Milí řešitelé!

Poněkud s předstihem dostáváte do rukou zadání třetí série našeho semináře. S ním dostáváte taktéž opravená řešení série první, takže špinavé a podlé fígly, které se z nich naučíte, můžete použít ještě při řešení série druhé :–)

Pro jistotu Vás upozorňujeme na *chybu*, která se vloudila do zadání druhé série – v zadání úlohy 17-2-1 „Prasátko Květomil“ má být v příkladě místo řádku `g = e * a` uvedeno `g = e * e`.

Termín odeslání Vašich řešení třetí série jest určen na 31. ledna 2005. Řešení můžete odevzdávat jak elektronicky na *http://ksp.mff.cuni.cz/submit/*, tak klasickou poštou na známou adresu:

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

Aktuální informace o KSP můžete nalézt na stránkách *http://ksp.mff.cuni.cz/*, dotazy organizátorům můžete posílat e-mailem na adresu *ksp@mff.cuni.cz*.

Zadání třetí série sedmnáctého ročníku KSP

17-3-1 Spisovatel Vilík — 10 bodů

Spisovatel Vilík Šekjrspír (v originále Shaker's Pear, neboli šejkařova hruška, oblíbený to alkoholický nápoj) byl velmi známý autor. Nicméně měl pocit, že jeho knihám se nedostává tolik pozornosti, kolik by si jí podle něj zasloužily. Nakonec se mu dokonce povedlo přijít na to, čím by to mohlo být. (A vzhledem k tomu, že ho dnes zná skoro každý, měl asi pravdu.)

Zjistil, že jeho knihy jsou poněkud nudné, protože se v nich často opakují celé kusy textu. A tak si řekl, že všechna opakování nějakého textu ze svých knih smaže, nebudou potom tak nudné a navíc budou mít otevřený konec.

Když takto upravil první knihu, zjistil, že z ní zbyla ještě celá třetina. Známý myslitel Cibulka mu tedy poradil, že za stejná slova může považovat i dva kusy textu, které mají stejnou délku a skládají se ze stejných znaků. (Nezáleží tedy na pořadí znaků v obou slovech.)

Vilík teď už ale sám nedokáže najít stejná slova, Cibulka mu sám také nechce pomoci (prý řeší zajímavější problém), a tak to zbyde na Vás.

Napište program, který dostane na vstupu číslo k a text délky N znaků skládající se z písmen '`a`'..'`z`', '`A`'..'`Z`', mezer, teček, otazníků a vykřičníků. Má spočítat délku nejdelšího začátečního úseku tohoto textu, ve kterém se ještě *nevyskytují* dvě *shodná* slova. Dvě slova jsou shodná, pokud to jsou souvislé podřetězce zadaného textu a obě se skládají z právě k stejných písmen, i když pořadí těchto písmen může být různé. Velká a malá písmena nerozlišujte, čili '`a`' = '`A`'.

Příklad: Pro $k = 3$, $N = 14$ a text '`Den␣bude␣hned.`' je správný výsledek 12 (v prvních dvanácti písmenech nejsou žádná shodná slova). Pro text '`Den␣je␣tu␣hned`' je správná odpověď 6 (protože '`␣je`'='`je␣`').

17-3-2 Popleta Truhlík — 10 bodů

Pan Truhlík byl veliký popleta. Nedokázal si zapamatovat, jak se jmenuje, kde pracuje, a často se mu stalo, že nepoznal svou vlastní dceru a ptal se jí: „Holčičko, nevíš, kde bydlím?“ (Zaměstnáním by se nejlépe hodil na matematika.)

Dokud žil s maminkou, bylo vše v pořádku, ale jakmile se odstěhoval z domu, začal mít se svou popleteností veliké problémy. A tak si řekl, že když by mamince občas zavolal, určitě by mu pomohla. Problém byl ale v tom, že i když si vzpomněl, že má maminku, nedokázal si vzpomenout na její telefonní číslo. Jednou se mu povedlo si celý problém uvědomit a svěřil se s ním prvnímu kolemjdoucímu, kterým byl zrovna myslitel Cibulka.

Pan Cibulka po chvíli hovoru zjistil, že Truhlík je sice veliký popleta (nebo pan Popletal je veliký truhlík?), ale pamatuje si všechny večerníčkové postavy. A tak vymyslel následující zlepšovák: místo telefonního čísla si bude Truhlík pamatovat větu, která se bude skládat ze jmen večerníčkových postav takovou, že když se napíše na klávesnici telefonu, vznikne chtěné číslo. Klávesnice telefonu vypadá následovně:

1	2	3
abc	de	fgh
4	5	6
ij	klm	no
7	8	9
pqr	st	uvw
	0	
	xyz	

(Číslo 7951140151651 jde zapsat jako „Rumcajz a Manka“.)

Jenomže Truhlík si sám takovou větu nedokáže vymyslet, Cibulku už o pomoc kvůli panu Popletalovi požádat nestihl, a tak zbýváte jen Vy.

Na vstupu dostanete seznam slov, které si pan Truhlík dokáže zapamatovat. Tato slova se skládají pouze ze znaků '`a`'..'`z`' a celková velikost slovníku je P písmen. Dále dostanete seznam N telefonních čísel, každé o délce C_i. Vaším úkolem je pro každé telefonní číslo najít posloupnost slov ze slovníku takovou, že pokud se napíše na klávesnici, vznikne kýžené telefonní číslo, případně říci, že to není možné.

Příklad: Jsou-li ve slovníku slova „brok, kuba, je“, číslo 5911425911 můžeme nahradit větou „kuba je kuba“, ale číslo 1765911 není možné žádnou větou složenou ze slovníkových slov nahradit.

Bonus: Pokud dokážete navíc vypsat pro každé telefonní číslo takovou větu, která má ze všech možných správných vět nejmenší počet slov, Truhlík se Vám určitě bohatě (bodově) odmění za to, že si ji zapamatuje brzy.

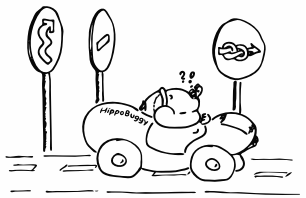
17-3-3 Starosta Hafák — 10 bodů

Pan Hafák se stal navzdory svému nevhodnému jménu starostou Kocourkova a jako starosta dostal za úkol starat se o bezpečnost chodců na silnicích. Nechal provést několik nezávislých odborných průzkumů a zjistil, že k největšímu počtu nehod dochází, když chodci přecházejí na zelenou. Rozhodl se tedy, že přikáže chodcům přecházet na červenou.

Slavný myslitel Cibulka mu ovšem vysvětlil, že takhle by rozhodně dopravních nehod neubylo, a poradil mu jiný způsob: pokud by všechny ulice v Kocourkově byly jednosměrné, bude šance, že nějakého chodce přejede auto, jenom poloviční.

To se Hafákovi velmi zalíbilo a ihned nechal udělal ze všech silnic jednosměrky. Při cestě domů ale zjistil, že to nebyl úplně dobrý nápad, protože už z první křižovatky, na kterou dojel, nevedla žádná silnice v jeho směru. A protože se Cibulka mezitím, mumlaje si nějaké nuly a jedničky, ztratil, budete muset Hafákovi poradit Vy.

Váš program dostane na vstupu popis silniční sítě v Kocourkově. Ta se skládá z N křižovatek a M silnic, každá silnice je obousměrná a spojuje dvě různé křižovatky. Žádné dvě silnice se mimo křižovatky nestýkají, mohou se ale mimoúrovňově křížit. Vaším úkolem je zjistit, kolik nejvýše silnic jde zjednosměrnit tak, aby bylo možné se ve výsledné zjednosměrněné síti dostat z nějaké křižovatky na jinou právě tehdy, když to šlo i v původní síti. A kromě počtu by měl Váš program vypsat, jak má zjednosměrněná síť vypadat.



Příklad: V síti $N = 3$, $M = 3$ se silnicemi spojujícími každé dvě křižovatky lze zjednosměrnit všechny tři silnice, výsledná síť bude vypadat třeba $(1 \to 2), (2 \to 3), (3 \to 1)$. Pokud by v síti byla ještě čtvrtá křižovatka, která by byla spojena silnicí s první křižovatkou, silnice mezi touto čtvrtou a první křižovatkou by zjednosměrnit nešla.

17-3-4 Myslitel Cibulka — 10 bodů

Poté, co jste snadno vyřešili problémy, které Vám myslitel Cibulka (vlastním jménem Filip Bonifác Narcis Cibulka) vymyslil, získali jste si jeho respekt, a tak se rozhodl obrátit se na Vás se svým vlastním problémem.

Cibulka si vymyslel zvláštní posloupnost čísel, kterou nazval po sobě Filipova Bonifácova Narcisova Cibulkova posloupnost. (Protože si to ale nikdo nemohl zamapatovat, zkrátil to na FiBoNaCiho.) Její první dva členy jsou 1 a 2 a každý další člen jest roven součtu dvou členů předchozích. Matematicky máme tedy posloupnost $\{F_n\}_{i=0}^{\infty}$, kde $F_0 = 1$, $F_1 = 2$ a $F_n = F_{n-1} + F_{n-2}$ pro $n \geq 2$.

Tato posloupnost se Cibulkovi tolik zalíbila, že se rozhodl zapisovat pomocí ní veškerá čísla, se kterými bude pracovat. Každé takto zapsané číslo je posloupnost nul a jedniček $a_n a_{n-1} \ldots a_1 a_0$ a jeho hodnota je $\sum_{i=0}^{n} a_i \cdot F_i$. Po krátké úvaze si uvědomil, že takový zápis nebude jednoznačný (třeba $011 = 100$), a proto vymyslil ještě *normalizovaný* zápis, který je stejný jako právě popsaný, jen se v něm nesmí vyskytnout dvě jedničky vedle sebe a nesmí začínat nulou (čili 11 ani 0100 není normalizované, ale 100 je).

Poté, co se dostatečně vychválil za svou genialitu, zjistil, že není schopen s takovými čísly vůbec pracovat. Už jenom je sečíst je veliký problém. A tak se nyní obrátil na Vás, zda byste mu nemohli vypomoci.

Zkuste napsat Cibulkovi program, který dostane na vstupu dvě čísla v nenormalizovaném FiBoNaCiho zápisu (tyto zápisy mají délky N a M) a vypíše zadaná čísla a jejich součet v normalizovaném tvaru. Není snad nutno dodávat, že skutečná hodnota zadaných čísel bude tak velká, že se nemůže vejít do žádného celočíselného typu (může jít o tisíce cifer ve FiBoNaCiho zápisu).

