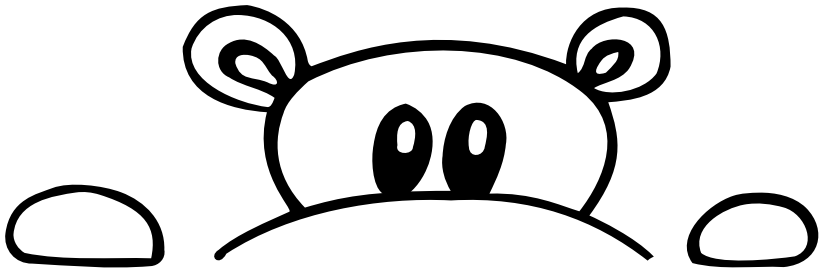
	řešitel	*škola*	*ročník*	*sérií*	*5-1*	*5-2*	*5-3*	*5-4*	*5-5*	*5-6*	*5-7*	*5-8*	*série*	*celkem*
57.–60.	David Blažek	SPŠÚžlabPH	3	1									0,0	8,0
	Lucie Kubíčková	GFXŠaldyLI	2	1									0,0	8,0
	Antonín Prantl	G Strakon	3	1									0,0	8,0
	Zuzana Svobodová	G FrýdlNOs	4	2									0,0	8,0
61.	Eliška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



matfyz

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

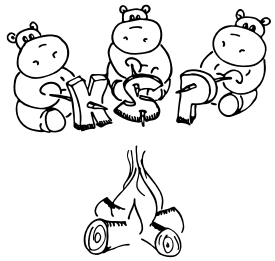
Webové stránky: *https://ksp.mff.cuni.cz/*
E-mail: *ksp@mff.cuni.cz*
Diskusní fórum: *https://ksp.mff.cuni.cz/forum/*

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** **Srpen 2016**

Milí řešitelé, řešitelky a řešitelčata!



Sedí takhle tři hroši u táboráku a vyprávějí si, co už o prázdninách prožili. A když už zbývají poslední řeřavé uhlíky, vzpomínají také na vás řešitele a je jim poněkud stydno, že vám ještě neposlali vzorová řešení poslední série.

Tož tady jsou, vonící ohýnkem a horskými loukami, zářící kapkami ranní rosy i hvězdami na půlnoční obloze.

Tím je 28. ročník našeho programovacího semináře u konce. S těmi nejlepšími z vás se těšíme na viděnou na podzimním soustředění či rovnou na Matfyzu. A nebojte, příští rok pokračujeme! Ostatně, zádání první série už visí na webu.

Přejeme vám příjemné počtení a krásný zbytek léta

Vaši organizátoři (též organizátorky a organizátorčata)

Vzorová řešení páté série dvacátého osmého ročníku KSP

28-5-1 Zaměřování místnosti

Nejprve si všimneme, že pokud obdélník existuje, pak nějaké dva body z libovolné pětice určují jednu stranu obdélníku. To vyplývá z tzv. Dirichletova principu (neboli principu holubníku), který v tomto případě říká, že pokud máme pět bodů, které leží na čtyřech stranách, tak nutně na alespoň jedné straně jsou alespoň dva body.

Vezmeme tedy dva body z náhodně vybrané pětice a zkusíme, jestli s jednou stranou určenou touto dvojicí bodů obdélník existuje. (Jak přesně to zjistíme, je vysvětleno v následujícím odstavci.) Když chceme vybrat dva body z pěti, máme celkem deset možností, jak to udělat. Proč? Pokud znáte kombinační čísla, tak jistě dokážete spočítat, že vybíráme-li dva z pěti bodů, možností, jak to udělat, je deset.

Pokud vám kombinační čísla nic neříkají, stačí jednoduchá úvaha: je pět možností, jak vybrat první bod, a pro každou z těchto pěti možností jsou ještě čtyři další, jak k nim vybrat druhý bod. Pět krát čtyři je tedy dvacet možností, když víme, jaký bod byl vybrán první a jaký druhý. Nám ale na jejich pořadí nezáleží a můžeme je mezi sebou zaměnit. Tedy dvacet vydělíme dvěma a dostáváme, že způsobů, jak vybrat dva z pěti bodů, když nezáleží na pořadí, je deset.

Teď si ukážeme, jak s libovolnou dvojicí bodů postupovat. U ostatních bodů zkontrolujeme, zdali neleží na této první přímce také. Pokud ano, můžeme je pro teď zapomenout. Pro všechny zbylé body si spočítáme vzdálenost od první přímky. Body (nebo bod) s největší vzdáleností budou ležet na opačné straně obdélníka a také je můžeme pro teď odstranit. Zbylé body by tedy měly ležet na nejvýše dvou přímkách kolmých na dosud určené přímky. Navíc musí být tyto kolmé přímky od sebe vzdálené alespoň tak, jak jsou od sebe vzdálené krajní body na původních přímkách, tak aby tvořily obdélník.

Pro každou dvojici tedy provedeme řádově N operací, kde N je počet bodů celkem. Různých dvojic je celkem deset, složitost je tak $10N$, tedy $\mathcal{O}(N)$ – lineární.

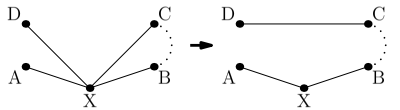
Zbývá vyřešit případy, kdy je bodů méně než pět. Pokud jsou body tři a méně, lze obdélník sestrojit vždy. Pokud jsou body čtyři, můžeme si pomoci například sestrojením konvexního obalu. Pokud na něm nějaký bod neleží, obdélník existovat nemůže.

Někteří zvolili k celému řešení úlohy sestrojení konvexního obalu a následně jej různě využili ke zjištění, zda jde o obdélník, nebo ne. Většina nápadů nebyla špatná a fungovala by, ale sestrojit konvexní obal trvá $\mathcal{O}(N \log N)$, a je to tedy pomalejší než výše popsané lineární řešení. Jediný řešitel, který správné lineární řešení poslal, byl Tomáš Domes, kterého tímto chceme veřejně pochválit.

