Teoretická informatika

Obor C, 3. ročník

David Weber

SPŠE Ječná

Poslední aktualizace: 16. srpna 2023

Obsah

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. Nejjednodušším 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.

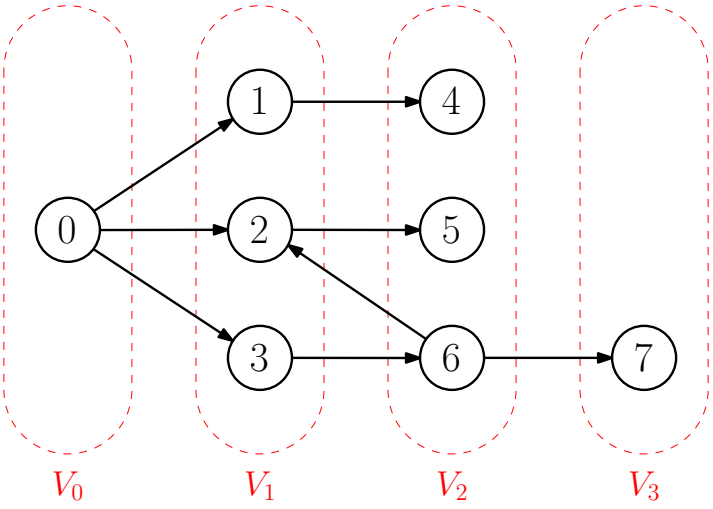
Algoritmus 1.3.1 (BFS)
Vstup: Graf $G = (V, E)$ a počáteční vrchol $v_0 \in V$.
// Inicializace
Pro každý vrchol $v \in V$ **opakuj**:
    $stav(v) \leftarrow$ *nenalezený*
    $D(v) \leftarrow \infty$
$stav(v_0) \leftarrow$ *otevřený*
$D(v_0) \leftarrow 0$

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

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

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

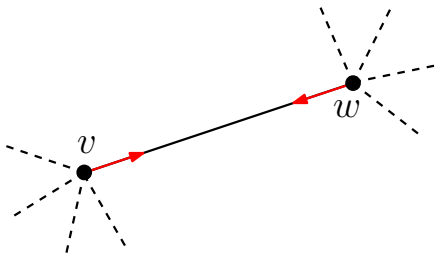
V druhém případě dojdeme ke stejnému výsledku, akorát zmíněná suma bude, kvůli orientaci hran, rovna přesně počtu hran, tj.

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

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 $v \in V$ **opakuj**:
    $stav(v) \leftarrow$ *nenalezený*
Zavolej DFS2(v_0)

// Rekurzivní volání na vrcholy
Funkce DFS2(vrchol v)
    $stav(v) \leftarrow$ *otevřený*
    Pro každý sousední vrchol w vrcholu v **opakuj**:
        Pokud $stav(w) =$ *nenalezený*, **proveď**:
            Zavolej DFS2(w)
    $stav(v) \leftarrow$ *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.

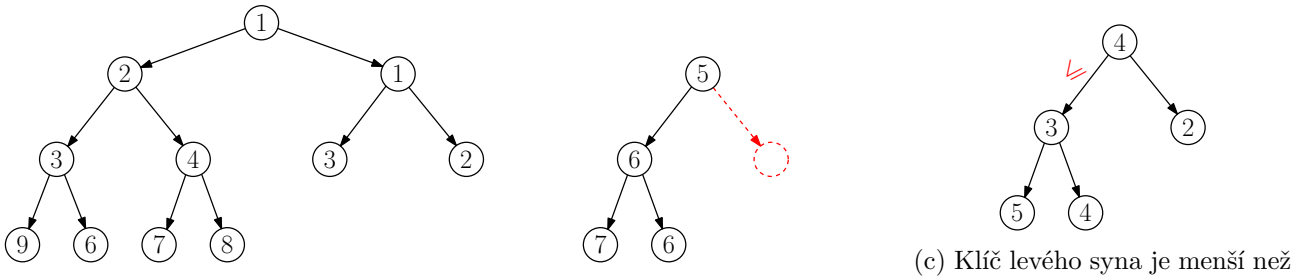
Algoritmus 1.5.1 (Max)
Vstup: Seznam čísel $x_1, x_2, \ldots, x_n$
$m \leftarrow x_1$
Pro $i = 1, 2, \ldots, n$ **opakuj**:
  Pokud $x_i > m$, **proveď**:
    $m \leftarrow x_i$

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 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.

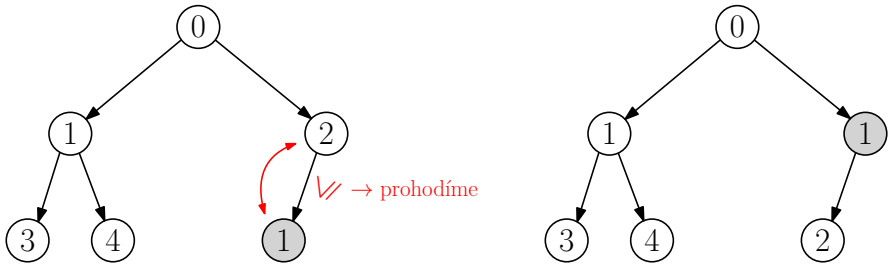
- Insert(x) – *vložení* nového vrcholu s klíčem x do haldy.
- Min() – *nalezení* vrcholu s nejmenším klíčem (už jsme zmínili).
- ExtractMin() – *odebrání* vrcholu s nejmenším klíčem.
- Increase(v) – *zvýšení* hodnoty klíče ve zvoleném vrcholu v.
- Decrease(v) – *snížení* hodnoty klíče ve zvoleném vrcholu 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). Postup můžeme zapsat zkráceně ve třech



Obrázek 1.4: Vkládání nového vrcholu do haldy.

bodech takto:

- Vložíme nový vrchol s klíčem x na první volnou pozici zleva na poslední hladině.
- pokud je klíč rodiče větší než x, prohodíme vrcholy (resp. jejich klíče)[2].
- Opakujeme druhý krok.

