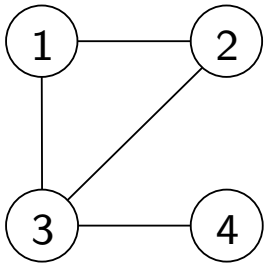
1 Průzkum grafů a grafová souvislost

- Průzkum do šířky
- Průzkum do hloubky
- Topologické uspořádání
- Silně souvislé komponenty

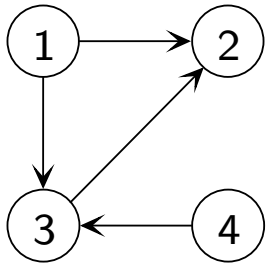
Graf a jeho reprezentace

- graf $G = (V, E)$
  - orientovaný / neorientovaný
  - ohodnocené hrany / vrcholy
  - jednoduché / násobné hrany
- reprezentace grafů
  - seznam následníků
  - incidenční matice
- složitost grafových algoritmů je funkcí počtu vrcholů a hran
- používáme zjednodušenou notaci, např. $\mathcal{O}(V + E)$

Incidenční matice



	1	2	3	4
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1
4	0	0	1	0



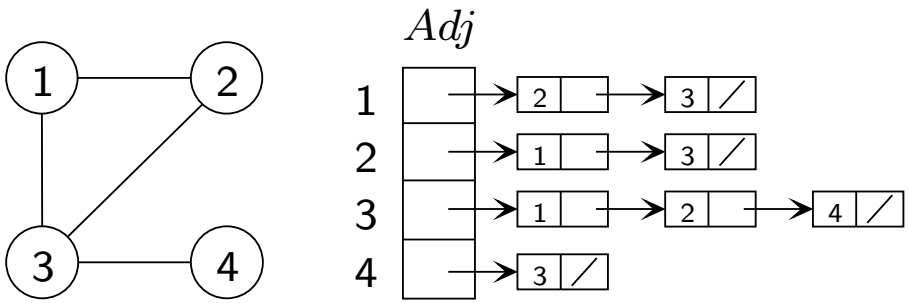
	1	2	3	4
1	0	1	1	0
2	0	0	0	0
3	0	1	0	0
4	0	0	1	0

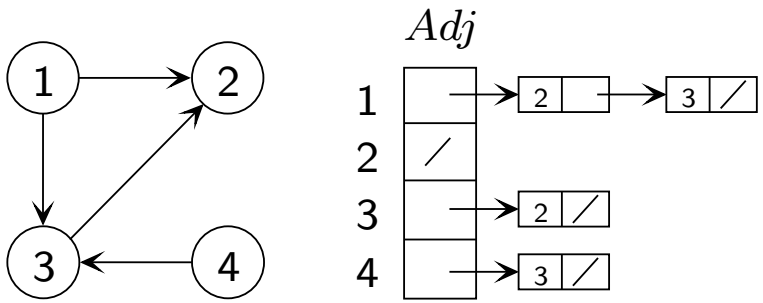
- matice $A = (a_{ij})$ rozměrů $|V| \times |V|$, kde

$$a_{ij} = \begin{cases} 1 & \text{pokud } (i,j) \in E \\ 0 & \text{jinak} \end{cases}$$

- prostorová složitost: $\Theta(V^2)$
- vhodné pro husté grafy
- časová složitost výpisu všech sousedů vrcholu u je $\Theta(V)$
- časová složitost ověření zda $(u,v) \in E$ je $\Theta(1)$

Seznam následníků





- pole Adj velikosti $|V|$
- prostorová složitost: $\Theta(V + E)$
- vhodné pro řídké grafy
- časová složitost výpisu všech sousedů vrcholu u je $\Theta(deg(u))$ ($deg(u)$ je stupeň vrcholu u)
- časová složitost ověření zda $(u, v) \in E$ je $\mathcal{O}(deg(u))$

varianta použít místo pole hašovací tabulku, zřetězené hašování nahradit otevřenou adresací

Srovnání

	incidenční matice	seznam následníků	hašovací tabulka
test $\{u, v\} \in E$	$\mathcal{O}(1)$	$\mathcal{O}(V)$	$\mathcal{O}(1)$
test $(u, v) \in E$	$\mathcal{O}(1)$	$\mathcal{O}(V)$	$\mathcal{O}(1)$
seznam sousedů vrcholu v	$\mathcal{O}(V)$	$\mathcal{O}(1 + deg(v))$	$\mathcal{O}(1 + deg(v))$
seznam hran	$\mathcal{O}(V^2)$	$\mathcal{O}(V + E)$	$\mathcal{O}(V + E)$
přidání hrany	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)^*$
odstranění hrany	$\mathcal{O}(1)$	$\mathcal{O}(V)$	$\mathcal{O}(1)^*$

* *očekávaná složitost*

Průzkum grafu

Everything on earth can be found, if only you do not let yourself be put off searching. *Philemon of Syracuse (ca. 360 BC–264 BC)*

- pro daný graf G a vrchol s grafu je cílem
  - navštívit všechny vrcholy grafu dosažitelné z vrcholu s, resp.
  - navštívit všechny vrcholy grafu
- průzkum realizovat maximálně efektivně, tj. se složitostí $\mathcal{O}(V + E)$ (*vyhnout se opakovaným návštěvám*)
- jaké další informace o grafu zjistíme v průběhu průzkumu???

Průzkum grafu do šířky vs do hloubky

Theseus si před bludištěm uváže jeden konec nitě na strom a vsoupí dovnitř. V prvním vrcholu (křižovatce) si vybere jednu možnou cestu / hranu a projde po ní do dalšího vrcholu. Aby Theseus neměl zmatek v tom, které hrany už prošel, tak si všechny hrany, které prochází označuje křídou – a to na obou koncích. V každém vrcholu, do kterého Theseus dorazí, provede následující:

- *Pokud na zemi najde položenou niť, tak ví, že už ve vrcholu byl a že se do něj při namotávání nitě zase vrátí. Odloží tedy další prozkoumávání tohoto vrcholu na později, provede čelem vzad a začne namotávat niť na klubko. To ho dovede zpátky do předchozího vrcholu.*
- *Pokud na zemi žádnou niť nenajde, tak se vydá první možnou neprošlou hranou. Pokud by taková hrana neexistovala, tak je vrchol zcela prozkoumán. V tom případě Theseus neztrácí čas a začne namotávat niť na klubko. Tím se dostane zpátky do předchozího vrcholu.*

Tímto postupem prozkoumá celé bludiště a nakonec se vrátí do výchozího vrcholu.[1]

[1] Jakub Černý: Základní grafové algoritmy `http://kam.mff.cuni.cz/~kuba/ka`

Průzkum grafu do šířky vs do hloubky

implementace

křída proměnná označující jestli jsme hranu prošli

klubko položená niť vyznačuje cestu z výchozího do aktuálního vrcholu, cestu si pamatujeme jako posloupnost vrcholů na této cestě. Pro uložení cesty použijeme zásobník. Odmotávání nitě odpovídá přidání vrcholu do zásobníka. Namotávání nitě při návratu zpět odpovídá odebrání vrcholu ze zásobníku.