Zuzka Drázdová

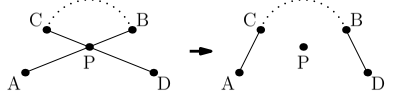
28-5-2 Místní přesuny

Nejprve si uvědomíme, že se žádný bod nevyplatí navštívit dvakrát. Druhou návštěvu můžeme vynechat, a tím si cestu vždy zkrátíme:



Z trojúhelníkové nerovnosti $|CD| < |CX|+|XD|$ tedy upravená cesta musí být kratší. Hledáme tedy nejkratší *hamiltonovskou cestu* v úplném grafu, jehož vrcholy jsou naše body a hrany úsečky spojující každou dvojici bodů. To je cesta, která navštíví každý vrchol právě jednou. Pro obecné grafy je její nalezení těžký problém,[1] ale ukážeme si, že v případě vrcholů v konvexní poloze to neplatí.

Dále si všimneme, že optimální cesta nikdy nekříží sebe sama. Uvažujme libovolnou cestu, na které se kříží nějaké dvě hrany AB, CD. Označme si P jejich průsečík. I takovou cestu snadno zkrátit:



Opět z trojúhelníkové nerovnosti: $|AC| < |AP| + |CP|$, $|BD| < |PD| + |PB|$. Sečtením dostáváme $|AC| + |BD| <$

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

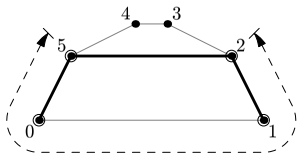
$|AP|+|PB|+|CP|+|PD| = |AB|+|CD|$, tedy nová cesta je kratší než původní.

Nyní k samotnému algoritmu. Začneme tím, že si vrcholy očíslujeme v pořadí, v jakém leží na obvodu konvexního mnohoúhelníku. K tomu můžeme použít libovolný algoritmus pro hledání konvexního obalu.[2] Protože máme body v konvexní poloze, budou všechny patřit do nalezeného konvexního obalu, ten nám ale navíc určí jejich pořadí. Zvládneme to v čase $\mathcal{O}(N \log N)$.

Očíslování budeme chápat jako cyklické a s čísly vrcholů pracovat modulo N, tedy např. $0 - 1 = N - 1$. Pak sousedé libovolného vrcholu X jsou $X + 1$ a $X - 1$.

Označení úlohy jako kuchařkové napovídá, že chceme použít dynamické programování.[3] Abychom to mohli udělat, musíme úlohu rozdělit na menší podproblémy, z řešení kterých lze poskládat řešení původního většího problému.

Na první pohled se zdá, že vhodným podproblémem by mohlo být hledání nejkratší cesty pokrývající nějaký souvislý úsek vrcholů mnohoúhelníka. Takový úsek můžeme identifikovat dvěma čísly (a, b) udávajícími počáteční a koncový vrchol. Pozor, že pokud jde úsek „přes nulu“, může být $b < a$. Na následujícím obrázku je zvýrazněn úsek (5, 2) tvořený vrcholy 5, 0, 1, 2 a nejkratší cesta jej pokrývající:

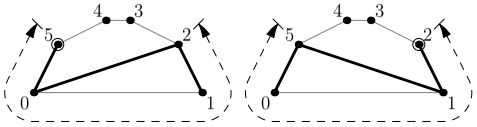


pokrývaný úsek

Na obrázku vidíme další problém: takováto nejkratší cesta je nám k ničemu, protože ji nedokážeme bez křížení prodloužit na nejkratší cestu pokrývající nějaký delší úsek. Aby se na cestu dalo ještě něco napojit, musí mít alespoň jeden konec v krajním bodě intervalu (zde 5 nebo 2).

Zkusme si tedy zesílit zadání: pro každý úsek (a, b) budeme hledat hned dvě cesty. První z nich bude nejkratší cesta navštěvující všechny vrcholy úseku a končící v a. O tom, kde má druhý konec, nic neříkáme. Může to být v b, ale taky někde uprostřed úseku. Označíme si ji $P_1(a, b)$, její délku pak $D_1(a, b)$. Analogicky $P_2(a, b)$ bude nejkratší pokrývající cesta končící v b.

Snadno si navíc rozmyslíte, že ani jedna z těchto cest nikdy nenavštíví vrchol mimo požadovaný úsek (protože takovou nepovinnou návštěvu můžeme vynechat, a tím cestu zkrátit). Může se stát, že $P_1(a, b) = P_2(a, b)$, ale obecně nemusí. Vše je vidět na následujícím obrázku.



$P_1(5, 2)$ $P_2(5, 2)$

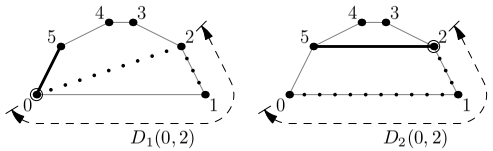
Pro jednoduchost budeme počítat jen délku nejkratší cesty, nalezení cesty samotné se snadno doplní. Jak spočítáme např. $D_1(a, b)$?

Uvažujme poslední hranu cesty $P_1(a, b)$, končící v a. To může být buď strana mnohoúhelníka, nebo úhlopříčka. Pokud je to strana, musí vést do vrcholu $a + 1$ (na obrázku 0). Vrchol $a - 1$ (4) leží mimo pokrývaný úsek a výše jsme ukázali, že takové se navštívit nevyplatí.

Po odebrání poslední hrany $P_1(a, b)$ dostaneme nějakou cestu pokrývající o jedna kratší úsek, $(a + 1, b)$, která navíc končí v $a + 1$. A musí to být nejkratší taková, tedy dlouhá $D_1(a + 1, b)$. Kdyby byla delší, můžeme ji zkrátit výměnou za $P_1(a + 1, b)$.

Pokud je poslední hrana úhlopříčka, musí naopak vést do vrcholu b. Kdyby vedla kamkoli jinam, rozdělí nám pokrývanou část mnohoúhelníka na dvě poloviny, mezi kterými nelze přejít bez křížení cest. Představte si na obrázku výše, co by se stalo, kdyby cesta končila hranou (1, 5). Analogicky po odebrání poslední hrany dostaneme nějakou cestu pokrývající zbylý úsek, ale tentokrát končící v b.