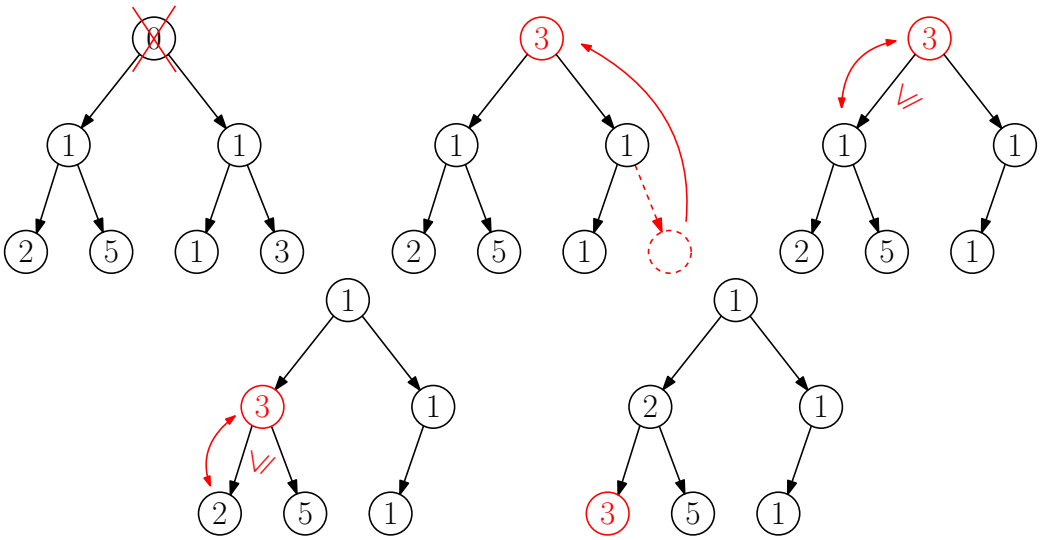
Odstranění minima

Co se týče získání minimálního klíče v haldě, jedná se o velmi triviální operaci. Zkrátka „přečteme“ klíč v kořeni a jsme hotovi. Ovšem pokud bychom chtěli minimum (EXTRACTMIN) ze stromu odstanit, bude to již o něco složtější. Kořen ze stromu nemůžeme „jen tak“ smazat, jeho pozici musí zastoupit jiný vrchol. Abychom neporušili vlastnosti haldy, nahradíme odstraněný kořen *nejlevějším vrcholem na poslední hladině*. Tím jsme jistě neporušili vlastnost (i), neboť poslední hladina, jako jediná, nemusí být plně obsazená.

Nyní však (stejně jako u vkládání vrcholu) nastává problém s druhou podmínkou, neboť touto operací jsme ji mohli porušit. Provedeme obdobný proces, jako pri vkládání vrcholu do haldy, ale vrchol nyní bude „probublávat“ směrem dolů. Podíváme se na syny daného vrcholu a pokud klíč některého z nich je menší než klíč nového kořene, pak jej s ním prohodíme (v případě, že klíče obou synů jsou menší, než klíč kořene, prohodíme s menším z nich). V bodech můžeme zapsat celou operaci následovně:

- Smažeme kořen a nahradíme jej nejlevějším vrcholem na poslední hladině.
- Tento vrchol (resp. jeho klíč) porovnáme s jeho syny a případně prohodíme s tím, jehož klíč je menší.

[2] Vrchol v podstatě takto „probublá“ do správné pozice ve stromě (příp. až do pozice kořene).



Obrázek 1.5: Odebírání vrcholu (kořene) s minimálním klíčem.

- Opakujeme druhý krok.

Operacím, které jsme prezentovali při vkládání vrcholu, resp. odebírání minima z haldy, kdy vrchol „probublával“ nahoru, resp. dolů, se někdy nazývají BubbleUp a BubbleDown. Ještě se nám budou hodit u operací Increase a Decrease.

Zvýšení a snížení hodnoty klíče ve vrcholu

Poslední dvojice operací, která se nám bude hodit, je zvýšení (Increase) a snížení (Decrease) hodnoty klíče ve vybraném vrcholu. K tomu potřebujeme akorát vyřešit, jak budeme k vrcholům přistupovat. Podle klíče vyhledávat neumíme, ale můžeme si zavést např. pomocný slovník, který budeme „adresovat“ pomocí příslušného vrcholu a vracenou hodnotou bude klíč ve vrcholu. Takto spotřebujeme navíc pouze $\mathcal{O}(n)$ paměti, kde n je počet vrcholů v haldě.

Realizace samotných operací je již jednoduchá. Pokud protovedeme Increase klíče v určitém vrcholu, pak z důvodu, že klíče v synech vrcholu jsou stejné nebo větší, se halda může pokazit pouze „směrem dolů“. Tedy provedeme nám již známé prohazování vrcholu s jeho syny, dokud nebude halda opravena (BubbleDown), stejně jako v případě odebírání kořene (ExtractMin).

Operaci Decrease provedeme zcela stejně, akorát nyní se halda bude kazit „směrem nahoru“, takže provedeme BubbleUp, jako při vkládání nového vrcholu (Insert).

V případě *maximové binární haldy* by operace vypadaly zcela stejně, akorát při operaci Increase budeme opravovat haldu směrem nahoru a u Decrease směrem dolů.

