13. Nejkratší cesty

Problém hledání nejkratší cesty v (obvykle ohodnoceném orientovaném) grafu provází teorii grafových algoritmů od samých počátků. Základní algoritmy pro hledání cest jsou nedílnou součástí základních kursů programování a algoritmů, my se budeme věnovat zejména různým jejich vylepšením.

Uvažujme tedy nějaký orientovaný graf, jehož každá hrana e je opatřena *délkou* $\ell(e) \in \mathbb{R}$. Množině hran (třeba sledu nebo cestě) pak přiřadíme délku rovnou součtu délek jednotlivých hran.

Pro vrcholy u, v definujeme jejich *vzdálenost* $d(u, v)$ jako nejmenší možnou délku cesty z u do v (jelikož cest je v grafu konečně mnoho, minimum vždy existuje). Pokud z u do v žádná cesta nevede, položíme $d(u, v) := \infty$.

Obvykle se studují následující tři problémy:

- **1-1** neboli **P2PSP** (Point to Point Shortest Path) – chceme nalézt nejkratší cestu z daného vrcholu u do daného vrcholu v. (Pokud je nejkratších cest více, tak libovolnou z nich.)
- **1-n** neboli **SSSP** (Single Source Shortest Paths) – pro daný vrchol u chceme nalézt nejkratší cesty do všech ostatních vrcholů.
- **n-n** neboli **APSP** (All Pairs Shortest Paths) – zajímají nás nejkratší cesty mezi všemi dvojicemi vrcholů.

Překvapivě, tak obecně, jak jsme si uvedené problémy definovali, je neumíme řešit v polynomiálním čase: pro grafy, které mohou obsahovat hrany záporných délek bez jakýchkoliv omezení, je totiž hledání nejkratší cesty NP-těžké (lze na něj snadno převést existenci hamiltonovské cesty). Všechny známé polynomiální algoritmy totiž místo nejkratší cesty hledají nejkratší sled – nijak nekontrolují, zda cesta neprojde jedním vrcholem vícekrát.

Naštěstí pro nás je to v grafech bez cyklů záporné délky totéž: pokud se v nalezeném sledu vyskytne cyklus, můžeme jej „vystřihnout“ a tím získat sled, který není delší a který má méně hran. Každý nejkratší sled tak můžeme upravit na stejně dlouhou cestu. V grafech bez záporných cyklů je tedy jedno, zda hledáme sled nebo cestu; naopak vyskytne-li se záporný cyklus dosažitelný z počátečního vrcholu, nejkratší sled ani neexistuje.

Navíc se nám bude hodit, že každý prefix nejkratší cesty je opět nejkratší cesta. Jinými slovy pokud některá z nejkratších cest z u do v vede přes nějaký vrchol w, pak její část z u do w je jednou z nejkratších cest z u do w. (V opačném případě bychom mohli úsek $u \dots w$ vyměnit za kratší.)

Díky této *prefixové vlastnosti* můžeme pro každý vrchol u sestrojit jeho *strom nejkratších cest* $\mathcal{T}(u)$. To je nějaký podgraf grafu G, který má tvar stromu zakořeněného v u a orientovaného směrem od kořene, a platí pro něj, že pro každý vrchol v je (jediná) cesta z u do v v tomto stromu jednou z nejkratších cest z u do v v původním grafu.

Pozorování: Strom nejkratších cest vždy existuje.

Důkaz: Nechť $u = v_1, \ldots, v_n$ jsou všechny vrcholy grafu G. Indukcí budeme dokazovat, že pro každé i existuje strom $\mathcal{T}_i$, v němž se nacházejí nejkratší cesty z vrcholu u do vrcholů $v_1, \ldots, v_i$. Pro $i = 1$ stačí uvážit strom obsahující jediný vrchol u. Ze stromu $\mathcal{T}_{i-1}$ pak vyrobíme strom $\mathcal{T}_i$ takto: Nalezneme v G nejkratší cestu z u do v_i a označíme z poslední vrchol na této cestě, který se ještě vyskytuje v $\mathcal{T}_{i-1}$. Úsek nejkratší cesty od z do v_i pak přidáme do $\mathcal{T}_{i-1}$ a díky prefixové vlastnosti bude i cesta z u do v_i v novém stromu nejkratší. ♡

Zbývá se dohodnout, v jakém tvaru mají naše algoritmy vydávat výsledek. U problémů typu 1-1 je nejjednodušší vypsat celou cestu, u 1-n můžeme jako výstup vydat strom nejkratších cest z daného počátku (všimněte si, že stačí uvést předchůdce každého vrcholu), u n-n vydáme strom nejkratších cest pro každý ze zdrojových vrcholů.

Často se ovšem ukáže, že podstatná část problému se skrývá v samotném výpočtu vzdáleností a sestrojení předchůdců je triviálním rozšířením algoritmu. Budeme tedy obvykle jen počítat vzdálenosti a samotnou rekonstrukci cest ponecháme čtenáři jako snadné cvičení.

Relaxační algoritmus

Začněme problémem 1-n a označme u výchozí vrchol. Většina známých algoritmů funguje tak, že pro každý vrchol v udržují ohodnocení $h(v)$, které v každém okamžiku odpovídá délce nějakého sledu z u do v. Postupně toto ohodnocení upravují, až se z něj stane vzdálenost $d(u, v)$ a algoritmus se může zastavit.

Vhodnou operací pro vylepšování ohodnocení je takzvaná *relaxace.* Vybereme si nějaký vrchol v a pro všechny jeho sousedy w spočítáme $h(v) + \ell(v, w)$, tedy délku sledu, který vznikne rozšířením aktuálního sledu do v o hranu (v, w). Pokud je tato hodnota menší než $h(w)$, tak jí $h(w)$ přepíšeme.

