Faktorové množiny

- jiný postup řešení základních grafových problémů 1. - 4.

- datová struktura DFU – Disjoint-Find-Union
  (Disjoint Sets, Union-Find Data Structure)
  = rozdělení množiny prvků na vzájemně disjunktní části
  (faktorové množiny) s operacemi
    * pro daný prvek určit číslo jeho faktorové množiny
    * sjednotit dané dvě faktorové množiny

- zde rozdělujeme množinu všech vrcholů grafu,
  faktorové množiny = postupně vytvářené komponenty souvislosti
- na začátku výpočtu máme graf bez hran
- hrany do grafu postupně přidáváme, tím se nám spojují faktory
- vhodná reprezentace grafu: seznam hran

Nejjednodušší implementace (nikoli nejefektivnější)

- každý vrchol grafu si přímo pamatuje číslo své faktorové množiny
- inicializace:
  graf je bez hran, takže každý vrchol je v jiné faktorové množině

```
faktor = [i for i in range(n)]
```

- určení čísla faktorové množiny pro vrchol *v* je snadné: `faktor[v]`
  konstantní časová složitost

- sjednocení faktorových množin, do nichž náležejí vrcholy *v, u*:
  obě přeznačíme na stejnou hodnotu (zvolíme hodnotu jedné z nich)

```
x = faktor[v]
for i in range(n):
    if faktor[i] == x:
        faktor[i] = faktor[u]
```

  časová složitost O(*N*)

Řešení základních grafových problémů 1. - 4.

1. + 2. souvislost grafu, komponenty souvislosti

Do grafu bez hran postupně přidáváme všechny jeho hrany, ve struktuře DFU při tom evidujeme aktuální čísla faktorových množin všech vrcholů (= postupně vytvářené komponenty souvislosti).

Po skončení výpočtu je v poli `faktor` uloženo rozdělení vrcholů do komponent souvislosti grafu.

Výpočet můžeme předčasně ukončit, když počet komponent klesne na 1 (tzn. graf je souvislý, zbývající hrany už ani nemusíme zpracovávat).

```
faktor = [i for i in range(n)]
pocet_komponent = n

pro všechny hrany (v, u):
    if faktor[v] != faktor[u]: # budeme sjednocovat
        pocet_komponent -= 1
        x = faktor[v]
        for i in range(n):
            if faktor[i] == x:
                faktor[i] = faktor[u]

print('Počet komponent:', pocet_komponent)
print('Komponenty souvislosti:', faktor)
```

Časová složitost

- postupně zpracováváme M hran – některé z nich v konstantním čase (pokud má hrana oba své koncové vrcholy ve stejné faktorové množině), ostatní v lineárním čase $O(N)$ potřebným na sjednocení faktorových množin
- sjednocení faktorových množin se ale může provádět celkem nejvýše N-1 krát (pak už jsou všechny vrcholy v jediné faktorové množině)
- celková časová složitost je proto $O(M + N^2) = O(N^2)$

3. existence cyklu v neorientovaném grafu

Provádíme stejný algoritmus (postupné přidávání hran a sjednocování faktorových množin). Pokud narazíme na hranu spojující dva vrcholy z téže faktorové množiny, našli jsme cyklus a výpočet ukončíme. Když žádnou takovou hranu nenajdeme, graf je lesem (příp. i stromem, je-li souvislý).

4. kostra souvislého neorientovaného grafu

Provádíme stejný algoritmus (postupné přidávání hran a sjednocování faktorových množin). Kdykoliv narazíme na hranu spojující dva vrcholy z různých faktorových množin, zařadíme tuto hranu do vytvářené kostry.

Jiná implementace struktury DFU
- převrácené stromy (Disjoint Set Forest)

- každá faktorová množina je reprezentována jedním orientovaným stromem (s neobvyklou orientací hran směrem „nahoru“)
- kořen odkazuje *na sebe*, ostatní uzly vždy na svého *otce*
- implementujeme v poli `faktor` pomocí indexů,
  `faktor[v]` = index otce prvku `v`

- inicializace:
  graf je bez hran, takže každý vrchol je v jiné faktorové množině

```
faktor = [i for i in range(n)]
```

- určení čísla faktorové množiny pro vrchol *v*:
  kořen stromu = reprezentant faktorové množiny

```
while faktor[v] != v: v = faktor[v]
```

- sjednocení faktorových množin, do nichž náležejí vrcholy *v, u*:
  kořen jednoho stromu se stane synem kořene druhého stromu

```
while faktor[v] != v: v = faktor[v]
while faktor[u] != u: u = faktor[u]
faktor[u] = v
```

