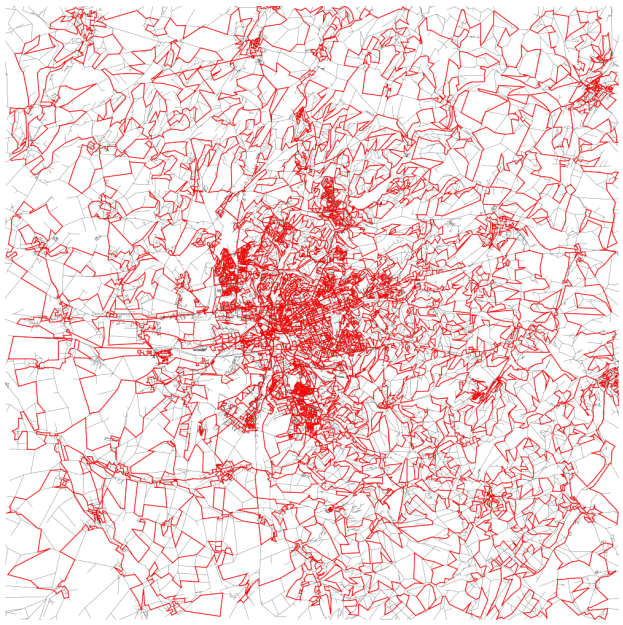
I když během běhu algoritmu vypisuji každou dosavadní nejdelší cestu, DFS se často dostává do míst grafu, kde zbytečně zkouší mnoho různých zakončení cesty, která nikdy nepovedou k lepšímu výsledku, protože algoritmus v nějakém vrcholu „špatně zabočil“ do slepého místa grafu, a trvá dlouho, než by se DFS opět propracovalo zásobníkem k onomu vrcholu. A to i přes odstranění mostů. Celkově se dá říct, že DFS trvá moc dlouho vracet se v zásobníku zpět. Tak mu trochu pomůžeme.

Pokud algoritmus nějaký počet iterací, třeba milion, nenajde novou nejlepší cestu, usoudíme, že je v nějaké „slepé uličce“ a z vrcholu zásobníku odstraníme několik položek. Pokud po dalším milionu iterací stále nenajde nic lepšího, odstraníme větší část zásobníku, a tak dále. Zahazováním kusů zásobníku samozřejmě odignorujeme mnoho různých cest, potenciálně těch nejlepších, nicméně v praxi tato heurestika funguje dobře. Algoritmus s ní dokáže najít cesty o délce stovek kilometrů v řádu minut, a poté, když dlouho nenachází nic lepšího, skončí na tom, že si vyprázdní celý zásobník.

I přes občasné mazání zásobníku se může stát následující scénář: algoritmus se dostane do „slepé uličky“, dlouho ji prohledává, po nějakém čase odstraní kus zásobníku a vrátí se před místo, kde do ní zabočil, jenže poté se může opět vrátit do stejné slepé uličky, jen tam dorazí trochu jinou cestou. Poslední heurestika opravuje tento problém. Pro každý vrchol si pamatuju, kolikrát jsem ho navštívil. Pokud tento počet překročí nějakou konstantnu, třeba tisíc, tak už do takového vrcholu algoritmus nikdy nevstoupí.

S těmito heurestikami algoritmus našel spirálu o délce zhruba dva a půl tisíce kilometrů.

Řešení Tomáše Slámy



Princip mého řešení je stejný jako řešení Jirky Sejkory – nejprve vygeneruji náhodnou cestu, kterou postupně vylepšuji vysekáváním souvislých úseků vrcholů.

Vysekávání úseku o délce d probíhá sekvenčně: nejprve zkusím odstranit vrcholy $(1, \ldots, d{+}1)$, poté $(2, \ldots, d{+}2)$, apod. Pro každý tento pokus ze startovního vrcholu hledám nejdelší cestu do koncového vrcholu pomocí DFS (s omezením na počet navštívení vrchlů, aby algoritmus nehledal do nekonečna). Zde používám zajímavou heuristiku: v daném vrcholu třídím sousedy podle toho, v jaké vzdálenosti jsou od koncového vrcholu (v euklidové vzdálenosti – používám zde `lat` a `lon`).

Pro $d = \{2 \ldots 50, 100, 120, 150\}$ tento algoritmus našel nejdelší cestu dlouhou 3 543 375 metrů.

Úlohu připravili: Jirka Kalvoda, Kuba Pelc, Jirka Sejkora, Tom Sláma

34-2-X1 Převod čísel

Pro řešení naší úlohy využijeme metody rozděl a panuj. Předpokládejme, že počet cifer zadaného čísla je mocnina dvojky. Tedy $N = 2^M$ pro celočíselné M. Když tomu tak není, můžeme před zadaný vstup doplnit dostatečné množství nul a tím délku doplnit na nejbližší větší mocninu dvojky. Délka vstupu se tím prodlouží nejvýše na dvojnásobek.

V případě, že na vstupu je jen jedna číslice, je problém triviální. Stačí postupně číslo modulit a dělit B.