Abychom zabránili opakovaným relaxacím téhož vrcholu, které nic nezmění, budeme rozlišovat tři stavy vrcholů: *neviděn* (ještě jsme ho nenavštívili), *otevřen* (změnilo se ohodnocení, časem chceme relaxovat) a *uzavřen* (už jsme relaxovali a není potřeba znovu).

Náš algoritmus bude fungovat následovně:

1. $h(*) \leftarrow \infty$, $h(u) \leftarrow 0$.
2. $stav(*) \leftarrow$ *neviděn*, $stav(u) \leftarrow$ *otevřen*.
3. Dokud existují otevřené vrcholy, opakujeme:
4. $\quad v \leftarrow$ libovolný otevřený vrchol.
5. $\quad stav(v) \leftarrow$ *uzavřen*.
6. $\quad$ Relaxujeme v:
7. $\quad\quad$ Pro všechny hrany vw opakujeme:
8. $\quad\quad\quad$ Je-li $h(w) > h(v) + \ell(v, w)$:
9. $\quad\quad\quad\quad h(w) \leftarrow h(v) + \ell(v, w)$.
10. $\quad\quad\quad\quad stav(w) \leftarrow$ *otevřen*.

11. Vrátíme výsledek $d(u, v) = h(v)$ pro všechna v.

Podobně jako u minimálních koster, i zde se jedná o meta-algoritmus, protože v kroku 4 nespecifikuje, který z otevřených vrcholů vybírá. Přesto ale můžeme dokázat několik zajímavých tvrzení, která na konkrétním způsobu výběru nezávisejí.

Věta: Spustíme-li meta-algoritmus na graf bez záporných cyklů, pak:

1) Ohodnocení $h(v)$ vždy odpovídá délce nějakého sledu z u do v.
2) $h(v)$ dokonce odpovídá délce nějaké cesty z u do v.
3) Algoritmus se vždy zastaví.
4) Po zastavení jsou označeny jako uzavřené právě ty vrcholy, které jsou dosažitelné z u.
5) Po zastavení mají konečné $h(v)$ právě všechny uzavřené vrcholy.
6) Pro každý dosažitelný vrchol je na konci $h(v)$ rovno $d(u, v)$.

Důkaz:

1) Dokážeme indukcí podle počtu kroků algoritmu.
2) Stačí rozmyslet, v jaké situaci by vytvořený sled mohl obsahovat cyklus.
3) Cest, a tím pádem i možných hodnot $h(v)$ pro každý v, je konečně mnoho.
4) Implikace $\Rightarrow$ je triviální, pro $\Leftarrow$ stačí uvážit neuzavřený vrchol, který je dosažitelný z u cestou o co nejmenším počtu hran.
5) $h(v)$ nastavujeme na konečnou hodnotu právě v okamžicích, kdy se vrchol stává otevřeným. Každý otevřený vrchol je časem uzavřen.
6) Kdyby tomu tak nebylo, vyberme si ze „špatných“ vrcholů v takový, pro nějž obsahuje nejkratší cesta z u do v nejmenší možný počet hran. Vrchol v je zajisté různý od u, takže má na této cestě nějakého předchůdce w. Přitom w už musí být ohodnocen správně a relaxace, která mu toto ohodnocení nastavila, ho musela prohlásit za otevřený. Jenže každý otevřený vrchol je později uzavřen, takže w poté musel být ještě alespoň jednou relaxován, což muselo snížit $h(v)$ na správnou vzdálenost. ♡

Dokázali jsme tedy, že meta-algoritmus pro libovolnou implementaci kroku 4 spočítá správné vzdálenosti.

Cvičení:

- Nechť do algoritmu doplníme udržování předchůdců tak, že v kroku 9 přenastavíme předchůdce vrcholu w na vrchol v. Dokažte, že předchůdci dosažitelných vrcholů budou tvořit strom a že tento strom bude stromem nejkratších cest z vrcholu u.
- Dokažte, že pro graf, v němž je alespoň jeden záporný cyklus dosažitelný z počátečního vrcholu, se algoritmus nezastaví a ohodnocení všech vrcholů na cyklu postupně klesnou libovolně hluboko. Nedosažitelné záporné cykly chod algoritmu samozřejmě nijak neovlivní.

Bellmanův-Fordův-Mooreův algoritmus

Bellman [1], Ford [5] a Moore objevili nezávisle na sobě algoritmus (říkejme mu BFM), který lze v řeči našeho meta-algoritmu formulovat takto: Otevřené vrcholy udržujeme ve frontě (vždy relaxujeme vrchol na počátku fronty, nově otevírané zařazujeme na konec). Co toto pravidlo způsobí?

Věta: Časová složitost algoritmu BFM činí $\mathcal{O}(nm)$.

Důkaz: Běh algoritmu rozdělíme na fáze. Nultá fáze sestává z vložení vrcholu u do fronty. V $(i+1)$-ní fázi relaxujeme ty vrcholy, které byly do fronty uloženy během i-té fáze.

Jelikož relaxace vrcholu trvá lineárně se stupněm vrcholu a každý vrchol se dané fáze účastní nejvýše jednou, trvá jedna fáze $\mathcal{O}(m)$. Zbývá ukázat, že fází provedeme nejvýše n.

