Korespondenční Seminář z Programování

28. ročník **KSP** **Prosinec 2015**

Milí řešitelé a řešitelky!

Dostává se vám do rukou vánoční zadání KSPčka jako šité na dlouhé zimní večery. Takže ať si s sebou KSPčko vezmete na cestu vlakem za babičkou, na zasněžený svah či k posezení v příjemném křesle u svítícího stromečku a mísy plné cukroví, přejeme vám do nastávajícího roku jen to nejlepší a hodně zdaru nejen při řešení KSP!

Připomínáme, že každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme propisku, blok, tužku, a možná i něco navíc.

Termín série: **Pondělí 8. února 2015 v 8:00 SEČ** (CodEx má termín stejný)
Odevzdávání: Přes web na adrese *https://ksp.mff.cuni.cz/submit/*.
Odměna série: Každému, kdo získá z alespoň **pěti úloh polovinu bodů**, pošleme **sladkou odměnu.**

Třetí série dvacátého osmého ročníku KSP

Gaius Fabius nebyl z tažení, které odvedlo legii od jeho rodné Florencie daleko na sever do barbarské Galie, vůbec nadšený. Jednak se nerad vzdaloval od své ženy a syna, ale taky se úplně nehrnul do předních řad. Pocit stát v čele a cítit, jak se o široký štít odrážejí meče barbarů, rád přenechal jiným. Gaius se raději nacházel v pozadí a jako řemeslník se staral o spoustu věcí od obléhacích strojů po stavbu opevnění.

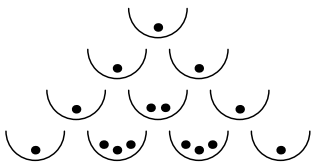
Tento den legie dopochodovala na planinu v lese, legát vyslal průzkumníky a zbytek se dal do stavby opevnění. Teprve před chvílí vztyčili poslední část dřevěného opevnění a Gaius unaveně padl na zem vedle ohně.

Část legionářů tu zrovna hrála hru se svými helmicemi. Stavěli z nich pyramidy a štouchali do nich kopím.

28-3-1 Pyramida z helmic — 7 bodů

Římští legionáři hrají o to, kdo bude mít v noci hlídku. Vždy hrají dva proti sobě, poberou si spoustu helmic a poskládají z nich pyramidu vysokou h (spodní patro má h helmic, patro nad ním $h-1$, ...).

Do každé helmice navíc vloží nějaký počet kamínků – do vrchní helmice přijde jeden a do každé další pak tolik kamínků, jako v helmicích vlevo a vpravo nad ní dohromady (pokud takové helmice jsou). Počet kamínků v helmicích tedy odpovídá Pascalově trojúhelníku[1] a následujícímu obrázku:



Legionáři se střídají po tazích. Vždy jeden z nich strčí kopím do helmice, kterou si vybere, a tím shodí ji i všechny helmice stojící na ní (levou vrchní, pravou vrchní a všechny, co podobně stojí na nich). Ze shozených helmic dostane všechny kamínky.

Hraje se tak dlouho, dokud stojí alespoň jedna helmice. Vyhrává ten, který má na konci méně kamínků. Existuje vyhrávající strategie pro některého z hráčů? A jak vypadá?

Hlídku „vyhráli“ Brutus a Marcus a ostatní vojáci centurie šli spát. Gaius měl jako řemeslník výhodu, že mohl spát ve svém stanu, který představoval vlastně i malou dílnu.

Zabalený pod přikrývky v tomto mrazivém počasí skoro usnul, když ho probudila podivná rána, takové lupnutí. Převalil se na bok a v tom ho uviděl! Divná postava, po níž ještě přebíhali nějací modří hadi. A v jeho stanu! Bohové!

Postava, asi muž, se zprudka nadechla a potřásla hlavou. Pravou rukou něco udělala na své levé ruce, nějak prapodivně hranaté, a pak se jí z levé ruky vyřinulo jasné světlo. Teď už bylo jasně vidět, že je to muž a že nemá hranatou ruku, jen na ní má zbroj, která svítí.

Muž se rozhlédl, spatřil Gaia a přiložil si prst na ústa. Jako by Gaia napadlo, že by mohl vydat nějaký zvuk. Teprve teď si všiml, že ve druhé ruce má muž masivní pánev.

Položil ji na pracovní stůl, popadl pár nástrojů a urazil jí držadlo. Pak chvíli něco kutil, sbalil si věci a chystal se asi k odchodu. Ještě se ale jednou ohlédl po zkoprnělém Gaiovi, něco zamumlal nějakou nesrozumitelnou řečí, popadl ze stojanu štít a kopí, a pak s lupnutím a modrým zábleskem zase zmizel.

Gaia konečně začaly poslouchat nohy a v děsu vyběhl z postele a nezastavil se, než doběhl do stanu Rufuse, jeho známého písaře. Chtěl si totiž nechat zapsat to, co slyšel, dokud si to pamatuje.

28-3-2 Líný písař — 9 bodů

Písař se pokouší zapsat vzkaz, který mu Gaius Fabius diktuje. Ten si však není příliš jistý tím, co slyšel. U každé věty si třeba pamatuje, že zněla trochu jako jedna věta, trochu jako jiná.

Písař by chtěl zapsat raději všechno, ale zase se při psaní nechce moc nadřít. Gaius nadiktuje písaři obě možné věty a písař chce najít co nejkratší větu (řetězec), kterou je možno vyškrtáním nějakých písmen převést na obě Gaiovy věty.

Příklad: Třeba pro věty (řetězce) `mujstit` a `mojesin` je řetězec `muojestitn` jedním z možných nejkratších řetězců obsahujících obě věty. Věty se z něj dají získat třeba takto:

```
muojestitn
mu_j_stit_
m_ojes_i_n
```

Ráno to ale bylo ještě horší...

[1] `https://cs.wikipedia.org/wiki/Pascalův_trojúhelník`

„Kde je moje zbroj? Tak kde?!?“

Gaius si pomyslel, že centurion vypadá dnes obzvláště naštvaně. Bohužel měl centurion ve zvyku posílat lidi, co ho naštvali, na nějaké speciální úkoly. Nemělo smysl pokoušet se mu vysvětlovat, že tu v noci byl nějaký divný cizinec, kterého neviděly hlídky, který mluvil divnou řečí, kterému svítila ruka a který shodou náhod ukradl centurionovu zbroj, kterou měl Gaius vyleštit. To prostě nemělo cenu.

Tak se Gaius smířil s tím, že bude muset někde v hlubinách zásobovací sítě legie sehnat zbroj novou, jinak že se prý nedožije dalšího rána. Bohužel sehnat novou zbroj pro centuriona nebylo tak jednoduché, vrchní zbrojíř se totiž vyžíval v neskutečné byrokracii.

28-3-3 Formulář na zbroj — 10 bodů

K sehnání zbroje je potřeba vyplnit spoustu formulářů. Každý formulář se dá vyplnit dvěma způsoby, a to kladně a záporně, a má navíc své číslo. Gaius má zmapováno, kdo a jak v táboře legie vydává formuláře.

Platí, že každý formulář vydává nejvýše jeden člověk. Někteří lidé v táboře své formuláře přímo vydají bez potřeby nečeho dalšího, ale ostatním je nutné nejdříve ukázat jiné již vyplněné formuláře, a teprve na jejich základě vydají svůj formulář (buď kladně, nebo záporně vyplněný).

To, jak vyplněný formulář vydají, je totiž dáno logickou funkcí (AND, OR, XOR nebo NOT) na formulářích, které dostanou k nahlédnutí.

Lidi v táboře máme očíslované a dostáváme popis byrokratické struktury v táboře zadaný jako:

- Člověk A vydává formulář F_a vyplněný kladně/záporně.
- Člověk B vydává formulář F_b s hodnotou danou logickou funkcí (třeba F_x XOR $F_y \to F_b$)

Zajímalo by nás, které všechny formuláře a jak ohodnocené umíme získat (předpokládejte, že formulář je vydán jen a pouze tehdy, pokud ukážeme všechny formuláře, na nichž závisí – tedy nestačí třeba pro formulář vydávaný podmínkou F_a AND F_b donést pouze negativní F_a s tím, že již určuje výsledek).

Nakonec se Gaiovi povedlo sehnat novou zbroj až odpoledne. Doufal, že si pak konečně odpočine, ale o tom si mohl nechat leda tak zdát. Legie se chystala na střet s barbarskou armádou, a tak si legát svolal všechny starší legionáře starající se o obléhací stroje.

Jeho plánem bylo vyvážit početní převahu barbarů lepší disciplínou legionářů, lepší výzbrojí a také použitím katapultů. Vybrané centurie měly v rozsáhlém údolí zaujmout pevné pozice a katapulty ostřelovat blížící se barbary.

No a rozmístění katapultů byl právě úkol pro legionáře starající se o obléhací stroje.

28-3-4 Katapulty — 11 bodů

Legát chce nechat po údolí rozděleném do čtvercové sítě $N \times N$ rozmístit K katapultů. Rozkázal, že každý katapult smí střílet jen rovnoběžně s čtvercovou sítí (tedy podle políček horizontálně nebo vertikálně, ne však našikmo) a chce, aby se katapulty vzájemně neohrožovaly (nebyly žádné dva ve stejném sloupci nebo řádku).

Údolí je ale trochu podmáčené, a tak máme pro každý katapult určený obdélník, ve kterém může stát.

Pro zadané N, K a pro určené obdélníky pro každý katapult najděte rozmístění katapultů tak, aby se vzájemně neohrožovaly, nebo rozhodněte, že takové rozmístění neexistuje.

Lehčí varianta (za 5 bodů): Řešte úlohu v jednorozměrné variantě: Pro každý katapult máme daný úsek, kde může stát, a nesmí stát dva na stejném políčku.

Další ráno část legie vyrazila. V táboře zůstalo dvacet centurií, zbylých čtyřicet odvedl legát do boje. Průzkumníci nelhali, skutečně se jim povedlo zaujmout výhodné postavení v širokém údolí a proti rozptýleným barbarům fungovala legátova rozptýlená taktika překvapivě dobře.

Osamocení barbaři se tříštili o pevně stojící hradbu štítů, větší skupinky padaly za oběť přesně mířeným zásahům košů s kamením z katapultů.

Gaius ale řešil problém se svým katapultem. Po každém výstřelu se v nestabilní půdě pohnul, sklouznul rohem do tůňky vedle a bylo potřeba ho zase vytáhnout a správně nasměrovat. To velmi snižovalo rychlost palby.

Pomocníkům se povedlo sehnat spoustu dřevěných fošen a Gaius je chtěl použít k zatížení katapultu, aby se nehýbal. Uprostřed několika desítek legionářů, kteří stáli kolem dokola v neproniknutelné hradbě, začal fošny osekávat a svazovat k sobě.

28-3-5 Závaží z fošen — 8 bodů

Máme několik silných dřevěných fošen. Všechny jsou stejně tlusté, ale liší se svými rozměry. Chtěli bychom z nich sestavit co možná nejtěžší závaží, neboli závaží s největším objemem, protože všechny fošny jsou ze stejně těžkého dřeva. Prkno o rozměrech 1×1 váží 1 jednotku.

Aby se nám ale závaží nerozpadalo, musí mít všechny vrstvy na sobě stejný rozměr. Jednotlivé fošny můžeme oříznout (rovnoběžně s jejich hranami a s tím, že odřezky zahazujeme), můžeme je otočit (prohodit šířku a výšku), nebo dokonce nepoužít vůbec. Nemůžeme však mít v závaží nějakou fošnu menší (ať už šířkou nebo výškou) než jinou.

Formát vstupu: Na prvním řádku vstupu obdržíte počet fošen, na dalších N řádcích pak rozměry každé fošny jako dvě čísla oddělená mezerou.

Formát výstupu: Na výstup vypište jediné číslo – maximální váhu závaží, kterou jsme schopni ze zadaných fošen poskládat.

Ukázkový vstup:	*Ukázkový výstup:*
6 5 11 1 1 5 6 1 2 6 4 4 6	96

Druhou a čtvrtou fošnu nepoužijeme vůbec, ostatní ořízneme na rozměry 6×4, což nám dohromady dá objem (a tedy i váhu) 96.

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

Nadšení z toho, že se povedlo katapult stabilizovat, však netrvalo dlouho. Vlastně trvalo docela krátce, protože barbarů najednou bylo příliš mnoho a legionáři začínali být vysílení. Ze svého vyvýšeného postavení Gaius viděl, jak hradby štítů několika centurií zakolísaly, barbaři se dostali skrz a jednotlivé legionáře svým počtem udolali. Takhle nemůžeme vydržet moc dlouho, pomyslel si Gaius.

Navíc začala padat tma a situace se stávala ještě nepřehlednější a prohra stále zřejmější. Asi si to uvědomil i legát a vzduchem se ke všem zbývajícím vojákům doneslo troubení signálu k ústupu.

Katapulty byly opuštěny a jednotlivé centurie se začaly v želvích formacích (a želvím tempem) sunout k jižní části údolí, odkud přišli.

Než se zbývající legionáři shromáždili do jedné skupiny, zbyla z legie sotva polovina. Nikdo nevěděl, kolik jejich druhů je mrtvých, kolik padlo do zajetí (pokud barbaři vůbec brali zajatce) a kolik se jich ztratilo.

Aby se zjistilo, kolik jich z každé centurie zbylo, poslal každý centurion mezi svými muži helmici. Každý, kdo přežil, do ní měl vhodit kamínek. A aby se jednotlivé centurie nepomíchaly, každý měl předat helmici jenom někomu, koho zná.

28-3-6 Počítání přeživších — 12 bodů

Přeživší legionáři se potřebují spočítat. Dělají to tak, že si mezi sebou posílají helmici a vkládají do ní kamínky. Helmice by měla obejít všechny legionáře a to tak, že u každého se objeví právě jednou.

Legionáři jsou ochotní helmici předat někomu, koho znají osobně, někomu, koho zná někdo koho znají, nebo někomu, koho zná někdo koho zná někdo koho znají. Zkráceně řečeno jsou ochotni předávat helmici jen někomu, kdo je od nich maximálně tři přátelství daleko (A předá helmici D, pokud se znají $A-B$, $B-C$ a $C-D$). Známosti jsou symetrické, tj. pokud A zná B, tak i B zná A.

Pro skupinu legionářů a neorientovaný graf toho, jak se znají, najděte takovou posloupnost předávání helmice, aby respektovala podmínku výše, každý legionář dostal helmici do rukou právě jednou a helmice se dostala zpátky do rukou centurionovi.

Lehčí varianta (za 6 bodů): Vyřešte úlohu, pokud víte, že graf známostí mezi legionáři je strom.

Když zjistili, kolik jich je, rozhodlo se, že se legie stáhne nazpět do opevněného tábora. Protože byla noc, utvořili velkou formaci ve tvaru kruhu ježící se kopími na všechny strany a v ní opatrně postupovali zpátky.

Útoky barbarů ustaly, ale o to byla noc zlověstnější. Kruhová formace se přiblížila k řídkému lesu v soutěsce a zastavila se. Legát tu cítil nějakou léčku, stromy byly vysoké a košaté, tak košaté, že se na každém z nich mohla skrývat spousta barbarů.

