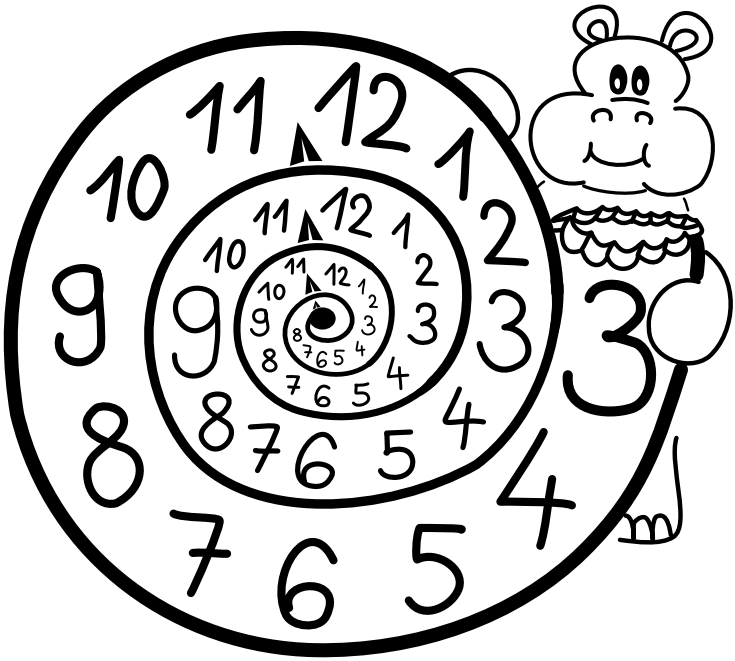
JIŘÍ SETNIČKA A KOLEKTIV

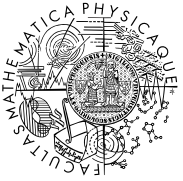
Korespondenční seminář z programování

XXVIII. ročník – 2015/2016



matfyzpress

NAKLADATELSTVÍ MATEMATICKO-FYZIKÁLNÍ FAKULTY
UNIVERZITY KARLOVY



MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

JIŘÍ SETNIČKA A KOLEKTIV

Korespondenční seminář z programování

XXVIII. ročník – 2015/2016

matfyzpress

Praha 2016

© Jiří Setnička a kolektiv, 2016
© MatfyzPress, nakladatelství Matematicko-fyzikální fakulty
Univerzity Karlovy, 2016

ISBN 978-80-7378-330-3

Úvod

Letos pokračoval Korespondenční seminář z programování (dále jen *KSP*) již svým dvacátým osmým ročníkem. Po dobu své historie patří k nejznámějším aktivitám pro zájemce o informatiku a programování z řad studentů (nejen) středních škol. Díky aktivnímu zapojování se do řešení úloh získalo mnoho řešitelů praxi ve zdolávání nejrůznějších algoritmických problémů, jakož i hlubší náhled do tajů informatiky.

Od 26. ročníku je KSP rozděleno do dvou kategorií, hlavní a začátečnické. Obě kategorie jsou rozděleny do několika *sérií*, hlavní do pěti, začátečnická do čtyř. Na začátku série pošleme řešitelům zadání sady úloh. Ty jsou různého typu, některé teoretické (úkolem je vymyslet a popsat efektivní algoritmus), některé praktické (úkolem je algoritmus nejen vymyslet, ale také naprogramovat a odladit). V hlavní kategorii bývá navíc zařazen *seriál*, který je kromě soutěžních úložek tvořen zejména povídáním o nějakém zajímavém informatickém tématu; seriál je rozložený do celého ročníku a jeho díly v jednotlivých sériích na sebe navzájem navazují.

Řešitelé pak mají několik týdnů na to, aby si úlohy rozmysleli a dali dokupy jejich řešení, které nám odevzdají. Výsledky praktických úloh mají řešitelé k dispozici hned po odevzdání, řešení teoretických úloh po termínu série opravíme, okomentujeme a pošleme zpět.

Jako primární metody komunikace používá KSP různé elektronické způsoby komunikace, webem počínaje a emaily konče, ale věříme i na metody papírové komunikace a tak každé zadání i řešení putuje v mnoha výtiscích řešitelům po celé republice.

Velkou událostí jsou dvě týdenní *soustředění*. Jarní je určené hlavně řešitelům začátečnické kategorie, ale přihlášky otvíráme i pro ty, kteří s programováním zatím nemají žádné zkušenosti a chtěli by se ho naučit. Podzimní soustředění probíhá na začátku následujícího ročníku a zveme na něj primárně nejlepší řešitele hlavní kategorie. V obou případech je pro účastníky soustředění připravený bohatý program od odborných přednášek na informatická témata až po zcela neodborné hraní a dovádění v přírodě. Navíc mají účastníci možnost potkat další lidi s podobnými zájmy.

KSP v průběhu let rozrůstá a mimo korespondenčního řešení úloh a přípravy soustředění stíháme toho během ročníku mnohem víc.

Na podzim proběhl ji čtvrtý ročník Putovních přednášek[1] na středních školách po celé republice. Naše spřátelená, ale nyní již samostatná soutěž Kasiopea[2] uspořádala svůj první ročník ve velkém formátu. A také již finišujeme přípravu nového knižního vydání našich Programátorských kuchařek.

[1] `http://ksp.mff.cuni.cz/akce/putovni-prednasky/2015/`

[2] `http://kasiopea.matfyz.cz`

(Nejen) u úloh v této knize lze zahlédnout několik značek pro urychlení orientace:

Některé značky používáme primárně k označení typu úlohy:


V začátečnické kategorii tímto symbolem označujeme teoretické úlohy, tedy ty „klasické“. Úkolem řešitelů je vymyslet efektivní algoritmus, slovně ho popsat a tento popis odevzdat. (V hlavní kategorii jsou teoretické všechny úlohy, které nejsou přímo označené jako praktické.)


Tento symbol označuje praktickou úlohu, konkrétně takovou, které říkáme *open-data*. Úkolem řešitelů je nejen vymyslet algoritmus, ale také ho zapsat jako program a tento program odladit. Odevzdání probíhá tak, že si řešitel stáhne vstupní data a odevzdá příslušný výstup, přičemž počet pokusů není omezen.


I tato úloha je praktická, obvykle pro ni ale používáme termín *codexovka*, podle systému, do kterého se odevzdává. Řešitel napíše a odladí program, který pak nahraje do systému, kde je puštěn na neveřejných datech.


V každém ročníku KSP rozebíráme na pokračování nějaké zajímavé informatické téma do hloubky. Poslední úloha série je pokračováním takového *seriálu* – obsahuje kromě samotného zadání ještě text, ve kterém se můžete dozvědět o tématu něco nového. Jelikož díly seriálu na sebe navazují, vyplatí se mít nastudované i předchozí série.

Jiné značky slouží k označení obtížnosti a doporučených zdrojů inspirace:


Takto označenou úlohu považujeme za řešitelnou i pro začátečníky, zkušení řešitelé ji jistě zvládnou levou zadní. Pro její vyřešení by neměly být potřeba žádné speciální znalosti.


Aby si i pokročilí přišli na své, zařazujeme někdy do zadání těžkou úlohu, která se může stát leckomu noční můrou. Na její pokoření jsou často potřeba hlubší znalosti algoritmů a datových struktur, odměnou je však vyšší bodový zisk.


Protože chápeme, že k „uvaření“ řešení jsou často potřeba znalosti základních algoritmů a datových struktur, obvykle též přikládáme do každé série tzv. *kuchařku,* ze které se můžete takové věci naučit. Často je také v zadání úloha, již lze řešit algoritmem z kuchařky. A pozor – další kuchařky najdete na našich webových stránkách.


Dále tímto symbolem označujeme místa, jejichž pochopení může vyžadovat větší zamyšlení, případně nějaké předchozí znalosti.

Chcete-li se na cokoliv zeptat, ať už ohledně semináře, studia na naší fakultě nebo nějakého informatického či programátorského problému, neváhejte a napište nám na diskusní fórum na stránce *http://ksp.mff.cuni.cz/forum/* nebo na naši poštovní adresu:

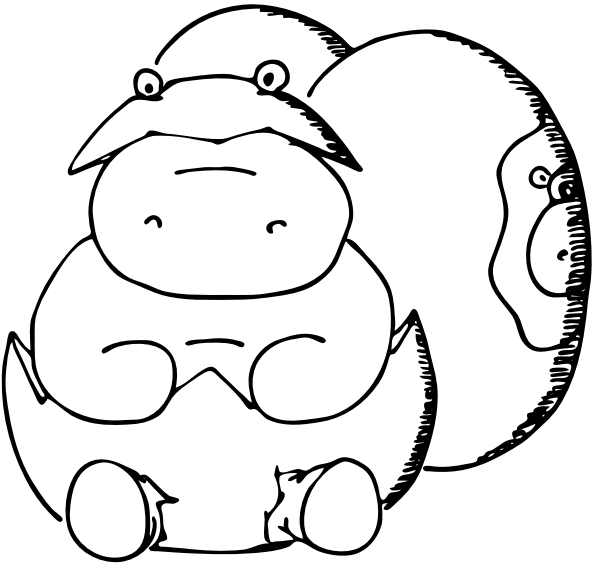
Korespondenční seminář z programování
KAM MFF UK
Malostranské náměstí 25
118 00 Praha 1

e-mail: *ksp@mff.cuni.cz*
www: *http://ksp.mff.cuni.cz/*

Obsah

KSP-Z

Začátečnická kategorie KSP



Zadání úloh KSP-Z

První série

28-Z1-1 Kevinův leták — 8 bodů

Kevin otevřel obálku (kterou našel ve schránce) a vyndal z ní všechny papíry. Jeden ho obzvláště zaujal (ten od KSP). Ihned si všiml, že (na začátku) obsahuje nějak mnoho závorek. Tak by ho zajímalo, jestli tam (autoři zadání) neudělali nějakou chybu. Pomůžete mu (s ověřením)?

Pro danou posloupnost symbolů `(` a `)`, tedy otevíracích a zavíracích závorek, najděte od začátku co nejdelší úsek, který je platným uzávorkováním – tedy každá závorka má svoji do páru.

Tvar vstupu: Na prvním řádku dostanete číslo N. Na dalších N řádcích budou jednotlivé testovací případy.

Tvar výstupu: Pro každý testovací případ vypište na řádek délku nalezeného uzávorkování.

Slibujeme, že N bude nejvýše 1 000, a každý řádek bude mít nejvýše 10^5 symbolů.

Ukázkový vstup:	*Ukázkový výstup:*

`3`
`(()())()()`
`(((()())()))()`
`(()()))()()()()()`
	`8`
	`12`
	`6`

Uzavírající závorka na deváté pozici nemá svůj protějšek. Druhý řádek je celý platné uzávorkování.

28-Z1-2 Sářina hra — 10 bodů

Znáte to, když jedete autobusem, cesta občas ubíhá strašlivě pomalu. Potom vymýšlíte možné i nemožné, abyste se zabavili. Při jedné takové cestě Sáru napadla hříčka.

Pořídila si seznam N zastávek po cestě. Seznam si očíslovala tak, že zastávka, na které nastoupila, dostala číslo jedna, a ta, na které vystoupila, číslo N. Pokaždé, když autobus přijel do zastávky k, si Sára zakroužkovala každou k-tou zastávku v seznamu – tedy zastávky číslo $k, 2k, 3k, \ldots$ Pokud už nějaká zastávka zakroužkovaná byla, zakroužkování zase vygumovala.

Uznejte sami, není to úplně zábavná hra. Také to Sáru před M-tou zastávkou přestalo bavit. Co ale zastávka, na které vystupuje? Je zakroužkovaná?

Tvar vstupu a výstupu: Na prvním řádku je číslo J udávající počet úloh, které budete řešit, těch bude nejvýše 1 000. Na dalších J řádcích najdete mezerou oddělená čísla N a M, kde $1 < M \leq N < 10^9$. Pro každý řádek si zahrajte

Sářinu hru, a vypište na výstup `ANO`, pokud byla N-tá zastávka na konci hry zakroužkovaná, jinak `NE`.

Ukázkový vstup:	*Ukázkový výstup:*
3	ANO
9 4	NE
6 4	NE
7 7	

Devátou zastávku Sára zakroužkovala po projetí třetí, zato šestou po třetí zastávce vygumovala. Ve třetím řádku ji kroužkování přestalo bavit těsně před tím, než by sedmičku zakroužkovala poprvé.

28-Z1-3 Petrovy stromy — 10 bodů

Petr raději při cestě autobusem kouká z okna a počítá. Co počítá? Srnky, posedy, stromy… Počkat, stromy?

Podél cesty je vysázeno několik druhů stromů. Následují po sobě pravidelně, jako neustále se opakující vzorec. Dub, topol, dub, topol, dva duby a buk. A tak pořád dokola. Potom, co tohle chvíli sleduje, se mu začínají dělat mžitky před očima.

Petr by rád věděl, za jak dlouho uvidí K-tý strom daného druhu. Protože ale nezná rychlost autobusu, stačí mu vědět, kolikátý strom v řadě to bude. Vypočítáte to pro něj?

Tvar vstupu: Na prvním řádku budou čísla B, J a K oddělená mezerou. B je počet stromů v bloku, který se opakuje, a J je číslo druhu stromu, jehož K-tý výskyt Petra zajímá. Máte slíbeno, že B bude nejvýše 10^4, J nejvýše 1 000 a K nebude větší než 10^8.

Na dalším řádku bude B čísel $1 \ldots 1\,000$ označujících druhy stromů, které se podél cesty opakují.

Tvar výstupu: Vypište jediné číslo L takové, že K-tý výskyt druhu J je L-tý v řadě všech stromů. Předpokládejte přitom, že řada stromů je dostatečně dlouhá, a daný strom se podél cesty vyskytuje.

Ukázkový vstup:

```
7 2 5
1 2 1 2 1 1 3
```

Ukázkový výstup:

```
16
```

Příklad odpovídá ukázce z textu. Pátý topol (druh číslo 2), který Petr uvidí, bude šestnáctý v řadě.

28-Z1-4 Zuzčina zvědavost — 12 bodů

Malá Zuzka nedávno oslavila šesté narozeniny, takže v září nastupuje do první třídy. Už byla u zápisu, stejně jako všichni z jejího ročníku. Každý věděl, s kým bude chodit do třídy, a s každým svým spolužákem se při zápisu seznámil. S nikým jiným se nikdo neseznamoval.

Zuzka s Kevinem chvíli pozorovali seznamování ostatních. Potom ale Zuzka položila zvídavý dotaz: „Kolik vlastně bude prvních tříd?“ Ještě, že si Kevin zapsal několik dvojic, které viděl se seznamovat. Pomozte mu ze zapsaných dat vykoukat odpověď na Zuzčinu otázku.

Tvar vstupu: První řádek obsahuje čísla N a M, kde N je počet různých dětí, které Kevin viděl, a M je počet seznamujících se dvojic, které si zapsal. Víte, že $1 < N < 1\,000$ a žádná dvojice není v seznamu dvakrát. Následuje seznam M dvojic čísel oddělených mezerou. Každé dítě je pro jednoduchost označeno číslem $1 \ldots N$.

Tvar výstupu: Vypište jedno číslo odpovídající největšímu možnému počtu tříd, které škola podle Kevinova pozorování může otevírat. Každá třída má alespoň jednoho žáka.

Ukázkový vstup:

```
5 3
1 2
2 3
4 5
```

Ukázkový výstup:

```
2
```

Jednu třídu tvoří žáci 1, 2 a 3, druhou pak 4 a 5.

28-Z1-5 Dvě fronty na oběd — 12 bodů

Když už měla Zuzka dost pozorování, šla s ostatními na oběd. V jídelně mají takové zvláštní pravidlo – fronta žáků čekajících na oběd musí být neustále seřazena podle výšky, aby kuchařky viděly všem najednou do obličeje.

Cestou na oběd se tedy skupina, ve které Zuzka šla, správně seřadila. Jenže možné cesty na oběd jsou dvě – u jídelny se potkali s jinou skupinkou. Ta byla přibližně stejně veliká, a také už byla správně seřazena. Jak se mají obě skupinky spojit do výsledné fronty tak, aby byli žáci seřazeni?

Protože jsou všichni hladoví, zkuste vymyslet postup, který vyžaduje co nejméně porovnání výšek – to je totiž to jediné, co s výškami můžete dělat.

Formálněji, máme dvě setříděná pole a chceme z nich sestrojit jedno obsahující všechny položky z obou, také správně setříděné.

Zajímá nás *asymptotická složitost* vašeho řešení v nejhorším případě. To znamená řádový počet operací pro nejhorší možný případ, vyjádřený pomocí proměnných. Například si můžete označit proměnnými A a B velikosti skupin a říct, že k seřazení potřebujete řádově $(A+B)^2$ operací. Podrobněji o složitosti se dočtete v naší kuchařce, kterou vřele doporučujeme přečíst od začátku – nejprve základní algoritmy,[3] poté složitost.[4]

Aby úloha nebyla zbytečně těžká, mají žáci k řazení dostatek prostoru. Tedy například můžete ze dvou front stavět jednu výslednou někde úplně jinde.

28-Z1-6 Osm čipů — 14 bodů

Zatímco Zuzka se mohla jít najíst, Kevin pomáhal s přípravou na prvňáčky. Jedním z úkolů, který řešil, bylo přidělení čipů na přístup do školy. Takový čip má zvolené unikátní číslo a jeho přečtení čtečkou otevře dveře.

Kevin si vymyslel speciální způsob přidělování čísel. Na začátku si zvolí tři libovolná celá čísla a, b a c. Následně vždy spočítá číslo nového čipu x_i z posledních třech čísel vzorcem $x_i = ax_{i-3} + bx_{i-2} + cx_{i-1}$. Čísla prvních tří čipů si prostě vymyslí libovolná, podle toho, jakou má náladu.

Pak si ale vzpomněl, že čip má v sobě omezenou paměť, takže nepojme libovolně velké číslo. Místo původního čísla tedy zapíše do čipu jeho zbytek po vydělení N, kde N je nějaké rozumně velké číslo.

Pak si ale uvědomil, že tento postup nezaručuje, že každý čip bude mít čísla různá od ostatních. Už kvůli tomu, že může existovat nejvýše N čipů s různým číslem, takže se jednou musí čísla začít opakovat.

Ze zvědavosti si Kevin vymyslel číslo K, které je řádově větší než N, a ptá se vás – jaké číslo by měl K-tý čip, který Kevin takto vytvoří? Vymyslete způsob, jak spočítat x_K co nejrychleji.

Předpokládejte, že máte počítač s dostatkem paměti. Naopak nepředpokládejte, že K je rozumně malé. Kevinův počítač umí s tak velkými čísly provádět základní operace, ale například nestihne do K napočítat po jedné.

Příklad: Zvolme si parametry: $a = 0$, $b = 1$, $c = 1$, $N = 3$ a $K = 52$. Prvním čipům dejme čísla 1, 1 a 2. Kevin bude postupně přidělovat čísla takto:

$$1, 1, 2, 0, 2, 2, 1, 0, \quad 1, 1, 2, 0, 2, 2, 1, 0, \quad 1, 1, 2, \ldots$$

Padesátý druhý čip dostane číslo 0.

[3] `http://ksp.mff.cuni.cz/viz/kucharky/zakladni-algoritmy`

[4] `http://ksp.mff.cuni.cz/viz/kucharky/slozitost`

Druhá série

28-Z2-1 Před muzeem — 8 bodů

„Hurá, jdeme na exkurzi!“ znělo třídou stojící před hlavním vchodem leteckého muzea. Učitelky obcházely v řadě stojící žáky a vybíraly z nich skupinky, které půjdou pohromadě.

Papírek se jmény, který dostala Kevinova učitelka, se ale cestou rozmočil a byla čitelná jen první písmenka křestních jmen. Učitelka si sice pamatovala, že vybraní žáci byli zapsaní v abecedním pořadí podle příjmení, ale nepamatovala si, kdo to přesně byl.

Seřadila si tedy své žáky podle abecedy a teď hledá alespoň nějakou skupinku, u které by souhlasila první písmena křestních jmen i pořadí (druhý na papíře musí stát v řadě až za prvním a tak dále).

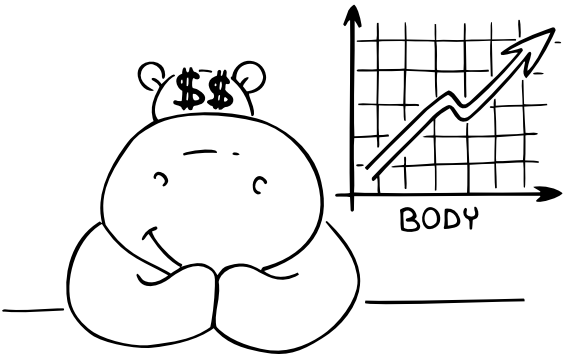
Tvar vstupu: Na vstupu dostanete na prvním řádku dvě čísla: počet žáků ve skupince a celkový počet žáků ve třídě. Na druhém řádku pak první písmena křestních jmen žáků ve skupince a na třetím první písmena křestních jmen všech žáků (tak, jak stojí za sebou v řadě).

Tvar výstupu: Vypište mezerou oddělené pořadí vybraných žáků (žáky číslujeme od jedničky). Pokud takových výběrů existuje více, vypište libovolný z nich.

Ukázkový vstup:	*Ukázkový výstup:*

`3 8`
`JAJ`
`AJJBAJAJ`
	`3 5 8`

Poznámka: Výběr `3 5 6` nebo třeba `2 5 6` fungují taky, `2 1 3` ne, protože žáci nestojí v tomto pořadí za sebou.



28-Z2-2 Práce pro Sáru — 10 bodů

V muzeu se ke Kevinovi připletla Sára a chtěla si s ním povídat. Kevin fascinovaný pohledem na motor z letadla si ale povídat nechtěl, a tak dal Sáře úkol.

Napsal jí na papír číslo a řekl jí, aby prováděla následující operace, než se dostane k jedničce:

- Pokud je číslo sudé, vyděl ho dvěma.
- Je-li liché, vynásob číslo třemi a přičti jedničku.

Doufal, že Sáru na chviličku zabavil, ale než se stihl otočit zpátky k motoru z letadla, tak už Sára jásala, že dosáhla jedničky. Kevin asi zvolil nějaké jednoduché číslo, chtěl by jí teď zadat nějaké těžší. Ideálně takové číslo z nějakého intervalu, pro které to bude trvat nejvíce kroků.

Tvar vstupu: První řádek vstupu bude obsahovat číslo K. Na každém z následujících K řádků bude jeden interval, který Kevina zajímá. Interval je zadaný jako dvojice čísel oddělených mezerou, přičemž krajní hodnoty do intervalu patří. Slibujeme, že spodní mez je vždy alespoň dva.

Tvar výstupu: Pro každý interval na vstupu vypište na samostatný řádek největší počet kroků a také první číslo z intervalu, pro něž se tohoto počtu kroků dosahuje.

Ukázkový vstup:

```
2
10 20
23 44
```

Ukázkový výstup:

```
20 18
111 27
```

28-Z2-3 Byli jsme tři — 10 bodů

Po obědě v bufetu leteckého muzea se Kevin, Sára a Petr posadili společně k jednomu stolu. Petr začal přemýšlet: „To je super, že se my tři tak dobře známe. Kolik si myslíte, že je ve škole ještě takových trojic?“

Sáře se najednou rozsvítily oči a vytáhla svůj sešit. Exponáty v muzeu ji tolik nezajímaly, tak si celý den zapisovala, kdo se s kým baví. Jejím tajným snem totiž bylo stát se špiónkou.

Trojice kamarádů se tedy podívala na to, co si Sára zapsala. Měla poznamenané všechny lidi ve škole, kteří se dobře znají, a my bychom chtěli spočítat počet trojic kamarádů, jako třeba Kevina, Sáru a Petra (zkrátka trojic, kde každý zná dobře každého).

Tvar vstupu: Na prvním řádku vstupu naleznete dvě čísla N a K oddělená mezerou – celkový počet žáků ve škole a počet dobrých přátelství, o kterých

víme. Na dalších K řádcích je pak vždy dvojice čísel popisující, kdo s kým se přátelí (dvojice čísel oddělených mezerou, číslujeme od 0 do $N-1$).

Platí, že $1 < N, K \leq 10^5$.

Tvar výstupu: Do výstupního souboru vypište jediné číslo a to počet různých trojic kamarádů.

Ukázkový vstup:

```
10 5
0 1
0 2
1 3
1 2
3 2
```

Ukázkový výstup:

```
2
```

Poznámka: Existují právě dvě trojice, a to `012` a `123`.

28-Z2-4 Rozsypaná turbína — 12 bodů

„Podíve... ach né!“ zakopl Kevin, když chtěl Sáře ukázat zajímavě vyskládané lopatky z turbíny proudového letadla. Lopatky se rozlétly do všech stran a zazvonily po podlaze haly. A kde se vzal, tu se vzal, stál nad Kevinem velký, zlý a tajemný hlídač.

Položil mu ruku na rameno se slovy: „Tak to si poskládáte, mladý muži. Doufám, že máte dostatek času.“

Každá lopatka z turbíny má na sobě dvě písmena (vrchní a spodní) a platí, že dvě lopatky mohou být za sebou jen tehdy, pokud vrchní písmeno na první je stejné jako spodní písmeno na druhé. Víme, že lopatky jdou určitě poskládat zpátky tak, aby na sebe všechny dokola navazovaly, pomozte je Kevinovi poskládat.

Tvar vstupu: Na prvním řádku dostanete počet lopatek. Na dalších řádcích je vždy dvojice písmen popisujících jednu lopatku (horní a dolní písmeno oddělené mezerou).

Tvar výstupu: Vypište lopatky v pořadí, v jakém se mají poskládat.

Ukázkový vstup:

```
6
A A
C B
A C
B A
B D
D B
```

Ukázkový výstup:

```
C B
B D
D B
B A
A A
A C
```

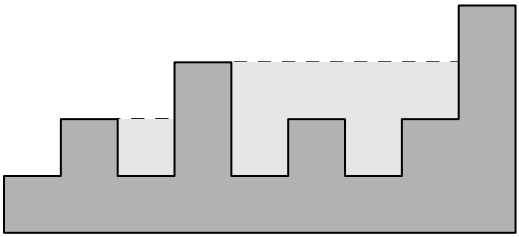
K této úloze jsme na Dnu otevřených dveří MFF UK ukázali drobnou nápovědu. Abychom byli spravedliví i k těm, kteří na DODu nebyli, můžete se podívat na záznam přednášky.[5]

28-Z2-5 Příkop u Tří soutěsek — 12 bodů

Byl už večer, když konečně Kevina pustili z muzea. Zrovna pršelo. Jak tak čekal na zastávce autobusu, sledoval příkop vedle silnice postupně se plnící vodou.

Příkop byl v různých částech různě hluboký a Kevina fascinovalo to, jak se voda různě přelévá a vytéká pryč. Když voda dotekla na nějaký okraj příkopu, zmizela v propustku pod silnicí, takže se držela jen v prohloubeninách.

Kevin se ještě chvíli díval a pak popadl tužku a papír s tím, že spočítá, kolik vody se může v příkopu zadržet. Předpokládejte, že máme podélný řez příkopem jako čtvercovou síť s přesností na decimetry (viz obrázek níže) a že příkop je široký také třeba decimetr. V takovém případě je objem vody v příkopu níže 7 litrů. Vymyslete algoritmus, který toto spočítá pro každý příkop.



28-Z2-6 Kalamita — 14 bodů

Jak nastoupil Kevin do autobusu, změnil se déšť ve sníh a během pár chvil celé město za okny autobusu pohltila sněhová vánice. Kevin se sice ještě dostal domů, ale dopravní podnik už začal plánovat, jaké zastávky bude muset přestat obsluhovat.

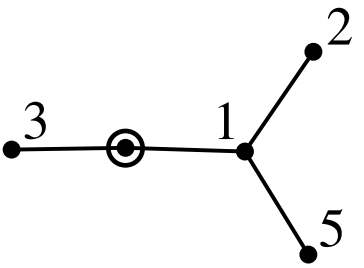
Dopravní podnik provozuje ve městě síť autobusů ve tvaru stromu (pokud neznáte stromy, podívejte se do základní kuchařky),[6] které se všechny rozbíhají z centrálního přestupního stanoviště u vlakového nádraží. Pro každou zastávku zná dopravní podnik počet cestujících, kteří zde za den nastupují. Pokud zrušíme obsluhu nějaké zastávky, tak tím současně zrušíme i obsluhu všech zastávek za touto zastávkou (směrem dál od centrálního přestupu).

[5] http://ksp.mff.cuni.cz/akce/dod/2015/

[6] http://ksp.mff.cuni.cz/viz/kucharky/zakladni-algoritmy

Dopravní podnik by potřeboval pořadí zastávek, v jakém je má postupně rušit, aby vždy odřízl od světa co nejméně cestujících.

Například pro síť na obrázku níže (čísla značí počet cestujících, kroužek centrální zastávku) se vyplatí postupně rušit zastávky se 2, 3, 5 a 1 cestujícím.



Třetí série

28-Z3-1 Místo oslavy — 8 bodů

Střední škola, na kterou chodí Kevin, slaví tento rok významné výročí: celých 500 let od svého založení. Její vedení proto plánuje uspořádat náležitě velkou oslavu, spojenou s výstavou o historii školy a ukázkami prací studentů.

Než se pustí do dalších příprav, musí se vyřešit jedna důležitá věc. Samotná škola totiž nemá žádné velké prostory, ve kterých by mohla takhle velkou akci uspořádat. Proto si chce někde ve městě pronajmout sál. Ale kde? Ne všechny obyvatele budou oslavy zajímat, a tak by bylo pěkné najít nějaké místo blízké těm, kteří určitě přijdou.

Město tvoří jedna dlouhá rovná ulice, okolo které se nacházejí domy. Pro určení pozice ve městě tedy stačí jediné číslo – vzdálenost od začátku města. Máme k dispozici pozice domů, kde bydlí lidé zajímající se o školní oslavu. Naším úkolem je najít takové místo, které má co nejmenší vzdálenost k tomu, kdo to má nejdále.

Tvar vstupu a výstupu: Na prvním řádku je číslo N, na druhém je N čísel, každé představuje dům zájemce, resp. jeho pozici ve městě. Vypište celé číslo označující nejlepší možnou pozici sálu. Pokud je možných odpovědí více, vypište libovolnou z nich.

Ukázkový vstup:	*Ukázkový výstup:*
`5` `3 5 17 24 25`	`14`

28-Z3-2 Zlomkovník — 10 bodů

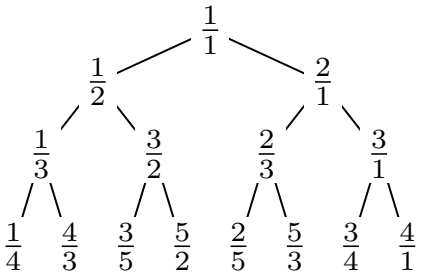
Po nalezení vhodného místa se přípravy rozjely naplno. Kevina nechávalo výročí své školy dlouho chladným (a rýpavě se ostatních ptal, zda 512 let není významnějším výročím než 500), ale pak jeho třída dostala zajímavý úkol. Měli vyrobit nějaké umělecké dílo připomínající Kevinův nejoblíbenější předmět – matematiku.

Jak by taková věc ale mohla vypadat? Kevin dlouho nemohl na nic přijít, ale pak si vzpomněl, jak se Zuzkou po Vánocích odstrojovali stromeček. Co si znovu pořídit strom, ale tentokrát pořádný, matematicky popsatelný? Po chvíli hledání na internetu skutečně našel web, kde prodávali pěkné modely binárních stromů.[7]

Takové dílo by bylo příliš strohé, a proto si Kevin řekl, že každý vrchol stromu označí nějakým zlomkem. Ale ne jen tak ledabyle. Pokud k jakémukoliv

[7] Nevíte-li, co je binární strom, můžete se podívat do naší základní kuchařky (`http://ksp.mff.cuni.cz/viz/kucharky/zakladni`).

vrcholu připíšeme zlomek $\frac{A}{B}$, kde A a B jsou přirozená čísla, v jeho levém synovi bude $\frac{A}{A+B}$ a pravém synovi $\frac{A+B}{B}$. Do kořene stromu zapíšeme $\frac{1}{1}$. Část takového zlomkovníku (jak jinak nazvete strom se zlomky?) bude vypadat takto:



Kevin si nakreslil ještě několik dalších úrovní a došlo mu, že se tímto postupem může dostat k jakémukoliv zlomku v základním tvaru. Jak to ale co nejrychleji provést?

Na vstupu dostanete řádek se dvěma přirozenými čísly A a B. Najděte nejkratší cestu z kořene zlomkovníku do vrcholu odpovídajícímu zlomku $\frac{A}{B}$. Výstup tvoří posloupnost znaků `R` a `L`, kde `R` je přechod do pravého a `L` do levého syna.

Ukázkový vstup:	*Ukázkový výstup:*
`2 5`	`RLL`

28-Z3-3 Posloupnost za trest — 10 bodů

Ajaj! prolétlo Kevinovi hlavou, když uviděl balík se stromem, který nechal poslat na adresu školy. Mělo mu dojít, že ten internetový obchod sídlící na druhém konci planety možná používá jiné jednotky. Očekával výšku okolo dvou metrů, ale tenhle kolos se snad nevejde ani do tělocvičny...

Stěhování obřího zlomkovníku na místo výstavy zabralo půl dne a vystřídal se na něm celý personál školy. Když se ukázalo, že strom je přece jen o něco nižší než výška stropu, všichni si oddechli. A hned poté požádali Kevinova učitele matematiky, aby dal svému nepozornému studentovi nějaký exemplární trest.

Teď je Kevin po škole a zabývá se touto posloupností čísel:

$$1, 11, 21, 1211, 111221, \ldots$$

Jak jsme na ni přišli? Začali jsme jednou jedničkou. Pak jsme ji přečetli: „jedna jednička“, takže píšeme jedničku a jedničku. To jsme opět přečetli: „dvě jedničky“, tedy 21. Pak „jedna dvojka, jedna jednička“, čili 1211. A tak dále.

Kevinovým úkolem je zapsat N-tý člen posloupnosti. Ale členy rostou rychle a kdyby byl poctivý, zůstal by ve škole až do večera. Chce si proto pomoci malým

podvodem: vypočte jen několik prvních číslic N-tého členu a bude doufat, že matikář delší část kontrolovat nebude.

Pro zadané N a K spočítejte prvních K číslic N-tého členu. Slibujeme, že $N, K \leq 300\,000$.

Ukázkový vstup:

```
8 10
```

Ukázkový výstup:

```
1113213211
```

28-Z3-4 Zbývající úkoly — 12 bodů

Korejské přísloví říká, že při odložení práce o jeden den uplyne ve skutečnosti dní deset. Kvůli stěhování zlomkovníku a podobným komplikacím čas ubýval a ubýval a s přibližujícím se dnem oslavy zůstávalo mnoho důležitých úkolů nesplněných. Ředitel proto svolal učitele a zástupce žáků k poradě do školní auly.

Když tam Kevin dorazil, schůze už začala a vládl tam zmatek. Učitelé na tabuli vypsali seznam úkolů, které bude třeba před oslavami splnit, a jejich časovou náročnost. Teď se ale nemohli shodnout na tom, zda je vůbec možné všechno stihnout. Problémem je mimo jiné to, že některé úkoly mohou být splněny až po dokončení jiných (např. není možné, aby se instalovala výstava o historii školy, pokud nejsou sepsané texty).

Tvar vstupu: Na prvním řádku dostanete celkový počet úkolů $N \leq 100\,000$ a počet závislostí K. Na dalším řádku je N čísel, kolik hodin bude splnění jakého úkolu trvat. Na dalších K řádcích se vyskytují dvojice čísel A a B: úkol s číslem B je možné začít až po dokončení úkolu A. Číslujeme od nuly.

Na úkolu může pracovat jen jeden člověk, ale více úkolů lze zpracovávat paralelně (předpokládejte, že dobrovolníků máme k dispozici nekonečně mnoho). Práce na úkolu začne v momentě, kdy všechny úkoly, na kterých závisí, jsou splněné.

Tvar výstupu: Vypište číslo označující počet hodin potřebných k dokončení všech úkolů.

Ukázkový vstup:

```
4 4
5 6 8 5
0 1
0 2
1 3
2 3
```

Ukázkový výstup:

```
18
```

Nejprve je třeba splnit úkol 0, na 1 a 2 se může pracovat současně, teprve po dokončení obou můžeme začít 3.

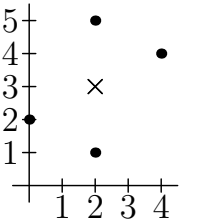
28-Z3-5 Ukotvení stromu — 12 bodů

Oslava výročí se povedla. Zvláště Kevinův obří zlomkovník vzbudil takovou pozornost, že se na výstavu došlo podívat takřka celé město. Proto se ředitel školy a jeho zástupci jednomyslně shodli na tom, že umělecká díla by se měla vystavit trvale na pozemku před školou.

Nejvíce práce samozřejmě zabralo převezení zlomkovníku. A protože se čekalo, že na novém místě nějaký čas setrvá, bylo nutné jej pořádně ukotvit. To se dělá tak, že se okolo stromu zatluče několik kolíků a od každého vedeme lano až ke kmeni, kde se připevní.

Aby celá konstrukce byla skutečně stabilní, chceme kolíky rozdělit do dvojic, kde oba body a kmen stromu leží v jedné přímce, vzdálenost obou bodů od kmenu je stejná a kmen se nachází mezi kolíky. Můžeme tedy říci, že všechny kotvící body jsou středově souměrné podle bodu označujícího kmen.

Kevin na místo přišel ve chvíli, kdy pracovníci už zatloukli všechny kolíky a čekali na přivezení stromu. Zapsal si souřadnice jednotlivých kolíků a rád by našel bod, podle kterého jsou všechny středově souměrné – nebo zjistil, že takové místo neexistuje.



Např. pro body $[4,4]$, $[2,5]$, $[0,2]$ a $[2,1]$ je správný střed $[2,3]$, viz obrázek.

28-Z3-6 Šíření drbů — 14 bodů

„Víš, co jsem zjistila? To tvoje veledílo se stejně bude muset brzy přesunout, protože město chce na pozemku dělat archeologické vykopávky!“ řekla Kevinovi Sára. „Vážně? A to jsi zjistila kde?“ zajímal se Kevin. „Od jedné kamarádky,“ odpověděla vyhýbavě.

Sářiny kamarádky ze třídy tvoří nerozlučnou partu a jejich společnou zálibou je drbání o všem možném i nemožném. Každá spolužačka přijde ráno do školy se svým jedinečným drbem. O přestávce se každá z nich zapovídá s kamarádkou dle svého výběru a v těchto dvojicích si povyměňují všechny drby, které v daný okamžik znají.

Znáte celkový počet spolužaček v partě, N. Pro jednoduchost předpokládejte, že N je mocnina dvojky (1, 2, 4, 8, 16, ...). Navrhněte, o které přestávce se má setkat kdo s kým tak, aby se co nejrychleji všechny kamarádky dozvěděly všechny drby.

Např. pro čtyři spolužačky A, B, C, D si mohou o první přestávce vyměnit drby dvojice A–B a C–D, o druhé pak A–C, B–D. Takto bude po dvou přestávkách znát každá všechny drby.

Čtvrtá série

28-Z4-1 Půdorys — 8 bodů

Když Kevin v minulé sérii hledal ideální místo pro zlomkovník, pobíhal s GPS po školním dvoře, aby si zapsal souřadnice kolíků. Tato aktivita se mu natolik zalíbila, že se rozhodl každý den chvíli sbírat data a večer je zanést do mapy.

Jenže ouha – na soustředění neměl připojení k internetu a po návratu už si nepamatuje, kudy chodil. Nyní sedí nad záznamy z GPS a přemýšlí, které z nich jsou obdélníkové domy a které například kruhové rybníky.

V zadání dostanete několik záznamů – seznamů bodů. Pro každý rozhodněte, jestli leží na nějakém společném obdélníku, který má hrany rovnoběžné se souřadnými osami. Jinými slovy hledáte pouze obdélníky, které nejsou „šikmo“.

Tvar vstupu a výstupu: Na prvním řádku dostanete počet záznamů T. Následuje $2T$ řádků, dva pro každý záznam. Na prvním z nich bude číslo K označující počet bodů v jednom měření. Na druhém řádku leží $2K$ čísel oddělených mezerou, po dvojicích souřadnice x a y jednoho z bodů. Body na řádku nemají žádné speciální pořadí.

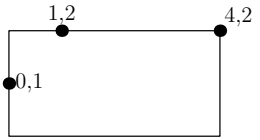
Výstup bude obsahovat pouze T řádků, na každém slovo `ANO`, pokud body leží na osovém obdélníku, `NE` pokud neleží.

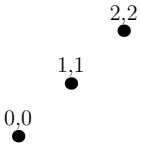
Ukázkový vstup:

```
2
3
0 1 4 2 1 2
3
2 2 1 1 0 0
```

Ukázkový výstup:

```
ANO
NE
```





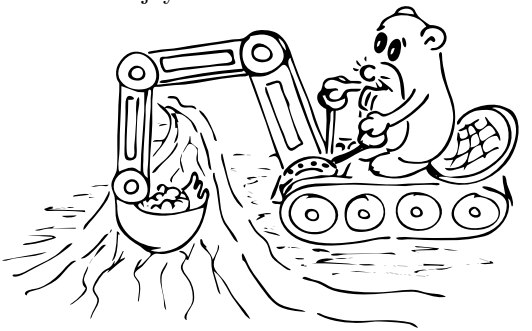
28-Z4-2 Vykopávky — 10 bodů

Mezitím započaly archeologické práce na místě, kde tak pracně zlomkovník umístili. Takové archelologické vykopávky nejsou jen tak nějaké kopání do země, ty mají pevný řád. Ptáte se jaký?

Vždy jsou potřeba čtyři archeologové. Určí si obdélníkovou oblast, rozdělí ji na čtvercová políčka a každý začne kopat v jednom z rohů oblasti. Z rohu pak

rozšiřuje svůj výkop (také ve tvaru obdélníku) políčko po políčku směrem do bodu S. V tomto bodě se mají všichni potkat.

Bod S určují archeologové jen tak od oka ze zkušenosti. Kevin se s tím ale nespokojí. Během práce kopáče pozoruje a zapisuje si, jak dlouho trvá práce na každém políčku. Až práce skončí, chce najít ideální bod S – takový, ke kterému by všichni kopáči došli ve stejný čas.



Tvar vstupu a výstupu: Na prvním řádku jsou čísla R a S, označující počet řádků a sloupečků. Na každém z dalších R řádků je S nezáporných celých čísel, každé říká, kolik hodin bude práce na tomto políčku trvat. Mezi čísly jsou mezery.

Hledáte rozdělení této tabulky čísel na čtyři obdélníkové části, které mají stejný součet. Výstupem budou dvě čísla – výška a šířka levé horní oblasti. Slibujeme, že to vždy půjde, a že tabulka bude mít nejvýše 200 000 prvků.

Ukázkový vstup:

```
4 4
6 4 2 8
5 1 2 1
1 1 1 3
1 1 2 1
```

Ukázkový výstup:

```
1 2
```

6	4	2	8
5	1	2	1
1	1	1	3
1	1	2	1

Součet čísel v každé části je 10.

28-Z4-3 Mocniny — 10 bodů

Od koukání z okna na dvůr Kevina vytrhla až hodina matematiky. Čísla, ta má rád. Zrovna si jich musí docela hodně zapamatovat, probírají totiž mocniny. Ke každému číslu až do dvaceti musí umět, kolik je jeho druhá a třetí mocnina.

Kevinova učitelka matematiky prosazuje alternativní metody zkoušení. V písemce zadá hromadu čísel a po žácích požaduje, aby mezi nimi našli číslo x, pro

které lze v hromadě najít i x^2 a x^3. Jen tak totiž pozná, kdo čísla umí zpaměti a kdo počítá.

Kevin je ale programátor, tak si na to napíše program. A když už, tak se neomezí jen na čísla od nuly do dvaceti.

Tvar vstupu a výstupu: Na prvním řádku je číslo N. Na dalším pak mezerou oddělených N nezáporných celých čísel. Všechna čísla se vejdou do 32 bitů.

Na výstup vypište nejmenší takové x, které lze mezi čísly najít. Součástí vstupu bude vždy alespoň jedno.

Ukázkový vstup:	*Ukázkový výstup:*
`8` `64 2 100 10 7 16 4 256`	`4`

28-Z4-4 Čtyřková — 12 bodů

U matematiky zůstaneme. Sára totiž přišla se zajímavým hlavolamem: zapsat číslo 42 jen pomocí sčítání, odčítání, násobení a dělení, ale především pouze pomocí čtyřek. A to ještě co nejkratším výrazem!

Operace ale fungují tak trochu podivně. Zaprvé, neplatí zde přednost násobení a dělení před sčítáním a odčítáním. Všechny operace se vyhodnocují striktně zleva doprava.

Zadruhé, po každé operaci se z mezivýsledku vezmou jen čtyři poslední číslice – Sářina kalkulačka neumí počítat s čísly nad 10 000. Pokud by náhodou vyšlo číslo záporné, přičítá se k němu 10 000. (Jinak řečeno, všechny operace počítají modulo 10 000.) Nakonec, dělit čtyřmi lze pouze tehdy, když výsledek bude celočíselný.

Kevinovi netrvalo dlouho uvědomit si, že popsaným způsobem lze vyrobit nejen číslo 42, ale i třeba $9\,997 = 4/4 - 4$. Ale co ostatní čísla?

Tvar vstupu a výstupu: Na vstupu je pouze jedno číslo x. Výstupem je také jedno číslo – počet čtyřek v nejkratším výrazu, který po vyhodnocení dává x.

Ukázkový vstup:	*Ukázkový výstup:*
`42`	`9`

Nejkratší výraz je `4 + 4 + 4 * 4 - 4 * 4 - 4 - 4 / 4`, používá 9 čtyřek.

28-Z4-5 Vyhlazení GPS — 12 bodů

Vraťme se k GPS. Pokud jste ji někdy měli v ruce, asi jste si všimli, že není úplně přesná. Když se podíváte na záznam bodů ze zařízení, hodnoty „skáčou“. Proto se používají různé techniky vyhlazení.

Jeden z možných způsobů je klouzavý průměr. Na vstupu budete dostávat potenciálně nekonečnou posloupnost čísel, reprezentujících například nadmořskou výšku. Na každé číslo byste měli odpovědět aritmetickým průměrem z posledních K obdržených hodnot, kde K je dopředu pevně dané číslo. Čím vyšší K, tím hladší bude průběh, ale o to pomaleji bude GPS reagovat.

Protože je posloupnost nekonečná, nedává dobrý smysl bavit se o časové složitosti programu. Zkuste vymyslet řešení, které bude mít co nejlepší časovou složitost jednoho kroku (tj. přijetí jednoho čísla a vypsání jednoho průměru).

Nemusíte se zabývat počátečními podmínkami, rozumné odpovědi se očekávají až po K-tém čísle na vstupu.

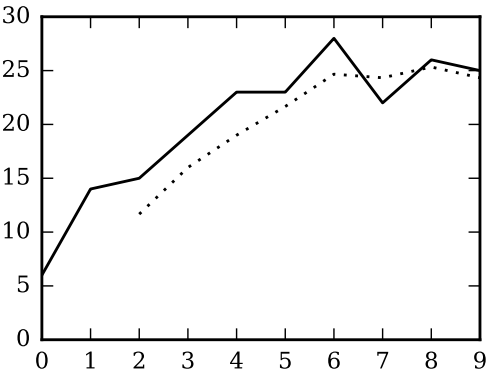
Například pro $K = 3$ a vstupní posloupnost

$$6 \quad 14 \quad 15 \quad 19 \quad 23 \quad 23 \quad 28 \quad 22 \quad 26 \quad 25 \quad \ldots$$

vypisujte postupně

$$? \quad ? \quad 11,7 \quad 16,0 \quad 19,0 \quad 21,7 \quad 24,7 \quad 24,3 \quad 25,3 \quad 24,3 \quad \ldots$$

Efekt vyhlazování je vidět na obrázku (plná čára představuje původní data, tečkovaná vyhlazená):



28-Z4-6 Klučičí drby — 14 bodů

V minulé sérii Kevin zkoumal chování kamarádek, které si vyprávěly drby. Zatímco holky mezi sebou drby šíří jako poplach, kluci se k drbům chovají trochu jinak.

Každý kluk má svého nejlepšího kamaráda. Takový je právě jeden, ale vztah „nejlepší kamarád“ nemusí být vzájemný. Řekněme, že na začátku má každý kluk nějaký skvělý drb, o kterém nikdo jiný neví. O první přestávce ho řekne svému nejlepšímu kamarádovi.

Zadání úloh KSP-Z – 4. série

V každé další přestávce pak kamarádovi řekne všechny nové drby, které slyšel přestávku minulou. Pokud nějaký kluk už slyší drb podruhé, řekne jen „To je starý!“ a nechá ho být, dále ho neposílá, aby se neztrapnil.

Kevin nad tímto mechanismem přemýšlel a za chvíli si uvědomil, že po pár přestávkách každý drb u někoho skončí. Dokážete pro každý drb spočítat, kolik kluků se ho dozví?

Na vstupu dostanete seznam kluků ve třídě a ke každému jeho nejlepšího kamaráda. Každý kluk má svého nejlepšího kamaráda ve třídě.

Vzorová řešení KSP-Z
První série

řešení

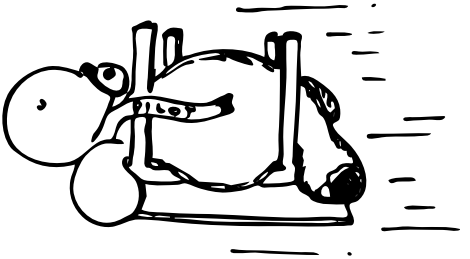
28-Z1-1 Kevinův leták

Nejprve si vyřešme jednodušší úlohu: jak o nějaké posloupnosti závorek poznáme, jestli je (celá) správně uzávorkovaná? Určitě musí obsahovat stejný počet levých a pravých závorek – jinak těžko mohou tvořit páry.

Ovšem to nestačí. Uvažte třeba posloupnost `())(`. Ta obsahuje dvě levé a dvě pravé závorky, ale korektním uzávorkováním určitě není. Na třetí pozici se pokoušíme ukončit závorku „dřív, než začala“.

Jak takovou situaci poznáme? Podíváme-li se na první tři znaky naší posloupnosti, vidíme, že obsahují jednu levou závorku a dvě pravé. Obecně problém nastane, pokud existuje v posloupnosti nějaké místo takové, že v úseku od začátku po toto místo je více pravých závorek než levých. Pak alespoň jedna z těch pravých nemá před sebou žádnou levou, se kterou bychom ji mohli spárovat.

Pokud posloupnost obsahuje stejně levých a pravých závorek a nenastane problém popsaný výše, pak už si snadno rozmyslíte, že je vždy jde správně spárovat, a tedy uzávorkování je korektní.



Nyní k původní úloze. Představme si, že si chceme označit všechna místa v posloupnosti, kde může končit korektní uzávorkování (začínající na začátku). Třeba pro příklad ze zadání to budou následující místa (označena hvězdičkou): `(()())*()*()`.

Jak je najdeme? Budeme postupně procházet posloupnost od začátku a průběžně si počítat, kolik levých a pravých závorek už jsme viděli. Pokud narazíme na místo, kde jsme napočítali víc pravých než levých, můžeme rovnou skončit. Už víme, že takové uzávorkování korektní není, a přidáním libovolných dalších závorek na konec už nemůžeme napravit chybějící závorky nalevo od aktuální pozice.

Pokud tato situace ještě nenastala a objevíme místo, před kterým je počet levých a pravých závorek stejný, označíme si jej. Podle toho, co jsme si řekli výše, uzávorkování končící na tomto místě je korektní.

Ukažme si průběh algoritmu na předchozím příkladu:

	(	(	)	(	)	)	(	)	)	(	)
Pozice	1	2	3	4	5	6	7	8	9	10	11
Počet levých	1	2	2	3	3	3	4	4	4	–	–
Počet pravých	0	0	1	1	2	3	3	4	5	–	–
Rozdíl	1	2	1	2	1	0	1	0	−1	–	–
Akce						∗		∗	`X`		

Hvězdička znamená označení místa s korektním uzávorkováním a `X` konec prohledávání, když žádné další uzávorkování korektní být nemůže.

Na konci jen vypíšeme nejpravější označené místo. Všimněme si, že si nemusíme ukládat všechna označená místa, stačí si vždy pamatovat poslední dosud nalezené. Další drobné zjednodušení je místo dvou počítadel použít jen jedno, uchovávající rozdíl počtu dosud načtených levých a pravých závorek (řádek „rozdíl“ v tabulce výše). Pak označujeme, když je toto počítadlo rovné nule, a končíme, pokud klesne pod nulu.

Program (Python 3) – počítání obou druhů závorek:
`http://ksp.mff.cuni.cz/viz/28-Z1-1-1.py`

Program (Python 3) – stačí nám počítat jen levé:
`http://ksp.mff.cuni.cz/viz/28-Z1-1-2.py`

Filip Štědronský

28-Z1-2 Sářina hra

Na vstupu dostaneme čísla N a M a máme zjistit, jestli po hraní $M-2$ kol hry bude zastávka číslo N zakroužkovaná (kol je $M-2$, protože Sára začne kroužkovat až ve druhé zastávce – v prvním kole kroužkuje zastávky dělitelné 2 a v posledním zastávky dělitelné $M-1$).

Pomalé řešení

Nejjednodušší řešení by bylo vyrobit si pole `zakrouzkovano` s N příznaky, kde příznak `zakrouzkovano[i]` značí, je-li i-tá zastávka zakroužkovaná. V tomhle poli $(M-2)$-krát nasimulujeme kolo Sářiny hry a nakonec přečteme příznak u N-té zastávky, čímž zjistíme, jestli skončila zakroužkovaná, nebo ne.

V Pythonu to jde napsat třeba takhle:

```
# N + 1, protože Python čísluje pole od nuly
zakrouzkovano = [False for _ in range(N + 1)]
for kolo in range(2, M):
  # Zakroužkuj každou k-tou zastávku.
  for zas in range(kolo, N + 1, kolo):
    zakrouzkovano[zas] = not zakrouzkovano[zas]
print("ANO" if zakrouzkovano[N] else "NE")
```

Jak dlouho to poběží? Určitě aspoň $\Omega(N)$ – první kolo Sářiny hry zakroužkuje aspoň $\lfloor N/2 \rfloor$ zastávek, kterým potřebujeme upravit příznaky.[8]

Kol je ale víc: první kolo potřebuje $N/2$ kroků, druhé $N/3$, třetí $N/4$, a tak dále. Kol je celkem M a každé potřebuje nejvýš N kroků, proto nebude naše řešení potřebovat více než $\mathcal{O}(N \cdot M)$ kroků. Pomalé řešení tedy poběží něco mezi $\Omega(N)$ a $\mathcal{O}(N \cdot M)$. (Přesná časová složitost je $\mathcal{O}(N \log M)$, ale na ní teď příliš nezáleží. Chcete-li zjistit, kde se vzalo $N \log M$, podívejte se do sekce o Eratosthenově sítě v kuchařce o teorii čísel.)[9]

Tímhle zvládneme první dva malé vstupy, ale větší nestíháme spočítat.

Druhý pokus

Zakroužkování žádné zastávky kromě N-té nás ale vlastně vůbec nezajímá, výsledek programu na nich nezáleží. Udržování příznaků, jestli jsou zastávky $2, \ldots N-1$ zakroužkované, je plýtvání časem a pamětí. Místo toho tedy projdeme kola $2, \ldots, M-1$ a pro každé z nich se podíváme, jestli změní zakroužkovanost N-té zastávky. Kroužkování každé k-té zastávky změní zakroužkovanost zastávky N právě tehdy, když je N dělitelné beze zbytku k, neboli (v Pythonu) `N % k == 0`.

```
# Je N-tá zastávka zakroužkovaná?
zakrouzkovana = False
for kolo in range(2, M):
  if N \% kolo == 0:
    zakrouzkovana = not zakrouzkovana
print("ANO" if zakrouzkovano[N] else "NE")
```

Tohle řešení poběží v čase $\mathcal{O}(M)$, a to je určitě zlepšení proti prvnímu pokusu. Na plný počet bodů ale ještě nestačí.

Řešení za všechny body

Každá změna zakroužkovanosti N-té zastávky odpovídá tomu, že číslo kola k dělí N. Zastávka N je na konci zakroužkovaná, pokud počet změn zakroužkovanosti byl lichý, jinak zakroužkovaná není. Úloha se nás tedy vlastně ptá, jestli má N sudé, nebo liché množství dělitelů, kteří jsou menší než M a velcí aspoň 2.

K dalšímu zrychlení si všimneme, že dělitelé se vyskytují v párech: ke každému k_1, které je dělitelem N, existuje nějaké k_2 takové, že $N = k_1 \cdot k_2$. Tohle k_2 je taky dělitelem N. Dále v každém páru (k_1, k_2) je aspoň jedno k menší nebo rovné $\sqrt{N}$: kdyby byly obě větší, tak $k_1 \cdot k_2$ by muselo být víc než N, ale dohodli jsme se, že to bude přesně N.

[8] Ω je příbuzná známějšího $\mathcal{O}$. Když řekneme, že nějaká funkce g je v $\mathcal{O}(f(x))$, znamená to, že se stoupajícím x neroste rychleji, než nějaký násobek $f(x)$, čili „g se dá shora odhadnout f“. Ω oproti tomu značí dolní odhady: když g je v $\Omega(f(x))$, tak g roste se stoupajícím x aspoň tak rychle, jako f.

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

Když nás tedy zajímají všichni různí dělitelé N, můžeme je generovat tak, že najdeme všechna k_1 mezi 1 a $\sqrt{N}$ a ke každému dopočítáme k_2. Pro každého z dělitelů můžeme zjistit, je-li menší než M a aspoň 2, a jestli je, přepneme za něj zakroužkovanost N-té zastávky. Zbývá doladit ještě jeden detail: když N je k_1^2, tak $k_2 = k_1$. V takovém případě musíme přepnout zaškrtnutost N-té zastávky jenom jednou. Tentokrát řešení doběhne za $\mathcal{O}(\sqrt{N})$, a to už si zaslouží 10 bodů.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-Z1-2.py`

Michal „Prvák“ Pokorný

28-Z1-3 Petrovy stromy

Zadání sice mohlo vypadat děsivě kvůli množství proměnných, které se v něm vyskytují, ve skutečnosti je ale prosté. Podél cesty se nám pravidelně střídají stromy a nás zajímá, kolik stromů se stihne prostřídat, než uvidíme K-tý strom konkrétního druhu. V řešení budeme mluvit o topolech, ale myslíme tím prostě strom druhu J.

Připomeňme ještě označení B pro délku bloku, který se nám pravidelně opakuje.

Přímočaré řešení je uložit si celý opakující se blok do pole. Následně můžeme toto pole procházet a pamatovat si, kolik topolů a kolik stromů celkem jsme už viděli. Až potkáme K-tý topol, zahlásíme výsledek.

Takové řešení má ale složitost až $\mathcal{O}(BK)$, to když bude v celém bloku jeden jediný topol. Jelikož vstup má velikost $\mathcal{O}(B)$ a K může být mnohem větší než B, můžeme pojmout podezření, že to musí jít rychleji.

Jde, a rychlejší řešení není nijak těžké, pojďme na něj. Nejprve uvidíme nějaký počet opakujících se bloků, pak ještě část stromů z dalšího bloku a pak ten očekávaný K-tý topol.

Kolik bude bloků, které uvidíme celé? Tolik, aby se do nich spolehlivě vešlo $K-1$ topolů, takže $(K/S)-1$, kde lomítko představuje celočíselné dělení (zbytek zahodíme) a S je počet topolů v bloku.

A kolik stromů uvidíme z dalšího bloku? Nejprve jiná otázka: kolik topolů ještě uvidíme? Tolik, kolik jich chybí do K. To většinou bude $K \bmod S$, kde „mod“ označuje zbytek po dělení. Ono slovo „většinou“ je v předochozí větě pro případ, kdy K je násobkem S, pak nám topolů chybí nikoliv 0, ale S.

Dejme tomu, že nám tedy do K chybí ještě T topolů. To ale znamená, že uvidíme tolik stromů, na kolikáté pozici je v bloku T-tý topol. Řečeno příkladem, pokud jsou třeba topoly na pozicích 2, 4 a 9 a chybí nám 2 topoly, uvidíme ještě 4 stromy, protože druhý topol je na pozici 4.

Zpočátku samozřejmě neznáme ani S, ani pozice jednotlivých topolů. Obojí ale můžeme jednoduše zjistit při načítání vstupu. Pamatujeme si, kolikátý strom

právě načítáme, a taky si udržujeme informaci o počtu už načtených topolů. Když máme na vstupu topol, zvýšíme počet topolů a zároveň si uložíme jeho pozici do pole.

Na konci pak jen využijeme tyto informace k dosazení a máme výsledek. Zbývá poslední věc, a to je složitost řešení. Musíme načíst vstup, to zvládneme v $\mathcal{O}(B)$ (i když načítáme topol, provádíme jen konstantně mnoho operací). Závěrečné spočtení výsledku stihneme v konstantním čase, máme tedy celkovou časovou složitost $\mathcal{O}(B)$. Paměťová složitost je $\mathcal{O}(S)$, protože si musíme pamatovat pozice pro jednotlivé topoly.

Program (C):

`http://ksp.mff.cuni.cz/viz/28-Z1-3.c`

Karolína „Karryanna“ Burešová

28-Z1-4 Zuzčina zvědavost

Zadání trochu krkolomně popisovalo graf, ve kterém hrany tvoří dvojice žáků, kteří se seznamovali. V tomto grafu jste měli najít počet komponent souvislosti. Pokud předchozí větě rozumíte, asi se zde nic nového nedozvíte. Jinak velmi doporučuji přečíst si naši kuchařku o grafech.[10]

Pokud se grafových zvířátek zatím bojíte, zkusím popsat řešení v jazyce zadání. Nejprve si celé zadání načteme do paměti. Nejlépe tak, že si pro každého žáka pořídíme seznam ostatních žáků, se kterými

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

se seznámil. Nezapomeňte, že pro každou dvojici musíte přidat dva záznamy – lidé se seznamují vzájemně.

Nyní budeme postupně procházet všechny žáky a pokud někoho ještě nemáme zařazeného v žádné třídě, vytvoříme pro něj novou (zvýšíme počítadlo tříd). Potom budeme procházet žáky, se kterými se seznámil, a u všech si označíme, že už jsme je zařadili. Tuhle operaci musíme provádět *rekurzivně*, tj. označit si i sousedy sousedů, sousedy sousedů sousedů a tak dále.

Prakticky si pořídíme funkci `označ(Ž)`, která označí žáka Ž, a zavolá sama sebe na všechny spolužáky Ž. Volání sebe sama se říká rekurze. Důležité ale je, aby někdy přestala – pokud už je žák označený, funkce ihned skončí a nic nedělá.

Popsanému algoritmu se říká prohledávání do hloubky, více o něm a jeho aplikacích najdete v kuchařkách.

Nejprve si pojďme uvědomit, kolik potřebujeme paměti. Pro uložení seznamů spolužáků potřebujeme N seznamů velkých nejvýše N – ale budou mít dohromady $2M$ prvků. Protože ale nevíme, které z čísel je větší, řekneme, že na tuto část potřebujeme $\mathcal{O}(N+M)$ paměti. Při dodržení podmínek ze zadání bude $M < N^2$, ale může být i řádově menší.

Na uložení příznaků označení nám stačí N proměnných, pak už potřebujeme jen pár počítadel. Dohromady se tedy vejdeme do $\mathcal{O}(N + M)$ paměti.

Časová složitost vyjde stejně. Při procházení si stačí uvědomit, že po každé dvojici projdeme jen dvakrát (jednou z každé strany), a jakmile třídu uzavřeme, už se na žádného žáka z ní nepodíváme. Třída s jedním žákem je ale také třída a prohledávání, které hned skončí, musíme také započítat. Všechna prohledávání dohromady spotřebují $\mathcal{O}(N + M)$ času – $\mathcal{O}(N)$ za žaky, $\mathcal{O}(M)$ za dvojice.

Dodejme jen, že v některých jazycích může být volání funkce zbytečně náročná operace a může chtít příliš mnoho paměti. Potom se hodí rekurzi nahradit cyklem a žáky si ukládat do zásobníku „ručně“. Co je to zásobník a jak ho použít se dočtete ve výše zmíněné kuchařce.

Pokud místo zásobníku použijeme frontu, dostaneme prohledávání do šířky, které používá vzorové řešení v Pythonu. Jinak jsou obě možnosti ekvivalentní.

Program (C++):
`http://ksp.mff.cuni.cz/viz/28-Z1-4.cpp`

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/28-Z1-4.py`

Ondra Hlavatý

KSP-Z

28-Z1-5 Dvě fronty na oběd

řešení

Jsme v jídelně a máme tu dvě fronty hladových studentů seřazených podle výšky. Potřebujeme, aby se spojili v jednu frontu (také seřazenou). Jak se ale mají žáci co nejrychleji srovnat?

Někteří z vás možná poznali, že se vlastně jedná o „slévací“ funkci `merge` z algoritmu MergeSort, o kterém si můžete více přečíst v naší kuchařce o třídění.[11] A pokud je to pro vás nové, pojďme si ukázat, jak na to.

První žák v nové frontě musí být nejmenší ze všech. To znamená, že to bude buď první žák první fronty, nebo první žák druhé fronty. Tito dva jsou nejmenší ve své skupince (která je seřazená), tudíž určitě nikdo za nimi není menší.

Porovnáme jejich výšku a menšího pošleme do nové fronty. Možná by vás mohlo napadnout poslat do nové fronty oba, ale to udělat nemůžeme. Žák druhý v pořadí může být klidně menší než první žák z jiné fronty. Posíláme tedy vždy jen jednoho.

U jedné skupinky se tak druhý nejmenší stal nejmenším a toho opět porovnáme s nejmenším z druhé fronty. Pokud by porovnávaní žáci byli stejně vysocí, můžeme poslat libovolného, nebo dokonce oba dva (ani v jedné frontě není nikdo menší než oni).

Vždy tedy porovnáme dva aktuálně nejmenší žáky a menšího z nich pošleme na konec nové fronty. Když už bude jedna fronta prázdná, můžeme zbytek té druhé poslat do nové fronty tak, jak je (žáci jsou už seřazení).

Než vám prozradíme časovou a paměťovou složitost, je tu jeden implementační detail. Poslat žáka na konec nové fronty znamená, že informaci pouze překopírujeme. „Odebraného“ žáka nemažeme, jenom se ve frontě podíváme na dalšího. Mazat data (a tím všechna ostatní posouvat) by bylo zbytečné a neefektivní.

Každým porovnáním vybereme jednoho žáka, kterého pak pošleme na konec nové fronty. Pokud bylo v první skupince A žáků a ve druhé B, potom bude v nové frontě $A + B$ žáků. Takže provedeme maximálně $A + B - 1$ porovnání, méně to bude, pokud po vyprázdnění jedné fronty zbude ve druhé více než jeden žák či pokud jsou porovnávaní žáci stejně vysocí a my je pošleme do fronty oba. Ale nás zajímá nejhorší případ, takže časová složitost bude $\mathcal{O}(A + B)$.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/28-Z1-5.py`

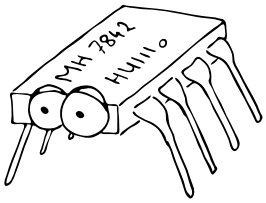
Katka Zákravská & Zuzka Drázdová

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

28-Z1-6 Osm čipů

Kevin si zvolí tři čísla: a, b a c. Když máme spočítat číslo k uložení do i-tého čipu, použijeme vzorec $a \cdot x_{i-3} + b \cdot x_{i-2} + c \cdot x_{i-1}$ a spočítáme, kolik vyjde po vydělení výsledku číslem N[12] Dostaneme zadaná čísla prvních tří čipů x_1, x_2 a x_3 a nějaké hodně velké K a snažíme se co nejrychleji spočítat x_K.

Nejjednodušší by bylo vyrobit si pole o K prvcích, ve kterém budeme držet hodnotu každého čipu. První tři hodnoty do něj uložíme rovnou a zbytek dopočítáme.



Takové řešení potřebuje $\mathcal{O}(K)$ paměti na uložení pole a bude trvat čas $\mathcal{O}(K)$. My ale vlastně nepotřebujeme znát všech K čísel ve všech čipech: zajímá nás jenom to poslední.

A když počítáme číslo K-tého čipu, stačí nám k tomu čísla čipů $(K-1)$, $(K-2)$ a $(K-3)$. Podobně když počítáme číslo v čipu $(K-1)$, potřebujeme znát jenom čísla čipů $(K-2)$, $(K-3)$ a $(K-4)$.

Obecně si stačí pamatovat čísla jenom tři kroky dozadu – díky tomu naše spotřeba paměti bude konstantní, nezávisle na K.[13] Proměnné, ve kterých budeme držet poslední tři čísla čipů, si označíme jako x, y a z. Můžeme třeba ve for-cyklu $(K-1)$-krát upravit hodnoty proměnných (x, y, z) na $(y, z, (a \cdot x + b \cdot y + c \cdot z) \bmod N)$.

```
# V x, y, z si teď budeme držet čísla
# posledních 3 čipů.
x, y, z = X1, X2, X3
for i in range(K - 1):
    x, y, z = y, z, (a * x + b * y + c * z) % N
print(x)
```

V zadání bylo napsáno, že do K nestihnete napočítat po jedné – to znamená, že cyklus provádějící K kroků bude příliš pomalý. Ukážeme si dva způsoby, které jsou rychlejší.

Když se pozorně podíváte na zadání, všimnete si, že se čísla po nějaké době opakují. V zadání je předěl mezi opakováními zvýrazněn větší mezerou. Pokud bychom znali délku opakujícího se úseku p, můžeme skoro s jistotou říci, že K-tý čip má stejné číslo, jako $(K \bmod p)$-tý. Číslo $K \bmod p$ bude určitě menší než p, takže bychom to stihnout mohli.

[12] Obecně se operaci „spočítat zbytek po dělení“ říká „modulo“ a často se dá v programovacích jazycích provést operátorem `%`.

[13] Program navíc v praxi nejspíš i poběží rychleji: přístupy do paměti jsou rychlejší, když zapisujeme a čteme pořád do stejného místa. Zatímco první program popíše K různých míst v paměti, druhý program jenom tři.

Nejprve učiníme pozorování: přestože zadání neslibovalo, jak velká budou x_1, x_2 a x_3, můžeme místo nich vzít rovnou jejich zbytek po dělení N. Pokud $a \bmod N = j$, pak existuje i takové, že $a = i \cdot N + j$. Potom $(a \cdot x) \bmod N = (i \cdot N \cdot x) \bmod N + (j \cdot x) \bmod N = 0 + (j \cdot x) \bmod N$. Cokoliv, co násobíme N, bude N dělitelné, tedy bude mít zbytek po dělení N nula.

Další důležitá myšlenka plyne už z odstavce výše. K určení čísla i-tého čipu nám stačí čísla třech čipů dozadu. Pokud je tedy trojice čísel před čipem i stejná jako před čipem j, musí být x_j stejné číslo jako x_i. A díky tomu se rovnají i x_{i+1} s x_{j+1}. A tak dále. Jakmile tedy najdeme stejnou trojici podruhé, nutně už musíme být v opakujícím se bloku, v *periodě*.

Hledání periody trochu komplikuje to, že posloupnost čísel může mít předperiodu – opakující se úsek nemusí začínat na začátku posloupnosti.

Například pro $(a, b, c) = (0, 1, 1)$, $N = 2$ posloupnost může vypadat takto:

$$1,\ 1, 1, 0,\ 1, 1, 0,\ 1, 1, 0, \ldots$$

Všimněte si, že kromě prvních se tři jedničky za sebou v posloupnosti už neobjeví. Jak se tedy s předperiodou vypořádat? Pořídíme si trojrozměrné pole čísel, v každém rozměru velké N. Do něj si budeme označovat trojice po sobě jdoucích čísel čipů a číslo kroku i, ve kterém jsme danou trojici potkali. Jakmile narazíme na stejnou trojici podruhé v kroku j, našli jsme periodu dlouhou $j - i$ s předperiodou délky i.

Když známe délku periody p a předperiody q, číslo K-tého čipu spočítáme hloupějším algoritmem výše, který spustíme pro $K' = q + (K - q) \bmod p$.

Zbývá si uvědomit, že K', tedy ani počet kroků výpočtu předperiody, nemůže přesáhnout N^3 už jen díky tomu, že více různých trojic se nemůže objevit. To zároveň slouží jako důkaz, že perioda se v posloupnosti vždy objeví.

Algoritmus, který najde periodu tedy poběží v čase $\mathcal{O}(N^3)$ a spotřebuje $\mathcal{O}(N^3)$ paměti. Může se to zdát hodně, ale při rozumně velkém N, které slibovalo zadání, je to velké zrychlení oproti $\mathcal{O}(K)$. Takové řešení mohlo být za plný počet bodů.

Program (C):

`http://ksp.mff.cuni.cz/viz/28-Z1-6.c`

Stále to jde ale rychleji.

Upozorňujeme čtenáře, že následující text obsahuje zvýšené množství matematiky. Doporučujeme!

Podívejme se, jak se hodnoty proměnných x, y a z změní během prvního kroku (staré hodnoty označíme indexem 1 a nové indexem 2):

$$x_2 = y_1,\ y_2 = z_1,\ z_2 = (a \cdot x_1 + b \cdot y_1 + c \cdot z_1) \bmod N$$

Rozepíšeme si tyto rovnice do tvaru, ve kterém výsledné proměnné jsou vážený součet vstupních:

$$\begin{aligned} x_2 &= (0 \cdot x_1 + 1 \cdot y_1 + 0 \cdot z_1) \bmod N \\ y_2 &= (0 \cdot x_1 + 0 \cdot y_1 + 1 \cdot z_1) \bmod N \\ z_2 &= (a \cdot x_1 + b \cdot y_1 + c \cdot z_1) \bmod N \end{aligned}$$

Tyto lineární rovnice, které z (x_1, y_1, z_1) počítají (x_2, y_2, z_2), se dají taky vyjádřit jako maticové násobení těchto vektorů zleva. Pokud vám matice a vektory nejsou známé, můžete si pro naše účely představit vektory jako skupinky čísel, se kterými budeme pracovat najednou (třeba (x_1, y_1, z_1)).

Matice jsou jenom zkratka za postup tohoto typu: z jednoho vektoru udělám jiný vektor, a složky nového vektoru jsou nějaké vážené součty složek prvního, kde váhy jsou konstanty. Právě tyhle konstanty v pořadí jako ve vzorečcích výše tvoří matici. Náš případ se dá maticově zapsat jako takovéto násobení:

$$\begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = \begin{pmatrix} (0 \cdot x_1 + 1 \cdot y_1 + 0 \cdot z_1) \bmod N \\ (0 \cdot x_1 + 0 \cdot y_1 + 1 \cdot z_1) \bmod N \\ (a \cdot x_1 + b \cdot y_1 + c \cdot z_1) \bmod N \end{pmatrix} =$$

$$= \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ a & b & c \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \bmod N$$

Tato matice se ještě bude hodit, tak si ji označíme:

$$M = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ a & b & c \end{pmatrix}$$

Když potom z (x_2, y_2, z_2) počítáme (x_3, y_3, z_3), provádíme stejné operace: $x_3 = y_2$, $y_3 = z_2$, $z_3 = (a \cdot y_2 + b \cdot y_2 + c \cdot z_2)$, neboli zapsáno maticově:

$$\begin{pmatrix} x_3 \\ y_3 \\ z_3 \end{pmatrix} = \left(M \cdot \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} \right) \bmod N =$$

$$= \left(M \cdot \left(M \cdot \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \right) \right) \bmod N$$

A podobně to vypadá i dál: posunutí proměnných x, y, z o jedno místo znamená jejich vynásobení zleva maticí M.

Obecně pro hodnoty x, y, z po K krocích platí:

$$\begin{pmatrix} x_K \\ y_K \\ z_K \end{pmatrix} = M^{i-1} \cdot \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix}$$

Co to ale vlastně znamená násobit matice? Násobení matic je trochu divná operace. Je vymyšlené tak, aby bylo asociativní, tedy aby pro libovolné čtvercové matice A a B a každý vektor x platilo $A \cdot (B \cdot x) = (A \cdot B) \cdot x$. Matice $(A \cdot B)$ tedy reprezentují stejnou operaci, jako nejdřív vektor „prohnat“ maticí B a pak maticí A.

Na vektorech o dvou dimenzích a maticích velkých 2×2 si z toho snadno můžeme odvodit, jak se musí násobit. Mějme pár matic a libovolný vektor:

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix},\ B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix},\ x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

Potom:

$$B \cdot x = \begin{pmatrix} b_{11} \cdot x_1 + b_{12} \cdot x_2 \\ b_{21} \cdot x_1 + b_{22} \cdot x_2 \end{pmatrix}$$

Po vynásobení A dostaneme:

$$A \cdot (B \cdot x) = \begin{pmatrix} a_{11}(b_{11}x_1 + b_{12}x_2) + a_{12}(b_{21}x_1 + b_{22}x_2) \\ a_{21}(b_{11}x_1 + b_{12}x_2) + a_{22}(b_{21}x_1 + b_{22}x_2) \end{pmatrix} =$$

$$= \begin{pmatrix} x_1(a_{11}b_{11} + a_{12}b_{21}) + x_2(a_{11}b_{12} + a_{12}b_{22}) \\ x_1(a_{21}b_{11} + a_{22}b_{21}) + x_2(a_{21}b_{12} + a_{22}b_{22}) \end{pmatrix}$$

Odsud si můžeme hned přečíst, jak musí matice $A \cdot B$ obecně vypadat. Pro rozměr 3 je koeficientů víc, ale dají se odvodit stejným způsobem.

Teď už víme, co dělá a jak se dělá násobení matic a že je asociativní. Proto $(K-1)$-násobné vynásobení x_1 zleva maticí M se dá napsat jako $M^{K-1} \cdot x_1$ – nezáleží na uzávorkování.

Naše první řešení se dá popsat jako „vezmi x_1, $(K-1)$-krát ho zleva vynásob M a pak vrať x_K“. Tohle znamená i-krát pronásobit třísložkový vektor maticí 3×3 (a promodulit výsledek N).

Rychlejší řešení to obejde tím, že *rychle spočítá* M^K (modulo N) a potom vynásobí x zleva M^K (a zase vymodulí). Bude nám na to stačit $\mathcal{O}(\log K)$ násobení matic 3×3 a jedno násobení vektoru. Násobení matice maticí sice potřebuje více operací než násobení matice vektorem, ale obě operace trvají konstantní čas: jenom násobíme a sčítáme konstantní počet čísel na vstupu.

Díky asociativitě násobení matic se můžeme spolehnout na to, že pro libovolná g a h platí $M^{g+h} = M^g \cdot M^h$. Číslo $(K-1)$ si vyjádříme jako součet mocnin dvojky (neboli ho převedeme do dvojkové soustavy) a příslušně si přeuspořádáme výpočet M^{K-1}. Například pro $K = 44$ tak dostaneme $M^{43} = M^{32} \cdot M^8 \cdot M^2 \cdot M^1$. Takhle si rozložíme M^{K-1} na nejvýše $\lceil \log_2 i \rceil$ násobení matic, které jsou „dvojkové mocniny M“.

Tyhle „dvojkové mocniny M“ budeme efektivně generovat: v prvním kroku budeme držet $M = M^1$. Jejím vynásobením samy se sebou dostaneme M^2, ze které po dalším umocnění dostaneme M^4, a tak dále.

V matici M' si budeme postupně stavět potřebnou matici M^{K-1}. Počáteční hodnota M' bude maticový ekvivalent jedničky. Takové matici se říká jednotková, značí se ve třech rozměrech I_3, a je to ta jediná, která se chová v násobení matic jako jednička v násobení čísel: když je T libovolná matice 3×3, potom $I_3 \cdot T = T \cdot I_3 = T$. Z vlastností násobení se dá uhodnout, jak I_3 vypadá. Má nuly všude kromě diagonály, kde má jedničky:

$$I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Budeme se posouvat v binárním zápisu čísla $(K-1)$ zprava doleva. Na začátku kroku číslo i budeme mít uloženou prozatímní matici M' a matici M^{2^i}. Podíváme se na i-té číslo zprava v binárním zápisu, a jestli tam uvidíme jedničku, přenásobíme M' maticí M^{2^i}.

Ať vidíme jedničku nebo nulu, umocníme M^{2^i} na druhou, čímž si spočítáme pro další krok $M^{2^{i+1}}$. Po každém kroku taky vymodulíme obsah $M^{2^{i+1}}$ i M' číslem N. Tohle modulení musíme provádět, abychom zajistili, že všechny použité proměnné budou omezeně velké, konkrétně mezi 0 a $N-1$. Bez modulení by totiž mohly růst hodně rychle a ukládat velká čísla nestojí konstantně mnoho paměti.

Z výsledku nás zajímá jenom $x_i \bmod N$ a to vyjde stejně ať počítáme přesně, nebo modulo N.[14]

Na konci bude M' součin těch mocnin M, které odpovídají vahám jedniček v binárním zápisu $(K-1)$, tedy bude M' přesně rovna M^K. Za každou číslici v binárním zápisu $(K-1)$ provedeme jedno nebo dvě násobení matic, což trvá konstantní čas, a protože číslic je logaritmicky mnoho, trvá náš algoritmus jenom $\mathcal{O}(\log K)$. Dá se navíc naimplementovat tak, že mu stačí jenom konstantní množství paměti.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-Z1-6.py`

Michal „Prvák“ Pokorný

[14] To platí, protože $(a+b) \bmod N = [(a \bmod N) + (b \bmod N)] \bmod N$ a $(a \cdot b) \bmod N = [(a \bmod N) \cdot (b \bmod N)] \bmod N$. x_i vznikne z (x_0, y_0, z_0) jenom jejich posloupností, proto můžeme všechny mezivýsledky včetně matic ukládat modulo N.

Druhá série

28-Z2-1 Před muzeem

řešení

U úloh tohoto typu, kde je na první pohled spousta různých správných výsledků, se často vyplatí hledat nějaké konkrétní, třeba v nějakém smyslu nejmenší nebo první. Lépe pak vyplyne, že pokud jsme řešení nenašli, tak neexistuje.

Ukážeme si jednoduché řešení, které nám nezabere více času než vůbec přečíst vstup. Pořídíme si dvě ukazovátka a pojmenujeme si je *jehla* a *seno*. *Jehla* bude ukazovat na prvního žáka v hledané skupince (první znak na druhém řádku) a *seno* na prvního žáka v řadě (první znak v řádku třetím).

Nyní budeme hledat jehlu v kupce sena. První žák, kterého vybereme do skupinky, musí začínat písmenem, které nám určuje *jehla*. Pokud do skupinky dáme prvního takového, určitě nic nezkazíme!

Budeme tedy posouvat ukazovátko *seno*, a jakmile se bude znak pod *jehlou* a *senem* shodovat, vypíšeme pozici *sena* na výstup. Pak posuneme jehlu doprava a opakujeme úvahu. Další vybraný žák může být bez újmy na obecnosti ten první, na kterého narazíme. Jakmile vybereme všechny žáky ze skupinky (dojedeme *jehlou* na konec řádky), končíme.

Pokud bychom vyjeli *senem* za konec třetího řádku, víme jistě, že žádná taková skupinka mezi žáky není. To ale zadání zakazovalo a vstupní data tento případ neobsahovala.

Všimněte si, že ukazovátka posouváme jen doprava a přes každé písmenko přejede ukazovátko jen jednou. Algoritmus běží v čase lineárním se součtem čísel na prvním řádku, tedy s velikostí vstupu.

Program je opravdu jednoduchý, proto si v tom Céčkovém dovolíme trochu magie. Zkuste si rozmyslet, co se tam děje.

Program (C):

`http://ksp.mff.cuni.cz/viz/28-Z2-1.c`

Program (Python):

`http://ksp.mff.cuni.cz/viz/28-Z2-1.py`

Ondra Hlavatý

28-Z2-2 Práce pro Sáru

Ještě než si ukážeme samotné správné řešení, zodpovíme otázku, která vám jistě vrtá hlavou: kde se tato jednoduchá hříčka s násobením a dělením čísel vlastně vzala? Ve skutečnosti je to matematický problém známý jako *Collatzova domněnka (Collatz conjecture)*.

