1. Základní grafové algoritmy

Teorie grafů, jak ji známe z diskrétní matematiky, nám dává elegantní nástroj k popisu situací ze skutečného i matematického světa. Díky tomu dovedeme různé praktické problémy překládat na otázky ohledně grafů. To je zajímavé i samo o sobě, ale jak uvidíme v této kapitole, často díky tomu můžeme snadno přijít k rychlému algoritmu.

V celé kapitole budeme pro grafy používat následující značení:

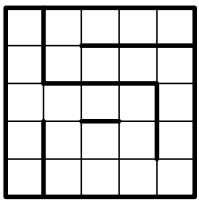
Definice:

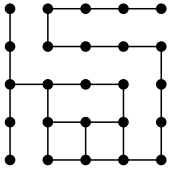
- G je graf, se kterým pracujeme (orientovaný nebo neorientovaný).
- V je množina vrcholů tohoto grafu, E množina jeho hran.
- n značí počet vrcholů, m počet hran.
- uv označuje hranu z vrcholu u do vrcholu v. Podle toho, zda pracujeme s orientovaným grafem, je to formálně buď uspořádaná dvojice (u, v), anebo dvojprvková množina $\{u, v\}$.
- *Následníci* vrcholu v budou vrcholy, do kterých z v vede hrana. Analogicky z *předchůdců* vede hrana do v. Předchůdcům a následníkům dohromady říkáme *sousedé* vrcholu v.

1.1. Pár grafů úvodem

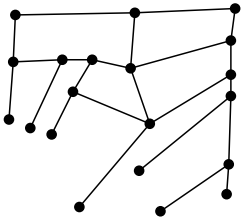
Pojďme se nejprve podívat na několik příkladů, jak praktické problémy modelovat pomocí grafů:

Bludiště na čtverečkovaném papíře: vrcholy jsou čtverečky, hranou jsou spojené sousední čtverečky, které nejsou oddělené zdí. Je přirozené ptát se na *komponenty souvislosti* bludiště, což jsou různé „místnosti“, mezi nimiž nelze přejít (alespoň bez prokopání nějaké zdi). Pokud jsou dva čtverečky v téže místnosti, chceme mezi nimi hledat *nejkratší cestu.*



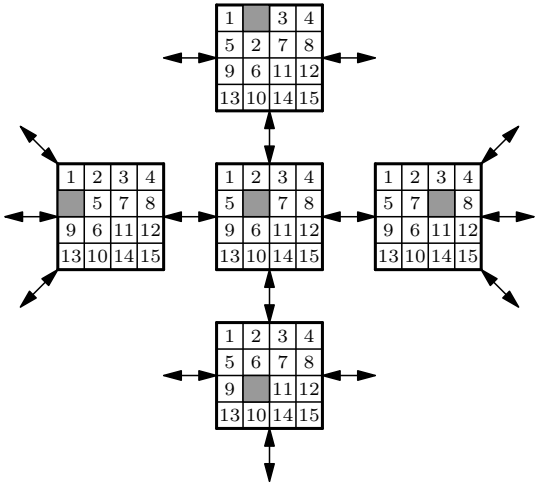


Mapa města je podobná bludišti: vrcholy odpovídají křižovatkám, hrany ulicím mezi nimi. Hrany se hodí *ohodnotit* délkami ulic nebo časy potřebnými na průjezd; pak nás opět zajímají nejkratší cesty. Když městečko zapadá sněhem, *minimální kostra* grafu nám řekne, které silnice chceme prohrnout jako první. *Mosty* a *artikulace* (hrany resp. vrcholy, po jejichž odebrání se graf rozpadne) mají také svůj přirozený význam. Pokud jsou ve městě jednosměrky, budeme uvažovat o orientovaném grafu.



Hlavolam „patnáctka“: v krabičce velikosti 4×4 je 15 očíslovaných jednotkových čtverečků a jedna jednotková díra. Jedním tahem smíme do díry přesunout libovolný čtvereček, který s ní sousedí; matematik by spíš řekl, že můžeme díru prohodit se sousedícím čtverečkem.

Opět se hodí graf: vrcholy jsou *konfigurace* (možná rozmístění čtverečků a díry v krabičce) a hrany popisují, mezi kterými konfiguracemi jde přejít jedním tahem. Tato konstrukce funguje i pro další hlavolamy a hry a obvykle se jí říká *stavový prostor* hry.



Šeherezádino číslo 1 001 je zajímavé například tím, že je nejmenším násobkem sedmi, jehož desítkový zápis sestává pouze z nul a jedniček. Co kdybychom obecně hledali nejmenší násobek nějakého čísla K tvořený jen nulami a jedničkami?

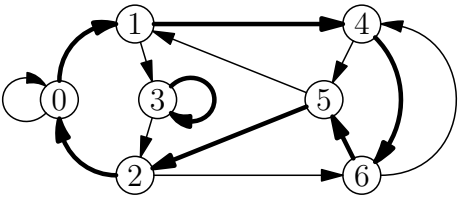
Zatím vyřešíme jednodušší otázku: spokojíme se s libovolným násobkem, ne tedy nutně nejmenším.

Představme si, že takové číslo vytváříme postupným připisováním číslic. Začneme jedničkou. Kdykoliv pak máme nějaké číslo x, umíme z něj vytvořit čísla $x\mathbf{0} = 10x$ a $x\mathbf{1} = 10x + 1$. Všímejme si zbytků po dělení číslem K:

$$(x\mathbf{0}) \bmod K = (10x) \bmod K = (10 \cdot (x \bmod K)) \bmod K,$$
$$(x\mathbf{1}) \bmod K = (10x + 1) \bmod K = (10 \cdot (x \bmod K) + 1) \bmod K.$$

Ejhle: nový zbytek je jednoznačně určen předchozím zbytkem.

V řeči zbytků tedy začínáme s jedničkou a chceme ji pomocí uvedených dvou pravidel postupně přetransformovat na nulu. To je přeci grafový problém: vrcholy jsou zbytky 0 až $K-1$, orientované hrany odpovídají našim pravidlům a hledáme cestu z vrcholu 1 do vrcholu 0.



Obr. 1.1: Graf zbytků pro $K = 7$. Tenké čáry připisují 0, tučné 1.

Cvičení:

1. Kolik nejvýše vrcholů a hran má graf, kterým jsme popsali bludiště z $n \times n$ čtverečků? A kolik nejméně?
2. Kolik vrcholů a hran má graf patnáctky? Vejde se do paměti vašeho počítače? Jak by to dopadlo pro menší verzi hlavolamu v krabičce 3×3?

3.* Kolik má graf patnáctky komponent souvislosti?

4. Kolik vrcholů a hran má graf Šeherezádina problému pro dané K?

5.* Dokažte, že Šeherezádin problém je pro každé $K > 0$ řešitelný.

6.* Jakému grafovému problému by odpovídalo hledání nejmenšího čísla z nul a jedniček dělitelného K? Pokud vás napadla nejkratší cesta, ještě chvíli přemýšlejte.

1.2. Prohledávání do šířky

Základním stavebním kamenem většiny grafových algoritmů je nějaký způsob *prohledávání grafu.* Tím myslíme postupné procházení grafu po hranách od určitého počátečního vrcholu. Možných způsobů prohledávání je víc, zatím ukážeme ten nejjednodušší: *prohledávání do šířky.* Často se mu říká zkratkou *BFS* z anglického *breadth-first search.*

Na vstupu dostaneme konečný orientovaný graf a počáteční vrchol v_0. Postupně nacházíme následníky vrcholu v_0, pak následníky těchto následníků, a tak dále, až objevíme všechny vrcholy, do nichž se dá z v_0 dojít po hranách. Obrazně řečeno, do grafu nalijeme vodu a sledujeme, jak postupuje vlna.

Během výpočtu rozlišujeme tři možné *stavy vrcholů:*

- *Nenalezené* – to jsou ty, které jsme na své cestě grafem dosud nepotkali.

- *Otevřené* – o těch už víme, ale ještě jsme neprozkoumali hrany, které z nich vedou.
- *Uzavřené* – už jsme prozkoumali i hrany, takže se vrcholem nemusíme nadále zabývat.

Na počátku výpočtu tedy chceme prohlásit v_0 za otevřený a ostatní vrcholy za nenalezené. Pak v_0 uzavřeme a otevřeme všechny jeho následníky. Poté procházíme tyto následníky, uzavíráme je a otevíráme jejich dosud nenalezené následníky. A tak dále. Otevřené vrcholy si přitom pamatujeme ve *frontě,* takže pokaždé zavřeme ten z nich, který je otevřený nejdéle.

Následuje zápis algoritmu v pseudokódu. Pomocná pole D a P budou hrát svou roli později (v oddílu 1.5), zatím můžete všechny operace s nimi přeskočit – chod algoritmu evidentně neovlivňují.

Algoritmus BFS
Vstup: Graf $G = (V, E)$ a počáteční vrchol $v_0 \in V$.

