Základní grafové algoritmy

Jakub Černý
KAM, MFF UK

24. listopadu 2010

Verze 0.95

Homepage `http://kam.mff.cuni.cz/~kuba/ka`
Kontakt: `kuba@kam.mff.cuni.cz`

Obsah

Úvod

Text je psán pro úplné začátečníky. To je pro kohokoliv, kdo má rozumné základy z programování a teorie grafů. Snažím se co nejnázorněji vysvětlovat fungování jednotlivých algoritmů, takže by tomu měli rozumět i středoškoláci. Na druhou stranu v ní svoje najdou i středně pokročilí.

Pro začátečníky jsou určeny počáteční kapitoly, které vysvětlí základní pojmy. A dále začátky ostatních kapitol, které obsahují základy daných problémů.

Pokročilejší mohou rovnou přeskočit na kapitoly, které je zajímají. Kromě základů na začátku každé kapitoly, dále najdou i alternativní řešení a souvislosti. Obecně se dá říci, že na začátku kapitoly jsou základy a postupně přituhuje.

Kniha obsahuje jak teoretické výsledky, analýzy (včetně důkazů), tak i finty jak daný algoritmus dobře naprogramovat. Algoritmy jsou popsané pseudokódem, který by měl být čitelný každému, kdo zná základy programování. Přepsání pseudokódu do jiného programovacího jazyka by pro něj už mělo být jednoduché.

Mým cílem je motivovat a hravě vysvětlit grafové algoritmy. Chtěl bych, aby kniha byla srozumitelná, plná příkladů a obrázků. Také bych chtěl, aby se kniha četla dobře a aby Vás snadno provedla obsaženými tématy. Zkrátka, abyste knihu četli pro radost.

Ono není tak těžké o něčem napsat knihu, ale je hodně těžké napsat knihu, která se dobře čte.[1] Jak to shrnuje jeden citát: „Rychlé psaní je těžké čtení, ale lehké čtení je zatraceně těžké psaní."

Knihu jsem se snažil psát tak, jak by se mi to samotnému líbilo, když bych ještě neznal žádné grafové algoritmy.

Je několik výborných zahraničních učebnic, ale ještě jsem neviděl rozumnou českou učebnici na grafové algoritmy. Z těchto důvodů jsem se rozhodl napsat tuto knihu. Když jsem na Karlově Univerzitě cvičil algoritmy, tak mi chyběla sbírka příkladů na procvičení grafových algoritmů. Proto je na konci každé kapitoly spousta příkladů.

Jakub Černý, Ph.D.

Co ještě obsahuje tato úvodní kapitola? Přehled anglickým učebnic, které mohu doporučit a které pro mě byly inspirací. Doporučení o tom, jak používat tuto knihu. Poděkování všem, kteří se zasloužili o to, aby tato kniha byla taková, jaká je.

Nezávisle na čtení knihy si můžete přečíst dodatky, které jsou na konci celé knihy. Dodatky obsahují

- moderní poznatky o tom, jak funguje učení.
- Proslov ke studentům, učitelům.
- Značení použité v této knize.

[1] Viz spousta nekvalitních překladů dobré zahraniční literatury nebo některé české rychlokvašky, kterým chybí dotažení do konce.

Podobné učebnice v angličtině a odkazy

Anglická literatura o grafových algoritmech:

Cormen, Leiserson, Rivest: *Introduction to Algorithms* [9]
Dasgupta, Papadimitriou, and Vazirani: *Algorithms* [10] (dá se stáhnout z webu)
Cook, Cunningham, Pulleyblank, Schriver: *Combinatorial Optimization* [8]
Schrijver: *Combinatorial Optimization* [27] (na webu jsou lecture notes [28])
Erickson: *Algorithms Course Materials* (jsou na webu jako lecture notes [12])

Sbírky příkladů: (často i s ukázkovým řešením).

Korespondenční seminář z programování (`http://ksp.mff.cuni.cz`)
ACM programming contest (například `http://uva.onlinejudge.org/`)

Odkazy na další literaturu najdete v jednotlivých kapitolách (tj. u věcí, se kterými to souvisí).

Doporučení

Jak už jsem psal na začátku. Tato kniha je pro kohokoliv, kdo má rozumné základy z programování a teorie grafů. Aspoň byste měli vědět, co je to zásobník, fronta, rekurze, pointer neboli ukazatel, dynamická alokace paměti a spojový seznam. Pro úvod do programování mohu doporučit knihu P. Töpfer: *Algoritmy a programovací techniky* [30]. Pro hezký úvod do teorie grafů mohu doporučit knihu Matoušek, Nešetřil: *Úvod do diskrétní matematiky* [23], nebo anglicky [24], [11].

Snažím se co nejnázorněji vysvětlovat fungování jednotlivých algoritmů, ale přeci jen má kniha svoje meze. Nedají se do ní vložit animace. Vřele každému doporučuji, aby si na internetu vyhledal animace, applety, které ilustrují průběh jednotlivých algoritmů. Často si do nich můžete zadat vlastní graf a krokovat jednotlivé kroky algoritmu. Pohrajte si s nimi. Vygoolujte si je.

Varování: Programy v celé knize jsou psány tak, aby byli co nejpochopitelnější a co nejnázornější. Zkrátka jsou psány pro snadnou výuku. To znamená, že nemusí být optimalizovány pro co nejrychlejší implementaci (ta stejně záleží na programovacím jazyce a operačním sytému). Když už dobře pochopíte daný algoritmus, tak patří k programátorskému umění dobře algoritmus implementovat. Neberte to tak, jak je psáno, ale přemýšlejte o tom!

Poděkování

Chtěl bych mockrát poděkovat ...
Tomáš Holan, Jan Šarson, Andrea Bachtíková
... za přečtení a korektury v textu.

Kapitola 1

Jak porovnávat algoritmy?

Co je to Algoritmus?
Adam i Božena nezávisle na sobě napsali program,
který hraje šachy. Jak poznat, který program je lepší?

1.1 Algoritmy + Datové struktury = Programy

A co je to ten algoritmus? Neformálně se dá říci, že algoritmus je recept. Dostaneme zadání (co chceme uvařit), přísady (ingredience) a poté provádíme posloupnost kroků podle receptu. Skončíme s požadovaným výsledkem (uvařeným jídlem). Recept, nebo také pracovní postup, by měl mít přesně stanoveno, jak vypadá vstup (suroviny), co má být výstupem (jméno jídla) a dále by měl srozumitelně popisovat, co se má se vstupem dělat, abychom dostali požadovaný výstup.

Odkud pochází název algoritmus? Ještě ve středověku se počítalo s římskými čísly. Umíte sečíst dvě římská čísla, aniž byste je přepsali do desítkové soustavy? Pokud budou čísla dostatečně malá, tak to zvládneme na prstech, ale zkuste sečíst $MCDXLVIII + DCCCXXII$. Moc to nejde a to jsme je ještě nezkoušeli násobit nebo dělit. Desítková soustava vznikla až 6. stol v Indii. Jeden perský astronom a matematik, Al-Khwarizmi, napsal někdy kolem roku 825 spis "O počítání s indickými čísly". Ve spise ukazuje, jak jednoduše sčítat, odčítat, násobit a dělit. Dokonce uměl počítat i druhé odmocniny a relativně dobře vyčíslit π. Jeho spis byl ve 12. století přeložen do latiny jako "Algoritmi de numero Indorum", což znamená něco jako "Algoritmi o číslech od Indů". Algoritmi bylo jméno autora. Lidé názvu špatně porozuměli a od té doby se ujal pojem algoritmus jako metoda výpočtu. Jiní tvrdí, že to bylo na počest autora.

Co jsou to ty datové struktury? Datová struktura je způsob, jak si v počítači organizovat a ukládat data, aby se s nimi dobře a efektivně pracovalo. Příkladem datových struktur je reprezentace čísel, se kterými chceme počítat. Můžeme si je zapsat jako římská čísla nebo v poziční desítkové soustavě. S datovými strukturami jsou přímo spojeny i algoritmy, které na nich pracují.

Samotný recept na jídlo ještě nezaručuje, že bude jídlo výborné. Záleží i na kuchaři. Podobně samotný algoritmus není zárukou skvělého programu, ale záleží na i programátorovi, který algoritmus implementuje pro konkrétní počítač a v konkrétním programovacím jazyce.

Program realizující algoritmus si můžeme představit jako takovou chytrou skříňku, které předhodíme vstup a ona nám po nějaké době "vyplivne" výstup. Čas, za jak dlouho nám skříňka vydá výstup, záleží na algoritmu a vstupu.

Když dostanete nový problém, ne který chcete vymyslet algoritmus, který ho řeší, tak si v první řadě musíte ujasnit, jak vypadají vstupy algoritmu a jak má

vypadat výstup. Teprve pak má smysl přemýšlet o samotném algoritmu.[1] Je dobré dát si nějaký čas na to, abychom si rozmysleli řešení, a pak terpve začít programovat. Řada programátorů se dopouští té chyby, že začnou příliš brzy psát řešení a až v půlce zjistí, že řeší jiný problém. Na druhou stranu nemá smysl příliš dlouho vymýšlet a plánovat. Klidně můžeme nejprve naprogramovat prototyp, získat nové poznatky a zkušenosti, a pak teprve napsat dokonalou verzi.

Při vymýšlení složitějších algoritmů často potřebujeme efektivně vyřešit řadu "základních problémů", jako je třídění, vyhledávání, fronta, zásobník a další. Protože se tyto problémy vyskytují opravdu často, tak se i hodně studují. Známe pro ně celou řada efektivních řešení v podobě algoritmů a datových struktur. Jejich dobrá znalost patří k základnímu programátorskému "know how". Více se o nich můžete dozvědět v literatuře. Pro jednoduchý úvod doporučujeme knihu P. Töpfer: Algoritmy a programovací techniky [30]. Pro znalce, hledající opravdové skvosty, doporučujeme D. Knuth: The Art of Computer Programming [18, 19, 20]. Poměrně dost informací najdete i na internetu. Internetová verze má o proti knize tu výhodu, že může obsahovat plně audiovizuální prezentaci, například animace průběhu algoritmů.

1.2 Jak poznat, který algoritmus je lepší?

Když Vy i Váš kamarád dostanete stejné zadání problému, tak nejspíš každý navrhnete jiný algoritmus. Někdy i Vás samotné napadne více řešení. Jak poznat, který algoritmus je lepší? Mohli byste navrhnout, že ten, který proběhne rychleji. A nebo to bude ten, který bude potřebovat méně paměti? A nebo oba algoritmy proběhnou tak rychle a spotřebují tak málo paměti, že vybereme ten, který má kratší a jednoduší zdrojový kód? Všechny odpovědi mohou být správné. Záleží, co chceme optimalizovat:

- **Rychlost výpočtu,** tj. za jak dlouho program proběhne. Jak dlouho budeme muset čekat, než se dozvíme výsledek?
- **Paměťovou náročnost.** Kolik paměti program zabere? Bude stačit paměť, kterou v počítači máme? Pokud program potřebuje více paměti, než je k dispozici, tak se chybějící paměť nahradí pamětí na pevném disku. Ta je výrazně pomalejší a proto se tím zpomalí výpočet.
- **Rychlost, za jak dlouho program napíšeme.** Některé věci potřebujeme spočítat jen jednou. Potom nezáleží tolik na rychlosti výpočtu, jako na celkovém čase, než program napíšeme a než program proběhne. Je jedno, jestli program poběží 1s nebo 5min, protože jeho naprogramování nám zabere podstatně více času.

  Do času, za jak dlouho program napíšeme, spadá i odladění chyb. A to někdy trvá hodně dlouho. V kratších a jednodušších algoritmech je menší šance, že uděláme chybu.

Dobrý programátor si nejprve promyslí, na které z těchto kritérií se chce zaměřit a podle toho vybere vhodný algoritmus.

První dvě kritéria nás přivádí ke dvěma důležitým pojmům – časové a prostorové složitosti. Zhruba řečeno, *časová složitost* říká, jak dlouho algoritmus poběží v závislosti na velikosti vstupních dat. *Prostorová (paměťová) složitost* říká, kolik paměti je potřeba k vykonání algoritmu v závislosti na velikosti vstupních dat.

[1] Odpovídá to i řízení projektů a komunikaci mezi lidmi vůbec. Nejprve se musíme shodnout na pojmech a jejich významu, v těchto pojmech si ujasníme, co po nás kdo chce a co očekává. Teprve když už tohle všechno víme, tak můžeme začít projekt řešit.

Tyto dvě věci spolu úzce souvisí. Zrychlení výpočtu můžeme dosáhnout tím, že si něco předpočítáme. Příkladem může být program, který na vstupu dostane $n \in \{1, \ldots, 50\}$, a má odpovědět jestli lze šachovnici velikosti $n \times n$ proskákat šachovým koněm tak, abychom každé políčko navštívili právě jednou. Všechny odpovědi si můžeme předpočítat a uložit do tabulky. Nejrychlejší program se jen podívá do tabulky a vrátí odpověď. Použitelnější příklad může být předpočítání si prvočísel.

Předpočítané věci si ale musíme někam uložit a to nám může zvýšit prostorovou složitost. Proto se často stává, že rychlejší algoritmy mají větší prostorovou složitost a naopak.[2]

1.2.1 Praktické porovnávání algoritmů

Porovnání algoritmů lze provádět teoreticky i prakticky. Teoreticky můžeme nalézt odhad na počet kroků algoritmu (tj. jeho rychlost) a na spotřebu paměti. Často už v teoretických odhadech dostaneme takové rozdíly, že nemá smysl algoritmy dále porovnávat. Ale pokud je algoritmus složitý, tak může být nalezení správných odhadů těžké. Nejlepší odhady, které umíme ukázat mohou být mnohokrát horší než reálné hodnoty. Také záleží na tom, na kterých datech/vstupech budeme algoritmy používat. Z lepší znalosti vstupních dat můžeme ukázat mnohem lepší odhady než pro obecná data. Proto je lepší brát odhady časových složitostí jen jako první a hrubé porovnání. Pro lepší porovnání algoritmů nám nezbude nic jiného, než oba algoritmy naprogramovat. Potom oba programy spustíme na používaných datech, změříme časy výpočtu a velikosti zabrané paměti, a naměřené údaje porovnáme.

Dokonce se může stát, že praktickým porovnáním algoritmů dostane opačný závěr než teoretickým porovnáním. Tedy že se algoritmus s teoreticky vysokou časovou složitostí může prakticky chovat lépe než algoritmus s teoreticky nízkou časovou složitostí.[3]

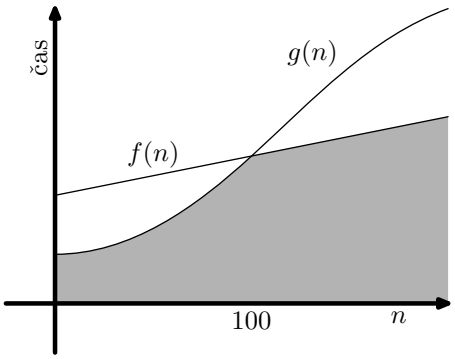
Praktickému porovnávání algoritmů se v tomto textu nebudeme příliš věnovat, ikdyž je to v praxi velmi důležité. Jenom poznamenejme, že je potřeba měřit rychlost na stejném počítači a také na stejných vstupních datech. Také dost záleží na tom, ve kterém programovacím jazyce a jak dobře je který algoritmus implementován. I elegantní algoritmus lze implementovat úplně neelegantně. Čas běhu programu je také značně závislý na hardwaru, na kterém program běží (například drobné zvětšení vstupních dat může výrazně zpomalit běh celého programu, protože se nám najednou data nevejdou do paměti a budou se muset ukládat na disk). Podobná je i závislost na operačním systému.

Pokud už máme dobře proměřené chování algoritmů (jejich časové složitosti) na různě velkých datech, tak je můžeme porovnat. Následující obrázek zachycuje grafy časové složitosti dvou programů v závisloti na velikosti vstupu n. Může se stát, že jeden algoritmus bude lepší pro menší data (například pro $n \leq 100$) a druhý pro

[2] V dnešní době se klade větší důraz na časovou složitost, protože paměti je relativně dost a není tak drahé ji dokoupit.

[3] Příkladem může být Simplexový algoritmus pro úlohy lineárního programování. V nejhorším případě má exponenciální časovou složitost, ale na praktických datech se chová daleko lépe než všechny známé polynomiální algoritmy pro lineární programování. Ty totiž mají časovou složitost něco jako $\mathcal{O}(n^{50})$ a to ještě nemluvíme o konstantách schovaných ve velkém O.

větší data (pro $n > 100$). Který algoritmus vybrat?



Oba. Nejlepšího výsledku dosáhneme kombinací obou algoritmů. Pro malá data použijeme první algoritmus a pro velká data ten druhý.

Poznámka: Výborným tréninkem praktického programování je programovací soutěž ACM. Na adrese `http://uva.onlinejudge.org/` si ji můžete vyzkoušet online mimo soutěž. (řada dalších serverů s podobnou službou vygooglíte při hledání"online judge"). Server obsahuje archiv úloh, které se už někdy objevily programovacích soutěžích.

Vyberete si úlohu a dostanete zadání, které obsahuje popis problému, formát vstupů a formát výstupů. Až napíšete program, který úlohu řeší, tak ho odešlete na server. Soudce na serveru posoudí, jestli vaše řešení vrací správné výsledky a také změří, jestli je dostatečně rychlé. Do pár vteřin vám oznámí výsledek. Pokud váš program běží déle, než je stanovený limit, tak se řešení odmítne a vy musíte přemýšlet, jak to naprogramovat lépe. Na tom se přesně naučíte prakticky srovnávat algoritmy na konkrétních vstupech (velikost vstupu stejně jako čísel na vstupu je u každé úlohy omezena konstantou ze zadání).

K tomu, abyste byli v soutěži dobří, potřebujete rychle analyzovat problém, vymyslet dostatečně dobré řešení a co nejrychleji ho naprogramovat a odladit.

1.2.2 Teoretické porovnávání algoritmů

K teoretickému porovnání dvou algoritmů (tj. aniž bychom oba algoritmy naprogramovali) stačí odhadnout počet kroků, které každý algoritmus udělá v závisloti na velikosti vstupu.

Jak to udělat si vysvětlíme v následující kapitole 2 o časové složitosti.

Kapitola 2

Časová složitost

... aneb jak dlouho program poběží, než dostanu výsledek.

Velikost vstupu: Jak měřit velikost vstupních dat? Velikost vstupních dat je počet bitů, které jsou potřeba k zápisu vstupu do počítače. Často ale velikost vstupních dat měříme hrubě a uvádíme jen počet čísel na vstupu, počet vrcholů grafu, počet hran grafu apod. Zápis každého čísla, vrcholu či hrany, odpovídá jedné proměnné v počítači a obsahuje jenom konstantní počet bitů. Proto je skutečná velikosti vstupu (počet bitů) jen konstantě krát větší. Později si ukážeme, že nám při výpočtu časové složitosti na multiplikativních konstantách nezáleží.

Na binární zápis čísla n je potřeba $\log n$ bitů. Proto musíme rozlišovat, jaká čísla dostaneme na vstupu. Čísla typu integer mají omezenou velikost 32 bitů (konstantní počet bitů). Pokud na vstupu dostaneme dlouhé číslo n, například s miliónem cifer, tak ho nemůžeme uložit do jedné proměnné typu integer, ale budeme ho muset reprezentovat v poli. Proto bude jeho velikost na vstupu odpovídat počtu políček pole, nebo přesněji počtu bitů, což je $\log n$.

Časová složitost: Časová složitost algoritmu spuštěného na vstup $\mathcal{D}$ je počet kroků, které algoritmus provede. Časová složitost algoritmu je funkce $T : \mathbb{N} \to \mathbb{N}$, kde $T(n)$ je (maximální) počet kroků, které provede algoritmus běžící na datech o velikosti n (pro všechny vstupy $\mathcal{D}$ velikosti n).

Co je to krok algoritmu? Krok algoritmu je jedna operace/instrukce daného stroje/počítače. Je to například přiřazení, aritmetická operace $+$, $-$, $*$, $/$, vyhodnocení podmínky apod. Zjednodušeně budeme za krok algoritmu považovat libovolnou operaci proveditelnou v konstantním čase.

Označme velikost vstupu jako n a nechť c je nějaká konstanta. Časové složitosti $c \cdot n$ budeme říkat lineární, časové složitosti $c \cdot n^2$ kvadratická, $c \cdot n^3$ kubická a $c \cdot a^n$ pro $a > 1$ exponenciální.

Jak můžeme jedno přiřazení považovat za krok algoritmu stejně jako padesát přiřazení? Proč můžeme zjednodušeně říci, že jedno číslo na vstupu má velikost jedna a ne 32, i když zabere 32 bitů? Copak na těchto konstantách v teoretických odhadech nezáleží? Ano je to tak. Na těchto konstantách moc nezáleží a ukážeme si proč (prakticky na nich záleží, ale to patří do praktického porovnávání algoritmů).[1]

Podívejme se do následující tabulky, která ukazuje, jak dlouho budou trvat výpočty algoritmů s různou časovou složitostí. Čísla v tabulce jsou přibližné hodnoty funkcí na vstupech o velikostech 10, 100, 1000 a 1000000.

[1] Ve skutečnosti trvá každá instrukce procesoru jiný čas (jiný počet taktů), a navíc je to na každém typu procesoru jinak. Některé instrukce trvají 1 takt, 2 takty, ale jsou i instrukce trvající 150 taktů. Je to takový guláš, že nám nic jiného než hrubý odhad času nezbude.

	$n = 10$	$n = 100$	$n = 1000$	n=1000000
$\log n$	3.3	6.7	10	20
$\sqrt{n}$	3.2	10	31.6	1000
n	10	100	1000	10^6
$n \log n$	33	664	9966	$20 \cdot 10^6$
n^2	100	10^4	10^6	10^{12}
n^3	1000	10^6	10^9	10^{18}
2^n	1024	$13 \cdot 10^{30}$	$11 \cdot 10^{302}$	$\approx \infty$
$n!$	$36 \cdot 10^5$	$93 \cdot 10^{157}$	$40 \cdot 10^{2567}$	$\approx \infty$

Běžný počítač zvládne spočítat 10^9 operací za vteřinu. Následující tabulka udává, jak dlouho budou trvat výpočty algoritmů na běžném počítači.

	$n = 10$	$n = 100$	$n = 1000$	n=1000000
$\log n$	$3.3ns$	$6.7ns$	$10ns$	$20ns$
$\sqrt{n}$	$3.2ns$	$10ns$	$31.6ns$	$1\mu s$
n	$10ns$	$100ns$	$1\mu s$	$1ms$
$n \log n$	$33ns$	$664ns$	$9.9\mu s$	$20ms$
n^2	$100ns$	$10\mu s$	$1ms$	$16,5min$
n^3	$1\mu s$	$1ms$	$1s$	$31let$
2^n	$1\mu s$	$3 \cdot 10^{14} let$	$3 \cdot 10^{286} let$	$\approx \infty$
$n!$	$3ms$	$3 \cdot 10^{142} let$	$\approx \infty$	$\approx \infty$

Co můžeme z tabulky vypozorovat? Z tabulky je vidět, že skoro všechny výpočty až na ty s časovou složitostí 2^n a $n!$, budou trvat rozumný čas. Proto budeme považovat algoritmy s polynomiální časovou složitostí za rozumné a těm s exponenciální časovou složitostí se budeme snažit vyhnout. Dále je vidět, že pro zpracování velkých dat jsou algoritmy s časovou složitostí menší než $c \cdot n \log n$ výrazně lepší než ostatní polynomiální algoritmy s vyšším stupněm.

To, jestli má program rozumnou časovou složitost, nerozhoduje jen o tom, jestli odpověď dostaneme hned a nebo jestli budeme muset chvilku čekat. Je otázkou, jestli se výsledku vůbec dožijeme. Vždyť i naše planeta Země existuje teprve $4,5$ miliardy let. To je $4,5 \cdot 10^9$ let což je zhruba jen $14 \cdot 10^{16}$ vteřin.[2]

Z tabulky je také vidět, že například funkce n^2, $5n^2$ a $30n^2$ porostou skoro stejně rychle. Na dostatečně velkých vstupech je snadno odlišíme od lineárních funkcí, ale i od kubických až exponenciálních funkcí. Proto není důležitá ta konstanta před funkcí, ale řád n^2, se kterým funkce roste. Z toho důvodu nebudeme při posuzování algoritmů brát ohled na konstanty. To nás přivádí k pojmu asymptotická časová složitost. (Prakticky záleží i na těchto konstantách. Přeci jen je rozdíl, jestli bude algoritmus počítat rok a nebo 10 let, případně 2 hodiny a nebo 1 den.)

Poznámka: Kdy se vyplatí koupit si nový počítač? O kolik větší data/vstupy budeme moci zpracovávat? Předpokládejme, že nový počítač bude dvakrát rychlejší než ten, co máme, a že používáme aplikaci, která musí nejpozději do minuty vydat odpověď. Pokud odpověď počítáme podle algoritmu s lineární časovou složitostí, tak stihneme spočítat dvakrát větší vstup. Naproti tomu pokud má algoritmus exponenciální časovou složitost, tak budeme rádi, když spočítáme výsledek pro o jedna větší vstup. A možná ani to ne.

Poznámka (Moorův zákon): Moorův zákon popisuje zajímavý trend v historii počítačového hardwaru. Říká, že se rychlost nových počítačů přibližně za každé

[2]Vzpomeňte si na Stopařova průvodce po galaxii. V této knize si myši nechaly postavit super počítač – planetu Zemi, aby jim odpověděl na základní otázku života, vesmíru a tak vůbec. Jejich výpočet byl o proti algoritmům s exponenciální časovou složitostí spuštěných na vstupu velikosti tisíc docela efektivní.

dva roky zdvojnásobí.[3] Mohli byste proto namítat, že můžeme výpočty zrychlovat tím, že si počkáme na výkonnější počítače. Jak jsme si ale ukázali v předchozí poznámce, tak nám některých případech dvojnásobné zrychlení výpočtu moc nepomůže. Proto je lepší se zaměřit na vývoj efektivnějších algoritmů.

Také se proslýchá, že se v brzké době Moorův zákon zastaví. Tranzistory na procesoru už jsou tak malé, že už se blíží k velikosti atomů.

Poznámka: Některé časové složitosti vůbec nemusí být rostoucí funkce. Klidně mohou oscilovat. Podívejme se například na algoritmus, který zjišťuje, zda je číslo n prvočíslo. Postupně bude procházet čísla 2 až $\sqrt{n}$ a testovat, zda dané číslo dělí n. Pokud uspěje, tak máme dělitele, můžeme skončit a odpovědět, že číslo n není prvočíslo. Jinak projdeme všechny možné dělitele a nakonec odpovíme, že n je prvočíslo.[4] Pro každé sudé n algoritmus skončí hned v prvním kroku. Na druhou stranu pro každé n prvočíslo algoritmus projde všech $\sqrt{n}$ čísel. Proto časová složitost osciluje mezi funkcemi 1 a $\sqrt{n}$.

2.1 Asymptotická časová složitost

Při určování asymptotické časové složitosti nás zajímá chování algoritmů na hodně velkých datech. Vezměte si papír, hodně velký papír, a nakreslete na něj grafy všech funkcí představujících časovou složitost, které vás napadnou. Při pohledu zblízka uvidíme velký rozdíl mezi funkcemi $\log n$, n, 2^n, ale i mezi n, $5n$ a $50n$. Asymptotická časová složitost se na tento papír dívá z velké dálky, třeba až z Marsu (musíme mít hodně velký papír). Při jejím pohledu všechny funkce, které se liší jen multiplikativní konstantou, „splynou“ v jednu „rozmazanou funkci“. Z takové dálky zůstane vidět jen propastný rozdíl mezi funkcemi $\log n$, n, 2^n (viz tabulka na straně 5 zachycující, jak rychle rostou některé funkce).

Chceme zavést značení, které bude říkat, že funkce lišící se pouze multiplikativní konstantou, patří do stejné třídy.

Definice: Nechť f a g jsou dvě funkce z přirozených čísel do přirozených čísel. Řekneme, že

$f(n) = \mathcal{O}(g(n))$ (čte se „f je velké O od funkce g“) právě tehdy, když existuje konstanta $c > 0$ a n_0 takové, že pro každé $n \geq n_0$ platí $f(n) \leq c \cdot g(n)$.

$f(n) = \Omega(g(n))$ (čte se „f je omega od funkce g“) právě tehdy, když existuje konstanta $c > 0$ a n_0 takové, že pro každé $n \geq n_0$ platí $f(n) \geq c \cdot g(n)$.

$f(n) = \Theta(g(n))$ (čte se „f je theta od funkce g“) právě tehdy, když zároveň $f(n) = \mathcal{O}(g(n))$ i $f(n) = \Omega(g(n))$.

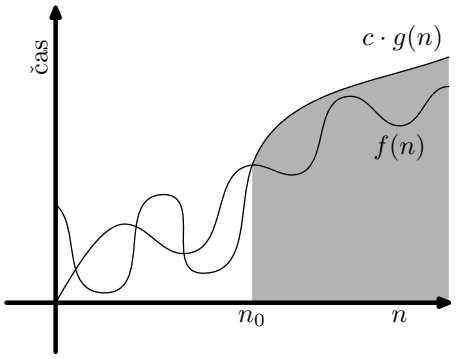
Slovy se dá asymptotická notace $f(n) = \mathcal{O}(g(n))$ popsat jako f neroste řádově rychleji než funkce g. Zápis $f(n) = \Omega(g(n))$ znamená, že funkce f roste řá-

[3] Podobný trend lze vypozorovat i pro velikost pevných disků, počtu pixelů ve foťácích apod.

[4] Samozřejmě si můžete vymyslet daleko lepší algoritmus. Tento je jen pro ukázku oscilující časové složitosti.

dově aspoň tak rychle jako funkce g a zápis $f(n) = \Theta(g(n))$, že obě funkce rostou řádově stejně rychle.

Můžeme si to vysvětlit i na obrázku vpravo. Notace $f(n) = \mathcal{O}(g(n))$ říká, že existuje taková konstanta c, že od určitého n_0 už leží graf funkce $f(n)$ pod grafem funkce $c \cdot g(n)$ (v šedě vyznačené oblasti).



Uveďme ještě pár příkladů. Platí $2^{56} = \mathcal{O}(1)$, $30n = \mathcal{O}(n)$, $n = \mathcal{O}(n^2)$, $n^{30} = \mathcal{O}(2^n)$, ale také $5n^2 + 30n = \mathcal{O}(n^2)$, protože $5n^2 + 30n \leq 35n^2 = \mathcal{O}(n^2)$.

Pozor na značení! Přesněji bychom měli psát $f \in \mathcal{O}(g)$ a říkat „f je třídy $\mathcal{O}(g)$“ a nebo „f patří do třídy $\mathcal{O}(g)$“. Značení s „=“ může být zavádějící, protože $\mathcal{O}(x) = \mathcal{O}(x^2)$, ale $\mathcal{O}(x^2) \neq \mathcal{O}(x)$. Musíme ho chápat spíše jako „$= \mathcal{O}$“ ve významu „$\preceq$“. Proto se také $f = \mathcal{O}(g)$ někdy čte jako „f je asymtoticky menší nebo rovno g“.

Lemma 1 *Nechť f_1, f_2, g_1, g_2 jsou funkce takové, že $f_1 \in \mathcal{O}(g_1)$, $f_2 \in \mathcal{O}(g_2)$, k je konstanta a h je rostoucí funkce. Potom*

- $f_1 \cdot f_2 \in \mathcal{O}(g_1 \cdot g_2)$
- $f_1 + f_2 \in \mathcal{O}(g_1 + g_2) = \mathcal{O}(\max(g_1, g_2))$
- $k \cdot f_1 \in \mathcal{O}(g_1)$
- $f_1(h(n)) \in \mathcal{O}(g_1(h(n)))$*, ale také* $h(f_1(n)) \in \mathcal{O}(h(g_1(n)))$.

Mezi často používané odhady patří následující. Pro každé $a \leq b$ platí $n^a = \mathcal{O}(n^b)$. Pro každé k platí $n^k = \mathcal{O}(2^n)$. To je ekvivalentní s $\log^k n = \mathcal{O}(n)$. Zkuste si tyto odhady i předchozí lemma dokázat (je to jen cvičení s funkcemi, limitami a použití definice velkého $\mathcal{O}$).

Definice vhodná pro výpočty: Asymptotickou notaci můžeme definovat i přes limitu.

$$f(n) = \mathcal{O}(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} \in \langle 0, +\infty)$$

$$f(n) = \Omega(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} \in (0, +\infty\rangle$$

Tato metoda je lepší pro počítání, ale musíme být trochu zběhlí v počítání limit. Zkuste si dokázat, že definice přes limitu implikuje námi uvedenou definici. Je to jednoduché cvičení na definici limity.