Ta říká, že pokud vezmeme jakékoliv přirozené číslo a stále dokola na něj aplikujeme pravidla ze zadání (pokud je sudé, vydělíme jej dvěma; pokud je liché, vynásobíme jej třemi a přičteme jedničku), tak se vždy nakonec dostaneme do jedničky.

Problém byl poprvé formulován v roce 1937 matematikem Lotharem Collatzem a doposud zůstává nevyřešený – tedy nikdo ještě nenašel důkaz, že domněnka platí pro jakékoliv číslo.

Počítačovou simulací se ověřila správnost pro všechny hodnoty menší než cca 2^{60} (v naší úloze jste dostávali daleko menší). To však neznamená, že se protipříklad, tedy nějaké číslo, které do jedničky nedojde, nemůže vyskytovat mezi ještě většími hodnotami.

A jak tedy vyřešit zadanou úlohu? Přímočarým řešením je na každé číslo z intervalu aplikovat pravidla a přitom počítat, kolik kroků potřebujeme pro dosažení jedničky. Následně vypíšeme číslo, které má tento počet nejvyšší. Takový postup stačil na vyřešení menších testovacích vstupů.

Pokud máme zájem o rychlejší řešení, musíme uvažovat, zda nepočítáme něco víckrát, než je nutné. Při počítání kroků se často dostaneme do čísla, v němž jsme se předtím již vyskytli, a postup zbytečně opakujeme. Například pro interval $4 \ldots 8$ je počet kroků pro osmičku roven počtu kroků pro čtyřku plus jedna – toho ale při počítání osmičky využít nemůžeme, jelikož si počty nikde nepamatujeme.

Optimálním řešením je si vytvořit pole, indexované všemi možnými hodnotami, do kterých se při počítání kroků můžeme dostat. Typicky budeme vycházet z maximálního možného čísla na vstupu.

Hodnoty, které nám vyjdou při počítání, ale mohou být ještě větší, takže musíme udělat rozumný odhad. Na indexu I bude zapsán počet kroků, než se z čísla I dostaneme do jedničky, nebo -1, pokud jej zatím neznáme. Na začátku budou všechny hodnoty v poli nastavené na -1.

Pro každé číslo z intervalu pak aplikujeme pravidla jako předtím, ale pamatujeme si hodnoty, přes které jsme prošli. Jakmile narazíme na číslo, jehož počet kroků již známe (a je tedy uložený v poli), vrátíme se přes zapamatované hodnoty zpět až do původního čísla a přitom každému nastavíme správný počet kroků. Zbytek probíhá stejně jako u původního řešení.

Protože si nejsme jisti, že algoritmus skončí, nebudeme ani obecně určovat časovou složitost.

Program (C):
`http://ksp.mff.cuni.cz/viz/28-Z2-2.c`

Kuba Maroušek

KSP-Z

28-Z2-3 Byli jsme tři

řešení

Jak můžeme mezi všemi lidmi ve škole najít ty trojice, které tvoří dobří přátelé? Připomeňme ještě, co dostaneme na vstupu: počet lidí N, počet dvojic lidí, kteří spolu kamarádí K, a následně K dvojic typu (a, b) říkajících, že osoba a a b jsou kamarádi.

Pokud jste už někdy slyšeli o teorii grafů, asi pro vás nebylo těžké si úlohu představit a pochopit správně zadání. Pokud jste o grafech ještě neslyšeli, velmi doporučujeme přečíst si základní kuchařku.[15] Pro pochopení tohoto textu sice není nezbytná, ale může vám pomoci při řešení dalších podobných úloh.

Ale zpět k úloze. Jak ji řešit? Může nás napadnout přímočaré řešení. Vyzkoušíme všechny trojice lidí a ověříme, zda se každá ze tří dvojic zná. Přímou implementací dostaneme sice funkční, ale velmi pomalé řešení. Vyzkoušení všech možných trojic zařídíme snadno pomocí tří vnořených cyklů. Všech možných trojic je řádově $\mathcal{O}(N^3)$. A pro každou z nich projdeme vždy celý vstup, abychom mezi všemi přátelstvími našli tři dvojice lidí (a, b), (b, c) a (c, a).

Musíme si dát pozor, kolikrát trojici započítáme. Na každou totiž narazíme postupně šestkrát, například jako na trojice `012, 021, 102, 120, 201, 210`. Tento problém můžeme vyřešit dvěma způsoby: buď výsledek před vypsáním vydělíme šesti, nebo trojici započítáme pouze v případě, že čísla jednotlivých lidí budou v rostoucím pořadí. Na to nesmíme zapomenout ani v následujících řešeních, v popisu to však již znovu rozebírat nebudeme.

Celková časová složitost popsaného algoritmu je $\mathcal{O}(N^3K)$. Za takovéto řešení ale moc bodů získat nešlo.

Zrychlujeme

Čím trávíme drahocenný čas zbytečně? Hledáním. Řešení výše pořád dokola hledá v celém seznamu přátelství jednotlivé dvojice. Co kdybychom o každé dvojici dokázali říct okamžitě, zda se dotyční přátelí, či nikoli? Rázem bychom dostali řešení s časovou složitostí $\mathcal{O}(N^3)$.

Jak to tedy zařídit? Pořídíme si tabulku $N \times N$ – dvourozměrné pole, kde na pozici (i, j) bude jednička, pokud jsou lidé i a j přátelé, v opačném případě nechápe políčko nulové. V řeči grafů bychom této tabulce říkali *matice sousednosti*.

Asi vám nemusíme složitě popisovat, jak takovou tabulku získat. Na začátku si jednoduše přečteme řádek po řádku celý vstup a vždy si do matice sousednosti zapíšeme na odpovídající pozice jedničku. Jen nezapomeňte, že přátelství jsou vzájemná, takže si chceme zapsat jedničku jak na pozici (i, j), tak zároveň na (j, i).

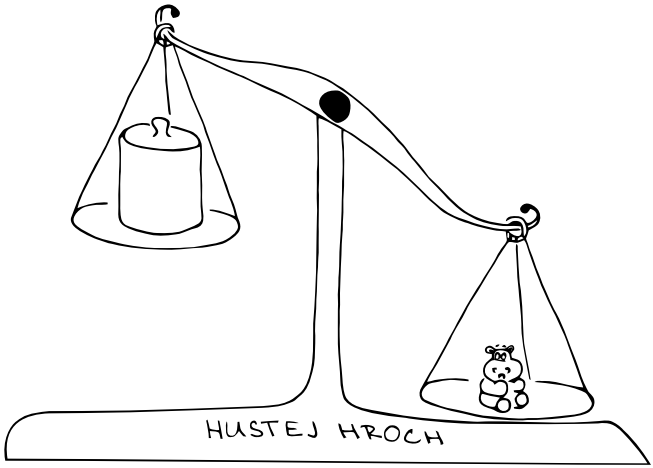
Sestavení tabulky nám zabere čas $\mathcal{O}(N^2 + K)$. Pokud totiž chceme nějakou paměť používat, musíme si ji nejprve připravit. A to zabere řádově tolik času,

[15] `http://ksp.mff.cuni.cz/viz/kucharky/zakladni-algoritmy`

kolik paměti chceme.[16] I tak je však příprava rychlejší, než samotné zkoušení všech trojúhelníků. Časová složitost je tedy $\mathcal{O}(N^3)$ a paměťová $\mathcal{O}(N^2)$.

Je dobré si uvědomit, že pokud by se každý přátelil s každým (v řeči teorie grafů se jedná o *úplný graf*), je celkový počet všech trojic $\frac{N\cdot(N-1)\cdot(N-2)}{6}$, tj. řádově N^3. V obecném případě tedy ani časovou složitost pod $\mathcal{O}(N^3)$ nezlepšíme.[17]

Máme tedy vyhráno? Ještě ne. V zadání byla společná horní hranice pro počet lidí i přátelství: $N, K \leq 10^5$. Rozhodně tedy nemůže nastat případ, kdy $N = 10^5$ a každý zná každého, protože pak by všech přátelství bylo téměř 10^{10}. Zajímáme se zkrátka o případy, ve kterých je řádově stejně lidí jako dvojic přátel. Můžeme tedy hovořit o takzvaném *řídkém grafu* – grafu, který neobsahuje mnoho hran.



Husté řešení pro řídké grafy

Z čeho se taková trojice přátel skládá? Ze tří lidí a tří přátelství. Zatím jsme zkoušeli brát postupně všechny trojice lidí a k nim dohledávali přátelství. Mohli bychom to také celé otočit. Vezmeme-li jedno konkrétní přátelství (řádek vstupu, čili jednu informaci, kterou si Sára zapsala), určuje nám jednoznačně hned dvě konkrétní osoby a a b. K nim už stačí jen vzít všechny přátele b a zjistit, zda jsou také přáteli a; pokud ano, našli jsme trojici.

Technicky zlepšení dosáhneme i jen díky tomu, že budeme existenci přátelství kontrolovat průběžně a ne až pro celou trojici dohromady.

[16] Toto se nemusí vždy projevit, protože paměť za nás může připravit operační systém. Často to dokonce udělá těsně před samotným zápisem do paměti.

[17] Zde předpokládáme, že program musí konkrétní trojúhelníky najít.

Tím dostaneme řešení s celkovou časovou složitostí $\mathcal{O}(K \cdot N)$, pro každý vztah vyzkoušíme N lidí na doplnění trojice. Abychom omezili i paměťovou náročnost, musíme se ještě zbavit matice sousednosti (a nerozbít si u toho složitost časovou).

Pro každého si budeme pamatovat seznam lidí, se kterými se přátelí. Pořídíme si k tomu N-prvkové pole (pojmenujme si je třeba `P`) obsahující spojové seznamy. Následně budeme pole procházet – tím získáme prvního z trojice, a. Procházením seznamu přátel a budeme dostávat b, takto tedy iterujeme přes všechna přátelství.

Nyní stačí projít všechny přátele b a zkontrolovat u nich pouze to, že se také přátelí s a. Ke kontrole přátelství s a jsme dříve používali právě matici sousednosti. Tu teď sice k dispozici nemáme, ale nic nám nebrání si vždy vytvořit jeden její řádek – ten s kamarády a.

Celé řešení proto bude vypadat takto: projdeme postupně naše pole `P` a pro každého člověka uděláme následující tři kroky. Nejprve vytvoříme jemu odpovídající řádek z matice sousednosti (projitím jeho seznamu přátel).

V druhém kroku budeme trojúhelníky počítat, postupně pro každého z již projitého seznamu kamarádů a. A to tak, že ověříme, kteří přátelé b jsou i přáteli a. Za každého společného přičteme jedničku k celkovému počítadlu trojic.

Na závěr projdeme seznam ještě jednou a řádek matice po sobě zase uklidíme (vynulujeme místa s jedničkou), tím si ho připravíme pro dalšího člověka.

Paměťová složitost je tedy $\mathcal{O}(K + N)$ a časová slíbených $\mathcal{O}(K \cdot N)$. A co říci závěrem? Nezapomeňte výsledek vydělit šesti.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-Z2-3.py`

Program (C):

`http://ksp.mff.cuni.cz/viz/28-Z2-3.c`

Jenda Hadrava

28-Z2-4 Rozsypaná turbína

Jako první si všimneme jedné zajímavé vlastnosti. Každé písmeno se objeví na horní straně lopatky právě tolikrát, kolikrát se objeví na spodní straně.

Proč tohle platí? Představme si třeba, že v poskládané turbíně nebudeme koukat na lopatky samotné, ale na jejich spojení – to vždy obsahuje stejné písmenko, jednou nahoře, jednou dole.

Zkusme teď lopatky poskládat. Jako první nás asi napadne nejprve vzít náhodnou lopatku, pod ní připojit libovolnou z těch, které pod ní připojit smíme, a tak dále.

Jestliže už mezi nezapojenými lopatkami není žádná, kterou bychom mohli přidat, musí být možné spojit poslední a první lopatku. To plyne právě z toho,

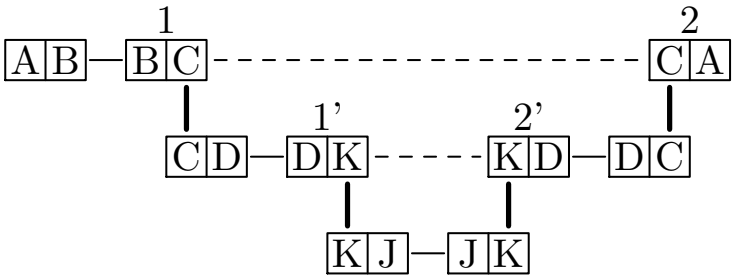
že všechna písmenka se vyskytují stejněkrát nahoře i dole. Pokud má poslední lopatka dole písmeno A a žádná nezapojená lopatka nemá A nahoře, někde nám jedno volné „horní A“ chybí. Na spoji zapojených lopatek to být nemůže, tam jsou písmenka „obsazená“, musí to tedy být na první lopatce.

Skvělé, sestavili jsme turbínu. Ta ale nemusí být stejně velká jako ta původní, klidně nám mohly zbýt nepoužité lopatky. Co s nimi uděláme? Někam je vložíme.

Když se budeme postupně dívat na jednotlivé lopatky hotové turbíny, dříve nebo později potkáme takovou, na kterou bychom (kdyby nebyla obsazená) mohli napojit nějakou z nezapojených lopatek.

Turbínu tedy rozpojíme. Označme si rozpojené lopatky třeba 1 a 2. Teď k lopatce 1 připojíme nezapojenou lopatku a k ní budeme opět přikládat libovolnou lopatku z těch, které přidat smíme (jsou nezapojené a mají stejné písmeno). Podobným argumentem jako prve dojdeme k tomu, že až nebudeme mít co připojit, musí být možné poslední přidanou lopatku spojit s lopatkou 2.

Stejně můžeme pokračovat a zapojit další zbývající lopatky. Jen pozor, při zkoumání lopatek v turbíně musíme pokračovat z lopatky 1, nikoliv 2, může se nám totiž stát něco podobného jako na obrázku:



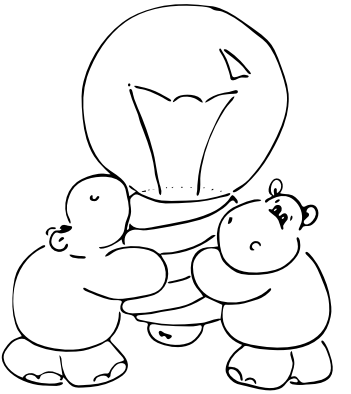
Na začátku máme nalezený cyklus `AB BC CA`. Do něj se nám podaří vložit lopatky `CD DK KD DC`, takže dostáváme `AB BC CD DK KD DC CA`. Ve vkládání pokračujeme od lopatky 1 (`BC`). Přímo k ní už nic dalšího nenajdeme, tak pokračujeme lopatkou `CD`. Turbínu ovšem rozpojíme až za `DK`.

Rozmysleme si, že takto můžeme postupně přidat opravdu všechny lopatky. Kdyby to možné nebylo, musí nám zůstat alespoň jedna lopatka označená písmenkem, které se vyskytuje v hotové turbíně. Jinak by vůbec nebylo možné všechny lopatky spojit, ale my víme, že to jít musí (předtím spojené byly).

Jenže my lopatku v hotové turbíně neopustíme, dokud na ni můžeme něco napojit, tedy bychom naši zbylou lopatku bývali zapojili. Takže nám žádná taková lopatka zbýt nemůže.

Dobře, teď musíme takový postup efektivně naimplementovat. K implementaci můžeme dojít dvěma úvahami, buď do nich zahrneme klasické grafy, nebo ne.

Úvahy bez grafů



Hotovou turbínu budeme reprezentovat jako obousměrný spojový seznam, abychom do ní uměli rychle vkládat nové lopatky.

Horší je to s nezapojenými lopatkami, ty si potřebujeme pamatovat a zároveň v nich chceme umět rychle najít libovolnou vhodnou lopatku, tedy lopatku označenou konkrétním písmenem.

Tady trochu zneužijeme, že písmenek je málo.[18] Pořídíme si pole indexované písmenky (resp. třeba jejich pořadím v abecedě). Prvky tohoto pole budou spojové seznamy lopatek, které mají na horní straně dané písmenko.

Když budeme potřebovat lopatku označenou daným písmenkem, jednoduše vezmeme první z příslušného spojového seznamu.

Jakou bude mít náš algoritmus složitost? Začněme od přidání lopatky do hotového kusu turbíny. To je konstantní, $\mathcal{O}(1)$, protože smazat první prvek ze spojového seznamu i vložit za konkrétní prvek zvládneme v konstantním čase.

Malá zrada přichází, když si rozmyslíme složitost projití celé turbíny. Mohlo by se zdát, že ke každé lopatce můžeme přinejhorším připojit všechny lopatky, tedy při N lopatkách by byla složitost $\mathcal{O}(N^2)$. Ale během celého algoritmu můžeme přidat jen N lopatek, takže celé napojování trvá $\mathcal{O}(N)$.

Načtení vstupu i výpis výstupu nám trvá lineárně, celý algoritmus má tedy časovou složitost $\mathcal{O}(N)$. Lineární je i paměťová složitost, celou dobu máme někde uložené informace o N lopatkách (byť se lopatky přesouvají mezi jednotlivými spojovými seznamy).

Úvahy s grafy

Jestli se grafů nebojíme, můžeme úlohu jednoduše převést na grafový problém. Pokud jste někdy slyšeli o hledání eulerovského tahu, měl by vám ho popsaný postup nápadně připomínat. V opačném případě vřele doporučujeme přečíst si kuchařku o procházkách v grafu.[19]

Jak ale převést lopatky na graf? Vcelku jednoduše, z písmenek uděláme vrcholy, z lopatek pak hrany (za každou lopatku povedeme hranu z písmene na její horní straně do písmene na dolní straně). Jenom pozor, že mezi dvěma vrcholy může vést více hran (můžeme mít dvě stejné lopatky). Grafu s takto násobnými hranami se říká *multigraf*.

[18] Podotkněme, že kdyby jich málo nebylo, můžeme udělat něco podobného, ale bude to technicky komplikovanější.

[19] `http://ksp.mff.cuni.cz/viz/kucharky/prochazky-po-grafech`

Použití všech hran teď odpovídá použití všech lopatek. Průchod přes vrchol pak zajišťuje, že lopatky na sebe budou správně napojené, tj. že první lopatka končí stejným písmenem, jakým začíná druhá lopatka.

Tím, že se písmeno musí vždy vyskytovat stejněkrát na horní i dolní straně, budou mít všechny vrcholy sudý stupeň, eulerovský tah tedy existuje. Stačí nám tak pustit na grafu klasický algoritmus.

Zbývá nám určit složitost. Jak slibuje kuchařka, algoritmus na hledání eulerovského tahu je lineární v počtu vrcholů a hran. Protože písmenek máme „málo“, můžeme jejich počet prohlásit za konstantní, do složitosti se nám tedy promítne jen počet hran, což je vlastně počet lopatek.

Pro úplnost dodejme, že vytvoření grafu zvládneme také v lineárním čase – pro každou lopatku v konstantním čase přidáme do grafu hranu (tj. prvek do spojového seznamu sousedů příslušného písmena). Celý algoritmus bude mít tedy pro N lopatek složitost $\mathcal{O}(N)$.

Poznámka na okraj

Snadno si můžeme rozmyslet, že obě úvahy vedou na velmi podobný program, liší se opravdu jen způsob uvažování a pojmy, které používáme. Kouzlo informatiky je, že v takovýchto případech si každý může vybrat ten přístup, který mu vyhovuje, a přesto máme všichni stejně dobré řešení.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-Z2-4.py`

Martin Španěl & Karolína „Karryanna“ Burešová

28-Z2-5 Příkop u Tří soutěsek

Způsobů, jak vyřešit tuto úlohu, bylo více. My zde ukážeme několik variant řešení, které jsou lineární s délkou příkopu. Předpokládáme, že vstupem je řada čísel, kde každé vyjadřuje výšku příkopu na jednom decimetru délky.

Jedno z možných řešení mohlo vypadat následovně – pro každý úsek (každý decimetr) zjistíme, jakou největší výšku má příkop od něj nalevo a napravo. Voda se na dané části může udržet do menší z nich, protože jinak odteče.

Spočítat to můžeme tak, že projdeme příkop zleva i zprava a při jednom průchodu si pro každou část zapisujeme doposud největší výšku. Nakonec si z hodnot na stejných pozicích zapamatujeme tu nižší.

Podívejme se na příklad:

Vstup:	`1, 3, 2, 5, 4, 1, 3, 2`
Maxima průchodu zleva:	`1, 3, 3, 5, 5, 5, 5, 5`
Maxima průchodu zprava:	`5, 5, 5, 5, 4, 3, 3, 2`
Výsledné max. výšky:	`1, 3, 3, 5, 4, 3, 3, 2`

Výsledek už dostaneme snadno – od čísla na i-té pozici v poli maximálních výšek odečteme výšku i-tého decimetru příkopu a tento rozdíl přičteme do výsledného objemu zadržené vody.

Pro příklad výše bychom postupně sečetli hodnoty `0, 0, 1, 0, 0, 2, 0, 0`, takže se zadrží 3 litry vody.

A proč to celé funguje? V každém sloupečku se udrží právě tolik litrů vody, kolik jsme za něj přičetli, tj. rozdíl mezi výškou hladiny a dnem příkopu. Zbývá tedy ukázat, že v poli maximálních výšek máme opravdu uloženou výšku vodní hladiny na dané části. Výška hladiny nemůže být větší, protože jinak by vodě nic nebránilo v odtečení na stranu s nižším maximem. A zároveň se zde voda udrží do této výšky, protože nalevo i napravo je překážka, která sahá alespoň takto vysoko.

Řekněme, že N je délka příkopu, tj. počet čísel na vstupu. Časová složitost je lineární. Při počítání maximálních výšek projdeme vstup třikrát a každé číslo pouze porovnáme s jedním jiným číslem. Při počítání objemu zadržené vody opět projdeme dvě pole o velikosti N a pro každý prvek provedeme konstantně operací. Paměťová složitost je také lineární, neboť si pamatujeme tři pole stejně velká jako vstup.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-Z2-5-1.py`

Tento postup lze ještě vylepšit. Můžeme si všimnout, že při počítání výšek zleva a zprava se hodnoty po přejití globálního maxima (nejvyššího místa příkopu) už nemění, takže do pole s maximálními výškami nikdy nevybereme hodnotu z levé, resp. pravé části. Pojďme se tedy podívat, jak výpočet trošku zrychlit a jak ušetřit paměť.

Nejprve najdeme místo, kde je příkop nejvyšší, a označíme si ho M, tj. M bude označovat pozici, nikoliv výšku. Také se nám bude hodit pomocná proměnná max, ve které budeme počítat maximum z doposud projitých hodnot výšek.

Algoritmus postupně projde jednotlivé části příkopu zleva, dokud nenarazí na nejvyšší místo M. Poté projde příkop zprava, opět do M. Před druhým průchodem je nutné proměnnou max vynulovat.

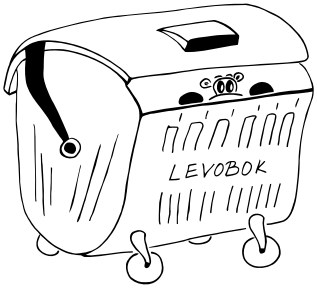
Pro každou část zaktualizuje proměnnou max (je-li aktuální část vyšší než max, uloží do max současnou výšku) a spočítá, kolik litrů vody se na ní drží. To znamená, že do výsledného objemu přičte rozdíl max a aktuální výšky, takže pokud se nacházíme na doposud nejvyšším bodě, všechna voda odsud odteče, ale pokud ne, nějaká voda se zde zadrží.

Připomeňme, že N je délka příkopu. Hledání nejvyššího místa M zvládneme v čase $\mathcal{O}(N)$, protože při jednom průchodu vstupu porovnáme každé dvě sousední výšky a uložíme si větší z nich. Samotný algoritmus vstup projde také pouze jednou a pro každou část vykoná konstantně operací, takže celková časová

složitost je $\mathcal{O}(N)$. Co se paměti týče, kromě samotného vstupu si pamatujeme pouze pár proměnných.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-Z2-5-2.py`



Na závěr dodáme, že i toto řešení se dá ještě vylepšit tak, abychom celý vstup prošli jen jednou. Na začátku bychom nehledali žádné maximum, ale místo toho bychom si pořídili dvě proměnné pro počítání maxima a dva ukazatele na pozici – jednu dvojici pro procházení zleva a jednu pro průchod zprava.

Na další úsek bychom posunuli ukazatel na té straně, kde je hodnota maxima menší. Do výsledného objemu přičteme to samé jako v předchozím algoritmu – rozdíl maxima a aktuální výšky. Tím spojíme všechny tři fáze (nalezení globálního maxima, počítání průběžných maxim zleva a zprava a počítání výsledku) dohromady.

Ohledně paměti platí to samé, co v předchozím případě. Toto poslední řešení by v realitě vyhrálo v rychlosti, alespoň pokud byste data četli z obrovského souboru na (pomalém) disku. Dva průchody by trvaly dvakrát tak dlouho, tři třikrát, atd.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-Z2-5-3.py`

Katka Zákravská

28-Z2-6 Kalamita

Jako první si uvědomíme jedno velmi důležité pozorování, nikdy se nevyplatí zrušit zastávku, která není listem (tj. není konečná). Proč tomu tak je? Inu, zrušení ne-konečné zastávky zruší i všechny zastávky za ní (viz zadání), tedy i alespoň jednu další (konečnou) zastávku.

To máme jisté kvůli faktu, že graf stanic je strom, tedy neobsahuje žádné kružnice. Přeci jen když nemáme kružnice a vydáme se po nějaké cestě z centrální stanice, třeba přes tu naši rušenou, tak jednou určitě dojedeme na konec, tj. na konečnou stanici. Koneckonců, naše město není nekonečné a absence kružnic zaručuje, že neprojedeme nic dvakrát.

Zrušení zastávky, která není konečná, tedy znamená, že odřízneme od světa lidi jak z té naší zastávky, tak i její odpovídající konečné. A to je určitě víc lidí, než by odřízlo zrušení jen konečné.

Tím jsme přišli na to, že určitě budeme vždycky rušit jen konečné zastávky a zastávky, které se staly konečnými zrušením nějakých předchozích.

KSP-Z

řešení

Pro jednodušší přemýšlení o zastávkách si ještě uvědomíme, že s centrální zastávkou můžeme počítat jako s úplně normální zastávkou, která má počet lidí nějaké extrémně velké číslo. To nám zajistí, že ji vždycky zrušíme jako poslední. To chceme, jelikož její zrušení automaticky zruší všechny ostatní zbývající zastávky, a tedy je, dle argumentu výše, nevýhodné.

Nyní už si stačí rozmyslet, v jakém pořadí to budeme dělat. I to je ovšem vcelku jasné, zadání nám totiž přikazuje, abychom vždy zrušili tu nejméně zaplněnou zastávku, tedy takovou, která má nejnižší číslo. V tuto chvíli by nás mohlo napadnout najít všechny konečné zastávky, v nich určit tu nejmenší, zrušit ji, znovu najít všechny konečné zastávky, a takhle dokud nám nezbyde jen centrální zastávka.

To by samozřejmě fungovalo, není to ovšem ani zdaleka efektivní řešení. Předně, konečné zastávky není třeba vždy hledat úplně od začátku. Zřejmě nám totiž platí, že jednou konečné zastávky budou konečné i poté, co nějakou odstraníme. A přibýt do klubu konečných může odstraněním nějaké konečné zastávky vždy jen jedna nová, a to konkrétně ta, která se zrušenou zastávkou sousedila.

Dále si také uvědomíme, že z množiny aktuálně konečných zastávek v každé fázi programu jen odstraňujeme minimum a přidáváme jeden prvek, tedy operace, které až nápadně volají po tom, abychom použili minimovou haldu. Pokud náhodou nevíte, co je minimová halda, pak si určitě přečtěte naši kuchařku.[20] Ve zkratce jde ale o datovou strukturu, která po vybudování (trvajícím lineární čas) dokáže v logaritmickém čase vracet nejmenší prvek a jejíž oprava po přidání libovolného prvku trvá taktéž nejhůř logaritmický čas.

Když si dáme tyto dvě úvahy dohromady, tak se konečně dostaneme na ideální řešení. Předně si najdeme všechny konečné zastávky a vytvoříme si z nich minimovou haldu. Jak hledání (stačí prostě projít graf a vedle si pamatovat zastávky, ze kterých už dál jít nelze), tak tvorba haldy nám zabere lineární čas, tedy $\mathcal{O}(N)$, kde N je počet zastávek.

Následně vždy z haldy odstraníme minimum za $\mathcal{O}(\log(N))$, podíváme se na jejího předchůdce, jestli nám vznikla nová konečná zastávka, a pokud ano, tak jej přidáme do haldy v čase $\mathcal{O}(\log(N))$.

Takovýchto kroků, nebo fází chcete-li, provedeme nejhůř tolik, kolik je zastávek. Pokaždé totiž zrušíme jednu zastávku a každou zastávku zrušíme maximálně jednou. Výsledná časová složitost je tedy $\mathcal{O}(N + N \cdot \log(N))$, což je stejné jako $\mathcal{O}(N \cdot \log(N))$.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-Z2-6.py`

Petr Houška

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

Třetí série

28-Z3-1 Místo oslavy

K vybrání vhodného místa pro pořádání oslavy stačí vedení školy najít první a poslední dům od začátku města, kde bydlí zájemci o oslavu. Je zřejmé, že právě obyvatelé těchto domů to budou mít nejdále. Ovšem to znamená, že stačí najít minimum a maximum v zadané posloupnosti čísel, což zvládneme během jediného průchodu vstupem.

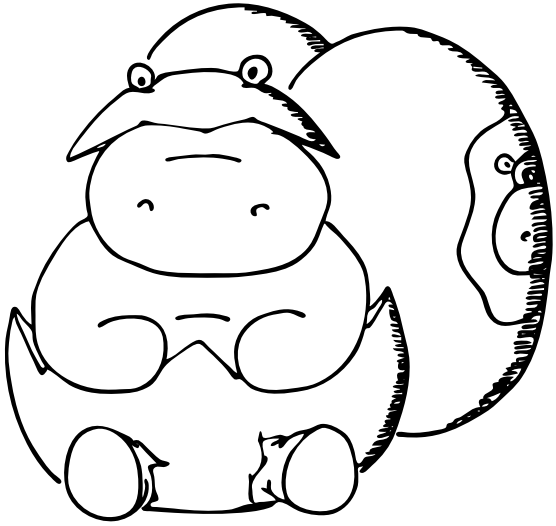
My ale ještě chceme, aby to oba nejvzdálenější zájemci měli co nejblíže, tj. aby tato nejdelší vzdálenost byla co nejmenší. Proto budeme hledat střed mezi dvěma zmíněnými domy, vzdálenost z obou pak bude stejná.

Vzhledem k tomu, že čísla budov jsou celočíselná, chceme opět celočíselný výsledek. Pokud je vzdálenost mezi prvním a posledním domem liché číslo, jsou dvě možnosti pro konání oslavy, obě stejně výhodné. My můžeme vybrat libovolnou z nich, zvolme si vždy třeba tu s nižším číslem. Nyní bude stejný postup pro obě varianty, tedy když je ona vzdálenost liché i sudé číslo. Minimum si označíme Min, maximum jako Max, výsledná pozice sálu bude $(Max + Min)$ div 2.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-Z3-1.py`

Zuzka Drázdová



KSP-Z

28-Z3-2 Zlomkovník

řešení

Z postupu, jakým byl strom tvořen, si všimneme, že pokud je čitatel menší než jmenovatel, byl poslední vybrán levý syn. Naopak, pokud je čitatel větší než jmenovatel, byl poslední pravý syn. Proč tomu tak je? Levý syn je tvořen $\frac{A}{A+B}$ a je zřejmé, že $A+B$ je větší než A (protože ve stromu jsou samá kladná čísla). Podobně u pravého syna, který je $\frac{A+B}{B}$, je $A+B$ větší než B.

Stačí nám tedy v podstatě projít zpět ke kořeni a zaznamenat, kde jsme byli. Jak taková cesta bude vypadat, zjistíme právě porovnáváním čitatele a jmenovatele. Zapamatované pořadí bude obrácené, než jaké máme vypsat v odpovědi, proto jej nesmíme zapomenout poté obrátit.

Jak přesně budeme postupovat? Nejprve porovnáme čitatele a jmenovatele, pokud je čitatel menší, tedy zlomek je ve stavu $\frac{A}{A+B}$, odečteme od jmenovatele čitatele a získáme tím otce: $\frac{A}{A+B-A} = \frac{A}{B}$. Pokud je čitatel větší, musíme k získání otce odečíst od čitatele jmenovatele: $\frac{A+B-B}{B} = \frac{A}{B}$. Podle porovnání nezapomene poznamenat, jestli jsme se jednalo o pravého nebo levého syna.

Pokračovat budeme opět k novému otci aktuálního syna. Tento postup budeme opakovat, až dokud se nedostaneme do kořene stromu. Poté nezapomeneme vypsat posloupnost R a L v obráceném pořadí a máme výsledek.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-Z3-2.py`

Zuzka Drázdová

28-Z3-3 Posloupnost za trest

Nejdříve zkusíme počítat postupně členy posloupnosti $P_1, P_2, \ldots, P_N$. První člen P_1 je triviální, P_{i+1} vyrobíme z P_i tak, že budeme procházet číslice zleva doprava a počítat, jak dlouhé úseky stejných číslic potkáme. To pokaždé zvládneme v čase $\mathcal{O}(\textit{počet cifer})$.

Zpočátku to půjde dobře, jenže cifer bude rychle přibývat. (Dokonce exponenciálně: i-tý člen má přibližně 1.304^i číslic. To dokázal John Conway fascinujícím způsobem. Pokud máte chuť na trochu pokročilejší matematiky, směle se začtěte do článku ve wikipedii[21] a odkazů, které z něj vedou.)

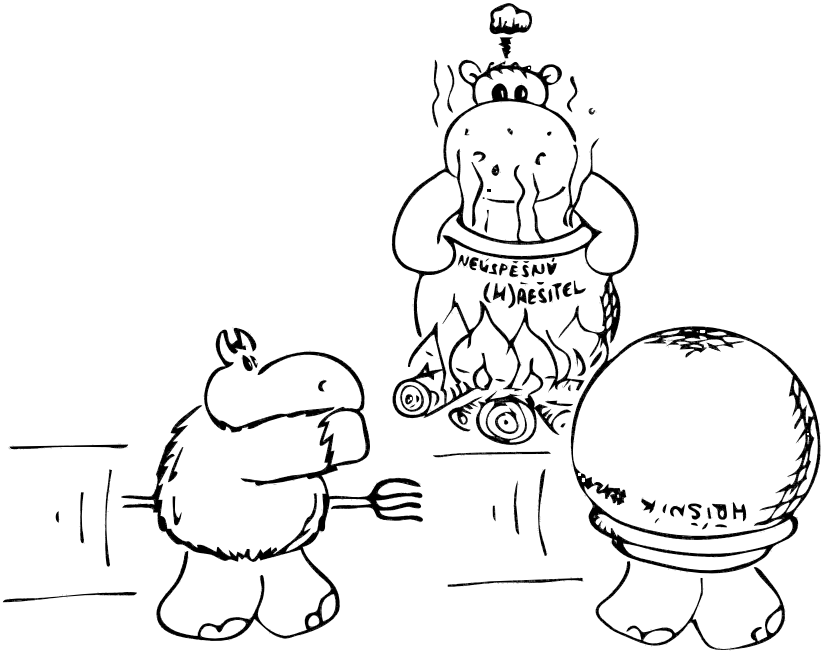
Nás naštěstí zajímá jen prvních K číslic výsledku. Zkusíme tedy počítat jen prvních K číslic každého členu posloupnosti. Je to trochu riskantní, protože by se mohlo stát, že při vytváření dalšího členu budeme potřebovat víc číslic předchozího členu, než jsme spočítali. Pokusem si ale můžeme ověřit, že posloupnost roste dost rychle, takže se to pro $K \geq 10$ nestane.

[21] `https://en.wikipedia.org/wiki/Look-and-say_sequence`

Z toho dostaneme řešení o složitosti $\mathcal{O}(NK)$, dost rychlé na to, aby vyřešilo všechny naše vstupy během pár desítek sekund. Přesto ho zkusíme ještě trochu zrychlit.

Označme Q_i hodnotu P_i zkrácenou na prvních K číslic. Pokud je K malé, začnou se hodnoty $Q_1, Q_2, \ldots$ brzy opakovat. Náš algoritmus si proto bude ve slovníku pamatovat všechna Q_i, která už viděl, a číhat na první opakování. Jakmile zjistí, že $Q_i = Q_j$ pro nějaké $j < i$, musí se od i-té pozice neustále opakovat úsek $Q_j, Q_{j+1}, \ldots, Q_{i-1}$. Stačí tedy zjistit, který z opakovaných členů vyjde na hledané Q_N, a vypsat ho.

Zdálo by se, že perioda bude pro velká K dostatečně daleko, takže tento trik nepomůže. Neváhejme a zkusme to. Překvapení: pro maximální povolené $K = 300\,000$ je $Q_{55} = Q_{52}$ a zjevně to platí i pro všechna menší K. Stačí tedy spočítat prvních 55 členů posloupnosti a známe odpověď pro jakékoliv N.



Program k základnímu řešení (Python 3):
`http://ksp.mff.cuni.cz/viz/28-Z3-3.py`

Program k rychlejšímu řešení (Python 3):
`http://ksp.mff.cuni.cz/viz/28-Z3-3-magic.py`

Martin „Medvěd“ Mareš

KSP-Z

28-Z3-4 Zbývající úkoly

řešení

Nejprve vymyslíme jednoduché řešení, a pak si jej prostým trikem velmi zrychlíme. Pojďme na to.

Začneme tím, že si načteme celý vstup. Pro každý úkol si budeme chtít pamatovat jeho délku a seznam úkolů, na kterých je závislý. Toto nejde udělat jinak, než že celý vstup načteme do paměti. V poli t uschováme délky, v poli *dep* potom seznamy úkolů, které musíme provést dříve. Jak to udělat pohodlně se můžete podívat do programu v Pythonu.

Potom si napíšeme rekurzivní funkci, která spočítá, kdy nejdříve může být úkol x dokončený:

```
def finish_time(x):
  deptime = 0
  for d in dep[x]:
    deptime = max(deptime, finish_time(d))
  return deptime + t[x]
```

V této funkci vezmeme maximum z času dokončení všech úkolů, na kterých je x závislý, a přičteme dobu trvání x. Jak ji použijeme?

Abychom si zjednodušili kód, můžeme použít podlý trik. Přidáme si neexistující úkol A, který bude trvat 0 jednotek času, zato ale bude závislý na všech ostatních. Jeho čas dokončení je pak přesně to, co po nás úloha žádá.

Jak to bude rychlé? Představte si takový vstup:

$$1 \leftarrow 2 \leftarrow 3 \leftarrow 4 \leftarrow 5$$

V takovém případě se funkce zavolá postupně s těmito parametry:

$$1, 2, 1, 3, 2, 1, 4, 3, 2, 1, 5, 4, 3, 2, 1$$

Zde je nutno si všimnout, že se funkce volá mnohokrát třeba pro jedničku, přestože její návratová hodnota bude vždy stejná. Vždyť si výsledky můžeme uložit:

```
def finish_time(x):
  if x not in fintime:
    deptime = 0
    for d in dep[x]:
      deptime = max(deptime, finish_time(d))
    fintime[x] = deptime + t[x]
  return fintime[x]
```

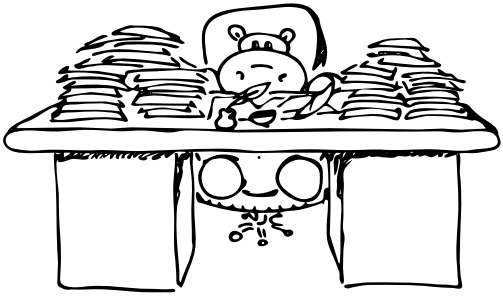
Jak bude vypadat seznam volání teď?

$$1, 2, 1, 3, 2, 4, 3, 5, 4$$

Nemůžeme říct, že by se nyní volala pro každé x pouze jednou. Čeho jsme se ale zbavili, je neustálé opakování „ocásků“, pro které už známe odpověď, a můžeme ji vrátit daleko dříve.

Pokud bychom chtěli časovou složitost vyčíslit, všimneme si že funkci zavoláme jednou jako závislost na virtuálním úkolu A, a jednou pro každou závislost jiného úkolu y na tomto úkolu x – to právě tehdy, když poprvé počítáme y. Dohromady víme, že závislostí je K, takže celkový počet volání bude $N + K$. N (počet úkolů) musíme započítat, protože může být klidně větší než K.

Úplně stejně se na to podíváme, pokud budeme počítat čas pro funkci samotnou. Všechna volání dohromady ve vnitřním for cyklu stráví $\mathcal{O}(K)$ času, všechno ostatní už je jen konstantní zpomalení. Proto bude mít výpočet časovou složitost $\mathcal{O}(N + K)$. Do tohoto času se vejdeme i s načtením vstupu a vypsáním výstupu, a stejnou funkcí odhadneme i prostorovou složitost.



Doteď se úloha zdála růžová. Bohužel jsme si při testování úlohy nevšimli výrazného omezení jazyka, který pro řešení úloh propagujeme – Pythonu. Ten totiž ve výchozím nastavení neumožňuje zanořit volání funkcí více než tisíckrát.

Tentokrát tedy přidáváme tři zdrojové kódy. První problém obchází tím, že nastavení změní. Bohužel, bude fungovat jen na Linuxu. O důvodech se můžeme pobavit na fóru.

Program (Python 3, Linux):

`http://ksp.mff.cuni.cz/viz/28-Z3-4-linux.py`

Druhý jej řeší tím, že nepoužívá volání funkcí, ale simuluje jej pomocí seznamu. Technika je to obecná a dá se použít vždy, jen zdrojový kód není úplně dobře čitelný.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-Z3-4-explicit.py`

Program (C++):

`http://ksp.mff.cuni.cz/viz/28-Z3-4.cpp`

Ondra Hlavatý

KSP-Z

28-Z3-5 Ukotvení stromu

řešení

Pro začátek předpokládejme, že žádné dva body nemají stejnou x-ovou ani y-ovou souřadnici.

Nejprve body seřadíme podle x-ové souřadnice.

Pokud jsou všechny body středově souměrné, musí být bod nejvíce vlevo souměrný s tím nejvíce vpravo. Tím pádem podle těchto dvou bodů jednoznačně určíme střed S – bude přesně v polovině mezi nimi.

Zbývá ověřit, že všechny ostatní body jsou podle S souměrné. Druhý musí být souměrný s předposledním, třetí se třetím od konce a tak dál pro všechny body. Pokud má být souměrný bod A s bodem B podle středu S, musí platit $S_x - A_x = B_x - S_x$ a zároveň $S_y - A_y = B_y - S_y$.

Pokud se jediná podmínka poruší, odpovíme, že nejsou souměrné. Když žádný test neselže, odevzdáme střed S.

Algoritmus funguje i pokud mají nějaké body stejnou x-ovou souřadnici. V takovém případě je potřeba při řazení porovnávat primárně podle x-ové a sekundárně podle y-ové souřadnice.

Proč to funguje? Protože přesně takové pořadí bychom dostali, kdybychom celou rovinu nepatrně pootočili, aby se žádné x-ové souřadnice neshodovaly.

Zbývá dořešit časovou složitost. Body, který je N, umíme seřadit v čase $\mathcal{O}(N \log N)$ třeba MergeSortem. Pokud je uchováváme v poli, bude každý z N testů trvat konstantní čas. Celková časová složitost je tedy $\mathcal{O}(N \log N)$.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-Z3-5.py`

Martin Španěl

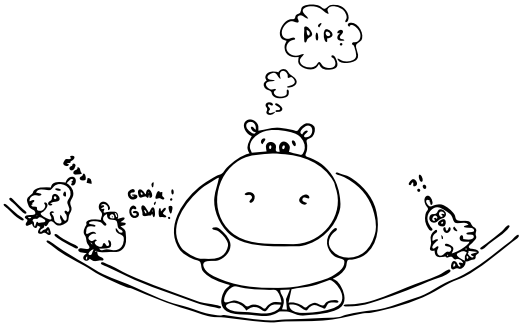
28-Z3-6 Šíření drbů

Dobrým prvním krokem může být zkusit správný počet potřebných přestávek odhadnout. Proto se pokusme spočítat správné výsledky pro malý počet kamarádek.

Máme-li jen jednu kamarádku, ta zná svůj drb hned (tedy nepotřebuje ani jednu přestávku). Dvě kamarádky potřebují jednu přestávku na to, aby si vyměnily své drby. Čtyři kamarádky (jak máme v zadání) potřebují dvě přestávky. S trochou trpělivosti můžeme zkoušením zjistit, že pro osm kamarádek jsou potřeba tři přestávky.

Z toho (a s nápovědou v zadání, že máme uvažovat pouze N, která jsou mocniny dvojky) můžeme odhadnout, že potřebujeme $\log_2 N$ přestávek. Jinak řečeno, pokud je 2^k kamarádek, klepy si vymění během k přestávek.

Pojďme se tedy podívat, jestli se nám podaří najít takové přiřazení kamarádek, aby se klepy šířili tak rychle. Začněme u malého počtu kamarádek. Když máme pouze dvě kamarádky, tak stačí když se potkají během jediné přestávky. Řešení pro čtyři kamarádky máme rovnou v zadání (nejprve se potkají A–B a C–D, poté A–C a B–D).



Jak můžeme přistupovat k tomu, když je ve třídě kamarádek osm? Rozdělme si je na dvě skupiny po čtyřech a nechme nejprve rozšířit drby v rámci jednotlivých skupinek. Jak jsme ukázaly výše, aby každá kamarádka znala všechny skupinové drby, stačí nám dvě přestávky.

Všimněte si, že na rozšíření drbů mezi skupinkami nám pak stačí už jen jedna přestávka – první kamarádka z první skupinky se podělí o drby s první kamarádkou z druhé skupiny, druhá s druhou a tak dále. Každá kamarádka pak bude vědět všechny drby ze své skupiny (ty znala i před touto poslední přestávkou) a tím, že se potkala s někým z druhé skupiny, tak se dozvěděla i všechny drby z druhé skupiny.

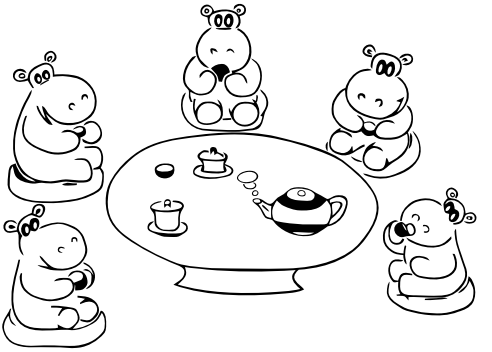
Stejný trik můžeme použít i pro šestnáct kamarádek – stačí je rozdělit na dvě skupiny. Každá skupinka během tří přestávek rozšíří drby v rámci své skupiny

KSP-Z

řešení

(tak jak bylo ukázáno v předchozím odstavci). Během následující přestávky si pak každá kamarádka popovídá s příslušnou kamarádkou z druhé skupiny, čímž se dozví všechny drby.

Ve skutečnosti takto můžeme postupovat pro libovolný počet kamarádek. Kamarádky rozdělíme na dvě skupiny a zamyslíme se nad tím, jak rozšířit drby v rámci skupin. Potom během jedné přestávky už rozšíříme drby napříč skupinami.



Tímto způsobem spotřebujeme za každé zdvojnásobení počtu kamarádek jen o jednu přestávku více. Dostali jsme se tedy přesně na náš vytyčený cíl v počtu přestávek.

Tato technika rozdělování problémů na několik menších, které se řeší vlastně stejnou technikou, je v informatice poměrně populární a nazývá se *Rozděl a panuj*. Pokud tě zaujala a chceš se o ní dozvědět více, tak další informace najdeš v naší kuchařce.[22]

Na závěr si pojďme ještě ukázat, že rychleji to skutečně nejde. Představme si na chvíli, že už proběhlo několik přestávek. Označme si d počet drbů, které zná nejpopulárnější kamarádka (tedy ta, která zná ze všech kamarádek nejvíce klepů). Druhá nejpopulárnější tudíž zná nejvýše také d drbů. Když se tyto dvě kamarádky během další přestávky potkají, obě budou znát nejvýše $2d$ drbů.

Všimněme si, že (jelikož se jednalo o nejpopulárnější kamarádky) nikdo nebude po další přestávce znát více jak $2d$ drbů. Také si uvědomme, že toto platí pro každou přestávku. Takže, bude-li p značit počet drbů, které zná nejpopulárnější kamarádka, p se nám každou přestávku nejvýše zdvojnásobí.

Toto už nám (s tím, že na začátku je p jedna), dává právě naší formulku, že po k přestávkách bude p nejvýše 2^k. Tedy při 2^k kamarádkách potřebuje i nejpopulárnější kamarádka alespoň k přestávek.

Janka Bátoryová & Dominik Smrž

[22] `http://ksp.mff.cuni.cz/viz/kucharky/rozdel-a-panuj`

Čtvrtá série

28-Z4-1 Půdorys

Důležitou součástí řešení bylo uvědomit si, že hledané obdélníky mají jednu zajímavou vlastnost – body ležící na jedné straně obdélníka mají vždy stejnou jednu souřadnici. Takže svislé hrany obdélníka mají stejnou x-ovou souřadnici a vodorovné zase y-ovou.

My musíme pro každý bod na vstupu ověřit, zda leží na nějaké hraně obdélníka. K tomu potřebujeme nejdříve znát souřadnice těchto hran. To už zvládneme jednoduše.

Všechny body na levé svislé hraně budou mít nejmenší x-ovou souřadnici, body na pravé svislé hraně budou mít největší x-ovou souřadnici. Stačí nám tedy pouze projít x-ové souřadnice všech bodů a zapamatovat si minimum a maximum. Pro vodorovné hrany analogicky s y.

Teď už jenom ověříme, že každý bod leží na nějaké hraně jednoduchým porovnáním jeho souřadnic se zjištěnými souřadnicemi hran.

Program (Python 3):

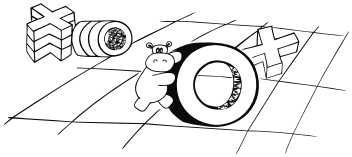
`http://ksp.mff.cuni.cz/viz/28-Z4-1.py`

Martin Šerý

28-Z4-2 Vykopávky

Je zřejmé, že když mají mít všechny čtyři obdélníkové oblasti stejný součet, bude to právě součet všech políček vydělený čtyřmi. Abychom ho tedy spočítali, projdeme si nejdřív jednou všechna čísla a sečteme je. Zatím nezáleží na tom, v jakém pořadí je projdeme.

Dále si všimneme, že část nahoře od bodu S (všichni se tu mají potkat) a část dole budou mít stejný součet, a to právě polovinu součtu celkového. Budeme tedy procházet čísla ještě jednou, tentokrát po řádcích, budeme je opět sčítat, ale zastavíme se, když se dosčítáme k polovině celkového součtu. Zapamatujeme si, kolik řádků jsme zatím prošli, to bude totiž výška obdélníku, kterou chceme zjistit.



Do třetice všeho dobrého projdeme čísla ještě jednou, tentokrát po sloupcích. Opět je budeme sčítat, dokud se nedostaneme k polovině celkového součtu. Počet zatím prošlých sloupců nám dá hledanou šířku obdélníku.

Protože jsme čísla na vstupu prošli třikrát, složitost bude $\mathcal{O}(RS)$, což je počet řádků vynásobený počtem sloupečků, tedy počet všech políček.

Ještě se zamyslíme, jak by se dala uspořit paměť – co kdybychom si nemuseli pamatovat všech R čísel? Vždy potřebujeme počítat součty celých řádků a celých

sloupců. Takže by stačilo si při načítání vstupu průběžně sčítat všechna čísla v každém řádku a sloupci, a pamatovat si jen tyto údaje, kterých je $R + S$.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-Z4-2.py`

Zuzka Drázdová

28-Z4-3 Mocniny

Přímočarým řešení Kevinova problému je si pro každé číslo x spočítat jeho druhou a třetí mocninu, a pak ověřit, že i ony leží v naší posloupnosti. To můžeme udělat třeba sekvenčním vyhledáním x^2 a x^3. Zapamatujeme si právě ta x, pro která jsme našli dané mocniny. Ze zapamatovaných x už jenom stačí najít minimum.

Toto řešení má kvadratickou složitost $\mathcal{O}(N^2)$, kde N je počet čísel na vstupu. Pro každé číslo projdeme všechna ostatní a zjistíme, jestli mezi nimi je i jeho druhá a třetí mocnina.

Rychlejší řešení získáme úpravou toho přímočarého. Nejvíce času strávíme hledáním x^2 a x^3. Tento krok můžeme urychlit tím, že si na začátku celou posloupnost setřídíme. Poté už můžeme mocniny hledat pomocí binárního vyhledávání.[23]

Navíc nám stačí se zastavit u prvního x v setříděné posloupnosti. Toto x je totiž první pro které jsme našli mocniny, takže všechna další čísla už budou pouze větší.[24]

Setřídění nám zabere $\mathcal{O}(N \log N)$ času. Binární vyhledání jednoho prvku má logaritmickou složitost a jelikož hledané x může být téměř až na konci setříděné posloupnosti, provedeme jej pro každý prvek. Tj. celé vyhledávání trvá $\mathcal{O}(N \log N)$ času, což je stejné jako složitost setřídění, bude to tedy i výsledná složitost.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-Z4-3-bin.py`

Pokud váš programovací jazyk disponuje datovou strukturou pro množinu, často pojmenovanou `set`, můžete ji s výhodou použít. Ta si interně udržuje čísla setřízená a vlastně na nich binární vyhledávání provádí – jen nemá čísla uložená v seznamu. Rychlé řešení pak vyjde velmi elegantně na jeden řádek.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-Z4-3-set.py`

Martin Šerý

[23] `http://ksp.mff.cuni.cz/viz/kucharky/zakladni-algoritmy`

[24] V Pythonu se hodí použít `set`, který je přímo dělaný na podobné vyhledávání.

28-Z4-4 Čtyřková

Úloha byla trochu podlá, protože zjistit odpověď pro jedno číslo je vlastně stejně těžké, jako ji zjistit pro všechna najednou. Pojďme se na to podívat.

Začínáme s jednou čtyřkou. Jaká čísla jsme schopni vyrobit, pokud použijeme čtyřky dvě? Jsou to $8 = 4 + 4$, $0 = 4 - 4$, $16 = 4 * 4$ a $1 = 4/4$. Protože tato čísla s jednou čtyřkou určitě nevyrobíme, zapamatujme si o nich že jdou vyrobit nejlépe se dvěmi.

A co když si dovolíme další čtyřku navíc? Pak se stačí podívat na ta čtyři čísla, která umíme vyrobit se dvěma čtyřkami, a zkusit k nim všemi čtyřmi operacemi přidat třetí. Tím, že se operace vyhodnocují zleva doprava, snadno spočítáme výsledek nezávisle na tom, jak jsme dokázali vyrobit číslo původní.

Vytvoříme si tedy pole výsledků `v` a pro každé číslo od 0 do 9 999 si do něj uložíme třeba -1, protože jej zatím vyrobit neumíme. Jen pro číslo 4 víme, že jej umíme vyrobit za pomocí jedné čtyřky.

Pak si pořídíme frontu čísel, která jsme dokázali vyrobit, ale ještě z nich neprozkoumali možnosti rozšíření o další čtyřku. Na začátku bude obsahovat jen tu první čtyřku.

Následně vždy vezmeme číslo x z fronty a podíváme se na výsledek operace $x+4 = y$. Pokud jsme y ještě neviděli, ve `v[y]` bude stále mínus jednička. V tom případě toto musí být nejkratší způsob, jak y vyrobit, a můžeme do `v[y]` uložit `v[x] + 1`. Současně y přidáme do fronty.

To samé provedeme pro ostatní operace. Dáme si při tom pozor, abychom přeskočili dělení, pokud x není dělitelné čtyřmi.

Ve výpočtech jsme trochu zanedbali to, že Sářina kalkukačka umí počítat jen s čtyřcifernými čísly. Jak bylo napovězeno v zadání, s výhodou použijeme operaci modulo. Každé y proženeme jednoduchým výrazem:

$$y_2 = (y_1 + 10\,000) \bmod 10\,000$$

Přičítat můžeme bezpečně, i když je číslo kladné, protože se 10 000 modulením zase odečte.

Až frontu vyprázdníme, určitě jsme našli všechna čísla, která vyrobit jdou, a v poli `v` máme výsledky, které chceme vracet. Stačí se do `v` podívat na správné místo a vypsat výsledek.

Pokud trochu znáte grafové algoritmy, můžete si všimnout, že popsaný způsob je vlastně prohledávání grafu do šířky. V tomto grafu jsou vrcholy čísla a z každého vedou tři nebo čtyři hrany za operace.

Můžeme prozradit, že každé číslo takto vytvořit jde, a dokonce je nejkratší výraz až na dvě čísla jednoznačný. Pokud byste chtěli zároveň výraz vypsat, můžete si do `v` uložit i x a operaci, kterou jste y vyrobili, a z těchto pak výraz zpětně poskládat.

Algoritmus má trochu překvapivě konstantní časovou složitost. Nejprve proběhne předvýpočet, který není závislý na vstupu, a tedy je z principu konstantní. Po přečtení vstupu už se jen podíváme na správné místo do pole. Pokud bychom neměli pevně zadaný rozsah 10 000 čísel, ale nějaké nejvyšší číslo K, časová složitost by byla lineární s K. Stejně tak paměťová.

Aby byl váš výpočet ještě rychlejší, mohli jste použít trik. Na co výsledky počítat při každém startu programu znovu, když si je můžete spočítat jednou a uložit pro příště například do souboru? Pro získání plného počtu bodů to ale vůbec nebylo nutné, 10 000 čísel spočítáte popsaným způsobem i v Pythonu bleskově.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/28-Z4-4.py`

Ondra Hlavatý

28-Z4-5 Vyhlazení GPS

Úlohu si trochu upravíme – místo toho, abychom počítali průměr, budeme počítat pouze součet posledních K hodnot. Každé číslo pak jen před vypsáním vydělíme K, takže vyřešíme původní zadání, ale přemýšlet se o tom bude lépe.

Použijeme myšlenku pohyblivého okénka. Prohlasíme na chvíli, že K je 3. Známe součet $S_1 = x_1 + x_2 + x_3$, a dostaneme x_4. Jak z nich spočítat součet $S_2 = x_2 + x_3 + x_4$?

Každý vidí, že stačí odečíst x_1 a přičíst x_4. Důležité je, že K může být úplně libovolné a vždy stačí nejstarší číslo odečíst a nové přičíst, tedy provést konstantní počet operací.

Pořídíme si proto pole, do kterého si budeme ukládat posledních K čísel, a jejich součet si budeme počítat průběžně. Vždy první číslo z pole vyhodíme a na konec jiné přidáme.

Počkat, tady něco nehraje. Vyhodit první číslo z pole nám zabere spoustu času, protože musíme ta ostatní posunout!

Představte si, že byste si mohli pořídit pole zatočené do kruhu, takový prstenec. Někdy se mu proto anglicky říká *ring buffer*, cyklický buffer. Prstenec bude stát před námi na stole a místo abychom my točili s prstencem, budeme obcházet okolo stolu.

Tolik představa, teď jak to implementovat. Pole zůstane rovné a bude mít K prvků jako předtím. V prvním kole budeme pracovat s prvním prvkem s indexem 0. V druhém s druhým… až v K-tém

s K-tým s indexem $K-1$. Potom by se nám hodilo, abychom pokračovali znovu od začátku. K tomu nám poslouží stará známá funkce modulo. V i-tém kole budeme pracovat s prvkem s indexem $i \bmod K$.

Nechť se tedy průběžný součet jmenuje S a udržujeme cyklický buffer B. Jak S, tak všechny prvky B budou na začátku nulové. V i-tém kole odečteme od S prvek $B[i \bmod K]$ a následně na jeho místo vložíme nové číslo x_i, a zpátky ho přičteme do S.

Pokud by se nám nelíbilo, že počítadlo kroků i roste do nekonečna a může přetéci, budeme ho průběžně modulit. To totiž ničemu nevadí, my ho k jinému účelu nepotřebujeme.

Takto se nám podařilo udržovat průběžný součet s konstantní časovou složitostí na jeden krok. K celému výpočtu potřebujeme paměť velikosti $\mathcal{O}(K)$, kterou si ale vynulujeme na začátku, a v každém kroku přistupujeme pouze k jednomu prvku.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-Z4-5.py`

Ondra Hlavatý

28-Z4-6 Klučičí drby

Nejprve si uvědomíme, že drb se vždy přestane šířit – kluků je konečně mnoho a každý říká drb právě jednomu, takže časem se určitě dostane k někomu, kdo už jej zná.

Navíc víme, že o přestávce je stejný drb sdělován nejvýše jednou (říkáme ho pouze svému nejlepšímu kamarádovi). Proto se každý drb dostane do skupiny kamarádů, kteří si drby předávají do kruhu. Drb se zastaví tehdy, když se jej dozví všichni z nějakého cyklu.

Zkusme pro každý drb spočítat, kolik kluků jej bude znát. Uděláme to přesně tak, jak se drby šíří. Začneme u našeho nejlepšího kamaráda, poté je na řadě nejlepší kamarád toho našeho, pak jeho nejlepší kamarád, … A tak dál, dokud nenarazíme na nějakého, který už drb zná.

A abychom věděli, kolik kluků se ho dozvědělo, stačí si postupně nejlepší kamarády počítat (v každém kroku přičteme jedničku k počítadlu).

Jak poznáme, že jsme se právě dostali k někomu, kdo už drb zná? Budeme si kluky značit. Pořídíme si třeba pole plné nul, kde máme pro každého kluka jedno políčko. Vždy, když se někdo dozví drb, uložíme si do jeho políčka jedničku. Před šířením dalšího drbu si celé pole vynulujeme.

Složitost tohoto řešení je kvadratická vzhledem k počtu kluků ve třídě – každý drb sledujeme zvlášť, drbů je stejně jako kluků. Drb se navíc může rozšířit až ke všem klukům.

KSP-Z

řešení

Ale jde to ještě zrychlit! Všimněte si, že často počítáme vícekrát, jak dlouho se šíří nějaký drb od stejného kluka. Mohli bychom si tedy pro každého kluka pamatovat, ke kolika dalším se od něj drb dostane. Tomuto číslu budeme říkat *společenský dopad*.

Jak společenské dopady získat a jak si je pamatovat? Jednoduše. Jako paměť nám poslouží další pole. Začneme opět simulovat šíření nějakého drbu. Postupně procházíme a počítáme nejlepší kamarády jako v prvním řešení, počítadlo si označíme jako K.

Rozdílem oproti kvadratickému řešení bude, že se zastavíme i tehdy, pokud řekneme drb někomu, jehož společenský dopad už známe (budiž to číslo L). $K+L$ nám udává celkový počet kluků, kteří se dozví právě simulovaný drb.

Skočíme zpátky na začátek a projdeme stejnou cestu znovu, ale budeme u toho ukládat dopady jednotlivých kluků – první má $K+L$, druhý $K+L-1$, další $K+L-2$, ... Těm, kteří už mají společenský dopad spočítaný, ho nepřepisujeme, dokonce u prvního takového se opět zastavíme.

Pořád se nám může stát, že se simulace zastaví kvůli podmínce z prvního řešení (přestane se i šířit drb). Připomeňme, že drb se zastaví, když se dostane do cyklu, a dozví-li se ho někdo z cyklu, dozví se ho i všichni ostatní z něj. To znamená, že společenský dopad všech kluků na cyklu bude stejný – délka cyklu (drb se z tohoto kruhu nikdy nikam dál nedostane).

Místo, kde se v cyklu zastavíme, je zároveň i to místo, kde do něj vstoupíme. Zapamatujeme si ho. Opět skočíme na začátek a ukládáme K, $K-1$, $K-2$, ... Jakmile znovu narazíme na cyklus, přestaneme odečítat jedničku a ukládáme stejné číslo (právě délku cyklu).

Poznámka na konec: Tuto vylepšenou simulaci postupně spustíme pro každý drb (nemusí totiž existovat kluk, jehož společenský dopad postihne všechny).

Zrychlili jsme řešení? Simulujeme opět všechny drby a simulace může trvat lineárně s počtem kluků ve třídě. Jenže čím déle trvá, tím více společenských dopadů spočítáme. Trvá-li K kroků, zapamatujeme si tím i K dosud neznámých dopadů. To ovšem znamená, že už je žádná další simulace nebude počítat – od každého kluka šíříme drb pouze jednou.

Dohromady uděláme tolik kroků, kolik je ve třídě kluků. Celková časová složitost je lineární, paměťová také.

Pokud by nás zajímala časová složitost jedné simulace, můžeme o ní tvrdit, že je amortizovaně konstantní. Sice můžou některé trvat dlouho, ale v průměru (když pustíme simulaci pro všechny drby) nám jedna trvá pouze konstatně dlouho.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-Z4-6.py`

Katka Zákravská & Jenda Hadrava

Pořadí řešitelů KSP-Z

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
0.				24	264.0
1.	Roman Bujdák	G JM Galanta	2	24	236.5
2.	Václav Brož	GZborovPH	1	24	233.0
3.	Pavel Koch	GTěš	4	24	226.7
4.	Daniel Skýpala	GTomkovaOL	−2	23	226.5
5.	Jakub Šťastný	G BO-Řeč	1	23	214.5
6.	Jiří Moravčík	GUHradiště	2	21	180.0
7.	Jiří Löffelmann	GLitoměřPH	2	16	150.5
8.	Vojtěch Hudec	G_ČTřebová	2	17	144.0
9.	Dennis Pražák	GJirsíkaČB	1	15	136.0
10.	Michael Bausano	GTěš	4	16	133.5
11.	Vendula Kuchyňová	GMLerchaBO	2	13	129.0
12.	Martin Bencko	GOhradníPH	−1	23	128.5
13.	Vojtěch Březina	GCoubTábor	−1	14	122.0
14.	Anna Hollmannová	GSRandyJN	−1	18	119.5
15.–16.	Jakub Jirkal	GJungmanLT	1	13	115.0
	Václav Pavlíček	ZŠ Ždírec nD	0	13	115.0
17.	Tomáš Terem	GTajBanBys	4	11	106.0
18.	Ondřej Gonzor	G Brandýs	−1	15	104.5
19.	Lukáš Riedel	G Bílina	4	13	104.0
20.	Antonín Prantl	G Strakon	3	14	101.0
21.	Michal Rickwood	G_ČTřebová	2	13	95.5
22.	Pavel Turinský	G Brandýs	3	8	84.0
23.	Jonáš Havelka	GJírovcČB	0	9	82.0
24.	Martin Picek	GJirsíkaČB	1	9	81.5
25.–26.	Vojtěch Káně	G Brandýs	0	9	80.0
	Jáchym Solecký	PORGPha	3	8	80.0
27.–28.	Tomáš Domes	MendelG_OP	3	8	78.5
	Samuel Schneider	GTajBanBys	4	9	78.5
29.–30.	Jan Kaifer	GČesBrod	0	9	77.5
	Michaela Štolová	G Sokolov	4	9	77.5
31.	Ondřej Krsička	Integra BO	0	14	76.0
32.	Jindřich Dítě	ZŠKom2ŽS	0	12	71.5
33.	Josef Pospíšil	GÚstavníPH	2	9	69.0
34.	Petr Dedek	MensaG	0	7	66.0
35.	Radoslav Hašek	G_Čáslav	2	8	62.0
36.	Vojtěch Kuchař	ZŠ Sobotka	−1	9	60.9
37.	Roman Solař	GJarošeBO	4	6	60.0
38.	Tomáš Troján	G Cheb	0	9	58.5

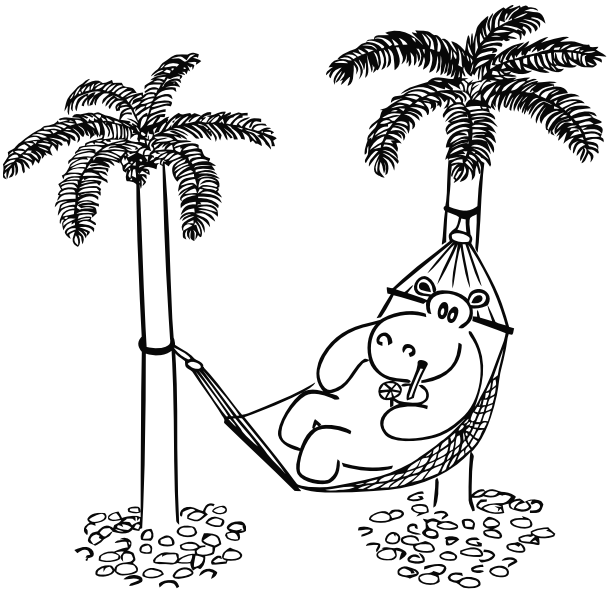
Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
39.–40.	Adam Husník	GArabskáPH	2	6	58.0
	Pavel Souček	G_Nymburk	4	6	58.0
41.	Ľuboš Kolumber	SpojŠ Popr	4	8	56.5
42.–43.	Michael Olšavský	GNadKavaPH	1	6	56.0
	Petra Štefaníková	GOlgHavl	4	7	56.0
44.–45.	David Nápravník	GLitoměřPH	3	6	54.0
	Jiří Sejkora	GVoděraPH	4	6	54.0
46.	Vít Gaďurek	Neuvedená	1	7	53.0
47.	David Blažek	SPŠÚžlabPH	3	5	52.0
48.–49.	Dominik Krasula	GKrnov	3	6	48.0
	Václav Šraier	GČeskoliPH	3	6	48.0
50.	Tomáš Novotný	G BO-Řeč	2	6	47.0
51.	Andrej Čermák	G JF Šaľa	2	5	46.0
52.	Janek Hlavatý	GJirsíkaČB	−3	6	44.3
53.	Michal Szymik	G Wicht	2	5	44.0
54.	Radek Jančík	GJarošeBO	3	6	43.0
55.	Ondřej Cach	ZŠPolab	0	6	40.5
56.–62.	Ondřej Baumgartner	GMost	4	4	40.0
	Ondřej Borýsek	GJarošeBO	3	4	40.0
	Hana Hladíková	GNadKavaPH	2	4	40.0
	Vojtěch Lanz	GZborovPH	2	4	40.0
	Jakub Matěna	GČeskoliPH	4	4	40.0
	Pavel Svoboda	ZŠJílovsPH	0	6	40.0
	Lucien Šíma	PORGPha	4	4	40.0
63.–65.	Robert Jaworski	GÚstavníPH	−2	4	36.0
	Jan Mráz	G Holice	2	5	36.0
	David Žáček	GZborovPH	3	4	36.0
66.	David Pavlík	GJarošeBO	3	3	30.0
67.–68.	Jakub Suchánek	GOpatovPHA	2	3	28.0
	Marek Černoch	GFPValMez	0	3	28.0
69.	Alexej Popovič	SlovanGOL	4	4	27.5
70.	Ondrej Potúček	G_GolNitra	2	4	27.0
71.–72.	Ludmila Bujnovská	MendelG_OP	2	4	26.0
	Matěj Šmíd	SPŠÚžlabPH	2	4	26.0
73.	Jiří Kvapil	GTomkovaOL	−2	3	24.0
74.	Eliška Vlčinská	GHladnov	1	4	23.7
75.	Filip Čermák	MendelG_OP	2	3	18.5
76.–78.	Radim Buráň	G UherBrod	1	2	18.0
	Jakub Dostál	SlovanGOL	2	4	18.0
	Pavel Martinec	GLesníZlín	2	2	18.0

Pořadí řešitelů KSP-Z

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
79.–82.	Jan Burda	G Holice	2	3	16.0
	Ladislav Töpfer	G DrJPekMB	1	2	16.0
	Jan Vaník	GRNK	1	2	16.0
	Adam Šánta	GJHroncaBA	1	2	16.0
83.	Milan Kubala	GTajBanBys	4	2	15.0
84.	Jan Vozár	G UherBrod	2	3	13.0
85.–87.	Vanda Hendrychová	GHeyrovPH	4	3	12.0
	Jakub Růžička	G_Nymburk	1	1	12.0
	Lenka Vincenová	GTomkovaOL	2	1	12.0
88.	Michal Mlčoch	G UherBrod	1	2	10.0
89.–93.	Martin Hofbauer	G BO-Řeč	1	1	8.0
	Filip Matějka	GZborovPH	2	1	8.0
	Václav Plavec	GTep	3	1	8.0
	Daniel Pluskal	G BO-Řeč	2	1	8.0
	Martin Zima	G Holice	2	1	8.0
94.–95.	Dominik Karas	GTěš	4	2	7.0
	Roman Ondráček	GBoskovice	2	3	7.0
96.	Matěj Hudec	CírkG Plzeň	4	4	6.5
97.–98.	Josef Martínek	GPelhřimov	2	1	5.0
	Robert Wiesner	GJosefskPH	4	2	5.0
99.–100.	Martin Hubata	GMikulášPL	0	1	2.0
	Daniel Perout	GJarošeBO	−1	1	2.0
101.–103.	Tomáš Baják	ZŠ VBílovice	−1	2	1.0
	Rajmund Hruška	GPošKošice	3	2	1.0
	Martin Kolář	GÚstavníPH	3	1	1.0
104.–108.	Lenka Duongová	SvobChebŠ	−2	1	0.5
	Martin Mikšík	SPŠ Bo	1	1	0.5
	Filip Novotný	GJMasar_JI	0	1	0.5
	Petr Zahradník	GaSOŠ ÚL	1	1	0.5
	Petr Šícho	GKepleraPH	−2	1	0.5

KSP

Hlavní kategorie KSP



Zadání úloh KSP

První série

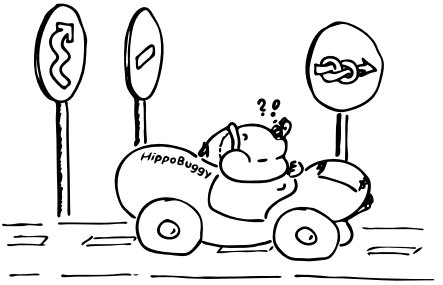
Vyhrocené sousedství

Je jasná, klidná letní noc. Téměř celé město spí, u řeky se prochází nevinné páry a na diskotékách vydrželi už jen ti největší tahouni. Silnice jsou prázdné. Jen občas na nich můžeme potkat zpívající skupinku, oslavující včerejší fotbalový úspěch, nebo taxík vezoucí ty, kdo už domů nezvládají dojít po svých.

Všechen klid tady kazí jen jeden zmateně pojíždějící motorkář a skupina policejních aut snažící se o jeho dopadení. Motorkář nejede nijak rychle, ale moc se nezajímá o protisměry, nemá boty, natož helmu, a už vůbec nehodlá zastavit. Hlídkám zdařile uniká a úspěšně objíždí i všechny zátarasy. Ale jen do chvíle, než dojede do slepé uličky, kde jej policie konečně dopadne.

28-1-1 Jízda na biomotorce — 10 bodů

Představte si, že jedete městem na motorce, jiné než té v příběhu: na biomotorce. Takové, která jezdí na pomeranče. Na jeden pomeranč je schopná ujet celých 100 metrů. Má to ale háček: pomeranče jsou velké a vejde se jich do nádrže jen 10. Naštěstí ale pomeranče ve městě jen tak rostou na stromech.



Mapa je tvořená křižovatkami a ulicemi, které je spojují. Můžeme si ji tedy představit jak neorientovaný ohodnocený graf, v němž navíc každý vrchol má daný počet pomerančů, které v něm rostou.

Napište program, který na vstupu dostane mapu města a najde trasu ze startu do cíle, během které co nejméněkrát projedeme ulicemi (tedy nás zajímá počet přejezdů a ne celková délka trasy). Během cesty můžete brát pomeranče z křižovatek. Nesmíte však překročit limit 10 pomerančů v nádrži. Křižovatky i ulice se mohou na trase opakovat.

Všechny sebrané pomeranče na křižovatce dorostou hned po jejím opuštění a dojetí na jinou křižovatku. Na začátku má motorka prázdnou nádrž, ale můžete ji naplnit pomeranči ze startovní křižovatky.

Zadání úloh KSP – 1. série

Formát vstupu: Na prvním řádku vstupu budou čísla N, M, S a C oddělená jednou mezerou, kde N je počet křižovatek ve městě, M počet ulic, S číslo startovní křižovatky a C číslo cílové křižovatky.

Na druhém řádku bude N mezerou oddělených čísel udávající počty pomerančů v jednotlivých křižovatkách. Na dalších M řádcích jsou popsány ulice, každá třemi čísly. První dvě udávají čísla křižovatek, mezi kterými ulice vede, a třetí udává délku ulice v metrech. Všechny ulice jsou obousměrné.

Pro hodnoty na vstupu dále platí:

- $1 \leq N \leq 30\,000$
- $1 \leq M \leq 1\,000\,000$
- Křižovatky jsou číslované od 1 do N.
- Na každé křižovatce leží maximálně 10 pomerančů.
- Ulice jsou dlouhé minimálně 100 a maximálně 1 000 metrů. Délky jsou násobky 100.

Formát výstupu: Na výstup vypište jedno celé číslo udávající délku nejkratší cesty v počtu projetých ulic ze startu do cíle, na které vám nikdy nedojdou pomeranče v nádrži.

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

Motorkář je mírně zakrvácený, vytřeštěný a stále opakuje, že musí utéct. Není mu to ale nic platné, je odveden na stanici a usazen na židli do rohu, kde čeká, než na něj přijde řada.

⋆ ⋆ ⋆

Tím motorkářem jsem já. Roztřeseně sedím, hledím do země a naslouchám okolnímu dění.

„Já nevim proč! Prostě najednou odněkud vyskočil, vyrval mi tu pochodeň z ruky, srazil mě na zem a zdrhnul někam do lesa!“ rozčiloval se muž na policejní stanici.

„A nemůžete prostě vzít kus klacku a vyrobit si novou? To nás musíte zdržovat takovýma prkotinama?“ reaguje znuděně policista.

„To nebyla jen tak nějaká pochodeň,“ brání se muž, „to byla speciální žonglovací pochodeň za tisíc korun!“

„Dobře, dobře. . . “ vzdává se policista, „tak to teda vezmeme znova od začátku.“ „Budeš zapisovat,“ říká kolegovi.

„Jak se incident odehrál?“ ptá se.

„Žongloval jsem v rámci fire show na louce u lesa. Najednou se vedle mě objevil chlap, vzal mi pochodeň, srazil mě na zem a zdrhl. To už jsem vám přece říkal!“ popisuje muž a praští naštvaně pěstí do stolu.

„Můžete pachatele nějak popsat?“ ptá se nenuceně dál policista.

„No… já jsem ho moc neviděl. Byla tma a soustředil jsem se na žonglování,“ vykládá muž, „takový normální chlap, o trochu vyšší než já a byl docela silný.“

„Skvělý! Napiš tam, že hledáme normálního muže vysokého asi 185 centimetrů, co chodí po městě s pochodní,“ vysmívá se policista, „proboha chlape, to na něm nebylo nic neobvyklého?“

„Ne,“ odpovídá muž, „vlastně… měl na ruce něco jako moderní hodinky. Takový technologický náramek – pořád to blikalo.“

Policista se pohrdavě otočí na kolegu: „Máš to?“

„Jo, mám,“ říká kolega.

„Tak děkujeme za nahlášení. V případě jakýchkoliv výsledků ve vašem případu vás ihned budeme informovat. Číslo na vás máme. Na shledanou!“ loučí se s mužem.

Muž se zvedl a trochu nesouhlasně odcházel. Asi tušil, že svou pochodeň již nikdy neuvidí. Jak by taky mohl, když dneska je problém najít ukradené auto, nebo třeba i kamion. A kamion se oproti pochodni sakra dobře hledá.

Ale vlastně ho trochu chápu. Hrátky s ohněm jsou fajn. Když mi bylo pět, tak jsme si s klukama ze sousedství tajně s ohněm hráli. Jen než mi jeden blbeček zapálil kraťasy a způsobil ošklivé popáleniny. Od té doby mám z ohně panickou hrůzu. Ono totiž utíkat před ohněm, který hoří přímo na vás, je čirá marnost.

28-1-2 Zapalování kostek — 8 bodů

Hrajeme následující hru. Máme postavenou pyramidu z dřevěných kostek o K patrech. To jest v prvním patře máme K kostek, na nich stojí $K-1$ kostek, na těch stojí $K-2$ kostek, až na špičce stojí jen jedna kostka.

Dva hráči se střídají v tazích. V jednom tahu hráč vybere jednu kostku v pyramidě a zapálí ji. To zapálí (obě) kostky, které na ní stojí, ty zas zapálí kostky, které stojí na nich a tak dále. Sousední kostky od aktuální nechytnou! Pak hraje druhý hráč. Vyhrává ten, kdo zapálí poslední kostku v pyramidě.

Pro dané K navrhněte vyhrávající strategii pro prvního hráče. To znamená takovou strategii, že ať druhý hráč dělá cokoliv, první vždy vyhraje.

⇧ **Lehčí varianta (za 3 body):** Popište konkrétní strategii pro $K = 4$ a $K = 5$.

„Tak teď vy,“ křikl na mě policista.

Hlídka mě odvádí k výslechovému stolu a sundává mi pouta. Rozklepaně pokládám ruce na stůl, sklopím hlavu a mlčím. Po chvíli se ozve policistův rozzářený hlas.

„Ale, ale… Vás jsme tady měli i včera, že jo? Nějaké sousedské problémy, jestli si správně pamatuji.“

„A-a-ano… S-s-soused mi-mi-mi z-zbořil m-můj ko-ko-komín.“

28-1-3 Bourání komínu I — 12 bodů

Na zahradě stojí V metrů vysoký komín, který chceme zbourat. K tomu máme k dispozici N bomb. Bomba i váží w_i kilogramů a zničí přesně d_i metrů komínu. Komín bouráme postupně odshora. Při boření vždy musíme vynést bombu až na vršek zbytku komínu a tam ji odpálit. Tím se komín sníží o d_i metrů (přitom nesmí vyjít záporná výška, nechceme skončit s jámou).

S nošením bomb se ale chceme co nejméně nadřít. Vynesení i-té bomby na komín vysoký x nás stojí $x \cdot w_i$ jednotek energie. Navrhněte algoritmus, který naplánuje boření komínů tak, abychom celkem použili nejmenší možné množství energie.

Lehčí varianta (za 7 bodů): Řešte případ, kdy všechny bomby dohromady zničí přesně V metrů. Tedy víme, že je všechny musíme použít.

28-1-4 Bourání komínu II — 8 bodů

Stejná úloha jako minulá, ale nyní ji řešte prakticky v CodExu. Stačí ovšem, když místo přesného postupu bourání budete vypisovat pouze minimální počet jednotek energie, kterou je nutno ke zbourání použít.

Formát vstupu: Na vstupu na prvním řádku dostanete dvě čísla V a N – výšku komínu a počet bomb. Na dalších N řádcích jsou vždy dvě čísla w_i a d_i udávající váhu a sílu bomby i. Dále platí:

- $1 \leq V \leq 10\,000$
- $1 \leq N \leq 10\,000$
- $1 \leq w_i \leq 100\,000$
- $1 \leq d_i \leq 10\,000$
- Pro několik prvních vstupů platí, že bomby dohromady zničí přesně V metrů.

Formát výstupu: Na výstup vypište jediné číslo udávající minimální námahu, se kterou je možno komín zbourat.

Ukázkový vstup:

```
12 3
4 6
2 2
12 4
```

Ukázkový výstup:

```
108
```

Nejdříve použijeme první bombu, stojí nás $12 \cdot 4 = 48$ jednotek energie, komín po ní bude mít výšku 6. Poté použijeme druhou, stát nás to bude $6 \cdot 2 = 12$, z komínu zbudou 4 metry. Nakonec třetí bombu, k použité energii přičteme $4 \cdot 12 = 48$, celkem jsme tedy využili 108 jednotek energie a z komínu nic nezbylo.