1. Pro všechny vrcholy v:
2. $\quad stav(v) \leftarrow$ *nenalezený*
3. $\quad D(v) = \emptyset$, $P(v) \leftarrow \emptyset$
4. $stav(v_0) \leftarrow$ *otevřený*
5. $D(v_0) \leftarrow 0$
6. Založíme frontu Q a vložíme do ní vrchol v_0.
7. Dokud je fronta Q neprázdná:
8. $\quad$ Odebereme první vrchol z Q a označíme ho v.
9. $\quad$ Pro všechny následníky w vrcholu v:
10. $\quad\quad$ Je-li $stav(w) =$ *nenalezený*:
11. $\quad\quad\quad stav(w) \leftarrow$ *otevřený*
12. $\quad\quad\quad D(w) \leftarrow D(v) + 1$, $P(w) \leftarrow v$
13. $\quad\quad\quad$ Přidáme w do fronty Q.
14. $\quad stav(v) \leftarrow$ *uzavřený*

Nyní dokážeme, že BFS skutečně dělá to, co jsme plánovali.

Lemma: Algoritmus BFS se vždy zastaví.

Důkaz: Vnitřní cyklus je evidentně konečný. V každém průchodu vnějším cyklem uzavřeme jeden otevřený vrchol. Jednou uzavřený vrchol už ale svůj stav nikdy nezmění, takže vnější cyklus proběhne nejvýše tolikrát, kolik je všech vrcholů. □

Definice: Vrchol v je *dosažitelný* z vrcholu u, pokud v grafu existuje cesta z u do v.

Poznámka: Cesta se obvykle definuje tak, že se na ní nesmí opakovat vrcholy ani hrany. Na dosažitelnosti se samozřejmě nic nezmění, pokud budeme místo cest uvažovat sledy, na nichž je opakování povoleno:

Lemma: Pokud z vrcholu u do vrcholu v vede sled, pak tam vede i cesta.

Důkaz: Ze všech sledů z u do v vybereme ten nejkratší (co do počtu hran) a nahlédneme, že se jedná o cestu. Vskutku: kdyby se v tomto sledu opakoval nějaký

vrchol t, mohli bychom část sledu mezi první a poslední návštěvou t „vystřihnout“ a získat tak sled o ještě menším počtu hran. A pokud se neopakují vrcholy, nemohou se opakovat ani hrany. □

Lemma: Když algoritmus doběhne, vrcholy dosažitelné z v_0 jsou *uzavřené* a všechny ostatní vrcholy *nenalezené*.

Důkaz: Nejprve si uvědomíme, že každý vrchol buďto po celou dobu běhu algoritmu zůstane nenalezený, nebo se nejprve stane otevřeným a později uzavřeným. Formálně bychom to mohli dokázat indukcí podle počtu iterací vnějšího cyklu.

Dále nahlédneme, že kdykoliv nějaký vrchol w otevřeme, musí být dosažitelný z v_0. Opět indukcí podle počtu iterací: na počátku je otevřený pouze vrchol v_0 sám. Kdykoliv pak otevíráme nějaký vrchol w, stalo se tak proto, že do něj vedla hrana z právě uzavíraného vrcholu v. Přitom podle indukčního předpokladu existuje sled z v_0 do v. Prodloužíme-li tento sled o hranu vw, vznikne sled z v_0 do w, takže i w je dosažitelný.

Zbývá dokázat, že se nemohlo stát, že by algoritmus nějaký dosažitelný vrchol neobjevil. Pro spor předpokládejme, že takové „špatné“ vrcholy existují. Vybereme z nich vrchol s, který je k v_0 nejbližší, tedy do kterého vede z v_0 sled o nejmenším možném počtu hran. Jelikož sám v_0 není špatný, musí existovat vrchol p, který je na sledu předposlední (z p do s vede poslední hrana sledu).

Vrchol p také nemůže být špatný, protože jinak bychom si jej vybrali místo s. Tím pádem ho algoritmus nalezl, otevřel a časem i zavřel. Při tomto zavírání ovšem musel prozkoumat všechny sousedy vrcholu p, tedy i vrchol s. Není proto možné, aby s unikl otevření. □

1.3. Reprezentace grafů

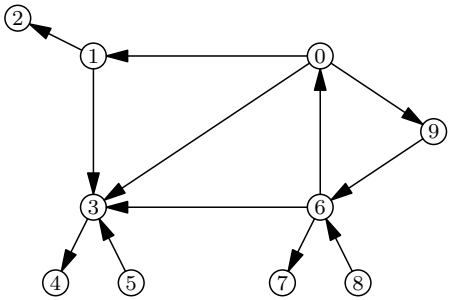
U algoritmu na prohledávání grafu do šířky jsme zatím nerozebrali časovou a paměťovou složitost. Není divu: algoritmus jsme popsali natolik abstraktně, že vůbec není jasné, jak dlouho trvá nalezení všech následníků vrcholu. Nyní tyto detaily doplníme.

Především se musíme rozhodnout, jak grafy reprezentovat v paměti počítače. Obvykle se používají následující způsoby:

Matice sousednosti. Vrcholy očíslujeme od 1 do n, hrany popíšeme maticí $n \times n$, která má na pozici i, j jedničku, je-li ij hrana, a jinak nulu. Jedničky v i-tém řádku tedy odpovídají následníkům vrcholu i, jedničky v j-tém sloupci předchůdcům vrcholu j. Pro neorientované grafy je matice symetrická.

Výhodou této reprezentace je, že dovedeme v konstantním čase zjistit, zda jsou dva vrcholy spojeny hranou. Vyjmenování hran vedoucích z daného vrcholu trvá $\Theta(n)$. Matice zabere prostor $\Theta(n^2)$.

Lepších parametrů obecně nemůžeme dosáhnout, protože sousedů může být až $n-1$ a všech hran až řádově n^2. Ale je to nešikovné, pokud pracujeme s *řídkými*




```
  0123456789
0 0101000001
1 0011000000
2 0000000000
3 0000100000
4 0000000000
5 0001000000
6 1001000100
7 0000000000
8 0000001000
9 0000001000
```

Obr. 1.2: „Prasátko“ a jeho matice sousednosti

grafy, tedy takovými, které mají méně než kvadraticky hran. Ty potkáváme nečekaně často – řídké jsou například všechny rovinné grafy, čili i stromy.

Seznamy sousedů. Vrcholy opět očíslujeme od 1 do n. Pro každý vrchol uložíme seznam čísel jeho následníků. Přesněji řečeno, pořídíme si pole S, jehož i-tý prvek bude ukazovat na seznam následníků vrcholu i. V neorientovaném grafu zařadíme hranu ij do seznamů $S[i]$ i $S[j]$.

0: 1, 3, 9	1: 2, 3	2:	3: 4	4:
5: 3	6: 0, 3, 7	7:	8: 6	9: 6

Obr. 1.3: Seznamy následníků pro prasátkový graf

Vyjmenování hran tentokrát stihneme lineárně s jejich počtem. Celá reprezentace zabírá prostor $\Theta(n + m)$. Ovšem test existence hrany ij se zpomalil: musíme projít všechny následníky vrcholu i nebo všechny předchůdce vrcholu j.

Často se používají různá rozšíření této reprezentace. Například můžeme přidat seznamy předchůdců, abychom uměli vyjmenovávat i hrany vedoucí *do* daného vrcholu. Také můžeme čísla vrcholů nahradit ukazateli, pole seznamem a získat tak možnost graf za běhu libovolně upravovat. Pro neorientované grafy se pak hodí, aby byly oba výskyty téže hrany navzájem propojené.

Komprimované seznamy sousedů. Pokud chceme šetřit pamětí, může se hodit zkomprimovat seznamy následníků (či všech sousedů) do polí. Pořídíme si pole $S[1 \dots m]$, ve kterém budou za sebou naskládaní nejdříve všichni následníci vrcholu 1, pak následníci dvojky, atd. Navíc založíme „rejstřík“ – pole $R[1 \dots n]$, jehož i-tý prvek ukazuje na prvního následníka vrcholu i v poli S. Pokud navíc dodefinujeme $R[n+1] = m+1$, bude vždy platit, že následníci vrcholu i jsou v S na pozicích $R[i]$ až $R[i+1] - 1$.

Fix!

Tím jsme ušetřili ukazatele, což sice asymptotickou paměťovou složitost nezměnilo, ale přesto se to u velkých grafů může hodit. Základní operace jsou řádově

Fix: Číslování od 0 nebo od 1? V příkladu nefunguje.

i	1	2	3	4	5	6	7	8	9	10	11	12
$S[i]$	1	3	9	2	3	4	3	0	3	7	6	6

i	0	1	2	3	4	5	6	7	8	9	10
$R[i]$	1	4	6	6	7	7	8	11	11	12	13

Obr. 1.4: Komprimované seznamy následníků pro prasátkový graf

stejně rychlé jako před kompresí, ovšem výrazně jsme zkomplikovali jakékoliv úpravy grafu.

Matice incidence. Často v literatuře narazíme na další maticovou reprezentaci grafů. Jedná se o matici tvaru $n \times m$, jejíž řádky jsou indexovány vrcholy a sloupce hranami. Sloupec, který popisuje hranu ij, má v i-tém řádku hodnotu -1, v j-tém řádku hodnotu 1 a všude jinde nuly. Pro neorientované grafy se znaménka buďto volí libovolně, nebo jsou obě kladná.

