Algoritmy 2 - grafové algoritmy

Petr Osička

KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

Reprezentace grafu v počítači

Budeme předpoládat, že $V = \{0, 1, \ldots, n-1\}$

Seznamy sousedů

- pole `adj` mající n prvků. Prvek `adj[u]` je seznam vrcholů, obsahující právě vrcholy z množiny $\{v \mid (u, v) \in V\}$.
- Suma délek všech seznamů `adj[u]` (sčítáme přes $u \in V$) je omezena $O(|H|)$.
- Další informace ke hranám (např. váhy), lze do seznamů přidat. Informace ke hraně (u, v) přidáme k vrcholu v do seznamu `adj[u]`.
- Tato reprezentace se hodí pro řídké grafy (kde je $|H|$ mnohem menší než n^2, protože její velikost závisí na počtu hran). Jak uvidíme později, reprezentace maticí sousednosti má velikost závislou na n^2.
- Nevýhodou je to, že nemůžeme v konstantním čase testovat, jestli graf obsahuje konkrétní hranu.

Matice sousednosti

- matice `A` velikosti $n \times n$ (budeme ji brát jako dvourozměrné pole).
- `A[u][v]` $=$ `1` pokud $(u, v) \in H$, jinak `A[u][v]` $=$ `0`
- V konstantním čase lze testovat existenci hrany.
- Zabírá $\Theta(n^2)$ paměti.
- Matice neorientovaného grafu je symetrická přes hlavní diagonálu.
- Přídavné informace ke hraně (u, v) lze přidat přímo k prvku `A[u][v]`.

V následujícím uvažujeme reprezentaci pomocí seznamů sousedů, zachycenou ve struktuře

```
struct Graph
    n — počet vrcholů
    adj — seznam sousedů
```

Průchod grafem obecně

Uvažujeme datovou strukturu `M` pro uchování množiny vrcholů. Operace:

- `insert(M, x)`. Vloží vrchol `x` do `M`.
- `pop(M)`. Odebere jeden vrchol a vrátí jej jako výsledek. (*Nespecifikujeme který vrchol, v tom je obecnost z nadpisu slajdu.*)
- `empty(M)`. Test prázdnosti `M`.

```
graph-search
 ⟵ g: Graph  — reprezentace pomocí matice sousednosti
 ⟵ s: Int  — počáteční vrchol
```

```
1  X ← [g.n]Bool — prvky inicializovány na false
2  vytvoříme prázdnou strukturu M
3  insert(M,s)
4  X[s] ← true
5  while (not empty(M))
6     u ← pop(M)
7     visit(u)
8     foreach w in g.adj[u] such that X[w] = false
9        X[w] ← true
10       insert(M,w)
```

Vlastnosti

Vrchol v je dosažitelný z s, pokud existuje cesta z s do v.

Věta

Volání `graph-search(G,s)` navštíví každý vrchol dosažitelný z vrcholu s právě jednou. Vrcholy, které dosažitelné nejsou, toto volání nenavštíví.

Důkaz.

Vrchol je navštíven poté, co je odebrán z `M`. Vrchol je vložen do `M` pouze pokud je jeho příznak v poli `X` nastaven na `false`. Protože po vložení vrcholu do `M` nastavíme tento příznak na `true`, je vrchol vložen do `M` nejvýše jednou, a tedy navštíven nejvýše jednou.

Každý vrchol, který je vložen do `M` je eventuálně navštíven: Z předchozího víme, že každý vrchol je vložen nejvýše jednou, počet vrcholů je konečný a v každé iteraci cyklu `while` jeden vrchol z `M` odebíráme. Pokud se tedy vrchol v během běhu algoritmu ocitne v `M`, ocitnou se tam i jeho potomci. Buď jsou přidáni před iterací cyklu `while`, ve které je v navštíveným vrcholem, nebo jsou přidání v cyklu `foreach`. Vidíme tedy, že s každým navštíveným vrcholem jsou navštíveni i všechny vrcholy, které jsou z něj dosažitelné. Současně z předchozího plyne, že vrchol, který není dosažitelný z s, se nemůže dostat do `M` a tím pádem nebude navštíven. □

Předpokládejme, že složitost operace `insert` je dána funkcí f_i, složitost operace `pop` funkcí f_p a složitost `empty` funkcí f_e.

V nejhorším případě je tedy složitost jednotlivých volání procedur omezena aplikací těchto funkcí na počet vrcholů grafu, protože v `M` může být každý vrchol nejvýše jednou.

