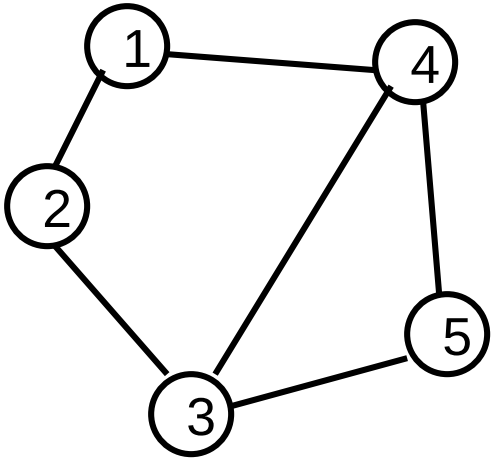
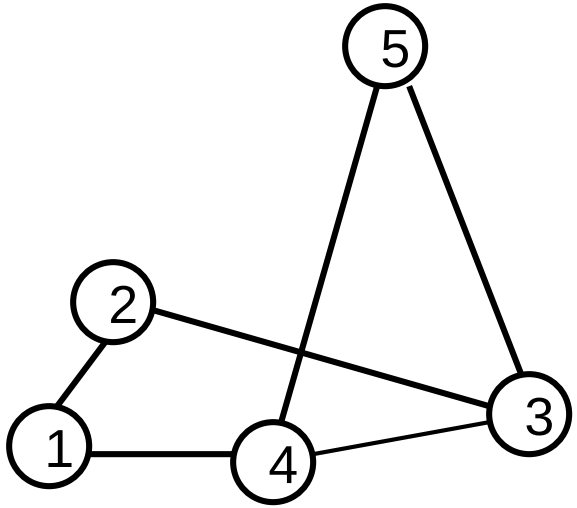
Grafy

- vrcholy (počet značíme *N*), hrany (počet značíme *M*)
- každá hrana spojuje právě dva vrcholy
- předpokládáme, že vrcholy jsou očíslovány od 1 do *N*
  (příp. v programech v Pythonu od 0 do N-1)

rozlišujte *graf* a *nakreslení grafu*
dvě různá nakreslení téhož grafu:





Základní pojmy

graf neorientovaný – orientovaný – obecný (smíšený)
graf neohodnocený – ohodnocený (hranově, vrcholově)
(my se omezíme jen na častěji používané hranové ohodnocení)

multigraf – obsahuje *multihrany* (dvě hrany spojující stejnou dvojici vrcholů, v případě orientovaných grafů i se stejnou orientací)
smyčka – hrana, jejíž počáteční a koncový vrchol jsou shodné
(my se omezíme na obvyklé grafy bez smyček a bez multihran)

Počet hran v grafu
- minimálně 0 (graf neobsahuje žádnou hranu)
- maximálně $N.(N-1)/2$ – úplný neorientovaný graf
- v případě orientovaného grafu maximálně $N.(N-1)$
  (hrana $u \rightarrow v$ a hrana $v \rightarrow u$ jsou různé hrany)

Cesty v grafu
= jak dojít po hranách z vrcholu *x* do vrcholu *y*
(v případě orientovaného grafu s dodržením orientace hran)

sled z vrcholu *x* do vrcholu *y* (libovolná opakování vrcholů i hran)
tah z vrcholu *x* do vrcholu *y* (hrany se neopakují, vrcholy mohou)
cesta z vrcholu *x* do vrcholu *y* (neopakují se na ni ani vrcholy)

vzdálenost dvojice vrcholů = délka nejkratší cesty mezi nimi
(minimální co do počtu vrcholů, resp. v případě ohodnoceného grafu minimalizujeme součet ohodnocení hran na cestě)