Tato matice hraje pozoruhodnou roli v důkazech různých vět na pomezí teorie grafů a lineární algebry (například Kirchhoffovy věty o počítání koster grafu pomocí determinantů). V algoritmech se ovšem nehodí – je obrovská a všechny základní grafové operace jsou s ní pomalé.

Vraťme se nyní k prohledávání do šířky. Pokud na vstupu dostane graf reprezentovaný seznamy sousedů, o jeho časové složitosti platí:

Lemma: BFS doběhne v čase $\mathcal{O}(n+m)$ a spotřebuje paměť $\Theta(n+m)$.

Důkaz: Inicializace algoritmu (kroky 1 až 6) trvá $\mathcal{O}(n)$. Jelikož každý vrchol uzavřeme nejvýše jednou, vnější cyklus proběhne nejvýše n-krát. Pokaždé spotřebuje konstantní čas na svou režii a navíc konstantní čas na každého nalezeného následníka. Celkem tedy $\mathcal{O}(n + \sum_i d_i)$, kde d_i je počet následníků vrcholu i. Tato suma je rovna počtu hran.

(Také si můžeme představovat, že algoritmus zkoumá vrcholy i hrany, obojí v konstantním čase. Každou hranu přitom prozkoumá v okamžiku, kdy uzavírá vrchol, z nějž tato hrana vede, čili právě jednou.)

Paměť jsme potřebovali na reprezentaci grafu, lineárně velkou frontu a lineárně velká pole. □

Cvičení:

1. Jak reprezentovat multigrafy (grafy s násobnými hranami) a jak grafy se smyčkami (hranami typu ii)?
2. Jakou časovou složitost by BFS mělo, pokud bychom graf reprezentovali maticí sousednosti?
3. Navrhněte reprezentaci grafu, která bude efektivní pro řídké grafy, a přitom dokáže rychle testovat existenci hrany mezi zadanými vrcholy.
4. Je-li A matice sousednosti grafu, co popisuje matice A^2? A co A^k? (Mocniny matic definujeme takto: $A^1 = A$, $A^{k+1} = A^k A$.)

5.* Na základě předchozího cvičení vytvořte algoritmus, který pomocí $\mathcal{O}(\log n)$ násobení matic spočítá *matici dosažitelnosti.* To je nula-jedničková matice A^*, v níž $A^*_{ij} = 1$ právě tehdy, když z i do j vede cesta. **Fix!**

6. Je-li I matice incidence grafu, co popisují matice $I^T I$ a II^T?

1.4. Komponenty souvislosti

Jakmile umíme prohledávat graf, hned dokážeme odpovídat na některé jednoduché otázky. Například umíme snadno zjistit, zda zadaný neorientovaný graf je souvislý: vybereme si libovolný vrchol a spustíme z něj BFS. Pokud jsme navštívili všechny vrcholy, graf je souvislý. V opačném případě jsme prošli celou jednu komponentu souvislosti. K nalezení ostatních komponent stačí opakovaně vybírat dosud nenavštívený vrchol a spouštět z něj prohledávání.

Následuje pseudokód algoritmu odvozený od BFS a mírně zjednodušený: zbytečně neinicializujeme vrcholy vícekrát a nerozlišujeme mezi otevřenými a uzavřenými vrcholy. Místo toho udržujeme pole C, které o navštívených vrcholech říká, do které komponenty patří; komponentu přitom identifikujeme nejmenším číslem vrcholu, který v ní leží.

Algoritmus KOMPONENTY
Vstup: Neorientovaný graf $G = (V, E)$

1. Pro všechny vrcholy v položíme $C(v) \leftarrow$ *nedefinováno*.
2. Pro všechny vrcholy u postupně provádíme:
3. Je-li $C(u)$ nedefinováno: *(nová komponenta, spustíme BFS)*
4. $C(u) \leftarrow u$
5. Založíme frontu Q a vložíme do ní vrchol u.
6. Dokud Q není prázdná:
7. Odebereme první vrchol z Q a označíme ho v.
8. Pro všechny následníky w vrcholu v:
9. Pokud $C(w)$ není definováno:
10. $C(w) \leftarrow u$
11. Přidáme w do fronty Q.

Výstup: Pole C přiřazující komponenty vrcholům

Korektnost algoritmu je zřejmá. Pro rozbor složitosti označme n_i a m_i počet vrcholů a hran v i-té nalezené komponentě. Prohledání této komponenty trvá $\Theta(n_i + m_i)$. Kromě toho algoritmus provádí inicializaci a hledá dosud neoznačené vrcholy, což se obojí týká každého vrcholu jen jednou. Celkově tedy spotřebuje čas $\Theta(n + \sum_i (n_i + m_i))$, což je rovno $\Theta(n + m)$, neboť každý vrchol i hrana leží v právě jedné komponentě. Paměti potřebujeme $\Theta(n + m)$.

Cvičení:

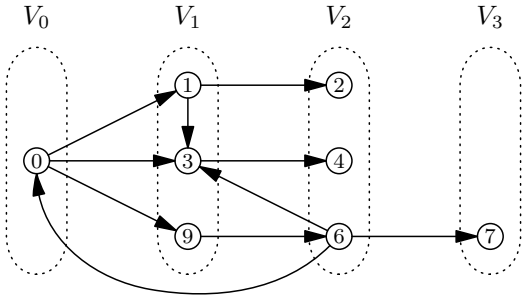
1. Navrhněte algoritmus, který v čase $\mathcal{O}(n+m)$ zjistí, zda zadaný graf je bipartitní.

Fix: Odkázat na Strassenův algoritmus z kapitoly Rozděl a panuj.

1.5. Vrstvy a vzdálenosti

Z průběhu prohledávání do šířky lze zjistit spoustu dalších zajímavých informací o grafu. K tomu se budou hodit pomocná pole D a P, jež jsme si už v algoritmu přichystali.

Především můžeme vrcholy rozdělit do *vrstev* podle toho, v jakém pořadí je BFS prochází: ve vrstvě V_0 bude ležet vrchol v_0 a kdykoliv budeme zavírat vrchol z vrstvy V_k, jeho otevírané následníky umístíme do V_{k+1}. Algoritmus má tedy v každém okamžiku ve frontě vrcholy z nějaké vrstvy V_k, které postupně uzavírá, a za nimi přibývají vrcholy tvořící vrstvu V_{k+1}.



Obr. 1.5: Vrstvy při prohledávání grafu z obr. 1.2 do šířky

Lemma: Je-li vrchol v dosažitelný, pak leží v nějaké vrstvě V_k a číslo $D(v)$ na konci výpočtu je rovno k. Navíc pokud $v \neq v_0$, pak $P(v)$ je nějaký předchůdce vrcholu v ležící ve vrstvě V_{k-1}. Tím pádem v, $P(v)$, $P(P(v))$, …, v_0 tvoří cestu z v_0 do v (zapsanou pozpátku).

Důkaz: Vše provedeme indukcí podle počtu kroků algoritmu. Využijeme toho, že $D(v)$ a $P(v)$ se nastavují v okamžiku uzavření vrcholu v a pak už se nikdy nezmění. □

Čísla vrstev mají ovšem zásadnější význam:

Lemma: Je-li vrchol v dosažitelný z v_0, pak na konci výpočtu $D(v)$ udává jeho vzdálenost od v_0, měřenou počtem hran na nejkratší cestě.

Důkaz: Označme $d(v)$ skutečnou vzdálenost z v_0 do v. Z předchozího lemmatu víme, že z v_0 do v existuje cesta délky $D(v)$. Proto $D(v) \geq d(v)$.

Opačnou nerovnost dokážeme sporem: Nechť existují vrcholy, pro které je $D(v) > d(v)$. Takovým budeme opět říkat *špatné vrcholy* a vybereme z nich vrchol s, jehož skutečná vzdálenost $d(s)$ je nejmenší.