Časové složitosti operací

Pojďme si nyní rozebrat časové složitosti zmíněných operací Insert, Min, ExtractMin, Increase a Decrease a podívat se, jak moc jsme si pomohli oproti práci s polem (seznamem). Pokud se pozorně podíváme na operace popsané výše, je zjevné, že nejvíce bude naše operace zpomalovat ono „probublávání“ vrcholu při opravování haldy (vyjma operace Min, kde jen přečteme klíč v kořeni), tj. BubbleUp

a BubbleDown. Na nich bude celková časová složitost záviset.

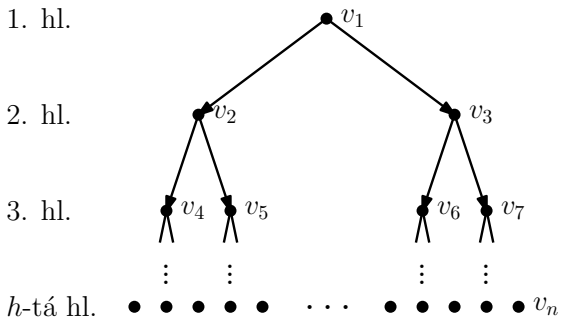
Všimněme si, že po každém prohození vrcholů při BubbleUp nebo BubbleDown se vrchol posune o jednu hladinu výše/níže.

Víme tak, že při opravování haldy se vrchol může posunout *maximálně o počet hladin*, který má daná halda. Časová složitost daných operací bude tak závislá na jejich počtu. Tím jsme se dostali o trochu blíže řešení, ale rádi bychom věděli, jak se bude době trvání daných operací měnit v závislosti na *počtu vrcholů v haldě*, nikoliv počtu hladin. Mezi těmito proměnnými však existuje hezká závislost.

Dejme tomu, že naše halda má n vrcholů v celkově h hladinách, přičemž poslední hladina *je plně obsazena* (viz obrázek 1.6). (Mohlo by se, v obecném případě, samozřejmě stát, že poslední hladina plně obsazena nebude, ale jediné, co se tím změní, bude, že mezi n a výsledným výrazem bude místo rovnosti nerovnost. S plnou hladinou se nám ale bude lépe počítat, protože při analýze časové složitost nás zajímá nejhorší možný případ.) Pokud budeme mít haldu s jednou hladinou, bude mít pouze jeden vrchol, a to kořen. S dvěma hladinami budou již 3 vrcholy v haldě. S třemi hladinami jich bude 7, atd (viz tabulka 1.1).

Počet hladin h	1	2	3	4	5	6
Počet vrcholů n	1	3	7	15	31	63

Tabulka 1.1: Vztah mezi počtem hladin h a počtem vrcholů n v plně obsazené haldě.



Obrázek 1.6: Odebírání vrcholu (kořene) s minimálním klíčem.

Z tabulky 1.1 si můžeme všimnout, že počet vrcholů je vždy *mocnina dvojky snížená o jedna*. Výsledná závislost je tedy $n = 2^h - 1$. V případě, že by poslední hladina nebyla plně obsazena, by se rovnost změnila v nerovnost, tj. $n \leqslant 2^h - 1$. Tím máme již vše připravené k tomu, abychom odvodili časovou složitost daných operací.

Věta 1.5.3 (Složitost operací v binární haldě)**.** Operace Insert, ExtractMin, Increase a Decrease mají časovou složitost $\mathcal{O}(\log n)$. Operace Min má časovou složitost $\mathcal{O}(1)$.

Důkaz. V případě Min je to jednoduché, zkrátka jen přečteme klíč v kořeni, což zjevně potrvá $\mathcal{O}(1)$. U zbývajících operací opravujeme haldu pomocí BubbleUp a BubbleDown. Předpokládejme, nastal nejhorší možný případ, tj. halda má h hladin a je plně obsazena (tzn. včetně poslední hladiny). Víme, že „probublání“ vrcholu haldou nahoru/dolů potrvá řádově $\mathcal{O}(h)$. My však víme, že $n = 2^h - 1$. Tedy zde bude stačit, když si vyjádříme h z rovnosti.

$$n = 2^h - 1 \Leftrightarrow 2^h = n + 1 \Leftrightarrow h = \log_2(n+1) \overset{*}{=} \frac{1}{\log 2} \cdot \log(n+1).$$

(V rovnosti $*$ jsme využili vzorec $\log_b a = \log a / \log b$.) Operace BubbleUp a BubbleDown tedy potrvají

$$\mathcal{O}(h) = \mathcal{O}(\frac{1}{\log 2} \cdot \log(n+1)) \overset{*}{=} \mathcal{O}(\log n).$$

(V $*$ je jednička v logaritmu zanedbatelná v rámci $\mathcal{O}$-notace. Výraz $1/\log 2$ představuje konstantu a tudíž je též zanedbatelný.) Tedy operace Insert, ExtractMin, Increase a Decrease mají logaritmickou časovou složitost. □

Zkusme si na závěr udělat menší porovnání oproti poli, se kterým jsme začínali (viz tabulka 1.2). Lo-

	Insert	Min	ExtractMin	Increase	Decrease
Pole (seznam)	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Binární halda	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

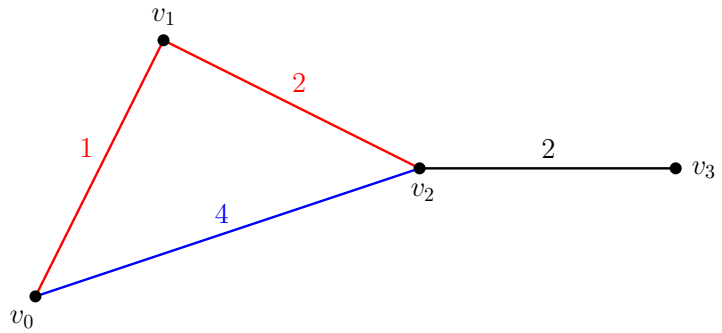
Tabulka 1.2: Porovnání časových složitostí operací v haldě a v poli.

garitmická časová složitost je určitě lepší, než lineární, je však vidět, že jsme si zároveň pohoršili v případě vkládání a úpravy hodnot klíčů. Je tomu tak z důvodu, že pole má svou výhodu v téměř nulové režii, kdežto halda (kvůli své pevně definované struktuře) spotřebuje nějaký čas, aby byla uvedena do korektního stavu.