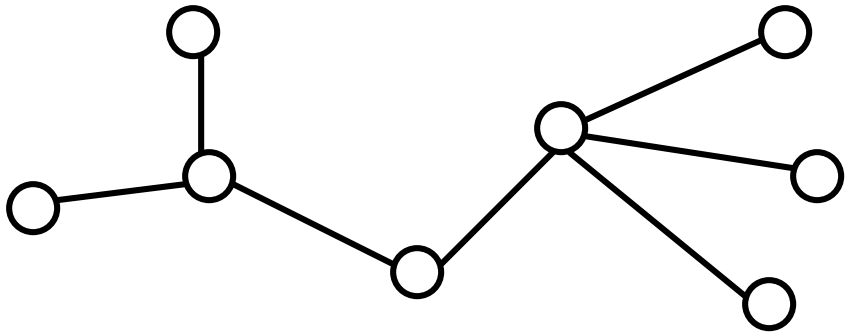
Jsou-li počáteční a koncový vrchol shodný:
uzavřený sled – uzavřený tah – uzavřená cesta = ***kružnice, cyklus***

Souvislost grafu

souvislý graf = mezi každými dvěma vrcholy existuje cesta
komponenta souvislosti = maximální souvislá část grafu

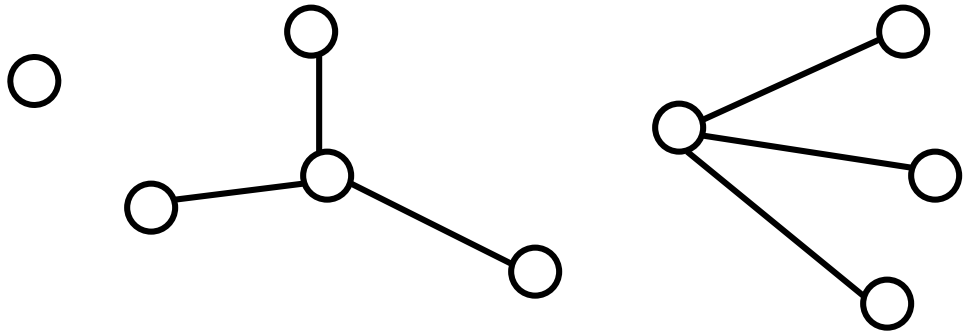
U orientovaných grafů je to složitější:
slabě souvislý = symetrizace grafu je souvislým grafem
slabé komponenty
silně souvislý = každé dva vrcholy jsou oboustranně dosažitelné
silné komponenty
(*touto věcí se nebudeme zabývat*)

Strom = souvislý neorientovaný graf bez cyklů
- souvislý graf s minimálním počtem hran
- mezi každými dvěma vrcholy existuje právě jedna cesta
- každý strom s *N* vrcholy má přesně *N*-1 hran



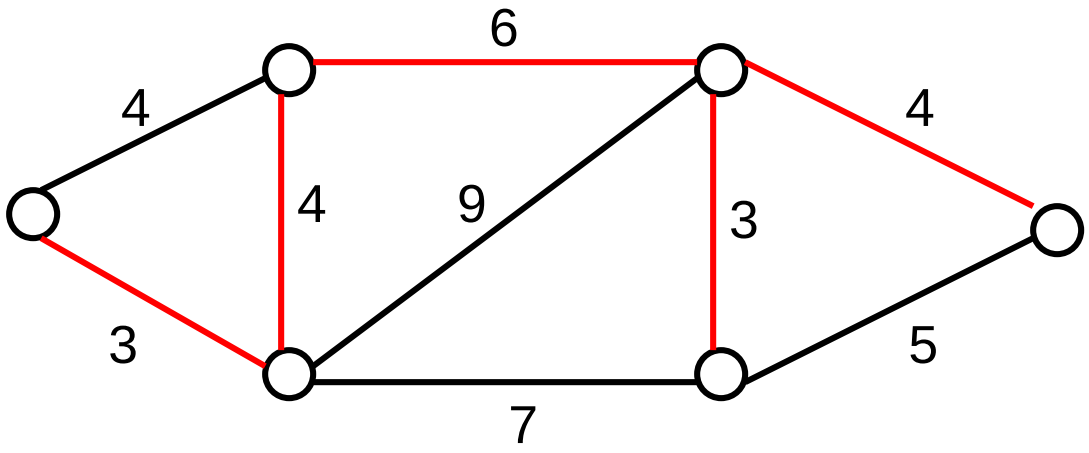
Poznámka:
Jeden libovolný vrchol můžeme prohlásit za kořen a všem hranám přiřadit orientaci směrem od kořene k listům
→ ***zakořeněný strom, orientovaný strom***
(těm jsme dosud říkali „stromy“).