Indukcí dokážeme, že na konci i-té fáze je každé ohodnocení $h(v)$ shora omezeno délkou nejkratšího z uv-sledů o nejvýše i hranách. Pro $i = 0$ to triviálně platí. Uvažujme nyní vrchol v na konci $(i+1)$-ní fáze a nějaký nejkratší uv-sled P o $i+1$ hranách. Označme wv poslední hranu tohoto sledu a P' sled bez této hrany, který tedy má délku i. Podle indukčního předpokladu je na konci i-té fáze $h(w) \leq \ell(P')$. Tuto hodnotu získalo $h(w)$ nejpozději v i-té fázi, při tom jsme vrchol w otevřeli, takže jsme ho nejpozději v $(i+1)$-ní fázi zavřeli a relaxovali. Po této relaxaci je ovšem $h(v) \leq h(w) + \ell(w,v) \leq \ell(P') + \ell(w,v) = \ell(P)$. ♡

Cvičení:

- Ukažte, že asymptoticky stejné časové složitosti by dosáhl algoritmus, který by vrcholy očísloval $v_1, \ldots, v_n$ a opakovaně by je v tomto pořadí relaxoval tak dlouho, dokud by se ohodnocení měnila.
- Jak algoritmus upravit, aby v i-té fázi spočítal minimální délky sledů o právě i hranách?
- Jak lze algoritmus BFM využít k nalezení záporného cyklu?

Dijkstrův algoritmus

Pokud jsou všechny délky hran nezáporné, můžeme použít efektivnější pravidlo pro výběr vrcholu navržené Dijkstrou [3]. To říká, že vždy relaxujeme ten z otevřených vrcholů, jehož ohodnocení je nejmenší.

Věta: Dijkstrův algoritmus uzavírá vrcholy v pořadí podle neklesající vzdálenosti od u a každý dosažitelný vrchol uzavře právě jednou.

Důkaz: Indukcí dokážeme, že v každém okamžiku mají všechny uzavřené vrcholy ohodnocení menší nebo rovné ohodnocením všech otevřených vrcholů. Na počátku to jistě platí. Nechť nyní uzavíráme vrchol v s minimálním $h(v)$ mezi otevřenými. Během jeho relaxace nemůžeme žádnou hodnotu snížit pod $h(v)$, jelikož v grafu s nezápornými hranami je $h(v) + \ell(v,w) \geq h(v)$. Hodnota zbývajících otevřených vrcholů tedy neklesne pod hodnotu tohoto nově uzavřeného. Hodnoty dříve uzavřených vrcholů se nemohou nijak změnit. ♡

Přímočará implementace Dijkstrova algoritmu by tedy pokaždé v čase $\mathcal{O}(n)$ vybrala otevřený vrchol s nejmenším ohodnocením, v čase $\mathcal{O}(n)$ ho relaxovala a toto by se opakovalo nejvýše n-krát. Algoritmus by tudíž doběhl v čase $\mathcal{O}(n^2)$, což je pro husté grafy zajisté optimální. Zkusíme nyní zrychlit výpočet na řídkých grafech.

Všechny relaxace trvají dohromady $\mathcal{O}(\sum_v \deg(v)) = \mathcal{O}(m)$, takže úzkým hrdlem je vybírání minima. Použijeme pro něj vhodnou datovou strukturu, v níž budeme udržovat množinu všech otevřených vrcholů spolu s jejich ohodnoceními. Od datové struktury potřebujeme, aby uměla operace *Insert* (vložení vrcholu), *ExtractMin* (nalezení a smazání minima) a *Decrease* (snížení hodnoty vrcholu). První dvě operace přitom voláme nejvýše n-krát a operaci *Decrease* nejvýše m-krát. Celý algoritmus tedy doběhne v čase

$$\mathcal{O}(nT_I(n) + nT_E(n) + mT_D(n)),$$

kde $T_I(n)$, $T_E(n)$ a $T_D(n)$ jsou časové složitosti jednotlivých operací na struktuře o nejvýše n prvcích (stačí amortizovaně).

Jaké možnosti máme pro volbu struktury?

- *pole* – *Insert* a *Decrease* stojí konstantu, *ExtractMin* trvá $\mathcal{O}(n)$, celkem tedy $\mathcal{O}(n^2)$.
- *(binární) halda* – všechny tři operace umíme provést v čase $\mathcal{O}(\log n)$, takže celkem $\mathcal{O}(m \log n)$. To je pro husté grafy horší, pro řídké lepší.
- *k-regulární halda* – pokud haldu upravíme tak, že každý prvek bude mít až k synů, hloubka haldy klesne na $\mathcal{O}(\log_k n)$. Operace „vybublávající“ prvky směrem nahoru, což je *Insert* a *Decrease*, se zrychlí na $\mathcal{O}(\log_k n)$. Ovšem *ExtractMin* potřebuje zkoumat všechny syny každého navštíveného prvku, takže se zpomalí na $\mathcal{O}(k \log_k n)$.

  Celková složitost tedy vyjde $\mathcal{O}(nk \log_k n + m \log_k n)$. Oba členy se vyrovnají pro $k = m/n$, čímž získáme $\mathcal{O}(m \log_{m/n} n)$. Člen $\log_{m/n} n$ je přitom $\mathcal{O}(1)$, kdykoliv je $m \geq n^{1+\varepsilon}$ pro nějaké $\varepsilon > 0$, takže pro dostatečně husté grafy jsme získali lineární algoritmus.

  (Všimněte si, že pro $m \approx n^2$ algoritmus zvolí $k \approx n$, takže halda degeneruje na jediné patro, tedy na pole, které se opravdu ukázalo být optimální volbou pro husté grafy.)
- *Fibonacciho halda* [6] – *Insert* a *Decrease* stojí $\mathcal{O}(1)$, *ExtractMin* má složitost $\mathcal{O}(\log n)$ [vše amortizovaně]. Dijkstrův algoritmus proto doběhne v čase $\mathcal{O}(m + n \log n)$. To je lineární pro grafy s hustotou $\Omega(\log n)$. Téže složitosti operací dosahují i jiné, méně známé haldy [8, 4], které mohou být v praxi výrazně rychlejší.
- *Monotónní haldy* – můžeme použít nějakou jinou haldu, která využívá toho, že posloupnost odebíraných prvků je neklesající. Pro celá čísla na RAMu to může být například Thorupova halda [10] se složitostí $\mathcal{O}(\log \log n)$ u operace *ExtractMin* a $\mathcal{O}(1)$ u ostatních operací. Dijkstra tedy běží v $\mathcal{O}(m + n \log \log n)$.
- *Datové struktury pro omezené universum* – prozkoumáme vzápětí.

