Korespondenční Seminář z Programování

33. ročník **KSP** **Listopad 2020**

Milí řešitelé, řešitelky a řešitelčata!

Dostává se k vám podzimní vydání KSP-H aneb druhé série hlavní kategorie 33. ročníku KSP.

Podobně jako v první tak i v této sérii naleznete **4 normální úlohy**, **bonusovou úlohu** a pak také pokračování **seriálu o počítačové grafice**. Všechny díly seriálu můžete **odevzdávat v průběhu celého roku**, takže vůbec nevadí, že jste třeba první díl nestihli. Detaily naleznete u zadání seriálu. Připomínáme, že oproti loňskému ročníku se do výsledkové listiny započítávají **všechny úlohy** a body se již nepřepočítávají podle počtu vyřešených sérií.

Odměny & na Matfyz bez přijímaček

Za úspěšné řešení KSP můžete být **přijati na MFF UK bez přijímacích zkoušek**. Úspěšným řešitelem se stává ten, kdo získá za celý ročník (této kategorie) alespoň 50 % bodů. Za letošní rok půjde získat maximálně 300 bodů, takže hranice pro úspěšné řešitele je 150. Maturanti pozor, pokud chcete prominutí využít letos, musíte to stihnout do konce čtvrté série, pátá už bude moc pozdě. Také každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok, placku a možná i další překvapení.

Termín série: **13. prosince 2020 ve 32:00**

Odevzdávání: Přes web na adrese *https://ksp.mff.cuni.cz/submit/*

Značky úloh: Lehčí úloha (či její část) vhodná pro začátečníky

Praktická open-data úloha

Úloha, u které doporučujeme začíst se do kuchařky

Seriálová úloha

Odměna série: Každému, kdo vyřeší 4 úlohy alespoň na polovinu bodů, pošleme **sladkou odměnu**.

Druhá série třicátého třetího ročníku KSP

Letos nebudou mít úlohy žádný velký spojující příběh, na který jste mohli být zvyklí z minulých let. Namísto toho budou mít jednotlivé úlohy své vlastní malé příběhy. Snad se vám budou líbit :)

33-2-1 Ostrovní království — 9 bodů

Za devatero horami, sedmero moři a třemi informatiky na odvážné výpravě se rozkládá malé ostrovní království. I když je malé co do rozlohy, tak rozhodně není malé co do počtu ostrovů. Ostrovů je dokonce tolik, že v královském archivu ani pořádně neví, kolik jich je. Krále by nyní zajímalo, z kolika obydlených ostrovů (s alespoň jedním městem) se jeho říše skládá. Ale počet měst ani cest nikdo nezaznamenával.

Jedinou záchranou by mohlo být královské ministerstvo cest, které si drží podrobný přehled o všech cestách, o které se musí starat. Na každém ostrově se nachází několik měst (vždy minimálně jedno, menší ostrovy bez města nás teď nezajímají), každé z měst má své unikátní sekvenční číslo. Města na jednom ostrově jsou spojena cestami, které jsou také sekvenčně očíslované (číslujeme od nuly dál, žádná cesta ani žádné město ještě v historii království nezanikly, takže v číslování nejsou žádné „díry“).

Vždy platí, že ze všech měst na jednom ostrově se dá po cestách dostat do všech dalších měst na tom stejném ostrově, ale na jiný ostrov je potřeba vždy dojet lodí (zkrátka mezi ostrovy žádná cesta nevede, v království nevěří v existenci mostů). Navíc na žádném ostrově cesty netvoří cyklus (ministerstvo cest nebuduje zbytečné cesty a na existenci kruhových objezdů v království nevěří).

Na vás dopadl úkol spočítat počet ostrovů. Jediné dva typy dotazů, které můžete pokládat archivářům ministerstva cest, jsou:

- „Mezi jakými městy vede cesta s číslem x?“ – dostanete jako odpověď dvě čísla měst spojených cestou x, případně informaci, že cesta x neexistuje.
- „Dá se z města x dostat po souši do města y?“ – dostanete odpověď `ANO` nebo `NE` (případně informaci, že některé z měst x nebo y neexistuje).

Vyhledávání v archivu ministerstva cest je ale náročné.[1] Proto byste měli vymyslet postup, jak získat počet obydlených ostrovů na co nejméně dotazů do archivu.

33-2-2 Hasičské stanice — 11 bodů

V právě postaveném městě inženýři zjistili, že zapomněli na jednu důležitou věc. Ve městě totiž nepostavili žádnou hasičskou stanici. Z vyhlášky EU však není dovoleno bydlet v budovách, které nejsou chráněné alespoň jednou hasičskou stanicí. Pomozte jim vymyslet, jak nejlépe stanice rozmístit.

Město si můžeme představit jako obdélník $N \times M$ budov. U každé budovy je dán maximální počet lidi, kteří zde mohou bydlet.

Každou budovu je možné přestavět na hasičskou stanici. Když se to stane, tak v dané budově již nebudou moct bydlet žádní lidé. Tato stanice pak bude chránit všechny budovy, které leží ve stejném řádku nebo sloupci (hasičská auta jsou moc veliká, a tedy nezvládnou zatáčet, aby dojela k budově za rohem).

[1] *Už jsme se zmiňovali, že mimo mostů a kruhových objezdů nevěří v tomto království ani na papír? Záznamy jsou vytesány na masivních mramorových deskách...*

Vaším úkolem je vymyslet algoritmus, který dostane zadanou mapu města a upraví ho přidáním hasičských stanic tak, aby byl počet lidí, kteří v něm mohou žít, co nejvyšší. Lidé mohou bydlet pouze v budovách chráněných alespoň jednou hasičskou stanicí a nemohou bydlet na hasičských stanicích.

33-2-3 Bludiště s turrety — 12 bodů

Tajný agent Hrochbond uvízl v komplikovaném bludišti postaveném doktorem Zlounem. Bludiště je tvořeno pravoúhlými políčky, na kterých je buď prázdné místo, zeď, nebo turret střílející rakety jedním ze čtyř směrů. Agentovi Hrochbondovi se povedlo získat mapu bludiště a dokonce i intervaly, ve kterých jednotlivé turrety střílejí, a potřeboval by najít nejrychlejší cestu k východu.

Každý turret střílí pouze jedním směrem a to v pravidelných intervalech. V čase nula všechny turrety vystřelí raketu (objeví na sousedním políčku turretu ve směru jeho střelby) a rakety postupně letí rychlostí jednoho políčka za sekundu v přímém směru, dokud nenarazí na pevnou překážku (zeď, turret nebo Hrochbonda). Turret s intervalem K pak vystřelí další raketu za K sekund (tedy turret s intervalem K vystřelí raketu v časech 0, K, $2K$, ...).

Rakety jsou malé, takže pokud se potkají dvě rakety na stejném políčku, tak se vyhnou a obě letí dál v původním směru. Políčko turretu se počítá jako zeď jak pro Hrochbonda, tak pro rakety.

Hrochbond se zvládne přemístit na vedlejší políčko také za jednu sekundu, takže celý průchod bludištěm probíhá v jednosekundových kolech. Hrochbond se však nesmí vyskytnout na stejném políčku, na kterém se vyskytuje raketa (vybuchl by, k radosti doktora Zlouna), nebo na políčku, které raketa právě opustila (raketa má za sebou ohnivý plamen, který by z Hrochbonda udělal v mžiku well-done steak), a to i pokud raketa ihned po opuštění políčka narazí do zdi.

Hrochbond může vykonat každé kolo jednu z pěti následujících akcí:

- `L`: posunout se o políčko doleva
- `P`: posunout se o políčko doprava
- `N`: posunout se o políčko nahoru
- `D`: posunout se o políčko dolů
- `-`: zůstat stát na místě bez pohybu

Je možné vstoupit jen na volná políčka a na políčko cíle, zdmi a turrety procházet nelze.