Příklad: Pro vstup 11101 a 1101 by měl Vás program vypsat $100101 + 10001 = 1001000$.

Hintík: Cibulka Vám ještě prozradil, že každé nezáporné číslo v normalizovaném tvaru zapsat jde.

17-3-5 Jazykozpytcova naděje — 9 bodů

Na jisté nejmenované univerzitě vědecky působil a samozřejmě též vyučoval nadšený lingvista, pan Choam Nomsky. Svým studentům často zadával domácí úkoly a nejčastějším úkolem bylo sestrojení konečného automatu, který má rozpoznávat nějaký zadaný jazyk. (Pro nezasvěcené: pokud netušíte, o čem je řeč, nahlédněte do zadání první série, kde jsou potřebné pojmy definovány.)

Jenže jeho studenti nepatřili zrovna k nejbystřejším a často nosili automaty, které používaly obrovské množství stavů, i když jazyk šlo rozpoznávat automatem s podstatně méně stavy. Kontrola správnosti obrovských automatů způsobovala panu Nomskému četné vrásky a bolesti hlavy, rozhodl se tedy, že před kontrolou správnosti si musí automat zjednodušit. Za zjednodušený automat, říkejme mu odborně *redukovaný*, považoval takový automat (Q, A, δ, q_0, F) ekvivalentní s původním, který neměl žádné nedosažitelné stavy a žádné dva stavy nebyly ekvivalentní. Co to znamená:

- Stav q je *nedosažitelný*, pokud neexistuje slovo u nad abecedou A, že by pro něj výpočet skončil ve stavu q.
- Dva různé stavy p a q jsou *ekvivalentní*, pokud pro každé slovo u nad abecedou A platí následující: výpočet nad slovem u startující ze stavu p skončí přijetím slova u, právě když výpočet nad slovem u startující ze stavu q skončí přijetím slova u.

O redukovaných automatech se dají dokázat některé zajímavé skutečnosti, například že libovolné dva ekvivalentní automaty se zredukují na stejně velký a „stejně vypadající“ automat, ale tím vás tentokrát zatěžovat nebudeme. Nyní po vás nechceme nic snazšího, než vymyslet co nejefektivnější algoritmus a posléze napsat program, který panu Choamu Nomskému usnadní jeho úděl, čili ke konečnému automatu zadanému na vstupu najde příslušný redukovaný automat (což vlastně není nic jiného než ekvivalentní automat s co nejmenším počtem stavů).

Program nejprve načte z první řádky vstupu počet stavů n (očíslujme si je tedy 1 až n), počet symbolů abecedy a (taktéž si je očíslujme 1 až a), číslo počátečního stavu p a počet přijímacích stavů f. Následuje řádek s f čísly, které udávají přijímací stavy. Pak je na vstupu n řádků, každý s a čísly. Číslo c umístěné v i-tém řádku a j-tém sloupci znamená, že $\delta(i, j) = c$. Program by měl na výstup vypsat redukovaný automat v podobném formátu.

Příklad: vstup je vlevo, vzorový výstup vpravo.

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

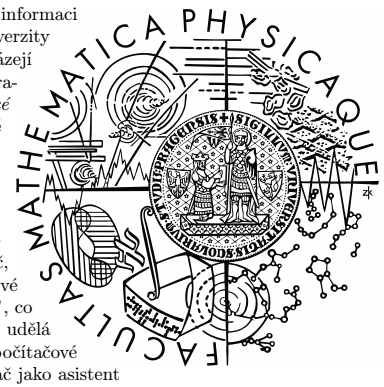
Recepty z programátorské kuchařky

V dnešním vydání známého bestselleru budeme péci grafy souvislé i nesouvislé, orientované i neorientované, ba i rovinné. Řekneme si o základním procházení grafem, komponentách souvislosti, topologickém uspořádání a dalších

Výsledková listina sedmnáctého ročníku KSP po první sérii

		škola	ročník	1711	1712	1713	1714	1715	suma	celkem
1.	Peter Černo	GLŠtúra	4	10	10	10	10	10	50	50
2.	Ondřej Bílka	G Zlín	3	10	10	9	9	10	48	48
3.	Miroslav Klimoš	G Lanškr	0	10	8	8	10	10	46	46
4.	Miroslav Cicko	GJGTajov	4	10	4	10	10	10	44	44
5.	Peter Perešíni	GJGTajov	3	10	10	7	6	7	40	40
6.	Zbyněk Konečný	GKptJaroš	2	10	1	5	10	10	36	36
7.	Martin Koníček	G UBrod	4	11			10	10	31	31
8. – 9.	Jan Pelc	G UBrod	3	6	1	6	6	10	29	29
	Josef Pihera	G Strakon	2	0	3	8	10	8	29	29
10. – 13.	Stanislav Basovník	G Kroměříž	4		10			10	20	20
	Pavel Klavík	G Chrudim	2	1	1	2	9	7	20	20
	Petr Kratochvíl	G SvětláNS	2	5	3	3	7	2	20	20
	Lukáš Lánský	GJKTyla	1	0		2	8	10	20	20
14.	Jan Bulánek	G Klatovy	4		10	6		3	19	19
15.	Zbyněk Falt	GNeumannov	4	10	3	5			18	18
16.	Adam Zivner	G UBrod	3	4	1		9	3	17	17
17. – 18.	Tomáš Herceg	G Třebíč	2	10	1	2	3		16	16
	Josef Špak	GJírovco	2	5	1			10	16	16
19. – 20.	Jiří Cabal	SPŠ DvKrál	2	5	1	2	6	1	15	15
	Martin Čech	G UBrod	4	10				5	15	15
21. – 23.	Jan Hrnčíř	GFXŠaldy	3	5	1	4	4		14	14
	Jan Palenčar	G Martin	2	4		10			14	14
	Lukáš Špalek	G Čadca	4	5	1	2	6		14	14
24.	Martin Kupec	GMendel	3	1	1	3	3	5	13	13
25. – 27.	Ondřej Garncarz	G Příbor	4	6		2	4		12	12
	Martin Podloucký	G Strážnic	4	1	1			10	12	12
	Roman Smrž	GOhradní	1	6		6			12	12
28. – 30.	Jakub Jenis	GsvCyrMet	1	3		1		7	11	11
	Jakub Kaplan	GJKTyla	1	6	1	4			11	11
	Ján Zahornadský	GZborov	4	10	1				11	11
31. – 35.	Lukáš Beleš	G Čadca	4	5	1		4		10	10
	Jakub Benda	GJNerudy	2	10					10	10
	Ondřej Bouda	GKptJaroš	2	10					10	10
	Martin Kahoun	GJNerudy	2	10					10	10
	Michal Vaner	G Turnov	3	10					10	10
36.	Marian Kaluža	GHavlíčkov	2	5	0	1	2	1	9	9
37. – 38.	Jiří Machálek	G Holešov	3	6			2		8	8
	Petr Soběslavský	GJHeyrovs	4	6	1		1		8	8
39. – 40.	Daniel Sedláček	SPŠE Hav	1	6				1	7	7
	Filip Šauer	G Klatovy	4	1			6		7	7
41. – 43.	Jiří Nohavec	G Domažl	4	4		2			6	6
	Michal Pavelčík	G UBrod	2	5	1				6	6
	Eva Schlosáriková	G Piešťany	4	1	2	1	2		6	6
44. – 46.	Cyril Hrubiš	G Bílovec	3	1				4	5	5
	Jakub Porod	G Týn nV	2	5					5	5
	Jan Staněk	GKptJaroš	3	3		2			5	5
47.	Zdeněk Vilušínský	G Turnov	4				3		3	3
48. – 51.	Tomáš Ehrlich	G Holešov	2	0	1	1			2	2
	Petr Musil	G MBuděj	3					2	2	2
	Adam Ráž	GBudějo	2	1	1				2	2
	Martin Vařák	G Bílovec	2	1	1				2	2
52. – 55.	Florián Danko	SPŠEtech	2			1			1	1
	Hana Klempová	GUBalvanJN	4	0	1				1	1
	Tamara Kuštárová	GBiling	0	0		1			1	1
	Petr Zimčík	G UBrod	1			1			1	1
56. – 57.	Miroslav Hovorka	GJateční	4	0					0	0
	Adrián Lachata	G Svidník	3	0		0	0		0	0

Studium informatiky na MFF UK Praha



Milí řešitelé, se zadáním 1. a 2. série úloh jste od nás dostali také přehlednou informaci o tom, jak vypadá studium informatiky na Matematicko-fyzikální fakultě Univerzity Karlovy v Praze. Už jsme Vás seznámili s dvěma pracovišti, které se nacházejí ve čtvrtém patře budovy na Malostranském náměstí 25 v Praze 1, a to s pracovišti *Kabinet software a výuky informatiky (KSVI)* a *Středisko informatické sítě a laboratoří (SISAL)*. Kromě nich se na čtvrtém patře nachází další dvě spřátelená pracoviště, *Ústav formální a aplikované lingvistiky (ÚFAL)* a *Centrum komputační lingvistiky (CKL)*, která jsou k sobě vzájemně připoutána počítačovou (komputační) lingvistikou. Pokud se ptáte *Co je to počítačová lingvistika?* a *Proč zrovna na matfyzu?*, tak se ptáte velmi dobře. Lingvistika je věda o jazyce, tedy jazykověda. Takto osamocená je vědou čistě teoretickou, která studuje strukturu vybraného jazyka. Když si k ní přidáme počítač, tak to neznamená, že by počítač sám od sebe studoval strukturu jazyka. Ze své podstaty počítač sám od sebe nikdy nic neudělá. Vždy mu musíme "vysvětlit", co má udělat. Na druhou stranu, když už počítač ví, co a jak má dělat, tak to udělá "hrozně" rychle. Nejinak je tomu i se studováním struktury jazyka, které se v počítačové lingvistice převádí na úlohy aplikačního charakteru - strojový překlad (počítač jako asistent překladatele), vyhledávání v textech, rozpoznávání mluvené řeči, větný rozbor, aj.

Na pracovištích ÚFAL a CKL spolu pracují informatici a lingvisté. Skládají dohromady znalosti charakteru informatického (algoritmy, datové struktury, výpočetní složitost) a lingvistického (tvarosloví jazyka, větná stavba, význam), aby společnými silami "vysvětlili" počítači, jak má např. správně přeložit anglickou větu *Time flies like an arrow.* do češtiny. Zkuste to také! Pokud se vám to podařilo, tak si udělejte větný rozbor věty *Chytal tlouště na višni.* a popřemýšlejte, jak byste vysvětlili počítači, aby to udělal za vás, pokud možno správně.

