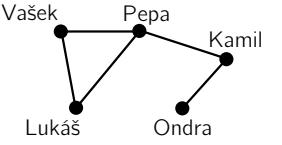
Grafy

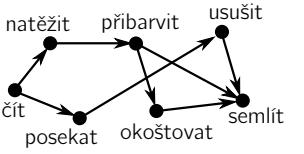
Co mají společného následující úlohy?

- Na filmovém festivalu se sešlo šest tisíc lidí, některé dvojice se znají, některé ne. Jak najít největší skupinu lidí, ve které se všichni znají?
- Podnikatel sepisuje procesy, které se pravidelně opakují při tvorbě jeho produktu. Některé úkony závisí na dokončení jiných, a tak by rád věděl, jak je uspořádat a jaké možnosti má při jejich paralelizování.
- Jak najít nejkratší cestu z Prahy do Brna, máme-li zadánu silniční síť České republiky jako trojice (město, město, délka)?

Obyčejný graf



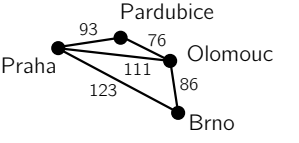
Orientovaný graf



Všechny problémy mají společné, že jejich vstup můžeme redukovat na dvě množiny: objekty a vztahy mezi dvojicemi těchto objektů. Objekty jsou po řadě lidé, úkony a města; vztahy jsou

- A se zná s B.
- A závisí na dokončení B.
- Mezi A a B existuje cesta dlouhá x.

Ohodnocený graf



Takovým zadáním říká moderní matematika *grafy.* Ten pojem nijak nesouvisí s grafy jakožto obrázky vývoje funkcí nebo vizualizacemi statistických dat. Graf je v jádru skutečně jen dvojice množin, které budeme v počítači často reprezentovat seznamy.

Na rozdíl od jiných částí nové matematiky jsou grafy názorné – krásně se kreslí. Ačkoliv se část teorie zabývá vlastnostmi takového kreslení a jedna z dalších kuchařek se věnuje grafům, které se dají dobře kreslit do roviny, nesmíme se nechat touto názorností zmást. Jako informatici si je kreslíme jen proto, že se nám o seznamech čísel špatně přemýšlí. To, jak jsme si je nakreslili, na seznamech čísel pranic nezmění.

Definování

Naše tři příklady zachycují tři obvyklé situace vztahů mezi objekty:

- Nejjednoduší případ nastává na filmovém festivalu. Jediné, co si o vztahu pamatujeme, je, zda-li existuje. Mezi dvěma lidmi navíc nemůže existovat víc vztahů. Grafům, které modelují takové situace, říkáme *obyčejné* a kreslíme je tak, že označené vrcholy spojujeme bezejmennými čarami.
- Situace podnikatele je složitější, protože vztahy mají směr. Ačkoliv v uvedené úloze existence dvou protisměrných vztahů mezi dvojicí vrcholů znamená neexistenci řešení, v obecnosti nám takové situace nevadí. Grafy tohoto typu označujeme jako *orientované* a místo čar kreslíme šipky.
- Případ hledání nejkratší cesty pracuje na nejobecnějším grafu (v jistém smyslu), grafu *ohodnoceném.* Ten si u každého vztahu pamatuje, zdali existuje a pokud

ano, jaké má ohodnocení, což může být libovolný údaj, zpravidla číslo. Při kreslení si toto ohodnocení připisujeme k čarám.

Nemá cenu dále zastírat obvyklou terminologii: objektům se říká *vrcholy* a vztahům *hrany*. Počátky teorie grafů se totiž vážou ke zkoumání pravidelných mnohostěnů.

Matematika definuje obyčejný graf jako uspořádanou dvojici (V, E), kde V je obecná (*nosná*) množina vrcholů (**v**ertices) a E množina neuspořádaných dvojic $\{u, v\}$ hran (**e**dges). V případě grafu orientovaného jsou dvojice z E uspořádané, (u, v). Zanést ohodnocení do definice můžeme prostě tím, že k (V, E) přidáme funkci $h : E \to M$, kde M je množina, ze které ohodnocení vybíráme. Ale jde to samozřejmě i jinak. Definice slouží jen jako kontrola, že nám intuice sloužila všem stejně.