Nalezněte pro zadanou mapu na vstupu nejrychlejší cestu (co do počtu kol) vedoucí ze startu do cíle bez toho, aniž by se Hrochbond během cesty dostal na políčko, na kterém se nachází raketa nebo se v minulém kole nacházela.

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

Formát vstupu: Na prvním řádku dostanete tři čísla – R a S udávající počet řádků a sloupců mapy (pozice $[0, 0]$ je vlevo nahoře) a číslo T udávající počet turretů. Bude následovat R řádků po S znacích popisující bludiště: `.` (tečka) značí volné políčko, `#` zeď, `T` turret, `S` startovní pozici Hrochbonda a `C` cílové políčko.

Na následujících T řádcích pak dostanete popis jednotlivých turretů. Každý řádek bude obsahovat pozici turretu (řádek a sloupec, indexujeme od nuly), natočení turretu, ve kterém střílí rakety (jedno z písmen `LPND`), a interval ve kterém rakety střílí (interval bude mezi čísly 1 až 6 včetně).

Formát výstupu: Na první řádek výstupu vypište délku vámi nalezené nejkratší cesty D a na druhý řádek pak bez mezer D znaků udávajících jednotlivé Hrochbondovy kroky. Slibujeme, že cesta ze startu do cíle vždy existuje.

Ukázkový vstup:

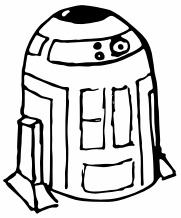
```
6 5 2
S....
#T...
....T
#.#..
.##..
.C..#
1 1 P 3
2 4 L 4
```

Ukázkový výstup:

```
12
--PPPDDDDDLL
```

33-2-4 Oprava satelitů — 13 bodů

Na vesmírné stanici se kvůli nárazu s asteroidem pomíchaly satelity a je potřeba je srovnat. V extrémních podmínkách, které panují na této planetě, však nemá žádný astronaut šanci přežít – proto je potřeba naprogramovat robota, který závadu opraví autonomně.



Satelity si lze představit jako orientovaný graf, kde z každého satelitu (vrcholu) vede jeden výstupní kabel (hrana). Na začátku robotovy práce tvoří satelity „řetěz“ se speciálními satelity Z na začátku a C na konci.

Robot začíná svojí práci na satelitu Z, může svoji práci ukončit kdekoliv a má k dispozici tyto čtyři typy příkazů:

- **Dopředu:** robot přejde přes výstupní kabel k dalšímu satelitu.
- **Položit X:** robot položí na aktuální satelit značku `X` (má k dispozici 52 značek značených písmeny `a-z` nebo `A-Z`); položení značky podruhé ji přesune na nové místo.
- **Teleportovat na X:** robot se teleportuje na značku `X`.
- **Nastavit na X:** robot nastaví (přepojí) výstupní kabel aktuálního satelitu na satelit s položenou značkou `X`. Tento příkaz nelze provést na satelitu C.

Aby začala komunikace se Zemí opět fungovat, tak je potřeba pro robota připravit sérii příkazů, které po K-ticích satelity otočí pozpátku (vyjma krajních satelitů Z a C, které zůstávají na místě). Například pro $K = 2$ se má provést následující:

$$Z \to 1 \to 2 \to 3 \to 4 \to C$$

$$\Downarrow$$

$$Z \to 2 \to 1 \to 4 \to 3 \to C$$

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

Formát vstupu: Na prvním řádku vstupu bude počet satelitů N (bez startovního a koncového) a velikost K-tic satelitů k otáčení. Slibujeme, že K dělí N (tj. nikde nezůstanou přebytečné satelity). V prvním testu je $K = 2$, ve všech ostatních je $K \geq 3$.

Formát výstupu: Posloupnost příkazů, která každou sousední K-tici satelitů prohodí. Příkazy mají tento tvar:

- `D`: dopředu
- `P <Z>`: položit značku `<Z>`
- `T <Z>`: teleportovat se na značku `<Z>`
- `N <Z>`: nastavit kabel na značku `<Z>`

Velikost programu pro robota může být nejvýše 50 MB (naše vzorová řešení dávají nejvýše malé jednotky MB).

Ukázkový vstup:

```
2 2
```

Znázornění:

```
Z -> 1 -> 2 -> C
---------------
Z -> 1 -> 2 -> C
A    B    C    D
---------------
Z -> 2 -> 1 -> C
```

Ukázkový výstup:

```
P A
D
P B
D
P C
D
P D
T C
N B
T B
N D
T A
N C
```

Body za poslední dva testy se odvíjí od toho, kolik různých značek robot na satelity položí – plné body lze získat pouze za řešení využívající nejvýše tři značky. Za použití více značek také nějaké body dostanete.

Na webu jsme připravili simulátor,[2] ve kterém si řešení můžete otestovat. Po zadání instrukcí simuluje robotův pohyb po satelitech a v případě chyb zobrazuje také dodatečné informace, které by se mohly hodit při řešení úlohy.

33-2-X1 Závody formulí — 10 bodů

Toto je bonusová úloha pro zkušenější řešitele, těžší než ostatní úlohy v této sérii. Nezískáte za ni klasické body, nýbrž dobrý pocit, že jste zdolali něco výjimečného. Kromě toho za správné řešení dostanete speciální odměnu a body se vám započítají do samostatných výsledků KSP-X.

Počítat se dá nejen klasickými algoritmy, ale třeba i pomocí logických (booleovských) formulí a obvodů. Pojďme se nad nimi zamyslet.

Formule

Nejprve definujeme *formuli.* To je výraz, ve kterém jsou logické *proměnné* (budeme značit řetězci písmen a číslic) pospojované *logickými spojkami.* Pro naše účely budou stačit spojky $\wedge$ (AND, konjunkce), $\vee$ (OR, disjunkce) a $\neg$ (NOT, negace). Formule může například vypadat takto:

$$(x \wedge \neg y) \vee (\neg x \wedge y).$$

Když chceme formuli „spustit“, dosadíme za proměnné logické hodnoty 0 (nepravda) nebo 1 (pravda). Výstupem formule pak je jediný bit, opět 0 nebo 1. Vyzkoušejte si, že naše ukázková formule dostane x a y a spočítá jejich XOR.

Nevýhodou formulí je, že „nemají proměnné“. Pokud chceme nějaký mezivýsledek použít víckrát na různých místech, nezbývá než zmáčknout pomyslné *copy & paste* a výpočet mezivýsledku zkopírovat. Pokud tento mezivýsledek závisí na jiném mezivýsledku a tak dále, může formule exponenciálně narůst. Proto budeme raději pracovat s obvody.

[2] `https://ksp.mff.cuni.cz/h/ulohy/33/satelity/`

Obvody

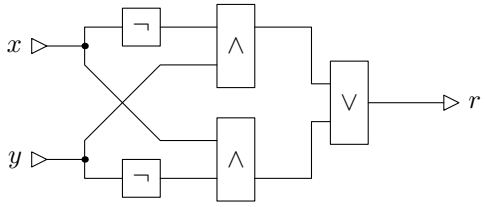
Obvod se skládá z následujících součástek:

- *vstupních portů* – každý z nich má nějaké jméno a přivádí do obvodu jeden bit zvenku (je to tedy ekvivalent proměnné ve formuli).
- *výstupních portů* – každý z nich předává jeden bit výsledku ven z obvodu. Nás budou zajímat jen obvody s jednobitovým výsledkem, takže budou mít jen jeden výstupní port.
- *hradel* – ta provádí logické operace. Budeme uvažovat hradla AND, OR a NOT. Každé hradlo má nějaké vstupy (AND a OR dva, NOT jeden) a jeden výstup.
- *konstant* – povolíme také „nula-vstupová hradla“, která mají jen výstup a vydávají na něm konstantní výsledek. Budeme jim říkat podle výsledků prostě 0 a 1.