Les = neorientovaný graf bez cyklů
- jeho komponenty souvislosti jsou stromy
- mezi každými dvěma vrcholy existuje nejvýše jedna cesta
- les s *N* vrcholy má nejméně 0 a nejvýše *N*-1 hran
  (počet hran = *N* – počet komponent)



Kostra grafu (souvislého neorientovaného)
= souvislý podgraf obsahující všechny vrcholy grafu a co nejméně hran (tzn. vždy přesně *N*-1 hran)
→ kostra je stromem

Hranově ohodnocené grafy: ***minimální kostra***
= ta z koster, v níž je součet ohodnocení hran co nejmenší
- nemusí být jednoznačná



Bipartitní graf (neorientovaný)
= všechny vrcholy grafu lze rozdělit do dvou skupin tak, že každá hrana grafu bude spojovat vždy dvojici vrcholů náležejících do různých skupin

Neboli: každý vrchol grafu obarvíme červeně nebo modře tak, aby každá hrana grafu spojovala červený vrchol s modrým.

Graf je bipartitní, právě když neobsahuje žádný cyklus liché délky.

Acyklický orientovaný graf
= neobsahuje žádný orientovaný cyklus, tj. orientovanou kružnici
(případnou smyčku nepokládáme za cyklus)

tzn. všechny silné komponenty grafu jsou jednobodové

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

Orientovaný graf je acyklický, právě když má topologické uspořádání.

Reprezentace grafu v programu

1. Matice sousednosti
= čtvercová matice velikosti N x N
$a[i][j] = 0 / 1$ právě když neexistuje / existuje hrana (i, j)

neorientovaný graf – symetrická matice
orientovaný graf – obecně není symetrická
ohodnocený graf – $a[i][j]$ = ohodnocení hrany (i, j)
$\rightarrow$ **matice délek hran**
(zvláštní pro ten účel vyčleněná hodnota potom kóduje,
že hrana neexistuje)

určení všech sousedů daného vrcholu: potřebujeme N operací
(projít řádek matice)

Vhodné použití:
malé grafy s velkým počtem hran

Kdy nevyhovuje:
graf s velkým počtem vrcholů a relativně málo hranami
(např. silniční síť nějakého regionu)
- matice je rozsáhlá (paměťově náročná) a přitom v ní jsou uloženy skoro samé nuly
- také při hledání sousedů daného vrcholu se musí přes všechny ty nuly procházet (pomalý výpočet)

Řešení:
použít jinou reprezentaci grafu

2. Seznamy následníků

- u každého vrcholu je uložen seznam čísel vrcholů,
  do nichž odsud vede hrana

- primárně určeno pro orientované grafy

- neorientovaná hrana se nahradí dvojicí orientovaných hran
  (vedoucích tam a zpět)

- ohodnocený graf:
  s číslem cílového vrcholu se ukládá i ohodnocení hrany

Realizace:

a) pole *N* seznamů (standardní seznamy v Pythonu nebo LSS)
- do nich ukládáme čísla následníků jednotlivých vrcholů
→ alokace paměti přesně potřebné délky

b) matice *N* x *N*-1 (z každého vrcholu může vést až *N*-1 hran)
+ jednorozměrné pole délky *N* (udává počet hran vycházejících z jednotlivých vrcholů)
→ oproti matici sousednosti se paměť neušetří, ale čas na procházení všech následníků ano

c) matice *N* x *R*
(když víme, že z každého vrcholu vede maximálně *R* hran)
např. *N* = 1000 měst, z každého vede nejvýše *R* = 10 silnic,
pak stačí matice 1000 x 10 místo dosavadní 1000 x 999
→ výrazná paměťová úspora

d) nemáme omezení *R* platné pro každý vrchol, ale víme, že celkově je v grafu nejvýše *M* hran

- princip *„vzájemné solidarity“* vrcholů … čísla následníků všech vrcholů se ukládají do jednoho společného pole E délky *M*