Uvážíme nejkratší cestu z v_0 do s a předposlední vrchol p na této cestě. Jelikož $d(p) = d(s) - 1$, musí p být dobrý (viz též cvičení 1.5.2), takže $D(p) = d(p)$. Nyní zaostřeme na okamžik, kdy algoritmus zavírá vrchol p. Tehdy musel objevit vrchol s jako následníka. Pokud byl v tomto okamžiku s dosud nenalezený, musel padnout do vrstvy $d(p) + 1 = d(s)$, což je spor. Jenže pokud už byl otevřený nebo dokonce uzavřený, musel dokonce padnout do nějaké dřívější vrstvy, což je spor tím spíš. □

Strom prohledávání a klasifikace hran

Vrstvy vypovídají nejen o vrcholech, ale také o hranách grafu. Je-li ij hrana, rozlišíme následující možnosti:

- $D(j) < D(i)$ – hrana vede do některé z minulých vrstev. V okamžiku uzavírání i byl j už uzavřený. Takovým hranám budeme říkat *zpětné.*
- $D(j) = D(i)$ – hrana vede v rámci téže vrstvy. V okamžiku uzavírání i byl j buďto uzavřený, nebo ještě otevřený. Tyto hrany se nazývají *příčné.*
- $D(j) = D(i) + 1$ – hrana vede do následující vrstvy (povšimněte si, že nemůže žádnou vrstvu přeskočit, protože by neplatila trojúhelníková nerovnost pro vzdálenost).
  - Pokud při uzavírání i byl j dosud nenalezený, tak jsme j právě otevřeli a nastavili $P(j) = i$. Tehdy budeme hraně ij říkat *stromová* a za chvíli prozradíme, proč.
  - V opačném případě byl j otevřený a hranu prohlásíme za *dopřednou.*

Lemma: Stromové hrany tvoří strom na všech dosažitelných vrcholech, orientovaný směrem od kořene, jímž je vrchol v_0. Cesta z libovolného vrcholu v do v_0 v tomto stromu je nejkratší cestou z v_0 do v v původním grafu. Proto se tomuto stromu říká *strom nejkratších cest.*

Důkaz: Graf stromových hran musí být strom s kořenem v_0, protože vzniká z vrcholu v_0 postupným přidáváním listů. Na každé hraně přitom roste číslo vrstvy o 1 a jak už víme, čísla vrstev odpovídají vzdálenostem, takže cesty ve stromu jsou nejkratší. □

Dodejme ještě, že stromů nejkratších cest může pro jeden graf existovat vícero (ani nejkratší cesty samotné nejsou jednoznačně určené, pouze vzdálenosti). Každý takový strom je ovšem kostrou prohledávaného grafu.

Pozorování: V neorientovaných grafech BFS potká každou dosažitelnou hranu dvakrát: buďto poprvé jako stromovou a podruhé jako zpětnou, nebo nejdříve jako dopřednou a pak jako zpětnou, anebo v obou případech jako příčnou. Tím pádem nemohou existovat zpětné hrany, které by se vracely o víc než jednu vrstvu.

Nyní pojďme vše, co jsme zjistili o algoritmu BFS, shrnout do následující věty:

Věta: Prohledávání do šířky doběhne v čase $\mathcal{O}(n+m)$ a spotřebuje prostor $\Theta(n+m)$. Po skončení výpočtu popisuje pole *stav* dosažitelnost z vrcholu v_0, pole D obsahuje vzdálenosti od vrcholu v_0 a pole P kóduje strom nejkratších cest.

Cvičení:

1. Upravte BFS tak, aby pro každý dosažitelný vrchol zjistilo, kolik do něj vede nejkratších cest z počátečního vrcholu. Zachovejte lineární časovou složitost.

2. V důkazu lemmatu o vzdálenostech jsme považovali za samozřejmost, že usekneme-li nejkratší cestu z v_0 do s v nějakém vrcholu p, zbude z ní nejkratší cesta z v_0 do p. Jinými slovy, prefix nejkratší cesty je zase nejkratší cesta. Dokažte formálně, že je to pravda.
3. BFS v každém okamžiku zavírá nejstarší otevřený vrchol. Jak by se chovalo, kdybychom vybírali otevřený vrchol podle nějakého jiného kriteria? Která z dokázaných lemmat by stále platila a která ne?

1.6. Prohledávání do hloubky

Dalším důležitým algoritmem k procházení grafů je *prohledávání do hloubky,* anglicky *depth-first search* čili *DFS.* Je založeno na podobném principu jako BFS, ale vrcholy zpracovává rekurzivně: kdykoliv narazí na dosud nenalezený vrchol, otevře ho, zavolá se rekurzivně na všechny jeho dosud nenalezené následníky, načež původní vrchol zavře a vrátí se z rekurze.

Algoritmus opět zapíšeme v pseudokódu a rovnou ho doplníme o pomocná pole *in* a *out*. Do nich zaznamenáme, v jakém pořadí jsme vrcholy otevírali a zavírali.

Algoritmus DFS
Vstup: Graf $G = (V, E)$ a počáteční vrchol $v_0 \in V$.

1. Pro všechny vrcholy v:
2. $\quad$ $stav(v) \leftarrow$ *nenalezený*
3. $\quad$ $in(v), out(v) \leftarrow$ *nedefinováno*
4. $T \leftarrow 0$ *(globální počítadlo kroků)*
5. Zavoláme DFS2(v_0).

Procedura DFS2(v)

1. $stav(v) \leftarrow$ *otevřený*
2. $T \leftarrow T + 1$, $in(v) \leftarrow T$
3. Pro všechny následníky w vrcholu v:
4. $\quad$ Je-li $stav(w) =$ *nenalezený*, zavoláme DFS2(w).
5. $stav(v) \leftarrow$ *uzavřený*
6. $T \leftarrow T + 1$, $out(v) \leftarrow T$

Rozbor algoritmu povedeme podobně jako u BFS.

Lemma: DFS doběhne v čase $\mathcal{O}(n + m)$ a prostoru $\Theta(n + m)$.

Důkaz: Každý vrchol, který nalezneme, přejde nejprve do otevřeného stavu a posléze do uzavřeného, kde už setrvá. Každý vrchol proto uzavíráme nejvýše jednou a projdeme při tom hrany, které z něj vedou. Strávíme tak konstantní čas nad každým vrcholem a každou hranou.

Kromě reprezentace grafu algoritmus potřebuje lineárně velkou paměť na pomocná pole a na zásobník rekurze. □

Lemma: DFS navštíví právě ty vrcholy, které jsou z v_0 dosažitelné.

Důkaz: Nejprve indukcí podle běhu algoritmu nahlédneme, že každý navštívený vrchol je dosažitelný. Opačnou implikaci zase dokážeme sporem: ze špatných vrcholů (dosažitelných, ale nenavštívených) vybereme ten, který je k v_0 nejbližší, a zvolíme jeho předchůdce na nejkratší cestě. Ten nemůže být špatný, takže byl otevřen, a tím pádem musel být posléze otevřen i náš špatný vrchol. Spor. □

Prohledávání do hloubky nemá žádnou přímou souvislost se vzdálenostmi v grafu. Přesto pořadí, v němž navštěvuje vrcholy, skýtá mnoho pozoruhodných vlastností. Ty nyní prozkoumáme.

Chod algoritmu můžeme elegantně popsat pomocí řetězce závorek. Kdykoliv vstoupíme do vrcholu, připíšeme k řetězci levou závorku; až budeme tento vrchol opouštět, připíšeme pravou. Z průběhu rekurze je vidět, že jednotlivé páry závorek se nebudou křížit – dostali jsme tedy dobře uzávorkovaný řetězec s n levými a n pravými závorkami. Hodnoty $in(v)$ a $out(v)$ nám řeknou, kde v tomto řetězci leží levá a pravá závorka přiřazená vrcholu v.

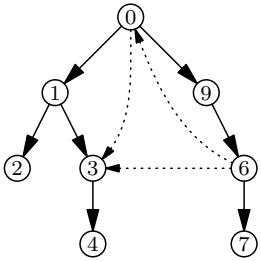
Každé uzávorkování odpovídá nějakému stromu. V našem případě je to tak zvaný *DFS strom,* který je tvořen hranami z právě uzavíraného vrcholu do jeho nově objevených následníků (těmi, po kterých jsme se rekurzivně volali). Je to tedy strom zakořeněný ve vrcholu v_0 a jeho hrany jsou orientované směrem od kořene. Budeme ho kreslit tak, že hrany vedoucí z každého vrcholu uspořádáme zleva doprava podle toho, jak je DFS postupně objevovalo.

Na průběh DFS se tedy také dá dívat jako na procházení DFS stromu. Vrcholy, které leží na cestě z kořene do aktuálního vrcholu, jsou přesně ty, které už jsme otevřeli, ale zatím nezavřeli. Rekurze je má na zásobníku a v závorkové reprezentaci odpovídají levým závorkám, jež jsme vypsali, ale dosud neuzavřeli pravými. Tyto vrcholy tvoří příslovečnou Ariadninu nit, po níž se vracíme směrem ke vchodu do bludiště.

Obrázek 1.6 ukazuje, jak vypadá průchod DFS stromem pro „prasátkový“ graf z obrázku 1.2. Začali jsme vrcholem 0 a pokaždé jsme probírali následníky od nejmenšího k největšímu.

Pozorování: Hranám grafu můžeme přiřadit typy podle toho, v jakém vztahu jsou odpovídající závorky, čili v jaké poloze je hrana vůči DFS stromu. Tomuto přiřazení se obvykle říká *DFS klasifikace hran.* Pro hranu xy rozlišíme tyto možnosti:

- $(_x \ldots (_y \ldots)_y \ldots)_x$. Tehdy mohou nastat dva případy:
  - Vrchol y byl při uzavírání x nově objeven. Taková hrana leží v DFS stromu, a proto jí říkáme *stromová.*
  - Vrchol y jsme už znali, takže v DFS stromu leží v nějakém podstromu pod vrcholem x. Těmto hranám říkáme *dopředné.*
- $(_y \ldots (_x \ldots)_x \ldots)_y$ – vrchol y leží na cestě ve stromu z kořene do x a je dosud otevřený. Takové hrany se nazývají *zpětné.*
- $(_y \ldots)_y \ldots (_x \ldots)_x$ – vrchol y byl už uzavřen a rekurze se z něj vrátila. Ve stromu není ani předkem, ani potomkem vrcholu x, nýbrž leží



$(_0(_1(_2)_2(_3(_4)_4)_3)_1(_9(_6(_7)_7)_6)_9)_0$

v	$in(v)$	$out(v)$
0	1	16
1	2	9
2	3	4
3	5	8
4	6	7
5	—	—
6	11	14
7	12	13
8	—	—
9	10	15

Obr. 1.6: Průběh DFS na prasátkovém grafu

v nějakém podstromu odpojujícím se doleva od cesty z kořene do x. Těmto hranám říkáme *příčné*.