Jednotlivé součástky jsou propojené „dráty“ podle následujících pravidel:

- Z každého vstupního portu a z každého výstupu hradla může vést libovolný počet drátů do výstupních portů a vstupů hradel.
- Do výstupního portu a na vstup hradla vede právě jeden drát.
- V drátech neexistuje cyklus (přesněji řečeno když součástky chápeme jako vrcholy a dráty jako orientované hrany, graf neobsahuje orientovaný cyklus).

Ukažme si příklad obvodu, který počítá XOR podle naší formule:



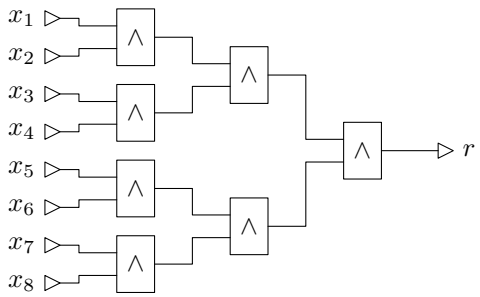
Výpočet obvodu probíhá v taktech. V každém z nich se *aktivují* ty součástky, které už mají k dispozici vstup z předchozích taktů, a vydají svůj výstup. V nultém taktu se tedy aktivují součástky, do kterých nevedou žádné dráty – to jsou vstupní porty a konstanty. Výpočet se zastaví, jakmile se aktivují všechny výstupní porty. Náš ukázkový obvod tedy počítá v pěti taktech: v nultém se aktivují porty x a y, v dalším negace, pak ANDy, pak OR a nakonec výstup.

Všimněte si, že počet taktů výpočtu nezávisí na hodnotách vstupů. Můžeme ho také spočítat jako délku nejdelší cesty mezi vstupními porty (případně konstantami) a výstupními porty. Proto se mu říká *hloubka obvodu.* Hloubku můžeme považovat za analogii časové složitosti. Podobně můžeme měřit prostor potřebný na výpočet *velikostí obvodu* vyjádřené počtem hradel.

Pokud byste zatoužili prozkoumat obvody detailněji, doporučujeme seriál 20. ročníku. Pro tuto úlohu by to ale nemělo být potřeba.

Stromečky

K jedné formuli obvykle existuje více obvodů, které se mohou lišit hloubkou. Třeba pro $x_1 \wedge x_2 \wedge \ldots \wedge x_8$ můžeme spočítat $x_1 \wedge x_2$, pak výsledek z ANDovat s x_3, pak výsledek s x_4 atd. Takový obvod bude mít hloubku 9. Nebo můžeme vyrobit „stromeček z ANDů“:



Ten spočítá totéž v hloubce 5. Obecně pro AND n proměnných dá první způsob hloubku $\Theta(n)$, zatímco druhý $\Theta(\log n)$.

Jak překládat formule

A teď konečně slíbená úloha. Budeme překládat formule na obvody. Dostaneme nějakou formuli délky n (délky budeme měřit řekněme počtem logických spojek) a budeme chtít vyrobit obvod, který počitá totéž s co nejmenší hloubkou.

Nějaký obvod určitě existuje. Stačí si nakreslit stromovou strukturu formule (v listech jsou proměnné, ve vnitřních vrcholech logické spojky. A to už vlastně je obvod, který počítá totéž: listy jsou vstupní porty (víc výskytů téže proměnné slepíme do jednoho portu), vnitřní vrcholy jsou hradla, z kořene vede drát do výstupního portu.

Tato konstrukce dává obvod hloubky $\Theta(n)$. Pro jeden konkrétní případ jsme ale viděli, že stačí $\Theta(\log n)$. Uměli byste obecný převod vylepšit, aby se k tomu přiblížil? Zajímavé je libovolné řešení s lepší než lineární hloubkou.

Poznámka: Také se můžete zamyslet nad tím, jak překládat naopak obvody na formule. Jak velká formule vyjde? Bodovat to nebudeme, ale i tak budeme rádi, když nám to napíšete.

33-2-S Šumy — 15 bodů

Toto je seriálová úloha, která navazuje na podobnou úlohu v minulé sérii. Pokud jste předchozí díl seriálu neřešili, pro pochopení tohoto dílu je dobré si jej nejméně přečíst. A pokud si chcete úlohy z minulého dílu také naprogramovat, stále za ně můžete získat polovinu bodů.

V tomto díle se podíváme na to, jak se dá z trošky náhody vytvořit něco zajímavého. Všechny přírodní struktury v sobě mají nějakou míru chaosu, ať už to je mech na skalách, oblaka na obloze, vzory ve dřevě nebo třeba povrch vašich dlaní. Právě náhodná podstata mnohých přírodních objektů dělá šumové funkce tak zajímavými a užitečnými. Nejprve si ale musíme nějakou tu náhodu pořídit.

Stejně jako v minulém díle budeme pracovat v Shadertoy,[3] což je webový editor shaderů. Jak jej ovládat (a jak psát shadery) se dočtete v prvním díle seriálu.

Opět odevzdávejte zdrojové kódy vašich shaderů zabalené do zipka a pojmenované nějak tak, aby bylo jasné, který shader řeší kterou úlohu. Vždy prosím odevzdávejte celý spustitelný kód shaderu, ne jenom jeho část.

Náhoda

Vyrábět náhodné hodnoty v počítačích je poměrně problém, zvlášť pokud chcete náhodu využít pro kryptografii. My si ale vystačíme s něčím, co náhodně jen vypadá.

V běžných programovacích jazycích máme k dispozici pseudonáhodné funkce, které nám při každém zavolání vrátí jinou, zdánlivě náhodnou hodnotu, nicméně uvnitř jsou tyto funkce deterministické. Před prvním použitím bývají inicializovány nějakou počáteční hodnotou, takzvaným *seedem*. Pro stejný seed funkce vrací stejnou posloupnost hodnot. Pokud chceme, aby se náhodný generátor choval při každém spuštění programu jinak, můžeme jako seed použít například systémový čas.

V shaderech žádný podobný nástroj není. Už jen zajistit, aby každé volání nějaké funkce vracelo pokaždé jinou hodnotu je problém, protože různé pixely se nesmí navzájem ovlivňovat (protože se počítají paralelně). A pokud by náhodná funkce vrátila pro každý pixel to samé, tak by moc užitečná nebyla.

Naše náhodná funkce tedy bude muset mít nějaké parametry, minimálně souřadnice aktuálního pixelu, ze kterých „náhodu“ spočítá.