Věta

Složitost `graph-search` pro graf $G = (V, E)$ je dána

$$O(|V| \cdot (f_i(|V|) + f_p(|V|) + f_e(|V|)) + |E|).$$

Důkaz.
Každý vrchol je vložen či odebrán nejvýše jednou, cyklus `while` proběhne nejvýše $|V|$-krát (každý vrchol je do `M` vložen nejvýše jednou, viz důkaz předchozího tvrzení). Počet iterací cyklu `foreach` je dán součtem délek seznamů v poli `g.adj`, pro každý vrchol totiž procházíme seznam jeho sousedů nejvýše jednou. Tento součet je $O(|E|)$. □

Průchod sestavuje strom

```
graph-search
⟵ g: Graph — reprezentace pomocí matice sousednosti
⟵ s: Int — počáteční vrchol
```

```
1  X ← [g.n]Bool — prvky inicializovány na false
2  P ← [g.n]Int — prvky inicializovány na -1
3  vytvoříme prázdnou strukturu M
4  insert(M,s)
5  X[s] ← true
6  while (not empty(M))
7     u ← pop(M)
8     visit(u)
9     foreach w in g.adj[u] such that X[w] = false
10       X[w] ← true
11       P[w] ← u
12       insert(M,w)
```

Uvažujme běh `graph-search` pro graf G a vrchol s, obsah `P` na jeho konci a graf $T = \{V_T, E_T\}$, kde

$$V_T = \{v \mid v \text{ je dosažitelný z } s\},$$
$$E_T = \{(u, v) \mid \texttt{P[v] = u}\}.$$

Věta

a. T je podgraf G.

b. Považujeme-li E_T za neorientované hrany, pak je T strom (vrchol s bereme za jeho kořen).

Důkaz.
(a) Z kódu algoritmu vidíme, že každý vrchol i hrana (modulo orientace), které jsou v T se museli nacházet i v G. U vrcholů je to zřejmé, u hran si stačí všimnout, že na řádku 11 musí `w` a `u` sousedit.
(b) *Absence cyklu.* Vrchol v se nemůže stát potomkem vrcholu v', který je navštíven později než v, protože v ten moment má již v nastaven příznak v poli `X`. Nelze tedy sestavit kružnici.
Souvislost. Indukcí snadno dokážeme, že v momentě, kdy je na řádku 12 vložen vrchol `w` do `M`, existuje v (tomuto kroku odpovídající části) T cesta z `s` do `w`. Platí to totiž před první iterací cyklu `while`, a cyklus tento invariant zachovává. Nastavujeme-li totiž na řádku 11 `u` jako rodiče `w`, jsou `u` a `w` sousedi. □

Používané průchody

Různé volby `M` dají průchody s dalšími vlastnosti (to pak určuje jejich použití). V přednášce projdeme

- *Průchod do šířky*, kde je `M` frontou,
- *Průchod do hloubky*, kde je `M` zásobníkem.

Za účelem analýzy a pro využití v aplikacích, algoritmy rozšíříme (analogicky rozšíření pomocí pole `P` výše).

`bfs` — *průchod do šířky*
⟵ `g: Graph` — *reprezentace pomocí matice sousednosti*
⟵ `s: Int` — *počáteční vrchol*

```
1  X ← [g.n]Bool — prvky inicializovány na false
2  P ← [g.n]Int — prvky inicializovány na 0
3  D ← [g.n]Int — prvky inicializovány na ω, ω > m pro všechna reálná m
4  vytvoříme prázdnou frontu Q
5  enqueue(Q,s)
6  X[s] ← true
7  D[s] ← 0
8  while (not empty(Q))
9     u ← dequeue(Q)
10    visit(u)
11    foreach w in g.adj[u] such that X[w] = false
12       X[w] ← true
13       P[w] ← u
14       D[w] ← D[u] + 1
15       insert(M,w)
```

Nejkratší vzdálenost $\delta(u, v)$ z vrcholu u do vrcholu v je nejmenší počet hran, které má nějaká cesta z u do v. Pokud cesta z u do v neexistuje, pak $\delta(u, v) = \omega$. Cesta z u do v, která má $\delta(u, v)$ hran je *nejkratší cesta* z u do v.

Dokážeme následující větu

Věta (`bfs` hledá nejkratší cesty)

Na konci běhu `bfs(G,s)` pro každý vrchol `v` platí

1. `D[v]` $= \delta($`s`$,$ `v`$)$.
2. Existuje-li cesta z `s` do `v`, je cesta, kterou sestavíme tak, že vezmeme nejkratší cestu z `s` do `P[v]` a připojíme k ní hranu do vrcholu `v`, nejkratší cestou z `s` do `v`.

Lemma

Nechť $G = (V, H)$ *je graf a* s *je jeho vrchol. Potom pro každou hranu* $(u, v) \in H$ *platí,* $\delta(s, v) \leqslant \delta(s, u) + 1$.