Příklad: Potřebujeme zjistit, jestli platí $2n^2+4n-5 = \mathcal{O}(n^2)$. Podíváme se tedy na $\lim_{n\to\infty} \frac{2n^2+4n-5}{n^2} = 2$. Když si zvolíme $\varepsilon > 0$, například $\varepsilon = 3$, tak z definice limity dostaneme, že existuje n_0 takové, že pro každé $n \geq n_0$ platí $\frac{2n^2+4n-5}{n^2} < 2 + \varepsilon = 5$. Po přenásobení nerovnosti faktorem n^2 je to přesně definice vztahu $2n^2 + 4n - 5 = \mathcal{O}(n^2)$.

2.2 Časová složitost v nejhorším případě

Spočítat časovou složitost přesně je docela těžké. Protože jsme si ukázali, že na multiplikativních konstantách tolik nezáleží, budeme časovou složitost jenom odhadovat ze shora.

Některé úlohy přijímají více různých vstupů stejné velikosti (například když chceme setřídit n čísel). Doba výpočtu může být závislá na konkrétním vstupu. Potřebujeme odhadnout čas, za jak dlouho program určitě doběhne ve všech případech. K tomu nám stačí odhadnout dobu výpočtu v nejhorším případě. Té budeme říkat časová složitost v nejhorším případě.

Například pokud program počítá navádění letadel a snaží se zabránit srážkám, tak musíme vždycky dostat odpověď do pár vteřin. Nemůžeme říkat, že to normálně funguje rychle a bez problémů. Že došlo k nehodě jen proto, že tam ta letadla letěla blbě, nastala vyjímečná situace a my jsme to nestihli spočítat.

Uvedeme si několik příkladů, na kterých si ukážeme, jak počítat časovou složitost v nejhorším případě.

2.2.1 Hledání minima v poli

V poli $\mathtt{A}[\,\cdot\,]$ dostaneme n čísel. Máme vrátit hodnotu nejmenšího z nich.

```
1: min := A[1]
2: for i = 2 to n do
3:         if A[i] < min then
4:             min := A[i]
5: return min
```

Řádky 3 a 4, tj. podmínka a případné zapamatování si nového minima, zaberou nejvýše konstantní čas. Označíme je za krok algoritmu. Spolu s testem uvnitř for-cyklu proběhnou $(n-1)$-krát. Ostatní řádky proběhnou jen jednou. Proto je celková časová složitost algoritmu $\mathcal{O}(n)$.

Umíte ukázat dolní odhad na časovou složitost hledání minima? Tím myslíme dokázali byste ukázat, kolik nejméně porovnání dvou čísel je potřeba k nalezení minima? Podívejme se na hledání minima jako na turnaj. Při utkání dvou prvků vyhrává ten menší. Chceme najít vítěze. Prvek, který je absolutním vítězem, musel porazit každý další prvek přímo a nebo nepřímo (porazit někoho, kdo ten prvek porazil – ať už přímo nebo nepřímo). Každý z $n-1$ prvků, které nevyhrály, musel být aspoň jednou poražen. Jinak o sobě může prohlašovat, že je také vítězem. Proto je potřeba alespoň $n-1$ porovnání (zápasů).

2.2.2 Sečtení prvků v matici

Dostaneme matici přirozených čísel o rozměrech $n \times n$ a chceme všechna čísla sečíst. Pomocí dvou for-cyklů projdeme celou matici a všechna čísla sečteme. Proto je časová složitost algoritmu $\mathcal{O}(n^2)$. Rychleji to nejde, protože musíme projít všech n^2 čísel.

Přesto můžeme o algoritmu říci, že je lineární ve velikosti vstupu. Musíme si uvědomit, že vstupem algoritmu je matice, která obsahuje n^2 čísel.

2.2.3 Vypisování n čísel

Dostaneme číslo n a úkolem je vypsat všechna čísla 1 až n. Jakou to bude mít časovou složitost? Pokud ji chceme vyjádřit v závislosti na čísle n, tak jednoduše $\mathcal{O}(n)$. Ovšem co když ji chceme vyjádřit vzhledem k velikosti vstupu? Velikost vstupu je $m := \log n$ bitů. Na výstup vypíšeme n čísel, každé o velikosti maximálně $\log n$. Časová složitost odpovídá počtu vypsaných bitů a to je $\mathcal{O}(n \log n) = \mathcal{O}(m2^m)$.

Vždy je potřeba si ujasnit, vzhledem k čemu budeme časovou složitost vyjadřovat.

2.2.4 Binární vyhledávání v setříděném poli

Máme pole `A[ · ]` obsahující n čísel setříděných od nejmenšího po největší. Chceme zjistit, jestli pole obsahuje číslo x.

```
1:  dolni := 1
2:  horni := n
3:  while horni ≥ dolni do
4:      stred := ⌊(dolni + horni)/2⌋
5:      if x = A[stred] then
6:          return TRUE
7:      else if x < A[stred] then
8:          horni := stred − 1
9:      else
10:         dolni := stred + 1
11: return FALSE
```

Invariant je vlastnost, která se v průběhu algoritmu nemění. Vhodných invariantů často využíváme v důkazech správnosti algoritmu, i při výpočtu časové složitosti.

Invariantem binárního vyhledávání je, že číslo x může být v poli `A[ · ]` pouze v úseku mezi indexy *dolni* a *horni*. Na začátku tento úsek obsahuje celé pole a v každé iteraci ho zmenšíme na polovinu. Pokud už je úsek tak malý, že neobsahuje žádný prvek, tak skončíme.

Díky půlení úseku proběhne while-cyklus nejvýše $\log n$ krát. To můžeme nahlédnout analýzou pozpátku. Průběh algoritmu sledujeme jako film, který si pustíme pozpátku. Nejprve má úsek jeden prvek, pak dvakrát tolik, to je dva prvky. A tak dále, až po $h = \lceil \log n \rceil$ krocích bude mít $2^h \geq n$ prvků.

Proto je časová složitost vyhledávání $\mathcal{O}(\log n)$.

2.2.5 Bublinkové třídění

V poli `A[ · ]` dostaneme n čísel. Čísla chceme setřídit pomocí porovnávání dvojic a prohazování prvků pole. Použijeme bublinkový algoritmus.

```
1: for j = 1 to n do
2: for i = 1 to n − j do
3:         if A[i] > A[i + 1] then
4:             prohoď(i, i + 1)
```

Prohoď(x,y) prohodí prvky v poli `A` na pozicích x a y. Algoritmus obsahuje dva for-cykly a každý z nich proběhne nejvýše n-krát. Řádky 3, 4 budou trvat konstantní čas a proto odhadneme časovou složitost jako $\mathcal{O}(n^2)$.

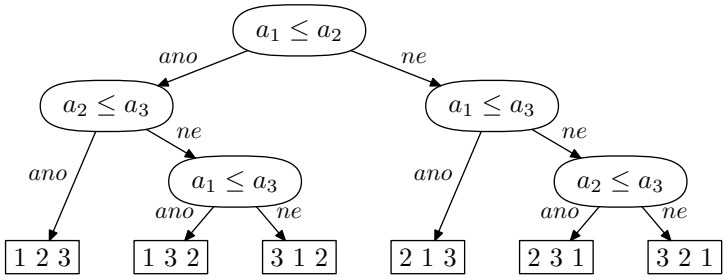
Mohli byste namítat, že jsme časovou složitost počítali příliš hrubě. Druhý for-cyklus přeci neproběhne vždycky n-krát, ale poprvé jen $(n-1)$-krát, pak $(n-2)$-krát,... až dvakrát a naposledy jen jednou. Přesný počet provedení řádek 3 a 4 je součtem aritmetické řady $(n-1) + (n-2) + \cdots + 2 + 1 = (n-1)\frac{(n-1)+1}{2} = {}^1\!/_2(n^2-n) = \mathcal{O}(n^2)$. I po přesnějším výpočtu nám vyšla stejná asymptotická časová složitost. Z toho vidíme, že si v určitých případech můžeme dovolit počítat hrubě.[5]

[5] Jak poznám, kdy mohu počítat hrubě? To se naučíte časem. Chce to jen trochu cviku.

2.2.6 Dolní odhad pro třídění

A co dolní odhad? Kolik nejméně porovnání (a případných prohození) je potřeba k setřídění n čísel a_1, a_2,..., a_n? Průběh každého deterministického algoritmu[6] lze zachytit rozhodovacím stromem.

Na následujícím obrázku je příklad rozhodovacího stromu pro setřídění a_1, a_2, a_3. Každý vnitřní vrchol odpovídá porovnávání dvou čísel a_i a a_j (a jejich případnému prohození). Toto porovnání můžeme vyjádřit otázkou „Je $a_i \leq a_j$?“ Listy stromu jsou označeny permutací, podle které musíme přerovnat vstup, abychom dostali setříděnou posloupnost. Tedy například listu označenému permutací „1 3 2“ odpovídá pořadí prvků $a_1 \leq a_3 \leq a_2$.



Jak funguje algoritmus odpovídající rozhodovacímu stromu? Algoritmus začne v kořenu stromu, kde se zeptá na první otázku. Odpověď mu určí větev stromu, kterou bude pokračovat. Po hraně dorazí k další otázce. Zeptá se na tuto otázku a podle odpovědi bude pokračovat dále. Postupně se ptá na otázky, které potká, a odpovědi mu určují cestu, kudy bude pokračovat. Nakonec dorazí do listu, kde si už musí být jist, jaké je uspořádání prvků ze vstupu podle velikosti.

Kontrolní otázka: Jakou permutaci setříděné posloupnosti musíme dát na vstup, aby algoritmus došel do listu s permutací π? Permutaci inverzní k π, to je π^{-1}.

Listy stromu musí obsahovat všechny možné permutace. Jinak by existovaly dvě permutace setříděné posloupnosti, které když dáme na vstup, tak povedou do stejného listu.[7] Algoritmus by nebyl schopen je rozlišit. Proto má rozhodovací strom alespoň $n!$ listů.

Rozhodovací strom je binární strom s alespoň $n!$ listy. Časová složitost v nejhorším případě je délka nejdelší cesty od kořene do listu. Ta je nejkratší, pokud je strom vyvážený a jeho listy obsahují každou permutaci jen jednou. Protože strom má $n!$ listů, tak je jeho výška alespoň

$$\log(n!) = \log n + \log(n-1) + \cdots + \log 1 \geq \frac{n}{2} \cdot \log(n/2) \geq \frac{1}{2} n \log n - \frac{n}{2} \log 2$$

(v první nerovnosti jsme si nechali jen první polovinu členů součtu a odhadli je zdola velikostí nejmenšího ze zbylých členů). Tím jsme ukázali, že každý algoritmus založený na porovnávání dvojic musí udělat $\Omega(n \log n)$ porovnání. Poznamenejme, že odhad je asymptoticky nejlepší možný, protože třídící algoritmy s časovou složitostí $\mathcal{O}(n \log n)$ v nejhorším případě existují (například heapsort nebo mergesort).

Pozor, jsou i algoritmy, které nejsou založené na porovnávání dvou čísel, ale využívají znalostí o tom, jak čísla vypadají. Například pokud na vstupu dostaneme n různých čísel z rozsahu 1 až N, tak si můžeme vytvořit pole $\mathtt{x}[\,\cdot\,]$ o velikosti N znázorňující charakteristický vektor množiny ze vstupu. Hodnota $\mathtt{x}[i]$ je 1, pokud

[6] To je takový algoritmus, který postupuje podle předem známého plánu. Nerozhoduje se ani náhodně, ani si nenechá věštit postup od vědmy či z orákula.

[7] Plyne to z Dirichletova holubníkového principu. Pro každý vstup se algoritmus dostane do nějakého listu. Pokud je vstupů více než listů, tak musí existovat list, do kterého vedou alespoň dva vstupy.

je číslo i součástí vstupu a 0 jinak. Pole $\mathtt{x}[\,\cdot\,]$ vyplníme jedním průchodem vstupu. Jedním průchodem pole $\mathtt{x}[\,\cdot\,]$ jsme schopni vypsat setříděnou posloupnost čísel. Časová složitost tohoto algoritmu je tedy $\mathcal{O}(n+N)$. Drobným vylepšením dostaneme například BucketSort, RadixSort, kterým se česky říká přihrádkové třídění. Nevýhodou těchto algoritmů jsou vyšší paměťové nároky a časová složitost závislá na velikosti čísel N. Záleží, co víme o datech, které chceme třídit.

2.3 Časová složitost v průměrném případě

Může se stát, že program bude na většině dat fungovat celkem rychle, akorát na pár „blbých“ výjimkách poběží hrozně dlouho. Abychom lépe popsali časovou složitost takového algoritmu, tak kromě časové složitosti v nejhorším případě uvedeme i průměrnou časovou složitost. Tu spočítáme jako průměr časových složitostí přes všechny možné vstupy.

Často není rychlost odpovědi otázkou života a smrti a proto nám nebude vadit, když si ve výjimečných případech počkáme déle. Důležité je, že v průměru budeme dostávat odpovědi relativně rychle.

Příkladem programu, který má průměrnou časovou složitost lepší než časovou složitost v nejhorším případě, je quicksort. Při nevhodné volbě pivota může mít až kvadratickou časovou složitost, ale v průměrném případě je jeho složitost $\mathcal{O}(n \log n)$. Přesným výpočtem se dá ukázat, že konstanta před $n \log n$ je o proti jiným třídícím algoritmům malá. To je důvod, proč se v praxi quicksort tolik používá.

2.3.1 QuickSort

Quicksort využívá techniku Rozděl a Panuj (Divide et Empera) o které si povíme později v sekci 3.

Algoritmus dostane n čísel v poli $\mathtt{A}[\,\cdot\,]$ a má je setřídit od nejmenšího po největší.

Quicksort funguje rekurzivně. Dostene úsek pole $\mathtt{A}[l..p]$, který nejprve rozdělí na dva podúseky $\mathtt{A}[l..q]$ a $\mathtt{A}[q+1..p]$ podle hodnoty, které se říká pivot. První podúsek obsahuje všechny prvky menší nebo rovné pivotu a druhý podúsek všechny prvky větší než pivot. Nakonec quicksort nechá oba podúseky setřídit rekurzivně.

```
1: Quicksort(l, p)
2:     if l < p then
3:         pivot := A[⌊(l + p)/2⌋]
4:         q :=Partition(l, r, pivot)
5:         Quicksort(l, q)
6:         Quicksort(q + 1, p)
```

Funkce q :=Partition(l, p, $pivot$) rozdělí úsek pole $\mathtt{A}[l..p]$ na dva úseky $\mathtt{A}[l..q]$ a $\mathtt{A}[q+1..p]$, kde první úsek obsahuje pouze prvky menší nebo rovny pivotu a druhý úsek pouze prvky větší než pivot. Funkce vrátí index q, který odpovídá hranici mezi podúseky.

```
1: Partition(l, p, pivot)
2:     while l < p do
3:         while A[l] ≤ pivot do l := l + 1
4:         while A[p] > pivot do p := p − 1
5:         Prohoď(l, p)
6:     return p
```

V ideálním případě, kdy se nám podaří vybrat všechny pivoty tak, aby se každý úsek pole rozdělil přesně napůl, dosteneme v nulté hladině rekurze 1 úsek o n prvcích, v první hladině rekurze 2 úseky o $n/2$ prvcích, ve druhé hladině rekurze

4 úseky o $n/4$ prvcích,... Časová složitost quicksortu při tomto dělení úseků je $\mathcal{O}(n + 2(n/2) + 4(n/4) + \cdots + 2^{\lceil \log n \rceil} \cdot 1) = \mathcal{O}(\lceil \log n \rceil \cdot n) = \mathcal{O}(n \log n)$.

Ovšem pokud budeme mít smůlu a pivot bude vždy nejmenším nebo největším prvkem z aktuálního úseku, tak bude bude časová složitost quicksortu $\mathcal{O}(n + (n - 1) + (n - 2) + \cdots = \mathcal{O}(n^2)$.

Quicksort má v nejhorším případě kvadratickou časovou složitost, ale dá se o něm ukázat, že časová složitost v průměrném případě je pouze $\mathcal{O}(n \log n)$. Upočítání vyžaduje dobrou znalost teorie pravděpodobnosti a proto ho přenecháme jiným knížkám (ale jinak je to jednoduché).

Poznámky k implementaci: Při implementaci quicksortu můžeme místo rekurze využít zásobníku, na který si budeme ukládat úseky pole, které je potřeba setřídit pomocí quicksortu. Při zpracování úseku uloženého na vrcholu zásobníku vyrobíme 2 nové podúseky. Myslíte, že záleží na pořadí, v jakém nové podúseky uložíme na zásobník? Samozřejmě, že ano. Vždy uložíme větší úsek pole dospod. Rozmyslete si, jak to ovlivní velikost paměti potřebné pro zásobník.

Druhým trikem při implementaci je pozorování, že pro malé vstupy je Insersort rychlejší než Quicksort. Proto od určité velikosti podúseků skončíme s rekurzivním voláním Quicksortu a podúsek dotřídíme Insertsortem.

2.4 Amortizovaná časová složitost

Amortizovaná časová složitost je v něčem podobná časové složitosti v průměrném případě. Podává lepší informaci o algoritmu než časová složitost v nejhorším případě, ale tentokrát nepotřebujeme nic počítat přes všechny možné vstupy, nepotřebujeme používat žádnou pravděpodobnost.

Když pracujeme s datovou strukturou (například s polem), tak můžeme veškerou práci s datovou strukturou realizovat pomocí několika operací. *Operace* je něco jako funkce pro práci s datovou strukturou. Operace provedou, co potřebujeme, a odstíní nás od znalosti fungování datové strukury. Příkladem operace pro práci s polem je vložení prvku do pole, smazání prvku, nalezení minima, apod.

Definice (Amortizovaná časová složitost): Je dána datová struktura D, na které postupně provádíme posloupnost stejných operací. Začneme s $D_0 := D$. První operace zavolaná na D_0 upraví datovou strukturu na D_1. Druhá operace zavolaná na D_1 upraví datovou strukturu na D_2. A tak dále. Postupně zavoláme i-tou operaci na D_{i-1} a ta upraví datovou strukturu na D_i. Některá operace může trvat krátce, jiná déle. Průměrný čas doby trvání operace nazveme amortizovanou časovou složitostí. *Amortizovanou časovou složitost* jedné operace spočítáme tak, že spočteme celkovou časovou složitost posloupnosti operací v nejhorším případě a vydělíme ji počtem operací.

K čemu je amortizovaná časová složitost? Pomůže nám lépe odhadnout časovou složitost některých algoritmů v nejhorším případě.

Příklad: Máme algoritmus, který používá jen jednu datovou strukturu a n krát volá operaci, která tuto datovou strukturou upravuje. Nic jiného nedělá. Časová složitost operace je v nejhorším případě $\mathcal{O}(n)$, ale její amortizovaná časová složitost je jen $\mathcal{O}(\log n)$. Celkovou časovou složitost celého algoritmu můžeme spočítat jako $n \cdot \mathcal{O}(n) = \mathcal{O}(n^2)$. To ale může být mnohem více než skutečná časová složitost algoritmu. Druhá možnost odhadu časové složitosti je s pomocí amortizované časové složitosti. Ikdyž bude jeden konkrétní průběh operace trvat $\mathcal{O}(n)$, můžeme odhadnout časovou složitost algoritmu v nejhorším případě jako n krát amortizovaná časová složitost jedné operace. To je jen $\mathcal{O}(n \log n)$.

Výpočet amortizované časové složitosti. K výpočtu amortizované časové složitosti jedné operace se nejčastěji používají následující metody. Pro lepší pochopení metod se podívejte na konkrétní příklady výpočtu v následujících podsekcích.

1. **Přímo z definice.** Spočteme časovou složitost posloupnosti n operací v nejhorším případě a podělíme ji n.

2. **Účetní metoda.** Předpokládejme, že uhodneme, kolik je amortizovaná časová složitost operace. Jak ověřit, že je to pravda? Představme si výpočet na stroji, kde musíme za každou časovou jednotku strávenou výpočtem zaplatit jednu korunu. Na začátku přidělíme každé operaci[8] tolik korun, kolik je její amortizovaná složitost. V algoritmu pracujeme s objekty jako je vrchol, hrana, políčko pole,... Každému objektu otevřeme účet.

   Pokud bude operace trvat kratší čas, než kolik je její amortizovaná složitost, tak zaplatí za svůj výpočet a ještě jí zbydou peníze. Ty uloží na účty objektů, na kterých pracuje. Pokud bude naopak trvat delší čas, než je její amortizovaná složitost, tak si peníze sebere z účtů objektů, na kterých pracuje. Díky tomu bude moci zaplatit za svůj výpočet.

   Metoda spočívá v nalezení takových pravidel ukládání a vybírání peněz z účtů, aby se žádný účet nedostal do mínusu a abychom byli schopni zaplatit za všechnu vykonanou práci. Pokud se nám to povede, tak pomocí pravidel ověříme, že je náš odhad hodnoty amortizované časové složitosti správný.

3. **Metoda potenciálu**. Využijeme účtů z účetní metody. Potenciál je celkový vklad v bance, kde máme účty. Jinými slovy je to součet aktuálních hodnot vkladů na všech účtech.

   Pojďme si to ale vysvětlit pořádně. Nechť a je amortizovaná cena jedné operace. Začneme s datovou strukturou D_0 a postupně budeme provádět n operací. Nechť c_i je skutečná cena provedení i-té operace a D_i je datová struktura, která vznikla z D_{i-1} zavoláním i-té operace. Potenciální funkce Φ přiřazuje každé datové struktuře D_i reálné číslo $\Phi(D_i)$, které nazveme potenciál spojený s datovou strukturou D_i. Každá operace dostala tolik korun, kolik je její amortizovaná složitost. Pokud trvala kratší čas, než je její amortizovaná časová složitost, tak ušetřila peníze. Ušetřené peníze vloží do banky. Tím se zvýší potenciál (vklad v bance) o $\Phi(D_i) - \Phi(D_{i-1})$. Pokud operace trvala déle, než je její amortizovaná složitost, tak si peníze z banky naopak vybrala. V tomto případě je potenciálový rozdíl $\Phi(D_i) - \Phi(D_{i-1})$ záporný. Amortizovaná cena i-té operace vzhledem k potenciálu Φ je $a = c_i + \Phi(D_i) - \Phi(D_{i-1})$.

   Celková amortizovaná cena všech n operací je $\sum_{i=1}^{n}(c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$. Pokud skončíme s $\Phi(D_n) - \Phi(D_0) \geq 0$, tak nám na účtech zůstaly ještě nějaké penízky a amortizovaná časová složitost všech n operací je horním odhadem časové složitosti. Pokud skončíme s $\Phi(D_n) - \Phi(D_0) < 0$, tak se banka zadlužila musíme tuto hodnotu započítat do celkové časové složitosti všech operací.

   Mohlo se stát, že jsme na začátku nezačínali s prázdnou datovou strukturou, ale už jsme ji přebrali odjinud. V takovém případě potřebujeme na začátku vložit na účty počáteční vklad $\Phi(D_0)$, aby fungovala pravidla pro vkládání a vybírání peněz (jinak se banka dostane do mínusu). Podobně pokud skončíme s datovou strukturou, o kterou jsme pečovali a pilně střádali penízky na budoucí drahé operace, tak nám na účtech zbyde $\Phi(D_n)$ peněz.

[8] Abychom byli korektní, měli bychom místo „každé operaci“ říkat „každé instanci operace“.

> Metoda potenciálu spočívá v nalezní potenciální funkce Φ a ověření, že potenciálový rozdíl $\Phi(D_{i-1}) - \Phi(D_i)$ spolu s amortizovanou složitostí operace pokryje náklady na provedení i-té operace.

Účetní metoda se používá i v následující variantě. Peníze rozdělíme na účty jednotlivých objektů. Operace nedostane žádné peníze, ale vždycky řekne, ze kterého účtu se bude její práce platit (většinou to jsou účty objektů, na kterých operace pracuje). Po provedení všech operací musíme ukázat, že se žádný účet nedostal do mínusu. Celková časová složitost posloupnosti operací bude odpovídat množství peněz, které jsme na začátku vložili na účty.

Amortizovanou složitost nemusíme počítat jen pro operace, ale i pro řádky zdrojového kódu, které se provádí několikrát. Například pro řádky v cyklu nebo v několika do sebe vnořených cyklech. Protože časová složitost vybraných řádek může být proměnlivá, tak nepočítáme celkovou časovou složitost podle počtu do sebe vnořených cyklů, ale přes účty. Díky tomu se nám často podaří ukázat lepší odhad.

V následujících podsekcích si na ukážeme výpočet amortizované časové složitosti pomocí všech tří metod na příkladech.

2.4.1 Kavárna „U Zavěšeného kafe“

V Praze je kavárna „U Zavěšeného kafe“. Na jedné zdi v kavárně visí několik hrnečků. Někteří hosté se mohou projevit jako dobrodinci. Když platí stovkou nebo jinou papírovou bankovkou, tak řeknou: „To je dobrý, nic mi nevracejte. Zbytek dejte do toho hrnku za zdi.“ A když do kavárny přijde nějaký chudý student, tak může barmana poprosit, jestli si může dát kafe na účet toho hrnku na zdi. Pokud je v hrnku dostatek peněz, tak dostane kafe.

Amortizovaná cena jednoho kafe je přesně tolik, kolik si účtuje kavárna. Ale jeho reálná cena je pro každého jiná. Bohatí dobrodinci vydají ze své peněženky za kafe víc. Normální lidé platí tolik, kolik kafe stojí a chudí studenti platí méně, protože část ceny hradí z hrnku na zdi.

2.4.2 Nafukovací pole

Potřebujeme navrhnout funkci $pridej(x)$, která do pole přidá prvek x.

Pokud dopředu nevíme, jak velké pole budeme potřebovat, tak začneme s malým polem velikosti jedna[9] a v případě potřeby ho zvětšíme. Vždy, když se nám přidávaná položka nevejde do aktuálního pole, tak vytvoříme nové pole o dvojnásobné velikosti. Všechny položky do něj zkopírujeme, staré pole zrušíme a x přidáme až do nového pole. Vytvoření nového pole, kopírování prvků a rušení starého pole dohromady zabere čas $tn = \mathcal{O}(n)$, kde n je velikost starého pole a t je nějaká konstanta. Tolik je i časová složitost operace $pridej$ v nejhorším případě. Časová složitost operace $pridej$ je v nejhorším případě výrazně větší než v obyčejném poli, kde přidání prvku trvá jen $\mathcal{O}(1)$.

Ve skutečnosti to není tak zlé, protože časově náročné vytváření nového pole nastává málo často. Pokud jsme právě vytvořili nové pole o $2n$ položkách, tak musíme přidat dalších n čísel, než se pole zaplní a bude ho potřeba znova nafouknout.

Pro přidání n čísel do prázdného pole musíme provést nafukování celkem k-krát, kde $k = \lfloor \log n \rfloor$. Jedno nafouknutí pole n čísel trvá čas tn. Celkový čas všech nafukování je $t(1 + 2 + 4 + 8 + \cdots + 2^{k-1}) = t \cdot 2^k \leq 2tn$. Dohromady s časem za samotné přidání prvků do pole dostáváme amortizovanou složitost jedné operace $(2t + 1)n/n = 2t + 1 = \mathcal{O}(1)$.

[9] Při konkrétním použití bychom začali s větším polem. Přeci jen jsme schopni nějak odhadnout minimální velikost pole.

Druhá možnost výpočtu amortizované složitosti je pomocí účetní metody. Na začátku dáme každé operaci *pridej* $(2t+1)$ korun a každému políčku v poli otevřeme účet. Operace zaplatí jednu korunu za vlastní přidání prvku do pole a zbylých $2t$ korun dá na účet aktuálního políčka. Pokud se pole velikosti k zaplní, znamená to, že jsme od posledního nafouknutí přidali dalších $k/2$ prvků. Na účtě tedy máme tk korun, které použijeme na vytvoření nového pole a zkopírování všech k prvků.

Podobně můžeme počítat i pomocí metody potenciálu. Za potenciál můžeme zvolit $\Phi = 2t \cdot \#\text{prvků v poli}$.

2.4.3 Přičítání jedničky

V čítači máme hodně dlouhé binární číslo x a postupně k němu přičítáme jedničku. Číslo x má n bitů a jeho bity jsou uloženy v poli $\mathtt{A}[\,\cdot\,]$. Nejnižší bit je uložen v $\mathtt{A}[0]$ a nejvyšší bit v $\mathtt{A}[n-1]$, takže $x = \sum_{i=0}^{n-1} \mathtt{A}[i] \cdot 2^i$.

Přičítání děláme tak, jak nás to učili na základní škole. Zkusíme přičíst jedničku k poslednímu bitu. Pokud je nultý bit nula, tak jej přepíšeme na jedničku a jsme hotoví. Pokud je nultý bit jednička, tak obě jedničky sečteme a dostaneme "10", zapíšeme nulu a jedničku přeneseme o bit výše. Tam ji přičítáme k prvnímu bitu úplně stejně, jako jsme to dělali u nultého bitu. Pokud dojde k dalšímu přenosu, tak pokračujeme analogicky. Práci na úrovni jednoho bitu označíme za jeden krok.

Jak dlouho bude trvat přičtení jedničky? To záleží na aktuálním čísle v čítači. Pokud bude na konci nula, tak jsme po jednom kroku hotoví. Ale pokud bude na konci čísla k jedniček, tak budeme muset provést $k+1$ kroků. V nejhorším případě bude přičtení jedničky trvat $\mathcal{O}(n)$.

Jak dlouho bude trvat přičtení m jedniček? Předpokládejme, že je čítač na začátku vynulován. V každém druhém přičtení je poslední bit čítače nula (v čítači je sudé číslo) a proto bude přičítání trvat jen jeden krok. Tuto úvahu můžeme zobecnit. Pokud dojde k přenosu na k-tém bitu, tak muselo dojít i k přenosu na všech nižších bitech a tím pádem jsou všechny nižší bity vynulovány. Aby se na pozici k-tého bitu dostala opět jednička, budeme muset aspoň 2^k krát přičíst jedničku.

Proto můžeme říci, že krok na úrovni nultého bitu proběhne pokaždé, krok na úrovni prvního bitu jen při každém druhém přičtení, krok na úrovni třetího bitu jen při každém čtvrtém přičtení a tak dále. Celkem tedy proběhne $m + m/2 + m/2^2 + m/2^3 + \cdots + m/2^l \leq 2m$ kroků, kde $l = \lfloor \log m \rfloor$ (použili jsme vzorec pro součet geometrické řady $1 + 1/2 + 1/4 + 1/8 + \cdots = 2$). Z toho dostáváme amortizovanou časovou složitost jednoho přičtení $2m/m = 2$.

Druhou možností, jak určit amortizovanou časovou složitost, je použití účetní metody. Každému bitu otevřeme účet. Po celou dobu bude platit, že hodnota bitu udává počet korun, které má na účtě. Začneme s vynulovaným čítačem. Každá operace přičtení jedničky dostane dvě koruny. Jednu použije na první krok a druhou vloží na účet nultého bitu. Pokud by při tomto přičítání došlo k přenosu, tak jsou na účtu nultého bitu dvě koruny. Ty vybereme, dáme je přenášené jedničce a nultý bit nastavíme na nulu. Přenášená jednička má zase dvě koruny a může je na úrovni vyššího bitu použít úplně stejně. Tímto způsobem je každá operace přičtení schopna zaplatit za svoji práci. Proto je amortizovaná složitost jedné operace přičtení jedničky dva.

Pokud na začátku nezačínáme s prázdným čítačem, tak musíme každé jedničce v čítači vložit na účet jednu korunu. Zbytek už proběhne stejně.

Třetí možnost výpočtu amortizované složitosti je přes potenciál. Za potenciál $\Phi(i)$ zvolíme počet jedničkových bitů binárního čísla (po přičtení i jedniček). To, že potenciál funguje, se ukáže stejně jako u účetní metody.

2.4.4 Počítání stupňů vrcholů

Dostaneme graf s vrcholy $\{1, 2, \ldots, n\}$ a m hranami reprezentovaný seznamem sousedů,[10] tj. pro každý vrchol dostaneme spojový seznam sousedních vrcholů. V tomto grafu bychom chtěli pro každý vrchol spočítat jeho stupeň.[11] Můžeme postupovat podle následujícího algoritmu.

```
1: for v = 1 to n do
2:     deg[v] := 0
3:     for all w ∈ sousedi[v] do
4:         deg[v] := deg[v] + 1
```

Jaká je časová složitost tohoto algoritmu? Mohli byste říci, že se algoritmus skládá ze dvou for-cyklů, každý for-cyklus proběhne nejvýše n-krát a proto bude časová složitost algoritmu $\mathcal{O}(n^2)$. Ale ono se dá ukázat, že časová složitost je jen $\mathcal{O}(n + m)$.

