	řešitel	*škola*	*ročník*	*sérií*	*4-1*	*4-2*	*4-3*	*4-4*	*4-5*	*4-6*	*4-7*	*4-8*	*série*	*celkem*
56.–59.	David Blažek	SPŠÚžlabPH	3	1									0,0	8,0
	Lucie Kubíčková	GFXŠaldyLI	2	1									0,0	8,0
	Antonín Prantl	G Strakon	3	1									0,0	8,0
	Zuzana Svobodová	G FrýdlNOs	4	2									0,0	8,0
60.	Eliška Vlčinská	GHladnov	1	1			5						7,7	7,7
61.	Kristýna Moudrá	GÚstavníPH	3	1	2	1							6,9	6,9
62.	Adam Husník	GArabskáPH	2	1									0,0	6,0
63.	Lukáš Mičan	GČeskáČB	2	1									0,0	5,8
64.–65.	Lukáš Beneda	GČeskáČB	2	1	1	1	0						5,0	5,0
	Anna Šebestíková	GČeskáČB	1	1	1	1							5,0	5,0
66.	Petra Štefaníková	GOlgHavl	4	1		3							4,8	4,8
67.	Michael Bausano	GTěš	4	1									0,0	4,5
68.	Jakub Matěna	GČeskoliPH	4	4									0,0	4,3
69.	Martin Zoula	GNadKavaPH	4	4									0,0	3,7
70.	Ondřej Borýsek	GJarošeBO	3	2									0,0	3,3
71.	Jiří Muller	G_Roudnice	3	1									0,0	2,5
72.	David Nápravník	GLitoměřPH	3	1									0,0	2,2
73.	Peter Matta	G KošiceS	4	1									0,0	1,0

KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.

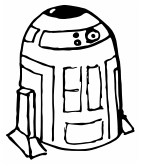
Webové stránky: *https://ksp.mff.cuni.cz/*
E-mail: *ksp@mff.cuni.cz*
Diskusní fórum: *https://ksp.mff.cuni.cz/forum/*

Chcete-li s námi komunikovat bezpečně, můžete si ověřit náš HTTPS certifikát – jeho SHA1 fingerprint je: `E9:DB:EE:C6:62:BC:14:DE:09:E4:E8:97:DC:36:0E:87:B3:50:B0:01`.

Korespondenční Seminář z Programování

28. ročník **KSP** **Květen 2016**

Milí řešitelé a řešitelky!



Poslední letošní série se vám právě dostává do rukou. Už se určitě těšíte na to, jak brzy začnou prázdniny a budete si moci dát od školy na chvíli pauzu, ale přesto si schovejte ještě pár chvil i pro řešení úloh z tohoto letáku. Vždyť vás za vyřešení úloh a získání dostatečného počtu bodů můžeme **pozvat na Podzimní soustředění**, a to se vyplatí!

Navíc připomínáme, že každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme propisku, blok, tužku, a možná i něco navíc. Tak si to poslední sérií nezkažte.

Termín série: **Pondělí 13. června 2016 v 8:00 SELČ** (CodEx má termín stejný)
Odevzdávání: Přes web na adrese *https://ksp.mff.cuni.cz/submit/*
Odměna série: **Sladkou odměnu** pošleme každému, kdo **získá z alespoň šesti úloh alespoň dva body**.

Pátá série dvacátého osmého ročníku KSP

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.

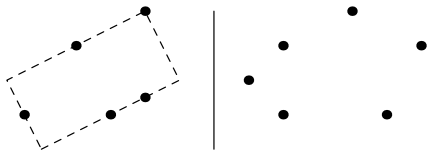
Tohle měl být první experiment, kdy časem pošlou člověka, konkrétně Willa. Zatím vyslali jen několik automatických sond. Ale asi selhal časoprostorový generátor a jádro generátoru, ve kterém se teď Will nacházel, se už vůbec nenacházelo ve stejném čase jako zbytek jeho týmu. Nikdo vlastně nevěděl, kde (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.



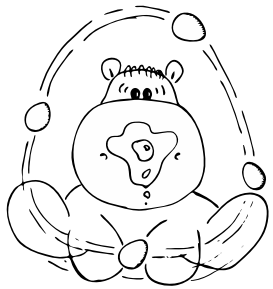
Výzkumníkům se sice povedlo zjistit, že je místnost jenom fázově posunutá a leží na okraji otevřeného časového víru, ale v záchraně Willa se nedostali o nic dál. Navíc skomírající generátor už dlouho časový vír pod kontrolou neudrží a nikdo nevěděl, jak velké škody v čase by jeho vymknutí se kontrole 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.