Důkaz. Pokud existuje cesta z s do u, pak existuje i cesta z s do v, kterou dostaneme prodloužením nejkratší cesty z s do u o hranu (u, v). Cestu tak prodloužíme o 1 hranu. Pokud cesta z s do v neexistuje, pak neexistuje ani cesta z s do u. V obou případech dokazovaná nerovnost platí. □

Lemma

Po skončení běhu `bfs(G,s)` *platí pro každý vrchol* `v` *nerovnost* $\texttt{D[v]} \geqslant \delta(\texttt{s}, \texttt{v})$.

Důkaz. Indukcí přes počet provedení `enqueue`. Víme, že že libovolný vrchol `x` je do fronty vložen maximálně jednou a tedy `D[x]` je změněno maximální jednou.

Při prvním provedení operace tvrzení platí, protože vkládáme vrchol `s`, kterému předtím nastavíme $\texttt{D[s]} \leftarrow \texttt{0}$.

V obecném kroku předpokládejme, že do fronty vkládáme vrchol `w`, který jsme našli při procházení seznamu sousedů vrcholu `u`. Vrchol `u` tak musel být vložen do fronty před `w` a platí pro něj indukční předpoklad, tedy $\texttt{D[u]} \geqslant \delta(\texttt{s}, \texttt{u})$. Na řádku 12 nastavíme $\texttt{D[w]} \leftarrow \texttt{D[u] + 1}$. S použitím predchozího lemma tak máme

$$\texttt{D[w]} = \texttt{D[u]} + 1 \;\geqslant\; \delta(\texttt{s}, \texttt{u}) + 1 \;\geqslant\; \delta(\texttt{s}, \texttt{w}).$$

Graf totiž musí obsahovat hranu `(u,w)`, protože `w` je obsažen v `G.adj[u]`. □

Lemma

Je-li během běhu `bfs` *obsah fronty* `Q` *uspořádaný od nejdříve vloženého vrcholu roven* $v_1, v_2, \ldots v_r$, *pak*

- $D[v_r] \leqslant D[v_1] + 1$,
- *pro* $i = 1, 2, \ldots, r-1$ *máme* $D[v_i] \leqslant D[v_{i+1}]$

Důkaz. Indukcí podle počtu operací s frontou. Po první operaci tvrzení očividně platí, protože obsahuje pouze jeden prvek.

Pokud v obecném případě provedeme odebrání vrcholu z fronty, fronta nově začíná od vrcholu v_2 a tvrzení očividně platí.

Uvažme první operaci vložení vrcholu (označíme jej v_{r+1}) v rámci cyklu na řádcích 8 až 15. Předchozí operace musela být odebrání vrcholu (před kterým tvrzení podle indukčního předpokladu tvrzení platilo). Označme odebraný vrchol u. Máme tak $D[u] \leqslant D[v_1]$ a $D[v_r] \leqslant D[u] + 1$. Díky řádku 14 máme

$$D[v_r] \leqslant (D[v_{r+1}] = D[u] + 1) \leqslant D[v_1] + 1.$$

a tvrzení tedy platí. Další operace přidání ve stejné iteraci vyřešíme analogicky. □

Věta (`bfs` hledá nejkratší cesty)

Na konci běhu `bfs(G,s)` pro každý vrchol v platí

1. `D[v]` $= \delta(s, v)$.
2. Existuje-li cesta z s do v, je cesta, kterou sestavíme tak, že vezmeme nejkratší cestu z s do `P[v]` a připojíme k ní hranu vrchol v, nejkratší cestou z s do v.

Důkaz (1) Sporem. Předpokládejme, že existuje vrchol v, pro který tvrzení neplatí a má mezi vrcholy, pro které tvrzení neplatí, minimální délku nejkratší cesty z vrcholu s. (Očividně, $v \neq s$.) Z předchozích lemmat máme `D[`v`]` $\geqslant \delta(s, v)$, a tedy musí být

$$\texttt{D[}v\texttt{]} > \delta(s, v).$$

Platí tak $\delta(s, v) \neq \omega$ a existuje cesta z s do v. Vybereme vrchol u, který leží na nejkratší cestě z s do v těsně před vrcholem v. Pro u tvrzení musí platit (protože $\delta(s, u) < \delta(s, v)$) máme tak `D[`$u$`]` $= \delta(s, u)$. Odtud dostaneme

$$\texttt{D[}v\texttt{]} > \delta(s, v) = \delta(s, u) + 1 = \texttt{D[}u\texttt{]} + 1 \tag{1}$$

Zaměříme se na moment, kdy algoritmus odebral vrchol u z fronty. Pro vrchol v máme dvě možnosti:

1. `X[v] == true`.
   Vrchol v je buď ve frontě, nebo už ve frontě v minulosti byl. Z předchozího lemmatu pak plyne $\texttt{D}[v] \leqslant \texttt{D}[u] + 1$, což je spor s (1).
2. `X[v] == false`.
   Vrchol v ještě nebyl ve frontě a na řádku 12 algoritmu nastavíme `D[v] ← D[u] + 1`, což je opět spor s (1).