Cvičení a poznámky

- Řešení všech třech úvodních úloh zde neuvádíme explicitně, ale v knize přítomná jsou. Zkuste je objevit.
- Můžeme ohodnotit graf orientovaný? Můžeme ohodnocovat vrcholy, ne hrany? Můžeme vést mezi dvěma vrcholy několik hran? Můžeme všechno! Existují dokonce i taková zobecnění, která jako hranu chápou množinu ne dvou, ale k vrcholů.

Programová reprezentace

Programátor si definuje obyčejný graf především podle možností svého programovacího jazyka. Zatímco součástí pythonistovy jazykové kultury jsou seznamy, díky kterým se nemusí starat o dynamické alokace paměti a může tak graf psát prostě jako seznam vrcholů, kde je vrchol seznamem sousedních vrcholů, pascalista takové řešení zpravidla nezvolí. Nechce totiž plýtvat pamětí statickou alokací, u níž musí předpokládat, že vrchol může mít tolik sousedů, kolik je v grafu vrcholů, ale ani se nechce párat se spojovými seznamy.

Pro mluvčí starších jazyků existuje standardní trik, jak neplýtvat pamětí a nemuset dynamicky alokovat. Uděláme si dvě pole, jedno bude velikosti N (tímto písmenkem se obvykle značí počet vrcholů), druhé velikosti M (počet hran). Do hranového pole budeme ukládat *cíle* hran, ve vrcholovém bude ke každému vrcholu uvedeno, kde v poli hran začínají hrany *z něj vycházející* – konec tohoto úseku je implicitně určen počátkem následujícího vrcholu. Pokud zrovna ukládáme graf neorientovaný, budeme každou hranu chápat jako dvojici hran orientovaných, protisměrných, což ostatně platí u každého způsobu uložení grafu *seznamem sousedů*, jak se probírané metodě říká.

V případě složitých grafových algoritmů, které za svého běhu graf extenzivně modifikují, se kvůli časové složitosti nevyhneme nutnosti použít spojový seznam pro seznamy hran vedoucích z jednotlivých vrcholů. Pokud si ke každé hraně zapamatujeme nejen, kam vede, ale i kde se nachází její *druhý konec* ve spojovém seznamu protějšího vrcholu, můžeme v konstantním čase hranu odebrat, což se nám v jednodušších reprezentacích bude dařit jen těžko.

V pokročilejších partiích teorie grafů se používají rozličné maticové reprezentace a máte-li rádi algebru, je doporučeníhodné se s nimi seznámit.

V následujících receptech budeme, vzhledem k použitému programovacímu jazyku, vždy používat seznamy sousedů uložené výše popsaným, trikovým způsobem. Hranovému poli budeme říkat `Sousedi`, poli vrcholovému `Zacatky` a nadeklarujeme si je takto:

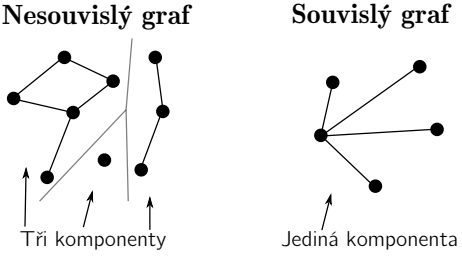
```
var N, M: integer;  { počet vrcholů a hran }
    Zacatky: array[1..MaxN+1] of integer;
    Sousedi: array[1..MaxM] of integer;
```

Souvislost

Grafová terminologie je bohatá a vtipná. Třeba *cesta* (délky $k-1$) je taková posloupnost různých vrcholů $u_1, u_2, \ldots, u_k$, kde jsou všechny těsně po sobě jdoucí vrcholy u_i, u_{i+1} spojeny hranou. Tedy vskutku cesta v netechnickém významu toho slova, pokud graf chápeme jako mapu, po které se můžeme procházet.