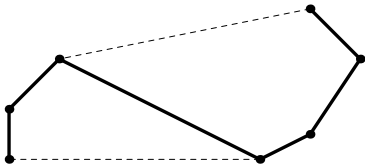
Chlapík se za ním rozeběhl a byl asi lepší běžec než Will. Pomalu ho začal dohánět, a tak Will za běhu hrábl po minipočítači na ruce. Skok časem dělat nechtěl, ale několika místními přesuny by ho setřást mohl. Stejně je chtěl použít k prozkoumání většího území.

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.

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.

* * *

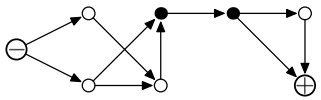
A tak se stalo, že se Will ocitl v podzemí nějaké banky 20. století před velkým sejfem, ve kterém měl být dostatečně velký kus čistého iridia získaný po dopadu nějakého meteoritu. Jenže trezor měl dost propracovaný alarm.

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. Naraził 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

Výsledková listina čtvrté série dvacátého osmého ročníku KSP

	řešitel	škola	ročník	sérií	4-1	4-2	4-3	4-4	4-5	4-6	4-7	4-8	série	celkem
0.					9	8	10	12	11	9	11	20	64,0	242,0
1.	Jakub Pelc	G UherBrod	2	4			10	12		9	8	20	58,7	199,4
2.	Richard Hladík	GOAMarLaz	3	19	7		10	12		9	11	20	62,0	196,0
3.	Jiří Sejkora	GVoděraPH	4	7		8	10	12			11	19	60,3	191,8
4.	Pavel Turek	GTomkovaOL	3	4	8	8	10	12	5				46,1	182,6
5.	Václav Volhejn	GKepleraPH	3	18									0,0	178,0
6.	Jan Bouček	GKepleraPH	3	8		8						19,5	27,7	169,0
7.	Stanislav Lukeš	GPísnickáPH	3	10				8			11		19,0	100,2
8.	Jonáš Fiala	GJungmanLT	3	4	3	1	0	12					19,0	97,5
9.	Daniel Herman	GŠKo	3	3	6	6	10	4	4	1	10	12	52,2	96,8
10.	Michal Töpfer	G DrJPekMB	3	9			10					2	11,6	84,1
11.	Leonard Mentzl	GŘíč	3	2									0,0	76,9
12.	Petr Chmel	G_Kralupy	3	4	2	8		0				8	24,1	75,1
13.	Josef Gajdůšek	SŠKKamPard	3	3									0,0	71,1
14.	Václav Pavlíček	ZŠ Ždírec nD	0	4	3	4		0	3				16,0	67,8
15.	Jakub Tětek	Dollar Ac	2	9						9	10		19,0	63,0
16.	Vojtěch Lukeš	GPikaPL	4	2									0,0	62,9
17.	Lukáš Rozsypal	GÚstavníPH	3	3	2	8	5				8		29,3	59,6
18.	Pavel Turinský	G Brandýs	3	8	7	8							15,2	51,5
19.	Přemysl Šťastný	GŽamberk	3	11	3	8	10	0					20,5	50,5
20.	Lukáš Vlček	GMikulášPL	2	3									0,0	48,9
21.	Jiří Löffelmann	GLitoměřPH	2	2	3	8	10	0		2			26,6	46,6
22.	Václav Šraier	GČeskoliPH	3	8			10	0					10,0	42,9
23.	Jakub Dobrý	GMikulášPL	2	3									0,0	41,5
24.	Jiří Vozár	G UherBrod	4	8		4					3	6	15,4	40,6
25.	Miroslav Hrabal	GTomkovaOL	2	2									0,0	40,2
26.	Viktor Fukala	GKepleraPH	−1	1	2	3,5	6					20	39,0	39,0
27.	Ján Chudý	GŽilina	4	1									0,0	38,5
28.	David Žáček	GZborovPH	3	3	3	8	10		3				26,4	36,4
29.	Ondrej Pudiš	GŽilina	4	1									0,0	35,0
30.	Michal Kodad	ZŠJílovsPH	0	4		4		0					5,6	31,1
31.–33.	Vladimír Bartovic	G AM Trnava	4	2									0,0	29,2
	Vanda Hendrychová	GHeyrovPH	4	1									0,0	29,2
	Jakub Lukeš	GNAlejíPH	3	4									0,0	29,2
34.	Jakub Suchánek	GOpatovPHA	2	1	6				3			8	27,5	27,5
35.	Roman Beňo	GJHroncaBA	3	2									0,0	25,6
36.	Vojtěch Hudec	G_ČTřebová	2	1	1	1	9	0	3	0,5	2		23,6	23,6
37.	Zdenko Čepan	GPartizans	3	2									0,0	22,6
38.	Michal Jireš	GRNK	1	1									0,0	22,5
39.	David Ucháč	eduSOŠ PA	3	2									0,0	21,9
40.	Marco Souza de Joode	GNadŠtolPH	−1	2									0,0	20,8
41.	Sam Friedlaender	GKepleraPH	−1	2									0,0	20,3
42.	František Kmječ	G Brandýs	0	2	2	7							11,8	19,8
43.	Jan Pokorný	G Bučovice	4	8									0,0	17,7
44.	Filip Geib	G MMH LM	2	2									0,0	17,1
45.	Alexej Popovič	SlovanGOL	4	2									0,0	16,1
46.	Alena Tesařová	GVídeňskBO	4	2									0,0	15,5
47.	Jan Kaifer	GČesBrod	0	2									0,0	15,4
48.	Josef Pospíšil	GÚstavníPH	2	1		1						7	12,6	12,6
49.	Petr Gebauer	GMělník	2	1									0,0	10,6
50.–53.	Jan Gocník	GJŠkodyPŘ	4	4									0,0	10,0
	Martin Kučera	GNeratov	4	1			10	0					10,0	10,0
	Jan Priessnitz	GJarošeBO	3	1									0,0	10,0
	Filip Šohajek	GUHradiště	−1	1									0,0	10,0
54.	Jan Neumann	GNAlejíPH	2	1									0,0	9,7
55.	Michal Rickwood	G_ČTřebová	2	2	2	1							6,0	8,3