1.6 Dijkstrův algoritmus

Již jsme si ukázali, že v *neohodnoceném grafu* $G = (V, E)$ umíme relativně jednoduše najít délku nejkratší cesty do libovolného vrcholu z nějakého výchozího vrcholu $v_0 \in V$. Obvykle však hrany nemusí mít stejnou váhu (cenu). Např. cesty (vrcholy) mezi městy (hrany) mohou být v různém stavu a některé jsou tak lepší než jiné.

Zde ji narážíme na problém, neboť obecně platí, že když nalezneme vrchol přes dvojici různých hran, nemusí být nalezené cesty stejně dlouhé. Na obrázku 1.7 si lze všimnout, že cesta (v_0, v_1, v_2) je kratší než (v_0, v_2), přestože má více hran.



Obrázek 1.7: Příklad ohodnoceného grafu.

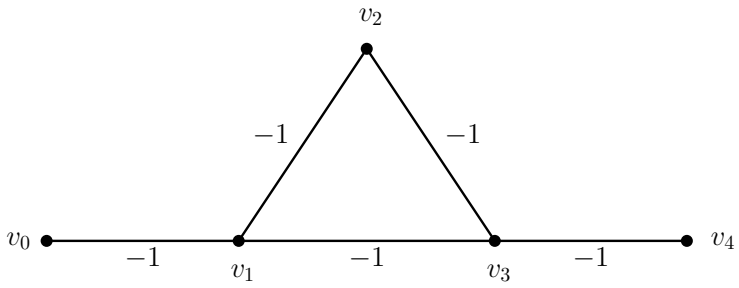
Nabízí se varianta převést ohodnocený graf na neohodnocený tak, že rozdělíme hranu na takový počet (neohodnocených) hran, kolik činí její původní váha. V čistě teoretické rovině se jedná o funkční řešení, neboť na nově vzniklý graf již lze aplikovat např. BFS, které jsme popsali v sekci 1.3. Ne každý graf však musí mít „malé“ váhy hran. Pokud bychom vzali např. graf, kde váhy hran jsou v řádech tisíců, bude pro

BFS potřeba provést zbytečně mnoho práce, neboť pro zpracování jedné ohodnocené hrany bude třeba provést tisíce iterací.

Jedním z nejznámějších algoritmů v tomto ohledu je tzv. *Dijkstrův algoritmus*, který popsal v roce 1958 holandský informatik *Edsger Wybe Dijkstra*.[3] Podobně jako u BFS a DFS, i zde budeme postupně *otevírat* a *uzavírat* vrcholy, přičemž každý uzavřeme *nejvýše jednou*, avšak je třeba mít na paměti důležitou věc.

První nalezená cesta do libovolného vrcholu již nemusí být nutně nejkratší. Cestu do daného vrcholu bude třeba postupně vylepšovat.

Zaměřme se na grafy, jejichž váhy hran jsou nezáporné. Záporně ohodnocené hrany mohou způsobovat problémy, neboť mezi určitou dvojicí vrcholů by již nemusela nutně existovat nejkratší cesta. Např. v obrázku 1.8 si lze všimnout, že mezi vrcholy v_0 a v_4 neexistuje nejratší cesta konečné délky, neboť cyklus (v_1, v_3, v_2, v_1) lze projít libovolněkrát, přičemž každým průchodem se zkrátí o 3. Odpověď na délku nejrakší cesty mezi těmito vrcholy by tak musela být $-\infty$. Podobným grafům se tak chceme



Obrázek 1.8: Graf, kde mezi v_0 a v_4 neexistuje (konečná) nejkratší cesta.

vyhnout, neboť by naše úvahy pak již nemusely fungovat (cestu mezi vrcholy bychom mohli potenciálně do nekonečně vylepšovat a algoritmus by se tak nikdy nezastavil).

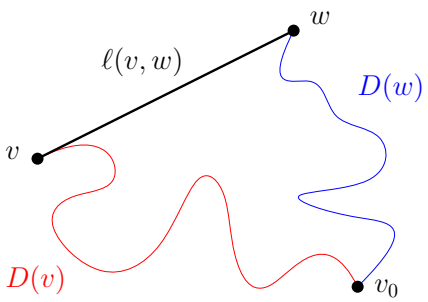
V dalších odstavcích budeme váhu hrany vedoucí z vrcholu u do vrcholu v značit $\ell(u, v)$.

Algoritmus 1.6.1 (Dijkstra)
Vstup: Nezáporně ohodnocený graf $G = (V, E)$ a počáteční vrchol $v_0 \in V$.

```
// Inicializace
Pro všechny vrcholy v ∈ V opakuj:
    stav(v) ← nenalezený
    D(v) ← ∞

stav(v_0) ← otevřený
D(v_0) ← 0
Dokud existují otevřené vrcholy, opakuj:
    // Kandidát na nejkraší cestu do sousedních vrcholů
    v ← otevřený vrchol s nejmenším D

    Pro všechny sousedy w ∈ V vrcholu v opakuj:
        // Našli jsme lepší cestu
        Pokud D(v) + ℓ(v, w) < D(w), proveď:
            D(w) ← D(v) + ℓ(v, w)
            stav(w) ← otevřený
    stav(v) ← uzavřený
```

[3] Jedná se o holandské jméno, čteme „dajkstra“.

Výstup: Seznam vzdáleností D.



Obrázek 1.9: Znázornění situace v průběhu Dijkstrova algoritmu.

