Teoretická informatika

Obor C, 3. ročník

David Weber

SPŠE Ječná

Poslední aktualizace: 24. července 2023

Obsah

Předmluva **2**

1 Grafové algoritmy **3**

1.1 Grafy a jejich reprezentace 3

1.2 Stromy 3

1.3 Prohledávání do šířky 3

1.4 Prohledávání do hloubky 5

1.5 Binární halda 6

1.6 Dijkstrův algoritmus 8

1.7 Algoritmus A* 8

2 Dynamické programování **9**

Předmluva

Kapitola 1

Grafové algoritmy

1.1 Grafy a jejich reprezentace

Definice 1.1.1 (Graf)**.** Grafem G nazveme uspořádanou dvojici (V, E), kde V je množina *vrcholů* (nebo také *uzlů*) a E množina *hran*, přičemž pokud

- $E \subseteq \{\{u, v\} \mid u, v \in V\}$, pak G nazýváme *neorientovaným* grafem (tj. po hraně lze pohybovat v obou směrech).
- $E \subseteq \{(u, v) \mid u, v \in V\}$, pak G nazýváme *orientovaným* grafem (tj. po hranách se lze pohybovat pouze v jednom směru).

1.2 Stromy

1.3 Prohledávání do šířky

Jednou ze základních úloh je procházení grafu z určitého vrcholu a zjištění dosažitelnosti ostatních vrcholů. Nejednodušším algoritmem v tomto ohledu je tzv. *prohledávání do šířky* (angl. *breadth-first search*, zkráceně BFS). Jeho základní princip spočívá v postupném objevování následníků již nalezených vrcholů. Na počátku dostaneme graf $G = (V, E)$ a nějaký počáteční vrchol $v_0 \in V$. Postupně objevíme všechny sousedy vrcholu v_0, poté všechny sousedy těchto nalezených sousedů, atd. Na BFS lze nahlížet tak, že do počátečního vrcholu nalijeme vodu a sledujeme, jak postupuje vzniklá vlna.

Pro každý vrchol si budeme uchovávat jeho *stav*.

- *Nenalezený* – vrchol jsme ještě během výpočtu neviděli.
- *Otevřený* – vrchol jsme viděli, ale ještě nejsme neprozkoumali všechny jeho sousedy.
- *Uzavřený* – vrchol jsme prozkoumali společně se všemi jeho sousedy a dál se jím již netřeba zabývat.

Na počátku začneme s jedním otevřeným vrcholem a to v_0 (zde začínáme). Po prozkoumání všech sousedních vrcholů se jejich stav změní na otevřený a počáteční vrchol v_0 se uzavře. Obdobně pokračujeme pro nově otevřené vrcholy. Pokud by náhodou mezi dvojicí otevřených vrcholů existovala hrana, pak si sousedního vrcholu všímat nebudeme, neboť byl již otevřen. Pro každý vrchol se ještě dodatečně můžeme uchovávat informaci, jak daleko se nachází od v_0, co do počtu hran ležících na cestě.

Algoritmus 1.3.1 (BFS)
Vstup: Graf $G = (V, E)$ a počáteční vrchol $v_0 \in V$.

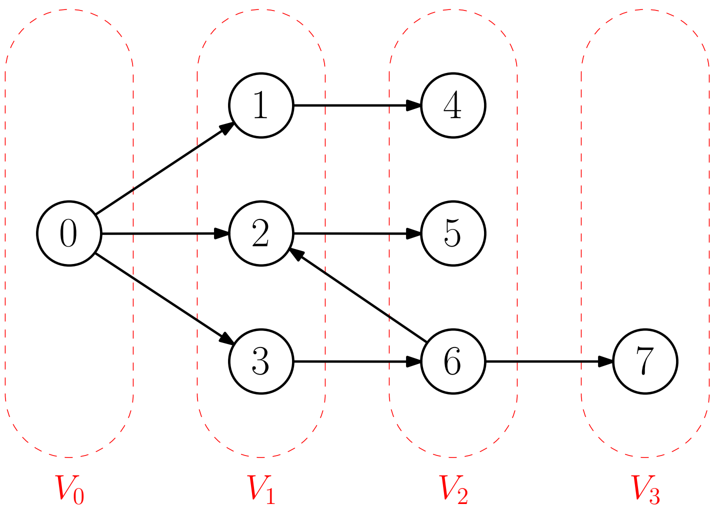
    // Inicializace
    Pro každý vrchol $v \in V$ **opakuj**:
        $stav(v) \leftarrow nenalezený$
        $D(v) \leftarrow \infty$
    $stav(v_0) \leftarrow otevřený$
    $D(v_0) \leftarrow 0$

    Založ frontu Q a přidej do ní vrchol v_0
    Dokud je fronta Q neprázdná, **opakuj**:
        $v \leftarrow$ první vrchol ve frontě Q, který z ní odebereme

        // Prozkomáváme všechny dosud neobjevené sousedy
        Pro každý sousední vrchol w vrcholu v **opakuj**:
            Pokud $stav(w) = nenalezený$, **proveď**:
                $stav(w) \leftarrow otevřený$
                $D(w) \leftarrow D(v) + 1$
                Přidej w do fronty Q
        $stav(v) \leftarrow uzavřený$

Výstup: Seznam vzdáleností D.