(2) je důsledkem (1) a toho, že `bfs` sestaví strom, který je podgrafem G.

Idea bfs v Dijkstrově algoritmu

Problém nalezení nejkratší cesty
Vstup: (orientovaný) graf $G = (V, H)$, vrchol s, funkce $c : H \to \mathbb{R}^+$ (ohodnocení hran)
Výstup: pro každý vrchol $t \in V$ spočítáme délku nejkratší cesty z s do t

Máme speciální hodnotu ω takovou, že

- pro každé $r \in \mathbb{R}$ platí $r < \omega$,
- pro každé $r \in \mathbb{R}$ platí $\omega + r = r + \omega = \omega$,
- $\omega + \omega = \omega$

Tato hodnota pro nás hraje roli *dostatečně velkého čísla*, pokud neexistuje cesta z s do t, je délka nejkratší cesty z s do t rovna ω.

Dijkstrův algoritmus

1. *inicializace*
   - nastavíme $d(s) \leftarrow 0$
   - nastavíme $A \leftarrow V$
   - pro $u \in V - \{s\}$ nastavíme $d(u) \leftarrow \omega$
2. *nalezneme minimální vrchol*
   - pokud $A = \emptyset$, algoritmus končí
   - nastavíme $m \leftarrow \text{argmin}_{u \in A}\, d(u)$
   - pokud $d(m) = \omega$, algoritmus končí
3. *update nalezených cest*
   - pro všechny sousedy u vrcholu m spočítáme $x \leftarrow d(m) + c(m, v)$ a nastavíme $d(u) \leftarrow \min(d(u), x)$.
   - nastavíme $A \leftarrow A - \{m\}$
   - Pokračujeme krokem 2.

Algoritmus lze zastavit i dříve, pokud hledáme cestu z s do t. Stačí v kroku 2 otestovat, jestli $m = t$.

Prioritní fronta

Uložení dvojic priorita, data.

Hlavní operace

- `insert`. Vložení dvojice.
- `extract-min`. Odebrání dvojice s nejmenší prioritou.
- `decrease-key` Zmenšení priority u dvojice, která je v prioritní frontě.

Prioritní frontu lze vytvořit například pomocí vyhledávacího stromu (kde povolíme více shodných klíčů):

- priority jsou klíče,
- `insert` děláme pomocí `tree-insert`,
- `extract-min` pomocí operací `tree-min` a `tree-remove` ve stromu,
- `decrease-key` pomocí `tree-remove` a `tree-insert`.

Pro prioritní fronty existují i specializované struktury. Např. binární halda z algoritmu heapsort.

`dijkstra`
⟵ `g: Graph` — *součástí je i ohodnocení hran* `g.c`
⟵ `s: Int` — *počáteční vrchol*
⟶ `[]Float` — *pole vzdáleností od* `s`

```
1  Y ← [g.n]Bool — navštíven? Inicializováno na false
2  X ← [g.n]Bool — je v prioritní frontě? Inicializováno na false
3  D ← [g.n]Float — inicializováno na ω
4  vytvoř prázdnou prioritní frontu Q
5  insert(Q, s, 0)
6  D[s] ← 0
7  X[s] ← true
8  Y[s] ← true
```

```
9  while (not empty(Q))
10    u, d ← extract-min(Q)
11    Y[u] ← true
12    D[u] ← d
13    foreach w in g.adj[u] such that Y[u] = false
14       if (not X[w])
15          insert(Q, w, d + g.c(u,w))
16          X[w] ← true
17       else
18          decrease-key(Q, w, d+g.c(u,w))
19 return D
```

Pro operaci `decrease-key` může být nutný mechanismus navíc, viz tabule.

Srovnání s `bfs`

Vidíme tedy, že na Dijkstrův algoritmus se můžeme dívat jako na variantu průchodu do šířky, kde používáme prioritní frontu a operaci `decrease-key`.

Pokud jsou ohodnocení hran v grafu rovná 1 (nebo jsou ohodnocení všech hran stejná), pak Dijkstrův algoritmus je průchodem do šířky. Změna priority na ř. 18 totiž nevede ke změně pořadí v prioritní frontě. Prioritu tak vrcholu přiřazujeme nejvýše jednou, a to při vkládání do prioritní fronty, tedy přesně tak, jak se to děje v algoritmu `bfs`. Jako u `bfs` pak lze ukázat, že pokud vkládáme vrchol do prioritní fronty, nemá menší prioritu, než je maximální priorita mezi vrcholy již v prioritní frontě obsaženými.