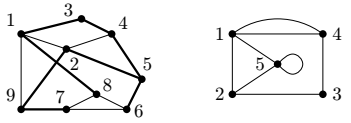
Mohlo by se zdát, že pracoviště jsou jenom výzkumná a ne pedagogická, ale není tomu tak. Pracoviště nabízejí přednášky a semináře, které pokrývají magisterské i doktorské studium, takže pokud budete studovat na naší fakultě, jistě se s nimi setkáte.

Příště budeme pokračovat konečně o patro níže a seznámíme se s *Katedrou aplikované matematiky (KAM)* a *Institutem teoretické informatiky (ITI)*.

grafových algoritmech. Abychom ale mohli začít, musíme si nejprve říci, s čím budeme pracovat.

Ingredience

Neorientovaný graf je určen množinou vrcholů V a množinou hran, což jsou neuspořádané dvojice vrcholů. Hrana $e = x, y$ spojuje vrcholy x a y. Většinou požadujeme, aby hrany nespojovaly vrchol se sebou samým (takovým hranám říkáme *smyčky*) a aby mezi dvěma vrcholy nevedla více než jedna hrana (pokud toto neplatí, mluvíme o *multigrafech*). Neorientovaný graf většinou zobrazujeme jako body pospojované čarami.



Neorientovaný graf a jeho kostra; multigraf

Podgrafem grafu G rozumíme graf G', který vznikl z grafu G vynecháním některých (a nebo žádných) hran a vrcholů.

Často nás zajímá, zda se dá z vrcholu x dojít po hranách do vrcholu y. Ovšem slovo „dojít“ by mohlo být trochu zavádějící, proto si zavedeme pár pojmů:

- *sled* budeme říkat takové posloupnosti vrcholů a hran tvaru $v_1, e_1, v_2, e_2, \ldots, e_{n-1}, v_n$, že $e_i = \{v_i, v_{i+1}\}$ pro každé i. Sled je tedy nějaká procházka po grafu. Délku sledu měříme počtem hran v této posloupnosti.
- *tah* je sled, ve kterém se neopakují hrany, tedy $e_i \neq e_j$ pro $i \neq j$.
- *cesta* je sled, ve kterém se neopakují vrcholy, čili $v_i \neq v_j$ pro $i \neq j$. Všimněte si, že se nemohou opakovat ani hrany.

Lehce nahlédneme, že pokud existuje sled z vrcholu x do y ($v_1 = x$, $v_n = y$), pak také existuje cesta z vrcholu x do vrcholu y. Každý sled, který není cestou, obsahuje nějaký vrchol u dvakrát, nechť $u = v_i = v_j$, $i < j$. Z takového sledu ale můžeme vypustit posloupnost $e_i, v_{i+1}, \ldots, e_{j-1}, v_j$ a dostaneme také sled spojující v_1 a v_n, který je určitě kratší než původní sled. Tak můžeme po konečném počtu úprav dospět až ke sledu, který neobsahuje žádný vrchol dvakrát, tedy k cestě.

Kružnicí nazýváme cestu délky alespoň 3, ve které oproti definici platí $v_1 = v_n$. Někdy se na cesty, tahy a kružnice v grafu také díváme jako na podgrafy, které získáme tak, že z grafu vypustíme všechny ostatní vrcholy a hrany.

Ještě si ukážeme, že pokud existuje cesta z vrcholu a do vrcholu b a z vrcholu b do vrcholu c, pak také existuje cesta z vrcholu a do vrcholu c. To vyplývá z faktu, že existuje sled z vrcholu a do vrcholu c, který můžeme dostat například tak, že spojíme za sebe cesty z a do b a z b do c. A jak jsme si ukázali, když existuje sled z a do c, existuje i cesta z a do c.

V mnoha grafech (například v těch na předchozím obrázku) je každý vrchol dosažitelný cestou z každého. Takovým grafům budeme říkat *souvislé*. Pokud je graf nesouvislý, můžeme ho rozložit na části, které již souvislé jsou a mezi kterými nevedou žádné další hrany. Takové podgrafy nazýváme *komponentami souvislosti*.

Teď se podívejme na pár pojmů z přírody: *Strom* je souvislý graf, který neobsahuje kružnici. *List* je vrchol, ze kterého vede pouze jedna hrana. Ukážeme, že každý strom s alespoň dvěma vrcholy má nejméně dva listy. Proč to? Stačí si najít nejdelší cestu (pokud je takových cest více, zvolíme libovolnou z nich). Oba koncové vrcholy této cesty musí být nutně listy: kdyby z některého z nich vedla hrana, musela by vést do vrcholu, který na cestě ještě neleží (jinak by ve stromu byla kružnice), ale o takovou hranu bychom cestu mohli prodloužit, takže by původní cesta nebyla nejdelší.

Grafům bez kružnic budeme obecně říkat *lesy*, jelikož každá komponenta souvislosti takového grafu je strom.

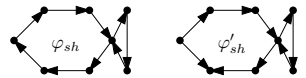
Les, jak ho vidí matematici

Někdy se hodí jeden z vrcholů stromu prohlásit za *kořen*, čímž jsme si v každém vrcholu určili směr nahoru (ke kořeni – je to zvláštní, ale grafoví teoretici obvykle kreslí stromy kořenem vzhůru) a dolů (od kořene). Souseda vrcholu směrem nahoru pak nazýváme jeho *otcem*, sousedy směrem dolů jeho *syny*.

Kostra souvislého grafu je strom, který spojuje všechny vrcholy. Pro nesouvislé grafy nazveme kostrou les tvořený kostrami jednotlivých komponent. Na prvním obrázku je kostra levého grafu znázorněna silnými hranami.

Orientované grafy

Často potřebujeme, aby hrany byly pouze jednosměrné. Takovému grafu říkáme *orientovaný graf*. Hrany jsou nyní uspořádané dvojice vrcholů (x, y) a říkáme, že hrana vede z vrcholu x do vrcholu y. Hrany (x, y) a (y, x) jsou tedy dvě různé hrany (i když se mohou vyskytovat v grafu obě najednou). Orientovaný graf většinou zobrazujeme jako body spojené šipkami. Většina pojmů, které jsme definovali pro neorientované grafy, platí i pro grafy orientované, jen si musíme dát pozor na směr hran. Kružnici v orientovaném grafu často nazýváme *cyklem*.



Silně a slabě souvislý orientovaný graf

Se souvislostí orientovaných grafů je to trochu složitější. Rozlišujeme *slabou a silnou souvislost*. Slabě souvislý je graf tehdy, když zapomeneme-li na orientaci hran, dostaneme souvislý orientovaný graf. Silně souvislým ho nazveme tehdy, vede-li mezi každými dvěma vrcholy x, y ($x \neq y$) orientovaná cesta v obou směrech. Pokud je graf silně souvislý, je i slabě souvislý, ale jak ukazuje náš obrázek, opačně to platit nemusí.

Komponenta silné souvislosti orientovaného grafu G je takový podgraf G', který je silně souvislý a není podgrafem žádného většího silně souvislého podgrafu grafu G. Komponenty silné souvislosti tedy mohou být mezi sebou propojeny, ale žádné dvě nemohou ležet na společném cyklu.

Reprezentace grafů

Nyní už víme o grafech hodně, ale ještě jsme si neřekli, jak graf reprezentovat v paměti počítače. Nejčastější jsou tyto dva způsoby:

- *matice sousednosti* – to je pole A velikosti $N \times N$ (kde N je počet vrcholů). Na pozici $A[i,j]$ uložíme hodnotu 0 nebo 1 podle toho, zda z vrcholu i do vrcholu j vede hrana (1) nebo nevede (0). S maticí sousednosti se zachází velmi snadno, ale má tu nevýhodu, že je vždy kvadraticky velká bez ohledu na to, kolik je hran. Výhodou naopak je, že místo jedniček můžeme ukládat nějaké další informace o hranách, třeba jejich délky. Vpravo od tohoto odstavce najdete matici sousednosti grafu z prvního obrázku.

```
011000001
100110001
100100000
011010000
010101000
000010110
000001011
100001100
110000100
```

- *seznam sousedů* se obvykle zapisuje dvěma poli: polem hran E, do kterého uložíme všechny hrany tak, aby hrany vedoucí z jednoho vrcholu tvořily souvislý úsek, a polem vrcholů V, které pro každý vrchol udává začátek odpovídajícího úseku v poli E. Pokud do $V[N+1]$ uložíme $M+1$, kde M je počet hran, platí, že hrany vycházející z vrcholu i jsou uloženy v $E[V[i]], \ldots, E[V[i+1]-1]$. Tato reprezentace má tu výhodu, že má velikost pouze $O(N+M)$ a sousedy každého vrcholu máme pěkně pohromadě a nemusíme je hledat. Pro graf z 1. obrázku:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$E[i].a$	1	1	1	2	2	2	2	2	3	3	4	4	4	5	5	5	6	6	6	7	7	7	8	8
$E[i].b$	2	3	8	9	1	4	5	9	1	4	2	3	5	2	4	6	5	7	8	6	8	9	1	6

i	25	26	27	28	29
$E[i].a$	8	8	9	9	9
$E[i].b$	7	1	2	7	

i	1	2	3	4	5	6	7	8	9	10
$V[i]$	1	5	9	11	14	17	20	23	26	29

Reprezentace grafu seznamem sousedů

Recepty

Naše povídání o grafových algoritmech začneme dvěma základními způsoby procházení grafem. K tomu budeme potřebovat dvě podobné jednoduché datové struktury: *Fronta* je konečná posloupnost prvků, která má označený začátek a konec. Když do ní přidáváme nový prvek, přidáme ho na konec posloupnosti. Když z ní prvek odebíráme, odebereme ten na začátku. Proto se tato struktura anglicky nazývá *first in, first out*, zkráceně *FIFO*. *Zásobník* je také konečná posloupnost prvků se začátkem a koncem, ale prvky přidáváme a odebíráme z konce zásobníku. Anglický název je (překvapivě) *last in, last out*, čili *LIFO*.