Podrobnou mapu lesa dodali průzkumníci už minulý den a legát by teď potřeboval vědět, jestli může legii lesem bezpečně provést, nebo musí les někudy obejít.

28-3-7 Legie v lese — 13 bodů

Máme legionáře v kruhové formaci s poloměrem r. Dále máme také podrobnou mapu lesa. Les na obou stranách svírá soutěska a stromy jsou vzhledem k velikosti legie tak malé, že jejich kmeny můžeme považovat pouze za body.

Legie chce projít od severu na jih lesa a to tak, aby se cestou vyhnula všem stromům (protože na nich mohou číhat barbaři). Do kruhu představujícího legii se tedy během postupu lesem nesmí dostat žádný strom a ani nesmí kruh zasahovat za některou z bočních hranic (protože les je v soutěsce).

Pro zadaný les rozhodněte, jestli taková cesta skrz les existuje, nebo ne.

Tato úloha je praktická a řeší se ve vyhodnocovacím systému CodEx.[2] Přesný formát vstupu a výstupu, povolené jazyky a další technické informace jsou uvedeny v CodExu přímo u úlohy.

Stalo se, to co legát očekával – legii v lese přepadli barbaři. Díky legátově prozíravosti sice neseskákali ze stromů přímo do středu legionářů, ale stejně se strhla bitva za světla měsíce a několika málo pochodní.

Kruh z legionářů se během boje přeléval a deformoval, ale držel. Vždy, když byli legionáři donuceni o pár metrů ustoupit, přišli další spolubojovníci a barbary zase vytlačili dál. Bohužel Gaius se při jednom z těhle výpadů ocitl mimo ochranný kruh štítů. Jediné štěstí bylo, že si ho nikdo z barbarů nevšiml, a tak se mu povedlo se rychle odkulit do nějaké jeskyně.

Teď ale sledoval, jak se od něj hradba štítů postupně vzdaluje a mezi ním a jeho druhy pobíhá mnoho barbarů. Bez meče nebo alespoň štítu se tam nemá šanci dostat! Ještě, že si ho zatím v té jeskyni nevšimli...

Jeho přemýšlení přerušil divný zvuk za jeho zády. Rychle se otočil a málem vykřikl. Opět se tu objevil ten tajemný cizinec, kterému po těle běhali postupně mizející modří hadi, v ruce nesl pochodeň. Také ho do nosu udeřil odporný zápach spáleného masa a všiml si, že na zem před cizincem dopadla zuhelnatělá lidská paže.

Cizinec vypadal sám docela zaskočený, ale rychle se vzpamatoval a s nějakým zamumláním odkopl spálenou ruku dále od sebe. Pak se rozhlédl, vytáhl zpoza pasu nějakou svítící krabičku a začal s ní obcházet stěny. O Gaia se nezajímal.

Po chvíli asi objevil to, co hledal, udeřil do stěny jeskyně kladivem, odloupl nějaký divný kus kamene, sáhl po svojí levé ruce a se zablesknutím zmizel.

Než se však Gaius stihl vzpamatovat a pořádně nadechnout, zablesklo se podruhé a cizinec se vrátil. Teď tam však místo pochodně stál s pánví v jedné ruce a centurionskou zbrojí ve druhé. Řekl něco dalšího nesrozumitelného a hodil zbroj i s mečem Gaiovi k nohám. Pak ho rukou pobídl, aby se do ní navlékl.

Gaius dlouze neváhal a cizince poslechl. Jakmile na sebe zbroj navěsil, podíval se na cizince, co teď. Ten se nahnul, hodil něco nalevo do lesa, na prstech odpočítal od tří do jedné a silně strčil Gaia do zad. Ten vyběhl současně s tím, co se zleva z lesa začaly ozývat divné zvuky.

Díky téhle diverzi se dostal zhruba do poloviny vzdálenosti k postupující legii, než si ho všimli barbaři. A pak přišel ke slovu meč, štít a zbroj. Z posledních sil se probojoval zpátky ke svým druhům a tak se stalo, že Gaia zachránil neznámý cizinec.

Příběh pro vás vyprávěl

Jirka Setnička

28-3-8 Inteligence hejna — 15 bodů

V tomto díle se budeme naposled věnovat evolučním algoritmům a oproti minulému dílu budeme opět více čerpat inspiraci z chování přírody. Konkrétně si budeme všímat struktur chování skupin živočichů. Tyto algoritmy patří do kategorie, která se souhrně nazývá *Inteligence hejna*.

[2] `http://ksp.mff.cuni.cz/viz/codex`

Co ale znamená samotný pojem inteligence hejna? V přírodě existuje řada živočichů, jejichž jedinci se chovají velmi jednoduše a řídí se jen následováním pár jednoduchých pravidel. Takoví jedinci obvykle nemají potenciál samotní přežít, a nebo dokázat velké věci. Když ale vezmeme celé hejno takových jedinců, kteří všichni usilují o stejnou věc, tak tím získáme něco velkého.

Například si představme mravence, který se snaží postavit své obydlí. Ten musí jít pro dřívko, odložit jej na hromadu, pak jít pro další, opět jej položit a tak dál. To vypadá jako jednoduchý úkol. Akorát jeden mravenec tyto dřívka nedokáže shánět dost rychle. Než donese druhé, tak mu první dřívko může sebrat jiný živočich, odfouknout vítr, a nebo se nám mraveneček může polámat. V každém případě tento jeden mravenec má jen pramalé šance na úspěšné dostavení celého obydlí.

Nyní si znova představme stejnou situaci, ale namísto jednoho mravence bude obydlí stavět milion mravenců, kteří všichni chtějí postavit společné obydlí. Tato situace je mnohem nadějnější. Obydlí sice musí postavit řádově větší, ale také jim práce půjde řádově rychleji a z nasbíraných dřívek se nám rychle stane hromádka. A když se náhodou jednomu mravenci něco stane, tak tam pořád máme statisíce dalších, kteří jej mohou nahradit.

Co z toho plyne pro informatiku? Máme jedince, kteří se chovají jednoduše. Ty zvládneme snadno simulovat pomocí sady jednoduchých pravidel. Když pak vezmeme hodně takových jednoduchých jedinců a budeme je simulovat všechny najednou v jednom prostředí (tak, aby se navzájem ovlivňovali), tak nám dohromady vytvoří složitou strukturu chování, kterou už nejspíš nedokážeme jednoduše popsat.

Z informatického pohledu zbývá jen jedno. Navrhnout takové chování jedinců, jejich cíle a takové prostředí, aby nám celé takové hejno vyřešilo zadaný problém. My pro chování jedinců budeme hledat inspiraci ve skutečných příkladech. Nepovede se nám dosáhnout toho, že najdeme tak skvělého živočicha, jehož simulací dokážeme vyřešit všechny problémy, ale uvidíme, že inspirace konkrétním živočichem nás dovede ke ke konkrétní třídě problémů, na které se simulace zrovna tohoto živočicha hodí.

Chování mravenčí kolonie

Mravenci jsou sociální hmyz, který žije ve skupinách velkých 2 až 25 milionů jedinců. My si budeme všímat, jak se chovají při shánění potravy. Mravenci začnou víceméně náhodně prohledávat okolí mraveniště a hledat potravu. Jakmile je některý z nich úspěšný, tak po své cestě vypouští feromony, jimiž dává ostatním mravencům najevo, že tato cesta je dobrá. Ostatní mravenci jsou pak schopni tyto feromony cítit a automaticky preferují cesty s vyšší koncentrací feromonů. Tomuto se říká mechanizmus pozitivní odezvy.

Je důležité si uvědomit, že tam nikde není žádný centrální mravenec, který by pohyb řídil, a že celé shánění potravy vyplyne na povrch automaticky z náhodného chození a za pomoci feromonů. Cesty k potravě postupně sílí a jakmile tam potrava dojde, tak mravenci přestanou produkovat feromony, ty začnou postupně vyprchávat a mravenci si najdou jinou cestu k další potravě.

Celá kolonie se tedy umí samoorganizovat bez jakékoliv centrální či vnější pomoci za využití produkování feromonů. Ty v prostředí pak fungují jako sdílená krátko/dlouhodobá paměť všech mravenců z mraveniště.

Optimalizace mravenčích kolonií

Hledání potravy mravenců budeme simulovat jako hledání cesty v grafu. Máme zadaný ohodnocený graf $G = (V, E)$, ve kterém bychom chtěli najít co nejkratší cestu z vrcholu s (mraveniště) do vrcholu t (potrava). My si popíšeme základní mravenčí algoritmus (Ant System), ze kterého pak vychází drtivá většina ostatních mravenčích algoritmů.

Každá hrana ij má svou délku d_{ij} a intenzitu feromonů f_{ij}. Mravenec začne ve vrcholu s a vydá se po grafu až dokud nedojde do t (případně nepřekročí maximální počet kroků). Pokud stojí ve vrcholu i, tak v dalším kroce přejde do vrcholu j s pravděpodobností

$$p_{ij} = \frac{f_{ij}^{\alpha} \cdot b_{ij}^{\beta}}{\sum_{ik \in E} f_{ik}^{\alpha} \cdot b_{ik}^{\beta}}$$

kde $b_{ij} = 1/d_{ij}$ je „vhodnost hrany“ – čím kratší, tím vhodnější, a α, β jsou parametry ovlivňující význam obou složek. Obvykle se α, β volí kolem 2.

Intenzita feromonů se na začátku výpočtu může zvolit třeba 1 či menší konstanta, nebo b_{ij} či úplné jinak. Záleží, jak nám to pro konkrétní algoritmus vyhovuje. Volbou $f_{ij} = b_{ij}$ obvykle nic nezkazíme.

Jedna iterace algoritmu má tři fáze:

1. Vytváření řešení (mravenci hledají cestu)
2. Aktualizace feromonů (vypařování a zvyšování mravenci)
3. Vnější zásahy (nepovinná část)

Vytváření řešení jsme si již popsali. To jen mravenci na základě aktuálních pravděpodobností hledají cestu v grafu. Pak se odpaří intenzita feromonů na všech hranách podle

$$f_{ij} = (1 - \rho) \cdot f_{ij}$$

kde $\rho \in [0, 1]$ je intenzita odpařování. Čím vyšší ρ, tím více se feromony odpařují. Po odpaření pak všichni mravenci znova projdou své cesty a intenzitu feromonů každé hrany ij na nich zvýší o $1/L$, kde L je délka nalezené cesty. Případně o rozumný násobek této hodnoty či dle jiné klesající funkce závisející na délce hledané cesty.

$$f_{ij} = f_{ij} + 1/L$$

Vnější zásahy jsou nepovinná část algoritmu. Jedná se o vylepšení, která se nedají udělat z pohledu mravence. Obvykle se jedná o různé zvýhodňování nejlepších mravenců, podrobnější hledání v okolí nejlepšího řešení a podobně.

Tím jsme popsali celý základní mravenčí algoritmus a nyní se podíváme na jeho aplikaci na reálný problém.

Aplikace v problému obchodního cestujícího

Zadání problému: Máme zadaný seznam n měst a vzdálenost každé dvojice z nich (tedy úplný graf o n vrcholech). Obchodní cestující by všechny tyto města chtěl navštívit (každé právě jednou) a vrátit se do toho, kde začal. V jakém pořadí je má projít?

Toto je slavný problém, pro který není znám žádný algoritmus, který by jej efektivně řešil. Tak nám nezbývá než se jej snažit vyřešit optimalizačně. Jelikož v problému hledáme nějakou cestu v grafu, tak se nabízí použít mravence.

Nalezení potravy v tomto případě bude znamenat projití všech vrcholů grafu, každým právě jednou. Čím kratší cestu

najdeme, tím více feromonů budeme vydávat. Jelikož se pohybujeme na úplném grafu, tak hledání takových cest nebude velký problém.

Jeden mravenec vždy začne v náhodném vrcholu a na základě feromonů přejde do dalšího vrcholu. Vždy ale bere v úvahu jen ty sousedy, které ještě při své cestě nenavštívil. Takže mravenec pokaždé nějakou cestu nalezne a ta bude procházet právě přes právě n vrcholů. Čímž jsme v ještě lepší situaci než při hledání nejkratší cesty, kde nám mravenci mohli i zabloudit.

Zbytek mravenčího algoritmu zůstává naprosto stejný.

Úkol 1 [15b]: Řešte problém obchodního cestujícího na obcích České Republiky. Data si můžete stáhnout na obvyklém místě na stránce seriálu.

Ve vstupním souboru najdete na každém řádku (je jich 6 251) popis jednoho města formou následujících údajů. Počet obyvatel obce, x-ová souřadnice obce, y-ová souřadnice obce a název obce. Jednotlivé údaje jsou oddělené mezerou. Za poskytnutí dat, která by měla být platná k začátku roku 2011, děkujeme ČSÚ – `https://www.czso.cz/`.

Pro jednoduchost vzdálenost mezi dvěma městy uvažujeme jako vzdálenost bodů v rovině. Úlohu řešte pro všechna města. Takový graf, ale pro první zkoušení algoritmu možná bude příliš velký, tak úlohu můžete zkusit řešit pro obce s více jak 2 500 obyvateli, případně pro obce s více jak 10 000 obyvateli.

Pro řešení můžete použít jak algoritmus mravenců, tak jakýkoliv jiný postup. Porovnejte výsledky, kterých jste jednotlivými postupy dosáhli.

Možné modifikace mravenčí kolonie

Elitářská strategie: Preferování doposud nejlepší cesty za celý dosavadní průběh algoritmu. Funguje tak, že si pamatujeme doposud nejkratší cestu a v každé iteraci během aktualizace intenzity feromonů přičteme ϵ/L_b ke všem hranám této cesty, kde ϵ určuje sílu vlivu nejlepší cesty a L_b je její délka.

Vliv pořadí: Lepší mravenci budou produkovat ještě více feromonů než ti horší. Mravence seřadíme podle délky jejich cesty a feromony produkuje jen w nejlepších z nich. r-tý mravenec produkuje feromony podle

$$f_{ij} = f_{ij} + (w - r + 1)/L$$

Další inteligence hejna

Existuje ještě řada dalších modifikací algoritmu mravenčí kolonie. Ty ale v tomto seriálu nezvládneme pokrýt. Raději si uděláme rychlý přehled dalších inspirací hejn, které v přírodě můžeme najít.

Optimalizace hejnem částic

Anglicky *Particle Swarm Optimization* je trochu podobné diferenciální evoluci z minulého dílu. Hejno sestává z jednotlivých částic, které se pohybují v prohledávacím prostoru. Každá má svou aktuální polohu a vektor rychlosti. Rychlost se pak stáčí k doposud nejlepšímu nalezenému bodu dané částice a nejlepší nalezené poloze všech částic.

Optimalizace včelím rojem

Zahrnuje celou skupinu algoritmů, které hledají inspiraci v rojích včel. Jedinci v algoritmu jsou obvykle rozděleny do několika skupin včel, kde každá hraje svou specifickou roli. Algoritmy nachází uplatnění jak při řešení reálných optimalizací, tak při řešení diskrétních problémů.