Průchod do hloubky

Pro analýzu a aplikace upravíme základní algoritmus následovně.

Přidáme globální data

- Proměnná `time`, inicializovaná na `0`,
- Pole `D` a `F` velikosti `g.n`, prvky inicializovány na `0`.
- Pole `P` velikosti `g.n`, prvky inicializovány na `nil`.
- Pole `X` velikosti `g.n`, prvky inicializovány na `false`.

Přepíšeme do rekurzivního tvaru

Navštívíme všechny vrcholy, ne jenom ty dosažitelné s počátečního vrcholu.

```
dfs-visit
⟵ g: Graph
⟵ t: Int
```

```
1  time ← time + 1
2  D[t] ← time
3  X[t] ← true
4  foreach u in g.adj[t] such that X[u] == false
5     P[u] ← t
6     dfs-visit(g, u)
7  time ← time + 1
8  F[t] ← time
```

```
dfs-all
⟵ g: Graph
```

```
1  for u in 0 ..< g.n
2     if (X[u] == false) then dfs-visit(g, u)
```

Věta

Volání `dfs-all(g,t)` sestaví les. (Podle pole `P`.)

Důkaz

Předpokládejme, že `dfs-all` volá `dfs-visit` postupně pro vrcholy `t1, t2, ...`.

Už víme, že `dfs-visit(g, t1)` sestaví strom s kořenem `t1`, jehož vrcholy jsou všechny vrcholy dosažitelné z `t1`. Díky poli `X` pak `dfs-visit(g, t2)` sestaví strom s kořenem `t2` obsahující vrcholy dosažitelné z `t2` mimo těch, které jsou dosažitelné z `t1`.

Obecně volání `dfs-visit(g, ti)` sestaví strom s kořenem `ti` obsahujícím vrcholy dosažitelné z `ti` mimo těch, které jsou dosažitelné z `tj` pro `j < i`.

Věta (Uzávorkovací)

Po provedení `dfs-all` platí pro libovolné různé vrcholy u, v a les vytvořený `dfs-all` právě jedna z následujících možností.

a. $\langle \mathtt{D}[u], \mathtt{F}[u] \rangle \cap \langle \mathtt{D}[v], \mathtt{F}[v] \rangle = \emptyset$, vrcholy u a v nejsou ve vztahu následovník/předek;
b. $\langle \mathtt{D}[u], \mathtt{F}[u] \rangle \subset \langle \mathtt{D}[v], \mathtt{F}[v] \rangle$, vrchol u je následovníkem vrcholu v;
c. $\langle \mathtt{D}[v], \mathtt{F}[v] \rangle \subset \langle \mathtt{D}[u], \mathtt{F}[u] \rangle$, vrchol v následovníkem vrcholu u.

Důkaz. Z algoritmu vidíme, že $\mathtt{D}[u]$,$\mathtt{D}[v]$,$\mathtt{F}[u]$,$\mathtt{F}[v]$ jsou po dvojicích různé a $\mathtt{D}[u] < \mathtt{F}[u]$ a $\mathtt{D}[v] < \mathtt{F}[v]$. Předpokládejme $\mathtt{D}[u] < \mathtt{D}[v]$.

Pokud $\mathtt{D}[v] < \mathtt{F}[u]$, pak celé rekurzivní volání `dfs-visit` pro vrchol v proběhlo v rámci volání pro vrchol u a tedy $\mathtt{F}[v] < \mathtt{D}[u]$. Oba vrcholy jsou tedy ve stejném stromu a u předkem v. (Připomeňme si důkaz toho, že obecný průchod vytvoří strom.)

Pokud naopak platí, že $\mathtt{D}[v] > \mathtt{F}[u]$, tak dostaneme $\mathtt{D}[u] < \mathtt{F}[u] < \mathtt{D}[v] < \mathtt{F}[v]$ a intervaly jsou disjunktní. Volání `dfs-visit` pro u skončilo před tím, než začalo voláni pro v. Vrcholy tedy leží v různých podstromech. □

Věta (O nenavštívené cestě)

Nechť T je les sestavený `dfs-all` pro graf G. Potom pro libovolné různé vrcholy u, v jsou následující podmínky ekvivalentní:

a. v je následníkem u v lese T,

b. v čase `D[u]` (tj. v momentě, kdy algoritmus nastavoval hodnotu `D[u]`) existovala cesta v grafu z vrcholu u do vrcholu v taková, že pro každý vrchol $w \neq u$ na této cestě platí `X[w]` $=$ `false`.