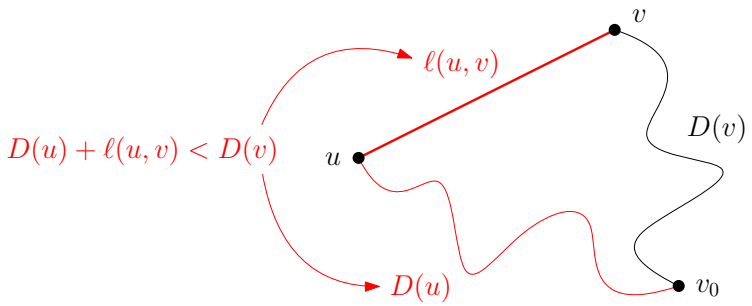
Správnost Dijkstrova algoritmu

Nejprve je dobré si uvědomit, proč algoritmus vlastně funguje. Kdykoliv uzavíráme vrchol (tzn. vybereme vrchol s nejmenší hodnotou $D(v)$), pak se spoléháme na to, že $D(v)$ opravdu odpovídá v daný moment délce nejkraší cesty z v_0 do v. Ale proč to tak musí být?

Zkusme uvážit opačnou situaci, tzn. kdyby nastalo, že jsme vybrali nějaký vrchol v, jehož $D(v)$ neodpovídá délce nejkratší cesty. Pak by musel existovat nějaký sousední vrchol u vrcholu v, přes nějž vede nejkratší cesta, jejíž délka je $D(u) + \ell(u, v)$ (viz obrázek 1.10). To znamená, že $D(v) > D(u) + \ell(u, v)$. Jenže v takovou chvíli by si algoritmus nutně musel vybrat vrchol u místo vrcholu v, protože

$$D(v) > D(u) + \ell(u, v) \overset{*}{>} D(u).$$

V kroku označeném $*$ vycházíme právě z toho, že váhy hran jsou nezáporné, tj. $\ell(u, v) > 0$ (proto tento argument funguje). Tedy stručně řečeno[4], pokud $D(v)$ neodpovídá délce nejkratší cesty, pak musí existovat jiný vrchol u s nižším $D(u)$.



Obrázek 1.10: Situace při výběru vrcholu v, kdy $D(v)$ neodpovídá délce nejkratší cesty.

Časová složitost Dijkstrova algoritmu

Zbývá nám ještě analyzovat časovou složitost. Pokud se zpětně podíváme na pseudokód algoritmu, lze si všimnout, že podstatnou roli bude hrát vybírání vrcholu s minimální hodnotou $D(v)$. Nabízí se tak otázka, jak danou operaci implementovat. První varintou je hodnoty D realizovat jako prostý *seznam (popř. pole).*

[4] Nebo spíš napsáno?

Věta 1.6.2 (Dijkstra se seznamem)**.** Dijkstrův algoritmus, kde hodnoty D ukládáme do seznamu (pole), doběhne v čase $\mathcal{O}(n^2 + m)$.

Důkaz.
- Inicializace zabere $\mathcal{O}(n)$ (máme n vrcholů).
- Každý vrchol uzavřeme opět nejvýše jednou, tzn. vnější cyklus se provede $\mathcal{O}(n)$-krát.
- Hledání minimálního D zabere v rámci každé iterace čas $\mathcal{O}(n)$.
- Stejně jako u BFS a DFS, i zde každou hranu zkontroluji nejvýše dvakrát (záleží, zda graf G je orientovaný), tzn. $\mathcal{O}(2m) = \mathcal{O}(m)$.

Celkově tedy máme

$$\overbrace{\mathcal{O}(n)}^{\text{Init}} + \underbrace{\mathcal{O}(n) \cdot \mathcal{O}(n)}_{\text{Výběr minima}} + \mathcal{O}(m) = \mathcal{O}(n^2 + n + m) = \mathcal{O}(n^2 + m).$$

□

Kvadratická časová složitost není úplně zlá, ale můžeme si ještě trochu přilepšit. Pokud si vzpomeneme na binární haldu z oddílu 1.5, všimneme si v konečném důsledku, že při použití v Dijkstrově algoritmu se některé operace zrychlí.

Věta 1.6.3 (Dijkstra s haldou)**.** Dijkstrův algoritmus, kde hodnoty D ukládáme do binární haldy, doběhne v čase $\mathcal{O}((n + m) \cdot \log n)$.

Důkaz. Inicializace trvá zase $\mathcal{O}(n)$. Při operacích s binární haldou víme, jak dlouho potrvají (vit tabulka 1.2). Akorát si nyní musíme rozmyslet, kdy se která operace provádí.

- Při *otevírání* vrcholu provádíme v haldě operaci INSERT[5]. Celkově vložíme do haldy max. všech n vrcholů, což potrvá $n \cdot \mathcal{O}(\log n) = \mathcal{O}(n \log n)$.
- Při *uzavírání* vrcholu provádíme operaci EXTRACTMIN (opět max. n-krát), tj. $n \cdot \mathcal{O}(\log n) = \mathcal{O}(n \log n)$.
- Při *aktualizaci vzdálenosti* $D(v)$ provádíme DECREASE (tato hodnota se nikdy nemůže zvýšit). To provádíme vždy při kontrole sousedů (tzn. daných hran), kterých je pro všechny vrcholy dohromady max. $\mathcal{O}(m)$ (stejný argument jako při započítávání hran). Tzn. $m \cdot \mathcal{O}(\log n) = \mathcal{O}(m \log n)$.

Po sečtení dostaneme

$$\begin{aligned}\mathcal{O}(n + n\log n + n\log n + m\log n) &= \mathcal{O}(n + 2n\log n + m\log n)\\ &= \mathcal{O}(n + n\log n + m\log n)\\ &= \mathcal{O}(n\log n + m\log n)\\ &= \mathcal{O}((n+m)\cdot\log n).\end{aligned}$$

□

Ačkoliv to není úplně zjevné, výraz $(n + m) \cdot \log n$ roste o dost pomaleji oproti $n^2 + m$, tedy binární haldou si skutečně pomůžeme oproti seznamu.