„Ano, čtu to tady: Pán nahlásil zboření svého výstavního komínu. Podezřívá z toho svého souseda kvůli dlouhodobým sporům a protože ten jej údajně neprávem obviňuje z otrávení svého psa... Hmmm... Koukám, že jsme panu sousedovi poslali předvolání. Tak nebojte, ono se to brzy vyřeší. Ale teď zpět k vám. Jaké vy jste dneska dělal problémy?“

„Jel na motorce jako blázen, opětovně ignoroval výzvy hlídek k zastavení a po dopadení blekotal nějaké nesmysly o útěku. Možná je pod vlivem drog,“ odpovídá rázně muž z hlídky.

„Tak se na to podíváme. Dělej zase zápis,“ ukazuje na kolegu.

„Proč jste ignoroval všechny výzvy k zastavení?“

„J-já se p-p-potřeboval co-co nejrychleji d-dostat sem... a-abych n-nahlásil, c-co se stalo.“

„Pokračujte,“ vybízí policista s náznakem zvědavosti.

„V-víte, já měl vždycky h-hroznou smůlu n-na své sousedy. Už v první třídě jsem seděl v l-lavici vedle kluka, který mi k-kradl svačiny a tr-trhal o-oblečení. T-to bylo vždycky d-doma problémů, že každý t-týden roztrhnu tričko. J-já ni-nikdy nepřiznal, že to dělá on. J-jsem se b-bál na něj ž-žalovat.“

„Proboha mluvte k věci. Mě zajímá dnešek a ne váš podělanej život.“

„A-ale t-to b-byla na-naprostá pr-prkotina o-oproti tomu, c-co se stalo t-teď,“ pokračuji, jako bych policistu vůbec neslyšel.

Začalo to hned včera, jak jsem vyšel z policejní stanice. Silně pršelo, tak jsem rozevřel deštník a vyrazil. Ze stanice to mám do práce kousek, jen pár bloků. Šlo se mi těžce, protože v ulicích foukal silný vítr a zápasil jsem s rozevřeným deštníkem. Nakonec jsem jej musel naklonit před sebe, abych jím prorážel vzduch. Tím jsem si zablokoval výhled. PRÁSK! Někdo přímo přede mnou vystupoval z auta a já, nevidě před sebe, mu kovovou špičkou deštníku vyryl do dveří pořádnou rýhu.

„Ty šmejde! Tys mi zničil auto! Nemůžeš se doprdele dívat na cestu?“ ozvalo se.

Nadzvedávám deštník, abych viděl, s kým mám tu čest, a mohl se dotyčnému omluvit. Stál tam asi padesátiletý pohublejší muž s vyholenou hlavou, který byl aspoň o pět čísel vyšší než já. Byl to můj soused.

„Ty?! Ty se ještě odvažuješ plést se mi do cesty, po tom, cos mi otrávil psa? To mi teda zaplatíš! To ti jen tak nedaruju!“ pokračuje.

„Já??? Já nikoho neotrávil. A nic vám platit nebudu! Tohle byla vaše chyba. Máte se koukat, jestli tam nejsou lidi, když otevíráte dveře,“ bráním se a s úšklebkem dodávám, „kromě toho vás v dohledné době přijdou vyšetřovat kvůli tomu komínu, který mi v noci někdo zbořil.“

Soused zasupěl a odměřeným zastrašujícím tónem řekl: „Ty mě jako budeš žalovat?“

„Jestli budu? Právě jsem to udělal, protože tohle vám už dál trpět nebudu!“

Soused zrudl, prohlídl si mě očima a potichu supěl: „Tos přehnal chlapečku. Mě tady nikdo žalovat nebude. Rozumíš? Nikdo! A už vůbec ne ty! . . . Ty o mě ještě uslyšíš.“

Najednou začal křičet: „TY JEŠTĚ POZNÁŠ, CO JÁ DOKÁŽU A NEBUDE SE TI TO LÍBIT! NA TO SE MŮŽEŠ SPOLEHNOUT!“

Přešel chodník, ukázal směrem ke mně výhružné gesto, agresivně trhl dveřmi a zmizel ve vedlejší budově.

Nevyděsilo mě to. Naopak mi to zvedlo náladu a s pocitem zadostiučinění jsem pokračoval v cestě do práce. Konečně dosáhnu spravedlnosti! V duchu jsem si představoval policejní důstojníky klepající na dveře souseda a jak mi přímo on dává do ruky peníze za způsobenou škodu. Nebo ještě líp, dostává příkaz k vystěhování.

S takto dobrou náladou jsem došel do práce – dokonce přestalo i pršet. To jsem ale ještě netušil, co mě později v ten den čeká. . .

* * *

Z práce jdu rovnou domů. Po průchodu brankou se ještě jednou smutně podívám na hromádku cihel, která mi zbyla po komínu. On to totiž nebyl jen tak obyčejný komín. Byl to umělecky navržený výstavní komín, který jsme vlastnili už po dvě generace a ke kterému se pojí nejedna vzpomínka. Například když do něj můj mladší bratr zapadl při hře na schovávanou a přes dvě hodiny jsme jej nemohli najít. Nebo méně veselá historka: jak se nám z něj šířila plíseň do vedlejších záhonů a museli jsme se jí celý víkend zbavovat různými chemikáliemi.

28-1-5 Likvidace plísně — 10 bodů

Na zahradě nám vyrostlo N hromádek mnohavrstvé plísně. Té bychom se chtěli co nejrychleji zbavit. K tomu máme plošný postřikovač naplněný přípravkem proti plísním. V jednom kroku můžeme buď použít postřikovač a zničit tak jednu vrstvu plísně z každé hromádky, nebo vzít několik vrstev z jedné hromádky a dát je na jinou (klidně i novou) hromádku. Hromádek takto můžeme vytvořit kolik chceme.

Napište program, který spočítá, v kolika nejméně krocích je možné se zbavit celé plísně.

Formát vstupu: Na prvním řádku vstupu dostanete číslo N udávající celkový počet hromádek plísně. Na druhém řádku dostanete čísla $v_1, \ldots, v_N$ udávající počet vrstev na jednotlivých hromádkách.

- $1 \leq N \leq 10\,000$
- $1 \leq v_i \leq 10\,000$

Formát výstupu: Na výstup vypište jedno celé číslo udávající minimální počet kroků, v jakém je možné se plísně zbavit.

Příklad: Máme-li tři hromádky o počtech 9, 3 a 3, je nejlepší nejdříve dvěma přesuny z první hromádky vyrobit tři po třech plísních, a pak všech pět třemi postřiky vyhubit.

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

Věnoval jsem hromádce cihel další, poslední pohled, když vtom mě popadla zlomyslnost. Proč já mám mít na zahrádce hromadu cihel a ten, kdo ji zničil, pořádek? Ať si soused taky trochu vychutná svou vlastní práci! Vzal jsem několik kusů a hodil je přes plot.

Jednu do růží. Jednu do okurek. Další mezi chryzantémy. Pak jednu do rajčat. Do umělého bazénu a poslední jsem zničil psí boudu. Stejně už toho blbýho čokla nemá. Haha! Takhle už jsem se dlouho nebavil!

Když jsem se vydováděl, šel jsem domů. Po namáhavém dni se natáhl na pohovku, pustil televizi a usnul. Spalo se mi krásně a klidně, ale nevydrželo to dlouho. Probudila mě obrovská rána, která šla z mojí kuchyně. Cože? Nebyl to sen? Nebyl, hned vzápětí se ozvala další, ještě větší!

Celý zmatený spadnu z pohovky na zem a snažím se vzpamatovat. Co se děje? Najednou oknem proletí velký, těžký předmět. Dopadne přímo do zapnuté televize a ta se celá rozpadne.

Ten magor mi hází do domu cihly! Další letěla přes pohovku a roztříštila skleněný stolek přímo přede mnou. Letící střepy mi pořezaly tvář, ruce a pravou nohu. Naštěstí jsem stihl zavřít oči.

Tohle už došlo moc daleko! Tady jde o život! Musím něco udělat dřív, než sem hodí další a stane se něco fatálního.

„Sousede! Dost! Chci se usmířit! Tohle už se nám vymklo z rukou!“ křičím.

Další rána, tentokrát z koupelny.

„Dost! To auto ti zaplatím! Zahradu taky! Jenom už mi prosimtě přestaň ničit barák.“

Nastala chvíle klidu. Z okna se ozve sousedův vyrovnaný hlas: „Ty si vážně myslíš, že penězi se dá něco urovnat po tom, co jsi proved’? Tak pojď ven a zkus to. Čekám na tebe.“ „A ať tě ani nenapadne volat pomoc, pár cihel mi tady pořád zbývá!“ dodal.

Zadání úloh KSP – 1. série

Seděl jsem na místě a přemýšlel, co mám dělat. Vzhledem k okolnostem mluvil soused až překvapivě klidně a to mě zneklidňovalo. Takto klidného ho neznám. Nakonec jsem se vyhrabal ze střepů a vydal se do ložnice pro peníze. Šel jsem opatrně, u zdi a při tom se pořád ohlížel po oknech. Vzal jsem peníze a zamířil na chodbu. Všude byl naprostý klid a po sousedovi žádné známky.

„Sousede, jsi tam?“ volám směrem ke dveřím.

Ticho, žádná odpověď. Že by bylo po všem? Že by si uvědomil váhu svých činů a šel domů? Radši se ještě podívám ven, abych měl jistotu.

„Jdu ven!“ volám.

Přistupuji ke dveřím, opatrně beru za kliku a pomalu otvírám. Rozhlížím se. Nikde jej nevidím. Pomalu a tiše dělám krok směrem ven. Druhý. Třetí.

Náhle mi ztuhnou ruce i nohy. Zvláštním způsobem mě zabrní celé tělo a padám na zem. Nemůžu se hnout. Jsem v jedné velké křeči. Nad sebou vidím stát souseda a pomalu se mi udělá temno před očima. Tak takový je to pocit... když vás někdo uzemní paralyzérem.

* * *

Ležím na tvrdé, nepohodlné, studené podložce a pomalu se probírám. Třeští mi hlava. Rozhlížím se a snažím se zjistit, kde to jsem. Jsem v potemnělé místnosti s jedním menším oknem, kterým prosvítá měsíční světlo. Ze všech stran kolem mě jsou mříže. Jsem zavřený v kleci.

Na zdech visí spousta různých středověkých nástrojů. Nůžky mnoha velikostí, kladivo zakončené hřebíky, řemdih, pouta... Na věšáku visí svěrací kazajka a v opačném rohu místnosti stojí... to je opravdu skřipec?

Jsem v mučírně. To nemůže být pravda! To je jenom sen! Hlava mi stále třeští, že nejsem schopný si ani kleknout. Tiše ležím na zemi a čekám, co se bude dít. Asi po půl hodině vchází soused.

„To je ale překvapení! Pán se nám konečně probral!“ zvolal s mírným nadšením v hlase. „To si spolu konečně můžeme užít trochu legrace,“ řekl a usmál se na mě. Pak hned zvážněl a zeptal se přísně: „Víš, proč jsi tady?“

Mlčím. Nejsem schopný slova. Vyčkávám. Najednou se mu v ruce objeví zbraň a střelí mě do paže.

„Na něco jsem se tě ptal!“

„Au!“ chytám se za paži. Nekrvácí. Ale nahmatávám pod kůží zarytou kuličku. Má airsoftku. A nepříjemně silnou!

„Neslyšíš, nebo co?“ střelí mě znova, tentokrát do břicha, „ptal jsem se, jestli víš, proč jsi tady!“

„Jo, tuším!“ chytám se za břicho a vzdychám, „asi kvůli vašemu autu a těm cihlám, co jsem k vám hodil. Omlouvám se! Všechno zaplatím! Jen už do mě prosim nestřílejte!“

„Špatně!“ zaburácel podrážděně a trefil mě do ramene, „za těma chryzantémama,“ rána do holeně, „na kterých ses vydováděl,“ rána pod lopatku, „si hrála moje vnučka!“ završil přímou ranou do čela.

Choulím se v bolestech na zemi. Nedokáži se natočit tak, aby mě nic nebolelo. On zatím odkládá zbraň. Vnučka? On měl nějakou vnučku? Ani jsem nevěděl, že měl ženu nebo děti.

„Dokážeš si představit, co se stane se čtyřletou holčičkou, když na ni dopadne cihla???“ dal hlavu co nejblíž mřížím a potichu a chladně pokračoval, „že nedokážeš? Tak počkej pár hodin a já ti to pomalu a velmi, velmi přesně ukážu.“

Odstoupil několik kroků, sedl si na židli a pozoroval mě svým chladným výrazem. Byl jsem šokován. Otřesen vším, co mi právě řekl, jsem zapomněl na všechny své bolesti a hlavou mi začala probíhat spousta nepříjemných otázek.

Opravdu měl vnučku? Opravdu bych ji přehlédl? Vždyť ta cihla dopadla přímo doprostřed záhonu, tam nikdo nemohl být! A nebo jen chci, aby to tak bylo, a ve skutečnosti jsem v rozrušení nedával pozor, kam co házím? Jak může mít vnučku v domě, kde má mučírnu? Jsme vůbec u něj doma? Jak dlouho jsem byl mimo po zásahu paralyzérem? Nevymyslel si svou vnučku, jen aby teď sledoval mě, užírajícího se pocitem viny? Ale co když si ji nevymyslel? Jak já s tím teď budu žít? Budu vůbec žít? Jak to myslel, že mi to pomalu a přesně ukáže? To mě plánuje něčím rozmáčknout?

Na žádnou z těch otázek jsem neuměl odpovědět a postupně padal do hlubšího a hlubšího pocitu agonie, který jsem doprovázel tichým sípáním. Soused mě neustále bedlivě sledoval. Pak se zvedl, přistoupil ke kleci a hodil mi nějaký ovladač s displejem a tlačítky.

„Na, vem si to. Sleduj displej. Když zasvítí zeleně, musíš zadat správné číslo.“

„Bav se. Musím si teď na tebe něco připravit,“ a odešel.

28-1-6 Úloha z ovladače — 12 bodů

Držíte v ruce ovladač, na jehož displeji se postupně objevují čísla. Jakmile displej zasvítí zeleně, musíte zadat nejmenší z posledních K čísel, které jste viděli. Toto číslo K je pevně dané.

Navrhněte datovou strukturu, která bude umět efektivně zpracovávat následující dvě operace: přidání prvku a vypsání nejmenšího z posledních K přidaných prvků.

Plný počet bodů můžete získat za řešení, které potřebuje průměrně konstantní čas na dotaz. Tedy některé operace struktury můžou být pomalejší, ale posloupnost D operací zabere čas $\mathcal{O}(D)$. Bonusové body můžete získat za řešení, které potřebuje konstantní čas na každý jeden dotaz.

Dělal jsem, jak řekl. Sledoval čísla na displeji. Najednou zasvítil zeleně. Rychle jsem něco zadal a potvrdil. V tu chvíli nade mnou něco zavrzalo a strop klece klesl asi o tři centimetry. Neee! Tak takovou já tady hraji hru. Takhle to myslel! Vždyť já vůbec nevim, co tam mam zadávat!

Během mého panikaření na displeji proběhlo dalších několik čísel a opět zezelenal. Zkusím nedělat nic, třeba to nezareaguje! Asi čtvrt minuty byl klid. Pak displej zablikal červeně a strop se znova snížil. Tentokrát asi o deset centimetrů.

To je ještě horší! Musím se teda snažit hrát. Vždy chvíli čekám a zkouším zadávat různá čísla. Nikdy jsem se netrefil a strop stále pomalu klesal. To je naprosto beznadějný, takhle nikdy nemám šanci nic uhodnout. A jde to vůbec? Nemá mi to jen dávat falešnou naději na moji záchranu? Padám do hluboké deprese a už prostě jen zadávám nějaká čísla, nevnímaje jaká.

Strop klece se již dostal tak nízko, že jsem si musel lehnout. Už jen pár poklesů a bude po všem. Už jen pár poklesů a nebude mě nic trápit. Strop už je tak blízko, že cítím narezlé železo, které ke mně stále klesá.

Hluboce dýchám. Snažím se každý další pokles nevnímat, ale nejde to. Zavřu oči a odhodím krabičku. Prosím, ať už to skončí! Já chci mít klid! Mříže naposled zavržou a zastaví se přesně ve výšce, že se jich má prsa při každém nádechu dotýkají. Dál se nic neděje. Otevřu oči. Mříže jsou až u mě. Mám tak málo místa, že se sotva můžu pohnout.

Najednou se za mnou zableskne a začnou se ozývat kroky. Už je zas tady! A má s sebou oheň. Já nesnášim oheň! Mám z něj panickou hrůzu! Vydávám vyděšené povzdechy. Představa upálení je pro mě naprosto zničující. To je ta nejhorší možná smrt, co může být!

Není to soused. Po místnosti se prochází neznámý muž asi po třicítce. V ruce drží pochodeň, rozhlíží se všude kolem a pečlivě si prohlíží předměty na zdech. Kdo to je? Je to snad otec sousedovy vnučky, který se mi taky přišel pomstít? Vybírá si teď vhodný mučící nástroj, kterým mi znepříjemní poslední hodiny života?

Sebral ze zdi dlouhé nůžky, kterými by se dalo ustřihnout i celé zápěstí a udělal pár kroků směrem k východu. Najednou se otočil a zadíval se na mě. Zastavil se mi dech. Vystrašeně se dívám jeho směrem a ležím bez nejmenší známky pohyblivosti. Vykročil směrem ke mně. Vyskočil na klec a skrčil se. Stále ty obrovské nůžky držel v ruce. Snažím se nemyslet na to, co s nimi hodlá dělat, ale nejde to. Furt dokola mi hlavou probíhá, jestli víc bolí ustřižené zápěstí, anebo prostřižené stehno.

Muž si nůžky strčil za opasek, koukl se na své moderní hodinky a něco na nich nastavil. Z hodinek vylezla dlouhá žhavá jehla. Tu vzal a začal s ní objíždět mříže. Mě se ani nedotkl, aspoň zatím. Dokončil okruh, škubl za mříže a vyrval je.

28-1-7 Vyříznutý kus mříže — 10 bodů

Ve čtvercové mřížce je nakreslený mnohoúhelník s N vrcholy, které jsou umístěny přesně v mřížových bodech. Navrhněte algoritmus, který na vstupu dostane souřadnice bodů mnohoúhelníka (v pořadí na obvodu) a spočítá, kolika mřížovými body prochází jeho strany.

Stále nehybně a vyčerpán ležím. Bojím se jakkoliv zareagovat. „Uteč!“ pošeptá mi, zvedne pochodeň a začne odcházet.

Nevěře, co se právě stalo, se pomalu zvedám a podezřívavě se dívám na muže. Jen aby to nebyla nějaká hra, dávající mi falešnou naději na svobodu. Vtom do dveří vejde soused.

„C-cože? Kdo jsi a co tady děláš?“ napůl překvapeně a napůl vyděšeně křičí na muže a začne jej ohrožovat pěstí.

Muž jej tvrdě odstrčí ke zdi a utíká ven. Soused hned vyrazí za ním a natahuje po něm ruku. Vtom za rohem zablýskne ostré světlo a soused začne neuvěřitelně řvát. Řval, jakoby mu tloukli hřebíky do kolen. Byl to takový úzkostný a dávivý řev, jaký jsem nikdy předtím neslyšel.

Já přestal na cokoliv čekat a zamířil k východu. Má záchrana bylo to jediné, co mě zajímalo.

Dobelhal jsem se do vedlejší místnosti. Tam se v rohu v bolestech svíjel zakrvácený soused. Chyběla mu půlka paže a z rány stříkala tmavě rudá krev. Prošel jsem místností, jak rychle to jen šlo, a začal hledat východ z domu.

Venku byla naprostá tma. U branky bylo zaparkované auto a vedle motorka. Po neznámém muži ani stopy. Bez rozmýšlení jsem sedl na motorku a ujížděl pryč. V hlavě mi zůstala jen jedna myšlenka a tou byl ÚTĚK!

Karel Tesař

Druhá série

Opojná vůně bankovek

„Dobrý oběd v restauraci.“ Co si pod touto frází představíte vy? Jistě se ve vašich myšlenkách objeví chutné jídlo, čisté stoly, cinkot příborů nebo přijatelná cena. Já, provozovatel restaurace v centru města, přitom cítím ještě něco navíc – pocit z dobře odvedené práce. Je jistě podobný tomu, co zažívá například programátor při doladění své nové aplikace, nebo dirigent, jenž posledním rozmáchlým gestem skončil symfonii a sklízí ovace diváků. A právě tímto pocitem začaly zvláštní události, o kterých vám chci vyprávět.

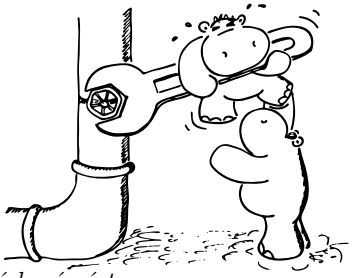
Mohly být tak dvě hodiny odpoledne, když jsem ve svém podniku od výčepu spokojeně sledoval hosty dojídající svůj oběd. Číšník odnášel prázdné talíře a obratem nosil na stoly dezerty. Většina lidí přišla ve skupinkách, jen blízko výčepu seděl osamoceně člověk s otevřenými novinami. Na zadní straně jsem si všiml reklamy na jakýsi katastrofický film.

28-2-1 Potopa ve městě — 10 bodů

Film se odehrává ve městě, jemuž hrozí katastrofa – mají přijít výjimečně silné deště. Město stojí na kopci a voda, která přeteče přes jeho okraj, bez problémů zmizí. Intenzivní srážky by však mohly poškodit budovy. Vědci naštěstí vymysleli způsob, jak zakrýt zdi domů, aby jimi voda neprošla. Potřebují však zjistit, jak bude celé zaplavení probíhat.

Dostanete popis města jako čtvercovou síť $N \times M$ a výšku budovy na každém poli (může být i nulová, třeba pokud se na něm nachází silnice). Voda, která nepřeteče přes okraj, je zadržena mezi budovami.

Ptáme se na celkový objem zadržené vody. Předpokládejte, že jí napršelo alespoň do výšky největší budovy.



Ukázkový vstup:

```
0430
0304
3223
0450
```

Ukázkový výstup:

```
5
```

Pro ukázkový příklad zůstane na „prostředních“ třech políčkách (dvě dvojky a pole nulové výšky nad pravou z nich) voda do výšky 3, vše ostatní odteče.

Dotyčný odložil noviny a já uviděl malého, shrbeného a celkem stydlivě vypadajícího muže. Mohlo mu být tak čtyřicet a byl pravidelným návštěvníkem. Všiml si, že ho sleduji, a plaše se ke mně otočil.

„Dobrý den... Jak se vám daří?“ zeptal se nejistým hlasem. „Jde to. Chutnalo vám?“ zajímalo mě.

„Jistě že ano,“ odpověděl rychle, jako by nad odpovědí nepřemýšlel a instinktivně ji vypustil z úst. Asi si to uvědomil a doplnil: „Á-ano, dnes mi chutnalo moc. Jistě, určitě.“

Byl to podivný člověk. Poprvé se tu objevil asi před dvěma měsíci a dlouhou dobu si sedal ke stolku blízko vchodu. Tvářil se, jako by chtěl mít jistotu, že může v případě potřeby rychle utéct. Až poslední dva týdny se rozhodl vyzkoušet pohodlnější místa a nakonec obsadil stůl blízko baru. Nic z toho mi nevadilo, ale jedna věc mi přišla zvláštní a nepříjemná: vypadalo to, že mě při jídle pokradmu sleduje a prohlíží. A dnes vypadal ještě nervózněji než kdy předtím.

„Nemám vám ještě něco přinést?“ s úsměvem jsem se zeptal. „Ne, já... stejně musím domů, mám nějakou práci s přerovnáváním své knihovny,“ řekl a nasadil křečovitý úsměv.

28-2-2 Řazení knih — 8 bodů

Mějme knihovnu. Je tvořena jednou dlouhou policí se spoustou knih, narovnaných těsně vedle sebe. Abychom v nich mohli vyhledávat, máme záznamy o knihách (reprezentovaných celými čísly) načtené v paměti počítače. Jsou uložené v poli a seřazené stejně jako odpovídající knihy v poličce.

Kvůli přerovnání knihovny jsme K knih z pravého konce vzali a přesunuli na levý konec. Vy máte za úkol provést změnu v záznamech v poli, aby odpovídala novému pořadí. Ale pozor, nemáte dostatek prostředků na vytvoření nového pole, změnu je třeba provést přímo v původní struktuře a smíte alokovat jen konstantně mnoho paměti navíc.

Příklad: Pro pole s hodnotami $(2, 3, 8, 7, 5, 1)$ a $K = 2$ je správným výsledkem $(5, 1, 2, 3, 8, 7)$.

„Vidíte, tohle mám na práci. Mimochodem, no, hm, jestli vás to zajímá, jmenuji se Konrád,“ představil se neznámý.

Konrád. To je ale jméno...

Usmál jsem se, také se představil a chystal se odejít do kuchyně, když vtom se ten člověk znovu ozval: „Já, vlastně, totiž, chtěl jsem se, víte, no, zeptat se...“

„Tak se vymáčkněte. Na co se chcete zeptat?“ pohlédnu na něj a v duchu si povzdechnu. Chce si na něco postěžovat? Tak ať to vybalí, já kvůli tomu nikoho nevyháním!

Chvilku se na mě dívá. Pak kývne hlavou. A pak tu otázku položí.

„Ronny. Ronny Tlouštík. Neznáte to jméno?“ vydechne.

Ale to ne! To jméno, zvuk toho jména proběhne celým mým tělem. To přeci ne! V mé mysli se rozvíří zapadlé vzpomínky a pocity překvapení. Vůbec si neuvědomuji, že na Konráda teď zírám s dokořán otevřenými ústy a zvednutým

obočím. Najednou nejsem majitelem restaurace, vracím se do zašedlé minulosti a znovu slyším ten hutný hlas —

* * *

Asi bych vám měl povědět o tom, čím jsem se kdysi živil.

Po letech strávených v rodném městečku jsem odjel studovat na vysokou školu. Ale do jejího výběru jsem až příliš nechal mluvit své rodiče. Sice jsem jakž takž plnil své povinnosti, ale obor mě vůbec nebavil. Většinu času jsem trávil osaměle, mrzutým procházením se po městě – až jsem se jednoho temného večera dostal do městské čtvrti, která neměla mezi obyvateli dobrou pověst.

Po hodině bloumání tmavými ulicemi jsem spatřil bar a řekl jsem si, že zakončím den skleničkou něčeho ostřejšího. Podnik byl umístěný ve sklepě omšelé budovy s popadanou omítkou, a proto mě překvapil čistý a kupodivu poměrně luxusně zařízený interiér. Jedinými hosty byla halasně oslavující skupinka mužů, sedící v rohu místnosti. Krátký pohled na nabídku nápojů prozradil, že se svým stavem financí si nemohu dovolit snad ani lahev vody.

„Bez peněz? Zvu vás!“ ozval se jasný, veselý hlas. Přede mnou najednou stál muž vysoké postavy, očividně silný, ale také tlustý. Hlavu měl plešatou. Byl oblečený ve stylovém bílém obleku a smál se od ucha k uchu.

„Oslavujeme!“ zvolal nadšeně a přivedl mě ke stolu. Jeho spolusedící, evidentně opilí, mě vzali mezi sebe. „Ronnymu se očividně líbíš,“ podotkl jeden z nich. „Možná by tě mohl pozvat na nějakou naši recesi,“ zasmál se druhý.

„Recesi?“ zeptal jsem se. Ale to už na stole přistál kalíšek whisky určený pro mě.

Oslava pokračovala, já zůstal a k jedné skleničce se přidala druhá, třetí. Ronny (o jeho nelichotivém přízvisku Tlouštík jsem se dozvěděl později) se mě vyptával na jméno, život, bydliště a neustále se smál. Když čas hodně pokročil, dal Ronny pokyn barmanovi a ten zamkl dveře.

„Ne abys mluvil o tom, co teď uvidíš,“ řekl. Neznělo to jako výhrůžka, ale jako nepsaná dohoda dvou kamarádů. Jeden z lidí na stůl přinesl dvě černé tašky, dosud nevinně stojící v koutě, a na stůl vytáhl jejich obsah.

Z jedné vypadla pistole s náboji. A ta druhá byla plná peněz, skutečných bankovek.

Tak tohle byla ta jejich recese. Sedím tady s kriminálníky, o kterých jsem předtím jen četl v novinách, a oslavuji s nimi jejich povedené vyloupení banky!

Ronny vzal několik bankovek vysoké hodnoty, srovnal je do úhledného balíčku a zeptal se: „Co bys řekl na to přidat se k nám? Co může člověk jako ty ztratit?“

Vrátil jsem se domů celý rozechvělý. Do rána jsem vystřízlivěl, ale opojení z peněz, jež ke mně takhle jednoduše doputovaly, zůstalo.

A tak jsem se brzy ke zločinecké partě připojil. Ronny, ač vypadal tak nevinně, byl jejich kápo a měl všude své kontakty. Setkání, kterému jsem byl přítomen, nedělali kvůli bezpečnosti příliš často. Zasílání zpráv všem členům obvykle probíhalo přes editora článků zaměstnaného v městských novinách.

28-2-3 Zprávy pro lupiče — 10 bodů

V novinovém článku jsou zakódované zprávy pro zločince. Určité slovo, představující zprávu, je do textu vložené jako vybraná podposloupnost – to znamená, že slovo získáme vybráním určitých znaků z textu článku (ale nesmíme změnit jejich pořadí). Celý systém je poměrně složitý a záleží i na tom, kolikrát je slovo do textu vloženo.

Obdržíte text článku a slovo a musíte najít všechny výskyty slova v článku. Pro jednoduchost předpokládáme, že se ve slově neopakují znaky. Na výstup vypište pozice písmen, které vyberete z článku a dají dohromady hledané slovo.

KSP

zadání

Ukázkový vstup:

```
11
aanubbcahoj
3
abc
```

Ukázkový výstup:

```
1 5 7
1 6 7
2 5 7
2 6 7
```

Netrvalo dlouho a ocitl jsem se v první akci. V nočních hodinách jsme vykrádali bankovní trezor. Ostatní členové se dostali dovnitř a já hlídal, zda se nikdo nepovolaný neblíží. Povedlo se a já s úsměvem poslouchal zprávy o bezradných vyšetřovatelích, když jsem si z tajného místa odnášel podíl z loupeže.

Pak jsem povýšil a učil jsem se, jak odemknout který zámek, jaký drát přestřihnout k vypnutí zabezpečovacího systému nebo jak správně omráčit nočního hlídače.

Odloučil jsem se od rodiny, pořídil si vlastní byt a užíval si toho, že mi stačí jednou za několik měsíců jít do nebezpečné akce a pak dlouho odpočívat a dělat, cokoliv se mi zlíbí. Ať už ve městě, nebo na ostrově v Karibiku. „Vydělaných" peněz přibývalo a ani jsem nevěděl, za co všechno je utratit.

Několikrát jsem se setkal s Ronnym. Vždy byl ve skvělé náladě, vždy byl skvěle oblečený a za celou dobu neshodil ani kilo. Věděl jsem, že je ve skutečnosti nekompromisní a dokáže být tvrdý ke svým nepřátelům. Ale říkal, že ve mě od samého začátku věřil a že se nemám ničeho bát. To, že by mě jen tak podrazil, mě nenapadlo ani ve chvíli, kdy trochu propadl megalomanii a začal plánovat vloupání se do několika bank současně.

28-2-4 Útěk z města — 10 bodů

Zločinecký gang plánuje velkou akci. Chtějí se simultánně vloupat do všech poboček banky ve městě, využít chaosu a utéct z města pryč do bezpečí. Poslední část není tak jednoduchá, jak se zdá: na místě, kterým proběhne lupič, se shromáždí policejní hlídky a přes ně už nikdo další neproběhne.

Město je reprezentováno čtvercovou sítí. Na každém poli je buď prázdné místo, jímž se dá proběhnout, nebo budova. Také máte zadané souřadnice jednotlivých poboček. Na každou pobočku připadá jeden zločinec.

Najděte pro každého z nich cestu z banky kamkoliv na okraj mapy tak, aby cesta vedla jen po průchozích polích a každé pole bylo použito maximálně jednou (nepočítejte s tím, že dva zločinci mohou jedním polem proběhnout současně – tak dobře se nemají šanci zkoordinovat). Není možné se pohybovat po diagonálách.

Já a ještě jeden člen gangu jsme dostali za úkol se postarat o centrální banku v samém srdci města. Nebezpečná akce začala tím, že jsme pronikli do honosné dvorany budovy. Odsud to bylo jen několik chodeb k lákavému obsahu, skrytému za kovovými dveřmi.

Samotné otevření trezoru a vyzvednutí peněz proběhlo až podezřele bez potíží. Teď bylo třeba utéct přes propojovací chodbu do vedlejší budovy. Vrátili jsme se do dvorany. Uprostřed rozlehlé, potemnělé místnosti se můj partner náhle zastavil. Z jeho tváře jsem vyčetl strach a zklamání. A já najednou cítil to samé.

„Je mi to líto," řekl. Sáhl po revolveru, ale pak ruku zase svěsil a potřásl hlavou. A takto celá melodramatická scéna skončila, protože těsně poté se ve dvoraně rozsvítila světla a já viděl početný zástup policistů, rozestoupených v rozích a mířících na nás zbraněmi.

* * *

Podivné období života s lehce vydělanými penězmi skončilo. Procitl jsem ze snu, uvědomil si, že ne všechno bylo takové, jak to vypadalo. Ale bylo pozdě. Ronny Tlouštík nás dva zradil a anonymně oznámil vloupání do centrální banky na policii. Policisté se na toto místo zaměřili a díky tomu měli lupiči z ostatních poboček volnou cestu k útěku.

Prošedivělý soudce mi oznámil dobu trestu a moje další cesta vedla do vězení. První rok v novém prostředí byl krušný. Těžko jsem si zvykal na stísněnou celu a osamělost. Vězeňští bachaři od nás vyžadovali naprostou poslušnost, ačkoliv sami byli poměrně vybíraví. Dlouhou dobu se například dohadovali, kdo bude hlídat který vězeňský blok.

28-2-5 Hlídání věznice 10 bodů


Věznice je rozdělena na mnoho menších bloků. Blok hlídá právě jeden bachař. Pracuje se na dvě směny, denní a noční, a každý hlídač pracuje v obou z nich.


Na začátku roku se rozpis hlídání mění a bachaři přišli se svými požadavky. Každý přinesl seznam svých oblíbených bloků, které je ochoten hlídat. A protože by se jen v jednom nudil, je třeba, aby blok přidělený ve dne a v noci byl odlišný.

Na základě požadavků přiřaďte každému bloku dva bachaře, z nichž jeden jej bude hlídat ve dne a druhý v noci. Nezapomeňte, že hlídač může v jednu chvíli střežit jen jeden blok.

KSP

zadání

Formát vstupu: Na prvním řádku dostanete čísla celá čísla B a P, udávající po řadě počet bloků/bachařů (musí být stejný, jinak by řešení neexistovalo) a počet preferencí. Následuje P řádků popisujících preference bachařů. Každý z nich obsahuje dvě čísla h_i a b_i ($0 \leq h_i, b_i \leq B - 1$) a znamená „bachař h_i je ochoten hlídat blok b_i“.

Platí $2 \leq B \leq 30\,000$ a $4 \leq P \leq 125\,000$, ale spousta vstupů je mnohem menší.

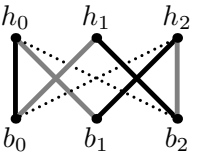
Formát výstupu: Na výstup vypište B řádků popisujících přiřazení jednotlivých bachařů. i-tý řádek popisuje i-tého bachaře a obsahuje dvě čísla d_i a n_i udávající po řadě blok, který bachař hlídá v denní a v noční směně (tedy záleží na pořadí těchto čísel).

Ukázkový vstup:

```
3 8
0 0
0 1
0 2
1 0
1 2
2 0
2 1
2 2
```

Ukázkový výstup:

```
0 1
2 0
1 2
```



Na obrázku silné čáry značí přiřazené směny (černé denní, šedé noční) a tečkované nepoužité preference. Pro jeden vstup existuje více správných výstupů. Například pokud všechny denní směny prohodíte za noční a naopak, dostanete opět platné řešení.

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

Vlastně vůbec nevím, co by se se mnou stalo, kdyby jednoho dne do mé cely nepřiřadili nového spoluvězně. Byl starší než já, očividně ve vězení strávil už

pěkně dlouhou dobu a vypadalo to, že tu zůstane ještě déle. O své minulosti nikdy moc nemluvil. Zdálo se mi však, že se smířil se svým osudem a že je z něho cítit neobyčejná vyrovnanost, jakou jsem předtím nikdy nezažil. I hlídači se k němu chovali s větší úctou než k ostatním.

„Já se odsud už nedostanu," říkal mi, „ale tebe jednou pustí, tak přemýšlej o tom, co budeš tam venku dělat!" Apeloval na mě, ať využiji každé možnosti pracovat a zjistit, jaká legální činnost mě na svobodě bude živit.

A tak jsem se dostal jako pomocník do vězeňské kantýny. Vaření mě začalo bavit a postupně jsem se zlepšoval. Stal jsem se kuchařem vařícím pro celé vězení a odsud byl jen krok k předčasnému propuštění za dobré chování. Personál kuchyně mi gratuloval a daroval mi nástroj, který jsem si oblíbil: velkou litinovou pánev.

Ale kam se mám teď vydat? přemýšlel jsem za branou věznice. K rodině jsem se jít styděl. Ke zločineckému gangu jsem se vrátit nemohl, i kdybych snad chtěl. Většina jeho členů byla zadržena při nepovedené loupeži, o kterou se pokusili, když jsem byl za mřížemi. Ronny Tlouštík byl za nespočetné množství loupeží a několik vražd odsouzen na doživotí a jeho sbírka dvaceti obleků pro každou příležitost se prodala v nějaké státní aukci.

Rozhodl jsem se jet někam, kde mě nikdo nezná, a odjel jsem do velkoměsta na druhé straně země. Zpočátku jsem tam zažíval krušné chvíle – než jsem si dokázal vydělat na nájem, nezbylo mi nic jiného, než přespávat v nočních linkách městské hromadné dopravy.

28-2-6 Cesta MHD — 12 bodů

Máme k dispozici kompletní jízdní řády všech vozidel městské hromadné dopravy. Trasa každého vozidla je popsána jako seznam dvojic zastávka-čas (předpokládáme, že čas odjezdu je stejný jako čas příjezdu). V zastávce lze mezi vozidly přestupovat, ale abychom měli jistotu, že vše stíháme, musí být mezi časem příjezdu prvního spoje a časem odjezdu druhého spoje rozdíl alespoň λ minut, jež také obdržíte na vstupu.

Najděte nejdelší možnou cestu v síti (délku měříme celkovým časem stráveným ve vozidle) při dodržení času na přestup.

⇧ **Lehčí varianta (za 6 bodů):** Řešte stejnou úlohu za předpokladu, že $\lambda = 0$.

A tím jsme se dostali až do současnosti. Vařením jsem se ve městě dokázal uživit, až jsem sehnal dostatek peněz na zakoupení své vlastní restaurace. A teď v té restauraci stojím a překvapeně poslouchám Konráda, který moji minulost zná...

* * *

Pomalu se vrátím do reality a vrhám na podivného hosta tázavý pohled.

„Já... pracuji v archivu...“ začal Konrád vysvětlovat. „Narazil jsem na... váš případ. Na slyšení před soudem... kdy jste vysvětloval, jak jste se prolomil do té poslední banky a otevřel trezor.“

Chvíli mi trvá, než si vzpomenu. „Jistě,“ odpovím. „Uzavřené pro veřejnost. Popisoval jsem chyby v jejich zabezpečení. Jak jste na to přišel?“ ptám se ho ostře.

Konrád se lekne mého zvýšeného tónu a znovu začne koktat. „Zvědavost, no, vždyť víte... V archivu, tam, tam se člověk nudí, začne, no, začne si číst... a pak nemůže přestat...“

Potřesu hlavou a ptám se: „A proč jste za mnou vlastně přišel?“

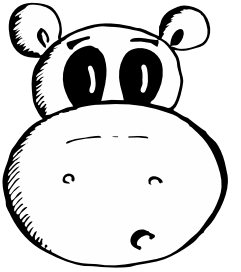
Odpovídá: „Můj blízký přítel v té bance... víte, pracuje tam. Je tam... říkal, že je tam pořád stejné zabezpečení... pořád... byste se tam uměl dostat.“

Je to, co říká, možné? Uvažuji o tom, ale pak mi dojde, co má na mysli. „Vy chcete, abych se tam vloupal znovu!“ šokovaně odpovídám.

„Ten... kamarád by nám s tím pomohl. Vy máte zkušenosti, on nám pomůže... a já to zorganizuji.“

Vyjeveně se na něj dívám. Ale najednou mi začnou do mysli pronikat vzpomínky. Bankovky, spousta bankovek a jejich typická vůně. Nadšení při každé povedené akci a každém otevřeném zámku. Slunné dny na Havaji, strávené nicněděláním. Podívám se po restauraci, kterou jsem si vlastníma rukama vybudoval, a najednou necítím žádný pocit z dobře odvedené práce.

„Přijďte sem později,“ zašeptám.



Je večer a já sedím v kuchyni svého podniku. Zbytek dne po rozhovoru jsem strávil celý nesvůj. Předtím jsem se nechtěl vrátit mezi kriminálníky, ale teď? Co se to se mnou vlastně děje? Z přemítání mě vyruší zaklepání na dveře. Je to Konrád. Tváří se napjatě a v ruce drží kufřík.

„Mám pro vás nějaké podklady,“ řekne mi. Položí kufřík na desku stolu. Na jeho boku se nachází displej a červeně na něm svítí tři čísla. „Nejnovější technika,“ usměje se a začne na malé klávesnici vedle displeje zadávat kód k otevření. Asi třikrát se splete, než trefí správné číslo.

28-2-7 Otevření kufříku — 10 bodů

Firma vyrábějící kufříky s přístupem na kód chce zajistit, aby číselné heslo nebylo konstantní. Typ kufříku, který používá Konrád, zobrazí na svém displeji několik čísel. Ty slouží jako parametry určité funkce a pro otevření je třeba zadat jejich výsledek.

Displej Konrádova kufříku zobrazí čísla A, B a K. Abyste kufřík otevřeli, zjistěte, kolik se mezi čísly A a B vyskytuje čísel, jejichž binární zápis obsahuje právě K jedniček.

Lehčí varianta (za 5 bodů): Předpokládejte, že $A = 0$ a $B = 2^n - 1, n \in \mathbb{N}$.

Zvědavě jsem nakoukl dovnitř. Na vrchu kufříku jsem uviděl několik složek. Vypadalo to, že obsahují nákresy vnitřních prostor banky a popis zabezpečení. To ale Konráda zatím nezajímalo. Odložil vrchní obsah na stranu a na dně kufru jsem spatřil – co je tohle za déjà vu? – ony zelené papírky, po kterých lidi tolik touží.

Na chvíli strnu a nasávám onu opojnou vůni peněz, jež mě zavedla do tolika problémů. Pak se podívám na Konráda, jenž má ve tváři tázavý výraz. „Jdete do toho?“

Nadechnu se a vztáhnu ruku po bankovkách. A v okamžiku, kdy se jich mám dotknout, se to stane.

Najednou slyším výkřik, pád a sypání krabic. Vyděsím se. Je tu někdo jiný! Odkud ten zvuk jde? Zaslechnu klení a další zvláštní zvuky. Vychází ze skladu, odděleného od kuchyně dveřmi. Než stihnu cokoliv udělat, dveře jsou s neskutečnou silou vyraženy a já se dívám do očí podivnému člověku. Má na sobě podivné černé oblečení a jeho tvář je rozezlená. Začne se rozhlížet po kuchyni.

„Tohle si vezmu,“ procedí mezi zuby a ze zdi něco sundá. Co to dělá? To je má památeční pánev, kterou jsem si odnesl z vězení! Chci mu ji vytrhnout z ruky, ale on rychle uskočí a pánví se po mně ožene. „Na tohle nemám čas,“ zamumlá a vyběhne přes restauraci ven do ulice.

Nemohu uvěřit tomu, co se právě stalo. Ohlížím se po Konrádovi, ale ten zbabělec je pryč. Včetně kufříku a jeho lákavého obsahu. Nevěřícně se dívám do ulice a najednou mi do duše padne příjemný klid.

⋆⋆⋆

Asi jste nečekali takový konec, že ano? Doufal jsem, že celá záležitost s tím mužem se nějak vysvětlí. Ale nevysvětlilo se nic. To, jak se dostal do skladu, zůstalo záhadou. Jediným vchodem do té místnosti, vyjma dveří do kuchyně, je mřížka ventilace. Přes ni by neproklouzl. Nebo že by přišel přes kuchyň ještě předtím, než jsem se tam dostal já? Ale co tam pohledával? Že by si chtěl něco odnést a pak se takhle hloupě prozradil? A proč potřeboval právě pánev. . . ?

Podobné otázky mě dlouho nenechávaly spát, a když jsem někdy večer zůstal sám doma, bál jsem se, co by na mě mohlo vylézt ze skříně. Alespoň že vím, že ten neznámý byl skutečně člověk.

Zem se slehla i po Konrádovi. Už nikdy nepřišel na oběd a ani jinde jsem ho nezahlédl. Zajímalo by mě, co se s tím mužem, jenž chtěl rychle přijít k penězům, vlastně stalo. Nemyslím si, že svůj plán někdy úspěšně dokonal. Chyběla mu jedna základní vlastnost každého zločince: drzost. Nejspíš zůstane pracovat v archivu a myšlenka na vloupání zůstane jenom snem.

Ale nenechte se mýlit: za to, co se nakonec stalo, jsem ve skutečnosti vděčný. Nechápu, jak jsem mohl uvažovat o tom, že bych se vrátil do světa zločinu. V každém případě, ukradení pánve vyvolalo vzpomínky na mého skvělého spoluvězně a na to, že ve mě věřil. Snad to bude dostatečná vzpruha do dalších let.

Pocit z dobře odvedené práce? Až se někdy půjdete najíst do mé restaurace a uvidíte mě stát u baru, zeptejte se, zda ho stále cítím. A kdybych se na vás tvářil rozpačitě a díval se po vaší peněžence? Pak mě raději rychle něčím přetáhněte.

Kuba Maroušek

Třetí série

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.

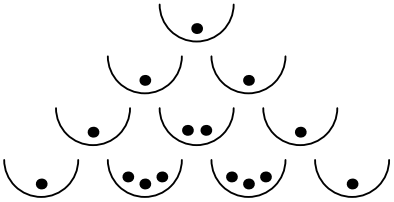
Tento den legie dopochodovala na planinu v lese, legát vyslal průzkumníky a zbytek se dal do stavby opevnění. Teprve před chvílí vztyčili poslední část dřevěného opevnění a Gaius unaveně padl na zem vedle ohně.

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[25] 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?

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

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
```

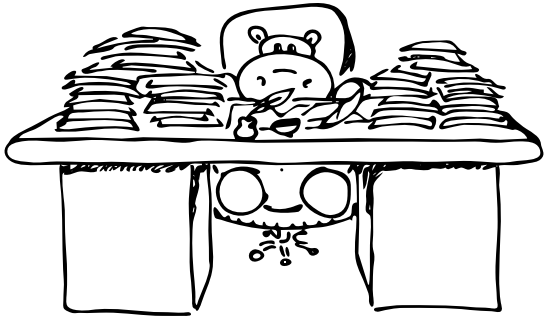
Ráno to ale bylo ještě horší...

„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 (například 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 ořínout (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:

```
6
5 11
1 1
5 6
1 2
6 4
4 6
```

Ukázkový výstup:

```
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 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.

Zadání úloh KSP – 3. série

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

Čtvrtá série

Příběh pěti domů

Utahuji poslední šroub, ještě pro jistotu naposled změřím napětí, balím si nářadí a vyrážím zpátky domů. Už se tu nechci zdržovat ani minutu. Je pátek po deváté večer a mě ještě doma čekají přípravy na víkend a taky jsem dětem slíbil pohádku.

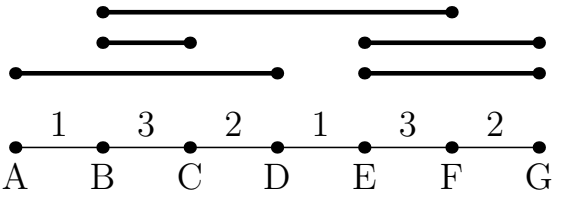
Pracuji jako technik pro jednu telefonní společnost. Práce to není špatná, docela mě i baví a hlavně mě v běžném životě příliš nezatěžuje. Akorát když v mém okruhu nastane náhlý výpadek, jako třeba teď, tak to musím hned jít opravit. Ale co nadělám, aspoň že se to tentokrát stalo přímo v našem bloku, tak jen stačí projít ulicí a jsem doma.

28-4-1 Sledování telefonů — 9 bodů

V ulici stojí v řadě N domů. Každý z nich má jeden telefon a je propojen telefonní linkou s dvěma sousedními domy (jedním v případě krajních). Grafově řečeno tvoří telefonní síť cestu: vrcholy představují domy a hrany spoje. Pokud zavoláme z domu a do b, musí hovor projít přes všechny spoje ležící na cestě mezi a a b.

U každého spoje víme, kolik přes něj za poslední týden prošlo hovorů. Na základě této informace bychom chtěli zjistit, kdo komu volal. To samozřejmě nelze určit jednoznačně, stejný provoz na spojích může vytvořit mnoho různých řešení. Vyberte to s nejmenším celkovým počtem hovorů (pokud je více takových, libovolné z nich).

Například pro $N = 7$ a počty hovorů na spojích postupně 1,3,2,1,3,2 muselo proběhnout nejméně pět hovorů a mohly vypadat například takto:



Tedy proběhly hovory A → D, B → C, B → F a E → G a znovu E → G.

S menším počtem hovorů nelze vytvořit zadané vytížení linek.

⇧ **Lehčí varianta (za 5 bodů):** Řešte za předpokladu, že se počty zachycených hovorů na libovolných dvou sousedních spojích liší maximálně o 10.

V mé ulici stojí pět domů a při takové večerní procházce si aspoň zas jednou udělám obrázek, jak se daří sousedům. Všichni zde žijeme už téměř deset let a k našemu nastěhování se váže společný příběh.

⋆⋆⋆

Bylo mi tehdy 25 let a o tomto roce můžu jednoznačně mluvit jako o roce smůly. Firma, ve které jsem byl zaměstnán, prodělávala, a tak musela čistit. Jako nezkušený mladík jsem to odnesl já a další práci dlouho nemohl sehnat. A aby toho nebylo málo, tak mi navíc o pár týdnů později vykradli byt a stopy zakryli tím, že ho prostě zapálili. Pachatele se dopadnout nepodařilo, vchodové kamery nikoho nezaznamenaly a ani nebyly patrné jakékoliv známky vniknutí. Prostě záhada a pojištěný jsem nebyl. Takže jsem najednou neměl ani práci, ani kde bydlet, a nápad, jak z toho ven, už vůbec ne.

Večer, když jsem svůj žal zapíjel v místním lokále, neměl jsem totiž kam jinam jít, si ke mně přisedl uhlazený muž v obleku a začal se vyptávat, co mě trápí. Stručně jsem mu popsal svou situaci, svěsil hlavu a zhrzele seděl dál. Opravdu jsem neměl náladu se s někým vykecávat.

Muž si na papír načmáral pár poznámek a pak povídal: „Pracuji pro společnost jménem Druhá šance a mám pro vás nabídku. S kolegy zrovna zakládáme nový projekt a vy byste pro nás byl ideální kandidát. Vaší účastí byste vyřešil všechny problémy s prací a bydlením, které vás momentálně sužují. Rozhodnutí je teď na vás. Pokud byste měl zájem se dozvědět více, zavolejte na toto číslo a domluvíme si schůzku. Moc ale neváhejte, naše prostředky jsou omezené a mohli bychom místo vás sehnat někoho jiného.“

Předal mi svou vizitku, rozloučil se a s úsměvem odešel. O nabídce jsem dlouze nepřemýšlel a hned ráno jsem volal, že přijímám. Schůzka byla další pondělí.

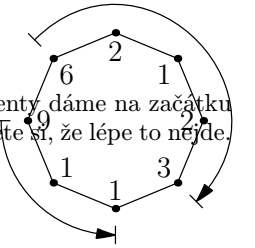
Přišel jsem na zadanou adresu a byl usazen do čekárny. Bylo nás tam celkem pět, tři muži a dvě ženy. Na pohled jsme všichni byli ve věku kolem pětadvaceti let. Z kanceláře vyšel muž, kterého jsem již znal, v ruce držel nějaké papíry a povídal: „Výborně! Tak jste tu všichni. Vybrali jsme do našeho projektu právě vás pět, protože si myslíme, že jste poslední dobou v životě neměli štěstí a potřebujete dostat svou »druhou šanci«. Tu vám můžeme nabídnout. Já si teď ještě rychle musím něco dopřipravit. Zatím si projděte a podepište tyto dokumenty a trochu se seznamte.“

28-4-2 Podepisování dokumentů — 8 bodů

V kruhu sedí N lidí, kterým potřebujeme předat dokumenty k podepsání. O každém víme přesně, jak dlouho mu bude trvat, než dokumenty zvládne projít. K dispozici máme dvě sady dokumentů, které dáme dvěma lidem, kteří v kruhu sedí vedle sebe. Ti pak dokumenty pošlou dále po kruhu. Navrhněte

algoritmus, který zjistí, kterým dvěma sousedům dokumenty podat, aby celkový čas procházení dokumentů všech lidí byl co nejmenší.

Například pro následující skupinku (čísla udávají dobu prohlížení v minutách):



je jedno z možných řešení naznačené šipkami (dokumenty dáme na začátku osobám s časy 6 a 9). Celkový čas bude 14 minut. Rozmyslete si, že lépe to nejde.

První žena byla bruneta oblečená v červených šatech a botách na vysokých podpatcích. Svou image měla doplněnou o černý módní klobouk, zlaté náušnice a náhrdélník. Nebyla vyloženě krásná, ale jak se takto vyšvihla, tak se rozhodně bylo na co koukat. Manžel se s ní prý rozvedl, protože dle jeho názoru zbytečně utrácela za nesmysly a sama toho moc nevydělala. Nakonec zmínila, že její manžel stejně byl jen takový blbeček, který ji nedokázal docenit, že si ji ani nezasloužil a že si určitě brzy najde lepšího.

Druhým byl pohublý muž, který pořád něco ťukal do tabletu a moc se s námi nechtěl bavit. Pracovával jako programátor, pak se ale nepohodl s nadřízenými, když chtěl všechny pracovní dny využívat home office a komunikovat pouze po chatu. S rodiči si také nerozumněl. Neměli totiž pochopení pro hraní online počítačových her a stravování pomocí objednávek přes internet. Prostě takový ajťák.

Třetí osobou byla rozcuchaná potetovaná slečna, oblečená v hipsterském stylu a vlasy obarvenými na tmavě zelenou. Dle svých slov žije pohodový život, ve kterém jí toho moc nechybí. Stálou práci nemá, ale když potřebuje, dopomáhá si brigádami, většinou roznášením letáků. Od rodičů odešla v sedmnácti, protože prý měli příliš oldschoolový pohled na svět a zbytečně ji omezovali. Sedí tu s námi, protože dostala nabídku a nové příležitosti neodmítá.

Čtvrtým přítomným byl muž, takový trochu podivín. Měl vystudovaná práva, sociologii a nyní začal chodit na ekonomku. V životě se dostal do slepé uličky, když přesáhl věk dvaceti šesti let a za svá studia musel začít platit. Neměl totiž z čeho. Navíc to byl vitarián, což také nevycházelo nejlevněji. Ve svém životě dodržoval přísná pravidla ohledně stravování, denního režimu, spánkového režimu a tak vůbec.

Pátý jsem byl já, muž s vyhořelým bytem, bez práce a s nulovou představou, jak tuto zoufalou situaci řešit. Jinak ale obyčejný chlap s ne příliš náročnou představou o životě. Chtěl jsem hlavně založit rodinu, o tu se postarat, najít si pár přátel a s těmi občas něco podniknout či v zimě vyjet na hory. Hlavně aby bylo pořád co dělat a za čím jít.

Po vyslechnutí příběhů všech přítomných bylo naprosto zjevné, že jsme všichni navzájem diametrálně odlišní a máme naprosto jinak seřazené své životní hodnoty.

28-4-3 Řazení životních hodnot — 10 bodů

Máme zadanou množinu X s N navzájem různými celými čísly a posloupnost $R_1, \ldots, R_{N-1}$ operátorů menší než (<) a větší než (>). Najděte uspořádání $x_1, \ldots, x_N$ čísel z množiny X takové, aby platilo

$$x_1 R_1 x_2 R_2 \cdots R_{N-1} x_N.$$

Formát vstupu: Na prvním řádku dostanete číslo N. Na druhém řádku najdete N čísel tvořících množinu X (v neurčeném pořadí) a na třetím $N-1$ znaků < nebo > bez mezer.

Formát výstupu: Jeden řádek obsahující čísla z X v takovém pořadí, že splňují zadané relace. Správných řešení může být víc, vypište libovolné z nich.

Ukázkový vstup:

```
6
42 2 3 8 1 5
>><><
```

Ukázkový výstup:

```
8 5 2 3 1 42
```

Výstup je správný, protože platí: $8 > 5 > 2 < 3 > 1 < 42$.

Po chvíli z kanceláře znova vyšel ulízlý muž v obleku. Vybral od nás podepsané dokumenty a povídal: „Jmenuji se Dalimír a mým úkolem je uvést vás do celého projektu a postarat se o všechny formality s ním spojené. Všem pěti z vás nějakým způsobem pomůžeme vyřešit váš aktuální problém s bydlením a financemi. . . “

„Heeej! Já nemám žádnej problém a určitě nepotřebuju ničí pomoc!“ ozvala se zelenovlasá hipsterka.

Dalimír ale pokračoval dál, jako by ji vůbec neslyšel: „Na kraji města v ulici Nádvorní jsme postavili pět domů a každý z nich se chystáme darovat jednomu z vás spolu ještě s dalšími výhodami. Jste připraveni se na ně jít podívat?“

Nezmohl jsem se ani na slovo. Tak úžasnou zprávu jsem vůbec nečekal. Ostatní na tom byli podobně. Vyhublý ajťák dokonce na chvíli přestal koukat do tabletu a dámě v klobouku v obličeji na okamžik zableskl výraz pokory a věku.

Po chvíli ticha se ozval věčný student: „A nebudeme mít pak problémy se zdaněním? Prošly ty domy oficiální kolaudací? A je legální v nich už bydlet?“

„Nebojte, o to vše jsme se postarali,“ uklidňoval jej Dalimír.

„Tak vyrazíme!“ pokračoval.

Dojeli jsme na místo a začali procházet ulicí. Dalimír nám vše pečlivě popisoval. Kde najdeme zastávky, kterým směrem je nejbližší obchod, a tak podobně. Až jsme došli k prvnímu, obrovskému domu.

„Nyní mi dovolte, abych vám představil první dům. Jedná se o tuto vilu se zahradou, bazénem a dvěma garážemi! Garáž samozřejmě není prázdná, ale najdete v ní nejnovější model auta Subaru XV Crosstrek. Samotný dům se pyšní dvěma

kuchyněmi, třemi koupelnami a ložnice pro hosty je samozřejmostí. Interiér je navíc zkrášlen řadou obrazů historického i moderního umění…“ představoval Dalimír.

28-4-4 Podivuhodný obraz — 12 bodů

V domě visí podivuhodný obraz. Je na něm nakreslených N bodů, které jsou spojeny dohromady M čarami. Celý obraz je černo-červený a má zajímavou vlastnost. Pokud z bodu vedou alespoň 2 čáry, některá z nich je černá a některá červená.

KSP

zadání

Co kdyby ale obraz vypadal jinak? Šel by pořád takto nakreslit? Vymyslete algoritmus, který na vstupu dostane neorientovaný graf a obarví jeho hrany dvěma barvami tak, že každý vrchol se dotýká hran obou barev (případně zjistěte, že to nejde). Pozor, vstupní graf nemusí být souvislý.

Lehčí varianta (za 8 bodů): Řešte za předpokladu, že všechny vrcholy mají sudé stupně.

Formát vstupu: Na prvním řádku dostanete dvě celá čísla N a M udávající počet vrcholů a hran. Každý z následujících řádků popisuje jednu hranu pomocí dvojice čísel vrcholů (vrcholy číslujeme od nuly).

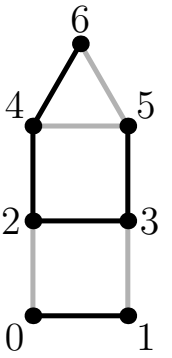
Formát výstupu: Vypište celkem M řádků popisujících barvy hran ve stejném pořadí, jako byly na vstupu. Barva každé hrany je buď 0 (černá), nebo 1 (červená, na obrázku šedá). Pokud graf nejde správně obarvit, vypište jediný řádek obsahující číslo -1.

Ukázkový vstup:

```
7 9
0 1
0 2
1 3
2 3
2 4
3 5
4 5
4 6
5 6
```

Ukázkový výstup:

```
0
1
1
0
0
0
1
0
1
```



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

„To zní úplně jako sen!“ vykřikla nadšením dáma v klobouku.

„A to ještě není všechno,“ pokračoval Dalimír, „starat se o takový dům není sranda. Proto také od nás máte k dispozici komorníka, uklízečku, zahradníka,

osobního šoféra a hlídače k vašim službám. Součástí je také finanční dar, který by měl stačit na veškerou údržbu na alespoň dalších pět až deset let.“

„A tento dům se všemi jeho výhodami jsme připravili... Chvíle napětí... Pro vás, slečno!“ ukázal na dámu v klobouku.

„To... to je naprosto úžasné,“ rozpačitě děkovala dáma, „určitě se budu snažit vás nezklamat a budu domu dělat dobré jméno. Určitě vám dokážu, že si jej zasloužím.“

„Ano, ano. Uvnitř na vás čeká komorník, který Vám vše ukáže,“ ukazuje na vchod Dalimír, „tak se běžte seznámit a my ostatní budeme pokračovat dál.“

Tak to je teda hustý, říkám si pro sebe. Jestli všechny domy budou takový, tak jsem totálně za vodou.

„Nyní přicházíme ke druhému domu,“ znova mluví Dalimír, „tento dům se pyšní těmi nejmodernějšími technologiemi, které doposud lidstvo vyvinulo. Nejen že budete mít k dispozici ty nejlepší počítače a další hračky, ale ještě k tomu vám každý rok budeme dodávat nové. Společnost vám budou dělat domácí roboti, kteří za vás vysají, automaticky ovládaní droni, kteří doletí ke dveřím pro poštovní zásilky, vyzvednou krabici s jídlem, a čeká na vás ještě fůra dalších technologických vychytávek. Navíc je všechno centrálně ovladatelné a synchronizované. Když na to přijde, tak celý den nebudete muset vylézt z postele, a přitom se o všechno zvládnete postarat. Vysokorychlostní internet a pokrytí Wi-Fi v celém areálu je samozřejmostí.“

„Tak to už se nemůžu dočkat, až se nastěhuju, to je přesně pro mě,“ ušklíbla se ironicky hipsterka. Všichni už jsme tušili, kdo bude novým vlastníkem domu. Vyhublému ajťákovi začaly svítit oči a pozorně poslouchal každé slovo jako do té doby nikdy.

Vtom se zablesklo a před domem se objevil urostlý muž s mečem v ruce. Na jeho rukou a nohou probleskovaly pruhy modrého světla. Pár vteřin tam stál, rozhlížel se, ale pak se zase zablesklo a byl fuč.

„A teď jste mohli vidět... To byla asi... ukázka automatického zastrašování pomocí holografické stráže,“ pokračoval Dalimír, „není to ale jediný bezpečnostní prvek, který zde je. Celý areál je pokrytý kamerami, které zevnitř můžete sledovat z libovolného přístroje a vstoupit můžete jen po identifikaci svou oční duhovkou.“

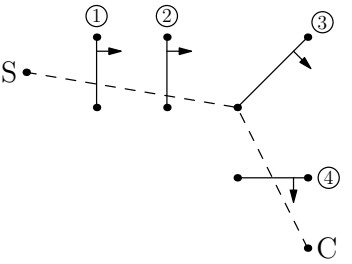
„Jak už asi tušíte, tak tento dům je přímo dělaný pro vás, pane,“ ukázal na ajťáka a ten se zaradoval: „Já... nemůžu se dočkat až si všechny tyhle super věci vyzkouším a pořádně nakonfiguruji! Tenhle dům je to nejlepší, co jsem zatím ve svém životě viděl!“

„Běžte ke vchodu. Tam vám kolega naskenuje duhovku a provede další inicializaci. My budeme pokračovat dál,“ povídá Dalimír.

„Nyní přicházíme ke třetímu domu. V tom se určitě nikdy nudit nebudete! Uvnitř najdete bowlingové dráhy, kulečník, saunu, minikino s vířivkou, venkovní párty bazén, minigolfové hřiště a kroketové hřiště. Ať budete chtít podniknout cokoliv, nikdy nebudete muset chodit moc daleko...“

28-4-5 Hra kroket 11 bodů

Ve hře kroket přesouváme míček po hřišti pomocí několika úderů holí. Míček považujme za bod nulové velikosti, který se po úderu pohybuje po úsečce. Celá trasa míčku tedy tvoří lomenou čáru. Na hřišti jsou také rozmístěné branky, které si budeme představovat jako úsečky. Cílem hry je, aby míček projel všechny branky, a to v předepsaném pořadí. Navíc každá branka má určený směr, kterým je třeba ji projet.



Na vstupu dostanete startovní a cílovou pozici míčku a seznam branek (každá je určena svými krajními body). Najděte nejkratší trasu míčku, která začně na startu, skončí v cílovém bodě a projede všechny branky správným směrem a ve správném pořadí.

Jednoduchý příklad vstupu naleznete na obrázku. Plnou čarou jsou značeny branky, šipky a čísla u nich udávají očekávané pořadí a směr průjezdu. Čárkovaná čára ukazuje správnou nejkratší trasu míčku.

Ještě dodáme, že je povoleno projet stejnou branku vícekrát, stačí, aby jeden z průjezdů dodržel správné pořadí a směr. Například je korektní projet branky v pořadí 1, 2, 4, 3, 4, protože když nepočítáme první průjezd čtvrtou brankou (který tím pádem může být i špatným směrem), dostaneme správné pořadí.

⇧ **Lehčí varianta (za 7 bodů):** Všechny branky jsou svislé, projíždějí se zleva doprava a jejich x-ové souřadnice tvoří rostoucí posloupnost (nejdřív je třeba projet nejlevější branku, potom druhou nejlevější, ..., až nakonec nejpravější). Start je nalevo od všech branek a cíl napravo. Jinými slovy, stačí hrát jen zleva doprava.

„... Součástí je také bar a taneční klub, který můžete použít jak pro soukromé účely, tak pro pořádání veřejných akcí. Personál včetně DJe od nás samozřejmě máte k dispozici. Pak vzadu můžete provozovat vlastní tetovací salón, vedle něhož najdete skleník, kde roste takové zelené překvapení,“ dokončuje vyčerpávající popis Dalimír.

„Tak počkat! Co myslíte tím zeleným překvapením? A vůbec mám takový pocit, že pořádání párty v obytné čtvrti je zakázané. Ty určitě budou narušovat noční klid...“ začal jej okamžitě napomínat student.

„Tak tím zeleným překvapením jsem samozřejmě myslel bio okurky a domácí zelí," vysvětluje Dalimír a při tom nepatrně mrkne na hipsterku, „co jste myslel vy? A o rušení nočního klidu nemějte obavy. Celý areál je vybaven moderním odhlučňovacím systémem, takže s tím by neměl být absolutně žádný problém."

Dalimír pokračuje: „A pro koho máme přichystaný tento dům? Ano, jste to vy, zelenovlasá dívko!"

Hipsterka radostně naběhla dovnitř a my ostatní pokračovali dál, k domu číslo čtyři.

„Nebudeme to dál napínat. Další dům je určen zde pro našeho věčného studenta a můžeme jej klidně nazývat Domem vědomostí," představuje čtvrtý dům Dalimír.

„Součástí tohoto domu je obrovská knihovna, ve které jsou k dispozici všechny studijní materiály, odborné články a literatura, které v posledních 50 letech lidstvo vyprodukovalo. K tomu je dodána přehledová brožura k bezproblémové orientaci v celém archivu. Navíc jsme Vám zařídili licenci pro sledování přednášek z pěti největších univerzit na světě. Kdyby Vám přeci jen něco chybělo, nebo vyšel nějaký nový článek, který byste chtěl, neváhejte nám říct a my jej pro vás obstaráme. Dále kolem domu máte dost prostoru pro pěstování ovoce, zeleniny, bylinek a obilovin, které byste k životu mohl potřebovat. Dodáme Vám ještě několik profesionálních ekologických zahradníků, kteří s Vámi na pěstování budou spolupracovat. Společně toho pravděpodobně vypěstujete více, než budete potřebovat, a jelikož pozemek patří Vám, tak vás přebytek nejspíš i uživí. A málem bych zapomněl, to auto, které tady stojí, je Váš nový elektromobil, šetrný k životnímu prostředí," dokončuje Dalimír.

„Tak to je naprosto boží! Tohle je doslova a do písmene můj sen! Živý, tady, přede mnou a ještě k tomu patří mně! Vůbec nevím, jak Vám mám poděkovat," povídá student, ale není na něm vidět žádná výraznější emoce.

„Jen mi neděkujte a raději se běžte zabydlet," posílá jej Dalimír dovnitř.

„Tak teď ještě vyřešit vás," říká směrem ke mně, „dům pro vás je ještě kousek dál po cestě."

Přicházíme ke znatelně menšímu domu, než byly všechny předchozí, který na první pohled vypadá naprosto obyčejně.

Dalimír začíná představovat: „Pro vás máme postavený klasický rodinný dům čtyři plus jedna. Oproti běžným tři plus jedna má navíc druhou ložnici, která se na takto odlehlém místě určitě bude hodit a případně se dá přestavět na dětský pokoj. K domu patří menší zahrada vhodná pro letní posezení venku. Jinak je to klasický byt. Kuchyň je vybavena sporákem a myčkou, obývák televizí a DVD. Také pro vás máme Volkswagena. Je sice trochu ojetý, ale zase málo žere ... A kdybyste měl zájem, tak vás doporučíme do jedné telefonní společnosti, kde zrovna shání zdatného technika. To jste vystudoval, je to tak?"

„Ano, je to tak," odpovídám.

„Tak to je asi všechno. Jak se Vám to líbí?“ dokončuje a ptá se Dalimír. Já na to rozpačitě odpovídám: „No... ano, líbí... Samozřejmě, že líbí. Ale... je tu jedna věc, která mi vrtá hlavou... Ostatní dostali výrazně větší a luxusnější domy a v podstatě se jim dneska splnily všechny sny, zatímco pro mě máte »jen« takovýhle »obyčejný dům«?“

Dalimír hned reaguje: „Rozumím. Tento dar od nás nemusíte přijmout. Pořád máte možnost jej odmítnout. Jsem si celkem jistý, že získáním tohoto domu budete ve značně lepší situaci, než jste byl před týdnem...“

„Ano, ano. Já si určitě nechci stěžovat. Jsem samozřejmě nadšen a vaši nabídku přijímám všemi deseti! Jen mě trochu zaskočil ten nepoměr vůči ostatním. Ale už raději mlčím,“ vymlouvám se.

„Tak to bude nejlepší,“ říká Dalimír, „já bych si dokonce vsadil, že se svým novým obydlím budete spokojený. A kdo ví, jestli ne dokonce víc než ostatní? Tak se běžte dovnitř trochu porozhlédnout, já teď ještě musím dořešit pár věcí a trochu si to všechno utřídit.“

28-4-6 Mediánové třídění — 9 bodů

Máme dohromady N čísel k setřídění. Také máme takovou speciální krabičku. Té na začátku zadáme fixní liché K a pak do ní začneme postupně vkládat nějaké prvky. Po každém vloženém prvku nám krabička řekne medián z posledních K vložených prvků, tedy takový prvek, který by se po jejich setřídění nacházel na prostředním místě. Krabička začne vracet mediány teprve poté, co do ní vložíme alespoň K prvků.

Například pokud pro $K = 3$ do krabičky vložíme postupně prvky 4, 7, 1, 2, 3, 5, řekne nám hodnoty $\emptyset$, $\emptyset$, 4, 2, 2, 3 (kde $\emptyset$ značí prázdný výsledek, když krabička obsahuje méně než K prvků). Například první dvojka je mediánem posloupnosti 7, 1, 2.

Vymyslete, jak s pomocí takovéto krabičky setřídit čísla v linárním čase. K si můžete zvolit libovolné, klidně pro každý vstup jiné. Předpokládejte, že přidání prvku do krabičky trvá konstantní čas.

Po tom dni a podepsání všech smluv jsem Dalimíra už nikdy neviděl a o Druhé šanci nikdy neslyšel.

⋆⋆⋆

Od té doby se u nás v ulici ledacos změnilo. Teď když procházím kolem prvního domu, tak z něj věčně slyším křik a hádky. Dáma už se za tu dobu stíhá potřetí rozvádět. Nikdo pro ní není dost dobrý. Co na tom, že vlastní úspěšný salón krásy a večeří kaviár? Co na tom, že vyhrává všechny soudní spory o majetek a děti, když žádnému z nich nemůže dopřát pocit fungující rodiny a namísto mateřského objetí jim pronajímá chůvu, aby se mohla naplno věnovat svému účesu a nehtům. Tato dáma dostala před deseti lety neuvěřitelné možnosti. Bohužel se

ale snažila urvat více, než dokázala sama unést. Dělala spoustu věcí, které nebyly zrovna správné, ale dělala je prostě jen proto, že si je mohla dovolit. Podle mě teď určitě nevede život, jaký si předtím vysnila.

Blížím se ke druhému domu. U něj je zaparkovaná blikající sanitka. Už zase... Vyhublý ajťák se totiž všemi technologiemi a vymoženostmi nechal natolik unést, že jen zřídkakdy vycházel ven. Dnešní svět to bohužel umožňuje. Když se nad tím zamyslíte, tak už vlastně neexistuje mnoho věcí, které by se nedaly zařídit online. Jeho život byl naprosto oddán virtuální realitě a moderním technologiím. Bohužel virtuálnímu světu dával výraznou přednost před tím fyzickým a ignoroval potřeby svého vlastního těla. A tak začal trpět ochabnutím svalů, záněty zápěstí a nakonec se mu začala bortit páteř.

Před necelým rokem ochrnul na dolní polovinu těla, když upadl na zem po cestě na záchod. Zachránili jej až kamarádi z online hry, když se do ní dva dny po sobě nepřipojil a ani o sobě nedal vědět. Dostat se pak k němu a poskytnout mu lékařskou pomoc byl také problém. Projít přes všechna technologická zabezpečení, která v domě měl, by byl úkol minimálně pro CIA. Ještě že měl jen obyčejná skleněná okna.

Po tomto incidentu mu byl přidělen osobní pečovatel. Ten se následně stal jeho nejbližším přítelem ve fyzickém světě za všechny ty roky. Vlastně byl zároveň jediným člověkem, který trávil v jeho přítomnosti více jak deset minut denně.

Když jsem míjel jeho dům, zrovna se sanitka rozhoukala a začala odjíždět.

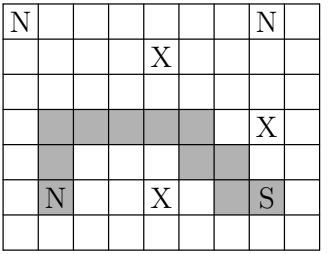
28-4-7 Jízda sanitkou — 11 bodů

Sanitka má naloženého stabilizovaného pacienta a potřebuje jej bezpečně odvézt z jeho domu do některé z nemocnic. Ve městě ale probíhají na různých místech opravy, v jejichž okolí to nebezpečně drncá a navíc poblíž nich hrozí uvíznutí v zácpě. Proto by se řidič rád držel od míst oprav co nejdál.

Máte zadanou mapu města jako čtvercovou mřížku s vyznačenou počáteční pozicí, pozicemi nemocnic a pozicemi všech míst, kde probíhají opravy. Najděte největší K takové, že existuje cesta ze startu do nějaké nemocnice, která se po celou dobu drží ve vzdálenosti alespoň K od všech míst oprav. Zároveň také najděte libovolnou cestu splňující toto omezení.

Po mapě se pohybujeme pouze vodorovně a svisle. Vzdálenost měříme v manhattanské metrice, tedy jako součet rozdílů x-ových a y-ových souřadnic. Například políčka (1,2) a (5,3) mají vzdálenost $4+1 = 5$.

Pro následující mapu (S značí start, N nemocnici a X místo oprav) je největší $K = 2$ a jedna z možných optimálních cest je vyznačena šedě. Snadno nahlédnete, že pro $K = 3$ žádná cesta neexistuje.



Ve třetím domě byl klid. Nedá se to vůbec srovnávat s tím, co se tam dělo před pěti až deseti lety. Bývala tam divoká párty minimálně každý druhý den, která se bez výjimky protahovala minimálně do pozdního rána dalšího dne. Já sám jsem se tam také dvakrát vydal a můžu říct, že to bylo super! Pak jsem ale další dva dny téměř jen ležel, spal a střízlivěl. Takovouhle akci bych dokázal přežít maximálně dvakrát měsíčně, tak je pro mě naprosto nepochopitelné, že to ta zelenovlasá, dnes už černovlasá, hipsterka zvládla táhnout přes čtyři roky v kuse zhruba čtyřikrát týdně.

Celou dobu to vypadalo, že si vše maximálně užívá. Pak ale jednou z ničeho nic, když se párty zase dobře rozjela, proskočila zavřeným oknem a dopadla přímo do bazénu. Nic vážného se jí nestalo a nikdo přesně neví, proč to vlastně udělala. Nikdo jí v té době nebyl dost blízký, aby dokázal něco tušit, natož pak předpovídat. Z tohoto incidentu se ale nedokázala vzpamatovat a další skoro dva roky strávila na psychiatrickém oddělení. Asi to holka s tou volností, spontánností a nevázaností přehnala. Žila naplno každý moment a jen pro ten moment. Už si ale nenašla chvilku, aby se sama ohlédla odkud, kam, za čím, pro koho a proč jde. A tak se jí tyhle nevyřešené a odkládané pocity hromadily tak dlouho, až to najednou vybouchlo.

Dneska ale žije docela jiný život. Našla si stálého přítele a společně přízemí jejího domu přestavěli z nočního klubu na moderní školku. Jsme spolu přátelé a občas se navštěvujeme. Myslím si, že navzdory svému divokému mládí je to celkem fajn a rozumná holka.

Teď už se blížím ke čtvrtému domu. Tam se toho za těch deset let moc nezměnilo. Předtím tam žil podivný věčný student, jehož život byl řízen přísnými pravidly. Dnes tam žije podivný věčný student, jehož život je řízen ještě přísnějšími pravidly. Akorát nyní má svou sbírku titulů obohacenou o mnohé další obory. Každý den vstává a chodí spát ve stejnou dobu, má pečlivě navržený jídelníček, třikrát týdně cvičí a vůbec v jeho životě není žádná nepravidelnost. Namísto kamarádů má pouze spolupracovníky a spolubadatele.

Ze začátku jsem jej párkrát zkusil někam pozvat, ale v jeho rozvrhu se těžko dá hledat minuta, která by nebyla zabraná v rámci jeho složitého týdenního režimu. Teď o něm absolutně nedokáži říct, zda je v životě šťastný a zda má to, co

hledal. Nerozumím mu, nemáme si spolu co říct, emoce na sobě nedává znát, ale třeba je tak šťastný.

Teď už se konečně blížím ke svému domu. Tam na mě čeká manželka a mé dvě krásné děti, pro které jsem na zahradě v posledním roce postavil malé dětské hřiště. Myslím, že teď žiju spokojený život, ve kterém ještě stále jdu za svým cílem. Posledních deset let pro mě nebylo vždy jednoduchých. Musel jsem se hodně snažit v práci a dělat přesčasy, abych získal vyšší kvalifikaci a zvedli mi plat. Doma nám zas nedávno prasklo potrubí a děti jsme museli složitě dovážet k babičkám a vůbec celkem často musíme řešit takové náhlé problémy. Ale zatím jsme to vždy všechno nějak zvládli.

Jsem opravdu rád, že mi kdysi byl přidělen právě tento dům a ne žádný jiný. Jsem opravdu spokojený. V tom se tehdy Dalimír trefil.

Všem mým sousedům se v ten den splnil jejich sen a začali si hlavně užívat a po nějaké době se toho přesytili. Nikdy v životě se nenaučili, jak o věci v životě bojovat a jak se o své cíle snažit. Já oproti nim dostal jen podmínky, které mi umožňovaly si za svými cíli jít a snažit se je plnit. Samotný fakt, že se mi to daří a postupně za mnou jsou vidět výsledky, mě naplňuje opravdovým štěstím. Takovým tím pocitem zadostiučinění, který moji sousedé dlouho nebo dokonce nikdy neměli šanci poznat.

Oni do náruče dostali svůj cíl a v tom okamžiku se přestali zajímat o jakoukoliv cestu, která by je někam mohla dovést. Já jsem dostal možnost stavět si svou vlastní cestu a jít po ní. A to je přesně ono, protože právě ta cesta je můj cíl! Ano, je to tak: „Cesta je cíl!“

Příběh pro vás napsal

Karel Tesař

Pátá série

Tento příběh završuje příběhy uplynulých sérií. Pokud vám něco nebude dávat smysl, přečtěte si minulé díly :-)

* * *

Will Knight, vedoucí výzkumník časoprostorového výzkumu, se vrávoravě sebral se země a oprášil ze sebe saze. Ještě než k němu skrz zalehlé uši dolehlo houkání sirén, viděl, jak všechno kolem zběsile bliká. To nebude dobré, pomyslel si.

„Wille, jsi v pohodě? Wille?!?“ ozýval se neodbytně hlas z interkomu. Will hrábl po tlačítku a přitom se rozhlížel okolo. To, co si v dalších minutách poskládal ze svých vzpomínek, ze stavu okolí a z toho, co mu řekli z druhé strany spojení, nebylo vůbec dobré.

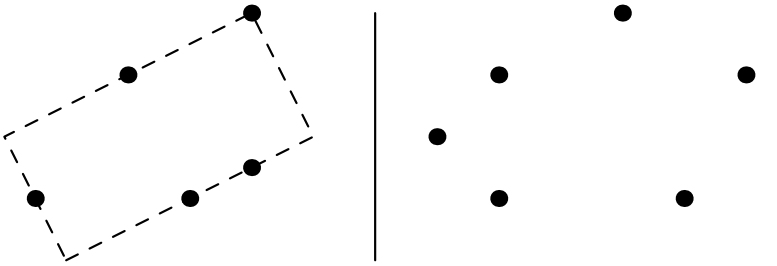
Tohle měl být první experiment, kdy časem pošlou člověka, konkrétně Willa. Zatím vyslali jen několik automatických sond. Ale asi selhal časoprostorový generátor a jádro generátoru, ve kterém se teď Will nacházel, se už vůbec nenacházelo ve stejném čase jako zbytek jeho týmu. Nikdo vlastně nevěděl, kde (a kdy) se ocitlo. Jediné, co měli k dispozici, byly údaje z několika časových majáků ve stěnách generátoru a spojení přes časoprostorový interkom.

28-5-1 Zaměřování místnosti — 9 bodů

Výzkumný tým získává souřadnice od časoprostorových majáků instalovaných ve stěnách místnosti posunuté v čase. Bohužel přijímají více sad údajů a potřebují rychle odlišit, které sady souřadnic jsou chybné a které by skutečně mohly označovat místnost.

Výzkumníci ví, že místnost má obdélníkový tvar a majáky jsou instalované v jejích stěnách. Od vás by potřebovali pomoc s tím, jak rychle určit, jestli všechny dané body leží na nějakém (jakkoliv otočeném) obdélníku, nebo ne. Souřadnice bodů jsou libovolná reálná čísla, zaokrouhlování neřešte.

Například pro body $[4, 4]$, $[2, 3]$, $[3, 1]$, $[0.5, 1]$ a $[4, 1.5]$ na levém obrázku takový obdélník nalezneme, pro body $[0, 2]$, $[1, 1]$, $[4, 1]$, $[1, 3]$, $[3, 4]$ a $[5, 3]$ napravo ne.