Důkaz

(a) implikuje (b): Pokud je vrchol w následníkem vrcholu u, pak z věty o uzávorkování plyne `D[u]` $<$ `D[w]`, že v čase `D[u]` tedy muselo být `X[w]` $=$ `false`. Dále musí z u do w existovat v grafu cesta, jinak by se w nemohl stát následníkem u.

(b) implikuje (a): Sporem. Před. že $w \neq u$ je první vrchol na cestě z u do v, který není následníkem u. Nechť z je vrchol, který je na této cestě těsně před w. Z uzávorkovací věty plyne `F[z]` $\leqslant$ `F[u]` (může být $z = u$). Vrchol w není navštíven dříve než vrchol u, proto `D[u]` $<$ `D[w]`. V grafu je hrana (z, w), proto je vrchol w navštíven nejpozději během volání `dfs-visit` pro vrchol z, odtud `D[w]` $<$ `F[z]`. Dohromady tedy `D[u]` $<$ `D[w]` $<$ `F[z]` $\leqslant$ `F[u]`. Z uzávorkovací věty ale plyne, že `F[w]` $<$ `F[u]` a w je potomek u. □

Klasifikace hran grafu

Uvažujme graf G a les T sestavený procedurou `dfs-all` spuštěnou pro G.
Hrana (u, v) je

1. *stromová*, pokud u je v lese T rodičem v,
2. *zpětná*, pokud v je v lese T předchůdcem u,
3. *dopředná*, pokud není stromová a v je v lese T následníkem u,
4. *křížová*, jinak.

Neorientované grafy:

- Pro neorientované grafy musíme doplnit podmínku určující, jestli je hrana $\{u, v\}$ zpětná nebo dopředná (podle předchozí definice může být obojí).
- Řekneme, že `dfs-all` zkoumá hranu $\{u, v\}$ při volání `dfs-visit` pro u, pokud v tomto zavolání při průchodu `adj[u]` narazíme na vrchol v.
- Hranám přiřadíme směr. Pokud `dfs-all` nejdříve zkoumal hranu $\{u, v\}$ ze zavolání `dfs-visit` pro u, bereme hranu orientovanou jako (u, v). Jinak je orientace opačná.

Věta

Každá hrana v neorientovaném grafu je buď stromová nebo zpětná.

Důkaz.

Uvažme libovolnou hranu $\{u, v\}$ a předpokládejme `D[u]` $<$ `D[v]`. Protože v je v seznamu `adj[u]`, musí `dfs-visit` pro v skončit dříve, než to pro u. Máme tak `F[v]` $<$ `F[u]` a podle uzávorkovací věty je v následníkem u.

Pokud algoritmus poprvé zkoumá hranu $\{u, v\}$ při volání `dfs-visit` pro u, musí být `X[v]` $=$ `false`, vrchol v se stane potomkem u a hrana je stromová.

Pokud naopak na tuto hranu narazíme poprvé až během volání pro v, přiřadíme jí orientaci (v, u) a je tak zpětná. □

Složitost algoritmů `dfs-all`, `bfs`

Složitost operací se zásobníkem a frontou je konstantní. Z popisu obecného průchodu pak plyne, že složitost je $O(|V| + |H|)$. (Naše rozšíření přidalo pouze konstantní počet operací provedených pro každý vrchol).

Topologické uspořádání

Orientovaný graf bez cyklů budeme nazývat *dag*. (To je zavedená anglická zkratka pojmu *directed acyclic graph*).

Topologické uspořádání dagu $G = (V, E)$ je lineární uspořádání vrcholů grafu takové, že pokud $(u, v) \in E$, pak u je v tomto uspořádání před v.

(*Jinak řečeno: hrany* G *vedou v tomto uspořádání dopředu.*)

Algoritmus `topol`

- Inicializujeme prázdný seznam vrcholů,
- Spustíme upravený průchod do hloubky. Úprava spočívá v tom, že vždycky, když nastavujeme `F[u]` pro vrchol u, připojíme u na začátek seznamu
- Po skončení průchodu obsahuje seznam vrcholy uspořádané sestupně podle hodnot položky `F`. Topologické uspořádání je dáno pořadím v tomto seznamu.

Složitost: Vkládání vrcholu na začátek seznamu je v konstantním čase a vkládáme $|V|$ vrcholů. To je jediná práce navíc oproti průchodu do hloubky. Proto je složitost $O(|V| + |E|)$.

Věta

Orientovaný graf G obsahuje cyklus, právě když `dfs-all`(G) vytvoří zpětnou hranu.