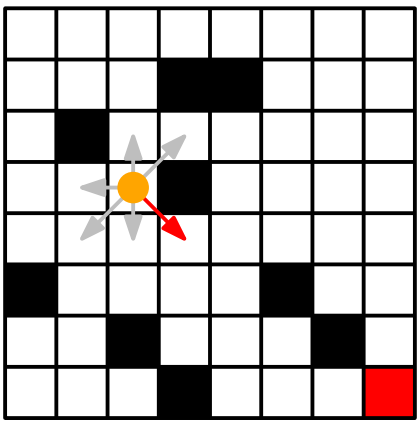
[5] Náš odhad $\mathcal{O}(n \log n)$ by šel dokonce zlepšit i na $\mathcal{O}(n)$, když do něj započítáme fakt, že hladin v haldě je ze začátku 0 a postupně přibývají (tzn. není jich z počátku „moc“), ale celkový odhad složitosti by vyšel stejně a navíc odvození tohoto faktu je již trochu náročnější.

Závěrem zde je ještě uveďme, že často nehledáme cestu do všech vrcholů, ale pouze mezi nějakou konkrétní dvojicí. V takovou chvíli můžeme výpočet algoritmu zarazit ve chvíli, kdy narazíme na cílový vrchol, protože jak již víme, při otevírání vrcholu je $D(v)$ skutečná délka nejkratší cesty. V tomto případě je možné ještě dále urychlit hledání nejkratší cesty, neboť Dijkstrův algoritmus může pak provádět hodně nadbytečné práce. Na to se podíváme v další sekci 1.7.

1.7 Algoritmus A*

Nyní omezíme naše soustředění na případy, kdy hledáme cestu pouze do nějakého konkrétního vrcholu[6] $v_G \in V$ (nikoliv do všech, jako tomu bylo doposud). Stále pracujeme s grafy, jejichž hrany mají nezáporné ohodnocení. Všechny dosavadně vysvětlené algoritmy by jistě fungovaly i v tomto případě (při otevírání cílového vrcholu jednoduše vyskočíme z hlavního cyklu). Nabízí se však otázka, zdali nemůžeme znalosti cílového vrcholu nějak využít. Pokud bychom např. hledali nejkratší cestu v mřížce s rozmístěnými překážkami (tj. na některá políčka nelze vstoupit), nejpíše by dávalo smysl upřednostňovat políčka, která jsou blíže cílovému políčku, než ty, která jsou od něj dál.

Mějme bludiště, v němž se nachází hráč (oranžový kruh), jež se může pohybovat libovolným směrem vždy o jedno pole, přičemž cílovým pole se nachází vpravo dole (červeně vyznačené). Situaci lze vidět na obrázku 1.11. Takovou situaci můžeme znázornit pomocí grafu, kde každá hrana má váhu 1, popř. lze použít neohodnocený graf. Pokud bychom aplikovali např. BFS, či Dijkstrův algoritmus, prozkoumali



Obrázek 1.11: Mřížka a optimální směr pohybu hráče.

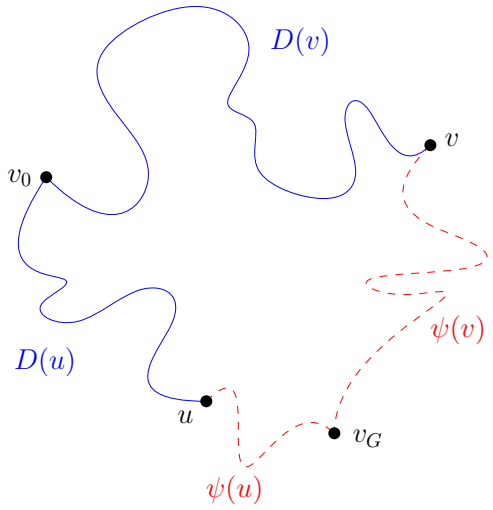
bychom postupně všechny směry (šipky) stejně. Je však zřejmé, že nejvýhodnější je vydat směrem vyznačeným červeně, neboť je nejblíže cíli. V roce 1968 tak přišli Peter Hart, Nils Nilsson a Bertam Raphael s úpravou původního Dijkstrova algoritmu, která nese název A*.

Myšlenka je stále stejná, avšak nyní budeme vybírat vrcholy podle hodnoty $D(v)+\psi(v)$ (místo pouhého $D(v)$), kde ψ je tzv. *heuristická funkce* nebo zkráceně *heuristika*, která slouží jako *odhad vzdálenosti od cílového vrcholu* v grafu (viz obrázek 1.12). Součtu $D(v) + \psi(v)$ budeme říkat tzv. *f-skóre*[7], značíme $f(v)$. V každé iteraci tak budeme vybírat vrchol s nejnižším f-skóre, k čemuž lze opět použít pole, nebo binární haldu.

Oproti Dijkstrově algoritmu tedy skutečně nenastává moc změn, akorát je potřeba si zvlášť uchovávat hodnoty f-skóre.

[6] „G“ od angl. *goal*

[7] V jiných textech, převážně anglických, se někdy používá pro heuristiku označení h a pro vzdálenost označení g, tzv. *g-skóre*, tj. $f(v) = g(v) + h(v)$.



Obrázek 1.12: Znázornění heuristické funkce ψ.