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:

$$\mathrm{P}[C = s|P_s = 1, P_v = -1, P_g = 1] =$$

$$= \mathrm{P}[C = s] \cdot \frac{\mathrm{P}[P_s = 1|C = s]}{\mathrm{P}[P_s = 1]} \cdot \frac{\mathrm{P}[P_v = -1|C = s]}{\mathrm{P}[P_v = -1]} \cdot$$

$$\cdot \frac{\mathrm{P}[P_g = 1|C = s]}{\mathrm{P}[P_g = 1]}$$

$\mathrm{P}[C = s]$ 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 mulu 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

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.

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.

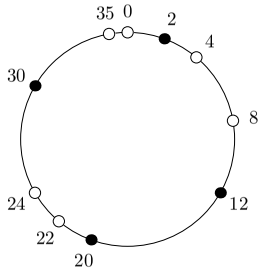
Ukázkový vstup:

```
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.

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

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.

Tato úloha je praktická a řeší se ve vyhodnocovacím systému CodEx.[1] Přesný formát vstupu a výstupu, povolené jazyky a další technické informace jsou uvedeny v CodExu přímo u úlohy.

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

vání 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.

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 „Meduněd“ Mareš & Honza Knížek

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:

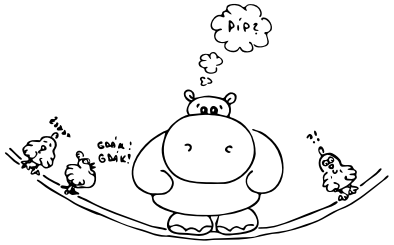
$$\mathrm{P}[C = s | P_s = 1, P_v = -1, P_g = 1]$$

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.



Rýsuje se nám tady základ algoritmu.

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,[6] 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.[7]

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 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,[8] 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.

Pokud bychom dopředu znali minimální K optimální cesty (označme toto optimum D), mohli bychom pustit prohledá-

[6] `http://ksp.mff.cuni.cz/viz/27-2-1`
[7] `http://ksp.mff.cuni.cz/viz/kucharky/trideni`
[8] `http://ksp.mff.cuni.cz/viz/28-2-1/reseni`

28-5-8 Neuronové sítě — 14 bodů

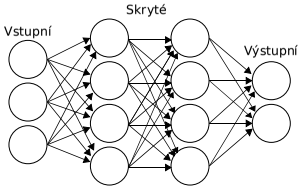
V minulém dílu seriálu jsme se zabývali modelem biologického neuronu – perceptronem. Ukázali jsme si, jak takový model „naučit“ a použít k jednoduché klasifikaci, tedy roztřídění vstupů do několika skupin. V zadané úloze jste nakonec bojovali s tím, že jeden perceptron dokáže klasifikovat jen do dvou skupin.

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:

$$E = \frac{1}{2} \sum_{i}^{data} \sum_{j}^{výstupy} (y_{j,i} - d_{j,i})^2$$

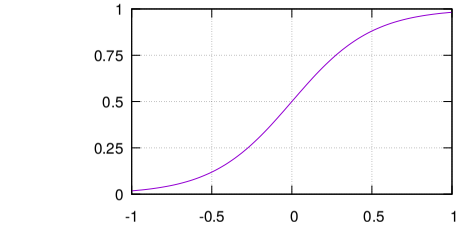
Kde $y_{j,i}$ je výstup j-tého neuronu při i-tém vstupu sítě a $d_{j,i}$ je požadovaný výstup ve stejném případě. Na předpisu této funkce je zajímavé, že chyba (nebo také odchylka) každého neuronu je umocněna na druhou, aby se zdůraznily velké chyby a zanedbaly malé. Sumy pak sečtou chyby ze všech neuronů 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:

$$w_{ij}(t+1) = w_{ij}(t) + \alpha\delta_j y_i$$

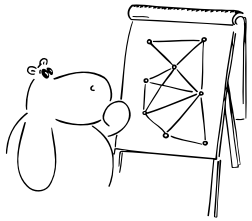
kde δ_j je chyba neuronu j. Ta se dá přímo spočítat pro výstupní vrstvu:

$$\delta_j = (d_j - y_j)\lambda y_j(1 - y_j)$$

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.

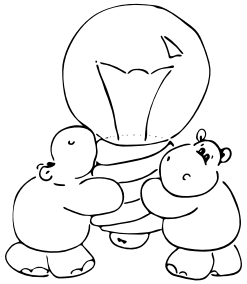
Pokud síť používáme ke klasifikaci do k skupin, často můžeme dosáhnout lepších výsledků, když místo jednoho výstupu sítě, který by udával číslo skupiny $1, .., k$, použijeme k výstupních neuronů. i-tá skupina se pak reprezentuje tak, že i-tý neuron má hodnotu 1 a ostatní 0 (nebo v blízkosti těchto hodnot).

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 $\alpha \in [0.2; 1]$.



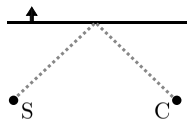
Zbývá vyřešit volbu počátečních vah. Ty bychom chtěli nastavit tak, aby průměrný neuron reagoval na vstupy v rozsahu $[0; 1]$ opět potenciálem v rozsahu $[0; 1]$.

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.