Nabízí se přemýšlet nad tím, kam až v takovém grafu/mapě můžeme z některého vrcholu po všech možných cestách dojít. Z Prahy se pěšky (suchou nohou) do Sydney nedostaneme, do Bratislavy ano.

Množině vrcholů grafu, pro které platí, že se z libovolného jejího vrcholu dostaneme po nějaké cestě do libovolného jiného vrcholu množiny, a která je maximální v tom smyslu, že už do ni nemůžeme přidat žádný jiný vrchol grafu, říkáme *komponenta souvislosti* grafu. Vrcholy každého grafu můžeme na komponenty beze zbytku rozdělit. Pokud má graf komponentu pouze jednu (z odkudkoliv se dostaneme kamkoliv), budeme jej označovat za graf *souvislý.*



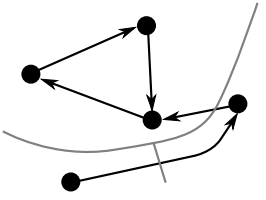
Uvědomte si, že fakt, že lze z každého vrcholu dojít do každého jiného *po cestě,* neznamená, že mezi každými dvěma vrcholy existuje hrana (cesty délky jedna). Kdyby mezi každými dvěma vrcholy vedla hrana, prohlásili bychom graf za *úplný* a značili ho K_n.

Jak najít komponenty souvislosti programově? Těžko vymyslet snazší úkol! Označíme si vrcholy příznakem *nenavštíveno* a pro libovolný nenavštívený vrchol spustíme rekurzivní funkci prohledávání, která nedělá nic jiného, než že odejme příznak nenavštívenosti vrcholu v argumentu a zavolá sama sebe na všechny sousedy, kteří příznak ještě mají.

Takový postup v čase lineárním vůči počtu hran komponenty objeví komponentu, ve které se nacházel vybraný vrchol, a můžeme ho pouštět znovu a znovu, dokud do komponent nerozbijeme celý graf. Systematicky se tímto prohledáním budeme zabývat v části *Prohledávání do hloubky.*

Souvislost orientovaného grafu je o dost složitější vlastnost. Existuje ve dvou variantách: definici *slabé souvislosti* získáme tak, že ze zkoumaného grafu „odmažeme šipky“ a na výsledném neorientovaném grafu identifikujeme komponenty souvislosti. *Silná souvislost* se definuje takřka stejně jako souvislost na obyčejném grafu, avšak využívá místo obyčejné cesty definici cesty orientované – je to to samé, akorát pořadí vrcholů musí respektovat orientaci šipek.

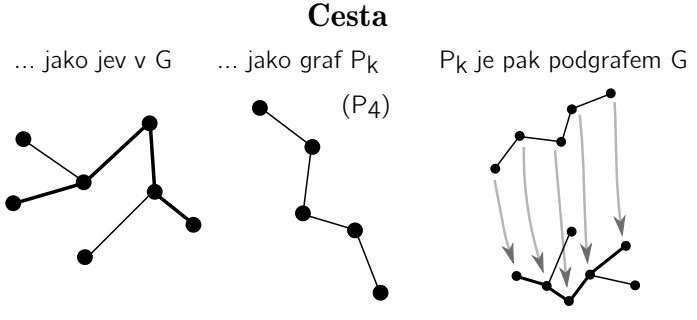
Silně souvislý graf je tedy i slabě souvislý, protože lze-li se z vrcholu do vrcholu dostat respektujíce směry šipek, jde to jistě, i když na ně zapomeneme. Naopak to ale neplatí, jak ukazuje zobrazený slabě souvislý graf s vyznačenými komponentami silné souvislosti.



Slabě souvislý graf se třemi komponentami silné souvislosti.