Například můžeme mít tři typy včel: dělnice, pozorovatelky a průzkumnice. Nejdříve dělnice vyletí do prostoru řešení a pomocí včelího tanečku dávají vědět pozorovatelkám, jak je jejich řešení dobré. Pozorovatelky z nich pomocí vážené pravděpodobnosti vyberou ty, které jsou úspěšné. Dělnice, které ve svém okolí dlouho nebyli úspěšné se pak změní na průzkumnice a vydají se zkoumat jinou oblast prostoru řešení.

Optimalizace hejnem světlušek

Každá světluška má svou svítivost v závislosti na její úspěšnosti. Čím úspěšnější, tím více září a přitahuje ke své pozici ostatní světlušky. Každá světluška má také omezenou vzdálenost, kam až dohlédne.

Karel Tesař

Recepty z programátorské kuchařky: Geometrie

Geometrické algoritmy

V dnešním díle našeho kuchařkového speciálu se budeme učit vařit geometrické problémy. A co že si představujeme pod pojmem geometrický problém? Trochu analytické geometrie, například zjištění, na které straně orientované přímky bod leží, trocha plotů, neboli konvexních obalů, a obecně mnoho zametání.

V celé kuchařce se omezíme pouze na dvourozměrné problémy, tedy na algoritmy v rovině. Některé postupy se dají zobecnit pro trojrozměrné, a většinou i pro n-rozměrné problémy, ale to je již nad rámec této kuchařky.

Geometrické základy

Nejdříve trocha středoškolské analytické geometrie pro ty, kdo ji ještě neměli. Ostatní mohou tuto sekci přeskočit.

Každý bod v rovině můžeme určit jeho souřadnicemi vůči osám. Nejběžněji se používá takzvaný *kartézský souřadný systém,* tedy dvě na sebe kolmé osy označované jako x-ová osa (vodorovná) a y-ová osa (svislá). Obvykle se uvažuje, že hodnoty na osách rostou směrem doprava (osa x) a směrem nahoru (osa y), my se toho budeme v naší kuchařce držet.

Místo, kde se obě osy protínají, se označuje jako *počátek* soustavy souřadnic. Samotné *souřadnice* bodu zapisujeme jako dvojici čísel, která udávají, o kolik jednotek se musíme posunout ve směru které z os, abychom z počátku dorazili do bodu, kterému souřadnice patří. Počátek má souřadnice $[0, 0]$. Bod se souřadnicemi $[a, b]$ leží na pozici, kterou získáme tak, že se od počátku posuneme o a jednotek ve směru první osy (x-ové) a o b jednotek ve směru druhé osy (y-ové).

Vše ostatní funguje tak, jak jsme se učili při geometrii na základní škole, tedy úsečka je určena dvěma krajními body, obdélník čtyřmi a podobně. Ještě si ale řekneme, co je to vektor, a zavedeme některé další pojmy.

Často potřebujeme popsat vzájemnou polohu dvou bodů. Můžeme například udat jejich vzdálenost a směr (třeba jako úhel vzhledem k ose x). Praktičtější ale bývá říci, o kolik se liší jejich x-ové a y-ové souřadnice. To nám dá dvojici čísel, které říkáme *vektor.*

Pokud například k bodu $[1, 1]$ přičteme vektor $a = (2, -1)$, dostaneme se do bodu $[3, 0]$. Stejně tak, pokud odečteme například bod $[4, 2]$ od bodu $[1, 3]$, tak dostaneme vektor $b = (-3, 1)$ udávající jejich vzájemnou polohu.

Pomocí vektoru a bodu tedy lze určit přímku. Bod nám určí, kam umístit vektor, a vektor nám určí směr přímky z daného bodu. Tomuto vektoru se říká *směrový vektor,* nebo také někdy *směrnice,* dané přímky nebo úsečky.

Samotné vyjádření přímky nebo úsečky poté může být ve dvou tvarech. Prvním z nich je *parametrický tvar.* Základem je nějaký bod $A = [a_x, a_y]$. Od toho se ve směru směrového vektoru $u = (u_x, u_y)$ můžeme pohybovat libovolně a stále budeme na přímce. To nám vede na následující tvar, kde t je libovolný reálný parametr, neboli proměnná, za kterou si můžeme dosadit jakékoliv reálné číslo a vždy nám vyjde bod na přímce. Parametrický tvar vypadá:

$$x = a_x + tu_x$$

$$y = a_y + tu_y$$

To samé můžeme vyjádřit i vektorově, tedy $X = A + tu$.

Pro ilustrování funkce parametru, když bude $t = 0$, tak dostaneme výchozí bod přímky. Pokud poté budeme s parametrem hýbat od $-\infty$ do $+\infty$, dostaneme postupně všechny body na přímce.

Druhým způsobem zápisu je *obecný tvar přímky.* K jeho vyjádření budeme potřebovat kolmý vektor ke směrovému vektoru, tomu se také říká *normálový vektor.* V rovině ho získáme jednoduše. Pokud je $v = (v_x, v_y)$ směrnice přímky, tak vektor na něj kolmý má tvar $n = (v_y, -v_x)$. Jako poznámku pro zvídavé můžeme uvést, že *skalární součin* těchto vektorů, tedy součin po složkách ($v \cdot n = ab + b(-a)$), je roven 0, což je také jedna z definic kolmosti.

A jak tedy vypadá slibovaný obecný tvar přímky? Pokud je $n = (a, b)$ normálový vektor přímky, tak obecný tvar přímky je rovnice $ax + by + c = 0$. Dobře, a a b máme, jak ale zjistit c? Normálový vektor určuje směr, kterým přímka povede, ale stále ji můžeme libovolně posouvat. Potřebujeme ještě znát jeden bod, který na naší přímce leží, aby byla určená jednoznačně.

Když dosadíme souřadnice takového bodu do rovnice přímky s neznámou c, získáme tak rovnici pro c, kterou vyřešíme. A máme hotovo, známe hodnoty všech koeficientů v rovnici. Ještě si můžeme všimnout, že pro $c = 0$ prochází přímka počátkem.

Takovéto tvary se hodí jednak pro nějaké zapsání přímek, ale také pro *zjištění jejich průsečíku.* Když hledáme průsečík, hledáme vlastně místo, kde mají obě přímky navzájem stejné x-ové a y-ové souřadnice. A to vede na jednoduché soustavy lineárních rovnic, které jistě již vyřešit umíte.

Ještě si ale zdůrazněme rozdíl úseček oproti přímkám. V případě parametrického tvaru omezujeme velikost parametru t (například $t \in \langle 0, 1 \rangle$) a v případě obecného tvaru omezujeme rozsah jedné ze souřadnic (například $x \in \langle -2, 2 \rangle$). V případě, že bychom chtěli vyjádřit polopřímku, si parametr nebo souřadnici omezíme pouze z jedné strany.

Nakonec si ukážeme jednu základní aplikaci parametru a parametrického vyjádření úsečky. Jak snadno spočítat střed nějaké úsečky AB? V takovém případě není nic jednodušší, než si vzít vektor $B - A$, přenásobit ho parametrem $1/2$ (střed úsečky je v polovině její délky) a přičíst k bodu A. Triviální úpravou pak zjistíme, že střed úsečky můžeme spočítat jako aritmetický průměr jejích krajních bodů:

$$A + \frac{1}{2} \cdot (B - A) = \frac{A + B}{2}$$

Jako příklad na rozkoukání si ukážeme, jak zjistit, na které straně přímky leží bod.

Zjištění polohy bodu vůči přímce

Nejdříve si zavedeme pojem orientovaná přímka. Když budeme mít přímku určenou dvojicí bodů A a B, budeme se na ni dívat, jako kdybychom stáli v prvním bodě (bod A) a dívali se směrem ke druhému (bod B). Pak již máme jasně definovanou pravou a levou stranu a můžeme říci, kde vůči přímce bod leží.

Vezměme si tedy přímku určenou body A a B a bod X. Určíme si vektory $u = X - A$ a $v = B - A$ (s prvky u_x, u_y, respektive v_x, v_y) a porovnáme úhel mezi nimi.

Pokud jste už měli analytickou geometrii, určitě znáte vzoreček na výpočet úhlu mezi dvěma vektory. Vzoreček má tvar:

$$\cos\alpha = \frac{u \cdot v}{|u||v|}$$

Jeho nevýhodou je, že výpočet inverzní funkce $\cos^{-1}$ trvá dlouho. Je proto lepší použít jiný způsob výpočtu, kde si vystačíme pouze s násobením.

Tím jiným způsobem je výpočet determinantu matice určené těmito vektory. *Matice* je pouze tabulka, kde jsou vektory poskládány pod sebe (ta naše tedy bude velká 2 na 2 políčka).

Determinant matice této velikosti nám udává obsah rovnoběžníku určeného zadanými vektory. A navíc znaménko determinantu nám říká, jestli je úhel mezi vektory (měřený v kladném směru, tedy proti směru hodinových ručiček) menší než π, nebo větší než π.

Kdo se ještě s determinanty nesetkal, může brát následující vzorec pro výpočet determinantu matice dva krát dva jako kouzelnou formuli. Kdo přesto chce zdůvodnění, může si zkusit udělat rozbor všech vzájemných poloh dvou přímek (a jejich směrových vektorů), které mohou nastat. Po chvíli dojdete ke vztahu přesně odpovídajícímu následujícímu vzorečku:

$$d = u_x v_y - u_y v_x.$$

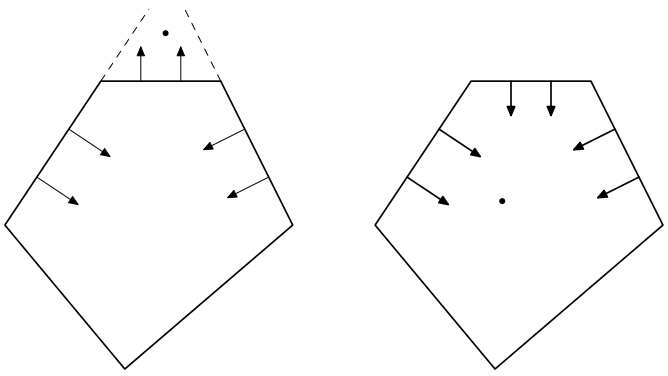
Pokud vyjde d kladné, je bod napravo od přímky, pokud vyjde d záporné, je bod nalevo od přímky, a konečně, pokud vyjde $d = 0$, tak bod leží na přímce.

Bod a konvexní mnohoúhelník

Konvexní mnohoúhelník je takový, který nemá žádný vnitřní úhel větší než 180°. Jinou definicí je, že pokud si zvolíme libovolné dva body v mnohoúhelníku a natáhneme úsečku mezi nimi, nikdy nám nevyleze z mnohoúhelníku ven.

Když už víme, co konvexní mnohoúhelník je, jak zjistíme, jestli nějaký bod leží v něm nebo ne? Využijeme vlastnosti konvexnosti. Stačí nám jít po hranách na obvodu a zjišťovat, jestli hledaný bod leží na stejné straně všech hran (tedy přímek určených koncovými body hran), nebo neleží.

Pokud bod leží na stejné straně všech hran, nachází se uvnitř mnohoúhelníku. Pokud se ale vůči jen jediné hraně nachází na jiné straně než vůči ostatním, leží bod vně mnohoúhelníku. Nejlépe to vysvětlí obrázek:



Tomuto postupu se také někdy říká test polorovinami. Každá kontrola nám zabere konstantně mnoho času. Časová složitost tohoto postupu je tedy lineární vzhledem k počtu hran, neboli $\mathcal{O}(N)$.

Bod a nekonvexní mnohoúhelník

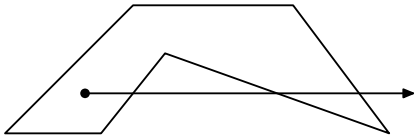
Pro nekonvexní útvary je již postup o něco těžší, protože jak si můžeme všimnout, postup s kontrolováním polohy bodu vůči všem hranám fungovat nebude.

Můžeme si ale na chvíli zahrát na Robina Hooda a ze zkoumaného bodu vystřelit šíp, respektive vést polopřímku. Pěkně se nám bude počítat, pokud polopřímku povedeme rovnoběžně s nějakou z os (třeba ve směru $(1, 0)$). Celé řešení pak spočívá v počítání, kolikrát polopřímka protne hranici mnohoúhelníku.

Můžeme si totiž všimnout, že finálně polopřímka skončí venku a nikdy více již do mnohoúhelníku nevstoupí. A pokaždé když do mnohoúhelníku vstoupí, musí z něj zase někdy vystoupit. Pokud tedy bod leží venku, začali jsme polopřímku vést zvenku, a tedy bude počet protnutí sudý, pokud bod leží uvnitř, tak bude počet protnutí hranice lichý.

Jediné, na co je potřeba dát pozor, je situace, kdy polopřímka povede přesně skrz nějaký vrchol. V takovém případě se musíme podívat na opačné krajní body hran, které se v tomto vrcholu stýkají. Pokud se obě nachází ve stejné polorovině určené polopřímkou, jen jsme se vrcholu dotkli, ale neprošli jsme skrz (a tedy nepočítáme žádný průsečík). Pokud se ale krajní body hran nachází v opačných polorovinách, znamená to, že jsme ve vrcholu hranici proťali a musíme započítat jeden průsečík.

Jako cvičení na rozmyšlenou necháme situaci, kdy se druhý krajní bod jedné z hran nachází na polopřímce.

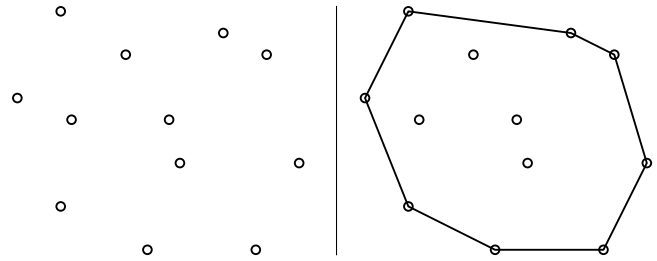


Opět musíme zkontrolovat polopřímku vůči všem hranám, takže časová složitost je znovu $\mathcal{O}(N)$ (i když s o něco vyšší konstantou, protože spočítání průsečíku je více početních operací, než jeden test polorovinou).

Konvexní obal a zametání roviny

Podíváme se na jeden z nejznámějších geometrických problémů, totiž hledání konvexního obalu množiny bodů v rovině. *Konvexní obal* je nejmenší konvexní mnohoúhelník, který obsahuje všechny zadané body. Můžeme si všimnout, že všechny vrcholy výsledného mnohoúhelníka musí být nějaké body ze zadané množiny, jinak bychom mohli mnohoúhelník ještě zmenšit (a nebyl by to konvexní obal).

Jako motivaci si představte třeba situaci, že máte sad ovocných stromů a chcete je oplotit co nejkratším plotem. Jak takový plot, nebo obecně obal, nalézt?



Vlevo neobalené body, vpravo obalené.

Ukážeme si postup, kterému se říká *zametání roviny.* Je to trik, který najde uplatnění u mnoha různých geometrických problémů a vyplatí se ho umět.