- $(_x\ldots)_x\ldots(_y\ldots)_y$ – případ, kdy by vedla hrana z uzavřeného vrcholu x do vrcholu y, který bude otevřen teprve v budoucnosti, nemůže nastat. Před uzavřením x totiž prozkoumáme všechny hrany vedoucí z x a do případných nenalezených vrcholů se rovnou vydáme.

Na obrázku 1.6 jsou stromové hrany nakresleny plnými čarami a ostatní tečkovaně. Hrana $(6,0)$ je zpětná, $(0,3)$ dopředná a $(6,3)$ příčná.

V neorientovaných grafech se situace zjednoduší. Každou hranu potkáme dvakrát: buďto poprvé jako stromovou a podruhé jako zpětnou, nebo poprvé jako zpětnou a podruhé jako dopřednou. Příčné hrany se neobjeví (k nim opačné by totiž byly toho druhu, který neexistuje).

K rozpoznání typu hrany vždy stačí porovnat hodnoty *in* a *out* a případně stavy obou krajních vrcholů. Zvládneme to tedy v konstantním čase.

Shrňme, co jsme v tomto oddílu zjistili:

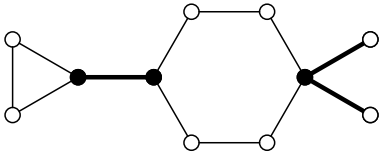
Věta: Prohledávání do hloubky doběhne v čase $\mathcal{O}(n+m)$ a spotřebuje prostor $\Theta(n+m)$. Jeho výsledkem je dosažitelnost z počátečního vrcholu, DFS strom a klasifikace všech hran.

Cvičení:

1. Jakou časovou složitost by DFS mělo, pokud bychom graf reprezentovali maticí sousednosti?
2. Nabízí se svůdná myšlenka, že DFS získáme z BFS nahrazením fronty zásobníkem. To by například znamenalo, že si můžeme ušetřit většinu analýzy algoritmu a jen se odkázat na obecný prohledávací algoritmus z cvičení 1.5.3. Na čem tento přístup selže?
3. Zkontrolujte, že DFS klasifikace je kompletní, tedy že jsme probrali všechny možné polohy hrany vzhledem ke stromu.

1.7. Mosty a artikulace

DFS klasifikaci hran lze elegantně použít pro hledání *mostů* a *artikulací* v souvislých neorientovaných grafech. Most se říká hraně, jejímž odstraněním se graf rozpadne na komponenty. Artikulace je vrchol s toutéž vlastností.



Obr. 1.7: Mosty a artikulace grafu

Mosty

Začneme klasickou charakteristikou mostů:

Lemma: Hrana *není most* právě tehdy, když leží na alespoň jedné kružnici.

Důkaz: Pokud hrana xy není most, musí po jejím odebrání stále existovat nějaká cesta mezi vrcholy x a y. Tato cesta spolu s hranou xy tvoří kružnici v původním grafu.

Naopak leží-li xy na nějaké kružnici C, nemůže se odebráním této hrany graf rozpadnout. V libovolném sledu, který používal hranu xy, totiž můžeme tuto hranu nahradíme zbytkem kružnice C. □

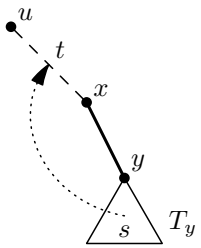
Nyní si rozmysleme, které typy hran podle DFS klasifikace mohou být mosty. Jelikož každou hranu potkáme v obou směrech, bude nás zajímat její typ při prvním setkání:

- Stromové hrany mohou, ale nemusí být mosty.
- Zpětné hrany nejsou mosty, protože spolu s cestou ze stromových hran uzavírají kružnici.
- Dopředné ani příčné hrany v neorientovaných grafech nepotkáme.

Stačí tedy umět rozhodnout, zda daná stromová hrana leží na kružnici. Jak by tato kružnice mohla vypadat? Nazveme x a y krajní vrcholy stromové hrany, přičemž x je vyšší z nich (bližší ke kořeni). Označme T_y podstrom DFS stromu tvořený vrcholem y a všemi jeho potomky. Pokud kružnice projde z x do y, právě vstoupila do podstromu T_y a než se vrátí do x, zase musí tento podstrom opustit. To ale může pouze po zpětné hraně: po jediné stromové jsme vešli a dopředné ani příčné neexistují.

Chceme tedy zjistit, zda existuje zpětná hrana vedoucí z podstromu T_y ven, to znamená na stromovou cestu mezi kořenem a x.

Pro konkrétní zpětnou hranu tuto podmínku ověříme snadno: Je-li st zpětná hrana a s leží v T_y, stačí otestovat, zda t je výše než y, nebo ekvivalentně zda $in(t) < in(y)$ – to je totéž, neboť in na každé stromové cestě shora dolů roste.



Obr. 1.8: Zpětná hrana st způsobuje, že xy není most

Abychom nemuseli pokaždé prohledávat všechny zpětné hrany z podstromu, provedeme jednoduchý předvýpočet: pro každý vrchol v spočítáme $low(v)$, což bude minimum z inů všech vrcholů, do nichž se lze dostat z T_v zpětnou hranou. Můžeme si představit, že to říká, jak vysoko lze z podstromu dosáhnout.

Pak už je testování mostů snadné: stromová hrana xy leží na kružnici právě tehdy, když $low(y) < in(y)$.

Předvýpočet hodnot $low(v)$ lze přitom snadno zabudovat do DFS: kdykoliv se z nějakého vrcholu v vracíme, spočítáme minimum z low jeho synů a z inů vrcholů, do nichž z v vedou zpětné hrany. Lépe je to vidět z následujícího zápisu algoritmu. DFS jsme mírně upravili, aby nepotřebovalo explicitně udržovat stavy vrcholů a vystačilo si s polem in.

Algoritmus MOSTY

Vstup: Souvislý neorientovaný graf $G = (V, E)$.

1. $M \leftarrow \emptyset$ *(seznam dosud nalezených mostů)*
2. $T \leftarrow 0$ *(počítadlo kroků)*
3. Pro všechny vrcholy v nastavíme $in(v) \leftarrow$ *nedefinováno*.
4. Zvolíme libovolně vrchol $u \in V$.
5. Zavoláme MOSTY2(u).

Výstup: Seznam mostů M.

Procedura MOSTY2(v)

1. $T \leftarrow T + 1$, $in(v) \leftarrow T$
2. $low(v) \leftarrow +\infty$
3. Pro všechny následníky w vrcholu v:
4. Pokud $in(w)$ není definován: *(hrana vw je stromová)*
5. Zavoláme MOSTY2(w).
6. Pokud $low(w) \geq in(w)$: *(vw je most)*
7. Přidáme hranu vw do seznamu M.
8. $low(v) \leftarrow \min(low(v), low(w))$
9. Jinak je-li $in(w) < in(v) - 1$: *(zpětná hrana)*
10. $low(v) \leftarrow \min(low(v), in(w))$

Snadno nahlédneme, že takto upravené DFS stále tráví konstantní čas nad každým vrcholem a hranou, takže běží v čase $\Theta(n+m)$. Paměti zabere $\Theta(n+m)$, neboť oproti DFS ukládá navíc pouze pole *low*.

Artikulace

I artikulace je možné charakterizovat pomocí kružnic a stromových/zpětných hran, jen je to maličko složitější:

Lemma A: Vrchol v *není artikulace,* pokud pro každé dva jeho různé sousedy x a y existuje kružnice, na níž leží hrany vx i vy.

Důkaz: Pokud v není artikulace, pak po odebrání vrcholu v (a tedy i hran vx a vy) musí mezi vrcholy x a y nadále existovat nějaká cesta. Doplněním hran xv a vy k této cestě dostaneme kýženou kružnici.

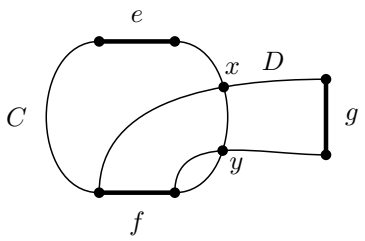
V opačném směru: Nechť každé dvě hrany incidentní s v leží na společné kružnici. Poté můžeme libovolnou cestu, která spojovala ostatní vrcholy a procházela při tom přes v, upravit na sled, který v nepoužije: pokud cesta do v vstoupila z nějakého vrcholu x a odchází do y, nahradíme hrany xv a vy opačným obloukem příslušné kružnice. Tím pádem graf zůstane po odebrání v souvislý a v není artikulace. □

Definice: Zavedeme binární relaci $\approx$ na hranách grafu tak, že hrany e a f jsou v relaci právě tehdy, když $e = f$ nebo e a f leží na společné kružnici.