Průzkum grafu do šířky vs do hloubky

Tento průchod (prohledání grafu) si můžeme představit tak, že se do výchozího vrcholu postaví miliarda Číňanů a všichni naráz začnou prohledávat graf. Když se cesta rozdělí, tak se rozdělí i dav řítící se hranou. Předpokládáme, že všechny hrany jsou stejně dlouhé. Graf prozkoumáváme „po vlnách“. V první vlně se všichni Číňané dostanou do vrcholů, dokterých vede z výchozího vrcholu hrana. V druhé vlně se dostanou do vrcholů, které jsou ve vzdálenosti 2 od výchozího vrcholu. Podobně v k-té vlně se všichni Číňané dostanou do vrcholů ve vzdálenosti k od výchozího vrcholu. Kvůli těmto vlnám se někdy průchodu do šířky říka algoritmus vlny. [2]

implementace

V počítači vlny nasimulujeme tak, že při vstupu do nového vrcholu uložíme všechny možné cesty do fronty. Frontu průběžně zpracováváme.

[2] Jakub Černý: Základní grafové algoritmy `http://kam.mff.cuni.cz/~kuba/ka`

Průzkum grafu do šířky a do hloubky

průzkum grafu do **šířky** vs do **hloubky** se liší pouze použitím **fronty** a **zásobníku**

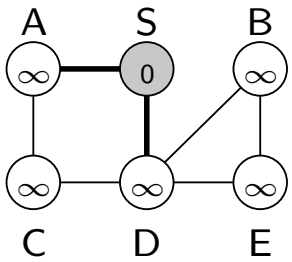
NE

Průzkum do šířky - strategie

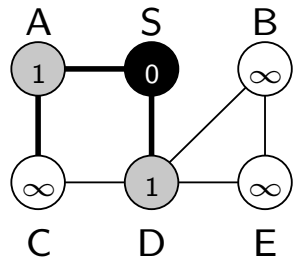
cílem je prozkoumat všechny vrcholy dosažitelné z daného iniciálního vrcholu s

- postupujeme od iniciálního vrcholu s po *vrstvách*
- nejdříve prozkoumáme všechny vrcholy dosažitelné z s po 1 hraně
- pak všechny vrcholy dosažitelné po 2 hranách, po 3 hranách atd.
- pro manipulaci s vrcholy používáme prioritní frontu Q
- $v \in Q$ právě když byl dosažen (objeven) ale ještě nebyl prozkoumán

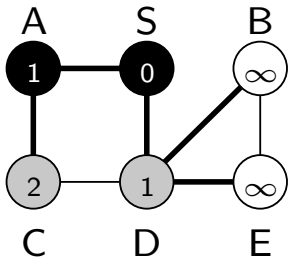
Průzkum do šířky - příklad



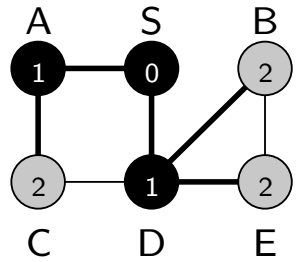
Q: S



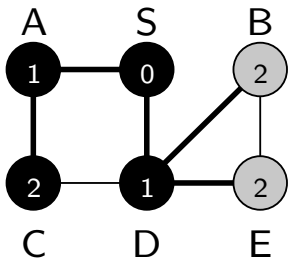
Q: AD



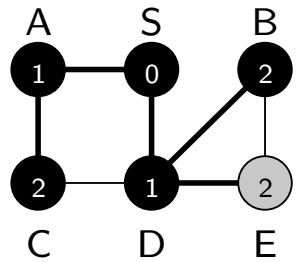
Q: DC



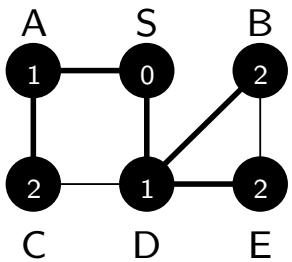
Q: CBE



Q: BE



Q: E



Q: ∅

Průzkum do šířky - atributy vrcholu

$v.color$

- v průběhu výpočtu je vrchol postupně objeven (je zařazen do fronty) a prozkoumán (všechny sousedící vrcholy jsou objeveny)
- vrchol má **černou** barvu právě když je dosažitelný z iniciálního vrcholu a byl již prozkoumán
- vrchol má **šedivou** barvu právě když je dosažitelný z iniciálního vrcholu, byl již objeven, ale nebyl ještě prozkoumán
- vrchol má **bílou** barvu právě když není dosažitelný z iniciálního vrcholu anebo ještě nebyl objeven

$v.\pi$

- vrchol, ze kterého byl v objeven

$v.d$

- délka (počet hran) cesty z s do v, na které byl v objeven
  ($=$ *délka nejkratší cesty z* s *do* v)

Průzkum do šířky - implementace

BFS(G, s)

```
1  foreach u ∈ V \ {s}
2     do u.color ← white;  u.d ← ∞;  u.π ← Nil od
3  s.color ← gray;  s.d ← 0;  s.π ← Nil
4  Q ← ∅
5  Enqueue(Q, s)
6  while Q ≠ ∅ do
7          u ← Dequeue(Q)
8          foreach v ∈ Adj[u] do
9             if v.color = white
10               then v.color ← gray
11                    v.d ← u.d + 1
12                    v.π ← u
13                    Enqueue(Q, v) fi
14            u.color ← black od
15 od
```