- seznamy následníků jsou umístěny v poli E jeden za druhým, v pomocném poli V délky *N* evidujeme, kde který seznam začíná: následníci vrcholu *u* jsou zapsáni v poli E na pozicích od indexu V[*u*] do indexu V[*u*+1]-1

3. Seznam hran

- jednorozměrné pole nebo seznam délky *M*
- každý záznam odpovídá jedné hraně grafu – jsou v něm uložena čísla koncových vrcholů a případně také ohodnocení hrany
- lze použít pro orientované i neorientované grafy
(v případě orientovaného grafu rozlišujeme odkud – kam vede hrana)

Vhodné použití: algoritmy založené na postupném zpracování všech hran grafu (např. algoritmy využívající strukturu DFU).

Nevhodné: pro algoritmy vyžadující určení všech sousedů daného vrcholu – časová složitost nalezení sousedů je O(*M*).

4. Matice incidence

= matice velikosti *M* x *N* (hrany x vrcholy)

neorientovaný graf – stačí matice typu boolean:

A [*h*] [*u*] = 0 právě když vrchol *u* neleží na hraně *h*

A [*h*] [*u*] = 1 právě když vrchol *u* leží na hraně *h*

orientovaný graf – rozlišíme orientaci hran:

A [*h*] [*u*] = +1 právě když hrana *h* vede z vrcholu *u*

A [*h*] [*u*] = -1 právě když hrana *h* vede do vrcholu *u*

ohodnocený graf: A [*h*] [*u*] přímo udává ohodnocení hrany

Použitelné jen pro velmi řídké grafy (méně hran než vrcholů), jinak paměťově i časově neefektivní.
Využití spíše v teorii grafů (např. pro některé důkazy).

5. Dynamická reprezentace

- je možná, ale není typická

- vrchol grafu je dynamicky alokovaný záznam (objekt),
  v něm je uložen seznam odkazů na sousední vrcholy

- datově stejné jako obecná reprezentace stromu,
  odpovídá orientovanému grafu

- neorientovaná hrana se reprezentuje dvojicí orientovaných hran
  vedoucích oběma směry

- v případě ohodnoceného grafu se do záznamu přidá položka,
  která představuje ohodnocení hrany

- na rozdíl od stromu nemá graf jeden vstupní bod (kořen), navíc nemusí být souvislý (resp. silně souvislý)

→ je třeba nějaká tabulka odkazů na všechny vrcholy (např. v podobě spojového seznamu, do něhož propojíme všechny vrcholy grafu)

- pro realizaci algoritmu průchodu potřebujeme mít v každém vrcholu navíc jednu položku typu boolean indikující, zda byl při průchodu tento vrchol již navštíven (pro detekci zacyklení)

Základní grafové problémy
(neorientovaný neohodnocený graf)

1. Je graf souvislý?
   Určit, které vrcholy grafy jsou dostupné z daného vrcholu
   (tzn. určit komponentu souvislosti)

2. Určit komponenty souvislosti grafu

3. Obsahuje graf nějaký cyklus?
   Jinými slovy: Je graf stromem nebo lesem?

4. Určit kostru souvislého grafu (jednu libovolnou)

5. Určit, zda je graf bipartitní

Problémy 1, 2, 3, 4, 5 lze řešit

procházením grafu do hloubky nebo do šířky

– obě metody jsou rovnocenné (stejně efektivní).

6. Určit délku nejkratší cesty mezi danými dvěma vrcholy
   Pro daný vrchol určit vzdálenosti všech ostatních vrcholů

7. Nalézt nejkratší cestu mezi danými dvěma vrcholy
   (jednu libovolnou)

Problémy 6 a 7 řešíme ***procházením grafu do šířky***
(v případě problému 7 ještě doplněno navíc o *zpětný chod*)

Problémy 6 a 7 mají smysl i pro orientované grafy
– řešíme je stejně jako u neorientovaných grafů, jenom musíme po hranách postupovat pouze podle jejich orientace
(při zpětném chodu proti orientaci, jak uvidíme později).