Lemma E: Relace $\approx$ je ekvivalence.

Důkaz: Reflexivita a symetrie jsou zřejmé z definice, ale potřebujeme ověřit tranzitivitu. Chceme tedy dokázat, že kdykoliv $e \approx f$ a $f \approx g$, pak také $e \approx g$. Víme, že existuje společná kružnice C pro e, f a společná kružnice D pro f a g. Potřebujeme najít kružnici, na níž leží současně e a g. Sledujme obrázek 1.9.

Vydáme se po kružnici D jedním směrem od hrany g, až narazíme na kružnici C (stát se to musí, protože hrana f leží na C i D). Vrchol, kde jsme se zastavili, označme x. Podobně při cestě opačným směrem získáme vrchol y. Snadno ověříme, že vrcholy x a y musí být různé.



Obr. 1.9: Situace v důkazu lemmatu E

Hledaná společná kružnice bude vypadat takto: začneme hranou g, pak se vydáme po kružnici D do vrcholu x, z něj po C směrem od vrcholu y ke hraně e, projdeme touto hranou, pokračujeme po C do vrcholu y a pak po D zpět ke hraně g. □

Ekvivalenčním třídám relace $\approx$ se říká *komponenty vrcholové 2-souvislosti* nebo také *bloky.* Pozor na to, že na rozdíl od komponent obyčejné souvislosti to jsou množiny hran, nikoliv vrcholů.

Pozorování: Vrchol v je artikulace právě tehdy, sousedí-li s hranami z alespoň dvou různých bloků.

Teď povoláme na pomoc DFS klasifikaci. Uvažme všechny hrany incidentní s nějakým vrcholem v. Nejprve si všimneme, že každá zpětná hrana je v bloku s některou ze stromových hran, takže stačí zkoumat pouze stromové hrany.

Dále nahlédneme, že pokud jsou dvě stromové hrany vedoucí z v dolů v témže bloku, pak musí v tomto bloku být i stromová hrana z v nahoru. Důvod je nasnadě: podstromy visící pod stromovými hranami nemohou být propojeny přímo (příčné hrany neexistují), takže je musíme nejprve opustit zpětnou hranou a pak se vrátit přes otce vrcholu v.

Zbývá tedy pro každou stromovou hranu mezi v a jeho synem zjistit, zda leží na kružnici se stromovou hranou z v nahoru. K tomu opět použijeme hodnoty *low*: je-li s syn vrcholu v, stačí otestovat, zda $low(s) < in(v)$.

Přímočarým důsledkem je, že má-li kořen DFS stromu více synů, pak je artikulací – hrany do jeho synů nemohou být propojeny ani přímo, ani přes vyšší patra stromu.

Stačí nám proto v algoritmu na hledání mostů vyměnit podmínku porovnávající *low* s *in* a hned hledá artikulace. Časová i paměťová složitost zůstávají lineární s velikostí grafu. Detailní zápis algoritmu ponechme jako cvičení.

Cvičení:

1. Dokažte, že pokud je v grafu na alespoň třech vrcholech most, pak je tam také artikulace. Ukažte, že opačná implikace neplatí.
2. Zapište v pseudokódu nebo naprogramujte algoritmus na hledání artikulací.
3. Definujme relaci $\sim$ na vrcholech tak, že $x \sim y$ právě tehdy, leží-li x a y na nějaké společné kružnici. Dokažte, že tato relace je ekvivalence. Jejím ekvivalenčním třídám se říká *komponenty hranové 2-souvislosti,* jednotlivé třídy jsou navzájem pospojovány mosty. Upravte algoritmus na hledání mostů, aby graf rozložil na tyto komponenty.
4. Rozšiřte algoritmus na hledání artikulací, aby graf rozložil na bloky.

1.8. Acyklické orientované grafy

Častým případem orientovaných grafů jsou *acyklické orientované grafy* neboli *DAGy* (z anglického *directed acyclic graph*). Pro ně umíme řadu problémů vyřešit efektivněji než pro obecné grafy. Mnohdy k tomu využíváme existenci topologického pořadí vrcholů, které zavedeme v tomto oddílu.

Detekce cyklů

Nejprve malá rozcvička: Jak poznáme, jestli zadaný orientovaný graf je DAG? K tomu použijeme DFS, které budeme opakovaně spouštět, než prozkoumáme celý

graf (buď stejně, jako jsme to dělali při testování souvislosti v oddílu 1.4, nebo trikem z cvičení 1.8.1).

Lemma: V grafu existuje cyklus právě tehdy, najde-li DFS alespoň jednu zpětnou hranu.

Důkaz: Pakliže DFS najde nějakou zpětnou hranu xy, doplněním cesty po stromových hranách z y do x vznikne cyklus. Teď naopak dokážeme, že na každém cyklu leží alespoň jedna zpětná hrana.

Mějme nějaký cyklus a označme x jeho vrchol s nejnižším *out*em. Tím pádem na hraně vedoucí z x do následujícího vrcholu na cyklu roste out, což je podle klasifikace možné pouze na zpětné hraně. □

Topologické uspořádání

Důležitou vlastností DAGů je, že jejich vrcholy lze efektivně uspořádat tak, aby všechny hrany vedly po směru tohoto uspořádání. (Nabízí se představa nakreslení vrcholů na přímku tak, že hrany směřují výhradně zleva doprava.)

Definice: Lineární uspořádání $\prec$ na vrcholech grafu nazveme *topologickým uspořádáním vrcholů,* pokud pro každou hranu xy platí, že $x \prec y$.

Věta: Orientovaný graf má topologické uspořádání právě tehdy, je-li to DAG.

Důkaz: Existuje-li v grafu cyklus, brání v existenci topologického uspořádání: pro vrcholy na cyklu by totiž muselo platit $v_1 \prec v_2 \prec \ldots \prec v_k \prec v_1$.

Naopak v acyklickém grafu můžeme vždy topologické uspořádání sestrojit. K tomu se bude hodit následující pomocné tvrzení:

> **Lemma:** V každém neprázdném DAGu existuje *zdroj,* což je vrchol, do kterého nevede žádná hrana.
>
> *Důkaz:* Zvolíme libovolný vrchol v a půjdeme z něj proti směru hran, dokud nenarazíme na zdroj. Tento proces ovšem nemůže pokračovat do nekonečna, protože vrcholů je jen konečně mnoho a kdyby se nějaký zopakoval, našli jsme v DAGu cyklus. □

Pokud je náš DAG prázdný, topologické uspořádání je triviální. V opačném případě nalezneme zdroj, prohlásíme ho za první vrchol v uspořádání a odstraníme ho včetně všech hran, které z něj vedou. Tím jsme opět získali DAG a postup můžeme iterovat, dokud zbývají vrcholy. □

Důkaz věty nám rovnou dává algoritmus pro konstrukci topologického uspořádání, s trochou snahy lineární (cvičení 1.8.2). My si ovšem všimneme, že takové uspořádání lze přímo vykoukat z průběhu DFS:

Věta: Pořadí, v němž DFS opouští vrcholy, je opačné topologické.

Důkaz: Stačí dokázat, že pro každou hranu xy platí $out(x) > out(y)$. Z klasifikace hran víme, že je to pravda pro všechny typy hran kromě zpětných. Zpětné hrany se nicméně v DAGu nemohou vyskytovat. □

Stačí tedy do DFS doplnit, aby kdykoliv opouští vrchol, připojilo ho na začátek seznamu popisujícího uspořádání. Časová i paměťová složitost zůstávají lineární.

Topologická indukce

Ukažme alespoň jednu z mnoha aplikací topologického uspořádání. Dostaneme DAG a nějaký vrchol u a chceme spočítat pro všechny vrcholy, kolik do nich z u vede cest. Označme $c(v)$ hledaný počet cest z u do v.

Nechť $v_1, \ldots, v_n$ je topologické pořadí vrcholů a $u = v_k$ pro nějaké k. Tehdy $c(v_1) = c(v_2) = \ldots = c(v_{k-1}) = 0$, neboť do těchto vrcholů se z u nelze dostat. Také jistě platí $c(v_k) = 1$. Dále můžeme pokračovat indukcí:

Předpokládejme, že už známe $c(v_1)$ až $c(v_{\ell-1})$ a chceme zjistit $c(v_\ell)$. Jak vypadají cesty z u do v_ℓ? Musí se skládat z cesty z u do nějakého předchůdce w vrcholu v_ℓ, na níž je napojena hrana wv_ℓ. Všichni předchůdci ovšem leží v topologickém uspořádání před v_ℓ, takže pro ně známe počty cest z u. Hledané $c(v_\ell)$ je tedy součtem hodnot $c(w)$ přes všechny předchůdce w vrcholu v_ℓ.

Tento výpočet proběhne v čase $\Theta(n + m)$, neboť součty přes předchůdce dohromady projdou po každé hraně právě jednou.

Další aplikace topologické indukce naleznete v cvičeních.