BFS rozděluje vrcholy do vrstev podle toho, v jaké vzdálenosti od počátečního vrcholu se nachází (viz obrázek 1.1). Nejdříve jsou prozkoumány vrcholy ve vzdálenosti 0 (tj. pouze v_0), poté ve vzdálenostech 1, 2, ... Z toho vyplývá, že kdykoliv při otevírání libovolného vrcholu v nastavujeme hodnotu $D(v)$, bude tato hodnota vždy odpovídat délce nejkratší cesty z v_0 do v. Tato hodnota bude však nastavena pouze u těch vrcholů, které jsou z v_0 dosažitelné (ostatní vrcholy zůstanou ve stavu nenalezený).



Obrázek 1.1: Vrcholy rozdělené do vrstev podle průběhu BFS.

Zbývá prozkoumat časovou a paměťovou složitost BFS. Označme si počet vrcholů grafu G na vstupu n a počet jeho hran m.

Věta 1.3.2 (Složitost BFS)**.** *Algoritmus BFS doběhne v čase $\mathcal{O}(n + m)$ a spotřebuje paměť $\mathcal{O}(n + m)$.*

Důkaz. Inicializace potrvá $\mathcal{O}(n)$, neboť cyklus iteruje přes všechny vrcholy. Vnější cyklus provede maximálně n iterací, protože každý z vrcholů uzavřeme nejvýše jednou, tj. $\mathcal{O}(n)$.

S vnitřním cyklem je to trochu složitější, protože jeho počet iterací závisí na tom, který z vrcholů otevíráme (resp. na počtu jeho sousedů). To znamená, že pokud si označíme d_i počet sousedů vrcholu i, pak celkový počet iterací vnitřního cyklu přes všechny vrcholy bude $\sum_i d_i$. Lze si ovšem všimnout jedné užitečné věci. Pokaždé, když prozkoumáváme sousední vrchol w nějakého vrcholu v, mohou nastat dva případy podle toho, jestli je G orientovaný graf, nebo neorientovaný.

(i) Graf G je neorientovaný. Pak hranu, která spojuje v a w prozkoumáme právě *dvakrát* (jednou z vrcholu v a podruhé z vrcholu w). Každá hrana se tak započítá dvakrát, tzn. vnitřní cyklus se celkově provede max $2m$-krát (po všech iteracích vnějšího cyklu).

(ii) Graf G je orientovaný. Pak se hrana započítá pouze jednou, a to z vrcholu, z něhož vede. Celkově se vnitřní cyklus provede m-krát.

V prvním případě tak pro časovou složitost bude platit

$$\mathcal{O}\left(n + \sum_i d_i\right) = \mathcal{O}(n + 2m) \stackrel{*}{=} \mathcal{O}(n + m).$$

(V kroku označeném $*$ zanedbáváme konstantu, neboť pracujeme s $\mathcal{O}$-notací.)

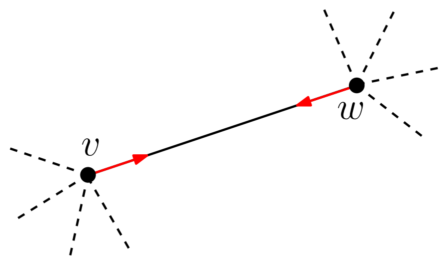
V druhém případě dojdeme ke stejnému výsledku, akorát zmíněná suma bude, kvůli orientaci hran, rovna přesně počtu hran, tj.

$$\mathcal{O}\left(n + \sum_i d_i\right) = \mathcal{O}(n + m).$$

Nyní k prostorové složitosti algoritmu. Na reprezentaci grafu (např. pomocí seznamu sousedů) spotřebujeme paměť $\mathcal{O}(n + m)$ (n vrcholů, m hran)[1]. Dále si uchováváme vrcholy ve frontě, v níž může být v jednu chvíli maximálně n vrcholů, tj. $\mathcal{O}(n)$, a k tomu máme seznam *stav*, kde je uloženo $\mathcal{O}(n)$ vrcholů. Celkově spotřebujeme paměti

$$\mathcal{O}(n + m) + \mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n + m).$$

□



Obrázek 1.2: Znázornění situace v důkazu části (i) věty 1.3.2.

