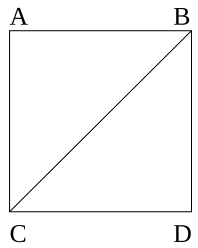
8. Základní grafové algoritmy *(Zapsali M. Petr, O.Tichý, Ľ. Magic)*

Reprezentace grafu

Označme vrcholy grafu na následujícím obrázku písmeny A, B, C, D. Pokud bychom chtěli tento graf uchovat v paměti počítače, máme na výběr hned několik způsobů, jak to udělat.



1. seznam hran

Jednoduše vypíšeme seznam všech dvojic vrcholů grafu spojených hranou.

AB, BC, DC, CA, BC

2. matice sousednosti

Matice sousednosti je pole A o velikosti $n \times n$, jehož prvky na souřadnicích i, j jsou dány následujícím předpisem:

$$A_{i,j} = \begin{cases} 0 \Leftrightarrow \{i,j\} \notin E \\ 1 \Leftrightarrow \{i,j\} \in E \end{cases}$$

Náš graf z obrázku výše by tedy v maticové reprezentaci vypadal takto:

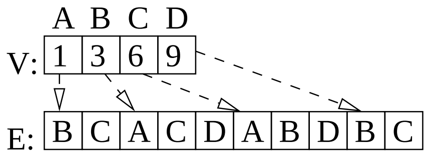
$$\begin{array}{c c} & \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \end{array} & \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

S touto matici se pracuje velmi snadno (např. všechny sousedy i-tého vrcholu zjistíme jednoduše tak, že projdeme i-tý řádek matice, což představuje časovou složitost $O(n)$), ale má i jednu zřejmou nevýhodu: její velikost je vždy kvadratická bez ohledu na to, jak "řídký" je graf. U grafu s mnoha vrcholy, ale s malým počtem hran, tedy budeme zbytečně plýtvat místem v paměti.

Nabízí se proto třetí, úspornější, způsob, a to:

3. seznam sousedů

V paměti počítače by se dal seznam sousedů uchovávat dvěma poli: polem vrcholů V grafu, jehož prvky postupně pro každý vrchol udávají index na začátek odpovídajícího úseku v poli E, ve kterém by byli uloženi jeho sousedé.



Znázornění polí seznamu sousedů.

Na tuto reprezentaci už stačí prostor $O(n + m)$, což už je, na rozdíl od předchozího kvadratického prostoru, docela příjemné.

Ještě dodejme, že ačkoliv jsme jednotlivé reprezentace ukazovali na příkladě neorientovaného grafu, orientované grafy se dají reprezentovat prakticky stejně. U orientovaných grafů je totiž jediný rozdíl: existuje-li hrana z vrcholu u do vrcholu v, neznamená to, že existuje i hrana z v do u (tedy (u, v) a (v, u) tvoří uspořádané dvojice).

Průchod grafem

Definice: Vrchol je dosažitelný, pokud do něj vede cesta z počátečního vrcholu.

Efektivní průchod by měl splňovat tyto podmínky:

- každou hranu projdu právě jednou (resp. maximálně dvakrát - 1x tam, 1x zpět)
- hranou se vracím zpět, až když už z vrcholu, na kterém momentálně jsem, nevede dosud neprozkoumaná hrana
- pokud jsem došel po hraně do již dříve navštíveného vrcholu, hned se vrátím

Konečnost

Důkaz: Plyne z prvního bodu. ♡

Korektnost (algoritmus projde celý graf)

Důkaz: Sporem: Nechť existuje vrchol, který je dosažitelný, ale algoritmus ho neprošel. Pak jsme ale při průchodu narazili na $\{u, v\} \in E$ takovou, že vrchol u je dosažitelný a w dosažitelný není. Tzn., že $\{u, w\}$ tvoří neprozkoumanou hranu $\Rightarrow$ spor s druhým bodem. ♡

Časová složitost

Z toho, že žádná hrana ani vrchol se neprojde vícekrát (první podmínka efektivního průchodu), plyne časová složitost $O(n + m)$.

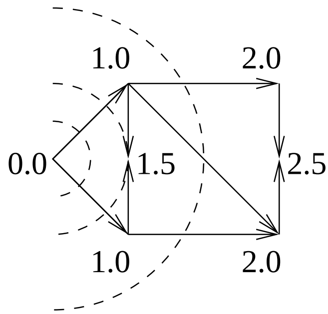
Nyní si něco povíme o dvou způsobech, kterak můžeme procházet grafy.