BFS - kompaktní verze

BFS(G, s)

```
1  foreach u ∈ V \ {s}
2     do u.d ← ∞ od
3  s.d ← 0
4  Q ← ∅
5  Enqueue(Q, s)
6  while Q ≠ ∅ do
7           u ← Dequeue(Q)
8           foreach v ∈ Adj[u] do
9              if v.d = ∞
10                then v.d ← u.d + 1
11                     Enqueue(Q, v) fi
12          od
13 od
```

BFS - složitost

- operace vložení a odstranění vrcholu z fronty mají konstantní složitost, každý vrchol je ve frontě maximálně jednou; celkově $\mathcal{O}(V)$
- seznam následníků každého vrcholu se prochází maximálně jednou; průzkum hrany má konstantní složitost; celkově $\mathcal{O}(E)$
- inicializace má složitost $\Theta(V)$
- celková složitost BFS je $\mathcal{O}(V + E)$

BFS a nejkratší cesty v neohodnoceném grafu

Nejkratší cesta v nehohodnoceném grafu

Délka nejkratší cesty z s do v, značíme $\delta(s, v)$, je definována jako minimální počet hran na cestě z s do v. Když neexistuje žádná cesta z s do v, tak $\delta(s, v) = \infty$.

Nejkratší cestou z s do v je každá cesta z s do v která má $\delta(s, v)$ hran.

BFS a nejkratší cesty v neohodnoceném grafu

Věta 1

Nechť $G = (V, E)$ je graf a $s \in V$ jeho vrchol, na které aplikujeme algoritmus BFS. *Pak po ukončení výpočtu pro každý vrchol $v \in V$ platí*

$$v.d = \delta(s, v)$$

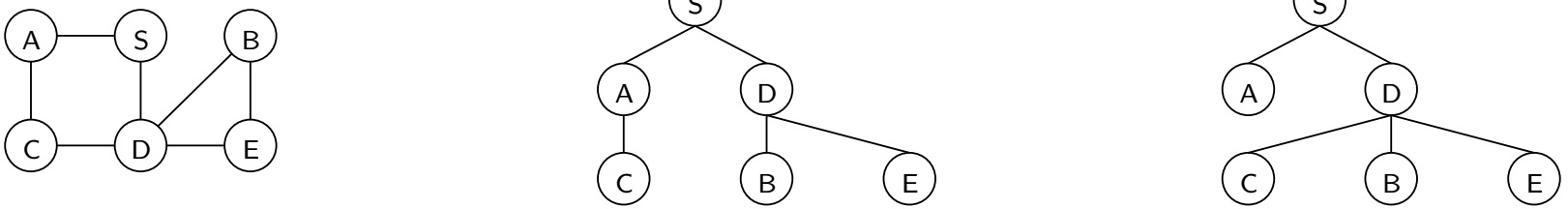
dokážeme indukcí podle hodnoty $v.d$

1. vrchol s má $v.s = 0$ a nejkratší cesta z s do s má nula hran
2. předpokládejme, že pro všechny vrcholy s hodnotou $v.d \leq k$ je $v.d$ délka nejkratší cesty do v
3. potřebujeme ukázat indukční krok a to je, že každý vrchol v s $v.d = k + 1$ leží ve vzdálenosti $k + 1$ od s
   - pokud ne, tak existuje kratší cesta z s do v a nechť (w, v) je poslední hrana na tého cestě
   - $w.d < k$
   - potom se ale měl algoritmus při zpracovávání vrcholu w podívat na hranu (w, v) a nastavit hodnotu $v.d$ na $w.d + 1$
   - $w.d + 1 < k + 1$, spor

BFS strom a nejkratší cesty

- algoritmus BFS definuje přes atributy π **graf předchůdců**
- formálně: pro graf $G = (V, E)$ a iniciální vrchol s je graf předchůdců $G_\pi = (V_\pi, E_\pi)$ definovaný předpisem
$$V_\pi = \{v \in V \mid v.\pi \neq Nil\} \cup \{s\}$$
$$E_\pi = \{(v.\pi, v) \mid v \in V_\pi \setminus \{s\}\}$$
- graf předchůdců se nazývá **BFS strom**

- BFS strom je kostrou grafu
- pro každý vrchol $v \in V_\pi$ obsahuje BFS strom jedinou cestu z s do v, která je současně **nejkratší cestou z s do v**



graf a jeho dva různé BFS stromy

BFS a graf s ohodnocenými hranami

namísto fronty použijeme prioritní frontu

- do fronty vkládáme dvojici (vrchol; délka hrany, po které by objeven)
- prioritou je délka hrany
- z fronty vybíráme vždy nejkratší hranu
- **BFS strom je nejlevnější kostrou grafu**
- Primův algoritmus

- vrcholu ve frontě aktualizujeme hodnotu $v.d$ pokaždé, když je po nějaké hraně objeven
- prioritou je hodnota $v.d$
- z fronty vybíráme vždy vrchol s nejnižší hodnotou $v.d$
- **BFS strom je strom nejkratších cest z s do ostatních vrcholů grafu**
- Dijkstrův algoritmus

podrobnosti o obou algoritmech později

Aplikace a algoritmy využívající BFS

- Peer to Peer Networks
- Crawlers in Search Engines
- Social Networking Websites - hledání osob *ve vzdálenosti* nejvíce k
- GPS navigační systémy
- broadcasting
- garbage collection
- Fordův Fulkersonův algoritmus pro hledání maximálního toku v síti
- testování bipartitnosti

Bipartitní grafy

Definice 2

Neorientovaný graf se nazývá bipartitní právě když se jeho množina vrcholů dá rozdělit na dvě disjunktní množiny tak, že žádné dva vrcholy patřící do stejné množiny nejsou spojeny hranou.

alternativní formulace: vrcholy grafu je možné obarvit dvěma různými barvami tak, že každé dva vrcholy spojené hranou mají různou barvu

aplikace: vytváření dvojic, rozvrhu, ...

Testování bipartitnosti využitím BFS

Bipartitní graf neobsahuje cyklus liché délky.