V opačném případě si vstupní číslo rozdělíme na poloviny. Jelikož délka původního vstupu byla mocninou dvojky, tak lze vstup rozdělit na dvě stejné části, a dokonce každá z nich bude mít také délku rovnou mocnině dvojky. Na každou z polovin samostatně zavoláme rekurzivně náš algoritmus. Tím získáme dvě čísla v soustavě o základu B, která zbývá spojit dohromady.

Původní číslo X jsme rozdělili na dvě části Y a Z, přičemž platilo $X = Y \cdot A^{N/2} + Z$. Převodem čísla do jiné soustavy se ovšem tato rovnost nezměnila. Číslo $A^{N/2}$ původně mělo hezký zápis (jednička a za ní samé nuly), ale to už v soustavě o základu B neplatí. Je tedy nutné nejprve spočítat $A^{N/2}$ a pak pomocí aritmetiky v soustavě o základu B spočítat výsledek jako $Y \cdot A^{N/2} + Z$.

Potřebné mocniny A si můžeme přepočítat dopředu pro celý algoritmus. Začneme s A a postupně budeme předchozí výsledek umocňovat na druhou (tedy násobit sám se sebou). Stačí nám totiž získat pouze mocniny A, jejichž exponent je mocnina dvojky.

Odhad složitosti

Délka vstupu je $N = 2^M$ a číslo A lze zapsat v soustavě o základu B do L buněk. Nechť používaný algoritmus na násobení má složitost $\Theta(X^k)$ pro násobení čísel délky X.

Na předvýpočet mocnin A budeme potřebovat $M - 1$ násobení. Ovšem budeme násobit čísla délek L, 2^1L, 2^2L, …, $2^{M-2}L$. Součet délek bude menší než $2^{M-1}L < NL$. Časová složitost předvýpočtu bude $\mathcal{O}((NL)^k)$.

Průběh algoritmu si můžeme představit jako vyvážený binární strom. Kořen reprezentuje převod celého čísla; jeho potomci pak jsou dvě části, na které se dané číslo rozdělilo. Takto to postupuje až do listů, které reprezentují triviální převod jednociferného čísla. Každý list nám trvá $\mathcal{O}(L)$ času, celá poslední úroveň stromu tedy $\mathcal{O}(NL)$.

Na i-té úrovni stromu (v nulté úrovni je kořen) je 2^i vrcholů, každý z nich vyžaduje násobení čísel délky $L\frac{N}{2^i}$. Celková složitost i-té úrovně tedy je:

$$\mathcal{O}\left(2^i \left(\frac{NL}{2^i}\right)^k\right) = \mathcal{O}\left(2^{i\cdot(1-k)}(NL)^k\right).$$

Pokud $k = 1$, tak součet přes všech $M = \log N$ úrovní, a tedy i celková složitost algoritmu je $\mathcal{O}(NL \cdot \log N)$.

Pokud však $k > 1$, tak se jedná o součet geometrické řady, a tedy celková složitost se sečte na $\mathcal{O}((NL)^k)$.

Paměťová složitost algoritmu je $\mathcal{O}(NL)$.

Drobné zrychlení na závěr

Pokud je B menší než A, tak můžeme udělat následující trik: Místo toho, abychom převáděli do soustavy o základu B, tak budeme převádět do soustavy o základu B^r, kde r je nejmenší možné celé číslo takové, aby bylo $B^r \geq A$. Při vypisování můžeme snadno výsledek převádět do soustavy o základu B, protože jedna cifra v soustavě o základu B^r bude odpovídat r cifrám v soustavě o základu B.

Ovšem čísla do B^r se stále vejdou do jedné buňky, protože $A^2 > B^r$. Jelikož interně v algoritmu bude $L = 1$, celková časová složitost tedy bude $\mathcal{O}(N^k + LN)$ a paměťová složitost bude jen $\mathcal{O}(N)$.