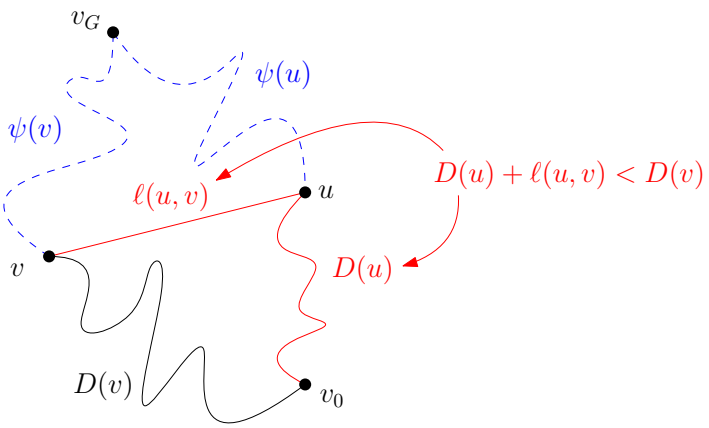
Algoritmus 1.7.1 (A*)
Vstup: Nezáporně ohodnocený graf $G = (V, E)$ a počáteční vrchol $v_0 \in V$.

```
// Inicializace
Pro všechny vrcholy v ∈ V opakuj:
    stav(v) ← nenalezený
    D(v) ← ∞, f(v) ← ∞

stav(v_0) ← otevřený
D(v_0) ← 0, f(v_0) ← ψ(v_0)
Dokud existují otevřené vrcholy, opakuj:
    v ← otevřený vrchol s nejmenším f-skóre

    Pokud v = v_G, proveď:
        Vrať D(v_G)

    Pro všechny sousedy w ∈ V vrcholu v opakuj:
        // Našli jsme lepší cestu
        Pokud D(v) + ℓ(v,w) < D(w), proveď:
            D(w) ← D(v) + ℓ(v,w)
            f(w) ← D(v) + ℓ(v,w) + ψ(w)
            stav(w) ← otevřený
    stav(v) ← uzavřený
Vrať D(v_G)
```

Výstup: Vzdálenost v_0 od v_G.

Správnost algoritmu A*

Podobně jako u Dijkstrova algoritmu i zde platí, že při výběru (tj. otevírání) vrcholu v odpovídá $D(v)$ délce nejkratší cesty. Argument pro tuto skutečnost je podobný. Předpokládejme, že otevíráme nějaký vrchol v, přičemž hodnota $D(v)$ je větší, než je skutečná délka nejkratší cesty. Pak existuje nějaký sousední vrchol u vrcholu v, přes nějž nutně vede nejkratší cesta do v, tj. platí $D(u) + \ell(u, v) < D(v)$ (viz obrázek 1.13). Jak to bude s f-skóre vrcholu v? Těžko říci. Co vůbec víme o funkci ψ? Zatím jsme si nespecifikovali žádné vlastnosti, které bychom od ní očekávali. Ukazuje se však, že ψ si libovolně zvolit



Obrázek 1.13: Situace při výběru vrcholu v při heuristice ψ, kdy $D(v)$ neodpovídá délce nejkratší cesty.

nemůžeme. Naše heuritika musí být v jistém smyslu „rozumná“. Pokud se vydáme z vrcholu u po hraně do vrcholu v a z něj poté do cílového vrcholu v_G, určitě výsledná cesta bude alespoň tak dlouhá, jako nejkratší cesta z u do v_G. To samé by mělo platit i pro náš odhad vzdáleností[8], tedy $\psi(u) \leqslant \ell(u,v)+\psi(v)$.

Pokud tedy budeme předpokládat, že ψ splňuje výše zmíněnou vlastnost, musí pak platit následující:

$$f(u) = D(u) + \overbrace{\psi(u)}^{\leqslant \ell(u,v)+\psi(v)} \leqslant \underbrace{D(u)+\ell(u,v)}_{<D(v)} + \psi(v) < D(v) + \psi(v) = f(v).$$

Z toho je ale již vidět, že $f(u) < f(v)$, tedy u musí mít nižší f-skóre než v. Tzn. algoritmus A* by upřednostnil vrchol u před v.

Upozorněme zde na fakt, že náš předpoklad o heuristice je velmi důležitý, neboť pokud bychom si zvolili heuristiku nevhodně, algoritmus již fungovat nebude a obecně může dojít k tomu, že nějaký z vrcholů vybereme „předčasně“.

Poznámka 1.7.2.
- Všimněme si, že Dijkstrův algoritmus je speciálním případem algoritmu A*. Stačí pro všechny vrcholy v položit[9] $\psi(v) = 0$.
- Je dobré dodat, že heuristiku bychom měli být schopni pro každý vrchol spočítat co nejrychleji, nejlépe v konstantním čase, tj. $\mathcal{O}(1)$.

Rozbor časové složitosti zde vynecháme. Je totiž silně závislá na tom, jakou heuristiku ψ si zvolíme. Při vhodné volbě doběhne A* zpravidla rychleji oproti Dijkstrově algoritmu. Může se však také stát, že ψ zvolíme nevhodně (i přesto, že splňuje trojúhelníkovou nerovnost výše), a algoritmus nakonec poběží déle.

Ukázky heuristických funkcí

Zbývá zodpovědět otázku, jakou heuristiku tedy použít? Odpověď není zcela jednoznačná, neboť záleží na situaci. Vraťme se opět k příkladu s bludištěm, kdy začínáme na určitém políčku a snažíme se dostat do cílového, přičemž hráč se smí pohybovat všemy směry vždy o jedno pole. Políčka identifikujeme podle jejich souřadnic, tj. dvojice (x, y).