K výpočtu amortizované časové složitosti použijeme variantu účetní metody. Za operace budeme považovat jednotlivé řádky algoritmu. Každému vrcholu grafu a každé hraně otevřeme v bance účet. Řádku 2 a průběh prvním for-cyklem budeme účtovat aktuálnímu vrcholu v. Řádku 4 a průběh druhým for-cyklem budeme účtovat hraně vw.

Po skončení algoritmu jsme z účtů zaplatili všechnu práci, kterou algoritmus vykonal. Každému vrcholu jsme z účtu strhli jen 2 koruny za provedení řádek 1 a 2. Každé hraně jsme strhli $2 + 2$ koruny za provedení řádek 3 a 4 dvakrát (na hranu jsme se podívali z každého koncového vrcholu jednou). Celkem jsme zaplatili $2n + 4m = \mathcal{O}(n + m)$ korun.

2.5 Příklady

2.5.1 Výpočet časové složitosti a asymptotické notace

1. (Procvičení notací velké $\mathcal{O}$, Ω a Θ) Pro následující dvojice funkcí f a g rozhodněte, jestli platí $f = \mathcal{O}(g)$ nebo $f = \Omega(g)$ a nebo oboje, tj. $f = \Theta(g)$.

(a)	$n - 100$	$n - 200$
(b)	$(n + 42)^8$	n^8
(c)	$n^{1/2}$	$n^{2/3}$
(d)	$100n + \log n$	$n + (\log^2 n)$
(e)	$n \log n$	$100n \log(16n)$
(f)	$\log 2n$	$\log 3n$
(g)	$10 \log n$	$\log(n^2)$
(h)	$\log \log n$	$\sqrt{\log n}$
(i)	$n^{1.01}$	$n \log n$
(j)	$n^2 / \log n$	$n \log^2 n$
(k)	$n^{0.1}$	$\log^{10} n$
(l)	$(\log n)^{\log n}$	$n / \log n$
(m)	$\sqrt{n}$	$\log^5 n$
(n)	$\sqrt{n}$	$n^{\sin n}$

[10] Reprezentace grafu seznamem sousedů je vysvětlena v kapitole 6 o reprezentacích grafu.
[11] Stupeň vrcholu v je počet hran, které z vrcholu v vycházejí.

(o)	$n^{1/2}$	$2^{\log_2 n}$
(p)	$n2^n$	3^n
(q)	2^n	2^{n+1}
(r)	$n!$	2^n
(s)	$(\log n)^{\log n}$	$2^{(\log_2 n)^2}$
(t)	$\sum_{i=1}^{n} i^k$	n^{k+1}
(u)	$2^{\mathcal{O}(n)}$	$5^{\mathcal{O}(n)}$

Pokud se chcete pocvičit ještě o něco více, tak seřaďte všechny výše uvedené funkce podle podle asymptotické časové složitosti (zápis "$= \mathcal{O}$" bereme jako uspořádání "$\preceq$").

2. (Najděte chybu) Indukcí dokážeme, že $f(n) = n^2 \in \mathcal{O}(n)$. Pro $n = 1$ to platí. Předpokládejme tedy, že tvrzení platí až do nějakého n. Ukážeme, že platí i pro následníka. $f(n) = n^2 = (2n+1)+(n-1)^2 = \mathcal{O}(n)+f(n-1) = \mathcal{O}(n)+\mathcal{O}(n) = \mathcal{O}(n)$. Čtvrtá rovnost platí díky indukčnímu předpokladu $f(n-1) = O(n)$.

3. (Procvičení výpočtu časové složitosti – násobení $a \cdot b$ pomocí sčítání)

   Dostaneme dvě čísla $a, b \in \mathbb{N}$ a chceme spočítat jejich součin. Předpokládejme, že pracujeme na počítači, kde má každá proměnná neomezený počet bitů. Práce s takovými proměnnými není jednoduchá a tak nemůžeme použít instrukci pro násobení ani instrukci pro bitový posun. Máme k dispozici pouze instrukci pro sčítání. Pro jednoduchost předpokládejme, že součet dvou čísel trvá konstantní čas.[12]

   (a) Triviální řešení je následující:

   ```
   mezivysledek := b
   for i = 2 to a do
       mezivysledek := mezivysledek + b
   return mezivysledek
   ```

   Ukažte, že tento algoritmu má exponenciální časovou složitost ve velikosti vstupu.

   (b) Vymyslete řešení s lineární časovou složitostí ve velikosti vstupu.

2.5.2 Dolní odhad časové složitosti

1. (Dolní odhad pro vyhledávání v setříděném poli na základě porovnávání)

   Pomocí binárního vyhledávání (půlení intervalu) umíme v čase $\mathcal{O}(\log n)$ zjistit, jestli setříděné pole $\mathtt{A}[1, \ldots, n]$ obsahuje číslo x. Ukažte, že každý algoritmus, který pouze porovnává prvky pole s číselnou hodnotou, tj. ptá se na "$\mathtt{A}[i] \leq z$?", má časovou složitost v nejhorším případě alespoň $\Omega(\log n)$. Jinými slovy každému takovému algoritmu může ďábel podstrčit ošklivý vstup, na kterém algoritmus vykoná alespoň $\log n$ kroků.

2.5.3 Hledání algoritmu s co nejlepší časovou složitostí

1. (Které číslo chybí?) Množina C obsahuje všechna čísla 1 až n kromě jednoho. Na vstupu postupně dostanete všechna čísla z množiny C a vaším úkolem je zjistit, které číslo v množině chybí.

[12] Ve skutečnosti takoví kouzelníci nejsme. Sečtení dlouhých čísel vyžaduje čas úměrný počtu bitů potřebných pro reprezentaci obou čísel.

(a) Umíte to v lineárním čase?

(b) A co když můžeme použít jen konstantně mnoho paměti?

(c) Co když v množině budou chybět 2 čísla? Jak rychle zjistíte, která to jsou?

2. (Max gap) Dostanete n reálných čísel z intervalu $\langle 0, 1\rangle$. První dvě z nich jsou vždy 0 a 1. Velikost mezery mezi čísly $a, b \in \mathbb{R}$ je $|b - a|$. Mezera je prázdná, pokud interval mezi a a b neobsahuje žádné jiné ze zadaných čísel. Zjistěte, která dvě zadaná čísla mají mezi sebou největší prázdnou mezeru, a vypište je spolu s velikostí mezery.

   Umíte je najít v čase $\mathcal{O}(n)$?

3. (Palindrom) Palindrom je slovo které se čte stejně zepředu i pozpátku. Na vstupu dostanete řětězec n znaků. Navrhněte algoritmus, který v řetězci co nejrychleji najde nejdelší palindrom a vypíše jeho délku. Své řešení otestujte například na řetězcích: “madam”, “mam”, “rotor”, “kuna nese nanuk”.

   Umíte to rozhodnout v čase $\mathcal{O}(n)$?

2.5.4 Amortizovaná časová složitost

1. (Nafukovací i smršťovací pole)

   V sekci o amortizované časové složitosti jsme si vysvětlili, jak funguje nafukovací pole. Co kdybychom chtěli přidat mazání prvků? Když už bude pole poloprázdné, tak bychom ho mohli smrštit na poloviční velikost. Rozmyslete si, při jaké obsazenosti pole by se mělo provádět nafukování nebo smršťování pole, aby amortizovaná časová složitost operací `delete(`x`)` a `insert(`x`)` byla stále konstantní.

2. (Binární vyhledávání + přidávání prvků)

   Chceme si reprezentovat n-prvkovou množinu čísel M. Často budeme volat operace `najdi(`x`)` a `pridej(`x`)`. Operace `najdi(`x`)` zjistí, jestli $x \in M$. Operace `pridej(`x`)` provede $M := M \cup \{x\}$.

   K reprezentaci množiny M použijeme následující datovou strukturu. Nechť $[n_{k-1}n_{k-2}\ldots n_1n_0]_2$ je binárním zápisem čísla n. Množinu M si místo jednoho pole reprezentujeme pomocí $k := \lceil \log(n+1) \rceil$ uspořádaných polí A_0, A_1, …, A_{k-1} o velikostech 1, 2, 4, …, 2^{k-1}. Velikost pole A_i je 2^i. Každé pole A_i je buď celé prázdné nebo celé plné, podle toho, jestli je $n_i = 0$ nebo $n_i = 1$. Celkový počet prvků ve všech polích je $\sum_{i=0}^{k-1} n_i 2^i = n$. Ačkoliv je každé pole setříděné, o velikostech prvků v různých polích nic nevíme.

   (a) Popište, jak v této datové struktuře provádět operaci `najdi(`x`)` a určete její časovou složitost v nejhorším případě.

   (b) Popište, jak v této datové struktuře provádět operaci `pridej(`x`)` a určete její časovou složitost v nejhorším případě. Kromě toho určete i její amortizovanou složitost.

   (c) Zkuste vymyslet, jakým způsobem by se v datové struktuře dalo provádět mazání prvků.

3. (Procvičení metody potenciálu: hledání následníka v binárním stromě)

Máme binární vyhledávací strom T a nějaký jeho vrchol v. Následník[13] vrcholu v je vrchol s nejmenším vyšším klíčem, než je hodnota klíče ve v. Nalezení následníka trvá v nejhorším případě čas $\mathcal{O}(\text{hloubka } T)$. Ukažte, že pokud budeme chtít najít k následníků, tak to zvládneme v čase $\mathcal{O}(k + \text{hloubka } T)$.

Nápověda: Nechť $w \in T$ je aktuální vrchol při hledání následníků. Zkuste potenciál $\Phi = (\text{hloubka } T - \text{hloubka } w \text{ v } T) + 2 \cdot \#\text{pravých hran na cestě z kořene do } w$.

[13] Pozor, následník vrcholu (syn) je něco jiného než zde zavedený následník. Pro definici hloubky stromu a podobných pojmů se podívejte do kapitoly 5.

Kapitola 3

Rozděl a panuj

Rozděl a panuj je programovací metoda. Často se označuje latinsky *"Divide et Empera"* nebo anglicky *"Divide and Conquer"*. Vychází z toho, že umíme zadaný problém rozložit na menší podproblémy stejného typu.

Na začátku rozdělíme práci a určíme, které podproblémy je potřeba vyřešit. Tyto podproblémy si necháme vyřešit rekurzivně. Složením jejich řešení dostaneme řešení původního problému. Algoritmus se volá rekurzivně tak dlouho, dokud se nedostane k problémům konstantní velikosti, které už umí hravě vyřešit.

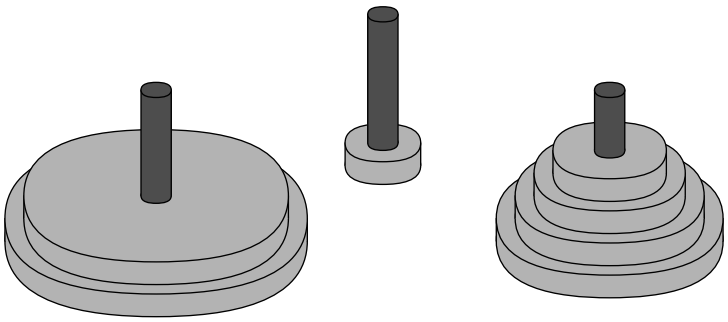
Vysvětlíme si to ještě jednou na pohádce o tom, jak si římský císař hraje na popelku. Inu i římskému císaři se někdy stane, že mu v sýpkách smíchají čočku a hrách. Jak to roztřídit? Císař přeci nemůže dělat takovou práci sám, má na to lidi. A tak si zavolá vojáky a nechá směs čočky a hrachu rozdělit na tři zhruba stejně velké hromádky. Potom zavolá své generály, každému přiřadí jednu hromádku a nechá je oddělit čočku od hrachu. Ale i takový generál se nebude dřít sám. Po vzoru císaře, rozdělí svojí hromádku na tři a rozdělí práci mezi své podřízené. Takto projde předávání práce celou hierarchií římské armády, až to skončí u otroků. Každý otrok dostane jednu misku směsi a tu roztřídí. Jeho pán si od všech svých otroků vybere zvlášť čočku a zvlášť hrách a předá je výše. Takto projde slučování hromádek zpátky až k císaři, který, ke své spokojenosti a rozmarnosti, nechá přesunout všechnu čočku zpátky do sýpky a hrách hodí sviním, aby se příště s čočkou nesmíchal.

Použití metody rozděl a panuj si ukážeme v následujících úlohách.

3.1 Hanojské věže

Příběh: V jednom indickém chrámu mají tři věže – Věž zrození, Věž života a Věž zkázy. Uvnitř každé z nich je kůl, na kterém je navlečeno několik zlatých disků. Ve všech třech věžích je dohromady 64 disků. Každý disk je jinak široký. Disky smí být na kůlu navlečeny pouze tak, že menší a užší disk leží na širším. Jinak to prý přinese smůlu. Proto každý kůl spolu s disky vypadá jako kužel.

Při stvoření chrámu byly všechny disky navlečeny na jeden kůl ve Věži stvoření. Kněží, kteří v chrámu přebývají, každý den přesunou jeden disk (víc jich přesunout nesmí) tak, aby se jim co nejdříve podařilo přesunout všechny disky na kůl ve Věži zkázy. Jinak by se zastavilo plynutí života. Stará legenda říká, že až se jim to podaří, tak nastane konec světa.



Úkol pro Vás: Dokázali byste napsat program, který kněžím vypíše instrukce, jak mají disky přesunovat? Už jsou z toho celí zmatení a moc by jim to pomohlo. Na začátku jsou všechny disky navlečeny na první kůl a máte je přestěhovat na poslední kůl.

Pokud se budeme držet zásady rozděl a panuj, tak zkusíme problém rozložit na podproblémy, jejichž vyřešení přehodíme na někoho jiného (třeba na rekurzi). Abychom mohli největší disk přesunout na správné místo, tak nejprve necháme všechny menší disky přestěhovat na třetí odkládací tyčku. Pak přesuneme největší disk. Na závěr necháme všechny odložené disky přesunout nad největší disk.

```
1: Hanoj(n, odkud, pres, kam)
2:     if n ≥ 1 then
3:         Hanoj(n − 1, odkud, kam, pres)
4:         Vypiš("Přeneste disk z  ",odkud," do  ", kam)
5:         Hanoj(n − 1, pres, odkud, kam)
```

Jaká je časová složitost tohoto algoritmu? Ta je úměrná tomu, kolikrát budou muset kněží přenést nějaký disk. Označme celkový počet přenesení disků pomocí $T(n)$. Z uvedeného algoritmu je vidět rekurence $T(n) = 2T(n-1) + 1$. Rekurenci můžeme vyřešit například postupným rozepisováním. Dostaneme

$$T(n) = 2T(n-1) + 1 = 2(2(T(n-2) + 1) + 1 =$$
$$= 2(2(2 \cdots (2T(1) + 1) \cdots + 1) + 1) + 1 = 2^n + 2^{n-1} + \cdots + 2 + 1 = 2^{n+1}.$$

Takže časová složitost algoritmu Hanoj je exponenciální. Legenda říká, že disků je 64. Jestliže byl chrám založen zhruba před 3000 lety, dokážete určit kdy nastane konec světa? Má smysl se toho obávat?

3.2 Mergesort

Mergesort je třídící algoritmus využívající metody rozděl a panuj. Dostane vstup, například posloupnost čísel, který rozdělí na dvě skoro stejné části. Ty nechá setřídit rekurzivně. Výsledné posloupnosti slije do jedné a tu vrátí jako setříděnou posloupnost.

Co znamená slít dvě setříděné posloupnosti P_1, P_2 do jedné? Cílem je vytvořit jednu posloupnost P, která obsahuje prvky obou předchozích posloupností a je setříděná. Menší prvek z počátečních (a nejmenších) prvků setříděných posloupností P_1, P_2 je určitě nejmenším prvkem výsledné posloupnosti P. Proto ho odtrhneme a přidáme na začátek P. Tím z P_1, P_2 vzniknou posloupnosti P_1', P_2'. Pro ty můžeme celý postup zopakovat a tím dostaneme druhý nejmenší prvek P. Dalším opakováním postupně odtrháme všechny prvky obou setříděných posloupností P_1, P_2 a sestavujeme požadovanou posloupnost P.

Slévání si můžeme představovat trochu jako zip na oblečení. Dvě ozubené části zipu (setříděné posloupnosti) přiložíme k sobě a zapneme je jezdcem. Jezdec kontroluje, že do sebe ozubené části zapadají podle velikosti.

Klasická implementace Mergesortu pracuje se spojovým seznamem prvků. Pokud chceme mergesort implementovat v poli, tak se neobejdeme bez pomocného pole $\texttt{B}[\cdot]$, kam budeme ukládat mezivýsledky. Procedura Mergesort(l, p) setřídí úsek pole $\texttt{A}[l..p]$ mezi indexy l a p. Pole $\texttt{A}[n]$ i pomocné pole $\texttt{B}[n]$ jsou globální proměnné.

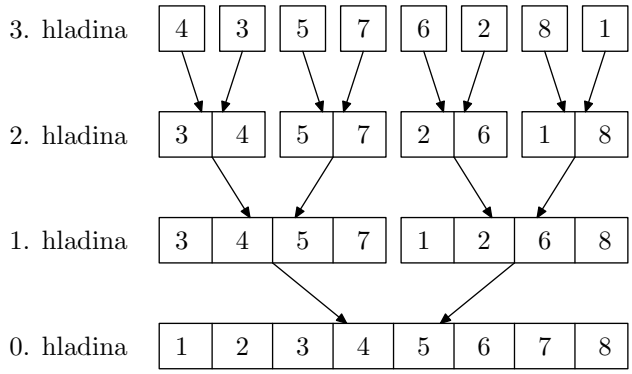
```
1: Mergesort(l, p)
2:     if l < p then
3:         stred := ⌊(l + p)/2⌋
4:         Mergesort(l, stred)
5:         Mergesort(stred + 1, p)
6:         Merge(l,p)
```

Procedura Merge(l, p) slije setříděné podposloupnosti $\texttt{A}[l..stred]$ a $\texttt{A}[stred + 1..p]$ do jedné setříděné posloupnosti $\texttt{A}[l..p]$. Pro ukládání mezivýsledků potřebuje pomocné pole $\texttt{B}[l..p]$. Procedura Merge má časovou složitost $\mathcal{O}(m)$, kde m je délka výsledné posloupnosti.

```
1: Merge(l, p)
2:     stred := ⌊(l + p)/2⌋
3:     i := l              {Index na začátek první setříděné posloupnosti. }
4:     j := stred + 1      {Index na začátek druhé setříděné posloupnosti.}
5:     k := l              {Index do pole B[ · ], kam zapisujeme výsledek.}
6:     while k ≤ p do
7:         if i > stred then                {První posloupnost došla.}
8:             B[k] := A[j]
9:             j := j + 1
10:        else if j > p then               {Druhá posloupnost došla.}
11:            B[k] := A[i]
12:            i := i + 1
13:        else if A[i] < A[j] then         {Žádná posloupnost nedošla.}
14:            B[k] := A[i]
15:            i := i + 1
16:        else
17:            B[k] := A[j]
18:            j := j + 1
19:        k := k + 1
20:    for k := l to p do {Kopírování výsledků z B[l..p] do A[l..p]}
21:        A[k] := B[i]
```

Časová složitost Mergesort je určena rekurencí $T(n) = 2T(n/2) + \mathcal{O}(n)$, protože dvakrát řešíme podúlohu poloviční velikosti a jednou sléváme výsledky dohromady.

Následující obrázek ilustruje průběh algoritmu a slévání jednotlivých částí. Strom rekurze máme nakreslený vzhůru nohama, protože se algoritmus nejprve zanoří, co nejhlouběji to jde, a teprve při zpáteční cestě provádí slévání podposloupností (úseků pole). Pro lidi je přirozenější ta zpáteční cesta, na které se něco děje.



Strom rekurze má hloubku nejvýše $\lceil \log n \rceil$, protože při každém zavolání Mergesort rozdělíme posloupnost v půlce. Na k-té hladině[1] se slévají dvě posloupnosti délky $n/2^{k+1}$ do posloupnosti délky $n/2^k$. Těchto slévání se na k-té hladině vykoná 2^k. Proto je celková práce na k-té hladině rovna $\mathcal{O}(n)$. Z toho dostáváme časovou složitost algoritmu mergesort $\mathcal{O}(n \log n)$.

Poznámka k implementaci: Na konci každého volání Merge kopírujeme výsledky z pomocného pole `B`$[\cdot]$ do pole `A`$[\cdot]$. Tomu se můžeme vyhnout tak, že budeme při rekurzivním volání Mergesortu střídat význam pole `A`$[\,\cdot\,]$ a `B`$[\,\cdot\,]$. Na sudých hladinách dostane Merge posloupnost v poli `B`$[\,\cdot\,]$ a vytvoří setříděnou posloupnost do pole `A`$[\,\cdot\,]$ a v lichých hladinách naopak. Prohazování můžeme zrealizovat tak, že funkci Merge předáme i ukazatele na tyto pole a při rekurzivním zavolání ukazatele prohodíme.

Mohlo by se stát, že střídání polí po hladinách vyjde tak, že budeme v poslední hladině chtít slévat položky z `B`$[\,\cdot\,]$ do pole `A`$[\,\cdot\,]$. Proto na začátku celého algoritmu nakopírujeme celý vstup z `A`$[\,\cdot\,]$ i do pole `B`$[\,\cdot\,]$.

3.3 Medián posloupnosti

Máme posloupnost n prvků, například čísel. *Medián* je takové číslo z posloupnosti čísel, že 50% čísel je větších nebo rovno mediánu, ale také 50% čísel je menších nebo rovno mediánu. Jinými slovy, když si posloupnost setřídíme, tak nazveme prvek, který leží uprostřed, *mediánem.* Pokud je prvků lichý počet, tak je medián ten prostřední prvek. Pokud je počet prvků sudý, tak máme mediány dva – spodní a horní medián. Abychom si zjednodušili výklad, budeme za medián považovat spodní medián.[2]

Medián se používá ve statistice, protože to je jedno číslo, které vypovídá něco typického o množině čísel. Podobným číslem je průměr.[3]

Poznámka: Kolik informace nám přinese údaj o průměrné mzdě v Heské republice? Pro běžné lidi je to jen číslo, kvůli kterému jsou nespokojeni, že nevydělávají dost. Pokud bude 10% obyvatel vydělávat 110tis. Kč/měsíc a zbylých 90% obyvatel jen 10tis. Kč/měsíc, tak je průměrná mzda v Heské republice 20tis. Kč. Ovšem kdo ji má? Naproti tomu medián platů v Heské republice je 10tis. Kč/měsíc. To je číslo, které mnohem lépe popisuje celou situaci.

[1] k-tou hladinou myslíme k-tou hladinu od kořene. Na obrázku je strom nakreslen vzhůru nohama a proto je to na obrázku k-tá hladina od zdola.

[2] Běžně se za medián bere průměr spodního a horního mediánu. To je ale hodnota, ne prvek.

[3] Víte o tom, že máte nadprůměrný počet očí? Tedy pokud nejste ten výjimečný případ.

Úkol: Dosteneme pole obsahující n různých čísel.[4] Jak co nejrychleji najít medián? Nebo když tento problém zobecníme, jak co nejrychleji najít k-tý nejmenší prvek?[5]

Řešení 1: Nejjednodušším řešením je, že si pole setřídíme a potom vypíšeme k-tý prvek. Třídění nám zabere čas $\mathcal{O}(n \log n)$.

Řešení 2: Můžeme upravit Quicksort ze strany 12 tak, aby po rozdělení úseku pole $\mathtt{A}[l..p]$ pivotem nezpracovával levý i pravý úsek pole, ale aby už pracoval jen s úsekem, ve kterém leží k-tý nejmenší prvek.

```
1:  QuickSelect(l, p, k)
2:      if l < p then
3:          pivot := A[⌊(l + p)/2⌋]
4:          q :=Partition(l, r, pivot)
5:          if q ≤ k then
6:              QuickSelect(l, q, k)
7:          else
8:              QuickSelect(q + 1, p, k)
9:      else
10:         Output(A[l])
```

Připomeňme, že funkce q :=Partition(l, p, $pivot$) rozdělí úsek pole $\mathtt{A}[l..p]$ na dva úseky $\mathtt{A}[l..q]$ a $\mathtt{A}[q+1..p]$, kde první úsek obsahuje pouze prvky menší nebo rovny pivotu a druhý úsek pouze prvky větší než pivot. Implementace funkce je popsána u quicksortu na straně 12.

V ideálním případě, kdy se nám podaří vybrat všechny pivoty tak, aby se každý úsek pole rozdělil přesně napůl, bude časová složitost tohoto řešení $\mathcal{O}(n + n/2 + n/4 + \dots) = \mathcal{O}(2n) = \mathcal{O}(n)$. Ovšem pokud budeme mít smůlu a pivot bude vždy nejmenším nebo největším prvkem z aktuálního úseku, tak bude časová složitost tohoto řešení $\mathcal{O}(n + (n-1) + (n-2) + \cdots + 1) = \mathcal{O}(n^2)$. Naštěstí, stejně jako u quicksortu, v průměrném případě nastane první varianta a časová složitost bude lineární.

Řešení 3: Předchozí řešení vylepšíme tak, že si v každé iteraci vybereme dobrého pivota, který leží dostatečně daleko od okrajů aktuálního úseku. Chceme, aby v úseku pole $\mathtt{A}[l..p]$ délky n vždy existovalo aspoň $3n/10$ prvků menších než pivot a $3n/10$ prvků větších než pivot. Pivota vybereme následovně:

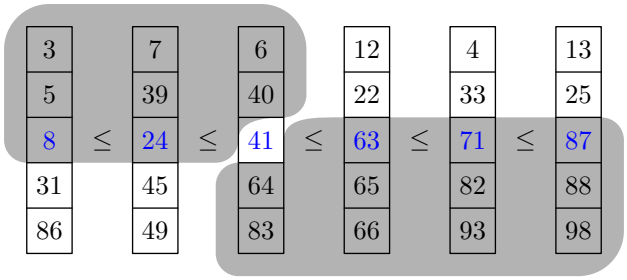
1. Prvky pole rozdělíme do pětic a v každé pětici nalezneme medián.

2. Vezmeme pouze mediány ze všech pětic a rekurzivně pomocí QuickSelect() na nich najdeme medián.

Podívejme se na následující obrázek. V každé pětici jsou hodnoty setříděné podle velikosti a sloupce s pěticemi jsou seřazeny podle hodnot jejich mediánů. Algoritmus nepotřebuje nic třídit, pouze nalezne v každé pětici medián (znázorněn modře) a potom nalezne medián mediánů (na tomto obrázku má hodnotu 41).

[4] Proč různých čísel? Pro různá čísla na vstupu nebudeme muset diskutovat okrajové případy u výpočtu mediánu přes pětice. Zjednoduší se tím výklad, ale algoritmus funguje i pro čísla, která nemusí být různá.

[5] U řady problémů, které se řeší pomocí metody Rozděl a panuj, je jednodušší vyřešit zobecněný problém, než hledat přímé řešení problému.



Všechny prvky, které leží v první šedé oblasti jsou menší než medián mediánů a všechny prvky ve druhé šedé oblasti jsou větší než medián mediánů. Prvků větších než medián mediánů je aspoň $3 \cdot n/10 - 1 \geq 3n/10 - 1$. Podobně prvků menších než medián mediánů je aspoň $3n/10 - 1$. (Protože n nemusí být dělitelné 5, může být v poslední pětici jen 1 prvek. Proto musíme počítat pečlivěji než podle obrázku).

Analýza časové složitosti: Nechť $T(n)$ označuje časovou složitost algoritmu QuickSelect. Nalezení mediána mediánů bude trvat $\mathcal{O}(n) + T(\lceil n/5 \rceil)$. Rekurzivní volání QuickSelect() na podúseku pole proběhne v čase $\mathcal{O}(n) + T(7n/10 + 1)$. Celkem dostáváme $T(n) = T(7n/10 + 1) + T(\lceil n/5 \rceil) + \mathcal{O}(n)$. Teď už stačí jen vyřešit tuto rekurenci.

Zkusíme hledat řešení rekurence ve tvaru $T(n) = cn$, pro nějakou zatím neznámou konstantu c. Dosazením do rekurence dostaneme $T(n) \leq c(7n/10 + 1) + c(\lceil n/5 \rceil) + an \leq c(9n/10 + 2) + an = cn - (cn/10 - 2c - an)$. Pokud bude $(cn/10 - 2c - an) \leq 0$, tak máme vyhráno. Tato podmínka je ekvivalentní s $c \leq 10a(n/(n - 20))$. Stačí zvolit $c = 10a$ a pro $n > 20$ bude řešení $T(n) = cn$ platit. Pro $n \leq 20$ je $T(n) = \mathcal{O}(1)$.

3.4 Master theorem, řešení rekurencí

Při určování časové složitosti algoritmu založeného na metodě Rozděl a panuj se typicky dostaneme k rekurenci, kterou potřebujeme vyřešit. Abychom ji nemuseli zdlouhavě rozepisovat a přemýšlet, jak ji vyřešit, naučíme se následující univerzální metodu.

Věta 1 (Master theorem) *Nechť $T : \mathbb{N} \to \mathbb{N}$ je funkce splňující*

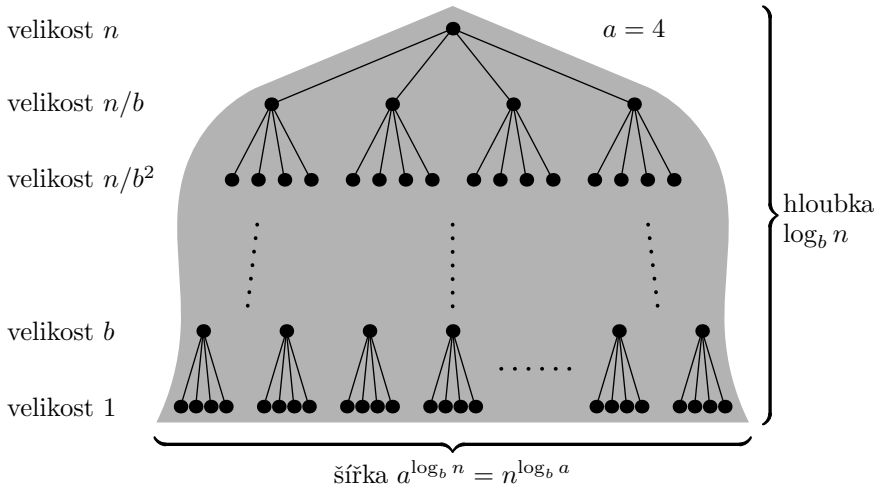
$$T(n) = aT(\lceil n/b \rceil) + \mathcal{O}(n^d)$$

pro nějaké konstanty $a > 0$, $b > 0$ a $d \geq 0$. Potom

$$T(n) = \begin{cases} \mathcal{O}(n^d) & \text{pokud } d > \log_b a \\ \mathcal{O}(n^d \log n) & \text{pokud } d = \log_b a \\ \mathcal{O}(n^{\log_b a}) & \text{pokud } d < \log_b a \end{cases}$$

Důkaz: Pro jednoduchost předpokládejme, že n je mocnina b. Díky je $\lceil n/b \rceil = n/b$.

Velikost podproblémů se při každém rekurzivním zanoření zmenší b krát. Proto se po nejvýše $\log_b n$ zanořeních zmenší až na konstantu (základní případ). Z toho dostáváme, že výška stromu rekurze je nejvýše $\log_b n$.