Důkaz. Pokud `dfs-all` vytvoří zpětnou hranu (u, w), pak vrchol u musí být potomkem w v příslušném stromu. Cesta z w do u v grafu G (která odpovídá cestě mezi těmito vrcholy ve stromu vytvořeném `dfs-all`) prodloužená o hranu (u, w) je kružnice.

Nechť G obsahuje cyklus c a w je první vrchol na tomto cyklu, pro který je zavoláno `dfs-visit`. Vrchol, který w na cyklu předchází označme u. V čase `D[`w`]` tvoří část c bez w nenavštívenou cestu. Podle věty o nenavštívené cestě pak musí být vrchol u následníkem vrcholu w ve stromu sestaveném `dfs-all`. Hrana (u, w) je proto zpětná. □

Korektnost algoritmu

Věta

Algoritmus `topol` nalezne topologické uspořádání.

Důkaz. Ukážeme, že pro každou hranu (u, v) platí, že `F[`u`]` $>$ `F[`v`]`.

V momentě, kdy `dfs-all` zkoumá hranu (u, v) (v rekurzivním zavolání `dfs-visit` pro u), máme dvě možnosti:

- `X[`v`]` $=$ `true`
  Potom algoritmus již nastavil `D[`v`]` a tak musí platit `D[`v`]` $<$ `F[`u`]`. Pokud bychom měli `F[`u`]` $<$ `F[`v`]`, tak v důsledku věty o uzávorkování máme `D[`v`]` $<$ `D[`u`]` $<$ `F[`u`]` $<$ `F[`v`]`, a hrana (u, v) je zpětná. To je ovšem podle předchozí věty spor, a proto musí plati `F[`v`]` $<$ `F[`u`]`.
- `X[`v`]` $=$ `false`
  Potom volání `dfs-visit` pro v proběhne v rámci volání `dfs-visit` pro u, a máme tak `F[`v`]` $<$ `F[`u`]`. □

Silně souvislé komponenty

Zafixujeme orientovaný graf $G = (V, H)$.

Zavedeme značení pro existenci cest v grafu:

- $u \rightsquigarrow v$ značí, že cesta z u do v existuje
- $u \not\rightsquigarrow v$ značí, že cesta z u do v neexistuje

Množina vrcholů C v orientovaném grafu je *silně souvislá*, pokud libovolné různé vrcholy $u, v \in C$ máme $u \rightsquigarrow v$ a $v \rightsquigarrow u$.

Množina vrcholů C v orientovaném grafu je *silně souvislá komponenta*, pokud

- je silně souvislá,
- neexistuje vrchol, který bychom mohli do C přidat tak, abychom po přidání dostali silně souvislou množinu.

Věta

Nechť C a C' jsou různé silně spojité komponenty. Pro libovolné $u, v \in C$, $u', v' \in C'$ pak $u \rightsquigarrow u'$ implikuje $v' \not\rightsquigarrow v$.

Důkaz.

$u \rightsquigarrow u'$ implikuje, že pro každé $x \in C, y \in C'$ máme $x \rightsquigarrow y$. Pokud bychom měli $v' \rightsquigarrow v$, pak pro každé $z \in C', w \in C$ máme $z \rightsquigarrow w$. Tedy C a C' by nemohli být různé silně spojité komponenty. □

Důsledkem předchozí věty je, že množina silných komponent v grafu je rozkladem množiny vrcholů grafu. Množina tvořená jedním vrcholem je totiž silně souvislá a proto je každý vrchol v nějaké silně souvislé komponentě.

Můžeme sestavit graf komponent $\overline{G} = (\overline{V}, \overline{H})$:

- $\overline{V}$ je množina všech silně souvislých komponent v G
- $(C, C') \in \overline{H}$ když existují $u \in C, v \in C'$ tak, že $(u, v) \in H$.

Z předchozí věty plyne, že $\overline{G}$ je dag.

Klíčová věta

Předp. že jsme pro G provedli `dfs-all` a máme k dispozici pole `X, D, F`.

Pro $W \subseteq V$ zavedeme: $\texttt{F}[W] = \max_{v \in W} \texttt{F}[v]$.

Věta

Pro libovolné různé silně souvislé komponenty C, C' platí: Pokud existuje hrana (u, v) taková, že $u \in C$, $v \in C'$, pak $\texttt{F}[C] > \texttt{F}[C']$.

Důkaz. Pro $W \subseteq V$ zavedeme: $\texttt{D}[W] = \min_{v \in W} \texttt{D}[v]$.