1.4 Prohledávání do hloubky

Na podobném přístupu, jako BFS, je založeno tzv. *prohledávání do hloubky* (anglicky *depth-first search*, zkráceně DFS). Vrcholy však tentokrát budeme zpracovávat rekurzivně. Pokaždé, když budeme otevírat

[1] Resp. každá hrana je v neorientovaných grafech započítána dvakrát, protože každý vrchol obsahuje v seznamu svých sousedů vždy „protější“ vrchol (stejný argument, jako u odvozování časové složitosti BFS).

nový vrchol, se rekurzivně zavoláme všechny jeho sousední vrcholy, u nichž opakujeme stejnou proceduru. Po prozkoumání všech sousedů daný vrchol uzavřeme. Stejně jako BFS si tak budeme uchovávat pole *stavů* pro jednotlivé vrcholy.

Algoritmus 1.4.1 (DFS)
Vstup: Graf $G = (V, E)$ a počáteční vrchol $v_0 \in V$.

```
// Inicializace
Pro každý vrchol v ∈ V opakuj:
    stav(v) ← nenalezený
Zavolej DFS2(v0)

// Rekurzivní volání na vrcholy
Funkce DFS2 (vrchol v)
    stav(v) ← otevřený
    Pro každý sousední vrchol w vrcholu v opakuj:
        Pokud stav(w) = nenalezený, proveď:
            Zavolej DFS2(w)
    stav(v) ← uzavřený
```

Věta 1.4.2 (Složitost DFS)**.** *Algoritmus DFS doběhne v čase $\mathcal{O}(n+m)$ a spotřebuje paměť $\mathcal{O}(n+m)$.*

Důkaz. Algoritmus DFS se oproti BFS liší pouze v pořadí, v jakém pořadí dosažitelné vrcholy zpracovává, to však nemá na časovou složitost žádný vliv. Argument pro její odvození je tak stejný jako u BFS, viz věta 1.3.2 v minulé sekci.

V paměti si musíme uchovávat reprezentaci grafu, to zabere $\mathcal{O}(n+m)$ paměti (opět např. pomocí seznamu sousedů) a máme seznam *stav* s n prvky (vrcholy). Zároveň si při volání DFS2 musíme na zásobník rekurze ukládat jednotlivé *aktivační záznamy*. Protože vrcholů je v grafu n, pak na zásobníku rekurze bude v jednu chvíli maximálně n záznamů, tj. $\mathcal{O}(n)$. Celkově spotřebujeme $\mathcal{O}(n+m) + \mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n+m)$ paměti. □

Je dobré si uvědomit, že byť algoritmy BFS a DFS mají stejnou časovou a prostorovou složitost, nelze zcela rovnocenně použít na stejné typy úloh. Např. BFS se hodí pro hledání nejkratší cesty *v neohodnoceném grafu* (pro ohodnocené grafy se používá např. Dijkstrův algoritmus, viz sekce 1.6), kdežto DFS se více hodí na prohledávání stavového prostoru, neboť typicky nezabere tolik paměti.

Cesta, kterou se DFS dostane do libovolného vrcholu v, nemusí být nutně nejkratší (silně závisí na pořadí, v jakém procházíme sousedy jednotlivých vrcholů).

1.5 Binární halda

Uveďme motivační příklad na úvod. Mějme seznam čísel, z něhož chceme (pokud možno co nejrychleji) vybrat minimální/maximální hodnotu. Pro maximum bychom sestavit následující jednoduchý algoritmus.

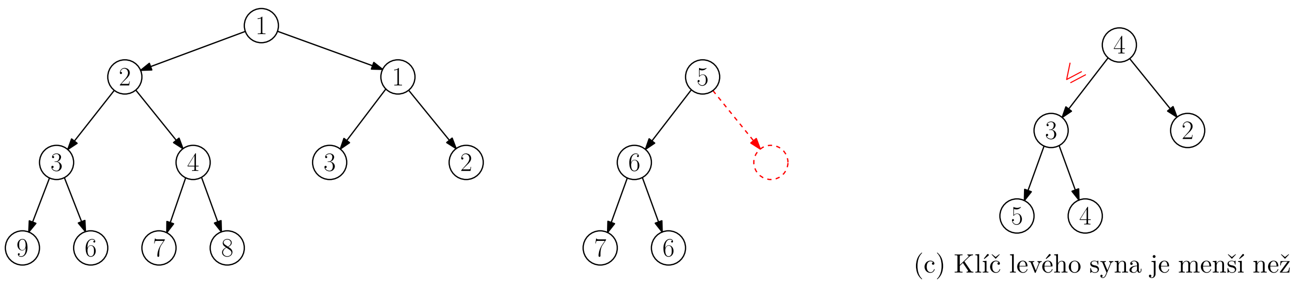
Algoritmus 1.5.1 (MAX)
Vstup: Seznam čísel $x_1, x_2, \ldots, x_n$
$m \leftarrow x_1$
Pro $i = 1, 2, \ldots, n$ **opakuj**:
Pokud $x_i > m$, **proveď**:
$m \leftarrow x_i$