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:

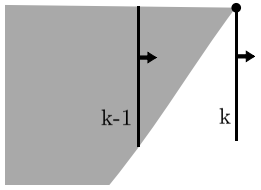


Pro účely našeho řešení uvažme, že úsečka tvořící branku (respektive přímka, na které leží) dělí celé hřiště na dvě poloroviny. Projet branku pak znamená, že míček se nejprve nachází v první a pak, skrz úsečku, přejde do druhé poloroviny. V obou stavech se míček alespoň na chvíli musí nacházet ostře, 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?

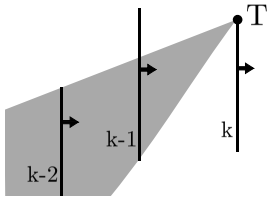
Považujme teď startovní a cílový bod za branky nulové délky. Algoritmus projde všechny tyčky branek a pro každou z nich zjistí, odkud k ní může přímým vrhem (tedy bez ohýbání) přiletět míček. Mějme tyčku T, která leží u K-té branky. Vezměme obě tyčky u branky $K - 1$ a veďme jimi z T polopřímky. Výseč mezi nimi odpovídá prostoru, ve kterém můžeme do T odpálit míček letící skrz branku $K - 1$:



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

[5] `http://ksp.mff.cuni.cz/viz/kucharky/geometrie`

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).

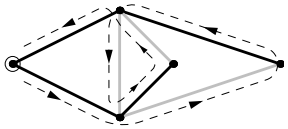
Pár poznámek k implementaci: výseče je možné v programu reprezentovat prostě vrcholem a dvojicí úhlů. Vzhledem k tomu, že nás nezajímá, jak přesně jsou tyto úhly velké, ale stačí nám je jen umět porovnávat, můžeme místo úhlů pracovat se *směrnicemi*.[4] Tím si ušetříme počítání s goniometrickými funkcemi. Pro určení toho, ve které polorovině od dané branky leží nějaký bod, můžeme použít techniky popsané v naší geometrické kuchařce.[5] Podrobněji ve vzorovém programu.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/28-4-5.py`

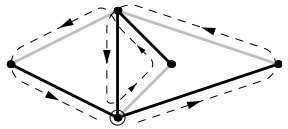
A na závěr doplnění k naší omluvě: Pokud by u jakékoliv úlohy vypadalo, že vás nutíme používat podobně chlupaté postupy, jako nekonečně malé kroky, raději se zeptejte na fóru. Orgové sice často jsou docela osobití lidé, ale takové šílenosti schválně nezadávají (případně si je nechávají v záloze na soustředění).

Kuba Maroušek

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.

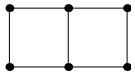
Nejprve však trocha názvosloví: nově přidaným vrcholům a hranám budeme říkat *virtuální*, původním *skutečné*. *Skutečný stupeň* vrcholu v je stupeň, který měl v ve vstupním grafu. Vrchol je dobrý, když se dotýká *skutečných* hran obou barev.

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.

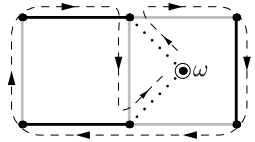
U vrcholů s lichým skutečným stupněm je to složitější, protože do nich můžeme přijít či z nich odejít po virtuální hraně. Ale protože původní graf neobsahoval listy, všechny mají skutečný stupeň alespoň 3, tedy v upraveném grafu stupeň alespoň 4. Každým takovým vrcholem musí tah projít alespoň dvakrát.

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.

Například následující graf:



obarvíme takto:

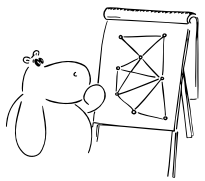


Na závěr shrňme celý algoritmus:

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



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`.

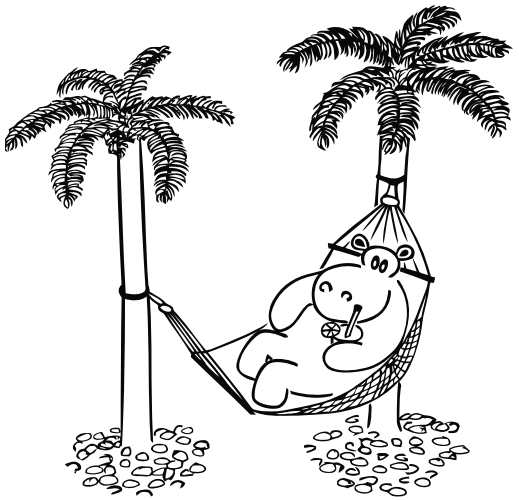
Úkol 3 [4b]:

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: Rekurzivní funkce a 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

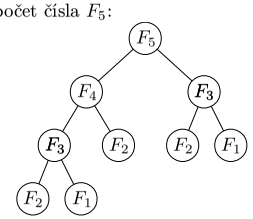
Budeme počítat n-té číslo Fibonacciho posloupnosti. To je posloupnost, jejímiž prvními dvěma členy jsou jedničky ($F_1 = 1$, $F_2 = 1$) a každý další člen je součtem dvou předchozích ($F_n = F_{n-1} + F_{n-2}$ pro $n > 2$). Začíná takto:

1 1 2 3 5 8 13 21 34 55 89 ...

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.

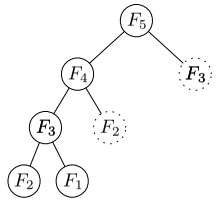
Právě zde je zmínka o králících příhodná.
Legenda o Fibonacciho číslech vypráví, že k jejich objevu došlo při výzkumu rozmnožování králíků.

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]
```