Cvičení:

- Najděte příklad grafu se zápornými hranami (ale bez záporných cyklů), na kterém Dijkstrův algoritmus selže.
- Rozmyslete si, že pokud nevyužijeme nějaké speciální vlastnosti čísel (celočíselnost, omezený rozsah), je mez $\mathcal{O}(m + n \log n)$ nejlepší možná, protože Dijkstrovým algoritmem můžeme třídit n-tici čísel. Thorup dokonce dokázal [11], že z každého třídícího algoritmu se složitostí $\mathcal{O}(nT(n))$ můžeme odvodit haldu se složitostí mazání $\mathcal{O}(T(n))$.
- Jsou-li délky hran celočíselné, můžeme se na Dijkstrův algoritmus dívat i takto: Představme si, že každou hranu nahradíme cestou tvořenou příslušným počtem hran jednotkové délky a na vzniklý neohodnocený graf spustíme prohledávání do šířky. To samozřejmě vydá správný výsledek, ale poměrně pomalu, protože bude většinu času trávit posouváním vlny „uvnitř“ původních hran. Můžeme si tedy pro každou původní hranu nařídit „budík“, který nám řekne, za kolik posunutí vlny dospějeme na její konec. Dokažte, že tento algoritmus je ekvivalentní s Dijkstrovým.

Celočíselné délky

Uvažujme nyní grafy, v nichž jsou všechny délky hran nezáporná celá čísla omezená nějakou konstantou L. Všechny vzdálenosti jsou tedy omezeny číslem nL, takže nám stačí datová struktura schopná uchovávat takto omezená celá čísla.

Použijeme jednoduchou přihrádkovou strukturu: pole indexované hodnotami od 0 do nL, jeho i-tý prvek bude obsahovat seznam vrcholů, jejichž ohodnocení je rovno i. Operace *Insert* a *Decrease* zvládneme v konstantním čase, budeme-li si u každého prvku pamatovat jeho polohu v seznamu. Operace *ExtractMin* potřebuje najít první neprázdnou přihrádku, ale jelikož víme, že posloupnost odebíraných minim je monotónní, stačí hledat od místa, kde se hledání zastavilo minule. Všechna hledání přihrádek tedy zaberou dohromady $\mathcal{O}(nL)$ a celý Dijkstrův algoritmus bude trvat $\mathcal{O}(nL + m)$.

Prostorová složitost $\mathcal{O}(nL + m)$ je nevalná, ale můžeme ji jednoduchým trikem snížit: Všimneme si, že všechna konečná ohodnocení vrcholů nemohou být větší než aktuální minimum zvětšené o L. Jinými slovy všechny neprázdné přihrádky se nacházejí v úseku pole dlouhém $L + 1$, takže stačí indexovat modulo $L + 1$. Pouze si musíme dávat pozor, abychom správně poznali, kdy se struktura vyprázdnila, což zjistíme například pomocí počítadla otevřených vrcholů. Čas si asymptoticky nezhoršíme, prostor klesne na $\mathcal{O}(L + m)$.

Podobný trik můžeme použít i pro libovolnou jinou datovou strukturu: rozsah čísel rozdělíme na „okna“ velikosti L (v i-tém okně jsou čísla $iL, iL + 1, \ldots, (i + 1)L - 1$). V libovolné chvíli mohou tedy být neprázdná nejvýše dvě okna. Stačí nám proto pořídit si dvě struktury pro ukládání čísel v rozsahu $0, \ldots, L - 1$ – jedna z nich bude reprezentovat aktuální okno (to, v němž leželo minulé minimum), druhá okno bezprostředně následující. Minimum mažeme z první struktury; pokud už je prázdná, obě struktury prohodíme. Operace *Insert* podle hodnoty určí, do které struktury se

má vkládat. S operací *Decrease* je to složitější – hodnota z vyšší struktury může přeskočit do té nižší, ale v takovém případě ji ve vyšší struktuře vymažeme (to je *Decrease* na $-\infty$ následovaný *ExtractMin*em) a do té nižší vložíme. To se každému prvku může přihodit nejvýše jednou, takže stále platí, že se každý prvek účastní $\mathcal{O}(1)$ vložení a $\mathcal{O}(1)$ extrakcí minima.

Ukázali jsme tedy, že pro naše potřeby postačuje struktura schopná uchovávání čísel menších nebo rovných L.

Nabízí se použít van Emde-Boasův strom z kapitoly o výpočetních modelech. Ten dosahuje složitosti $\mathcal{O}(\log \log L)$ pro operace *Insert* a *ExtractMin*, operaci *Decrease* musíme překládat na *Delete* a *Insert*. Celková složitost Dijkstrova algoritmu vyjde $\mathcal{O}(L + m \log \log L)$, přičemž čas L spotřebujeme na inicializaci struktury (té se lze za jistých podmínek vyhnout, viz zmíněná kapitola).

Vraťme se ale ještě k využití přihrádek...