- zvolíme libovolný vrchol grafu jako iniciální vrchol s
- BFS průzkum z vrcholu s definuje vrstvy $L_0, L_1, L_2, \ldots$
- do vrstvy L_i patří vrcholy, jejichž vzdálenost od s je i (t.j. $v.d = i$)

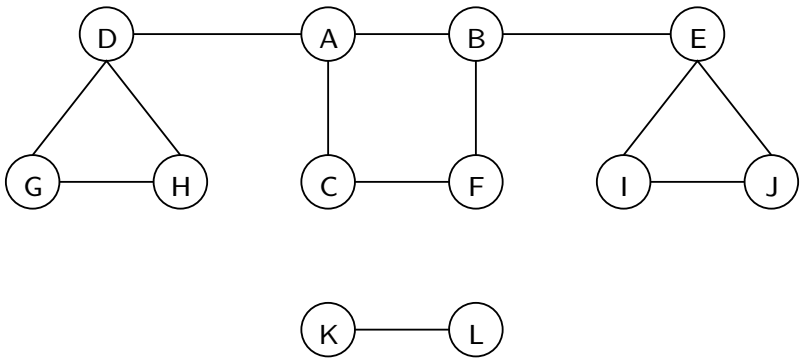
žádné dva vrcholy patřící do stejné vrstvy nejsou spojeny hranou

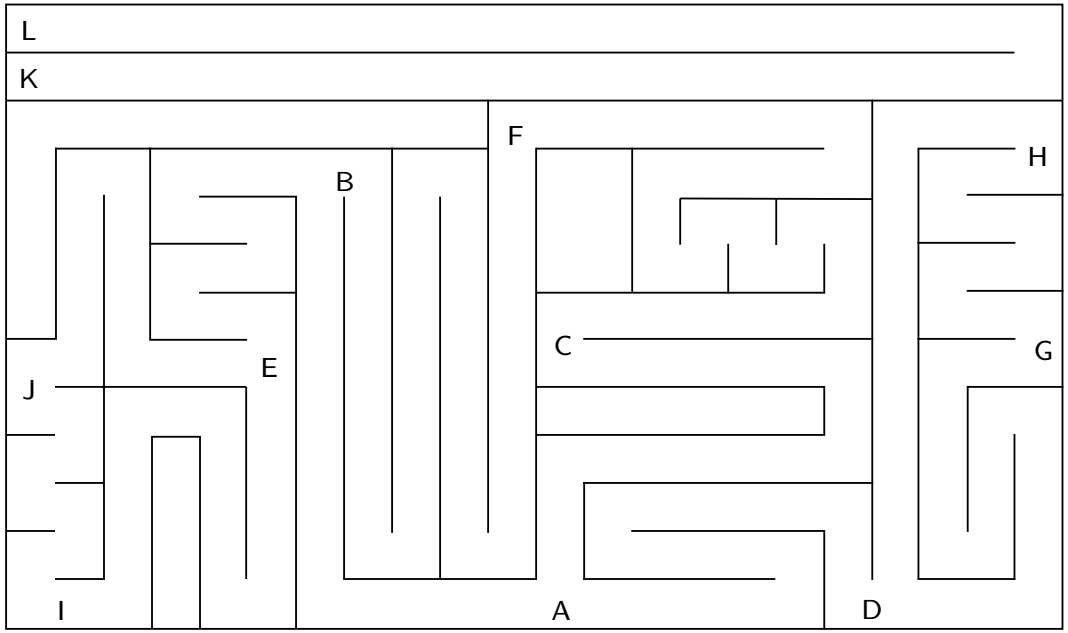
- obarvení vrcholů je určené vrstvami: vrcholy jejichž vzdálenost od s je **sudá** (**lichá**) mají **modrou** (**červenou**) barvu
- korektnost obarvení plyne z předpokladu o neexistenci hrany mezi vrcholy ze stejné vrstvy

existují dva vrcholy spojeny hranou a patřící do stejné vrstvy

- nechť u, v jsou vrcholy takové, že $u, v \in L_i$ a $\{u, v\} \in E$
- nechť y je nejmenší společný předchůdce vrcholů u a v v BFS stromu
- cesta z y do u, hrana $\{u, v\}$ a cesta z v do y tvoří cyklus, jehož délka je lichá (protože cesta z y do u a cesta z v do y mají stejnou délku)
- graf není bipartitní

Průzkum grafu do hloubky- motivace





pořadí, v němž BFS zkoumá vrcholy, netvoří souvislou cestu v grafu

Formulace problému

- průzkum do šířky a stejně tak i průzkum do hloubky je možné použít buď k prozkoumání té části grafu, která je dosažitelná z iniciálního vrcholu, anebo k prozkoumání celého grafu
- průzkum se dá aplikovat na orientované i neorientované grafy

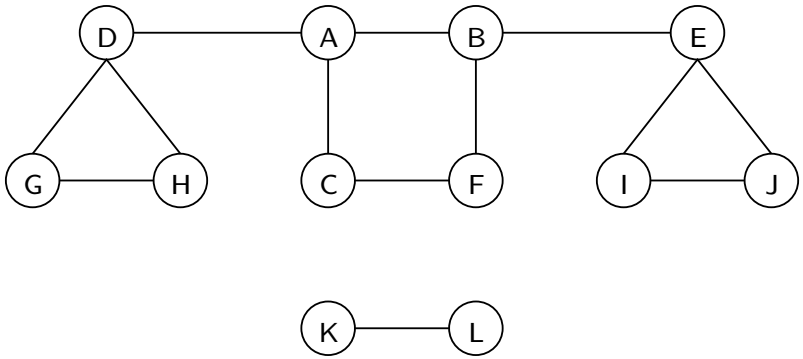
- prezentace průzkumu do hloubky předpokládá, že
  - vstupem je orientovaný graf a
  - cílem je prozkoumat celý graf