Při každém rekurzivním zavolání se rozvětvíme na a podproblémů. Proto bude na k-té hladině stromu rekurze a^k podproblémů, každý o velikosti n/b^k. Všechna práce prováděná na k-té hladině trvá

$$a^k \cdot \mathcal{O}(\left(\frac{n}{b^k}\right)^d) = \mathcal{O}(n^d) \cdot \left(\frac{a}{b^d}\right)^k.$$

Časy strávené v hladinách 0 až $\log_b n$ tvoří geometrickou posloupnost s kvocientem a/b^d. Celková časová složitost $T(n) = \mathcal{O}(n^d) \cdot \sum_{k=0}^{\log_b n} \left(\frac{a}{b^d}\right)^k$. Podle hodnoty kvocientu nastanou následující tři případy:

- $a/b^d < 1$. Posloupnost je klesající. Největší je první člen $\mathcal{O}(n^d)$. Součet prvních $\log_b n$ členů geometrické posloupnosti je nejvýše kvocient-krát větší než první člen. Proto $T(n) = \mathcal{O}(n^d)$.
- $a/b^d = 1$. Všechny členy posloupnosti mají stejnou velikost a to $\mathcal{O}(n^d)$. Proto $T(n) = \mathcal{O}(n^d) \cdot \log_b n$.
- $a/b^d > 1$. Posloupnost je rostoucí. Součet prvních $\log_b n$ členů geometrické posloupnosti je nejvýše kvocient-krát větší než poslední a největší člen. Jeho hodnota je $\mathcal{O}(n^d \left(a/b^d\right)^{\log_b n})$.

  $$n^d \left(\frac{a}{b^d}\right)^{\log_b n} = n^d \left(\frac{a^{\log_b n}}{(b^{\log_b n})^d}\right) = a^{\log_b n} = a^{(\log_a n)(\log_b a)} = n^{\log_b a}.$$

  Proto $T(n) = \mathcal{O}(n^{log_b a})$.

Tyto případy přesně odpovídají rozdělení případů z věty.

Zbývá ukázat, že věta platí i pro n, které není mocninou b. Už víme, že věta platí pro n, která jsou mocninou b. Toho využijeme. Nechť n_+ je nejbližší mocnina b větší než n. Podívejme se na první případ, kdy má řešení rekurence tvar $T(n) = \mathcal{O}(n^d)$. Ostaní případy si odůvodníte analogicky.

Funkce $T(n)$ je rostoucí a proto $T(n) < T(n_+)$. Z nerovnosti $n_+ < bn$ plyne, že $\mathcal{O}(n_+^d) = \mathcal{O}((bn)^d) = \mathcal{O}(n^d)$. Poskládáním nerovností dostaneme $T(n) < T(n_+) = \mathcal{O}(n_+^d) = \mathcal{O}(n^d)$. Proto $T(n) = \mathcal{O}(n^d)$. ∎

Jak se dá Master Theorem použít v algoritmech z předchozích sekcí knihy, u kterých už jsme časovou složitost spočítali jinak? Časová složitost binárního vyhledávání (vyhledávání půlením intervalu) je určena rekurencí $T(n) = T(n/2) + \mathcal{O}(1)$.

Pro její vyřešení použijeme Master Theorem s konstantami $a = 1$, $b = 2$, $d = 0$ a dostaneme $T(n) = \mathcal{O}(\log n)$. Časová složitost mergesortu je určena rekurencí $T(n) = 2T(n/2) + \mathcal{O}(n)$. Pro její vyřešení použijeme Master Theorem s konstantami $a = 2$, $b = 2$, $d = 1$ a dostaneme $T(n) = \mathcal{O}(n \log n)$.

Příklad: Vyřešte rekurenci $T(n) = 2T(\sqrt{n}) + \log n$.

Tato rekurence vypadá jako ošklivá čarodějnice, ale můžeme z ní udělat Popelku, když vhodně převlékneme proměnné. Nejprve použijeme substituci proměnné $m = \log n$ a dostaneme $T(2^m) = 2T(2^{m/2}) + m$. Pak použijeme substituci funkce $S(m) = T(2^m)$ a dostaneme $S(m) = 2S(m/2) + m$. Tuhle krásku už umíme vyřešit například pomocí Master Theoremu. $S(m) = \mathcal{O}(m \log m)$. Změnou $S(m)$ zpět na $T(n)$ dostaneme $T(n) = T(2^m) = S(m) = \mathcal{O}(\log n \log \log n)$. (Všechny substituce jsou korektní, protože jsme použili rostoucí funkce.)

3.5 Příklady

1. (Ruční třídění karet)

   Dostanete balíček zamíchaných karet. Jakým způsobem ho setřídíte? Dokážete něco říci o tomto algoritmu? Jakou bude mít časovou složitost? Jaké budou jeho nároky na prostor (paměť)?

2. (Hledání pevného bodu zobrazení)

   Dostaneme setříděné pole různých čísel $\mathtt{A}[1, \ldots, n]$. Chceme zjistit, jestli existuje index i, pro který $\mathtt{A}[i] = i$. Pomocí metody rozděl a panuj navrhněte algoritmus běžící v čase $\mathcal{O}(\log n)$.

3. (Většinový prvek)

   Dostaneme pole n prvků $\mathtt{A}[1, \ldots, n]$. Prvky nemusíme nutně umět porovnávat,[6] ale pro každé dva indexy i, j umíme zjistit, jestli $\mathtt{A}[i] = \mathtt{A}[j]$. Prvek má v poli většinu, pokud se vyskytuje na více než polovině políček. Navrhněte algoritmus, který zjistí, jestli pole obsahuje prvek mající většinu a případně ho vypíše.

   (a) Ukažte, jak vyřešit tento problém v čase $\mathcal{O}(n \log n)$.
   Nápověda: Rozdělte pole na dvě pole poloviční velikosti a zkuste v nich najít prvek mající většinu.

   (b) Najděte algoritmus běžící v lineálním čase.
   Nápověda: Prvky libovolně spárujte. Dostanete $n/2$ párů. Pro každý pár zjistěte, jestli jsou prvky stejné. Pokud ano, tak nechte jeden z nich a druhý prvek smažte. Pokud jsou prvky různé, tak smažte oba. Takto získáme nejvýše $n/2$ prvků. Ukažte, že nově získané prvky mají většinový prvek právě tehdy, když ho má i původních n prvků.

4. (Házení vajíčka z mrakodrapu)

   Na návštěvě Singapuru jste dostali jako dárek na uvítanou "nerozbitelné vajíčko". Tvrdí Vám, že je vyrobeno nejmodernější technologií a i když má jen tuhou skořápku, tak prý nejde rozbít. Rozhodnete se to vyzkoušet a proto budete vajíčko házet z mrakodrapu na chodník.[7]

[6] Prvky mohou odpovídat obrázkům či jiným souborům. Místo dlouhého souboru si stačí pamatovat hodnotu "dlouhé" hašovací funkce. Existují hašovací funkce (například MD5), které nám s velmi vysokou pravděpodobností zaručí, že každý obrázek dostene jinou hodnotu hašovací funkce.

[7] Singapur má jedny z nejvyšších mrakodrapů světa. Než to ale budete realizovat, tak si pořádně prostudujte, co zakazují přísné singapurské zákony. Pokud byste z mrakodrapu plivali, tak vás každé plivnutí vyjde na $1000.

Mrakodrap má n pater. Jaké je nejnižší patro, ze kterého už se vajíčko rozbije? (Klidně se může stát, že se vajíčko nerozbije ani z n-ého patra.) Kolik nejméně pokusů budete muset provést, aby jste to zjistili? Pokusem myslíme jedno hození vajíčka na chodník.

(a) Máte jen jedno vajíčko.

(b) A co když budete mít dvě, naprosto stejná vajíčka?

(c) Jak to zobecnit pro k vajíček?

Nejprve zkuste najít co nejlepší horní odhad na počet pokusů. Teprve pak se zamyslete nad tím, jak dokázat, že to lépe nejde.

Nápověda: Formulka pro počet pokusů není jednoduchá. Stačí, když ukážete, jak spočítat počet pokusů pro n-patrový mrakodrap například na počítači.

5. (Procvičení řešení rekurencí)

Vyřešte následující rekurence. Řešení určete s přesností Θ-notace.

(a) $T(n) = 2T(n/3) + 1$

(b) $T(n) = 5T(n/4) + n$

(c) $T(n) = 7T(n/7) + n$

(d) $T(n) = 9T(n/3) + n^2$

(e) $T(n) = 8T(n/2) + n^3$

(f) $T(n) = 49T(n/25) + n^{3/2} \log n$

(g) $T(n) = 2T(n/2) + n/\log n$

(h) $T(n) = 16T(n/4) + n!$

(i) $T(n) = T(n-1) + 2$

(j) $T(n) = T(n-1) + n^c$, kde $c \geq 1$ je konstanta

(k) $T(n) = T(n-1) + c^n$, kde $c > 1$ je konstanta

(l) $T(n) = 2T(n-1) + 1$

(m) $T(n) = T(\sqrt{n}) + 1$

(n) $T(n) = \sqrt{n}T(\sqrt{n}) + 1$

Nápověda: (k předposlední rekurenci) Zkuste použít substituci $S(m) = T(2^m)$. Substituce je korektní, protože 2^m je rostoucí funkce.

Kapitola 4

Jak zrychlovat programy?

Zde si předvedeme efektivní programovací techniky.
Asymptoticky rychlejším algoritmem zrychlíme program nejvíce.
Uvedeme i praktické informace, kterými program zrychlíme 2krát, 3krát, ale někdy i 10krát.

Možností je několik. Nejprve je ale potřeba zjistit, ve které části programu strávíme nejvíce času. Tím zjistíme, která část je úzkým hrdlem (bottle neck), a tu budeme zrychlovat. Pokud budeme zrychlovat část, ve které výpočet stráví jen 2% celkového času, tak si celkově moc nepomůžeme. Akorát ztratíme spoustu času programováním úprav. Lepší je se zaměřit na proceduru, ve které strávíme 80% celkového času.[1]

Zjistit, ve kterých funkcích strávíme nejvíce času, je dnes velmi jednoduché. Nemusíte nic programovat. Stačí použít nástroj, který se jmenuje profiler. Bývá součástí vývojového prostředí. Spustíte profiler společně se svým programem a rovnou sledujete přehledné statistiky, kolikrát byla spuštěna která funkce a kolik času jste v ní strávili.

Zlaté pravidlo pro optimalizace kódu zní: „Hrajte si s tím, vyzkoušejte všechno možné, porovnejte to a nakonec vyberte to nejlepší řešení.“ Raději ještě jednou zopakujeme nejdůležitější myšlenku celé kapitoly: „Je potřeba mít jasno v tom, které části programu má smysl optimalizovat a které ne.“

Největších zrychlení dosáhneme lepším algoritmem. Teprve nakonec, když už lepší algoritmus nevymyslíme, nebo když si dokonce dokážeme, že lepší algoritmus neexistuje, má smysl se vrhnout na optimalizaci zdrojového kódu a optimalizaci programu pro hardware a operační systém.

Upozornění: kusy kódu algoritmů v této knize nejsou vždy ty nejefektivnější a nejelegantnější. Je to z toho důvodu, že cílem této knihy je hlavně jednoduše a srozumitelně prezentovat grafové algoritmy. Jejich elegantní implementace už hodně závisí na konkrétním programovacím jazyce.

4.1 Předpočítání si výsledků do paměti

Nejrychlejší program řešící zadaný problém je ten, který nic nepočítá a rovnou vypíše správný výsledek. Zdá se vám to nemožné? Výsledky si můžeme spočítat dopředu a uložit si je do tabulky. Tabulku můžeme vložit buď přímo do zdrojového

[1] Pravidlo 80 na 20. Toto pravidlo často používají ekonomové a manažeři při svých rozhodnutích. Empiricky je ověřeno, že zhruba 80% zákazníků přinese 20% zisku a 20% zákazníků přinese 80% zisku. Na které zákazníky se máme zaměřit? Zkuste se zamyslet nad tím, jak toto pravidlo může využít programátor.

kódu programu (jako statické inicializované pole)[2] nebo ji můžeme na začátku programu načíst ze souboru. Další možností je, že tabulku vygenerujeme po spuštění programu a oželíme krátké zdržení.

Nalezení výsledku v tabulce a jeho vypsání nám často zabere jen konstantní čas, případně čas úměrný velikosti odpovědi. Ovšem má to jednu mouchu. Zatím jsme tiše předpokládali, že výsledky půjdou poskládat do tabulky a že tabulka bude rozumně velká. To nemusí být u každé úlohy splněno. Ne každá úloha je pro předpočítání výsledků vhodná. Například řešení úlohy, která na vstupu dostane velký graf, se bude do tabulky ukládat špatně. Na druhou stranu úloha, která dostane číslo $n \in \{1, \ldots, 1000\}$ a má spočítat $f(n)$ pro nějakou složitou funkci f, vhodná je. Předpočítané hodnoty $f(n)$ si hravě uložíme do pole velikosti 1000.

Příklad: (Hašování do nafukovacího pole) – viz. nafukovací pole v kapitole 2.4 o amortizované časová složitosti. Je spousta možností, jak zvolit hašovací funkci. Běžně se používá funkce, kterou počítáme x modulo prvočíslo p. Při nafukování pole na velikost $2n$ bychom potřebovali znát největší prvočíslo menší nebo rovno $2n$. Jak takové prvočíslo rychle najít? Otázka je, proč ho vůbec během nafukování pole hledat. Můžeme si přeci dopředu pro každou velikost pole najít vhodné prvočíslo a uložit si ho do tabulky.

4.2 Výpočet hodnoty na základě předchozí

Úloha: Dostaneme polynom $P(x) = ax^2+bx+c$, kde a, b, $c \in \mathbb{Z}$ jsou předem známé konstanty. Pro všechna $x \in \{1, 2, \ldots, n\}$ bychom chtěli spočítat hodnotu polynomu. Například proto, abychom si mohli nakreslit graf funkce $P(x)$.

Řešení: (Hörnerovo schéma) Pro každé x spočítáme hodnotu $P(x)$ Hörnerovým schématem. Hörnerovo schéma vychází z možnosti částečně povytýkat x. Dostaneme $P(x) = (((a)x + b)x + c)$. Uzávorkování nám dává návod, jak vyhodnotit polynom v bodě x. Budeme postupovat v cyklu, začneme od nejvnitřnější závorky. V každém kroku přenásobíme vnitřní závorku hodnotou x a přičteme k ní odpovídající konstantu.

Pro vyhodnocení polynomu ve všech $x \in \{1, \ldots, n\}$ budeme $2n$ krát násobit a $2n$ krát sčítat.

Řešení: (výpočet hodnoty na základě předchozí) Tentokrát zkusíme hodnotu $P(x + 1)$ spočítat na základě předchozí hodnoty $P(x)$. Pro vypočtení $P(1)$ potřebujeme jen dvakrát sčítat. $P(x + 1) = a(x + 1)^2 + b(x + 1) + c = P(x) + (2a)x + (a + b)$. Pro výpočet $P(x + 1)$ tedy úplně stačí, když k předchozí hodnotě $P(x)$ přičteme $(2a)x + (a + b)$. Označíme $Q(x) = (2a)x + (a + b)$. Zbývá ukázat, jak rychle spočítat hodnotu $Q(x)$. Spočteme ji stejným trikem. Pro výpočet $Q(1)$ nám stačí tři sčítání. Pro výpočet hodnoty $Q(x+1)$ na základě předchozí nám stačí jednou sčítat, protože $Q(x + 1) = Q(x) + 2a$.

```
P[1] := a + b + c; Q[1] := 3a + b
for i = 2 to n do
    P[i] := P[i − 1] + Q[i − 1]
    Q[i] := Q[i − 1] + 2a
```

Takto vypočítáme hodnotu $P(x)$ ve všech bodech $x \in \{1, \ldots, n\}$ pomocí pouhých $2n + 5$ sčítání.[3] Pokud by a, b, c byly opravdu předem známé konstanty a

[2] Pomocný program, který předpočítá výsledky do tabulky, může rovnou vypisovat zdrojový kód pro vložení tabulky.

[3] Na většině počítačů je instrukce pro násobení časově náročnější než instrukce pro sčítání. Takže je ve skutečnosti druhý výpočet ještě rychlejší, než se zdá.

ne proměnné, tak můžeme ušetřit i těch prvních 5 sčítání tím, že si $P(1)$, $Q(1)$ předpočítáme.

Dokážete úlohu zobecnit pro vyhodnocování polynomu stupně 3, stupně 4 nebo obecně pro polynomy stupně k?

4.3 Využití předchozích hodnot

Palindrom je slovo které se čte stejně zepředu i pozpátku. Příkladem palindromů jsou řetězce: „madam“, „mam“, „rotor“, „kuna nese nanuk“, „kobyla ma maly bok“ (po vynechání mezer).

Úloha: (Nejdelší palindrom) Na vstupu dostanete řětězec n znaků. Navrhněte algoritmus, který v řetězci co nejrychleji nalezne nejdelší palindrom.

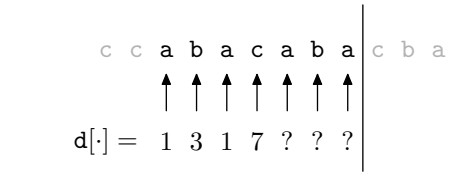
Řešení:(jednoduché kvadratické) Každý palindrom má svůj střed kolem kterého je symetrický (levá půlka palindromu je zrcadlovým obrazem pravé půlky). Středy palindromu jsou dvou druhů. Palindromy liché délky mají uprostřed písmeno, palindromy sudé délky mají uprostřed mezeru mezi písmeny.

Řešení, které nás hned napadne, je vyzkoušet všechny možné středy a z každého středu expandovat dosud nalezený palindrom do stran. Takové řešení má v nejhorším případě časovou složitost $\mathcal{O}(n^2)$. Příkladem řetězce, na kterém algoritmus dosáhneme kvadratické časové složitosti je řetězec se samých `a`, například `aaaaaaaaa`.

Řešení: (v lineárlním čase) Jak se dá předchozí algoritmus vylepšit, abychom dosáhli lineální časové složitosti? Základní myšlenka zůstane stejná, chceme pro všechny středy s vyplnit

$$\mathtt{d}[s] = \text{ délka nejdelšího palindromu se středem } s.$$

Ale jak to udělat? Podívejme se nejprve na malý příklad. Předpokládejme, že už jsme nalezli nejdelší palindrom se středem na pozici s. Říkejme mu aktuální master-palindrom. Na obrázku je vyznačen tučně, má délku 7 znaků a jeho střed je na pozici 6.



Jaké hodnoty vyplnit do $\mathtt{d}[7]$, $\mathtt{d}[8]$,...? Kdybychom tyto hodnoty hledali expanzí, tak se vrátíme zpátky ke kvadratické časové složitosti. Využijeme malý trik, kterým je zrcadlení. Protože je pravá půlka palindromu zrcadlovým obrazem levé půlky palindromu, platí $\mathtt{d}[s+i] = \mathtt{d}[s-i]$ pokud pravý okraj ozrcadleného palindromu nepřesáhne pravý okraj master-palindromu.

Na obrázku je pravý okraj master-palindromu vyznačen čárou. Ta zároveň odděluje oblast řetězce ze vstupu, která ještě nebyla prozkoumána. V příkladu na obrázku můžeme nastavit $\mathtt{d}[7] := 1$. Podobně $\mathtt{d}[8] := 3$, ale to ještě nemusí být délka nejdelšího palindromu se středem na pozici 8. Tento palindrom délky 3 se dotýká čáry označující neprozkoumanou oblast a proto možná půjde rozšířit.

Pro všechny středy, jejichž palindromy jsou celé uvnitř aktuálního master-palindromu, můžeme nastavit $\mathtt{d}[s+i] := \mathtt{d}[s-i]$. První střed z leva, jehož palindrom se dotýká nebo překračuje hranici master-palindromu, se stane středem S nového master-palindromu.

Nový master-palindrom nalezneme expanzí do stran. Nemusíme začínat úplně od začátku. Už víme, že $\mathtt{d}[S]$ znaků kolem středu S tvoří palindrom a proto stačí tento palindrom rozšiřovat do stran. První znak řetězce ze vstupu, na který „sáhneme“, bude ten napravo od posledního znaku původního master-palindromu (první znak za čárou).

Algoritmus začne s master-palindromem, jehož střed je na prvním znaku řetězce ze vstupu. Postupně hledá následující master-palindromy a za každý master-palindrom vyplní příslušný úsek pole $\mathtt{d}[\,\cdot\,]$. Na závěr už jen stačí najít maximální hodnotu v tomto poli.

Algoritmus má lineální časovou složitost, protože na každý znak řetězce ze vstupu sáhneme nejvýše dvakrát a protože každé políčko pole $\mathtt{d}[\,\cdot\,]$ vyplňujeme jen jednou.

Poznámka: Nezapomeňte, že palindromy mají dva druhy středů. Jedny leží na pozici písmenka a druhé mezi sousedními písmenky.

4.4 Přímé generování výsledků

Úloha: Množina M obsahuje pouze taková přirozená čísla, která nejsou dělitelná jiným prvočíslem než 2, 3 a 5. Vypište co nejrychleji prvních n nejmenších čísel množiny M.

Řešení: První řešení, které člověka napadne, je procházet postupně všechna přirozená čísla a testovat, zda nejsou dělitelná jiným prvočíslem než 2, 3 a 5. Nebudeme si vysvětlovat detaily tohoto řešení a raději si ukážeme daleko rychlejší řešení.

Řešení: Nejprve učiníme několik pozorování. Teprve v posledním odstavci si ukážeme, jak provést jednu iteraci algoritmu.

Množina M obsahuje pouze čísla tvaru $2^i 3^j 5^k$, pro všechny možné indexy $i, j, k \geq 0$. Dejme tomu, že už jsme vypsali prvních s čísel m_1, m_2, m_3, …, m_s množiny M. Vydělením čísla m_{s+1} jedním z čísel $2, 3, 5$ dostaneme menší číslo patřící do M, které už bylo vypsáno. Proto následující číslo m_{s+1} dostaneme tak, že jedno z předchozích čísel m_i pro $i \leq s$ vynásobíme buď 2 nebo 3 a nebo 5.

Protože $m_1 < m_2 < m_3 < \ldots$, tak i $2m_1 < 2m_2 < 2m_3 < \ldots$. Najdeme si takový index I, že $2m_i \leq m_s$ pro všechna $i < I$ a $2m_i > m_s$ pro všechna $i \geq I$. Podobně pro čísla tvaru $3m_j$ a $5m_k$ najdeme indexy J a K.

Následující číslo, které máme vypsat, je $m_{s+1} = \min\{\, 2m_I, 3m_J, 5m_K \,\}$. Až ho vypíšeme, tak stačí posunout příslušný index o 1. Tím nastavíme indexy I, J, K na správnou hodnotu. Pro výpis dalšího čísla můžeme postup zopakovat.

4.5 Předzpracování dat

Výpočet můžeme často zrychlit tím, že si vstupní data nejprve upravíme do vhodné podoby, případně si něco dalšího předpočítáme a teprve z těchto dat počítáme výstup. Celý výpočet tedy rozdělíme do dvou fází – předzpracování a vlastního výpočtu.

Úloha: (nejčastěji se vyskytující číslo) Dostanete pole obsahující n celých čísel a máte vypsat číslo, které se v poli vyskytuje nejčastěji.

Řešení: Hloupé řešení si pro každý prvek pole spočítá, kolikrát se v poli vyskytuje (pro každý prvek projde celé pole), a vybere ten s největším počtem výskytů. Jeho časová složitost je $\mathcal{O}(n^2)$.

Řešení: Určitě vás napadne nejjednodušší možné předzpracování a to je setřídění dat. Setřídit pole umíme v čase $\mathcal{O}(n \log n)$. Potom bude k nalezení nejčastěji se

vyskytujícího čísla stačit jen jeden průchod pole. Ten proběhne v čase $\mathcal{O}(n)$. Celkem toto řešení zabere čas $\mathcal{O}(n \log n)$.

Úloha: (největší jedničková podmatice) Matice A velikosti $n \times m$ obsahuje nuly a jedničky. Podmatice je souvislý obdélníkový výřez z matice A (určený levým horním a pravým dolním rohem). Jedničková podmatice je podmatice obsahující samé jedničky. Najděte největší jedničkovou podmatici matice A.

$$A = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{pmatrix}$$

Na obrázku je matice A velikosti 6×9. Největší jedničková podmatice obsahuje 9 jedniček a její levý horní roh leží na pozici $(3, 4)$.[4]

Řešení: Nejjednodušší řešení vyzkouší všechny možné polohy levého horního i pravého dolního rohu a pro jimi určenou podmatici zkontroluje, jestli se skládá ze samých jedniček. Této kontrole budeme jednoduše říkat testování podmatice.

Všech možných poloh levého horního rohu je mn. Stejně tak i poloh pravého dolního rohu. Otestování, jestli se jimi určená podmatice skládá ze samých jedniček vyžaduje průchod celé podmatice a trvá v nejhorším případě čas $\mathcal{O}(mn)$. Celkem je časová složitost tohoto řešení $\mathcal{O}(m^3 n^3)$.

Řešení: Zamysleme se nad tím, co je na výše uvedeném řešení neefektivní. Často testujeme podmatice, které se překrývají. Testování průniku obou podmatic by stačilo dělat jen jednou. Chceme-li se vyhnout opakovanému testování některých podmatic, tak si vhodně předzpracujeme vstupní data. Ke každé jedničce si spočítáme, kolik jedniček leží v souvislé řadě pod ní. Tento počet k jedničce přičteme. Získané údaje si můžeme uložit přímo do matice A (nepotřebujeme pomocnou datovou strukturu), protože nuly zůstanou tam, kde byly a místo jedniček budeme mít v matici uložena nenulová čísla.

$$\text{Z matice } A = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \text{ dostaneme matici } B = \begin{pmatrix} 2 & 0 & 4 & 3 \\ 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}.$$

Toto předzpracování nám zabere čas $\mathcal{O}(mn)$. Stačí každým sloupcem projít jednou od zdola nahoru, nuly ponechat beze změny a ke každé jedničce přičíst hodnotu prvku ležícího pod ní.

Jak hledat největší jedničkovou podmatici? Opět vyzkoušíme všechny polohy levého horního rohu (mn možností). Pro každý levý horní roh budeme zkoumat všechny možné polohy pravého horního rohu (až m možností). Pravý horní roh leží ve stejném řádku jako levý horní roh a mezi oběma rohy nesmí ležet žádná nula, jinak podmatice nebude jedničková.

Když už budeme znát levý i pravý horní roh podmatice, jak zjistíme velikost největší podmatice s těmito rohy? K tomu využijeme předvýpočtu. Každé číslo v řádku mezi levým a pravým horním rohem určuje velikost souvislého úseku jedniček ležícího ve sloupci pod ním (včetně jeho pozice). Takže stačí vzít minimum z těchto čísel. Velikost největší jedničkové podmatice určené horními rohy bude součin tohoto minima a vzdálenosti mezi horními rohy.

[4] Pozor, pozice se udává jako (řádek, sloupec) – jak už je u matic zvykem. Je to naopak než souřadnice (x, y).

Nyní shrneme, jak bude probíhat celý výpočet. Vyzkoušíme všechny možné pozice levého horního rohu. Pro každý levý horní roh stačí postupovat vpravo, dokud nenarazíme na nulu nebo na pravý okraj matice A. V průběhu postupu zkoušíme polohy pravého horního rohu a počítáme celkové minimum na základě předchozí hodnoty.

Vlastní hledání jedničkové podmatice bude trvat čas $mn \cdot \mathcal{O}(m) = \mathcal{O}(m^2 n)$. Dohromady s předzpracováním toto řešení vyžaduje čas $\mathcal{O}(m^2 n)$.

Řešení: Předchozí řešení můžeme ještě zrychlit. Provedeme ještě jedno předzpracování a spočítáme si i počet jedniček ležících nad každou pozicí. Kromě matice $B = (b_{ij})_n^m$, kde b_{ij} je délka souvislého úseku jedniček začínajícího na ij a ležícího pod pozicí ij, si vytvoříme matici $C = (c_{ij})_n^m$, kde c_{ij} je délka souvislého úseku jedniček začínajícího na ij a ležícího nad pozicí ij. Obě předzpracování stihneme provést v čase $\mathcal{O}(mn)$.

Klíčovým pojmem pro celé řešení je význačná pozice. Pozice ij je *význačná*, pokud za prvé $a_{ij} = 1$ a za druhé buď $a_{i,j-1} = 0$ nebo $a_{i,j-1}$ leží mimo matici A. Jinými slovy na význačné pozici leží 1 a vlevo vedle ní je buď 0 a nebo je pozice ij na levém okraji matice A.

Každá maximální jedničková podmatice nejde rozšířit o sloupec doleva, protože se vedle její levé hranice nachází 0 a nebo už tam končí matice A. To ale neznamená nic jiného, než že levý sloupec každé maximální jedničkové podmatice obsahuje význačnou pozici.

Jak bude probíhat hledání největší jedničkové podmatice? Pro každou význačnou pozici prozkoumáme všechny jedničkové podmatice, které ji obsahují ve svém levém sloupci. Provedeme to následovně. Začneme ve význačné pozici ij. Postupně budeme procházet doprava, dokud nenarazíme na nulu nebo pravý okraj matice A. V každém kroku, tedy na pozici ik, $k \geq j$, zjistíme velikost největší jedničkové podmatice obsahující obě pozice ij, ik. To provedeme stejně jako v předchozím řešení. Najdeme $b = \min\{b_{i,j}, b_{i,j+1}, \ldots, b_{i,k-1}, b_{i,k}\}$ a $c = \min\{c_{i,j}, c_{i,j+1}, \ldots, c_{i,k-1}, c_{i,k}\}$. Velikost největší takové jedničkové podmatice bude $(b+c-1)\cdot(k-j+1)$. Všechny pozice, na které jsme při postupu vpravo z význačné pozice vstoupili, nemohou být význačné, protože vlevo vedle nich leží 1.

Když si vše dáme dohromady, tak stačí matici A procházet po řádcích zleva doprava. V každém kroku jsme v jednom ze 3 stavů. Buď stojíme na nule, nebo na význačné pozici a nebo rozšiřujeme předchozí význačnou pozici doprava. V každém kroku děláme jen konstantně mnoho práce. Proto bude nalezení největší jedničkové podmatice trvat jen čas $\mathcal{O}(mn) + \mathcal{O}(mn)$. To je obdivuhodné zrychlení, ne?

4.6 Odstranění rekurze

Nejprve připomeňme základní znalost o fungování rekurze a předávání parametrů. Rekurze je realizována zásobníkem.[5] Všechny rekurzivně volané funkce se ukládají na zásobník. Na zásobníku se pro každou volanou funkci vytvoří záznam, který obsahuje informace o volané funkci a předané parametry. Po jejím skončení se tento záznam odebere ze zásobníku a program se podle předchozího záznamu (ten co je na vrcholu zásobníku) vrátí k předchozí funkci, a to do místa za rekurzivní volání právě proběhlé funkce.

Všechny parametry volané funkce se kopírují na zásobník a odtamtud je volaná funkce teprve využívá. Říká se tomu *předávání parametrů hodnotou.* Pokud bychom chtěli volané funkci předat pole, tak se nejprve celé pole zkopíruje na zásobník a

[5] Tak zvaný `STACK`. Většinou leží na začátku paměťového prostoru programu. Z druhé strany proti němu roste `HEAP` – paměť která je přidělována dynamicky alokovaným proměnným.

teprve pak s ním začne funkce pracovat. Abychom se vyhnuli zbytečnému kopírování, tak můžeme použít *předávání parametrů odkazem.* To funguje tak, že funkci předáme pouze ukazatel na předávané pole (adresu paměti, kde pole bydlí). Místo kopírovaní celého pole tedy stačí uložit na zásobník jeden ukazatel.

V některých případech můžeme rekurzi odstranit tím, že ji nahradíme jednoduchým cyklem.