To jsou jediné dvě možnosti, jak může $P_1(a, b)$ vypadat. Pak stačí vybrat tu kratší:



$D_1(0, 2)$ $D_2(0, 2)$

Dostáváme jednoduchou rekurenci:

$$D_1(a, b) = \min \begin{cases} d(a, a+1) + D_1(a+1, b) \\ d(a, b) + D_2(a+1, b), \end{cases}$$

kde $d(x, y)$ značí délku příslušné hrany.

Pro D_2 to dopadne analogicky. Nyní stačí hodnoty D_1 a D_2 spočítat pro všech N^2 dvojic dynamickým prograovaním. Jeden výpočet trvá konstantně dlouho, takže vše zvládneme v čase $\mathcal{O}(N^2)$. Nakonec stačí z nalezených cest pokrývajících celý mnohoúhelník (rozmyslete si, že to jsou právě $D_1(a, a - 1)$ a $D_2(a, a - 1)$ pro všechna a) vybrat tu nejkratší.

Spotřebujeme též kvadratické množství paměti. Ale pokud bychom si namísto nejkratší cesty vystačili s její délkou, můžeme paměťovou složitost zlepšit na lineární. Všimneme si totiž, že kdykoli během výpočtu si stačí pamatovat poslední dvě patra rekurze. Když počítáme hodnoty pro úseky délky k, stačí nám k tomu znát hodnoty pro úseky délky $k - 1$. Všechny kratší můžeme zahodit.

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

Filip Štědronský

28-5-3 Trezor s alarmem

Schéma zapojenia alarmu predstavuje acyklický orientovaný graf s dvomi vyznačenými vrcholmi – pólmi. Tento graf si označme ako G, záporný začiatočný vrchol ako z a kladný koncový ako k. Našou úlohou je nájsť v grafe G kritické vrcholy. Kritický vrchol je ten, ktorý leží na každej jednej možnej ceste zo z do k. Počet vrcholov grafu G budeme značiť ako N a počet hrán ako M.

Aby sa nám lepšie s grafom pracovalo, tak ho na začiatku prečistíme. Odstránime si z neho vrcholy (a k ním pripojené

[2] `http://ksp.mff.cuni.cz/viz/kucharky/geometrie`
[3] `http://ksp.mff.cuni.cz/viz/kucharky/dynamika`

Závěrečná výsledková listina 28. ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*5-1*	*5-2*	*5-3*	*5-4*	*5-5*	*5-6*	*5-7*	*5-8*	*série*	*celkem*
0.					9	11	10	10	7	11	12	14	58,0	300,0
1.	Richard Hladík	GOAMarLaz	3	20	8	10,5	10	10		1	8		40,6	236,6
2.	Pavel Turek	GTomkovaOL	3	5	8	9	10	4,5	5	1			41,2	223,8
3.	Jiří Sejkora	GVoděraPH	4	8		2	1	5		3	4	12	29,0	220,8
4.	Jakub Pelc	G UherBrod	2	5						0		12,5	12,1	211,5
5.	Václav Volhejn	GKepleraPH	3	19							12		12,0	190,0
6.	Jan Bouček	GKepleraPH	3	8									0,0	169,0
7.	Jonáš Fiala	GJungmanLT	3	5			9	5	5				22,5	120,0
8.	Stanislav Lukeš	GPísnickáPH	3	10									0,0	100,2
9.	Daniel Herman	GŠKo	3	3									0,0	96,8
10.	Michal Töpfer	G DrJPekMB	3	10			1		7			0,5	8,0	92,1
11.	Václav Pavlíček	ZŠ Ždírec nD	0	5				5	3	1			13,2	81,0
12.	Leonard Mentzl	GŘíč	3	2									0,0	76,9
13.	Petr Chmel	G_Kralupy	3	4									0,0	75,1
14.	Josef Gajdůšek	SŠKKamPard	3	3									0,0	71,1
15.	Jakub Těték	Dollar Ac	2	9									0,0	63,0
16.	Vojtěch Lukeš	GPikaPL	4	2									0,0	62,9
17.	Pavel Turinský	G Brandýs	3	9	6				5				11,0	62,5
18.	Lukáš Rozsypal	GÚstavníPH	3	3									0,0	59,6
19.	Přemysl Šťastný	GŽamberk	3	12	3				6				7,9	58,4
20.	Václav Šraier	GČeskoliPH	3	9	6				7				13,0	55,9
21.	Jakub Suchánek	GOpatovPHA	2	2	7		8		6	1			26,5	54,0
22.	Lukáš Vlček	GMikulášPL	2	3									0,0	48,9
23.	Jiří Löffelmann	GLitoměřPH	2	3						1	0		1,9	48,5
24.	Jakub Dobrý	GMikulášPL	2	3									0,0	41,5
25.	Jiří Vozár	G UherBrod	4	8									0,0	40,6
26.	Miroslav Hrabal	GTomkovaOL	2	2									0,0	40,2
27.	Viktor Fukala	GKepleraPH	−1	1									0,0	39,0
28.	Ján Chudý	GŽilina	4	1									0,0	38,5
29.	Vanda Hendrychová	GHeyrovPH	4	2	7					0			8,4	37,6
30.	David Žáček	GZborovPH	3	3									0,0	36,4
31.	Vojtěch Hudec	G_ČTřebová	2	2	5	0		1	2				12,4	36,0
32.	Ondrej Pudiš	GŽilina	4	1									0,0	35,0
33.	Michal Kodad	ZŠJílovsPH	0	4									0,0	31,1
34.–35.	Vladimír Bartovic	G AM Trnava	4	2									0,0	29,2
	Jakub Lukeš	GNAlejíPH	3	4									0,0	29,2
36.	Tomáš Domes	MendelG_OP	3	1	9		1	4	7	1			26,4	26,4
37.	Roman Beňo	GJHroncaBA	3	2									0,0	25,6
38.	Zdenko Čepan	GPartizans	3	2									0,0	22,6
39.	Michal Jireš	GRNK	1	1									0,0	22,5
40.	David Ucháč	eduSOŠ PA	3	2									0,0	21,9
41.	Marco Souza de Joode	GNadŠtolPH	−1	2									0,0	20,8
42.	Sam Friedlaender	GKepleraPH	−1	2									0,0	20,3
43.	František Kmječ	G Brandýs	0	2									0,0	19,8
44.	Jan Pokorný	G Bučovice	4	8									0,0	17,7
45.	Filip Geib	G MMH LM	2	2									0,0	17,1
46.	Alexej Popovič	SlovanGOL	4	2									0,0	16,1
47.	Alena Tesařová	GVídeňskBO	4	2									0,0	15,5
48.	Jan Kaifer	GČesBrod	0	2									0,0	15,4
49.	Josef Pospíšil	GÚstavníPH	2	1									0,0	12,6
50.	Petr Gebauer	GMělník	2	1									0,0	10,6
51.–54.	Jan Gocník	GJŠkodyPŘ	4	4									0,0	10,0
	Martin Kučera	GNeratov	4	1									0,0	10,0
	Jan Priessnitz	GJarošeBO	3	1									0,0	10,0
	Filip Šohajek	GUHradiště	−1	1									0,0	10,0
55.	Jan Neumann	GNAlejíPH	2	1									0,0	9,7
56.	Michal Rickwood	G_ČTřebová	2	2									0,0	8,3