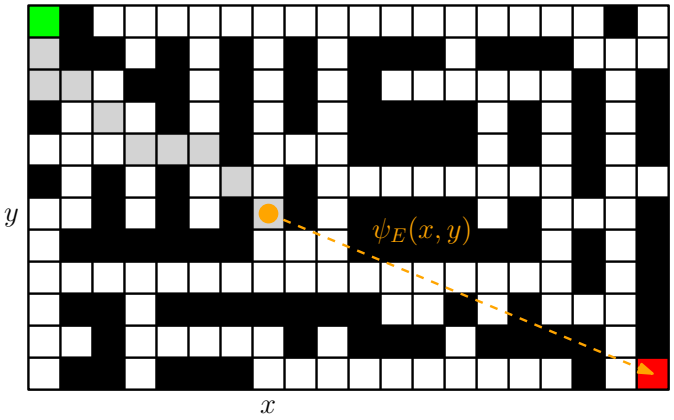
[8] Tomu se obecně říká tzv. *trojúhelníková nerovnost*. Pro libovolnou trojici vrcholů platí $d(u,v) \leqslant d(u,w) + d(w,v)$, kde d značí vzdálenost daných vrcholů v grafu G.

[9] Lze zvolit klidně i obecnou konstantu $c \in \mathbb{R}$, tj. $\psi(v) = c$.

Jednou možností je tzv. *eukleidovská vzdálenost*. Ta je pro libovolnou dvojici bodů v rovině $X = (x_1, y_1)$ a $Y = (x_2, y_2)$ definována jako

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

V případě cílového políčka známe jeho souřadnice, takže můžeme snadno vypočítat eukleidovskou vzdálenost libovolného políčka od cílového. To lze použít jako heuristiku[10], označme si ji ψ_E. Situaci lze sledovat na obrázku 1.14, kde šedě je vyznačena nejkratší cesta do pole se souřadnicemi (x, y). Pokud

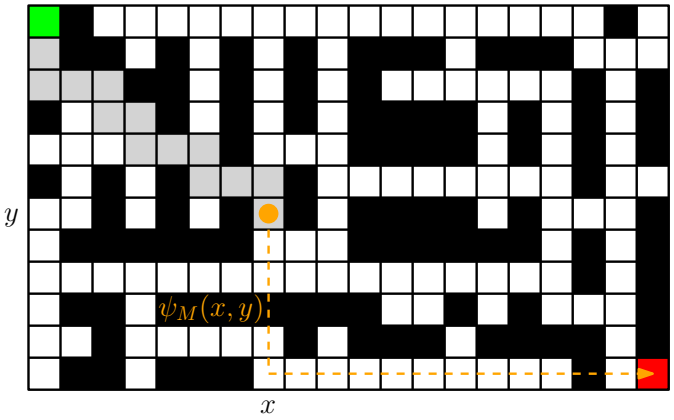


Obrázek 1.14: Příklad bludiště s eukleidovskou heuristikou.

bychom však pracovali s bludištěm, kde se *nelze pohybovat po diagonálách* nebo by nám to pravidla neumožňovala, můžeme výpočet heuristické funkce nahradit něčím jednodušším. V tomto ohledu se pak často využívá tzv. *manhattanská vzdálenost* (označme ψ_M) definovaná vztahem

$$|\, x_1 - x_2 \,| + |\, y_1 - y_2 \,|\,.$$

Stejně jako eukleidovská vzdálenost, i manhattanská vzdálenost splňuje trojúhelníkovou nerovnost.



Obrázek 1.15: Příklad bludiště s manhattnskou heuristikou.

Její výhodou je nižší náročnost při výpočtu (výpočet odmocniny[11] je již trochu pomalejší) a to hlavně z důvodu, že pracujeme vždy s celými čísly. Její nevýhodou však může být, že oproti eukleidovské vzdálenosti nabývá větších hodnot.

[10] Dokonce bychom mohli použít jako heuristiku pouze výraz pod odmocninou, tj. $(x_1 - x_2)^2 + (y_1 - y_2)^2$.

[11] V mnoha případech, např. v některých starších videohrách jako Quake III, se vývojáři snažili vyhýbat výpočtu odmocniny, nebo volili některé sofistikovanější výpočty, které byly rychlé a poskytovaly uspokojivý odhad hledané hodnoty.

Kapitola 2

Algoritmicky těžké problémy

Mnoho problémů lze v informatice řešit pomocí polynomiálních algoritmů, tzn. běžících v čase $\mathcal{O}(n^k)$ pro nějaké prvné k. Např. problém řazení prvků v poli jsme schopni řešit pomocí různých algoritmů, např. *BubbleSortem*, jehož časová složitost je v nejhorším případě $\mathcal{O}(n^2)$, kde n je počet prvků pole. Podobně hledání nejkratší cesty v grafu dokážeme řešit polynomiálně např. Dijkstrovým algoritmem, který i v případě implementace pomocí pole běží v čase $\mathcal{O}(n^2 + m)$, kde n je počet vrcholů a m počet hran[1]. S jistou rezervou lze říci, že polynomiální algoritmy jsou prakticky dobře použitelné.

Bohužel ne vždy je situace takto příznivá. Lze totiž narazit na problémy, na které není známý polynomiální algoritmus a o některých dokonce s jistotou víme, že je v polynomiálním čase řešit nelze. Dobrým příkladem jsou v tomto ohledu např. *Hanojské věže*[2], kde nejlepší algoritmus je exponenciální, tj. $\mathcal{O}(2^n)$, neboť je vždy potřeba exponenciálně mnoho kroků k vyřešení.

Nás budou zajímat ty nejdůležitější problémy z této oblasti, protože mezi nimi lze nalézt zajímavé vztahy, a posléze si uděláme menší ochutnávku z problematiky P a NP.

2.1 Problém SAT

[1] Může se zdát matoucí, že zde figurují dvě proměnné n a m místo jedné, ale počet hran je omezený (nejvýše mohou být každé dva vrcholy spojeny hranou) a to výrazem $n(n+1)/2$, což je polynom. Tedy všechny prezentované grafové algoritmy běží v polynomiálním čase

[2] Pro zájemce podrobnější vysvětlení: `https://en.wikipedia.org/wiki/Tower_of_Hanoi`