Například funkci faktoriál naprogramovanou pomocí rekurze nahradíme for-cyklem. Ušetříme čas za volání funkce včetně předávání parametrů. Druhou výhodou je paměťová úspora. Zásobník, přes který se rekurzivní volání realizuje, zabere paměť $\mathcal{O}(n)$. Má to ještě jednu výhodu. Pro cyklus může překladač uplatňovat optimalizace (predikce podmínky), kdežto pro rekurzivní volání ne.

```
Faktorial(n):
   if n > 1 then
      return(n·Faktorial(n − 1))
   else
      return 1
```

```
Faktorial(n):
   fakt := 1
   for i := 2 to n do
      fakt := fakt · i
   return fakt
```

Ne vždy můžeme rekurzi nahradit cyklem. Co ale funguje vždy je nahrazení rekurze vlastním zásobníkem. Jak jsme si už vysvětlili, rekurze je už sama o sobě realizována zásobníkem. Na ten se ale ukládají úplně všechny funkce, které program rekurzivně volá. Každá rekurzivně volaná funkce má jiný počet různě velkých parametrů. Proto se na tento zásobník ukládá více informací, než je v některých případech potřeba. Z těchto a ještě i dalších důvodů můžeme realizací vlastního zásobníku něco ušetřit.[6]

Podobného efektu docílíme i správnou volbou programovacího jazyka, nepoužíváním věcí, které nejsou potřeba (například objektů).[7]

4.7 Odstranění opakujících se výpočtů

Fibonacciho číslo F_n je určeno rekurencí následovně: $F_0 = 0$, $F_1 = 1$ a $F_n = F_{n-1} + F_{n-2}$ pro $n \geq 2$. Fibonacciho čísla jsou 0, 1, 1, 2, 3, 5, 8, ... Pokud bychom naprogramovali funkci vracející n-té Fibonacciho číslo pomocí rekurze[8], tak přepsáním do for-cyklu dosáhneme podstatného zrychlení, které je způsobeno odstraněním opakujících se výpočtů.

```
Fib(n):
   if n > 1 then
      return(Fib(n − 1)+Fib(n − 2))
   else
      return n
```

```
Fib(n):
   F[0] := 0
   F[1] := 1
   for i := 2 to n do
      F[i] := F[i − 1] + F[i − 2]
   return F[n]
```

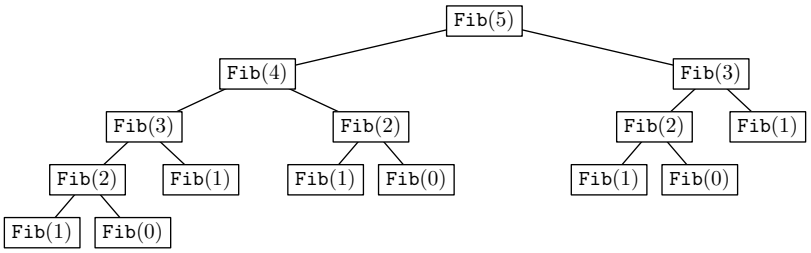
Na následujícím obrázku je strom větvení rekurzivního řešení. Zkuste si podle něj počítat.[9]

[6] Zkuste si například změřit, jak dlouho poběží quicksort naprogramovaný pomocí rekurze a jak dlouho poběží quicksort realizovaný pomocí zásobníku.

[7] To byste se divili, jak často studenti programují i triviální věci v objektech. Jakoby nic jiného neuměli. Napíší dvakrát tolik řádek zdrojového kódu, který je navíc pomalejší.

[8] To je typický odstrašující příklad.

[9] Kolega Kryl o efektivnosti algoritmů říká: „Zkuste si podle toho algoritmu počítat. A to doslova. Nevynechejte jediný příkaz. Pokud se při tom poblijete, tak to není efektivní.“.



Z obrázku vidíme, že jsme F_5 počítali jednou, F_4 také jednou, F_3 dvakrát, F_2 třikrát a F_1 pětkrát. Připomínají vám tyto čísla něco? Od toho pozorování už není daleko k tomu, abychom ukázali, že rekurzivní řešení má časovou složitost alespoň $\Omega(F_n)$. O Fibonacciho číslech je známo, že $F_n \approx \varphi^n$, kde $\varphi = \frac{1+\sqrt{5}}{2}$. Není těžké ukázat, že rekurzivní řešení má časovou složitost $\Theta(\varphi^n)$. O proti němu má nerekurzivní řešení jen lineální časovou složitost.[10] Dosáli jsme tedy exponenciálního zrychlení. A světe div se, nejlepší řešení s časovou složitostí $\mathcal{O}(\log n)$ je ještě exponenciálně krát rychlejší! (viz. cvičení).

Odstranění opakujících se výpočtů tím, že si je uložíme do tabulky, je jedna ze základních myšlenek *dynamického programování* (viz kapitola **??**).

4.8 Optimalizace pro hardware a operační systém

Fungování hardwaru je jedna velká magie.

Tuto optimalizaci nejčastěji provádíme tím, že optimalizujeme zdrojový kód. Příkazy a možnosti překladače uvedené v této sekci fungují například překladači `gcc`.

Často bereme počítač jen jako černou skříňku, která dobře počítá, a nezajímáme se o to, jak funguje vevnitř. Pokud ale chceme psát optimální kód, tak se neobejdeme bez hlubších znalosti chování hardwaru a fungování operačního systému. Jinak se nám může stát, že budeme naší černé skříňce házet klacky pod nohy a ona bude muset uvnitř pracovat tím nejsložitějším a taky nejpomalejším způsobem.

Nejprve bychom si měli rozmyslet, jestli je optimalizace kódu dané části programu opravdu nutná. Optimalizace totiž často svádí programátory k takzvaným „prasárnám“, které dokonale znečitelňují zdrojový kód. Nečitelný kód může pěkně potrápit další čtenáře, nebo po pár týdnech i nás samotné. Navíc se zvyšuje riziko, že někde uděláme chybu.

Optimalizace zdrojového kódu má smysl, pokud je daný kus kódu „úzkým hrdlem (bottle neckem)“ celého programu a pokud zvládneme kód optimalizovat lépe než překladač.[11] Přiznejme si, že dnešní překladače jsou na optimalizace profíci. Je těžké být lepší. Optimalizace kódu se většinou píší přímo v Assembleru. Často dosahují zrychlení tím, že využijí i specializované instrukce procesoru/grafického procesoru, které se běžně nepoužívají (případně počítají paralelně – buď na nezávislých obvodech téhož procesoru a nebo klidně i na více procesorech, CPU, GPU – graphical processing unit, ...).

[10] Poznamenejme, že u nerekurzivního řešení můžeme ještě snížit paměťovou složitost, protože si stačí místo celého pole pamatovat poslední dvě hodnoty.

[11] Například překladač `gcc` už dělá řadu základních optimalizací sám od sebe. Úroveň optimalizací můžete nastavit pomocí parametru na vstupu.

4.8.1 Jak to funguje uvnitř počítače?

Jak to zhruba funguje uvnitř černé skříňky? Procesor funguje na určité frekvenci. Frekvence procesoru je počet taktů za vteřinu. Každý procesor má svoji instrukční sadu. To jsou příkazy (instrukce), které jsou „zadrátované“ do elektrických obvodů. Každá instrukce trvá jiný počet taktů. Některé instrukce trvají jeden takt, jiné třeba až 150 taktů. Aby to nebylo jednoduché, tak má každý typ procesoru jinou instrukční sadu. Některé instrukce mohou být společné pro více typů procesorů, ale pro změnu mohou trvat jiný počet taktů (a to i výrazně).

Procesor potřebuje komunikovat s pamětí. Rychlá paměť je drahá a proto je paměť rozdělena do hierarchie podle klesající rychlosti, ale rostoucí velikosti: registry procesoru, cache procesoru, rozšířená paměť,[12] paměť na pevném disku. První dvě paměti jsou pro běžného programátora skryté, ale jsou výrazně rychlejší než rozšířená paměť.[13] Rozšířená paměť je zase výrazně rychlejší než pevný disk.

Procesor pracuje pouze s registry a s cache. Pokud potřebuje proměnnou z rozšířené paměti, tak si ji nejprve nechá nahrát do cache (v cache se vytvoří kopie) a tam s ní pracuje (s rychlejším přístupem). Často se stane, že s proměnnou pracuje více než jednou. Procesor pracuje pouze s lokální kopií v cache. Hodnota odpovídající proměnné v rozšířené paměti se zatím neaktualizuje. Aktualizace rozšířené paměti proběhne až v momentě, kdy chceme její lokální kopii v cache zrušit a uvolnit.

Cache často funguje na principu LRU (Least Recently Used). To znamená, že pokud už je cache plná a další proměnná se do ní nevejde, tak poslední dobou nejméně používaná proměnná uvolní místo nově příchozí proměnné. Při vymazání proměnné z cache se přepíše její aktuální hodnota zpátky do externí paměti.[14] [15]

Překladač sám provádí určité optimalizace kódu. Ovšem nic není dokonalé. Překladač se snaží pochopit strukturu kódu a když najde něco, co umí vylepšit, tak to vylepší. Stále se ale vyplatí optimalizovat si časově náročné části sám. Přeci jen toho o fungování algoritmu víme více než překladač, který to musí rozpoznat či odhadnout ze zdrojového kódu. Na druhou stranu použitím vlastních optimalizací můžeme zablokovat optimalizace, které mohl provést překladač (zneprůhlednili jsme mu tím kód) a ve výsledku můžeme dostat pomalejší kód.

To je vše, co si k fungování černé skříňky řekneme. Bližší zájemce odkazuji na literaturu o operačních systémech, hardwaru a překladačích.

Jestli ale nevíte, jak to na vašem počítači s vaším překladačem vašeho oblíbeného programovacího jazyka opravdu funguje, tak si s tím hrajte. Experimentujte. Napište si vlastní prográmek, ve kterém si změříte, jak dlouho které příkazy trvají. (Dobrý programátor by měl vědět, jak dlouho trvá sčítání oproti násobení. Jak pomalé je čtení z disku, apod.)

4.8.2 Zásady pro psaní efektivního kódu

Ačkoliv je situace složitá, je pár zásad, kterých se můžeme držet, abychom psali efektivní kód. V následujícím textu používáme slovo „drahý“ ve významu časově náročný.

- **Přístup do paměti je drahý.** Měli bychom se snažit vejít do cache. Případně bychom měli nejprve dokončit práci s proměnnými, které už jsou v cache, a pak se teprve vrhnout na nová data. Například pokud pracujeme s hodně

[12] Rozšířené paměti běžně říkáme „RAM“.

[13] Třeba až desetkrát.

[14] Při optimalizaci kódu můžeme speciálními příkazy určit, které proměnné se mají/nemají ukládat do cache.

[15] Pokud píšeme kód v assembleru, tak můžeme určovat přiřazení registrů proměnným. Běžně to za nás udělá překladač. A musíme uznat, že to dělá dobře.

dlouhým polem, tak je rychlejší dělat více práce na malých kusech, než vícekrát procházet celé pole.

Varování: V předchozích doporučeních jsme uváděli slova jako velký, malý apod. Ta správná velikost není univerzální, ale je na každém počítači jiná. Proto je nejlepší experimentovat s různými hodnotami parametrů a vybrat tu optimální. Další připomínkou je, že některá zlepšení se vyplatí až na hodně velkých datech.

Pokud potřebujeme vynulovat či překopírovat velký kus paměti, tak je lepší použít funkce k tomu určené (`memset`, `memcopy`).

Pokud se vše nevejde do cache, tak může být rychlejší, když si jednoduché věci dopočítáme znova, než když je hledáme v rozšířené paměti.

- **Zapisování a čtení ze souboru je drahé.** Připomeňme, že vypisování na obrazovku se chová stejně jako zápis do souboru a tudíž, je také celkem drahé. Hodně pomůže zavedení bufferů. Potom stačí zapisovat do souboru méně krát a po větších kusech. To vede k výraznému zrychlení. Také můžeme využít možnosti, namapovat si kus souboru do paměti, a s ním teprve pracovat.

- **Podmínky jsou drahé.** Aby procesor fungoval rychleji, tak si některé věci předpočítává. Například ví, že o pár instrukcí dál bude potřebovat určitou proměnnou, tak si ji už dopředu začne nahrávat do cache.[16] Pokud je ale následující nezpracovaná instrukce podmínka (větvení programu), tak se dopředu neví, jak dopadne a tudíž nevíme, co si předpočítat. Procesor se často připravuje na obě varianty a po vyhodnocení podmínky nepotřebnou variantu zahodí. Tím promarní část svého času.

  Příklad: Máme proměnnou a obsahující buď 0 nebo 1. Pokud potřebujeme změnit její hodnotu na opačnou, tak se dá podmínka s kódem nalevo přepsat na kód napravo.[17]

  ```
  if a = 1 then a := 0          a := 1 − a
  else a := 1
  ```

- Registry procesoru jsou dnes 64-bitové, proto je lepší zpracovávat data po 64 bitech a ne po menších dílech. Toho se dá využít například při zpracovávání řetězců, obrázků a dalších médií.

  Z podobných důvodů se paměť zarovnává na 64 bitů. To znamená, že první bit proměnné začíná na bitu v paměti, který je dělitelný 64. Pro jednoduchost si můžeme představit, že paměť funguje jako pole 64-bitových položek. Pokud bychom chtěli přistoupit k nezarovnané 64-bitové proměnné, tak budeme muset přečíst dvě políčka pole a z nich si vytáhnout potřebné bity.

- **Některé aritmetické operace jsou drahé.** Například dělení. Dělení mocninou dvojky, se dá obejít bitovým posunem doprava (bitová operace shift doprava). Dělení známou konstantou už ale stejným způsobem optimalizuje překladač. Problém nastává, když dělíme dopředu neznámou proměnou. To se při kompilaci programu optimalizovat nedá.

  Celkově je dobré se dělení vyhnout. Například místo výpočtu hashovací funkce modulo prvočíslo, kde se prvočíslo spočítá až za běhu programu a uloží do proměnné, si můžeme dopředu zjistit všechna prvočísla, které budeme používat. Při výpočtu hashovací funkce použijeme `switch`, kterým se podle prvočísla

[16] Leckdy natažení proměnné do cache trvá déle než samotný výpočet. Ovšem záleží na tom, co s proměnnou počítáme. Některé zběsilé výpočty (instrukce) trvají mnohem déle než natahování proměnné do cache.

[17] V takto jednoduchých případech to za nás udělá optimalizátor překladače.

rozskočíme na optimalizovanou rutinu modulení (modulo konstanta – optimalizace při dělení konstantou už provede překladač).

- K zajímavým trikům patří **použití zarážky.** Například když procházíme pole velikosti N a hledáme hodnotu x, tak běžně v podmínce cyklu kontrolujeme dvě věci: jestli index aktuálního políčka nepřekročil N a jestli není hodnota aktuálního políčka rovná x. Hledání můžeme zrychlit tím, že si do pole na pozici $N+1$ uložíme x, které bude fungovat jako zarážka. Potom máme jistotu, že x vždy najdeme, a můžeme podmínku v cyklu zjednodušit na test, jestli není hodnota aktuálního políčka rovná x. Až x najdeme, tak se podíváme, na kterém indexu leží. Podle toho zjistíme, jestli jsme x našli a nebo jen neúspěšně došli až na konec pole.

- Mezi **další triky** patří použití `inline` funkcí, nahrazení jednoduchých funkcí makrem (např. pro výpočet minima, maxima, absolutní hodnoty apod.)

4.9 Spousta dalších možností

Moto: „Vše co se učíme, se učíme tím, že to děláme.“

Řešte příklady z KSP (korespondenční seminář z programování), programátorských olympiád či soutěží ACM programming contest a učte se tím nové finty a triky. Jejich zadání najdete na webu. Můžete ho řešit i mimo soutěž, prostě jen pro radost. U prvních dvou najdete na webu i vzorové řešení.

4.10 Příklady

1. (Vyhodnocování polynomu) V sekci o výpočtu hodnoty na základě předchozí jsme si ukázali, jak rychle spočítat hodnoty polynomu $P(x) = ax^2 + bx + c$ ve všech bodech $x \in \{1, \ldots, n\}$. Potřebovali jsme je znát například proto, abychom si nakreslili graf funkce. Ale co kdybychom si chtěli nakreslit přesnější graf funkce $P(x)$? Co kdybychom chtěli znát hodnotu $P(x)$ ve všech bodech $0, 0.1, 0.2, 0.3, \ldots, n$? Jak to co nejrychleji spočítat?

2. (Výpočet hodnoty na základě předchozí) Je dána posloupnost celých čísel. Délka posloupnosti není známa. Posloupnost může být i tak dlouhá, že se nevejde do paměti, ale jen na pevný disk.

   (a) (Úsek posloupnosti s největším součtem) Nalezněte v posloupnosti souvislý úsek s největším součtem. Výstupem algoritmu bude jen hodnota největšího součtu. Například pro posloupnost 1 3 −1 1 −5 8 2 −3 4 −8 bude výsledkem 11 (to je délka úseku 8 2 −3 4).

   (b) (Nejdelší hladký úsek posloupnosti) Určete délku nejdelšího souvislého úseku, v němž se libovolná dvě čísla liší pouze o 1. Například pro posloupnost 5 7 6 7 7 8 8 7 9 9 9 bude výsledkem 5 (to je délka úseku 7 7 8 8 7).

   (c) (Nejdelší D-hladký úsek posloupnosti) Určete délku nejdelšího souvislého úseku, v němž se libovolná dvě čísla liší pouze o D.

   Nápověda: existuje řešení s časovou složitostí $\mathcal{O}(n)$.

3. (Přímé generování výsledku) Množina Q obsahuje pouze taková přirozená čísla, která nejsou dělitelná ani jedním z prvočísel 2, 3 a 5.[18] Vypište co nejrychleji n-té nejmenší číslo množiny Q.

   (a) Zkusme si to nejprve zjednodušit. Co kdybychom chtěli vypsat n-té nejmenší přirozené číslo, které není dělitelné 2? Je to moc jednoduché? Tak jak co nejrychleji vypsat n-té nejmenší přirozené číslo, které není dělitelné 2 ani 3?

   (b) Vyřešte úlohu pro zadaná prvočísla 2, 3 a 5. Umíte odpovědět v čase $\mathcal{O}(1)$? Pokud ano, tak ještě dokažte, že je odpověď správně. To je, že jste žádné číslo množiny Q nevynechali.

   (c) Zobecněte řešení úlohy pro k různých prvočísel.

4. (Rozklad čísla na součet třetích mocnin) Rozložte číslo $n \in \mathbb{N}$ na součet dvou třetích mocnin přirozených čísel. Jinými slovy najděte $a, b \in \mathbb{N}$ splňující $n = a^3 + b^3$. Vypište všechna řešení.

   (a) Někdo by mohl najít následující řešení. Procházíme všechna možná $a \in \{0, 1, \ldots, \lfloor \sqrt[3]{n} \rfloor\}$ a pro každé a spočteme $\sqrt[3]{n - a^3}$. Pokud bude výsledek celočíselný, tak jsme nalezli řešení. Toto řešení je samozřejmě správně a má časovou složitost $\mathcal{O}(\sqrt[3]{n})$.

   Zkuste ale vymyslet řešení, ve kterém budeme počítat pouze s celočíselnými proměnnými a vystačíme si se sčítáním a násobením. Dá se vymyslet řešení s časovou složitostí $\mathcal{O}(\sqrt[3]{n})$.

   (b) Obě řešení naprogramuje a porovnejte, jak rychle počítají pro konkrétní n. Vyvoďte z toho závěr, jestli je opravdu výhodnější počítat v celočíselných proměnných, nebo jestli to vyjde stejně, jako když počítáme v plovoucí čárce a navíc počítáme funkce jako je odmocnina.

5. (Kružnice) Na kružnici je rozmístěno n bodů. Vzdálenost dvou bodů budeme měřit po obvodu kružnice. Vzdálenosti libovolných dvou bodů jsou celočíselné. Začneme v nejvyšším bodě a body si očíslujeme čísly 1 až n po směru hodinových ručiček. Pro každé dva sousední body (v pořadí podle očíslování) dostanete jejich vzájemnou vzdálenost. Například pro kružnici s 5 body, můžete dostat vzdálenosti $d(1,2) = 1$, $d(2,3) = 4$, $d(3,4) = 2$, $d(4,5) = 5$, $d(5,1) = 2$.

   (a) Rozhodněte, jestli na kružnici existují dva body takové, že jejich spojnice prochází středem kružnice. Pokud ano, tak takovou dvojici bodů vypište. V příkladu kružnice s 5 body bychom mohli odpovědět $(1,4)$, ale také $(3,5)$, protože jejich spojnice prochází středem kružnice.

   (b) Rozhodněte, jestli na kružnici existují čtyři body takové, že tvoří čtverec. Pokud ano, tak takové 4 body vypište.

   Nápověda: Existuje řešení s časovou složitostí $\mathcal{O}(n)$.

6. (Fibonacciho čísla přes mocnění matic) Fibonacciho čísla jsou 0, 1, 1, 2, 3, 5, 8, ... Jsou definovány rekurencí $F_0 = 0$, $F_1 = 1$ a $F_n = F_{n-1} + F_{n-2}$ pro $n \geq 2$. Rovnost $F_1 = F_1$ a $F_2 := F_1 + F_0$ můžeme zapsat i pomocí matice:

[18] Pozor na odlišnost od ukázkové úlohy ze sekce o přímém generování výsledku. Tam jsme vyžadovali, aby číslo $x \in M$ bylo dělitelné pouze prvočísly 2, 3 a 5. Tady vyžadujeme, aby nebylo dělitelné ani jedním z prvočísel 2, 3 a 5. Není pravda, že $Q = \mathbb{N} \setminus M$, protože například číslo $21 = 7 \cdot 3$ nepatří ani do jedné množiny.

$$\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}.$$

Podobně

$$\begin{pmatrix} F_2 \\ F_3 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

až zobecněním dostaneme

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}.$$

Označme tuto matici tvaru 2×2 jako A. Pro výpočet n-tého Fibonacciho čísla tedy stačí vymyslet, jak se dá rychle spočítat n-tá mocnina matice A.

(a) Ukažte, že n-tá mocnina libovolné matice B velikosti 2×2 se dá spočítat v čase $\mathcal{O}(\log n)$.

Nápověda: Zamyslete se nad tím, jak byste počítali B^2, B^4, B^8.

(b) Jak rychle dokážete spočítat číslo X_n, které je určeno následující rekurencí: $X_0 = a$, $X_1 = b$, $X_2 = c$ a $X_n = dX_{n-1} + eX_{n-2} + fX_{n-3}$ pro $n \geq 3$. Čísla a, b, c, d, e, f jsou pevně dané konstanty.

(c) Fibonacciho čísla se dají spočítat i podle následujícího vzorce:

$$F_n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n$$

Ukažte, jak tento vzorec souvisí s vlastními čísly matice A.

Nebylo by rychlejší počítat Fibonacciho čísla přímo z tohoto vzorce než přes mocnění matic? Nebylo. Čísla ve vzorci jsou iracionální a jejich výpočet s dostatečnou přesností by byl stejně pracný jako mocnění matic. Vyzkoušejte si to naprogramovat.[19]

Nápověda: Jaká jsou vlastní čísla matice A?

Poznámka: Když si vzorec vyčíslíte, tak zjistíte, že F_n je přibližně $0,44 \cdot (1,61^n + (-0,61)^n)$. Proto můžeme F_n počítat jen jako $0,44 \cdot 1,61^n$ a výsledek zaokrouhlit na nejbližší celé číslo.

7. (Počítání prvních n prvočísel) Napište program, který vypíše na obrazovku prvních $N := 1000$ prvočísel. U každého přístupu odhadněte časovou složitost.

(a) Procházejte postupně čísla 1, 2, 3,... (procházení kandidátů) a aktuálně testované číslo k zkoušejte dělit všemi čísly 2,3,..., $\sqrt{k}$ (testování prvočíselnosti).

(b) Jak můžeme předchozí výpočet zrychlit? Ukládejte si nalezená prvočísla do paměti a při testování prvočíselnosti zkoušejte dělit testované číslo k pouze dosud nalezenými prvočísly.

(c) Jak to urychlit ještě více? Můžeme předem zamítnout některé kandidáty. Například všechna sudá čísla kromě 2 určitě nebudou prvočísly. Jak se dají rychle generovat kandidáti, kteří nejsou dělitelní 2 ani 3?

[19] V algebře se můžete dozvědět, že počítání s celými čísly, ke kterým přidáme iracionální číslo tvaru $\sqrt{a}$ můžeme nahradit počítáním s maticemi velikosti 2×2. Hovoříme o okruhu celých čísel a okruhu celých čísel rozšířeném o $\sqrt{a}$.

Pro odhad časové složitosti se vám bude hodit vědět, že počet prvočísel menších než n je $\pi(n) \approx n/\ln n$.

8. (Největší společná podmatice) Dostaneme dvě matice A a B obsahující přirozená čísla.

   (a) (největší společná podmatice ležící na stejné pozici) Předpokládejte, že obě matice mají stejnou velikost $n \times m$. Najděte největší obdélník určený levým horním a pravým dolním rohem takový, aby podmatice matic A a B určené tímto obdélníkem obsahovaly stejná čísla.

   (b) (největší společná podmatice ležící na libovolných pozicích) Tentokrát mohou mít matice A a B různé velikosti. Najděte rozměry obdélníka s největším možným obsahem takové, aby matice A, B obsahovaly podmatice těchto rozměrů, které jsou stejné.

   Rozmyslete si, o kolik složitější by bylo zároveň vypsat i polohy společných podmatic.

9. (Podmatice s největším součtem) Dostanete matici A velikosti $n \times m$ obsahující přirozená čísla. Najděte podmatici s největším součtem.

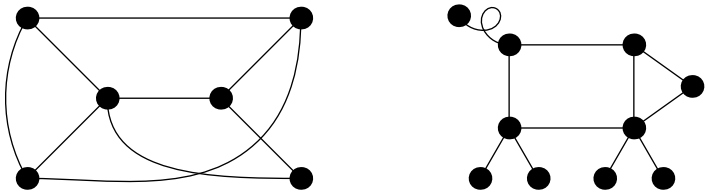
Kapitola 5

Grafy a stromy

Co je to graf? Grafem v teorii grafů myslíme něco jiného než je graf funkce nebo například sloupcový graf. *Graf* G je dvojice (V, E). Prvkům množiny V říkáme *vrcholy* a prvkům $E \subseteq \{\, \{u, v\} \mid u, v \in V \,\}$ *hrany*.

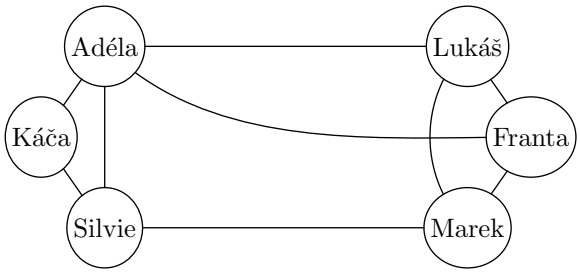
Hrana $e \in E$ je neuspořádaná dvojice vrcholů, tedy $e = \{u, v\}$ pro $u, v \in V$. Vrcholy u, v jsou *konce hrany* e. Protože $v \in e$, tak říkáme že v náleží do e, nebo použijeme cizí slovo a řekneme, že v je *incidentní* s e. Často existenci hrany vyjadřujeme slovy: „vrcholy u, v jsou spojeny hranou e“, „z u vede hrana do v“, „vrchol u sousedí s vrcholem v“. Vrcholy, se kterými je vrchol u spojen hranou, se nazývají *sousedi* vrcholu u. Množinu všech neuspořádaných dvojic vrcholů V budeme značit $\binom{V}{2}$.[1] Potom se dá napsat, že $E \subseteq \binom{V}{2}$.

Ačkoliv je graf abstraktní pojem, často ho kreslíme na papír. Vrcholy kreslíme jako „puntíky“ a hrany jako čáry, které je spojují.[2]



Grafem můžeme jednoduše zachytit celou řadu souvislostí mezi různými objekty:

Vztahy mezi lidmi. Vrcholy grafu jsou lidé, konkrétně $V =$ (Adéla, Franta, Káča, Lukáš, Marek, Silvie). Dva lidé jsou spolu spojeny hranou, pokud mezi sebou mají určitý vztah. Například pokud spolu kamarádí.

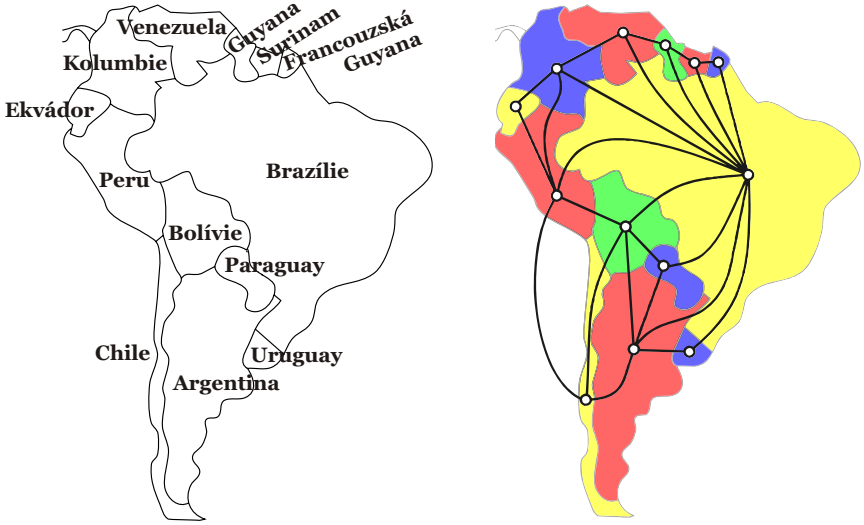


Barvení mapy. Zjednodušení struktury nám může pomoci lépe a přehledněji vyřešit požadovaný problém. Například chceme obarvit mapu Jižní Ameriky tak,

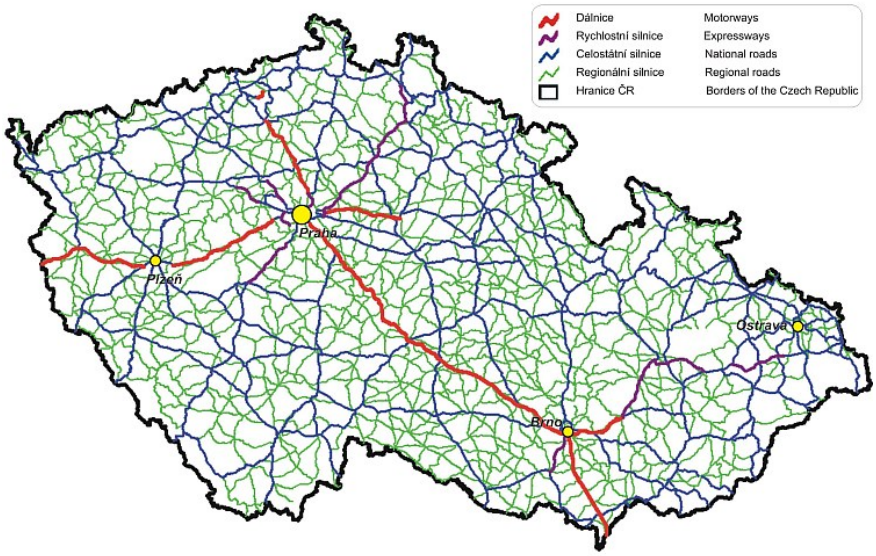
[1] Protože takových dvojic je přesně $\binom{|V|}{2}$.

[2] Nakreslení grafu je přesně definovaný pojem, o který se můžeme opřít v důkazech. To už ale patří k pokročilejší matematice.

aby sousední státy měly jinou barvu (jinak by na první pohled vypadaly jako 1 velký stát). Nejprve si vytvoříme pomocný graf. Do každého státu umístíme vrchol (například do hlavního města) a vrcholy dvou států spojíme hranou, pokud spolu sousedí. Teď stačí obarvit vrcholy grafu tak, aby vrcholy spojené hranou měly různou barvu.



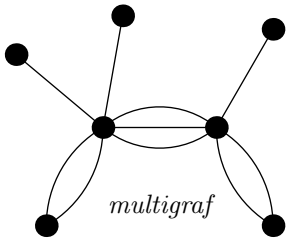
Silniční síť si také můžeme zjednodušeně zachytit grafem. Křižovatky znázorníme jako vrcholy, a silnice, které je spojují, jako hrany.



Každá silnice má svojí délku. Tuto délku můžeme připsat ke každé hraně. Dostaneme tím *ohodnocený graf*. Ohodnocený graf G je graf $G = (V, E)$, ke kterému přidáme funkci $h : E \to \mathbb{R}$, která každé hraně e přiřadí hodnotu $h(e)$.

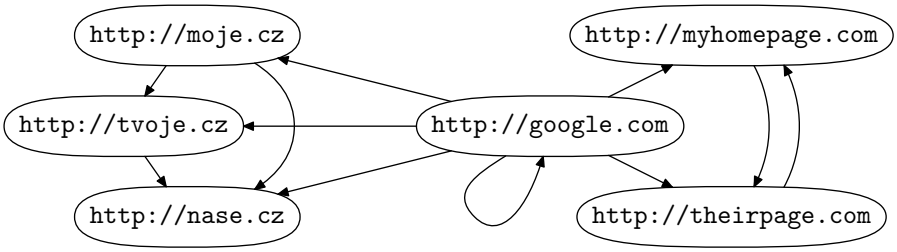
Teoreticky by se mohlo stát, že dvě křižovatky budou spojeny dvěmi různými silnicemi. Jinými slovy, mohlo by se stát, že dva vrcholy budou spojeny dvěmi různými hranami. Běžné grafy nám neumožňují takové věci zachytit. Museli bychom pojem grafu zobecnit tak, že každé hraně přiřadíme její *násobnost* $N : E \to \mathbb{N}$. Násobnost vyjadřuje, kolikrát je hrana v grafu zastoupena. Tím dostaneme *multigrafy*.