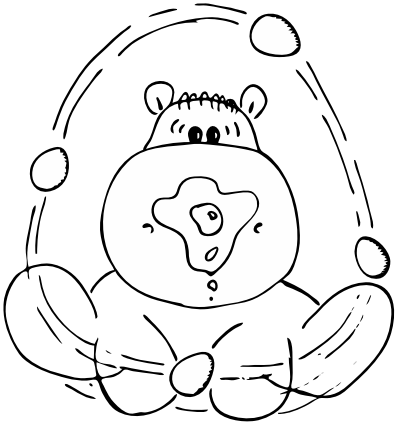
Výzkumníkům se sice povedlo zjistit, že je místnost jenom fázově posunutá a leží na okraji otevřeného časového víru, ale v záchraně Willa se nedostali o nic dál. Navíc skomírající generátor už dlouho časový vír pod kontrolou neudrží a nikdo nevěděl, jak velké škody v čase by jeho vymknutí se kontrole mohlo způsobit.

Unikající chladící kapalina z generátoru tvořila stále silnější a silnější ledový povlak na jedné z jeho částí a krystal dilithia v jádru byl nebezpečně popraskaný. Na opravu by stačil jakýkoliv dostatečně velký dilithiový krystal, problém tkvěl v tom, že k Willovi nemohli nic dostat. Ale jeden z výzkumníků dostal skvělý nápad – Willovi tak skvělý nepřipadal – a to, aby si Will obstaral potřebné věci v minulosti.

A tak se stalo, že se Will znovu postavil na plošinu ve středu generátoru, zadoufal, že krystal tuhle jednu cestu ještě zvládne, a stiskl tlačítko ovládání na své levé ruce.

* * *

Se zábleskem se zjevil na nějaké louce. Byla mu nepříjemná zima a netušil, kde se ocitl. Kus vedle ale žongloval nějaký člověk s pochodněmi a Will se rozhodl, že si nějakou vypůjčí. Vyhlédl si jednu opuštěnou a rozběhl se pro ni. Ale zrovna v tu chvíli se chlapík otočil a začal ji brát do ruky. Will ho v rozeběhu odstrčil, pochodeň popadl a nabral to směrem k nejbližšímu lesu.



Chlapík se za ním rozeběhl a byl asi lepší běžec než Will. Pomalu ho začal dohánět, a tak Will za běhu hrábl po minipočítači na ruce. Skok časem dělat nechtěl, ale několika místními přesuny by ho setřást mohl. Stejně je chtěl použít k prozkoumání většího území.

28-5-2 Místní přesuny 11 bodů

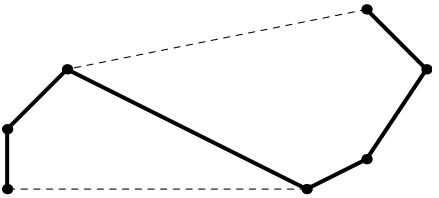
S pomocí lokálního transportéru se může Will přemisťovat mezi libovolnými dvěma body, které ho zajímají. Při průzkumu okolí by rád navštívil N bodů na mapě nacházejících se v konvexní poloze – to jsou takové body, které můžeme všechny umístit na obvod nějakého konvexního mnohoúhelníku, neboli napnout kolem nich gumičku tak, aby všechny body obepínala zvenku.

KSP

zadání

Na přesun mezi dvěma body spotřebuje Will tolik energie, jak jsou body od sebe na mapě daleko (počítáme vzdálenost vzdušnou čarou). Chtěl by začít v libovolném bodě, navštívit všechny ostatní body a spotřebovat přitom co nejméně energie.

Vymyslete algoritmus, který mu pomůže najít výše popsanou cestu. Níže můžete vidět ukázku bodů v konvexní poloze a nejlepší nalezené cesty mezi nimi (plná čára značí cestu, čárkovaná napnutou gumičku).



Po setřesení chlapíka udělal Will ještě pár skoků po okolí, než zaměřil svoji časoprostorovou pozici, a posledním skokem se ocitl v nějakém divném sklepě.

„Dílna, skvělé!“ zamumlal si potichu a šel se podívat, jestli by se mu něco nehodilo. Cestou si všiml divné klece v temném rohu, ale nevěnoval jí pozornost. Našel kladívko, krásné nůžky na plech a chtěl už odcházet, když mu ale něco nedalo, otočil se a pozorněji se podíval na klec. Uvnitř byl muž, k smrti vystrašený muž!

Will vyskočil na nízkou klec, z náramkového počítače vytáhl žhavou krájecí jehlu a začal zpracovávat mříže. Nakonec za ně trhl a vyrval je.

V tom zaslechl na schodech kroky a rozhodl se rychle zmizet. Otočil se k muži v kleci, řekl jenom „Uteč!“ a sám se rychle vydal s pochodní v ruce ke schodům, zadávaje přitom sekvenci k časovému přesunu na minipočítači.

Těsně, než došel ke dveřím, vyšel z nich druhý muž. Teď už se přesunu nedalo zabránit a Will muži nechtěl ublížit, tak se ho pokusil odstrčit do bezpečné vzdálenosti a vyběhnout z domu. V půlce schodů po něm muž natáhl ruku, ale přesně v tu chvíli se obvody nabily a okolo Willa vyrostla modrá koule...

***

Objevil se v nějaké jeskyni, ale přesunem neprošel sám. Před něj dopadla na zem zuhelnatělá lidská ruka. „Sakra!“ zaklel a pak si všiml, že v jeskyni je ještě nějaký římský voják, potlučený, s potrhanou zbrojí a beze zbraně.

Potom ale jeho zájem přilákal geologický senzor za pasem, který se rozblikal. V okolí se nachází velký dilithiový krystal, přesně to, co hledal! Začal obcházet okolní zdi, než objevil pořádně velký kus. Ten s vítězoslavným rozmachem kladívka vyloupl, sebral ho, hodil ještě jeden pohled po zkoprnělém Římanovi a stiskem návratového tlačítka se přesunul zpět ke generátoru.

⋆⋆⋆

S pomocí pochodně rozmrazil zamrzlé potrubí, nůžkami na plech se mu povedlo uvolnit vzpříčené kusy kovu nad nouzovým ventilem a tím pak zastavil únik chladící kapaliny. Nakonec vyměnil krystal v jádru, ten původní už byl skoro rozpadlý na prach, a spojil se interkomem s ostatními.

Willovo nadšení bylo ale vzápětí zchlazeno – zbytek týmu mezitím zjistil, co způsobilo celou havárii: Klíčová část generátoru se sama přenesla do minulosti, spálila svůj iridiový obal a vybuchla, což poničilo celý časoprostor. Se strojem času se půjde přenést do okamžiku těsně předtím, než dojde k výbuchu, ale je potřeba nové iridium k zastavení reakce.

⋆⋆⋆

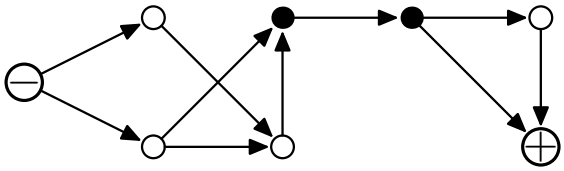
A tak se stalo, že se Will ocitl v podzemí nějaké banky 20. století před velkým sejfem, ve kterém měl být dostatečně velký kus čistého iridia získaný po dopadu nějakého meteoritu. Jenže trezor měl dost propracovaný alarm.

28-5-3 Trezor s alarmem — 10 bodů

Alarm v trezoru je soustava různě propojených obvodů, kterými teče stejnosměrný elektrický proud. Do jednoho místa soustavy je připojen záporný pól a od něj teče proud elektronů různými cestami ke kladnému pólu připojenému také na jedno místo v soustavě. Pokud se tok proudu přeruší, alarm vyhlásí poplach.

Jednotlivé dráty jsou pospojovány v uzlech a pro každý drát víme, kterým směrem v něm teče proud. Opačným směrem téci nemůže a v zapojení nejsou žádné orientované cykly.

Zapojení alarmu může vypadat jako na obrázku níže. Najděte ve schématu zapojení všechny uzly, kterých se za žádných okolností nesmíme dotknout (neboli takové, jejichž přerušení by přerušilo všechny cesty od záporného ke kladnému pólu). V příkladu níže jsou takové uzly vyznačeny černě.



Willovi se povedlo alarm rychle obejít, byla to celkem zastaralá technologie, i když na svoji dobu docela špička. V trezoru nikým nepozorován vzal trezorovou schránku, kde podle jeho záznamů mělo být iridium, otevřel ji a vyndal kus našedlého kovu. Pak ho schoval do kapsy kombinézy, schránku zase vrátil na místo a vyšel z trezoru ven.

Než se dostal do dvorany banky, zjistil, že je banka přepadena. No aspoň to lupiči budou mít o něco lehčí, řekl si při pomyšlení na odblokovaný trezor a vyřazený alarm v něm. Ale to už sahal znovu po svém minipočítači a skočil časem i s iridiem v kapse dříve, než si toho mohl kdokoliv všimnout.

⋆⋆⋆

Tentokrát se Will zjevil v nějakém skladu plném zásob jídla a s hrozným lomozem přitom shodil několik polic. Narazil si u toho nohu a zaklel: „Co to sakra? Vždyť jsem se měl vrátit zpátky!“

Teď o tom ovšem nebyl čas přemýšlet dál. Dveře do skladu byly vzpříčené, ale vykopnutí je otevřelo a jemu se naskytl pohled na kuchyň. Současně s tím jeho minipočítač zapípal, když v místnosti zdetekoval podivný časový vír. Jakoby vycházel z pánve visící na zdi. „Tohle si vezmu,“ zamumlal a pánev sundal z hřebíku.

Nějaký kuchař se mu pokusil pánev vytrhnout z ruky, ale Will ho odstrčil a se slovy „Na tohle nemám čas,“ vyběhl na ulici a zmizel v temné boční uličce. Tam začal zkoumat, proč ho pánev přitáhla na tohle místo.

Zmapoval, že mimo pánev existuje ještě několik dalších časových vírů, které se asi při výbuchu časového generátoru náhodně rozmístily po časoprostoru a tvoří teď jakési časoprostorové mosty. Willa by zajímalo, co se bude při jejich odstraňování s časoprostorem dít.

28-5-4 Časoprostorové mosty — 10 bodů

Will zjistil, že různé časy a místa jsou náhodně svázány časoprostorovými mosty. Každé místo v sobě nese jistý náboj energie a na počátku jsou všechna spojena tak, že tvoří souvislý graf.

Will si vymyslel pořadí, v jakém chce časoprostorové mosty odstraňovat, což povede k tomu, že se původně souvislý graf bude rozpojovat na menší a menší souvislé komponenty. Potřeboval by zjistit, jak se bude množství energie v jednotlivých komponentách chovat vzhledem k odstraňování mostů – vždy nás bude zajímat součet energie v rámci jednotlivých komponent.

Dostanete zadáno, kolik energie je v každém místě, a pak pořadí časoprostorových mostů, ve kterém je chce Will odstraňovat. Po každé operaci byste měli vypsat, co se stalo – buď nic (operace nerozpojila žádné dvě komponenty), nebo že došlo k rozpojení komponenty na dvě s takovou a takovou energií.

Zadání úloh KSP – 5. série

Příklad: Pokud budeme mít místa A, B, C a D s energiemi po řadě 1, 2, 3 a 4 a tato místa budou spojená do čtverce $ABCD$, tak následující posloupnost operací provede toto:

- Zrušení $A - D$: Nic.
- Zrušení $B - C$: Rozpojení na 2 části s energiemi 3 a 7.
- Zrušení $A - B$: Rozpojení na 2 části s energiemi 1 a 2.
- Zrušení $C - D$: Rozpojení na 2 části s energiemi 3 a 4.

Po naplánování ideálního pořadí rušení časoprostorových mostů přišla řada na realizaci. Will pečlivě navolil, na jaké místo a čas se chce dostat, spustil přesun...

***

... a ocitl se na místě, kde určitě nechtěl být, v nějakém římském stanu. Potřásl hlavou a rozsvítil si svítilnu na ruce. Všiml si na posteli ležícího překvapeného Římana. Náramně se podobal tomu, kterého potkal v jeskyni. Teď se začal zvedat, a tak si Will přiložil prst na ústa v pradávném gestu pro mlčení.

Pak si všiml vybaveného stolu s množstvím kladiv, kleští a dalších věcí. Rozhodl se, že si pánev trochu vylepší, urazil z ní držadlo a chvíli ji upravoval, aby ji později mohl použít k rozdrcení iridia na menší kusy. Pak si sbalil věci a chystal se k odchodu, tentokrát jen aby udělal místní přesun.

Ale ještě než zmizel, ohlédl se po Římanovi. Ujistil se, že je to ten z jeskyně, z věšáku vedle jeho postele popadl zbroj, štít a zbraň a přesunul se.

Přesun to nebyl daleký, chtěl jenom vypátrat, jak hluboko v minulosti se ocitl a jak by měl nakalibrovat svůj minipočítač, aby už dělal časové přesuny přesně.

28-5-5 Kalibrace 7 bodů

Willův minipočítač má v paměti uložen seznam časoprostorových souřadnic uspořádaných původně podle času. Ale vypadá to, že se vlivem nějaké poruchy seznam o něco zrotoval, a Will by rád zjistil, o kolik pozic.

Víme, že všechny časy jsou navzájem různé a že mimo zrotování se nic jiného nestalo. Z původní posloupnosti

$$1, 3, 4, 7, 9, 12, 13, 14, 15$$

tak mohla vzniknout třeba tato (zrotováním doprava o tři):

$$13, 14, 15, 1, 3, 4, 7, 9, 12$$

Bohužel jediné, co může Will dělat, je podívat se na nějaký index v posloupnosti, přesunout se časem na tyto souřadnice a určit, jakému odpovídají času. Rád by zjistil, o kolik se posloupnost souřadnic přesně zrotovala, ale chce při tom vykonat co možná nejméně cest časem (neboli podívat se na co nejméně pozic v seznamu). Vaším úkolem je navrhnout mu nejlepší postup, tedy postup s řádově co nejméně nutnými přesuny časem.

Po provedení několika pokusů, při kterých se zjevil s mečem v ruce před nějakým domem, pak ve sklepě Bílého domu a nakonec v Disneylandu, se Willovi konečně povedlo nakalibrovat správně minipočítač.

Teď už mohl přesně zaměřit místo a čas, kde mělo dojít k výbuchu části časového generátoru, která se přenesla do minulosti. Nastavil s velkou přesností stejné místo a čas o pět minut předtím. Iridium pomocí kladívka a pánve přeměnil na drobnější kousky, které plánoval nacpat do jádra, aby zastavil reakci. A potom zmáčkl tlačítko.

⋆⋆⋆

Objevil se v docela běžném panelákovém obývacím pokoji. Tedy byl by běžný, kdyby v půlce zdi nevisel podivný třímetrový kovový válec, který se materializoval uprostřed železobetonového panelu, jedním koncem v televizi. Válec vydával hluboký pulzující zvuk, který se začal zrychlovat. Will odklopil kryt na boku a doslova ho srazilo na zem teplo, které se vyvalilo ven. Bylo tak veliké, že skoro okamžitě chytly plamenem blízké záclony.

Will si zastínil obličej rukou a přiblížil se k válci. Po otevření krytu se ven vysunuly regulační sloty, do kterých by se mělo umístit iridium, ale nevysunuly se všechny (a Will stejně neměl tolik kousků iridia, aby je umístil do všech).

28-5-6 Sloty na iridium — 11 bodů

Máme k dispozici K kousků iridia a S slotů, do kterých se dá iridium umístit. Slotů je alespoň tolik, kolik je iridia ($K \leq S$), ale nejsou rozmístěny rovnoměrně. Všechny sice leží na obvodu kruhu, ale jsou různě daleko od sebe.

Iridium do nich potřebujeme rozmístit co možná nejvíce rovnoměrně. Za co nejvíce rovnoměrné rozmístění budeme považovat takové, které umístí každý kousek iridia do jiného slotu a obsazené sloty budou co nejdále od sebe – minimum ze vzdáleností mezi obsazenými sloty bude největší možné. Vzdálenost počítáme po obvodu kruhu (a to i přes začátek kruhu).

Sloty mají celočíselné souřadnice (mohly by to být třeba stupně) a naším úkolem je z nich rychle vybrat ty, které použijeme. Pokud bude možných několik stejně výhodných kombinací, můžeme použít libovolnou z nich.

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

Formát vstupu: Na prvním řádku vstupu budou tři čísla: obvod kruhu O, počet slotů S a počet kousků iridia k umístění K. Na druhém řádku vstupu pak bude S celých čísel s_i označujících pozice slotů na kruhu z rozsahu $0 \leq s_i < O$. Všechna čísla budou oddělená mezerami a pozice budou seřazené od nejmenší.

Formát výstupu: Na jediný řádek výstupu vypište K mezerou oddělených indexů označujících sloty, které budou použité. Indexujeme od 0.

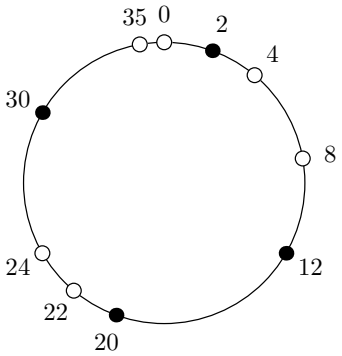
Ukázkový vstup:

```
36 10 4
0 2 4 8 12 20 22 24 30 35
```

Ukázkový výstup:

```
1 4 5 8
```

Ukázkový vstup a výstup si můžeme přiblížit obrázkem níže. Použité sloty jsou označeny plným puntíkem, minimální vzdálenosti mezi použitými sloty je 8 a větší vzdálenosti dosáhnout nelze.



Co nejrychleji vložil kousky iridia do slotů a udeřil do oranžového tlačítka. Sloty se zasunuly a Will sebou pro jistotu praštil o zem. Uvnitř došlo k bouřlivé reakci a ven vylétly rozžhavené jiskry, které zapálily blízké okolí. Ještě že kombinézy fasované v institutu byly nehořlavé! Po pár sekundách ale začala reakce ustávat, iridium zafungovalo jako regulátor.

Will se uprostřed hořícího bytu postavil, podíval se na nyní už neškodný válec a aktivoval v něm malou autodestrukční nálož. Pak s modrým zablesknutím z hořícího bytu rychle zmizel, dole už totiž zaslechl houkání sirén.

⋆⋆⋆

Z modré koule se vyloupl opět v místnosti s reaktorem, teď navíc s krátkým efektem šlehajících plamenů, které vtáhl s sebou. Ušklíbl se nad tím a spojil se s ostatními.

„Tak časové havárii jsme snad zabránili, zamezil jsem výbuchu!“ oznámil.

„Skvěle, tak teď tě už jenom dostat zpátky,“ přišla mu odpověď.

Aby vrátili místnost s reaktorem (a Willem) zpátky do normální fáze, museli omezit vzniklý časový vír. K tomu ale bylo potřeba provést velmi mnoho rychlých časoprostorových výpočtů v krátkém čase – když už se operace zahájí, nejde přerušit ani pozastavit. Proto se rozhodli si něco předpočítat předem.

28-5-7 Tajemná operace — 12 bodů

Máme počítač počítající tajemnou operaci $\otimes$, o které víme jenom to, že je binární a asociativní (tedy že je to operace mezi dvěma prvky a že nezáleží na uzávorkování operací ve výrazu). Co si pod tím představit? Může to být například obyčejné násobení nebo sčítání, nebo třeba násobení modulo 10^9 (všimněte si, že na rozdíl od běžného násobení k tomuto neexistuje inverzní operace – není tedy možné dělit).

Běh programu se bude skládat ze dvou částí. V první části si může program v nějakém rozumném čase předpočítat, co bude chtít, a pak bude ve druhé (ostré) části běhu dostávat dotazy. Problémem je, že výpočet tajemné operace $\otimes$ je náročný a během ostrého běhu zvládne počítač v čase každého dotazu použít operaci $\otimes$ pouze jedenkrát.

Na začátku běhu dostane program pevně daná čísla a_1 až a_n, na nichž si můžeme spočítat, co bude chtít, a při ostrém běhu pak bude dostávat mnoho dotazů typu

$$a_i \otimes a_{i+1} \otimes \ldots \otimes a_{j-1} \otimes a_j$$

neboli dotazů na výpočet operace $\otimes$ na nějakém intervalu mezi i a j (kde $i \leq j$).

Chtěli bychom si tedy vybudovat nějakou datovou strukturu, která nám při ostrém běhu umožní odpovídat na takové dotazy pouze s jedním zavoláním funkce $\otimes$. Ale výpočet takové struktury by taky měl být co možná nejrychlejší.

Výpočty byly dokončeny, místnost s generátorem vrácena do správné fáze a Will už se chystal k tomu, že si po náročném dni půjde dát horkou sprchu. V tom jeho pohled padl na římské brnění a meč, které přinesl a položil s otlučenou pánví do kouta. Chvilku přemýšlel, pak to vše vzal do náruče a do interkomu oznámil: „Ještě moment, mám nějakou nevyřízenou práci. . . “

⋆⋆⋆

Než se 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. . .

Příběh pro vás dovyprávěl

Jirka Setnička

Seriál – Evoluční algoritmy

28-1-8 Programování podle Darwina — 15 bodů

V letošním seriálu se budeme věnovat přírodou inspirovaným algoritmům, jakými jsou například *evoluční algoritmy* či *neuronové sítě*. Téma je velmi široké a obsahuje v sobě velkou spoustu postupů, ze kterých my se budeme věnovat jen těm základním a často používaným.

Úvod do evolučních algoritmů

Proč se vlastně informatici snaží přírodou inspirovat? Jednak určitě hraje roli motivace vyzkoušet si něco neobvyklého, ale jedním z hlavních důvodů je, že tradičními (exaktními) informatickými postupy mnoho problémů neumíme vůbec řešit. Přitom příroda má v zásobě spoustu poměrně silných a obecných technik pro řešení problémů, nepodobných čemukoli, na co jsme v informatice zvyklí. A při pohledu na výsledky se zdá, že fungují docela dobře. Patří mezi ně třeba právě proces evoluce (který je vlastně takovým algoritmem na hledání nejschopnějších forem života).

Nabízí se tedy otázky: „Mohly by nám tyto techniky nějak pomoci při řešení těžkých algoritmických problémů?“, „Dají se jednoduše popsat, nebo dokonce naprogramovat?“, „Dokážeme naprogramovat mravence, kteří by společně namísto stavění mraveniště hledali cestu v grafu?“, „Může se i počítač pomocí simulace neuronů něco naučit?“ a tak dále. Ukazuje se, že odpovědí na většinu těchto otázek je: „Ano, je to možné!“

To všechno vypadá skvěle! Ale možná si teď někteří z vás pro sebe řekli: „Budu já tomu rozumět? Já biologii moc neumím.“ Tak toho se přesně nemusíte bát. Všechny algoritmy, kterým se během seriálu budeme věnovat, se přírodou pouze inspirují. To znamená, že sledují nějaké její základní chování, a pak si jej vysvětlí svým informatickým způsobem. Tedy prakticky žádné biologické znalosti nejsou potřeba.

Evoluční algoritmy – část 1

Prvních několik dílů seriálu se společně budeme věnovat *evolučním algoritmům*. V tomto díle si konkrétně popíšeme a naučíme se používat vůbec první typ: takzvaný *genetický algoritmus*. Řekneme si, čím je motivován, podrobně popíšeme jeho hlavní části a pokusíme se pomocí něj vyřešit pár problémů. Do otázek ohledně toho, proč by takový algoritmus vůbec měl mít šanci fungovat, zatím nebudeme příliš zabíhat – ty si necháme na příště.

S genetickým algoritmem poprvé přišel John Holland v roce 1970. Genetický algoritmus je inspirován myšlenkou evoluce. V přírodě platí pravidlo „silnější přežijí,“ což znamená, že nejsilnější jedinci obstojí v konkurenci ostatních, reprodukují se a zvládnou tak přenést své geny do dalších generací. Tím v každé

další generaci dostáváme lepší jedince, protože nám zůstanou geny jen těch, kteří dokázali přežít.

Dále budeme předpokládat, že výkonost jedince ovlivňují pouze jeho geny a nic jiného (tomu se v biologii říká *darwinismus*). Tento předpoklad znamená, že při reprodukci jsou potomci a jejich vlastnosti závislí pouze na genech rodičů a ne na jejich životních zkušenostech. Nyní se pojďme podívat, jak vypadá nějaký informatický genetický algoritmus.

Genetický algoritmus sestává z populace *jedinců*, funkce ohodnocující výkonost těchto jedinců (*fitness funkce*) a tří hlavních genetických operátorů pro manipulaci s jedinci: *selekce*, *křížení* a *mutace*. V dalším textu si tyto jednotlivé části podrobněji popíšeme.

Jedinec je tvořen posloupností genů, která představuje nějaké řešení našeho problému. Je důležité, aby tato posloupnost měla danou fixní délku. Zatím navíc budeme předpokládat, že tato posloupnost je binární (skládá se pouze z 0 a 1). Pro binární soustavu dokážeme jednoduše popsat další operátory.

Fitness funkce ohodnocuje výkonost (kvalitu) jednotlivých jedinců. Jinými slovy říká, jak jsou jedinci dobří pro řešení našeho problému. Zpravidla vyhodnocuje tak, že vyzkouší, jak dobře genetický kód jedince řeší zadaný problém a ohodnotí jej reálným číslem.

Selekce

Pomocí *selekce* vybíráme, které jedince použijeme pro vytvoření další populace. Bereme přitom v úvahu fitness jedinců, ale zároveň ponecháváme i určitou míru náhody. My si uvedeme dva druhy selekcí: *ruletovou selekci* a *turnajovou selekci*.

Ruletovou selekci si představíme jako opravdovou ruletu. Máme kruh rozsekaný na různě velké části, kde každá odpovídá jednomu jedinci. Velikost části kruhu je přímo úměrná fitness jedince. Do rulety pak hodíme kuličku a vybereme toho jedince, v jehož části kulička skončí. Tento proces opakujeme tolikrát, kolik jedinců potřebujeme vybrat. Nevadí nám, pokud jednoho jedince vybereme vícekrát.[26]

V praxi je ruletová selekce počítána tak, že se vygeneruje náhodné číslo od 1 do součtu všech fitness a podle toho se vybere příslušný jedinec. Z toho důvodu je pro ruletovou selekci nutné, aby fitness funkce byla vždy kladná.

Pak ale existuje ještě *turnajová selekce*, u které hodnoty fitness funkce mohou být libovolné. Ta funguje tak, že vezme dva náhodné jedince a z nich vybere toho s lepší fitness. To opět zopakuje tolikrát, kolik chceme vybrat jedinců. (Turnajová selekce nemusí vždy vybírat jen lepšího ze dvou jedinců, ale klidně obecně nejlepšího z k jedinců.)

[26] V přírodě by to sice nešlo, ale my si to v informatice klidně můžeme dovolit a jednoho jedince si nakopírovat, kolikrát chceme.

Křížení

Křížení je jedním z nejdůležitějších genetických operátorů. To vezme dva jedince a nějakým způsobem je zkombinuje. Nejčastěji se používá takzvané *jednobodové křížení*, které vybere náhodný bod, tam jedince rozpůlí a prohodí jejich druhé části. Obdobně také funguje *dvoubodové křížení*, které náhodně zvolí dva body a pak prohodí tu část jedinců, která je mezi nimi.

Také se může použít křížení, které se pro každý bit zvlášť rozhodne, zda jej prohodí nebo ne. Takové křížení se ale pro některé problémy moc nepoužívá. Později si sami můžete vyzkoušet, které z křížení vám bude fungovat lépe.

Křížení se neaplikuje na všech jedincích, ale probíhá s pravděpodobností p_k, která se obvykle pohybuje od 0,7 do 0,9. Křížení se často považuje za hlavní pohon genetických algoritmů. Na druhou stranu existuje mnoho verzí a odvození, které křížení vůbec nezahrnují.

Mutace

Mutace je operátor, který náhodně změní jednoho jedince. To jest každý bit s nějakou malou pravděpodobností změní. Tato pravděpodobnost je často volena kolem $1/d$, kde d je délka jedince. Taková volba pravděpodobnosti způsobí, že během mutace v průměru prohodíme právě jeden bit jedince. Mutace se na jedince aplikuje s pravděpodobností p_m, která se obvykle volí v rozmezí 0,001 až 0,05.

Poslední pojem, který si definujeme, je *generace*. Generací je myšlena populace, která existuje v jedné iteraci. Na začátku máme generaci 0, z té je pak pomocí selekce, křížení a mutace vytvořena generace 1, z té pak generace 2 a tak dále. Předchozí generace vždy celá umírá a dále se používá pouze nová generace.

Pseudokód algoritmu

Nyní jsme si představili všechny důležité části genetického algoritmu, tak se pojďme podívat, jak dohromady fungují. Zde je pseudokód algoritmu.

1. Vygeneruj náhodných n jedinců velikosti d do generace 0 a spočítej jejich fitness.
2. $t = 0$
3. Opakuj následující:
4. Pomocí selekce vyber m jedinců z generace t.
5. Na každého z těchto m jedinců aplikuj křížení s pravděpodobností p_k.
6. Na každého dále aplikuj mutaci s pravděpodobností p_m.
7. Spočítej fitness výsledných jedinců a nejlepších n prohlaš za generaci $t+1$.
8. $t = t + 1$

Jelikož využíváme náhodu, vyplatí se algoritmus pustit několikrát za sebou a ze všech běhů vzít ten nejlepší výsledek. Při každém běhu začínáme s jinak nagenerovanou počáteční populací, a tedy můžeme získat i jinak dobré řešení.

A to je celé. Poslední, co zbývá říct, je, kdy se algoritmus zastaví. To může být buď ve chvíli, kdy vyvineme jedince reprezentujícího optimální řešení problému (tj. s maximální možnou fitness, pokud ji známe) nebo po určitém počtu iterací. Často se také zohledňuje počet vyhodnocení fitness funkce, protože právě ta bývá tou časově nejnáročnější částí algoritmu. Ta se ale v našem případě pustí v každé iteraci právě m-krát, takže výpočet budeme řídit počtem iterací a velikostí populace. ($m \geq n$, ale často $m = n$)

Elitismus

Než si algoritmus vyzkoušíme, zmíníme ještě poslední věc. V algoritmu, tak jak jsme jej popsali, se lehce může stát, že během přesunu na další generaci přijdeme o nejlepšího jedince – můžeme jej nevybrat, může se špatně zkřížit a může zmutovat. Z tohoto důvodu se do algoritmu přidává ještě další vlastnost, která se jmenuje *elitismus*. Ta funguje tak, že do výběru pro generaci $t+1$ přidáme ještě $p_e \cdot n$ nejlepších jedinců z generace t. Tím určitě zachováme doposud nejlepší geny. Hodnota p_e se obvykle volí maximálně 0,1. Nemůžeme brát moc velkou část, protože pak by nám celá populace postupně konvergovala jen k jednomu aktuálně nejlepšímu jedinci.

Nyní si algoritmus pojďme vyzkoušet. Protože genetické algoritmy obsahují mnoho parametrů, které se musí správně nastavit, aby dobře fungovaly, ladí se nejdříve na jednoduchých problémech. Na takových, na kterých je dobře vidět, jak algoritmus funguje, a pro které umíme efektivně vyhodnocovat fitness funkci. My se pokusíme navrhnout genetický algoritmus, který se bude snažit vyvinout posloupnost samých jedniček.

Pro takový problém je fitness funkce jednoduchá – prostě jen spočítá, kolik jedniček jedinec obsahuje. Selekce, křížení a mutace fungují přesně tak, jak je popsáno výše. Zbývá vyladit parametry: velikost populace n, počet iterací t a hodnoty p_k, p_m, p_e. Vaším úkolem teď bude si různé kombinace těchto parametrů vyzkoušet a zjistit, jak se algoritmus chová.

Úkol 1 [6b]: Pomocí genetického algoritmu vyviňte posloupnost samých jedniček. Tedy začněte s populací náhodných jedinců a pomocí genetického algoritmu se snažte vyvinout jedince, který je složen pouze z jedniček.

Vyzkoušejte to pro velikosti jedince $d = 20, 100, 500$. Zkuste různé velikosti populace a různé kombinace pravděpodobností. Jak se algoritmus chová?

Sledujte, jak se během výpočtu mění maximální a průměrná fitness generací.

Vyzkoušejte si také napsat nějaký vlastní způsob křížení nebo mutace. Jak to bylo úspěšné? Tato křížení a mutace by neměly nijak záviset na znalosti řešeného problému. Cílem je odladit operátory, které by pak mohly fungovat i pro jiné, už ne tak jednoduché problémy.

Vyzkoušejte také, jak je algoritmus úspěšný, pokud má vyvinout jedince, kde se 0 a 1 střídají. (Přičemž je jedno, čím se začne.) Mělo by stačit změnit fitness funkci. Funguje váš algoritmus stále stejně dobře?

Během řešení můžete použít naše šablony genetického algoritmu ze stránky `http://ksp.mff.cuni.cz/viz/evoluce`.

Odevzdávejte soubor typu ZIP s popisem řešení, průběhem algoritmu a pokud chcete, tak i se zdrojovým kódem. V popisu rozeberte, co jste zkoušeli a jaké jste použili parametry, aby algoritmus byl co nejlepší.

Průběhem algoritmu je myšlen textový soubor, kde na každém řádku budou mezerou či tabulátorem oddělené hodnoty: číslo generace, hodnota průměrné fitness, hodnota maximální fitness. Nemusíte logovat každou generaci, stačí každá desátá.

Tím jsme si vyzkoušeli, jak se genetický algoritmus ladí na jednoduchém problému. Často při vymýšlení nového operátoru či dokonce celého algoritmu se hodí jej nejdříve vyzkoušet a odladit na něčem takto jednoduchém. To se především týká operátorů, které jsou nezávislé na řešeném problému. O takové operátory se snažíme, protože je pak můžeme aplikovat i na složitější problémy.

Někdy ale můžeme znalost řešeného problému využít a přímo ji zahrnout do genetických operátorů, jako jsou křížení nebo mutace. To na jednu stranu může značně urychlit výpočet, na druhou stranu nás to ale může v řešení zahnat někam do suboptimálního řešení, ze kterého se nebudeme moci dostat.

To vše si vyzkoušíme na problému sedmi loupežníků. Skupinka sedmi loupežníků vyloupila vesnici a získala z ní dohromady d předmětů, každý z nich ohodnotila nějakou cenou. Vaším úkolem je tyto předměty rozdělit na 7 hromádek tak, aby jejich součet byl co nejbližší. Konkrétně tak, aby rozdíl nejhodnotnější a nejméně hodnotné hromádky byl co nejmenší.

Než se pustíme do řešení, musíme vyřešit několik problémů: Jak budeme kódovat jedince? Jak v tomto kódování bude fungovat křížení a mutace? Jak zvolit fitness funkci?

Jedinci budou délky d a budeme je kódovat čísly $0, 1, \ldots, 6$. Pokud na i-té pozici máme číslo 4, tak to znamená, že i-tý předmět přidělíme do hromádky 4. Křížení může fungovat stejně jako s binárními jedinci a mutace změní číslo na náhodnou hodnotu od 0 do 6 namísto překlopení bitu. To, jak dobře tyto operátory budou fungovat, je jiná otázka.

A co s fitness funkcí? V tomto případě máme za úkol minimalizovat rozdíl největší a nejmenší hromádky. Náš genetický algoritmus se ale snaží fitness funkci f maximalizovat a ne minimalizovat.

U turnajové selekce problém můžeme vyřešit jednoduše – prostě použijeme hodnoty $-f(x)$ namísto $f(x)$. Co ale dělat, pokud chceme použít ruletovou selekci? Tam všechny fitness navíc musí být kladná čísla. Máme několik možností, jak toho dosáhnout. Často se používá hodnota $1/(f(x)+1)$ namísto $f(x)$. Nebo hodnota $A - f(x)$ pro vhodně zvolené A tak, že výsledné hodnoty určitě budou kladné. Musíme ale dát pozor, aby A nebylo příliš velké, protože pak by výsledné hodnoty byly příliš blízko u sebe a z pohledu ruletové selekce byly „skoro stejné“.

Tím jsme si poradili se všemi problémy a tedy genetický algoritmus můžeme zkusit použít.

Úkol 2 [9b]: Pomocí genetického algoritmu řešte problém sedmi loupežníků pro data, která naleznete na stránce se šablonami. Data jsme vygenerovali troje: lehká, střední a těžká. Doporučujeme je řešit postupně. Tj. až si budete myslet, že máte dost dobré řešení pro lehká data, zkuste, jak vám algoritmus funguje pro střední, atd.

Na prvním řádku dat jsou dvě celá čísla: počet loupežníků (vždy 7) a počet nakradených věcí D. Na druhém řádku je D mezerou oddělených čísel udávajících váhy jednotlivých věcí.

Opět zkoušejte různé kombinace parametrů. Také si vyzkoušejte, jak nejlépe převést fitness funkci na maximalizační – můžete vymyslet i vlastní způsob nebo nějaké modifikace způsobů popsaných výše.

Nalezněte co nejlepší řešení daného problému. Můžete použít i vlastní genetické operátory, které libovolně využívají znalost problému a provádějí křížení nebo mutace „cíleně“ na specifických částech jedinců.

V ZIPu odevzdejte nejlepší vyvinuté řešení společně s popisem, jak jste jej dosáhli a proč si myslíte, že takový postup funguje. Také můžete přidat záznam průběhu řešení (jako v minulé úloze).

Při řešení obou úloh můžete upravit námi vytvořenou šablonu v jazycích C++, Java, ..., anebo použít svou vlastní.

Naše kódy obsahují základní verzi genetického algoritmu se všemi jeho částmi. Navíc logují, jak se během výpočtu mění průměrná a maximální fitness, a ukládají doposud nejlepšího vyvinutého jedince.

Karel Tesař

28-2-8 Genetika vs. procházení krajiny — 16 bodů

V prvním díle seriálu jsme si představili genetické algoritmy, jejich operátory a základní funkčnost. V tomto díle se postupně dostaneme k verzi algoritmů, které pracují s jedinci tvořenými reálnými čísly, a získáme aspoň základní porozumění, proč by vůbec takový algoritmus měl fungovat.

Než se však dostaneme přímo k těmto otázkám, představíme si základní postupy optimalizačního prohledávání prostoru řešení.

Všechny optimalizační problémy se dají formulovat tak, že máme zadanou n-rozměrnou funkci f, která jako vstupní parametry dostane n reálných čísel a odpoví jedním reálným číslem. Funkci f se pak snažíme maximalizovat, resp. minimalizovat. To jest hledáme takové vstupní parametry, pro které bude výsledek funkce největší, resp. nejmenší možný.

Takovou funkcí může být například:

$$f(x, y, z) = 3x^4 + 2(y + 4)^2(z - 2)^2$$

Pokud ji chceme minimalizovat, optimálním řešením jsou hodnoty $x = 0$, $y = -4$, $z = 2$, pro které $f(0, -4, 2) = 0$.

To je jednoduché, ne? Bohužel ale jen v tomto případě. My obvykle nemáme žádný takto jasný předpis a často ani nevíme, jaká je optimální hodnota funkce. Většinou jen známe počet vstupních parametrů a pro jejich konkrétní hodnoty umíme spočítat hodnotu funkce.

Příkladem takových funkcí mohou být všechny fitness funkce z minulého dílu seriálu a také všechny cílové funkce, které se objeví v tomto díle.

Metoda horolezení (hill climbing)

Budeme pracovat s funkcemi reálných proměnných, které popisují, jak dobré je řešení nějakého problému. Takové funkce jsou obvykle rozumně spojité. To znamená, že kdybychom si graf funkce nakreslili, tak nám její povrch bude připomínat krajinu. Budou na ní kopce, údolí, nadmořská výška bude plynule přecházet a zřídkakdy narazíme na svislý útes nebo propadliště. Prostě taková obyčejná krajina, kterou všichni známe.

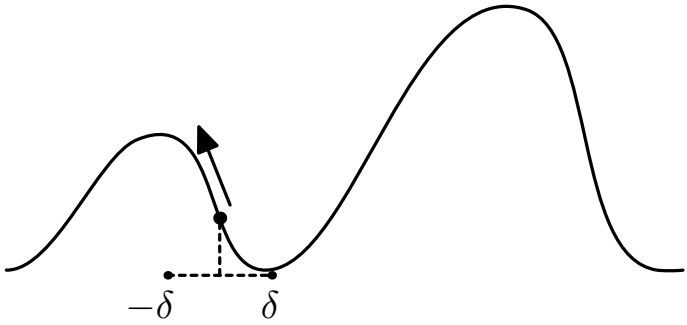
Nadále si dovolíme předpokládat, že všechny funkce mají tvar nějaké takové krajiny. Pak si pod optimalizační úlohou pro takovou funkci si můžeme představit hledání nejvyšší hory (v případě maximalizace) nebo nejhlubšího údolí (v případě minimalizace).

My se budeme věnovat hledání nejvyšších hor a ukážeme si metodu, která se nazývá *hill climbing* (česky metoda horolezení). Metoda si na začátku náhodně vybere start (náhodný bod v krajině) a z něj začne šplhat nahoru na kopec, dokud to jde.

Šplhání probíhá tak, že se náhodně zvolí bod z okolí místa, kde právě stojíme, spočítáme v něm hodnotu funkce, a pokud je stejná nebo vyšší než aktuální, přesuneme se tam. Celý postup opakujeme po daný počet iterací.

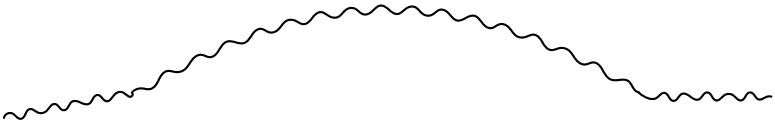
Bod přesunu vybíráme tak, že ke každé souřadnici přičteme náhodnou hodnotu z rozmezí $\langle -\delta, \delta \rangle$, kde δ je námi určená konstanta. Ta se pro začátek algoritmu volí trochu větší (povolujeme velké skoky a hledáme kopec) a v závěrečné fázi naopak hodně malá (už jsme na kopci a jen se přibližujeme vrcholu).

Tento algoritmus má však jednu značnou nevýhodu: skončíme na prvním kopci, který najdeme, a vůbec nevíme, zda třeba někde není další a ještě vyšší. Řečí matematiky najdeme nějaký lokální extrém, o kterém nevíme, jestli je i globálním.



To můžeme napravit tak, že metodu pustíme vícekrát za sebou a ze všech pokusů vybereme ten nejlepší. To už vypadá lépe – když vylezeme na více kopců, tak tím zvýšíme pravděpodobnost, že jsme se alespoň jednou ocitli na tom úplně nejvyšším. Ale...

Co když naše krajina bude vypadat jako zorané pole? Tam pak jsou všude samé malé kopečky, a ať začneme na jakémkoliv místě, hned na některém z nich skončíme. Tomu bychom chtěli nějak zamezit.



Simulované žíhání

Problém „malých kopečků“ řeší další metoda, která se nazývá *simulované žíhání*. Ta dělá přesně to, co metoda horolezení, jen navíc dovoluje s určitou pravděpodobností přejít i do bodu nižšího než aktuální.

Pravděpodobnost, s jakou si dovolíme přejít níž, se řídí dvěma faktory. Prvním je velikost změny od aktuální hodnoty (tu budeme značit Δf) a druhým je takzvaná *teplota* (značená T). Čím vyšší teplota, tím vyšší pravděpodobnost, že změnu přijmeme.

Na začátku algoritmu nastavíme teplotu vysokou a v průběhu ji pomalu snižujeme až skoro na nulu. Snižováním teploty snižujeme toleranci na velikost poklesu. Pro danou velikost snížení Δf a danou teplotu T má pravděpodobnost přijetí přesně hodnotu $e^{-\Delta f/T}$.

Snižování teploty můžeme provádět přenásobením konstantou α, kterou zvolíme z intervalu $(0, 1)$. Tomu se říká chladící schéma. Následuje pseudokód algoritmu simulovaného žíhání pro jednorozměrnou funkci (funkci jedné proměnné).

1. $T = T_0, \delta = \delta_0$
2. Vyber počáteční bod $x = x_0$.
3. Urči maximální počet iterací M a číslo iterace $i = 0$.

4. Dokud $i < M$
5. $y = x + \text{rand}(-\delta, \delta)$ *(náhodný posun)*
6. Pokud $f(y) > f(x)$
7. $x = y$ *(je vyšší, hned přijmi)*
8. Jinak
9. Vygeneruj náhodně hodnotu $r \in \langle 0, 1 \rangle$.
10. Pokud $r < e^{\Delta f / T}$, pak $x = y$ *(přijmeme s danou pravděpodobností)*
11. Pokud x je zatím nejlepší řešení, zapamatuj si jej.
12. $T = \alpha T$
13. Volitelně můžeme snížit δ.

Algoritmus pro více proměnných je naprosto shodný, jen s proměnnou x pracujeme jako s vektorem a operace provádíme zvlášť na všechny jeho složky. Celý algoritmus stejně jako u horolezení pouštíme vícekrát a ze všech řešení vybereme to nejlepší.

Na závěr této sekce ještě poznamenáme, že pro výběr dalšího bodu se často používá posun podle Gaussova normálního rozdělení. Protože to se ale na středních školách často nevyučuje, zvolili jsme jednoduší postup, který také funguje dobře.

Úkol 1 [7b]: Zkuste pomocí metody horolezení, simulovaného žíhání nebo nějakým vlastním způsobem vyřešit následující úlohu.

V rovině máme rozmístěných dohromady p bodů. Chtěli bychom tam přidat k centrálních stanic tak, aby součet vzdáleností všech bodů do jejich nejbližší stanice byl minimální.

Pro zadané vstupní body[27] řešte postupně pro $k = 1, 3, 5$. Na prvním řádku najdete počet bodů p. Na dalších p řádcích jsou vždy dvě čísla udávající souřadnice jednoho z bodů.

Na úlohu můžete vyzkoušet i algoritmy z dalšího textu. Napište, co jste zkoušeli a jak vám to fungovalo. Který z přístupů vám fungoval nejefektivněji?

Společně s popisem řešení pošlete i průběh vašeho algoritmu společně s nejlepšími dosaženými řešeními.

Explorace versus exploatace

Nyní odhalíme, proč jsme vůbec simulované žíhání a metodu horolezení probírali v souvislosti s genetickými algoritmy. Prvním důvodem je, že všechny tyto algoritmy mají společný cíl, totiž maximalizaci či minimalizaci cílové funkce. Druhým důvodem pak je, že oběma těmto skupinám se pokusíme porozumět pomocí pojmů explorace a exploatace.

Pojem *explorace* zastřešuje objevování nových částí prostoru řešení. To jest pokud se náš algoritmus podívá na hodně míst krajiny funkce, tak hodně exploroval.

[27] `http://ksp.mff.cuni.cz/viz/evoluce`

Pojem *exploatace* naproti tomu zastřešuje lokální prohledávání a využívání informací, které jsme již objevili. Tedy hledání kopce na nějakém lokálním místě v krajině patří do exploatace.

Při návrhu optimalizačního algoritmu se snažíme o vyvážení explorace a exploatace. Pokud algoritmus bude málo explorovat a hodně exploatovat, tak bude mít tendence nalézat lokální extrémy a držet se jich. Naopak pokud bude hodně explorovat a málo exploatovat, tak se bude blížit náhodnému tipování bodů. Sice jich hodně vyzkouší, ale nevyužije pořádně informace o tvaru jejich okolí.

Metoda horolezení i simulované žíhání v prvním kroce explorují (vyberou náhodný bod) a pak už jen exploatují (zkoumají aktuální okolí). Tento nedostatek explorace se pak snaží dohnat opakovaným spuštěním celého algoritmu.

Nyní pojďme rozebrat genetické algoritmy. Generování počáteční populace a opakované spouštění algoritmu patří jednoznačně do explorace. Operátor selekce se na druhou stranu řadí do exploatace (z aktuálních jedinců ponecháváme jen ty nejlepší). Zbývá nám zařadit operátory, které s jedinci přímo manipulují: křížení a mutaci.

Mutace je v genetickém algoritmu považována za představitele exploračního operátoru – přinášíme do jedince náhodnou novou informaci. Křížení se naopak považuje za operátor exploatační, protože pouze skládá dohromady informace, které již v jedincích máme.

Pohled na křížení jako na exploatační operátor může na první pohled vypadat neintuitivně. Vždyť přece křížením dvou jedinců se najednou dostaneme na úplně nové místo v krajině… To je sice pravda a z tohoto pohledu křížení může vypadat trochu exploračně, ale stále platí fakt, že jsme využili jen informace, které jsme již měli. Takže křížení v jistém smyslu funguje lokálně, ale naprosto jiným způsobem a v jiném rozsahu než například metoda horolezení nebo simulované žíhání.

Důvodem, proč např. genetické algoritmy vnímáme nadějně, je právě dobré zastoupení jak explorace, tak exploatace. Algoritmus prohledává okolí hned na několika místech najednou, tyto informace kombinuje dohromady a při tom se zároveň snaží objevovat nová místa.

Genetické algoritmy v reálných číslech

Nyní se podíváme na to, jak bychom genetickým algoritmem mohli řešit problém vyžadující reálné jedince (vektor reálných čísel). Pak jej budeme moci porovnat s metodami výše. Jednotlivé hodnoty jedinců budeme nazývat složky.

Takový genetický algoritmus bude fungovat naprosto stejně jako ten z prvního dílu seriálu, jen pro něj musíme navrhnout operátory křížení a mutace, které dokáží s reálnými jedinci pracovat.

Mutaci můžeme realizovat obdobně – náhodně pohneme s jednou či více složkami jedince. Realizujeme přičtením náhodné hodnoty z intervalu $\langle -\delta, \delta \rangle$. Další možností je vygenerovat novou hodnotu z daného rozsahu.

Křížení můžeme dělat jednobodové, stejně jako v minulém díle, anebo s jedinci můžeme pracovat více jako s vektory a počítat například jejich průměr. Případně místo průměru můžeme počítat konvexní kombinaci, která pro jedince x a y vypadá následovně:

$$z = ax + (1 - a)y \; ; \; a \in \langle 0, 1 \rangle \, ,$$

kde hodnotu a můžeme mít fixní, volit náhodně, nebo volit na základě fitness obou jedinců.

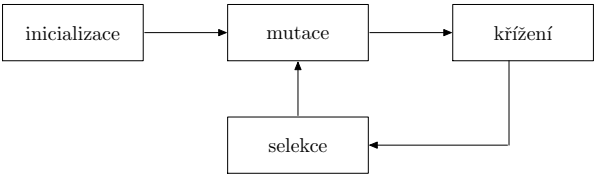
Oba operátory se dají uchopit ještě mnoha dalšími způsoby. Mně se například na operátoru křížení nelíbí to, že opakovaným průměrováním hodnot si jedinci budou navzájem čím dál podobnější. Časem budou všichni téměř stejní, blízko průměru původních hodnot. Přesto bych ale chtěl pracovat s jedinci jako s vektory hodnot a nějakým způsobem využívat informace, které v sobě uchovávají, a dostávat z nich nová, doposud nepoznaná řešení. Práci s vektory hojně využívá diferenciální evoluce.

Diferenciální evoluce

Diferenciální evoluce je specifická verze genetického algoritmu, ve které se s jedinci pracuje jako s vektory reálných čísel. Také využívá operátory selekce, křížení a mutace, ale přistupuje k nim jiným způsobem než genetické algoritmy.

Průběh diferenciální evoluce vypadá následovně:

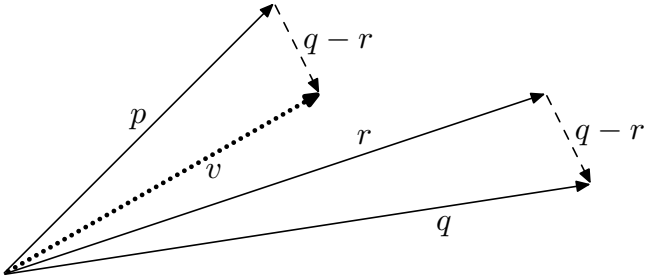
1. Inicializuj populaci n náhodnými jedinci o velikosti d.
2. Opakuj následující:
3. Proveď mutace.
4. Proveď křížení.
5. Proveď selekci.



Nejdříve aplikujeme mutaci, ta probíhá tak, že pro každého jedince vytvoříme takzvaného dárce (donor) z dalších tří náhodných jedinců. Pro jedince x vytvoříme donora v z náhodných jedinců p, q, r:

$$v = p + F \cdot (q - r),$$

kde F je reálný parametr z intervalu $[0, 2]$, kterému se říká *diferenciální váha*. Sice se teoreticky povoluje váha až 2, ale v praxi se vyplatí používat hodnoty jen do 1.



Operace mutace probíhá s celými vektory po složkách. Vezmeme směr vektoru jednoho jedince, přičteme k němu rozdíl směrů dalších dvou jedinců a získáme našeho dárce, kterého využijeme v dalších fázích.

Dále je na řadě křížení. To nastává s pravděpodobností $C \in \langle 0, 1\rangle$. Křížíme původního jedince x s jeho dárcem v a vytvoříme tak výsledného jedince u. To provedeme tak, že vygenerujeme náhodné číslo $r \in \langle 0, 1\rangle$. Pokud $r < C$, tak položíme $u = v$, jinak $u = x$, až na jednu náhodnou složku j, kterou vezmeme z v.

Neboli:

$$u_i = \begin{cases} v_i & \text{pokud } r < C \text{ nebo } i = j \\ x_i & \text{pokud } r \geq C \text{ a } i \neq j \end{cases}$$

Pro hodnotu C je většinou dobrá první volba $C = 0.5$.

Jako poslední v sérii operací je selekce. V té pouze porovnáme fitness původního jedince x s fitness výsledného jedince u a do další generace vezmeme jen lepšího z nich.

Takto vypadá celá jedna iterace. Náhodní jedinci pro dárce se určí na základě tří náhodných permutací jedinců. To znamená, že během jedné iterace se každý z jedinců použije právě jednou jako p, právě jednou jako q a právě jednou jako r. Navíc existuje i verze algoritmu, kdy se za jedince p vždy dosadí aktuálně nejlepší jedinec z populace (tím všichni dárci vychází ze směru stejného, nejlepšího jedince, což nemusí být vždy výhodné).

To je celý algoritmus. Má výhodu i v tom, že se díky rovnicovému zápisu dá naprogramovat o něco lépe než například klasický genetický algoritmus. Všimněte si, že je nám dokonce i jedno, zda fitness funkci maximalizujeme či minimalizujeme.

Závěrem okomentujeme, jak volit velikost populace. Ta z logiky algoritmu musí být alespoň 4. Avšak většinou se n volí velikostí mezi $5d$ a $10d$, kde d je

velikost (počet složek) jedince. Je to ale pouze doporučení, není žádný důvod, proč nezkusit třeba fixní hodnotu mezi 40 a 100.

Úkol 2 [9b]: Pomocí genetického algoritmu v reálných číslech a diferenciální evoluce zkuste řešit následující úlohu.

Máme zadaný velký obdélník o rozměrech $W \times H$ a sadu k malých obdélníků o rozměrech $w_1 \times h_1, \ldots, w_k \times h_k$. Naskládejte malé obdélníky do velkého tak, aby se celkově co nejméně překrývaly. Celkový překryv je součtem překryvů všech dvojic obdélníků. Za překryv se navíc počítá i vybočení ven z velkého obdélníka.

KSP

Úlohu řešte pro data, která najdete na stránce seriálu.[28] Na prvním řádku jsou čísla W a H, na druhém řádku pak počet obdélníků k a na dalších k řádcích jsou vždy dvě čísla: w_i, h_i – rozměry obdélníka i.

seriál

Opět vyzkoušejte různé kombinace parametrů. Úlohu můžete zkusit vyřešit i jiným, neevolučním způsobem, váš výsledek do evoluce uměle dosadit a zkusit jej ještě zlepšit.

Při řešení můžete využívat šablonu genetického algoritmu z minulého dílu nebo novou šablonu pro diferenciální evoluci. Obě najdete na stejné stránce jako vstupní data.

Společně s popisem řešení pošlete i průběh vašeho algoritmu a nejlepší řešení, jakého jste dosáhli.

Karel Tesař

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*.

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.

[27] `http://ksp.mff.cuni.cz/viz/evoluce`

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

28-4-8 Strojové učení — 20 bodů

Co je to strojové učení? Lidské učení je schopnost adaptace na nové situace. Když budeme poprvé hrát šachy, asi nám to nepůjde moc dobře, ale podruhé to půjde trochu lépe, a když budeme hrát dost dlouho, můžeme se stát experty. Můžeme se naučit hrát dobře šachy, i když na začátku dostaneme jenom jejich pravidla, která nám umožňují dělat špatné i dobré tahy. Nikdo nám nedá přesný postup, jak být expert – v naší hlavě hraním šachů vznikne schopnost hrát je dobře.

Strojové učení se snaží replikovat tuhle schopnost v počítačích. Je velmi užitečné umět řešit problémy, na které neznáme žádné jasné algoritmy. To platí obzvlášť, když se snažíme automatizovat nějaký aspekt lidské inteligence. Jedna praktická aplikace je třeba *počítačové vidění.* Dnes počítače umí číst psaný text, rozeznávat lidské tváře, nebo třeba hledat ve fotkách dopravní značky, a to všechno s nadlidskou přesností. Algoritmus strojového učení například dostane 5 000 příkladů 10 různých dopravních značek a jeho výstupem bude postup, jak je od sebe rozpoznat.

Většina materiálů o strojovém učení je buď v angličtině nebo používá anglickou terminologii. Aby pro vás bylo snadnější si dle zájmu dohledat víc informací, budeme české termíny zavádět i s jejich anglickými ekvivalenty. Jestli byste chtě-

li strojovému učení věnovat víc samostudia, můžete si třeba najít materiály ke kurzu Machine Learning na Courseře.[29]

Druhy strojového učení

Strojové učení se dělí na tři široké kategorie: *učení s učitelem* (*supervised learning*), *bez učitele* (*unsupervised learning*) a *zpětnovazební učení* (*reinforcement learning*).

Zmíněný problém klasifikace dopravních značek patří pod *učení s učitelem.* Jakýsi *učitel* nám ukázal příklady a řekl nám „tohle je stopka“, „tohle je zákaz vjezdu“, a tak dále. Naším úkolem je podle těchto *trénovacích dat* vyrobit algoritmus, který bude fungovat dobře nejen na příkladech, které jsme dostali od učitele, ale i na těch, které jsme ještě nikdy neviděli.

Učení bez učitele se snaží najít nějaké pravidelnosti, ale nemá zvenku zadáno, čím se takové pravidelnosti budou vyznačovat. Když máme nějakou sadu dat, o které nic nevíme, učení bez učitele nám třeba může pomoct najít nějaké jejich „významné vlastnosti“, na které se pak můžeme zaměřit zvlášť. Mezi učení bez učitele patří třeba *detekce anomálií,* která hledá ve vstupních datech vzorky, které významně vybočují z „pravidelné struktury“. Když třeba máme datacentrum plné serverů, můžeme se snažit detekcí anomálií najít servery, se kterými je něco nějakým způsobem špatně, i když předem nevíme, jakým způsobem se mohou porouchat a jak se to projeví v parametrech, které měříme.

Zpětnovazební učení je o něco speciálnější. Používá se na učení chování v prostředí, ve kterém nemusí být podle výstupu naučeného systému hned jasné, jestli se chová chytře. Představte si třeba, že se snažíme naučit hrát Pacmana. Algoritmu říkáme, jak vypadá labyrint, kde se Pacman nachází, kde jsou duchové a kde je jídlo a on nám říká, kam chce, abychom šli. Chceme, aby se naučil, jak se má hýbat, aby sežral co nejvíc jídla a pokud možno neumřel. Když algoritmus řekne „jdi vlevo,“ tak nevíme hned, jestli to byl dlouhodobě dobrý krok, nebo špatný krok – to záleží na tom, jak se algoritmus zachová v budoucnosti (například jestli ho vlivem tohoto kroku za 5 tahů zabije duch). Pacman provádí posloupnost akcí v nějakém prostředí a snaží se, aby se nakonec naučil co nejlepší algoritmus pro hraní.

Učení bez učitele na to použít zřejmě nejde (protože máme konkrétní cíl – snažíme se maximalizovat snědené jidlo). Učení s učitelem se taky nehodí. Intuitivně bychom se totiž neučili hrát dobře – učili bychom se jenom napodobovat učitele. Pokud by tedy učitel byl třeba mistr světa v Pacmanovi, tak by ho náš naučený algoritmus neuměl porazit, protože se jenom naučí to, co umí učitel. Náš cíl je najít co nejlepší algoritmus, který je v rámci pravidel Pacmana možný.

Dnes si ukážeme pár základních algoritmů učení s učitelem.

[29] `http://coursera.org/learn/machine-learning`, nejbližší termín kurzu začíná 21. března.

Učení s učitelem

Zkusme se třeba naučit podle výšky a obvodu pasu předpovídat váhu lidí. Nejdříve najdeme nějaké dobrovolníky (nechť je jich N) a posbíráme jejich výšky, obvody a váhy. Vstupním informacím, podle kterých se snažíme předpovídat hmotnost, říkáme *příznaky* (*features*). Počet příznaků, tedy u nás 2, označíme jako p; první příznak je výška a druhý je obvod pasu. Uložíme si je do vektorů $x^{(1)}, x^{(2)}, \ldots, x^{(N)}$. To, co se snažíme naučit předpovídat – tedy váhu – si uložíme do $y^{(1)}, \ldots y^{(N)}$. Všem vstupním i výstupním datům, se kterými pracujeme, se říká souhrnně *dataset*. Označíme jej jako $\mathcal{D}$.

Když třeba $x^{(8)} = (1.87, 93)$ a $y^{(8)} = 85$, tak osmý dobrovolník měřil 1.87 m, měl obvod pasu 93 cm a vážil 85 kg. Informacím, které máme o jednom dobrovolníkovi, tedy dvojici $(x^{(8)}, y^{(8)})$, říkáme společně *vzorek* (*sample*). Používáme standardní vektorové značení, takže $x_1^{(8)} = 1.87$ a $x_2^{(8)} = 93$.

Většina aplikací učení s učitelem je *klasifikace* nebo *regrese*. Klasifikace je přiřazení vstupního vzorku do jedné z *kategorií*. Regrese je předpověď hodnoty nějaké funkce, která vede do reálných čísel. Předpovídání hmotnosti je regrese. Určování, jaký typ dopravní značky jsme vyfotili, je klasifikace.

Náš cíl je najít nějakou funkci f takovou, že pokud možno pro každý vzorek (x, y) je $y = f(x)$, nebo aspoň mezi nimi nebude velký rozdíl. Kromě toho ale chceme ještě jednu důležitou vlastnost: f by měla *dobře generalizovat*. Učení je hledání vhodné funkce f.

První požadavek, tedy aby f na známých vzorcích odpovídala správně, se dá splnit spoustou způsobů. Jeden z nich je třeba ten, že by si f při učení zapamatovala všechna trénovací data, a když by dostala vstup x, tak by se podívala, jestli tenhle vstup byl v trénovacích datech, a jestli byl, vrátila by jeho příslušný výstup, a jinak by vrátila třeba 9 999. Máte-li nepříjemný pocit, že na tomhle nápadu je něčo špatně, máte ho zcela správně.

Taková funkce f by fungovala perfektně na trénovacích datech, ale jakmile bychom chtěli předpovědět hmotnost někoho, koho jsme ještě neviděli, byla by zcela k ničemu. *Generalizace* je právě tato schopnost fungovat dobře i na lidi, které jsme při učení ještě neviděli.

Příznaky

Když chceme mít dobré předpovědi, velmi záleží na tom, abychom zvolili dobré příznaky. Musí obsahovat nějakou užitečnou informaci, na které závisí ta veličina, kterou předpovídáme. Když se snažíme naučit předpovídat hmotnost lidí, nejspíš nám pomůže vědět fyzikální vlastnosti, které s hmotností souvisí: třeba výšku, obvod pasu, věk nebo procento tuku v těle. Naopak nám asi nepomůže vědět barvu očí nebo oblíbenou kapelu.

Než spustíme algoritmus strojového učení, vyplatí se nejdřív zamyslet nad reálnou strukturou problému a zkusit pro něj vymyslet co nejužitečnější příznaky. Zkusme třeba předpovídat podle výšky a hmotnosti pravděpodobnost srdeční

příhody v následujícím roce. Samotná hmotnost a výška sice jsou užitečné informace (když vážím 250 kg, jsem rizikovější, než kdybych vážil 70 kg), ale lepší nejspíš bude si přidat jako příznak BMI ((hmotnost v kg)/(výška v m)^2). BMI zohledňuje, že různí lidé mají různou postavu – je asi zdravější vážit 100 kg a být vysoký 2 metry, než vážit 80 kg a být vysoký 1.5 metru.

Příznaky jde široce dělit na kategorické a numerické. Numerické příznaky se dají přirozeně vyjádřit jako číslo, třeba výška v centimetrech nebo barva pixelu v obrázku. Kategorické příznaky můžou být třeba krevní skupina nebo státní občanství. Existuje pro ně nějaký poměrně malý počet možných hodnot (třeba krevní skupiny jsou $\{0^+, 0^-, \mathrm{A}^+, \mathrm{A}^-, \mathrm{B}^+, \mathrm{B}^-, \mathrm{AB}^+, \mathrm{AB}^-\}$) a na těhle hodnotách nemusí dávat smysl obvyklé číselné operace. Mohli bychom sice třeba označit krevní skupiny místo názvů pořadovými čísly ($0^+ = 1, \ldots, \mathrm{AB}^- = 8$), ale operace nad takovým označením nejsou užitečné – sice nad naším označením můžeme tvrdit, že $(0^-) + (\mathrm{B}^-) = (\mathrm{AB}^-)$, ale to neodpovídá žádnému vztahu v reálném světě. Stejně tak není B^- v žádném smyslu „větší než“ 0^+. Oproti tomu třeba může dávat smysl porovnávat výšky dvou lidí nebo počítat jejich rozdíly.

Hodně algoritmů potřebuje, aby jejich vstupy byla čísla. Tehdy potřebujeme kategorické příznaky „zakódovat“ do číselných. Nejobvyklejší takové kódování z příznaku, který obsahuje jednu z K kategorií, vyrobí K číselných příznaků, jeden pro každou kategorii. Když vzorek patří do i-té kategorie, nastavíme i-tý příznak na 1 a ostatní na 0.

Předpokládejme, že existuje nějaký *skutečný* vztah $\hat{f}$, který z x dělá y (tedy z výšek a obvodů pasu dává hmotnost). Příznaky, které měříme, ale nemusí stačit na zcela přesnou odpověď: i když mají Jana a Katka stejnou výšku a obvod pasu, můžou mít jinou hmotnost, protože se Katka ráno nenasnídala. Existuje nějaký vliv vnějších příznaků, které neměříme (nebo možná ani z principu měřit nejdou – třeba máme nepřesnou váhu). Dá se to neformálně napsat jako $y = \hat{f}(x)+\varepsilon$: předpovídaná hodnota y se skládá ze složky, která závisí na x, a nějakého náhodného (a snad malého) ε, ve kterém jsou schované vlivy, které neumíme měřit.

Funkci f, kterou se snažíme naučit, se říká *model* – snaží se co nejpřesněji *modelovat,* co by dělala $\hat{f}$, kdybychom se jí mohli zeptat.

Měření chyby modelu

Po modelech chceme, aby byly co nejpřesnější a aby dobře generalizovaly. Chceme tedy, aby na *neznámém vzorku,* který *nebyl k dispozici učícímu algoritmu,* daly dobrou předpověď.

Existují různé metriky pro to, jak *dobrá* předpověď je. Většina z nich jsou nějaká míra *chyby.*

Pro naše předpovídání hmotnosti se třeba hodí velmi obvykle používaná *kvadratická odchylka.* Kvadratická odchylka modelu f na vzorku (x, y) je rovna

$(y - f(x))^2$. Intuitivně jí velké odchylky vadí mnohem víc než malé.

Pro klasifikační úlohy se jako metrika hodí *accuracy*.[30] Accuracy na jednom vzorku je 1 tehdy, pokud jej f předpoví správně, a 0, když na něm udělá chybu. Když třeba předpovídáme, jakou dopravní značku obsahuje obrázek, zajímá nás jenom, jestli najdeme tu správnou. Když si spleteme stopku se zákazem vjezdu, je to pro accuracy stejně špatné, jako bychom si ji spletli s přikázaným směrem jízdy.

Zatím jsme si ukázali definice dvou různých chyb modelu na jednom vzorku. Po modelu chceme, aby měl co nejmenší *střední hodnotu chyby,* neboli aby na náhodně vybraném neznámém vzorku byl co nejpřesnější.

Když máme nějaký model f, jak zjistíme střední hodnotu chyby na neznámém vzorku? Neznámé vzorky jsou pro nás nedostupné – nemůžeme jít změřit 7 miliard lidí a spočítat, jakou chybu průměrně děláme. Máme k dispozici jenom data od dobrovolníků. Dělá se to tak, že učícímu algoritmu nedáme všechna data, která máme. Rozdělíme dataset $\mathcal{D}$ na *trénovací množinu* $\mathcal{S}$ a *testovací množinu* $\mathcal{T}$, třeba v poměru 90 %/10 %. Učícím algoritmům dáme k dispozici jenom trénovací data. Testovací vzorky před ním skryjeme. Až nám učící algoritmus dá model f, spočítáme jeho průměrnou chybu na testovací množině. S trochou statistiky se dá ukázat, že průměrná chyba na testovací množině rozumně odhaduje střední chybu na *všech neznámých vzorcích.*

Úkol 1 [1b]: Je potřeba, aby rozdělení na testovací a trénovací množinu bylo *náhodné.* Ať dataset $\mathcal{D}$ obsahuje nejdřív 100 značek „stop“, pak 100 značek „dej přednost v jízdě“ a nakonec 100 značek „slepá ulice“. Vymyslete, co a jak by se mohlo rozbít, kdybychom testovací množinu vybrali nenáhodně.

Když třeba po modelu chceme malé kvadratické odchylky, chceme malou střední kvadratickou odchylku na neznámých datech. Tu odhadneme podle střední kvadratické odchylky na testovací množině $\mathcal{T}$:

$$E \approx E_{\mathcal{T}} = \mathrm{MSE}_{\mathcal{T}} = \frac{1}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} (y - f(x))^2.$$

Střední kvadratické odchylce se říká anglicky *mean square error* (*MSE*). Když nás zajímá accuracy na neznámých datech, odhadneme ji podobně: pomocí průměrné accuracy na $\mathcal{T}$.

Čím více dat dáme k dispozici algoritmu strojového učení, tím více se jim bude moct přizpůsobit a tím bude naučený model dávat přesnější předpovědi. Na druhou stranu, čím větší máme testovací množinu, tím přesněji chyba na testovací množině $E_{\mathcal{T}}$ odhadne skutečnou chybu na neznámých vzorcích E.

Vzpomeňte si na f, která si uložila všechna trénovací data do tabulky a na všechno kromě nich vrátila 9 999. Takový model má na trénovacích datech

[30] Kromě accuracy se pro klasifikátory měří i metriky *precision* a *recall.* České překlady tady bohužel nemají tak dobře zavedený význam, jako anglické termíny.

nulovou chybu ($E_{\mathcal{S}} = 0$). Na testovacích datech $\mathcal{T}$, která nemá v tabulce, ale odpoví strašně špatně, takže průměrná chyba na testovacích datech $E_{\mathcal{T}}$ nám správně řekne, jak strašně moc špatný tenhle model je.

Přeučení a porovnávání modelů

Pokud se naučíme model, který je hodně dobrý na trénovacích datech, ale podstatně horší na testovacích datech, může to být kvůli *přeučení* (neboli *overfittingu*). K přeučení dochází, pokud učícímu algoritmu umožníme, aby se příliš silně adaptoval na nějaké zvláštnosti trénovacích dat, které ale obecně neplatí.

Hodně algoritmů strojového učení funguje tak, že postupně po *epochách* víc a víc adaptuje model na trénovací data. Vyrobí tedy posloupnost modelů $f_1, f_2, \ldots$, ve které jsou modely postupně čím dál tím adaptovanější na trénovací data, ale po nějaké době se začnou přeučovat, a tedy začnou být míň užitečné pro obecné použití. Podobná situace nastane, když zkoušíme na jedněch datech různé algoritmy strojového učení a chceme z naučených modelů vybrat ten nejvhodnější.

Očividný přístup, jak vybrat z modelů $f_1, f_2, \ldots$ ten nejlepší, je změřit chybu všech modelů na testovacích množině a vrátit ten, který ji má nejmenší, ale tenhle přístup je rozbitý.

Proč je rozbitý? Vzpomeňte si, že testovací množina se používá k *odhadování skutečné chyby*. Když si z modelů vybereme ten, který na nejmenší testovací chybu, bude jeho testovací chyba *příliš optimistický* odhad skutečné chyby.

Ilustrujeme si tenhle problém malým myšlenkovým experimentem. Představte si, že naše modely jsou tři férové mince. Pokud padne panna, model dá správný výsledek, a když padne orel, dá špatný výsledek. Každý model tedy ve skutečnosti dá správný výsledek v 50 % případů.

Testovací množinu vyrobíme tak, že každou mincí desetkrát hodíme. Na první vyjdou 3 orli, na druhé 7 orlů a na třetí 5 orlů. Vybereme si tedy druhý model a budeme si o něm myslet, že je přesný v 70 % případů. To je víc než jeho skutečných 50 % – druhý model jenom měl to štěstí, že při našem testu naházel nejvíc orlů. Jsme moc optimističtí o jeho výkonu, a to o 20 %.

Co kdybychom neházeli třemi mincemi, ale 10 000 mincemi? Skoro určitě (s pravděpodobností asi 0.99994) se stane, že náhodou některá z nich nahází 10 orlů. Byli bychom *extrémně optimističtí* – mysleli bychom si, že jsme našli minci, na které vždycky padají orli, i když je férová. Přidání dalších modelů způsobilo, že náš odhad je *horší* – teď už se mýlíme o 50 % místo 20 %.

Správné řešení je udělat „výběr nejlepší mince“ a „odhad pravděpodobnosti“ jako nezávislé experimenty: nejdřív 10× hodit a vybrat nejperspektivnější minci, a pak jí znova 10× hodit a podle druhých hodů odhadnout její „cinklost“. Když jsme v první fázi vybrali minci, co naházela 10 orlů, ale ve skutečnosti je férová a jenom měla štěstí, tak druhá fáze pořád bude 10 hodů férovou mincí a nejspíš nám řekne, že skutečná pravděpodobnost padnutí orla na vybrané minci je 0.5.

Když vybíráme z více modelů ten nejlepší a pak chceme vědět, jaký výkon od něho můžeme očekávat na nových datech, jedno ze správných řešení je rozdělit data na tři množiny: *trénovací, validační* a *testovací* (třeba v poměru 80 %/10 %/10 %). Trénovací množina se dá k dispozici učícím algoritmům (nebo nad ní iteruje jeden algoritmus a leze z něj posloupnost modelů). Jako nejlepší model vybereme ten, co má nejmenší chybu na *validační* množině. Tím pádem „neznečistíme testovací množinu“ a budeme ji moci dál používat k dobrému odhadování skutečné chyby.

Chyby na validační a testovací množině odpovídají naměřeným „cinklostem“ v první a druhé fázi myšlenkového experimentu s mincemi.

Teď už se trochu vyznáme v základních termínech a souvislostích, tak se konečně vrhneme na něco programovacího.

Lineární regrese

Algoritmy obecného učení jsou si obecně hodně podobné:

- Předepíší, jaký obecný tvar budou mít modely f, které z nich budou padat. Tenhle předpis bude obsahovat vstupní vektor x a nějaký vektor *parametrů,* který se označuje β. Konkrétní model dostaneme, když do předpisu dosadíme parametry.
- Říkají, jakou chybu se snaží minimalizovat.
- Popisují, jak efektivně najít takové β, že předpis f s dosazenými parametry β bude mít co nejmenší chybu.

Lineární regrese konkrétně:

- Hledá *lineární model.* Lineární model je funkce f, která na vstupu x dá jako výstup $f(x) = \vartheta + \sum_{i=1}^{p} \alpha_i x_i$. Konstanty ϑ a α_i jsou parametry určující konkrétní lineární model. Dohromady je těchto parametrů $(p+1)$, tedy vektor parametrů β je $(p+1)$-rozměrný: nejdříve obsahuje p složek $\beta_1 = \alpha_1, \beta_2 = \alpha_2, \ldots \beta_p = \alpha_p$ a pak jednu složku $\beta_{(p+1)} = \vartheta$. Graf lineárního modelu je přímka v p-rozměrném prostoru .[31]
- Chyba, kterou minimalizuje, je *střední kvadratická odchylka* na trénovací množině, proto se se jí taky říká *metoda nejmenších čtverců*:

$$E = \mathrm{MSE}_{\mathcal{S}}(\beta) = \frac{1}{|\mathcal{S}|} \sum_{(x,y)\in\mathcal{S}} (y - f_\beta(x))^2.$$

- Efektivně najde dobrou hodnotu vektoru β pomocí *gradientové metody*.[32]

[31] Pokud znáte skalární součin, všimněte si, že $f(\vec{x}) = (\vec{x}, 1) \cdot (\vec{\alpha}, \vartheta) = (\vec{x}, 1) \cdot \beta$.

[32] Pro lineární regresi existuje i vzorec, ze kterého jde nejlepší model přímo spočítat, ale gradientová metoda je jednodušší na pochopení a obecnější – používá se ve spoustě dalších učících algoritmů.

Kdyby nám nezáleželo na času na naučení, stačily by nám první dva „moduly“. Mohli bychom si jenom vygenerovat biliardu modelů (třeba pro všechny hodnoty všech složek β od -100 do 100 s krokem 0.1), pro každý spočítat chybu a vybrat ten, co ji má nejmenší. To ale rychle přestane být efektivní pro hodně parametrů.

Úkol 2 [1b]: Jak závisí časová složitost tohohle hloupého přístupu „vygenerujeme si tabulku se spoustou modelů a vybereme ten nejlepší“ na velikosti trénovací množiny a počtu příznaků p?

Efektivnější minimalizace chyby vyžaduje trochu matematické analýzy. O kvadratické chybě se dají dokázat užitečné vlastnosti. Zaprvé když si vyjádříme E jako funkci parametrů $\beta = (\alpha_1, \ldots \alpha_p, \vartheta)$, zjistíme, že je *spojitá.* Zadruhé má E jediné lokální minimum, které je i její jediné globální minimum. Zatřetí nemá žádné inflexní body.

Gradientová metoda

Gradientová metoda (*gradient descent*) umí takové funkce minimalizovat. Obecně tato metoda funguje tak, že začneme v nějakém libovolném, třeba nulovém počátečním bodu β_0. Potom se podíváme, kterým směrem bychom mohli trošku posunout β_0 tak, abychom tím co nejvíc snížili E.

Je vám ten nápad nějaký povědomý? Je vám povědomý zcela správně. Ve druhém díle seriálu jsme si ukazovali, jak hledat „horolezením“ extrémy reálných funkcí. Gradientová metoda je horolezení velmi podobná. Liší se tím, že „má kompas“ – umí najít zlepšující bod v okolí efektivněji než náhodným zkoušením.

Kompasu se říká *gradient.* Gradient E v bodu β_0 se značí $\nabla E(\beta_0)$ a je to vektor, který říká, kterým směrem máme jít z β_0, abychom šli co nejstrměji směrem zvyšující se E. V každém bodu může obecně gradient vést jiným směrem.

Když gradient otočíme, dostaneme směr největšího *poklesu* E. Když je délka gradientu v bodě β velká, znamená to, že E v β rychle roste/klesá. Naopak malý gradient znamená, že se nacházíme v placaté oblasti a nulový gradient znamená, že β je lokální minimum, maximum, nebo inflexní bod. Protože E inflexní body nemá, značí v ní nulový gradient lokální minimum nebo maximum.

Krok gradientové metody číslo t začne v souřadnicích β_{t-1}. Spočítá si v tomhle bodě gradient $\nabla E(\beta_{t-1})$, nějaký jeho γ-násobek odečte od souřadnic a tím spočítá β_t.

Jak velké má být γ? Čím menší, tím déle bude gradientová metoda běžet, ale tím přesněji se zase trefí do lokálního optima. Když bude γ moc velká, budou kroky gradientové metody lokální minimum „přeskakovat“, což se projeví tak, že se po pár iteracích dostaneme k nekonečně velkým hodnotám parametrů a nic se nenaučíme. Pokud se vám taková věc stane, zkuste γ zmenšit o pár řádů. Hodně učících algoritmů má podobný parametr ovlivňující velikost jednoho učícího kroku. Většinou se mu říká *learning rate,* rychlost učení.

Po odečtení γ-násobku gradientu přejdeme na krok $(t+1)$: spočítáme gradient v β_t, odečteme jeho γ-násobek, a tak dále. Skončíme, když je splněné nějaké kritérium, například když už jsme počítali dost dlouho nebo když je gradient hodně malý. Pokud gradientová metoda skončí v místě, kde je gradient skoro nulový, našla lokální minimum.

Gradient je rovný nule i v lokálním maximu. Když bychom měli tu neuvěřitelnou smůlu, že bychom začali gradientovou metodu dokonale přesně v lokálním maximu, tak bychom hned skončili a vrátili lokální maximum. To se ale skoro určitě nestane. Stane se možná, že začneme gradientovou metodu blízko lokálního maxima. Potom nás od něho ale první krok trochu oddálí (protože jde směrem klesající E), druhý krok nás oddálí ještě víc a tak dále, takže v lokálním maximu neskončíme.

Gradientová metoda tedy skončí poblíž lokálního minima, které je kvůli vlastnostem E i globální minimum, a tedy bod, ve kterém skončíme, bude určovat parametry lineárního modelu s malou chybou.

Zbývá už poslední kousek skládačky: jak gradient E spočítáme?

Pokud bychom o E nevěděli nic, mohli bychom místo počítání gradientu zkusit jenom o trošku pohnout každou složkou β a vybrat to drobné pohnutí, které E nejvíc zmenší. To bude poměrně pomalé, ale jednoduché a bude to fungovat na všechna E, co jsou spojitá, mají jediné lokální minimum, které je zároveň i globální minimum a nemají inflexní body.

Gradient se dá pro naši definici chyby spočítat explicitně. Je to vektor o $(p+1)$ složkách a jeho i-tá složka se spočítá tímto průměrem přes všechny trénovací vzorky:

$$\nabla E(\beta)_i = 2 \cdot \frac{1}{|\mathcal{S}|} \sum_{(x,y)\in\mathcal{S}} x_i(f_\beta(x) - y).$$

Poslední složka (odpovídající členu ϑ) se spočítá jako:

$$\nabla E(\beta)_{(p+1)} = 2 \cdot \frac{1}{|\mathcal{S}|} \sum_{(x,y)\in\mathcal{S}} (f_\beta(x) - y).$$

V obou rovnicích je f_β lineární model určený parametry β.

Bitevní plán: Začneme v libovolném (třeba nulovém) bodě $\beta_0 = (\vec{\alpha}_0, \vartheta_0)$. Spočítáme gradient, odečteme jeho γ-násobek a opakujeme, dokud gradient není malý vektor nebo nás to nepřestane bavit. Uložíme výsledný model.

Je zaručeno, že gradientová metoda minimalizuje spojité funkce, které mají jediné globální a lokální minimum a nemají inflexní body. Často se ale používá i na spojité funkce, které tuhle vlastnost nemají. Potom může skončit v lokálním, ale ne nutně globálním optimu, nebo v inflexním bodě. V praxi nám to ale často nevadí – i tak dává použitelně dobré výsledky. Gradientové metodě také můžeme pomoct *náhodnými restarty* – pustíme ji několikrát za sebou z různých náhodně

zvolených β_0. Dá se pak doufat, že projdeme větší kus definičního oboru E, takže v nejlepším nalezeném bodě budeme blíž globálnímu minimu.

Úkol 3 [6b]: Stáhněte si ze stránky seriálu[33] dataset o spotřebě benzínu v USA. Soubor má 5 sloupců dat oddělených čárkami. Každý řádek jsou data z jednoho státu. První sloupec je vybíraná daň z benzínu v centech na galon, druhý je průměrná mzda v dolarech, třetí délka dálnic ve státu v mílích, čtvrtý je poměr obyvatel s řidičským průkazem a pátý je roční spotřeba benzínu ve státě v milionech galonů.

Naprogramujte lineární regresi a pomocí ní odvoďte vzorec na předpovídání spotřeby benzínu podle ostatních hodnot. Data nemusíte dělit na testovací a trénovací množinu. Pro nás fungovalo dobře začít v nulových parametrech a pak provést 100 000 iterací gradientové metody s $\gamma = 10^{-8}$. Pošlete svůj zdrojový kód a nalezenou střední kvadratickou odchylku na celém datasetu. Zkuste ji mít menší než 21 800.