Cvičení:

1. Opakované spouštění DFS můžeme nahradit následujícím trikem: přidáme nový vrchol a hrany z tohoto vrcholu do všech ostatních. DFS spuštěné z tohoto „superzdroje“ projde na jedno zavolání celý graf. Nahlédněte, že jsme tím zachovali acykličnost grafu, a všimněte si, že chod tohoto algoritmu je stejný jako chod opakovaného DFS.
2. Ukažte, jak konstrukci topologického uspořádání postupným otrháváním zdrojů provést v čase $\mathcal{O}(n + m)$.
3. Příklad topologické indukce z tohoto oddílu by šel vyřešit i jednoduchou úpravou DFS, která by hodnoty $c(v)$ počítala rovnou při opouštění vrcholů. Ukažte jak.
4. Vymyslete, jak pomocí topologické indukce najít v lineárním čase délku nejkratší cesty mezi vrcholy u a v v DAGu s ohodnocenými hranami. **Fix!**
5. Ukažte totéž pro nejdelší cestu, což je problém, který v obecných grafech zatím neumíme řešit v polynomiálním čase.
6. Jak spočítat, kolik mezi danými dvěma vrcholy obecného orientovaného grafu vede nejkratších cest?

1.9.* Silná souvislost a její komponenty

Nyní se zamyslíme nad tím, jak rozšířit pojem souvislosti na orientované grafy. Intuitivně můžeme souvislost vnímat dvojím způsobem: Buď tak, že graf nelze rozdělit na dvě části, mezi kterými nevedou žádné hrany. Anebo chtít, aby mezi každými dvěma vrcholy šlo přejít po cestě. Zatímco pro neorientované grafy tyto vlastnosti splývají, v orientovaných se liší, což vede na dvě různé definice souvislosti:

Fix: Odkaz na kapitolu o Dijkstrovi, až nějaká bude.

Definice: Orientovaný graf je *slabě souvislý,* pokud zrušením orientace hran dostaneme souvislý neorientovaný graf.

Definice: Orientovaný graf je *silně souvislý,* jestliže pro každé dva vrcholy x a y existuje orientovaná cesta jak z x do y, tak opačně.

Slabá souvislost je algoritmicky triviální. V tomto oddílu ukážeme, jak lze rychle ověřovat silnou souvislost. Nejprve pomocí vhodné ekvivalence zavedeme její komponenty.

Definice: Buď $\leftrightarrow$ binární relace na vrcholech grafu definovaná tak, že $x \leftrightarrow y$ právě tehdy, existuje-li orientovaná cesta jak z x do y, tak z y do x.

Snadno nahlédneme, že relace $\leftrightarrow$ je ekvivalence (cvičení 1.9.1). Třídám této ekvivalence se říká *komponenty silné souvislosti* (v tomto oddílu říkejme prostě *komponenty*). Graf je tedy silně souvislý, pokud má právě jednu komponentu, čili pokud $u \leftrightarrow v$ pro každé dva vrcholy u a v. Vzájemné vztahy komponent můžeme popsat opět grafem:

Definice: *Graf komponent* $\mathcal{C}(G)$ má za vrcholy komponenty grafu G, z komponenty C_i vede hrana do C_j právě tehdy, když v původním grafu G existuje hrana z nějakého vrcholu $u \in C_i$ do nějakého $v \in C_j$.

Na graf $\mathcal{C}(G)$ se také můžeme dívat tak, že vznikl z G kontrakcí každé komponenty do jednoho vrcholu a odstraněním násobných hran.

Lemma: Graf komponent $\mathcal{C}(G)$ každého grafu G je acyklický.

Důkaz: Sporem. Nechť $C_1, C_2, \ldots C_k$ tvoří cyklus v $\mathcal{C}(G)$. Podle definice grafu komponent musí existovat vrcholy $x_1, \ldots, x_k$ ($x_i \in C_i$) a $y_1, \ldots, y_k$ ($y_i \in C_{i+1}$, indexujeme modulo k) takové, že $x_i y_i$ jsou hranami grafu G.

Jelikož každá komponenta C_i je silně souvislá, existuje cesta z y_{i-1} do x_i v C_i. Slepením těchto cest s hranami $x_i y_i$ vznikne cyklus v grafu G tvaru

$$x_1, y_1, \text{cesta v } C_2, x_2, y_2, \text{cesta v } C_3, x_3, \ldots, x_k, y_k, \text{cesta v } C_1, x_1.$$

To je ovšem spor s tím, že vrcholy x_i leží v různých komponentách. □

Podle toho, co jsme o acyklických grafech zjistili v minulém oddílu, musí v $\mathcal{C}(G)$ existovat alespoň jeden zdroj (vrchol bez předchůdců) a stok (vrchol bez následníků). Proto vždy existují komponenty s následujícími vlastnostmi:

Definice: Komponenta je *zdrojová,* pokud do ní nevede žádná hrana, a *stoková,* pokud nevede žádná hrana z ní.

Představme si nyní, že jsme našli nějaký vrchol, který leží ve stokové komponentě. Spustíme-li z tohoto vrcholu DFS, navštíví právě celou tuto komponentu (ven se dostat nemůže, hrany vedou v protisměru).

Jak ale vrchol ze stokové komponenty najít? Se zdrojovou by to bylo snazší: prohledáme-li graf do hloubky (opakovaně, nedostalo-li se na všechny vrcholy), vrchol s maximálním $out(v)$ musí ležet ve zdrojové komponentě (rozmyslete si, proč). Pomůžeme si proto následujícím trikem:

Pozorování: Nechť G^T je graf, který vznikne z G otočením orientace všech hran. Potom G^T má tytéž komponenty jako G a platí $\mathcal{C}(G^T) = (\mathcal{C}(G))^T$. Mimo to se prohodily zdrojové komponenty se stokovými.

Nabízí se tedy spustit DFS na graf G^T, vybrat v něm vrchol s maximálním *out*em a spustit z něj DFS v grafu G. Tím najdeme jednu stokovou komponentu. Tu můžeme odstranit a postup opakovat.

Je ale zbytečné stokovou komponentu hledat pokaždé znovu. Ukážeme, že postačí procházet vrcholy v pořadí klesajících *out*ů v G^T. Ty vrcholy, které jsme do nějaké komponenty zařadili, budeme přeskakovat, z ostatních vrcholů budeme spouštět DFS v G a objevovat nové komponenty. Následující tvrzení zaručuje, že takto budeme komponenty procházet v opačném topologickém pořadí:

Lemma: Pokud v $\mathcal{C}(G)$ vede hrana z komponenty C_1 do C_2, pak

$$\max_{x \in C_1} out(x) > \max_{y \in C_2} out(y).$$

Důkaz: Nejprve rozeberme případ, kdy DFS vstoupí do C_1 dříve než do C_2. Začne tedy zkoumat C_1, během toho objeví hranu do C_2, po té projde, načež zpracuje celou komponentu C_2, než se opět vrátí do C_1. (Víme totiž, že z C_2 do C_1 nevede žádná orientovaná cesta, takže DFS může zpět přejít pouze návratem z rekurze.) V tomto případě tvrzení platí.

Nebo naopak vstoupí nejdříve do C_2. Odtamtud nemůže dojít po hranách do C_1, takže se nejprve vrátí z celé C_2, než do C_1 poprvé vstoupí. I tehdy tvrzení lemmatu platí. □

Nyní je vše připraveno a můžeme algoritmus zapsat:

Algoritmus KompSilnéSouvislosti
Vstup: Orientovaný graf G

1. Sestrojíme graf G^T s obrácenými hranami.
2. $Z \leftarrow$ *prázdný zásobník*
3. Pro všechny vrcholy v nastavíme $komp(v) \leftarrow$ *nedefinováno*.
4. Spouštíme DFS v G^T opakovaně, než prozkoumáme všechny vrcholy. Kdykoliv přitom opouštíme vrchol, vložíme ho do Z. Vrcholy v zásobníku jsou tedy setříděné podle $out(v)$.
5. Postupně odebíráme vrcholy ze zásobníku Z a pro každý vrchol v:
6. Pokud $komp(v) =$ *nedefinováno*:
7. Spustíme DFS(v) v G, přičemž vstupujeme pouze do vrcholů s nedefinovanou hodnotou $komp(\ldots)$ a tuto hodnotu přepisujeme na v.

Výstup: Pro každý vrchol v vrátíme identifikátor komponenty $komp(v)$.

Věta: Algoritmus KompSilnéSouvislosti rozloží zadaný graf na komponenty silné souvislosti v čase $\Theta(n+m)$ a prostoru $\Theta(n+m)$.

Důkaz: Korektnost algoritmu vyplývá z toho, jak jsme jej odvodili. Všechna volání DFS dohromady navštíví každý vrchol a hranu právě dvakrát, práce se zásobníkem trvá také lineárně dlouho. Paměť kromě reprezentace grafu potřebujeme na pomocná pole a zásobníky (Z a zásobník rekurze), což je celkem lineárně velké. □

Dodejme ještě, že tento algoritmus objevil v roce 1978 Sambasiva Rao Kosaraju a nezávisle na něm v roce 1981 Micha Sharir.