Víceúrovňové přihrádky

Podobně jako u třídění čísel, i zde se vyplácí stavět přihrádkové struktury víceúrovňově (původně popsáno Goldbergem a Silversteinem [7]). Jednotlivé hodnoty budeme zapisovat v soustavě o základu B, který zvolíme jako nějakou mocninu dvojky, abychom mohli s číslicemi tohoto zápisu snadno zacházet pomocí bitových operací. Každé číslo tedy zabere nejvýše $d = 1 + \lfloor \log_B L \rfloor$ číslic; pokud bude kratší, doplníme ho zleva nulami.

Nejvyšší patro struktury bude tvořeno polem B přihrádek, v i-té z nich budou uložena ta čísla, jejichž číslice nejvyššího řádu je rovna i. Za *aktivní* prohlásíme tu přihrádku, která obsahuje aktuální minimum. Přihrádky s menšími indexy jsou prázdné a zůstanou takové až do konce výpočtu, protože halda je monotónní. Pokud v přihrádce obsahující minimum bude více prvků, budeme ji rozkládat podle druhého nejvyššího řádu na dalších B přihrádek atd. Celkem tak vznikne až d úrovní.

Struktura bude obsahovat následující data:

- Parametry L, B a d.
- Pole úrovní (nejvyšší má číslo $d - 1$, nejnižší 0), každá úroveň je buďto prázdná (a pak jsou prázdné i všechny nižší), nebo obsahuje pole U_i o B přihrádkách a v každé z nich seznam prvků. Pole úrovní používáme jako zásobník, udržujeme si číslo nejnižší neprázdné úrovně.
- Hodnotu μ předchozího odebraného minima.

Operace *Insert* vloží hodnotu do nejhlubší možné přihrádky. Podívá se tedy na nejvyšší úroveň: pokud hodnota patří do přihrádky, která není aktivní, vloží ji přímo. Jinak přejde o úroveň níže a zopakuje stejný postup. To vše lze provést v konstantním čase: stačí se podívat, jaký je nejvyšší jedničkový bit ve XORu nové hodnoty s číslem μ (opět viz kapitola o výpočetních modelech), a tím zjistit číslo úrovně, kam hodnota patří.

Pokud chceme provést *Decrease*, odstraníme hodnotu z přihrádky, ve které se právě nachází (polohu si můžeme u každé hodnoty pamatovat zvlášť), a znovu ji

vložíme.

Většinu práce samozřejmě přenecháme funkci *ExtractMin*. Ta začne prohledávat nejnižší obsazenou úroveň od aktivní přihrádky dál (to, která přihrádka je na které úrovni aktivní, poznáme z číslic hodnoty μ). Pokud přihrádky na této úrovni dojdou, prázdnou úroveň zrušíme a pokračujeme o patro výše.

Jakmile najdeme neprázdnou přihrádku, nalezneme v ní minimum a to se stane novým μ. Pokud v přihrádce nebyly žádné další prvky, skončíme. V opačném případě zbývající prvky rozprostřeme do přihrádek na bezprostředně nižší úrovni, kterou tím založíme.

Čas strávený hledáním minima můžeme rozdělit na několik částí:

- $\mathcal{O}(B)$ na inicializaci nové úrovně – to naúčtujeme prvku, který jsme právě mazali;
- hledání neprázdných přihrádek – prozkoumání každé prázdné přihrádky naúčtujeme jejímu vytvoření, což se rozpustí v $\mathcal{O}(B)$ na inicializaci úrovně;
- zrušení úrovně – opět naúčtujeme jejímu vzniku;
- rozhazování prvků do přihrádek – jelikož prvky v hierarchii přihrádek putují během operací pouze doleva a dolů (jejich hodnoty se nikdy nezvětšují), klesne každý prvek nejvýše d-krát, takže stačí, když na všechna rozhazování přispěje časem $\mathcal{O}(d)$;
- hledání minima – minimum naúčtujeme smazanému prvku, ostatní prvky, které jsme museli projít, naúčtujeme jejich rozhazování.

Stačí tedy, aby každý prvek při *Insert*u zaplatil čas $\mathcal{O}(B + d)$ a jak *Decrease*, tak *ExtractMin* budou mít konstantní amortizovanou složitost. Dijkstrův algoritmus pak poběží v $\mathcal{O}(m + n(B + d))$.

Zbývá nastavit parametry tak, abychom minimalizovali výraz $B + d = B + \log L / \log B$. Vhodná volba je $B = \log L / \log\log L$. Při ní platí

$$\frac{\log L}{\log B} = \frac{\log L}{\log\left(\log L / \log\log L\right)} = \frac{\log L}{\log\log L - \log\log\log L} = \Theta(B).$$

Tehdy Dijkstra vydá výsledek v čase $\mathcal{O}(m + n \cdot \frac{\log L}{\log\log L})$.

HOT Queue – kombinace přihrádek s haldou

Cherkassky, Goldberg a Silverstein [2] si všimli, že ve víceúrovňových přihrádkových strukturách platíme příliš mnoho za úrovně, ve kterých se za dobu jejich existence objeví jen malé množství prvků. Navrhli tedy ukládat prvky z „malých“ úrovní do společné haldy. Výsledné struktuře se říká HOT (Heap-on-Top) Queue. My si předvedeme její upravenou variantu (v té původní se skrývá několik chyb).

Pořídíme si haldu, řekněme Fibonacciho, a navíc ke každé úrovni počítadlo udávající, kolik prvků z této úrovně jsme uložili do haldy. Dokud toto počítadlo nepřeroste nějaký parametr H, přihrádky nebudeme zakládat a prvky poputují do haldy. Čas $\mathcal{O}(B)$ na založení úrovně a její procházení proto můžeme rozpočítat mezi H prvků, které se musely na dané úrovni objevit, než byla rozpřihrádkována. (Povšimněte

si, že počítadlo nikdy nesnižujeme, pouze ho vynulujeme, když je úroveň zrušena. Proto vůbec nemusí odpovídat skutečnému počtu prvků z příslušné úrovně, které jsou právě uloženy v haldě. To ovšem vůbec nevadí – počítadlo pouze hlídá, aby se úroveň nevytvořila příliš brzy, tedy abychom měli dost prvků k rozúčtování času.)