Algoritmus K nejbližších sousedů

Algoritmus K nejbližších sousedů (anglicky *K-nearest neighbors* nebo *KNN*) je založený na jednoduché myšlence: když chceme podle mojí výšky a obvodu pasu předpovědět mou hmotnost, odhadneme ji podle lidí, kteří jsou mi podobní výškou a obvodem pasu.

Model si bude pamatovat všechna trénovací data a jeho jediný parametr je přirozené číslo K. Když nám přijde nový vstup x, najdeme ve trénovacích datech K vzorků, které jsou mu nejbližší. Podíváme se na známé výstupy pro nejbližší sousedy, nějak je zagregujeme a výsledek vrátíme.

Konkrétní použití se liší podle toho, jak nadefinujeme, že je vstup x „blízký“ k nějakému trénovacímu vzorku, a podle toho, jak z výstupů na nejbližších sousedech vyrobíme předpověď pro neznámý vstup.

Blízkost se může definovat třeba podle malé euklidovské vzdálenosti

$$d(x, x^{(i)}) = \sum_{j=1}^{p} (x_j - x_j^{(i)})^2.$$

Euklidovská vzdálenost se hodí na numerické příznaky. Pokud ji ale použijeme, je potřeba dát si pozor na to, aby rozdíly v příznacích byly stejně významné.

Co tím myslím? Vzpomeňme si na příklad s předpovídáním hmotnosti a podívejme se na tři vzorky s těmito vstupními příznaky:

příznak	$i = 1$	$i = 2$	$i = 3$
$x_1^{(i)}$: výška v m	1.93	1.92	1.4
$x_2^{(i)}$: obvod pasu v cm	83	86	82

[33] `http://ksp.mff.cuni.cz/viz/evoluce`

Dobrovolník 1 se liší od dobrovolníka 2 jedním centimetrem výšky a třemi centimetry obvodu pasu. Dobrovolník 1 se od dobrovolníka 3 liší 53 centimetry výšky a jedním centimetrem obvodu pasu. Nemusíme být experti na antropologii, abychom očekávali, že hmotnost dobrovolníka 1 bude podobnější hmotnosti dobrovolníka 2 než dobrovolníka 3.

Rozhodli jsme se, že první složka (výška) bude číslo v metrech a druhá složka (obvod pasu) v centimetrech. Když si spočítáme euklidovskou vzdálenost tak, jak jsme si ji nadefinovali, dostaneme:

$$d(x^{(1)}, x^{(2)}) = (1.93 - 1.92)^2 + (83 - 86)^2 = 9.0001$$

$$d(x^{(1)}, x^{(3)}) = (1.93 - 1.4)^2 + (83 - 82)^2 = 1.2809$$

Dobrovolník 3 je tedy navzdory naší intuici dobrovolníkovi 1 mnohem „euklidovsky blíž“ než dobrovolník 2. Je to tím, že jsme zvolili pro příznaky taková měřítka, že změna ve výšce je mnohem podstatnější než numericky stejně velká změna v obvodu pasu. Kdybychom tedy hledali blízké vzorky podle euklidovské metriky, neodpovídalo by „blízké okolí“ našim představám.

Tohle se dá částečně řešit tím, že si vstupní data *normalizujeme*. Jeden ze způsobů normalizace je spočítat pro každý příznak jeho minimum a maximum na trénovací množině, pak všechny jeho hodnoty přeškálovat do intervalu $[0; 1]$ a počítat euklidovské vzdálenosti na přeškálovaných příznacích. Stejné přeškálování provedeme, když máme udělat předpověď pro nový vstup.

Když jsou všechny naše příznaky diskrétní, můžeme definovat *blízkost* podle takzvané Hammingovy vzdálenosti. Hammingova vzdálenost dvou vektorů je počet jejich složek, které se liší.

Zbývá agregace předpovědí z okolí. Pro regresi je nejjednodušší vzít výstupy z nejbližších sousedů a vrátit jejich průměr. Pro klasifikaci můžeme třeba vrátit tu kategorii, která má mezi K sousedy nejvíce hlasů (a když jich nejvíce hlasů dostalo několik, vybereme z takových třeba tu první). Počítání průměru na kategoriích totiž nemusí být dobře definovaná operace.

Úkol 4 [6b]: Stáhněte si ze stránky seriálu dataset o kvalitě bílého vína. Obsahuje 12 sloupečků oddělených čárkami. První řádek popisuje význam sloupců. Prvních 11 sloupců obsahuje různé chemické vlastnosti vína a poslední jeho kvalitu mezi 1 a 10, kterou se naučíme předpovídat.

Naprogramujte algoritmus K nejbližších sousedů s normalizací příznaků a euklidovskou vzdáleností. Z kvality od K sousedů vytvářejte předpověď prostým aritmetickým průměrem.

Rozdělte dataset na trénovací, validační a testovací množinu v poměru 60 %/20 %/20 %.

Měřte střední kvadratickou chybu na trénovací a validační množině v závislosti na K. Zkuste všechna K od 1 do 40.

Jak na K záleží chyba na trénovací množině? Jak na K záleží chyba na validační množině? Proč?

Vyberte nejslibněji vypadající K a pomocí testovací množiny odhadněte, jaká bude skutečná accuracy vašeho modelu.

Tahle úloha nejspíš poběží celkem dlouho. Jestli jí chcete pomoct, můžete zkusit využít více jader procesoru. K řešení přibalte svůj program a výsledky.

Opisování od evoluce

Nejúspěšnější nám známý učenlivý systém byl vytvořen přibližně čtyřmi miliardami let evoluce. Inspirováni jeho architekturou jsme vymysleli mnoho forem *umělých neuronových sítí.* Obzvlášť v posledních několika letech jsme díky několika novým trikům a rychlejším paralelním počítačům dosáhli úžasných výsledků, mimo jiné ve zpracování obrazu a zvuku, ale třeba i přirozeného jazyka.

Informace v lidském lidským mozku posílají elektrické a chemické signály. Domníváme se, že nejdůležitější „výpočetní jednotkou" je neuron. Většina neuronů má nějaké „vstupní kanály", kterým se říká *dendrity,* a jeden „výstup" – *axon.* Dendrity tvoří jakousi „stromovitou strukturu", která sahá řádově stovky mikrometrů od těla neuronu. Zatímco dendritický strom některých neuronů má až tisíc větví, některé neurony jich mají pouhé jednotky. Délka některých lidských axonů dosahuje až 1 metru. Každý neuron má sice nejvýše jeden axon, ale ten se může větvit a posílat jeho signály až stovkám jiných neuronů.

Elektrické signály v nervovém systému jsou kódované pulzně. Neuron „sbírá" signály od svých vstupů a pokud se v neuronu nahromadí za určitou dobu dostatečné množství potenciálu, „vystřelí" signál na výstupu. Spojením mezi neurony se říká *synapse.* Některé synapse jsou *excitační* a některé jsou *inhibiční.* Signály poslané po excitačních synapsích „zvyšuje vnitřní potenciál" cílového neuronu, zatímco inhibice mu „brání vystřelit". Možná překvapivě se informace v mozku šíří relativně pomalu. V závislosti na typu signálu jsou to řádově jednotky až stovky metrů za sekundu.[34]

Systému neuronových spojení v mozku se souhrnně říká *konektom.* Můžeme si jej představit velmi zjednodušeně jako poněkud gigantický orientovaný graf, jehož vrcholy tvoří jednotlivé neurony. Hrany reprezentují spojení a vedou směrem, jakým se přenáší signály: od neuronu, který je vypálí, do neuronu, který je má na vstupu. Kromě toho, že hrany mohou reprezentovat excitační nebo inhibiční spojení, existuje samozřejmě velmi mnoho dalších parametrů, které mají vliv na

[34] Například signály od receptorů bolesti nebo teploty se šíří rychlostí asi 0.5 až 2 m/s. Oproti tomu signály od proprioreceptorů – detektorů relativní pozice jednotlivých částí těla – se šíří rychlostí mezi 80 a 120 m/s. Lidský mozek obecně funguje mnohem pomaleji než sériové počítače. Procesy v něm jsou však vysoce paralelizované.

zpracování signálů. Učení probíhá změnou vlastností konektomu, například přidáváním nových synapsí nebo změnami parametrů těch synapsí, co už existují. Když synapticky spojené neurony pálí společně, jejich spojení se posiluje.

Umělé neuronové sítě modelují skutečnost s různou úrovní detailu a mají mnoho různých aplikací, podle kterých se odvíjí jejich architektura. Začneme od nejjednoduššího modelu, který dělá něco užitečného.

Umělý neuron

Namísto posílání pulzních signálů budeme reprezentovat informaci pomocí čísel. Umělý neuron je pro nás jednotka, která má n číselných vstupů $x_1, \ldots x_n$ a jeden výstup y.

Neuron bude ze svých vstupů počítat *potenciál* ξ. Některé vstupy budou k potenciálu přispívat pozitivně, některé negativně. Výstup y bude záviset na potenciálu podle takzvané *aktivační funkce,* značené a: $y = a(\xi)$. Směr příspěvku, tedy míra excitace nebo inhibice, bude pro každý vstup jiná a bude určená takzvanou *vahou* (*weight*). Váha i-tého vstupu se značí w_i.

Standardně se potenciál počítá jako součet vážených příspěvků všech vstupů. K tomuto součtu se také přidá takzvaný *bias* (český překlad by mohl být „předsudek“ nebo „sklon“) ϑ:

$$\xi = \vartheta + \sum_i w_i x_i.$$

Aktivační funkce se používají různé. Vždy jsou definované na celém $\mathbb{R}$, omezené (většinou na $[-1;1]$ nebo $[0;1]$) a rostoucí (respektive neklesající). První model, který si ukážeme, bude vracet 1, pokud je $\xi \geq 0$, a jindy -1 (tedy jeho aktivační funkce je funkce signum).

Perceptron

Takovému umělému neuronu se říká *perceptron.* Jeho výstup je 1, pokud $\vartheta + \sum_i w_i x_i \geq 0$, a jindy -1. Jsou-li vstupy perceptronu dva, je $\vartheta + \sum_i w_i x_i = 0$ rovnice přímky ve dvou rozměrech. Perceptron touhle přímkou rozděluje dvojrozměrný prostor vstupů na dvě poloroviny. V jedné vrací 1 a ve druhé vrací -1. Přidáme-li další vstupy, analogicky perceptron dělí n-rozměrný vstupní prostor na dva poloprostory.

Perceptron lze použít jako jednoduchý klasifikátor dvou kategorií, pro něž bude jeho očekávaný výstup 1 a -1. Rozdělíme si vstupní příznaky trénovacích dat do *pozitivní kategorie* $\mathcal{P}$ a do *negativní kategorie* $\mathcal{N}$.

Ukážeme si *algoritmus perceptronového učení,* o kterém víme, že pokud jdou kategorie $\mathcal{P}$ a $\mathcal{N}$ od sebe bez chyb rozdělit perceptronem (neboli nějakou nadrovinou), pak náš algoritmus nakonec najde váhy nějakého takového perceptronu.

Nejdříve ke všem vzorkům v $\mathcal{P}$ a $\mathcal{N}$ přidáme jako novou složku na začátek jedničku. Tím pádem můžeme zapomenout na bias – stane se z něho nová první váha. Vzorky „vylepšené“ o jedničky označíme jako $\mathcal{P}'$ a $\mathcal{N}'$.

Potom sjednotíme vzorky do jedné množiny. $\mathcal{N}'$ mají být vzorky, pro které $\sum_i w_i x_i < 0$, a to platí právě tehdy, když $\sum_i w_i \cdot (-x_i) > 0$. Vyrobíme si množinu $\mathcal{R}$, která se bude skládat z $\mathcal{P}'$ a minus jedničkou vynásobených vektorů z $\mathcal{N}'$. Perceptron, ve kterém jsou všechny vektory v $\mathcal{R}$ v pozitivní polorovině, bude správně klasifikovat $\mathcal{P}'$ i $\mathcal{N}'$.

Zinicializujeme perceptron libovolnými vahami, třeba nulovými. Učení probíhá tak, že iterujeme přes množinu $\mathcal{R}$ a postupně perceptron učíme jednotlivé vzorky x. Chceme po něm, aby na každém vzorku bylo $\sum_i w_i x_i$ větší než 0.

Pokud na zkoumaném vzorku je $\sum_i w_i x_i > 0$, je vzorek perceptronem rozpoznaný správně, neděláme nic a jdeme na další vzorek. Pokud je suma menší než 0, pak ke každé váze w_i přičteme $x_i \cdot \gamma$ a jdeme se učit další vzorek. γ je tady opět rychlost učení, konstanta mezi 0 a 1. Pokud najdeme perceptron, který správně klasifikuje všechny vstupy, vrátíme ho. Pokud jenom chceme dobrý perceptron, který ne nutně klasifikuje všechno dobře, můžeme místo toho jenom běžet, dokud nám nedojde čas, a pak vrátit ten perceptron, který dokázal klasifikovat nejvíc vzorků po sobě správně.

Úkol 5 [6b]: Stáhněte si ze stránky seriálu dataset o kosatcích. Tento slavný „Iris dataset“ poprvé použil v 30. letech významný biolog a statistik Ronald Fischer. Každý řádek reprezentuje vlastnosti jednoho kosatce. První řádek popisuje význam sloupečků: první dva jsou délky a šířky kališních lístků, další dva jsou délky a šířky okvětních lístků. Máme za úkol předpovědět poslední sloupec – konkrétní druh kosatce: `setosa`, `versicolor`, nebo `virginica`.

Zkuste naprogramovat rozpoznávání kosatců pomocí perceptronů. Jeden perceptron dokáže od sebe rozpoznat jenom 2 třídy, budete muset být kreativní. Není jediné správné řešení – překvapte nás! Jakou zvládnete accuracy na testovacích datech? Naše autorské řešení umí z 60 trénovacích vzorků mít na zbytku datasetu 93.42 %.

Perceptrony jsou užitečné, ale pořád docela slabé – pokud mají kategorie „složitou hranici“ a nejdou lineárně oddělit, potřebujeme na jejich naučení našim modelům povolit složitější strukturu.

Zábavné věci se začnou dít třeba, když začneme neurony *vrstvit* do takzvaných *dopředných vrstevnatých neuronových sítí* (*feedforward neural networks*). První vrstva takové sítě jsou vstupní data. Druhá vrsta konzumuje výstup první vrstvy jako svůj vstup. Sedí v ní kupříkladu 10 neuronů a každý z nich má vlastní váhy a transformuje vstupy jiným způsobem. Výstupy druhé vrstvy jsou vstupy pro třetí vrstvu, a tak dále až do výstupní vrstvy. Informace se šíří jenom z nižších vrstev do vyšších. Vyšší vrstvy jsou schopné počítat složitější a složitější transformace vstupních dat.

Michal Pokorný

28-5-8 Neuronové sítě 14 bodů

V minulém dílu seriálu jsme se zabývali modelem biologického neuronu – perceptronem. Ukázali jsme si, jak takový model „naučit“ a použít k jednoduché klasifikaci, tedy roztřídění vstupů do několika skupin. V zadané úloze jste nakonec bojovali s tím, že jeden perceptron dokáže klasifikovat jen do dvou skupin.

KSP

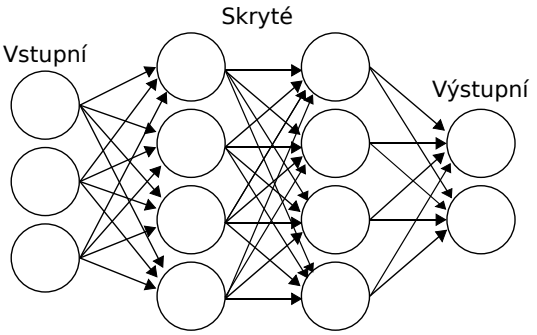
seriál

Dnes se podíváme na struktury z umělých neuronů složené, neuronové sítě, jejichž schopnosti jsou mnohem širší. Používají se k obecné klasifikaci, rozpoznávání zapamatovaných vzorů, organizaci dat do skupin podle podobnosti nebo třeba k řešení optimalizačních úloh.

Dopředné vrstevnaté neuronové sítě

Nejoblíbenější způsob, jak zapojit neurony do sítě, je rozdělit je do vrstev $1, 2, ..., N$. Mezi vrstvami napojíme výstup každého neuronu z vrstvy k na vstup každého neuronu z vrstvy $k + 1$. Každý z těchto vstupů má svoji váhu a každý z neuronů má svůj práh – stejně jako u jednoduchého perceptronu.

První vrstvě říkáme *vstupní*. Je trochu speciální, protože nepřijímá signály od jiných neuronů, ale přímo vstupní data a ta bez úpravy předává dál. Podobně poslední vrstva se nazývá *výstupní*, protože z ní čteme výstup sítě. Ostatní vrstvy jsou pro okolní svět *skryté*.



Tuto neuronovou síť bychom teď rádi naučili něco užitečného, tedy našli nastavení vah, které by zajistilo, že pro vstupy z trénovacích dat bude výstup sítě co nejpodobnější požadovanému výstupu. Tuto podobnost měříme pomocí chybové funkce neuronové sítě:

$$E = \frac{1}{2} \sum_{i}^{data} \sum_{j}^{výstupy} (y_{j,i} - d_{j,i})^2$$

Kde $y_{j,i}$ je výstup j-tého neuronu při i-tém vstupu sítě a $d_{j,i}$ je požadovaný výstup ve stejném případě. Na předpisu této funkce je zajímavé, že chyba (nebo

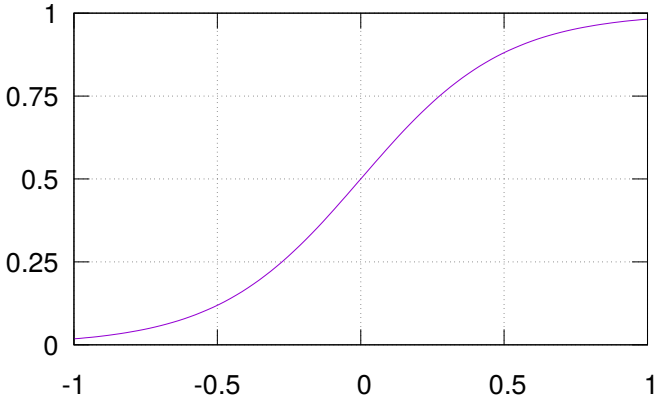
také odchylka) každého neuronu je umocněna na druhou, aby se zdůraznily velké chyby a zanedbaly malé. Sumy pak sečtou chyby ze všech neuronů pro všechna trénovací data. Polovina je v tomto vztahu jen proto, aby lépe vyšla jeho derivace – to zde ale dělat nebudeme.

Aby učící algoritmus mohl postupně snižovat chybu úpravou váhy neuronů, hodilo by se nám, aby aktivační funkce neuronu postupně a spojitě rostla s rostoucím potenciálem neuronu. Místo skokové funkce signum z minulého dílu tedy v našem seriálu použijeme sigmoidu:

$$f(\xi) = \frac{1}{1 + e^{-\lambda\xi}}$$

λ je v této funkci konstanta, jejíž nastavením můžeme měnit strmost funkce. K jejímu vlivu na učení se ještě dostaneme. Písmenkem ξ (*ksí*) značíme potenciál, e možná už znáte jako Eulerovo číslo.

Níže můžete vidět průběh sigmoidy pro $\lambda = 4$:



Výpočet výstupu sítě

Algoritmus pro výpočet výstupu sítě vychází přímo ze vztahů platících pro perceptron. Postupně počítá výstupy jednotlivých vrstev počínaje první skrytou (která vstupní vrstva data nijak nezpracovává). K výpočtu potenciálu ξ_j podle výstupů předchozí j-té vrstvy používá vztah z minulého dílu: $\xi_j = \sum_i y_i w_{ij}$. Potenciál se pak dosadí do aktivační funkce, v našem případě sigmoidy.

Nikoho, kdo četl předchozí díl, by teď nemělo překvapit, že zvlášť nepočítáme s prahem (anglicky *bias*). Ten je totiž schovaný v přidaném neuronu s konstantní hodnotou -1, ze kterého vede vstupní hrana s váhou odpovídající velikosti prahu.

Pro implementaci se může hodit si všimnout, že výpočet potenciálu je skalární součin dvou vektorů. Pokud váš programovací jazyk tedy vektory (v matematickém smyslu) a operace s nimi podporuje, můžete si jejich použitím občas

ušetřit trochu práce. Kromě toho můžete zjednodušením kódu ušetřit práci i čtenáři, což je v programování často ještě důležitější princip.

Úkol 1 [2b]: Vymyslete neuronovou síť modelující logickou funkci XOR. Síť bude mít dva vstupy a jeden výstup. Počet vrstev a skrytých neuronů je na vás, ale snažte se o minimalitu. Pošlete nám kresbu sítě včetně vah a vysvětlení, proč je vaše síť nejmenší možná.

Zpětná propagace

K učení, tedy minimalizaci chybové funkce E, se používá mnoho různých algoritmů. My se podíváme na nejzákladnější z nich, *zpětnou propagaci*. Nejdříve si popíšeme její myšlenku.

Na počátku náhodně nastavíme všechny váhy. Učení pak probíhá tak, že náhodně vybíráme z trénovacích dat, pro každý vstup necháme síť vypočíst výstup a porovnáme ho s požadovaným výstupem z trénovacích dat. Na základě tohoto porovnání upravíme váhy sítě tak, aby se snížila chyba sítě – výstup se přiblížil k požadovanému.

Postupně procházíme vrstvami od výstupní k vstupní a upravujeme váhy každé vrstvy tak, abychom snížili chybu té následující směrem k výstupní.

Převést tuto myšlenku do řeči matematiky vyžaduje trochu matematické analýzy, takže na to pro účely seriálu půjdeme trochu od lesa. Nejdříve vám ukážeme vzorce, které zpětná propagace na úpravu vah používá, a poté trochu objasníme jejich části.

Chceme určit novou váhu $w_{ij}(t+1)$ spoje z neuronu i do neuronu j v kroku $t+1$. Změna této váhy bude záviset na chybě neuronu j a výstupu neuronu i:

$$w_{ij}(t+1) = w_{ij}(t) + \alpha\delta_j y_i$$

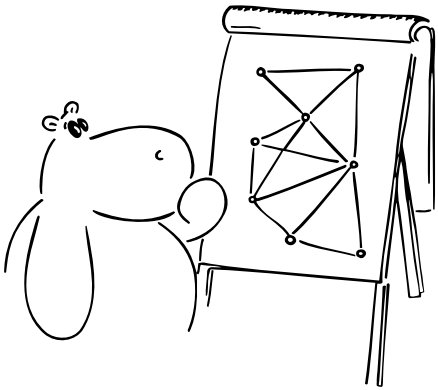
kde δ_j je chyba neuronu j. Ta se dá přímo spočítat pro výstupní vrstvu:

$$\delta_j = (d_j - y_j)\lambda y_j(1-y_j)$$

Pro skryté vrstvy je ale třeba ji odvodit na základě chyby, kterou už máme spočítanou pro sousední vrstvu směrem k výstupní:

$$\delta_j = \left(\sum_k \delta_k w_{jk}\right) \lambda y_j(1-y_j)$$

Kdy učení skončit a prohlásit síť za naučenou nebo naopak nepoučitelnou? Jedna z možností je sledovat chybu sítě. Pokud se průměrná chyba testovacích výstupů dlouho nesnižuje, lepší už to asi nebude.



Lepší možnost je vyzkoušet síť na vstupy, které v trénovacích datech nejsou. Takovým vstupům se říká testovací data, protože na nich testujeme naučenost sítě. Proč je lepší oddělit testovací a trénovací data? Představte si, že chcete neuronovou síť naučit třeba rozpoznat sudá od lichých čísel.

I pokud síť rozpozná naše trénovací čísla bez chyby, je možné, že se naučila jen seznam všech sudých nebo lichých čísel a jiná čísla může dál rozpoznávat špatně. Takovému stavu, kdy se síť příliš zaměřila na trénovací data, se říká *přeučení*. Naopak schopnosti sítě zobecnit řešení říkáme *generalizace*. Díky testovacím datům umíme tyto dva jevy rozlišit.

Pokud nám neuronová síť dává dobré výsledky na trénovacích i testovacích datech, znamená to že bude mít takto malou chybu i na dalších vstupech? Ani to ne! Učení totiž zastavujeme právě tehdy, když budou tyto chyby malé. Pokud chceme odhadnout chybu, se kterou bude síť pracovat pro další data, je dobré ji znovu změřit pro oddělenou sadu dat, *validační data*.

Uff, dat už potřebujeme pěknou hromádku, kde je ale vzít? To je společně s výpočetní náročností jednou z největších výzev strojového učení. Stroje svoji neobratnost v učení kompenzují velkým množstvím trénovacích dat. Zatímco dítěti stačí vidět pár čísel, aby se naučilo rozlišovat sudá a lichá, stroje jich potřebují řádově víc.

Mnoho současných technologických společností je schopno vytvářet „inteligentní“ řešení právě díky tomu, že mají od uživatelů svých služeb sesbíráno obrovské množství dat. Nám budou muset stačit veřejné *datasety* dostupné na internetu. Jak je rozdělit na trénovací, testovací a validační část? Náhodně; přičemž většina se obvykle používá na trénování.

Předzpracování dat

Narozdíl od perceptronů, neuronové sítě obvykle pracují s čísly v rozsahu $[0; 1]$, a i naše aktivační funkce má obor hodnot v tomto rozsahu. To pro nás znamená, že bychom si data pro neuronovou síť měli předzpracovat, aby byly

v tomto rozsahu. Těmto technikám předzpracování se říká *normalizace* a my ji budeme provádět stejně jako v minulém díle seriálu.

Pokud například chceme, aby neuronová síť pracovala s výškami dospělého člověka, které se v našich datech pohybují od 140 po 200 cm, přepočítáme tyto výšky tak, aby 140 odpovídalo nule, 200 odpovídalo jedné a vzájemné poměry výšek zůstaly stejné.

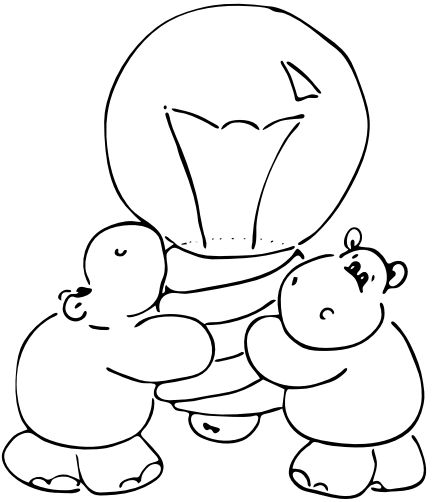
Pokud síť používáme ke klasifikaci do k skupin, často můžeme dosáhnout lepších výsledků, když místo jednoho výstupu sítě, který by udával číslo skupiny $1, .., k$, použijeme k výstupních neuronů. i-tá skupina se pak reprezentuje tak, že i-tý neuron má hodnotu 1 a ostatní 0 (nebo v blízkosti těchto hodnot).

Počáteční parametry

Chování zpětné propagace ovlivňuje několik parametrů a způsob generování počátečních vah. Parametry mají vliv hlavně na rychlost učení. Pokud učení nastavíte příliš rychlé, může algoritmus úplně ztratit schopnost chybu sítě zlepšovat a učení nebude fungovat. Konkrétní volba parametrů závisí na řešeném problému a často se s ní experimentuje.

Parametr λ nastavuje strmost aktivační funkce, v našem případě $f(\xi) = \frac{1}{1+e^{-\lambda\xi}}$. Jak si můžete sami vyzkoušet dosazováním, větší λ znamená strmější funkci. Strmá funkce pak zrychluje učení, protože menší potenciál (a tedy menší rozdíl vah) stačí k tomu, aby neuron vydal výstup blízko ± 1. Učení tedy pro velkou λ nemusí váhy měnit o tolik, což jej zrychlí.

Parametr α nastavuje rychlost změny vah v každém kroku učení. Jak je vidět ve vztahu pro aktualizaci vah: $w_{ij}(t+1) = w_{ij}(t) + \alpha\delta_j y_i$, větší α způsobí větší změny vah. Obvykle používáme $\lambda \in [0.5; 4]$ a $\alpha \in [0.2; 1]$.



Zbývá vyřešit volbu počátečních vah. Ty bychom chtěli nastavit tak, aby průměrný neuron reagoval na vstupy v rozsahu $[0; 1]$ opět potenciálem v rozsahu $[0; 1]$.

Nechceme tedy například, aby neurony s většími počty vstupů generovaly potenciály (v absolutní hodnotě) mimo zmíněný rozsah. Proto by takové neurony měly mít menší váhy svých vstupů. Doporučujeme volit počáteční váhy v rozsahu $\pm\frac{-2}{\sqrt{N}}$, kde N je počet vstupů neuronu. K přesnému odvození tohoto vztahu je potřeba smíchat trochu matematické statistiky a integrálního počtu, takže jej zde zatajíme.

Úkol 2 [8b]: Podívejme se znovu na dataset o kosatcích ze stránky seriálu. Zkusme klasifikovat kosatce pomocí neuronové sítě a porovnat přesnost s řešením pomocí perceptronů. Nezapomeňte na předzpracování dat. Původní autorské řešení, které mělo z 60 trénovacích vzorů na zbytku datasetu 93.42 % přesnost, byste měli bez problémů překonat.

Zkuste si pohrát s architekturou sítě (počtem neuronů a vrstev) a parametry a do řešení připište, na co jste přišli. V odevzdaném řešení rozdělte dataset v poměru 50:25:25 % pro trénovací, testovací a validační část.

Pokud jste úlohu z minulé série řešili, stáhněte si prosím dataset znovu, opravili jsme v něm dvě drobné chyby. Od minula také připomínáme, že každý řádek datasetu reprezentuje vlastnosti jednoho kosatce.

První řádek popisuje význam sloupečků: první dva jsou délky a šířky kališních lístků, další dva jsou délky a šířky okvětních lístků. Máme za úkol předpovědět poslední sloupec – konkrétní druh kosatce: `setosa`, `versicolor`, nebo `virginica`.

Úkol 3 [4b]:

Nakonec se podíváme na úlohu, která je pro náš „umělý mozek“ těžká: rozpoznání srdce. Mějme neuronovou síť se dvěma vstupy, jednou skrytou vrstvou s N neurony a jedním výstupem.

Síť bude na vstupu přijímat souřadnice x a y bodu v rovině a její výstup se bude blížit 1, pokud bod leží uvnitř srdce, a 0 v opačném případě. Srdce budeme pro naše účely znázorňovat známým piktogramem tvořeným čtvercem a dvěma půlkružnicemi umístěnými nad jeho dvěma sousedními stranami.

Zvolte si počet skrytých neuronů N (alespoň 6) a najděte nastavení vah této sítě. Rozměry a souřadnice srdce si zvolte libovolně. Úlohu můžete vyřešit jak pomocí zpětné propagace, tak pomocí tužky a papíru.

Jan Škoda

Recepty z programátorské kuchařky

Kuchařka první série – základní algoritmy

Tato naše kuchařka je nejzákladnější ze základních a je určena hlavně pro začínající řešitele. To však neznamená, že zkušenější řešitelé do ní nahlédnout nemůžou – třeba na nějakou konkrétní programátorskou techniku, kterou by si potřebovali osvěžit.

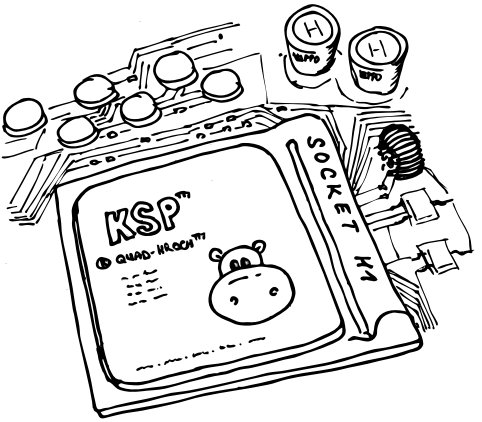
KSP

V první části kuchařky se seznámíme hlavně se základními principy programování, uchovávání dat v počítači a základy rychlé manipulace s nimi. Po přečtení této části bychom měli být schopní převést své myšlenky z hlavy na papír či do počítače a měli bychom vědět, proč je námi zvolený postup rozumný.

Druhá část nás poté seznámí se základními postupy, jak řešit určité konkrétní problémy. Naučíme se například, jak rychle vyhledávat v uspořádané posloupnosti hodnot, nebo jak si pomocí předpočítání usnadnit řešení těžké úlohy.

recepty

Většinu klíčových částí se pokusíme též ukazovat v podobě zdrojového kódu ve dvou různých jazycích (v nízkoúrovňovém C, kde je zápis blízký tomu, jak počítač doopravdy pracuje, a v Pythonu, ve kterém se píše o něco příjemněji). Nebudeme ale probírat základy syntaxe těchto jazyků, ty si případně můžete nastudovat jinde.[35] Pokud žádný z těchto jazyků neumíte, nezoufejte. KSP můžete řešit i bez toho, stačí když svá řešení důkladně slovně popíšete (konkrétní jazyk se pak můžete naučit až během dalších sérií).



[35] `http://ksp.mff.cuni.cz/study/odkazy.html`

Část první: Základní pojmy

Algoritmus a program

Pod tajemným slovem *algoritmus* se skrývá jen jiný výraz pro postup. Můžete si to představit jako příkaz od maminky „Běž do krámu, kup chleba, a když budou mít měkké rohlíky, tak jich vem tucet“.[36]

Takovýto příkaz klidně můžeme nazvat algoritmem, ačkoliv to bude asi znít nezvykle – pojem algoritmus se totiž používá hlavně ve světě počítačů. Je to tedy nějaká posloupnost základních příkazů, která řeší nějaký problém. Výběr konkrétního programovacího jazyka rozhoduje o tom, jaké základní příkazy budeme mít k dispozici. Většinou jsou ale skoro stejné.

Mezi základní příkazy patří:

- Manipulace s daty v paměti (uložení či načtení hodnoty, detailněji v další kapitole).
- Provedení nějakého numerického výpočtu $(+, -, *, /)$.
- Vyhodnocení určité podmínky a odpovídající větvení programu: *Pokud platí A, tak proveď B, jinak proveď C.* Přitom B i C mohou být klidně celé *bloky kódu*, tedy libovolně mnoho dalších základních příkazů.
- Opakování nějakého příkazu: *Dokud platí A, dělej B.* Takové konstrukci říkáme *cyklus* a podobně jako u podmínky může být B blok kódu, který se celý opakuje.
- Vstup a výstup programu (typicky vstup od uživatele z klávesnice či načtení vstupu ze souboru; výstup pak může znamenat vypsání výsledku na obrazovku nebo třeba zapsání dat do souboru).

Z těchto základních stavebních kamenů se skládá každý algoritmus. Programem potom rozumíme realizaci algoritmu v nějakém konkrétním programovacím jazyce.

U složitějších programů se pak často setkáte s problémem, že budete mít nějakou posloupnost příkazů, která se bude na spoustě míst programu opakovat, což zbytečně prodlužuje a znepřehledňuje kód.

Řešením tohoto problému je použití *funkcí*. Funkci si můžeme představit jako nějakou pojmenovanou část programu (s vlastní pamětí), kterou můžeme opakovaně použít tím, že ji v různých částech programu *zavoláme*. Funkci při zavolání předáme parametry (například seznam čísel), které se dostanou do její vnitřní paměti.

Funkce pak na základě obdržených parametrů může provádět nějaké operace, při kterých pracuje se svojí vnitřní pamětí (mluvíme o *lokální* paměti, změny

[36] A jako slušně vychovaní se tedy vydáte do krámu a koupíte tucet chlebů, protože měli měkké rohlíky :-)

v ní se neprojeví nikde mimo funkci). Na konci nám funkce může vrátit nějaký výsledek. Pokud funkce během svého běhu změní i nějaká data v *globální* paměti, či provede nějakou globální operaci (například výpis textu na monitor), mluvíme pak o funkci s *vedlejšími efekty* (neboli side-efekty).

Konkrétním příkladem může být funkce, která nám spočítá odmocninu ze zadaného čísla. Ta dostane jako svůj parametr číslo, uvnitř si provede nějaký výpočet, o který se jako uživatel funkce nemusíme starat, a jako výstup nám vrátí spočtenou odmocninu.

KSP

Reprezentace dat v počítači

recepty

Celkem často si v průběhu výpočtu našeho algoritmu potřebujeme pamatovat nějaké hodnoty. K tomu nám programovací jazyky dávají nástroj s názvem *proměnná*. Ta představuje jakési pojmenované místo v paměti (přihrádku), do kterého si můžeme data ukládat a pak je odtud zase načítat.

Typickým příkladem může být počítání součtu čísel, která nám uživatel zadá na vstupu. Na začátku nejdříve do nějakého místa v paměti uložíme hodnotu 0. Poté postupně, jak nám uživatel zadává čísla, tuto proměnnou pokaždé přečteme, k její hodnotě přičteme nově zadané číslo a výsledek opět uložíme na stejné místo.

Takovéto použití jedné proměnné je velmi jednoduché (tak jednoduché, že ho takto podrobně do řešení KSPčka nepište, není to potřeba), ale také celkem omezené. Co kdybychom si chtěli pamatovat třeba celou zadanou posloupnost čísel? Mohlo by nám stačit vyrobit si spoustu různě pojmenovaných proměnných, ale nejde to lépe? Jde.

Jednotlivé proměnné se mohou kombinovat do složitějších konstrukcí, které obecně nazýváme *datovými strukturami*. Zkusíme si ty nejzákladnější představit.

Pole

První datovou strukturou, kterou si představíme a která se na výše nastíněnou situaci náramně hodí, je *pole*. To představuje spoustu přihrádek (proměnných) naskládaných v paměti za sebou, ke kterým typicky přistupujeme přes jeden společný název pole a jejich pořadové číslo neboli index (jako `NazevPole[0], NazevPole[1], ...`).[37]

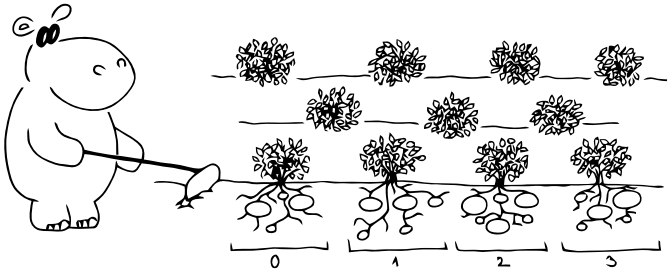
Ve většině základních jazyků je pole jen *statické*, tedy v okamžiku jeho vytváření musíme počítači říct, jak ho chceme velké. Některé vyšší jazyky ale nabízejí i pole, které se dynamicky zvětšuje, takovou konstrukci si ukážeme ve druhé části kuchařky.

Abychom nebyli omezeni jen jedním rozměrem, můžeme si klidně vyrobit pole dvourozměrné (případně obecně n-rozměrné). Dvourozměrné pole je vlastně tabulka hodnot, nazýváme ji také někdy *matice*, a může se nám hodit napří-

[37] Pozor, ve světě počítačů se velmi často indexuje od nuly, tedy první prvek má v tomto případě index 0.

klad při reprezentaci různých map (plán bludiště) nebo, jak uvidíme níže, pro reprezentaci dalších datových struktur.

U pole již má smysl přemýšlet, jak dlouho bude která operace trvat. Díky tomu, že jsou jednotlivé prvky v poli naskládané pevně za sebou, když se počítače zeptáme na obsah přihrádky `pole[42]`, přesně ví, na kterém místě v paměti se její obsah nachází, a proto nám hodnotu vrátí ihned.



Tomu budeme říkat *operace v konstantním čase* a budeme značit, že trvá čas $\mathcal{O}(1)$. Efektivitu programu totiž nepočítáme v sekundách (protože každý z nás má asi jinak rychlý počítač), ale v počtu základních operací, které musí program řádově vykonat. Více o časové složitosti si můžete přečíst v kuchařce o složitosti,[38] nejdříve však doporučujeme dočíst tuto kuchařku.

Přidání nového prvku na konec pole také zvládneme v konstantním čase. Problém je přidání nového prvku někam doprostřed (což se nám typicky stane, pokud budeme chtít udržovat hodnoty v poli seřazené a zároveň do něj vkládat nové). V takovém případě se totiž všechny prvky za vkládaným musí posunout o jednu pozici dál, aby se vkládaný prvek vešel na své místo. Taková operace tedy může pro pole délky N prvků trvat řádově až N kroků, což zapisujeme jako $\mathcal{O}(N)$ a říkáme, že je to vzhledem k N *lineární časová složitost*.

To je docela značná nevýhoda oproti struktuře, kterou si ukážeme za chvíli. Určitě ale pole nezavrhujme. Je to základní datová struktura, která nalezne použití ve spoustě programů, a jak si ve druhé části kuchařky ukážeme, můžeme ho použít třeba k rychlému hledání hodnoty metodou *binárního vyhledávání*. Nyní ale již slibovaná další datová struktura.

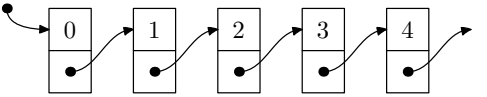
Spojový seznam a ukazatele

Pole jsme měli v paměti určené jenom tím, že počítač věděl, kde je jeho začátek a kolik místa v paměti zabírají jeho prvky. Při dotazování na konkrétní index pak podle indexu a podle velikosti prvků počítač přesně věděl, kam do paměti se má podívat, aby našel námi požadovaný prvek (to vše zvládl v konstantním

[38] `http://ksp.mff.cuni.cz/viz/kucharky/slozitost`

čase). Jednotlivé prvky si tedy vůbec nemusely pamatovat, kde se nachází jejich sousedi, protože všechny prvky seděly v paměti za sebou.

Představme si ale teď situaci, kdy by si každý prvek ještě pamatoval pozice sousedů. Pak bychom mohli mít prvky libovolně rozházené v paměti a jen by se na sebe vzájemně odkazovaly (první prvek by tvrdil, že druhý je na pozici X, druhý by tvrdil, že třetí je na pozici Y, a tak dále).

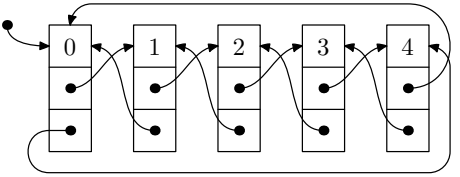


K lepšímu pochopení tohoto principu je důležité si vysvětlit, co to je *ukazatel* (nebo také *odkaz* či anglicky *pointer*). Každé místo v paměti počítače má své číselné označení, kterému říkáme *adresa*. Když si vytváříme nějakou pojmenovanou proměnnou, ta se vlastně odkazuje na určité místo v paměti a na tomto místě v paměti je její hodnota.

Co kdyby ale hodnota proměnné byla adresa nějakého jiného místa v paměti? Pak takové proměnné říkáme *pointer* a umožňuje nám vytvářet výše popsanou strukturu rozházených prvků v paměti.

Spojový seznam je tedy určený svým prvním prvkem (máme v jedné proměnné pointer na tento prvek, který se často nazývá *kořen*, protože z něj „vyrůstá“ zbytek struktury) a poté u každého dalšího prvku máme za sebou uloženou hodnotu tohoto prvku a odkaz (pointer) na další prvek. Odkazy mezi prvky mohou být i obousměrné, mohou vést dokola (poslední ukazuje na první) či mohou dokonce tvořit nějakou složitější strukturu (pak to ale již nebude čistý spojový seznam).

Pokud pointer nemá nikam ukazovat, realizuje se to odkázáním tohoto pointeru na adresu `NULL`. To skoro doslovně říká „Neukazuji nikam“.



Co nám takto vystavěná struktura umožňuje v porovnání s polem? Přístup na konkrétní prvek v ní stojí lineárně času, protože ho musíme „odkrokovat“ od prvního prvku (na který máme pointer), tedy musíme udělat až $\mathcal{O}(N)$ kroků. Pokud bychom však pointer na daný prvek už nějak měli, samozřejmě na něj můžeme přistoupit v konstantním čase.

Naopak přidávání prvků na konkrétní místo (i jejich odebírání) máme v podstatě zadarmo a spojový seznam můžeme rozšiřovat, dokud na něj máme v počítači paměť. Ve chvíli, kdy chceme přidat nový prvek za prvek, na který máme

pointer, jen šikovně přepojíme ukazatele. Pokud předtím ukazatele vedly $A \to B$, teď povedou $A \to C \to B$ (a při odebírání naopak).

Zde můžete vidět ukázku pointerů a spojových seznamů v jazyce C, kde jsou tyto věci mnohem více nízkoúrovňové (ale zato rychlejší):

```
#include <stdio.h>
#include <stdlib.h>
// Příkazy výše načetly do programu
// standardní knihovny a funkce z˜nich.

// Struktura pro prvek obsahující dopředné
// i zpětné odkazy. Zkráceně tomuto typu
// budeme říkat "tprvek".
typedef struct prvek tprvek;
struct prvek {
    int hodnota;
    tprvek *dalsi;
    tprvek *predchozi;
};

// Vytvoří nový prvek:
tprvek *novy(int i) {
    tprvek *aktualni =
        malloc(sizeof(tprvek));
    aktualni->dalsi = NULL;
    aktualni->predchozi = NULL;
    aktualni->hodnota = i;
    return aktualni;
}

// Odstraní prvek a vrátí pointer na další
// prvek (vrácení pointeru se hodí při
// odstraňování kořene):
tprvek *odstran(tprvek *aktualni) {
    if (aktualni->predchozi != NULL)
        aktualni->predchozi->dalsi =
            aktualni->dalsi;
    if (aktualni->dalsi != NULL)
        aktualni->dalsi->predchozi =
            aktualni->predchozi;

    tprvek *pomocna = aktualni->dalsi;
    free(aktualni);
    return pomocna;
}
```

```
// Vloží a vrátí pointer na nový prvek:
tprvek *vloz_za(tprvek *aktualni, int i) {
    tprvek *pomocna = aktualni->dalsi;
    aktualni->dalsi = novy(i);

    if (pomocna != NULL)
       pomocna->predchozi = aktualni->dalsi;

    aktualni->dalsi->dalsi = pomocna;
    return aktualni->dalsi;
}

// Použití:
int main(void) {
    tprvek *koren = novy(1);
    tprvek *aktualni = vloz_za(koren, 2);

    aktualni = koren;
    while (aktualni != NULL) {
        printf("%d\n",aktualni->hodnota);
        aktualni = aktualni->dalsi;
    }

    return 0;
}
```


Zde je ukázka spojových seznamů v Pythonu, kdybychom si je podobně jako v C chtěli naprogramovat sami (Python totiž obsahuje spoustu základních struktur již hotových, podívejte se na modul jménem `collections`):

```
class Prvek:
    def __init__(self, hodnota):
        self.hodnota = hodnota
        self.dalsi = None
        self.predchozi = None

class Spojak:
    def __init__(self):
        self.koren = None

    def Vypis(self, aktualni):
        if aktualni is not None:
            print aktualni.hodnota
            self.Vypis(aktualni.dalsi)

    def VlozPo(self, prvek, zaPrvek = None):
        if zaPrvek is not None:
            prvek.dalsi = zaPrvek.dalsi
```

```
                prvek.predchozi = zaPrvek
                zaPrvek.dalsi = prvek
            if prvek.dalsi is not None:
                prvek.dalsi.predchozi = prvek
            if self.koren is None:
                self.koren = prvek

        def Odstran(self, prvek):
            if prvek.predchozi is not None:
                prvek.predchozi.dalsi = \
                    prvek.dalsi
            if prvek.dalsi is not None:
                prvek.dalsi.predchozi = \
                    prvek.predchozi

# Použití:
prvekA = Prvek("A")
prvekB = Prvek("B")
prvekC = Prvek("C")
prvekD = Prvek("D")

seznam = Spojak()

seznam.VlozPo(prvekB)
seznam.VlozPo(prvekD, prvekB)
seznam.VlozPo(prvekC, prvekD)
seznam.VlozPo(prvekA, prvekC)
seznam.Odstran(prvekC)

seznam.Vypis(seznam.koren)
```


Fronta a zásobník

S použitím spojových seznamů (nebo v jednodušším případě dokonce i polí) můžeme zkonstruovat dvě velmi užitečné datové struktury, frontu a zásobník.

Fronta funguje tak, jak si ji asi každý z nás představuje: ten, kdo se do fronty postaví první, ten také první přijde na řadu. Můžeme si ji také představit jako trubku, do které na jedné straně sypeme nějaké věci a na druhé je odebíráme. Anglicky je též nazývaná *FIFO* (*„First In, First Out“*).

Praktickou realizaci uděláme jednoduše spojovým seznamem. Budeme si držet dva ukazatele, jeden na začátek seznamu, druhý na konec. Když se objeví nový prvek, který do fronty budeme chtít vložit, přidáme ho na konec, zatímco při odebírání z fronty využijeme druhého ukazatele a vezmeme prvek ze začátku.

Druhou velmi podobnou datovou strukturou je *zásobník*. Jak už ale plyne z anglického názvu *LIFO* (*„Last In, First Out“*), funguje spíše jako plný šuplík: Nahoru na něj přidáváme nové prvky, a když chceme nějaký odebrat, vezmeme

také zvrchu. To znamená, že první se na řadu dostane naposledy vložený prvek.

Implementace je velmi obdobná jako u fronty, jen bude ukazatel pouze jeden a bude ukazovat jenom na jeden konec spojového seznamu.

Knihovny

Tyto základní struktury už jsou často předpřipravené jako součást nějakých *knihoven* v daném jazyce. Knihovna je většinou sbírka nějakých navzájem souvisejících funkcí, které již někdo sepsal a které si můžeme do našeho programu načíst a používat. Ukázku načtení knihoven můžete vidět například ve výše zmíněném kódu v jazyce C.

Je ale velmi důležité rozumět tomu, jak knihovní funkce vnitřně fungují. Protože jedině když budeme vědět, co je jak rychlé a efektivní, budeme schopni psát rychlé programy.

Teď již víme, jak reprezentovat nejzákladnější datové struktury v počítači, ale mohlo by se nám hodit zastavit se ještě chvíli u dalších struktur. Tentokrát je už budeme studovat trochu teoretičtěji.

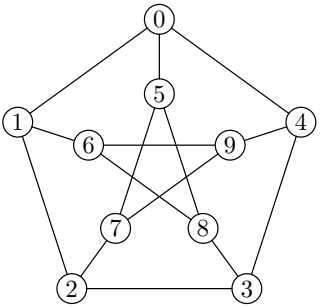
Stromy a grafy v informatice

Grafy

S nějakými grafy jste se již možná potkali, ale tento pojem je bohužel docela přetěžovaný. Jedním jeho významem jsou „koláčové grafy“ a jiné další diagramy znázorňující nějaký poměr (ať už to jsou výsledky voleb, nebo poměr lidí, kteří sledovali v televizi Večerníček).

Další význam můžeme nalézt v analytické matematice, kde se potkáme s grafy průběhu nějakých funkcí. My však nemáme na mysli ani jedno z výše zmíněných, teď se budeme bavit o *kombinatorických grafech*.

Grafem tedy máme na mysli nějakou množinu objektů, říkejme jim *vrcholy*, a nějaké vztahy mezi nimi. Tyto vztahy nazýváme *hranami* a jsou vyjádřené dvojicemi vrcholů, mezi kterými vedou. Ukázku takového grafu vidíme třeba na následujícím obrázku.



Jako praktickou ukázku grafu si můžeme například představit silniční síť nějakého státu: vrcholy budou města a hrany budou silnice, které mezi nimi vedou.

Občas se můžete setkat s pojmem *souvislý* graf. Ten znamená jen to, že mezi každými dvěma vrcholy existuje nějaká cesta. Pokud tomu tak není, je graf *nesouvislý* a dá se rozložit na několik menších grafů, které již souvislé jsou a říká se jim *komponenty souvislosti*.

Samotný graf poté můžeme doplnit tím, že si v každém vrcholu nebo na každé hraně budeme pamatovat nějakou hodnotu (například cenu nejlevnějšího benzínu ve městech a délku v kilometrech na silnicích). Pamatování si hodnot ve vrcholech je docela obvyklá technika a nemá speciální název, ale pokud budeme mít graf, který si pamatuje hodnoty na hranách, budeme o něm mluvit jako o *ohodnoceném grafu*.

Další možnou úpravou je, že každá hrana povede jen jedním směrem (jednosměrné silnice), takovým grafům říkáme *orientované* (pokud pak v orientovaném grafu chceme silnici oběma směry, prostě do něj přidáme dvě hrany, jednu v každém směru).

Poslední, co nám schází k praktickému použití grafů, je naučit se, jak je reprezentovat v počítači. Existuje několik možností (n bude značit počet vrcholů, m počet hran):

- **Seznam sousedů** – vrcholy grafu budeme mít uložené v poli a u každého vrcholu budeme mít (spojový) seznam čísel dalších vrcholů, do kterých z aktuálního vrcholu vede hrana. Zabírá místo $\mathcal{O}(n+m)$ a hodí se pro řídké grafy (tedy grafy, kde je m řádově stejné jako n).
- **Matice sousednosti** – tabulka $n \times n$, kde na souřadnicích $[i,j]$ je jednička (případně jiná hodnota, v případě ohodnoceného grafu), pokud z i do j vede hrana, a nula, pokud tam hrana není (u neorientovaných grafů je navíc matice symetrická – je jedno, jestli vezmeme $[i,j]$ nebo $[j,i]$). Hodí se pro husté grafy, kde $m \sim n^2$.
- **Matice incidence** – řádky reprezentují vrcholy, sloupce hrany. V každém sloupci jsou právě dvě jedničky – indexy vrcholů, mezi kterými hrana vede. Zabírá však $\mathcal{O}(mn)$ a její použití bývá dost neohrabané, takže je většinou lepší dát přednost jiné reprezentaci grafu. Je však dobré o ní vědět.

Grafy jsou velmi široké téma. Můžeme hledat jejich minimální kostry, můžeme v nich hledat nejkratší cesty či skrze ně pouštět pod tlakem vodu. Více o nich si tedy můžete přečíst v některé z našich specializovaných grafových kuchařek, které odkazujeme z našeho kuchařkového rozcestníku.[39]

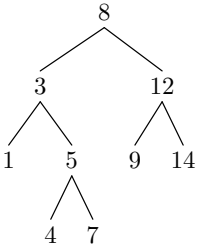
[39] `http://ksp.mff.cuni.cz/study/cooks/`

Stromy

Možná si říkáte, co má informatika u všech elektronů společného s lesnictvím? Kupodivu celkem mnoho a bez stromů bychom se v leckterém případě jen těžko obešli. Informatické stromy sice nejsou většinou tak zelené, mají ale, na rozdíl od svých dřevnatých sourozenců, mnoho jiných pěkných vlastností.

Strom je vlastně speciálním případem souvislého grafu, který neobsahuje žádnou kružnici (cyklus). To znamená, že mezi každými dvěma vrcholy stromu existuje právě jedna cesta.

Díky této vlastnosti pak můžeme nějaký zvolený vrchol prohlásit za *kořen* a strom za něj pomyslně zavěsit (tak, že strom roste od kořene směrem dolů), této operaci se říká *zakořenění*. Pak můžeme mluvit o tom, že z kořene směrem dolů (informatické stromy mají tradičně kořen nahoře) vyrůstají nějaké *podstromy*.



Pokud je strom zakořeněný, můžeme v něm mluvit o *hloubce* každého vrcholu, neboli o jeho vzdálenosti od kořene. Hloubka celého stromu je pak nejdelší ze vzdáleností od kořene k nějakému z *listů* (tak říkáme vrcholům, které již nemají žádné *syny*, tedy vrcholy, které by z nich vyrůstaly). Podle hloubky poté můžeme vrcholy stromu uspořádat do jednotlivých *hladin*.

Velmi často používáme stromy, které jsou nějak pravidelné. Příkladem jsou třeba *binární stromy*, které mají v každém vrcholu maximálně dva syny (říkáme jim *levý a pravý podstrom*). Reprezentovat se dají buď obecně jako každý jiný strom (v každém vrcholu spojový seznam podstromů), nebo velmi pěkně i v poli.

Stačí si pomyslně doplnit binární strom na *úplný* (to je takový, který má všechny své hladiny plné) a pak ho od kořene směrem dolů po hladinách očíslovat (kořen dostane číslo nula, jeho synové čísla jedna a dva, další hladina čísla tři až šest, atd.).

Můžeme si všimnout, že pokud si v takovém očíslování vezmeme jakýkoliv vrchol s číslem (indexem) i, tak jeho synové jsou právě vrcholy s indexy $2i + 1$ a $2i + 2$. Do pole níže je zapsaný binární strom z obrázku výše.

index	0	1	2	3	4	5	6	7	8	9	10
hodnota	8	3	12	1	5	9	14	–	–	4	7

Jak plyne z očíslování, pro úplný binární strom je uložení v poli efektivní a neplýtváme místem. Pokud ale strom úplný nebude, zůstanou nám v poli volná místa. Uložení v poli se tedy vyplatí jen pro stromy, které se od úplných příliš neliší.

Speciálním případem binárních stromů jsou pak ještě *binární vyhledávací stromy*. Jsou to normální binární stromy, pro něž navíc platí, že ať si vezmeme libovolný vrchol, budou hodnoty vrcholů v jeho levém podstromě menší než hodnota tohoto vrcholu, a hodnoty v jeho pravém podstromě naopak větší.

V takovém stromě pak zvládneme snadno vyhledávat. Budeme ho postupně procházet od kořene a v jednotlivých vrcholech budeme porovnávat hledanou a aktuální hodnotu a podle toho sestupovat do správného podstromu. Podobná technika je detailněji popsaná ve druhé části kuchařky, v kapitolce *Rozděl a panuj*.

Na složitější datové struktury stavějící na těchto základních (haldy, intervalové stromy, ...) se můžete podívat do některé z našich dalších kuchařek, na jejichž přehled jsme vás už odkázali o kapitolu výše.

Část druhá: Programátorské techniky

Tato část by měla sloužit jako rychlý přehled a ukázka různých technik, které se dají použít při řešení úloh z KSPčka, nebo při programování obecně.

Rekurze

Rekurze je velmi důležitá programátorská technika. V podstatě znamená definování nějaké věci (ať už je to nějaký objekt či postup výpočtu) pomocí sebe sama.

Rekurzivně může být například zadána nějaká datová struktura. Například stromy jsou pěkným příkladem rekurzivně definované datové struktury – každý vrchol stromu může mít syny, a každý z těchto synů je sám o sobě strom (tedy i osamocený vrchol bez synů je stromem).

Prakticky je to realizováno tak, že každý vrchol má svou hodnotu a pak ještě seznam ukazatelů vedoucích na další případné podstromy. S ukazateli jsme se již potkali a s jejich pomocí jsme si postavili spojový seznam. A přesně tak, spojový seznam je také ve své podstatě rekurzivní datová struktura.

Mimo rekurzivních datových struktur se ale často potkáváme i s rekurzivním postupem výpočtu nějakého programu, nejčastěji realizovaným ve formě funkce, která volá sama sebe (většinou s jinými parametry, jinak by to asi nemělo smysl), takové funkci se říká *rekurzivní funkce*.

U rekurzivních funkcí je nejdůležitější věc definovat nějakou *koncovou podmínku*, tedy podmínku, při níž už se rekurze zastaví. Jinak by se totiž mohlo stát, že by rekurze běžela donekonečna.

Přesněji rekurze by se i tak v nějakou chvíli zastavila, ale skončila by chybou, protože by jí došla paměť – každá volání funkce si totiž ukousne kus paměti (musí si pamatovat, kam se po skončení má vrátit) a pokud má rekurzivní funkce ještě nějaké lokální proměnné, musíme si někde uložit všechny lokální proměnné funkcí, z kterých jsme se doposud nevrátili.

Rekurzivní funkce a převod na nerekurzivní cyklus

Typickým příkladem rekurzivní funkce je výpočet Fibonacciho čísel. Ta jsou definována tak, že $f_0 = 1$, $f_1 = 1$ a n-té Fibonacciho číslo je součtem dvou předchozích ($f_n = f_{n-1} + f_{n-2}$). To nám dává posloupnost čísel 0, 1, 1, 2, 3, 5, 8, 13, ... pokračující donekonečna. Pokud toto přepíšeme do programového kódu, tak dostáváme následující zápisy:

V jazyce C:

```
int fib(int n) {
        if (n==0) return 0;
        else if (n==1) return 1;
        else return fib(n-1) + fib(n-2);
}
```

V Pythonu:

```
def fib(n):
        if n==0:
                return 0
        elif n==1:
                return 1
        else:
                return fib(n-1) + fib(n-2)
```

Jak vidíme, je přepis celkem přímočarý. Pokud by se nám však rekurze v nějakém případě nelíbila, můžeme se každé rekurze zbavit. Rekurzivní volání totiž můžeme šikovně přepsat na nějaký cyklus se *zásobníkem*.

Pak jen v cyklu odebíráme prvky ze zásobníku, dokud není prázdný, a za každé rekurzivní zavolání do zásobníku přidáme parametry, se kterými bychom naši funkci volali. Tímto postupem převedeme každou rekurzivní funkci na nerekurzivní.

Ještě doplníme poznámku, že ve většině programovacích jazyků každé volání funkce stojí nějaký čas, sice malý, ale když se volání provádí opakovaně, tak se to už nasčítá. Pro reálnou implementaci je tedy nejlepší pokusit se rekurzi převést na nerekurzivní volání, pokud to nějak rozumně jde.

Občas to jde dokonce i jednodušeji a bez zásobníku. Podívejte se na alternativní variantu výpočtu Fibonacciho čísel níže a rozmyslete si, co dělá.

V jazyce C:

```
int fib2(int n) {
        if (n==0) return 0;
        else if (n==1) return 1;

        int a = 0; int b = 1;
        for(int i = 2; i<=n; i++) {
                int c = a + b;
                a = b;
                b = c;
        }
        return b;
}
```

V Pythonu:

```
def fib2(n):
        if n==0:
                return 0
        elif n==1:
                return 1

        a = 0; b = 1
        while n>1:
                (a, b) = (b, a+b)
                n-=1
        return b
```

Jak vidíte, je i tato funkce elegantní a navíc běhá mnohem rychleji, než její rekurzivní varianta. Tato funkce běhá v $\mathcal{O}(n)$, kdežto rekurzivní varianta počítala stejné věci mnohokrát dokola (zkuste si nakreslit nějaký strom volání předchozí funkce, případně se podívat dopředu do kapitoly Předpočítané mezivýsledky).

Rekurzivní varianta tedy běžela až v čase $\mathcal{O}(2^n)$, což je pro velká n mnohem pomaleji než $\mathcal{O}(n)$ (avšak šla by celkem snadno zachránit, aby běžela také v $\mathcal{O}(n)$, zkuste si rozmyslet jak).

Backtracking

S rekurzí silně souvisí i pojem *backtracking*, česky by se snad dalo říci „metoda pokusu a omylu“. Tímto pojmem označujeme proces, kdy postupně zkoušíme všechny možnosti, jak vyřešit nějaký problém.

Metoda pokusu a omylu se tento proces nazývá proto, že pokud již nemůžeme pokračovat dál (třeba v případě, že v bludišti dojdeme do slepé uličky), vrátíme se kus zpět a zkusíme jinou (zatím nevyzkoušenou) možnost. Takto postupně zkusíme každou možnost, a buď nalezneme námi hledané řešení, nebo se vrátíme až na výchozí pozici a zjistíme, že řešení neexistuje.

Backtracking bývá často realizován pomocí rekurze, ukážeme si to na příkladu hledání rozkladu zadané částky na mince o hodnotách 5 Kč a 3 Kč (všimněte si, že v takto omezeném peněžním systému nejde složit třeba částka 7 Kč). Naše funkce dostane jako parametr zbývající částku a zkusí rekurzivně provést rozklad na jednotlivé mince:

V jazyce C:

```
bool rozloz(int castka) {
        // Koncová podmínka rekurze
        if (castka == 0) return true;
        else if (castka < 0) return false;
        else if (rozloz(castka-5)) {
                printf(" 5 Kc");
                return true;
        } else if (rozloz(castka-3)) {
                printf(" 3 Kc");
                return true;
        } else return false;
}
```

V Pythonu:

```
def rozloz(castka):
        if castka == 0:
                return True
        elif castka < 0:
                return False
        elif rozloz(castka-5):
                print " 5 Kc"
                return True
        elif rozloz(castka-3):
                print " 3 Kc"
                return True
        else:
                return False
```

V každém kroku zkusíme nejdříve použít pětikorunovou minci a zavoláme se na zbylou částku, a když náš rozklad nevyjde, zkusíme v tomto kroku použít ještě tříkorunu. Takto se rozhodujeme v každém kroku rekurze a případně se vracíme z neúspěšných větví výpočtu a zkoušíme další možnosti.

Takovým postupem ale vyzkoušíme až exponenciálně mnoho různých možností ($\mathcal{O}(2^n)$), což není moc rychlé. Proto je doporučováno se backtrackování raději vyhnout, nebo ho nějak chytře vylepšit. Je však dobré o backtrackování vědět, protože existují problémy, které efektivněji řešit neumíme.

Rozděl a panuj

Jednou ze základních technik je rozdělení složitějšího problému na menší části, které opět můžeme rozdělit na menší a tak dále, dokud se nedostaneme k problémům tak malým, že je už umíme triviálně vyřešit.

Binární vyhledávání v poli

Představte si, že máme seřazené pole n prvků a chceme zjistit, jestli se v něm nachází prvek s hodnotou k. Určitě můžeme projít celé pole v lineárním čase (tím, že budeme brát jeden prvek za druhým a kontrolovat, zda je roven hodnotě k), ale to je zbytečně pomalé a nevyužívá toho, že máme pole seřazené.

Můžeme totiž začít s velkým problémem a ten postupně zmenšovat na stále menší a menší. Nejdříve hledáme k v celém poli. Podíváme se na jeho prostřední prvek:

- Pokud je roven k, jsme hotovi.
- Je-li větší než k, víme, že se k musí nacházet nalevo od něj. Můžeme tedy hledat znovu, ale tentokrát se omezit jen na levou polovinu pole.
- Analogicky, je-li menší než k, můžeme hledat jen v pravé polovině.

Když tímto postupným dělením problémů na menší dojdeme až k poli o velikosti jednoho prvku, stačí tento prvek jenom porovnat, dál už se pole nepokoušíme rozdělovat.

Jelikož se nám každým krokem problém zmenší na polovinu, tak se maximálně po $\log n$ krocích dostaneme na pole velikosti jedna. Říkáme, že algoritmus má *logaritmickou časovou složitost*, píšeme $\mathcal{O}(\log n)$.[40]

Prakticky postup provádíme tak, že si udržujeme levý a pravý okraj aktuálně zpracovávaného úseku a postupně je k sobě přibližujeme.

[40] Pokud není řečeno jinak, znamená pro nás v informatice značka log *dvojkový logaritmus*, což je funkce opačná k funkci 2^n a roste o hodně pomaleji než funkce lineární. Pro velká n platí: $1 < \log n < n$ a například $\log 2 = 1, \log 8 = 3, \log 1024 = 10$.

Ukázka hlavní smyčky v C:

```
int pole[] = {1,2,5,7,12,16,42};
int hledane = 8;
int L = 0, R = 6;
int x;

do {
    int prostredni = (L+R)/2;
    x = pole[prostredni];
    if (x == hledane)
        printf("Pole obsahuje hledane\n");
    else if (x < hledane)
        L = prostredni + 1;
    else
        R = prostredni;
} while (L < R && x != hledane);

if (x != hledane)
    printf("Hledane neni v poli\n");
```


Ukázka v Pythonu jako funkce vracející index prvku nebo -1, pokud hledané číslo nenalezne:

```
def bin_vyhled(pole, hledane, L=0, R=None):
    if R is None:
        R = len(pole)
    while L < R:
        prostredni = (L+R)//2
        x = pole[prostredni]
        if x < hledane:
            L = prostredni + 1
        elif x > hledane:
            R = prostredni
        else:
            return prostredni
    return -1

# Zavolání:
print bin_vyhled([1,2,5,7,12,16,42], 8)
```

Další aplikace

Další typickou aplikací postupu rozděl a panuj je například třídění posloupnosti pomocí *Mergesortu*. Ten v základu funguje tak, že každou posloupnost, kterou dostane k setřídění, rozdělí na poloviny a každou z nich setřídí rekurzivním zavoláním sebe sama. Zanořování se zastaví ve chvíli, kdy třídíme posloupnost délky jedna (ta už je z podstaty setříděná). Pak jen v každém kroku ze dvou

setříděných menších posloupností vyrobí jejich sléváním setříděnou posloupnost dvojnásobné délky.

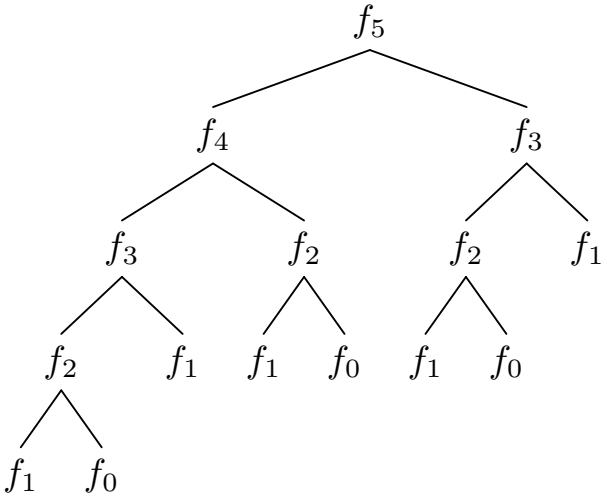
Více se o metodě Rozděl a panuj můžete dozvědět ve stejnojmenné kuchařce.[41]

Předpočítané mezivýsledky

Motivací k této kapitole je následující motto: „Proč počítat něco vícekrát, když nám to stačí spočítat jednou a zapamatovat si to?“.

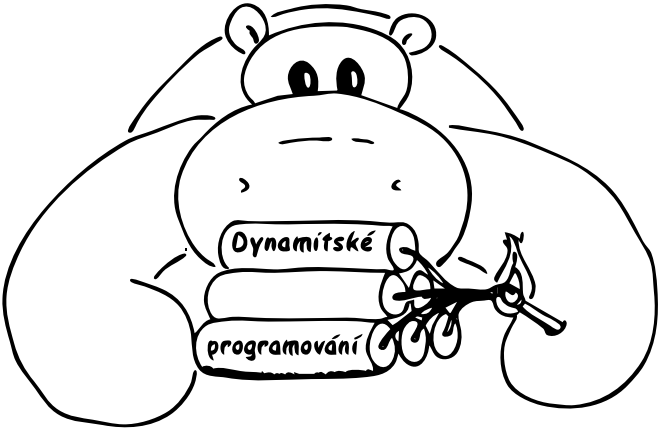
Velmi často se totiž setkáváme s tím, že něco počítáme stále dokola. Jako příklad si můžeme vzít naši rekurzivní implementaci počítání Fibonacciho čísel z kapitolky Rekurze.

Když se podíváme na výpočet čísla `fib(5)`, vidíme, že pro něj voláme `fib(4)` a `fib(3)`, `fib(4)` volá `fib(3)` a `fib(2)`, `fib(3)` volá `fib(2)` a `fib(1)` a tak dále. Všimli jste si, kolikrát se nám tyhle výpočty opakují? Některá Fibonacciho čísla spočteme totiž zbytečně mnohokrát.



Kdybychom si je namísto opakovaného počítání někde pamatovali, mohli bychom pak odpověď na dotaz na již vypočtené číslo vytáhnout jako králíka z klobouku v konstantním čase. Zavedením jednoho globálního pole, ve kterém si tyto hodnoty pro jednotlivá n budeme pamatovat, nám sníží časovou složitost z $\mathcal{O}(2^n)$ na pěkných $\mathcal{O}(n)$. Takovému postupu se obecně říká *dynamické programování*.

[41] `http://ksp.mff.cuni.cz/viz/kucharky/rozdel-a-panuj`



Nejprve uveďme na pravou váhu výraz „dynamické“ v názvu. Nevystihuje tak úplně podstatu této techniky a jeho historické pozadí je celkem složité, avšak dnes je tento název již tak zažitý, že se už pravděpodobně nezmění.

Slovo „dynamické“ částečně odkazuje na to, že se dynamicky (za běhu programu) postupně staví řešení jednodušších problémů, která jsou následně použita pro řešení složitějších. Jeho hlavní podstatou je tedy ukládání a opětovné použití již jednou vypočtených údajů.

Hodí se na úlohy, které se dají dělit na podúlohy, které jsou si podobné a mohou se opakovat. Výsledky takovýchto podúloh si poté ukládáme a při dotazu na stejnou podúlohu vrátíme jen uložený výsledek a výpočet již neprovádíme.

Pro další prohloubení znalostí můžete na našem webu nahlédnout do další kuchařky, tentokrát nesoucí (překvapivě) název Dynamické programování.[42]

Prefixové součty

Velmi často se nám hodí si ještě před samotným výpočtem předpočítat a uložit nějaké hodnoty, které poté použijeme.

Představme si například problém nalezení souvislého úseku s největším součtem v nějaké posloupnosti kladných i záporných čísel. Že to není úplně jednoduchý příklad, si ukažme na následující posloupnosti:

$$1, -2, 4, 5, -1, -5, 2, 7$$

[42] `http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani`

Máme zde dvě ryze kladné souvislé posloupnosti, každou se součtem 9 (4, 5 a 2, 7). Ale přesto je výhodnější vzít i nějaké záporné hodnoty a vytvořit tak souvislou posloupnost o součtu 12 (zkuste ji nalézt).

Mohlo by nás napadnout, že prostě zkusíme vzít všechny možné začátky a všechny možné konce. To nám dává $\mathcal{O}(n^2)$ možných posloupností (máme n možných začátků a ke každému z nich řádově n možných konců), pro každou posloupnost si spočteme součet (to zvládneme v $\mathcal{O}(n)$) a budeme si pamatovat ten největší nalezený. Celý náš postup tak trvá $\mathcal{O}(n^3)$.

To není pro takhle jednoduchou úlohu zrovna ten nejpěknější čas, zkusme ho zlepšit. Ukážeme si, jak vypočítat součet libovolné posloupnosti v konstantním čase. Celý princip je vlastně až kouzelně jednoduchý, ale zároveň velmi mocný. Na začátku výpočtu si do pomocného pole P stejné délky jako posloupnost na vstupu (té říkejme S) uložíme takzvané *prefixové součty:* i-tý prefixový součet je součet prvních $i+1$ prvků S, neboli $P[i] = S[0] + S[1] + \ldots + S[i]$.

Pro náš ukázkový případ a pro vstupní pole označené S by to dopadlo takto:

i	-1	0	1	2	3	4	5	6	7
$S[i]$		1	-2	4	5	-1	-5	2	7
$P[i]$	0	1	-1	3	8	7	2	4	11

Pole prefixových součtů umíme získat v linerárním čase – prostě jen od začátku procházíme vstupní pole, počítáme si průběžný součet a ten zapisujeme.

Součet libovolného úseku $a \ldots b$ pak získáme v konstantním čase jako prefixový součet od začátku do indexu b minus prefixový součet od začátku do indexu a. Zapsáno programově to pak je:

```
soucet = P[b] - P[a-1];
```

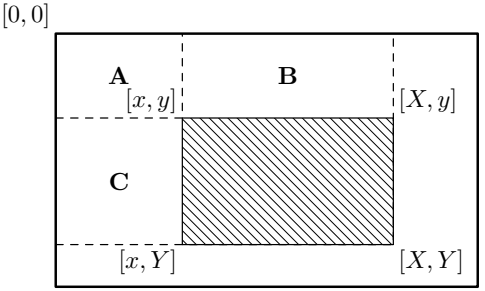
To nám umožňuje snížit čas potřebný na řešení této úlohy na $\mathcal{O}(n^2)$. To je už lepší čas, prozradíme však, že tuto úlohu lze řešit dokonce v lineárním čase, ale to je již nad rámec této kuchařky.

Dvourozměrné prefixové součty

Prefixové součty se dají zobecnit i do více rozměrů, ale princip je vždy stejný. Například dvourozměrné prefixové součty u matice fungují tak, že si předpočítáme součty podmatic začínajících levým vrchním políčkem a končící na indexu $[x, y]$.

Z toho je vidět, že prefixový součet zpravidla obsadí stejně velký prostor jako původní data, v tomto případě tedy budeme mít matici hodnot prefixových součtů končících na zadaných souřadnicích. Jak ale získat součet nějaké podmatice, která se nachází někde „uprostřed“ naší matice?

Použijeme stejný princip jako u jednorozměrného případu: Přičteme větší část, kterou chceme započítat, a odečteme od ní části, které započítat nechceme. Pro případ podmatice začínající vlevo nahoře na pozici $[x, y]$ a končící napravo dole na $[X, Y]$ to ilustruje následující obrázek:



Nejdříve přičteme celý prefixový součet končící na pozici $[X, Y]$. Tím jsme ale započítali i části A, B a C z obrázku, které započítat nechceme. Tak odečteme prefixové součty končící na indexech $[X, y]$ a $[x, Y]$. Ale pozor, teď jsme odečetli jednou $A + B$ a jednou $A + C$, tedy část A (prefixový součet končící na pozici $[x, y]$) jsme odečetli dvakrát, musíme ji proto ještě jednou přičíst.

Celý vzorec tedy vypadá takto:

```
soucet = P[X,Y] - P[X,y] - P[x,Y] + P[x,y];
```

Tento princip přičítání a odečítání se dá zobecnit i pro libovolné vyšší rozměry, ale chce to již trošku představivosti, co se má přičíst a kolikrát. Říká se tomu také *princip inkluze a exkluze* a najde použití nejen u vícerozměrných prefixových součtů.

Vyvážení délky předvýpočtu a hlavního výpočtu

Správně vyvážit, kolik času můžeme věnovat na předvýpočet a kolik času na hlavní výpočet, je velice důležitá věc a spousta i zkušenějších řešitelů v tom občas chybuje. Přitom to při troše počítání není vůbec nic složitého.

Jako první je potřeba vědět, kolikrát nám předvýpočet během běhu programu pomůže. Předvýpočtem si totiž vybudujeme za nějaký čas určitou datovou strukturu, pomocí které pak dokážeme rychle odpovídat na zadané dotazy.

Označme si počet takovýchto dotazů, které program za běhu dostane, jako Q. Buď to může být hodnota přímo ze zadání typu „Zkonstruujte datovou strukturu pro n hodnot, která zvládne rychle odpovídat na dotazy daného typu, a očekávejte řádově m dotazů“, nebo se může jednat o nějaký interní dotaz v rámci běhu programu (příklad interního dotazu je třeba výše uvedený algoritmus na hledání souvislé podposloupnosti s co největším součtem, který se za běhu ptal na součty nějakých úseků).

Dále si označme jako $\mathcal{O}_p$ čas, který nám zabere předvýpočet a jako $\mathcal{O}_q$ čas, který nám ušetří každý předvypočítaný dotaz. Celkový čas, který ušetříme, je pak vlastně $Q \cdot \mathcal{O}_q$. Pokud je tento čas řádově větší než $\mathcal{O}_p$, pak má předvýpočet smysl.

Naopak nemá smysl trávit předvýpočtem řádově více času, než by trval samotný výpočet bez použití předpočítaných hodnot.

Jako příklad uvažujme problém o velikosti n, u kterého máme tři možnosti, které můžeme zvolit. Můžeme buď předvýpočet úplně vynechat a na každý dotaz odpovídat v čase $\mathcal{O}(n)$, nebo provést předvýpočet v čase $\mathcal{O}(n \log n)$ a poté odpovídat na každý dotaz v čase $\mathcal{O}(\log n)$, nebo provést předvýpočet v čase $\mathcal{O}(n^2)$ a pak odpovídat v čase $\mathcal{O}(1)$ na dotaz.

Kdy se nám co hodí?

- Pokud bychom dostali jen jeden dotaz, nemá smysl si cokoliv předpočítávat a odpovíme jednou v čase $\mathcal{O}(n)$.
- Pokud bude dotazů řádově n, má smysl použít první předvýpočet. Pak budeme mít čas na předvýpočet i na samotný výpočet $\mathcal{O}(n \log n)$, což je optimum.
- Naopak pokud by dotazů bylo řádově n^2 nebo víc, tak se nám již první předvýpočet nevyplatí, dostali bychom se totiž na čas $\mathcal{O}(n^2 \log n)$. Zde se hodí použít druhý, delší předvýpočet a pak se dostat na časovou složitost $\mathcal{O}(n^2 + n^2 \cdot 1) = \mathcal{O}(n^2)$.

Hladové algoritmy

Věřte nebo ne, ale i počítač se někdy cítí hladový. Po namáhavé práci mu můžeme dopřát to potěšení, aby si ukousl co největší kus dat. A ukážeme, že někdy je to i ku prospěchu. Řeč bude o *hladových algoritmech*.

Takovými algoritmy rozumíme ty, které hledají řešení celé úlohy po jednotlivých krocích a splňují následující dvě podmínky:

- V každém kroku zvolí lokálně nejlepší řešení.
- Provedené rozhodnutí již nikdy neodvolává (tedy nebacktrackuje).

Lokálně nejlepší řešení je takové, které v aktuálním kroku vybere tu možnost, která nám na tomto místě nejvíce pomůže (bez jakéhokoliv ohledu na globální stav). Může to být třeba nejvyšší hodnota, nebo nejkratší cesta v grafu.

Pokud ale od hladového algoritmu chceme, aby nám našel globálně nejlepší řešení, musí naše úloha splnit předpoklad, že si výběrem lokálně nejlepšího řešení nezhoršíme to globální. Tento předpoklad se nedá formulovat obecně a je nutné se nad ním zamyslet zvlášť u každé úlohy.

Příklady hladových algoritmů

První hladovou úlohou bude (jak jinak) automat na jídlo vracející mince. Automat by měl vracet peníze nazpět tak, aby vrátil daný obnos v co možná nejmenším počtu mincí. Pro náš měnový systém (máme mince hodnot 1, 2, 5, 10, 20 a 50 Kč) lze tuto úlohu řešit hladovým algoritmem – v každém kroku algoritmu vrátíme tu největší minci, kterou můžeme (tedy pro vrácení 42 Kč to bude $42 = 20 + 20 + 2$ Kč).

Měnové systémy většiny států jsou postavené tak, aby fungovaly takto pěkně, neplatí to ale obecně. Zkusme si vrátit 42 Kč, pokud bychom měli jen mince

hodnoty 20, 10 a 4 Kč. Správným řešením je $42 = 20 + 10 + 4 + 4 + 4$ Kč, hladový algoritmus by ale zkusil vrátit $42 = 20 + 20 + \ldots$ a tady by selhal.

Dále se velmi často dají hladovým algoritmem řešit nějaké úlohy přidávání nebo odebírání skupin prvků. Typickým příkladem je třeba rozvržení naplánovaných přednášek do učeben. Seřadíme si začátky přednášek podle času a postupně bereme jednu za druhou a umísťujeme je do volných učeben s nejnižším číslem.

Tím jsme si určitě nic nerozbili, protože v nějaké učebně přednáška být musí. Určitě budeme potřebovat tolik učeben, kolik je maximálně přednášek v jeden čas, a díky tomu si umístěním přednášky do nějaké učebny nezablokujeme místo pro jinou přednášku, jelikož nám vždy zbude dostatek volných učeben.

Kdybychom ale naopak měli pevně zadaný počet učeben a chtěli jsme do nich umístit co možná nejvíce přednášek, tak se již nejedná o úlohu řešitelnou hladovým algoritmem, v takovém případě je potřeba zvolit nějaký chytřejší postup.

Závěr

Doufám, že jste si z tohoto rozsáhlého textu odnesli nějaké nové znalosti a poznatky, které vám pomohou nejen v řešení KSP.

Pokud jste začínajícími řešiteli, zkuste s pomocí kuchařky vyřešit několik lehčích úloh a jejich řešení poslat – nově nabyté znalosti je totiž nejlepší co nejdříve protrénovat. Nic si nedělejte z toho, pokud napoprvé nevyřešíte všechno, s postupným zkoušením se budou vaše znalosti jen zlepšovat. Zkušenější řešitelé možná v kuchařce nalezli nějaké ujasnění pojmů, či si některé techniky osvěžili.

A pokud tento text považujete za dobrý, budeme jen rádi, pokud ho doporučíte svým kamarádům a spolužákům, kteří chtějí s programováním začít.