Průchod do hloubky (Depth First Search, DFS)

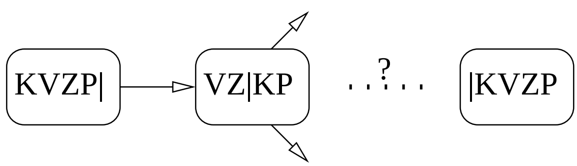
Průchod do hloubky je prvním základním postupem při procházení grafem. Pokud bychom chtěli DFS trochu přiblížit, mohli bychom ho přirovnat k Théseově postupu při průchodu Minotaurovho bludište za pomoci Ariadniny nitě.

Průchod do šířky (Breadth First Search, BFS)

Na rozdíl od prohledávání do hloubky využívá prohledávání do šířky namísto zásobníku frontu. Motivační úlohou by mohlo být např. hledání nejkratší cesty mezi dvěma vrcholy v grafu, nebo známý převozníkův problém.



Př. Koza, vlk a zelí



Řešení ulohy by se dalo popsat jako hledání cesty grafem stavů od počátečního stavu (koza, vlk i zelí jsou na jednom břehu) do koncového (všichni byli bez újmy převezeni na druhý břeh řeky).

DFS na neorientovaném grafu

Při průchodu grafu do hloubky používáme zásobník. Na začátku se v něm nachází pouze vstupní vrchol v. Dokud není zásobník prázdný, tak z něj odebíráme vrchol; ten označíme jako zpracovaný (abychom ho v budoucnu, kdybychom na něj ještě někdy narazili, nezpracovávali znovu) a do zásobníku přidáme všechny jeho sousedy takové, kteří ještě nebyli označeni jako zpracovaní.

Vrcholy značíme těmito barvami

- bílá - dosud nenavštívěné vrcholy (na začátku jsou tedy bílé všechny)
- šedá - označuje právě zpracovávaný vrchol

- černá - už zpracované vrcholy

U každého vrcholu si dále také zapamatujeme čas příchodu a čas odchodu ($in[v]$, $out[v]$).

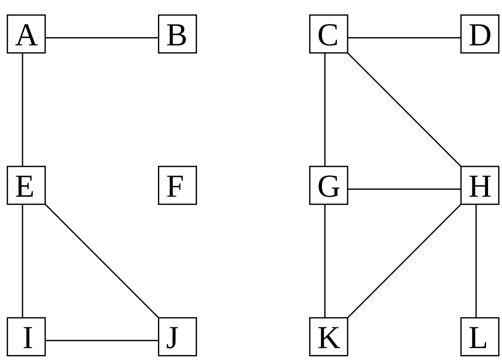
Algoritmus

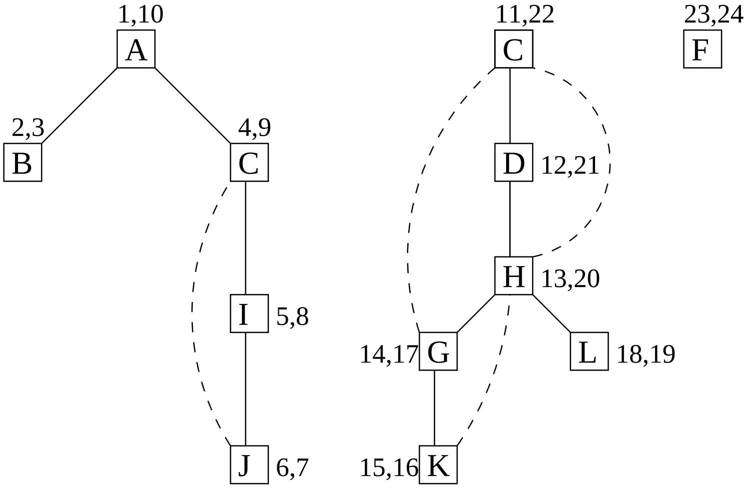
```
procedure Projdi(v)
    color[v] := šedá
    time := time + 1
    in[v] := time
    ∀ w ∈ Sousedi(v):
        if color[w] = bílá ⇒
            Projdi(w)
    color[v] := černá
    time := time + 1
    out[v] := time

procedure DFS()
    ∀ v: color[v] := bílá
    time := 0
    ∀ v ∈ V :
        if color[v] = bílá ⇒
            Projdi(v)
```

Implementaci zásobníku jsme v tomto meta-algoritmu nahradili použitím rekurze (která ale funguje na podobném principu, kdy se na zásobník vkládají jednotlivá volání rekurzivní funkce).

Na následujících obrázcích je znázorněno, jak probíhá průchod grafu pomocí DFS:





Znázornění průběhu algoritmu.