(těch je též $\mathcal{O}(n)$), ale s tím narazíme: na zodpovězení dotazu bychom potřebovali složit tři mezivýsledky, tedy zavolat operaci $\otimes$ dvakrát, což je moc.

Proto si místo obyčejných suffixů předpočítáme všechny intervaly, které vzniknou rozšířením suffixu o nějaký počet celých bloků. Každý suffix má $\sqrt{n}$ takových rozšíření, úplně všechny suffixy pak $\mathcal{O}(n^{3/2})$. Pak stačí skládat rozšířený suffix s obyčejným prefixem.

Získali jsme tedy řešení s předvýpočtem v čase $\mathcal{O}(n^{3/2})$ a dotazem v čase $\mathcal{O}(1)$ s jedním voláním operace $\otimes$.

Cesta jde pořád dál a dál...

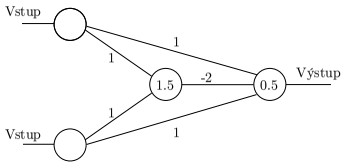
Na závěr ještě dodejme, že kdybychom povolili větší (ale stále konstantní) počet použití $\otimes$, úloha by byla mnohem zajímavější, ovšem také mnohem náročnější. Například pro 3 volání $\otimes$ stačí předzpracování v čase $\mathcal{O}(n \log^* n)$, kde $\log^*$ je funkce zmíněná v řešení čtvrté úlohy.

Martin „Medvěd“ Mareš

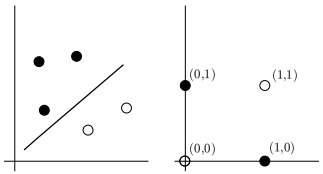
28-5-8 Neuronové sítě

Úkol 1

Cílem první úlohy je nalézt co nejmenší neuronovou síť modelující XOR. Překvapilo nás, že žádné ze zaslaných řešení nebylo optimální, takže tady je:



Důkaz minimality provedeme sporem. Kdyby byla síť ještě menší, měla by jen dva vstupní a jeden výstupní neuron bez skryté vrstvy. Jeden neuron dokáže, stejně jako perceptron, v dvourozměrném vstupním prostoru oddělovat dvě skupiny vzorů přímkou, jak jsme psali v minulém dílu seriálu. U funkce XOR ale vstupní vzory přímkou oddělit nelze, říkáme, že nejsou *lineárně separovatelné*. To ilustrují následující dva grafy lineárně separovatelné množiny a množiny možných vstupů xoru.



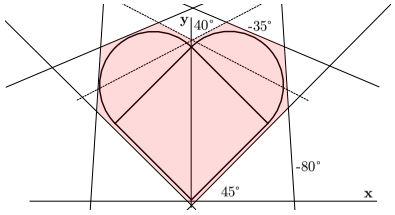
Úkol 2

Druhým úkolem bylo implementovat neuronovou síť pro klasifikaci kosatců. Na následujícím odkazu si můžete stáhnout vzorové řešení v jazyce Python. Náš program využívá neuronovou síť o 3 skrytých neuronech v jedné skryté vrstvě a dosahuje přesnosti klasifikace na validační množině kolem 98% až 100%.

Program (Python):
`http://ksp.mff.cuni.cz/viz/28-5-8-2.py`

Úkol 3

Poslední úlohu bylo možné řešit pomocí zpětné propagace nebo navrhnout síť pomocí analytické geometrie. Jelikož zpětnou propagací jsme řešili minulou úlohu a používala ji i všechna vaše řešení, podíváme se na geometrický přístup. Kolem srdce zkonstruujeme „obal“ z několik přímek, které budou oddělovat body uvnitř obrazce od ostatních. Tyto přímky pak převedeme na neurony, které budou dělat totéž v rámci neuronové sítě.



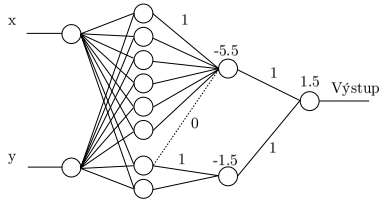
Z analytické geometrie víme, že přímka má rovnici $y = ax + b$, také víme, že a se dá spočítat pomocí úhlu α, který přímka svírá s osou x: $a = \tan(\alpha)$. Hodnotu b pak můžeme dopočítat dosazením nějakého bodu na přímce do rovnice přímky. Díky tomu můžeme spočítat rovnice všech osmi v obrázku zakreslených přímek. Ukažme si to na pravé spodní přímce, která má sklon 45° a leží na ní bod $[0;0]$:

$$\begin{aligned} a &= \tan(\alpha) = \tan(45^\circ) = 1 \\ y &= x + b \\ 0 &= 0 + b \\ b &= 0 \end{aligned}$$

Tuto rovnici přímky nyní můžeme upravit do tvaru $0 = ax - 1y + b$ a porovnáním s rovnicí neuronu $0 = \xi = w_1 x + w_2 y + \delta$ z ní určit jeho váhy w a práh δ. Pro naši první přímku $y = x + 0$ to bude $w = [1; -1]$ a $\delta = 0$. Všimněte si, že nemáme zajištěno, na které straně přímky bude potenciál neuronu kladný a na které záporný. V tomto případě bude kladný v oblasti mimo srdce. Pokud bychom chtěli hodnoty prohodit, můžeme vynásobit uvedenou rovnici neuronu -1, čímž dostaneme váhy a práh také vynásobený -1.