Proměnnou μ necháme ukazovat na místo, kde jsme se při hledání minima v přihrádkách zastavili. Současné globální minimum struktury může být nižší, nachází-li se minimum zrovna v haldě. Stále je však zaručeno, že před μ se nenachází žádná neprázdná přihrádka.

Operace budou fungovat takto:

Insert:

1. Spočítáme, do které úrovně i má prvek padnout (bitovými operacemi).
2. Pokud je počítadlo této úrovně menší než H, zvýšíme ho, vložíme prvek do haldy a skočíme.
3. Nebyly-li ještě pro tuto úroveň založeny přihrádky, vyrobíme prázdné.
4. Vložíme prvek do příslušné přihrádky.

Decrease:

1. Pokud se prvek nachází v haldě (to si u každého prvku pamatujeme), provedeme *Decrease* v haldě a skončíme.
2. Smažeme prvek z jeho přihrádky a provedeme *Insert* s novou hodnotou.

ExtractMin:

1. Dokud je μ menší než aktuální minimum haldy, opakujeme:
2. Najdeme přihrádku odpovídající hodnotě μ.
3. Je-li tato přihrádka prázdná, přejdeme na další (upravíme μ). Jsme-li na konci úrovně, zrušíme ji, vynulujeme její počítadlo a pokračujeme o úroveň výš.
4. Není-li prázdná, rozprostřeme ji o úroveň níž (stejným způsobem jako při *Insert*u, takže prvních H prvků vložíme do haldy).
5. Smažeme minimum z haldy a vrátíme ho jako výsledek.

Pustíme se do analýzy složitosti. Jako parametry si zvolíme počet hladin d (takže počet přihrádek B na jedné úrovni je roven $L^{1/d}$) a „haldovou konstantu“ H.

Nejprve si všimneme, že než počítadlo nějaké úrovně vynulujeme, jsou bezpečně z haldy pryč všechny prvky patřící do této úrovně. Celkem se tedy v haldě vyskytuje nejvýše $\mathcal{O}(dH)$ prvků. *ExtractMin* v haldě proto trvá $\mathcal{O}(\log(dH))$, ostatní haldové operace $\mathcal{O}(1)$.

Nyní rozúčtujeme čas operací mezi jednotlivé prvky (opět si vše předplatíme v *Insert*u a ostatní operace poběží v amortizovaně konstantním čase):

- Každý prvek může být nejvýše jednou za dobu svého života vložen do haldy a nejvýše jednou z ní vyjmut. Na obojí dohromady přispěje $\mathcal{O}(\log dH)$.

- Prvek se účastní nejvýše d přehození do nižší úrovně. Pokud byl přihozen do haldy, už tam setrvá, jinak pokaždé zaplatí $\mathcal{O}(1)$ za zařazení do přihrádky, celkem tedy $\mathcal{O}(d)$ na prvek.
- Vytvoření, projití a smazání přihrádek na jedné úrovni nastane až tehdy, co bylo H prvků patřících do této úrovně vloženo do haldy. Stačí tedy, aby každý prvek přispěl časem $\mathcal{O}(B/H) = \mathcal{O}(L^{1/d}/H)$.

Každý prvek tedy platí $\mathcal{O}(d + \log dH + L^{1/d}/H) = \mathcal{O}(d + \log H + L^{1/d}/H)$. Pojďme najít nastavení parametrů, které tento výraz minimalizuje. Nejprve zvolme H tak, aby se d vyrovnalo s $L^{1/d}/H$. Tedy $H = L^{1/d}/d$. Celý výraz tím zjednodušíme na $\mathcal{O}(d + \log(L^{1/d}/d)) = \mathcal{O}(d + 1/d \cdot \log L - \log d) = \mathcal{O}(d + 1/d \cdot \log L)$. Oba sčítance volbou $d = \sqrt{\log L}$ vyrovnáme.

HOT Queue tedy zvládne *Insert* s amortizovanou časovou složitostí $\mathcal{O}(\sqrt{\log L})$ a ostatní operace v čase amortizovaně konstantním. Použijeme-li tuto strukturu v Dijkstrově algoritmu, spočte vzdálenosti v čase $\mathcal{O}(m + n\sqrt{\log L})$.

Dinicův algoritmus

Zajímavé vylepšení Dijkstrova algoritmu navrhl Dinic. Všiml si, že pokud je každá hrana dlouhá alespoň δ, můžeme uzavírat nejen vrchol s minimálním ohodnocením μ, ale i všechny s ohodnocením menším než $\mu + \delta$.

Pro takto upravený Dijkstrův algoritmus bude stále platit, že při uzavírání vrcholu odpovídá ohodnocení skutečné vzdálenosti, takže uzavřené vrcholy již nejsou znovu otevírány.

O monotonii vzdáleností jsme sice přišli, ale v přihrádkové struktuře nebo haldě můžeme klíče nahradit hodnotami $h'(v) := \lfloor h(v)/\delta \rfloor$. Tyto hodnoty se totiž chovají monotónně a vrcholy se stejným $h'(v)$ můžeme libovolně zaměňovat.

Kteroukoli z popsaných přihrádkových struktur můžeme tedy použít, pouze v rozboru časové složitosti nahradíme L výrazem L/δ. Tento přístup přitom funguje i pro neceločíselné délky hran, pouze potřebujeme mít předem k dispozici netriviální dolní odhad na všechny délky.