Základní myšlenka spočívá v tom, že nějakou přímkou, říkejme jí *zametací přímka,* přejedeme přes celou rovinu (od minus nekonečna do plus nekonečna, zleva doprava nebo shora dolů) a vždy když zametací přímka protne nějaký pro nás zajímavý bod, zpracujeme příslušnou událost. *Událost* je něco významného, co souvisí s příslušným bodem (průsečík přímek, vrchol mnohoúhelníka apod.)

Ale jak jet přímkou postupně od minus nekonečna do plus nekonečna? To není vůbec nutné. Pohyb přímky můžeme začít v nějakém startovním bodě (většinou první událost v setříděné posloupnosti událostí) a ukončit ho po zpracování všech událostí. Navíc nebudeme přímkou pohybovat plynule, ale budeme jí vždy skákat z události na událost (protože mezi událostmi se nic zajímavého neděje).

Vraťme se k našemu problému s konvexním obalem. Jako události budeme brát všechny body, které dostaneme na vstupu. V tomto případě nám žádné nové události v průběhu výpočtu vznikat nebudou, takže frontu událostí můžeme implementovat jako lineární spojový seznam.

Na začátku si body setřídíme podle jejich x-ové souřadnice (zatím budeme pro jednoduchost předpokládat, že žádné dva body nemají stejnou x-ovou souřadnici), začneme je zametací přímkou postupně procházet zleva doprava a budeme si udržovat konvexní obal bodů, které jsme už zpracovali.

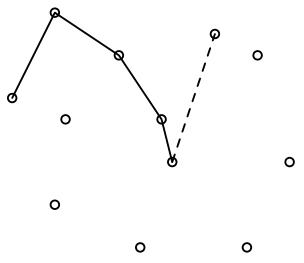
V průběhu výpočtu si budeme konstruovat horní a dolní *obálku.* Obě obálky budou určitě začínat v nejlevějším a končit v nejpravějším bodě (jednoduchým pozorováním lze nahlédnout, že tyto body do obalu určitě patří). A jak už název napovídá, horní obálka půjde vrchem a bude se zatáčet stále doprava, a dolní obálka naopak půjde spodem a bude se stále zatáčet doleva.

Můžeme se pro zjednodušení dohodnout, že nejlevější i nejpravější bod patří do obou obálek. Když pak horní a dolní obálku spojíme, dostaneme konvexní obal.

Horní (respektive dolní) obálku si budeme udržovat jako lineární seznam vrcholů.

Teď si ukážeme, jak bude probíhat jeden krok zpracování. Výpočet se bude provádět samostatně pro horní a dolní obálku, my si ho ukážeme jen pro horní (pro dolní je až na zrcadlení stejný).

Uvažujme, že už máme nějakou část horní obálky, skočili jsme zametací přímkou na další bod a ten teď chceme přidat. Podíváme se na poslední bod v horní obálce a zkontrolujeme úhel poslední hrany v obálce a úsečky mezi posledním bodem obalu a novým bodem.



K tomu můžeme využít například test polorovinami z úvodu kuchařky (pokud nový bod leží vůči poslední hraně horní obálky napravo, je vnitřní úhel konvexní, pokud nalevo, je úhel konkávní). Jestliže se horní obálka zatáčí doprava, máme vyhráno, přidáme nový bod do obálky a můžeme se posunout na další bod. Zajímavější je ale situace, kdy se nám obálka stočí doleva a vznikne konkávní úhel.

Pokud se podíváme na obrázek výše, jasně vidíme, že je potřeba dosavadní poslední bod obálky odebrat a zkusit spojit nově přidávaný bod s předposledním. Odstraníme tedy poslední bod obálky a budeme test opakovat s předposledním bodem.

Takto budeme pokračovat (a případně vyhazovat další body), než buď bude úhel hran konvexní, nebo dokud nám v obálce nezůstane pouze jeden bod (počáteční). Pak nový bod přidáme do obálky a pokračujeme s dalším.

Výše popsaný postup je nejvýhodnější provádět najednou pro obě dvě obálky. Tedy každý bod se pokusím připojit k horní i dolní obálce a podle toho obě obálky příslušně upravím.

Proč tento postup funguje? Postupně projdeme všechny body a každý z nich se alespoň na chvíli stane posledním bodem obálky. Při změně obálky se obsažená plocha v konvexním obalu vždy pouze zvětší a žádný bod tedy nám tedy nemůže zůstat mimo konvexní obal.

Ještě jsme zapomněli na případ, kdy úhel není ani konvexní, ani konkávní. V takovém případě se rozhodneme, jestli budeme vrchol tohoto úhlu započítávat mezi vrcholy konvexního obalu. Obvykle se takový vrchol z konvexního obalu vyhazuje, ale nakonec vždycky záleží, k čemu ten konvexní obal vlastně potřebujeme.

Skončíme, až zametací přímkou skočíme na poslední bod a zpracujeme ho. V tomto bodě se nám obálky spojí a dostaneme celý konvexní obal. Teď ale přichází otázka, kolik času nám tento postup zabere?

Může se zdát, že hodně, protože při vyhazování bodů z obálky můžeme postupně vyhodit skoro všechny body. Označme si velikost zadané množiny (počet bodů na vstupu programu) N. Musíme si uvědomit, že každý bod do obálky přidáme pouze jednou a vyhodíme ho také maximálně jednou, tedy časová složitost je lineární k velikosti množiny, tedy $\mathcal{O}(N)$, v případě, že již máme setříděný vstup. Pokud ne, musíme ještě přičíst čas potřebný k setřídění bodů, tedy $\mathcal{O}(N \log N)$ při použití nějakého rychlého třídicího algoritmu.[3]

Nakonec ještě zbývá dořešit více bodů se stejnou x-ovou souřadnicí. Pokud to nejsou krajní body, tak nám to v postupu nevadí. Menším problémem je, když to jsou počáteční, nebo koncové body. Problém ale snadno vyřešíme tím, když body seřadíme lexikograficky, tedy nejdříve podle x a pokud je stejné, pak podle y. To nám jednoznačně určí pořadí bodů a počáteční i koncový bod.

Také si to můžeme představit tak, že rovinu nepatrně natočíme. Tím se určitě konvexní obal (až na natočení) nezmění, nikde nebudou dva body nad sebou a z pohledu algoritmu je to vlastně totéž, jako bychom prošli body v lexikografickém pořadí.

Hledání průsečíků úseček

Nakonec si ukážeme ještě jeden typický zametací problém, který principu zametání využívá o trochu více než konvexní obal. Představte si, že máte v rovině N úseček a chcete najít všechny jejich průsečíky.

Hledáme samozřejmě co nejrychlejší algoritmus vzhledem k N a počtu průsečíků P.

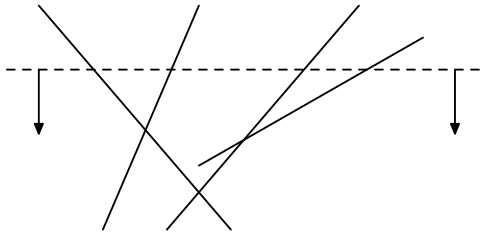
[3] `http://ksp.mff.cuni.cz/viz/kucharky/trideni`

Bystří si jistě již spočítali, že průsečíků může být v extrémním případě až N^2 a tedy nic rychlejšího než zkontrolovat každou úsečku se všemi dalšími v tomto případě není.

Ale takové případy se moc často nestávají, spíše naopak. Uvažujme tedy, že průsečíků je řádově tolik, kolik je úseček a v tom případě je výše popsaný algoritmus již pomalý.

Předpokládejme pro zjednodušení, že v žádném bodě se neprotínají tři a více úseček, žádné dvě úsečky nemají více než jeden společný bod (neleží přes sebe) a žádná úsečka není ani přesně svislá, ani přesně vodorovná. Vyřešení takovýchto případů spočívá v snadných úpravách uvedeného řešení.

Použijeme opět zametací přímku (pro lepší představu teď jdoucí shora dolů, obecně ale nemá směr zametání význam), kterou budeme skákat přes události, a na ní si budeme udržovat aktuální stav. Nazvěme ji třeba *průřezem.* Jak už název napovídá, bude udržovat pořadí úseček, které aktuálně protínají zametací přímku. Jelikož se průřez bude po každé události měnit, budeme pro něj potřebovat šikovnou datovou strukturu. Ale na to se podíváme až potom, co si rozebereme události, ať víme, co od průřezu budeme chtít.



Stejně jako v minulém případě budou mezi událostmi všechny body na vstupu (tedy počáteční i koncové body úseček), vyskytnou se tam ale i další. Pojďme si tedy trochu lépe rozebrat události a akce, které se při nich mají stát:

- *Začátek úsečky:* Přidáme úsečku na správné místo do průřezu, spočítáme případné průsečíky s okolními úsečkami a přidáme je do seznamu událostí.
- *Konec úsečky:* Smažeme úsečku z průřezu, a jelikož se nám dvě okolní úsečky dostanou smazáním této k sobě, musíme ještě spočítat jejich případný průsečík a přidat ho do seznamu událostí.
- *Průsečík:* Započítáme a zapíšeme si průsečík úseček, prohodíme pořadí těchto dvou úseček na průřezu, a jelikož se nám k sobě na průřezu dostaly nové úsečky, musíme spočítat, jestli se někde protínají, a případně průsečíky přidat do seznamu událostí.

Spočítání průsečíků úseček je jednoduchá analytická geometrie. Nejdříve porovnáme jejich směrnice. Pokud jdou od sebe, nemusíme se o nic starat, pokud jdou k sobě, spočítáme, ve kterém bodě se protnou. A když máme tento bod, jenom ověříme, jestli leží na obou úsečkách (neboli že úsečky nekončí ještě před spočítaným průsečíkem).

Když se podíváme na požadavky, hodilo by se nám umět v průřezu rychle vyhledávat, přidávat a mazat, k čemuž nám nejlépe poslouží vyhledávací strom. Ale co za informace si budeme o úsečkách ve vrcholech stromu pamatovat? Jejich aktuální x-ovou pozici (tedy přesněji x-ovou souřadnici bodu této úsečky na úrovni zametací přímky)? Tu bychom museli po každé události u všech úseček přepočítat, budeme na to tedy muset jít chytřeji.

Ve vrcholech stromu si budeme ukládat pouze nějaký rovnicový tvar úsečky (například její obecnou rovnici, nebo směrnici a bod) a vždy, když budeme vyhledávat ve stromu, tak si na základě aktuální y-ové pozice zametací přímky spočítáme v konstantním čase aktuální x-ovou pozici úsečky (jednoduchým doplněním do obecné rovnice) a podle toho se budeme ve vyhledávacím stromu pohybovat.

Máme tedy datovou strukturu pro průřez, ale jak dlouho budou trvat operace s ním? Jelikož v každou chvíli bude ve vyhledávacím stromu maximálně N vrcholů (tedy maximálně tolik, kolik je úseček), budou všechny operace se stromem trvat $\mathcal{O}(\log N)$.

Do seznamu událostí budeme potřebovat také přidávat prvky, takže tentokrát se nám mnohem více hodí použití nějaké haldy. Opět si můžeme uvědomit, že v haldě bude najednou pouze $\mathcal{O}(N)$ prvků (za každou úsečku její začátek a konec a průsečíky úseček vedle sebe na průřezu, tedy maximálně $N-1$ průsečíků) a tedy operace v ní bude trvat $\mathcal{O}(\log N)$.

Když už máme vybudované datové struktury, podívejme se na to, jak algoritmus poběží. Na začátku přidáme do průřezu první úsečku a do seznamu událostí všechny začátky i konce úseček. Pak již jen postupujeme po událostech, každou událost zpracujeme podle postupu výše a skončíme ve chvíli, kdy nám dojdou všechny události.

Algoritmus funguje správně, jelikož postupně projde přes všechny průsečíky (když jedna úsečka protíná více dalších, tak postupným prohazováním v průřezu se dostanou všechny tyto dvojice vedle sebe a všechny průsečíky přidáme do událostí) a žádný průsečík neprojdeme vícekrát.

Zpracování jakékoliv události nás stojí konstantní množství operací s datovými strukturami, a protože každá z těchto operací stojí maximálně $\mathcal{O}(\log N)$, tak nás zpracování jedné události stojí $\mathcal{O}(\log N)$. Počet událostí je $2N + P$ kde N je počet úseček a P počet průsečíků na výstupu, tedy celková časová složitost je $\mathcal{O}((N+P)\log N)$. Pro pořádek ještě uveďme paměťovou složitost, které je díky použitým datovým strukturám $\mathcal{O}(N)$.

Můžeme si všimnout, že pokud by průsečíků bylo řádově N^2, tak jsme si vlastně pohoršili. Předpokládali jsme ale situaci, kdy je průsečíků řádově stejně jako úseček. V tomto případě je náš algoritmus výrazně rychlejší.

Závěr

Prošli jsme si základní geometrické algoritmy pro rovinné problémy a ukázali jejich základní myšlenky. Různou aplikací a kombinací těchto postupů můžeme řešit většinu lehčích geometrických problémů v rovině, které potkáme.

Jen jako ochutnávku si ještě uvedeme například *Voroného diagramy,* což je rozklad roviny na oblasti, které jsou vždy nejblíž danému bodu (motivací může být například přiřazení obcí na mapě k nejbližšímu krajskému městu). Při jejich konstrukci se také uplatní zametání roviny, ale tentokrát již ne přímkou, ale pomocí zametacích parabol.

A jak jsme si uvedli na začátku, mnohé z uvedených postupů lze zobecnit z roviny i do prostoru, ale o tom někdy jindy. Pokud máte zájem o další informace o geometrických algoritmech, tak vás mohu odkázat na studijní text k přednášce ADS[4] na stránkách Martina Mareše.

Pokud stále nemáte geometrie dost, můžete si ještě zkusit vyhledat pojmy *kombinatorická a výpočetní geometrie.* Dostanete se tak ke spoustě dalších zajímavých materiálů.

Jirka Setnička

[4] `http://mj.ucw.cz/vyuka/ads/43-geom.pdf`

Vzorová řešení druhé série dvacátého osmého ročníku KSP

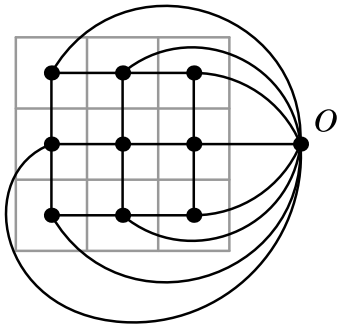
28-2-1 Potopa ve městě

Představme si, že městečko už je zatopené a zadržuje maximální možné množství vody. Zaměřme se nyní na jedno konkrétní políčko x a označme si h výšku hladiny na tomto políčku (měřeno od země). Pokud na toto políčko přilijeme nějakou vodu navíc (řekněme do výšky $h+1$), musí všechna odtéct až za okraj městečka (jinak by se zadržený objem zvýšil, a tedy ten původní by nemohl být maximální).