Jiné zavedení multigrafů pracuje se zobrazením, které každé hraně přiřadí dva vrcholy, které jsou její konce. To už je blíže tomu, jak multigrafy používá programátor.



V některých aplikacích se můžeme použití multigrafů (a násobných hran) vyhnout tím, že na jednu hranu umístíme fiktivní vrchol/křižovatku, který hranu rozdělí na dvě.

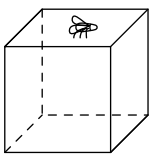
Web. Chtěli bychom si znázornit „strukturu“ webových stránek. Vrcholy grafu budou webové stránky. Dva vrcholy (webové stránky) spojíme hranou, pokud z jedné stránky vede odkaz do druhé. Jak je ale vidět, tento vztah není symetrický. Ze stránky A může vést odkaz na stránku B, ale naopak už to platit nemusí. Abychom lépe zachytili strukturu webu, tak z každé hrany uděláme šipku. Z vrcholu A povede šipka do vrcholu B, pokud ze stránky A vede odkaz na stránku B. Dostaneme tím *orientovaný graf.*

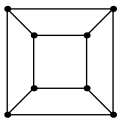


Orientovaný graf je graf, kde každé hraně udělíme orientaci. Na obrázku se dá říci, že z hran uděláme šipky. Formálně, *orientovaný graf* $G = (V, E)$, kde V jsou vrcholy a $E \subseteq V \times V$ jsou hrany. Hrany jsou orientované, v prvním vrcholu začínají a ve druhém končí. Proto $(u, v) \neq (v, u)$.

V orientovaných grafech se může stát, že existuje hrana $xy \in E$, ale i $yx \in E$. Dokonce se může stát, že v grafu bude hrana $xx \in E$. Takovým hranám říkáme *smyčky.*

Odkud pochází název hrana, vrchol? Pochází z mnohostěnů. Mnohostěny, například krychle, mají také vrcholy a hrany. Vrcholy jsou body a hrany jsou úsečky spojující dva vrcholy. Jak spolu souvisí mnohostěn a graf? Ke každému mnohostěnu můžeme najít graf popisující jeho strukturu. Představte si mouchu, která si sedne na stěnu velké skleněné krychle. Skrz skleněnou krychli moucha uvidí obrázek napravo. Ten odpovídá nakreslení grafu.





Značení používané této knize. Pokud mluvíme pouze o grafu, tak myslíme obyčejný neorientovaný graf. Když bychom chtěli mluvit o jiném grafu, tak to vždy zdůrazníme.

Protože budeme grafy používat hodně často a většinou budeme pracovat jen s jedním grafem G, tak budeme počet vrcholů grafu G označovat písmenem n a počet hran písmenem m. Také budeme zjednodušeně psát $uv \in E$ místo $\{u, v\} \in E$ a v orientovaných grafech $uv \in E$ místo $(u, v) \in E$.

5.1 Grafové pojmy

Abychom mohli s grafy pracovat a jednoduše popsat, co s grafem děláme, potřebujeme znát základní grafové pojmy.

- **Izomorfismus:** Izomorfismus není nic jiného než přejmenování vrcholů. Graf G *je izomorfní s grafem* H právě tehdy když existuje bijekce $f : V(G) \to V(H)$ taková, že $xy \in E(G) \iff f(x)f(y) \in E(H)$.

  Jsou výše nakreslené grafy izomorfní? Najděte bijekci.

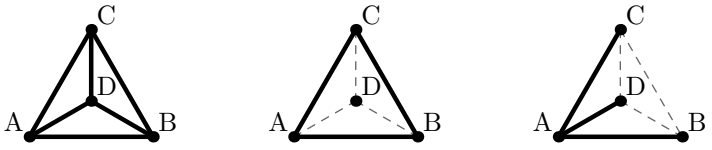
- **Doplněk grafu:** *Doplněk grafu* G (značíme $\overline{G}$) je graf $(V(G), \overline{E})$, kde $\overline{E} = \binom{V(G)}{2} \setminus E(G) = \{\, xy \,|\, xy \notin E(G) \,\}$. Jinými slovy, hranami doplňku G jsou „nehrany“ grafu G.

- **Stupeň vrcholu:** Stupeň vrcholu v v grafu G je počet jeho sousedů:

$$deg_G v := |\{vx \in E(G) \,:\, \text{pro} \forall\, x \in V\}|.$$

- **Podgraf grafu:** H je *podgraf* grafu G (značíme $H \subseteq G$) právě tehdy když H je graf a $V(H) \subseteq V(G)$, $E(H) \subseteq E(G)$. Jinými slovy, podgraf H vznikne z grafu G vymazáním některých hran a některých vrcholů včetně hran z nich vedoucích. Obráceně bychom mohli říci, že G je *nadgraf* grafu H. Místo „H je podgraf G“ někdy říkáme, že graf G *obsahuje* H.

- **Indukovaný podgraf:** $H := G[W]$ je *podgraf grafu* G *indukovaný množinou* $W \subseteq V$ právě tehdy když $V(H) = W \subseteq V(G)$ a $E(H) \subseteq E(G) \cap \binom{W}{2}$. Jinými slovy, graf $G[W]$ vznikne z G vymazáním vrcholů $V \setminus W$ včetně hran z nich vedoucích. Pokud nechceme množinu W vysloveně uvádět, tak řekneme, že H je indukovaný podgraf grafu G (indukovaný nějakou množinou, která se stane $V(H)$).



Na obrázku je graf G, jeho indukovaný podgraf $H_1 = G[\{A, B, C\}]$ a podgraf $H_2 \subseteq G$, který není indukovaným podgrafem.

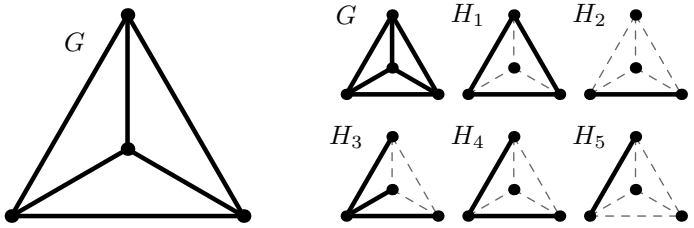
- **Jednoduché operace na grafu $G = (V, E)$:**
  1. přidání hrany: $G + e := (V, E \cup \{e\})$
  2. smazání hrany: $G - e := (V, E \setminus \{e\})$
  3. smazání vrcholu: $G - v := (V \setminus \{v\}, E \setminus \{vw \in E \,:\, \text{pro} \forall\, w \in V\})$

- **Hranově maximální graf:** Graf G je *hranově maximální* pro nějakou grafovou vlastnost A, pokud G splňuje vlastnost A a žádný z grafů $G + xy$, $xy \notin E(G)$, už ji nesplňuje.

  Příkladem grafové vlastnosti je „graf obsahuje nejvýše 5 hran“. Složitější vlastnosti naleznete v následující sekci s grafovou botanickou.

- **Maximální/minimální graf:** Graf G je *maximální/minimální* pro nějakou grafovou vlastnost A, pokud G splňuje vlastnost A a žádný nadgraf/podgraf už ji nesplňuje.

  V algoritmech často hledáme maximální/minimální graf s určitou vlastností. Jak ale grafy porovnáváme? Který je menší a který větší? Za uspořádání na grafech bereme relaci býti podgrafem. Relace býti podgrafem je relace částečného uspořádání.[3] Podívejme se příklad na obrázku. Napravo máme 6 podgrafů grafu G, který je nalevo (velký graf).



  Pro grafy G, H_1, H_2 platí $H_2 \subseteq H_1 \subseteq G$. Podobně $H_5 \subseteq H_4 \subseteq H_3$. To určuje uspořádání na podgrafech. Ne každé dva podgrafy jsou porovnatelné, například H_2 a H_5 (ani jeden není podgrafem druhého).

  Pozor, je potřeba rozlišovat dva pojmy – maximální a největší.[4] Prvek je maximální, právě tehdy když neexistuje žádný větší. Prvek je největší, právě tehdy když je větší než všechny ostatní. Podobně pro minimální a nejmenší prvek.

  Na obrázku je G největším podgrafem, protože obsahuje všechny ostatní jako podgraf. Podgrafy H_2 a H_5 jsou minimálními neprázdnými podgrafy, ale ani jeden z nich není nejmenší neprázdný podgraf. Nejmenším podgrafem je prázdný podgraf, který obsahuje všechny vrcholy, ale ani 1 hranu.

Jako základní učebnici teorie grafů můžeme doporučit Matouška, Nešetřila: Kapitoly z diskrétní matematiky [23]. Pro pokročilejší teorii anglickou učebnici Diestel: Graph Theory [11].

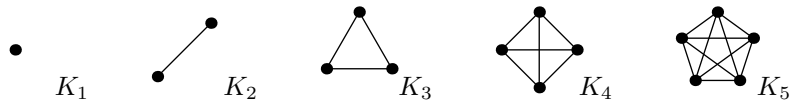
[3] Relace R je *částečné uspořádání,* pokud je reflexivní (xRx pro každé x), antisymetrická (xRy a $yRx \Rightarrow x = y$ pro každé x, y) a tranzitivní (xRy a $yRz \Rightarrow xRz$ pro každé x, y, z). Více například viz Matoušek, Nešetřil [23].

[4] Angličtina pro to má dva pojmy maximal a maximum.

5.2 Grafová botanická

Grafy, které splňují určitou vlastnost, mají svůj název. Mezi základní „druhy“ grafů patří:

- **Úplné grafy:**[5] Úplný graf obsahuje všechny možné hrany: $K_n = ([n], \binom{[n]}{2})$, kde $[n] = \{1, 2, \ldots, n\}$.



- **Cesty:** *Cesta* je neprázdný graf $P = (V, E)$, kde $V = \{x_0, x_1, \ldots, x_k\}$ a $E = \{x_0x_1, x_1x_2, x_2x_3, \ldots, x_{k-1}x_k\}$. Vrcholům x_0 a x_k říkáme *koncové vrcholy cesty.* Ostatní vrcholy jsou *vnitřní. Délka cesty* je počet hran na cestě.

- **Kružnice:** *Kružnice* je neprázdný graf $P = (V, E)$, kde $V = \{x_0, x_1, \ldots, x_k\}$ a $E = \{x_0x_1, x_1x_2, x_2x_3, \ldots, x_{k-1}x_k\} \cup \{x_kx_0\}$. *Délka kružnice* je její počet hran.

  Podgrafy grafu G, které jsou izomorfní výše uvedeným druhům, často bereme jakou součást grafu. Například cesta v grafu je podgraf izomorfní cestě.

- **Cesta v grafu:** Cesta v grafu G je podgraf grafu G, který je izomorfní cestě. Někdy za cestu P v G bereme posloupnost vrcholů $(x_0, x_1, \ldots, x_k)$ takovou, že $x_{i-1}x_i \in E(G)$ pro $i \in \{1, 2, \ldots, k\}$. Říkáme, že cesta P vede mezi vrcholy x_0 a x_k. Proto ji někdy značíme jako x_0Px_k. Díky tomu můžeme jednoduše označovat „podcesty“ (pro $0 \leq i \leq j \leq k$):

$$\begin{aligned} Px_i &:= (x_0 \ldots x_i) \\ x_iP &:= (x_i \ldots x_k) \\ x_iPx_j &:= (x_i \ldots x_j) \end{aligned}$$

  ale i některé operace, například zřetězení úseků xPy a yQz je cesta $xPyQz$.

- **Kružnice v grafu:** Kružnice v grafu G je podgraf grafu G, který je izomorfní kružnici. Někdy za kružnici C v G bereme posloupnost vrcholů $(x_0, x_1, \ldots, x_k)$ takovou, že $x_{i-1}x_i \in E(G)$ pro $i \in \{1, 2, \ldots, k\}$ a $x_0x_k \in E(G)$.

- **Hamiltonovské kružnice:** Kružnice je *hamiltovnoská* právě tehdy když obsahuje všech n vrcholů grafu.

- **Souvislost grafu:** Graf G je *souvislý,* pokud mezi každou dvojicí vrcholů $x, y \in V$ existuje cesta.

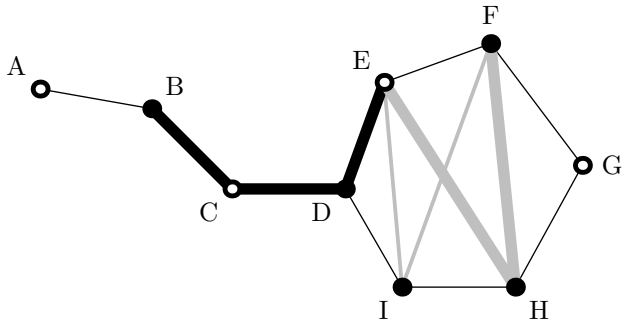
- **Klika v grafu:** *Klika* v grafu G je indukovaný podgraf grafu G, který je izomorfní úplnému grafu. Někdy za kliku bereme pouze podmnožinu vrcholů $Q \subseteq V(G)$ takovou, že každé dva vrcholy $x, y \in Q$ jsou spojeny hranou $xy \in E(G)$.

[5] Hovorově „úplňáky“.

- **Nezávislá množina v grafu:** Vrcholy grafu G, mezi kterými nevede žádná hrana, se nazývají *nezávislé. Nezávislá množina* $W \subseteq V(G)$ v grafu G je množina navzájem nezávislých vrcholů. Podgraf G indukovaný nezávislou množinou je doplňkem úplného grafu.
- **Stromy:** *Strom* T je souvislý graf bez kružnic. Pokud zakážeme pouze kružnice, ale nevyžadujeme souvislost, tak dostaneme graf skládající se z několika stromů. Takovému grafu se říká *les.* Vrcholy stromu stupně 1 se nazývají *listy.*

  O stromech koluje i několik grafových vtipů: Víte, proč pařez není strom? Protože obsahuje kružnice.[6] A víte, že souvislý les je strom?
- **Kostry:** *Kostra* grafu G je strom $T \subseteq G$ na všech n vrcholech.
- **Bipartitní graf:** Graf $G = (V, E)$ je *bipartitní,* právě tehdy když můžeme jeho vrcholy rozdělit do dvou množin V_1 a V_2 tak, že každá hrana vede mezi V_1 a V_2. $V = V_1 \cup V_2$. Množinám V_1 a V_2 se říká partity grafu.

Příklad: V následujícím grafu je cesta (B, C, D, E, H, F) vyznačena tučně. Dále graf obsahuje kružnici (D, E, F, G, H, I), ale i kružnici (E, I, F, H). Klika $\{E, F, H, I\}$ je vyznačena šedě a nezávislá množina $\{A, C, E, G\}$ má vrcholy označeny bíle.



Rozmyslete si, jestli je vyznačená klika i nezávislá množina maximální. Také si rozmyslete, jak vypadá minimální souvislý podgraf tohoto grafu.

5.3 Rovinné grafy

Graf často kreslíme do roviny (na papír). Vrcholy kreslíme jako „puntíky“ a hrany jako čáry, které je spojují. Obrázku říkáme *nakreslení grafu* do roviny. Graf je *rovinný*, pokud lze nakreslit na papír bez křížení hran. Nakreslení do roviny bez křížení hran se říká *rovinné nakreslední* grafu. Jako příklad rovinných grafů uveďme graf sousedících států Jižní Ameriky, graf silniční sítě bez mostů a tunelů.

Pokud dostaneme nakreslení grafu, které není rovinné, tak to ještě neznamená, že graf není rovinný. Každý graf má řadu špatných nakreslení. Pro rovinnost stačí, aby existovalo jedno nakreslení bez křížení hran. Někdy můžeme křížící se hranu překreslit jako na následujícím obrázku.

Teď vezme do ruky nůžky a papír, na kterém je nakreslen graf, rozstříháme obrázek podél nakreslených hran. *Stěny* rovinného grafu jsou souvislé kusy roviny,

[6] letokruhy

na které se rovina rozpadne. Například graf na předchozím obrázku vpravo má 4 stěny (3 vnitřní a 1 vnější). Pozor, stěna je definována pouze pro rovinné nakreslení.

Věta 2 (Eulerova formule) *V rovinném grafu $G = (V, E)$ platí*

1. $|V| + |S| = |E| + 2$.
2. $|E| \leq 3|V| - 6$.

Stejná formule platí o pro konvexní mnohostěny (krychle, osmistěn, ...). Není divu, vždyť moucha sedící na stěně skleněného mnohostěnu vidí vrcholy a hrany jako rovinný graf (rozmyslete si, proč je graf rovinný).

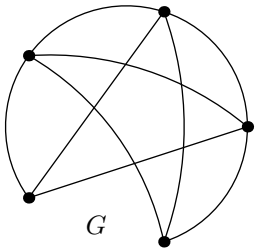
Druhé tvrzení věty je důsledkem prvního. Má velký význam pro odhad časové složitosti algoritmů pracujích na rovinných grafech. Ukazuje totiž, že rovinný graf nemůže mít příliš mnoho hran. Počet hran je $m \leq 3n - 6 = \mathcal{O}(n)$.

Řada problémů se dá v rovinných grafech řešit rychleji nebo jednodušeji než v obecných grafech. Příkladem je barvení vrcholů grafu co nejmenším počtem barev tak, aby každé dva vrcholy spojené hranou měli různou barvu. V obecných grafech je to velmi těžké.[7]

Věta 3 (o 4 barvách) *Vrcholy rovinného grafu lze obarvit 4 barvami tak, aby žádná hrana nespojovala dva vrcholy stejné barvy.*

Věta o 4 barvách je jednou z prvních vět, která byla dokázána s pomocí počítače.[8] Zkuste najít příklad rovinného grafu, jehož obarvení potřebuje alespoň 4 barvy.

O rovinném nakreslení grafu. Podívejte se na graf na obrázku. Je to rovinný graf?



Jak poznat, jestli je graf rovinný? Rovinné grafy jsou poměrně krásně charakterizovány následující větou.

Věta 4 (Kuratowski) *Graf G je rovinný právě tehdy když neobsahuje dělení K_5 nebo dělení $K_{3,3}$ jako podgraf.*

Dělení K_5 je graf, který vznikne z K_5 tak, že některé jeho hrany podrozdělíme. Podrozdělení hrany $uv \in E$ je nahrazení hrany uv cestou uPv, kde cesta P obsahuje samé nově přidané vrcholy (kromě u a v).

Kuratowského věta se dá použít v teoretických důkazech nebo pro nalezení „certifikátu“ nerovinnosti grafu (když se někdo zeptá, proč graf není rovinný, tak mu ukážete dělení K_5 nebo $K_{3,3}$, které je podgrafem). Algoritmicky je věta téměř nepoužitelná.

[7] Barvení obecného grafu je NP-úplný problém.

[8] Počítač nalezl 4-obarvení menších grafů z důkazu, pomocí kterých se dá obarvit libovolný rovinný graf.

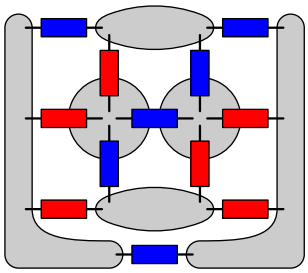
Jak najít rovinné nakreslení grafu? Existuje několik algoritmů, které nám naleznou rovinné nakreslení grafu a nebo odpoví, že graf není rovinný. Ačkoliv je jejich časová složitost jen $\mathcal{O}(n)$, jsou poměrně komplikované (jejich docela dobrý přehled naleznete v anglické Wikipedii pod heslem „planarity testing“). Pokud nevyžadujeme dobrou časovou složitost, tak nám poměrně dobře poslouží následující heuristika.

Pružinková heuristika: Heuristika využívá fyzikálních zákonů. Model, který si vytvoříme, odsimulujeme na počítači. Po určitém množtví kroků (iterací) dostaneme řešení, které sice není přesné, ale je mu velmi blízko.

Začneme s libovolným nakreslením grafu. Hrany grafu nahradíme pružinkami a do vrcholů umístíme náboje, které se vzájemně odpuzují. Potom necháme síly působit (odpudivé síly mezi vrcholy proti přitažlivým sílám pružinek). Působení sil realizujeme tak, že v několika iteracích pro každý vrchol spočítáme, kam se pohne. Skončíme, až nalezneme „téměř“ rovnovážný stav.[9] Odpudivé síly rozprostřou graf do co největší plochy, ale pružiny drží vrcholy spojené hranou blízko u sebe. Díky tomu dostaneme rovinné nakreslení (pokud je graf rovinný).

Věta 5 (Fáry) *Každý rovinný graf G se dá nakreslit do roviny tak, aby vrcholy odpovídaly bodům a hrany odpovídaly rovným úsečkám.*

Rovinné nakreslení grafu má řadu aplikací. Například při návrhu jednovstvých tištěných spojů, anglicky VLSI designs.



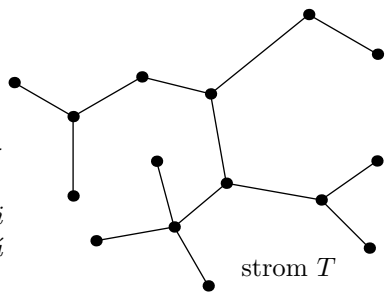
5.4 Stromy

Stromy jsou velmi důležitou třídou grafů. Mají velmi jednoduchou strukturu a proto se v nich řada věcí spočítá velmi jednoduše – téměř triviálně. Díky následujícím vlastnostem je používáme i při návrhu složitějších algoritmů.

Následující Lemma shrnuje základní vlastnosti stromu, které lze brát i jako ekvivalentní definice stromu.

Lemma 2 *Následující tvrzení jsou ekvivalentní:*

- *T je strom*
- *T je souvislý graf s $n-1$ hranami.*
- *(minimální souvislý podgraf) Přidáním libovolné hrany k T vznikne kružnice.*
- *(jednoznačnost cesty) Mezi libovolnými dvěmi vrcholy T vede jednoznačně určená cesta.*



[9] Pozor na oscilace! Soustava pružinek se často rozkmitá. Proto je vhodné k pohybu vrcholů přidat tření, které oscilace ztlumí.

Jednoznačnou cestu z vrcholu x do vrcholu y ve stromě T budeme značit xTy. Vrcholy stromu stupně 1 se nazývají *listy*.

Každý strom můžeme zkonstruovat z jednoho vrcholu postupným přilepováním listů. Tato rekurzivní konstrukce se velmi hodí jak v důkazech, tak v algoritmech. Nahlédnout to můžeme analýzou pozpátku. Natočíme si film o tom, jak postupně odtrháváme listy (najít list a odtrhnout je jednoduché). Když si film pustíme pozpátku, uvidíme jak strom vzniká z jednoho vrcholu postupným přilepováním listů.

5.5 Zakořeněné stromy

V algoritmech se častěji setkáme se stromy, které mají jeden význačný vrchol – kořen. Když pracujeme se stromem, tak většinou v jednom vrcholu začneme a z něj postupujeme do dalších vrcholů. Příkladem takového stromu je strom rekurzivních volání (vrcholy jsou instance funkcí[10]; 2 instance funkce jsou spojeny hranou, pokud jedna zavolala druhou). Začneme v hlavní funkci a odtamtud voláme všechny ostatní funkce.

Při popisu algoritmů se bez zakořeněných stromů neobejdeme. Často je používáme, aniž bychom rozuměli všem pojmům, a proto si je pojďme ujasnit.

Zakořeněný strom T je strom s jedním význačným vrcholem, který nazýváme *kořen*. Místo slova vrchol, se někdy používá termín *uzel*, oba termíny mají stejný význam. Představme si, že jsme hrany stromu zorientovali směrem od kořene (z hran se staly šipky). Vrcholy, ze kterých nevychází žádná šipka, se nazývají *listy*. Ostatní vrcholy jsou *vnitřní vrcholy stromu*.

Vrchol w je *syn (přímý následník)* vrcholu v, pokud z v vede šipka do w. Naopak v je *otec (přímý předchůdce)* vrcholu w.[11] Každý vrchol kromě kořene má právě jednoho otce, ale obráceně jeden vrchol může mít libovolný počet synů. Vrchol u je *následník (potomek)* vrcholu v, pokud v leží na cestě z kořene do u. Naopak v je *předchůdce* u.

Podstrom určený vrcholem x je podgraf T indukovaný všemi následníky vrcholu x. Tento podstrom je opět zakořeněným stromem s kořenem x. Podstrom určený x je ten kus grafu, který odpadne od zakořeněné části, když přeřízneme hranu vedoucí mezi x a jeho otcem y. Proto někdy podstromu určenému x říkáme větev y.

- *Stupeň vrcholu* x v zakořeněném stromě T je počet jeho synů.
- *Hloubka vrcholu* x v T je délka cesty z kořene do x. Kořen je tedy v hloubce nula.
- *Hloubka stromu* T je největší hloubka v T, tedy délka nejdelší cesty od kořene k listu.
- *k-tá hladina* stromu T je množina všech vrcholů stromu T ležících v hloubce k. Hladiny začínáme počítat od nulté.
- *Šířka stromu* T je velikost největší hladiny ve stromě T.
- *Výška stromu* T je počet hladin, což je hloubka stromu T plus jedna.

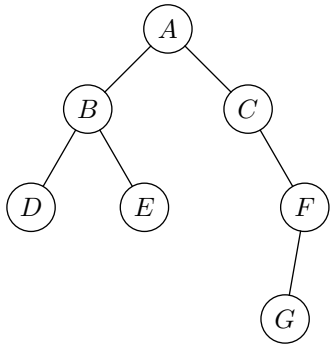
Do obrázku orientaci hran často nekreslíme. Hrany jsou orientovány z vrcholu ležícího výše do vrcholu ležícího níže. Proto kreslíme syny vrcholu x pod vrchol

[10] To, co se ukládá na zásobník.

[11] Někdy se ve stejném významu ještě používají pojmy „rodič“ a „děti“.

x, navíc často rovnáme zleva doprava. Můžete si to představit tak, že graf uvázaný z kuliček a provázků chytneme za kořen, který zvedneme. Když tento model připlácneme a otiskneme na tabuli, dostaneme správné nakreslení.

Příklad: Na obrázku vpravo je zakořeněný strom T o 7 vrcholech s kořenem A. Vrchol B je otec vrcholů D, E. Naopak D, E jsou synové vrcholu B. Vrcholy A, C jsou předchůdci vrcholu F a G je jeho jediný následník/potomek. Podstrom určený vrcholem B je indukován vrcholy B, D, E. Podstrom určený vrcholem C je indukován vrcholy C, F, G.
Šířka stromu T je 3, hloubka stromu T je také 3 a výška stromu T je 4. V nulté hladině je pouze vrchol A. Vrcholy ve 2. hladině jsou $\{D, E, F\}$



Často pracujeme se speciálními stromy. *Binární strom* je strom, ve kterém má každý je vrchol nejvýše dva syny. Ty potom často označujeme jako *levého* a *pravého syna.* Obecně, *k-ární strom* je strom, ve kterém má každý vrchol nejvýše k synů.

Stromová datová struktura je reprezentace stromu v počítači. V každém vrcholu stromu v si budeme pamatovat hodnotu $x(v)$, které se říká klíč (key).

Při reprezentaci stromu v počítači je důležité, abychom se z každého vrcholu uměli dostat k jeho synům a z každého vrcholu, kromě kořene, k jeho rodiči. Je jedno, jestli si strom reprezentujeme dynamicky pomocí ukazatelů (pointrů) a nebo staticky v poli.

V reprezentaci stromu si často nepamatujeme pamatovat odkazy na otce. A to proto, že je můžeme zjistit jinak. Téměř vždy procházíme strom od kořene až se dostaneme do vrcholu v. Během průchodu si můžeme vrcholy na cestě od kořene k v ukládat na zásobník. Tím získáme dokonce všechny předchůdce vrcholu v.

5.6 Příklady

1. (Graf známostí) Vrcholy grafu jsou jména lidí a dva vrcholy jsou spojeny hranou, pokud se 2 lidé odpovídající vrcholům znají. Dokažte následující tvrzení:

   (a) V každé skupině $n \geq 2$ lidí jsou dva, kteří znají stejný počet lidí ze skupiny.

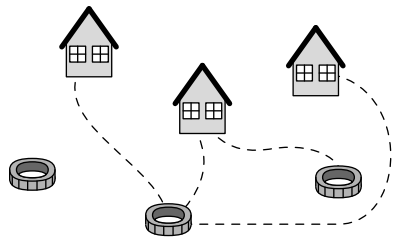
   (b) Mezi šesti lidmi vždy existují 3 lidé, kteří se vzájemně znají a nebo 3 lidé, kteří se vzájemně neznají.[12]

   (c) Každá skupina lidí může být rozdělena na dvě skupiny tak, aby aspoň polovina známých člověka X byla v jiné skupině než X, pro každého člověka X.

   (d) Pokud každý člověk zná alespoň polovinu lidí ze skupiny, tak můžeme lidi posadit kolem kulatého stolu tak, aby každý seděl mezi lidmi, které zná.

2. (Nakreslení grafu) Na Obrázku jsou 3 domečky a 3 studny. Vaším úkolem spojit každý domeček s každou studnou pěšinou tak, aby se pěšiny vzájemně nekřížili.

[12] Toto pozorování jde zobecnit. Existuje přirozené číslo $R(m, n)$ takové, že mezi alespoň $R(m, n)$ lidmi vždy existuje m lidí, kteří se vzájemně znají a nebo n lidí tak, že se vzájemně neznají. Říká se tomu Ramseyova věta.



Nezkoušejte to ale déle jak 10 minut, raději dokažte, proč to nejde.

3. Jaký je maximální počet vrcholů binárního stromu o h hladinách? A jaký je maximální počet vrcholů k-árního stromu o h hladinách? Co z toho plyne pro minimální výšku těchto stromů? Kolik může být maximální výška binárního stromu?

4. Dokažte, že v binárním stromě je počet vnitřních uzlů stupně dva roven počtu listů mínus jedna.

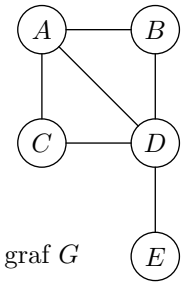
5. Dokažte, že v každém binárním stromě s L listy existuje podstrom s $l \in \langle L/3, 2L/3 \rangle$ listy.

Kapitola 6

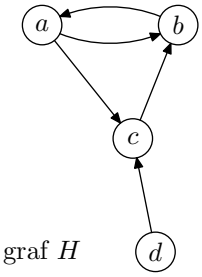
Reprezentace grafu

V kapitole 5 jsme se dozvěděli, co to jsou grafy a k čemu jsou dobré. Brzo budeme chtít napsat nějaký program, který s grafy pracuje. Ale jak si takový graf uložit do počítače? To si teď vysvětlíme.

Všechny možné reprezentace si budeme vysvětlovat na následujícím grafu $G = (V, E)$ a orientovaném grafu $H = (W, F)$. V každé sekci budou jejich reprezentace uvedeny jako příklad.



graf G



graf H

V této knize označujeme vrcholy grafu písmeny abecedy. Je to tak lepší pro výklad a diskuse nad grafy v příkladech. V počítači označujeme vrcholy pomocí čísel 1, 2 až n.[1]

Výhody a nevýhody jednotlivých reprezentací jsou shrnuty na konci kapitoly.

6.1 Seznam hran

Uložíme si počet vrcholů a seznam všech hran. Hrana je dvojice vrcholů. V neorientovaných grafech je jedno, jestli si hranu $uv \in E$ pamatujeme jako dvojici uv nebo vu. Ale v případě orientovaných grafů už musíme dodržet správné pořadí vrcholů. Většinou za správné pořadí považujeme to po směru šipky. Orientovaný graf může kromě šipky uv obsahovat i opačnou šipku vu.

Příklad: Neorientovaný graf na obrázku má 5 vrcholů A až E a seznam hran AB, AC, AD, BD, CD, DE.

Orientovaný graf má 4 vrcholy a až d a seznam hran ab, ac, ba, cb a dc.

Tato reprezentace je vhodná spíše pro zadávání grafu na vstupu nebo pro skladování grafu na pevném disku.

[1] Často ale začínáme od 0 a končíme vrcholem $n - 1$.

6.2 Matice sousednosti

Matice sousednosti zachycuje, které vrcholy spolu sousedí. V matici si pro každou dvojici vrcholů (u, v) pamatujeme, jestli z vrcholu u vede hrana do vrcholu v. Z toho je vidět, že matice reprezentuje orientované grafy.

Neorientovaný graf G si můžeme reprezentovat tak, že ho nejprve převedeme na orientovaný graf $\vec{G}$ a ten teprve reprezentujeme pomocí matice sousednosti. Orientovaný graf $\vec{G}$ dostaneme z G tak, že každou neorientovanou hranu $uv \in E$ nahradíme dvojicí šipek uv a vu, jdoucích proti sobě.