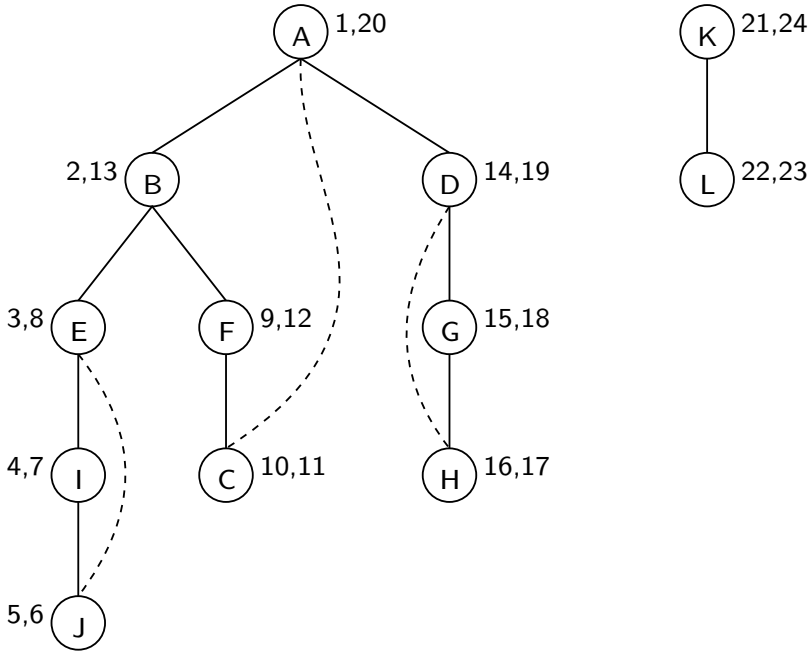
Průzkum do hloubky - strategie

- na začátku výpočtu a vždy po dokončení průzkumu vybereme jeden z dosud neprozkoumaných vrcholů a zvolíme ho za nový iniciální vrchol
- označ iniciální vrchol jako objevený
- vyber neprozkoumanou hranu (u, v), která vychází z naposledy objeveného vrcholu u, a když její koncový vrchol v ještě nebyl prozkoumán, tak ho označ jako objevený
- když všechny hrany vycházející z naposledy objeveného vrcholu u byly prozkoumány, tak ukonči průzkum vrcholu u a pokračuj vrcholem, ze kterého byl vrchol u objeven
- průzkum končí když jsou prozkoumány všechny vrcholy dosažitelné z iniciálního vrcholu

- pro manipulaci s vrcholy používáme zásobník

Průzkum grafu do hloubky - příklad





Průzkum do hloubky - atributy vrcholu

$v.color$

- vrchol má **černou** barvu právě když je dosažitelný z iniciálního vrcholu a byl již prozkoumán, tj. byly prozkoumány všechny hrany vycházející z vrcholu
- vrchol má **šedivou** barvu právě když je dosažitelný z iniciálního vrcholu, byl již objeven, ale nebyl ještě prozkoumán
- vrchol má **bílou** barvu právě když není dosažitelný z iniciálního vrcholu anebo ještě nebyl objeven

$v.\pi$

- vrchol, ze kterého byl v objeven

$v.d$

- časová značka, která zaznamenává čas první návštěvy vrcholu *(discovery time)*

$v.f$

- časová značka, která zaznamenává čas ukončení průzkumu vrcholu *(finishing time)*

Průzkum do hloubky - implementace

DFS(G)

```
1 foreach u ∈ V do u.color ← white;   u.π ← Nil od
2 time ← 0
3 foreach u ∈ V do
4     if u.color = white then DFS_Visit(G, u) fi od
```

DFS_Visit(G, u)

```
1 time ← time + 1
2 u.d ← time
3 u.color ← gray
4 foreach v ∈ Adj[u] do
5     if v.color = white then v.π ← u
6                             DFS_Visit(G, v) fi od
7 u.color ← black
8 time ← time + 1
9 u.f ← time
```

DFS - složitost

- oba cykly v DFS mají složitost $\Theta(V)$
- $\mathrm{DFS\_Visit}$ se pro každý vrchol grafu volá jednou, protože bezprostředně po zavolání dostává vrchol šedivou barvu
- každá hrana se v cyklu procedury $\mathrm{DFS\_Visit}$ prozkoumá právě jednou; ostatní operace mají konstantní složitost
- celková složitost DFS je $\mathcal{O}(V + E)$

Průzkum do hloubky - iterativní implementace

DFS_Iterative_Visit(G, u)

```
1  S ← ∅
2  S.push(u)
3  time ← time + 1; u.d ← time
4  u.color ← gray
5  while S ≠ ∅ do
6          u ← S.pop()
7          if  existuje hrana (u, v) taková, že v.color = white
8            then S.push(u)
9                 S.push(v)
10                v.color ← gray
11                v.π ← u
12                time ← time + 1; v.d ← time
13           else u.color ← black
14                time ← time + 1; u.f ← time fi od
```

DFS strom

- analogicky jako u BFS definují atributy $.\pi$ graf předchůdců
- protože prohledáváme celý graf, který nemusí být nutně souvislý, graf předchůdců je DFS les, který se skládá z DFS stromů
- $G_\pi = (V, E_\pi)$

  $E_\pi = \{(v.\pi, v) \mid v \in V \text{ a } v.\pi \neq Nil\}$

DFS - vlastnosti časových značek

časové značky, které DFS přiřadí vrcholům grafu, obsahují informace o struktuře grafu a DFS stromů

- pro každý vrchol u platí $u.d < u.f$
- s každým vrcholem u je asociovaný interval $[u.d, u.f]$

časové značky určují uspořádání vrcholů

preoder uspořádání podle značky $.d$ (discovery time) v rostoucím pořadí

postorder uspořádání podle značky $.f$ (finishing time) v rostoucím pořadí

reverse postorder uspořádání podle značky $.f$ (finishing time) v klesajícím pořadí

DFS - vlastnosti časových značek

podmínky správného uzávorkování

pro každé dva vrcholy u, v platí právě jedna z podmínek

- intervaly $[u.d, u.f]$ a $[v.d, v.f]$ jsou disjunktní
  u není následníkem v v DFS stromu a symetricky
  v není následníkem u v DFS stromu
- interval $[u.d, u.f]$ je celý obsažen v intervalu $[v.d, v.f]$
  u je následníkem v v DFS stromu
- interval $[v.d, v.f]$ je celý obsažen v intervalu $[u.d, u.f]$
  v je následníkem u v DFS stromu

DFS - vlastnosti časových značek

- vrchol v je dosažitelný z vrcholu u v DFS stromu grafu G právě když

$$u.d < v.d < v.f < u.f$$

- **vlastnost bílé cesty**
  v DFS stromu grafu G je vrchol v následníkem vrcholu u právě když v čase $u.d$ existuje cesta z u do v obsahující jenom bílé vrcholy

DFS - klasifikace hran

stromová hrana (*tree edge*) je hrana (u, v) obsažená v DFS lese
při průzkumu hrany je vrchol v bílý
$u.d < v.d < v.f < u.f$