Tímto postupem můžeme určit váhy a prahy neuronů všech osmi přímek. Řekněme, že tedy máme neurony s aktivační funkcí signum, které mají pro body směrem dovnitř obrazce kladný výstup $+1$. Bod je uvnitř srdce tehdy, když všechny neurony mají kladný výstup, kromě dvou neuronů, jejichž přímky jsou na obrázku vykresleny přerušovanou čarou. Z těch stačí jeden s výstupem $+1$.

Abychom tuto podmínku realizovali neurony, pořídíme si vrstevnatou neuronovou síť jako na následujícím obrázku. Neurony přímek jsou použity v první skryté vrstvě. Aktivační funkce všech neuronů je signum. Nezakreslené hrany můžeme v síti formálně ponechat s nastavenou váhou 0 (ilustrováno přerušovanou čarou).



Jan Škoda

hrany), ktoré neležia na žiadnej ceste zo z do k. Je zrejmé, že vrcholy na odstránenie budú práve tie, z ktorých nevedie cesta do k, a tie, do ktorých sa zo z nedá dostať. Žiadne ďalšie to byť nemôžu, lebo vrchol, do ktorého sa dá dostať zo z a zároveň z neho vedie cesta do k, tvorí cestu zo z do k. Graf, ktorý ostane po tomto prečistení, si označme ako G', jeho počet vrcholov N' a hrán M'.

Graf G' je naďalej orientovaný a neexistuje v ňom ani žiaden orientovaný cyklus (odstraňovaním vrcholov a hrán cyklus vzniknúť nemôže). Teda vrcholy grafu G' je možné topologicky zoradiť. To znamená, že vrcholy sa očíslujú od 1 do N', a to tak, že každá orientovaná hrana bude viesť vždy od vrcholu s menším číslom do vrcholu s väčším číslom. Všimnime si, že z dostane číslo 1 a k číslo N', lebo žiaden vrchol nemôže byť väčší ako k a žiaden menší ako z.

Posledný krok bude v grafe G' nájsť kritické vrcholy. Vrchol v bude kritický práve vtedy, keď nebude existovať žiadna hrana vedúca od vrcholu s číslom menším ako v do vrcholu s číslom väčším ako v. Ak by takáto hrana existovala, potom cesta zo z do k, ktorá obsahuje túto hranu nemôže zároveň obsahovať vrchol v (kvôli topologickému usporiadaniu vrcholov). Zároveň vrchol, ktorý neobchádza žiadna hrana, sa bude nachádzať na každej ceste zo z do k, keďže ako jediný spája vrcholy s menšími číslami ako je on sám a s väčšími číslami ako je on sám.

Ostáva nám už iba popis algoritmov na jednotlivé časti rozboru. Prečistenie grafu spravíme dvojitým prehľadaním grafu do hĺbky. Prvý spustíme zo z a označíme si dosiahnuteľné vrcholy po smere hrán. Druhý spustíme z k, ale proti smeru orientácií hrán, čím získame vrcholy, z ktorých sa dá dostať do k. To zvládneme v čase $O(N + M)$, lineárne od počtu hrán a vrcholov v G. Topologické zoradenie vrcholov acyklického orientovaného grafu vieme uskutočniť taktiež v čase $O(N + M)$, napríklad pomocou algoritmu z našej kuchárky o rovinných grafoch.[4]

V poslednom kroku už budeme mať dostupný topologicky zoradený zoznam vrcholov grafu G'. Tento zoznam vrcholov budeme postupne spracovávať od najmenšieho po najväčší. Pri spracovávaní si budeme pamätať a priebežne aktualizovať maximálne číslo m. To bude reprezentovať najvzdialenejší vrchol od začiatku, na ktorý už viedla nejaká hrana.

Spracovávanie každého vrcholu začneme porovnaním čísla tohoto vrcholu a hodnoty m. Ak sú hodnoty rovnaké, potom bude aktuálny vrchol kritický. Platí to, pretože neexistuje hrana zo žiadneho vrcholu s menším číslom ako m do vrcholu s väčším číslom ako m. Následne sa pozrieme na všetkých susedov spracovávaného vrcholu. Vždy, keď bude číslo niektorého suseda väčšie ako m, tak zmeníme m práve na číslo toho suseda. Týmto docielime, že na konci spracovávania vrcholu, bude v m opäť najväčšie číslo, do ktorého sa vieme dostať.

Aby sa nám spracovávanie zoznamu vrcholov nedostalo do nedefinovaného stavu, tak si na začiatku nastavíme hodnotu m na 1. Zároveň budeme ignorovať informáciu, že kritický vrchol by mal byť aj z (rovný 1) a aj k (najväčší). Tie v skutočnosti netvoria uzly sústavy obvodov zo zadania.

Posledný krok zvládneme taktiež v čase $O(N + M)$, keďže každý vrchol a každú hranu spracujeme práve raz. Teda celková časová zložitosť bude lineárna od súčtu počtu hrán a vrcholov vo vstupnom grafe G. Ak použijeme klasickú reprezentáciu grafu G a G' ako zoznam susedov pre každý vrchol, tak rovnako bude na tom aj pamäťová zložitosť.

Program (C++):
`http://ksp.mff.cuni.cz/viz/28-5-3.cpp`

Pali Rohár

[4] `http://ksp.mff.cuni.cz/viz/kucharky/rovinne-grafy`

28-5-4 Časoprostorové mosty

Pro řešení této úlohy existuje poměrně jednoduchý, leč neefektivní algoritmus: Po každém odstranění mostu zkontrolujeme graf prohledáním vrcholů, zda se komponenta obsahující tento most rozpadla na dvě.