Chceme tedy něco, co pro málo se lišící hodnoty (například souřadnice sousedních pixelů) vrátí velmi různé výsledky. Tuto vlastnost mají hešovací funkce a jako zdroj náhody můžeme použít některou z nich. Také si můžeme jednu velmi jednoduchou hešovací funkci vyrobit ze staré dobré funkce *sinus*. Spusťte si následující shader:

```
float func(float x)
{
    float y = sin(x);
    //y = y * 10.0;
    //y = fract(y);
    return y;
}

void mainImage(out vec4 fragColor,
               in vec2 fragCoord)
{
    vec2 uv = fragCoord/iResolution.xy;
    // Střed souřadnic chceme uprostřed obrazu
    uv = uv * 2.0 - 1.0;
    // Korekce pro správný poměr stran
    uv.x *= iResolution.x / iResolution.y;

    vec3 col = vec3(0.0);

    // Rozsah obrazu je asi -4..4 v obou osách
    // (u X ve skutečnosti trochu jiný, aby byl
    // zachován poměr stran)
    uv *= 4.0;

    // Osa X
    col += clamp(
            1.0 - abs(uv.y)*40.0, 0.0, 1.0
        ) * vec3(1.0, 0.5, 0.5);
    // Osa Y
    col += clamp(
            1.0 - abs(uv.x)*40.0, 0.0, 1.0
        ) * vec3(0.5, 1.0, 0.5);
    // f(x)
    col += clamp(
            1.0 - abs(uv.y - func(uv.x)) * 20.0,
            0.0, 1.0
        ) * vec3(0.5, 0.5, 1.0);

    // Vybarvíme pozadí současnou hodnotou
    // funkce
    col += clamp(func(uv.x), 0.0, 1.0) * 0.25;

    fragColor = vec4(col,1.0);
}
```

[3] `https://www.shadertoy.com/new`

Shader zobrazuje graf funkce `func`, která je ve své neupravené podobě jen *sinem*. Pozadí je vybarveno hodnotou funkce pro x-ovou souřadnici daného pixelu.

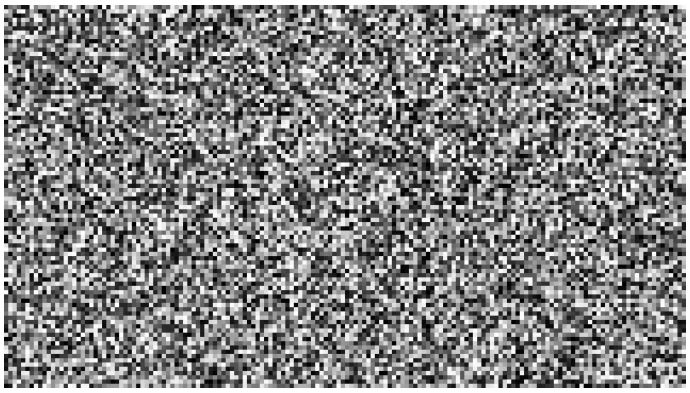
Zkuste odkomentovat řádek `y = fract(y);` ve `func` a podívat se, jak se graf funkce změní. Funkce `fract` vrací desetinnou část čísla, existují i funkce `floor` vracející dolní celou část a `ceil` vracející horní celou část.

Nyní odkomentujte řádek `y = y * 10.0;`. Výsledek zatím nevypadá příliš zajímavě. Teď zvětšete konstantu `10.0`, přidejte k ní pár nul. Výsledek pro třeba `10000.0` je poměrně chaotický, že? Právě jsme si vyrobili pseudonáhodnou funkci!

Poznámka: Toto demo je založené na podobné vizualizaci ve webové knize The Book of Shaders, kterou je inspirován i celý tento seriál. Neváhejte si ji přečíst, zvlášť 4. kapitolu, která je velmi zajímavá a relevantní k tomuto dílu seriálu.

Máme tedy něco, co nám z jednoho floatového parametru vyrobí „náhodu“. To ale nestačí. Vyrábíme dvojrozměrný obraz, a proto by se nám hodila funkce, která udělá náhodu z `vec2`. Tu vyrobíme následujícím trikem:

```
float random(vec2 st)
{
    float s = dot(st, vec2(12.3456, 34.1415));
    return fract(sin(s) * 45678.9);
}

void mainImage(out vec4 fragColor,
               in vec2 fragCoord)
{
    vec2 uv = fragCoord/iResolution.xy;
    // Střed souřadnic chceme uprostřed obrazu
    uv = uv * 2.0 - 1.0;
    // Korekce pro správný poměr stran
    uv.x *= iResolution.x / iResolution.y;

    vec3 col = vec3(random(uv));

    fragColor = vec4(col, 1.0);
}
```



Z vektoru vyrábíme skalár tím, že provedeme skalární součin s nějakým konstantním vektorem. V tomto případě je to ekvivalentní (z definice skalárního součinu) zápisu `float s = st.x * 12.3456 + st.y * 34.1415;`. Pokud vám není jasné, co to skalární součin je, odpověď najdete v našem krátkém úvodu do vektorů.[4]

Tedy změny jak v x-ové, tak v y-ové souřadnici vedou k jinému číslu dosazenému do *sinu*. Zároveň jsou obě čísla vynásobená různými a hlavně velkými konstantami, takže nevznikají čárovité artefakty (schválně zkuste místo toho použít `vec2(1.0, 1.0)`). Ve skutečnosti ty artefakty stále vznikají, jen jsou té správné velikosti a směru, že se v naší pravidelné mřížce pixelů neprojevují, a to nám stačí. Každý pixel padá do jiné části kopečků *sinu*. Kdybychom dostatečně zazoomovali, stále bychom artefakty viděli.

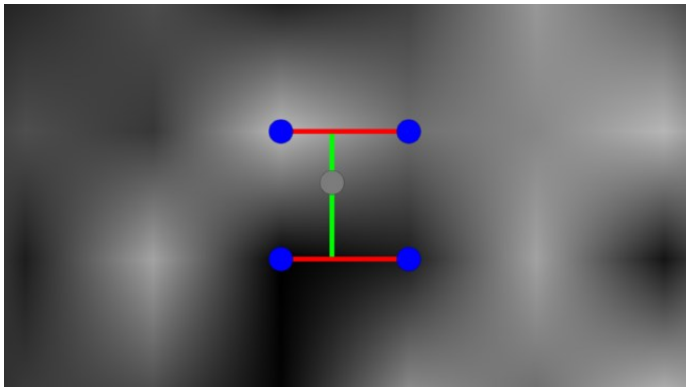
Neváhejte místo tohoto triku se *sinem* použít i jiné hešovací funkce. Spoustu jich naleznete například v tomto shaderu (přepněte se v editoru do záložky „Common“).

Perlin Noise

Náhoda je sice hezká, ale není až tak… hezká. Můžeme ji ale použít k vytvoření vizuálně zajímavější šumové funkce známé jako **Perlin Noise** či **Perlinův šum**.

Ideou Perlinova šumu je vytvořit pravidelnou mřížku bodů a spočítat náhodu jen pro tyto body. Pro místa mezi body interpolujeme (děláme plynulý přechod) hodnoty ze čtyř rohů aktuální „buňky“.

```
float random(vec2 st)
{
    float s = dot(st, vec2(12.3456, 34.1415));
    return fract(sin(s) * 45678.9);
}

float perlinNoise(vec2 st)
{
    vec2 cell = floor(st);
    vec2 cell2 = ceil(st);
    vec2 f = fract(st);

    float s00 = random(cell);
    float s01 = random(vec2(cell.x, cell2.y));
    float s10 = random(vec2(cell2.x, cell.y));
    float s11 = random(cell2);

    return mix(
        mix(s00, s10, f.x),
        mix(s01, s11, f.x),
        f.y
    );
}
```



Naše buňky mají velikost jedné jednotky. Pro získání souřadnic rohu buňky používáme již zmíněnou funkci `floor` (dolní celá část) a pro souřadnice uvnitř buňky funkci `fract` (desetinná část). Poté si spočítáme náhodu ve všech čtyřech rozích buňky a hodnotu lineárně interpolujeme pomocí funkcí `mix`, nejdříve dle souřadnice x, poté dle y.