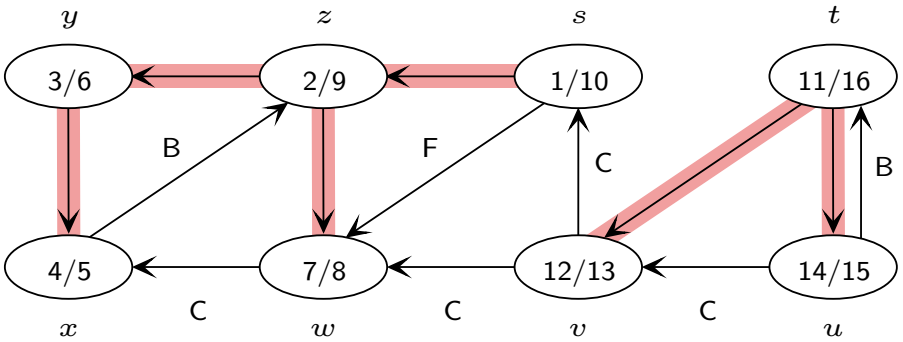
zpětná hrana (*back edge*) je hrana (u, v) která spojuje vrchol u s jeho předchůdcem v v DFS stromu
při průzkumu hrany je vrchol v šedivý
$v.d < u.d < u.f < v.f$

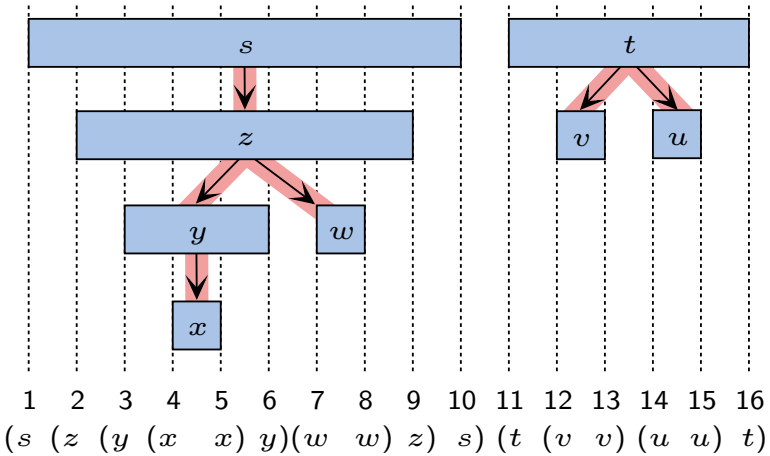
dopředná hrana (*forward edge*) je hrana (u, v), která nepatří do DFS stromu a která spojuje vrchol u s jeho následníkem v DFS stromu
při průzkumu hrany je vrchol v černý
$u.d < v.d < v.f < u.f$

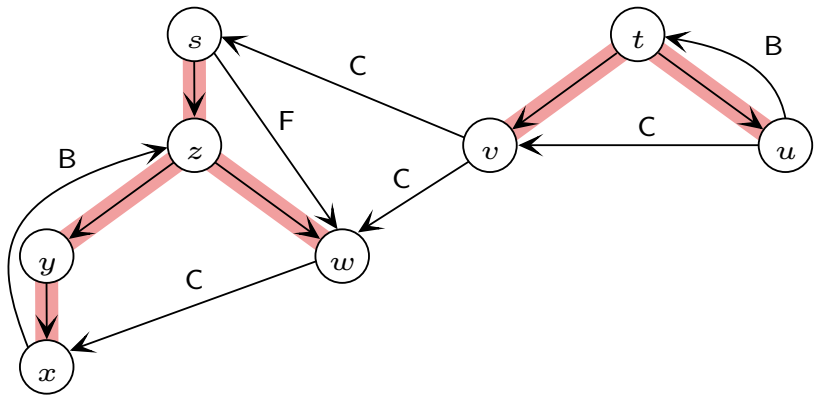
příčná hrana (*cross edge*) všechny ostatní hrany
při průzkumu hrany je vrchol v černý
$v.d < v.f < u.d < u.f$

všechny hrany v neorientovaném grafu jsou buď stromové anebo zpětné

Časové značky a klasifikace hrán - příklad







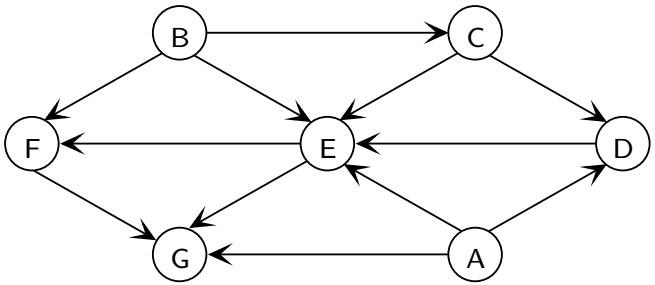
stromové hrany jsou zvýrazněné, zpětné hrany jsou označeny písmenem B, dopředné písmenem F a příčné písmenem C

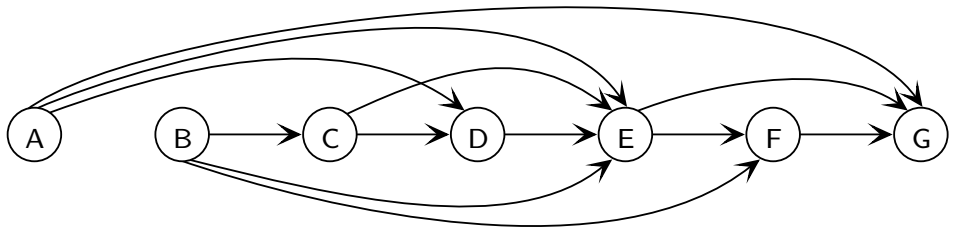
Aplikace a algoritmy využívající DFS

- topologické uspořádání
- komponenty souvislosti
- artikulace a mosty
- testování planarity
- hledání cesty v bludišti
- generování bludiště

Topologické uspořádání vrcholů grafu

Topologické uspořádání vrcholů v orientovaném acyklickém grafu je takové očíslování vrcholů čísly 1 až n (n je počet vrcholů grafu), že každá hrana vede z vrcholu s nižším číslem do vrcholu s vyšším číslem.





$A < B < C < D < E < F < G$

aplikace: modelování procesů, které jsou částečně uspořádané

problém

- existuje v daném grafu topologické uspořádání?
- když ano, tak nalezení takového uspořádání

Topologické uspořádaní - podmínky

Lema 3

Orientovaný graf G má topologické uspořádání právě když je acyklický.