Algoritmus prohledávání grafu do hloubky:

1. Na začátku máme v zásobníku pouze vstupní vrchol w. Dále si u každého vrcholu v pamatujeme značku z_v, která říká, zda jsme vrchol již navštívili. Vstupní vrchol je označený, ostatní vrcholy nikoliv.
2. Odebereme vrchol ze zásobníku, nazvěme ho u.
3. Každý neoznačený vrchol, do kterého vede hrana z u, přidáme na zásobník a označíme.
4. Body 2 a 3 opakujeme, dokud není zásobník prázdný.

Na konci algoritmu budou označeny všechny vrcholy dosažitelné z vrcholu w, tedy v případě neorientovaného grafu celá komponenta souvislosti obsahující w. To můžeme snadno dokázat sporem: Předpokládáme, že existuje vrchol x, který není označen, ale do kterého vede cesta z w. Pokud je takových vrcholů více, vezmeme si ten nejbližší k w. Označme si y předchůdce vrcholu x na nejkratší cestě z w; y je určitě označený (jinak by x nebyl nejbližší neoznačený). Vrchol y se tedy musel někdy objevit na zásobníku, tím pádem jsme ho také museli ze zásobníku odebrat a v kroku 3 označit všechny jeho sousedy, tedy i vrchol x, což je ovšem spor.

To, že algoritmus někdy skončí, nahlédneme snadno: v kroku 3 na zásobník přidáváme pouze vrcholy, které dosud nejsou označeny, a hned je značíme. Proto se každý vrchol může na zásobníku objevit nejvýše jednou, a jelikož v bodě 2 pokaždé odebereme jeden vrchol ze zásobníku, musí vrcholy někdy (konkrétně po nejvýše N opakováních cyklu) dojít. V bodu 3 probereme každou hranu grafu nejvýše dvakrát (v každém směru jednou). Časová složitost celého algoritmu je tedy lineární v počtu vrcholů N a počtu hran M – $O(N+M)$. Paměťová složitost je stejná, protože si musíme hrany a vrcholy pamatovat.

Nejjednodušší implementace prohledávání do hloubky je rekurzivní funkcí. Jako zásobník v tom případě požíváme přímo zásobník programu, kde si program ukládá návratové adresy funkcí. Může to vypadat třeba následovně:

```
var Hrany: array[1..MaxN + 1] of Integer;
  Sousedi: array[1..MaxM] of Integer;
  Oznacen: array[1..MaxN] of Boolean;

procedure Projdi(V: Integer);
var I: Integer;
begin
  Oznacen[V]:= True;
  for I:= Hrany[V] to Hrany[V + 1] do
    if not Oznacen[Sousedi[I]] then
      Rekurze(Sousedi[I]);
end;
```

Rozdělit neorientovaný graf na komponenty souvislosti je pak už jednoduché. Projdeme postupně všechny vrcholy grafu a pokud nejsou v žádné z dosud označených komponent grafu, přidáme novou komponentu tak, že graf z tohoto vrcholu prohledáme do hloubky. Vrcholy značíme přímo číslem komponenty, do které patří. Protože prohledáváme do hloubky několik oddělených částí grafu, každou se složitostí $O(N_i + M_i)$, kde N_i a M_i je počet vrcholů a hran komponenty, vyjde dohromady složitost $O(N+M)$. Nic nového si ukládat nemusíme, a proto je paměťová složitost také $O(N+M)$.

```
var Komponenta: array[1..MaxN] of Integer;
  Hrany: array[1..MaxN + 1] of Integer;
  Sousedi: array[1..MaxM] of Integer;
  NovaKomponenta: Integer;

procedure Projdi(V: Integer);
var I: Integer;
begin
  Komponenta[V]:= NovaKomponenta;
  for I:= Hrany[V] to Hrany[V + 1] do
    if (Komponenta[Sousedi[I]] = -1) then
      Rekurze(Sousedi[I]);
end;
```

```
    for (int i=P, j=N; i>0; i--)
        if (batoh[i][j] > batoh[i-1][j]) {          /* předmět i je v batohu, vylepšil celkovou cenu */
            printf ("␣%d", i);
            j -= m[i];
        }
    return 0;
}
```

Úloha 17-1-4 – Paloučkova výhra – program

```
var N:integer;                                      {počet měst}
    D,E:array[1..N] of array[1..N] of integer;      {D - délky silnic mezi městy,
                                                     D[i][i]="nekonečno",
                                                     místo neexistujících je také "nekonečno",
                                                     E - nastaveny na 0}
    P:array[1..N] of integer;                       {posloupnost křižovatek v min kružnici, končí nulou}
    i,j,k,min_i,min_j,min:integer;
begin
  Nacti_a_inicializuj(E,D,N);
  min:=maxint;                                      {zatím žádná kružnice}
  for k:=3 to N do begin                            {S=(1..k-1)}
    for i:=1 to k-2 do                              {hledám nejmenší kružnice obsahující k}
      for j:=i+1 to k-1 do
        if (D[i][k]+D[k][j]+D[i][j]<min) then       {i-k a j-k musí být přímo hrany}
        begin
          min:=D[i][k]+D[k][j]+D[i][j];
          min_i:=i;
          min_j:=j;
        end;
    Pamatuj(P,E,i,j,k);                             {do P si zapamatuj si novou min. kružnici z i do j s k}
    for i:=1 to k-1 do                              {podle upraveného F-W zjisti nové lepší cesty}
      for j:=1 to k-1 do begin
        if D[i][k]+D[k][j]<D[i][j] then begin D[i][j]:=D[i][k]+D[k][j]; E[i][j]:=k; end;
        if D[i][j]+D[j][k]<D[i][k] then begin D[i][k]:=D[i][j]+D[j][k]; E[i][k]:=MAX(E[i][j],j); end;
      end;
  end;
  Vypis(P,min);                                     {pokud min="nekonečno", žádná kružnice neexistuje}
end.
```

```
      queue[w++] = i;
   while (r < w) {                                  /* Procházíme frontu */
      i = queue[r++];
      for (j=first[i]; j; j=next[j])                /* ... všechny hrany z vrcholu i */
         if (--deg[dest[j]] == 5)                   /* Nový vrchol nízkého stupně */
            queue[w++] = dest[j];
   }
   if (r != N) {
      puts ("Graf nebyl rovinný, to není fair!");
      exit (1);
   }
}
void paint (void) {                                 /* Obarví vrcholy */
   int r, i, j;
   int avail[7];                                    /* Které barvy jsou zatím volné */
   for (r=N-1; r>=0; --r) {                         /* Projdeme frontu pozpátku */
      i = queue[r];
      for (j=1; j<=6; j++)                          /* Zatím jsou všechny barvy volné */
         avail[j] = 1;
      for (j=first[i]; j; j=next[j])                /* Mimo barev použitých sousedy */
         avail[color[dest[j]]] = 0;
      j = 1;                                        /* Zvolíme první nepoužitou */
      while (!avail[j])
         j++;
      color[i] = j;
   }
   for (i=0; i<N; i++)
      printf ("Ve městě %d se bude mluvit jazykem %d.\n", i, color[i]);
}
int main (void)
{
   melolontha ();
   scan ();
   paint ();
```

Úloha 17-1-3 – Chmatákův lup – program

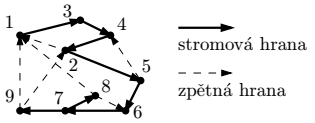
```
#include <stdio.h>

#define maxP 100
#define maxN 100
#define MAX (a, b)  ( ( (a)> (b)) ? (a): (b))

int main () {
        int N;                                      /* kolik zloděj unese */
        int P;                                      /* počet předmětů */
        int m[maxP];                                /* hmotnosti předmětů */
        float c[maxP];                              /* ceny předmětů */

        /* maximální hodnota lupu z prvních p předmětů v batohu s kapacitou n */
        float batoh[maxP][maxN];

        scanf ("%d %d", &N, &P);
        for (int i=1; i<=P; i++)
                scanf ("%d %f", &m[i], &c[i]);

        for (int j=0; j<=N; j++)
                batoh[0][j] = 0;

        /* postupně přidáváme předměty */
        for (int i=1; i<=P; i++) {
                for (int j=0; j<m[i]; j++)          /* nízké hmotnosti jenom zkopírujeme */
                        batoh[i][j] = batoh[i-1][j];
                for (int j=m[i]; j<=N; j++)         /* pokud lze, zlepšíme i-tým předmětem */
                        batoh[i][j] = MAX (batoh[i-1][j], batoh[i-1][j-m[i]]+c[i]);
        }

        printf ("cena: %f\n", batoh[P][N]);
        printf ("předměty:");
```



```
var I: Integer;
begin
   ...
   for I:= 1 to N do Komponenta[I]:= -1;
   NovaKomponenta:= 1;
   for I:= 1 to N do
      if Komponenta[I] = -1 then
         begin
            Projdi(I);
            Inc(NovaKomponenta);
         end;
   ...
end.
```

Průběh prohledávání grafu do hloubky můžeme znázornit stromem. Z počátečního vrchol w učiníme kořen. Pak budeme graf procházet do hloubky a vrcholy zakreslujeme jako syny vrcholů, ze kterých jsme přišli. Hranám mezi těmito vrcholy budeme říkat *stromové hrany*. Protože jsme do žádného vrcholu nešli dvakrát, vznikne nám strom. Hrany, které vedou do již navštívených vrcholů na cestě, kterou jsme přišli z kořene, jsou tzv. *zpětné hrany. Dopředné hrany* vedou naopak z vrcholu blíže kořeni do už označeného vrcholu dále od kořene. A konečně *příčné hrany* vedou mezi dvěma různými podstromy grafu.

Všimněte si, že při prohledávání neorientovaného grafu objevíme každou hranu dvakrát: buďto poprvé jako stromovou a podruhé jako zpětnou, a nebo jednou jako zpětnou a podruhé jako dopřednou. Příčné hrany se objevit nemohou – pokud by příčná hrana vedla „doprava“, vedla by do dosud neoznačeného vrcholu, takže by se prohledávání vydalo touto hranou a nevznikl by oddělený podstrom; „doleva“ rovněž vést nemůže: představme si stav prohledávání v okamžiku, kdy jsme opouštěli levý vrchol této hrany. Tehdy by naše hrana musela být příčnou vedoucí doprava, ale o té už víme, že neexistuje.



Strom prohledávání do hloubky

Prohledávání do hloubky lze tedy využít na hledání kostry neorientovaného grafu, což je strom, který jsme prošli, a rovnou při tom také zjistíme, zda graf neobsahuje cyklus, což je v případě, kdy nalezneme zpětnou hranu různou od té stromové, kterou jsme do vrcholu přišli.