Matice sousednosti A má velikost $n \times n$ a je definována jako $A = (a_{u,v})$, kde

$$a_{u,v} = \begin{cases} 0 & \iff uv \notin E \\ 1 & \iff uv \in E \end{cases}$$

Protože graf neobsahuje smyčky (hrany z v do v), tak je $a_{v,v} = 0$ pro každý vrchol $v \in V$. Podobně je vidět, že pro neorientované grafy dostaneme symetrickou matici.

Příklad:

$$A(G) = \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \begin{array}{c} A \\ B \\ C \\ D \\ E \end{array} \qquad A(H) = \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \begin{array}{c} a \\ b \\ c \\ d \end{array}$$

Matici sousednosti budeme celkem často používat, protože se s ní jednoduše pracuje. Hlavně velmi rychle zjistíme, jestli je uv hranou grafu.

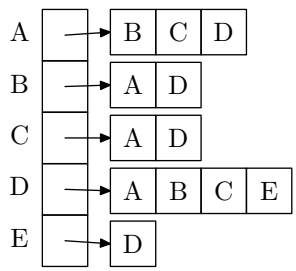
Matice sousednosti obsahuje jen nuly nebo jedničky. Často si ale boolovskou matici rozšíříme na matici integerů, protože pak si v a_{ij} můžeme pamatovat ohodnocení hrany ij. Pokud graf není úplný, tak musíme určit jednu hodnotu, která znamená, že hrana neexistuje (běžně se bere „počítačové ∞“, které pro integery je MAXINT; podobně pokud jsou všechna ohodnocení hran nezáporná, tak můžeme použít -1). Pokud si do a_{ij} uložíme vzdálenost vrcholu i od vrcholu j, tak dostaneme *matici vzdáleností.*

6.3 Seznam sousedů

Reprezentaci si vysvětlíme pro orientované grafy. Neorientovaný graf G bychom si reprezentovali tak, že ho nejprve převedeme na orientovaný graf $\vec{G}$, který už budeme umět reprezentovat (viz jak to bylo popsáno u matice sousednosti).

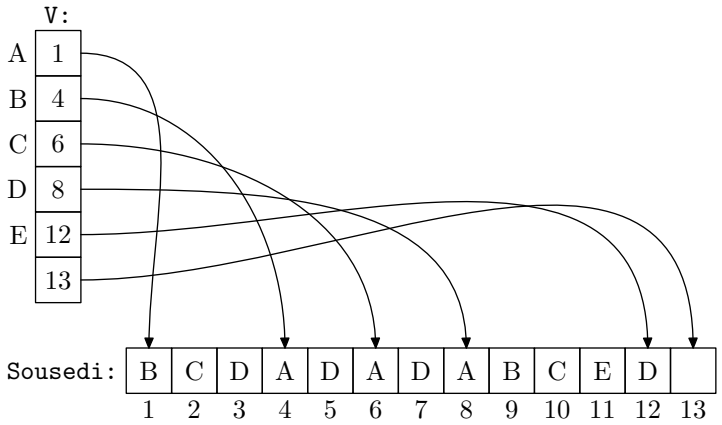
Teď se podívejme, jak reprezentovat orientovaný graf $G = (V, E)$. Pro každý vrchol $v \in V$ si budeme pamatovat seznam jeho sousedů. To je vrcholy, do kterých z v vede hrana.

Příklad:



Abychom se vyhnuli spojovým seznamům a přitom neplýtvali pamětí, tak jednotlivé seznamy poskládáme za sebe do jednoho pole. Abychom neztratili přehled o tom, kde který seznam sousedů začíná, tak si stále ponecháme pole `V`$[\cdot]$ obsahující ukazatele na začátky seznamů. Políčko `V`$[i]$ bude obsahovat index do pole `Sousedi`$[\cdot]$, na kterém začíná seznam sousedů vrcholu i. Seznam sousedů vrcholu i bude končit o jednu pozici dříve než začíná seznam sousedů vrcholu $i+1$. Abychom věděli kde končí seznam sousedů posledního vrcholu, tak rozšíříme obě pole o jedna (odpovídá to přidání fiktivního vrcholu s prázdným seznamem sousedů).

Příklad:



6.4 Výhody a nevýhody jednotlivých reprezentací

Nejprve si musíme rozmyslet, co chceme v grafu dělat. Mezi nejčastější operace patří testování, jestli jsou dva vrcholy $u \in V$ a $v \in V$ spojeny hranou. Jinými slovy potřebujeme zjistit, zda je uv hrana. Druhou častou operací je průchod všech sousedů vrcholu v.

Následujíc tabulka shrnuje, jak dlouho tyto dotazy trvají v jednotlivých reprezentacích. Zároveň udává i prostorovou složitost každé reprezentace.

Reprezentace	Je uv hrana?	Projít sousedy v	Prostorová složitost
seznam hran	$\mathcal{O}(m)$	$\mathcal{O}(m)$	$\mathcal{O}(m)$
matice sousednosti	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
seznam sousedů	$\mathcal{O}(n)$	$\mathcal{O}(\#\text{sousedů})$	$\mathcal{O}(n+m)$

Matice sousednosti nám jako jediná reprezentace umožňuje v konstantním čase určit, jestli je uv hrana grafu (však je tomu šitá na míru). Pro řídké grafy (grafy, které mají málo hran) za to zaplatíme vyšší paměťovou náročností. Příkladem řídkých grafů jsou rovinné grafy, které mají nejvýše $3n-6 = \mathcal{O}(n)$ hran. Na druhou stranu pro husté grafy (grafy, které mají alespoň $c \cdot n^2$ hran pro nějaké $c > 0$) je to ideální reprezentace. Zabere lineárně mnoho prostoru ve velikosti grafu, testování, jestli je uv hrana, proběhne v čase $\mathcal{O}(1)$ a dokonce i průměrný počet sousedů je $\Omega(n)$.

V některých algoritmech hodně často procházíme všechny sousedy některých vrcholů a testy, zda je uv hrana, ani nepotřebujeme. V takových případech je nejlepší použít reprezentaci grafu seznamem sousedů.

Z předchozích odstavců je vidět, že je každá reprezentace někdy výhodná, každá jindy. Záleží na tom, co s grafem chceme provádět. Pro každý algoritmus si musíme vybrat individuálně.

Poznámka: Někdy si spolu s grafem chceme reprezentovat i jeho další vlastnosti. Například pro rovinné grafy chceme znát seznam všech stěn. Pro každou

stěnu chceme rychle zjistit seznam hran a vrcholů, které obsahuje. Naopak pro každou hranu chceme vědět, ve kterých stěnách je obsažena. Návrh takové datové struktury už není složitý a proto necháváme na čtenáři, aby si rozmyslel, jak takové věci efektivně reprezentovat.

6.5 Příklady

1. (Převody mezi reprezentacemi) Pro každé dvě reprezentace grafu navrhněte nejefektivnější způsob převodu z první reprezentace do druhé. Určete časovou složitost převodu.

2. (Obrácení hran v orientovaném grafu) Dostanete orientovaný graf reprezentovaný maticí sousednosti A. Obrátit všechny šipky na druhou stranu je jednoduché, stačí vzít matici transponovanou A^{T}. *Matice transponovaná* k matici $A = (a_{i,j})$ je matice $A^{\mathrm{T}} = (a_{j,i})$ (prohodíme řádky za sloupce a naopak). Ale co když dostaneme graf reprezentovaný seznamem sousedů? Jak rychle obrátíte hrany v této reprezentaci?

3. (Mocnina orientovaného grafu) Mocnina orientovaného grafu $G = (V, E)$ je graf $G^2 = (V, E^2)$, kde $uv \in E^2$ právě tehdy když z u vede orientovaná cesta délky 2 do v (neboli existuje vrchol $w \in V$ takový, že $uw \in E$ a $wv \in E$). Navrhněte efektivní algoritmus, který dostane graf G reprezentovaný buď maticí sousednosti nebo seznamem sousedů, a vytvoří graf G^2 reprezentovaný stejným způsobem. Jaká je časová složitost vašeho algoritmu?

4. (Hledání stoku) Dostanete orientovaný graf reprezentovaný maticí sousednosti. Většina algoritmů pracujících s maticí sousednosti má časovou složitost alespoň $\Omega(n^2)$. Ale jsou i výjimky.

   (a) Zkuste co nejrychleji najít univerzální stok grafu. *Univerzální stok* je takový vrchol, do kterého vede $n-1$ hran ze všech ostatních vrcholů, ale ze kterého nevedou žádné hrany ven. Zvládnete to v čase $\mathcal{O}(n)$?

   (b) Zkuste co nejrychleji najít stok grafu. *Stok* je takový vrchol, ze kterého nevedou žádné hrany ven. Jaká bude časová složitost vašeho algoritmu?

5. (a) (*Matice incidence*) Česky bychom mohli říci matice náležení. Tato matice zachycuje, kdy platí $v \in e$ pro $v \in V$ a $e \in E$. Matice incidence I má velikost $n \times m$ a je definována jako $I = (i_{v,e})_{v \in V, e \in E}$, kde

$$i_{v,e} = \begin{cases} 0 & \iff v \notin e \\ 1 & \iff v \in e \end{cases}$$

   Pro graf G ze začátku kapitoly dostáváme následující matici.

$$I(G) = \begin{array}{c} \begin{array}{cccccc} e_{AB} & e_{AC} & e_{AD} & e_{BD} & e_{CD} & e_{DE} \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array}$$

   V každém sloupci jsou právě dvě jedničky, protože každá hrana má právě dva konce. Pokud bychom chtěli takto reprezentovat orientovaný graf, tak musíme jedničku u výchozího vrcholu šipky změnit na mínus jedničku.

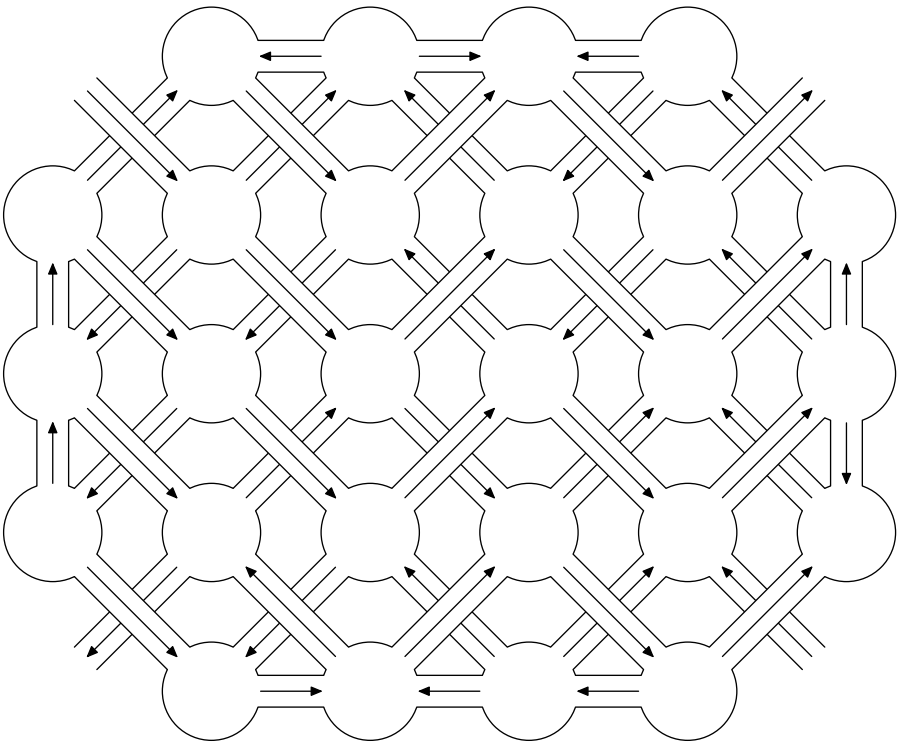
(b) *Matice incidence* orientovaného grafu $G = V, E)$ je matice tvaru $n \times m$ $B = (b_{v,e})$, kde

$$b_{v,e} = \begin{cases} -1 & \text{pokud hrana } e \text{ vychází z vrcholu } v \\ 1 & \text{pokud hrana } e \text{ přichází do vrcholu } v \\ 0 & \text{jinak.} \end{cases}$$

Zjistěte, jaký význam mají položky matice $B \cdot B^{\mathrm{T}}$, kde B^{T} značí matici transponovanou k matici B.

Kapitola 7

Průchod grafu



Na obrázku je speciální bludiště. Skládá se z několika místností, které jsou propojeny chodbami. Každá chodba se dá projít jen ve směru šipky, protože je zavřená dveřmi, které mají z jedné strany kliku a z druhé strany kouli. Dokážete projít bludištěm? Jestli se Vám to zdá jednoduché, tak zkuste tohle: Do 10 minut zjistěte, jak projít celé bludiště tak, abyste každou místnost navštívili právě jednou. Pokud to do 10 minut nestihnete, nebo některou místnost projdete dvakrát, tak spustíte alarm.

7.1 Efektivní průchod grafu

Průchod bludištěm je pro lidi výzvou už po několik století. Vzpomeňme si například na řecké báje a pověsti, na pověst o Mínotaurovi. Theseus chtěl zachránit Ariadnu a proto musel projít bludiště, najít Minutaura, zabít ho a zase se vrátit zpátky. Aby se v bludišti neztratil, dostal od Ariadny niť, pomocí které si označoval cestu. Když

postupoval do neznámých míst bludiště, tak niť odmotával. Když se vracel, tak niť namotával zpátky na klubko. Namotávání niťě ho vyvedlo zpátky před vchod do bludiště.

Asi každý by dokázal pomocí nitě a křídy projít bludiště, ale jak to naučit počítač?[1] Bludiště si v počítači reprezentujeme jako graf. Vrcholy, do kterých se lze dostat z výchozího místa nazveme *dosažitelné*. Pro jednoduchost předpokládejme, že všechny vrcholy bludiště jsou dosažitelné. Později si ukážeme, jak se tohoto předpokladu zbavit.

Efektivní průchod grafu musí splňovat následující body:

- každou hranou projdeme maximálně jednou (jednou tam a jednou zpět)
- hranou se vrátíme až když z vrcholu nevede další cesta
- hranou vedoucí do navštíveného vrcholu se ihned vrátíme

Algoritmus splňující výše uvedené body je konečný a korektní. Konečnost plyne z prvního bodu, protože v každém kroku projdeme po jedné hraně, každou hranu jen jednou a hran je jen konečně mnoho. Korektnost algoritmu v našem případě znamená, že projdeme všechny vrcholy a hrany, které jsou dosažitelné z výchozího vrcholu. Pokud by existoval algoritmem nenavštívený, ale jinak dosažitelný, vrchol v, tak se podívejme na cestu P vedoucí z výchozího vrcholu do v. Taková cesta musí existovat, protože vrchol v je dosažitelný. Protože začátek cesty P byl navštívený a konec nebyl, tak na cestě existuje neprošlá hrana vedoucí z navštíveného vrcholu do nenavštíveného. To je ale spor s druhým bodem.

Časová složitost algoritmu je $\mathcal{O}(n + m)$, protože čas za průchod naúčtujeme[2] hranám, které procházíme, a případnou práci ve vrcholech vrcholům.[3]

Efektivní průchod grafu má dvě možné implementace:

1. **Průchod do hloubky (DFS z anglického Depth First Search)** – podle tohoto algoritmu postupuje i Theseus. Před bludištěm si uváže jeden konec nitě na strom a vstoupí dovnitř. V prvním vrcholu si vybere jednu možnou cestu/hranu a projde po ní do dalšího vrcholu. Aby Theseus neměl zmatek v tom, které hrany už prošel, tak si všechny hrany, které prochází označuje křídou — a to na obou koncích. V každém vrcholu, do kterého Theseus dorazí, provede následující:
   - Pokud na zemi najde položenou niť, tak ví, že už ve vrcholu byl a že se do něj při namotávání nitě zase vrátí. Odloží tedy další prozkoumávání tohoto vrcholu na později, provede čelem vzad a začne namotávat niť na klubko. To ho dovede zpátky do předchozího vrcholu.
   - Pokud na zemi žádnou niť nenajde, tak se vydá první možnou neprošlou hranou. Pokud by taková hrana neexistovala, tak je vrchol zcela prozkoumán. V tom případě Theseus neztrácí čas a začne namotávat niť na klubko. Tím se dostane zpátky do předchozího vrcholu.

[1] Někteří lidé si myslí, že pro průchod zahradního bludiště stačí použít následující pravidlo pravé ruky: Při průchodu bludištěm se budeme pravou rukou neustále držet zdi. Takto podél zdi projdeme celé bludiště a zase se vrátíme na začátek. Mají pravdu, že se vrátíme zpátky na začátek bludiště, ale bohužel ho neprojdeme celé. Dovedete najít příklad bludiště, kde pravidlo pravé ruky selže?

[2] Viz počítání amortizované časové složitosti v kapitole **??**.

[3] Práce ve vrcholech se může vykonávat při prvním vstupu do vrcholu, mezi návratem z jedné hrany a odchodem do další hrany nebo při posledním opouštění vrcholu. Podle toho označujeme práci ve vrcholu jako *preorder, inorder* a *postorder*. Čas za práci preorder a postorder naúčtujeme vrcholům a čas za práci vykonávanou inorder hranám, které následně projdeme.

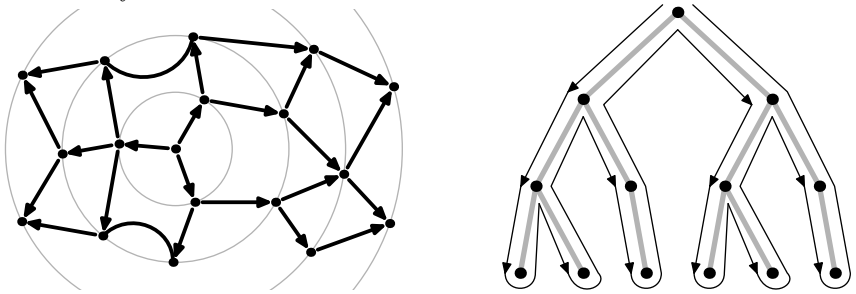
Tímto postupem prozkoumá celé bludiště a nakonec se vrátí do výchozího vrcholu.

Jak to provést v počítači? Křídu potřebujeme proto, abychom se nezacyklili. Abychom každou hranu prošli jen jednou. Místo křídy si pro každou hranu zřídíme proměnnou označující, jestli jsme hranu prošli. Klubíčko a niť potřebujeme proto, abychom našli cestu ven z bludiště. Položená niť nám vyznačuje cestu z výchozího vrcholu do aktuálního vrcholu. V počítači nám bude úplně stačit, když si tuto cestu budeme pamatovat jako posloupnost vrcholů na této cestě. Pro uložení této cesty použijeme zásobník. Odmotávání nitě z klubka při postupu do dalšího vrcholu odpovídá přidání vrcholu na zásobník. Namotávání nitě na klubko při návratu zpět odpovídá odebrání vrcholu ze zásobníku.

Stejně postupuje i *backtracking* (to je ta metoda „Hrr, na ně! A když to nepůjde, tak coufnem."). Backtracking je metoda hrubé síly, která vyzkouší všechny možnosti.

2. **Průchod do šířky (BFS z anglického Breadth First Search)** – tento průchod (prohledání grafu) si můžeme představit tak, že se do výchozího vrcholu postaví miliarda Číňanů a všichni naráz začnou prohledávat graf. Když se cesta rozdělí, tak se rozdělí i dav řítící se hranou. Předpokládáme, že všechny hrany jsou stejně dlouhé. Graf prozkoumáváme „po vlnách". V první vlně se všichni Číňané dostanou do vrcholů, do kterých vede z výchozího vrcholu hrana. V druhé vlně se dostanou do vrcholů, které jsou ve vzdálenosti 2 od výchozího vrcholu. Podobně v k-té vlně se všichni Číňané dostanou do vrcholů ve vzdálenosti k od výchozího vrcholu. Kvůli těmto vlnám se někdy průchodu do šířky říká *algoritmus vlny.* V počítači vlnu nasimulujeme tak, že při vstupu do nového vrcholu uložíme všechny možné cesty do fronty. Frontu průběžně zpracováváme.

Na následujícím obrázku nalevo je znázorněn průchod do šířky a napravo průchod do hloubky.



Který průchod grafu je lepší? Jak kdy a jak na co. To záleží na tom, jak vypadá graf, který prohledáváme. Někdy je výhodnější jedna metoda o proti druhé, protože budeme potřebovat výrazně méně paměti na uložení zásobníku/fronty. Jindy je to naopak.

Příklad: Dostaneme bludiště, které má tvar čtvercové sítě o rozměrech $n \times n$, a jsou v něm pouze obvodové zdi. Je to takové fotbalové hřiště s mantinely. Někde v bludišti jsme ztratili míč, protože se udělala hrozná mlha. V mlze je vidět jen o malinko víc než na jedno políčko bludiště. Chtěli bychom projít celé bludiště, navštívit každé políčko, a míč najít.[4] Vchod do bludiště je pravém horním rohu.

Při použití DFS, ve kterém z každého prozkoumávaného políčka zkoušíme cesty v pořadí nahoru, doprava, dolů a doleva, bude neustále možné pokračovat dále. A

[4] Úlohu můžeme přeformulovat i tak, že chceme obarvit všechna políčka bludiště tím, že na první políčko vylijeme plechovku barvy. Barva se rozlije po všech dostupných políčkách. Proto tomuto způsobu barvení říkáme záplavové vyplňování (floodfill).

to tak dlouho, dokud nezaplníme skoro všechna políčka. Jednoduše řečeno, průchod bludištěm bude vypadat jako dlouhá cesta naskládaná zig-zag po celém bludišti. (Pokud to není jasné, tak si zkuste průchod odkrokovat.) Všechna políčka budou na zásobníku a proto budeme potřebovat zásobník velikosti $\mathcal{O}(n^2)$. Při použití BFS budeme bludiště procházet v soustředných vlnách a bude nám stačit fronta velikosti $4n$ (nejvýše obvod bludiště). Ve fotbalové analogii to odpovídá vytvoření rojnice.

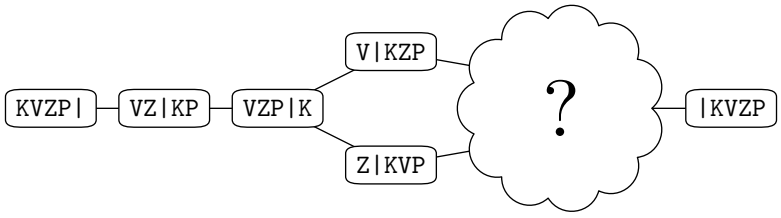
Takto se zdá, že z paměťových důvodů je lepší použít BFS. Toto zdání ale klame. Podívejme se na další příklad.

Příklad: Jsme na dovolené a bydlíme v hotelu. Hotel má 5 pater, do kterých se dostaneme výtahem. Na každém patře je dlouhá chodba se dveřmi do 20 pokojů. Potřebovali bychom zjistit, na kterém pokoji bydlí ta hezká slečna, co jsme ji viděli u snídaně, ale nevíme jak se jmenuje. Rozhodneme se proto projít celý hotel a slečnu najít. Potřebujeme projít všechny místnosti hotelu a v každé nechat vzkaz. Pokud použijeme DFS, vystačíme si se zásobníkem velikosti 3 (výtah, chodba, pokoj). Při použití BFS budeme potřebovat tak velkou frontu, kolik je v hotelu pokojů, tedy 100.

Výhodnou BFS je, že prochází vrcholy v pořadí podle nejkratší vzdálenosti od počátku. Pokud tedy hledáme nejkratší cestu, nejmenší počet tahů apod., tak je výhodnější použít BFS.

Příklad: (Koza, vlk a zelí) Na břehu řeky má převozník kozu, vlka a zelí. Do loďky se mu vejde jen jedno zvíře nebo zelí. Jak má převézt kozu, vlka a zelí na druhý břeh, když na žádném břehu nesmí nechat nehlídaného vlka s kozou (vlk by sežral kozu) ani kozu se zelím (koza by snědla zelí)?

Vrcholy grafu jsou stavy které označují, kdo je na kterém břehu. Například `VZ|KP` značí, že na prvním břehu zůstal vlk se zelím a na druhém břehu převozník s kozou. Přechod po hraně grafu odpovídá převozu zvířete, zelí nebo přejezdu převozníka na protější břeh. Například na začátku jsme se převozem kozy dostali ze stavu `KVZP|` do `VZ|KP`. Řešení úlohy lze najít průchodem tohoto grafu a nalezením cesty do vrcholu `|KVZP`.



Při hledání dalších stavů a kreslení obrázku nevidíme, jak to bude vypadat dál. Chová se to jako opravdové bludiště. Takovéto úlohy se vyskytují celkem často. Graf je „skrytý“ a odpovídá možným větvením programu.

Prohledáváme takzvaný *„stavový prostor“* programu. Pozor, často se necháme zmýlit tím, co je „stav“. Stav zachycuje celou situaci. V předchozí úloze nebylo stavem jen to zvíře, které převážíme přes řeku (koza nebo vlk či zelí). Stav musí obsahovat polohu všech zvířátek a zelí, včetně převozníka.

Podobnou úlohou je proskákání šachovnice šachovým koněm tak, abychom každé políčka navštívili právě jednou. V této úloze není stavem jen poloha naposledy položeného koně a počet umístěných koní. Stavem je „fotka“ celé šachovnice. To je poloha všech dosud umístěných koní.

V řadě úloh nám ujasnění si toho, co je stav, pomůže k nalezení řešení.

7.2 DFS na neorientovaném grafu

V implementaci DFS můžeme místo zásobníku využít rekurze. Ta je ve skutečnosti implementována zase pomocí zásobníku, ale je příjemnější pro programátora. Abychom si ulehčili výklad, tak budeme během průchodu grafu barvit vrcholy. Vrcholy, které jsme ještě nenavštívili budou mít bílou barvu. Vrcholy, které zpracováváme (už jsme se do nich dostali, ale ještě jsme neprozkoumali všechny cesty z nich vedoucí) barvu šedou a hotové vrcholy barvu černou. Šedé jsou přesně ty vrcholy, které jsou na zásobníku.

V polích `in[v]` a `out[v]` si budeme pro každý vrchol pamatovat čas, kdy jsme do něj poprvé vstoupili a kdy jsme ho nadobro opustili.

Při praktickém použití není nutné rozlišovat všechny tři barvy, ani si pamatovat `in[v]` a `out[v]`. Stačí, když budeme schopni rozlišit, které vrcholy jsme už navštívili a které ještě ne.

Graf si reprezentujeme seznamem sousedů, takže seznam `sousedi[v]` je seznam sousedních vrcholů vrcholu v.

```
Projdi(v):
    color[v]=GREY
    time=time+1
    in[v]=time
    for each w ∈ Sousedi[v] do
        if color[w]=WHITE then
            Projdi(w)
    color[v]=BLACK
    time=time+1
    out[v]=time
```

Procedura `Projdi(v)` projde tu část grafu, které je dosažitelná z výchozího vrcholu. Pokud by graf měl několik oddělených částí, tak musíme do každé části převézt Thesea helikoptérou. Postupně v každé části zvolíme výchozí místo a z něj graf projdeme. To obstará následující procedura DFS.

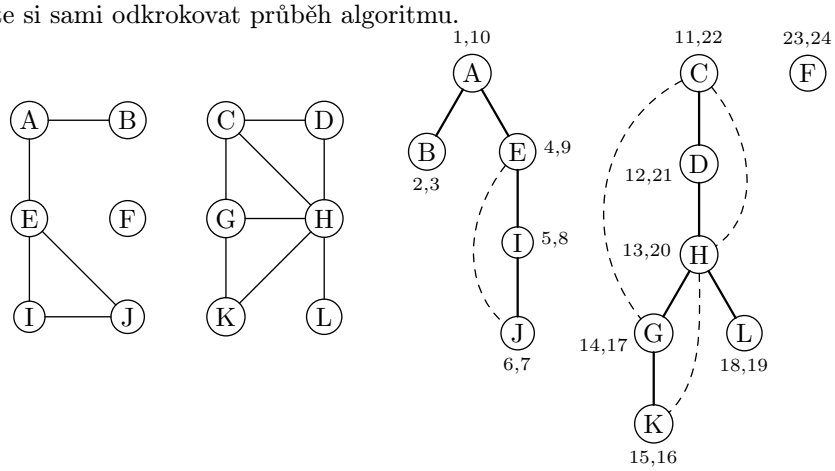
```
DFS():
    ∀v ∈ V : color[v]=WHITE
    time=0
    for each v ∈ V do
        if color[w]=WHITE then
            Projdi(w)
```

Podívejme se nejprve na tu část grafu, kterou jsme prošli procedurou `Projdi(v)`. Výchozí vrchol v nazveme *kořen*. Hrany, po kterých jsme v průběhu algoritmu prošli do nenavštíveného bílého vrcholu tvoří strom. Proto tyto hrany nazveme *stromové* a jim odpovídající strom nazveme *DFS stromem* (stromem průchodu do hloubky). Ostatním hranám budeme říkat *zpětné,* protože vedou zpět do již navštívených vrcholů. Stromy průchodu všech oddělených částí dohromady tvoří *DFS les.*

Pokud při prozkoumávání vrcholu v projdeme stromovou hranou do vrcholu u, tak řekneme, že u je *synem* v. Naopak v je *otcem* u. Všechny synové vrcholu v, synové synů a tak dále jsou *potomci* vrcholu v. Obráceně se dá říci, že v je jejich *předchůdce.* Někdy místo potomci říkáme *následníci.* Ještě jednou a přesněji, u je následník vrcholu v v zakořeněném stromě, pokud v leží na jednoznačné cestě z kořene do u.

Na obrázku budeme syny každého vrcholu kreslit pod jejich otce a v pořadí zleva doprava tak, jak jsme na ně při průchodu do hloubky narazili.

Příklad: Na následujícím obrázku vlevo je graf G, který projdeme pomocí DFS. Pokaždé, když si algoritmus můžete vybrat z několika vrcholů, tak vybereme abecedně nejmenší vrchol. Na obrázku vpravo je strom průchodu grafu G do hloubky. Čísla u každého vrcholu v jsou hodnoty `in[v]`, `out[v]`, tj. časy příchodu a odchodu. Zkuste si sami odkrokovat průběh algoritmu.



7.3 Komponenty souvislosti

Motivace: Máme ohromné bludiště, které vzniklo tak, že někdo v ohromné hale postavil z cihel spoustu zdí. Do haly vede několik vchodových dveří. Chtěli bychom zjistit, kolik oddělených bludišť v hale je (několik vchodů může vést do stejného bludiště). Dále chceme zjistit, kolik nejméně zdí musíme probourat, abychom dostali jedno velké bludiště.

Definice: Graf je *souvislý*, pokud mezi každýma dvěma vrcholy vede cesta. Pokud graf není souvislý, tak se skládá z jednotlivých částí, které už souvislé jsou. Těmto částem budeme říkat *komponenty souvislosti.* Jinak řečeno, komponenta souvislosti je maximální souvislý podgraf.

Pro nalezení komponent souvislosti použijeme průchod do hloubky s malou úpravou. Víme, že procedura `Projdi(`v`)` projde všechny vrcholy dosažitelné z v tj. vrcholy komponenty obsahující vrchol v. Proto stačí, když si při tomto průchodu budeme vrcholy označovat číslem komponenty.

Časová složitost nalezení komponent souvislosti stejná jako časová složitost DFS a to je $\mathcal{O}(n + m)$.

7.4 Komponenty 2-souvislosti

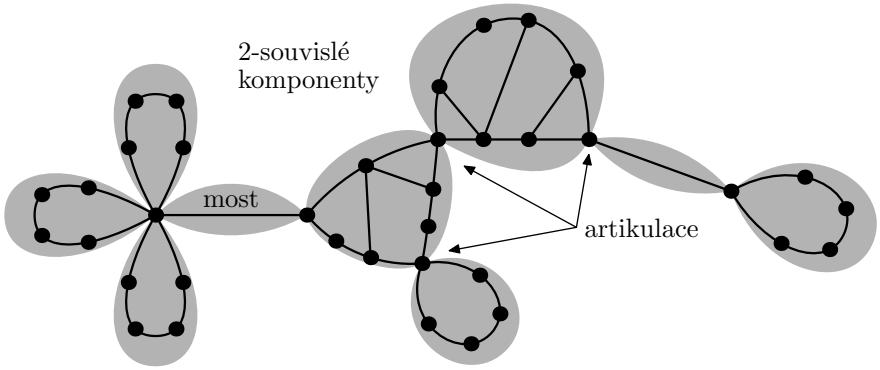
Motivace: Představme si situaci za války, kdy partyzáni znemožňovali postup vojsk tak, že vyhodili do povětří vhodný most a tím přerušili silniční spojení. Dostaneme graf silniční sítě a ptáme se, jestli v ní existuje vrchol (křižovatka) nebo hrana (most) taková, že se po jejím vyhození graf rozpadne na několik komponent.