Tento způsob určitě funguje, ale je velmi pomalý. Každé odstranění mostu znamená $\mathcal{O}(N)$ operací, tedy celkově tento algoritmus zabere až $O(M \cdot N)$ času. Nejvíce nás zdržuje právě zjišťování, zda odstranění daného mostu způsobilo rozpad komponent. Jak z této neefektivity vybruslíme?

Pojďme se na chvíli podívat na algoritmy pro hledání nejmenších koster, konkrétněji Kruskalův algoritmus, o kterém se lze více dočíst v kuchařce o kostrách.[5] Jeho princip spočívá v tom, že postupně bere nejlevnější hranu. Poté se ptá, zda její přidání do kostry způsobí sloučení dvou komponent do jedné. A to je přesně opačný problém.

K tomuto problému existuje datová struktura *Union-Find*, která poskytuje dvě hlavní operace:

- *Union* spojí dvě komponenty určené vrcholy A a B.
- *Find* pro dva vrcholy A a B odpoví, zda tyto vrcholy leží ve stejné komponentě souvislosti.

O tom, jak tato datová struktura funguje a jak je složitá, si můžete více přečíst ve výše zmíněné kuchařce.

Jak nám ale tato datová struktura pomůže? Dělá přece opak toho, co chceme. Nezoufejme však, jelikož můžeme použít chytrý trik. Vzhledem k tomu, že známe dopředu posloupnost odstranění mostů, můžeme celý postup obrátit. Začneme s grafem podle toho, jak vypadá po odstranění všech mostů a postupně voláme operace *Union* a *Find*.

*Find*em zjistíme, zda most vede uvnitř jedné komponenty. Pokud ano, nemusíme dělat nic. Jinak *Union*em komponenty sloučíme, sečteme jejich energie a součet prohlásíme za energii nové komponenty. Takto postupujeme, dokud nepřidáme všechny mosty. Nakonec vypíšeme výsledek jednotlivých operací pozpátku.

Celkově nás tedy odebrání (přidání) jednoho mostu stojí tolik času, co provedení operací *Union* a *Find*. To je pro dostatečně slušnou implementaci této datové struktury $\mathcal{O}(\log N)$. Dohromady má tedy celý algoritmus časovou složitost $\mathcal{O}(M \log N)$.

Jirka Setnička & Václav Končický

Poznámka M.M.: Složitost $\mathcal{O}(\log N)$ pro Union-Find je velmi velkorysá. Ve skutečnosti i jednoduchá implementace struktury pomocí stromečků, kterou popisujeme v kuchařce, pracuje daleko efektivněji. Složitost lze omezit například funkcí $\mathcal{O}(\log^*)$. Tento „hvězdičkový logaritmus“ je definovaný jako funkce inverzní k takzvané věžové funkci:

$$2 \uparrow 1 = 2, \quad 2 \uparrow 2 = 2^2, \quad 2 \uparrow 3 = 2^{2^2}, \quad 2 \uparrow (k+1) = 2^{2 \uparrow k}.$$

Funkce $\log^* n$ tedy roste velice pomalu, ale ani ona není nejlepší možná. Více už zde neprozradíme a raději vás odkážeme na detaily v kuchařce.

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

28-5-5 Kalibrace

Máme původně setříděnou posloupnost N čísel, která se zrotovala o neznámý počet pozic, a přesně tento počet bychom chtěli určit. Než se do toho pustíme, podotkněme, že čísla už máme skutečně uložená v paměti. Nemusíme se tedy starat o načtení vstupu (to v praxi odpovídá třeba tomu, že píšeme ne celý program, ale nějakou funkci).

Pravděpodobně nás záhy napadne použít nějakou formu binárního vyhledávání, které je popsané třeba v kuchařce základních algoritmů.[6]

Co s ním ale budeme hledat? Špatně uspořádanou dvojici čísel. Můžeme si rozmyslet, co dostaneme, když zrotujeme posloupnost doprava. Na začátku bude nějaký správně uspořádaný kus, pak špatně uspořádaná dvojice (konkrétně největší prvek následovaný nejmenším) a pak zase správně uspořádaný kus.

Budeme tedy hledat tohle špatné uspořádání, přesněji druhé z těchto špatně uspořádaných čísel. Jeho pozice nutně odpovídá rotaci (počtu pozic, o které byla posloupnost zrotovaná).

V každém kroku se díváme na úsek. Nejprve porovnáme levý krajní prvek L a pravý krajní prvek R tohoto úseku. Je-li $R > L$, úsek je správně uspořádaný. Pokud víme, že v něm je špatně uspořádaný prvek, musí být hned na začátku a můžeme vrátit pozici L.

V opačném případě se podíváme na prostřední prvek M. Je-li $M > L$, první polovina je správně uspořádaná. Problém tedy bude ve druhé, takže se zanoříme do pravé poloviny. Pro $M < L$ naopak. Dostaneme-li se k úseku velikosti 1, určitě je problematickým prvkem právě ten jediný v něm.

Voilà, máme algoritmus, který najde problematickou pozici. Jelikož vždy rozdělíme aktuální úsek alespoň na polovinu, skončíme nejpozději po $\log N$ krocích. Časová složitost algoritmu je tak $\mathcal{O}(\log N)$.

Paměťová složitost je $\mathcal{O}(N)$, pokud uvažujeme uloženou posloupnost. Můžeme ale říct, že tak jako zanedbáváme načtení vstupu, zanedbáme uložení posloupnosti, a pak je paměťová složitost konstantní, $\mathcal{O}(1)$.

Karry Burešová

28-5-6 Sloty na iridium

Rozmisťování iridia do slotů se ukázalo být zapeklitějším problémem, než se zprvu zdálo. Většina z odvážných, kdo se úloze vydali vstříc, vyřešila první triviální vstup, ale přes druhý už se dostal jen jediný z vás (gratulujeme tímto Jirkovi Sejkorovi) a dál už se nedostal vůbec nikdo.