Procházení (prohledávání) grafu do hloubky
(DFS – depth-first search)

- stejný postup jako u binárních a obecných stromů

- změna: musíme značit navštívené vrcholy a kontrolovat, zda do vrcholu nepřicházíme opakovaně (pokud ano, nejdeme tam)

- každému vrcholu proto přiřadíme značku, zda už byl navštíven; na začátku mají všechny vrcholy značku **`False`**, pouze výchozí vrchol *s* má značku **`True`**

- implementace prohledávání grafu do hloubky: stejně jako u stromů buď rekurzivní funkce, nebo zásobník + cyklus

Průchod s počátečním vrcholem *s* realizovaný voláním rekurzivní funkce `Pruchod()`

```
navstiven = [False] * n

def Pruchod(v):
    navstiven[v] = True
    zpracuj_vrchol(v)   # provede požadovanou akci
    pro všechny hrany (v, u):
        if not navstiven[u]:
            Pruchod(u)

Pruchod(s)
```

Průchod s počátečním vrcholem *s* pomocí zásobníku
(zásobník je implementován seznamem `zasob`)

```
navstiven = [False] * n
navstiven[s] = True
zasob = [s]

while len(zasob) > 0:    # zásobník není prázdný
    v = zasob.pop()
    zpracuj_vrchol(v)    # provede požadovanou akci
    pro všechny hrany (v, u):
        if not navstiven[u]:
            navstiven[u] = True
            zasob.append(u)
```

Poznámka 1

Obě uvedené varianty implementace algoritmu DFS správně prohledají celý graf, ale pořadí navštívených vrcholů je jiné. Proto je jiný také strom (seznam hran), který tomuto prohledávání odpovídá.

Pro řešení mnoha grafových problémů (viz např. problémy 1. - 5.) to nevadí.

Poznámka 2 – pouze pro znalce

Strom získaný rekurzivní implementací algoritmu DFS má některé zajímavé vlastnosti, které druhá implementace nemá (zejména vzhledem k němu neexistují příčné hrany).
Některá běžná rozšíření DFS s nerekurzivním algoritmem nefungují (např. algoritmus na hledání mostů).

Časová složitost

V obou variantách implementace algoritmus jednou zpracuje každý dostupný vrchol, tzn. provede maximálně *N* kroků výpočtu.

Při zpracování každého vrcholu *v* musíme projít všechny jeho sousední vrcholy:

- při reprezentaci grafu pomocí matice sousednosti má průchod přes sousedy vrcholu *v* časovou složitost O(*N*)
  → celková časová složitost algoritmu je proto O(N^2)

- při reprezentaci grafu pomocí seznamů následníků budeme během všech *N* kroků výpočtu každou z *M* hran grafu procházet dvakrát (jednou v každém směru)
  → celková časová složitost algoritmu je proto O(*N*+*M*)

Procházení (prohledávání) grafu do šířky
(BFS – breadth-first search)

- opět stejný postup jako u binárních a obecných stromů

- stejně jako u prohledávání do hloubky musíme značit navštívené vrcholy a kontrolovat, zda do vrcholu nepřicházíme opakovaně

- úplně stejný algoritmus jako prohledávání do hloubky, jenom se místo zásobníku použije fronta *(to už známe ze stromů)*

- proto i stejná časová složitost $O(N^2)$ resp. $O(N+M)$

Průchod s počátečním vrcholem *s* pomocí fronty
(fronta je implementována seznamem `fronta`)

```
navstiven = [False] * n
navstiven[s] = True
fronta = [s]

while len(fronta) > 0:  # fronta není prázdná
    v = fronta.pop(0)
    zpracuj_vrchol(v)   # provede požadovanou akci
    pro všechny hrany (v, u):
        if not navstiven[u]:
            navstiven[u] = True
            fronta.append(u)
```

Řešení základních grafových problémů 1. - 7.

1. souvislost neorientovaného grafu

Provedeme průchod grafem do hloubky nebo do šířky,
při zpracování každého vrcholu zvýšíme globální počítadlo navštívených vrcholů o 1. Nakonec otestujeme, zda má toto počítadlo hodnotu *N*.

určení dostupných vrcholů (komponenty souvislosti)

Postup viz výše, všechny vrcholy dostupné z počátečního vrcholu *s* budou mít po skončení průchodu značku **`True`**
(v seznamu `navstiven`).