Výstup: Maximální hodnota seznamu m

Časová složitost bude zjevně $\mathcal{O}(n)$, neboť algoritmus prochází všech n prvků. Takovou úlohu lze však řešit rychleji, pokud si prvky vhodně uspořádáme. K tomu můžeme použít tzv. *haldu.*

Definice 1.5.2 (Minimová binární halda)**.** Minimová binární halda je datová struktura tvaru binárního stromu, kde v každém vrcholu je uložena *právě jedna* hodnota (tzv. *klíč*, pro vrchol v budeme značit jeho klíč $k(v)$) a navíc platí:

(i) každá hladina je *plně obsazena*, kromě poslední, pričemž hladiny jsou zaplněny *zleva*
(ii) a je-li v libovolný vrchol a s jeho syn, pak $k(v) \leqslant k(s)$.



(a) Korektní halda.

(b) Chybějící vrchol v 1. hladině.

(c) Klíč levého syna je menší než klíč kořene.

Obrázek 1.3: Příklady korektních a nekorektních hald.

Zde se nyní na chvíli pozastavme. Podmínka (ii) v definici binární haldy 1.5.2 má za následek totiž velmi příjemnou vlastnost, když se podíváme, kde se v haldě nachází minimum. Pokud se budeme pohybovat od kořene směrem „dolů“, hodnoty ve vrcholech se budou pouze zvětšovat, neboť klíče synů mají musí mít vždy stejnou nebo větší hodnotu než klíč v rodiči. Minimum se tak vždy nachází *v kořeni stromu*, což znamená, že zjistění minima tak můžeme provést v *konstantním čase* $\mathcal{O}(1)$.

> Pokud bychom chtěli naopak *maximovou binární haldu*, bude definice 1.5.2 vypadat obdobně, akorát v podmínce (ii) bude obrácená nerovnost, tj. $k(v) \geqslant k(s)$ (klíč v rodiči má vždy stejnou nebo vyšší hodnotu než klíče v jeho synech). Maximum se bude opět nacházet v kořeni haldy.

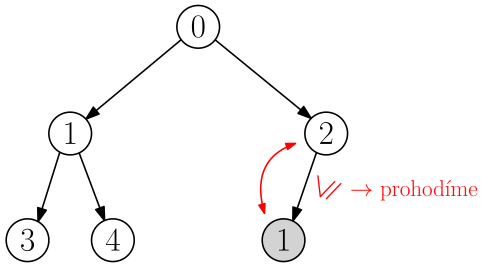
Je však více operací, které bychom rádi s haldou prováděli. Vypišme si všechny, které nás budou zajímat.

- INSERT(x) – *vložení* nového vrcholu s klíčem x do haldy.
- MIN() – *nalezení* vrcholu s nejmenším klíčem (už jsme zmínili).
- EXTRACTMIN() – *odebrání* vrcholu s nejmenším klíčem.
- INCREASE(v) – *zvýšení* hodnoty klíče ve zvoleném vrcholu v.
- DECREASE(v) – *snížení* hodnoty klíče ve zvoleném vrcholu v.

Podívejme se postupně na jednotlivé operace a jejich realizaci. S dovolením si zde nyní odpustíme zápis pomocí pseudokódu a pro jednoduchost si dané operace pouze slovně vysvětlíme. Pro následné odvození časové složitosti nám to bude stačit.

Vkládání nového vrcholu

Při vkládání nového vrcholu (Insert) se nejdříve potřebujeme vypořádat s tím, kam vrchol v haldě umístíme. S tím nám ale pomůže podmínka (i) v definici binární haldy. Všechny hladiny stromu jsou vždy plně obsazené, kromě poslední, která se všechny vrcholy nachází vlevo. To znamená, že nový vrchol můžeme umístit na první volnou pozici vlevo na poslední hladině. To však nemůžeme provést zcela beztrestně, neboť nemáme nijak zaručeno, že bude splněna podmínka (ii). Klíč v novém vrcholu může mít hodnotu *ostře menší* než jeho rodič. Haldu „opravíme“ postupným prohazováním vrcholu s jeho rodičem, dokud nebude podmínka splněna (viz obrázek 1.4).



Obrázek 1.4: Vkládání nového vrcholu do haldy.

1.6 Dijkstrův algoritmus

1.7 Algoritmus A*

Kapitola 2

Dynamické programování