Důkaz

$\Rightarrow$ z definice

$\Leftarrow$ důkaz indukcí k počtu vrcholů grafu

- tvrzení platí pro $n = 1$
- v orientovaném grafu G s $n > 1$ vrcholy najdi vrchol v, do kterého nevstupuje žádná hrana (*kdyby takový neexistoval, tak by jsme mohli z libovolného vrcholu jít donekonečna „pozpátku“ a našli by jsme cyklus*)
- $G - v$ je acyklický (*odstranění vrcholu nemůže způsobit vznik cyklu*)
- z indukčního předpokladu $G - v$ má topologické uspořádání
- topologické uspořádání pro G má na prvním místě v následované uspořádáním pro $G - v$

Topologické uspořádání - naivní algoritmus

Topological_Sort_Visit(G)

```
1 n ← |V|
2 for i = 1 to n do
3     v ← libovolný vrchol, do kterého nevstupuje žádná hrana
4     S[i] ← v
5     odstraň z G vrchol v a všechny jeho hrany
6 od
7 return S[1...n]
```

- algoritmus pro acyklický graf
- nalezení vrcholu do kterého nevstupuje žádná hrana má složitost ???
- celková složitost algoritmu je ???

- existuje efektivnější algoritmus (ideálně s lineární složitostí)???

- *symetricky se dá postupovat podle vrcholů z nichž nevychází žádná hrana*

Acyklický graf - testování

Lema 4

Orientovaný graf G je acyklický právě když DFS průzkum grafu neoznačí žádnou hranu jako zpětnou.

Důkaz

$\Rightarrow$ zpětná hrana (u, v) spojuje vrchol u s jeho předchůdcem v v DFS stromu, tj. uzavírá cyklus

$\Leftarrow$

- nechť v grafu existuje cyklus c, nechť v je první vrchol cyklu c navštívený při DFS průzkumu grafu a nechť (u, v) je hrana cyklu c
- v čase $v.d$ vrcholy cesty c tvoří bílou cestu z v do u co implikuje, že u je následníkem v v DFS stromu
- (u, v) je zpětná hrana

Topologické uspořádání - algoritmus

1. aplikuj DFS na G
2. když průzkum označí některou hranu jako zpětnou, tak graf nemá topologické uspořádání
3. v opačném případě vypiš vrcholy v uspořádání *reverse postorder*, tj. podle značky $.f$ (finishing time) v klesajícím pořadí

Korektnost

potřebujeme dokázat, že pro libovolnou dvojici vrcholů u, v platí

jestliže G obsahuje hranu (u, v), tak $u.f > v.f$

jaké jsou barvy vrcholů u a v při průzkumu hrany (u, v)?

- u je šedivý
- v je

  šedivý nemůže nastat, protože (u, v) by v takovém případě byla zpětnou hranou a graf by nebyl acyklický

  bílý v takovém případě je (u, v) stromová hrana, v je následníkem u v DFS stromu a $u.d < v.d < v.f < u.f$

  černý v takovém případě je průzkum vrcholu v ukončený a průzkum vrcholu u ještě není ukončený a proto $v.f < u.f$

Souvislost v orientovaném grafu

orientovaný graf $G = (V, E)$

- **vrchol v je dosažitelný z vrcholu u**, značíme $u \rightsquigarrow v$, právě když v G existuje orientovaná cesta z u do v
- $Reach(u)$ je množina **všech** vrcholů dosažitelných z u
- vrcholy u a v jsou **silně dosažitelné** (*strongly connected*) právě když u je dosažitelný z v a současně v je dosažitelný z u
- silná dosažitelnost - relace ekvivalence
- **silně souvislá komponenta** grafu je třída ekvivalence relace silné dosažitelnosti, tj. maximální množina vrcholů $C \subseteq V$ taková, že pro každé $u, v \in C$ platí $u \rightsquigarrow v$ a současně $v \rightsquigarrow u$
- graf nazýváme silně souvislý právě když má jedinou silně souvislou komponentu

problém

najít všechny silně souvislé komponenty grafu

Souvislost v neorientovaném grafu

- v neorientovaném grafu jsou pojmy dosažitelnosti a silné dosažitelnost totožné
- pro hledání silně souvislé komponenty grafu můžeme použít BFS nebo DFS
- jednotlivé DFS (BFS) stromy korespondují s komponentami souvislosti
- složitost $\mathcal{O}(V + E)$

Silně souvislé komponenty v orientovaném grafu

výpočet silně souvislé komponenty obsahující daný vrchol u

- najdi množinu $Reach(u)$ všech vrcholů dosažitelných z u aplikací $\mathrm{DFS\_Visit}(G, u)$
- najdi množinu $Reach^{-1}(u)$ všech vrcholů, ze kterých je dosažitelný u
- pro výpočet $Reach^{-1}(u)$ využijeme transponovaný graf[3] $rev(G)$, na který aplikujeme $\mathrm{DFS\_Visit}(rev(G), u)$
- silně souvislá komponenta obsahující u je průnikem $Reach(u) \cap Reach^{-1}(u)$
- časová složitost výpočtu je $\mathcal{O}(V + E)$

[3] transponovaný graf $rev(G) = (V, rev(E))$ je graf G s obrácenou orientací hran, $rev(E) = \{(x, y) \mid (y, x) \in E\}$

Silně souvislé komponenty v orientovaném grafu

výpočet všech silně souvislých komponent grafu

- můžeme *zabalit* popsaný postup (podobně jako je $\mathrm{DFS\_VISIT}(G, u)$ *zabalené* do DFS)
- v nejhorším případě má graf $|V|$ komponent souvislosti a detekce každé z nich má časovou složitost $\mathcal{O}(E)$
- celková časová složitost výpočtu je $\mathcal{O}(V \cdot E)$

- existuje efektivnější algoritmus, optimálně s lineární časovou složitostí ???
- motivace: v čase $\mathcal{O}(V + E)$ dokážeme rozhodnout, zda všechny silně souvislé komponenty grafu jsou jednoduché (obsahují pouze jeden vrchol)