Úvodním kurzem vaření podle kuchařky vás provedl

Jirka Setnička

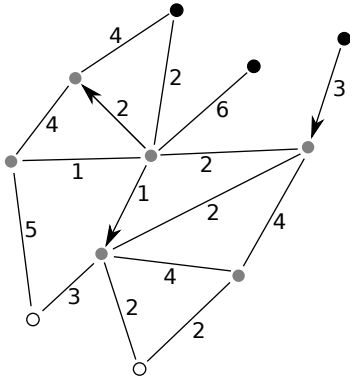
Kuchařka druhé série – toky v sítích

Ukážeme si uměle znějící úlohu, kterou posléze zmatematizujeme, vyřešíme a dokážeme vlastnosti řešení. Nakonec přijdou četná užití, která ozřejmí, proč jsme se snažili.

Látka je lehce pokročilá, takže vězte, že budete potřebovat znát grafy.

Uměle znějící úloha

Ruský petrobaron vlastní ropná naleziště na Sibiři a trubky vedoucí do Evropy. Trubky vedou mezi nalezišti, uzlovými body a koncovými body, kde ropu přebírají odběratelé.



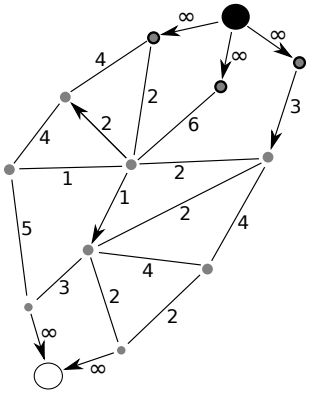
Každá trubka může a nemusí mít definováno, kterým směrem jí má téci ropa. Pro každou trubku zvlášť víme, kolik nejvýše jí za hodinu protlačíme.

Naleziště jsou bezedná a mohou posílat neomezená množství ropy. Odběratelé také dokáží neomezená množství ropy z koncových bodů odebírat. Petrobaron čelí problému, jak protlačit danou distribuční sítí co nejvíce ropy za hodinu ze zdrojů k odběratelům.

Zapeklité je to zejména kvůli tomu, že v uzlových bodech nelze ropu hromadit, ani pálit – rozhodně tedy nejde bez rozmyslu přikázat, ať každou trubkou teče maximum, protože bychom poškodili cenná zařízení a v uniklé ropě utopili vše živé.

Zmatematizování

V zadání vidíme graf, který obsahuje orientované i neorientované hrany, kde je nějaká podmnožina vrcholů označená jako zdroje a jiná jako… říkejme tomu třeba stoky.



Abychom měli situaci jednodušší, zbavíme se hned na úvod mnohočetnosti zdrojů a stoků. Přikreslíme si dva nové vrcholy – z nadzdroje budeme posílat ropu do všech zdrojů, do nadstoku budeme posílat ropu ze všech stoků. Kapacitu přikreslených hran pak nastavíme na nekonečno.

Teď nám stačí vymyslet algoritmus, který řeší problém s právě jedním zdrojem a právě jedním stokem.

Každý vstup totiž popsaným způsobem převedeme, pošleme ho algoritmu a z výstupu prostě jen odstraníme dva přidané vrcholy a připojené hrany.

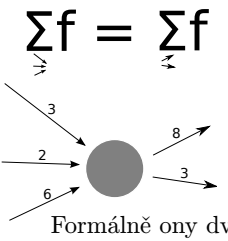
Podobně se zbavíme neorientovaných hran.

Každou takovou hranu v každém zadání změníme na dvojici protisměrných orientovaných hran se stejnou kapacitou. V algoritmu pak už můžeme počítat jen s hranami orientovanými.

Dostáváme se nyní k nejdůležitějšímu – podmínkám na hledaný tok.

Na vstupu dostáváme ohodnocení hran nezápornými čísly a naším úkolem je sestavit jiné ohodnocení těch samých (všech) hran.

Je důležité, aby se nám to nepletlo – ohodnocení ze vstupu se říká kapacita a značí se $c(e)$, konstruované ohodnocení se jmenuje tok a říkáme mu $f(e)$.



Konstruované ohodnocení se snažíme maximalizovat, ale omezuje nás kapacita a Kirchhoffův zákon.

Tak budeme říkat podmínce na to, že součet toku na hranách, které do vrcholu vstupují, musí být stejný jako součet toku na hranách, které z vrcholu vystupují. Máte-li rádi fyziku nebo berete-li školu vážně, důvod k takovému pojmenování jistě chápete.

Formálně ony dvě podmínky vypadají takto:

$$\forall e \in E : f(e) \leq c(e)$$

$$\forall v \in V \setminus \{z, s\} : \sum_{\overrightarrow{uv} \in E} f(\overrightarrow{uv}) = \sum_{\overrightarrow{vu} \in E} f(\overrightarrow{vu})$$

Kirchhoffova podmínka se samozřejmě netýká ani zdroje, ani stoku – tam nám naopak jde o to ji co nejvíce porušit. Velikost toku je nejsnazší měřit na nich. Budeme ji definovat jako rozdíl mezi součtem odtoků a součtem přítoků ve zdroji.

K zamyšlení

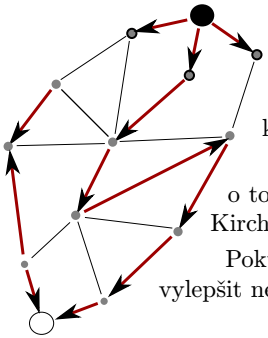
- Nastavit ohodnocení hrany (kapacitu) na skutečné nekonečno v našem programovacím jazyce nemusí jít. Pak se to řeší tím, že se zvolí dostatečně velké číslo. Jak co nejmenší, ale stále bezpečné, rychle ze zadání určit? Stejný problém se řeší třeba v Dijkstrově algoritmu, ale i ve spoustě dalších.
- Neorientované hrany, neboli obousměrné trubky, si zaslouží podrobnější rozbor, než jaký jsme jim věnovali v textu. Jak spolehlivě převedeme řešení algoritmu do původní sítě?
- Vymysleli jsme, jak vyřešit více zdrojů a stoků a jak ošetřit obousměrné trubky. Co kdyby bylo v zadání omezení na průtok vrcholy?
- Umíte dokázat, že je absolutní hodnota rozdílu přítoků a odtoků stejná na zdroji i na stoku? Tedy že bychom mohli velikost toku stejně tak dobře měřit i na stoku?

Řešení

Problém je velmi studovaný a k jeho řešení existují dva velké přístupy, které jsou humorně protikladné. Ten první vezme nulový tok a opatrně ho zlepšuje. Druhý si napíská veliké ohodnocení hran, které ani tokem není, a pak ho opravuje.

Předvedeme si onen první způsob a algoritmus, který se podle svých autorů jmenuje Fordův-Fulkersonův. Bude se nám odteď hodit tvářit se, jako že mezi každými dvěma vrcholy vede oběma směry hrana. Tam, kde ze vstupu nepřišla, si domyslíme jednu s nulovou kapacitou.

Představme si graf, na kterém počítáme tok, a dejme tomu, že už nějaký tok máme – třeba prázdný. Představme si, že jsme ropný magnát a každý rozdíl mezi kapacitou potrubí a jejím využitím (tokem) nás stojí miliony dolarů.

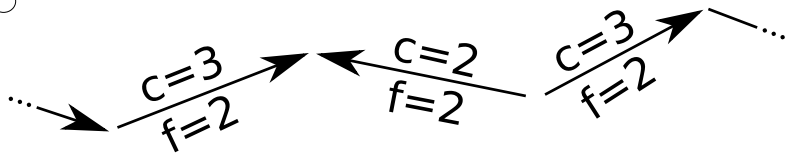


Už jsme se smířili s tím, že každá trubka nemůže být využita na maximum, ale zkusme si vyznačit ty hrany, kde $c(e) \neq f(e)$.

Co když existuje cesta z nadzdroje do nadstoku, která vede pouze po takových hranách?

Můžeme vzít minimum z rozdílů na každé hraně a o toto číslo navýšit tok na každé z nich! Ani kapacitní, ani Kirchhoffovu podmínku to jistě nepoškodí.

Pokud žádnou takovou cestu nevidíme, znamená to, že tok vylepšit nejde? Ne úplně. Představte si následující situaci:



Copak nejde zlepšit? Jde! Není na to první pohled úplně jasné, ale můžeme zlepšovat výsledný tok i tím, že ho na protisměrné části cesty snížíme. Samozřejmě však nesmíme nastavovat tok záporný.

(Je smutné, že si teď trochu kazíme grafovou terminologii – co je to za cestu v orientovaném grafu, která nemusí respektovat orientaci hran?)

Takže jaká je přesně podmínka pro „vyznačení“ hrany $u\vec{v}$? Nastává $f(u\vec{v}) < c(u\vec{v})$ nebo $f(v\vec{u}) > 0$. Potom ji lze zlepšit o $c(u\vec{v}) - f(u\vec{v}) + f(v\vec{u})$.

Hledání všech vhodných („zlepšujících“) cest tedy můžeme dělat prostým prohledáváním do šířky přes vyznačené hrany. Budeme to dělat opakovaně znovu a znovu, až žádnou takovou nenajdeme, a pak vrátíme získaný tok jako výsledek.

Analýza algoritmu

Správnost

Zavolali jsme algoritmus na prázdný tok, ten ho zlepšil do situace, ve které neexistuje zlepšující cesta.

Znamená tato neexistence, že je výsledný tok maximální? Opačná implikace je jasná – maximální tok zlepšit žádným způsobem nepůjde, takže ani přes zlepšující cestičky.

Když zkusíme algoritmus pustit na graf, kde už žádná taková cesta není, můžeme si poznamenat všechny vrcholy, kam jsme se pomocí prohledávání zlepšitelných hran ještě dostali. Tato množina bude jistě obsahovat zdroj (tam jsme začali) a jistě nebude obsahovat stok (to by existovala zlepšující cesta).

Na hranách mezi touto množinou a jejím doplňkem nemůžeme zlepšovat, jinak by se po nich náš program pustil dál a množinu vrcholů, kam se dostal, by rozšířil. Všechny hrany směřující ven tedy mají $f(e) = c(e)$, pro všechny hrany směřující dovnitř platí $f(e) = 0$.

Tyto hrany tvoří *řez* naším grafem. Odvoláme se v tuto chvíli na vaši intuici – tok nemůže být větší než libovolný řez. Z toho už dostáváme, že náš algoritmus našel tok maximální, protože našel také řez, který zaručuje, že nemůže existovat tok větší.

Formálnější předvedení najdete ve skriptíčkách z kombinatoriky.[43]

Časová složitost

Je možné dobu běhu omezit počtem vrcholů a hran? Výše uvedeným postupem na grafu s celočíselnými kapacitami každou nalezenou cestou zvýšíme tok alespoň o jednotku, takže program nebude běžet déle, než je součet všech kapacit. Ale to není moc uspokojivý odhad, protože záleží na ohodnocení.

Pokud budeme hledat cesty skutečně prohledáváním do šířky, bude počet kroků v $\mathcal{O}(nm^2)$, protože se dá ukázat, že se hrany, které při zlepšování cesty tvoří minimum, postupně vzdalují od zdroje. Pak máme $\mathcal{O}(m)$ času k nalezení cesty a m hran, které se nejvýše n-krát mohou vzdálit. Že to tak skutečně je, je lehce zdlouhavé intelektuální cvičení. Nechat si prozradit postup můžete třeba v druhém vydání Introduction to Algorithms na straně 662.

O vylepšení daného postupu si můžete přečíst v kapitole o tocích[44] Medvědových skriptíček o algoritmech a datových strukturách, ukázka druhého přístupu k řešení hledání maximálního toku je tam také.

K zamyšlení

- Důležitou vlastností algoritmu je, že když dostane celočíselné kapacity, vrátí celočíselný tok. Bude se nám to hodit v aplikacích. Dokážete to?

[43] `http://kam.mff.cuni.cz/~valla/kg.html`

[44] `http://mj.ucw.cz/vyuka/ads/41-toky.pdf`

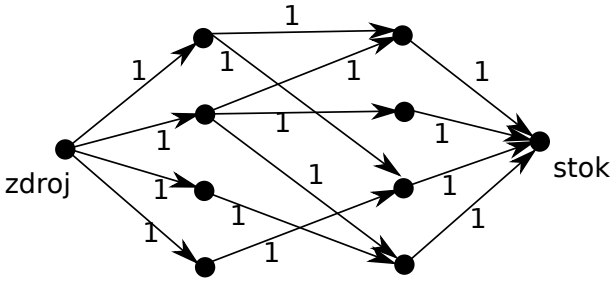
- Rozdíl mezi Fordem-Fulkersonem, který hledá cesty obecným způsobem, a takovým, který to dělá prohledáváním do šířky, je ze složitostního hlediska docela velký, a proto se tomu druhému občas říká Edmondsův-Karpův. Najděte malý graf a nevhodnou posloupnost cest, která způsobí, že F-F poběží skutečně v závislosti na velikosti kapacit.
- Můžete dokonce zkusit využít zlatého řezu k nalezení grafu s reálnými kapacitami, na kterém F-F pro danou (nešikovnou) posloupnost cest nikdy neskončí.
- Skončí algoritmus v konečném čase, jsou-li kapacity čísla racionální?

Užití

Párování v bipartitních grafech

Máme-li za úkol najít na plese co nejvíce tanečnicím tanečníka, kterého znají, stojíme před zásadním a nelehkým úkolem.

Co třeba postavit na základě známosti bipartitní graf mezi partitou tanečníků a partitou tanečnic, přidat zdroj za kluky a stok za holky, tyto k nim připojit hranami s jednotkovou kapacitou, hranám v bipartitním grafu také nastavit jednotkové kapacity a nakonec všechno zorientovat směrem do stoku?



Maximální celočíselný tok, který na tomto grafu získáme, nám hrany bipartitního grafu rozdělí na nevybrané s tokem 0 a vybrané s tokem 1. Můžou vybrané hrany sdílet tanečníka? Těžko, když do něj teče nejvýše jednotkový tok a musí platit Kirchhoffův zákon. A podobně s tanečnicemi.

Vybrané hrany nám proto vytvoří párování. A protože jsme našli maximální tok, jde o párování největší. Kdyby existovalo párování větší, dokázali bychom z něj zvětšit tok.

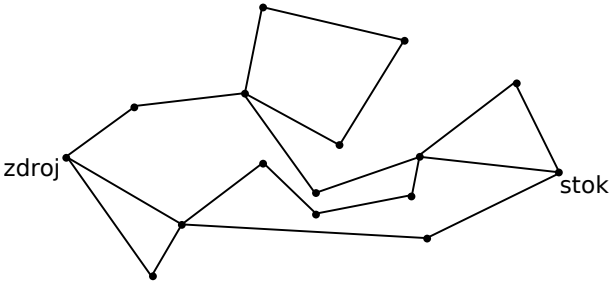
Hledání hranově a vrcholově disjunktních cest

Chceme-li se v grafu G dostat z vrcholu u do vrcholu v, může nás zajímat (třeba kvůli spolehlivosti, s jakou se umíme dostat do cíle), kolik mezi nimi existuje cest, které:

- nesdílí hrany, nebo
- nesdílí vrcholy. (Tato podmínka je silnější. Když dvě cesty nesdílí vrcholy, nesdílí hrany.)

Oba tyto problémy lze převést na hledání maximálního toku. V obou případech nastavíme u jako zdroj a v jako stok. V prvním případě nastavíme jednotkové kapacity všem hranám, v druhém navíc všem vrcholům.

Ford-Fulkerson nastavil některým hranám jednotkový tok, některým nulový. Nulové nyní z grafu vyhodíme. Pokud jsme hledali hranově disjunktní cesty, můžeme nyní získat třeba takovýto graf:



Jak z něj vykřesat kýžený výsledek? Začneme procházet ze zdroje zbylé hrany. Vždy, když se dostaneme do vrcholu, ve kterém už jsme v tom samém průchodu byli, vyhodíme z grafu všechny hrany cyklu, který jsme tímto objevili. (Hodnota toku se tím nezmění.)

Průchodem grafu se vždy můžeme dostat až do stoku (všude jinde budeme moci podle Kirchhoffova zákona jít dál – dost to připomíná úvahu o eulerovských tazích),[45] a protože jsme mezitím agilně odstraňovali cykly, dostali jsme cestu. Vrátíme ji jako jeden výsledek, smažeme její hrany, a pokud ještě tok není nulový, pokračujeme dál.

Počet cest je tedy velikost toku. Podle Mengerovy věty je navíc počet hranově/vrcholově disjunktních cest roven stupni hranové/vrcholové souvislosti grafu – máme tedy nyní algoritmus, který ji najde.

K zamyšlení

- Úvaha nebyla naprosto přímočará kvůli cyklům v nalezeném toku. Říká se jim cirkulace. Je jasné, že v případě hledání hranově disjunktních cest vzniknout mohou. Co v případě vrcholově disjunktních, tedy v situaci, kdy jsme omezili tok vrcholy?
- Nepracuje náhodou neupravený Edmondsův-Karpův algoritmus rychleji, pokud je graf, jak jsme teď opakovaně viděli, ohodnocený toliko nulami a jedničkami?

Dnešní menu servíroval

Lukáš Lánský

[45] `http://ksp.mff.cuni.cz/viz/kucharky/eulerovske-tahy`

Kuchařka třetí série – 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 jednoduššího, 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.

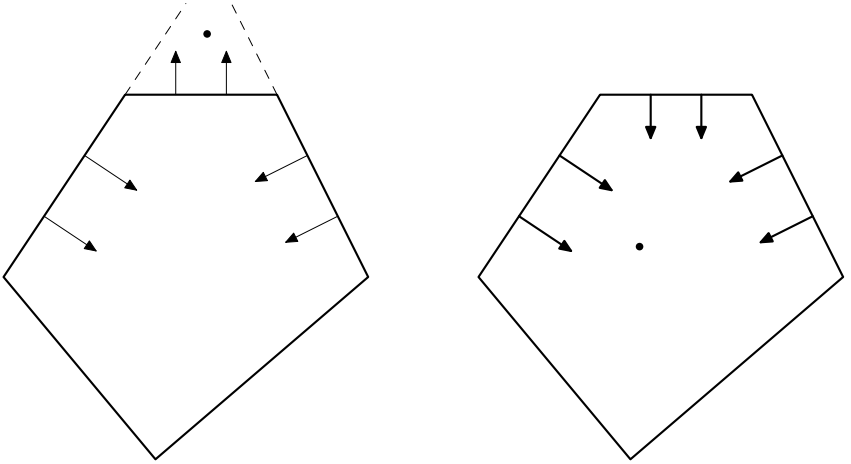
Bod a konvexní mnohoúhelník

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.

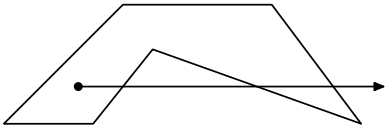
Můžeme si ale na chvíli zahrát na Robina Hooda a ze zkoumaného bodu vystřelit šíp, respektive vést polopřímku. Pěkně se nám bude počítat, pokud polopřímku povedeme rovnoběžně s nějakou z os (třeba ve směru $(1, 0)$). Celé řešení pak spočívá v počítání, kolikrát polopřímka protne hranici mnohoúhelníku.

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.

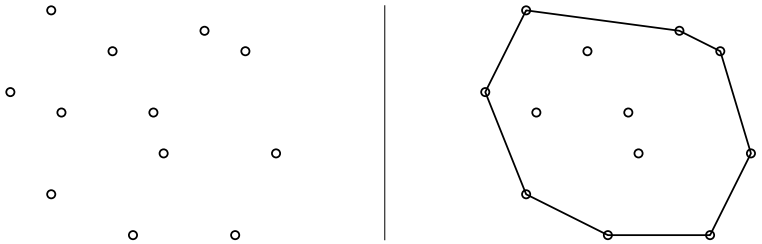


Opět musíme zkontrolovat polopřímku vůči všem hranám, takže časová složitost je znovu $\mathcal{O}(N)$ (i když s o něco vyšší konstantou, protože spočítání průsečíku je více početních operací, než jeden test polorovinou).

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.

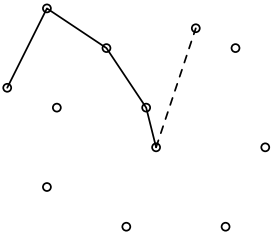
V průběhu výpočtu si budeme konstruovat horní a dolní *obálku*. Obě obálky budou určitě začínat v nejlevějším a končit v nejpravějším bodě (jednoduchým pozorováním lze nahlédnout, že tyto body do obalu určitě patří). A jak už název napovídá, horní obálka půjde vrchem a bude se zatáčet stále doprava, 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.[46]

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.

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

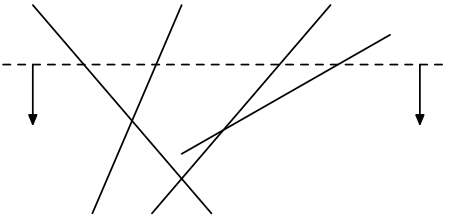
Hledáme samozřejmě co nejrychlejší algoritmus vzhledem k N a počtu průsečíků P.

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 rovinnné 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[47] 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.

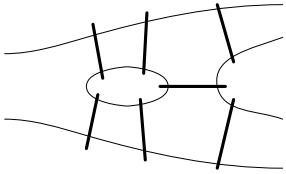
Jirka Setnička

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

Kuchařka čtvrté série – eulerovské tahy

Historický problém

V roce 1735 se švýcarskému matematikovi Leonhardu Eulerovi na stůl dostal na první pohled jednoduchý problém, který mu předložil starosta města Královec (dnešní Kaliningrad). Královcem teče řeka Pregola, na ní je několik ostrovů a ostrovy jsou spojeny se zbytkem města mosty. Dobová ilustrace situaci vystihla takto (schematická kresba):



Pan starosta se pana matematika v dopise tázal, jestli je možné začít na některém z břehů (nebo ostrovů) a udělat si vycházku po městě tak, že se každým mostem projde právě jednou. Navíc chtěl procházku skončit na kusu suché země, ze kterého vyšel.

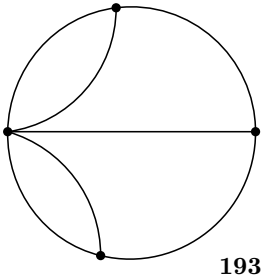
Profesor Euler jej nejprve chtěl poslat k šípku – problém jde snadno vyřešit rozborem případů, což by zvládli i tehdejší studenti střední školy (natož pak ti dnešní). Zachoval se ovšem jako pravý matematik – přišel na to, jak problém zobecnit, a mistrně vyřešil hádanku i pro všechna možná města, která kdy budou chtít pořádat podobné procházky.

Eulerovský tah

Pojďme si nyní problém popsat abstraktně a tím si připomenout grafovou terminologii. Vrcholy našeho grafu jsou kusy pevniny, ať už to budou části města nebo ostrovy. Mezi dvěma vrcholy povede hrana, pokud jsou spojeny mostem, a onen most odpovídá hraně.

V tomto zadání má smysl uvážit, že mezi dvěma kusy pevniny povede mostů více – například v Praze jich vede tolik, že se na to ptají v leckteré zeměpisné olympiádě. Graf, kde mezi vrcholy vede více hran, nazýváme *multigraf,* a pokud dvě hrany vedou mezi stejnými vrcholy, mluvíme o nich jako o *paralelních* hranách.

Obecná procházka v grafu z vrcholu A do vrcholu B (posloupnost hran taková, že cílový vrchol předchozí hrany je počáteční vrchol hrany následující) se nazývá *sled* z A do B. Ve sledu se mohou opakovat jak hrany, tak vrcholy; sled tedy není řešením našeho problému (ve sledu je možné se vrátit po hraně, ze které jsme právě přišli).



Pro naši úlohu se hodí posloupnost hran taková, že vrcholy se opakovat mohou, ale hrany nikoli. Této posloupnosti se říká *tah* z A do B. Kdyby se neopakovaly ani vrcholy, pak posloupnost označujeme jako *cestu*. Tah (respektive sled) je *uzavřený,* pokud začíná v A a končí také v A.

Podíváme-li se tedy na mapu Královce jako na multigraf, ptáme se, zdali existuje uzavřený tah takový, že každou hranu navštíví právě jednou. Takovému tahu pak říkáme *uzavřený eulerovský*.

Mimochodem, tahu se „tah“ neříká jen tak náhodou. Děti se často ve školce překonávají v umění nakreslit obrázek jedním tahem, aby se tužkou nemuselo vracet po už nakreslené čáře. Pokud si obrázek představíme jako graf (čáry jsou hrany, místa jejich setkání vrcholy), pak eulerovský tah nalezneme jen v tom obrázku, který lze nakreslit jedním tahem. V uzavřeném eulerovském tahu se pak vrátíme i do místa, kde jsme začali.

Podmínky tahu

Je na čase poodhalit řešení našeho problému s eulerovským tahem. Půjdeme na to jako matematici – nejprve ukážeme *nutnou* a hned nato *postačující* podmínku. Nutná vlastnost grafu je taková, že bez ní eulerovský tah není možné najít; postačující vlastnost je ta, se kterou vždy eulerovský tah najít umíme. Jsou-li obě podmínky stejné, pak se jedná o ekvivalenci, a tak tomu bude i nyní.

Představme si, že jsme kouzlem nějaký uzavřený eulerovský tah našli, ať už je jakýkoli. Vždy, když se dostaneme do jednoho vrcholu (a není důležité, jestli už jsme v něm byli, nebo ne), tak z něj musíme hned také odejít, abychom tah uzavřeli. A protože tah je eulerovský, každou hranou projdeme jen jednou, takže tyto dvě hrany (tu příchozí a odchozí) už nepoužijeme. U každého vrcholu mimo výchozí tedy platí, že hrany tvoří dvojice – jedna, co vedla dovnitř, a jedna, která z něj vedla ven.

Podobná věc platí i pro startovní vrchol. Sice do něj nevstoupíme poprvé pomocí hrany, takže počet navštívených hran u něj bude stále lichý – ale jen do chvíle, než se do něj naposledy vrátíme a skončíme, protože skončením jsme použili poslední hranu, která bude tvořit dvojici s hranou první.

Jakou vlastnost grafu jsme odhalili? Neplatí, že graf má sudý počet hran (protože trojúhelník jedním tahem nakreslíme a přesto má 3 hrany), ale platí, že do každého vrcholu vede sudý počet hran, tedy že graf má *všechny stupně sudé*. Nezapomeňme také na to, že graf musí být souvislý – dva oddělené obrázky jedním tahem bez zvednutí tužky nenakreslíme. Máme nutné podmínky!

Nalezení tahu

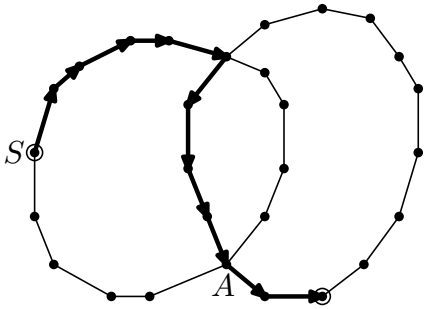
Zbývá tedy ověřit, že podmínky jsou i postačující. Mějme souvislý graf, který má všechny stupně sudé. Umíme v něm vždy najít uzavřený eulerovský tah? Ověřme to, jak se na informatiky patří – algoritmem.

Recepty z programátorské kuchařky – Eulerovské tahy

Předložený algoritmus je založený na vylepšeném prohledáváni do hloubky, tedy DFS, o kterém jste si mohli přečíst v první grafové kuchařce.[48]

Vyberme si vrchol, v něm začneme. Náš algoritmus musí umět označovat hrany jako „probrané“, jako to dělá DFS. Vyberme si tedy jednu hranu a pokračujme dále, zatím bez vypisování.

Po nějakém tom procházení se jistě stane, že jsme se zastavili – vrchol už nemá žádné nepoužité hrany. Nutně to znamená, že to je ten vrchol, ve kterém jsme začínali. V procházení do hloubky se vracíme zpět, ale my k tomu přidáme vypisování cesty – postupně pozpátku vypisujeme hrany, kterými se vracíme zpět v prohledávání.



Na obrázku výše je příklad právě probíhajícího algoritmu. Začal ve zvýrazněném vrcholu vlevo, procházel po šipkách až do bodu A, kde volil hrany tak, že hned skončil na začátku. Dále pokračoval vypisováním hran pozpátku, až došel zase do bodu A. Zde si vybral jednu ještě nepoužitou hranu a po ní prošel celou druhou kružnici – zbytek hran – zpět do bodu A. Nyní vypisuje hrany pozpátku od bodu A.

Buď tímto výpisem dojdeme až na začátek, nebo se dostaneme do vrcholu, který má ještě nějaké nepoužité hrany (situace může vypadat třeba jako na obrázku). Potom vypisování zastavíme a pokračujeme v prohledávání DFS přes nepoužitou hranu. I tam se to může zastavit (a zastaví), i tam začneme vypisovat pozpátku. Nakonec dojdeme do původního místa rozbočení, a budeme opět pozpátku vypisovat hrany, které nás nakonec dostanou až na počátek, kde skončíme.

Najde tento algoritmus opravdu korektní uzavřený eulerovský tah? Graf byl souvislý a o algoritmu DFS se ví, že v takovém případě navštíví každou hranu právě jednou. Algoritmus opravdu vypisuje cyklus – jen je u něj trochu zvláštní způsob, jak ho vypisuje. Když dojde na křižovatku s ještě nepoužitými hranami,

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

tak výpis zastaví, tiše po nich kráčí, označuje si je a vypisuje, až když se po nich vrací. Ověřme si, že hrany opravdu navazují.

V duchu argumentů z předcházející části víme, že jediný vrchol grafu s lichým počtem nepoužitých hran je právě ona křižovatka – a algoritmus DFS prochází graf podobně, jako jsme ho procházeli v minulé sekci, takže právě do tohoto vrcholu algoritmus dojde, až se průchod touto částí grafu zastaví.

Jakmile sem program dojde (a nezbudou mu volné hrany), začne cestovat zpět a hrany vypisovat – a opravdu, pokračuje se tedy z místa, kde naposledy přestal, a program vskutku vypíše tah přes všechny hrany v grafu – uzavřený eulerovský tah.

Věta o eulerovském tahu v celé své kráse tedy zní: *(Multi)graf obsahuje uzavřený eulerovský tah právě tehdy, když má všechny stupně sudé a je souvislý.*

Je třeba podotknout, že složitost našeho algoritmu na bázi DFS je lineární vůči velikosti grafu (počtu vrcholů a hran). Existují i jiné algoritmy pro hledání eulerovského tahu, jedna varianta například prochází grafem a vybírá si na křižovatkách takové hrany, které souvislost grafu pokud možno nepoškodí. Tyto algoritmy už nemusí mít nutně lineární časovou složitost.

Jiné druhy procházek

Nejen kreslením obrázků ze stejného bodu živ je člověk. Co kdybychom mohli začít a skončit v jiném místě, tedy ptali se po neuzavřených eulerovských tazích, změnilo by se něco? Není tomu tak, pouze nutné a postačující podmínky si vyžádají, aby všechny vrcholy měly sudý stupeň až na právě dva vrcholy, které mají lichý stupeň. Pokud nám to nevěříte, zkuste si to rozmyslet sami, opravdu to není těžké.

Smysl také dává zkusit najít ne uzavřený tah, ale uzavřenou cestu – uzavřenou cestu přes všechny vrcholy, která navštíví každý vrchol právě jednou (říká se jí „Hamiltonovská cesta“). Bohužel, ačkoli jsou problémy příbuzné, musíme vás zklamat – není znám žádný efektivní (polynomiální) algoritmus na tento problém, a kdyby jej někdo z vás nalezl, vyřešil by otázku „P vs. NP“, o níž se více dočtete v kuchařce o těžkých problémech.[49]

V matematice se také někdy zmiňují „náhodné procházky“ po grafech – můžete si je představit tak, že se po mostech města Královce motá opilec, který si hází (opilou nebo spravedlivou) mincí a podle toho se rozhoduje, přes který most jít dál. Použití mají tyto modely hlavně v matematické teorii grafů a teorii pravděpodobnosti. O tom si můžeme povědět zase někdy jindy.

Martin Böhm

[49] `http://ksp.mff.cuni.cz/viz/kucharky/tezke-problemy`

Kuchařka páté série – dynamické programování

Rekurzivní funkce je taková funkce, která při svém běhu volá sama sebe, často i více než jednou. To typicky vede na exponenciální časovou složitost algoritmu.

Dynamické programování je technika, kterou lze z pomalého rekurzivního algoritmu vyrobit pěkný polynomiální, tedy až na výjimečné případy. Ale nepředbíhejme, nejdříve se podíváme na jednoduchý příklad rekurze.

Fibonacciho čísla

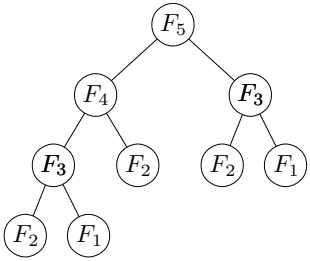
Budeme počítat n-té číslo Fibonacciho posloupnosti. To je posloupnost, jejímiž prvními dvěma členy jsou jedničky ($F_1 = 1$, $F_2 = 1$) a každý další člen je součtem dvou předchozích ($F_n = F_{n-1} + F_{n-2}$ pro $n > 2$). Začíná takto:

$$1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad 55 \quad 89 \quad \ldots$$

Pro nalezení n-tého členu (ten budeme značit F_n) si napíšeme rekurzivní funkci `Fibonacci(n)`, která bude postupovat přesně podle definice – zeptá se sama sebe rekurzivně, jaká jsou dvě předchozí čísla, a pak je sečte. Možná více řekne program:

```
def fibonacci(n):
  if n <= 2:
    return 1
  else:
    return fibonacci(n-1) + fibonacci(n-2)
```

To, jak funkce volá sama sebe, si můžeme snadno nakreslit třeba pro výpočet čísla F_5:



Vidíme, že se program rozvětvuje, což tvoří strom volání. V každém vrcholu tohoto stromu trávíme konstantní čas, takže časová složitost celého algoritmu je až na konstantu rovna počtu vrcholů tohoto stromu. Kolik to je, spočítáme jednoduchou úvahou.

Každý vrchol stromu vrací hodnotu, která je součtem hodnot v jeho synech. Proto je hodnota v kořeni rovna součtu hodnot v listech. V listech jsou ovšem jedničky (F_1 a F_2), takže listů musí být právě F_n a všech vrcholů dohromady aspoň F_n.

Proto na spočítání n-tého Fibonacciho čísla spotřebujeme čas alespoň takový, kolik je ono číslo samo. Ale jak velké takové F_n vlastně je? Můžeme třeba využít toho, že

$$F_n = F_{n-1} + F_{n-2} \geq 2 \cdot F_{n-2},$$

z čehož indukcí dokážeme

$$F_n \geq 2^{n/2} \quad \text{pro } n \geq 6.$$

Funkce `Fibonacci` má tedy alespoň exponenciální časovou složitost což není nic vítaného.

Jak najít efektivnější algoritmus? Všimneme si, že některé podstromy jsou shodné. Zřejmě to budou ty části, které reprezentují výpočet stejného Fibonacciho čísla – v našem příkladě třeba třetího. Tyto výpočty opakujeme stále dokola.

Nenabízí se proto nic snazšího, než si jejich výsledky uložit a pak je kdykoliv vytáhnout jako pověstného králíka z klobouku s minimem námahy.

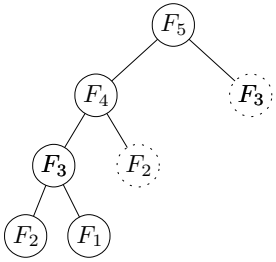
Právě zde je zmínka o králících příhodná. Legenda o Fibonacciho číslech vypráví, že k jejich objevu došlo při výzkumu rozmnožování králíků.

Leonardo Pisánský (známý též jako Fibonacci) totiž pěstoval králíky. První dva měsíce měl 1 pár, další měsíc měl 2 páry, pak 3, pak 5, ...

Bude nám k tomu stačit jednoduché pole P o n prvcích, na počátku inicializované nulami. Kdykoliv budeme chtít spočítat některý člen, nejdříve se podíváme do pole, zda jsme ho již jednou nespočetli. A naopak jakmile hodnotu spočítáme, hned si ji do pole poznamenáme:

```
P = [0] * (N+1)
def fibonacci(n):
  if P[n] == 0:
    if n <= 2:
      P[n] = 1
    else:
      P[n] = fibonacci(n-1) + fibonacci(n-2)
  return P[n]
```

Podívejme se, jak vypadá strom volání nyní:



Na každý člen posloupnosti se tentokrát ptáme maximálně dvakrát – k výpočtu ho potřebují dva následující členy. To ale znamená, že funkci `Fibonacci` zavoláme maximálně $2n$-krát, čili jsme touto jednoduchou úpravou zlepšili exponenciální složitost na lineární.

Zdálo by se, že abychom získali čas, museli jsme obětovat paměť, ale to není tak úplně pravda. V prvním příkladu sice nepoužíváme žádné pole, ale při volání funkce si musíme zapamatovat některé údaje, jako je třeba návratová adresa, parametry funkce a její lokální proměnné, a na to samotné potřebujeme určitě paměť lineární s hloubkou vnoření, v našem případě tedy lineární s n.

Určitě vás už také napadlo, že n-té Fibonacciho číslo se dá snadno spočítat i bez rekurze. Stačí prvky našeho pole P plnit od začátku – kdykoli známe $P[1 \ldots k] = F_{1 \ldots k}$ (všechny prvky pole na pozicích od 1 do k), dokážeme snadno spočítat i $P[k+1] = F_{k+1}$:

```
P = [0] * (N+1)
def fibonacci(n):
  P[1] = 1
  P[2] = 1
  for i in range(3, n):
    P[i] = P[i-1] + P[i-2]
  return P[n]
```

Zopakujme si, co jsme postupně udělali – nejprve jsme vymysleli pomalou rekurzivní funkci, kterou jsme zrychlili zapamatováváním si mezivýsledků.

Nakonec jsme ale celou rekurzi „obrátili naruby“ a mezivýsledky počítali od nejmenšího k největšímu, aniž bychom se starali o to, jak se na ně původní rekurze ptala.

V případě Fibonacciho čísel je samozřejmě snadné přijít rovnou na nerekurzivní řešení a dokonce si všimnout, že si stačí pamatovat jen poslední dvě hodnoty, a paměťovou složitost tak zredukovat na konstantní.

Zmíněný obecný postup zrychlování rekurze nebo řešení úlohy od nejmenších podproblémů k těm největším funguje i pro řadu složitějších úloh. Obvykle se mu říká *dynamické programování*.

Problém batohu

Je dáno N předmětů o hmotnostech $m_1, \ldots, m_N$ (celočíselných) a také číslo M (nosnost batohu). Úkolem je vybrat některé z předmětů tak, aby součet jejich hmotností byl co největší, ale přitom nepřekročil M. Předvedeme si algoritmus, který tento problém řeší v čase $\mathcal{O}(MN)$.

Náš algoritmus bude používat pomocné pole $A[0 \ldots M]$ a jeho činnost bude rozdělena do N kroků. Na konci k-tého kroku bude prvek $A[i]$ nenulový právě tehdy, jestliže z prvních k předmětů lze vybrat předměty, jejichž součet hmotností je přesně i.

Před prvním krokem (po nultém kroku) jsou všechny hodnoty $A[i]$ pro $i > 0$ nulové a $A[0]$ má nějakou nenulovou hodnotu, řekněme -1.

Všimněme si, jak kroky algoritmu odpovídají podúlohám, které řešíme – v prvním kroku vyřešíme podúlohu tvořenou jen prvním předmětem, ve druhém kroku prvními dvěma předměty, pak prvními třemi předměty atd.

Popišme si nyní k-tý krok algoritmu. Pole A budeme procházet od konce, tj. od $i = M$. Pokud je hodnota $A[i]$ stále nulová, ale hodnota $A[i-m_k]$ je nenulová, změníme hodnotu uloženou v $A[i]$ na k (později si vysvětlíme, proč zrovna na k).

Nyní si rozmyslíme, že po provedení k-tého kroku odpovídají nenulové hodnoty v poli A hmotnostem podmnožin z prvních k předmětů (*podmnožina* je v podstatě jen výběr nějaké části předmětů).

Pokud je hodnota $A[i]$ nenulová, pak buď byla nenulová před k-tým krokem (a v tom případě odpovídá hmotnosti nějaké podmnožiny prvních $k-1$ předmětů) anebo se stala nenulovou v k-tém kroku.

Potom ale hodnota $A[i - m_k]$ byla před k-tým krokem nenulová, a tedy existuje podmnožina prvních $k-1$ předmětů, jejíž hmotnost je $i-m_k$. Přidáním k-tého předmětu k této podmnožině vytvoříme podmnožinu předmětů hmotnosti přesně i.

Naopak, pokud lze vytvořit podmnožinu X hmotnosti i z prvních k předmětů, pak takovou podmnožinu X lze buď vytvořit jen z prvních $k-1$ předmětů, a tedy hodnota $A[i]$ je nenulová již před k-tým krokem, anebo k-tý předmět je obsažen v takové množině X.

Potom ale hodnota $A[i - m_k]$ je nenulová před k-tým krokem (hmotnost podmnožiny X bez k-tého prvku je $i - m_k$) a hodnota $A[i]$ se stane nenulovou v k-tém kroku.

Po provedení všech N kroků odpovídají nenulové hodnoty $A[i]$ přesně hmotnostem podmnožin ze všech předmětů, co máme k dispozici. Speciálně největší index i_0 takový, že hodnota $A[i_0]$ je nenulová, odpovídá hmotnosti nejtěžší podmnožiny předmětů, která nepřekročí hmotnost M.

Nalézt jednu množinu této hmotnosti také není obtížné: V k-tém kroku jsme měnili nulové hodnoty v poli A na hodnotu k, takže v $A[i_0]$ je uloženo číslo jednoho z předmětů nějaké takové množiny, v $A[i_0 - m_{A[i_0]}]$ číslo dalšího předmětu atd. Zdrojový kód tohoto algoritmu lze nalézt na další straně.

Časová složitost algoritmu je $\mathcal{O}(NM)$, neboť se skládá z N kroků, z nichž každý vyžaduje čas $\mathcal{O}(M)$. Paměťová složitost činí $\mathcal{O}(N + M)$, což představuje paměť potřebnou pro uložení pomocného pole A a hmotností daných předmětů.


```
# Již existující proměnné:
# N - počet předmětů
# M - hmotnostní omezení
# hmotnosti - pole vah jednotlivých předmětů

A = [0] * M

A[0] = 1
for k in range(N):
  for i in range(M, hmotnost[k]-1, -1):
    if (A[i-hmotnost[k]] != 0) and (A[i] == 0):
      A[i] = k
  i = M
  while A[i] == 0:
    i -= 1
  print("Maximální hmotnost: {}" .format(i))
  print("Předměty v množině:", end="")
  while A[i] != -1:
    print(" {}" .format(A[i]), end="")
    i = i - hmotnost[A[i]]
```

Cvičení a poznámky

- Proč pole A procházíme pozadu a ne popředu?
- Složitost algoritmu vypadá jako polynomiální, ale to je trochu podvod. Závisí totiž na hodnotě M. Pokud tuto hodnotu na vstupu zapíšeme obvyklým způsobem, tedy v desítkové nebo dvojkové soustavě, použijeme řádově $\log M$ cifer.

  Naše M proto bude vzhledem k délce vstupu až exponenciálně velké. To je typický příklad takzvaného *pseudopolynomiálního* algoritmu – tedy takového,

jenž je vzhledem k hodnotám na vstupu polynomiální, ale k délce vstupu exponenciální. Podrobnosti si můžete přečíst v kuchařce o těžkých úlohách.[50]

Nejkratší cesty a Floydův-Warshallův algoritmus

Náš další příklad bude z oblasti grafových algoritmů, ale zkusíme si jej nejdříve říci bez grafů:

Bylo-nebylo-je N měst. Mezi některými dvojicemi měst vedou obousměrné *silnice,* jejichž (nezáporné) délky jsou dány na vstupu. Předpokládáme, že silnice se jinde než ve městech nepotkávají (pokud se kříží, tak mimoúrovňově).

KSP

Úkolem je spočítat nejkratší vzdálenosti mezi všemi dvojicemi měst, tj. délky nejkratších cest mezi všemi dvojicemi měst.

Cestou rozumíme posloupnost měst takovou, že každá dvě po sobě následující města jsou spojené silnicí, a délka cesty je součet délek silnic, které tato města spojují.

recepty

V grafové terminologii tedy máme daný ohodnocený neorientovaný graf a chceme zjistit délky nejkratších cest mezi všemi dvojicemi jeho vrcholů.

Půjdeme na to následovně – vzdálenosti mezi městy jsou na začátku algoritmu uloženy ve dvourozměrném poli D, tj. $D[i][j]$ je vzdálenost z města i do města j. Pokud mezi městy i a j nevede žádná silnice, bude $D[i][j] = \infty$ (v programu bude tato hodnota rovna nějakému dostatečně velkému číslu).

V průběhu výpočtu si budeme na pozici $D[i][j]$ udržovat délku nejkratší dosud nalezené cesty mezi městy i a j.

Algoritmus se skládá z N fází. Na konci k-té fáze bude v $D[i][j]$ uložena délka nejkratší cesty mezi městy i a j, která může procházet skrz libovolná z měst $1, \ldots, k$.

V průběhu k-té fáze tedy stačí vyzkoušet, zda je mezi městy i a j kratší stávající cesta přes města $1, \ldots, k-1$, jejíž délka je uložena v $D[i][j]$, nebo nová cesta přes město k.

Pokud nejkratší cesta prochází přes město k, můžeme si ji rozdělit na nejkratší cestu z i do k a nejkratší cestu z k do j. Délka takové cesty je tedy rovna $D[i][k] + D[k][j]$.

[50] `http://ksp.mff.cuni.cz/viz/kucharky/tezke-problemy`

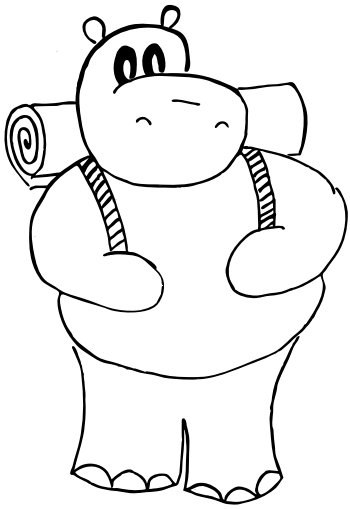
Takže pokud je součet $D[i][k] + D[k][j]$ menší než stávající hodnota $D[i][j]$, nahradíme hodnotu na pozici $D[i][j]$ tímto součtem, jinak ji ponecháme.

Z popisu algoritmu přímo plyne, že po N-té fázi je na pozici $D[i][j]$ uložena délka nejkratší cesty z města i do města j.

Protože v každé z N fází algoritmu musíme vyzkoušet všechny dvojice i a j, vyžaduje každá fáze čas $\mathcal{O}(N^2)$. Celková časová složitost našeho algoritmu tedy je $\mathcal{O}(N^3)$. Co se paměti týče, vystačíme si s polem D a to má velikost $\mathcal{O}(N^2)$.

Program bude vypadat následovně:


```
for k in range(N):
  for i in range(N):
    for j in range(N):
      if d[i][k] + d[k][j] < d[i][j]:
        d[i][j] = d[i][k] + d[k][j]
```



Popišme si ještě, jak bychom postupovali, kdybychom kromě vzdáleností mezi městy chtěli nalézt i nejkratší cesty mezi nimi.

To lze jednoduše vyřešit například tak, že si navíc budeme udržovat pomocné pole $E[i][j]$ a do něj při změně hodnoty $D[i][j]$ uložíme nejvyšší číslo města na cestě z i do j délky $D[i][j]$ (při změně v k-té fázi je to číslo k).

Máme-li pak vypsat nejkratší cestu z i do j, vypíšeme nejprve cestu z i do $E[i][j]$ a pak cestu z $E[i][j]$ do j. Tyto cesty nalezneme stejným (rekurzivním) postupem.

Poznámky

- Popis algoritmu vysloveně svádí k „rejpnutí“: Jak víme, že spojením dvou cest, které provádíme, vznikne zase cesta (tj. že se na ní nemohou nějaké vrcholy opakovat)?

  To samozřejmě nevíme, ale všimněme si, že kdykoliv by to cesta nebyla, tak si ji nevybereme, protože původní cesta bez vrcholu k bude vždy kratší nebo alespoň stejně dlouhá… tedy dokud se v naší zemi nevyskytuje cyklus záporné délky. To bychom měli přidat do předpokladů našeho algoritmu, kdybychom byli pedanti.

- Pozor na pořadí cyklů – program vysloveně svádí k tomu, abychom psali cyklus pro `k` jako vnitřní… jenže pak samozřejmě nebude fungovat.

Cvičení

- Jak by algoritmus fungoval, kdyby silnice byly jednosměrné?
- Na první pohled nejpřirozenější hodnota, kterou bychom mohli použít pro ∞, je `maxint`. To ovšem nebude fungovat, protože $\infty + \infty$ přeteče. Stačí `maxint div 2`?
- Hodnoty v poli si přepisujeme pod rukama, takže by se nám mohly poplést hodnoty z předchozí fáze s těmi z fáze současné. Ale zachrání nás to, že čísla, o která jde, vyjdou v obou fázích stejně. Proč?

Nejdelší společná podposloupnost

Poslední příklad dynamického programování, který si předvedeme, se bude týkat posloupností. Mějme dvě posloupnosti čísel A a B.

Chceme najít jejich nejdelší společnou podposloupnost, tedy takovou posloupnost, kterou můžeme získat z A i B odstraněním některých prvků. Například pro posloupnosti

$$A = 2\ 3\ 3\ 1\ 2\ 3\ 2\ 2\ 3\ 1\ 1\ 2$$
$$B = 3\ 2\ 2\ 1\ 3\ 1\ 2\ 2\ 3\ 3\ 1\ 2\ 2\ 3$$

je jednou z nejdelších společných podposloupností tato posloupnost:

$$C = 2\ 3\ 1\ 2\ 2\ 3\ 1\ 2.$$

Jakým způsobem můžeme takovou podposloupnost najít? Nejdříve nás asi napadne vygenerovat všechny podposloupnosti a ty pak porovnat.

Jakmile si ale spočítáme, že všech podposloupností posloupnosti o délce n je 2^n (každý prvek nezávisle na ostatních buď použijeme, nebo ne), najdeme raději nějaké rychlejší řešení.

Zkusme využít následující myšlenku: vyřešíme tento problém pouze pro první prvek posloupnosti A. Pak najdeme řešení pro první dva prvky A, přičemž

využijeme předchozích výsledků. Takto pokračujeme pro první tři, čtyři, ... až n prvků.

Nejprve si rozmyslíme, co všechno si musíme v každém kroku pamatovat, abychom z toho dokázali spočíst krok následující. Určitě nám nebude stačit pamatovat si pouze nejdelší podposloupnost, jenže množina všech společných podposloupností je už zase moc velká.

Podívejme se tedy detailněji, jak se změní tato množina při přidání dalšího prvku k A: Všechny podposloupnosti, které v množině byly, tam zůstanou a navíc přibude několik nových, končících právě přidaným prvkem.

Ovšem my si podposloupnosti pamatujeme proto, abychom je časem rozšířili na nejdelší společnou podposloupnost.

Takže pokud známe nějaké dvě stejně dlouhé podposloupnosti P a Q končící nově přidaným prvkem v A a víme, že P končí v B dříve než Q, stačí si z nich pamatovat pouze P.

V libovolném rozšíření Q-čka totiž můžeme Q vyměnit za P a získat tím stejně dlouhou společnou podposloupnost.

Proto si stačí pro již zpracovaných a prvků posloupnosti A pamatovat pro každou délku l tu ze společných podposloupností $A[1 \ldots a]$ a B délky l, která v B končí na nejlevějším možném místě.

Dokonce nám bude stačit si místo celé podposloupnosti uložit jen pozici jejího konce v B. K tomu použijeme dvojrozměrné pole $D[a, l]$.

Ještě si dovolíme jedno malé pozorování: Koncové pozice uložené v poli D se zvětšují s rostoucí délkou podposloupnosti, čili $D[a, l] < D[a, l + 1]$, protože posloupnosti délky $l + 1$ nejsou ničím jiným než rozšířeními posloupností délky l o 1 prvek.

Teď již výpočet samotný:

Pokud už známe celý a-tý řádek pole D, můžeme z něj získat $(a + 1)$-ní řádek. Projdeme postupně posloupnost B. Když najdeme v B prvek $A[a + 1]$ (ten právě přidávaný do A), můžeme rozšířit všechny podposloupnosti končící před aktuální pozicí v B.

Nás bude zajímat pouze ta nejdelší z nich, protože rozšířením všech kratších získáme posloupnost, jejíž koncová pozice je větší než koncová pozice některé posloupnosti, kterou již známe. Rozšíříme tedy tu nejdelší podposloupnost a uložíme ji místo původní podposloupnosti.

Toto provedeme pro každý výskyt nového prvku v posloupnosti B. Všimněme si, že nemusíme procházet pole s podposloupnostmi stále od začátku, ale můžeme se v něm posouvat od nejmenší délky k největší.

Ukážeme si, jak vypadá zaplněné pole hodnotami při řešení problému s posloupnostmi z našeho příkladu. Řádky jsou pozice v A, sloupce délky podposloupností.

D	1	2	3	4	5	6	7	8	9	10	11	12
1	2	–	–	–	–	–	–	–	–	–	–	–
2	1	5	–	–	–	–	–	–	–	–	–	–
3	1	5	9	–	–	–	–	–	–	–	–	–
4	1	4	6	11	–	–	–	–	–	–	–	–
5	1	2	5	7	12	–	–	–	–	–	–	–
6	1	2	3	7	9	14	–	–	–	–	–	–
7	1	2	3	7	8	12	–	–	–	–	–	–
8	1	2	3	7	8	12	13	–	–	–	–	–
9	1	2	3	5	8	9	13	14	–	–	–	–
10	1	2	3	4	6	9	11	14	–	–	–	–
11	1	2	3	4	6	9	11	14	–	–	–	–
12	1	2	3	4	6	7	11	12	–	–	–	–

Zbývá popsat, jak z těchto dat zvládneme rekonstruovat hledanou nejdelší společnou podposloupnost (NSP).

Ukážeme si to na našem příkladu – jelikož poslední nenulové číslo na posledním řádku je v 8. sloupci, má hledaná NSP délku 8.

$D[12, 8] = 12$ říká, že poslední písmeno NSP je na pozici 12 v posloupnosti B. Jeho pozici v posloupnosti A určuje nejvyšší řádek, ve kterém se tato hodnota také vyskytuje, v našem případě je to řádek 12. Druhé písmeno tedy budeme určovat z $D[10, 7]$, třetí z $D[9, 6]$, atd.

Jednou z hledaných podposloupností je tedy:

poslupnost:	2	3	1	2	2	3	1	2
indexy v A:	1	2	4	5	7	9	10	12
indexy v B:	2	5	6	7	8	9	11	12

Již zbývá jen odhadnout složitost algoritmu. Časově nejnáročnější byl vlastní výpočet hodnot v poli, který se skládá ze dvou hlavních cyklů o délce $|A|$ a $|B|$, což jsou délky posloupností A a B.

Vnořený cyklus while proběhne celkem maximálně $|A|$-krát a časovou složitost nám nezhorší. Můžeme tedy říct, že časová složitost je $\mathcal{O}(|A| \cdot |B|)$.

Posloupnosti jsme si prohodili tak, aby první byla ta kratší, protože pak je maximální délka společné podposloupnosti i počet kroků algoritmu roven délce kratší posloupnosti, a tedy i velikost pole s daty je kvadrát této délky.

Paměťovou složitost odhadneme $\mathcal{O}(N^2 + M)$, kde N je délka kratší posloupnosti a M té delší.

```
...
if LA > LB:
  (C, A, B) = (A, B, C)
  (T, LA, LB) = (LA, LB, T)

for i in range(LA):
  d[0][i] = LB

L = 0
MaxL = 0
for i in range(LA):
  for j in range(LA):
    d[i][j] = d[i-1][j]

  L = 0
  for j in range(LB):
    if B[j] == A[i-1]:
      while (L == 0) or (d[i-1][L] < j):
        L += 1
      if d[i][L] >= j:
        d[i][L] = j
    if L > MaxL:
      MaxL = L

  LC = MaxL
  j = LA
  for i in range(LC, 0, -1):
    while d[j-1][i] == d[j][i]:
      j -= 1
    C[i-1] = A[j-1]
    j -= 1

...
```

Dnešní menu servírovali

Martin Mareš a Petr Škoda

Vzorová řešení KSP

První série

28-1-1 Jízda na biomotorce

Naše mapa města je představována neorientovaným grafem, v němž hledáme nejkratší cestu mezi počátečním a cílovým vrcholem. Délka cesty je určena počtem hran, které musíme projít. Toho jste se takřka všichni správně chytli a usoudili jste, že řešením úlohy bude modifikace prohledávání do šířky.[51] Tento algoritmus, zvaný také BFS (podle anglického *breadth-first search*) nám najde vzdálenosti všech vrcholů grafu od nějakého počátečního vrcholu.

Použijeme frontu, což je datová struktura, do které můžeme přidávat prvky a zase je odebírat. Důležité je, aby prvky byly vyjmuty v pořadí, v jakém byly do fronty přidány – stačí si představit frontu lidí na poště. V naší frontě si budeme uchovávat vrcholy grafu.

U každého vrcholu bude uložené číslo odpovídající jeho vzdálenosti od počátečního vrcholu. Ten bude mít nastavenou vzdálenost na nulu (nemusíme jít přes žádnou hranu). Ostatním vrcholům na začátku přiřadíme vzdálenost *nekonečno*. Počáteční vrchol vložíme do fronty.

Samotný algoritmus probíhá tak dlouho, dokud je fronta neprázdná: vyjmeme z fronty vrchol U a podíváme se na jeho sousední vrcholy (ty, které jsou s ním spojeny hranou). Pokud nějaký sousední vrchol má nekonečnou vzdálenost, pak ji nastavíme na vzdálenost U zvětšenou o jedna. Navíc vložíme takový vrchol do fronty.

Všechny vrcholy, jež jsou z toho počátečního dosažitelné (je mezi nimi cesta složená z hran), mají na konci správnou vzdálenost. Jak je to možné? Všimněte si, že vrcholy jsou ve frontě seřazeny v pořadí dle vzdálenosti – nejprve uložíme počáteční vrchol se vzdáleností nula, následují jeho sousední vrcholy se vzdáleností rovnou jedné, následně sousední vrcholy těchto vrcholů atd. Pokud tedy do

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

nějakého vrcholu vede nejkratší cesta složená z H hran, bude na této cestě nejprve počáteční vrchol, pak nějaký jeho soused, dále jeho soused... a vzdálenost každého vrcholu se bude po jedné zvětšovat, až nakonec ve vzdálenosti H bude vrchol, do kterého jsme hledali cestu.

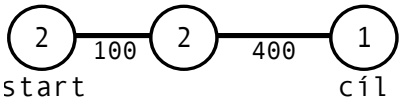
Nedostupným vrcholům zůstane nekonečná vzdálenost, nicméně v naší úloze jsme předpokládali, že celý graf je souvislý (mezi každými dvěma vrcholy vede nějaká cesta).

Pokud jako N označíme počet vrcholů a M počet hran (zapamatujte si toto označení, používá se poměrně často), celý algoritmus má časovou složitost $\mathcal{O}(N + M)$. Hlavní cyklus se opakuje tolikrát, kolik máme v grafu vrcholů – a každý z nich vložíme do fronty jen jednou. V každé iteraci cyklu prozkoumáme hrany vedoucí z určitého vrcholu, v celém průběhu algoritmu se ale na každou hranu takto podíváme jen dvakrát (z jednoho a druhého vrcholu, které spojuje).

Že by tedy stačilo spustit na startovní křižovatce BFS a vypsat vzdálenost cílového vrcholu? Nikoliv, věc nám totiž komplikují pomeranče. Při cestě mezi sousedními vrcholy (nazvěme je V a W) je třeba zkontrolovat, zda jich máme dostatek na cestu. To je však stále snadné, prostě srovnáme počet pomerančů, jež máme ve V, a délku hrany (vydělenou stem, abychom dostali číslo odpovídající spotřebě).

Pokud je počet pomerančů větší nebo roven, můžeme se přesunout do W. Zde se stav nádrže může změnit – vypočteme jej tak, že od počtu pomerančů ve V odečteme pomeranče spotřebované na cestě a naopak přičteme ty, jež obdržíme na křižovatce W. Jen nesmíme zapomenout na limit deseti pomerančů v nádrži. Pokud je počet pomerančů ve V menší, pak W není, alespoň prozatím, dostupné.

Někteří řešitelé však zapomněli na ještě jeden důležitý a na první pohled nepříliš viditelný fakt. Protože některé hrany nelze projet z důvodu nedostatku pomerančů, může se stát, že do některých vrcholů-křižovatek budeme chtít zajet víckrát než jednou. Jeden obrázek nám řekne více než odstavec slov:



Do prostředního vrcholu se dostaneme s počtem tří pomerančů, což nám pro pokračování do cíle nestačí. Je ale možné se otočit a vrátit se do startovního vrcholu, kde opět dostaneme pomeranče, tudíž máme čtyři. Pak cestujeme opět do prostředního vrcholu, kde přistaneme s pěti pomeranči – a to už je dostatek pro překonání hrany dlouhé 400 metrů. Do cíle tedy dojedeme, kvůli oklice budeme potřebovat čtyři hrany.

Při ignorování této možnosti vám často program nefungoval na čtvrtém testovacím vstupu generujícím strom, tedy graf, kde se nenacházejí cykly. V tako-

vém případě vedla mezi startem a cílem jen jedna cesta a pro nalezení řešení bylo často nutné se popsaným způsobem „vracet“.

Jak tedy vypadá kompletně správné řešení? Abychom odlišili různé možnosti, které ve vrcholu máme v závislosti na stavu nádrže, pohybujeme se mezi *stavy*. To jsou v tomto případě dvojice (V, P), kde V je vrchol, v němž se nacházíme, a P počet pomerančů, které v něm máme. Všechny možné dvojice dohromady tvoří *stavový prostor*. Všimněte si, že počet dvojic je konečný (vrcholů máme omezeně mnoho a počet pomerančů je shora omezen). Prostor lze reprezentovat jako graf – jednotlivé stavy jsou vrcholy a hrana vede ze stavu (V, P_1) do (W, P_2) právě tehdy, když z křižovatky V s P_1 pomeranči v nádrži můžete dojet na křižovatku W a mít zde P_2 pomerančů.

Takový graf už nebude neorientovaný, ale prohledávání do šířky můžeme spustit i zde: začneme ve stavu odpovídajícímu naší výchozí situaci (startovní vrchol a tolik pomerančů, kolik jsme zde sebrali) a procházíme do té doby, než dojdeme do stavu, jehož vrchol je cílový (počet pomerančů nás zde nezajímá, chceme se dostat co nejrychleji do cíle bez závislosti na tom, kolik nám jich zbyde). Správnou odpovědí je pak vzdálenost do tohoto cílového stavu.

Jak se nám změní časová složitost? Vrcholů tohoto grafu je desetkrát více než předtím, počet hran se může také takto zvětšit. Desítka je ale stále rozumná konstanta, která se „vejde do óčka“, a tak složitost paměťová i časová je stále $\mathcal{O}(10N + 10M) = \mathcal{O}(N + M)$, kde N je počet vrcholů a M počet hran.

Zbývá jen doplnit, že při implementaci algoritmu není nutné si explicitně takový graf stavět. Stačí si udržovat původní síť představující mapu města a ke každému vrcholu připojit pole indexované od 0 do 10, což odpovídá počtu pomerančů – prvky pole jsou vzdálenosti příslušných stavů.

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

Kuba Maroušek

28-1-2 Zapalování kostek

Se soupeřem se střídáme v zapalování kostek v pyramidě a chtěli bychom být tím, kdo zapálí poslední kostku. Jak něco takového udělat?

Zadání sice mělo lehčí variantu, ale pojďme rovnou vyřešit úlohu bez dalších omezení, strategie pro konkrétní K nám z toho vypadnou samy.

Kdyby byly pyramidy dvě, obě stejné a každá pro jednoho hráče, nemá začínající hráč šanci. Jeho soupeř se totiž může „opičit“, tedy zahrát na své pyramidě vždy to, co hrál první hráč na té své. Tím pádem dokud by měl první hráč co hrát, měl by co hrát i druhý hráč, až by druhý hráč nakonec vyhrál.

To ale znamená, že kdyby se nám prvním tahem podařilo rozdělit pyramidu na dvě stejné menší, můžeme se my opičit po soupeři a tím vyhrát. A přesně

to zvládneme. Pro lichá K je postup přímočarý – zapálíme prostřední kostku v prvním patře. Pro sudá nám stále stačí rychlé zamyšlení a zjistíme, že chceme pálit prostřední kostku ve druhém patře.

Tím nám vzniknou dvě stejné, izolované pyramidy (pro sudá K se jejich spodní řady dotýkají, ale to nevadí, protože sousední kostky se nemohou navzájem zapálit). Bez ohledu na to, jaké kostky zapálíme v jedné z nich, druhá zůstane nedotčena.

Po rozdělení tedy kopírujeme soupeřovy tahy. Soupeř sice může hrát v kterékoliv z menších pyramid, ale to my také. Vybereme si tedy vždy tu opačnou, takže po našem tahu budou opět obě pyramidy vypadat úplně stejně.

Soupeř musí dříve nebo později zapálit poslední nehořící kus jedné pyramidy. V té chvíli my zapálíme stejný kus druhé pyramidy, čímž vyhrajeme. A jelikož kostek je konečně mnoho a v každém tahu se zapálí alespoň jedna kostka, dojde i k naší výhře v konečném čase.

Karolína „Karryanna“ Burešová

28-1-3 Bourání komínu I

28-1-4 Bourání komínu II

Jak seřadit bomby, aby celková energie potřebná na zbourání komínu byla co nejmenší?

Vyřešme nejdříve lehčí variantu úlohy, kdy bomby dohromady přesně vystačí na zbourání komínu. Představme si, že už máme nějaké (libovolné) pořadí bomb zvolené. Zaměřme se nyní na dvě bomby, které použijeme jako poslední (označme si je A a B). Ty váží w_A a w_B a zničí d_A, resp. d_B metrů komínu. Před jejich použitím je komín vysoký $d_A + d_B$.

Pokud bomby použijeme v původním pořadí, tedy nejdřív A a potom B, spotřebujeme $w_A \cdot (d_A + d_B) + w_B \cdot d_B$ jednotek energie. Kolik energie spotřebujeme, pokud je prohodíme? Přece $w_B \cdot (d_A + d_B) + w_A \cdot d_A$.

Prohození se tedy vyplatí, pokud platí následující nerovnost:

$$w_A \cdot (d_A + d_B) + w_B \cdot d_B \leq w_B \cdot (d_A + d_B) + w_A \cdot d_A$$

Závorky si roznásobíme, odečteme stejné členy z obou stran a tím nám zůstane $w_A \cdot d_B \leq w_B \cdot d_A$ (1).

KSP

To si upravíme na $w_A/d_A \leq w_B/d_B$. Číslu w_A/d_A říkejme *nákladnost* bomby A. Už víme, že z posledních dvou bomb se vyplatí použít nejdříve tu méně nákladnou.

Tato úvaha ovšem neplatí jen pro poslední dvě bomby. Podíváme-li se na libovolné dvě po sobě následující bomby A a B, dostaneme nerovnost podobnou té předchozí. Jen pokud po jejich použití zbude místo holé země komín výšky D, musíme obě vynést do výšky o D vyšší než v minulém případě. Tedy na obou stranách nerovnosti přibude člen $(w_A + w_B) \cdot D$, který ale můžeme hned odečíst.

řešení

Opět tedy platí, že pokud je w_A/d_A větší než w_B/d_B, vyplatí se A a B prohodit.

Takovéto prohazování bychom mohli opakovat, dokud je někde v naší posloupnosti nákladnější bomba těsně před méně nákladnou. To ale není nic jiného než bublinkové třídění![52] Z tohoto příměru jsou hned jasné dvě věci: prohazování se časem zastaví a výsledná posloupnost bomb bude setříděná vzestupně podle nákladnost. Ta je tedy také správným řešením, protože kdykoli posloupnost není setříděná, existuje dvojice, kterou můžeme prohodit a řešení zlepšit.

My samozřejmě nemusíme třídit bublinkově, ale můžeme použít nějaký lepší algoritmus (např. MergeSort). Tak získáme řešení s časovou složitostí $\mathcal{O}(N \log N)$, kde N je počet bomb.

Drobná poznámka na okraj: nerovnici (1) jsme mohli zrovna tak upravit na $d_B/w_B \leq d_A/w_A$. Poměru d_A/w_A, vyjádřenému v „metrech na kilo“, říkejme třeba *efektivita* bomby A. V tomto případě naopak platí, že bomby s největší efektivitou chceme použít jako první, tedy musíme třídit sestupně.

Těžší varianta

Co ale uděláme, když jsme si pořídili bomb víc a všechny naráz by byly schopné zbourat komín větší než náš?

Nejprv si rozmyslete, že bez ohledu na to, které bomby si vybereme, opět je chceme seřadit nejvýhodnějším způsobem, tedy vzestupně podle nákladnosti. Tak si bomby setřiďme už na začátku, a pak teprve vybírejme, které použít.

Ukážeme si nejprve „hloupé“ řešení, které zkouší všechny možnosti, a pak ho zkusíme vylepšit. Bomby procházíme postupně od nejméně nákladné. U každé se můžeme rozhodnout, jestli ji použít, nebo ne. Pro obě z těchto možností

[52] `http://cs.wikipedia.org/wiki/Bubblesort`

pokračujeme rekurzivně v procházení zbylých bomb. Průběžně si hlídáme, aby celková síla vybraných bomb nepřekročila výšku komínu, a počítáme spotřebovanou energii. Na konci jen vybereme nejlevnější variantu. Takové řešení určitě funguje, ale možností, jak vybrat bomby, je celkem 2^N. To je moc na to, abychom je stihli vyzkoušet všechny.

Když se na průběh vybírání podíváme podrobněji, zjistíme, že se často opakovaně dostaneme do podobné situace. Například se může více způsoby stát, že po m krocích výběru skončíme s bombami, které sníží komín na výšku h. Pro každou z těchto variant pak zkoušíme všechna možná pokračování, přestože zřejmě stačí uvažovat jen tu nejlevnější a ostatní zahodit.

Neboli pro každé m a h bychom chtěli spočítat, jak nejlevněji snížit komín na výšku h za použití některých z m nejefektivnějších bomb. K tomu se nám bude hodit pozorování, že výška komínu v průběhu bourání bude vždycky mezi 0 a K.

Tyto hodnoty si tedy můžeme ukládat do dvourozměrného pole P velikosti $(N+1) \times (K+1)$. Budeme je počítat postupně pro $m = 0, \ldots, N$. Na začátku víme, že $P[0][K] = 0$ (když nic neuděláme, „zbouráme“ komín na původní výšku K s nulovou cenou), a na ostatních políčkách $P[0]$ bude „nekonečno“, tedy nějaká nekřesťanská cena znamenající, že zbourat komín na takovouto výšku zatím neumíme.

Když zpracováváme m-tou bombu, už známe všechny hodnoty v $P[m-1]$ a rádi bychom vyplnili $P[m]$. To uděláme tak, že budeme procházet pole $P[m-1]$ pro h od nuly do K, a když na výšce h narazíme na cenu bourání z minula (což je minimální cena, jakou jsme schopni vydat za zbourání komínu do výšky h pouze s využitím bomb před m), tak do pole $P[m]$ zapíšeme nové ceny dvou bourání.

Jedna cena bude za bourání, u kterého jsme bombu m nepoužili (tedy $P[m][h] = P[m-1][h]$, cena zůstává stejná), druhá, když ji zkusíme použít. Do pole $P[m][h-d_m]$ zapíšeme $P[m-1][h]+h \cdot w_m$, tedy cenu bourání s předchozími bombami plus cenu za vynesení bomby m na vršek komínu.

Psali jsme ale, že bombu m zkusíme použít. Ono se to nemusí povést, třeba když je moc silná a $h - d_m < 0$. Pak to samozřejmě zapisovat nebudeme. Nebo jsme se s předchozími bombami dostali na stejnou výšku levněji. Pak bombu m zahodíme. (Ale opatrně, bude se ještě hodit.)

A to je celé. Když přežijete zahazování nepoužitých bomb, můžete si na konci na políčku $P[N][0]$ přečíst cenu nejlevnějšího postupu na zbourání, nebo tu nekřesťanskou cenu za ruční rozebrání, protože bombami to nejde.

Ještě si všimněte, že nemusíme udržovat v paměti celé P. V každém kroku pracujeme vždy jen s posledními dvěma řádky (z jednoho čteme a do druhého zapisujeme), stačí si tedy pamatovat ty.

To můžeme udělat například tak, že pole P bude velké jen $2 \times (K+1)$ a místo $P[m]$ budeme používat $P[m \bmod 2]$, kde $m \bmod 2$ je parita m (zbytek po

dělení dvěma). Vzhledem k tomu, že dvě po sobě jdoucí m mají opačnou paritu, ve chvíli, kdy zapisujeme do jednoho řádku P, bude ten druhý obsahovat právě čísla zapsaná v minulém kroku.

Paměťová složitost je $\mathcal{O}(N + K)$, časová $\mathcal{O}(N \cdot \log N + NK)$.

Známe cenu, ale kterých bomb?

Takto jsme tedy získali nejlevnější cenu. Kdybychom chtěli vědět i to, jaké bomby jsme použili, pak můžeme buďto použít pole čísel 0 až K pro každou bombu, a ne jen dvě, a ke každé ceně si zapisovat i předchozí použitou bombu (podobně jako u hledání nejkratší cesty). Tím se ale zhorší paměťová složitost na $\mathcal{O}(NK)$.

KSP

Nebo můžeme mít pole jen dvě, ale použité bomby si udržovat ve spojových seznamech vedle ceny. V nejhorším případě ovšem bude potřeba také $\mathcal{O}(NK)$ paměti.

A co s těmi nepoužitými bombami? Odneste si je v batohu! Máte batoh s nosností K, bomby mají váhy w_i a na černém trhu za ně dostanete v_i. Odneste si takové bomby, aby jejich cena byla co největší a abyste nepřekročili nosnost (a mohli do letadla ;)). To je známý kombinatorický „problém batohu“. Pokud je K tak malé, že si můžete dovolit mít v paměti takto dlouhé pole, pak se dá problém řešit dynamickým programováním. Je to velmi podobné jako naše úloha. A to už přece umíte!

řešení

Program (C):

`http://ksp.mff.cuni.cz/viz/28-1-4.c`

Dominik Macháček

28-1-5 Likvidace plísně

Úloha byla vskutku jednoduchá a neskrýval se za ní žádný složitý trik. Pokud jste si uvědomili několik jednoduchých pozorování, napsání programu již bylo hračkou.

Prvním krokem je uvědomit si, že se nám vždy vyplatí přesunovat jenom na úplně nové (prázdné) hromádky. Pokud bychom chtěli přesunout nějaké vrstvy na existující hromádku, tak přesunem vrstev na novou můžeme jen zkrátit celkovou dobu odstraňování („nezaplácneme“ si druhou hromádku dalšími vrstvami).

Druhým pozorováním je, že všechno odstraňování plísně má smysl provádět až po všech přesunech. Použijeme klasický postup důkazu a budeme uvažovat, že by existovalo nějaké optimální řešení, které by po nějakých krocích odstraňování plísně provádělo ještě přesuny. Ale u takového řešení můžeme modifikovat přesuny tak, aby za každý krok odstraňování braly o jednu vrstvu více (o tu, kterou by odstraňování odmazalo), a přemístíme všechna odstraňování na konec.

Díky tomu odmažeme minimálně stejný počet vrstev, takže jsme tím převedli libovolné jiné řešení na alespoň stejně dobré splňující naší podmínku. Nějaké z optimálních řešení má tedy všechna mazání až na konci.

Stačí nám jen postupně pro všechny délky mazací části (označme si ji k) zkusit spočítat, kolik kroků potřebujeme na rozdělení hromádek tak, aby byly všechny maximálně k vrstev vysoké. To spočítáme jednoduchým cyklem (počet přesunů na rozdělení hromádky vysoké h je $\lceil h/k-1 \rceil$). Ze všech výšek k vybereme tu nejlepší.

Pokud N udává počet hromádek a V maximální výšku plísně, tak časová složitost algoritmu je $\mathcal{O}(V \cdot N)$.

Program (C):

`http://ksp.mff.cuni.cz/viz/28-1-5.c`

Jirka Setnička

28-1-6 Úloha z ovladače

Úlohu si můžeme představit tak, že posouváme okénkem délky K po nekonečně dlouhé posloupnosti čísel a hlásíme, jaké je zrovna minimum uvnitř okénka.

Kdybychom minimum pokaždé počítali znovu, trvalo by to $\mathcal{O}(K)$. Jak to udělat lépe? Rozmysleme si, jak se posunutím okénka může minimum změnit. Označme původní prvky $x_0, \ldots, x_{K-1}$ (x_m byl nejmenší) a nové $x_1, \ldots, x_K$.

- Pokud $x_K \leq x_m$, je novým minimem x_K.
- Pokud $x_K > x_m$ a $m > 0$ (staré minimum dosud okénko neopustilo), zůstává minimem x_m.
- Pokud $x_K > x_m$ a $m = 0$ (staré minimum právě vypadlo), nevíme, kde minimum leží.

V posledním případě tedy musíme všechny prvky okénka znovu projít, což zase trvá $\mathcal{O}(K)$. A co hůř, může se to stát pokaždé: představte si případ, kdy zadané prvky tvoří rostoucí posloupnost.

Mohli bychom si pomoci haldou (nebo vyhledávacím stromem), kde bychom si pamatovali celý obsah okénka. Pak by jedno posunutí trvalo $\mathcal{O}(\log K)$. My to ale umíme lépe, poslyšte, jak.

Přemýšlejme nad tím, jak přesně vypadá situace, kdy jsme o minimum právě přišli. Čím ho nahradíme? Nejmenším z prvků, které leží napravo od něj. Prvek s touto vlastností si opět můžeme průběžně udržovat, jenže co když pak vypadne i ten? Tak si pojďme pamatovat náhradu i za něj, atd.

Přehledněji řečeno, budeme si udržovat následující posloupnost:

m_0 := poloha aktuálního minima,
m_1 := poloha nejmenšího z prvků ležících vpravo od m_0,
$\vdots$
m_r := poloha nejmenšího z prvků vpravo od m_{r-1}.

Při každém posunutí okénka nyní provedeme toto:

- Pokud m_0 vypadlo z okénka, smažeme ho z posloupnosti.
- Dokud je nový prvek menší než x_{m_r}, smažeme m_r.
- Přidáme na konec posloupnosti pozici nového prvku.
- Nahlásíme prvek x_{m_0} jako minimum nového okénka.

Snadno si rozmyslíme, že tím vznikne posloupnost požadovaných vlastností pro nové okénko.

Opravdu jsme si tím pomohli? V nejhorším případě přeci můžeme mazat až K hodnot m_i, takže časová složitost je stále $\mathcal{O}(K)$! V této temnotě se ale mihotá světélko naděje: ke smazání všech prvků nemůže docházet často, protože prvky se doplňují jen po jednom.

Dokážeme, že posunutí okénka má *konstantní amortizovanou složitost.* Tím se myslí, že provedeme-li n posunutí, trvají dohromady $\mathcal{O}(n)$. Všechny operace totiž buďto přidávají prvky do posloupnosti, nebo je odebírají. Těch přidávacích je nejvýše n, neboť pokaždé přidáme právě jeden prvek. A odebíracích je nejvýše tolik, kolik je přidávacích, protože žádný prvek nemůžeme smazat víckrát.

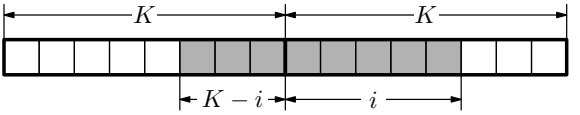
Program (C):
`http://ksp.mff.cuni.cz/viz/28-1-6-amort.c`

Řešení rozkladem na bloky

Ukážeme si ještě jedno amortizovaně konstantní řešení, založené na úplně jiné myšlence.

Vstup budeme dělit na bloky o K prvcích. Pro každý blok $x_1, \ldots, x_K$ si budeme průběžně počítat *prefixová minima* $\min(x_1, \ldots, x_i)$. Jakmile blok skončí, dopočítáme i *suffixová minima* $\min(x_j, \ldots, x_K)$.

Jak je vidět na následujícím obrázku, okénko délky K můžeme vždy rozdělit na prefix aktuálního bloku a suffix toho předchozího:



Minimum aktuálního okénka tedy můžeme spočítat v konstantním čase z předpočítaných hodnot.

Časová složitost vyjde opět amortizovaně konstantní, protože předvýpočet nás stojí $\mathcal{O}(K)$ na konci bloku o velikosti K. Stačí tedy, aby každý prvek přispěl časem $\mathcal{O}(1)$ na budoucí zpracování hotového bloku.

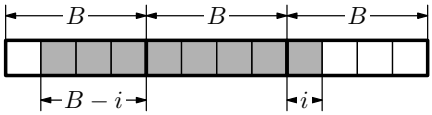
Čistokrevné konstantní řešení

Ačkoliv to může znít neuvěřitelně, existuje i řešení, kterému stačí konstantní čas na posunutí okénka i v nejhorším případě. Zkusíme „deamortizovat“ trik s rozkladem na bloky (mimochodem, první řešení s posloupností náhradních prvků deamortizovat neumíme).

Místo toho, abychom postupně střádali prvky a pak najednou zpracovali celý blok, zkusíme ho zpracovávat postupně: s každým novým prvkem kousek (nejdřív suffix délky 1, pak délky 2, atd.). To ovšem nemůže vyjít, protože hned u prvního prvku následujícího bloku potřebujeme nejdelší suffix toho předchozího.

Zachráníme to elegantním trikem: zvolíme jinou velikost bloku než K, konkrétně $B = K/2$ (pro jednoduchost předpokládejme, že K je sudé; lichá K dořešíme později).

Jak ukazuje následující obrázek, okénko velikosti K zasahuje do tří bloků: z aktuálního používá prefix, minulý pokrývá celý a z předminulého používá suffix:



Tím jsme si sice výpočet trochu zkomplikovali, ale teď mezi ukončením bloku a okamžikem, kdy potřebujeme jeho suffixová minima, leží alespoň jeden další blok, během kterého můžeme minima počítat.

Přesněji řečeno: pokud jsme právě dostali i-tý prvek aktuálního bloku B_t, provedeme následující:

1. Spočítáme i-té prefixové minimum bloku B_t.
2. Spočítáme i-té suffixové minimum bloku B_{t-1}.
3. Vrátíme jako výsledek minimum z těchto hodnot:
4. i-té prefixové minimum bloku B_t,
5. minimum celého bloku B_{t-1} (to je také prefixové minimum, takže už je spočítané),
6. i-té suffixové minimum bloku B_{t-2}.

Toto vše spočítáme v čase $\mathcal{O}(1)$, na udržování prefixových a suffixových minim spotřebujeme $\mathcal{O}(B) = \mathcal{O}(K)$ buněk paměti.

Zbývá dořešit okénka liché délky: pro ta postačí nastavit $B = \lceil K/2 \rceil$. Podle obrázku si rozmyslíme (dobře se to představuje třeba pro $K = 7$, kdy bude $B = 4$), že vše vyjde správně. Pokud blok právě začal ($i = 0$), potřebujeme celý minulý blok a suffix předminulého. Pokud se blok právě chystá skončit ($i = K - 1$), stále

minulý blok pokrýváme celý, takže se nestane, že bychom potřebovali suffixové minimum, které jsme dosud nespočítali.

(Mimochodem, i kdyby nám to pro lichá K takhle hezky nevyšlo, mohli bychom si pomoci oklikou: okénko bychom zmenšili na $K-1$ prvků a navíc bychom si pamatovali seznam posledních K prvků. Pak bychom jako výsledek vraceli minimum z minima okénka a nejstaršího prvku seznamu.)

Gratulujeme Jirkovi Sejkorovi, který jako jediný vymyslel řešení sice složitější než naše vzorové, ale také pracující v konstantním čase.