Aby mohla odtéct, musí odtéct *někudy*, po nějaké cestě z x na okraj. Uvažujme libovolnou takovou cestu P – to je prostě souvislá posloupnost políček, která začíná v x a končí na okraji města. Pokud na některém z políček P je budova výšky alespoň $h+1$, vodu zadrží a ta tudy odtéci nemůže. Tedy aby voda ve výšce $h+1$ mohla odtéci po cestě P, musí maximum z výšek budov na P být nejvýše h. Tomuto maximu budeme říkat *zádržnost* cesty P (protože udává maximální výšku vody, kterou daná cesta zadrží) a značit jej $z(P)$.

Aby mohla přidaná voda odtéct, musí tedy z x existovat alespoň jedna cesta na okraj se zádržností nejvýše h. Zároveň ovšem nesmí existovat cesta se zádržností menší než h, protože pak by se na x neudržela voda ve výšce h. Obě tyto podmínky lze shrnout tak, že minimum ze zádržností přes všechny cesty z x na okraj musí být rovno h.

To nám dává návod, jak h spočítat. Stačí najít z x na okraj cestu P s nejmenší zádržností (té budeme říkat *nejpropustnější cesta*), pak $h = z(P)$. To se dost podobá problému hledání nejkratší cesty – chceme najít cestu, která minimalizuje nějakou vlastnost, jen namísto délky je to zádržnost. Pokud budeme hledat cesty, mohlo by se nám hodit dívat se na město jako na graf (pro každé políčko jeden vrchol, sousední políčka spojena hranou). Navíc přidáme jeden virtuální vrchol o reprezentující oblast za okrajem městečka, který bude sousedit se všemi okrajovými políčky:



Nyní můžeme prostě hledat nejpropustnější cestu z x do o. Od hledání nejkratší cesty se náš problém liší dvěma věcmi:

- Ohodnocené jsou vrcholy namísto hran.
- Ohodnocení celé cesty je maximem z ohodnocení jednotlivých vrcholů, nikoli součtem.

Zkusíme podle toho přímočaře upravit Dijkstrův algoritmus:[5] prostě v něm sčítání přepíšeme na maximum a délku hrany na výšku budovy ve vrcholu. Výsledek by mohl v pseudokódu vypadat takto (s je startovní vrchol, $b(v)$ je výška budovy na políčku reprezentovaném v a $Z(v)$ je zádržnost nejpropustnější zatím nalezené cesty do v):

1. $Z(*) := \infty, Z(s) := b(s)$
2. Dokud nejsou všechny vrcholy definitivní:
3. Vyber vrchol v s nejmenším $Z(v)$
4. Prohlas v za definitivní
5. Pro každého souseda w vrcholu v:
6. Pokud $\max(Z(v), b(w)) < Z(w)$ (našli jsme propustnější cestu do w):
7. $Z(w) := \max(Z(v), b(w))$

Chtěli bychom ukázat, že takto upravený algoritmus stále funguje, tedy že na konci je $D(v)$ délka nejpropustnější cesty z s do v pro každé v. Začátek algoritmu (krok 1) a úpravy D (takzvané *relaxace*, kroky 5-7) jsou správně z definice zádržnosti. Co si musíme rozmyslet, je, zda oprávněně prohlašujeme vrcholy za definitivní. Ale zde je důkaz úplně stejný jako u klasického Dijkstry, takže si jej přečtěte v kuchařce a rozmyslete si, že funguje i pro naši variantu. S podobnou úpravou Dijkstrova algoritmu jste se mohli setkat v úloze 27-2-3 Průjezd jeřábu,[6] kde byly oproti naší úloze prohozené minimum a maximum.

Mohli bychom tedy z každého políčka spustit modifikovaného Dijkstru a tak určit výšku hladiny na tomto políčku jako zádržnost nejpropustnější cesty do o. Ale opakované spouštění je docela neefektivní. Namísto toho si všimneme, že zádržnost je symetrická, takže namísto hledání nejpropustnější cesty z každého vrcholu do o můžeme hledat nejpropustnější cestu z o do všech vrcholů. A na to nám stačí jednou spustit náš algoritmus s o jako startovním vrcholem.

Když takto pro každé políčko x spočítáme výšku vodní hladiny nad zemí $h(x)$, stačí od ní odečíst výšku budovy na daném políčku $b(x)$ a dostaneme výšku zadrženého vodního sloupce. Pokud by měla vyjít záporná (budova vyčnívající nad zadrženou hladinu), budeme ji považovat za nulovou. Jelikož políčka mají jednotkovou plochu, tato výška se rovná i objemu vodního sloupce na daném políčku. Tyto dílčí objemy pak stačí posčítat přes všechna políčka a získáme hledaný celkový objem zadržené vody.

Pokud má město rozměry $M \times N$, pak náš graf má $\mathcal{O}(MN)$ vrcholů i hran, a naše řešení tedy bude potřebovat čas $\mathcal{O}(MN \log(MN))$. Paměťová složitost bude lineární. Graf nemusíme explicitně sestrojovat: uvnitř Dijkstry můžeme vrcholy identifikovat dvojicí (i, j) a sousedy najdeme prostým přičítáním/odečítáním jedničky.

Program (C):
`http://ksp.mff.cuni.cz/viz/28-2-1.c`

Filip Štědronský

[5] `http://ksp.mff.cuni.cz/viz/kucharky/haldy-a-cesty`
[6] `http://ksp.mff.cuni.cz/viz/27-2-3`

28-2-2 Řazení knih

Úloha o posunutí knih pro vás nebyla složitá a počet došlých řešení to jen dokazuje. Pojďme si nyní některé postupy vedoucí k vyřešení této úlohy rozebrat.

V textu budeme používat některé pojmy (jako třeba společné dělitele nebo zbytky po dělení), které si můžete připomenout v kuchařce o teorii čísel.[7]

Když se na problém podíváme informaticky, tak úkolem bylo posunout všechny záznamy v poli doprava o K pozic s tím, že co přeteče napravo, to se objeví na začátku. V tom nám pomůže, pokud výpočty souřadnic budeme brát jako zbytky po dělení počtem prvků v poli. Posunutí o jedna doprava z posledního prvku tak povede zpátky na prvek s indexem nula: $(N-1)+1 \bmod N = 0$.

Pokud bychom posouvali záznamy jen o jednu pozici doprava, umíme to jednoduše: Budeme si držet proměnnou `minule` pro hodnotu minulého políčka. V cyklu půjdeme přes všechny pozice, vždy si ji zapamatujeme do dočasné proměnné a přepíšeme hodnotou z proměnné `minule`. Pak přesuneme hodnotu z dočasné proměnné do proměnné `minule` a pokračujeme další pozicí. Kdybychom to prováděli od konce, bude nám dokonce stačit jen jedna pomocná proměnná, ale směr od začátku pole se nám bude hodit víc:

```
minule = pole[N-1]
for i in range(N):
        docasna = pole[i]
        pole[i] = minule
        minule = docasna
```

Co pokud je chceme přesouvat o více pozic doprava? Nemůžeme si dovolit více než konstantně mnoho pomocných proměnných, takže si nemůžeme uložit celý posouvaný úsek délky K. Mohli bychom sice provést posunutí o jedna doprava K-krát, ale to by vedlo na čas $O(NK)$, což je moc.

Jako první si dovolíme předpoklad, že $K < N$ (kdyby ne, můžeme za K vzít zbytek po dělení N a nic se nezmění). Prvek na indexu 0 má finálně přijít na pozici K, prvek na indexu 1 se má ocitnout na $K+1$ a tak dále. A prvek na indexu K patří na pozici $2K$, tento prvek pak na pozici $3K$ a tak dále. Toho využijeme.

Nebudeme posouvat souvislé bloky, ale budeme po poli různě poskakovat a zařídíme, aby se prvky dostaly na své správné pozice. Začneme na nějakém indexu a jako při posouvání o jedno políčko výše využijeme pomocných proměnných, jen přeskoky budou dlouhé K a budeme při nich modulit (dovolíme si přeskočit z konce pole opět na jeho začátek, jako kdyby pole bylo cyklické). Navštívíme tak postupně pozice $0, K \bmod N, 2K \bmod N, \ldots$

Zastavíme se ve chvíli, kdy dojdeme opět na startovní index – na ten dojdeme nejdéle po N krocích, protože platí $N \cdot K \bmod N = 0$. Vlastně na něj dojdeme poprvé už po počtu kroků, který odpovídá nejmenšímu společnému násobku N a K. Pokud si jako L označíme počet kroků, kdy se poprvé vrátíme na startovní index (a platí tak $L \cdot K \bmod N = 0$), tak $L \cdot K$ bude přesně nejmenší společný násobek K a N.

Všechny prvky na právě projitém cyklu jsme umístili na správné pozice. Nemuseli jsme ale takto umístit všechny prvky, speciálně když N a K budou mít nějakého společného dělitele většího než 1.

Potřebovali bychom takto projít i všechny ostatní cykly (a každý z nich právě jednou). Ale tady narážíme na problém: Jak si pamatovat cykly, které jsme již prošli? Nemáme dost paměti na to si je značit. Můžeme si ale všimnout pěkné matematické vlastnosti.

Jakýkoliv jiný cyklus bude stejně dlouhý (po stejně mnoha přičtení K se vyrovná s délkou pole opět na své výchozí pozici), cykly tak budou od sebe jen o něco posunuté. Protože pro největší společný dělitel a nejmenší společný násobek platí vztah

$$\mathrm{nsd}(K,N) = \frac{K \cdot N}{\mathrm{nsn}(K,N)} = \frac{K \cdot N}{K \cdot L} = \frac{N}{L},$$

tak víme, že počet potřebných opakování cyklu k přesunu všech N pozic je vlastně největší společný dělitel délky pole a K.

Stačí nám tedy spustit cyklus postupně od všech pozic menších než největší společný dělitel. Všimneme si, že všechny pozice v jednom cyklu mají po dělení $\mathrm{nsd}(K,N)$ stejný zbytek, takže určitě žádné dva z vybraných počátečních bodů neleží na stejném cyklu a dohromady pokrývají právě všech $\frac{N}{L} \cdot L = N$ pozic.

Největší společný dělitel dokonce ani nemusíme počítat. Stačí nám jen držet si v jedné proměnné počet již přesunutých prvků a spouštět další cykly tak dlouho, dokud nedosáhne N. Paměťová složitost je konstantní a časová je $O(N)$.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/28-2-2.py`

Jirka Setnička

Poněkud magické řešení

Máte rádi kouzla? My také, a tak si jedno ukážeme. Rozmyslete si, proč funguje.

```
def zrcadli(A, i, j):
    for k in range((j-i) // 2):
        A[i+k], A[j-1-k] = A[j-1-k], A[i+k]

def posun(A, n, k):
    zrcadli(A, 0, n)
    zrcadli(A, 0, k)
    zrcadli(A, k, n)
```

Martin „Medvěd“ Mareš

28-2-3 Zprávy pro lupiče

Nejprve předvedeme velmi pomalé řešení, které určitě funguje, načež ukážeme, jak ho předpočítáváním různých hodnot zrychlit. To mimochodem bývá dobrá strategie pro skoro všechny CodExové úlohy, kde nevidíte vstup: výstup chytrých řešení můžete snadno srovnávat s výstupem toho prvního hloupého a sami objevit většinu chyb.

Nadále budeme článku textu říkat *seno,* značit ho S a jeho délku s. Hledanému slovu budeme říkat *jehla* J a její délku označíme j. Celkový počet výskytů jehly v seně označíme v.

Exponenciální řešení

Začneme jednoduchým rekurzivním řešením: Nejprve se pokusíme najít všechny výskyty prvního znaku jehly, pro každý z nich pak všechny napravo ležící výskyty druhého znaku jehly, a tak dále. Pokaždé, když najdeme j-té písmeno jehly, vypíšeme pozice všech nalezených písmen.

[7] `http://ksp.mff.cuni.cz/viz/kucharky/teorie-cisel`

Pro jehlu `abc...z` a seno `aabbcc...zz` bude tento algoritmus mít exponenciální časovou složitost – to by nevadilo, protože i výskytů jehly je exponenciálně mnoho (2^{26}). Horší ale je, že pro tutéž jehlu a seno `aabbcc...yy` strávíme exponenciální čas, načež (správně) vypíšeme, že v seně žádná jehla není.

Přeskakujeme slepé větve

Abychom pochopili, proč je předchozí algoritmus pomalý, představíme si jeho strom rekurze: kořen odpovídá startu algoritmu, jeho synové jsou jednotlivé výskyty prvního znaku jehly, jejich synové příslušné výskyty druhého znaku, atd. Pokud nějaká větev stromu pokračuje až do j-té hladiny, skončí vypsáním výskytu celé jehly. Jiné větve ale mohou skončit předčasně, takže nás stojí spoustu času, aniž přispějí k výsledku.

Hodilo by se tedy průběžně kontrolovat, zda zatím nalezenou část jehly jde rozšířit na aspoň jeden výskyt celé jehly – jinými slovy zda v podstromu, do kterého se právě chystáme zalézt, leží aspoň jeden list v hloubce j.

Půjde to snadno: pro $i = 1, \ldots, j$ předpočítáme $r(i)$, což bude nejpravější pozice v seně, za kterou ještě leží alespoň jeden výskyt znaků $J[i \ldots j]$. Zjevně $r(j)$ je nejpravější výskyt znaku $J[j]$, $r(j-1)$ nejpravější z výskytů znaku $J[j-1]$ ležících nalevo od $r(j)$, atd.

Kdykoliv pak v našem rekurzivním algoritmu hledáme znak $J[i]$, zastavíme se na pozici $r(i)$: kterýkoliv předchozí výskyt $J[i]$ půjde rozšířit na celou jehlu, kterýkoliv následující určitě nepůjde.

Jaké časové složitosti jsme dosáhli? Nejprve strávíme čas $\mathcal{O}(s)$ předvýpočtem všech $r(i)$. Poté spustíme rekurzi, ta vypíše všech v výskytů a na cestě do každého z nich projde nejvýše s znaků sena. Jelikož strom rekurze už neobsahuje žádné slepé větve, můžeme celkovou složitost omezit funkcí $\mathcal{O}(sv)$.

Předpočítáváme polohy

Krátká zkouška programátorské intuice: je předchozí řešení optimální? To je obecně těžká otázka, ale někdy nám pomohou úvahy typu „je potřeba aspoň přečíst celý vstup“ nebo „musíme aspoň vypsat celý výstup“. Přečtení vstupu trvá $\Omega(j+s)$, vypsání výstupu $\Omega(jv)$ – vypisujeme v výskytů a pro každý z nich j pozic znaků. Složitost našeho řešení je ale větší (aspoň pro $s \gg j$), tak pokračujme v přemýšlení.

Brzy přijdeme na to, že další pomalé místo je hledání výskytů znaků v seně. Hezky je to vidět na jehle `ab` a seně `a...ac...cb...b`: postupně zkoušíme všechna `a` a pro každé z nich musíme přeskočit všechna `c`, než se dobereme k prvnímu `b`.

Nabízí se předpočítat si pro každý znak seznam všech jeho výskytů v seně. Tento seznam ale nemůžeme používat celý, zajímají nás vždy jen výskyty ležící napravo od aktuální pozice v seně.