Vstupy se postupně zvětšovaly, ale některé z nich měly i jisté speciální vlastnosti. Třeba pátý vstup měl na pár místech veliké mezery a nebo hned třetí vstup měl všechny dostupné sloty rozmístěné jenom v jedné polovině obvodu. Zvlášť možností využít jen jednu polovinu obvodu se úloha pěkně zjednodušila, protože se pak problém choval jako rozmisťování iridia do slotů na přímce namísto kružnice. Vyřešme si tento jednodušší problém.

Sloty na přímce

Pokud budeme mít sloty na přímce, můžeme bez obav umístit iridium do prvního slotu, protože tím nic nepokazíme (pokud by v nějakém optimálním rozmístění nebylo první iridium v prvním slotu, můžeme ho bez zhoršení minimálních vzdáleností do prvního slotu posunout).

Nyní vyzbrojeni tímto pozorováním zkusme vyřešit úlohu trošku odlišnou – totiž otázku, jestli pro zadanou minimální vzdálenost d lze rozmístit iridium do slotů tak, aby byla tato minimální vzdálenost dodržena. Až vyřešíme tuto úlohu, ukážeme si, jak nám pomůže vyřešit původní problém.

Z pozorování výše víme, že první iridium můžeme umístit do prvního slotu a nic si tím nepokazíme. Další iridium můžeme umístit nejblíže ve vzdálenosti d. Takže přeskočíme všechny bližší sloty a iridium umístíme do prvního slotu, který má vzdálenost větší nebo rovnou d.

A tímto způsobem postupujeme dál, plníme vlastně sloty hladově zleva. Pokud se nám takto povede všechno iridium umístit, zahlásíme úspěch, pokud nám ale dojdou sloty a ještě nám bude zbývat neumístěné iridium, tak víme, že rozmístit s touto vzdáleností nejde (žádné iridium již nelze posunout více doleva, abychom si na konci uvolnili nějaké sloty).

V čase $\mathcal{O}(S)$ tedy umíme lineárním průchodem přes sloty vyřešit tuto podúlohu (připomeňme, že S je počet dostupných slotů). Jak nám to pomůže se řešením původního problému? Můžeme zkusit najít největší vzdálenost, pro kterou se nám ještě iridium povede rozmístit, a toto rozmístění pak vypsat. Hledání největší vzdálenosti můžeme dělat postupným zvětšováním o jedničku a zkoušením, ale to by pro obvod O trvalo až $\mathcal{O}(OS)$ a to je moc dlouho.

Víme, že se tento problém hledání vzdálenosti chová lineárně – když pro nějakou vzdálenost rozmístění iridia existuje, tak existuje i pro všechny menší vzdálenosti, když naopak neexistuje, tak neexistuje ani pro žádnou větší vzdálenost. Můžeme tedy maximální možnou vzdálenost *binárně vyhledat*.

Budeme si držet minimum a maximum zkoušeného rozsahu (minimum bude na začátku 1, maximum bude obvod dělený počtem kousků iridia). V každém kroku se podíváme na vzdálenost $d = \frac{min+max}{2}$ a ozkoušíme rozmístit iridium s touto minimální vzdáleností:

- Pokud se iridium povede rozmístit $\rightarrow$ hledaná vzdálenost je větší nebo rovna d, nastavíme $min = d$
- Pokud se iridium nepovede rozmístit $\rightarrow$ hledaná vzdálenost je menší než d, nastavíme $max = d - 1$

Takto pokračujeme, dokud nedojdeme na krok velikosti jedna. Pak již máme vzdálenost určenou jednoznačně. Binárním vyhledáváním uděláme $\mathcal{O}(\log O)$ kroků, takže celkově dosáhneme času $\mathcal{O}(S \log O)$.

Sloty na obvodu kruhu

Při rozmisťování iridia do slotů na obvodu kruhu použijeme úplně stejný postup, jen se už nemůžeme spolehnout na pozorování o umístění prvního iridia do prvního slotu. Nyní totiž jde i o vzdálenost prvního a posledního obsazeného slotu. Může být například výhodné prvních pár slotů přeskočit, aby nám vzdálenost vyšla.

Jak si s tím poradit? Pokud nám při zkoušení, jestli iridium umíme rozmístit se vzdáleností d, dojdou sloty před rozmístěním všech kousků, můžeme rovnou oznámit neúspěch. Pokud se nám naopak iridium rozmístit povede a vzdálenost mezi prvním a posledním obsazeným slotem je dostatečně velká, můžeme rovnou oznámit úspěch. To byly ty jednodušší případy.

Složitější je, pokud sice všechno iridium rozmístíme, ale vzdálenost mezi prvním a posledním bude příliš malá. Pak můžeme zkusit první posunout po obvodu dál tak, abychom tuto vzdálenost dostatečně zvětšili. Pokud tím neporušíme minimální vzdálenost k dalšímu obsazenému slotu, uspěli jsme, jinak to samé opakujeme (opět posuneme další iridium tak daleko, aby byla dodržená minimální vzdálenost a opakujeme).

Toto posouvání zastavíme ve chvíli, kdy se nám rozmístění buď povede opravit (v tom případě oznámíme úspěch), nebo když se pokusíme nějaké iridium umístit opět do prvního slotu. V tom případě totiž opět dostáváme stejnou situaci, jako při prvním rozmístění, a zacyklili jsme se bez nalezení fungujícího rozmístění.

Na implementaci tohoto postupu se můžete podívat v ukázkovém programu, nyní se zamysleme nad časovou složitostí. Hlavní pozorování je, že se při posouvání iridia mezi sloty pokusíme vložit iridium do každého ze slotů maximálně dvakrát. Při druhém pokusu o vložení do stejného slotu totiž dojde k tomu, že se i všechny zbylé kousky rozmístí tak, jak byly, a dojde k zacyklení (při kterém se zastavíme s neúspěchem).