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.

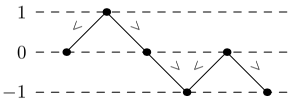
Pokud je mezi třemi pozicemi relace $a > b < c$, víme, že na pozici b nemůžeme umístit největší číslo ze vstupu, protože potřebujeme alespoň dvě větší čísla pro umístění na pozice a a c. Naopak pro pozice a a c žádné takové omezení nemáme a je nám dokonce jedno, v jakém vztahu budou čísla na těchto dvou 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.

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

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.

Vzorová řešení čtvrté série dvacátého osmého ročníku KSP

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).[3]

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

[3] Viz kuchařku na straně 8 tohoto letáku.

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 rovnou 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.

[2] `http://ksp.mff.cuni.cz/viz/kucharky/tezke-problemy`

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.[2]

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).

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.

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 $D[i][k] + D[k][j]$.

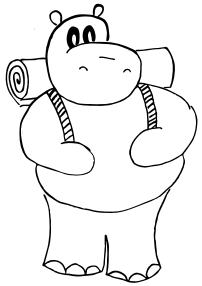
Takže pokud je součet $D[i][k] + D[k][j]$ menší než stávající hodnota $D[i][j]$, nahradíme hodnotu na pozici $D[i][j]$ tímto součtem, jinak ji ponecháme.

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:

posloupnost:	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