Sestrojíme si proto pomocný graf. Každý vrchol bude odpovídat jednomu výskytu písmene jehly v seně. Z vrcholu povedou dvě hrany (bude-li kam): *zelená* hrana do nejbližšího dalšího výskytu téhož znaku jehly, *červená* hrana do nejbližšího dalšího výskytu následujícího znaku jehly.

Tento graf snadno vytvoříme při průchodu senem pozpátku, přičemž si budeme pro každý znak jehly udržovat, kde jsme ho naposledy viděli. A abychom uměli rychle poznat, zda aktuální znak sena leží v jehle, předpočítáme si tabulku překládající znaky abecedy na pozice v jehle. Takto vytvoříme celý graf v čase $\mathcal{O}(j+s)$. Navíc můžeme do konstrukce grafu rovnou zabudovat ořezávání neperspektivních větví (rozmyslete si, jak).

Náš algoritmus na hledání všech výskytů pak naučíme pamatovat si, ve kterém vrcholu grafu se nachází, a kdykoliv bude chtít vyjmenovat všechny výskyty dalšího znaku, přejde jednou po červené hraně (tím najde první výskyt) a pak půjde po zelených hranách, dokud to půjde (aby našel ostatní výskyty).

Časovou složitost odvodíme opět ze stromu rekurze: strom má v listů, do každého z nich vede z kořene cesta délky j a v každém jejím vrcholu strávíme konstantní množství času nalezením znaku. Celkem tedy $\mathcal{O}(j+s+jv)$ včetně předvýpočtu, což je jistě optimální.

Program (C):
`http://ksp.mff.cuni.cz/viz/28-2-3.c`

Alternativní program (C) bez explicitní konstrukce grafu:
`http://ksp.mff.cuni.cz/viz/28-2-3-mj.c`

Martin „Medvěd“ Mareš & Vojta Sejkora

28-2-4 Útěk z města

Chceme pro všechny zločince najít únikovou cestu z města, tedy cestu za okraj mapy. Čtvercová síť nám zavání převodem na graf, byť si ještě budeme muset rozmyslet, jak přesně bude takový převod vypadat.

Hledání cesty by nás pak mohlo svádět porozhlédnout se mezi algoritmy právě na hledání cest, ale zkusme si problém nejprve trochu přeformulovat. Je vůbec možné, aby v daném městě (při dané mapě) uprchli všichni zločinci?

Máme velké množství možných cest a omezení na to, kolik zločinců může jednotlivými částmi cest proběhnout (každým políčkem jeden). Zajímá nás, jestli můžeme vybrat takové cesty, aby unikli všichni, resp. kolik zločinců dokáže uniknout. To už zní mnohem víc jako toky.

Ano, dopustili jsme se na vás drobné ošklivosti a zadali dvě kuchařkové úlohy, u této jsme to ale v zadání nepřiznali. Více informací o tocích naleznete v příslušné kuchařce,[8] tady se podíváme, jak je uplatnit na naši úlohu.

Pojďme začít s převodem čtvercové sítě na graf. Nabízí se standardní postup, kdy se z políček udělají vrcholy a sousední políčka se spojí hranou (resp. dvojicí orientovaných hran, protože toky obvykle definujeme pro orientovaný graf). Tyto hrany pak mohou dostat jednotkovou kapacitu, takže je smí použít jen jeden zločinec. Z políček s budovami a na ně žádné hrany nepovedou.

Tímto trikem jsme ovšem omezili hrany, nikoliv vrcholy. O něco formálněji, vynucujeme hranově disjunktní cesty, ale ne vrcholově disjunktní. Můžeme si třeba představit, že v původní síti jeden zločinec proběhne políčko svisle, druhý vodorovně. Jak z toho ven?

Každý vrchol si rozdělíme na dva, jeden bude sloužit jako vstup, druhý jako výstup, a tyto dva vrcholy spojíme hranou s jednotkovou kapacitou. Teď už platí, že každý vrchol bude použit nejvýše jednou. Musíme si ale pohlídat, že hrany spojující jednotlivá pojíčka povedeme mezi správnými „půlkami vrcholů“.

[8] `http://ksp.mff.cuni.cz/viz/kucharky/toky`

Řešení toho, že máme více počátečních i koncových vrcholů, je přímočaré. Přidáme si umělý zdroj, který spojíme hranou s jednotkovou kapacitou se všemi pobočkami, a podobně přidáme umělý stok, který spojíme se všemi okrajovými políčky.

Na takto upravený graf pak pustíme nějaký klasický tokový algoritmus, třeba Fordův-Fulkersonův z kuchařky. Jestli bude velikost nalezeného toku rovna počtu zločinců, mohou všichni uniknout (a naopak, pokud je tok menší, všichni uprchnout nemohou).

Naším původním úkolem ale bylo přímo nalézt cestu pro každého zločince. To už zvládneme jednoduše: z každé pobočky prohledáme graf tak, že se vydáme dál po hraně s jednotkovým tokem (ta z vlastností grafu musí být jen jedna), dokud se nedostaneme na okraj mapy.

Zbývá nám zamyslet se už jen nad složitostí. O Fordově-Fulkersonově algoritmu kuchařka slibuje, že má časovou složitost $\mathcal{O}(nm^2)$, kde n je počet vrcholů a m je počet hran. Dá se ale jednoduše ukázat, že pro jednotkové kapacity má složitost $\mathcal{O}(nm)$. Označíme-li počet políček jako N, máme $\mathcal{O}(2N) = \mathcal{O}(N)$ vrcholů a $\mathcal{O}(4N) = \mathcal{O}(N)$ hran, tedy složitost bude $\mathcal{O}(N^2)$.

Rekonstrukci cest pak zvládneme v čase $\mathcal{O}(n+m)$ (jelikož tok smí každý vrchol použít maximálně jednou, můžeme ho i my při prohledávání navštívit maximálně jednou, podobně s hranami), čili $\mathcal{O}(N)$. Celková časová složitost je tak $\mathcal{O}(N^2)$. Paměťová složitost je $\mathcal{O}(N)$.

Zmiňme ještě, že maximální tok lze hledat také pomocí Dinicova algoritmu, který má v našem případě časovou složitost $O(N^{\frac{3}{2}})$. Ostatní odhady zůstanou stejné, celková časová složitost se tedy zlepší, paměťová zůstane.

Program (C):
`http://ksp.mff.cuni.cz/viz/28-2-4.c`

Karolína „Karryanna“ Burešová

28-2-5 Hlídání věznice

Kdybychom přiřazovali jen denní směny, jednalo by se o úplně obyčejné hledání párování v bipartitním grafu: jednu partitu by tvořili bachaři, druhou bloky věznice. Chtěli bychom najít *perfektní* párování, tedy takové, jež spáruje všechny vrcholy. Jak bachařů, tak bloků proto musí být stejný počet, říkejme mu n.

V kuchařce jsme ukázali, jak tento problém převést na hledání maximálního toku ve vhodné síti: přidali jsme zdroj, z něj jsme dovedli hrany do všech vrcholů partity bachařů, a spotřebič, do kterého zase vedou hrany ze všech vrcholů partity směn. Všem hranám jsme nastavili jednotkovou kapacitu.

V této úloze potřebujeme místo jednoho perfektního párování najít dvě různá perfektní párování, která nemají společné hrany. Všimněte si, že ve sjednocení těchto dvou párování má každý vrchol grafu stupeň přesně 2. (Však to také grafoví teoretici rádi zobecňují: k-faktor se říká podgrafu, který obsahuje všechny vrcholy původního grafu a všechny mají stupeň přesně k. Perfektní párování je tedy 1-faktor, my hledáme 2-faktor.)

K nalezení 2-faktoru poslouží snadná úprava kuchařkového algoritmu: hranám ze zdroje a do spotřebiče nastavíme kapacitu 2, původním hranám grafu ponecháme kapacitu 1. Co se stalo? Každou hranu smíme použít jenom jednou, vrcholy v obou partitách až dvakrát. Takže hledáme tok, jehož velikost bude přesně $2n$. Rozmyslete si, že takový tok existuje právě tehdy, je-li v grafu 2-faktor.

Stačí nám tedy spustit Fordův-Fulkersonův algoritmus, aby nám našel maximální tok. Jak dlouho to potrvá? Pokud měl původní graf $2n$ vrcholů a m hran, naše síť má $n' = 2n+2$ vrcholů a $m' = m+2n$ hran. Jedna iterace Fordova-Fulkersonova algoritmu poběží v čase $\mathcal{O}(m')$ a zvětší tok alespoň o 1 (nezapomeňte, že všechno je celočíselné). Počet iterací proto nebude větší, než je velikost maximálního toku, což nepřesáhne $2n$ (omezeno např. hranami kolem zdroje). Celkem tedy algoritmus poběží v čase $\mathcal{O}(m'n') = \mathcal{O}(mn)$.

Zbývá nalezený 2-faktor rozebrat na denní a noční služby. Nejprve ho rozebereme na komponenty souvislosti a všimneme si, že každá komponenta musí být kružnice (souvislé grafy, jejichž všechny vrcholy mají právě 2 hrany, nemohou vypadat jinak). Navíc kružnice sudé délky, neboť se na ní pravidelně střídají partity. Stačí tedy (řekněme) sudé hrany prohlásit za denní služby a liché za noční. To zajisté zvládneme v čase $\mathcal{O}(m+n)$, takže nám to složitost nezhorší.

Program (C):
`http://ksp.mff.cuni.cz/viz/28-2-5.c`

Martin Mareš

28-2-6 Cesta MHD

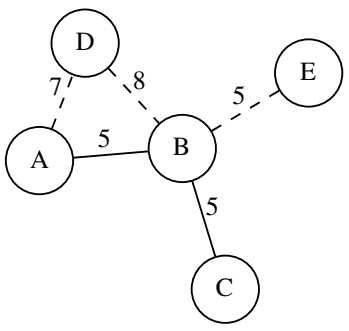
Hledáme nejdelší cestu v síti MHD, měřeno časem stráveným ve vozidlech. Vyřešíme nejprve lehčí verzi úlohy, tedy situaci, kdy si nemusíme nechávat časovou rezervu na přestup.

Formát vstupu

Zadání nespecifikovalo, v jakém formátu dostaneme jízdní řád. Asi nejobvyklejší způsob uložení jízdních řádů je jako množina *spojů*. Každý spoj popisuje jednu cestu vozidla na nějaké lince z konečné na konečnou. Tato cesta je zapsána jako posloupnost dvojic (zastávka, čas), v pořadí, v jakém je vozidlo na své cestě projede. Každé z těchto dvojic budeme říkat *zastavení* daného spoje.

Zastávky nechť máme očíslované 0 až $Z-1$. Časy budeme reprezentovat jako počet minut od půlnoci (tedy např. čas 270 představuje 4:30), tak s nimi můžeme pracovat jako s celými čísly (například je snadno porovnávat a odčítat). Spoje si očíslujeme 0 až $S-1$. Celkovou velikost jízdního řádu (tedy celkový počet zastavení přes všechny spoje) si označíme N.

To si zaslouží příklad. Představme si následující síť:



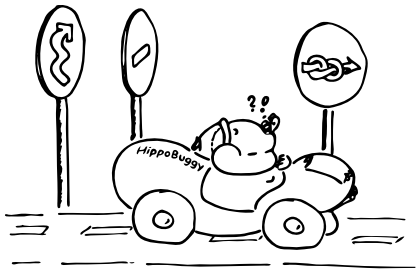
Síť je tvořena dvěma linkami, X (plná čára, jezdí každých 20 minut) a Y (čárkovaná čára, jezdí každých 30 minut). Zastávky jsou pro přehlednost označeny písmeny namísto čísel. Čísla na hranách značí jízdní dobu mezi příslušnými zastávkami.

Část jízdního řádu této sítě pro dobu mezi pátou a šestou hodinou (časy 300 a 360) by mohla vypadat takto (jeden řádek představuje jeden spoj):

```
A 300 B 305 C 310
A 320 B 325 C 330
A 340 B 345 C 350
C 300 B 305 A 310
C 320 B 325 A 330
C 340 B 345 A 350
A 300 D 307 B 315 E 320
A 330 D 337 B 345 E 350
E 300 B 305 D 313 A 320
E 330 B 335 D 343 A 350
```

Všimněte si, že nás vůbec nezajímá, že doprava je organizována do nějakých linek. Linka je prostě jen spousta spojů jedoucích ve stejné či podobné trase v určitém časovém odstupu. Každý z nich je v našem jízdním řádu uveden zvlášť, včetně vyjmenování všech zastávek na trase.

Může se zdát neefektivní pravidelnosti linek nevyužít, ale časem uvidíme, že optimálnímu algoritmu by stejně příliš nepomohla. Navíc ona ve skutečnosti zas tak pravidelná není. Třeba jízdní doby na dané lince se často různí v závislosti na denní době, aby odpovídaly hustotě dopravy.



Grafová reprezentace

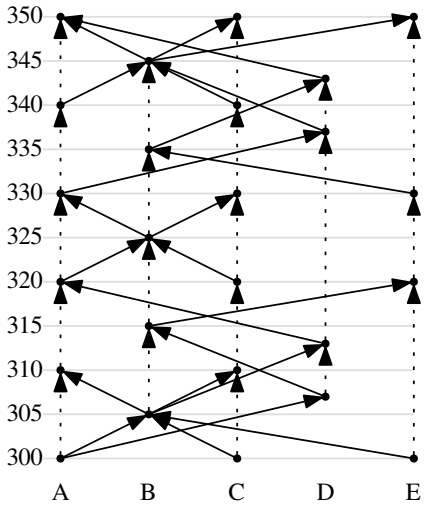
Hledáme cesty, to zavání nějakým grafem. Zkusme si tedy vytvořit graf popisující naši síť, takový, že cesty v něm budou odpovídat korektním cestám MHD. Určitě si nevystačíme s jednoduchým grafem, který má za vrcholy zastávky (jako ten na obrázku v předchozí sekci). Protože na jednu zastávku můžeme během našeho putování přijet víckrát, taková jízda by v našem grafu netvořila cestu, nýbrž sled (mohou se opakovat vrcholy). Se sledy se obvykle špatně pracuje, zkusme to tedy jinak.

Vytvoříme si takzvaný *stavový prostor*. To je graf, jehož vrcholy popisují nějaký stav (situaci), ve kterém se můžeme nacházet, a hrany mezi nimi určují dovolené změny stavu. V našem případě budou stavy dvojice (z, t) popisující „jsem na zastávce z v čase t“.

Jak se takový stav může změnit? Pokud v čase t odjíždí ze z nějaký spoj, jehož nejbližší další zastávka je z' a přijede na ni v čase t', pak určitě ze (z, t) povede hrana do (z', t'). Můžeme se tímto spojem svézt a tím se ocitnout na zastávce z' v čase t', tedy ve stavu (z', t').