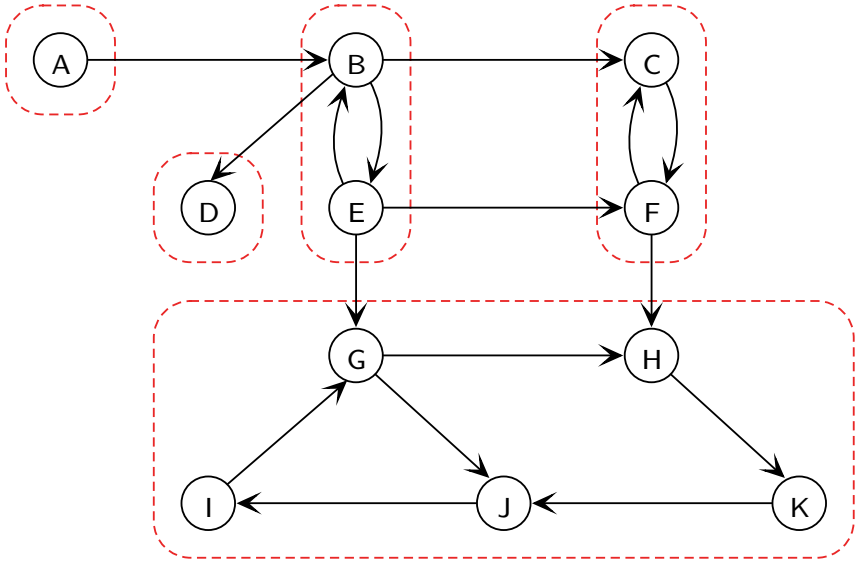
Komponentový graf

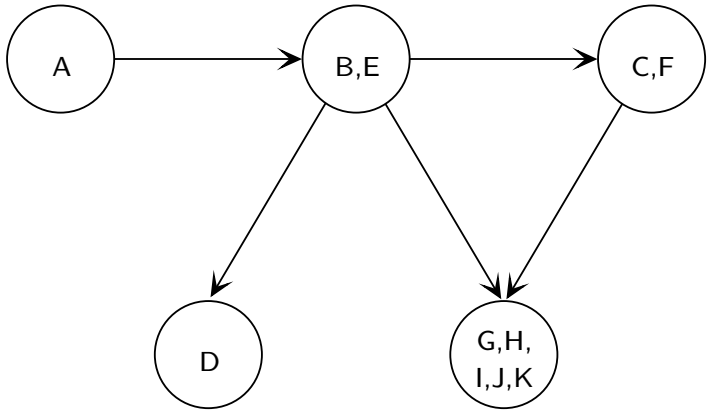
- komponentový graf (*aka graf silně souvislých komponent*) je orientovaný graf, který vznikne kontrakcí každé silně souvislé komponenty grafu do jednoho vrcholu a kontrakcí paralelních hran do jedné hrany, značíme $scc(G)$
- $scc(G)$ je orientovaný acyklický graf

- vrcholy $scc(G)$ můžeme topologicky uspořádat
- listová komponenta == list grafu $scc(G)$
- kořenová komponenta == kořen grafu $scc(G)$

- platí $rev(scc(G)) = scc(rev(G))$

Příklad





kořenová komponenta A
listové komponenty D, CF, GHIJK

Komponentový graf

- **listová komponenta grafu = list grafu** $scc(G)$
- z každého vrcholu u listové komponenty C jsou dosažitelné všechny vrcholy C, tj. DFS průzkum z u navštíví všechny vrcholy z C
- naopak, z vrcholu u není dosažitelný žádný vrchol nepatřící do C (C je listová komponenta!!), tj . DFS průzkum z u navštíví pouze všechny vrcholy z C
- na tomto pozorování můžeme postavit algoritmus pro detekci komponent

Strong_Components(G)

```
1 count ← 0
2 while G není prázdný do
3         count ← count + 1
4         v ← libovolný vrchol z listové komponenty
5         C ← One_Component(v, count)
6         odstraň z grafu G vrcholy z C a hrany vstupující do C od
```

potřebujeme (efektivně) najít vrchol patřící listové komponentě

Komponentový graf

- **potřebujeme** (efektivně) najít vrchol patřící listové komponentě
- **umíme** (efektivně) najít vrchol patřící kořenové komponentě

Věta 5

Vrchol s nejvyšší časovou značkou $.f$ leží v kořenové komponentě.

Důkaz tvrzení je důsledkem následujícího obecnějšího tvrzení

Komponentový graf

Věta 6

Nechť C_1 a C_2 jsou silně souvislé komponenty takové, že z C_1 vede hrana do C_2. Potom největší hodnota $.f$ v komponentě C_1 je větší než největší hodnota $.f$ v komponentě C_2.

Důkaz

mohou nastat dva případy

- v prvním případě navštíví DFS komponentu C_2 jako první; pak ale DFS zůstane v komponentě C_2 dokud ji celou neprozkoumá, teprve pak se dostane do C_1

- v prvním případě navštíví DFS komponentu C_1 jako první; nechť v je vrchol, který byl v C_1 navštíven jako první; DFS opustí vrchol v, až když prozkoumá všechny vrcholy, které jsou z v dosažitelné a které dosud nebyly navštíveny; proto nejprve projde celou komponentu C_2 a pad se teprve naposledy vrátí do v

Algoritmus pro detekci silně souvislých komponent

- vrcholy grafu uspořádej v obráceném postorder pořadí, tj. podle značky $.f$ v klesajícím pořadí
- proveď DFS průzkum z vrcholů v daném pořadí

Rao Kosaraju 1978, Micha Sharir 1981

Kosaraju_Sharir(G)

```
1 foreach U ∈ V do v.color ← white od
2 foreach U ∈ V do if u.color = white then REV_PUSH_DFS(u) fi od
3 foreach U ∈ V do v.color ← white od
4 count ← 0
5 while S ≠ ∅ do
6         u ← S.pop()
7         if u.color = white then count ← count + 1
8                                 LABEL_ONE_DFS(u, count) fi od
```

Rev_Push_DFS(u)

```
1 u.color ← black
2 foreach  hranu (u, v)∈ Rev(G) do
3    if v.color = white then REV_PUSH_DFS(v) fi
4    S.push(u) od
```

Label_One_DFS($u, count$)

```
1 u.color ← black
2 u.label ← count
3 foreach  hranu (u, v)∈ G do
4    if v.color = white then LABEL_ONE_DFS(v, count) fi od
```