Poznámky

- Vedle *cesty* existují ještě dva podobné pojmy související s procházkami – jde o *tah* a *sled*. Zatímco v cestě se nemohou opakovat vrcholy, a tedy ani hrany (to si rozmyslete), v tahu se nesmí opakovat pouze hrany a sled je obecná procházka po grafu, která se může sestávat z poskakování mezi dvěma vrcholy.
- Vypadá to, jako by pojem „cesta“ měl dva významy: je to jednak jev, který nastává v obecném grafu (tak jsme si jej definovali), druhak je cesta délky k graf sám o sobě označovaný jako P_k (z anglického **p**ath).



Souvislost obou použití je v tom, že můžeme P_k prohlásit za *podgraf* libovolného grafu, v němž nastává v úvodu kapitoly popisovaná situace (existuje v něm cesta délky k). Graf G_1 je přitom podgrafem grafu G_2 tehdy, existuje-li v G_2 množina vrcholů V taková, že můžeme sestrojit vzájemně jednoznačné přiřazení mezi vrcholy V a všemi vrcholy G_1 takové, že kdykoliv mezi dvěma vrcholy v G_1 vede hrana, existuje hrana i mezi příslušnými vrcholy v G_2.

Totožná situace nastává u kružnic, o kterých mluví další část. Obecně ale tímto způsobem zaměňovat výskyt v grafu a graf sám nezní dostatečně odborně – vyskytne-li se kupříkladu úplný graf K_n v grafu H, začne se mu říkat *klika.*

Kružnice a stromy

Kružnice je, podobně jako cesta, posloupnost různých vrcholů $u_1, u_2, \ldots, u_k$, kde platí, že mezi u_i a u_{i+1} vede hrana. Navíc ale musí vést hrana i mezi u_k a u_1.

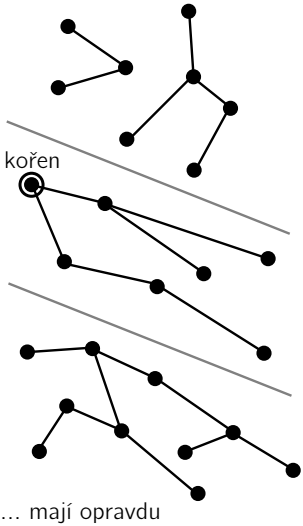
Jsou-li dva vrcholy v obyčejném grafu na kružnici, znamená to, že mezi nimi existují alespoň dvě cesty – jedna po prvním oblouku kružnice, druhá po druhém. Považujeme-li hrany za spojení (dopravní, komunikační), je to dobrá zpráva, protože výpadek jedné z hran kružnice nezpůsobí nedostupnost (graf zůstane souvislý – viz kapitolu o hranové 2-souvislosti).

Neexistence kružnice má pro strukturu grafu silné důsledky. Platí, že právě pokud v souvislém grafu není kružnice, pak

- mezi každými dvěma vrcholy vede právě jedna cesta,
- odebrání libovolné hrany způsobí ztrátu souvislosti,
- přidání libovolné hrany vytvoří kružnici,
- vrcholů je v takovém grafu právě o jeden víc než hran.

Těmto souvislým grafům bez kružnic se říká *stromy* a je zábavným cvičením si dokázat ekvivalenci mezi čtyřmi uvedenými body.

Stromy



… mají opravdu jednoduchou strukturu.

Stromy jsou velmi oblíbené datové struktury – jen mezi kuchařkami najdete dvě ne náhodou pojmenované „vyhledávací stromy“ a „intervalové stromy“. Protože je užitečné mít ve stromu zvláštní vrchol, ze kterého budeme strom prohledávat a skrze který na něj budeme v programu odkazovat, dostalo se mu zvláštního pojmenování – kořen.

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: Předpokládáme, že existuje vrchol x, který není označen, ale do kterého vede cesta z w. Pokud je takových vrcholů více, vezmeme si ten nejbližší k w. Označme si y předchůdce vrcholu x na nejkratší cestě z w; y je určitě označený (jinak by x nebyl nejbližší neoznačený). Vrchol y se tedy musel někdy objevit na zásobníku, tím pádem jsme ho také museli ze zásobníku odebrat a v kroku 3 označit všechny jeho sousedy, tedy i vrchol x, což je ovšem spor.