Úlohu připravili: Jirka Kalvoda, Martin „Medvěd“ Mareš

Výsledková listina druhé série třicátého čtvrtého ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*2-1*	*2-2*	*2-3*	*2-4*	*2-S*	*2-X1*	*série*	*KSP-X*	*celkem*
0.					9	10	12	14	15	10	60,0	26,0	120,0
1.	Benjamin Swart	MensaG	3	2	9	8	12	1	16	8	46,0	24,5	102,0
2.	Daniel Skýpala	GTomkovaOL	4	22	9	10	12	1	16		48,0	8,0	99,5
3.	Robert Jaworski	GÚstavníPH	4	9	9	10	8	1	17	2	45,0	6,0	94,0
4.	Lukáš Tomoszek	GTři	4	3	9	8	12	1	14,9		44,9	4,0	89,9
5.	Jakub Ondroušek	GTomkovaOL	2	7	9	8		1	15		33,0	0,0	82,0
6.	Jáchym Kouba	GJŠkodyPŘ	2	2	6	8	10	1	15		40,0	4,0	80,0
7.	Jan Kotovský	GPísnickáPH	3	8	9	8	9	1	15		42,0	0,0	79,0
8.	Adam Kolník	SSŠVTPraha	3	7	9	8		1	15,5		33,5	8,0	71,0
9.	Prokop Randáček	GFXŠaldyLI	3	8	6	8	8		13,5		35,5	0,0	70,0
10.	Lukáš Veškrna	GKepleraPH	4	7	9	8		1	15		33,0	0,0	69,5
11.	Ján Plachý	G VBN Prie	4	2	9	10	7				26,0	4,0	58,5
12.	Vít Skalický	GPísnickáPH	4	21	4				15		19,0	0,0	56,0
13.	Petr Šícho	GKepleraPH	4	4	9	9			15		33,0	0,0	55,0
14.	Jan Slíva	MensaG	1	2	9	3	12				24,0	0,5	51,5
15.–16.	Viktor Číhal	SPŠSmíchov	2	2	9	8	9	1			27,0	0,0	49,0
	Vladimír Sklenár	GTerVans	2	2	6			1	14		21,0	4,0	49,0
17.	Jakub Mikeš	GJŠkodyPŘ	4	2					15,5		15,5	0,0	46,5
18.	Richard Tichý	SG Kladno	0	2	9	4	8				21,0	0,5	46,0
19.	Jiří Kvapil	GTomkovaOL	4	20							0,0	0,0	44,5
20.	Jan Černohorský	G Brandýs	4	3	9		5		15		29,0	0,0	44,0
21.	Eliška Macáková	CENADA BA	2	4							0,0	0,0	43,5
22.–23.	Patrik Herman	GTomkovaOL	3	8	9	0		1			10,0	0,0	42,5
	David Kolář	GJírovcČB	3	2	9		8	1	12,5		30,5	0,0	42,5
24.	Kryštof Maxera	GJírovcČB	1	4	9	8		1			18,0	0,0	42,0
25.–27.	Matúš Duchyňa	GGrössBA	3	2	4			0			4,0	0,0	26,0
	Nikolay Fomichev	SSŠVTPraha	3	2	2				9		11,0	0,0	26,0
	Alex Michaud	GJírovcČB	3	2	9	3	2	1			15,0	0,5	26,0
28.–29.	Vojtěch Lančarič	???	3	1	9	8	7				24,0	0,0	24,0
	Daniel Šoltýs	GTřeKošice	4	5	2						2,0	0,0	24,0
30.	Martin Belluš	GGrössBA	3	1							0,0	0,0	22,0
31.	Jonáš Dej	G Wicht	3	1							0,0	0,0	18,0
32.	Jakub Švojgr	GČeskáČB	3	1	6	8					14,0	0,0	14,0
33.–36.	Adam Kuča	PORG Krč	4	2		2				1	2,0	1,0	12,0
	Michal Pavlíček	MendelGOP	4	2							0,0	0,0	12,0
	Matúš Púll	GZborovPH	2	1							0,0	0,0	12,0
	Ondřej Skácel	GTomkovaOL	3	6							0,0	0,0	12,0
37.	Vojtěch Franc	GArabskáPH	2	1							0,0	0,0	11,0
38.	Pavel Jordán	GPOA Znojmo	3	3							0,0	0,0	10,0
39.–40.	Veronika Jůzková	MensaG	4	5	0						0,0	0,0	9,0
	Ivan Trenčanský	GLSáru	3	1	9						9,0	0,0	9,0
41.–42.	Martin Fof	MendelGOP	4	1							0,0	0,0	8,0
	Jakub Kopčil	GMikulášPL	3	1							0,0	0,0	8,0
43.	Marek Maškarinec	SPŠEMasLI	1	1	4						4,0	0,0	4,0
44.	Jan Šuráň	GZborovPH	4	1				1			1,0	0,0	1,0

Bonusové úlohy označené „X“ mají svou vlastní výsledkovou listinu a nepočítají se do normálního bodování ročníku.

Výsledková listina KSP-X po druhé sérii třicátého čtvrtého ročníku

	řešitel	*škola*	*ročník*	*sérií*	*1-X1*	*1-X2*	*2-X1*	*celkem*
0.					8	8	10	26,0
1.	Benjamin Swart	MensaG	3	2	8,5	8	8	24,5
2.–3.	Adam Kolník	SSŠVTPraha	3	7		8		8,0
	Daniel Skýpala	GTomkovaOL	4	22		8		8,0
4.	Robert Jaworski	GÚstavníPH	4	9		4	2	6,0
5.–8.	Jáchym Kouba	GJŠkodyPŘ	2	2		4		4,0
	Ján Plachý	G VBN Prie	4	2		4		4,0
	Vladimír Sklenár	GTerVans	2	2		4		4,0
	Lukáš Tomoszek	GTři	4	3		4		4,0
9.	Adam Kuča	PORG Krč	4	2			1	1,0
10.–12.	Alex Michaud	GJírovcČB	3	2	0,5			0,5
	Jan Slíva	MensaG	1	2	0,5	0		0,5
	Richard Tichý	SG Kladno	0	2	0,5			0,5

Bonusové úlohy z jednotlivých sérií se nepočítají do bodování ročníku. Mají svou vlastní výsledkovou listinu a za jejich úspěšné vyřešení (alespoň polovina bodů za úlohu) udělujeme speciální odměny.