KSP

Program (C):

`http://ksp.mff.cuni.cz/viz/28-1-6-worst.c`

Martin „Medvěd“ Mareš & Václav Končický

28-1-7 Vyříznutý kus mříže

řešení

Máme mnohoúhelník s N vrcholy v mřížových bodech a chceme spočítat, kolik mřížových bodů se nachází na jeho obvodu. Vrcholy máme na vstupu v pořadí, v jakém jsou na obvodu. Naše řešení postavíme z krabičky, která umí počítat mřížové body na jedné straně. Pustíme ji na každém páru po sobě následujících bodů na obvodu. Kdybychom ale sečetli počet mřížových bodů na všech stranách, nedostali bychom správný výsledek: každý vrchol je mřížový bod, který leží na dvou stranách, proto musíme ještě od součtu odečíst N.

Jak spočítáme počet mřížových bodů na straně? Nechť naše strana vede z bodu (a, b) do bodu (c, d). Označme si $c-a$ jako Δx a $d-b$ jako Δy, a odteď jenom počítejme počet mřížových bodů na úsečce $(0, 0) \ldots (\Delta x, \Delta y)$. Tím jsme jenom posunuli úsečku do počátku – to počet mřížových bodů nijak nezmění.

Vodorovné a svislé úsečky, tj. ty, kde Δx nebo Δy je 0, můžeme ošetřit zvlášť (počet mřížových bodů na nich je absolutní hodnota toho Δx nebo Δy, které není nulové).

Jeden ze způsobů, jak spočítat počet mřížových bodů na úsečce, je zkoušet všechny hodnoty x od 0 do Δx. Ke každému takovému x na naší úsečce leží právě jeden bod (x, y) a pro něj platí $y = x \cdot \Delta y / \Delta x$. Pokud nám pro nějaké x vyjde y celočíselné, započítáme si najitý mřížový bod. Protože se chceme vyhnout nepřesnému dělení čísel s plovoucí čárkou, test na celočíselnost můžeme napsat třeba jako `(deltaY * x) % deltaX == 0`.

Takové řešení funguje, ale má jednu nevýhodu: každou stranu projíždíme celou od 0 do Δx, a proto je jeho složitost přímo úměrná obvodu mnohoúhelníka (respektive součtu Δx přes všechny strany). Počítat mřížové body na úsečce $(0, 0) \ldots (1, 1)$ bude mnohem rychlejší, než kdyby končila v $(100, 100)$.

Podívejme se na ten mřížový bod, který má z vnitřních mřížových bodů nejmenší obě souřadnice. Ať je to bod (p, q).

Použijeme dvě zajímavé vlastnosti bodu (p, q). Zaprvé pokud si vybereme jakýkoliv jiný mřížový bod (s, t), tak musí být souřadnice (s, t) násobek (p, q).

Zkusme si představit, že (s, t) není násobek (p, q). Když je (s, t) mřížový bod na úsečce, tak určitě i $(s - p, t - q)$ je mřížový bod na úsečce. Odečítejme od (s, t) násobky (p, q) tak dlouho, dokud to už nejde bez toho, abychom šli do záporných souřadnic. Dostaneme z toho mřížový bod $(s', t') = (s, t) - k \cdot (p, q)$. Protože další odečtení (p, q) by šlo do záporných souřadnic, je jedna ze souřadnic (s', t') menší než souřadnice (p, q), a to je zase spor s volbou (p, q). Proto je určitě (s, t) násobek (p, q).

Tohle je super: když najdeme správně bod (p, q), jsou všechny mřížové body na úsečce jeho násobky, takže nám stačí podělit $\Delta x/p$, přičíst jedničku za mřížový bod v počátku a máme počet mřížových bodů na celé úsečce.

Jak ale najdeme (p, q)? K tomu použijeme druhou vlastnost: p a q musí být nesoudělná čísla. Měla-li by společného dělitele r, pak by i $(p/r, q/r)$ byl mřížový bod, ale to by byl spor: my jsme si přece vybrali (p, q) tak, aby první souřadnice byla co nejmenší a $p/r < p$.

Čísla p a q jsou tedy nesoudělná a $p/q = \Delta x/\Delta y$, protože leží na úsečce. K nalezení p a q nám vlastně stačí zkrátit zlomek $\Delta x/\Delta y$ do základního tvaru!

Krácení zlomku se dělá tak, že najdeme největšího společného dělitele Δx a Δy. Po vydělení Δx a Δy jejich největším společným dělitelem dostaneme p a q. Na hledání největšího společného dělitele použijeme Euklidův algoritmus, který doběhne v čase $\mathcal{O}(\log \min\{\Delta x, \Delta y\})$. Jeho detaily si můžete dohledat v naší kuchařce o teorii čísel.[53] To je podstatné zlepšení proti posouvání se po všech potenciálních mřížových bodech, které stojí $\Theta(\Delta x)$.

Trochu si ještě zjednodušíme počítání. Určení souřadnic (p, q) vlastně vůbec není potřeba: poté, co spočítáme p jako $\Delta x/\operatorname{nsd}(\Delta x, \Delta y)$, ho používáme na určení počtu mřížových bodů: $1 + (\Delta x/p)$. Stačí to trochu upravit, a vidíme, že výsledek je rovný $1 + \operatorname{nsd}(\Delta x, \Delta y)$ a p nikde nepotřebujeme. Nakonec se vyhneme nehezkému odečítání N v posledním kroku: místo $\operatorname{nsd}(\Delta x, \Delta y) + 1$ budeme sčítat jenom $\operatorname{nsd}(\Delta x, \Delta y)$, což N od součtu automagicky odečte.

Celý algoritmus doběhne v čase $\Theta(N \cdot \log \min\{\Delta x, \Delta y\})$. Navíc protože si nepotřebujeme pamatovat pozice všech bodů, můžeme pro všechny úsečky na vstupu spočítat počet mřížových bodů a úsečku rovnou zahodit. Stačí nám konstantní paměť.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-1-7.py`

Michal „Prvák“ Pokorný

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

28-1-8 Programování podle Darwina

Úloha 1

V této úloze bylo cílem vyzkoušet si genetický algoritmus, jaké má vlastnosti a jak jednotlivé parametry ovlivňují jeho chování. Z mnohých věcí jste mohli vypozorovat například následující:

Mutace: Pokud zvýšíme pravděpodobnost mutace, tak se algoritmus chová více náhodně. Ze začátku má sice rychlý nástup, ale pak pro náš problém platí, že čím máme lepšího jedince, tím je menší pravděpodobnost, že jej vylepšíme a větší pravděpodobnost, že jej zhoršíme. Pak pokud nastavíme pravděpodobnost na změnu jednoho bitu příliš velkou, tak mutace dělá větší změny a tím se i chová méně konzistentně.

Selekce: Při řešení jedničkového problému by použití ruletové vs. turnajové selekce nemělo mít velký vliv na výsledek. Ruletová má výhodu v tom, že více preferuje lepší jedince, zatímco turnajová je celkově jednodušeji implementovatelná a paměťově i časově efektivnější.

Ti bystřejší z vás si navíc všimli, že pokud předem víme, kolik jedinců pomocí ruletové selekce chceme vybrat, tak ji celou můžeme implementovat efektivněji. To jest v čase $O(P+N \log N)$, kde P je velikost populace a N je počet vybíraných jedinců.

Křížení a velikost populace: Tyto dva parametry spolu implementačně příliš nesouvisí, ale fakticky spolu souvisí hodně. Velikost populace určuje variabilitu dat, která na začátku v jedincích máme. Čím více jedinců, tím více různých dat. Křížení pak během algoritmu tato data dává různě dohromady a díky selekci tvoří lepší jedince. Při malé populaci se nám stane, že za chvíli nemáme co nového mezi jedinci křížit a musíme čekat na štěstí v mutaci, zatímco ve velkých populacích se informace zbytečně opakují a výpočet nám to akorát zpomalí.

Velikost jedince a velikost populace: Tyto dva parametry spolu také souvisí. Velikost populace bychom měli volit v závislosti na velikosti jedince. Tj. aby byla dost velká šance, že na začátku budou jedinci dohromady obsahovat správná data pro všechny bity a že je jednou nešťastnou mutací či křížením zase neztratíme.

Fitness funkce: Ta v prvním případě počítala jen počet jedniček. Zajímavější byl druhý případ. Pokud se snažíme vyvinout jedince, kde se jedničky a nuly střídají, dostaneme fitness funkci se dvěma možnými maximy, které jsou k sobě inverzní. Pak v průběhu algoritmu nám „jdou proti sobě“ a jedinci obou frakcí spolu „soupeří“. Jedni se snaží mít na sudých pozicích nuly, na lichých jedničky a druzí naopak.

Na tom jste si měli hlavně vyzkoušet, jak genetický algoritmus (ne)funguje dobře, pokud máme fitness funkci s více rozdílnými maximy, které jsou velmi odlišného tvaru. V tomto případě to lze vyřešit třeba fintou, že postavíme fitness

funkci tak, aby preferovala jedince s jedničkami na lichých pozicích. Obecně ale do fitness funkce tolik nevidíme, abychom podobné finty mohli dělat.

Úloha 2

V této úloze již byla potřeba jistá modifikace řešení a kreativní náhled na problém.

Celý genetický algoritmus jednoduše přepíšeme tak, aby namísto s binárními jedinci fungoval s jedinci celých čísel 0 až 6. Křížení zůstane nezměněno a mutace místo překlopení generuje náhodnou novou hodnotu.

Fitness funkci naprogramujeme podle zadání. Ohodnotíme jedince podle toho, jak dobře rozděluje zadané věci mezi sedm loupežníků. Pokud používáme ruletovou selekci, tak navíc hodnotu překlopíme na $1/(x+1)$, abychom mohli maximalizovat a ne minimalizovat.

Tak to by bylo. Pustíme algoritmus a ono to moc dobře nefunguje. Proč?

Zkusme se zamyslet, co fakticky znamenají křížení a mutace v rámci tohoto problému. Jednobodové křížení vezme dvě různá řešení a v náhodném bodě je zkříží. Proč by takové křížení mělo jakkoliv směřovat ke stejným součtům různých hromádek? Třeba díky fitness funkci, ale...

Stejně jako v případě střídajících se nul a jedniček, i tady máme více různých nejlepších řešení, které jdou „proti sobě“. Těch je alespoň 7! a faktoriál sedmi je 5 040, k tomu obsahuje ještě víc lokálních neoptimálních maxim, ze kterých se nedá jednoduše dostat. Křížení teda vypadá, že nám moc nepomůže.

A co mutace? Ta zas jen provede malou náhodnou změnu. Ta nám sice občas může přinést něco dobrého, ale samotná určitě nestačí, chce to něčím doplnit.

My ji doplníme chytrou mutací, to jest takovou, která využívá znalost řešeného problému. Ta si pro daného jedince spočítá velikosti jednotlivých hromádek, pak vybere náhodný předmět z nejtěžší hromádky a přendá jej na nejlehčí hromádku.

Taková mutace uměle cílí na část problému, která by měla pozitivně ovlivnit fitness funkci. To má výhodu v tom, že se na začátku algoritmu budeme efektivně blížit k relativně dobrému řešení. Bohužel nevýhodou je, že můžeme lehce uvíznout v nějakém suboptimálním řešení, ze kterého se přehozením jednoho předmětu nedostaneme. Jak ale uvidíme dále, nám tento operátor bude stačit.

V celém řešení použijeme pouze chytrou mutaci a normální mutaci. Křížení vůbec používat nebudeme. Populaci nepotřebujeme příliš velkou, bude stačit 50 jedinců, protože jedinci mezi sebou stejně neinteragují. Na druhou stranu jich bereme 50, abychom procházeli více náhodných změn najednou a ne jen jednu.

Celé to necháme běžet velké množství iterací, třeba 10 000, abychom určitě zkonvergovali a případně měli dost času se dostat z lokálních minim. To celé pustíme opakovaně v několika bězích, abychom několikrát začali s jinak nagenerovanými jedinci.

Na výsledcích nejtěžšího vstupu pak můžeme vidět, že za 10 000 iterací k optimálnímu řešení obvykle dojdeme. V nejlepších případech se tak stane řádově po stovkách iterací, jindy řádově po tisících iterací a jindy k optimu vůbec nedojdeme. Záleží na tom, jak se nám zrovna nagenerovala vstupní data a kam se výpočet nasměroval. Proto také nevsázíme jen na jeden výpočet a pouštíme algoritmus opakovaně s různě nagenerovanými počátky.

Toto řešení je implementované ve vzorovém kódu.

Program (C++):

`http://ksp.mff.cuni.cz/viz/28-1-8.cpp`

Karel Tesař

Druhá série

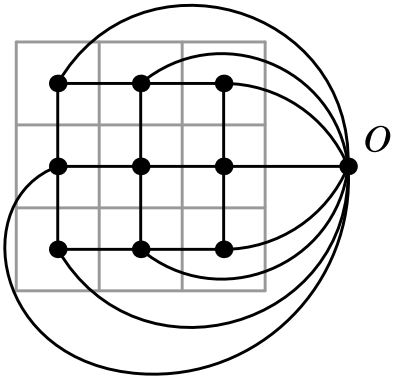
28-2-1 Potopa ve městě

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:[54] 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,[55] kde byly oproti naší úloze prohozené minimum a maximum.

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

[55] `http://ksp.mff.cuni.cz/viz/27-2-3`

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

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.[56]

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

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

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 nalézt 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.

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): *ze-*

lená 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,[57] 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

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

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".

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

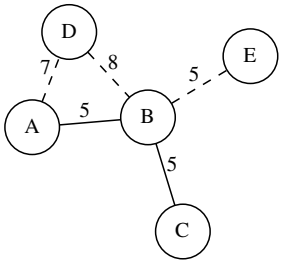
Hledáme nejdelší cestu v síti MHD, měřeno časem stráveným ve vozidlech. Vyřešíme nejprve lehčí verzi úlohy, tedy situaci, kdy si nemusíme nechávat časovou rezervu na přestup.

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:



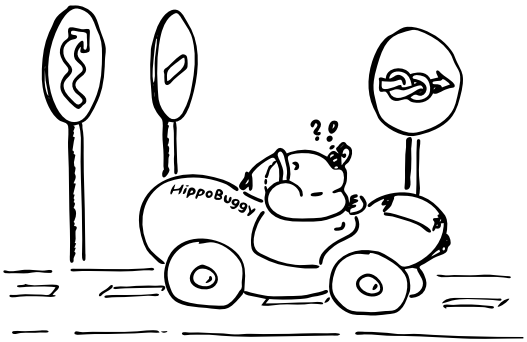
Síť je tvořena dvěma linkami, X (plná čára, jezdí každých 20 minut) a Y (čárkovaná čára, jezdí každých 30 minut). Zastávky jsou pro přehlednost označeny písmeny namísto čísel. Čísla na hranách značí jízdní dobu mezi zastávkami.

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.

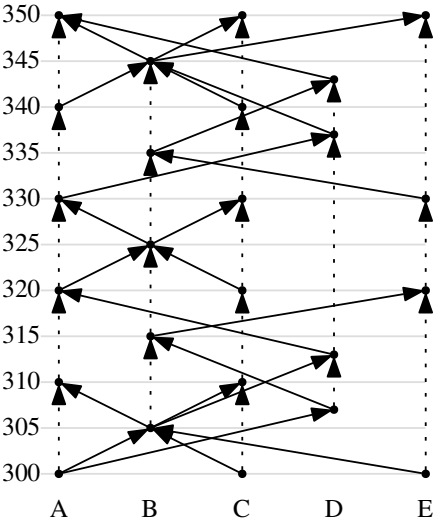
Vytvoříme si takzvaný *stavový prostor*. To je graf, jehož vrcholy popisují nějaký stav (situaci), ve kterém se můžeme nacházet, a hrany mezi nimi určují dovolené změny stavu. V našem případě budou stavy dvojice (z, t) popisující „jsem na zastávce z v čase t“.

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)[58] 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).[59] 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 $D(u) = 0$.

[58] `http://ksp.mff.cuni.cz/viz/kucharky/tezke-problemy`

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

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 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.

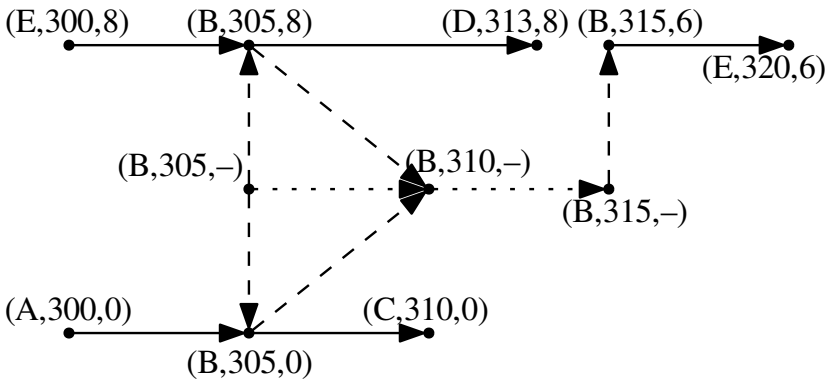
Namísto původní hrubé informace „jsem na zastávce z v čase t“ se budeme muset naučit rozlišovat mezi „stojím na zastávce z (venku) v čase t“ a „jsem ve vozidle spoje s, které právě zastavilo na z v čase t“. Tedy stav bude popsán trojicí (z, t, s), kde s je číslo spoje nebo „$-$“ reprezentující „stojím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

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

KSP

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`

řešení

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 nám nezbývá, než 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*.[60]

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.

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

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 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 bězích, 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 bězích 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

Třetí série

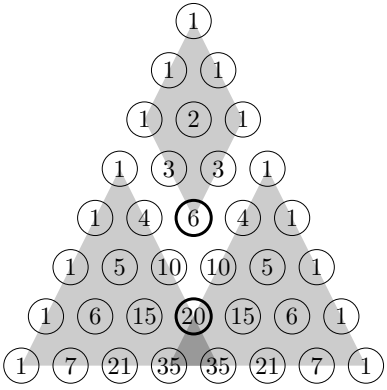
28-3-1 Pyramida z helmic

Úvodem dnešních řešení se vám všem musíme omluvit za naprosto neodpovídající ohodnocení pyramidové úlohy (a to ještě vypadalo krásně, že nejlehčí úloha vyšla jako první!). Pokud jste si tedy nad úlohou lámali hlavu a nemohli na nic kloudného přijít, možná jste jen projevili víc rozumu než orgové, kteří kdysi uvěřili tomu, že úloha je přece jednoduchoučká.

Doufáme, že příště bude bodové ohodnocení lépe vypovídat o obtížnosti úlohy, a hlavně že jsme nikoho od řešení neodradili. Ale teď už honem na řešení.

Snadno si můžeme rozmyslet, že pro $N \leq 2$ vždy vyhraje druhý hráč. Naopak dost komplikovaně můžeme dojít k tomu, že pro $N \geq 3$ existuje vyhrávající strategie pro prvního hráče.

Nejprve tuto vyhrávající strategii popíšeme, a pak teprve dokážeme, že opravdu funguje a soupeř nemá šanci. Zaveďme si teď pár označení, ať se nám strategie popisuje snáz. Předně, *lichým řádkem* budeme myslet řádek obsahující lichý počet helmic. *Drahý střed* pak bude prostřední helmice na nejspodnějším lichém řádku a *levný střed* prostřední helmice na druhém lichém řádku odspodu. Také si dovolme používat pojem pyramida, přestože tvar, který budou helmice v průběhu hry vytvářet, nebude vždy pyramidu připomínat.



1. Hned v prvním tahu shodíme levný střed. Tím jakoby vznikly dvě shodné pyramidy, které se částečně překrývají.

2. Potom budeme „kopírovat“ soupeřovy tahy, dokud soupeř nestrčí do drahého středu. Jinými slovy, soupeř strčí do nějaké helmice v jedné z pyramid, my strčíme do stejné helmice ve druhé pyramidě.
3. Když soupeř strčí do drahého středu, začneme hrát hladově. To znamená, že vždy shodíme tu helmici, která právě obsahuje nejméně kamínků.

Dokazování vezměme trochu na přeskáčku. Začněme tím, že opravdu můžeme hrát podle druhého bodu. Pyramidy vzniklé po našem prvním tahu totiž mají průnik (překryv) a nemusí být jasné, že pokud soupeř strčí do něčeho z tohoto průniku, můžeme kopírovat.

KSP

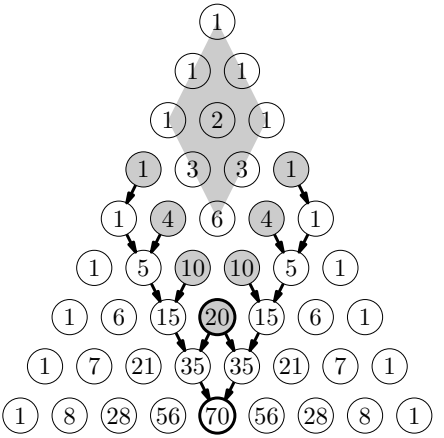
Ten průnik ale není velký: obsahuje drahý střed a případně (je-li N sudé) dvě helmice pod ním. Drahý střed máme ošetřený zvlášť. Helmice nikdy neshodí helmici stojící vedle ní, takže shodí-li soupeř jednu, my shodíme druhou. Kopírovat tedy můžeme vždy.

To, že každý tah kromě strčení do drahého středu umíme zkopírovat, je důležité. Právě proto to musí být soupeř, kdo strčí do drahého středu, případně ho shodí spolu s helmicí pod ním.

řešení

Jak se kopírovací část strategie projeví na počtu sebraných kamínků? Určitě platí, že když okopírujeme tah, sebereme maximálně tolik, kolik sebral soupeř. Ve skutečnosti sebereme ostře méně právě tehdy, když soupeř shodí něco z průniku pyramid. V takovém případě ale určitě shodil drahý střed. To dohromady znamená, že po konci kopírovací strategie se rozdíl našich a soupeřových kamínků zvýší alespoň o hodnotu drahého středu.

Nesmíme ale zapomínat, že před kopírovací částí jsme sami sebrali levný střed. My ovšem ukážeme, že součet kamínků v helmicích, které shodíme shozením levného středu, je o jedna menší než hodnota drahého středu. Když tohle platí, máme po kopírovací strategii alespoň o jeden kamínek méně než soupeř.



Indukcí podle r budeme dokazovat obecnější tvrzení: Počet kamínků získaných shozením střední helmice v r-tém lichém řádku shora je o jedna menší než počet kamínků ve střední helmici na $(r+1)$-ním lichém řádku.

Pro $r = 1$ naše tvrzení evidentně platí $(1 = 2 - 1)$. Indukční krok sledujme na obrázku. Pokud shodíme střed 3. lichého řádku, což je číslo 6, spadne celý „diamant“ a z indukce už víme, že jeho součet je 19.

Shodíme-li střed o řádek níž, tedy číslo 20, spadnou navíc dvě diagonály vedoucí od okraje ke středu 20. Šipky na obrázku ukazují, že každá z těchto diagonál se sečte na 35, takže obě diagonály dohromady na 70, což je přesně střed následujícího lichého řádku.

Ovšem číslo 20 leží v obou diagonálách, takže jsme ho započítali dvakrát. To je ale správně: jednou reprezentuje samo sebe, podruhé součet diamantu nad diagonálami zvětšený o jedničku. Analogická úvaha funguje pro libovolné r.

Hurá! Tím jsme ukázali, že po kopírovací části bude mít soupeř alespoň o jeden kamínek víc. Pokud ale soupeř shodí drahý střed přímo, pokračujeme třetí částí (hladovou). Teď musíme ukázat, že i z téhle části získáme nejvýše tolik, kolik soupeř.

Máme před sebou dvě symetrické pyramidy (které už možná jako pyramidy vůbec nevypadají) a chceme získat maximálně stejně, to by svádělo právě k nějaké kopírovací strategii. Jenže my začínáme a nemáme ani tu nejmenší jistotu, že soupeř bude naše tahy kopírovat. Co víc, nemáme ani jistotu, že soupeř bude hrát poslední.

Budeme tedy muset na důkaz toho, že naše strategie funguje, jít jinak. Držte si klobouky, pojedeme z kopce.

Nejprve si zavedeme další označení, tentokrát pro helmice, na kterých neleží žádná další (takže jejich shozením neshodíme nic dalšího). Těm budeme říkat *volné*.

Všimněme si, že mezi všemi helmicemi s momentálně minimální hodnotou je alespoň jedna volná. To plyne z toho, že směrem dolů hodnoty helmic neklesají. Také si snadno rozmyslíme, že my bereme vždy volnou helmici – alespoň nějaká z nich má nejmenší hodnotu, takže se nám určitě nevyplatí shazovat helmic víc.

Přiřaďme teď jednotlivým helmicím v jedné ze symetrických pyramid označení x_i taková, že $x_i \leq x_{i+1}$. Každé x_i se ve hře (resp. této hladové části hry) vyskytuje dvakrát – na začátku je jednou v první pyramidě, jednou ve druhé.

Helmice můžeme rozdělit na naše a na soupeřovy podle toho, kdo je shodil. K tomu si zavedeme další označení, a to *hlavní helmice*. To bude pro každý tah nejmenší z helmic, které hráč v daném tahu shodil. Pokud helmic shodil více, označíme ty ostatní jako *vedlejší*.

Snadno si všimneme, že všechny naše helmice jsou hlavní (jelikož nikdy neshodíme víc než jednu helmici naráz).

Nyní bychom chtěli spárovat své helmice se soupeřovými, a to tak, aby vždy naše helmice měla maximálně takovou hodnotu, jakou má spárovaná soupeřova helmice. Některé soupeřovy helmice možná zůstanou nespárované, ale to nám nijak nevadí. Kdyby se nám takové párování podařilo najít, musí být náš počet kamínků maximálně stejný jako soupeřův.

Představme si, že vždy spárujeme naši hlavní helmici s hlavní helmicí, kterou soupeř vezme v příštím tahu. Pro takové párování by nerovnost určitě platila. Jenže se může stát, že v posledním tahu hrajeme my, tedy zbývá jedna nespárovaná helmice, kterou už není s čím spárovat. Ukážeme ale, že v takovém případě můžeme helmice přepárovat.

Označme nespárovanou helmici jako X a její hodnotu jako x_i. Nyní se podíváme na suffix $x_i, \ldots, x_N$ všech hodnot, které byly ve hře. Existují dvě možnosti.

Zaprvé, existuje vedlejší soupeřova helmice s hodnotou z tohoto suffixu. Jelikož jsme zatím párovali vždy s hlavními helmicemi, je tato vedlejší helmice nespárovaná, a navíc má určitě aspoň stejnou hodnotu jako X. Tedy můžeme X spárovat s touto vedlejší helmicí. Tím máme vše spárováno a nerovnosti platí.

Zadruhé, žádná taková vedlejší helmice neexistuje. To znamená, že suffixu odpovídají pouze hodnoty hlavních helmic. Zároveň ovšem suffixu odpovídají hodnoty sudého počtu helmic, a jedna z těchto helmic je X. To znamená, že alespoň jedna hlavní helmice je spárovaná s nějakou, jejíž hodnota je menší než x_i.

Označme helmice v tomto páru jako A a B, nechť $A \leq B$ (tedy hodnota A v suffixu neleží, hodnota B ano). V dosavadním párování platí, že hodnota naší helmice je menší než soupeřovy, tedy A je naše. Přepárujme nyní X s B; A se stane nespárovanou.

Jelikož x_i bylo v suffixu nejlevěji, je určitě hodnota X neostře menší než hodnota B, takže nerovnosti nám stále platí. Může se zdát, že jsme si nepomohli, opět máme jednu nespárovanou helmici. Její hodnota ale určitě leží v řadě hodnot o jedna víc než vlevo, tedy opakováním přepárování spárujeme všechny helmice v konečném čase.

Tady ještě zdůrazněme, že pokud se dostaneme až do situace, kdy má nespárovaná helmice hodnotu x_1, musí mít soupeř nutně alespoň jednu vedlejší helmici. V opačném případě by muselo existovat párování vedoucí před suffix, ale před x_1 už nic není.

Vždy tedy umíme vytvořit párování takové, že naše helmice má maximálně stejnou hodnotu, jakou soupeřova, tedy i z hladové části hry budeme mít maximálně tolik, kolik získá soupeř.

Tím jsme dokázali, že popsaná strategie je skutečně vyhrávající. Ufff!

Karry Burešová

28-3-2 Líný písař

Označme si věty jako řetězce A a B, bez újmy na obecnosti předpokládejme, že $|A| \geq |B|$. Podívejme se nejprve, jakým způsobem sestrojíme řetězec S takový, aby obsahoval po odstranění některých znaků každou větu a také byl nejkratší, neboli aby tento řetězec byl nejkratší společnou nadposloupností A, B. Jak ale tento řetězec bude vypadat?

Určitě se nám vůbec nevyplatí mít délku $|S| > |A| + |B|$, jelikož bychom museli přidat další zbytečné znaky. Řetězec dále můžeme zkrátit tím, že znaky společné oběma větám zapíšeme do našeho řetězce S jen jednou.

Tyto společné znaky však vůbec nemusí být na stejných pozicích u obou řetězců. Příkladem jsou řetězce

$$A = \texttt{xAxBxCxDxE}, \quad B = \texttt{klmABCDkElm},$$

kde společné znaky `ABCDE` tvoří nejdelší společnou podposloupnost (zkráceně NSP). Znaky této NSP se ale v obou řetězcích vyskytují na různých indexech: v A na indexech $[1, 3, 5, 7, 9]$ a v B na indexech $[3, 4, 5, 6, 8]$. My potřebujeme zjistit jak samotnou NSP, tak indexy jejích znaků v obou řetězcích.

K vyřešení problému nalezení NSP lze využít dynamického programování, algoritmus je přímo popsán v kuchařce. Tento algoritmus nás stojí $\mathcal{O}(|A| \cdot |B|)$ času.

Jakmile máme indexy společných znaků z obou vět, můžeme sestrojit náš požadovaný řetězec S. Mějme i-tý společný znak, a_i a b_i indexy tohoto společného znaku v A a B. Potom postupně budeme zvyšovat i od 1 po délku NSP a při každé iteraci vypíšeme všechny znaky z A, B mezi $(i-1)$-ním a i-tým společným znakem a nakonec samotný i-tý znak. Tento algoritmus nám zabere $\mathcal{O}(|A| + |B|)$ času.

Celkově algoritmus zabere $O(|A| \cdot |B|)$ času. Důvod je takový, že nám právě nejvíce času potrvá vyhledání NSP. Samotné vypsání řetězce trvá jen lineárně, což nám asymptotickou složitost nezmění.

Jiný pohled na úlohu

Na tuto úlohu existuje i řešení založené na odlišné myšlence, které nakonec bude stejně efektivní. Představme si orientovaný graf, jehož vrcholy jsou reprezentovány uspořádanou dvojicí $[a_i, b_i]$. Tyto vrcholy si můžeme graficky znázornit, že jsou položené na mřížce o velikosti $|A|$, $|B|$, ve které souřadnice vrcholů určují indexy jednotlivých řetězců s tím, že vlevo nahoře máme vrchol $S = [0, 0]$ a vpravo dole je vrchol $C = [a_{n-1}, b_{n-1}]$.

Pro každý vrchol dále bude platit, že z něj vede hrana doprava a dolů vždy, pokud není poslední ve své souřadnici, a dále si zavedeme zkratkové hrany, které nám budou vést přes diagonálu dolů doprava. Tyto hrany budou vést z těch vrcholů, které nám označují znak společný oběma řetězcům.

Nyní, když máme takový graf postavený, nalezneme nejkratší cestu z vrcholu $[0,0]$ do vrcholu $[A,B]$. Jakmile jsme jednu takovou cestu našli, vydáme se po ní a zařídíme se podle toho, kterým směrem se rozhodneme vydat z vrcholu $[a_i,b_j]$:

- Pokud je následující vrchol cesty napravo od aktuálního, vypíšeme na výstup znak a_i.
- Pokud je následující vrchol cesty směrem dolů od aktuálního, vypíšeme znak b_i.
- Pokud se následující vrchol cesty nachází směrem po diagonále, je celkem jedno, který znak necháme vypsat, protože aktuální vrchol označuje společný znak.

KSP

Tímto jsme vypsali na výstup hledanou nejkratší společnou nadposloupnost. Samotné sestavení grafu nám potrvá $\mathcal{O}(|A| \cdot |B|)$, jelikož tento graf má počet vrcholů stejný, jako je součin délky obou řetězců $A \cdot B$. Nejkratší cestu potom umíme nalézt v lineárním čase jednoduchým prohledáním, takže nám to časovou složitost neporuší. Výsledný algoritmus má tímto stejnou asymptotickou složitost jako algoritmus popsaný výše.

řešení

Program (C):

`http://ksp.mff.cuni.cz/viz/28-3-2.c`

Program (C++):

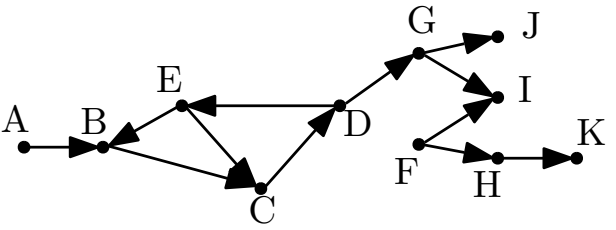
`http://ksp.mff.cuni.cz/viz/28-3-2.cpp`

Václav Končický a Karry Burešová

28-3-3 Formulář na zbroj

Nejprve si úlohu převedeme na grafovou. Pro každý formulář vytvoříme jeden vrchol. Říkejme, že formulář x (přímo) *závisí* na formuláři y, pokud k získání x musíme předložit vyplněné y. Pro každou takovouto dvojici vytvoříme v grafu orientovanou hranu $x \to y$. Některé formuláře jsou vydávány „zadarmo“ (nemají závislosti). Odpovídající vrcholy v grafu nemají žádné výstupní hrany – takovéto vrcholy obvykle nazýváme *stoky*.

Graf závislostí může vypadat třeba takto:



Stoky jsou v tomto případě vrcholy I, J, K. Dále řekneme, že nějaký formulář x *nepřímo závisí* na y, pokud z x vede orientovaná cesta do y. Takový x

nemůžeme získat, aniž bychom někdy předtím získali y. Například F nepřímo závisí na K, a vskutku: bez K nedostaneme H, a tedy ani F.

Pokud nějaký formulář leží na orientovaném cyklu, nepřímo závisí sám na sobě. Takový určitě nemůžeme získat: například abychom získali E, potřebujeme C, který vyžaduje D, a ten zase E. Jinými slovy abychom dostali E, museli bychom nejdřív mít E. To určitě nejde, protože úředníci nevydávají formuláře na dluh.

Dalším formulářem, který určitě nemůžeme získat, je A, protože závisí na B, který leží na cyklu.

Řešení prohledáváním do hloubky

Pro každý formulář budeme chtít určit jeho *ohodnocení,* které může nabývat jedné ze tří následujících hodnot:

- 0, pokud formulář lze získat vyplněný záporně
- 1, pokud formulář lze získat vyplněný kladně
- **X**, pokud formulář není možné získat

Ohodnocení formuláře x najdeme pomocí rekurzivní funkce Ohodnoť(x). Ta nejprve rekurzivním zavoláním získá ohodnocení všech přímých závislostí x, na jejich základě určí ohodnocení x, a to vrátí. To vlastně není nic jiného než prohledávání do hloubky.[61]

Protože funkce Ohodnoť může být na tentýž formulář zavolána mnohokrát a nechceme plýtvat časem opakováním toho samého výpočtu, použijeme dynamické programování.[62] Již spočítaná ohodnocení si budeme ukládat do pole H indexovaného čísly formulářů. Pokud je při zavolání Ohodnoť(x) hodnota $H[x]$ již známa, prostě ji rovnou vrátíme a nepočítáme znovu.

Kostra ohodnocovací logiky je jednoduchá: pokud některou ze závislostí x nejde získat (má ohodnocení **X**), ani x nemůžeme získat, tedy vrátíme rovněž **X**. Pokud naopak jde získat všechny, pak jde získat i x a jeho ohodnocení vytvoříme tak, že ohodnocení závislostí zkombinujeme předepsanou logickou funkcí.

Ale to nestačí. Pokud graf obsahuje cykly, takováto implementace by skončila nekonečnou rekurzí. Potřebujeme umět nějak cykly detekovat a vrcholy na nich označit jako nezískatelné.

To můžeme udělat například tak, že na začátku funkce Ohodnoť(x) si vrchol x označíme jako *rozpracovaný* (a na konci toto označení zase zrušíme). Pokud potom někdy při procházení závislostí narazíme na rozpracovaný vrchol, víme, že leží na cyklu. Proč? Pokud narazíme při prohledávání na rozpracovaný vrchol x, znamená to, že funkce Ohodnoť(x) aktuálně běží (v nějakém vyšším patře rekurze). Všechny vrcholy, které navštívíme, zatímco běží Ohodnoť(x),

[61] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`
[62] `http://ksp.mff.cuni.cz/viz/kucharky/dynamika`

patří mezi (nepřímé) závislosti x. Takže pokud narazíme na x, znamená to, že x závisí na x.

Zavedeme si dvě pomocné hodnoty, které se mohou vyskytovat v poli H, pokud ohodnocení daného vrcholu ještě neznáme:

- **?**, pokud jsme dané pole ještě nenavštívili
- **R**, pokud je vrchol právě rozpracovaný

Upravený algoritmus bude vypadat následovně:
OHODNOŤ(x):

1. Pokud $H[x] = \mathbf{R}$:
2. $\quad H[x] \leftarrow \mathbf{X}$, vrať $\mathbf{X}$.
3. Pokud $H[x] \neq$ **?**: vrať $H[x]$.
4. $H[x] \leftarrow \mathbf{R}$
5. Označ z_1 až z_k všechny závislosti x.
6. Pro $i = 1$ až k:
7. $\quad h \leftarrow$ OHODNOŤ(z_i)
8. $\quad$ Pokud $h = \mathbf{X}$:
9. $\quad\quad H[x] \leftarrow \mathbf{X}$, vrať $\mathbf{X}$.
10. $H[x] \leftarrow f(H[z_1], \ldots, H[z_k])$, přičemž f je logická funkce předepsaná pro formulář x (AND, OR nebo XOR).
11. Vrať $H[x]$.

Protože chceme zjistit ohodnocení všech formulářů, prostě funkci OHODNOŤ zavoláme postupně na všechny vrcholy. Celkem budeme potřebovat lineární čas, přesněji $\mathcal{O}(N + M)$, kde N je počet formulářů a M je celkový počet závislostí. To nahlédneme následovně. Vnitřní část funkce OHODNOŤ provedeme pro každý vrchol nejvýše jednou. Tím pádem nejvýše jednou provedeme vnitřní cyklus přes všechny hrany vycházející z daného vrcholu. Tudíž na každou hranu se podíváme maximálně jednou.

Ještě musíme uvážit čas strávený na prvních třech řádcích funkce OHODNOŤ, které mohou být provedeny víckrát. Ale funkce OHODNOŤ je volána jen ze dvou míst:

- Z hlavní smyčky, kde je volána jednou pro každý vrchol (celkem $N\times$).
- Rekurzivně z jiného volání OHODNOŤ, ale to se děje právě v místě, kde procházíme výstupní hrany. A už víme, že každou hranu navštívíme maximálně jednou, tedy těchto volání funkce OHODNOŤ je dohromady maximálně M.

Z tohoto odhadu je taky vidět, že teď už se náš algoritmus nemůže zacyklit.

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

Filip Štědronský

28-3-4 Katapulty

Podívejme se prvně na lehčí variantu. Nabízí se umisťovat katapulty hladově, tedy dát každý katapult co nejlevěji to půjde.

Řekněme, že projdeme bitevní lajnu zleva a při procházení si udržujeme množinu dosud neumístěných katapultů, které můžeme umístit na aktuální políčko.

Na každém políčku pak přidáme do množiny katapulty, které můžeme nově umístit, a poté z téhle množiny vezmeme katapult, jehož úsek končí nejdříve. Zkontrolujeme, že pravá hranice povoleného úseku je maximálně rovna aktuální pozici, a pokud ano, katapult umístíme na toto políčko. Pokud náhodou ne, zahlásíme, že řešení neexistuje. Budeme-li mít zrovna prázdnou množinu, jednoduše se posuneme na další políčko.

Snadno nahlédneme, že pokud náš algoritmus vydá řešení, bude toto řešení korektní – každý katapult bude ve svém úseku a zároveň bude na každém políčku jen jeden.

Musíme ale také ukázat, že pokud naopak ohlásíme neexistenci řešení, skutečně žádné řešení neexistuje. Když jsme vyhlásili neúspěch, máme nějaký katapult P, který jsme nedokázali umístit. Na každém políčku v úseku, kam smíme P umístit, už se ale vyskytuje jiný katapult, jehož úsek končí nejpozději stejně, tedy žádný z nich nemůžeme přesunout doprava.

Ještě musíme uvážit, jestli jsme některý z těch katapultů nemohli umístit více vlevo. Podobným argumentem ale ukážeme, že ne – když vezmeme největší levou hranici těchto katapultů a podíváme se na úsek mezi ní a začátkem úseku pro P, opět máme na každém políčku katapult, který nemůžeme přesunout doprava.

Takto dojdeme až na začátek lajny. Vlastně to znamená, že se snažíme umístit $L + 1$ katapultů na L políček, a to evidentně jít nemůže. Náš algoritmus tedy ohlásí neúspěch, pouze když řešení neexistuje.

Připomeňme teď, že počet katapultů označujeme K a délku lajny N.

Nejprve si všechny katapulty seřadíme podle levé hranice vzestupně. Tím pádem se můžeme postupně posouvat v poli katapultů a nalezení katapultů, které máme přidat do množiny, trvá lineárně v jejich počtu, za celý algoritmus tedy $\mathcal{O}(K)$. Samotné řazení nám zabere $O(K \log K)$.

K udržování katapultů, resp. k jejich přidávání a nalezení toho s minimální pravou hranicí, nám dobře poslouží halda. Tak bude každá operace trvat $\mathcal{O}(\log K)$.

Nejprve tedy seřadíme katapulty a pak projdeme celou bojovou lajnu. Tento průchod nám trvá $\mathcal{O}(N)$, na každém políčku se ptáme na katapult s minimální souřadnicí a možná přidáváme katapulty do množiny. Takových přidání je ale dohromady $\mathcal{O}(K)$, takže celková časová složitost algoritmu je $\mathcal{O}(N \log K)$. Přesněji

tedy $(K \log K + N \log K)$, ale klidně předpokládejme $K \leq N$. Paměťová složitost je lineární v počtu katapultů, tedy $O(K)$.

Pěkné je si všimnout, že políčka, na kterých se nic neděje, jsou nezajímavá a vlastně nás jen zdržují. Líbilo by se nám umět skákat jen po těch, kde se něco děje. To ale umíme zařídit – o každém katapultu víme, kde jeho úsek začíná a končí, takže se můžeme buď posunout o jedno políčko (máme-li po umístění katapultu neprázdnou množinu), nebo rovnou skočit na příští událost.

KSP

Události si můžeme udržovat v haldě, na každém políčku tedy ještě přidáme případné zjištění příští události. Jelikož pro umístění K katapultů použijeme nejvýš K políček, bude celková složitost $\mathcal{O}(K \log K)$.

Teď těžší varianta, a pozor, přijde trik :) Všimneme si, že souřadnice katapultů jsou nezávislé – bez ohledu na to, do kterého řádku katapult umístíme, můžeme ho umístit do stejných sloupců.

Řešením úlohy tak není nic jiného než dvojí spuštění lehčí varianty, jednou pro řádky, jednou pro sloupce. A jelikož dvojka je konstanta, časová složitost zůstává $\mathcal{O}(K \log K)$.

řešení

Program (C):

`http://ksp.mff.cuni.cz/viz/28-3-4.c`

Program (C++):

`http://ksp.mff.cuni.cz/viz/28-3-4.cpp`

Karry Burešová

28-3-5 Závaží z fošen

Nejprve fošny ze zadání ohoblujeme až na strohou geometrickou realitu: Dostali jsme n desek, každá má danou šířku a výšku, všechny mají jednotkovou tloušťku. Chceme vytvořit kvádr. Uděláme to tak, že si vybereme podmnožinu desek, položíme je na sebe a případně některé z nich o 90 ° otočíme. Nakonec je ořízneme na šířku nejužší desky a výšku té nejnižší. Chceme přitom, aby vznikl kvádr o největším možném objemu.

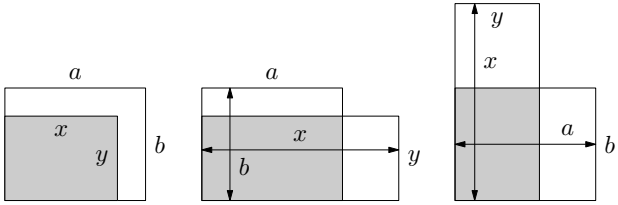
Jak se zbavit otáčení

Úloha má nepříjemně mnoho stupňů volnosti: nejen, že si vybíráme, které desky použijeme, ale pro každou ještě určujeme natočení. Pojďme se otáčení zbavit. Dokážeme, že pokud každou desku předem natočíme „naležato“, tedy tak, aby její šířka byla větší nebo rovná výšce, o optimální řešení nepřijdeme.

Nejprve uvažujme situaci se dvěma deskami rozměrů $a \times b$ a $x \times y$ (první rozměr je šířka, druhý výška). Jistě můžeme předpokládat, že $a \geq b$ a $x \geq y$ (jinak desku otočíme) a navíc $b \geq y$ (první deska je ta vyšší). Aby vznikl kvádr maximálního objemu, chceme desky posunout a otočit tak, aby se překrývaly

v co největší ploše. Překryv jistě nezmenšíme, přiložíme-li na sebe levé dolní rohy obou desek.

Nyní rozlišíme dva případy. V prvním je $x \leq a$, čili nižší deska je i užší (levý obrázek). Tehdy se desky překrývají celou plochou té menší z nich, takže se jistě nevyplatí kteroukoliv desku otáčet.



Zbývá nám případ $x > a$. Pokud obě desky ponecháme orientované stejně (prostřední obrázek), jejich společná část bude mít obsah $a \cdot y$. Pakliže je vůči sobě otočíme (pravý obrázek), vyjde obsah $b \cdot y$. První varianta je ovšem stejná nebo lepší, neboť $a \geq b$.

V obou případech platí, že překrývající se část má opět šířku větší nebo rovnou výšce, takže postup můžeme opakovat a o všech použitých deskách dokázat, že jsme je mohli nechat naležato. Ve zbytku řešení tedy budeme bezelstně předpokládat, že deskami není možné otáčet.

Elementární řešení

Náš první pokus o algoritmus bude založený na jednoduchém pozorování: výsledný kvádr zdědí jak svou šířku, tak svou výšku od některé desky, každou možná od jiné. Stačí tedy vyzkoušet n možných šířek, k nim n možných výšek a pokaždé probrat všechny desky a použít ty, které jsou dostatečně široké a vysoké. To zvládneme v čase $\mathcal{O}(n^3)$ a získáme za to pár bodů.

Mimochodem, řešení tohoto druhu by fungovalo, i kdybychom brali v úvahu otáčení. Výšku i šířku kvádru bychom vybírali ze všech výšek i šířek (celkem $4n^2$ možností) a při testování, zda se deska vejde, bychom zkoušeli obě možné orientace.

Chytřejší řešení

V minulém řešení počítáme stále dokola podobné věci, takže zkusíme výpočet přeuspořádat, abychom se tomu vyhnuli.

Nadále budeme zkoušet všech n možných šířek. Pro každou šířku w vybereme všechny dostatečně široké desky. Budeme je procházet od nejvyšší k nejnižší a průběžně přepočítávat aktuální kvádr. Pro nejvyšší desku samotnou má kvádr šířku w, výšku této desky a tloušťku 1. Když přidáme další, nižší desku, šířka zůstane, výška klesne a tloušťka vzroste o 1. Tak pokračujeme a pokaždé spočítáme objem aktuálního kvádru a započítáme ho do průběžného maxima.

Toto vše se dá stihnout v čase $\mathcal{O}(n^2)$: na počátku výpočtu setřídíme všechny desky podle výšky. Pak zkoušíme (v libovolném pořadí) všech n možných šířek, pro každou z nich probíráme desky v setříděném pořadí, přeskakujeme ty příliš úzké a pro ostatní v konstantním čase přepočítáváme objem kvádru.

Autor přiznává, že stále věří v existenci rychlejšího řešení, ale postupně si všechna taková vyvrátil. Kdybyste o nějakém věděli, dejte nám prosím vědět.

Program (C):

`http://ksp.mff.cuni.cz/viz/28-3-5.c`

KSP

Martin „Medvěd“ Mareš

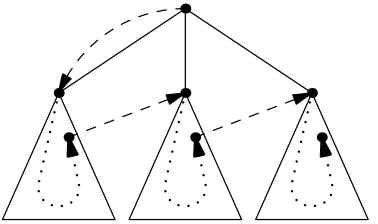
28-3-6 Počítání přeživších

Aby řešení mohlo existovat, graf musí být souvislý. To budeme nadále předpokládat. Nejprve vyřešíme jednodušší variantu, kdy graf přátelství je stromem. Hledanému předpisu, kdo komu má helmici předat, budeme říkat *plán* pro daný strom. Ten si lze představit prostě jako posloupnost vrcholů, kde dva sousední jsou vždy vzdálené ve stromě nejvýše tři hrany.

řešení

Strom si zakořeníme a použijeme obvyklý trik na stromové úlohy. Nejprve rekurzivně najdeme plán pro každý podstrom kořene a pak tyto plány nějak napojíme, abychom z nich poskládali plán pro celý strom. Předpokládejme pro začátek (později se ukáže, že je to trochu složitější), že každým podstromem začne helmice putovat od kořene.

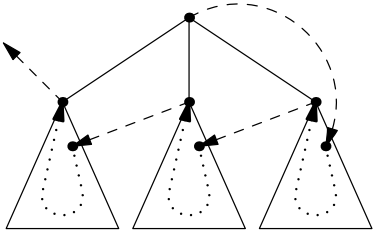
To znamená, že když helmice opouští první podstrom, musíme ji předat rovnou kořeni druhého podstromu (přes kořen celého stromu to nejde, tam už byla):



Tečkované čáry značí rekurzivně spočtené plány pro jednotlivé podstromy, čárkované jejich napojení. Aby předání mezi podstromy bylo korektní, musí plán pro každý podstrom končit v prvním patře tohoto podstromu (hned pod jeho kořenem). Kdyby končil hlouběji, předání bude přes víc než tři hrany.

Abychom mohli naše řešení použít rekurzivně, musí tedy i výsledný plán pro celý strom končit v prvním patře. Ale plán nastíněný v obrázku výše zřejmě končí v patře druhém...

Leč není třeba propadat panice. Všimneme si, že kdybychom z kořene poslali helmici do vrcholu, který je aktuálně poslední, a celý průchod průchod podstromy obrátili (díky neorientovanosti hran můžeme), skončíme v kýženém prvním patře:



Máme tedy algoritmus, který nalezne plán pro daný strom začínající v kořeni a končící v prvním patře: Rekurzivním zavoláním téhož algoritmu nalezneme plán pro první podstrom kořene a obrátíme v něm pořadí vrcholů (jako první helmici obdrží nějaký vrchol v prvním patře tohoto podstromu, jako poslední jeho kořen). Za tento převrácený plán připojíme rovněž převrácené plány pro druhý, … až poslední podstrom. Helmice tedy skončí v kořeni posledního podstromu, tedy v prvním patře celého stromu, což přesně potřebujeme.

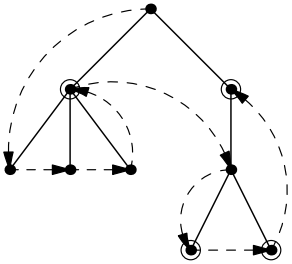
Pro zjednodušení implementace nemusíme plány ani dodatečně obracet. Stačí rekurzivnímu volání navíc předat jeden parametr, zda má generovat normální (z kořene do prvního patra), nebo obrácený (z prvního patra do kořene) plán. Díky tomu si nemusíme plány ukládat, nýbrž můžeme vrcholy rovnou vypisovat.

Snadno si rozmyslíte, že sousední vrcholy plánu jsou vzdálené maximálně tři hrany. Zbývá ještě vyřešit okrajové podmínky rekurze. Zastavíme se na jednovrcholovém stromě (tvořeném listem původního stromu), pro který je plánem prostě jednoprvková posloupnost obsahující tento vrchol.

Celý algoritmus v pseudokódu:

NajdiPlán(u, *obrať*):

1. Pokud *obrať* $= 0$: vypiš u.
2. Pro všechny v syny u:
3. NajdiPlán(v, $1 -$ *obrať*)
4. Pokud *obrať* $= 1$: vypiš u.



Rozmyslete si, že opravdu odpovídá výše popsanému. Výsledný plán může vypadat třeba takto:

Zakroužkované jsou ty vrcholy, ve kterých hledáme obrácený plán. To jsou právě vrcholy v lichých patrech.

Obecné grafy

Zobecnit řešení na všechny grafy už je jednoduché. Z libovolného grafu můžeme udělat strom tak, že najdeme jeho kostru (např. prohledáním do hloubky) a

na ni spustíme výše popsaný algoritmus. Protože pro každý strom existuje řešení, hrany mimo kostru jsou přebytečné a můžeme je ignorovat.

Navíc hledání kostry a konstrukci plánu nemusíme provádět odděleně, nýbrž je můžeme spojit do jednoho DFS průchodu grafem. Podrobněji ve vzorovém programu.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-3-6-dfs.py`

KSP

Filip Štědronský

28-3-7 Legie v lese

Les je obdélník o hraně L, v němž leží S bodových stromů. Horní stranou do obdélníku vstoupí kruhová legie o průměru D a chce vylézt spodní stranou. Nesmí přitom narazit na žádný strom, ani na levou či pravou stranu obdélníku. Střed legie tedy musí stále udržovat vzdálenost alespoň $D/2$ od všech stromů i od bočních stran lesa.

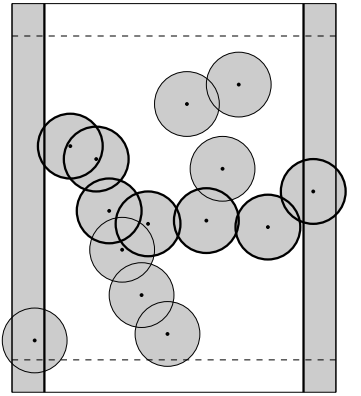
řešení

Překážky a ploty

Navigační problémy tohoto druhu se řeší snáz, pokud se lesem místo kruhové formace pohybuje jediný bod. Úlohu proto trochu přeformulujeme: každý strom „nafoukneme“ na kruh o poloměru $D/2$ a boční strany lesa roztáhneme směrem dovnitř na obdélníky široké rovněž $D/2$. Aby se všechny překážky vešly do lesa, roztáhneme les nahoře i dole o $D/2$.

Legii naopak smrskneme do jediného bodu. Ten může stát právě na těch místech, která neleží v žádném kruhu ani obdélníku.

Dostaneme tedy nějaké překážky ve tvaru kruhů a obdélníků a ptáme se, zda bod může projít shora dolů a vyhnout se všem překážkám.

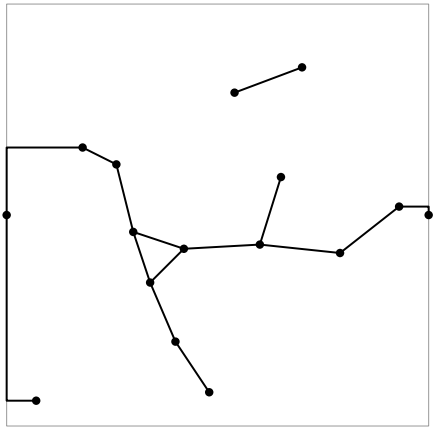


Na předchozím obrázku to možné není, protože v něm existuje *plot* – posloupnost na sebe navazujících překážek, která spojuje levý okraj s pravým. Každá trasa shora dolů musí plot alespoň na jednom místě protnout, takže trasa narazí na alespoň jednu překážku.

Dokonce platí, že kdykoliv nejde projít shora dolů, existuje nějaký plot, který tomu brání. Vskutku: množina bodů, do kterých se dá shora dojít, tvoří nějakou uzavřenou oblast. Hranice této oblasti se skládá z částí horní, levé a pravé strany lesa a nějakých částí překážek. Představme si, že hranici této oblasti obcházíme od levého horního rohu lesa proti směru hodinových ručiček. Mezi posledním odchodem z levé strany a následujícím příchodem na tu pravou jsme museli projít přes na sebe navazující části překážek. To ovšem znamená, že tyto překážky tvoří plot.

Gigantický graf

Potřebujeme tedy vymyslet, jak hledat plot. Pomůžeme si grafem: jeho vrcholy budou reprezentovat stromy, navíc přidáme vrchol ℓ pro levou stranu lesa a p pro pravou. Hranou spojíme každé dva vrcholy, jejichž vzdálenost v rovině je menší než D. To nastane právě tehdy, když se příslušné překážky protínají. Plot pak odpovídá cestě z vrcholu ℓ do vrcholu p v tomto grafu. Pro příklad z předchozího obrázku vypadá graf následovně:



Stačí tedy zjistit, zda ℓ a p leží v téže komponentě souvislosti. To můžeme otestovat třeba prohledáváním do šířky. Jistou nevýhodou ovšem je, že náš graf může mít až $\mathcal{O}(S^2)$ hran (rozmyslete si, jak by takový les vypadal). Bude tedy lepší nedržet si celý graf v paměti a vytvářet ho podle potřeby za běhu.

Budeme si udržovat frontu stromů, které máme zpracovat. Na počátku do ní vložíme stromy dosažitelné z ℓ, čili ty, jejichž x-ová souřadnice je menší než D. Pak postupně odebíráme stromy z fronty. Pro každý strom najdeme všechny jeho

sousedy, tedy stromy ve vzdálenosti menší než D, a pokud jsme je ještě neviděli, přidáme je do fronty. Skončíme, pokud se fronta vyprázdní, nebo pokud objevíme strom s x-ovou souřadnicí větší než $L - D$.

To je všechno. Snadné, že? Ale pomalé... Navštívíme až S vrcholů a pro každý z nich procházíme S potenciálních sousedů. Jednoho souseda naštěstí stihneme zpracovat v konstantním čase (místo vzdáleností porovnáváme jejich druhé mocniny, abychom nemuseli odmocňovat), takže celý výpočet trvá $\mathcal{O}(S^2)$.

KSP

Program (C):

`http://ksp.mff.cuni.cz/viz/28-3-7-quad.c`

Pomohou políčka?

Existuje řada technik, kterými se dá hledání sousedních stromů urychlit. Můžeme například rozdělit les na políčka velikosti $D \times D$ a pro každé políčko si předpočítat, jaké stromy v něm leží. Sousedé daného stromu se pak nacházejí buď ve stejném políčku, nebo v některém z osmi okolních.

řešení

Tento trik může pro některé kombinace L a D výrazně pomoci. Limity v zadání úlohy bohužel připouštěly i případy, kdy je D řádově stejné jako L. Tehdy je políček konstantní počet, takže okolní políčka obsahují podstatnou část všech stromů. Škoda, tudy cesta nevede.

Znovu zametání

Není náhoda, že jsme k této sérii přibalili geometrickou kuchařku, v níž se ukazuje princip zametání roviny. Hodí se i pro tuto úlohu.

Les budeme zametat zleva doprava svislou přímkou. Budeme si při tom pamatovat, které stromy jsme už potkali. Pokaždé, když zametací přímka narazí na nový strom, najdeme jeho sousedy mezi předchozími stromy a natáhneme do nich hrany. Hledání si urychlíme dvěma triky:

- Nemá smysl zkoušet stromy, které se v y-ové souřadnici liší od nového stromu o více než D.
- Pokud se dva stromy vyskytly v témže řádku, stačí uvažovat pravý z nich. Ten levý je buďto moc daleko, nebo už leží v téže komponentě jako pravý, takže stačí hranu přivést do pravého. (Nezapomeňte, že souřadnice stromů jsou celočíselné, a proto je možných řádků málo.)

Pořídíme si tedy pole indexované číslem řádku $(0 \ldots L-1)$. V něm si budeme pro každý řádek pamatovat, jaký nejpravější strom jsme tam viděli. Objeví-li se nový strom v y-tém řádku, stačí se podívat na nejpravější stromy v řádcích $y-D$ až $y+D$ a pro každý z nich zkontrolovat, jak je daleko.

Sousedy jednoho vrcholu tedy projdeme v čase $\mathcal{O}(D)$. Celý algoritmus nejprve stráví čas $\mathcal{O}(S \log S)$ setříděním všech stromů a $\mathcal{O}(L)$ inicializací pomocného pole. Poté prochází S stromů a nad každým z nich stráví čas $\mathcal{O}(D)$. Tím vytvoří graf o nejvýše $\mathcal{O}(SD)$ hranách, který pak prohledá do šířky v čase $\mathcal{O}(SD)$.

Celková časová složitost je tedy $\mathcal{O}(S \log S + L + SD)$.

Implementační (in)triky

Náš ukázkový program se drží předchozího nápadu, ale pro jednoduchost se v některých detailech odchyluje.

Především si místo třídění stromů podle souřadnic vytvoří matici $L \times L$, která o jednotlivých bodech lesa říká, zda tam leží strom. Zametání lesa pak prostě prochází tuto matici po sloupcích a hledá jedničky. Místo $\mathcal{O}(S \log S)$ tedy potřebuje čas $\mathcal{O}(L^2)$. To pro limity ze zadání je spíše lepší.

Také místo toho, abychom graf udržovali v paměti a pak ho celý prohledali, udržujeme komponenty souvislosti průběžně v datové struktuře Union-Find. Pro každou z $\mathcal{O}(SD)$ hran tedy voláme Union, který trvá $\mathcal{O}(\log S)$. Kdybychom byli pilnější, použili jsme rychlejší Union-Find se složitostí $\mathcal{O}(\log^* S)$[63] nebo $\mathcal{O}(\alpha(S))$, který je popsaný v kuchařce o minimálních kostrách.[64]

I s jednoduchým Union-Findem má náš program složitost $\mathcal{O}(L^2 + SD \log S)$, což je pro slíbené velikosti vstupu naprosto postačující.

Program (C):

`http://ksp.mff.cuni.cz/viz/28-3-7-grid.c`

Divotvorný diagram

Zbývá hrstka pochybností, zda by úlohu šlo řešit efektivněji, zejména v případě, kdy by nám nikdo neslíbil celočíselné souřadnice. Pochybnosti samozřejmě nejsou na místě, známe řešení v čase $\mathcal{O}(S \log S)$ s použitím jednoho milého králíka z kouzelnického klobouku výpočetních geometrů. Jen je to trochu pracnější, takže vše pouze načrtneme. Detaily můžete najít v geometrické kapitole medvědí knížky.[65]

Nechť $\mathcal{S}$ je množina stromů v rovině. Pro každý strom $s \in S$ definujeme jeho *oblast* R_s jako množinu všech bodů roviny, pro něž je s nejbližší strom (respektive jeden z nejbližších stromů, pokud jich je víc).

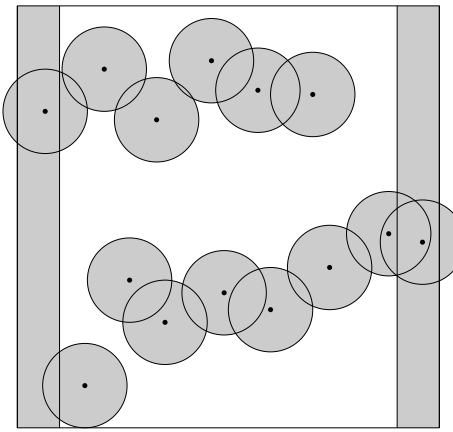
Podívejme se na nějaké dva stromy x a y. Všechny body roviny, které mají k x blíže než k y, tvoří polorovinu (hraniční přímkou této poloroviny je osa úsečky xy). Proto musí každá oblast R_s být průnikem konečně mnoha polorovin. Oblasti tedy mají tvar konvexních mnohoúhelníků, případně na jedné straně otevřených do nekonečna.

Vyzkoušíme si to na tomto lese:

[63] Funkce $\log^* x$ říká, kolikrát musíme číslo x opakovaně zlogaritmovat, než se poprvé dostaneme pod 1. Roste tedy velice pomalu. Funkci $\alpha(x)$ zde definovat nebudeme, ale prozradíme, že roste ještě mnohem pomaleji.

[64] `http://ksp.mff.cuni.cz/viz/kucharky/kostry`

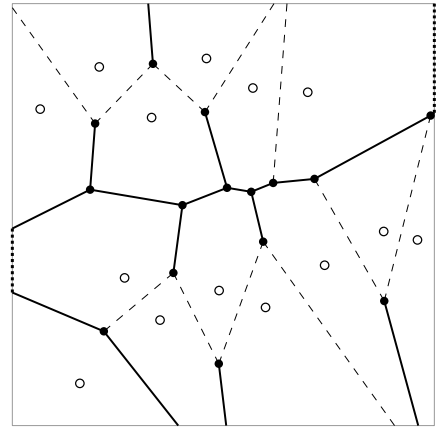
[65] `http://mj.ucw.cz/vyuka/ads/43-geom.pdf`



KSP

řešení

Diagram pro něj vyjde následovně. (Plné i čárkované čáry jsou hranice oblastí, brzy uvidíme, v čem se liší. Obrázek jsme ořízli okrajem lesa.)



Rozkladu roviny na tyto oblasti se říká *Voroného diagram* (zkoumal ho začátkem 20. století ruský matematik Georgij Voronoj). Je dobré vědět, že je to rovinný graf (až na neomezené stěny, ale to je detail), tudíž pro S stromů má $\mathcal{O}(S)$ vrcholů a $\mathcal{O}(S)$ hran. Také se hodí znát algoritmus pro jeho konstrukci v čase $\mathcal{O}(S \log S)$, jako obvykle mazaným zametením roviny – detaily viz medvědí knížka.

Na chvíli zapomeňme, že les má nějaké okraje. Pak platí, že může-li lesem projít kruhová legie, pak jím může projít po hranách Voroného diagramu. Není divu: každá hrana diagramu je osa úsečky mezi dvěma stromy (nebo její část),

takže jdeme-li po hraně, dodržujeme maximální možnou vzdálenost od této dvojice stromů.

Stačilo by tedy sestrojit diagram, pro každou hranu spočítat vzdálenost stromů, které tato hrana odděluje, a pokud je to méně než D, hranu smazat (na obrázku jsou tyto hrany čárkované). Tím by vznikl nějaký lineárně velký graf, který bychom vzápětí prohledali do šířky. To dává hezké řešení v čase $\mathcal{O}(S \log S)$.

Jenže teď si na okraje zase musíme vzpomenout. Co to způsobí? Část diagramu skončí za okrajem, takže ji odřízneme. Naopak máme povoleno chodit podél okraje (v uctivé vzdálenosti $D/2$). Proto se podíváme na konce hran, které se zarazily o okraj, a zkusíme je propojit. Každá po sobě jdoucí dvojice takových hran ohraničuje společnou stěnu diagramu, uvnitř níž leží nějaký strom. Pokud je tento strom vzdálený od okraje aspoň D, můžeme mezi hranami projít podél okraje. To znázorníme dalším typem hran (na obrázku tečkované). Bude jich nejvýše $\mathcal{O}(S)$ a spočítáme je také v čase $\mathcal{O}(S)$.

Shrneme-li tyto úvahy (a doplníme spoustu detailů, o které jsme vás ošidili), získáme algoritmus řešící úlohu v čase $\mathcal{O}(S \log S)$ bez jakýchkoliv požadavků na rozsah souřadnic a celočíselnost.

Martin „Medvěd“ Mareš

28-3-8 Inteligence hejna

Měli jsme za úkol hledat co nejkratší kružnici, která prochází všemi městy České republiky, přičemž text seriálu napovídal, že by se mohl použít mravenčí algoritmus. Tím to také zkusíme, ale nebude to úplně jednoduché. Úlohu budeme řešit jen pro všechna města. Pro jejich menší část by řešení bylo obdobné.

Část z vás se spletla a namísto kružnice hledala cestu. To ale nebyl velký problém. Myšlenka v následujících řešeních je v zásadě stejná, jen se na konci musíme vrátit do počátečního vrcholu.

Základní mravenčí algoritmus

První problém, na který můžeme narazit, je, že výpočet trvá hrozně dlouho. Průchod jednoho mravence přes všechna města může zabrat až vteřinu. To nás motivuje použít málo mravenců, protože chceme algoritmus nechat běžet více iterací a dozvědět se aspoň nějaký výsledek.

Abychom ušetřili aspoň trochu času, předpočítáme si matici vzdáleností měst, abychom je nemuseli počítat stále dokola.

Další problém může nastat s hodnotami vhodnosti hran b_{ij} a intenzitou feromonů f_{ij}. V textu seriálu se doporučuje dát $b_{ij} = 1/d_{ij}$, kde d_ij je délka hrany, a k f_{ij} přičítat hodnoty $1/L$, kde L je délka nalezené cesty.

V praxi je dobré se snažit obě tyto hodnoty držet řádově stejné a při tom respektovat používané datové typy. Při spuštění algoritmu můžeme zjistit, že vzdálenosti měst se pohybují v rozmezí $[501; 485\,824]$, zatímco délky okružních

jízd dosahují řádu 10 000 000. Pokud tedy použijeme $b_{ij} = 1\,000/d_{ij}$ a k f_{ij} budeme přičítat $10^7/L$ dostaneme se vždy zhruba do řádu jednotek.

Takto upraveným základním algoritmem spolu s parametry $\alpha = 2$, $\beta = 2$ a vaporizací $\rho = 0{,}5$ jsem za několik desítek iterací a zhruba s jednotkami mravenců dosáhl řešení 23 495 526.

Vícekrát jsem to nespouštěl a ladit parametry se mi nechtělo, protože to běželo dost pomalu.

Hladový algoritmus

Teď chvíli zapomeneme na mravence a zkusíme úlohu vyřešit hladově. Začneme v náhodném městě a vždy se přesuneme do nejbližšího nenavštíveného města. Takto pokračujeme až dokud všechny neprojdeme. Kvalita řešení se může lišit v závislosti na výběru počátečního města. Tak se také vyplatí algoritmus spustit vícekrát.

Vašek Volhejn vyzkoušel pro hladový algoritmus všechny možné začátky a jako nejlepší řešení mu vyšlo 20 763 908.

Hladoví mravenci

A co takhle oba postupy zkombinovat? Nabízí se například vzít několik hladových řešení a vzít je jako počáteční populaci. Výpočet je ale stále pomalý a je jen malá šance, že tak hladové řešení překonáme. Hlavně bychom si museli trochu déle počkat.

Co uděláme my, je, že znovu použijeme mravenčí algoritmus, ale omezíme množství měst, ze kterých budeme vybírat následníka. Jelikož se pohybujeme v rovině, tak se nám takřka nikdy nevyplatí jít moc daleko. Další město můžeme tedy vybírat například z nejbližších K, kde K bude rozumně malé. Třeba 10.

Tím se algoritmus dost zrychlí. Už si můžeme dovolit použít více mravenců. Použijeme jich aspoň 100 až 1 000, aby v jedné iteraci zvládli označit rozumné množství hran feromony a další iterace tak měly větší variabilitu.

Pokud takové řešení s parametry $\alpha = 2$, $\beta = 2$, $\rho = 0.5$ a pro 1 000 mravenců necháme běžet 500 iterací (několik hodin), dostaneme se na řešení o hodnotě 20 550 678, což není o moc lepší než hladový algoritmus, ale aspoň něco. Tento algoritmus můžete vidět ve zdrojovém kódu.

Já pak ještě experimentoval s tím, že feromony zanechává jen W nejlepších mravenců z jedné iterace, ale dostal jsem se tak jen na víceméně stejně dobrý výsledek.

Program (C++):

`http://ksp.mff.cuni.cz/viz/28-3-8.cpp`

Další možné postupy

Evoluční a jim podobné algoritmy jsou většinou navržené obecně, aby se jimi dala řešit širší škála problémů. Proto nevyužívají charakteristiky žádného konkrétního z nich. Na jednu stranu je výhoda, že se dají prakticky vždy nějakým

způsobem použít. Na druhou stranu ale mají nevýhodu v tom, že jsou málokdy nejlepší. Když se totiž zaměříme na konkrétní problém a navrhneme algoritmus přímo vhodný pro něj, tak obvykle dosáhneme vyššího úspěchu.

Kromě toho existují ještě heuristiky, které se snaží vylepšovat nějaké již existující řešení. Pro takové heuristiky se hodí jako počáteční řešení zvolit buď výsledek nějakého hladového algoritmu nebo právě výsledek některého z evolučních algoritmů.

Pro náš problém se dají použít například následující heuristiky:

- Prohodíme 2 hrany. Pokud se řešení zlepší, necháme je tak, jinak je vrátíme zpátky. (Díky této heuristice se postupně zbavujeme i křížení hran.)
- Prohodíme 3 hrany. Pakliže už nefunguje prohazování dvou hran, můžeme začít prohazovat 3, což nám dává další možnosti.
- Přeházení K po sobě jdoucích měst. Také může pomoci.

U všech těchto heuristik nové řešení přijmeme pouze pokud nám heuristika pomohla. Tyto heuristiky uvádíme jen pro příklad. Ty nijak nesouvisí s evolučními algoritmy. Pomocí nich se řešení dalo vylepšit až na 18 000 000.

Karel Tesař

Čtvrtá série

28-4-1 Sledování telefonů

Pokrytí hovory budeme říkat libovolnému seznamu hovorů, který při posčítání přes spoje odpovídá vstupním datům. *Minimální pokrytí* pak bude takové, které mezi všemi pokrytími má minimální počet hovorů. Úloha po nás chce najít libovolné minimální pokrytí. Pro zjednodušení uvažujme, že před prvním a za posledním domem je imaginární spoj, přes který neproběhl žádný hovor.

Podívejme se teď na libovolné pokrytí, a vezměme nějaký dům. Z něj vede doleva l a doprava p hovorů. Pokud $l = p$, tímto domem můžou všechny hovory jenom procházet. Pokud ale $l < p$, pak doprava vede více hovorů, tedy zde alespoň $p - l$ musí začínat (vést od tohoto domu někam doprava). Naopak, pokud $l > p$, pak zde alespoň $l - p$ hovorů musí končit.

Na chvilku si představme, že počty hovorů přes spoj jsou nadmořské výšky. Pokud začneme vlevo v nule, půjdeme stejně metrů do kopce jako z kopce, protože na konci zase skončíme v nule. Z toho vidíme, že začátků je stejně jako konců. Navíc, budeme-li zleva doprava průběžně počítat počet začátků a konců, počet začátků bude v každou chvíli alespoň tak velký jako počet konců.

V libovolném pokrytí musí každý hovor někde začínat. Posčítáme-li tedy nutné začátky, dostaneme odhad na minimální počet hovorů H. Pokud by se nám podařilo sestrojit pokrytí, které je složeno z H hovorů, víme, že je minimální.

Nyní si ukážeme algoritmus, který právě takové pokrytí sestrojí. Půjdeme přes domy zleva doprava, a každý začátek si poznamenáme jako nevyřešený. Jakmile narazíme na nějaký konec, přidělíme ho libovolnému z nevyřešených začátků.

Pokud za „libovolný“ prohlásíme ten první, můžeme nevyřešené začátky udržovat ve frontě, ale stejně dobře bude fungovat třeba zásobník. Jak už jsme si všimli, nevyřešených začátků bude vždy dostatek a na konci vyřešíme všechny. Tím jsme sestrojili pokrytí právě H hovory, tedy to musí být minimální pokrytí.

Řešení projde přes N domů, a k tomu H-krát vloží číslo na zásobník. Časová složitost tedy bude $\mathcal{O}(N + H)$, stejně tak paměťová.

Někteří z vás si v tuto chvíli řekli, že to je nejlepší možné, protože $\mathcal{O}(N)$ nám sebere čtení vstupu, a $\mathcal{O}(H)$ vypisování výstupu. V tom případě jste si ale špatně zvolili formát výstupu, který byl na vás :)

My si jako formát výstupu zvolíme seznam trojic (A, B, x). Každá říká, že z A do B bylo provedeno x hovorů. Součet všech x bude H, ale ukážeme, že námi vygenerované pokrytí lze popsat pomocí nejvýše $2N$ trojic.

Nevyřešené začátky si budeme ukládat do fronty, ale místo toho, abychom si do ní vkládali každý zvlášť, uložíme si tam dvojice (A, a) vyjadřující, že z A máme ještě a nevyřešených začátků.

Jak potom postupujeme, když nalezneme vrchol Z, kde končí nějaké hovory? V každém takovém vrcholu je $l-p=z$ konců, které je potřeba propojit se začátky. Tyto začátky poskládáme z toho, co najdeme na začátku fronty.

Podíváme se na první (A, a) ve frontě. Pokud $a \leq z$, všechny nevyřešené začátky z A spojíme s Z, vypíšeme (A, Z, a) a odstraníme z fronty (A, a). Navíc snížíme z o a. Toto opakujeme, dokud neodstraníme všechny začátky, které dokážeme vyřešit celé.

Až poslední začátek, kdy $a > z$, nedokážeme vyřešit natolik, abychom ho mohli vyhodit z fronty. V tu chvíli a ve frontě snížíme o z, a naposledy vypíšeme (A, Z, z). Tím jsme vyřešili všechny konce v domě Z.

Všimněte si, že částečných snížení ve frontě bude nejvýše tolik, kolik je vrcholů s konci, tedy nejvýše N. Zároveň, odstranit z fronty konkrétní dům lze jen jednou, a do fronty jsme vložili každý dům nejvýše jednou. Dohromady tedy vypíšeme nejvýše $2N$ řádků. A protože jsme si tím zároveň omezili velikost fronty na N, takto upravený algoritmus už má časovou i paměťovou složitost $\mathcal{O}(N)$.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-4-1.py`

Ondra Hlavatý

28-4-2 Podepisování dokumentů

Na začátek zvolme jednoduchý přístup. Vybereme si dvojici a dáme jí dokumenty. Potom se podíváme na to, jak dlouho by podepisování trvalo. Takto to zkusíme pro každou spolusedící dvojici a vybereme z nich tu nejlepší.

A jak zjišťovat celkovou dobu podepisování dokumentu pro danou startovní dvojici? Nejjednodušší je krokovat po minutě. Rychlejší přístup rovnou skáče na ty minuty, kdy se nějaký člověk dokončí svůj podpis.

Mějme tedy dvě čísla (každé pro jednu sadu dokumentů). Tato čísla nám udávají, ve které minutě přibude další podpis pod danou sadu dokumentů. Pokaždé vybereme menší z těchto časů, v takovém čase se sada posune k dalšímu člověku. Dokument tedy pošleme dalšímu člověku a čas příštího podpisu aktualizujeme (přičteme dobu podepisování tímto člověkem). Toto opakujeme, dokud se nepodepíší všichni.

Jeden průběh dokumentu jsme schopni zpracovat v lineárním čase a musíme vyzkoušet N dvojic. Dostáváme tedy časovou složitost $\mathcal{O}(N^2)$.

Zrychlujeme

Pojďme se podívat na rychlejší řešení. Budeme využívat již spočítané případy, čímž si ušetříme jejich opětovné počítání (této technice se říká dynamické programování).[66]

[66] `http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani`

Nejprve si, stejně jako v předchozím případě, vybereme libovolnou dvojici a pro ni si algoritmem z předchozí části spočítáme celkovou dobu podepisování. Navíc si ale zapamatujeme, kde dokumenty skončí a kdy přibyl poslední podpis pod první sadu a druhou sadu.

Nyní si představme, že bychom dokumenty dali o jednu pozici vedle tak, že první člověk, co podepisoval druhou sadu, bude nyní první, který bude podepisovat první sadu. Ukážeme si, že tato situace se v určitém smyslu příliš neliší od té předchozí, kterou již máme spočtenou.

Podívejme se na skupinu lidí, která podepíše druhou sadu. Tato skupina oproti předchozímu řešení přišla o svého prvního člena, který nyní podepíše první sadu. Budeme-li tedy vycházet z předchozích výsledků, stačí od času posledního podpisu pod druhou sadou odečíst dobu, kterou tento první člen strávil podepisováním.

To ale nemusí být jediná změna. Do skupiny lidí podepisujících druhou sadu může několik lidí přibýt. To se může stát tím, že druhá sada se nyní k poslednímu člověku dostane dřív a tedy se může stihnout předat ještě následujícím lidem v kruhu. Zkoušíme tedy postupně takové lidi přesouvat do naší skupiny a příslušně upravovat časy.

Kdy se zastavit? Všimněte si, že tímto přesouváním se čas podepisování druhé sady zvyšuje, zatímco u první sady se snižuje. Rozdíl těchto časů se tedy bude postupně snižovat dokud čas podepisování druhé sady nebude větší než čas podepisování té první, poté se bude tento rozdíl jen zvyšovat. Stačí se tedy zastavit v okamžiku, kdy doba podepisování druhé sady překročí dobu podepisování první sady.

Rozmyslete si, že jedno ze dvou posledních řešení (tj. to první, kdy doba podepisování druhé sady je větší než doba podepisování první sady, nebo to poslední, kdy tomu tak ještě není) je nejrychlejší řešení, a popisuje tedy i skutečnou dobu, kterou by dokumenty kolovaly, pokud bychom je rozdali zvolené startovní dvojici.

Takto budeme postupně zkoušet všechny startovní dvojice. Stačí z nich pak vybrat to celkově nejrychlejší a to nám říká, které dvojici máme dokumenty dát.

V celém řešení vlastně postupně posouváme místo, kde dokumenty rozdáme a místo kde se sady dokumentů opět střetnou. Všimněte si, že místo střetnutí nikdy nepřekročí místo rozdání dokumentů. A jelikož zkoušíme každé výchozí místo jen jednou, tak místo střetnutí udělá nanejvýš jeden okruh. Čas trávíme pouze při posouvání některých z těchto míst (a hledání pro první dvojici, které zvládneme lineárně) a těchto posunů je lineárně, celková časová složitost tedy bude $\mathcal{O}(N)$.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-4-2.py`

Janka Bátoryová & Dominik Smrž

28-4-3 Řazení životních hodnot

Úkolem v této úloze bylo seřadit životní hodnoty reprezentované čísly tak, aby zapadly do zadaných relací „menší než“ a „větší než“. Díky tomu, že číselné hodnoty byly různé, nebyla úloha příliš složitá na vyřešení a mnoho z vás se s ní úspěšně popralo.

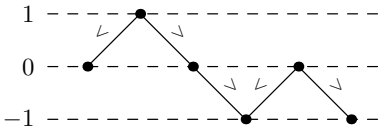
Pokud je mezi třemi pozicemi relace $a > b < c$, víme, že na pozici b nemůžeme umístit největší číslo ze vstupu, protože potřebujeme alespoň dvě větší čísla pro umístění na pozice a a c. Naopak pro pozice a a c žádné takové omezení nemáme a je nám dokonce jedno, v jakém vztahu budou čísla na těchto dvou místech – stačí, že obě budou větší, než číslo na b.

Po zvolení čísla na pozici b se nám dokonce na tomto místě rozpadnou zbylé nerovnosti na levou a pravou část, které můžeme řešit samostatně – nevadí nám, pokud budou všechna čísla v levé polovině menší, než v pravé (nebo naopak větší, či nějak promíchaná). Zkusme z toho vymyslet algoritmus.

Mohli bychom vždy nalézt nějakou pozici, která není ničím zdola omezená, na tuto pozici umístit nejmenší zatím nepoužité číslo ze vstupu a tento postup opakovat pro vzniklou levou a pravou část. Tím dostaneme rozmístění čísel, které respektuje všechny nerovnosti.

Abychom nemuseli pokaždé hledat, která pozice je zrovna zdola neomezená, můžeme si pozice zkusit očíslovat v pořadí, v jakém budou přicházet za sebou. Nejlevější pozici přiřadíme číslo 0 a každé další pak číslo o jedna větší (pokud zde byla relace $<$), nebo o jedna menší (pokud zde byla relace $>$), než měla pozice předchozí. Například pokud na vstupu dostaneme relace `< > > < >`, očíslování bude: 0, 1, 0, −1, 0, −1.