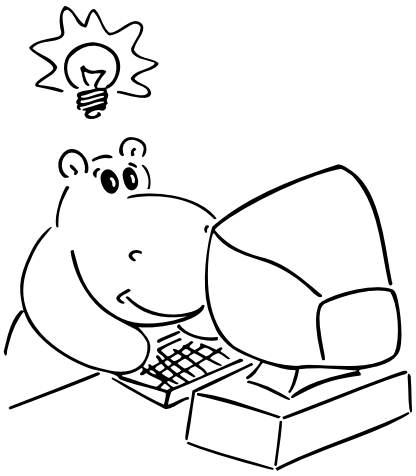
Obrázek vizualizuje funkčnost interpolace hodnot z rohů buňky. Modré puntíky značí rohy, ve kterých počítáme náhodu. Na červených horizontálních čarách interpolujeme

[4] `http://ksp.mff.cuni.cz/encyklopedie/vektory.html`

podle x-ové souřadnice uvnitř buňky, poté mezi dvěma výslednými hodnotami interpolujeme podle y-ové osy. Celé to odpovídá kódu: `mix(mix(s00, s10, f.x), mix(s01, s11, f.x), f.y)`. (Pamatujte, že `mix(x, y, a)` není nic jiného než `x * (1 - a) + y * a`.)

Zkuste tento šum spustit. Protože je velikost buňky jedna jednotka, je potřeba souřadnice pixelu něčím velkým vynásobit, aby bylo něco zajímavého vidět, třeba `perlinNoise(uv * 16.0)`.

Toto zatím nevypadá nic moc – všimněte si kosočtvercových artefaktů. Ty jsou způsobeny právě lineární interpolací (občas je lze vidět i ve hrách, pokud má nějaká textura malé rozlišení). To opravíme tím, že použijeme místo křivky lineární křivku kubickou (v GLSL zabudovaná jako funkce `smoothStep`). Dejte před „mixování“ tento řádek: `f = smoothstep(0.0, 1.0, f);`. Výsledek už vypadá o něco lépe.



Gradient Noise

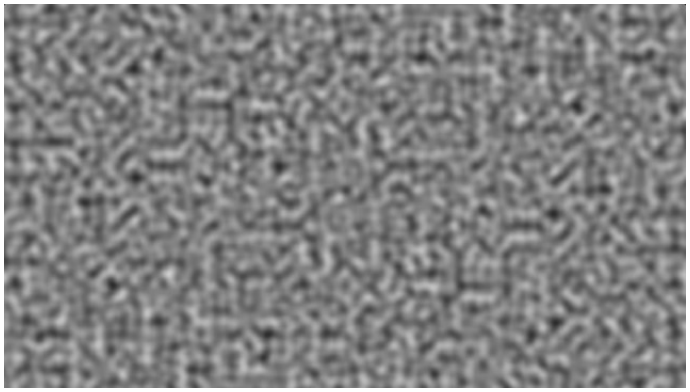
Existují různá vylepšení základního Perlinova šumu, například **Simplex Noise** nebo **Gradient Noise**. Simplex Noise používá místo čtvercové mřížky mřížku z rovnostranných trojúhelníků, což je pro vyšší dimenze výrazně rychlejší (počet bodů, co musíme spočítat, pak roste s dimenzemi lineárně, nikoliv exponenciálně).

Gradient Noise používá čtvercovou mřížku, ale neinterpoluje náhodné hodnoty jako takové, místo toho v každém vrcholu mřížky vytvoří nějaký přechod (gradient) a ten interpoluje. Více se o nich dočtete například v 11. kapitole The Book of Shaders.

Ve zbytku seriálu doporučuji používat místo základního Perlina právě jednu z těchto vylepšených variant. Níže je zdrojový kód pro Gradient Noise. Všimněte si, že náhodnou funkci, která místo skaláru vrací `vec2`, jsme vyrobili prostě tak, že náhodu počítáme dvakrát a vstupní `st` „dotujeme“ různými vektory, aby byly výsledky různé i pro stejné `st.x` a `st.y`. Také nyní v `random2` vracíme hodnoty v rozsahu -1 až 1, což se nám pro Gradient Noise hodí.

```
vec2 random2(vec2 st)
{
    vec2 s = vec2(
        dot(st, vec2(12.3456, 34.1415)),
        dot(st, vec2(42.2154, 15.2854))
    );
    return fract(sin(s) * 45678.9) * 2.0 - 1.0;
}

float gradientNoise(vec2 st)
{
    vec2 cell = floor(st);
    vec2 f = fract(st);

    vec2 s00 = random2(cell);
    vec2 s01 = random2(cell + vec2(0, 1));
    vec2 s10 = random2(cell + vec2(1, 0));
    vec2 s11 = random2(cell + vec2(1, 1));

    float d00 = dot(s00, f);
    float d01 = dot(s01, f - vec2(0, 1));
    float d10 = dot(s10, f - vec2(1, 0));
    float d11 = dot(s11, f - vec2(1, 1));

    vec2 u = smoothstep(0.0, 1.0, f);

    float noise = mix(
        mix(d00, d10, u.x),
        mix(d01, d11, u.x),
        u.y
    );
    return noise * 0.5 + 0.5;
}
```

Také si všimněte, že ve své základní variantě je Gradient noise docela šedivý a chybí mu kontrast. To lze opravit tím, že poslední řádek funkce `gradientNoise` nahradíte tímto: `return clamp(noise * 0.8 + 0.5, 0.0, 1.0);`. Hlavní rozdíl je v tom, že se hodnota `noise` při převodu na rozsah 0 až 1 násobí něčím větším než 0.5 a pokryje tedy větší rozsah hodnot. Nechceme ale omylem vybočit z rozsahu 0 až 1, proto ještě finální hodnotu „clampneme“ do tohoto rozsahu.

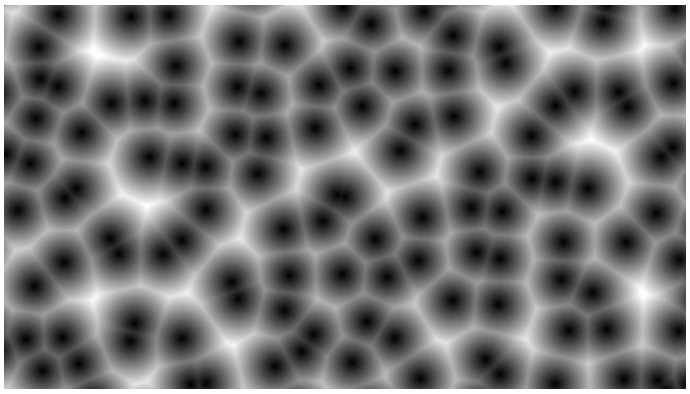


Cellular Noise

Dalším typem šumu je šum buňkový, anglicky známy jako **Cellular Noise** či **Worley Noise**. Jako u Perlina i zde rozdělíme plochu na čtvercové buňky. V každé buňce vygenerujeme jeden náhodný bod. Šum samotný je potom vzdálenost k nejbližšímu bodu (ať už leží v libovolné buňce).

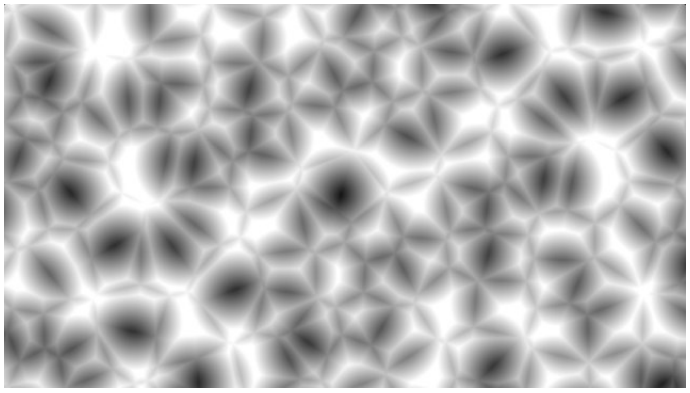
```
vec2 random2(vec2 st)
{
  vec2 s = vec2(
        dot(st, vec2(12.3456, 34.1415)),
        dot(st, vec2(42.2154, 15.2854))
  );
  return fract(sin(s) * 45678.9) * 2.0 - 1.0;
}
float cellularNoise(vec2 st)
{
  ivec2 cell = ivec2(floor(st));
  vec2 f = st - vec2(cell);
  float minDist = 1.0;
  for (int y = -1; y <= 1; y++) {
    for (int x = -1; x <= 1; x++) {
      ivec2 localCell = cell + ivec2(x, y);
      vec2 localPoint = random2(vec2(localCell))
                      * 0.5 + 0.5 + vec2(x, y);
      minDist = min(
          minDist, length(localPoint - f));
    }
  }
  return minDist;
}
```

Všimněte si, že stačí zkontrolovat jen 9 buněk pro každý dotaz – buňku aktuálního pixelu a osm sousedních.



