6. nejkratší vzdálenosti v grafu

Provedeme průchod grafem do šířky z výchozího vrcholu *s* (má hodnotu 0, zatímco nenavštívené vrcholy označíme hodnotou -1). Každému navštívenému vrcholu při tom přiřadíme hodnotu o 1 větší, než je hodnota vrcholu, ze kterého právě přicházíme. Takto projdeme všechny dostupné vrcholy (komponentu souvislosti). Jejich výsledné hodnoty udávají nejkratší vzdálenosti jednotlivých vrcholů od výchozího vrcholu.

Časová složitost algoritmu je stejná jako při určování souvislosti grafu
– tedy $O(N^2)$ nebo $O(N+M)$ podle zvolené reprezentace grafu.

```
navstiven = [-1] * n
navstiven[s] = 0
fronta = [s]

while len(fronta) > 0:     # fronta není prázdná
    v = fronta.pop(0)
    pro všechny hrany (v, u):
        if navstiven[u] == -1:
            navstiven[u] = navstiven[v] + 1
            fronta.append(u)

print(navstiven)
# vzdálenosti všech vrcholů od počátečního vrcholu
```

Modifikace: vzdálenost daných dvou vrcholů v grafu

Pokud nepotřebujeme znát vzdálenosti všech vrcholů od výchozího vrcholu, ale jen vzdálenost daných dvou vrcholů *A*, *B*, spustíme průchod do šířky z vrcholu *A* (algoritmus viz výše) a můžeme ho předčasně ukončit ve chvíli, kdy navštívíme a ohodnotíme vrchol *B*.

Asymptotická časová složitost algoritmu v nejhorším případě se touto optimalizací nezmění (do vrcholu *B* se můžeme dostat až na konci prohledávání), v průměru se ale výpočet urychlí (ne vždy je třeba projít celou komponentu souvislosti).

Změna v kódu: místo dosavadního příkazu

```
while len(fronta) > 0:      # fronta není prázdná
```

zde bude

```
while navstiven[B] == -1:   # nepřišli jsme do B
```

7. nejkratší cesta v grafu

Hledáme nejkratší cestu v grafu z vrcholu *A* do vrcholu *B*.
Výpočet probíhá ve dvou po sobě jdoucích fázích:

a) ***algoritmus vlny*** = průchod do šířky s určením nejkratší vzdálenosti vrcholů *A*, *B* – viz předchozí bod 6.
Navíc si ke každému vrcholu při vložení do fronty poznamenáme číslo jeho předchůdce na nejkratší cestě (odkud jsme do tohoto vrcholu poprvé přišli).

Realizace: k příkazům

```
navstiven[u] = navstiven[v] + 1
fronta.append(u)
```

doplnit příkaz

```
predchudce[u] = v
```

```
navstiven = [-1] * n
navstiven[A] = 0
predchudce = [0] * n
fronta = [A]

while navstiven[B] == -1:  # nepřišli jsme do B
    v = fronta.pop(0)
    pro všechny hrany (v, u):
        if navstiven[u] == -1:
            navstiven[u] = navstiven[v] + 1
            predchudce[u] = v
            fronta.append(u)
```

b) ***zpětný chod*** = rekonstrukce cesty odzadu pomocí zaznamenaných předchůdců

```
cesta = [B]
v = B
while v != A:
    v = predchudce[v]
    cesta.append(v)

print(cesta)    # cesta pozpátku, tedy od B do A
```

Časová složitost zpětného chodu je O(*N*), takže celková složitost algoritmu nalezení cesty je stejná jako při procházení grafu – tedy O(N^2) nebo O(*N*+*M*) podle zvolené reprezentace grafu.

Poznámky:

Nejkratší cesta v grafu nemusí být určena jednoznačně. Pokud existuje více různých cest téže minimální délky, algoritmus určí jednu z nich.

Stejný postup lze použít pro neorientované i pro orientované grafy. Pokud je graf orientovaný, algoritmus vlny postupuje po hranách ve směru jejich orientace.
Zpětný chod pak sice vede proti směru orientace hran, ale díky zaznamenaným předchůdcům to nevadí (při zpětném chodu již nepoužíváme vlastní reprezentaci grafu v paměti).

Faktorové množiny

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

- datová struktura DFU – Disjoint-Find-Union (Disjoint Sets)
  = 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 sovislosti:', 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.

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 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 prohledávání 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)

Metody ukládání a vyhledávání dat – shrnutí

data = záznamy s klíčem, podle kterého vyhledáváme

- pole: vyhledávání O(*N*), vkládání O(1)
- uspořádané pole: binární vyhledávání O(log *N*), vkládání O(*N*)
- lineární spojový seznam: vyhledávání O(*N*), vkládání O(1)
  jako v poli
- uspořádaný LSS: vyhledávání O(*N*), vkládání O(N)
  průchod seznamem lze předčasně ukončit

- binární vyhledávací strom: všechny operace v průměru O(log *N*),
  v nejhorším případě O(*N*)
- vyvážený binární vyhledávací strom: všechny operace O(log *N*)
- vícecestný vyhledávací strom

Metody ukládání a vyhledávání dat – shrnutí

data = záznamy s klíčem, podle kterého vyhledáváme

- přímé indexování pole klíčem: všechny operace O(1),
klíčem musí být celé číslo z předem známého dost malého rozsahu

- hešování: transformace klíče do předem známého menšího rozsahu celých čísel, jimiž lze indexovat pole

Hešování

klíč $\rightarrow$ **hešovací funkce** $\rightarrow$ index v poli (**hešovací tabulka**)

rozsah možných klíčů bývá výrazně větší než rozsah přípustných indexů – např. klíč je rodné číslo člověka, pro uložení několika set lidí stačí pole indexované 0..999

$\Rightarrow$ hešovací funkce nebývá prostá $\Rightarrow$ vznikají **kolize**
(hešovací funkce více záznamům přidělí stejný index v poli)

1. Minimalizace počtu kolizí:
- dobrá rovnoměrně rozptylující hešovací funkce
- dostatečný rozsah indexů vzhledem k počtu ukládaných záznamů
(zaplněnost hešovací tabulky nejvýše do 90%)

2. Řešení kolizí: oblast přetečení nebo sekundární transformace klíče

Řešení kolizí pomocí oblasti přetečení:

- v hešovací tabulce je na indexu *x* uložen seznam všech záznamů, jejichž klíč hešovací funkce zobrazila právě na index *x*

- při vyhledávání záznamu podle známého klíče se na tento klíč použije hešovací funkce, tím získáme index do hešovací tabulky a na tomto indexu najdeme seznam uložených záznamů; tento seznam sekvenčně projdeme a podle známého klíče v něm vyhledáme náš záznam