Pro orientované grafy je opět situace složitější: stromové a dopředné hrany jsou orientované vždy ve stromě shora dolů, zpětné zdola nahorů a příčné hrany mohou existovat, ovšem vždy vedou „zprava doleva“, tedy pouze do podstromů, které jsme již prošli (nahlédneme opět stejně).

Prohledávání do šířky

Prohledávání do šířky je založené na trochu jiné myšlence a narozdíl od prohledávání do šírky používá jinou datovou strukturu, a to frontu.

1. Na začátku máme ve frontě pouze jeden prvek, a to zadaný vrchol w. Dále si u každého vrcholu x pamatujeme číslo $H[x]$. Všechny vrcholy budou mít na začátku $H[x] = -1$, jen $H[w] = 0$.
2. Odebereme vrchol z fronty, označme ho u.
3. Každý vrchol v, do kterého vede hrana z u a jeho $H[v] = -1$, přidáme do fronty a dáme nastavíme jeho $H[v]$ na $H[u] + 1$.
4. Body 2 a 3 opakujeme, dokud není fronta prázdná.

Podobně jako u prohledávání do hloubky jsme se dostali právě do těch vrcholů, do kterých vede cesta z w (a označili jsme je nezápornými čísly). Rovněž je každému vrcholu přiřazeno nezáporné číslo maximálně jednou.

Vrcholy se stejným číslem tvoří ve frontě jeden souvislý celek, protože nejprve odebereme z fronty všechny vrcholy s číslem n, než začneme odebírat vrcholy s číslem $n + 1$. Navíc platí, že $H[v]$ udává délku nejkratší cesty z vrcholu w do x. Že neexistuje kratší cesta, dokážeme sporem: Pokud existuje nějaký vrchol v, pro který $H[v]$ neodpovídá délce nejkratší cesty z w do v, čili vzdálenosti $D[v]$, vybereme si z takových v to, jehož $D[v]$ je nejmenší. Pak nalezneme nejkratší cestu z w do v a předposlední vrchol z této cesty. Vrchol z je bližší než v, takže pro něj už musí být $D[z] = H[z]$. Ovšem když jsme z fronty vrchol z odebírali, museli jsme objevit i jeho souseda v, který ještě nemohl být označený, tudíž jsme mu museli přidělit $H[v] = H[z] + 1 = D[v]$, což je spor.

Prohledávání do šířky má časovou složitost taktéž lineární s počtem hran a vrcholů. Na každou hranu se také ptáme dvakrát. Fronta má lineární velikost k počtu vrcholů, takže jsme si oproti prohledávání do hloubky nepohoršili a i paměťová složitost je $O(N+M)$. Algoritmus implementujeme nejsnáze cyklem, který bude pracovat s vrcholy v poli, které nám bude představovat frontu.

```
var Fronta, Delka: array[1..MaxN] of Integer;
  Oznacen: array[1..MaxN] of Boolean;
  Hrany: array[1..MaxN + 1] of Integer;
  Sousedi: array[1..MaxM] of Integer;
  I, Prvni, Posledni, PocatecniVrchol: Integer;
begin
  ...
  Prvni:= 1;
  Posledni:= 1;
  Fronta[Prvni]:= PocatecniVrchol;
  Delka[Prvni]:= 0;

  repeat
  for I:= Hrany[Fronta[Prvni]] to
                Hrany[Fronta[Prvni]+1]-1 do
    if not Oznacen[Sousedi[I]] then begin
      Oznacen[Sousedi[I]]:= True;
      Delka[Sousedi[I]]:=Delka[Fronta[Prvni]]+1;
      Inc(Posledni);
      Fronta[Posledni]:= Sousedi[I];
    end;
  Inc(Prvni);
  until Prvni > Posledni;  {Fronta je prazdna}
  ...
end.
```

Prohledávání do šířky lze také použít na hledání komponent souvislosti a hledání kostry grafu.

Topologické uspořádání

Teď si vysvětlíme, co je to *topologické uspořádání*. Máme orientovaný graf G s N vrcholy a chceme očíslovat vrcholy čísly 1 až N tak, aby všechny hrany vedly z vrcholu s větším číslem do vrcholu s menším číslem, tedy pro každou hranu

$e = (v_i, v_j)$ platí $i > j$. Představme si ho jako srovnání vrcholů grafu na přímku tak, aby „šipky“ vedly pouze zprava doleva.

Nejprve si ukážeme, že pro žádný orientovaný graf, který obsahuje cyklus, nelze takovéto topologické pořadí vytvořit. Označme vrcholy tohoto cyklu $v_1, \ldots, v_n$, takže hrana vede z vrcholu v_i do vrcholu v_{i-1}, resp z v_1 do v_n. Pak vrchol v_2 musí dostat vyšší číslo než vrchol v_1, v_3 než $v_2, \ldots, v_n$ než v_{n-1}. Ale vrchol v_1 musí mít zároveň vyšší číslo než v_n, což je spor.

Pokud graf cyklus neobsahuje, lze ho vždycky topologicky uspořádat. Zde je algoritmus:

1. Na začátku máme orientovaný graf G a proměnnou $p = 1$.
2. Najdeme takový vrchol v, ze kterého nevede žádná hrana. Pokud žádný takový není, výpočet končí, protože jsme našli cyklus.
3. Odebereme z grafu vrchol v a všechny hrany, které do něj vedou.
4. Přiřadíme vrcholu v číslo p.
5. Proměnnou p zvýšíme o 1.
6. Opakujeme kroky 2 až 5, dokud graf obsahuje alespoň jeden vrchol.

Nejprve si ukážeme, že neprázdný graf, který neobsahuje cyklus, vždy obsahuje vrchol, ze kterého nevede žádná hrana. Pro spor předpokládejme, že žádný takový vrchol neexistuje. Pak si vyberme libovolný vrchol v_1. Z něj vede hrana do dalších vrcholů, vybereme jeden z nich a označme ho v_2. Z v_2 vybereme další hranu a takto pokračujeme. Protože je vrcholů konečný počet, dospějeme k jednomu z těchto případů:

- Z některého vrcholu v_i nevede žádná hrana.
- Některé dva vrcholy v_i, v_j jsou stejné, a graf tedy obsahuje cyklus.

Což je spor s našimi předpoklady. Graf G má konečně mnoho vrcholů a protože v bodě 3 pokaždé odebereme další vrchol grafu, musí algoritmus skončit. Z vrcholu v, který přidáváme do posloupnosti, nevedou žádné hrany, a proto může mít nižší číslo než zbývající vrcholy grafu. To platí pro každý takový vrchol v, a proto je uspořádání korektní.

Algoritmus můžeme snadno upravit tak, aby netratil příliš času hledáním vrcholů, z nichž nic nevede – stačí si takové vrcholy pamatovat ve frontě, a kdykoliv nějaký takový vrchol odstraňujeme, zkontrolujeme si, zda jsme nějakému jinému vrcholu nezrušili poslední hranu, která z něj vedla, a pokud ano, přidáme takový vrchol na konec fronty. Celé topologické třídění pak zvládneme v čase $O(N + M)$.

Také můžeme graf prohledat do hloubky a všimnout si, že pořadí, ve kterém jsme se z vrcholů vraceli, je právě topologické pořadí. Pokud právě opouštíme nějaký vrchol a číslujeme ho dalším číslem v pořadí, rozmysleme si, jaké druhy hran z něj mohou vést: stromová nebo dopředná hrana vede do vrcholu, kterému jsme již přiřadili vyšší číslo, zpětná existovat nemůže (v grafu by byl cyklus) a příčné hrany vedou pouze zprava doleva, takže také do již očíslovaných vrcholů. Časová složitost je opět $O(N + M)$.

```
var Hrany: array[1..MaxN + 1] of Integer;
  Sousedi: array[1..MaxM] of Integer;
  Ocislovani: array[1..MaxN] of Integer;
  Posledni: Integer;
  I: Integer;

procedure Projdi(V: Integer);
var I: Integer;
begin
  for I:= Hrany[V] to Hrany[V+1]-1 do
    if Ocislovani[Sousedi[I]] = 0 then
      Projdi(Sousedi[I]);

  Inc(Posledni);
  Ocislovani[Vrchol]:= Posledni;
end;

begin
  ...
  for I:= 1 to N do
  Ocislovani[I]:= 0;

  Posledni:= 0;
  for I:= 1 to N do
  if Ocislovani[I] = 0 then Projdi(I);
  ...
end.
```

Hranová a vrcholová souvislost

Nyní se podíváme na trochu komplikovanější formu souvislosti. Říkáme, že neorientovaný graf je *hranově 2-souvislý*, když platí, že:

- má 3 a více vrcholů,
- je souvislý,
- zůstane souvislý po odebrání libovolné hrany.

Hranu, jejíž odebrání by způsobilo zvýšení počtu komponent souvislosti grafu, nazýváme *most*.

Na hledání mostů nám poslouží opět upravené prohledávání do hloubky. Pokud by v grafu nebyly žádné zpětné hrany, byla by mostem každá hrana – rozdělila by graf na část obsahující kořen a podstrom „visící“ pod touto hranou. Aby nevznikly dvě komponenty souvislosti, musí mezi těmito částmi vést další hrana (a může to být jedině zpětná hrana).

Proto si pro každý vrchol spočítáme hladinu, ve které se nachází (kořen je na hladině 0, jeho synové na hladině 1, jejich synové 2, ...). Dále si pro každý vrchol v spočítáme, do jaké nejnižší hladiny vedou hrany z podstromu s kořenem v. To můžeme udělat přímo při procházení do hloubky, protože než se vrátíme z v, projdeme celý podstrom pod v. Pokud všechny zpětné hrany vedou do hladiny stejné nebo větší než té, na které je v, pak odebráním hrany vedoucí do v z jeho otce vzniknou dvě komponenty souvislosti, čili tato hrana je mostem. V opačném případě jsme nalezli kružnici, na níž tato hrana leží, takže to most být nemůže. Výjimku tvoří kořen, který žádného otce nemá a nemusíme se o něj starat.