Úkol 1 [3b]:

Další zajímavou variantou této funkce je nevracet vzdálenost nejbližšího bodu, ale druhého nejbližšího. A právě to je vaším úkolem. Výsledek by měl vypadat nějak takto:



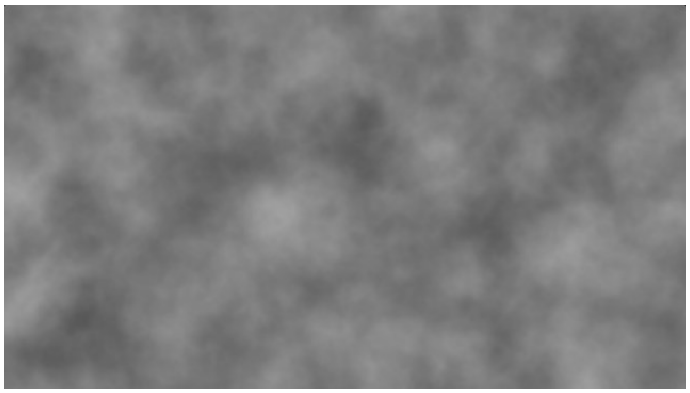
Fractal brownian motion

Ještě se podíváme na jedno velmi užitečné „šumové primitivum“. Na Perlin (či Gradient) Noise se dá dívat jako na „sinusoidu“. Kopečky sice mají různou velikost, ale rychlost změny hodnoty (frekvence) zůstává stejná. Co když sečteme několik šumů o různých frekvencích dohromady? Výsledek se nazývá **Fractal brownian motion**, zkráceně **fbm**. Kdyby byla suma nekonečná, skutečně by se jednalo o fraktál.

```
float fbm(vec2 st)
{
    float val = 0.0;

    float p = 0.5;

    const float angle = 1.0;

    mat2 rotation = mat2(
        cos(angle), sin(angle),
       -sin(angle), cos(angle)
    );

    for (int i = 0; i < 5; i++)
    {
        val += gradientNoise(st) * p;
        p *= 0.5;
        st *= 2.0;

        st += vec2(1.181, 0.57);
        st = rotation * st;
    }

    return val;
}
```

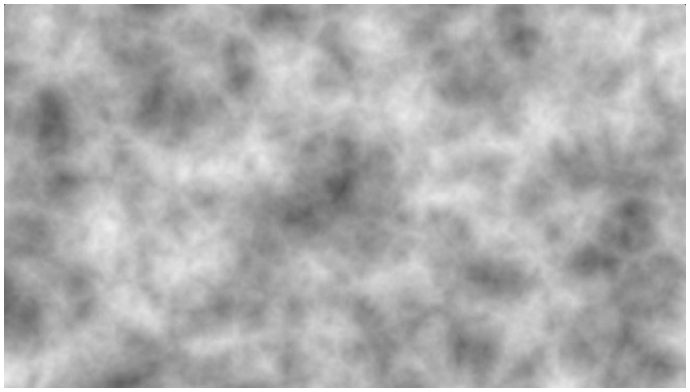
Ve funkci sčítáme několikrát Gradient Noise. Každá tato iterace se nazývá **oktáva**. Následující oktáva má vždy poloviční amplitudu (`p *= 0.5;`) a dvojnásobnou frekvenci (`st *= 2.0;`) než předchozí. Také souřadnice `st` v každé iteraci nějak posuneme a otočíme.

Výsledek této funkce by měl připomínat kouř či mraky. Zkuste ubrat nebo naopak přidat oktávy.



Úkol 2 [3b]:

Dalším vaším úkolem je aplikovat stejný princip jako u *fbm* i na buňkový šum - sečtěte několik buňkových šumů o různých frekvencích. Výsledek je další užitečné „šumové primitivum“:



Úkol 3 [3b]:

Zkuste co se stane, když do sebe „zapojíte“ tři *fbm*.

Bude potřeba vytvořit si vlastní „barevnou“ verzi funkce *fbm*, která nevrací jedinou hodnotu, ale alespoň `vec2`, aby šla přičítat k souřadnicím. Také je lepší, když tato hodnota bude v rozsahu -1 až 1 místo 0 až 1. Ve skutečnosti se hodí, aby vracela rovnou `vec3`, ve vnitřních funkcích z něj použijeme jen první dvě složky a ve vnější z něj získáme barvu (nezapomeňte ji převést do rozsahu 0 až 1).

Též můžete výsledky některé z vnořených `fbm` něčím pronásobit, aby měly větší vliv na vnější `fbm`.

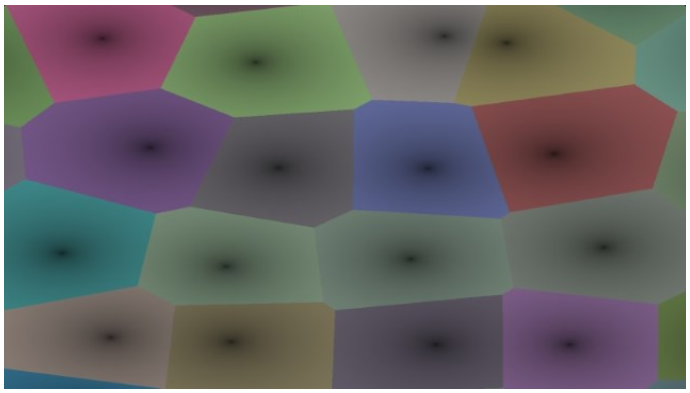
Také lze místo některé `fbm` použít vícevrstvý buňkový šum (opět upravený na barevný) z předchozího úkolu.

Úkol 4 [2b]:

Animujte (posunujte) souřadnice `uv` z předchozího úkolu v čase. Zkuste souřadnice v různě vnořených `fbm` (či Worleyho šumech) animovat různě rychle a různým směrem.

Úkol 5 [4b]:

Vraťme se k základnímu buňkovému šumu. Co když jako zdroj pozic bodů v buňkách použijeme místo úplné náhody (`random2`) něco sofistikovanějšího, třeba `fbm`? Ve statickém obrázku se nich moc nestane, proto zkuste pozice těchto bodů animovat v čase (třeba posunujte souřadnice, ze kterých se čte `fbm`). Navíc z buňkového šumu vracejte, který bod byl nakonec nejbližší a různé buňky poté různě obarvěte.



Tím jsme si prošli základní šumové funkce. Jak jsme zmínili na začátku, jejich využití je především procedurální generování různých (nejen) přírodních útvarů. A na to, jak se něco takového dělá, se lépe podíváme v příštím díle. Také se v příštím díle podíváme, jak se zhruba chová světlo a jak naše výtvory nasvětlit.