- časová složitost:
  v nejhorším případě strom degeneruje do seznamu délky *N*
  jeden test náležení nebo sjednocení O(*N*)
  → pro zpracování *M* hran grafu celkem časová složitost O(*N.M*)

Vyrovnávání výšek stromů

- zvýšení efektivity u předchozí varianty:
  při sjednocování stromů se kořen stromu s menší výškou
  stane synem kořene stromu s větší výškou

- k tomu je třeba evidovat v každém kořenu výšku jeho stromu
  (aktualizovat údaj vždy při sjednocení)

- stromy si tak udržují logaritmickou výšku *(důkaz je obtížnější)*
  $\rightarrow$ časová složitost jedné operace v nejhorším případě O(log *N*),
  pro zpracování *M* hran grafu celkem časová složitost O(*M*.log *N*)

Existují další možnosti pro zvýšení efektivity
(založené na průběžném zkracování cest ve stromech)

Další grafové problémy pro orientovaný graf

Topologické uspořádání orientovaného grafu
= očíslování vrcholů tak, že pro každou hranu $i \rightarrow j$ platí $i < j$

- algoritmus založený na postupném odebírání těch vrcholů, do nichž nevede žádná hrana – *topologické třídění*

Zjištění, zda je orientovaný graf acyklický
= neobsahuje žádný orientovaný cyklus

- stejný algoritmus jako v předchozím případě, pokusíme se graf topologicky uspořádat

Topologické třídění

Algoritmus sloužící k nalezení topologického uspořádání orientovaného grafu, tzn. takového očíslování všech vrcholů grafu čísly od 1 do N, aby pro každou hranu $i \rightarrow j$ platilo, že $i < j$.

Pozorování:
- topologické uspořádání nemusí existovat (je-li v grafu cyklus)
- topologické uspořádání nemusí být jednoznačné
  (např. graf se dvěma vrcholy a žádnou hranou má dvě topologická uspořádání)

Naším úkolem tedy je nalézt jedno libovolné topologické upořádání daného orientovaného grafu nebo ohlásit, že žádné topologické uspořádání neexistuje.

Tvrzení:
Orientovaný graf lze topologicky uspořádat, právě když je acyklický.

Důkaz:
→ triviální: kdyby graf obsahoval cyklus (např. $i \rightarrow j \rightarrow k \rightarrow i$), přímo z definice plyne, že nemůže existovat jeho topologické upořádání (muselo by platit $i < j < k < i$, což je spor)
← konstrukčně: ukážeme algoritmus topologického třídění, který libovolný acyklický graf topologicky uspořádá

Příklad použití:
- dány návaznosti výrobních operací (orientované hrany), sestavit výrobní proces (zvolit správné pořadí jednotlivých operací)

Pozorování:
V orientovaném acyklickém grafu existuje vrchol bez předchůdců (tzn. vrchol, do kterého nevede žádná hrana).

Dokážeme sporem – nechť tvrzení neplatí a v orientovaném acyklickém grafu do každého vrcholu nějaká hrana vede.

Zvolíme libovolný vrchol v_1, do něj vede hrana z nějakého vrcholu v_2, do něj vede hrana z vrcholu v_3, atd. V grafu je jen konečně mnoho vrcholů, takže nejvýše po N krocích se takto dostaneme do vrcholu grafu, ve kterém jsem už byli
→ graf obsahuje cyklus, což je spor s předpokladem.

Algoritmus:

- zvolíme libovolný vrchol bez předchůdců
(v acyklickém grafu musí existovat, jinak konec → je tam cyklus)

- zvolenému vrcholu přiřadíme nejbližší volné pořadové číslo v topologickém uspořádání a z grafu tento vrchol vypustíme včetně hran z něj vedoucích

- tím dostaneme graf (opět orientovaný a acyklický, když původní graf byl orientovaný a acyklický!), který má o jeden vrchol méně, a na něj aplikujeme stejný postup

- pokud takto postupně očíslujeme a vypustíme všechny vrcholy grafu, máme topologické uspořádání

Realizace:
- graf uložen pomocí seznamů následníků (případně matice sousednosti)
- pomocné pole P indexované čísly vrcholů od 1 do *N*,
kde je pro každý vrchol uložen **počet jeho předchůdců**
(vrchol bez předchůdců tam tedy má 0, vypuštěnému vrcholu uložíme do pole P pro odlišení nějakou zvláštní hodnotu, např. -1)

Časová složitost: $O(N^2)$
následující postup se opakuje nejvýše *N*-krát:

1. vybrat vrchol bez předchůdce – průchod polem P $O(N)$
2. vrchol vyřadit – vložení „-1“ do pole P $O(1)$
3. vyřadit hrany z něj vedoucí – projít jeho seznam následníků a snížit jim o 1 údaj v poli P $O(N)$