Toto očíslování má jednu důležitou vlastnost: kdykoli nějaká relace předepisuje, že číslo na i-té pozici má být větší než na j-té, bude očíslování i větší než očíslování j. To přímo plyne z toho, jak očíslování vytváříme, ale možná lépe je to vidět na následujícím obrázku:



Všechny relace jsou orientované „větším koncem nahoru“. Kdykoli nějaká relace předepisuje nerovnost dvou pozic, ta větší bude v obrázku výš, neboli bude mít větší očíslování.

Když si pak pozice seřadíme podle tohoto očíslování od nejmenšího, tak víme, že každé pozici, která přijde na řadu, můžeme dát nejmenší zatím nepoužité číslo ze vstupu. Všechny prvky, které měly vynucené menší číslo než ona, už své

číslo dostaly, a zbylé pozice mají vynucené číslo větší, nebo s ní ve vztahu vůbec nejsou.

Celý algoritmus tedy spočívá v tom si očíslovat pozice podle relací, seřadit pozice podle tohoto očíslování, seřadit vstupní čísla od nejmenšího a pak je postupně přiřazovat.

Na celém řešení je nejpomalejší třídění, které zabere čas $\mathcal{O}(N \log N)$, paměti spotřebujeme $\mathcal{O}(N)$. Na vzorovou implementaci z CodExu se můžete podívat níže.

KSP

Na závěr ještě dodáme, že na úlohu se lze dívat i grafově. Pro každou pozici si vytvoříme jeden vrchol a mezi sousedními vrcholy povede hrana orientovaná „od menšího konce relace k většímu“. Výše popsané očíslování pak není nic jiného než topologické uspořádání na tomto grafu.

Program (C):

`http://ksp.mff.cuni.cz/viz/28-4-3.c`

Jirka Setnička

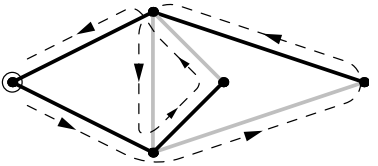
řešení

28-4-4 Podivuhodný obraz

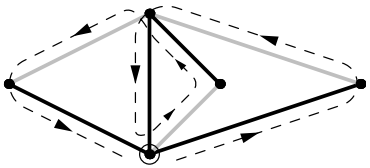
Jako první si všimneme, že obarvení v jednotlivých komponentách souvislosti můžeme volit nezávisle: pokud najdeme správné obarvení pro každou komponentu zvlášť a dáme je dohromady, získáme korektní obarvení celého grafu. Ve zbytku řešení se budeme zabývat tím, jak obarvit jednu komponentu, můžeme tedy předpokládat, že barvený graf je souvislý.

Nejprve vyřešíme jednodušší variantu. Označení úlohy jako kuchařkové nabádá k tomu najít eulerovský tah – což v souvislém grafu se sudými stupni můžeme udělat. Nyní stačí projít hrany v pořadí určeném eulerovským tahem a obarvovat je střídavě (červená, černá, červená, ...). Jako *dobré* budeme označovat vrcholy dotýkající se hran obou barev. Libovolný vrchol uvnitř tahu bude dobrý, protože jsme z něj odejdeme po hraně opačné barvy, než jsme do něj přišli.

To ovšem nemusí platit pro počáteční vrchol tahu (který je zároveň koncovým). Pokud má tah lichou délku (v grafu je lichý počet hran), vrátíme se do počátečního vrcholu hranou stejné barvy, jako jsme z něj vyšli. V případě, že do tohoto vrcholu nevedou žádné jiné hrany (má stupeň 2), výsledné obarvení není správné:



Ovšem pokud má větší stupeň, navštíví jej tah nejen na začátku a konci, ale alespoň jednou jej projde „skrz“ – a v tu chvíli korektně vystřídá barvy. Pokud tedy graf obsahuje vrchol stupně většího než 2, stačí začít obarvovat z tohoto vrcholu a najdeme správné obarvení:



Pokud takový vrchol v grafu není (všechny stupně jsou 2), graf musí být kružnice. Má-li sudou délku, obarvíme ji střídavě a máme vyhráno. Má-li lichou délku, snadno si rozmyslíte, že správné obarvení neexistuje.

Liché stupně

Nyní se vrhněme na těžší variantu. Nejprve jednoduché pozorování: listy (vrcholy stupně 1) nikdy nemohou být dobré, graf který obsahuje list, tedy nejde obarvit. Dále budeme uvažovat jen grafy bez listů.

Když graf obsahuje vrcholy lichého stupně, eulerovský tah neexistuje. Asi by se hodilo nějak graf upravit, aby měl opět všechny stupně sudé.

Lidi v takovéto situaci často napadá zkoušet vrcholy lichého stupně umazávat, obarvit zbytek a pak je tam nějakým způsobem vracet. Ale to je slepá ulička. Například proto, že graf může mít klidně *všechny* stupně liché…

My si poradíme jinak. Do grafu přidáme nový vrchol ω a spojíme s ním všechny vrcholy, které měly původně lichý stupeň. Tím se jejich stupeň změní na sudý. Ale nemůže se stát, že by nový vrchol ω měl lichý stupeň?

Nikoliv, neboť v každém grafu platí, že součet stupňů všech vrcholů je sudý. Pokud bychom každou hranu pomyslně rozsekli uprostřed na dvě poloviny, snadno nahlédnete, že součet stupňů je rovný počtu půlhran. A ten je určitě sudý. Proto žádný graf nemůže mít právě jeden vrchol lichého stupně.

Upravený graf má tedy všechny stupně sudé a můžeme v něm najít eulerovský tah. Opět budeme barvit podél tahu střídavě, ale tentokrát začneme z vrcholu ω. Tím se zbavíme speciálního ošetřování počátečního vrcholu, protože ω nemusí splňovat podmínky na obarvení. Zbývá si rozmyslet, že takto získáme správné obarvení.

Nejprve však trocha názvosloví: nově přidaným vrcholům a hranám budeme říkat *virtuální*, původním *skutečné*. *Skutečný stupeň* vrcholu v je stupeň, který měl v ve vstupním grafu. Vrchol je dobrý, když se dotýká *skutečných* hran obou barev.

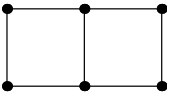
Vrcholy se sudým skutečným stupněm jsou určitě dobré, protože jsme do nich museli přijít i opustit je po skutečné hraně (žádné virtuální hrany nemají). A tyto hrany mají opačnou barvu.

U vrcholů s lichým skutečným stupněm je to složitější, protože do nich můžeme přijít či z nich odejít po virtuální hraně. Ale protože původní graf neobsahoval listy, všechny mají skutečný stupeň alespoň 3, tedy v upraveném grafu stupeň alespoň 4. Každým takovým vrcholem musí tah projít alespoň dvakrát.

A pouze jednoho z těchto průchodů se může účastnit virtuální hrana (každý vrchol má nejvýše jednu virtuální hranu). Při tom druhém tedy přijdeme i odejdeme po skutečné hraně, a tyto hrany mají opačnou barvu. Tedy i všechny vrcholy s lichým skutečným stupněm jsou dobré a naše obarvení je korektní.

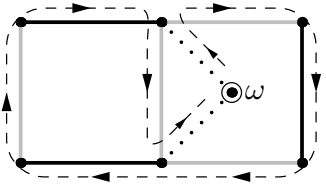
KSP

Například následující graf:



obarvíme takto:

řešení



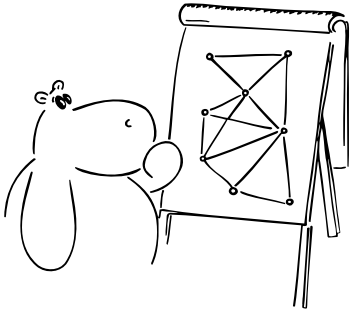
Na závěr shrňme celý algoritmus:

1. Najdeme komponenty souvislosti G.
2. Pro každou komponentu C:
3. Pokud C je lichá kružnice nebo obsahuje listy, vypíšeme „řešení neexistuje“ a skončíme.
4. Pokud C obsahuje vrcholy lichého stupně, všechny je spojíme s nově vytvořeným vrcholem ω.
5. Určíme výchozí vrchol z jako:
6. ω, pokud jsme tento vrchol přidávali;
7. jinak libovolný vrchol stupně alespoň čtyři, pokud takový existuje;
8. jinak zcela libovolný vrchol (C je kružnice).
9. Najdeme uzavřený eulerovský tah t ze z do z.
10. Projdeme hrany v pořadí určeném t a barvíme je střídavě.

Program (C):

`http://ksp.mff.cuni.cz/viz/28-4-4.c`

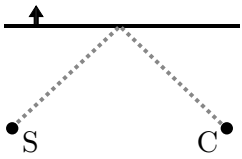
Filip Štědronský



28-4-5 Hra kroket

Zase po nějaké době musíte začít autorské řešení omluvou: až nad šesti přijatými pracemi nám došlo, že jsme pořádně nespecifikovali zadání. O co konkrétně šlo?

Chtěli jsme na co nejmenší vzdálenost projet všechny branky v předepsaném směru. Sporným bodem ale byla definice toho, co přesně znamená *projet branku*. Stačí do ní ťuknout míčkem z jedné strany a ihned se vrátit, nebo se musíme dostat i na druhou stranu? Následující trasa míčku by v prvním případě byla korektní, v tom druhém ne:

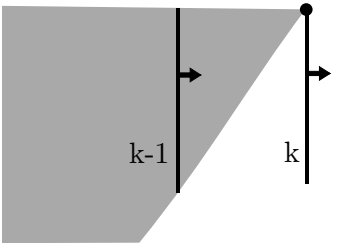


Pro účely našeho řešení uvažme, že úsečka tvořící branku (respektive přímka, na které leží) dělí celé hřiště na dvě poloroviny. Projet branku pak znamená, že míček se nejprve nachází v první a pak, skrz úsečku, přejde do druhé poloroviny. V obou stavech se míček alespoň na chvíli musí nacházet ostře, tedy ne na přímce, která roviny dělí.

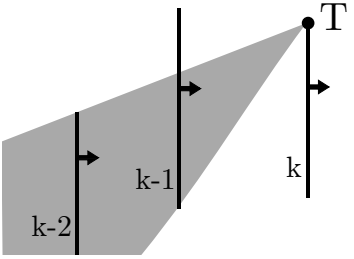
Jak v takovém případě vyřešit situaci z obrázku? Značený tah naše podmínky nesplňuje. Někteří řešitelé si snažili pomoci tím, že při průniku s brankou popojeli nahoru a dolů o nejmenší možnou vzdálenost – tím ji skutečně prošli v obou směrech. Klasická eukleidovská rovina, v níž hru hrajeme, ale takový posun neumožňuje. Stačí uvážit, že pokud uděláte jakkoliv malý krok, můžete jej zkrátit tím, že jeho délku vydělíte dvěma. K jakékoliv trase míčku pak dokážete najít kratší řešení: minimum tedy neexistuje a úloha řešení nemá. Podobným argumentem můžete ukázat nesprávným i postup, kdy míčkem „objíždíme“ koncovou tyčku – ani zde neexistuje minimální délka.

A teď už k popisu našeho algoritmu. Nejprve jedno pozorování: při naší definici nejkratší cesta skrz branky vytvoří posloupnost rovných čar, lámající se jen v tyčkách (krajních bodech) branky. Exaktní důkaz by byl trochu složitější. Zkuste si alespoň představit, že ke startu přivážete provázek a ten pak vedete všemi brankami podle pořadí až do cíle, a následně ho napnete. Kde se bude ohýbat?

Považujme teď startovní a cílový bod za branky nulové délky. Algoritmus projde všechny tyčky branek a pro každou z nich zjistí, odkud k ní může přímým vrhem (tedy bez ohýbání) přiletět míček. Mějme tyčku T, která leží u K-té branky. Vezměme obě tyčky u branky $K-1$ a veďme jimi z T polopřímky. Výseč mezi nimi odpovídá prostoru, ve kterém můžeme do T odpálit míček letící skrz branku $K-1$:



Pokračujeme u branky $K-2$ a dalších. Vždy určíme polopřímky vedoucí z T do tyček, čímž vznikne další výseč. Uděláme průnik s prostorem, který jsme dostali v předešlém kroku. Ve výsledné výseči se můžeme dostat do T přes mezilehlé branky:



Pokračujeme tak dlouho, dokud nedojdeme do startu, nebo nedostaneme prázdnou výseč. Navíc musíme kontrolovat povolený směr průchodu brankou: Již na začátku se podle něj omezíme na jednu z polorovin určených brankou u T. Pro branku $K-1$ a další ověříme, že skrze správný směr by míček směřoval do části výseče, kde se nachází T. Jinak průchod ukončíme.

Čeho jsme tím dosáhli? Dokážeme pro dvě libovolné tyčky (včetně startu a cíle) určit, zda mezi nimi může přímo proletět míček. Proto si založíme graf, jehož vrcholy odpovídají tyčkám, a hrana mezi nimi povede, pokud se z jedné tyčky do

druhé můžeme dostat „na jedno ťuknutí“. Hrany budou ohodnocené vzdáleností tyček. Pak už jen stačí zaměstnat Dijkstrův algoritmus a najít nejkratší cestu ze startovní do cílové tyčky. (Rozmyslete si, že v případě odpovídajícím prvnímu obrázku se žádná cesta nenajde.)

Při N brankách tedy máme $2N + 2 = \mathcal{O}(N)$ tyček, procházení u každé z nich zabere $\mathcal{O}(N)$, takže dohromady $\mathcal{O}(N^2)$. Složitost Dijkstry je stejná (máme nejvýše $\mathcal{O}(N^2)$ hran). Celková časová složitost algoritmu je tedy $\mathcal{O}(N^2)$, stejně tak paměťová složitost (tu navyšuje graf).

Pár poznámek k implementaci: výseče je možné v programu reprezentovat prostě vrcholem a dvojicí úhlů. Vzhledem k tomu, že nás nezajímá, jak přesně jsou tyto úhly velké, ale stačí nám je jen umět porovnávat, můžeme místo úhlů pracovat se *směrnicemi*.[67] Tím si ušetříme počítání s goniometrickými funkcemi. Pro určení toho, ve které polorovině od dané branky leží nějaký bod, můžeme použít techniky popsané v naší geometrické kuchařce.[68] Podrobněji ve vzorovém programu.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-4-5.py`

A na závěr doplnění k naší omluvě: Pokud by u jakékoliv úlohy vypadalo, že vás nutíme používat podobně chlupaté postupy, jako nekonečně malé kroky, raději se zeptejte na fóru. Orgové sice často jsou docela osobití lidé, ale takové šílenosti schválně nezadávají (případně si je nechávají v záloze na soustředění).

Kuba Maroušek

28-4-6 Mediánové třídění

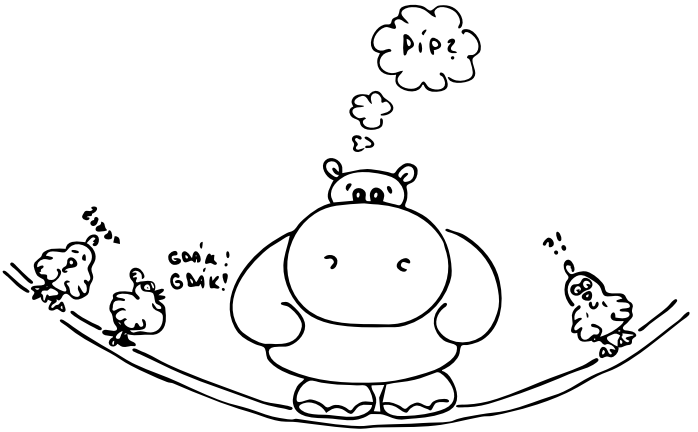
Jak užít mediánový blackbox na třídění, není na první pohled úplně jasné. Proto se místo tahání králíka z klobouku pokusíme nastínit postup, jakým se na řešení dalo vlastně přijít.

Když známe medián z (fixních) K čísel, krabička nám říká, že v ní je právě $\frac{K-1}{2}$ menších čísel než medián. (A stejný počet větších.) Kdybych tedy chtěl zjistit, které číslo v krabičce je o jedno větší než medián, stačí do krabičky nyní vložit nějaké hodně velké číslo, větší než všechna čísla v krabičce. Tím z krabičky vyhodíme … první prvek, který jsme do ní vložili.

Chtěli bychom vyhodit ten nejmenší. Tak si K o pár prvků zvětšíme a nejdřív do krabičky vložíme nějaká hodně malá čísla (menší než všechna v krabičce). Tím si zajistíme, že vyhozený prvek bude určitě menší než medián.

[67] `https://en.wikipedia.org/wiki/Slope`

[68] `http://ksp.mff.cuni.cz/viz/kucharky/geometrie`



KSP

Rýsuje se nám tady základ algoritmu.

řešení

1. Nechť $K = V + N$.
2. Vlož do krabičky V malých čísel
3. Vlož do krabičky celou vstupní posloupnost (N čísel).
4. V-krát:
5. Vypiš medián z krabičky.
6. Vlož do krabičky velké číslo (tím vypadne jedno malé, které jsme vložili na začátku).
7. Vypiš medián z krabičky.

Tím jsme právě vypsali setříděnou posloupnost $V + 1$ čísel v okolí mediánu. Můžeme tedy zvolit $V = N - 1$ a výše uvedený algoritmus nám vydá setříděnou posloupnost N čísel, což je přesně setříděná vstupní posloupnost.

Například pokud dostaneme k setřídění posloupnost 4, 2, 3, 1, nastavíme $V = 3$ a $K = 7$. Vývoj obsahu krabičky je popsán v následující tabulce. Medián je tučně zvýrazněn v posledním sloupci, $-\infty$ a ∞ značí ona hodně malá/velká čísla.

Krok	Obsah krabičky	Setříděný obsah
7	$-\infty$, $-\infty$, $-\infty$, 4, 2, 3, 1	$-\infty$, $-\infty$, $-\infty$, **1**, 2, 3, 4
8	$-\infty$, $-\infty$, 4, 2, 3, 1, ∞	$-\infty$, $-\infty$, 1, **2**, 3, 4, ∞
9	$-\infty$, 4, 2, 3, 1, ∞, ∞	$-\infty$, 1, 2, **3**, 4, ∞, ∞
10	4, 2, 3, 1, ∞, ∞, ∞	1, 2, 3, **4**, ∞, ∞, ∞

Dostaneme postupně mediány 1, 2, 3, 4, tedy přesně setříděnou posloupnost. A máme téměř vyřešeno.

Potřebujeme ještě vymyslet, jak najít dostatečně malá a dostatečně velká čísla na posouvání mediánem v krabičce. Jednoduše. Najdeme minimum a maximum ze vstupu a to použijeme.

Finální algoritmus tedy vypadá takto:

1. Nechť $K = 2N - 1$.
2. Projdi celý vstup, spočítej z něj minimum (označíme m) a maximum (označíme M) v čase $\mathcal{O}(N)$.
3. Vlož do krabičky $(N-1)$-krát m v celkovém čase $\mathcal{O}(N)$.
4. Vlož do krabičky vstupní posloupnost v celkovém čase $\mathcal{O}(N)$.
5. $(N-1)$-krát:
6. Vypiš medián z krabičky.
7. Vlož do krabičky M.
8. Vypiš medián z krabičky.

Na závěr bychom rádi uvedli na pravou míru, proč jsme vlastně takovouto na první pohled podivnou úlohu zadávali. Představte si, že bychom mediánovou krabičku nedostali jako kouzelnou skříňku, ale chtěli si ji poctivě implementovat. To jsme po vás chtěli například v úloze 27-2-1,[69] kde autorské řešení zvládlo jednu operaci (přidání prvku a vypsání mediánu) zpracovat v čase $\Theta(\log N)$. Nemohlo by to jít lépe?

Nemohlo. Označme si složitost jedné operace $T(k)$. Z řešení naší úlohy víme, že s pomocí takovéto krabičky dokážeme setřídit N čísel v čase $N \cdot T(N)$. Z toho tedy plyne, že $T(N)$ nemůže být lepší než $\Omega(\log N)$, protože kdyby byla, uměli bychom třídit rychleji než v čase $\Omega(N \log N)$. A je všeobecně známo, že to (za určitých předpokladů) není možné. Přečíst si o tom můžete například v naší kuchařce o třídění.[70]

Jan „Moskyto“ Matějka

28-4-7 Jízda sanitkou

Pro každé políčko určitě budeme potřebovat umět rychle zjistit vzdálenost od nejbližšího místa opravy (označme si toto číslo K). Pokud bychom měli jen jedno místo opravy, stačí z tohoto políčka pustit prohledávání do šířky a u každého políčka si poznamenávat vzdálenost. Pokud máme míst opravy více, všimneme si, že stačí na začátku přidat do fronty všechna místa opravy a použít stejný algoritmus. Postupně takto navštívíme políčka s $K = 1, 2, 3, \ldots$

Máme mapu, pro každé políčko známe jeho K a chceme najít takovou cestu ze startovního políčka do libovolné nemocnice, aby nejmenší K na této cestě bylo

[69] `http://ksp.mff.cuni.cz/viz/27-2-1`

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

co největší. Přímočaré řešení využívá Dijkstrův algoritmus, jen délku cesty budeme počítat jako minimum z aktuální délky cesty a K nového políčka. Dijkstrův algoritmus postupně hledá cesty s největším ohodnocením (v našem případě je ohodnocení cesty rovno nejmenšímu K na této cestě) do všech vrcholů. Postupně vybírá vrcholy, do kterých aktuálně známe nejlepší cestu, a aktualizuje sousedy těchto vrcholů tak, že pro každého souseda zkusíme, jestli do něj nevede lepší cesta přes aktuální vrchol. Pro přesný popis algoritmu viz řešení úlohy 28-2-1,[71] kde místo maximalizace minimálního K minimalizuje maximální K. Stačí tedy jen prohodit maximum a minimum. Algoritmus běží v čase $\mathcal{O}(RS \log(RS))$, kde R a S jsou rozměry mapy. Jde to ale rychleji, pojďme se podívat, jak.

KSP

Pokud bychom dopředu znali minimální K optimální cesty (označme toto optimum D), mohli bychom pustit prohledávání do šířky ze startu S a políčka s $K < D$ považovat na neprůchozí. Pokud bychom během průchodu nenarazili na žádnou nemocnici, cesta přes vrcholy s $K \geq D$ neexistuje. Můžeme postupně zkoušet všechna možná D přes hodnoty $R+S, R+S-1, R+S-2\ldots$, dokud nenarazíme na nemocnici. Ve chvíli, kdy jsme našli nemocnici, máme určitě optimální cestu (pokud by existovala lepší, našli bychom ji už v nějakém předchozím průchodu). Protože K je maximálně $\mathcal{O}(R+S)$, dostáváme kvadratický algoritmus vůči počtu políček.

řešení

Zamyslíme-li se nad průchodem algoritmu, zjistíme, že zbytečně prochází opakovaně části mapy. Pak si všimneme, že K dvou sousedních políček se může lišit maximálně o jedna, tedy políčka, která jsme při nějakém průchodu objevili jako nedostupná, budou hned v příštím průchodu dostupná. Navíc, políčka, která jsme aktuálně prošli, již znovu procházet nemusíme, protože víme, že na žádném z nich určitě není nemocnice (jinak bychom skončili). Pořídíme si tedy pomocnou frontu a při průchodu do ní budeme vkládat všechna objevená nedostupná políčka. Ve chvíli, kdy vyprázdníme frontu pro průchod do šířky, můžeme zmenšit požadované D o 1 a zpřístupnit tak políčka v pomocné frontě. Prohodíme obě fronty a pokračujeme v průchodu.

Takto postupně procházíme cesty s čím dál tím nižšími D. První hodnota, kterou vyzkoušíme, je K startovního políčka (vyšší hodnoty nemá smysl zkoušet).

Každé políčko vložíme maximálně jednou do jedné z front a jednou jej vyjmeme. Protože každé políčko má maximálně čtyři sousedy, máme algoritmus běžící v čase $\mathcal{O}(RS)$ s paměťovou složitostí $\mathcal{O}(RS)$, což už jistě zlepšit nepůjde.

Program (C++):

`http://ksp.mff.cuni.cz/viz/28-4-7-bfs.cpp`

Martin „Medvěd“ Mareš & Honza Knížek

[71] `http://ksp.mff.cuni.cz/viz/28-2-1/reseni`

28-4-8 Strojové učení

Náhodné rozdělení datasetu

Ptali jsme se vás, proč je potřeba dataset rozdělovat na trénovací a testovací množinu náhodně. Na našem příkladu to není těžké ukázat: v datasetu $\mathcal{D}$ máme nejdřív 100 značek „stop“, pak 100 značek „dej přednost v jízdě“ a nakonec 100 značek „slepá ulice“.

Kdybychom například prvních 90 % datasetu použili na trénování a posledních 10 % na testování, testovací množina by obsahovala jenom značky „slepá ulice“ a tudíž by měření výkonu na testovací množině jenom měřilo, jak dobře poznáváme tenhle jediný druh značek, místo toho, aby obsahovala rovnoměrný vzorek.

S dostatečně malým štěstím se může rozbít ještě jedna věc. Ať rozpoznáváme 50 druhů značek a od každé máme 100 vzorků. Kdybychom je měli v datasetu postupně za sebou a vybrali bychom prvních 90 % na natrénování, model by uměl rozpoznat prvních 45 druhů značek. Na posledních 5 druzích by ale samozřejmě neuměl nic, protože by neměl vůbec příležitost se je naučit.

Hloupé učení

Ať zkoušíme do všech složek vektoru β dosadit všechny hodnoty od -100 do 100 s krokem 0.1, kterých je 2001. Vektor β má p složek, takže budeme testovat 2001^p různých modelů. Ohodnocení jednoho modelu trvá čas $\Theta(|S|)$. Dohromady by tohle „triviální učení“ trvalo čas $\Theta(|S|\cdot 2001^p)$, neboli, odborně řečeno, dlouho.

Spotřeba benzínu

Na lineární regresi nic nebylo a všem, kdo se o ni pokusili, se povedla. Stačí naimplementovat algoritmus podle našeho návodu.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-4-8-3.py`

Kvalita vína

Opět stačilo následovat návod. Klíčové bylo použít trénovací set jako data pro modely, vybrat model s nejlepším K podle nejmenší kvadratické chyby a nakonec odhadnout skutečnou accuracy nejlepšího modelu s pomocí testovací množiny.

Na trénovací množině bude nejmenší chyba pro $K = 1$, protože pak bude kvalita každého vína X v trénovací množině předpovězena podle jeho jediného nejbližšího souseda v trénovací množině, tedy podle X. Proto z definice bude chyba na trénovací množině pro $K = 1$ rovná nule. Se zvyšujícím se K se bude chyba zvětšovat.

Na validační množině je nejmenší chyba pro K „někde uprostřed“. Pro menší K se nechá model více ovlivňovat lokálním šumem a pro větší K naopak průměruje tak moc blízkých vzorků, že si nevšimne lokálních vlastností datasetu a čím dál tím víc jenom počítá průměr ze všech vín.

To, které konkrétní K dá nejmenší chybu na validační množině, záleží na náhodě (tj. na rozdělení datasetu). Nám například vyšlo $K = 15$, které mělo na validační množině střední kvadratickou chybu 0.6343.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-4-8-4.py`

Kosatce

Naše autorské řešení funguje tak, že si natrénujeme tři perceptrony, jeden pro každý druh kosatce. Úkolem perceptronů je říct 1 na kosatcích jejich druhu a -1 na všech ostatních.

V ideálním světě bychom doufali, že výstup perceptronů bude $[1; -1; -1]$, $[-1; 1; -1]$, nebo $[-1; -1; 1]$, a jako výstupní třídu bychom zvolili tu, jejíž perceptron řekl „1“.

To se ale bohužel nepovede. Někdy se nám stane, že třeba dostaneme výstupy $[1; 1; -1]$. Kterou třídu zvolíme potom?

V autorském řešení jsme k rozseknutí těchhle sporných případů použili pár pravděpodobnostních triků. V téhle úloze nebyly nutně potřeba. Uvádíme je spíše pro zajímavost, protože podobný princip se používá například na bayesovské rozpoznávání spamu, které jste mohli vidět na Jarním soustředění na přednášce o strojovém učení.

Nechť nám přijde ke klasifikaci nový kosatec. Jeho skutečnou třídu si označíme jako C. Budeme se na ni dívat jako na náhodnou proměnnou, která má jednu ze tří hodnot: s jako setosa, v jako versicolor, nebo g jako virginica.

Parametry kosatce vložíme do tří natrénovaných perceptronů a jejich výstupy, které jsou rovné 1 nebo -1, si označíme jako P_s, P_v a P_g. Výstupy perceptronů použijeme jako signály, podle kterých spočítáme pravděpodobnosti, že $C = s$, $C = v$ a $C = g$.

Když na kosatci dostaneme výstupy $P_s = 1$, $P_v = -1$ a $P_g = 1$, zajímají nás *podmíněné pravděpodobnosti*, že při takovýchto výstupech perceptronů je kosatec skutečně setosa, versicolor, nebo virginica. Podmíněná pravděpodobnost, že za našich podmínek je kosatec setosa, se zapisuje:

$$\mathrm{P}[C = s | P_s = 1, P_v = -1, P_g = 1]$$

Jako výstupní třídu vrátíme tu, která bude mít tuto podmíněnou pravděpodobnost největší.

Podmíněná pravděpodobnost jevu X za předpokladu jevu Y je definovaná jako pravděpodobnost, že se stane jev X i jev Y, lomeno pravděpodobnost, že se stane Y:

$$\mathrm{P}[X|Y] = \frac{\mathrm{P}[X, Y]}{\mathrm{P}[Y]}$$

Dále použijeme *Bayesovu větu*, která říká:

$$\mathrm{P}[X|Y] = \frac{\mathrm{P}[Y|X] \cdot \mathrm{P}[X]}{\mathrm{P}[Y]}$$

V našem případě, kde X je $C = s$ a Y je jev $P_s = 1, P_v = -1, P_g = 1$, rozepíšeme podmíněnou pravděpodobnost následovně:

$$\mathrm{P}[C = s|P_s = 1, P_v = -1, P_g = 1] =$$

$$= \frac{\mathrm{P}[P_s = 1, P_v = -1, P_g = 1|C = s] \cdot \mathrm{P}[C = s]}{\mathrm{P}[P_s = 1, P_v = -1, P_g = 1]}$$

Protože perceptrony jsou natrénované nezávisle na sobě, budeme předpokládat, že P_s, P_v, P_g jsou *nezávislé náhodné proměnné*. To znamená dvě věci, jednak:

$$\mathrm{P}[P_s = 1, P_v = -1, P_g = 1] =$$

$$= \mathrm{P}[P_s = 1] \cdot \mathrm{P}[P_v = -1] \cdot \mathrm{P}[P_g = 1]$$

a druhak:

$$\mathrm{P}[P_s = 1, P_v = -1, P_g = 1|C = s] =$$

$$= \mathrm{P}[P_s = 1|C = s] \cdot \mathrm{P}[P_v = -1|C = s] \cdot \mathrm{P}[P_g = 1|C = s]$$

Celá podmíněná pravděpodobnost, že daný kosatec je setosa, tedy bude vypadat takhle:

$$\mathrm{P}[C = s|P_s = 1, P_v = -1, P_g = 1] =$$

$$= \mathrm{P}[C = s] \cdot \frac{\mathrm{P}[P_s = 1|C = s]}{\mathrm{P}[P_s = 1]} \cdot \frac{\mathrm{P}[P_v = -1|C = s]}{\mathrm{P}[P_v = -1]} \cdot \frac{\mathrm{P}[P_g = 1|C = s]}{\mathrm{P}[P_g = 1]}$$

$\mathrm{P}[C = s]$ je pravděpodobnost, že náhodně vybraný kosatec je setosa. Dataset obsahuje 50 kosatců setosa, 50 kosatců versicolor a 50 kosatců virginica, takže v našem případě $\mathrm{P}[C = s] = \mathrm{P}[C = v] = \mathrm{P}[C = g] = 1/3$.

Teď potřebujeme pro každý z perceptronů zjistit pravděpodobnost, že dá na náhodném kosatci výstup 1, resp. -1. Taky potřebujeme všechny podmíněné pravděpodobnosti typu $\mathrm{P}[P_s = 1|C_s]$ (to je pravděpodobnost, že pokud náhodně vybereme kosatec typu setosa, perceptron na rozpoznávání typu setosa na něm vrátí 1).

Rozdělíme si dataset na trénovací, *kalibrační* a testovací množinu. Na trénovací množině natrénujeme perceptrony. Potom použijeme kalibrační množinu na určení pravděpodobností typu $\mathrm{P}[P_s = 1]$ a $\mathrm{P}[P_s = 1|C_s]$.

Budeme si budovat třírozměrné pole A. Jeho první index bude *skutečná třída kosatce*, druhý index bude označení perceptronu, třetí index bude označovat, jestli perceptron vrátil 1, nebo -1. Pole nejdřív naplníme nulami, a pak pojedeme přes každý kosatec v kalibrační množině. Každý kosatec strčíme do všech perceptronů a podíváme se na jejich výstupy. Pro každý perceptron připočteme jedničku do příslušného místa v poli. Například, když uvidíme kosatec typu virginica, na kterém perceptrony řekly $P_s = 1, P_v = 1, P_g = -1$, tak připočteme jedničku k buňkám pole $A[v][s][1]$, $A[v][v][1]$, $A[v][g][-1]$.

Tohle pole použijeme k odhadnutí chybějících pravděpodobností. Konkrétně:

$$\mathrm{P}[P_s = 1|C_s] \simeq \frac{A[s][s][1]}{(A[s][s][1] + A[s][s][-1])}$$

$$\mathrm{P}[P_s = 1] = \mathrm{P}[P_s = 1|C_s] + \mathrm{P}[P_s = 1|C_v] + \mathrm{P}[P_s = 1|C_g]$$

Zbývá jenom jeden problém. Představme si, že na kalibrační množině se všechny perceptrony náhodou chovají dokonale, tedy vrátí 1 na svém druhu -1 na ostatních kosatcích z kalibrační množiny. Potom bude naše kalibrace odhadovat, že například $\mathrm{P}[P_v = 1|C = s] = 0$, protože v kalibrační množině nikdy nedával perceptron pro virginicu výsledek 1 zatímco kosatec byl ve skutečnosti setosa. Stejně tak $\mathrm{P}[P_s = 1|C = v] = 0$ a $\mathrm{P}[P_s = 1|C = g] = 0$.

Zkusme potom do takhle zkalibrovaného modelu strčit kosatec, na kterém perceptrony dají výsledek $P_s = 1$, $P_v = 1$, $P_g = -1$. Když počítáme podmíněnou pravděpodobnost $\mathrm{P}[C = s|P_s = 1, P_v = 1, P_g = -1]$, tak jeden ze členů, kterými nahoře násobíme, je $\mathrm{P}[P_v = 1|C = s]$, což je 0, takže i celá podmíněná pravděpodobnost vyjde 0. Podobně se nám na nulu zredukuje i pravděpodobnost $\mathrm{P}[C = v|P_s = 1, P_v = 1, P_g = -1]$ a $\mathrm{P}[C = g|P_s = 1, P_v = 1, P_g = -1]$. Všechny podmíněné pravděpodobnosti odhadneme na 0 a nevíme nic. To je problém ze dvou důvodů: jednak by součet pravděpodobností měl vyjít 1 (protože přece ten kosatec musí někam skutečně patřit), a druhak přece i takhle máme nějakou informaci, kterou bychom mohli nějak využít: $P_g = -1$, takže to virginica spíš nebude, než bude.

Náš program tady používá takzvaný *add-one smoothing*. Je to jednoduché: místo toho, abychom začali s polem P plným nul, naplníme ho místo toho jedničkami. Tím zajistíme, že z modelu nikdy nevypadne pravděpodobnost nula, a to i když uvidí nějakou situaci, jaká není v kalibrační množině.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/28-4-8-5.py`

Michal Pokorný

Pátá série

28-5-1 Zaměřování místnosti

Nejprve si všimneme, že pokud obdélník existuje, pak nějaké dva body z libovolné pětice určují jednu stranu obdélníku. To vyplývá z tzv. Dirichletova principu (neboli principu holubníku), který v tomto případě říká, že pokud máme pět bodů, které leží na čtyřech stranách, tak nutně na alespoň jedné straně jsou alespoň dva body.

Vezmeme tedy dva body z náhodně vybrané pětice a zkusíme, jestli s jednou stranou určenou touto dvojicí bodů obdélník existuje. (Jak přesně to zjistíme, je vysvětleno v následujícím odstavci.) Když chceme vybrat dva body z pěti, máme celkem deset možností, jak to udělat. Proč? Pokud znáte kombinační čísla, tak jistě dokážete spočítat, že vybíráme-li dva z pěti bodů, možností, jak to udělat, je deset.

Pokud vám kombinační čísla nic neříkají, stačí jednoduchá úvaha: je pět možností, jak vybrat první bod, a pro každou z těchto pěti možností jsou čtyři další, jak k nim vybrat druhý bod. Pět krát čtyři je tedy dvacet možností, když víme, jaký bod byl vybrán první a jaký druhý. Nám ale na jejich pořadí nezáleží a můžeme je mezi sebou zaměnit. Tedy dvacet vydělíme dvěma a dostáváme, že způsobů, jak vybrat dva z pěti bodů, když nezáleží na pořadí, je deset.

Teď si ukážeme, jak s libovolnou dvojicí bodů postupovat. U ostatních bodů zkontrolujeme, zdali neleží na této první přímce také. Pokud ano, můžeme je pro teď zapomenout. Pro všechny zbylé body si spočítáme vzdálenost od první přímky. Body (nebo bod) s největší vzdáleností budou ležet na opačné straně obdélníka a také je můžeme pro teď odstranit. Zbylé body by tedy měly ležet na nejvýše dvou přímkách kolmých na dosud určené přímky. Navíc musí být tyto kolmé přímky od sebe vzdálené alespoň tak, jak jsou od sebe vzdálené krajní body na původních přímkách, tak aby tvořily obdélník.

Pro každou dvojici tedy provedeme řádově N operací, kde N je počet bodů celkem. Různých dvojic je celkem deset, složitost je tak $10N$, tedy $\mathcal{O}(N)$ – lineární.

Zbývá vyřešit případy, kdy je bodů méně než pět. Pokud jsou body tři a méně, lze obdélník sestrojit vždy. Pokud jsou body čtyři, můžeme si pomoci například sestrojením konvexního obalu. Pokud na něm nějaký bod neleží, obdélník existovat nemůže.

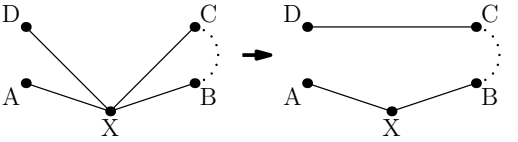
Někteří zvolili k celému řešení úlohy sestrojení konvexního obalu a následně jej různě využili ke zjištění, zda jde o obdélník, nebo ne. Většina nápadů nebyla špatná a fungovala by, ale sestrojit konvexní obal trvá $\mathcal{O}(N \log N)$, a je to tedy pomalejší než výše popsané lineární řešení. Jediný řešitel, který správné lineární řešení poslal, byl Tomáš Domes, kterého tímto chceme veřejně pochválit.

Zuzka Drázdová

28-5-2 Místní přesuny

KSP

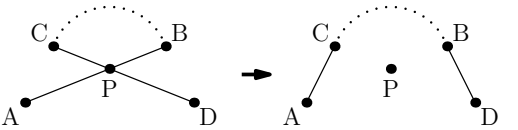
Nejprve si uvědomíme, že se žádný bod nevyplatí navštívit dvakrát. Druhou návštěvu můžeme vynechat, a tím si cestu vždy zkrátíme:



Z trojúhelníkové nerovnosti $|CD| < |CX| + |XD|$ tedy upravená cesta musí být kratší. Hledáme tedy nejkratší *hamiltonovskou cestu* v úplném grafu, jehož vrcholy jsou naše body a hrany úsečky spojující každou dvojici bodů. To je cesta, která navštíví každý vrchol právě jednou. Pro obecné grafy je její nalezení těžký problém,[72] ale ukážeme si, že v případě vrcholů v konvexní poloze to neplatí.

řešení

Dále si všimneme, že optimální cesta nikdy nekříží sebe sama. Uvažujme libovolnou cestu, na které se kříží nějaké dvě hrany AB, CD. Označme si P jejich průsečík. I takovou cestu snadno zkrátit:



Opět z trojúhelníkové nerovnosti: $|AC| < |AP| + |CP|$, $|BD| < |PD| + |PB|$. Sečtením dostáváme $|AC| + |BD| < |AP| + |PB| + |CP| + |PD| = |AB| + |CD|$, tedy nová cesta je kratší než původní.

Nyní k samotnému algoritmu. Začneme tím, že si vrcholy očíslujeme v pořadí, v jakém leží na obvodu konvexního mnohoúhelníku. K tomu můžeme použít libovolný algoritmus pro hledání konvexního obalu.[73] Protože máme body v konvexní poloze, budou všechny patřit do nalezeného konvexního obalu, ten nám ale navíc určí jejich pořadí. Zvládneme to v čase $\mathcal{O}(N \log N)$.

Očíslování budeme chápat jako cyklické a s čísly vrcholů pracovat modulo N, tedy např. $0 - 1 = N - 1$. Pak sousedé libovolného vrcholu X jsou $X + 1$ a $X - 1$.

Označení úlohy jako kuchařkové napovídá, že chceme použít dynamické programování.[74] Abychom to mohli udělat, musíme úlohu rozdělit na menší podproblémy, z řešení kterýchž lze poskládat řešení původního většího problému.

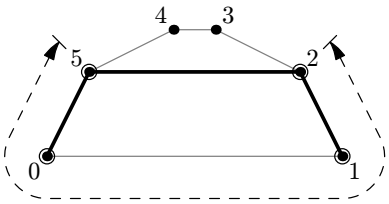
Na první pohled se zdá, že vhodným podproblémem by mohlo být hledání nejkratší cesty pokrývající nějaký souvislý úsek vrcholů mnohoúhelníka. Takový

[72] `http://ksp.mff.cuni.cz/viz/kucharky/tezke-problemy`

[73] `http://ksp.mff.cuni.cz/viz/kucharky/geometrie`

[74] `http://ksp.mff.cuni.cz/viz/kucharky/dynamika`

úsek můžeme identifikovat dvěma čísly (a, b) udávajícími počáteční a koncový vrchol. Pozor, že pokud jde úsek „přes nulu“, může být $b < a$. Na následujícím obrázku je zvýrazněn úsek (5, 2) tvořený vrcholy 5, 0, 1, 2 a nejkratší cesta jej pokrývající:

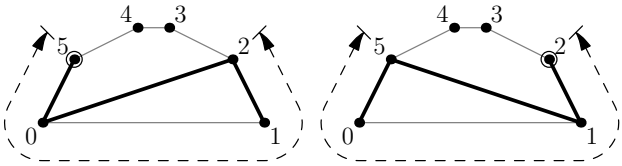


pokrývaný úsek

Na obrázku vidíme další problém: takováto nejkratší cesta je nám k ničemu, protože ji nedokážeme bez křížení prodloužit na nejkratší cestu pokrývající nějaký delší úsek. Aby se na cestu dalo ještě něco napojit, musí mít alespoň jeden konec v krajním bodě intervalu (zde 5 nebo 2).

Zkusme si tedy zesílit zadání: pro každý úsek (a, b) budeme hledat hned dvě cesty. První z nich bude nejkratší cesta navštěvující všechny vrcholy úseku a končící v a. O tom, kde má druhý konec, nic neříkáme. Může to být v b, ale taky někde uprostřed úseku. Označíme si ji $P_1(a, b)$, její délku pak $D_1(a, b)$. Analogicky $P_2(a, b)$ bude nejkratší pokrývající cesta končící v b.

Snadno si navíc rozmyslíte, že ani jedna z těchto cest nikdy nenavštíví vrchol mimo požadovaný úsek (protože takovou nepovinnou návštěvu můžeme vynechat, a tím cestu zkrátit). Může se stát, že $P_1(a, b) = P_2(a, b)$, ale obecně nemusí. Vše je vidět na následujícím obrázku.



$P_1(5, 2)$ $P_2(5, 2)$

Pro jednoduchost budeme počítat jen délku nejkratší cesty, nalezení cesty samotné se snadno doplní. Jak spočítáme např. $D_1(a, b)$?

Uvažujme poslední hranu cesty $P_1(a, b)$, končící v a. To může být buď strana mnohoúhelníka, nebo úhlopříčka. Pokud je to strana, musí vést do vrcholu $a + 1$ (na obrázku 0). Vrchol $a - 1$ (4) leží mimo pokrývaný úsek a výše jsme ukázali, že takové se navštívit nevyplatí.

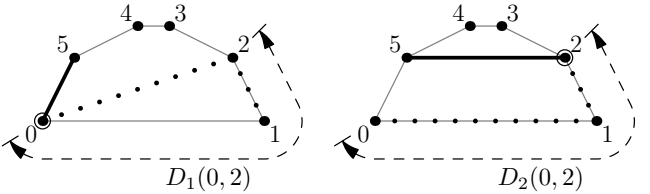
Po odebrání poslední hrany $P_1(a, b)$ dostaneme nějakou cestu pokrývající o jedna kratší úsek, $(a + 1, b)$, která navíc končí v $a + 1$. A musí to být nejkratší

taková, tedy dlouhá $D_1(a+1,b)$. Kdyby byla delší, můžeme ji zkrátit výměnou za $P_1(a+1,b)$.

Pokud je poslední hrana úhlopříčka, musí naopak vést do vrcholu b. Kdyby vedla kamkoli jinam, rozdělí nám pokrývanou část mnohoúhelníka na dvě poloviny, mezi kterými nelze přejít bez křížení cest. Představte si na obrázku výše, co by se stalo, kdyby cesta končila hranou (1, 5). Analogicky po odebrání poslední hrany dostaneme nějakou cestu pokrývající zbylý úsek, ale tentokrát končící v b.

KSP

To jsou jediné dvě možnosti, jak může $P_1(a,b)$ vypadat. Pak stačí vybrat tu kratší:



řešení

Dostáváme jednoduchou rekurenci:

$$D_1(a,b) = \min \begin{cases} d(a,a+1) + D_1(a+1,b) \\ d(a,b) + D_2(a+1,b), \end{cases}$$

kde $d(x,y)$ značí délku příslušné hrany.

Pro D_2 to dopadne analogicky. Nyní stačí hodnoty D_1 a D_2 spočítat pro všech N^2 dvojic dynamickým prograováním. Jeden výpočet trvá konstantně dlouho, takže vše zvládneme v čase $\mathcal{O}(N^2)$. Nakonec stačí z nalezených cest pokrývajících celý mnohoúhelník (rozmyslete si, že to jsou právě $D_1(a,a-1)$ a $D_2(a,a-1)$ pro všechna a) vybrat tu nejkratší.

Spotřebujeme též kvadratické množství paměti. Ale pokud bychom si namísto nejkratší cesty vystačili s její délkou, můžeme paměťovou složitost zlepšit na lineární. Všimneme si totiž, že kdykoli během výpočtu si stačí pamatovat poslední dvě patra rekurze. Když počítáme hodnoty pro úseky délky k, stačí nám k tomu znát hodnoty pro úseky délky $k-1$. Všechny kratší můžeme zahodit.

Program (C):

`http://ksp.mff.cuni.cz/viz/28-5-2.c`

Filip Štědronský

28-5-3 Trezor s alarmem

Schéma zapojenia alarmu predstavuje acyklický orientovaný graf s dvomi vyznačenými vrcholmi – pólmi. Tento graf si označme ako G, záporný začiatočný vrchol ako z a kladný koncový ako k. Našou úlohou je nájsť v grafe G kritické

vrcholy. Kritický vrchol je ten, ktorý leží na každej jednej možnej ceste zo z do k. Počet vrcholov grafu G budeme značiť ako N a počet hrán ako M.

Aby sa nám lepšie s grafom pracovalo, tak ho na začiatku prečistíme. Odstránime si z neho vrcholy (a k ním pripojené hrany), ktoré neležia na žiadnej ceste zo z do k. Je zrejmé, že vrcholy na odstránenie budú práve tie, z ktorých nevedie cesta do k, a tie, do ktorých sa zo z nedá dostať. Žiadne ďalšie to byť nemôžu, lebo vrchol, do ktorého sa dá dostať zo z a zároveň z neho vedie cesta do k, tvorí cestu zo z do k. Graf, ktorý ostane po tomto prečistení, si označme ako G', jeho počet vrcholov N' a hrán M'.

Graf G' je naďalej orientovaný a neexistuje v ňom ani žiaden orientovaný cyklus (odstraňovaním vrcholov a hrán cyklus vzniknúť nemôže). Teda vrcholy grafu G' je možné topologicky zoradiť. To znamená, že vrcholy sa očíslujú od 1 do N', a to tak, že každá orientovaná hrana bude viesť vždy od vrcholu s menším číslom do vrcholu s väčším číslom. Všimnime si, že z dostane číslo 1 a k číslo N', lebo žiaden vrchol nemôže byť väčší ako k a žiaden menší ako z.

Posledný krok bude v grafe G' nájsť kritické vrcholy. Vrchol v bude kritický práve vtedy, keď nebude existovať žiadna hrana vedúca od vrcholu s číslom menším ako v do vrcholu s číslom väčším ako v. Ak by takáto hrana existovala, potom cesta zo z do k, ktorá obsahuje túto hranu nemôže zároveň obsahovať vrchol v (kvôli topologickému usporiadaniu vrcholov). Zároveň vrchol, ktorý neobchádza žiadna hrana, sa bude nachádzať na každej ceste zo z do k, keďže ako jediný spája vrcholy s menšími číslami ako je on sám a s väčšími číslami ako je on sám.

Ostáva nám už iba popis algoritmov na jednotlivé časti rozboru. Prečistenie grafu spravíme dvojitým prehľadaním grafu do hĺbky. Prvý spustíme zo z a označíme si dosiahnuteľné vrcholy po smere hrán. Druhý spustíme z k, ale proti smeru orientácií hrán, čím získame vrcholy, z ktorých sa dá dostať do k. To zvládneme v čase $O(N + M)$, lineárne od počtu hrán a vrcholov v G. Topologické zoradenie vrcholov acyklického orientovaného grafu vieme uskutočniť taktiež v čase $O(N + M)$, napríklad pomocou algoritmu z našej kuchárky o rovinných grafoch.[75]

V poslednom kroku už budeme mať dostupný topologicky zoradený zoznam vrcholov grafu G'. Tento zoznam vrcholov budeme postupne spracovávať od najmenšieho po najväčší. Pri spracovávaní si budeme pamätať a priebežne aktualizovať maximálne číslo m. To bude reprezentovať najvzdialenejší vrchol od začiatku, na ktorý už viedla nejaká hrana.

Spracovávanie každého vrcholu začneme porovnaním čísla tohoto vrcholu a hodnoty m. Ak sú hodnoty rovnaké, potom bude aktuálny vrchol kritický. Platí to, pretože neexistuje hrana zo žiadneho vrcholu s menším číslom ako m do vrcholu s väčším číslom ako m. Následne sa pozrieme na všetkých susedov spra-

[75] `http://ksp.mff.cuni.cz/viz/kucharky/rovinne-grafy`

covávaného vrcholu. Vždy, keď bude číslo niektorého suseda väčšie ako m, tak zmeníme m práve na číslo toho suseda. Týmto docielime, že na konci spracovávania vrcholu, bude v m opäť najväčšie číslo, do ktorého sa vieme dostať.

Aby sa nám spracovávanie zoznamu vrcholov nedostalo do nedefinovaného stavu, tak si na začiatku nastavíme hodnotu m na 1. Zároveň budeme ignorovať informáciu, že kritický vrchol by mal byť aj z (rovný 1) a aj k (najväčší). Tie v skutočnosti netvoria uzly sústavy obvodov zo zadania.

KSP

Posledný krok zvládneme taktiež v čase $O(N + M)$, keďže každý vrchol a každú hranu spracujeme práve raz. Teda celková časová zložitosť bude lineárna od súčtu počtu hrán a vrcholov vo vstupnom grafe G. Ak použijeme klasickú reprezentáciu grafu G a G' ako zoznam susedov pre každý vrchol, tak rovnako bude na tom aj pamäťová zložitosť.

Program (C++):

`http://ksp.mff.cuni.cz/viz/28-5-3.cpp`

Pali Rohár

řešení

28-5-4 Časoprostorové mosty

Pro řešení této úlohy existuje poměrně jednoduchý, leč neefektivní algoritmus: Po každém odstranění mostu zkontrolujeme graf prohledáním vrcholů, zda se komponenta obsahující tento most rozpadla na dvě.

Tento způsob určitě funguje, ale je velmi pomalý. Každé odstranění mostu znamená $\mathcal{O}(N)$ operací, tedy celkově tento algoritmus zabere až $O(M \cdot N)$ času. Nejvíce nás zdržuje právě zjišťování, zda odstranění daného mostu způsobilo rozpad komponent. Jak z této neefektivity vybruslíme?

Pojďme se na chvíli podívat na algoritmy pro hledání nejmenších koster, konkrétněji Kruskalův algoritmus, o kterém se lze více dočíst v kuchařce o kostrách.[76] Jeho princip spočívá v tom, že postupně bere nejlevnější hranu. Poté se ptá, zda její přidání do kostry způsobí sloučení dvou komponent do jedné. A to je přesně opačný problém.

K tomuto problému existuje datová struktura *Union-Find*, která poskytuje dvě hlavní operace:

- *Union* spojí dvě komponenty určené vrcholy A a B.
- *Find* pro dva vrcholy A a B odpoví, zda tyto vrcholy leží ve stejné komponentě souvislosti.

O tom, jak tato datová struktura funguje a jak je složitá, si můžete více přečíst ve výše zmíněné kuchařce.

[76] `http://ksp.mff.cuni.cz/viz/kucharky/kostry`

Jak nám ale tato datová struktura pomůže? Dělá přece opak toho, co chceme. Nezoufejme však, jelikož můžeme použít chytrý trik. Vzhledem k tomu, že známe dopředu posloupnost odstranění mostů, můžeme celý postup obrátit. Začneme s grafem podle toho, jak vypadá po odstranění všech mostů a postupně voláme operace *Union* a *Find*.

*Find*em zjistíme, zda most vede uvnitř jedné komponenty. Pokud ano, nemusíme dělat nic. Jinak *Union*em komponenty sloučíme, sečteme jejich energie a součet prohlásíme za energii nové komponenty. Takto postupujeme, dokud nepřidáme všechny mosty. Nakonec vypíšeme výsledek jednotlivých operací pozpátku.

Celkově nás tedy odebrání (přidání) jednoho mostu stojí tolik času, co provedení operací *Union* a *Find*. To je pro dostatečně slušnou implementaci této datové struktury $\mathcal{O}(\log N)$. Dohromady má tedy celý algoritmus časovou složitost $\mathcal{O}(M \log N)$.

Jirka Setnička & Václav Končický

Poznámka M.M.: Složitost $\mathcal{O}(\log N)$ pro Union-Find je velmi velkorysá. Ve skutečnosti i jednoduchá implementace struktury pomocí stromečků, kterou popisujeme v kuchařce, pracuje daleko efektivněji. Složitost lze omezit například funkcí $\mathcal{O}(\log^*)$. Tento „hvězdičkový logaritmus“ je definovaný jako funkce inverzní k takzvané věžové funkci:

$$2 \uparrow 1 = 2, \quad 2 \uparrow 2 = 2^2, \quad 2 \uparrow 3 = 2^{2^2}, \quad 2 \uparrow (k+1) = 2^{2 \uparrow k}.$$

Funkce $\log^* n$ tedy roste velice pomalu, ale ani ona není nejlepší možná. Více už zde neprozradíme a raději vás odkážeme na detaily v kuchařce.

28-5-5 Kalibrace

Máme původně setříděnou posloupnost N čísel, která se zrotovala o neznámý počet pozic, a přesně tento počet bychom chtěli určit. Než se do toho pustíme, podotkněme, že čísla už máme skutečně uložená v paměti. Nemusíme se tedy starat o načtení vstupu (to v praxi odpovídá třeba tomu, že píšeme ne celý program, ale nějakou funkci).

Pravděpodobně nás záhy napadne použít nějakou formu binárního vyhledávání, které je popsané třeba v kuchařce základních algoritmů.[77]

Co s ním ale budeme hledat? Špatně uspořádanou dvojici čísel. Můžeme si rozmyslet, co dostaneme, když zrotujeme posloupnost doprava. Na začátku bude nějaký správně uspořádaný kus, pak špatně uspořádaná dvojice (konkrétně největší prvek následovaný nejmenším) a pak zase správně uspořádaný kus.

Budeme tedy hledat tohle špatné uspořádání, přesněji druhé z těchto špatně uspořádaných čísel. Jeho pozice nutně odpovídá rotaci (počtu pozic, o které byla posloupnost zrotovaná).

[77] `http://ksp.mff.cuni.cz/viz/kucharky/zakladni-algoritmy`

V každém kroku se díváme na úsek. Nejprve porovnáme levý krajní prvek L a pravý krajní prvek R tohoto úseku. Je-li $R > L$, úsek je správně uspořádaný. Pokud víme, že v něm je špatně uspořádaný prvek, musí být hned na začátku a můžeme vrátit pozici L.

V opačném případě se podíváme na prostřední prvek M. Je-li $M > L$, první polovina je správně uspořádaná. Problém tedy bude ve druhé, takže se zanoříme do pravé poloviny. Pro $M < L$ naopak. Dostaneme-li se k úseku velikosti 1, určitě je problematickým prvkem právě ten jediný v něm.

KSP

Voilà, máme algoritmus, který najde problematickou pozici. Jelikož vždy rozdělíme aktuální úsek alespoň na polovinu, skončíme nejpozději po $\log N$ krocích. Časová složitost algoritmu je tak $\mathcal{O}(\log N)$.

Paměťová složitost je $\mathcal{O}(N)$, pokud uvažujeme uloženou posloupnost. Můžeme ale říct, že tak jako zanedbáváme načtení vstupu, zanedbáme uložení posloupnosti, a pak je paměťová složitost konstantní, $\mathcal{O}(1)$.

Karry Burešová

řešení

28-5-6 Sloty na iridium

Rozmísťování iridia do slotů se ukázalo být zapeklitějším problémem, než se zprvu zdálo. Většina z odvážných, kdo se úloze vydali vstříc, vyřešila první triviální vstup, ale přes druhý už se dostal jen jediný z vás (gratulujeme tímto Jirkovi Sejkorovi) a dál už se nedostal vůbec nikdo.

Vstupy se postupně zvětšovaly, ale některé z nich měly i jisté speciální vlastnosti. Třeba pátý vstup měl na pár místech veliké mezery a nebo hned třetí vstup měl všechny dostupné sloty rozmístěné jenom v jedné polovině obvodu. Zvlášť možností využít jen jednu polovinu obvodu se úloha pěkně zjednodušila, protože se pak problém choval jako rozmísťování iridia do slotů na přímce namísto kružnice. Vyřešme si tento jednodušší problém.

Sloty na přímce

Pokud budeme mít sloty na přímce, můžeme bez obav umístit iridium do prvního slotu, protože tím nic nepokazíme (pokud by v nějakém optimálním rozmístění nebylo první iridium v prvním slotu, můžeme ho bez zhoršení minimálních vzdáleností do prvního slotu posunout).

Nyní vyzbrojeni tímto pozorováním zkusme vyřešit úlohu trošku odlišnou – totiž otázku, jestli pro zadanou minimální vzdálenost d lze rozmístit iridium do slotů tak, aby byla tato minimální vzdálenost dodržena. Až vyřešíme tuto úlohu, ukážeme si, jak nám pomůže vyřešit původní problém.

Z pozorování výše víme, že první iridium můžeme umístit do prvního slotu a nic si tím nepokazíme. Další iridium můžeme umístit nejblíže ve vzdálenosti d. Takže přeskočíme všechny bližší sloty a iridium umístíme do prvního slotu, který má vzdálenost větší nebo rovnou d.

A tímto způsobem postupujeme dál, plníme vlastně sloty hladově zleva. Pokud se nám takto povede všechno iridium umístit, zahlásíme úspěch, pokud nám ale dojdou sloty a ještě nám bude zbývat neumístěné iridium, tak víme, že rozmístit s touto vzdáleností nejde (žádné iridium již nelze posunout více doleva, abychom si na konci uvolnili nějaké sloty).

V čase $\mathcal{O}(S)$ tedy umíme lineárním průchodem přes sloty vyřešit tuto podúlohu (připomeňme, že S je počet dostupných slotů). Jak nám to pomůže se řešením původního problému? Můžeme zkusit najít největší vzdálenost, pro kterou se nám ještě iridium povede rozmístit, a toto rozmístění pak vypsat. Hledání největší vzdálenosti můžeme dělat postupným zvětšováním o jedničku a zkoušením, ale to by pro obvod O trvalo až $\mathcal{O}(OS)$ a to je moc dlouho.

Víme, že se tento problém hledání vzdálenosti chová lineárně – když pro nějakou vzdálenost rozmístění iridia existuje, tak existuje i pro všechny menší vzdálenosti, když naopak neexistuje, tak neexistuje ani pro žádnou větší vzdálenost. Můžeme tedy maximální možnou vzdálenost *binárně vyhledat*.

Budeme si držet minimum a maximum zkoušeného rozsahu (minimum bude na začátku 1, maximum bude obvod dělený počtem kousků iridia). V každém kroku se podíváme na vzdálenost $d = \frac{min+max}{2}$ a ozkoušíme rozmístit iridium s touto minimální vzdáleností:

- Pokud se iridium povede rozmístit $\rightarrow$ hledaná vzdálenost je větší nebo rovna d, nastavíme $min = d$
- Pokud se iridium nepovede rozmístit $\rightarrow$ hledaná vzdálenost je menší než d, nastavíme $max = d - 1$

Takto pokračujeme, dokud nedojdeme na krok velikosti jedna. Pak již máme vzdálenost určenou jednoznačně. Binárním vyhledáváním uděláme $\mathcal{O}(\log O)$ kroků, takže celkově dosáhneme času $\mathcal{O}(S \log O)$.

Sloty na obvodu kruhu

Při rozmisťování iridia do slotů na obvodu kruhu použijeme úplně stejný postup, jen se už nemůžeme spolehnout na pozorování o umístění prvního iridia do prvního slotu. Nyní totiž jde i o vzdálenost prvního a posledního obsazeného slotu. Může být například výhodné prvních pár slotů přeskočit, aby nám vzdálenost vyšla.

Jak si s tím poradit? Pokud nám při zkoušení, jestli iridium umíme rozmístit se vzdáleností d, dojdou sloty před rozmístěním všech kousků, můžeme rovnou oznámit neúspěch. Pokud se nám naopak iridium rozmístit povede a vzdálenost mezi prvním a posledním obsazeným slotem je dostatečně velká, můžeme rovnou oznámit úspěch. To byly ty jednodušší případy.

Složitější je, pokud sice všechno iridium rozmístíme, ale vzdálenost mezi prvním a posledním bude příliš malá. Pak můžeme zkusit první posunout po obvodu dál tak, abychom tuto vzdálenost dostatečně zvětšili. Pokud tím neporušíme minimální vzdálenost k dalšímu obsazenému slotu, uspěli jsme, jinak to

samé opakujeme (opět posuneme další iridium tak daleko, aby byla dodržená minimální vzdálenost a opakujeme).

Toto posouvání zastavíme ve chvíli, kdy se nám rozmístění buď povede opravit (v tom případě oznámíme úspěch), nebo když se pokusíme nějaké iridium umístit opět do prvního slotu. V tom případě totiž opět dostáváme stejnou situaci, jako při prvním rozmístění, a zacyklili jsme se bez nalezení fungujícího rozmístění.

KSP

Na implementaci tohoto postupu se můžete podívat v ukázkovém programu, nyní se zamysleme nad časovou složitostí. Hlavní pozorování je, že se při posouvání iridia mezi sloty pokusíme vložit iridium do každého ze slotů maximálně dvakrát. Při druhém pokusu o vložení do stejného slotu totiž dojde k tomu, že se i všechny zbylé kousky rozmístí tak, jak byly, a dojde k zacyklení (při kterém se zastavíme s neúspěchem).

Čas jednoho kroku jsme si tak oproti jednodušší verzi na přímce zhoršili jen konstantně a vnější část s binárním vyhledáváním zůstává stejná. Celkově tak dosáhneme časové složitosti $\mathcal{O}(S \log O)$.

řešení

Program (C):

`http://ksp.mff.cuni.cz/viz/28-5-6.c`

Jirka Setnička

28-5-7 Tajemná operace

Zopakujme si zadání: Dostaneme nějakou magickou operaci $\otimes$ a posloupnost $a_1, \ldots, a_n$. Chceme si něco předpočítat, abychom uměli vyhodnocovat intervalové dotazy typu $a_i \otimes a_{i+1} \otimes \ldots \otimes a_j$ a stačilo nám jediné zavolání operace $\otimes$.

Pochopitelně si můžeme předpočítat výsledky pro úplně všechny intervaly $a_i, \ldots, a_j$. Pak při odpovídání na dotazy dokonce nemusíme $\otimes$ volat vůbec. Ovšem předvýpočet trvá čas $\mathcal{O}(n^2)$, což nám nepromine sebetrpělivější uživatel, natož pak přísné elektronické oko CodExovo.

Tak zkusíme sáhnout po různých standardních technikách, jako je třeba rozklad na bloky nebo intervalové stromy. Z těch také použitelné řešení nekápne: na kombinování bloků nebo podstromů sice postačí malý počet volání $\otimes$, ale rozhodně více než povolené jedno. Přesto z myšlenky rekurzivního rozkladu na podproblémy něco vykřešeme. Tedy poslyšte…

Rekurzivní řešení

Učiníme myšlenkový pokus: zadanou posloupnost rozdělíme přibližně uprostřed na *levou polovinu* $a_1, \ldots, a_s$ a *pravou polovinu* $a_{s+1}, \ldots, a_n$. Dotazy rozdělíme na dva druhy podle toho, zda leží přes střed, či nikoliv.

Pokud interval $a_i, \ldots, a_j$ jde přes střed, skládá se ze suffixu levé poloviny (to je nějaký interval $a_i, a_{i+1}, \ldots, a_s$) a zprefixu té pravé ($a_{s+1}, \ldots, a_j$). Předpočí-

táme-li si tedy výsledky pro všechny suffixy levé poloviny a všechny prefixy pravé, umíme je na jedno zavolání $\otimes$ složit do výsledku celého dotazu.

A pokud interval neleží přes střed, přesuneme se do levé či pravé poloviny a tam rekurzivně aplikujeme tentýž postup. Pojďme spočítat, jak je to efektivní.

Označme $P(n)$ časovou složitost předvýpočtu pro posloupnost délky n. Jistě platí, že $P(n) = 2 \cdot P(n/2) + \mathcal{O}(n)$, neboť pro interval délky n v čase $\mathcal{O}(n)$ zpracujeme prefixy a suffixy a rekurzivně předpočítáme zvlášť levou a pravou polovinu. Tuto rovnici pro $P(n)$ můžeme vyřešit třeba analýzou stromu rekurze. Raději použijeme prostý trik: všimneme si, že úplně stejně se chová známý algoritmus MergeSort (třídění sléváním) – také spotřebuje lineární čas a pak rekurzivně zpracuje dva poloviční podproblémy. MergeSort běží v čase $\mathcal{O}(n \log n)$, takže náš předvýpočet také.

Vyhodnocování dotazu buďto skončí ihned (jde-li dotaz přes střed), nebo se rekurzivně zavolá na jednu z polovin. Během $\mathcal{O}(\log n)$ kroků tedy rekurze musí skončit. Jelikož každý krok trvá konstantně dlouho, časová složitost dotazu činí celkem $\mathcal{O}(\log n)$. Podmínku na nejvýše jedno použití $\otimes$ jistě splňujeme.

Program (C):

`http://ksp.mff.cuni.cz/viz/28-5-7-fast.c`

Rychlejší řešení

Nevýhodou předchozího řešení je, že vyhodnocení dotazu sice volá $\otimes$ jenom jednou, ale kromě toho tráví logaritmický čas nalezením správného podintervalu, přes jehož střed zadaný dotaz leží. To lze s trochou šikovnosti zvládnout i v konstantním čase.

Předně si představujme, že délka posloupnosti je mocnina dvojky a že prvky očíslujeme od nuly: $a_0, \ldots, a_{n-1}$. Pokud se na indexy prvků podíváme ve dvojkové soustavě, v levé polovině všechny začínají nulou, v pravé jedničkou. Dotaz tedy leží přes střed právě tehdy, pokud se jeho levý a pravý okraj liší v nejvyšším bitu.

Rekurzivní rozklad intervalů můžeme elegantně popsat binárním stromem, který na h-té hladině testuje h-tý nejvyšší bit čísla. Nejvyšší interval, v němž jde dotaz přes střed, pak odpovídá nejvyššímu bitu, v němž se okraje dotazu i a j liší. To je nejvyšší jedničkový bit v čísle $i \operatorname{XOR} j$. Pro zjišťování, kde leží nejvyšší jednička, si přitom můžeme snadno předpočítat tabulku.

Celkově tedy v konstantním čase spočítáme $i \operatorname{XOR} j$, pomocí tabulky zjistíme hladinu h stromu rekurze, kam se chceme podívat. A nejvyšších h bitů čísel i a j nám řekne, kolikátý vrchol zleva nás na dané hladině zajímá. Vrcholy tedy můžeme mít uložené v poli a indexovat je též v konstantním čase.

Program (C):

`http://ksp.mff.cuni.cz/viz/28-5-7-faster.c`

Rozklad na bloky

Pro zajímavost načrtneme ještě jedno řešení pomocí rozkladu na bloky. To je nejspíš jednodušší na vymyšlení, ale o něco pomalejší.

Posloupnost rozřežeme na $\sqrt{n}$ bloků velikosti $\sqrt{n}$. Dotazy, které leží celé uvnitř jednoho bloku, předpočítáme všechny. Je jich $\mathcal{O}(n)$ na blok, celkem tedy $\mathcal{O}(n^{3/2})$.

Ostatní dotazy chceme skládat ze suffixu prvního bloku, nějakých celých bloků a prefixu posledního bloku. Suffixy a prefixy si můžeme předpočítat (je jich celkem $\mathcal{O}(n)$). Nabízí se předpočítat si také všechny intervaly celých bloků (těch je též $\mathcal{O}(n)$), ale s tím narazíme: na zodpovězení dotazu bychom potřebovali složit tři mezivýsledky, tedy zavolat operaci $\otimes$ dvakrát, což je moc.

KSP

Proto si místo obyčejných suffixů předpočítáme všechny intervaly, které vzniknou rozšířením suffixu o nějaký počet celých bloků. Každý suffix má $\sqrt{n}$ takových rozšíření, úplně všechny suffixy pak $\mathcal{O}(n^{3/2})$. Pak stačí skládat rozšířený suffix s obyčejným prefixem.

Získali jsme tedy řešení s předvýpočtem v čase $\mathcal{O}(n^{3/2})$ a dotazem v čase $\mathcal{O}(1)$ s jedním voláním operace $\otimes$.

řešení

Cesta jde pořád dál a dál...

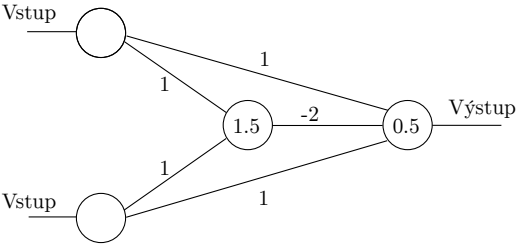
Na závěr ještě dodejme, že kdybychom povolili větší (ale stále konstantní) počet použití $\otimes$, úloha by byla mnohem zajímavější, ovšem také mnohem náročnější. Například pro 3 volání $\otimes$ stačí předzpracování v čase $\mathcal{O}(n \log^* n)$, kde $\log^*$ je funkce zmíněná v řešení čtvrté úlohy.

Martin „Medvěd“ Mareš

28-5-8 Neuronové sítě

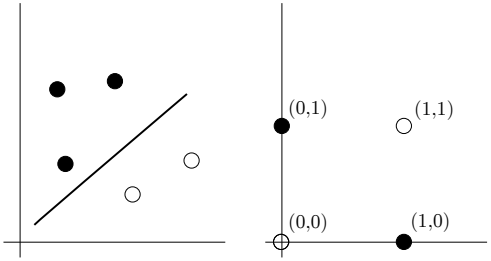
Úkol 1

Cílem první úlohy je nalézt co nejmenší neuronovou síť modelující XOR. Překvapilo nás, že žádné ze zaslaných řešení nebylo optimální, takže tady je:



Důkaz minimality provedeme sporem. Kdyby byla síť ještě menší, měla by jen dva vstupní a jeden výstupní neuron bez skryté vrstvy. Jeden neuron dokáže, stejně jako perceptron, v dvourozměrném vstupním prostoru oddělovat dvě

skupiny vzorů přímkou, jak jsme psali v minulém dílu seriálu. U funkce XOR ale vstupní vzory přímkou oddělit nelze, říkáme, že nejsou *lineárně separovatelné*. To ilustrují následující dva grafy lineárně separovatelné množiny a množiny možných vstupů xoru.



Úkol 2

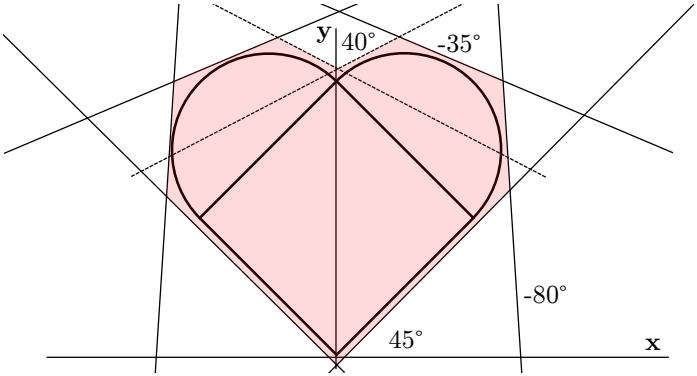
Druhým úkolem bylo implementovat neuronovou síť pro klasifikaci kosatců. Na následujícím odkazu si můžete stáhnout vzorové řešení v jazyce Python. Náš program využívá neuronovou síť o 3 skrytých neuronech v jedné skryté vrstvě a dosahuje přesnosti klasifikace na validační množině kolem 98% až 100%.

Program (Python):

`http://ksp.mff.cuni.cz/viz/28-5-8-2.py`

Úkol 3

Poslední úlohu bylo možné řešit pomocí zpětné propagace nebo navrhnout síť pomocí analytické geometrie. Jelikož zpětnou propagací jsme řešili minulou úlohu a používala ji i všechna vaše řešení, podíváme se na geometrický přístup. Kolem srdce zkonstruujeme „obal“ z několik přímek, které budou oddělovat body uvnitř obrazce od ostatních. Tyto přímky pak převedeme na neurony, které budou dělat totéž v rámci neuronové sítě.



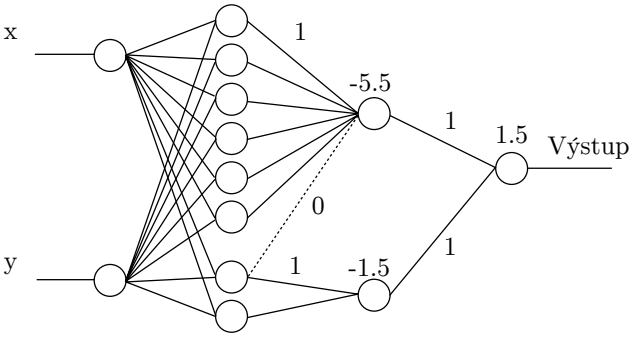
Z analytické geometrie víme, že přímka má rovnici $y = ax + b$, také víme, že a se dá spočítat pomocí úhlu α, který přímka svírá s osou x: $a = \tan(\alpha)$. Hodnotu b pak můžeme dopočítat dosazením nějakého bodu na přímce do rovnice přímky. Díky tomu můžeme spočítat rovnice všech osmi v obrázku zakreslených přímek. Ukažme si to na pravé spodní přímce, která má sklon 45° a leží na ní bod $[0; 0]$:

$$\begin{aligned} a &= \tan(\alpha) = \tan(45°) = 1 \\ y &= x + b \\ 0 &= 0 + b \\ b &= 0 \end{aligned}$$

Tuto rovnici přímky nyní můžeme upravit do tvaru $0 = ax - 1y + b$ a porovnáním s rovnicí neuronu $0 = \xi = w_1 x + w_2 y + \delta$ z ní určit jeho váhy w a práh δ. Pro naši první přímku $y = x + 0$ to bude $w = [1; -1]$ a $\delta = 0$. Všimněte si, že nemáme zajištěno, na které straně přímky bude potenciál neuronu kladný a na které záporný. V tomto případě bude kladný v oblasti mimo srdce. Pokud bychom chtěli hodnoty prohodit, můžeme vynásobit uvedenou rovnici neuronu -1, čímž dostaneme váhy a práh také vynásobený -1.

Tímto postupem můžeme určit váhy a prahy neuronů všech osmi přímek. Řekněme, že tedy máme neurony s aktivační funkcí signum, které mají pro body směrem dovnitř obrazce kladný výstup +1. Bod je uvnitř srdce tehdy, když všechny neurony mají kladný výstup, kromě dvou neuronů, jejichž přímky jsou na obrázku vykresleny přerušovanou čarou. Z těch stačí jeden s výstupem +1.

Abychom tuto podmínku realizovali neurony, pořídíme si vrstevnatou neuronovou síť jako na následujícím obrázku. Neurony přímek jsou použity v první skryté vrstvě. Aktivační funkce všech neuronů je signum. Nezakreslené hrany můžeme v síti formálně ponechat s nastavenou váhou 0 (ilustrováno přerušovanou čarou).



Jan Škoda

Pořadí řešitelů KSP

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
0.				40	300.0
1.	Richard Hladík	GOAMarLaz	3	28	236.6
2.	Pavel Turek	GTomkovaOL	3	29	223.8
3.	Jiří Sejkora	GVoděraPH	4	24	220.8
4.	Jakub Pelc	G UherBrod	2	23	211.5
5.	Václav Volhejn	GKepleraPH	3	25	190.0
6.	Jan Bouček	GKepleraPH	3	18	169.0
7.	Jonáš Fiala	GJungmanLT	3	18	120.0
8.	Stanislav Lukeš	GPísnickáPH	3	13	100.2
9.	Daniel Herman	GŠKo	3	17	96.8
10.	Michal Töpfer	G DrJPekMB	3	16	92.1
11.	Václav Pavlíček	ZŠ Ždírec nD	0	18	81.0
12.	Leonard Mentzl	GŘíč	3	9	76.9
13.	Petr Chmel	G_Kralupy	3	11	75.1
14.	Josef Gajdůšek	SŠKKamPard	3	10	71.1
15.	Jakub Tětek	Dollar Ac	2	6	63.0
16.	Vojtěch Lukeš	GPikaPL	4	8	62.9
17.	Pavel Turinský	G Brandýs	3	10	62.5
18.	Lukáš Rozsypal	GÚstavníPH	3	10	59.6
19.	Přemysl Šťastný	GŽamberk	3	11	58.4
20.	Václav Šraier	GČeskoliPH	3	10	55.9
21.	Jakub Suchánek	GOpatovPHA	2	7	54.0
22.	Lukáš Vlček	GMikulášPL	2	10	48.9
23.	Jiří Löffelmann	GLitoměřPH	2	10	48.5
24.	Jakub Dobrý	GMikulášPL	2	8	41.5
25.	Jiří Vozár	G UherBrod	4	8	40.6
26.	Miroslav Hrabal	GTomkovaOL	2	5	40.2
27.	Viktor Fukala	GKepleraPH	−1	4	39.0
28.	Ján Chudý	GŽilina	4	6	38.5
29.	Vanda Hendrychová	GHeyrovPH	4	8	37.6
30.	David Žáček	GZborovPH	3	5	36.4
31.	Vojtěch Hudec	G_ČTřebová	2	11	36.0
32.	Ondrej Pudiš	GŽilina	4	6	35.0
33.	Michal Kodad	ZŠJílovsPH	0	9	31.1
34.–35.	Vladimír Bartovic	G AM Trnava	4	7	29.2
	Jakub Lukeš	GNAlejíPH	3	7	29.2
36.	Tomáš Domes	MendelG_OP	3	5	26.4
37.	Roman Beňo	GJHroncaBA	3	3	25.6
38.	Zdenko Čepan	GPartizans	3	9	22.6

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
39.	Michal Jireš	GRNK	1	3	22.5
40.	David Ucháč	eduSOŠ PA	3	4	21.9
41.	Marco Souza de Joode	GNadŠtolPH	−1	4	20.8
42.	Sam Friedlaender	GKepleraPH	−1	3	20.3
43.	František Kmječ	G Brandýs	0	3	19.8
44.	Jan Pokorný	G Bučovice	4	2	17.7
45.	Filip Geib	G MMH LM	2	5	17.1
46.	Alexej Popovič	SlovanGOL	4	3	16.1
47.	Alena Tesařová	GVídeňskBO	4	6	15.5
48.	Jan Kaifer	GČesBrod	0	3	15.4
49.	Josef Pospíšil	GÚstavníPH	2	2	12.6
50.	Petr Gebauer	GMělník	2	2	10.6
51.–54.	Jan Gocník	GJŠkodyPŘ	4	1	10.0
	Martin Kučera	GNeratov	4	2	10.0
	Jan Priessnitz	GJarošeBO	3	1	10.0
	Filip Šohajek	GUHradiště	−1	1	10.0
55.	Jan Neumann	GNAlejíPH	2	1	9.7
56.	Michal Rickwood	G_ČTřebová	2	3	8.3
57.–60.	David Blažek	SPŠÚžlabPH	3	1	8.0
	Lucie Kubíčková	GFXŠaldyLI	2	1	8.0
	Antonín Prantl	G Strakon	3	3	8.0
	Zuzana Svobodová	G FrýdlNOs	4	1	8.0
61.	Eliška Vlčinská	GHladnov	1	1	7.7
62.	Kristýna Moudrá	GÚstavníPH	3	2	6.9
63.	Adam Husník	GArabskáPH	2	1	6.0
64.	Lukáš Mičan	GČeskáČB	2	2	5.8
65.–66.	Lukáš Beneda	GČeskáČB	2	3	5.0
	Anna Šebestíková	GČeskáČB	1	2	5.0
67.	Petra Štefaníková	GOlgHavl	4	1	4.8
68.	Michael Bausano	GTěš	4	1	4.5
69.	Jakub Matěna	GČeskoliPH	4	1	4.3
70.	Martin Zoula	GNadKavaPH	4	1	3.7
71.	Ondřej Borýsek	GJarošeBO	3	2	3.3
72.	Jiří Muller	G_Roudnice	3	1	2.5
73.	David Nápravník	GLitoměřPH	3	1	2.2
74.	Peter Matta	G KošiceS	4	1	1.0

Jiří Setnička a kolektiv

Korespondenční seminář z programování XXVIII. ročník

Vydal MatfyzPress
vydavatelství Matematicko-fyzikální fakulty Univerzity Karlovy
Sokolovská 83, 186 75 Praha 8
jako svou 524. publikaci.

Vytisklo Reprostředisko MFF UK
Sokolovská 83, 186 75 Praha 8.

Publikace neprošla recenzním ani lektorským řízením.
Nakladatelství neodpovídá za kvalitu a obsah textu.
Vydáno pro vnitřní potřebu MFF UK.
Publikace není určena k prodeji.

Autoři a opravující úlohy:
Jana Bátoryová, Karolína Burešová, Zuzana Drázdová, Jan Hadrava
Ondřej Hlavatý, Petr Houška, Jan Knížek, Václav Končický,
Dominik Macháček, Martin Mareš, Jakub Maroušek, Jan Matějka
Michael Pokorný, Pavol Rohár, Vojtěch Sejkora, Jiří Setnička
Dominik Smrž, Martin Šerý, Jan Škoda, Martin Španěl
Filip Štědronský, Karel Tesař, Kateřina Zákravská

Autoři příběhů v zadání:
Karel Tesař, Jakub Maroušek, Jiří Setnička

Autoři seriálu:
Karel Tesař, Michael Pokorný, Jan Škoda

TEX-ová makra pro sazbu ročenky vytvořili Martin Mareš, Jan Matějka, Radim Cajzl a Jiří Setnička.

S jejich pomocí ročenku vysázel Jiří Setnička.

Obrázek na obálce nakreslila Petra Pelikánová.

Sazba byla provedena písmem Computer Modern v programu TEX.

První vydání
Praha 2016

ISBN 978-80-7378-330-3

ISBN 978-80-7378-330-3