To, že algoritmus někdy skončí, nahlédneme snadno: v kroku 3 na zásobník přidáváme pouze vrcholy, které dosud nejsou označeny, a hned je značíme. Proto se každý vrchol může na zásobníku objevit nejvýše jednou, a jelikož 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:

```
var Oznacen: array[1..MaxN] of boolean;

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

Rozdělit neorientovaný graf na komponenty souvislosti je pak už jednoduché. Projdeme postupně všechny vrcholy grafu, a pokud nejsou v žádné z dosud označených komponent grafu, přidáme novou komponentu tak, že graf z tohoto vrcholu prohledáme do hloubky. Vrcholy značíme přímo číslem komponenty, do které patří. Protože prohledáváme do hloubky několik oddělených částí grafu, každou se složitostí $\mathcal{O}(N_i + M_i)$, 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 $\mathcal{O}(N + M)$.

```
var Komponenta: array[1..MaxN] of integer;
    NovaKomponenta: integer;

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

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

Průběh prohledávání grafu do hloubky můžeme znázornit stromem (podle anglického názvu pro prohledávání do hloubky „Depth-First Search“ se mu říká *DFS strom*). 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, anebo 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.

Pro orientované grafy je situace opět trochu složitější: stromové a dopředné hrany jsou orientované vždy ve stromě 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).

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 jeho $H[v] = -1$, 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.

Neexistenci kratší cesty 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.

```
var Fronta, H: array[1..MaxN] of integer;
    I, V, Prvni, Posledni: integer;
    PocatecniVrchol: integer;
begin
  ...
  for I := 1 to N do H[I] := -1;
  Prvni := 1;
  Posledni := 1;
  Fronta[Prvni] := PocatecniVrchol;
```

```
  H[PocatecniVrchol] := 0;
  repeat
    V := Fronta[Prvni];
    for I := Zacatky[V] to Zacatky[V+1]-1 do
      if H[Sousedi[I]] < 0 then begin
        H[Sousedi[I]] := H[V]+1;
        Posledni := Posledni + 1;
        Fronta[Posledni] := Sousedi[I];
      end;
    Prvni := Prvni + 1;
  until Prvni > Posledni;  { Fronta je prázdná }
  ...
end.
```

Prohledávání do šířky lze také použít na hledání komponent souvislosti a hledání kostry grafu (viz speciální kuchařka).

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 menším číslem do vrcholu s větším 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 zleva doprava.

Nejprve si ukážeme, že pro žádný orientovaný graf, který obsahuje cyklus, nelze takovéto topologické pořadí vytvořit. Označme vrcholy cyklu $v_1, \ldots, v_k$ tak, že hrana vede z vrcholu v_{i-1} do vrcholu v_i, resp. z v_k do v_1. Pak vrchol v_2 musí dostat vyšší číslo než vrchol v_1, v_3 než $v_2, \ldots, v_k$ než v_{k-1}. Ale vrchol v_1 musí mít zároveň vyšší číslo než v_k, 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 $p = N$.
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 snížíme 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 větší 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 a 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 vyšší číslo, zpětná existovat nemůže (v grafu by byl cyklus) a příčné hrany vedou pouze zprava doleva, takže také do již očíslovaných vrcholů. Časová složitost je opět $\mathcal{O}(N + M)$.

```
var Ocislovani: array[1..MaxN] of integer;
    Posledni: integer;
    I: integer;

procedure Projdi(V: integer);
var I: integer;
begin
  Ocislovani[V] := 0; { zatím V jen označíme }
  for I := Zacatky[V] to Zacatky[V+1]-1 do
    if Ocislovani[Sousedi[I]] = -1 then
      Projdi(Sousedi[I]);
  Ocislovani[V] := Posledni;
  Posledni := Posledni - 1;
end;

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

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:

```
var Hladina, Spojeno: array[1..MaxN] of integer;
    DvojSouvisle: Boolean;
    I: integer;

procedure Projdi(V, NovaHladina: integer);
var I, W: integer;
begin
  Hladina[V] := NovaHladina;
  Spojeno[V] := Hladina[V];

  for I := Zacatky[V] to Zacatky[V+1]-1 do begin
    W := Sousedi[I];
    if Hladina[W] = -1 then
    begin { stromová hrana }
      Projdi(W, NovaHladina + 1);
      if Spojeno[W] < Spojeno[V] then
```

```
        Spojeno[V] := Spojeno[W];
      if Spojeno[W] > Hladina[V] then
        DvojSouvisle := False; { máme most }
    end else { zpětná nebo dopředná hrana }
    if (Hladina[W] < NovaHladina-1) and
       (Hladina[W] < Spojeno[V]) then
      Spojeno[V] := Hladina[W];
  end;
end;

begin
  ...
  for I := 1 to N do
    Hladina[I] := -1;
  DvojSouvisle := True;
  Projdi(1, 0);
  ...
end.
```

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.

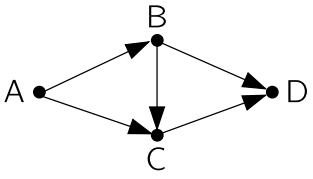
Algoritmus pro zjištění vrcholové 2-souvislosti grafu je velmi podobný algoritmu na zjišťování hranové 2-souvislosti, jen si musíme uvědomit, že odebíráme celý vrchol. Ze stromu procházení do hloubky může odebráním vrcholu vzniknout až několik podstromů, které všechny musí být spojeny zpětnou hranou 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.

Martin Mareš, David Matoušek, Petr Škoda a Lukáš Lánský

Úloha 20-3-4: Orientace na mapě

Na vstupu váš program dostane popis orientovaného grafu znázorňujícího mapu. Víte, že tento graf neobsahuje žádný orientovaný cyklus, čili že neexistuje žádná orientovaná cesta délky alespoň 1, která by začínala a končila ve stejném vrcholu. Úkolem programu je vypsat dvojici vrcholů, mezi kterými vede nejvíce různých cest v celém grafu. Za různé jsou považovány libovolné dvě cesty, které se liší alespoň jednou hranou. Pokud je takových dvojic vrcholů více, stačí libovolná z nich.

Příklad: Pro tento graf je řešením dvojice vrcholů A a D:



Úloha 18-2-5: Krokoběh

Krokoběh se skládá z několika jezírek, ve kterých mohou krokodýli odpočívat, a kanálů mezi nimi. Kanály jsou obousměrné, vedou vždy mezi dvěma jezírky a žádné dva kanály se mimo jezírka neprotínají (mimoúrovňově ale mohou).

Nějaká jezírka a kanály jsou již postaveny. Pokud se ovšem stane, že se krokodýl nemůže dostat do nějakého jezírka (nevede k němu žádná cesta), je velmi nerudný a žere vše kolem. (*Kvák!*) Protože Potrhlík nechce dopadnout stejně jako Skrblikvák, chtěl by dostavět potřebné kanály tak, aby byli krokodýli spokojení. Ti budou spokojení, pokud i když jeden libovolný kanál vyschne, pořád se budou moci dostat z každého jezírka do kteréhokoliv jiného. A protože stavba krokoběhu Potrhlíka finančně velmi vyčerpala, chtěl by postavit nových kanálů co nejméně.

Váš program dostane na vstupu popis existujícího krokoběhu. Ten se skládá z $N > 2$ jezírek a M kanálů, každý kanál spojuje dvojici jezírek. Vaším cílem je zjistit, kolik nejméně kanálů je třeba přidat, aby i když libovolný jeden kanál vyschne, bylo pořád možné dostat se z každého jezírka do každého.

Příklad: Pro $N = 6$ jezírek a $M = 4$ kanály vedoucí mezi jezírky $(1, 2)$, $(2, 3)$, $(3, 1)$ a $(4, 5)$ je třeba postavit alespoň další 3 kanály. (Jsou to například $(1, 4)$, $(5, 6)$, $(6, 2)$.)