Algoritmus je tedy pouhou modifikací procházení do hloubky a má i stejnou časovou a paměťovou složitost $O(N+M)$. Zde jsou důležité části programu:

```
var Hrany: array[1..MaxN + 1] of Integer;
  Sousedi: array[1..MaxM] of Integer;
  Hladina, Spojeno: array[1..MaxN] of Integer;
  DvaSouvisle: Boolean;
  I: Integer;

procedure Projdi(V, NovaHladina: Integer);
var I: Integer;
begin
  Hladina[V]:= NovaHladina;
```


Úloha 17-1-1 – Výdělek bratří Součků – program

```
var Ain,Kabel : string;
    Perioda   : longint;
    index     : longint;
    Shodne    : boolean;

function NSD(a,b:longint):longint;
begin
   while (a<>0) and (b<>0) do
      if (a>b) then
         a:=a mod b
      else
         b:=b mod a;
   NSD:=a+b;
end;

begin
   write('Ain:');readln(Ain);
   write('Kabel:');readln(Kabel);
   Perioda:=NSD(length(Ain),length(Kabel));
   Shodne:=true;

   for index:=1 to length(Ain) do             { generuje podřetězec délky NSD záznamy Aina a Kábela? }
      Shodne:=Shodne and (Ain[index]=Ain[((index-1) mod Perioda)+1]);
   for index:=1 to length(Kabel) do
      Shodne:=Shodne and (Kabel[index]=Kabel[((index-1) mod Perioda)+1]);
   for index:=1 to Perioda do                 { jsou generující podřetězce shodné? }
      Shodne:=Shodne and (Ain[index]=Kabel[index]);
   if Shodne then
      writeln('Záznamy jsou stejné.')
   else
      writeln('Záznamy jsou různé.');
end.
```

Úloha 17-1-2 – Bůhdhova odměna – program

```
#include <stdio.h>
#include <stdlib.h>

#define MAXN 100
#define MAXM (6*MAXN)

int N;                                  /* Počet vrcholů */
int first[MAXN];                        /* První hrana z vrcholu */
int dest[MAXM], next[MAXM];             /* Pro hranu: kam vede a další hrana */
int deg[MAXN];                          /* Stupně vrcholů */
int queue[MAXN];                        /* Fronta vrcholů nízkého stupně */
int color[MAXN];                        /* Přiřazené barvy */

void melolontha (void) {                /* Schroustá vstup */
  int i, j, M=1;
  scanf ("%d", &N);
  while (scanf ("%d%d", &i, &j) == 2) {
    /* Zařadíme novou hranu */
    deg[i]++, deg[j]++;
    dest[M]=j; next[M]=first[i]; first[i]=M;
    M++;
    dest[M]=i; next[M]=first[j]; first[j]=M;
    M++;
  }
}

void scan (void) {                      /* Projde graf podle stupňů */
  int r=0, w=0;                         /* Čtecí a zápisový index fronty */
  int i, j;
  for (i=0; i<N; i++)                   /* Na počátku nízké stupně */
    if (deg[i] < 6)
```

ší kružnici nijak přeskočit, protože jakmile právě přidávám do S její poslední vrchol, potom i a j se jednou trefí do jejich hran a pokud by její součástí měla být delší cesta než mnou nalezená $i \cdots j$, nebyl by výsledek nejmenší kružnice. Už zbývá jen projet všechna $i, j \in S$ a zjistit, zda nejsou $i \cdots k \cdots j$ nebo $i \cdots j - k$ kratší než původní $i \cdots j$ nebo $i \cdots k$ a případně je vylepšit. Časová složitost jednodušší verze je $O(N^3)$, paměťová složitost $O(N^2)$.

Další možnost řešení byla vybrat postupně každý vrchol jako start a spustit z něj Dijkstrův algoritmus (kterým najdu nejkratší cesty do všech ostatních vrcholů) a potom pro každou hranu nevedoucí z/do startu prozkoumat nejkratší cesty vedoucí z jejich konců na start. Pokud ani jedna z těchto cest není podmnožinou té druhé (na to se stačí podívat na jejich první hrany, jestli některá není ta prozkoumávaná), určitě tvoří buď kružnici nebo pseudokružnici (s „ocáskem“). Tak jako tak si z těchto všech mohu zapamatovat nejkratší (pseudo)kružnici, protože platí, že stejně musím (pro nějaký jiný start) objevit i kružnici bez toho ocásku, a ta bude určitě kratší. Časová složitost je zde (podle implementace Dijkstrova algoritmu) $O(N^3)$ při nejjednodušší implementaci, $O(MN \log N)$ při použití haldy a $O(MN + N^2 \log N)$ při použití Fibonnacciho haldy. Rozdíl těchto časových složitostí není sice velký (ty jsou v nejhorším vždy $O(N^3)$ až $O(N^3 \log N)$), ale některé z těchto implementací jsou mnohem lepší pro řídké grafy. Paměťová složitost je $O(N^2)$.

Tomáš Gavenčiak

17-1-5 Jazykozpytcův poklad

Obě zadané úlohy patřily spíše k těm snazším a řešitelé, kteří úlohy odeslali, byli převážně dvou druhů. První, malá skupina těch, kteří si nejspíše pořádně nepřečetli zadání, řešila povětšinou něco úplně jiného. Skupina druhá, naštěstí podstatně větší, která se prokousala povídáním o jazycích a správně pochopila definici konečného automatu a gramatiky, po přečtení zadání bez problémů obě úlohy vyřešila.

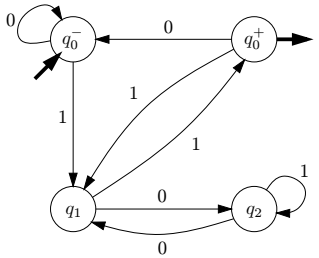
Jak tedy na konstrukci konečného automatu, který rozpoznává binární čísla dělitelná třemi a nedělitelná dvěma? Použijeme velice jednoduchý trik. Budeme automatem postupně cifru po cifře načítat číslo a průběžně si budeme pamatovat nikoli jeho hodnotu, nýbrž pouze zbytek po dělení šesti. Zjevně nám potom budou vyhovovat pouze čísla tvaru $6k + 3$ pro nějaké k, ostatní jsou buďto dělitelná dvojkou nebo nedělitelná trojkou. Když máme načtené číslo s aktuálním zbytkem z (tedy tvaru $6k + z$), po načtení 0 dostaneme číslo $2(6k + z)$, po načtení 1 číslo $2(6k + z) + 1$. Zbývá si tedy pouze spočítat, jak se pro každé z a načtenou cifru zbytek změní.

Náš stroj tedy bude používat stavy $Q = \{q_0, \ldots, q_5\}$, kde stav q_i značí, že aktuální zbytek je i. Počáteční stav bude q_0 a jediný přijímací stav bude $F = \{q_3\}$. Přechodová funkce δ bude vypadat takto:

δ	0	1
q_0	q_0	q_1
q_1	q_2	q_3
q_2	q_4	q_5
q_3	q_0	q_1
q_4	q_2	q_3
q_5	q_4	q_5

Každý sám už si asi domyslí, že podobný postup by fungoval, pokud bychom měli zadanou libovolnou konečnou množinu M čísel, které musí, resp. nesmí dělit vstup. Stačí vzít tolik stavů, kolik je nejmenší společný násobek čísel z M, přepočítávat při načítání zbytek a vhodně zvolit přijímací stavy.

Nicméně náš konkrétní automat lze navrhnout ještě jednodušší. Dělitelnost dvojkou je totiž ekvivalentní tomu, že poslední načtená cifra je 0. Stačí si tedy průběžně počítat zbytek při dělení třemi a to, zda poslední načtená cifra byla 0 nebo 1, nás zajímá jen v případě, že bychom se pomocí ní dostali do stavu reprezentujícího zbytek nula. To lze vyřešit tak, že tento stav rozštěpíme na dva, q_0^- a q_0^+, podle toho, zda jsme zbytku nula dosáhli načtením 0 či 1. Přijímací stav pak bude pouze q_0^+.



Když už máme hotový automat (Q, A, δ, q_0, F), je, jak si také většina řešitelů správně všimla, velmi jednoduché podle něj zkonstruovat ekvivalentní gramatiku (V_N, V_T, S, P). Pro stavy automatu $Q = \{q_0, \ldots, q_k\}$, zavedeme do gramatiky neterminální symboly $V_N = \{Q_0, \ldots, Q_k\}$, terminální symboly $V_T = A$ budou písmena abecedy, počáteční symbol $S = Q_0$ bude neterminál odpovídající počátečnímu stavu automatu.

Pro každý stav q_i a každé písmeno p se podíváme, jaký nový stav $q_j = \delta(q_i, p)$ vrací přechodová funkce. Do množiny přepisovacích pravidel P potom dáme příslušné pravidlo $Q_i \to pQ_j$. Pokud je navíc stav q_j přijímací, přidáme ještě pravidlo $Q_i \to p$. Každá expanze gramatiky zjevně následuje výpočet automatu, a tudíž generuje stejný jazyk.

Konkrétně v našem případě dostaneme gramatiku tvaru $(V_N, \{0, 1\}, Q_0^-, P)$ s neterminály $V_N = \{Q_0^-, Q_0^+, Q_1, Q_2\}$ a pravidly P:

$$\begin{aligned} Q_0^- &\to 0Q_0^- \mid 1Q_1 \\ Q_0^+ &\to 0Q_0^- \mid 1Q_1 \\ Q_1 &\to 0Q_2 \mid 1Q_0^+ \mid 1 \\ Q_2 &\to 0Q_1 \mid 1Q_2 \end{aligned}$$

První dva řádky jsou víceméně stejné, lze je tedy dokonce nahradit jediným řádkem $Q_0 \to 0Q_0 \mid 1Q_1$, příslušně modifikovat pravidlo pro Q_1 a ušetřit tak jeden neterminální symbol.

V zadání jsme tvrdili, že regulárním jazykům odpovídají právě gramatiky s pravidly tvaru $N \to uM$ a $N \to u$, kde N, M jsou neterminály a u je terminál. První část tohoto tvrzení, konstrukce ekvivalentní gramatiky ze zadaného automatu, jsme právě ukázali. Zbývá popsat konstrukci ekvivalentního automatu ze zadané gramatiky výše uvedeného tvaru. Ta se provede opačným postupem než při převodu automat → gramatika a detaily si jistě rozmyslí každý sám.