Případ I: $\texttt{D}[C] < \texttt{D}[C']$
Vybereme vrchol $x \in C$, pro který $\texttt{D}[C] = \texttt{D}[x]$. Potom pro libovolný $y \in C \cup C'$ různý od x máme $\texttt{D}[x] < \texttt{D}[y]$ (protože $x \rightsquigarrow y$). V čase $\texttt{D}[x]$ (tedy na začátku volání `dfs-visit` pro vrchol x) tedy existuje nenavštívená cesta z x do y a tedy y je ve stromu sestaveném `dfs-all` potomkem vrcholu x. Z uzávorkovací věty potom plyne, že $\texttt{F}[x] > \texttt{F}[y]$.

Případ 2: $\texttt{D[C]} > \texttt{D[C']}$
Vybereme vrchol $x \in C'$, pro který $\texttt{D[C]} = \texttt{D[x]}$. Potom pro libovolný $y \in C'$ máme $\texttt{D[x]} < \texttt{D[y]}$ a v čase $\texttt{D[x]}$ existuje nenavštívená cesta do y, vrchol y je tak ve stromu sestaveném `dfs-all` potomkem x a podle uzávorkovací věty tak máme $\texttt{F[x]} < \texttt{F[y]}$ a tedy $\texttt{F[x]} = \texttt{F[C']}$. Protože $x \not\rightsquigarrow z$ pro libovolný vrchol $z \in C$, máme $\texttt{D[C]} > \texttt{F[C']}$ a tedy $\texttt{F[C]} > \texttt{F[C']}$. □

Důsledek předchozí věty: Pokud bychom v grafu komponent procházeli vrcholy podle příslušných hodnot `F[]`, projdeme je v topologickém uspořádání.

Idea algoritmu:

1. Najdeme vrchol u s maximálním `F[]` v celém grafu.
2. Nalezneme silně souvislou komponentu, ve které se u nachází.
3. Odstraníme tuto komponentu z grafu a pokračujeme od 1 kroku, dokud je graf neprázdný.

Krok 2 lze realizovat pomocí `dfs-visit`, pokud dokážeme zajistit, aby `dfs-visit` nenavštívil vrcholy z jiných komponent. Toho lze dosáhnout obrácením směr hran!

V kroku 3 potom nemusíme z grafu nic odstraňovat, odstranění supluje práce s polem `X` v algoritmu `dfs-visit`.

Transponovaný graf

Transpozicí orientovaného grafu $G = (V, H)$ je orientovaný graf $G^T = (V, H^T)$, kde $H^T = \{(u, v) \mid (v, u) \in H\}$.

Věta

G a G^T mají stejné množiny silně souvislých komponent.

Důkaz. Pro libovolné vrcholy máme $u \rightsquigarrow v$ v grafu G, právě když $v \rightsquigarrow u$ v grafu G^T. □

Důsledek: $(\overline{G})^T = \overline{(G^T)}$

Algoritmus `strong-components`

1. Spočítáme topologické uspořádání pomocí (`dfs-all`) pro graf G.
2. Zavoláme `dfs-all` pro graf G^T s tím, že v hlavní smyčce `dfs-all` procházíme vrcholy v pořadí podle uspořádání získaného v prvním kroku.
3. Silně souvislé komponenty pak odpovídají stromům v lese sestaveném `dfs-all`.

Složitost: Kroky 1 a 2 jsou v čase $\Theta(|V| + |E|)$, sestavení G^T je v čase $\Theta(|E|)$, celkem tedy $\Theta(|V| + |E|)$.

Korektnost algoritmu

Věta

Algoritmus `strong-components` nalezne právě všechny silně souvislé komponenty grafu.

Důkaz

Značení: `F[]`, `D[]` patří k `dfs-all` řádku 1 algoritmu. `F[]'`, `D[]'` k průchodu na řádku 2.
Indukcí přes počet spuštění `dfs-visit` v hlavní smyčce algoritmu `dfs-all` z řádku 2 ukážeme, že doposud vytvořené stromy odpovídají právě silně souvislým komponentám.

Pro 0 spuštění je tvrzení triviálně pravdivé.

Předpokládejme, že tvrzení platí pro prvních k spuštění, a že další spuštění `dfs-visit` je pro vrchol x patřící do silně souvislé komponenty C.
V čase `D[`x`]'` existuje nenavštívená cesta z vrcholu x do libovolného jiného vrcholu v komponentě C (z indukčního předpokladu plyne, že žádný vrchol v C dosud nebyl navštíven).
Všechny vrcholy v C tedy patří podle věty o nenavštívené cestě do jednoho stromu.
Zbývá ukázat, že do tohoto stromu nepatří vrchol mimo C. To plyne z toho, že v grafu G^T pro všechny hrany (u, v) takové, že $u \in C, v \notin C$ platí, že v patří do komponenty C' s `F[`C'`]` $>$ `F[`C`]`, tzn. v čase `D[`x`]'` už byl vrchol v navštíven. □