Hrany v tomto znázornění dělíme na:

- stromové - v momentě průchodu vedou do dosud nenavštíveného, bílého, vrcholu
- zpětné - ostatní hrany

Pozorování: plné hrany tvoří strom (DFS strom, strom průchodu do hloubky).

Aplikace:

(1) hledání kostry

(2) hledání komponent souvislosti

Časová složitost těchto aplikací DFS je stejná jako složitost původního meta-algoritmu, tedy $O(n + m)$.

(3) hledání artikulací

Tuto aplikaci si teď rozebereme trošku podrobněji. Nejprve však několik definic.

Definice: *Artikulace* je vrchol v grafu, jehož odebráním se graf rozpadne na alespoň dvě komponenty souvislosti.

Definice: *Most* je hrana v grafu, jejímž odebráním se graf rozpadne na dvě komponenty souvislosti.

Definice: Graf je *2-souvislý,* jestliže po odstranění libovolného vrcholu zůstane souvislý.

Definice: *2-souvislá komponenta* je maximální 2-souvislý podgraf.

Jak bychom tedy při hledání artikulací v grafu mohli postupovat? Jako první se nabízí tento algoritmus:

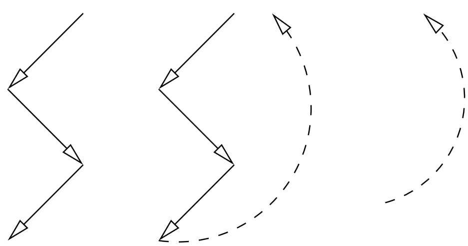
Hloupý algoritmus

Berme si postupně všechny vrcholy grafu; pro každý zjistíme, jestli se jeho odebráním zvýšil počet komponent. K tomu bychom mohli použít průchod do hloubky, který jsme si už ukázali. Procedura Projdi(v) projde a označí všechny vrcholy dosažitelné z v, tedy všechny vrcholy komponenty obsahující vrchol v. Můžeme tedy udělat takovou změnu, že si budeme navštívené vrcholy označovat číslem komponenty.

Takový algoritmus by měl složitost $n \times O(n + m))$ (pro každý vrchol grafu bychom n-krát spouštěli DFS), což se nám vůbec nelíbí. Pokusme se tedy najít něco lepšího. Nejprve učiňme toto pozorování:

Pozorování: v je artikulace $\Leftrightarrow$ v je kořen DFS stromu a má alespoň dva potomky nebo v není kořen a má syna w takového, že ze žádného jeho potomka nevede hrana do předchůdce v.

Definice: Přípustná cesta je taková cesta, pokud vede přes stromové hrany DFS stromu (může končit i skokem zpět).



Příklady přípustných cest.

Dále si definujme funkci low pro každý vrchol v, takto:

low(v) = min(in[w], z v do w vede přípustná cesta)

Pozorování: v je artikulace $\Leftrightarrow$ v je kořen DFS stromu a má alespoň dva potomky nebo má syna w takového, že platí $low(w) \geq in[w]$.

Jak ale hodnotu funkce low(w) spočítat?

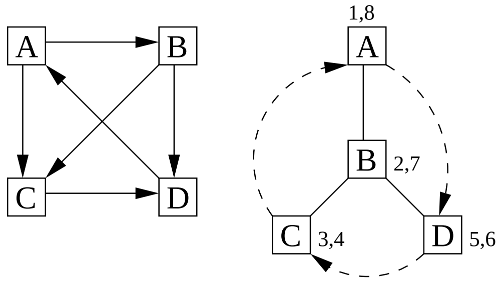
low(v) = min(x, y, z)

x = in[v]

y = min(in[w], $\{w, v\}$ je zpětná hrana)

z = min(low(w), $\{v, w\}$ je stromová hrana)

DFS na orientovaném grafu



Graf a znázornení průbehu:

Hrany v dělíme na:

- stromové $\{A, B\}$, $\{B, C\}$, $\{B, D\}$
- zpětné: vedou do předchůdce v DFS stromě (do šedého vrcholu)
- příčné: vedou do vrcholu v jiném podstromě (do černého vrcholu)
- dopředné: vedou do svého potomka (do černého vrcholu v témže podstromě)

Na závěr několik **pozorování:**

Intervaly (in[v], out[v]) $\forall v \in V$ tvoří dobré uzávorkování.

v je následník u $\Leftrightarrow (in[v], out[v]) \subseteq (in[u], out[u])$

$\{u, v\}$ je zpětná hrana $\Leftrightarrow (in[u], out[u]) \leq (in[v], out[v])$

$\{u, v\}$ je příčná hrana $\Leftrightarrow in[v] < out[v] < in[u] < out[u]$