Tomáš Valla


```
  Spojeno[V]:= Hladina[V];

  for I:= Hrany[V] to Hrany[V + 1] do
    if Hladina[Sousedi[I]] = -1 then
    begin
      Projdi(Sousedi[I], NovaHladina + 1);
      if Spojeno[Sousedi[I]] < Spojeno[V] then
        Spojeno[V]:= Spojeno[Sousedi[I]];
      if Spojeno[Sousedi[I]] > Hladina[V] then
        DvaSouvisle:= False;
    end else
    if Hladina[Sousedi[I]] < Spojeno[V] then
      Spojeno[V]:= Hladina[Sousedi[I]];
end;

begin
  ...
  for I:= 1 to N do
    Hladina[I]:= -1;

  DvaSouvisle:= True;
  Projdi(1, 0);
  ...
end.
```

Další formou souvislosti je *vrcholová souvislost*. Graf je *vrcholově 2-souvislý*, právě když:

- má 3 a více vrcholů,
- je souvislý,
- zůstane souvislý po odebrání libovolného vrcholu.

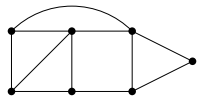
Artikulace je takový vrchol, který když odebereme, zvýší se nám počet komponent souvislosti.

Algoritmus pro zjištění vrcholové 2-souvislosti grafu je velmi podobný algoritmu na zjišťování hranové 2-souvislosti. Jen si musíme uvědomit, že odebíráme celý vrchol. Ze stromu procházení do hloubky může odebráním vrcholu vzniknout až několik podstromů, které všechny musí být spojeny zpětnou hranou s hlavním stromem. Proto musí zpětné hrany z podstromu určeného vrcholem v vést až *nad* vrchol v. Speciálně pro kořen nám vychází, že může mít pouze jednoho syna, jinak bychom ho mohli odebrat a vytvořit tak dvě nebo více komponent souvislosti. Algoritmus se od hledání hranové 2-souvislosti liší jedinou změnou ostré nerovnosti na neostrou, sami zkuste najít, které nerovnosti.

O rovinných grafech

Rovinný graf je graf, který můžeme nakreslit do roviny tak, že vrcholům přiřadíme vhodné body a hrany nakreslíme jako křivky spojující příslušné body, a to tak, že se žádné dvě křivky neprotínají mimo své krajní body. Ne každý graf takto nakreslit můžeme – sami si rozmyslete, že například graf K_5, což je 5 vrcholů spojených každý s každým, žádné rovinné nakreslení nemá. Na druhou stranu například každý strom určitě rovinný je.

Vezměme si tedy nějaký graf a jeho rovinné nakreslení, například tento:



Hrany nakreslení dělí rovinu na několik oblastí, těm budeme říkat *stěny*. Náš graf má 6 stěn: jednu čtvercovou, čtyři „trojúhelníkové“ (tedy ohraničené třemi hranami, byť to nejsou vždy úsečky) a jednu 6-úhelníkovou (to je celý zbytek roviny okolo grafu, tzv. *vnější stěna*). Například libovolné rovinné nakreslení stromu by mělo pouze jednu stěnu, a to tu vnější. Všimněte si, že pokud v grafu nejsou mosty ani akrtikulace, je každá stěna ohraničena nějakou kružnicí. [Pozor, to, jak vypadají stěny, závisí na konkrétním nakreslení do roviny!]

O rovinných grafech platí několik důležitých vět, které se často hodí při vytváření grafových algoritmů:

Věta: *(o počtu hran stromu)* Pro každý strom platí, že $e = v - 1$, kde v je počet vrcholů a e počet hran.

Důkaz: Indukcí podle počtu vrcholů. Pro strom s jedním vrcholem formulka určitě platí. Strom s $v > 1$ vrcholy má jistě list, tak jej odtrhneme [poněkud vandalské, nicméně účinné], čímž získáme strom s menším počtem vrcholů, pro který podle indukčního předpokladu formulka platí, a opětovným přidáním listu platit nepřestane, protože k oběma stranám přičteme jedničku.

Věta: *(Eulerova formule)* Pro každý souvislý graf nakreslený do roviny platí, že $v + f = e + 2$, kde v je počet vrcholů, e počet hran a f počet stěn.

Důkaz: Opět indukcí, tentokráte podle počtu hran. Každý souvislý graf má alespoň $v - 1$ hran a pokud jich má právě tolik, je to strom. (Kdyby ne, stačí se podívat na kostru grafu, což musí být strom a ty, jak už víme, mají právě tolik hran a náš graf měl hran více.) Jenže každé rovinné nakreslení stromu má právě jednu stěnu, takže Eulerova formule platí.

Pokud máme nakreslení grafu, který je souvislý a není to strom, znamená to, že obsahuje alespoň jednu kružnici. A každá hrana na kružnici jistě odděluje nějaké dvě stěny. Zvolme si tedy nějakou takovou hranu h a z grafu ji odeberme. Tím získáme graf s menším počtem hran (opět nakreslený do roviny), použijeme indukční předpoklad, Eulerova formule pro něj tedy již platí, a vrátíme hranu zpět. Levá strana rovnosti se tím zvětší o 1 (přidali jsme stěnu), pravá také (přidali jsme hranu), tedy rovnost stále platí.

Věta: *(o hustotě rovinných grafů)* O každém rovinném grafu platí, že $e \leq 3v - 6$.

Důkaz: Zvolme si libovolné nakreslení grafu do roviny. Nejprve předpokládejme, že je to triangulace, čili že každá stěna je trojúhelník. V takovém grafu patří každá hrana k právě dvěma trojúhelníkovým stěnám, takže $e = f \cdot 3/2$, čili $f = e \cdot 2/3$. Dosazením do Eulerovy formule získáme $v + (2/3)e = e + 2$, tedy $e = 3v - 6$.

Není-li náš graf triangulace, může to mít několik důvodů. Buďto není souvislý (pak ale stačí větu dokázat pro jednotlivé komponenty a nerovnosti sečíst), nebo je moc malý (má nejvýše dva vrcholy, proto to musí být jedna samotná hrana a pro tu naše věta určitě platí) a nebo obsahuje nějakou stěnu ohraničenou více než třemi hranami. Dovnitř takové stěny ovšem můžeme dokreslit další hrany a tím ji rozdělit na trojúhelníčky. Tím tedy dokážeme graf doplnit hranami na triangulaci, pro tu, jak už víme, platí dokonce rovnost, a když přidané hrany opět odebereme, snížíme pouze počet hran a uděláme tak z rovnosti nerovnost.

Věta: *(o vrcholu nízkého stupně)* V každém rovinném grafu existuje vrchol stupně maximálně 5. (Stupeň vrcholu je počet hran, které s vrcholem sousedí.)

Důkaz: Sporem. Kdyby všechny vrcholy měly stupeň alespoň 6, byl by součet stupňů alespoň $6v$. Jenže součet stupňů je přesně dvojnásobek počtu hran (každá hrana má dva konce), takže $e \geq 3v$, což je spor s předchozí větou.

Poznámky na okraj:

- K čemu je to všechno dobré, zjistíte třeba v řešení úlohy 17-1-2.
- Kdybychom definici rovinného nakreslení změnili a dovolili hrany kreslit pouze jako úsečky místo libovolných křivek, překvapivě se nic nezmění: každý rovinný graf má rovinné nakreslení, v němž jsou všechny hrany úsečky. Ale není to zrovna jednoduché dokázat.
- Stejně jako do roviny bychom mohli grafy kreslit třeba na povrch koule. Tím se také nic nezmění, zkuste sami vymyslet, jak z rovinného nakreslení udělat „kulové“ a naopak. Ale třeba anuloid (povrch pneumatiky) se už chová jinak, například zmíněný nerovinný graf K_5 se na anuloid dá nakreslit bez křížení hran.
- Rovinné grafy, jejichž všechny vrcholy mají stupeň právě 5, opravdu existují, je to například graf odpovídající pravidelnému dvacetistěnu [má 12 vrcholů stupně 5 a 20 trojúhelníkových stěn]. V jistém smyslu je tedy naše poslední věta nejlepší možná.
- Více informací o teorii (nejen rovinných) grafů najdete například v knížce pánů Matouška a Nešetřila Kapitoly z diskrétní matematiky.

Dnešní menu Vám servírovali
Martin Mareš & Petr Škoda

Vzorová řešení první série sedmnáctého ročníku KSP

17-1-1 Výdělek bratří Součků

Řešení této úlohy by se dala rozdělit do tří skupin. Lineární, kvadratické (vůči délce vstupu) a nefunkční. Kromě toho několik lidí předpokládalo, že řetězce mohou být vůči sobě posunuté. Nechápu proč. Bratři byli vyslání na koncert společně a začali zapisovat oba hned na začátku.

No a jak mělo řešení vypadat? Nejprve je třeba si uvědomit, že pokud mají být řetězce záznamem toho samého, pak musí obsahovat stejný podřetězec, jehož opakováním vytvoříme Ainův i Kábelův záznam. No a jak dlouhý tento podřetězec může být? Jelikož jeho několikanásobným zopakováním musíme dostat jak Ainův, tak Kábelův záznam, musí jeho délka být největší společný dělitel délek záznamů Aina a Kábela, resp. NSD musí být jeho celočíselným násobkem.

Pokud bude tedy Ainův i Kábelův záznam složen z opakujícího se podřetězce délky NSD a tyto podřetězce budou u Aina i Kábela stejné, víme, že zápisy budou stejné. A pokud podmínka splněná nebude, zaznamenali oba něco jiného.

Algoritmus řešící úlohu je jenom přímočarou implementací popsaného postupu, jeho časová i paměťová složitost je lineární vůči délce zápisů obou bratří.

Toto mělo být řešení této úlohy, ale při pročítání řešení účastníků jsem nalezl jedno výrazně elegantnější:

```
var Ain,Kabel:string;
begin
    write('Ain:');readln(Ain);
    write('Kabel:');readln(Kabel);
    if (Ain+Kabel = Kabel+Ain) then
      writeln('Záznamy jsou stejné.')
   else
      writeln('Záznamy jsou ruzné.');
end.
```

A proč toto funguje? BÚNO (bez újmy na obecnosti) předpokládejme, že Kábelův záznam (označme jej K a jeho délku L_K) je delší než Ainův (A, L_A). Záznamy budeme indexovat od nuly. Zřejmě ono porovnání vypadá takto:

$$\begin{array}{llll} A[0] & A[1] \ldots A[L_A-1] & K[0] \ldots K[L_K-L_A-1] & K[L_K-L_A] \ldots K[L_K-1] = \\ = K[0] & K[1] \ldots K[L_A-1] & K[L_A] \ldots K[L_K-1] & A[0] \quad \ldots A[L_A-1] \end{array}$$

Z porovnání těchto řádků dostaneme jednoduše:

$$K[i] = K[i \bmod L_A] \quad (1)$$

$$A[i] = K[i] \quad (2)$$

$$A[i] = K[L_K - L_A + i] \quad (3)$$

Zkombinováním (1) a (3):

$$A[i] = K[(L_K + i) \bmod L_A]$$

A po uvažování (2):

$$A[i] = A[(i + (L_K \bmod L_A)) \bmod L_A]$$

Iterováním této rovnosti pro libovolné celé c:

$$A[i] = A[(i + (c * (L_K \bmod L_A))) \bmod L_A]$$

Z toho je již vidět (po chvilce zamyšlení), že:

$$A[i] = A[i \bmod \mathrm{NSD}(L_A, L_K)] = K[i]$$

a tedy rovnost je splněna, právě když jsou záznamy stejné.