Potenciály

Viděli jsme, že v grafech s nezápornými délkami hran se nejkratší cesty hledají snáze. Nabízí se nalézt nějakou transformaci, která by dovedla délky hran upravit tak, aby byly nejkratší cesty zachovány (samozřejmě ne jejich délky, ale alespoň to, které cesty jsou nejkratší). Nabízí se následující fyzikální inspirace:

Definice: *Potenciál* budeme říkat libovolné funkci $\psi : V \to \mathbb{R}$. Pro každý potenciál zavedeme *redukované délky hran* $\ell_\psi(u, v) := \ell(u, v) + \psi(u) - \psi(v)$. Potenciál nazveme *přípustný,* pokud žádná hrana nemá zápornou redukovanou délku.

Pozorování: Pro redukovanou délku libovolné cesty P z u do v platí: $\ell_\psi(P) = \ell(P) + \psi(u) - \psi(v)$.

Důkaz: Nechť cesta P prochází přes vrcholy $u = w_1, \ldots, w_k = v$. Potom:

$$\ell_\psi(P) = \sum_i \ell_\psi(w_i, w_{i+1}) = \sum_i \left(\ell(w_i, w_{i+1}) + \psi(w_i) - \psi(w_{i+1})\right).$$

Tato suma je ovšem teleskopická, takže z ní zbude

$$\sum_i \ell(w_i, w_{i+1}) + \psi(w_1) - \psi(w_k) = \ell(P) + \psi(u) - \psi(v).$$

♡

Důsledek: Potenciálovou redukcí se délky všech cest mezi u a v změní o tutéž konstantu, takže struktura nejkratších cest zůstane nezměněna.

Zbývá přijít na to, kde si obstarat nějaký přípustný potenciál. Přidejme do grafu nový vrchol z, přiveďme z něj hrany nulové délky do všech ostatních vrcholů a označme $\psi(v)$ vzdálenost ze z do v v tomto grafu. Aby byl tento potenciál přípustný, musí pro každou hranu uv platit $\ell_\psi(u,v) = \ell(u,v) + \psi(u) - \psi(v) \geq 0$. Tuto nerovnost můžeme upravit na $\ell(u,v) + d(z,u) - d(z,v) \geq 0$, čili $d(z,u) + \ell(u,v) \geq d(z,v)$, což je ale obyčejná trojúhelníková nerovnost pro vzdálenost v grafu, která jistě platí.

Jedním výpočtem nejkratších cest v grafu se zápornými hranami (třeba algoritmem BFM) tedy dokážeme spočítat potenciál, který nám poslouží k redukci všech hran na nezáporné délky. To nám samozřejmě nepomůže, chceme-li jednorázově hledat nejkratší cestu, ale může nám to výrazně zjednodušit další, složitější práci s grafem. Jak uvidíme v příští kapitole, můžeme tak například nalézt vzdálenosti mezi všemi dvojicemi vrcholů v čase $\mathcal{O}(nm + n^2 \log n)$.

Na závěr tohoto oddílu dokážeme jedno pomocné tvrzení o potenciálech, které nám pomůže v konstrukci algoritmů typu 1-1:

Lemma: Pokud f a g jsou přípustné potenciály, pak jsou jimi i:

1. konvexní kombinace f a g, tedy $\alpha f + \beta g$ pro libovolné $\alpha, \beta \geq 0, \alpha + \beta = 1$;
2. $\min(f, g)$;
3. $\max(f, g)$.

Důkaz: Buď uv hrana. Potom z definice přípustnosti platí $\ell(u,v) \geq f(v) - f(u)$ a $\ell(u,v) \geq g(v) - g(u)$. Jednotlivé části tvrzení dokážeme takto:

1. Pokud obě strany nerovnosti pro f vynásobíme konstantou α, u nerovnosti pro g konstantou β a obě nerovnosti sečteme, dostaneme:

   $$(\alpha + \beta) \cdot \ell(u,v) \geq (\alpha f(v) + \beta g(v)) - (\alpha f(u) + \beta g(u)),$$

   což je přesně požadovaná nerovnost pro přípustnost funkce $\alpha f + \beta g$.
2. Označme $h := \min(f, g)$. Nechť bez újmy na obecnosti $f(u) \leq g(u)$. Pokud také platí $f(v) \leq g(v)$, shoduje se minimum s funkcí f a není co dokazovat. V opačném případě je $h(u) = f(u)$, $h(v) = g(v)$. Tehdy si stačí všimnout, že $h(v) - h(u) = g(v) - f(u) < f(v) - f(u) \leq \ell(u,v)$, takže potenciál h je přípustný.
3. Dokážeme analogicky. ♡

Algoritmy pro problém typu 1-1

Zaměřme se nyní na případ, kdy chceme hledat nejkratší cesty mezi zadanou dvojicí vrcholů s, t. Obvykle se i v této situaci používají algoritmy 1-n (SSSP) a

v obecném případě ani není efektivnější řešení známo. Existuje ovšem velké množství heuristik, kterými lze obvykle výpočet zrychlit. Některé z nich si předvedeme na Dijkstrově algoritmu.

Dijkstrův algoritmus ve své klasické podobě neví vůbec nic o cílovém vrcholu a prohledá celý graf. Hned se nabízí využít toho, že od okamžiku uzavření kteréhokoliv vrcholu se již jeho ohodnocení nezmění. Pokud tedy uzavřeme cílový vrchol, můžeme se zastavit.

Jakou část grafu prohledáváme teď? V metrice dané vzdálenostmi v grafu je to nejmenší koule se středem ve vrcholu u, která obsahuje nejkratší cestu (dobře se to představuje na hledání v silniční síti na rovinné mapě).