Ale nemusíme nastoupit do prvního spoje, který jede, potřebujeme umět reprezentovat i to, že počkáme na nějaký další. To by se dalo udělat například tak, že vždy ze (z, t) povede hrana do $(z, t + 1)$. Pak bychom ale u každé zastávky museli mít vrcholy pro všechny možné časy, což by bylo neúsporné. Místo toho je vytvoříme jen pro ty časy, kdy v z něco zastavuje. A hrany pak povedou vždy ze (z, t) do (z, t'), kde t' je čas nejbližšího dalšího odjezdu/příjezdu po t.

Pro síť z příkladu výše bude stavový prostor vypadat takto:



Tečkované hrany odpovídají čekání na zastávce, plné přesunům vozidly.

Hledání nejdelší cesty

Pokud si teď cestovní hrany ohodnotíme dobou jízdy a čekací hrany nulou, bude délka každé cesty odpovídat času, který strávíme ve vozidlech. Tedy nám stačí najít nejdelší cestu a máme vyhráno. Hledání nejdelší cesty v grafu je obecně těžký (přesněji NP-úplný)[9] problém. Ale můžeme si všimnout, že náš stavový prostor je acyklický orientovaný graf (DAG) – neobsahuje žádné orientované cykly. Tedy alespoň pokud součástí MHD nejsou stroje času.

V DAGu umíme nejdelší cesty hledat snadno pomocí topologického uspořádání (pokud jste tento pojem nikdy neslyšeli, nahlédněte do naší grafové kuchařky).[10] Budeme chtít každý vrchol u ohodnotit délkou $D(u)$ nejdelší cesty z něj vycházející. Snadno si rozmyslíte, že pokud $S(u)$ je množina následníků u (tedy vrcholů, do kterých vede z u hrana), pak

$$D(u) = \max_{v \in S(u)} d_{uv} + D(v),$$

kde d_{uv} je délka hrany uv. Pokud u nemá žádného následníka, zřejmě $D(u) = 0$.

Pokud budeme vrcholy postupně ohodnocovat v obráceném topologickém pořadí (od posledního), budeme při zpracování u už znát ohodnocení všech následníků, takže stačí dosadit do vzorečku a máme $D(u)$.

Budeme si navíc průběžně udržovat maximální dosud nalezené D, to bude na konci právě délka nejdelší cesty. Pokud bychom kromě délky chtěli vypsat i celou nalezenou cestu, můžeme si navíc ke každému vrcholu ukládat, přes kterého následníka nejdelší cesta vede (pro které v nabyl maxima výraz výše). Podrobněji ve zdrojáku.

Vytvoření grafu

Už umíme za pomoci stavového grafu úlohu vyřešit, ale jak jej vytvořit? Budeme postupně načítat vstup a dvojicím (z, t) přiřazovat čísla vrcholů (již přiřazená čísla si pamatujeme ve slovníku, nově objevené dvojici přiřadíme další volné číslo v pořadí). Zároveň si pro každý vrchol budeme

[9] http://ksp.mff.cuni.cz/viz/kucharky/tezke-problemy
[10] http://ksp.mff.cuni.cz/viz/kucharky/grafy

postupně vytvářet seznam jeho následníků a pro každou zastávku seznam všech vrcholů, které k ní patří. Potom pro každou zastávku tento seznam setřídíme a vytvoříme čekací hrany (každému vrcholu přidáme do seznamu jeho následníků nejbližší další vrchol v seznamu pro danou zastávku). Podrobněji opět ve zdrojáku.

Jaká bude časová složitost? Na třídění spotřebujeme čas $\mathcal{O}(N \log N)$, kde N je celková velikost jízdního řádu. Zbytek vytváření grafu, topologické seřazení i nalezení nejdelší cesty zvládneme v lineárním čase, tedy celé řešení stihneme v $\mathcal{O}(N \log N)$. Paměti spotřebujeme lineárně.

Těžší varianta

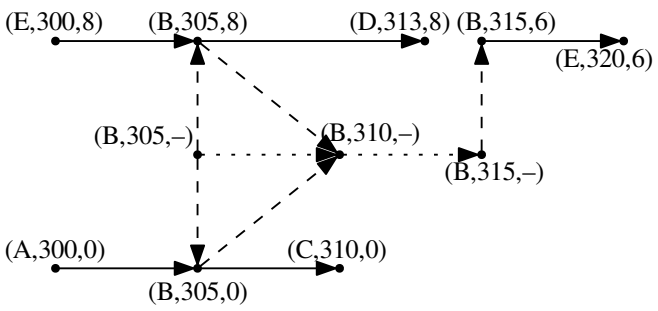
V těžší variantě si chceme na každý přestup nechat λ minut rezervu. Tady je náš dosavadní stavový prostor neadekvátní. Protože to, kam můžeme pokračovat např. z vrcholu $(B, 305)$ záleží na tom, kudy jsme do něj přijeli. Pokud jsme přijeli z vrcholu $(A, 300)$ spojem linky X, můžeme například pokračovat dál stejným spojem do vrcholu $(C, 310)$. Ale pokud jsme přijeli z $(E, 300)$ linkou Y, do $(C, 310)$ se vydat nemůžeme, protože bychom museli v B přestoupit s nulovou rezervou.

Namísto původní hrubé informace „jsem na zastávce z v čase t“ se budeme muset naučit rozlišovat mezi „stojím na zastávce z (venku) v čase t“ a „jsem ve vozidle spoje s, které právě zastavilo na z v čase t“. Tedy stav bude popsán trojicí (z, t, s), kde s je číslo spoje nebo „$-$“ reprezentující „stojím venku“.

Zbývá správně natahat hrany. Čekací hrany povedou jen mezi „venkovními“ vrcholy, tedy ze $(z, t, -)$ do $(z, t', -)$, kde t' je opět čas nejbližšího dalšího odjezdu/příjezdu. Zároveň přidáme hrany popisující nástup do vozidla: pro vozidlo spoje s odjíždějící ze z v čase t přidáme hranu $(z, t, -) \to (z, t, s)$. Ještě celkem přímočaré budou hrany popisující cestu ve vozidle: pokud spoj s jede ze z (odj. t) do z' (příj. t'), přidáme hranu $(z, t, s) \to (z', t', s)$.

Hlavní trik spočívá ve hranách popisujících výstup z vozidla. Ty budou mít tvar $(z, t, s) \to (z, t+\lambda, -)$. Můžeme si to představovat tak (formulace z řešení Václava Volhejna), že každé vozidlo na zastávku přijede λ minut po tom, co z ní odjede. Případně pokud je to na vás příliš sci-fi, můžete si představit, že vám λ minut trvá vystoupit z vozidla. Každopádně si snadno rozmyslíte, že touto úpravou zařídíme dodržení času na přestup.

Malý kousek nového grafu ukazující přestupy v B okolo času 305 (pro $\lambda = 5$):



Tečkované hrany jsou opět čekací a plné jízdní, přibyly čárkované nástupní a výstupní. V tomto grafu už se z $(A, 300)$ dostaneme do $(C, 310)$, ale z $(E, 300)$ nikoli. V obou případech si navíc můžeme v B počkat do 315, přestoupit na spoj 6 a pokračovat dál.

Graf sestrojíme analogicky lehké verzi a zbytek algoritmu je stejný. Stejná zůstává i složitost.

Na závěr ještě podotkněme, že tímtéž algoritmem byste mohli hledat nejen nejdelší cestu, nýbrž i nejkratší (jen v definici D vyměníte maximum za minimum), a to i mezi konkrétní dvojicí vrcholů. Tím byste si vyrobili jednoduché vyhledávátko spojení typu IDOS.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/28-2-6.py`

Filip Štědronský

28-2-7 Otevření kufříku

Úloha po nás chce spočítat, kolik z čísel $A, A+1, \ldots, B$ má ve svém dvojkovém zápisu právě k jedniček. Hledaný počet označíme $p_k(A, B)$. Rovnou si všimneme, že úlohu stačí umět vyřešit pro $A = 0$, protože platí $p_k(A, B) = p_k(0, B) - p_k(0, A-1)$.

Jednodušší varianta: pomohou kombinační čísla

V lehčí variantě úlohy máme spočítat $p_k(0, 2^n - 1)$. Všimneme si, že čísla $0, \ldots, 2^n - 1$ jsou přesně ta, která se ve dvojkové soustavě dají zapsat pomocí n číslic (povolíme-li nuly na začátku čísla). Ptáme se tedy, kolika způsoby lze z n míst vybrat k, na nichž budou jedničky.

Kdo už se někdy potkal s kombinatorikou, ví, že tento počet je roven *kombinačnímu číslu* „n nad k“:

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}{k \cdot (k-1) \cdot \ldots \cdot 1}.$$

Pokud se ještě s kombinačními čísly neznáte, případně pokud jste pro ně viděli nějaký jiný vzoreček, zde je stručné vysvětlení: Představme si na chvíli, že na n pozic chceme místo k nerozlišitelných jedniček umísťovat čísla 1 až k. Nejprve umístíme 1: to lze udělat n způsoby. Pro 2 už je volných pouze $n-1$ pozic, …, až pro k jich je $n-k+1$. To nám celkem dává $n \cdot (n-1) \cdot \ldots \cdot (n-k+1)$ možností.

Tentýž počet ale musí vyjít, když nejprve zvolíme k-tici pozic, na kterém chceme něco umístit (to lze udělat $\binom{n}{k}$ způsoby), a poté budeme vybírat pouze z nich: 1 můžeme umístit na k pozic, 2 na $k-1$, …, až pro k už zbývá jen jediná pozice. Proto musí platit $\binom{n}{k} \cdot k \cdot \ldots \cdot 1 = n \cdot \ldots \cdot (n-k+1)$, což je ekvivalentní s naším vzorcem.

Dobrá, vzoreček už máme, tak ho pojďme použít v algoritmu: jak čitatele, tak jmenovatele spočítáme v čase $\mathcal{O}(k)$ a pak je v konstantním čase vydělíme.

Jenže ouha …

Předchozí řešení má jeden háček, neřkuli hák: čitatel i jmenoval zlomku mohou být ohromná čísla, a to i v případech, kdy finální výsledek vyjde maličký: rozmyslete si třeba, co se stane pro $k = n-1$.

Pomůže přeházet pořadí násobení a dělení a počítat

$$\binom{n}{k} = n/1 \cdot (n-1)/2 \cdot (n-2)/3 \cdot \ldots \cdot (n-k+1)/k.$$

Všechny mezivýsledky jsou přitom celočíselné, protože to jsou kombinační čísla $\binom{n}{1}$, $\binom{n}{2}$, atd. Navíc je-li $k \leq n/2$, pak tyto mezivýsledky postupně rostou, takže nejsou nikdy větší než finální výsledek.

Pro $k > n/2$ použijeme válečnou lest: všimneme si, že $\binom{n}{k}$ je totéž jako $\binom{n}{n-k}$. To proto, že hledat k míst pro jedničky vyjde nastejno jako hledat $n-k$ míst pro nuly. Opět jsme dostali algoritmus se složitostí $\mathcal{O}(k)$, tentokrát už bez aritmetiky s obřími čísly.

Obecný případ

Uvažujme nyní, jak spočítat $p_k(0, B)$ pro obecné B. Označme h pozici nejvyššího jedničkového bitu čísla B (pozice číslujeme zprava od nuly, takže tento bit má váhu 2^h). Nyní čísla od 0 do B rozdělíme na dvě skupiny podle toho, jaký bit mají na h-té pozici:

- Nulu tam mají čísla $0, \ldots, 2^h - 1$ a mezi nimi je přesně $p_k(0, 2^h - 1) = \binom{h}{k}$ čísel s právě k jedničkami.
- Jedničku tam mají čísla $2^h, \ldots, B$. Ta mají jednu jedničku jistou, takže potřebujeme na zbývajících h míst naskládat $k-1$ jedniček, a to tak, aby zbytek čísla nepřesáhl $B - 2^h$.

Proto musí platit:

$$p_k(0, B) = \binom{h}{k} + p_{k-1}(0, B - 2^h).$$

To nám dává hezký rekurzivní algoritmus, který se zastaví buďto o $p_0(0, B) = 1$ (číslo s 0 jedničkami je právě jedno, a to 0), nebo o $p_k(0, 0) = 0$, $k > 0$.

Rekurze má hloubku k, pokaždé strávíme čas $\mathcal{O}(k)$ počítáním kombinačního čísla a $\mathcal{O}(\log B)$ hledáním pozice nejvyšší jedničky. Celkem tedy $\mathcal{O}(k \cdot (k + \log B))$, což můžeme zjednodušit na $\mathcal{O}(k \log B)$, protože pro $k > \log B$ je výsledek evidentně nulový.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/28-2-7-slow.py`

Rychlejší řešení

Rekurzivní vztah z předchozího řešení můžeme snadno rozepsat, uvědomíme-li si, že $B - 2^h$ je číslo B bez nejvyšší jedničky. Vyjde nám

$$p_k(0, B) = \binom{h_1}{k} + \binom{h_2}{k-1} + \cdots + \binom{h_{k+1}}{0}, \qquad (*)$$

kde h_i je pozice i-té jedničky zleva v čísle B. (Pokud by v B bylo méně než $k + 1$ jedniček, součet zkrátíme.)

Hned je jasné, že všechny jedničky můžeme najít jedním průchodem dvojkovým zápisem čísla B v čase $\mathcal{O}(\log B)$. Pak stačí spočíst k kombinačních čísel, každé v čase $\mathcal{O}(k)$. Celý výpočet tedy potrvá $\mathcal{O}(k^2 + \log B)$.

I zde je stále prostor pro zlepšování. Podle našeho vztahu pro kombinační čísla totiž platí:

$$\binom{n-1}{k} = \binom{n}{k} \cdot \frac{n-k}{n},$$
$$\binom{n-1}{k-1} = \binom{n}{k} \cdot \frac{k}{n}.$$

Díky tomu můžeme v čase $\mathcal{O}(k)$ spočítat $\binom{h_1}{k}$, z něj „doskákat“ do $\binom{h_2}{k-1}$, z něj do $\binom{h_3}{k-2}$, atd. Každý skok přitom trvá $\mathcal{O}(1)$ a sníží horní parametr kombinačního čísla o 1, takže všechny skoky dohromady nemohou trvat více než $\mathcal{O}(h_1) = \mathcal{O}(\log B)$.

Algoritmus tedy stráví $\mathcal{O}(\log B)$ hledáním jedniček, pak $\mathcal{O}(k)$ výpočtem prvního kombinačního čísla a $\mathcal{O}(\log B)$ skákáním k těm dalším. Jelikož $k \leq \log B$, vše se sečte na $\mathcal{O}(\log B)$.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/28-2-7-fast.py`

Dezert na závěr hostiny: případy s málo jedničkami

V případech, kdy je k mnohem menší než $\log B$, existuje ještě rychlejší, byť o trochu šílenější algoritmus. Vraťme se ke vztahu $(*)$ z minulého řešení. Co nás pro malé k při jeho výpočtu brzdí? Kombinační čísla to nejsou, ta hravě spočítáme v $\mathcal{O}(k^2)$. Ale potřebujeme najít pozice všech jedniček v čísle B, což jsme zatím dělali v čase $\mathcal{O}(\log B)$. Ukážeme, jak to provést rychleji.