Zlepšení realizace:
- přidáme seznam vrcholů Q, které nemají předchůdce
  (tzn. mají v poli P hodnotu 0)
- implementujeme ho například zásobníkem

Časová složitost v této reprezentaci: O(*N* + *M*)
následující postup se opakuje nejvýše *N*-krát:

1. vybrat vrchol bez předchůdce – ze seznamu Q O(1)
2. vrchol vyřadit – vložení „-1“ do P a vyřazení z Q O(1)
3. vyřadit hrany z něj vedoucí – projít seznam následníků a snížit jim o 1 údaj v poli P (když přitom hodnota následníka uložená v P klesne na 0, zařadíme ho do Q)
   dohromady ve všech krocích výpočtu O(*M*)

Jiný algoritmus nalezení topologického uspořádání grafu

- prohledáváme graf do hloubky (DFS)
- pořadí, v němž uzavíráme (tzn. opouštíme) vrcholy, je přesně opačné oproti topologickému uspořádání grafu
- *proč:* vrchol uzavíráme ve chvíli, když už jsou uzavřeni všichni jeho následníci
- *problém:* takto prohledáme a uspořádáme pouze vrcholy, které jsou dostupné ze zvoleného výchozího vrcholu
- *řešení:* do grafu přidáme navíc jeden pomocný vrchol a hrany vedoucí z něj do všech ostatních vrcholů, v tomto vrcholu začneme průchod grafem
- *časová složitost* je dána složitostí procházení grafu do hloubky – tedy $O(N^2)$ nebo $O(N+M)$ podle zvolené reprezentace grafu

Ohodnocené grafy

- existuje hranové a vrcholové ohodnocení grafu
- my se omezíme na hranové ohodnocení
  (častější – délka silnice, doba jízdy, cena jízdného, …)

Základní řešené problémy

- minimální kostra v ohodnoceném souvislém grafu
  (minimální součet ohodnocení hran zařazených do kostry)
- nejkratší cesta v ohodnoceném grafu
  (minimální součet ohodnocení hran tvořících cestu z vrcholu *A* do *B*)
- vzdálenost vrcholů
  (délka nejkratší cesty mezi danými dvěma vrcholy)

Minimální kostra grafu
(souvislého, neorientovaného, hranově ohodnoceného)

Použijeme stejný algoritmus, jako při hledání kostry
v neohodnoceném grafu pomocí faktorových množin *(bylo dříve)*,
jenom hrany nejprve seřadíme vzestupně podle jejich ohodnocení
a procházíme je pak od nejkratší po nejdelší
(tzn. hladový algoritmus)

Kruskalův algoritmus (1956)

Časová složitost:
seřazení hran $O(M.\log M) = O(N^2.\log N)$
hledání kostry (při jednoduché implementaci faktorových množin)
- čas $O(N^2)$

→ celkem $O(N^2.\log N)$

Jiné možnosti nalezení minimální kostry:

Jarníkův algoritmus (1930), též Primův (1957)
- libovolný výchozí vrchol, z něj se rozrůstá strom
- vždy vybíráme hranu s nejmenším ohodnocením vedoucí z již sestrojené části kostry do zbytku grafu
- časová složitost O(*N.M*) … *N* kroků, každý má složitost O(*M*)

Borůvkův algoritmus (1926)
- postupně spojuje vytvářené stromy
- na začátku máme jednovrcholové stromy bez hran
- pro každý strom vybereme hranu s nejmenším ohodnocením, která z něj vede ven, všechny tyto hrany přidáme do kostry, tím se stávající stromy propojí do větších stromů
- uvedený postup opakujeme, až zbyde jediný strom
- časová složitost O(*M*.log *N*) … log *N* kroků, každý má složitost O(*M*)

Nejkratší cesta v ohodnoceném grafu

nejkratší cesta z vrcholu *u* do vrcholu *v*
= taková cesta z *u* do *v*, na níž je součet ohodnocení všech hran co nejmenší

Dijkstrův algoritmus – podobný postup jako procházení do šířky v neohodnoceném grafu, vlna se ale nešíří z výchozího vrcholu *u* rovnoměrně do všech stran, nýbrž je vhodně „deformována“ pomocí ohodnocení

Nutná podmínka: nezáporné ohodnocení hran

Dijkstrův algoritmus (1959)

- lze použít buď pro nalezení nejkratší vzdálenosti z výchozího vrcholu *u* do cílového vrcholu *v*, nebo pro určení nejkratších vzdáleností z výchozího vrcholu *u* do všech dostupných vrcholů grafu (oboje při stejné asymptotické časové složitosti)