Čas jednoho kroku jsme si tak oproti jednodušší verzi na přímce zhoršili jen konstantně a vnější část s binárním vyhledáváním zůstává stejná. Celkově tak dosáhneme časové složitosti $\mathcal{O}(S \log O)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/28-5-6.c`

Jirka Setnička

28-5-7 Tajemná operace

Zopakujme si zadání: Dostaneme nějakou magickou operaci $\otimes$ a posloupnost $a_1, \ldots, a_n$. Chceme si něco předpočítat, abychom uměli vyhodnocovat intervalové dotazy typu $a_i \otimes a_{i+1} \otimes \ldots \otimes a_j$ a stačilo nám jediné zavolání operace $\otimes$.

Pochopitelně si můžeme předpočítat výsledky pro úplně všechny intervaly $a_i, \ldots, a_j$. Pak při odpovídání na dotazy dokonce nemusíme $\otimes$ volat vůbec. Ovšem předvýpočet trvá čas $\mathcal{O}(n^2)$, což nám nepromine sebetrpělivější uživatel, natož pak přísné elektronické oko CodExovo.

Tak zkusíme sáhnout po různých standardních technikách, jako je třeba rozklad na bloky nebo intervalové stromy. Z těch také použitelné řešení nekápne: na kombinování bloků nebo podstromů sice postačí malý počet volání $\otimes$, ale rozhodně více než povolené jedno. Přesto z myšlenky rekurzivního rozkladu na podproblémy něco vykřešeme. Tedy poslyšte...

Rekurzivní řešení

Učiníme myšlenkový pokus: zadanou posloupnost rozdělíme přibližně uprostřed na *levou polovinu* $a_1, \ldots, a_s$ a *pravou polovinu* $a_{s+1}, \ldots, a_n$. Dotazy rozdělíme na dva druhy podle toho, zda leží přes střed, či nikoliv.

Pokud interval $a_i, \ldots, a_j$ jde přes střed, skládá se ze suffixu levé poloviny (to je nějaký interval $a_i, a_{i+1}, \ldots, a_s$) a prefixu té pravé ($a_{s+1}, \ldots, a_j$). Předpočítáme-li si tedy výsledky pro všechny suffixy levé poloviny a všechny prefixy pravé, umíme je na jedno zavolání $\otimes$ složit do výsledku celého dotazu.

A pokud interval neleží přes střed, přesuneme se do levé či pravé poloviny a tam rekurzivně aplikujeme tentýž postup. Pojďme spočítat, jak je to efektivní.

Označme $P(n)$ časovou složitost předvýpočtu pro posloupnost délky n. Jistě platí, že $P(n) = 2 \cdot P(n/2) + \mathcal{O}(n)$, neboť pro interval délky n v čase $\mathcal{O}(n)$ zpracujeme prefixy a suffixy a rekurzivně předpočítáme zvlášť levou a pravou polovinu. Tuto rovnici pro $P(n)$ můžeme vyřešit třeba analýzou stromu rekurze. Raději použijeme prostý trik: všimneme si, že úplně stejně se chová známý algoritmus MergeSort (třídění sléváním) – také spotřebuje lineární čas a pak rekurzivně zpracuje dva poloviční podproblémy. MergeSort běží v čase $\mathcal{O}(n \log n)$, takže náš předvýpočet také.

Vyhodnocování dotazu buďto skončí ihned (jde-li dotaz přes střed), nebo se rekurzivně zavolá na jednu z polovin. Během $\mathcal{O}(\log n)$ kroků tedy rekurze musí skončit. Jelikož každý krok trvá konstantně dlouho, časová složitost dotazu činí celkem $\mathcal{O}(\log n)$. Podmínku na nejvýše jedno použití $\otimes$ jistě splňujeme.

Program (C):
`http://ksp.mff.cuni.cz/viz/28-5-7-fast.c`

Rychlejší řešení

Nevýhodou předchozího řešení je, že vyhodnocení dotazu sice volá $\otimes$ jenom jednou, ale kromě toho tráví logaritmický čas nalezením správného podintervalu, přes jehož střed zadaný dotaz leží. To lze s trochou šikovnosti zvládnout i v konstantním čase.

Předně si představujme, že délka posloupnosti je mocnina dvojky a že prvky očíslujeme od nuly: $a_0, \ldots, a_{n-1}$. Pokud se na indexy prvků podíváme ve dvojkové soustavě, v levé polovině všechny začínají nulou, v pravé jedničkou. Dotaz tedy leží přes střed právě tehdy, pokud se jeho levý a pravý okraj liší v nejvyšším bitu.

Rekurzivní rozklad intervalů můžeme elegantně popsat binárním stromem, který na h-té hladině testuje h-tý nejvyšší bit čísla. Nejvyšší interval, v němž jde dotaz přes střed, pak odpovídá nejvyššímu bitu, v němž se okraje dotazu i a j liší. To je nejvyšší jedničkový bit v čísle $i \operatorname{XOR} j$. Pro zjišťování, kde leží nejvyšší jednička, si přitom můžeme snadno předpočítat tabulku.

Celkově tedy v konstantním čase spočítáme $i \operatorname{XOR} j$, pomocí tabulky zjistíme hladinu h stromu rekurze, kam se chceme podívat. A nejvyšších h bitů čísel i a j nám řekne, kolikátý vrchol zleva nás na dané hladině zajímá. Vrcholy tedy můžeme mít uložené v poli a indexovat je též v konstantním čase.

Program (C):
`http://ksp.mff.cuni.cz/viz/28-5-7-faster.c`

Rozklad na bloky

Pro zajímavost načrtneme ještě jedno řešení pomocí rozkladu na bloky. To je nejspíš jednodušší na vymyšlení, ale o něco pomalejší.

Posloupnost rozřežeme na $\sqrt{n}$ bloků velikosti $\sqrt{n}$. Dotazy, které leží celé uvnitř jednoho bloku, předpočítáme všechny. Je jich $\mathcal{O}(n)$ na blok, celkem tedy $\mathcal{O}(n^{3/2})$.

Ostatní dotazy chceme skládat ze suffixu prvního bloku, nějakých celých bloků a prefixu posledního bloku. Suffixy a prefixy si můžeme předpočítat (je jich celkem $\mathcal{O}(n)$). Nabízí se předpočítat si také všechny intervaly celých bloků

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