Cvičení:

1. Dokažte, že relace $\leftrightarrow$ z tohoto oddílu je opravdu ekvivalence.
2. Opakovanému spouštění DFS, dokud není celý graf prohledán, se dá i zde vyhnout přidáním „superzdroje“ jako v cvičení 1.8.1. Co se přitom stane s grafem komponent?
3. V orientovaném grafu jsou některé vrcholy obarvené zeleně. Jak zjistit, jestli existuje cyklus obsahující alespoň jeden zelený vrchol?

4.* O orientovaném grafu řekneme, že je *polosouvislý,* pokud mezi každými dvěma vrcholy vede orientovaná cesta alespoň jedním směrem. Navrhněte lineární algoritmus, který polosouvislost grafu rozhoduje.

1.10.* Silná souvislost podruhé: Tarjanův algoritmus

Předvedeme ještě jeden lineární algoritmus na hledání komponent silné souvislosti. Je založen na několika hlubokých pozorováních o vztahu komponent s DFS stromem, jejichž odvození je pracnější. Samotný algoritmus je pak jednodušší a nepotřebuje konstrukci obráceného grafu. Objevil ho Robert Endre Tarjan v roce 1972.

Stejně jako v minulém oddílu budeme používat relaci $\leftrightarrow$ a komponentám silné souvislosti budeme říkat prostě komponenty.

Lemma: Každá komponenta indukuje v DFS stromu slabě souvislý podgraf.

Důkaz: Nechť x a y jsou vrcholy ležící v téže komponentě C. Rozebereme jejich možné polohy v DFS stromu a pokaždé ukážeme, že (neorientovaná) cesta P spojující ve stromu x s y leží celá uvnitř C.

Nejprve uvažme případ, kdy je x „nad“ y, čili z x do y lze dojít po směru stromových hran. Nechť t je libovolný vrchol cesty P. Jistě jde dojít z x do t – stačí následovat cestu P. Ale také z t do x – můžeme dojít po cestě P do y, odkud se už do x dostaneme (x a y jsou přeci oba v C). Takže vrchol t musí také ležet v C.

Pokud y je nad x, postupujeme symetricky.

Zbývá případ, kdy x a y mají nějakého společného předka $p \neq x, y$. Kdyby tento předek ležel v C, máme vyhráno: p se totiž nachází nad x i nad y, takže podle předchozího argumentu leží v C i všechny ostatní vrcholy cesty P.

Pojďme dokázat, že p se nemůže nacházet mimo C. Pozastavíme DFS v okamžiku, kdy už se vrátilo z jednoho z vrcholů x a y (BÚNO x), stojí ve vrcholu p a právě se chystá odejít stromovou hranou směrem k y. Použijeme následující:

Pozorování: Kdykoliv v průběhu DFS vedou z uzavřených vrcholů hrany pouze do uzavřených a otevřených.

Důkaz: Přímo z klasifikace hran. □

Víme, že z x vede orientovaná cesta do y. Přitom x je už uzavřený a y dosud nenalezený. Podle pozorování tato cesta musí projít přes nějaký otevřený vrchol. Ten se ve stromu nutně nachází nad p (neostře), takže přes něj jde z x dojít do p. Ovšem z p lze dojít po stromových hranách do x, takže x a p leží v téže komponentě.

(Intuitivně: stromová cesta z kořene do p, na níž leží všechny otevřené vrcholy, tvoří přirozenou hranici mezi už uzavřenou částí grafu a dosud neprozkoumaným zbytkem. Cesta z x do y musí tuto hranici někde překročit.) □

Stačí tedy umět poznat, které stromové hrany leží na rozhraní komponent. K tomu se hodí „chytit“ každou komponentu za její nejvyšší vrchol:

Definice: *Kořenem komponenty* nazveme vrchol, v němž do ní DFS poprvé vstoupilo. Tedy ten, jehož *in* je nejmenší.

Pokud odstraníme hrany, za které „visí“ kořeny komponent, DFS strom se rozpadne na jednotlivé komponenty. Ukážeme, jak v okamžiku, kdy nějaký vrchol v opouštíme, poznat, zda je kořenem své komponenty. Označíme T_v podstrom DFS stromu obsahující v a všechny jeho potomky.

Lemma: Pokud z T_v vede zpětná hrana ven, není v kořenem komponenty.

Důkaz: Zpětná hrana vede z T_v do nějakého vrcholu p, který leží nad v a má menší *in* než v. Přitom z v se jde dostat do p přes zpětnou hranu a zároveň z p do v po stromové cestě, takže p i v leží v téže komponentě. □

S příčnými hranami je to složitější, protože mohou vést i do jiných komponent. Zařídíme tedy, aby v okamžiku, kdy opouštíme kořen komponenty, byly již ke komponentě přiřazeny všechny její vrcholy. Pak můžeme použít:

Lemma: Pokud z T_v vede příčná hrana ven do dosud neopuštěné komponenty, pak v není kořenem komponenty.

Důkaz: Vrchol w, který je cílem příčné hrany, má nižší *in* než v a už byl uzavřen. Jeho komponenta ale dosud nebyla opuštěna, takže jejím kořenem musí být některý z otevřených vrcholů. Vrchol v je s touto komponentou obousměrně propojen, tedy v ní také leží, ovšem níže než kořen. □

Lemma: Pokud nenastane situace podle předchozích dvou lemmat, v je kořenem komponenty.

Důkaz: Kdyby nebyl kořenem, musel by skutečný kořen ležet někde nad v (komponenta je přeci ve stromu souvislá). Z v by do tohoto kořene musela vést cesta, která by někudy musela opustit podstrom T_v. To ale lze pouze zpětnou nebo příčnou hranou. □

Nyní máme vše připraveno k formulaci algoritmu. Graf budeme procházet do hloubky. Vrcholy, které jsme dosud nezařadili do žádné komponenty, budeme ukládat do pomocného zásobníku. Kdykoliv při návratu z vrcholu zjistíme, že je kořenem

komponenty, odstraníme ze zásobníku všechny vrcholy, které leží v DFS stromu pod tímto kořenem, a zařadíme je do nové komponenty.

Pro rozhodování, zda z T_v vede zpětná nebo příčná hrana, budeme používat hodnoty $esc(v)$, které budou fungovat podobně jako $low(v)$ v algoritmu na hledání mostů.

Definice: $esc(v)$ udává minimum z *in*ů vrcholů, do nichž z podstromu T_v vede buď zpětná hrana, nebo příčná hrana do ještě neuzavřené komponenty.

Následuje zápis algoritmu. Pro zjednodušení implementace si vystačíme s polem *in* a neukládáme explicitně ani *out*, ani stav vrcholů. Při aktualizaci pole *esc* nebudeme rozlišovat mezi zpětnými, příčnými a dopřednými hranami – uvědomte si, že to nevadí.

Algoritmus KompSSTarjan
Vstup: Orientovaný graf G

1. Pro všechny vrcholy v nastavíme:
2. $\quad in(v) \leftarrow$ *nedefinováno*
3. $\quad komp(v) \leftarrow$ *nedefinováno*
4. $T \leftarrow 0$
5. $Z \leftarrow$ *prázdný zásobník*
6. Pro všechny vrcholy u:
7. $\quad$ Pokud $stav(u) =$ *nenalezený*:
8. $\quad\quad$ Zavoláme KSST(u).

Výstup: Pro každý vrchol v vrátíme identifikátor komponenty $komp(v)$.

Procedura KSST(v)

1. $T \leftarrow T + 1$, $in(v) \leftarrow T$
2. Do zásobníku Z přidáme vrchol v.
3. $esc(v) \leftarrow +\infty$
4. Pro všechny následníky w vrcholu v:
5. $\quad$ Pokud $in(w)$ není definován: *(hrana vw je stromová)*
6. $\quad\quad$ Zavoláme KSST(w).
7. $\quad\quad$ $esc(v) \leftarrow \min(esc(v), esc(w))$
8. $\quad$ Jinak: *(zpětná, příčná nebo dopředná hrana)*
9. $\quad\quad$ Není-li $komp(w)$ definovaná:
10. $\quad\quad\quad$ $esc(v) \leftarrow \min(esc(v), in(w))$
11. Je-li $esc(v) \geq in(v)$: *(v je kořen komponenty)*
12. $\quad$ Opakujeme:
13. $\quad\quad$ Odebereme vrchol ze zásobníku Z a označíme ho t.
14. $\quad\quad$ $komp(t) \leftarrow v$
15. $\quad\quad$ Cyklus ukončíme, pokud $t = v$.

Věta: Tarjanův algoritmus nalezne komponenty silné souvislosti v čase $\Theta(n + m)$ a prostoru $\Theta(n + m)$.

Důkaz: Správnost algoritmu plyne z jeho odvození. Časová složitost se od DFS liší pouze obsluhou zásobníku Z. Každý vrchol se do Z dostane právě jednou při svém otevření a pak ho jednou vyjmeme, takže celkem prací se zásobníkem strávíme čas $\mathcal{O}(n)$. Jediná paměť, kterou k DFS potřebujeme navíc, je na zásobník Z a pole *esc*, což je obojí velké $\mathcal{O}(n)$. □