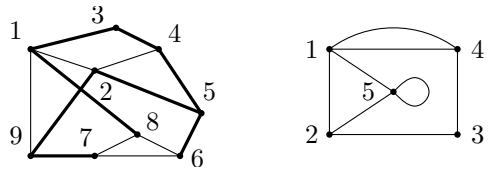
Kuba Pelc

Recepty z programátorské kuchařky: Grafy

V dnešním vydání známého bestselleru budeme péci grafy souvislé i nesouvislé, orientované i neorientované. Řekneme si o základním procházení grafem, komponentách souvislosti, topologickém uspořádání a dalších 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 E, 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*). Obvykle také předpokládáme, že vrcholů je konečně mnoho. Neorientovaný graf většinou zobrazujeme jako body pospojované čarami.



Neorientovaný graf a 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, …, 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, totiž obsahuje nějaký vrchol u dvakrát. Existuje tedy $i < j$ takové, že $u = v_i = v_j$. Pak ale můžeme z našeho sledu vypustit posloupnost e_i, v_{i+1}, …, 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.

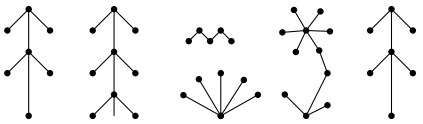
Kružnicí neboli *cyklem* nazýváme cestu délky alespoň 3, ve které oproti definici cesty navíc platí $v_1 = v_n$. Někdy se na cesty, tahy a kružnice v grafu také díváme jako na podgrafy, které získáme tak, že z grafu vypustíme všechny ostatní vrcholy a hrany.

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

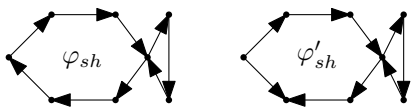
Někdy se hodí jeden z vrcholů stromu prohlásit za *kořen,* čímž jsme si v každém vrcholu určili směr nahoru (ke kořeni – je to zvláštní, ale matematici obvykle kreslí stromy kořenem vzhůru) a dolů (od kořene). Souseda vrcholu směrem nahoru pak nazýváme jeho *otcem,* sousedy směrem dolů jeho *syny.*

Kostra souvislého grafu říkáme každému jeho podgrafu, který je stromem a spojuje všechny vrcholy grafu. Můžeme ji například získat tak, že dokud jsou v grafu kružnice, odebíráme hrany ležící na nějaké kružnici. Pro nesouvislé grafy nazveme kostrou les tvořený kostrami jednotlivých komponent. Na prvním obrázku je jedna z koster levého grafu znázorněna silnými hranami.

Cvičení: Zkuste si dokázat, že stromy jsou právě grafy, které jsou souvislé a mají o jedna méně hran než vrcholů.

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. Orientovaný graf většinou zobrazujeme jako body spojené šipkami. Většina pojmů, které jsme definovali pro neorientované grafy, dává smysl i pro grafy orientované, jen si musíme dát pozor na směr hran.



Silně a slabě souvislý orientovaný graf

Se souvislostí orientovaných grafů je to trochu složitější. Rozlišujeme slabou a silnou souvislost: *slabě souvislý* je graf tehdy, pokud se z něj zapomenutím orientace hran stane souvislý neorientovaný graf. *Silně souvislým* ho nazveme tehdy, vede-li mezi každými dvěma vrcholy x a 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.

Ohodnocené grafy

Další možností, jak si graf „vyzdobit“, je ohodnotit jeho hrany čísly. Například v grafu silniční sítě (vrcholy jsou města, hrany silnice mezi nimi) je zcela přirozené ohodnotit hrany délkami silnic nebo třeba mýtným vybíraným za průjezd silnicí. Přiřazeným číslům se proto často říká *délky* hran nebo jejich *ceny*. Pojmy, které jsme si před chvílí nadefinovali pro obyčejné grafy, můžeme opět snadno rozšířit pro grafy ohodnocené – např. délku sledu budeme namísto počtu hran sledu počítat jako součet jejich ohodnocení. Neohodnocený graf pak odpovídá grafu, v němž mají všechny hrany jednotkovou délku.

Podobně můžeme přiřazovat ohodnocení i vrcholům, ale raději si všechny operace s ohodnocenými grafy necháme na některé z dalších dílů Kuchařky. I tak budeme mít práce dost a dost.

Reprezentace grafů

Nyní už víme o grafech hodně, ale ještě jsme si neřekli, jak graf reprezentovat v paměti počítače. To můžeme udělat například tak, že vrcholy očíslujeme přirozenými čísly od 1 do N, hrany od 1 do M a odkud kam vedou hrany, popíšeme jedním z následujících tří způsobů:

```
  123456789
1 011000011
2 100110001
3 100100000
4 011010000
5 010101000
6 000010110
7 000001011
8 100001100
9 110000100
```

- *matice sousednosti* – to je pole A velikosti $N \times N$. Na pozici $A[i,j]$ uložíme hodnotu 0 nebo 1 podle toho, zda z vrcholu i do vrcholu j hrana vede (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.
- *seznam sousedů* je obvykle tvořen dvěma poli: polem sousedů $S[1 \ldots M]$ obsahujícím postupně čísla všech vrcholů, do kterých vede hrana z vrcholu 1, pak z vrcholu 2 atd., a polem začátků $Z[1 \ldots N]$, v němž se pro každý vrchol dozvíme začátek odpovídajícího úseku v poli S. Pokud navíc do $Z[N+1]$ uložíme $M+1$, bude platit, že sousedé vrcholu i jsou uloženi v $S[Z[i]], \ldots, S[Z[i+1]-1]$. Tato reprezentace má tu výhodu, že zabírá pouze prostor $\mathcal{O}(N+M)$ 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
$S[i]$	2	3	8	9	1	4	5	9	1	4	2	3	5	2

i	15	16	17	18	19	20	21	22	23	24	25	26	27	28
$S[i]$	4	6	5	7	8	6	8	9	1	6	7	1	2	7

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

Reprezentace grafu seznamem sousedů

- *půlhranami* – tato reprezentace se používá tehdy, pokud potřebujeme během výpočtu graf složitě upravovat. Je univerzální, ale dost pracná na naprogramování. Spočívá v tom, že si každou hranu uložíme jako dvě půlhrany (začátek a konec hrany), každý vrchol bude obsahovat spojové seznamy přicházejících a odcházejících půlhran a každá půlhrana bude ukazovat na svou druhou polovici a na vrchol, ze kterého vychází.

V následujících receptech budeme vždy používat seznamy sousedů, poli S budeme říkat `sousedi`, poli Z `zacatky`. Pole `sousedi` bude mít rozsah od 0 do $M-1$, pole `zacatky` od 0 do N (kde N je počet vrcholů a M počet hran).

Prohledávání do hloubky

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 zatímco prvky přidáváme také na konec, odebíráme je z téhož konce. Anglický název je (překvapivě) *last in, first 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 do zásobníku a označíme.
4. Kroky 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: Předpokládejme, 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 ve 2. kroku pokaždé odebereme jeden vrchol ze zásobníku, musí vrcholy někdy (konkrétně po nejvýše N opakováních cyklu) dojít. Ve 3. kroku 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, čili $\mathcal{O}(N + M)$. Paměťová složitost je stejná, protože si tak jako tak musíme hrany a vrcholy pamatovat a zásobník není větší než paměť na vrcholy.

Prohledávání do hloubky implementujeme nejsnáze rekurzivní funkcí. Jako zásobník v tom případě používáme přímo zásobník programu, kde si program ukládá návratové adresy funkcí. Může to vypadat třeba následovně:

```
oznacen = [False] * N

def projdi(v):
  oznacen[v] = True
  for i in range(zacatky[v], zacatky[v+1]):
    if not oznacen[sousedi[i]]:
      projdi(sousedi[i])
```

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 $\mathcal{O}(N_i + M_i)$, kde N_i a M_i je počet vrcholů a hran komponenty, vyjde dohromady složitost $\mathcal{O}(N + M)$. Nic nového si ukládat nemusíme, a proto je paměťová složitost stále $\mathcal{O}(N + M)$.