2. všechny komponenty souvislosti neorientovaného grafu

Opakovaně provádíme průchod grafem (do hloubky nebo do šířky) vždy z nějakého dosud nenavštíveného vrcholu, dokud nebudou navštíveny všechny vrcholy grafu.
Při tom počítáme komponenty (proměnná `komponenta`).

Do seznamu příznaků `navstiven` se budou ukládat čísla 1, 2, … značící jednotlivé komponenty souvislosti.
Hodnota 0 označuje dosud nenavštívený vrchol.

Po skončení výpočtu známe počet komponent souvislosti (proměnná `komponenta`) a rozdělení vrcholů do jednotlivých komponent (seznam `navstiven`).

```
navstiven = [0] * n
komponenta = 0
pocet_navstivenych = 0

def Pruchod(v):
    global pocet_navstivenych
    pocet_navstivenych += 1
    pro vsechny hrany (v, u):
        if navstiven[u] == 0:
            navstiven[u] = navstiven[v]
            Pruchod(u)

while pocet_navstivenych < n:
    komponenta += 1
    s = nenavstiveny_vrchol()
    navstiven[s] = komponenta
    Pruchod(s)
```

Časová složitost

Každý z N vrcholů bude navštíven právě jednou, bez ohledu na počet komponent souvislosti grafu. Výsledná časová složitost algoritmu je proto stejná, jako v případě testu souvislosti grafu
– tedy $O(N^2)$ nebo $O(N+M)$ podle zvolené reprezentace grafu.

3. existence cyklu v neorientovaném grafu

Provedeme průchod celým grafem do hloubky nebo do šířky (přes všechny komponenty souvislosti jako v předchozím bodu 2.). Pokud při zpracování některého vrcholu narazíme na hranu vedoucí do již dříve navštíveného vrcholu, našli jsme cyklus a výpočet ukončíme. Když žádnou takovou hranu nenajdeme, graf je lesem (příp. i stromem, je-li souvislý).

Problém: Při zpracování vrcholu *w* narazíme také na tu hranu, po níž jsme do vrcholu *w* přišli → „falešný cyklus“.

Řešení: Při procházení grafem si budeme u navštívených vrcholů pamatovat, odkud jsme do nich přišli. Hrany vedoucí zpět do předchůdce nebudeme používat.

4. kostra souvislého neorientovaného grafu

Provedeme průchod grafem do hloubky nebo do šířky. Kdykoliv při zpracování některého vrcholu narazíme na hranu vedoucí do zatím nenavštíveného vrcholu, zařadíme tuto hranu do vytvářené kostry.

5. bipartitnost grafu

Provedeme průchod celým grafem do hloubky nebo do šířky (přes všechny komponenty souvislosti jako v bodech 2. a 3.). Nerozlišujeme přitom čísla komponent, ale místo toho navštívené vrcholy značíme střídavě čísly 1 a -1. Tyto hodnoty v seznamu `navstiven` určují rozdělení vrcholů do výsledných dvou skupin. Pokud narazíme na hranu spojující dva vrcholy se stejnou hodnotou (tzn. ležící ve stejné skupině), graf není bipartitní a výpočet předčasně ukončíme.

Časová složitost zůstává stejná jako při určování komponent souvislosti.

```
import sys
navstiven = [0] * n
pocet_navstivenych = 0

def Pruchod(v):
    global pocet_navstivenych
    pocet_navstivenych += 1
    pro všechny hrany (v, u):
        if navstiven[u] == 0:
            navstiven[u] = - navstiven[v]
            Pruchod(u)
        elif navstiven[u] == navstiven[v]:
            print('Graf není bipartitní')
            sys.exit()      # ukončení výpočtu
```

```
while pocet_navstivenych < n:
    s = nenavstiveny_vrchol()
    navstiven[s] = 1
    Pruchod(s)

print('Graf je bipratitní')
print('Rozdělení vrcholů:', navstiven)
```

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 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(reversed(cesta))
```

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).