Také můžeme spustit prohledávání z obou konců zároveň, tedy zkombinovat hledání od s v původním grafu s hledáním od t v grafu s obrácenou orientací hran. Oba postupy můžeme libovolně střídat a zastavíme se v okamžiku, kdy jsme jeden vrchol uzavřeli v obou směrech. Pozor ovšem na to, že součet obou ohodnocení tohoto vrcholu nemusí být roven $d(v,u)$ – zkuste vymyslet protipříklad. Nejkratší cesta ještě může vypadat tak, že přechází po nějaké hraně mezi vrcholem uzavřeným v jednom směru a vrcholem uzavřeným ve směru druhém (ponechme bez důkazu). Stačí tedy během relaxace zjistit, zda je konec hrany uzavřený v opačném směru prohledávání, a pokud ano, započítat cestu do průběžného minima.

Obousměrný Dijkstrův algoritmus projde sjednocení nějaké koule okolo s s nějakou koulí okolo t, které obsahuje nejkratší cestu. Průměry koulí přitom závisí na tom, jak budeme během výpočtu střídat oba směry prohledávání. V nejhorším případě samozřejmě můžeme prohledat celý graf.

Algoritmus A*

V umělé inteligenci se často pro hledání nejkratší cesty v rozsáhlých grafech (obvykle stavových prostorech úloh) používá algoritmus nazývaný A* [9]. Jedná se o modifikaci Dijkstrova algoritmu, která využívá heuristickou funkci pro dolní odhad vzdálenosti do cíle; označme si ji $\psi(v)$. V každém kroku pak uzavíra vrchol v s nejmenším možným součtem $h(v)+\psi(v)$ aktuálního ohodnocení s heuristikou.

Intuice za tímto algoritmem je jasná: pokud víme, že nějaký vrchol je blízko od počátačního vrcholu s, ale bude z něj určitě daleko do cíle t, zatím ho odložíme a budeme zkoumat nadějnější varianty.

Heuristika se přitom volí podle konkrétního problému – např. hledáme-li cestu v mapě, můžeme použít vzdálenost do cíle vzdušnou čarou.

Je u tohoto algoritmu zaručeno, že vždy najde nejkratší cestu? Na to nám dá odpověď teorie potenciálů:

Věta: Běh algoritmu A* odpovídá průběhu Dijkstrova algoritmu na grafu redukovaném potenciálem $-\psi$. Přesněji, pokud označíme h^* aktuální ohodnocení v A* a h aktuální ohodnocení v synchronně běžícím Dijkstrovi, bude vždy platit $h(v) = h^*(v) - \psi(s) + \psi(v)$.

Důkaz: Indukcí podle doby běhu obou algoritmů. Na počátku je $h^*(s)$ i $h(s)$ nulové a všechna ostatní h^* a h jsou nekonečná, takže tvrzení platí. V každém dalším kroku

A* vybere vrchol v s nejmenším $h^*(v) + \psi(v)$, což je tentýž vrchol, který vybere Dijkstra ($\psi(s)$ je stále stejné).

Uvažujme, co se stane během relaxace hrany vw: Dijkstra se pokusí snížit ohodnocení $h(w)$ o $\delta = h(w) - h(v) - \ell_{-\psi}(v, w)$ a provede to, pokud $\delta > 0$. Ukážeme, že A* udělá totéž:

$$\begin{aligned}\delta &= (h^*(w) - \psi(s) + \psi(w)) - (h^*(v) - \psi(s) + \psi(v)) - (\ell(v, w) - \psi(v) + \psi(w)) \\ &= h^*(w) - \psi(s) + \psi(w) - h^*(v) + \psi(s) - \psi(v) - \ell(v, w) + \psi(v) - \psi(w) \\ &= h^*(w) - h^*(v) - \ell(v, w).\end{aligned}$$

Oba algoritmy tedy až na posunutí dané potenciálem počítají totéž. ♡

Důsledek: Algoritmus A* funguje jen tehdy, je-li potenciál $-\psi$ přípustný.

Například pro rovinnou mapu to heuristika daná euklidovskou vzdáleností ϱ, tedy $\psi(v) := \varrho(v, t)$, splňuje: Přípustnost požaduje pro každou hranu uv nerovnost $\ell(u, v) - \psi(v) + \psi(u) \geq 0$, tedy $\ell(u, v) - \varrho(v, t) + \varrho(u, t) \geq 0$. Jelikož $\ell(u, v) \geq \varrho(u, v)$, stačí dokázat, že $\varrho(u, v) - \varrho(v, t) + \varrho(u, t) \geq 0$, což je ovšem trojúhelníková nerovnost pro metriku ϱ.

Literatura

[1] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[2] B. Cherkassky, A. Goldberg, and C. Silverstein. Buckets, heaps, lists, and monotone priority queues. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 83–92. Society for Industrial and Applied Mathematics, 1997.

[3] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[4] A. Elmasry. The violation heap: a relaxed Fibonacci-like heap. *Computing and Combinatorics*, pages 479–488, 2010.

[5] L. R. Ford Jr. Network flow theory. Paper P-923, The RAND Corperation, Santa Moncia, California, August 1956.

[6] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[7] A. Goldberg and C. Silverstein. Implementations of Dijkstra's algorithm based on multi-level buckets. *Network optimization*, pages 292–327, 1997.

[8] B. Haeupler, S. Sen, and R. Tarjan. Rank-pairing heaps. *Algorithms-ESA 2009*, pages 659–670, 2009.

[9] P. Hart, N. Nilsson, and B. Raphael. Correction to a formal basis for the heuristic determination of minimum cost paths. *ACM SIGART Bulletin*, (37):28–29, 1972.

[10] M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 149–158, 2003.

[11] M. Thorup. Equivalence between priority queues and sorting. *Journal of the ACM*, 54(6):28, 2007.