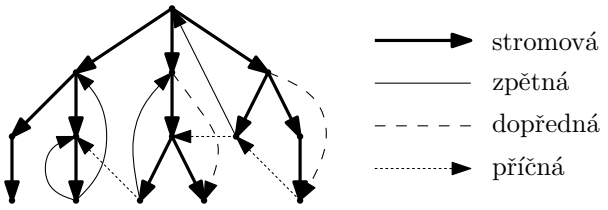
```
def projdi(v):
  komponenta[v] = nova_komponenta
  for i in range(zacatky[v], zacatky[v+1]):
    if komponenta[sousedi[i]] == -1:
      projdi(sousedi[i])

...
for i in range(N):
  komponenta[i] = -1
nova_komponenta = 1
for i in range(N):
  if komponenta[i] == -1:
    projdi(i)
    nova_komponenta += 1
...
```

Průběh prohledávání grafu do hloubky můžeme znázornit stromem (říká se mu DFS strom – podle anglického názvu Depth-First Search pro prohledávání do hloubky). Z počátečního vrcholu w učiníme kořen. Pak budeme graf procházet do hloubky a vrcholy zakreslovat jako syny vrcholů, ze kterých jsme přišli. Syny každého vrcholu si uspořádáme v pořadí, v němž jsme je navštívili; tomuto pořadí budeme říkat *zleva doprava* a také ho tak budeme kreslit. Hranám mezi otci a syny budeme říkat *stromové hrany*. Protože jsme do žádného vrcholu nešli dvakrát, budou opravdu tvořit strom. Hrany, které vedou do již navštívených vrcholů na cestě, kterou jsme přišli z kořene, nazveme *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.

Prohledávání do hloubky lze tedy také využít k nalezení kostry neorientovaného grafu, což je strom, který jsme prošli. Rovnou při tom také zjistíme, zda graf neobsahuje cyklus: to poznáme tak, že nalezneme zpětnou hranu různou od té stromové, po níž jsme do vrcholu přišli.

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



Strom prohledávání do hloubky a typy hran

Prohledávání do šířky

Prohledávání do šířky je založené na podobné myšlence jako prohledávání do hloubky, pouze místo zásobníku používá 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 jehož $H[v] = -1$, přidáme do fronty a nastavíme jeho $H[v]$ na $H[u]+1$.
4. Kroky 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. To vše se dokazuje podobně, jako jsme dokázali správnost prohledávání do hloubky.

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 v. 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 její předposlední vrchol z. 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 $H[v] = H[z] + 1 = D[v]$, a to 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 $\mathcal{O}(N+M)$. Algoritmus implementujeme nejsnáze cyklem, který bude pracovat s vrcholy v poli představujícím frontu.

```
...
for i in range(N):
  H[i] = -1
prvni = 1
posledni = 1
fronta[prvni] = pocatecni_vrchol
h[pocatecni_vrchol] = 0

while True:
  v = fronta[prvni]
  for i in range(zacatky[v], zacatky[v+1]):
    if h[sousedi[i]] < 0:
      h[sousedi[i]] = h[v] + 1
      posledni += 1
      fronta[posledni] = sousedi[i]
  prvni += 1
  if prvni > posledni: break
...
```

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

Topologické uspořádání

Teď si vysvětlíme, co je *topologické uspořádání* grafu. 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 aby pro každou hranu $e = (v_i, v_j)$ bylo $i > j$. Představme si to 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 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 nelze splnit.

Cyklus je ovšem to jediné, co může existenci topologického uspořádání zabránit. Libovolný acyklický graf lze uspořádat následujícím algoritmem:

1. Na začátku máme orientovaný graf G a proměnnou $p = 1$.
2. Najdeme takový vrchol v, ze kterého nevede žádná hrana (budeme mu říkat *stok*). Pokud v grafu žádný stok 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.

Proč tento algoritmus funguje? Pokud v grafu nalezneme stok, můžeme mu určitě přiřadit číslo menší než všem ostatním vrcholům, protože překážet by nám v tom mohly pouze hrany vedoucí ze stoku ven a ty neexistují. Jakmile stok očíslujeme, můžeme jej z grafu odstranit a pokračovat číslováním ostatních vrcholů. Tento postup musí někdy skončit, jelikož v grafu je pouze konečně mnoho vrcholů.

Zbývá si uvědomit, že v neprázdném grafu, který neobsahuje cyklus, vždy existuje alespoň jeden stok: Vezměme libovolný vrchol v_1. Pokud z něj vede nějaká hrana, pokračujme po ní do nějakého vrcholu v_2, z něj do v_3 atd. Co se při tom může stát?

- Dostaneme se do vrcholu v_i, ze kterého nevede žádná hrana. Vyhráli jsme, máme stok.
- Narazíme na v_i, ve kterém jsme už jednou byli. To by ale znamenalo, že graf obsahuje cyklus, což, jak víme, není pravda.
- Budeme objevovat stále nové a nové vrcholy. V konečném grafu nemožno.

Algoritmus můžeme navíc 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, zkontrolovat si, zda jsme nějakému jinému vrcholu nezrušili poslední hranu, která z něj vedla, a pokud ano, přidat takový vrchol na konec fronty. Celé topologické třídění pak zvládneme v čase $\mathcal{O}(N+M)$.

Jiná možnost je prohledat graf do hloubky a všimnout si, že pořadí, ve kterém jsme se z vrcholů vraceli, je právě topologické pořadí. Pokud zrovna 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 nižší 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 $\mathcal{O}(N+M)$.

```
def projdi(v):
  # zatím V jen označíme
  ocislovani[v] = 0
  for i in range(zacatky[v], zacatky[v+1]):
    if ocislovani[sousedi[i]] == -1:
      projdi(sousedi[i]]
  posledni += 1
  ocislovani[v] = posledni

...
for i in range(N):
  ocislovani[i] = -1
posledni = 0
for i in range(N):
  if ocislovani[i] == -1:
    projdi(i)
...
```

Hranová a vrcholová 2-souvislost

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

- má alespoň 3 vrcholy,
- 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 a DFS strom. Všimněme si, že mostem může být jedině stromová hrana – každá jiná hrana totiž leží na nějaké kružnici. Odebráním mostu se graf rozpadne na část obsahující kořen DFS stromu a podstrom „visící“

pod touto hranou. Jediné, co tomu může zabránit, je existence nějaké další hrany mezi podstromem a hlavní částí, což musí být zpětná hrana, navíc taková, která není jenom stromovou hranou viděnou z druhé strany. Takovým hranám budeme říkat *ryzí zpětné hrany.*

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 nejvyšší hladiny (s nejmenším číslem) vedou ryzí zpětné 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 proto starat.

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

```
def projdi(v, nova_hladina):
  hladina[v] = nova_hladina
  spojeno[v] = hladina

  for i in range(zacatky[v], zacatky[v+1]):
    w = sousedi[i]
    if hladina[w] == -1:
      projdi(w, nova_hladina + 1)
      if spojeno[w] < spojeno[v]:
        spojeno[v] = spojeno[w]
      if spojeno[w] > hladina[v]
        dvoj_souvisle = False
    else:
      if hladina[w] < nova_hladina - 1
         and hladina[w] < spojeno[v]:
        spojeno[v] = hladina[w]

...
for i in range(N):
  hladina[i] = -1
dvoj_souvisle = True
projdi(1, 0)
...
```

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

- má alespoň 3 vrcholy,
- je souvislý,
- zůstane souvislý po odebrání libovolného vrcholu.

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

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.

Dnešní menu servírovali

Martin Mareš, David Matoušek a Petr Škoda