- Dijkstrův algoritmus určí pouze délku nejkratší cesty z výchozího do cílového vrcholu, následovat musí ještě *zpětný chod*, při kterém určíme vlastní cestu (podobně jako při prohledávání do šířky u neohodnocených grafů)

- pokud existuje více různých cest téže minimální délky, algoritmus určí jednu libovolnou z nich

- stejný postup lze použít pro neorientované i pro orientované grafy

Reprezentace grafu
seznamy následníků nebo matice vzdáleností $d_{i,j}$
- potřebujeme umět rychle určit, kam vedou z daného vrcholu hrany

Další uložené údaje
u každého vrcholu si potřebujeme evidovat přiřazenou hodnotu
h_i = délka nejkratší cesty z výchozího vrcholu *u* do vrcholu *i*,
jakou jsme zatím našli
nejjednodušší realizace: pole H indexované čísly vrcholů

o této hodnotě si dále pamatujeme, zda je ***dočasná*** (tzn. možná ještě půjde zlepšit, tj. snížit) nebo ***trvalá*** (je to už definitivní hodnota)
realizace: druhé pole příznaků (hodnoty boolean)
nebo dočasné/trvalé hodnoty v poli H rozlišíme podle znaménka

Inicializace dat

$h_u = 0$ (výchozí vrchol)

h_i = „+∞“ pro všechna $i \neq u$

hodnoty všech vrcholů jsou dočasné

Algoritmus

dokud má cílový vrchol dočasnou hodnotu

1. nechť *j* je vrchol s nejmenší dočasnou hodnotou
2. hodnotu vrcholu *j* prohlásíme za trvalou
3. všem následníkům vrcholu *j*, které mají ještě dočasnou hodnotu, přepočítáme jejich hodnotu podle vztahu

$$h_i = \min (h_i, h_j + d_{j,i})$$

(tzn. snížíme dočasné hodnoty všech vrcholů grafu, které je možné snížit díky cestě vedoucí přes vrchol *j*)

Pokud chceme znát vzdálenosti z *u* do všech dostupných vrcholů:

dokud existuje dostupný vrchol s dočasnou hodnotou

Správnost

I. konečnost: při každé obrátce cyklu se jeden vrchol stane trvalým
$\rightarrow$ nejvýše po *N* krocích výpočet skončí

II. korektnost výsledku – proč můžeme minimální dočasnou hodnotu vrcholu prohlásit za trvalou:

- dočasná hodnota představuje vždy délku nejkratší cesty vedoucí přes trvale ohodnocené vrcholy (plyne z přepočítávání hodnot v bodě 3.)

- minimální dočasná hodnota už nepůjde zlepšit, neboť ostatní dočasné vrcholy mají hodnotu větší a hrany grafu jsou ohodnoceny nezáporně

Časová složitost

cyklus se opakuje nejvýše *N*-krát, v něm jsou vnořené kroky:

1. nalezení vrcholu s minimální dočasnou hodnotu O(*N*)
2. označení této hodnoty za trvalou O(1)
3. obejdeme nejvýše *N* následníků, každý přepočítáme O(*N*)

$\rightarrow$ celková časová složitost algoritmu $O(N^2)$

Implementace algoritmu pomocí binární haldy

- pro uchovávání dočasných hodnot vrcholů grafu můžeme použít haldu
- na začátku je v ní všech N vrcholů, v každém kroku odebereme jeden vrchol (ten s nejmenší hodnotou)
- vrcholy s upravenou dočasnou hodnotou přemístíme v haldě na správné místo,
  to se provede nejvýše M-krát dohromady ve všech krocích
- každé odebrání i přemístění vrcholu v haldě má složitost $O(\log N)$

$\rightarrow$ celková časová složitost algoritmu je v tomto případě $O((N+M).\log N)$, což je pro „řídké“ grafy lepší

Určení cesty

U každého vrcholu *i* si budeme pamatovat číslo vrcholu p_i, který je jeho předchůdcem na minimální cestě z výchozího vrcholu.
Realizace: další pole P indexované čísly vrcholů

Inicializace: $p_u = 0$ (výchozí vrchol nemá předchůdce)

Stanovení hodnot předchůdců: vždy při snížení hodnoty h_i na hodnotu $h_j + d_{j,i}$ definujeme rovněž novou hodnotu $p_i = j$

Po skončení Dijkstrova algoritmu následuje zpětný chod pomocí hodnot *p* stejně, jako v případě neohodnocených grafů.

Časovou složitost celého algoritmu to nijak neovlivní,
zpětný chod má časovou složitost pouze O(*N*).