Pavel Čížek

17-1-2 Bůhdhova odměna

V této úloze nám nejde o nic jiného než o tak řečené *barvení grafu,* čili o přiřazení nějakých čísel (barev) $1, \ldots, k$ vrcholům grafu tak, aby žádné dva vrcholy spojené hranou nedostaly stejnou barvu. Zjistit, jestli je graf nějakými k barvami obarvitelný, je obecně velice těžký problém (pro obecné grafy a $k > 2$ ho nikdo neumí vyřešit v polynomiálním čase; případy $k = 1$ a $k = 2$ si rozmyslete sami, ty jsou pro změnu triviální). Ale my o našem grafu naštěstí víme, že je rovinný (viz povídání v tomto vydání naši kuchařky), což situaci značně mění.

Každý rovinný graf lze obarvit čtyřmi barvami (to ale vůbec není triviální dokázat, matematikové se s tím trápili více než 100 let a nejkratší známý důkaz má přes sto stránek a rozebírá 633 různých případů). My si dokážeme, že vždy stačí 6 barev a že Bůhdha má tedy vždy šanci svou odměnu rozdělit:

Věta: *(o šesti barvách)* Každý rovinný graf je možno obarvit šesti barvami.

Důkaz: Indukcí podle počtu vrcholů. Pokud má graf nejvýše 6 vrcholů, obarvit ho dozajista můžeme. Pokud je graf větší, věta dokázaná v kuchařce nám říká, že v něm vždy existuje nějaký vrchol v stupně maximálně 5. Když takový vrchol odstraníme, dostaneme menší graf a ten je již podle indukčního předpokladu obarvitelný. Následně do obarveného grafu vrchol v vrátíme, a jelikož má nejvýše 5 sousedů, vždy je pro v alespoň jedna barva volná.

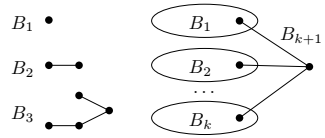
Již tento důkaz nám dává jednoduchý algoritmus pro Bůhdhův problém, ale ještě si musíme rozmyslet, jak neztrácet příliš mnoho času hledáním vrcholů nízkého stupně. Předem si spočítáme stupně všech vrcholů a do fronty uložíme ty vrcholy, které měly už na počátku stupeň ≤ 5. Poté postupně čteme vrcholy z fronty, snižujeme stupně jejich sousedům a jakmile některému sousedovi klesne stupeň pod 6, přidáme jej na konec fronty. Tím jsme vlastně sestrojili takové uspořádání vrcholů, že z každého vrcholu vede „doprava“ nejvýše 5 hran. Stačí tedy frontu projít pozpátku a postupně přidělovat volné barvy.

Tento algoritmus má lineární časovou i prostorovou složitost $O(N)$ (všimněte si, že jelikož je graf rovinný, má $O(N)$ hran). V programu stojí za zmínku snad jediné způsob reprezentace grafu: hrany máme uloženy v poli (každou dvakrát – v obou směrech), každý vrchol si pamatuje číslo první hrany z něj vycházející a každá hrana číslo následující hrany vycházející z téhož vrcholu. Při odtrhávání vrcholů hrany neodstraňujeme, pouze snižujeme stupně.

[P.S.: Po chvíli přemýšlení můžete náš důkaz upravit tak, aby ukazoval, že stačí 5 barev. Bůhdha se ale spokojí s šesti, takže mu nebudeme komplikovat život.]

Na závěr ještě ukážeme několik variant hladového barvícího algoritmu, který se často objevoval ve vašich řešeních a bohužel pro rovinné grafy nemůže fungovat.

Hladové barvení funguje takto: probíráme vrcholy jeden po druhém a každému přidělíme nejnižší barvu, která není použitá již obarvenými sousedy tohoto vrcholu. Zkusme si rozmyslet, jak obarví následující grafy:



Zde B_1 je graf z jednoho vrcholu a B_{k+1} zkonstruujeme tak, že vezmeme $B_1, \ldots, B_k$ a přidáme nový koncový vrchol v_k spojený s koncovými vrcholy všech B_i hranou. Vrcholy nového grafu uspořádáme tak, že nejdříve půjdou vrcholy grafů B_1, pak $B_2, \ldots, B_k$ a na konec umístíme nově vytvořený vrchol.

Všimneme si, že zadáme-li hladovému barvícímu algoritmu graf B_k, spotřebuje k barev a vrchol v_k obarví barvou k. I tentokrát nám k důkazu pomůže indukce: B_1 obarvíme jednou barvou, B_2 dvěma; pokud spustíme algoritmus na graf B_k, nejdříve obarví B_1 až B_{k-1}, a jelikož jsou v tom správném pořadí a nevedou mezi nimi žádné hrany, dopadne to stejně, jako bychom barvili každý zvlášť, čili podle indukčního předpokladu budou jejich koncové vrcholy obarveny barvami 1 až $k-1$, proto na zbývající vrchol v_k zbude barva k.

Jenže B_k je určitě rovinný graf, takže se dá obarvit šesti barvami (on je to dokonce strom, a tak stačí barvy dvě). Proto na něm pro $k > 6$ nemůže hladový barvící algoritmus fungovat.

Hladové barvení se ještě můžeme pokusit zachránit tím, že si zvolíme nějaké šikovné pořadí vrcholů (konec konců i náš správný barvící algoritmus je vlastně toho druhu). Ale jen tak ledajaké pořadí neposlouží:

- Pořadí podle rostoucích stupňů: v našem grafu pouze zparalelizuje barvení podgrafů $B_1, \ldots, B_k$ – jenže mezi nimi nevedou žádné hrany, takže výsledek musí vyjít stejný.
- Podle klesajících stupňů: stačí k vrcholům přivěsit spoustu listů (vrcholů stupně 1) a tím algoritmus donutit k takovému pořadí, jaké chceme (možná bude potřebovat ještě jednu barvu na listy, ale tím hůř pro něj).
- Podle prohledávání do šířky: přidáme každému B_i ještě počáteční vrchol w_i a při vytváření B_{k+1} připojíme nový počáteční vrchol w_{k+1} k počátečním vrcholům $w_1, \ldots, w_k$ cestami délek zvolených tak, aby koncové vrcholy všech B_i byly ve stejné vzdálenosti od w_{k+1}. Tehdy bude nový koncový vrchol v_{k+1} obarven až po všech ostatních vrcholech, čehož jsme přesně potřebovali dosáhnout. Graf přitom zůstane rovinný.
- Podle prohledávání do hloubky: tentokrát budeme přidávat hrany z v_i do w_{i+1} a zařadíme je tak, aby je prohledávací algoritmus objevil vždy před hranou do v_{k+1}. Takové cesty donutí prohledávání zpracovat nejdřív B_1 až B_k a teprve pak barvit vrchol v_{k+1}. Přitom vrchol w_i má blokovánu pouze barvu $i-1$, takže ho určitě obarvíme jedničkou. Protipříklad opět zachráněn, graf opět zůstane rovinný. (Jediný problém je s hranou v_1w_2, na tu musíme přidat ještě jeden vrchol, jinak bude w_2 obarven barvou 2, což nechceme.)

Martin Mareš

17-1-3 Chmatákův lup

Při řešení tohoto příkladu se mnoho z Vás inspirovalo příkladem v kuchařce. Avšak asi jenom polovina řešení byla správně. Hlavním problémem nefunkčních řešení bylo zejména nesprávné ošetření vícenásobného započtení jednoho předmětu.

Klíčové pozorování, které vede k pěknému řešení: Máme-li maximální dosažitelné ceny lupu pro všechny celočíselné kapacity batohu pomocí prvních k předmětů, snadno spočteme maximální ceny pro prvních $k+1$ předmětů. A to tak, že zkusíme přidat $k+1$-ní předmět do batohu s k předměty a kapacitou nižší o hmotnost tohoto předmětu. Předmět nepřidáme, pokud cena takového lupu není vyšší od ceny lupu z prvních k předmětů v batohu se stejnou kapacitou. První předmět je možné vložit do batohu s kapacitou rovnou alespoň hmotnosti předmětu a cena lupu je maximální možná. Pro další předměty to plyne z indukce.

Budeme postupně zaplňovat tabulku podle popisu v pozorování. Z této tabulky snadno zrekonstruujeme vložené předměty. Stačí si uvědomit, že předmět i byl použit, pokud vylepšil cenu lupu v batohu se stejnou nosností bez tohoto předmětu.

Časová i paměťová složitost je $O(P \cdot N)$.

Miroslav „miEro“ Rudišín

17-1-4 Paloučkova Výhra

V této úloze jde o nalezení nejkratší kružnice v ohodnoceném grafu. Celkem často se vyskytovalo jedno popsaných řešení, ale našla se i originální řešení v $O(N^4)$, $O(N^5)$.

Jednodušší řešení je dynamické, upravím vlastně Floyd-Warshalla tak, že postupně rozšiřuji množinu S už prozkoumaných vrcholů. Na začátku do ní vložím libovolné 2 vrcholy a do matice vzdáleností D si uložím délky hran nebo příp. nekonečna (jako v kuchařce). Potom vždy vezmu vrchol k, který ještě není v S, a pro všechny jeho sousedy i,j, kteří už jsou v S (jsou-li takoví), prozkoumám cestu $i \cdots j$. Pokud ji totiž už znám, určitě nevede přes k a $k - i \cdots j - k$ je tedy kružnice. Ze všech takto postupně nacházených kružnic si vyberu nejmenší a tu si zapíšu (pamatuji si vždy jen tu nejlepší) – to udělám tak, že si (stejně jako v kuchařce) pamatuji město s největším číslem na každé cestě $E[i,j]$, updatuji ho při změně cesty, vypisuji rekurzí. Nemůžu nejmen-