Pozici h nejvyšší jedničky v čísle B budeme hledat půlením intervalu. Na chvíli předpokládejme, že víme, že číslo B má nejvýše t bitů. Pozici h tedy můžeme hledat binárně v intervalu $\langle 0, t - 1\rangle$. Provádíme $\mathcal{O}(\log t)$ pokusů, v každém potřebujeme zjistit, zda nejvyšší jednička leží vlevo nebo vpravo od nějaké pozice i. To ověříme v konstantním čase porovnáním $B < 2^i$. Celkem hledaním strávíme čas $\mathcal{O}(\log t)$.

Jenže ve skutečnosti t neznáme. Tak budeme zkoušet postupně $t = 2^0, 2^1, 2^2, \ldots$, až najdeme takové t, pro něž $2^{t/2} \leq B < 2^t$. Pak spustíme půlení intervalu $\langle t/2, t - 1\rangle$. Nalezené t přitom leží někde v intervalu $\langle 1/2 \cdot \log B, \log B\rangle$, takže jak hledání t, tak následné půlení trvají $\mathcal{O}(\log t) = \mathcal{O}(\log \log B)$.

Tak jsme našli nejvyšší jedničku, další získáme jejím odstraněním a opakováním postupu. Celkem tedy k-krát trávíme čas $\mathcal{O}(\log \log B)$ hledáním jedničky a k-krát $\mathcal{O}(k)$ výpočtem kombinačního čísla, což dá dohromady $\mathcal{O}(k^2 + k \cdot \log \log B)$.

Nedosti na tom, nejvyšší jedničku lze najít i v konstantním čase a získat tak algoritmus o složitosti $\mathcal{O}(k^2)$, naprosto nezávislý na B. Zájemce o detaily odkážeme na kapitolu o výpočetních modelech v průvodci *Krajinou grafových algoritmů.*[11]

Martin „Medvěd“ Mareš

28-2-8 Genetika vs. procházení krajiny

Bez zbytečného okecávání pojďme rovnou na řešení úloh :)

Úloha 1

Úloha přímo vybízela k aplikaci metody horolezení či simulovaného žíhání. Souřadnice základen budou určovat bod v krajině a náhodné změny budeme dělat pomocí jejich náhodného posunutí o kousek vedle.

Oba algoritmy mohly fungovat dobře, ale pojďme se zamyslet, který z nich je vhodnější. Pro $k = 1$ má funkce jen jedno lokální minimum, které je zároveň globálním. Ať tedy začneme kdekoliv, tak se do globálního maxima určitě dostaneme tak, že „půjdeme pořád z kopce“, tj. budeme přijímat jen změny k lepšímu.

Pro $k = 3$ a $k = 5$ již sice lokálních minim máme více, ale pořád ne až tak moc a funkce je stále pěkně hladká. Takže pokud použijeme metodu horolezení, tak řádově za desítky pokusů natrefíme na globální minimum. (Aspoň mně to stačilo :-P)

Simulované žíhání samozřejmě bude fungovat taky. Akorát na začátku, kdy máme s velkou pravděpodobností povolené změny k horšímu, nám to bude zpomalovat postup. Až ale teplota dost klesne, tak také sklouzne do některého minima.

Nyní k maximální velikosti skoku. Tu na začátku zvolíme větší, třeba 100, aby se algoritmus mohl rychle rozběhnout do správné oblasti, a pak ji pomalu snižujeme, abychom více a více konvergovali do jednoho místa. Já jsem například

[11] `http://mj.ucw.cz/vyuka/ga/`

každou desátou iteraci povolenou velikost skoku vynásobil 0,95 s tím, že jsem zakázal jít pod hodnotu 10^{-6}. Tím děláme čím dál menší skoky a hodnoty postupně upřesňujeme. Za 10 000 iterací dost jistě zkonvergujeme i pro $k = 5$.

Poslední, nad čím se zamyslíme, je, jak budeme hledat okolní body změny. Určitě můžeme každou ze stanic náhodně posunout v každé souřadnici. Určitě se občas stane, že se takto posuneme do lepšího řešení, a metoda horolezení bude fungovat.

Posunutí všech základen je ale přeci jen celkem velká a náhodná změna. Co když jednu posuneme dobrým směrem a další dvě špatným? Pak výsledná změna bude nevýhodná a musíme tipovat znova. Nebylo by lepší posouvat vždy jen jednu základnu? Ano, pro konvergenci je to určitě lepší, protože u jedné základny máme větší šanci, že se trefíme do správného směru, než když hýbeme se všemi najednou.

Tuto úlohu šlo vyřešit posouváním vždy jen jedné základny. Na druhou stranu bychom ale posouvání více/všech základen neměli zatracovat, protože těmi se naopak můžeme dobře dostat z „mrtvých bodů“. Například když nám posunutí jen jedné základny nepomůže, zatímco správné posunutí více základen pomoct může.

Optimální hodnoty řešení byly 43 315,3 pro $k = 1$, dále 19 932,7 pro $k = 3$ a konečně 15 889,0 pro $k = 5$, což většině z vás vyšlo. Můžete také nahlédnout do vzorového kódu.

Program (C++):
`http://ksp.mff.cuni.cz/viz/28-2-8a.cpp`

Úloha 2

Tato úloha byla náročná. Vyžadovala velkou míru trpělivosti, snahy, generování a zkoušení nových nápadů. Už vůbec pro samotné napsání správné fitness funkce byl potřeba dostatek soustředěnosti. Ta se dala napsat v $\mathcal{O}(n)$, ale my si vystačíme s její přímočarou kvadratickou verzí (pro tento počet obdélníků to zas takové zdržení není). A jak napovídá učební text, zkusíme úlohu řešit pomocí diferenciální evoluce.

Pokud jsme pustili program s hodnotami ze šablony, mohli jsme se za několik běhů dostat na řešení s celkovým překryvem kolem 3 000 – 3 500. Důležité ale bylo zvednout počet iterací aspoň do řádů jednotek tisíců, protože řešení konvergovalo celkem pomalu.

Laděním parametrů jsme řešení mohli vylepšit až řádově na 2 200 – 2 800, pokud jsme byli trpěliví a nechali algoritmus běžet v hodně iteracích a hodně běžích, tak ještě trochu více.

To bylo ale v případě, kdy jsme generovali náhodnou počáteční populaci, tedy když jsme začínali s naprosto náhodně rozházenými obdélníky. My teď na moment opustíme evoluci a zkusíme na sebe obdélníky naskládat nějak „hezčeji“ algoritmicky. Třeba můžeme obdélníky brát v náhodném pořadí a dávat je za sebe po řádcích, přičemž další řádek začneme podle výšky nejvyššího obdélníka z řádku předchozího. Když nám dojde místo, tak začneme stavět druhou vrstvu. To samé taky můžeme zkusit po sloupcích.

Předchozím způsobem jsme mohli nagenerovat řadu různých řešení s překryvem zhruba 2 300 – 4 000. Pokud jsme navíc obdélníky přidávali v pořadí podle výšky dostali jsme se na řešení 1 875. To je dokonce lepší, než jsme to dokázali evolučně!

Takovým způsobem si můžeme nagenerovat řadu relativně pěkných řešení, kde každé má obdélníky jinak rozmístěné. Tak co takovou sadu řešení zkusit použít jako počáteční populaci? Když to zkusíme, tak hned dostaneme řešení o hodnotě kolem 1 200 – 1 300.

A co dál? Zkusíme pokračovat v podobné myšlence. Zjistili jsme, že když použijeme sadu relativně dobrých jedinců, tak tím získáme ještě lepšího. Řešený problém má navíc takovou povahu, že má velkou spoustu optimálních minim. Možností, jak vedle sebe naskládat obdélníky, je zkrátka hodně. Tak můžeme v několika běžích vypěstovat různě vypadající dobré jedince, ty pak vzít a použít je jako novou počáteční populaci. A to provádět stále dokola.

Ale ještě k nim vždy přidáme pár náhodných, protože ti nám můžou přinést nějakou náhodnou užitečnou informaci. Dokonce i můžeme vždy vzít jen jednoho nejlepšího a zbytek náhodný, to sice bude mít menší variabilitu, ale práci to značně urychlí. Podobnými postupy se můžeme dostat na řešení o hodnotách zhruba 600 – 800, opět záleží, jak moc budeme trpěliví.

Nejlepší řešení odevzdal Vašek Volhejn, který se dostal na překryv 423, čímž mu gratulujeme. Ten použil myšlenku iterovaného opakování evoluce s nejlepšími a novými náhodně generovanými jedinci plus do řešení přidal další operátor s následující myšlenkou. Pokud v řešení máme dva obdélníky, tak se může vyplatit tyto obdélníky vyměnit (protože by oba měly být na relativně dobrých pozicích). Tento operátor dává v úloze velmi dobrý smysl a pomohl mu dostat se ze stavů, kdy evoluce stagnovala na místě a nepřicházela na nic nového.

Teoretické optimum byl překryv 26, kterého kvůli velké různosti obdélníků pravděpodobně nešlo dosáhnout. Řešení v řádu několika stovek tedy určitě není žádná ostuda. Vzorovým kódem algoritmu je pouze jedno puštění diferenciální evoluce a zalogování nejlepšího jedince. Iterovaná evoluce v kódu není přímo implementována.

Program (C++):
`http://ksp.mff.cuni.cz/viz/28-2-8b.cpp`

Karel Tesař

Výsledková listina druhé série dvacátého osmého ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*2-1*	*2-2*	*2-3*	*2-4*	*2-5*	*2-6*	*2-7*	*2-8*	*série*	*celkem*
0.					10	8	10	10	10	12	10	16	58,0	117,0
1.	Václav Volhejn	GKepleraPH	3	17	10	8	10	10	10	12	9	16	58,0	117,0
2.	Jakub Pelc	G UherBrod	2	2	9	7,5	10		10			12	49,1	103,7
3.	Jan Bouček	GKepleraPH	3	6		8		7	1		9	15	42,3	97,4
4.	Pavel Turek	GTomkovaOL	3	2	6	8	1			9	8		39,2	87,5
5.	Richard Hladík	GOAMarLaz	3	17	10	8	4				10		30,0	84,0

	řešitel	*škola*	*ročník*	*sérií*	*2-1*	*2-2*	*2-3*	*2-4*	*2-5*	*2-6*	*2-7*	*2-8*	*série*	*celkem*
6.	Jiří Sejkora	GVoděraPH	4	5	9	1	10		10			11	44,3	80,4
7.	Leonard Mentzl	GŘíč	3	2	2,5	7		4		8	7		39,1	76,9
8.	Stanislav Lukeš	GPísnickáPH	3	8	10	8				6	1		26,1	74,8
9.	Josef Gajdůšek	SŠKKamPard	3	2	5	0	4		1		9		25,2	65,9
10.	Vojtěch Lukeš	GPikaPL	4	2			0		8				8,0	62,0
11.	Jonáš Fiala	GJungmanLT	3	2	5	1	0		5				15,0	55,5
12.	Michal Töpfer	G DrJPekMB	3	7		4					5	6	16,3	48,5
13.	Václav Pavlíček	ZŠ Ždírec nD	0	2		2			0	1	3		12,1	46,7
14.	Petr Chmel	G_Kralupy	3	2			1		5				7,4	43,3
15.	Lukáš Vlček	GMikulášPL	2	2	2,5	1	0		2				8,0	39,1
16.	Jakub Dobrý	GMikulášPL	2	2	2,5	1							7,3	38,5
17.	Ján Chudý	GŽilina	4	1									0,0	37,7
18.	Pavel Turinský	G Brandýs	3	7	4	8			1		3		17,9	35,9
19.	Ondrej Pudiš	GŽilina	4	1									0,0	34,2
20.	Václav Šraier	GČeskoliPH	3	7			1						1,4	32,9
21.	Miroslav Hrabal	GTomkovaOL	2	1									0,0	29,7
22.	Jakub Lukeš	GNAlejíPH	3	3			0						0,0	29,2
23.	Vanda Hendrychová	GHeyrovPH	4	1									0,0	28,7
24.	Vladimír Bartovic	G AM Trnava	4	2			0		1				1,0	28,0
25.	Roman Beňo	GJHroncaBA	3	2		7							7,6	25,6
26.	Jiří Vozár	G UherBrod	4	7		1	1						2,8	25,2
27.	Zdenko Čepan	GPartizans	3	2		1			0		3		8,0	22,6
28.	Michal Jireš	GRNK	1	1									0,0	22,5
29.–30.	Jakub Tětek	Dollar Ac	2	7	10								10,0	22,0
	Přemysl Šťastný	GŽamberk	3	9	5	6	10				1		22,0	22,0
31.	Marco Souza de Joode	GNadŠtolPH	−1	2		1							2,3	20,8
32.	Michal Kodad	ZŠJílovsPH	0	2		6			1				8,3	20,7
33.	Sam Friedlaender	GKepleraPH	−1	2		1							2,3	20,3
34.–35.	Lukáš Rozsypal	GÚstavníPH	3	1	6	8	1		1				20,1	20,1
	David Ucháč	eduSOŠ PA	3	1									0,0	20,1
36.	Jiří Löffelmann	GLitoměřPH	2	1									0,0	20,0
37.	Jan Pokorný	G Bučovice	4	8	10	7,5							17,7	17,7
38.	Alexej Popovič	SlovanGOL	4	2		1							2,1	16,1
39.	Alena Tesařová	GVídeňskBO	4	1		1	0			2	3		13,2	13,2
40.	Daniel Herman	GŠKo	3	1	2,5	7							13,1	13,1
41.	Filip Geib	G MMH LM	2	1									0,0	12,5
42.	Petr Gebauer	GMělník	2	1									0,0	10,6
43.–46.	Jan Gocník	GJŠkodyPŘ	4	4									0,0	10,0
	Jan Priessnitz	GJarošeBO	3	1									0,0	10,0
	Filip Šohajek	GUHradiště	−1	1									0,0	10,0
	David Žáček	GZborovPH	3	2									0,0	10,0
47.	Jan Neumann	GNAlejíPH	2	1									0,0	9,0
48.–52.	David Blažek	SPŠÚžlabPH	3	1									0,0	8,0
	František Kmječ	G Brandýs	0	1									0,0	8,0
	Lucie Kubíčková	GFXŠaldyLI	2	1									0,0	8,0
	Antonín Prantl	G Strakon	3	1									0,0	8,0
	Zuzana Svobodová	G FrýdlNOs	4	2									0,0	8,0
53.	Jakub Matěna	GČeskoliPH	4	4									0,0	4,3
54.	Lukáš Mičan	GČeskáČB	2	1									0,0	4,0
55.	Ondřej Borýsek	GJarošeBO	3	2									0,0	3,3
56.	Adam Husník	GArabskáPH	2	1									0,0	3,0
57.	Jiří Muller	G_Roudnice	3	1									0,0	2,5
58.	David Nápravník	GLitoměřPH	3	1			1						2,2	2,2
59.–60.	Michael Bausano	GTěš	4	1									0,0	2,0
	Martin Zoula	GNadKavaPH	4	4									0,0	2,0
61.	Peter Matta	G KošiceS	4	1									0,0	1,0