Pokud se Vám zdá motivace příliš zastaralá, tak místo partyzánů představte teroristy a místo silniční sítě síť počítačovou.

Definice: Graf je 2-*souvislý*, pokud po vyhození libovolného vrcholu zůstane souvislý. Platí věta, že graf je 2-souvislý právě tehdy, když každé dva vrcholy leží na společné kružnici. 2-*souvislá komponenta* je maximální 2-souvislý podgraf. *Artikulace* je vrchol, po jehož vyhození se graf rozpadne na komponenty souvislosti.

Podobně *most* je hrana, po jejímž vyhození se graf rozpadne. Most je také 2-souvislá komponenta.

Na následujícím obrázku jsou 2-souvislé komponenty znázorněny šedými obláčky. Jedna artikulace může patřit do několika 2-souvislých komponent.



Úkol: Dostaneme graf a chtěli bychom najít všechny jeho artikulace, případně vypsat 2-souvislé komponenty.

Triviálním řešením je postupně zkusit vyhodit každý vrchol a testovat souvislost výsledného grafu. To by nám ale zabralo čas $\mathcal{O}(n(n+m))$. Podívejme se na lepší řešení pomocí DFS:

Pozorování 1 *v je artikulace $\iff$ v je kořen DFS stromu a má alespoň dva potomky nebo není kořen a má syna w takového, že z žádného jeho potomka (včetně w) nevede zpětná hrana do předchůdce v.*

Přípustná cesta je cesta vedoucí po stromových hranách směrem od kořene, která může končit jedním skokem zpět po zpětné hraně. Pro každý vrchol grafu spočítáme funkci $\texttt{LOW}(v) = \min\{\texttt{in}[w] \mid \text{z } v \text{ do } w \text{ vede přípustná cesta}\}$. Funkci $\texttt{LOW}(v)$ můžeme počítat už při průchodu do hloubky, protože při opuštění vrcholu v známe všechny potřebné hodnoty `in[w]`. Funkci spočítáme jako $\texttt{LOW}(v) = \min\{x, y, z\}$, kde $x = \texttt{in}[v]$, $y = \min\{\texttt{in}[w] \mid vw \text{ je zpětná hrana}\}$ a $z = \min\{\texttt{LOW}[w] \mid vw \text{ je stromová hrana}\}$.

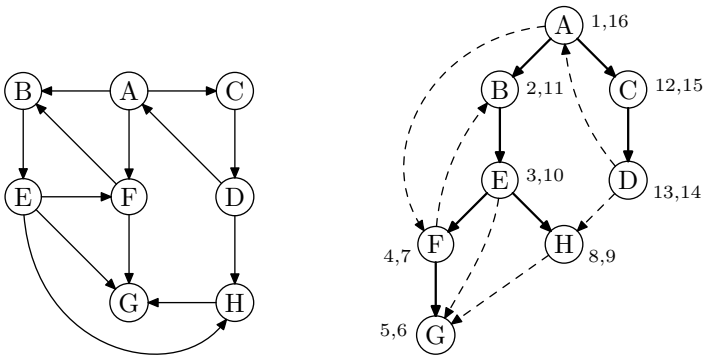
Předchozí pozorování můžeme přeformulovat pomocí funkce `LOW`.

Pozorování 2 *v je artikulace $\iff$ v je kořen DFS stromu a má alespoň dva potomky nebo není kořen a má syna w s* $\texttt{LOW}(w) \geq \texttt{in[v]}$

Na základě pozorování poznáme už při průchodu grafu do hloubky, které vrcholy jsou artikulace. Dodejme, že artikulace nemůžeme vypisovat rovnou, protože bychom mohli některé vypsat dvakrát. Musíme si u každého vrcholu pamatovat, jestli je artikulace, a výsledek vypsat až nakonec. Pokud bychom chtěli vypsat i hrany každé 2-souvislé komponenty, tak stačí během DFS ukládat procházené hrany na druhý zásobník. Při návratu do artikulace vypíšeme všechny hrany, které přibyly na zásobníku od posledního opuštění artikulace (stačí vhodně uříznout vršek zásobníku).

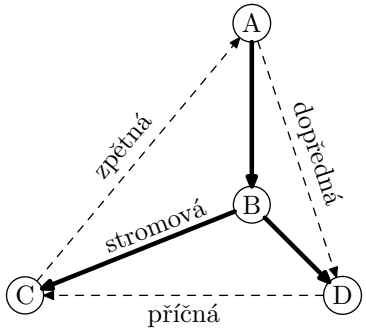
7.5 DFS na orientovaném grafu

Průchod do hloubky bude beze změny fungovat i pro orientované grafy. Na následujícím obrázku je nalevo orientovaný graf a napravo jeho strom průchodu do hloubky. Opět, pokud jsme během DFS měli na výběr z několika vrcholů, tak jsme si vybrali abecedně nejmenší vrchol.



V případě neorientovaných grafů jsme rozlišovali mezi stromovými a zpětnými hranami. U orientovaných grafů budeme muset rozlišit více případů nestromových hran. Typ hrany určíme ve chvíli, kdy hranu procházíme pomocí DFS a to podle barvy vrcholu, do kterého hrana vede, případně podle časů příchodu a odchodu z koncových vrcholů hrany.

- **stromové hrany** – vedou z právě prozkoumávaného vrcholu do nenavštíveného (bílého) vrcholu; tvoří DFS strom.
- **zpětné hrany** – vedou z právě prozkoumávaného vrcholu do šedivého vrcholu, tj. do předchůdce v DFS stromě.
- **dopředné hrany** – vedou z právě prozkoumávaného vrcholu do černého vrcholu v témže podstromě, tj. do svého potomka.
- **příčné hrany** – vedou z právě prozkoumávaného vrcholu do černého vrcholu v jiném podstromě, tj. mezi dvěmi různými podstromy téhož grafu.



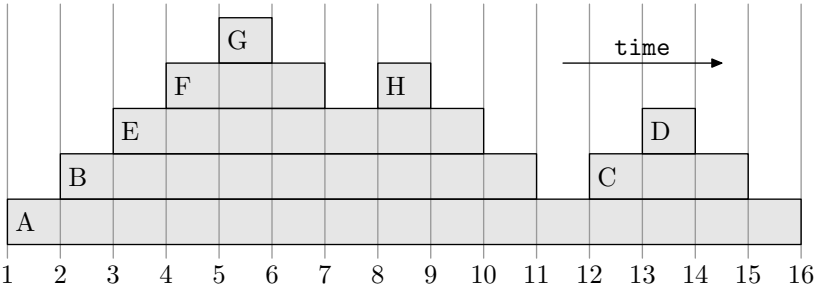
Hloubavý čtenář vidí, že barvy vrcholů ve skutečnosti nepotřebujeme a že jsme schopni barvu vrcholu určit pomocí hodnot $\texttt{in}[v]$, $\texttt{out}[v]$. Lepší charakterizaci hran při DFS dává následující pozorování.

Pozorování 3 *Intervaly $(in[v], out[v])$ pro všechny $v \in V$ tvoří dobré uzávorkování. To znamená, že dva intervaly jsou buď disjunktní a nebo je jeden podmnožinou druhého. Pro každou hranu $uv \in E$ platí jedna z následujících možností:*

- *uv je stromová nebo dopředná hrana $\iff (in[v], out[v]) \subseteq (in[u], out[u])$*
- *uv je zpětná hrana $\iff (in[u], out[u]) \subseteq (in[v], out[v])$*
- *uv je příčná hrana $\iff in[v] < out[v] < in[u] < out[u]$ (intervaly jsou disjunktní)*

Interval $(in[v], out[v])$ znázorňuje dobu, po kterou byl vrchol v uložen na zásobníku. Vlastnost korektního uzávorkování plyne přímo z toho, jak pracujeme se zásobníkem. Vrchol neopustíme a neodebereme ze zásobníku dříve, než jsou zpracováni všichni jeho potomci.

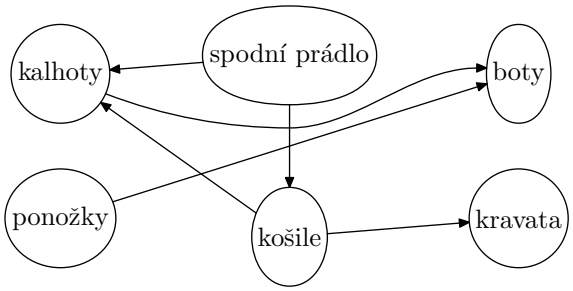
Průběh zaplnění zásobníku pro graf ze začátku sekce 7.5 je na následujícím obrázku. Stavu zásobníku odpovídá jen jeden sloupeček. Obrázek zachycuje stav zásobníku v různých časech. Vrchol X je vložen na zásobník v čase $\texttt{in}[X]$ a je ze zásobníku odebrán v čase $\texttt{out}[X]$.



Zkuste si rozmyslet, v jakém vztahu jsou intervaly $(in[u], out[u])$ a $(in[v], out[v])$, pokud je u následníkem v.

7.6 Topologické uspořádání

Motivace: Pomocí orientovaného grafu snadno znázorníme závislosti. Orientovanou hranou (šipkou) mezi úkoly vyjádříme, že druhý úkol můžeme začít provádět až po skončení prvního úkolu. Například když se chceme ráno obléknout, tak si ponožky musíme obléci dříve než boty. Dostaneme seznam úkolů a závislosti mezi nimi. Zajímalo by nás, v jakém pořadí úkoly vykonávat, abychom neporušili závislosti. Zkuste úlohu vyřešit pro následující orientovaný graf.

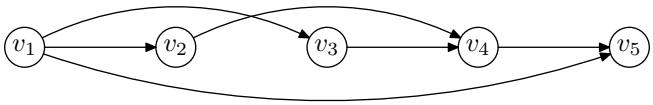


Jiným příkladem je uspořádání látky probírané na přednáškách tak, aby student nejprve slyšel všechny základy a potřebné ingredience a teprve pak si poslechl navazující výsledky.

Podobně funguje Makefile – soubor se závislostmi pro unixový program `make`. Často, když chceme ze vstupních souborů dostat požadované výstupy, musíme spustit více programů nebo utilit. Vstupem jednoho programu může být výstup z jiného programu. To určuje závislosti mezi použitými programy. Abychom dostali správný výstup, musíme programy pouštět ve správném pořadí. Závislosti napíšeme do souboru Makefile. Podle něj `Make` spustí všechny použité programy ve správném pořadí. Při dalším volání, `make` nespouští vše znova, ale podívá se, které vstupy i mezivýsledky se změnily. Potom spustí pouze ty programy, které přepočítají výstupy ke změněným vstupům. To je hlavní výhoda `Make`, která často výrazně zrychlí přepočítávání.

Proč se tomu říká topologické uspořádání? Šipky mezi vrcholy můžeme představit jako relaci '$\succeq$'. Potom hledáme lineární uspořádání '$\geq$', které je rozšířením '$\succeq$'. To znamená, že když $x \succeq y$, pak i $x \geq y$.

Definice: Topologické uspořádání vrcholů orientovaného grafu je seřazení vrcholů do řady v_1, v_2, v_3, ..., v_n tak, že každá hrana vede zleva doprava. Jinými slovy je to takové očíslování vrcholů čísly 1 až n takové, že každá hrana vede z vrcholu s nižším číslem do vrcholu s vyšším číslem.



Úkol: Pro zadaný orientovaný graf nalezněte topologické uspořádání. Přeskládávání vrcholů do topologického uspořádání se někdy říká *topologické třídění.*

Definice: *Orientovaný cyklus* je posloupnost $v_0 \to v_1 \to v_2 \to \cdots \to v_{n-1} \to v_n$ taková, že $v_0 = v_n$ a $v_{i-1}v_i$ je orientovaná hrana pro každé $i \in \{1, \ldots, n\}$. Orientovaným grafům, které neobsahují orientovaný cyklus, říkáme *orientované acyklické grafy.*[5]

Pozorování 4 *Graf G obsahuje orientovaný cyklus $\iff$ DFS najde zpětnou hranu.*

Důkaz: Jeden směr je jednoduchý, pokud je uv zpětná hrana, tak společně s cestou z v do u v DFS stromě tvoří cyklus. Na druhou stranu pokud graf obsahuje orientovaný cyklus, tak označme pomocí r první vrchol cyklu, který byl nalezen při průchodu DFS. Všechny ostatní vrcholy cyklu jsou potomky r, protože do nich z r vede orientovaná cesta. Proto je hrana cyklu, vedoucí do r, zpětná. ∎

Lemma 3 *Topologické uspořádání orientovaného acyklického grafu nalezneme pomocí DFS tak, že každý vrchol vypíšeme v momentě, kdy ho opouštíme. Na závěr výslednou posloupnost obrátíme. Jinými slovy uspořádání vrcholů podle klesajících časů* `out[`v`]` *je topologické.*

Důkaz: K důkazu předchozího tvrzení stačí ukázat, že je ve výsledném pořadí každá hrana zorientována správně. Nechť ij je libovolná orientovaná hrana. Podle klasifikace hran z pozorování 3 je hrana ij buď

- stromová nebo dopředná a potom $\texttt{out}[i] > \texttt{out}[j]$, nebo je
- příčná, ale pak také $\texttt{out}[i] > \texttt{out}[j]$. Hrana ij by ještě mohla být
- zpětná, to ale není podle předchozího pozorování možné, protože by graf obsahoval orientovaný cyklus. ∎

Rozpoznávání acykličnosti a hledání topologického uspořádání můžeme spojit do jednoho průchodu DFS. Budeme hledat topologické uspořádání pomocí průchodu do hloubky. Pokud objevíme zpětnou hranu, tak můžeme skončit a odpovědět, že topologické uspořádání neexistuje. V opačném případě topologické uspořádání najdeme. Časová složitost nalezení topologického uspořádání je $\mathcal{O}(n + m)$.

Poznamenejme na závěr, že topologické uspořádání je pro práci s acyklickými grafy (DAG, directed acyclic graph) stejně důležité, jako třídění pro práci s polem. Řada pěkných algoritmů v acyklických grafech vyžaduje topologicky uspořádané vrcholy. Aplikace najdte ve cvičeních na konci kapitoly.

Alternativním algoritmem pro nalezení topologického uspořádání může být postupné odtrhávání zdrojů (stoků) grafu a jejich vypisování na výstup. *Zdroj* je vrchol, do kterého nevede žádná hrana. Hrany z něj pouze vychází (vytékají). Naopak do *stoku* hrany pouze vedou a žádná hrana z něj nevede ven. Zkuste si rozmyslet, proč tento alternativní algoritmus funguje a jaká může být jeho nejrychlejší implementace.

Který algoritmus byste si vybrali pro řešení úlohy na papíře?

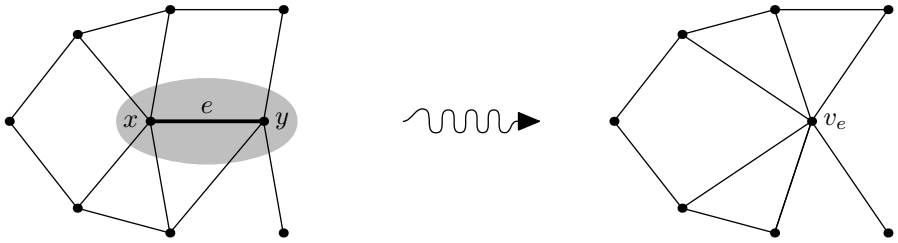
[5] Anglicky se označují jako DAG (Directed Acyclic Graph).

7.7 Intermezzo o kontrakcích

Začneme s kontrakcí hrany v neorientovaném grafu.

Nechť $e = xy$ je hrana grafu $G = (V, E)$. Kontrakci hrany e si můžeme představit tak, že se hrana e zkracuje a zkracuje, až už je tak krátká, že vrcholy x a y splynou v jeden vrchol. Pokud by vznikly násobné hrany nebo smyčky, tak je vyhodíme.

Přesněji řekneme, že graf $G.e$ vznikne kontrakcí hrany e do nového vrcholu v_e tak, že z grafu G vymažeme vrcholy x, y i s hranami, které z nich vedou, a naopak přidáme nový vrchol v_e, který spojíme se všemi zbylými vrcholy grafu G, se kterými byl spojen vrchol x nebo y. Formálněji $G.e = (V', E')$, kde $V' = V \setminus \{x, y\} \cup \{v_e\}$ a $E' = \{uv \in E \mid \{u, v\} \cap \{x, y\} = \emptyset\} \cup \{v_e w \mid xw \in E \setminus \{e\} \text{ nebo } yw \in E \setminus \{e\}\}$.



Nechť $W \subseteq V$ je podmnožina vrcholů taková, že $G[W]$ je souvislý podgraf.[6] Kontrakci množiny W do jednoho nového vrcholu v_W si opět můžeme představit tak, že všechny vrcholy W přibližujeme k sobě natolik, že splynou v jeden vrchol. Opět pokud by vznikly násobné hrany nebo smyčky, tak je vyhodíme.

Přesněji řekneme, že graf $G.W$ vznikne postupnou kontrakcí všech hran mezi vrcholy W. Kontrakce hrany $e = xy$ je v podstatě kontrakcí dvouprvkové množiny vrcholů $\{x, y\}$.

V orientovaných grafech kontrakce funguje stejně a zachovává směr hran.

7.8 Silně souvislé komponenty

Motivace: V jednom (raději) nejmenovaném městě, se starosta rozhodl, že ze všech ulic udělá jednosměrky. Odůvodnil to vznikem většího počtu parkovacích míst. Po jisté době se ale začalo proslýchat, že se z určitých míst nedá dojet do jiných míst ve městě jinak, něž porušením předpisů. Pokud by se to prokázalo, tak by s toho mohl mít starosta průšvih. Rád by to zkontroloval a případně napravil. Protože se rychle blíží volební období, tak už není moc času a nemůžete použít jiný algoritmus, než s lineární časovou složitostí. Znáte vhodný algoritmus, který ověří, že se dá dostat z kteréhokoliv místa ve městě kamkoliv?

Neorientovaný graf G je souvislý, pokud mezi každou dvojicí vrcholů existuje cesta. Jinými slovy, z každého vrcholu $u \in V(G)$ se dostaneme do libovolného jiného vrcholu $v \in V(G)$ po nějaké cestě. Graf G můžeme rozložit na komponenty souvislosti. To jsou největší „kusy“ (podgrafy) grafu G, které jsou souvislé.

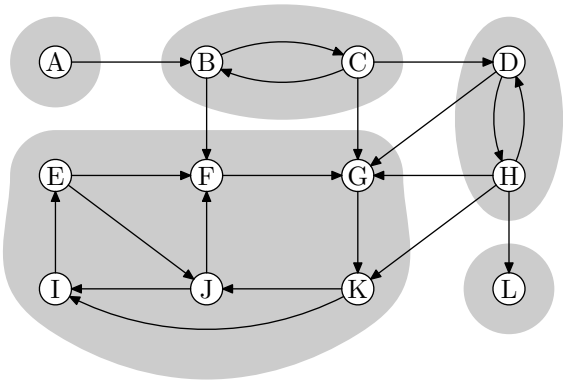
Ale co je to souvislost v orientovaném grafu? Po šipkách můžeme projít jen jedním směrem. Opět bychom chtěli zavést „souvislost“ tak, aby se bylo možné po šipkách dostat z libovolného vrcholu $u \in V(G)$ do libovolného vrcholu $v \in V(G)$. Tentokrát se ale může stát, že orientovaná cesta z u do v povede jinudy než cesta z v do u.

[6] Připomeňme, že $G[W]$ značí podgraf grafu G indukovaný vrcholy W.

Definice: Dva vrcholy $u, v \in V(G)$ jsou spolu *silně propojeny*, pokud existuje orientovaná cesta z u do v a i orientovaná cesta z v do u. Orientovaný graf G je *silně souvislý*, pokud je každá dvojice vrcholů $u, v \in V(G)$ silně propojena.

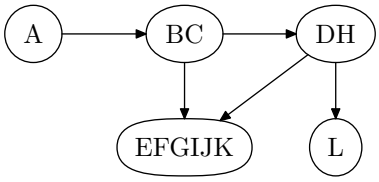
Relace být silně propojen je relace ekvivalence. Třídám této ekvivalence budeme říkat *silně souvislé komponenty.* Jinými slovy, silně souvislá komponenta je maximální podgraf grafu G, který je silně souvislý.

Příklad:



V grafu na obrázku máme 5 silně souvislých komponent.

Kontrakcí každé silně souvislé komponenty do jednoho meta-vrcholu dostaneme meta-graf.



Pozorování 5 *Meta-graf vzniklý kontrakcí silně souvislých komponent je acyklický orientovaný graf.*

Důkaz: Kdyby obsahoval orientovanou kružnici, tak všechny meta-vrcholy na kružnici tvoří jednu silně souvislou komponentu. To je spor s tím, že jsme všechny silně souvislé komponenty zkontrahovali do meta-vrcholů. ∎

Tato struktura orientovaných grafů nám umožňuje topologicky uspořádat silně souvislé komponenty. Silně souvislé komponentě, která odpovídá zdroji/stoku meta-grafu, budeme říkat zdrojová/stoková silně souvislá komponenta.

Při realizaci DFS jsme používali proceduru `Projdi(v)`. Ta projde právě všechny vrcholy, které jsou dostupné z vrcholu v. Pokud pustíme `Projdi(v)` na vrchol, který leží v silně souvislé komponentě odpovídající stoku meta-grafu, tak se dostaneme právě do všech vrcholů této silně souvislé komponenty.

Konkrétně na výše uvedeném příkladu: pokud v G pustíme `Projdi(E)` na vrchol E, tak projdeme právě vrcholy silně souvislé komponenty obsahující $\{E, F, G, I, J, K\}$. Nešlo by toho nějak využít? Ano, ale budeme muset vyřešit dva problémy.

(A) Jak najít vrchol, který určitě leží v silně souvislé komponentě, která je stokem?

(B) Jak pokračovat dále po té, co najdeme první silně souvislou komponentu?

Začneme s problémem (A). Vrchol, který je ve stokové silně souvislé komponentě se přímo najít nedá. Co se ale dá snadno najít, je vrchol, který leží ve zdrojové silně souvislé komponentě.

Pozorování 6 *Vrchol, který při DFS průchodu grafu G dostane největší opouštěcí čas* `out`$[\,\cdot\,]$*, leží ve zdrojové silně souvislé komponentě grafu G.*

Pozorování přímo plyne z následujícího obecnějšího pozorování.

Pozorování 7 *Nechť* `out`$[v]$ *jsou opouštěcí časy vrcholů při DFS průchodu grafu G. Pokud C_1 a C_2 jsou dvě silně souvislé komponenty takové, že z C_1 vede hrana do C_2, tak potom největší hodnota* `out`$[]$ *v první komponentě je větší než největší hodnota* `out`$[\,\cdot\,]$ *ve druhé komponentě.*

Důkaz: Mohou nastat dva případy. V prvním případě DFS navštíví komponentu C_2 jako první. Potom v ní ale zůstane, dokud ji celou neprozkoumá (to je vlastnost procedury `Projdi`()). Teprve pak se DFS dostane do C_2.

Ve druhém případě DFS nejprve navštíví C_1. Nechť v je vrchol, který byl v C_1 navštíven jako první. DFS opustí vrchol v, až když prozkoumá všechny vrcholy, které jsou z v dosažitelné a které nebyly dosud navštíveny. Proto nejprve projde celou komponentu C_2 a pak se teprve naposledy vrátí do v. ∎

Pozorování 7 vlastně říká, že sestupné uspořádání silně souvislých komponent podle jejich největšího čísla `out`$[\,\cdot\,]$ je topologické.

Pozorování 6 nám ukazuje, jak najít vrchol, který leží ve zdrojové silně souvislé komponentě. To je přesně naopak, než potřebujeme! Potřebujeme vrchol ležící ve stokové silně souvislé komponentě. Nevadí, vrchol budeme hledat v grafu G^R. To je v grafu, který vznikne z G tak, že obrátíme všechny hrany. Graf G^R má stejné silně souvislé komponenty jako graf G. Rozmyslete si proč.

Pustíme DFS na graf G^R. Vrchol v s největším `out`$[v]$ bude ležet ve stokové silně souvislé komponentě grafu G. Tím jsme vyřešili problém (A).

Zbývá vyřešit problém (B). Jak pokračovat po tom, co určíme stokovou silně souvislou komponentu? Znovu využijeme pozorování 7. Po té, co z grafu G vymažeme první nalezenou silně souvislou komponentu, tak vrchol w s největším `out`$[w]$ (ze zbylých vrcholů) patří do stokové silně souvislé komponenty zbytku grafu G. Pokud si budeme pamatovat hodnoty `out`$[\,\cdot\,]$, které jsme získali DFS průchodem grafu G^R, tak můžeme z G snadno odtrhnout druhou silně souvislou komponentu, třetí, čtvrtou a tak dál.

Celý algoritmus vypadá následovně:

1. Pusť `DFS` na graf G^R a ulož si posloupnost `out`$[\,\cdot\,]$ v klesajícím pořadí.

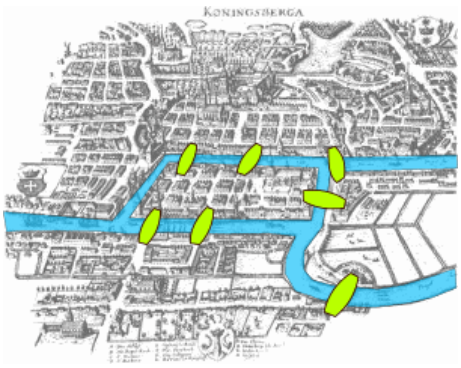
2. Pusť `DFS` na graf G, ve kterém procházíme vrcholy v pořadí podle klesajících hodnot `out`$[\,\cdot\,]$, a vypisuj nalezené komponenty souvislosti (jako v neorientovaném grafu).

Algoritmu běží v lineárním čase $\mathcal{O}(m+n)$. V podstatě je to dvakrát čas průchodu do hloubky (DFS). (Jak rychle vytvoříme reprezentaci grafu G^R? Vejdeme se do času $\mathcal{O}(m+n)$?)

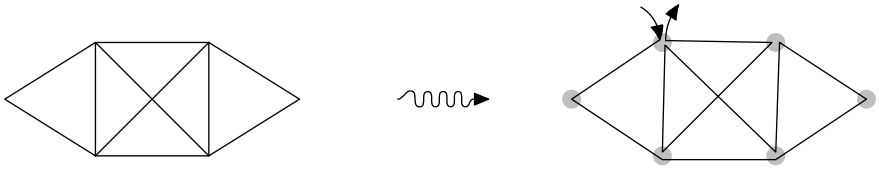
Zkusme algoritmus pustit na grafu G z příkladu. Pokud si DFS může vybrat z více vrcholů, tak si vybere ten abecedně menší. Pořadí vrcholů podle klesajících časů `out`$[\,\cdot\,]$ v průchodu G^R je L, F, G, K, J, I, E, H, D, B, C, A. Silně souvislé komponenty, které algoritmus nalezne, jsou postupně $\{L\}$, $\{E,F,G,I,J,K\}$, $\{D,H\}$, $\{B,C\}$, $\{A\}$.

7.9 Eulerovský tah

Motivace: Městem Královec protéká řeka Préva, na které leží sedm mostů. Stará mapa Královce je na vedlejším obrázku. Šlechtici 18. století už nevěděli coby a tak chtěli během večerní projížďky projet město v kočáře tak, aby přes každý most projeli právě jednou. Nevěděli jak na to a slibovali velkou odměnu tomu, kdo jim ukáže, jak to udělat. Nakonec přišel pan Euler, úlohu vyřešil a odpověděl, že to nejde. Odměnu samozřejmě nedostal.



Jinou motivací pro hledání Eulerovského tahu jsou jednotažky. Dostaneme obrázek a chceme ho nakreslit jedním tahem aniž bychom zvedli tužku z papíru. Známou jednotažkou je například domeček a nebo následující obrázek.



Sled je posloupnost v_0, e_0, v_1, e_1,... v_n taková, že $e_i = \{v_i, v_{i+1}\}$ (neboli každé 2 po sobě jdoucí hrany mají společný vrchol). Pokud se v posloupnosti neopakují hrany, tak ji nazveme *tah* a pokud se neopakují ani vrcholy, tak ji nazveme *cesta.* Uzavřený tah je tah, který začíná i končí ve stejném vrcholu. Uzavřený tah nazveme *cyklus. Uzavřený Eulerovský tah* je tah, který projde všechny hrany grafu a vrátí se do výchozího vrcholu. Graf, který obsahuje alespoň jeden uzavřený Eulerovský tah se nazývá Eulerovský.

Věta 6 *Souvislý graf G je Eulerovský $\iff$ všechny jeho vrcholy mají sudý stupeň.*

Jak najít Eulerovský tah? Na začátku pro jednoduchost předpokládejme, že má každý vrchol sudý stupeň. Pro nalezení uzavřeného Eulerovského tahu použijeme proceduru `Euler(`v`)`. Procedura vypíše vrcholy v pořadí, jak po sobě následují v uzavřeném Eulerovském tahu (vrchol, kterým tah projde k-krát, se na výstupu objeví také k-krát). Procedura `Euler()` funguje hodně podobně jako DFS. Tentokrát můžeme vrcholem projít několikrát, ale každou hranou stále jen jednou.

```
Euler(v):
    while ∃ neprošlá hrana vw do
        označ hranu vw
        Euler(w)
        vypiš(v)
```

Začneme ve vrcholu v a projdeme po libovolné hraně. Každou hranu, po které procházíme, označíme. Vždy když přijdeme do vrcholu různého od v, tak z něj můžeme odejít po neoznačené hraně. To platí proto, že všechny vrcholy mají sudý stupeň a každý průchod přes vrchol označí právě dvě hrany (po jedné jsme přišli a po druhé odešli). Jedinou výjimkou je výchozí vrchol v, protože má navíc označenu hranu, na které jsme tah začali. Po příchodu do výchozího vrcholu se napojíme na začátek tahu a dostaneme cyklus.

Pokud tento cyklus z grafu vyhodíme, dostaneme komponenty souvislosti, které jsou zase Eulerovské (z každého vrcholu jsme vyhodili sudý počet hran). Ve skutečnosti hrany nevyhazujeme, jen je označujeme za prošlé. V neoznačených hranách najdeme stejným způsobem další cyklus. Opakováním postupu nacházíme cykly tak dlouho, dokud jsou v grafu neoznačené hrany.

Slepením nalezených cyklů dohromady dostaneme uzavřený Eulerovský tah. Lepení probíhá podobně, jako když si na výletě s naplánovanou trasou vyrazíte na neplánovaný vyhlídkový okruh. Na chvíli necháte průchodu po naplánované trase, případně odložíte batohy, a vyrazíte na vyhlídkový okruh. Když se vrátíte k batohům, tak pokračujete dále po naplánované trase. Toto lepení za nás v proceduře `Euler(`v`)` provede rekurze.

Jak přesně ta rekurze funguje? Nejprve udělá plán a projde první cyklus (vrcholy naplánovaného tahu jsou uloženy na zásobníku rekurze a hrany cyklu jsou označeny za prošlé). Potom se rekurze začne vracet a vypisovat tah. Když se vrátí do vrcholu, odkud vedou neoznačené hrany, zavolá kamarádku rekurzi, aby zajistila vypsání "výletního" okruhu. Až se kamarádka rekurze vrátí, vypíše se vrchol a pokračuje se v návratu z rekurze.

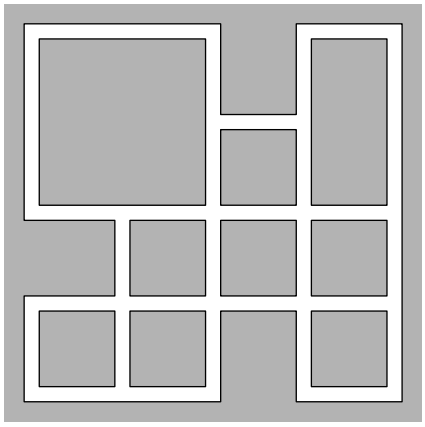
Kdybychom dopředu nevěděli, jestli má každý vrchol grafu sudý stupeň, tak to můžeme během algoritmu snadno zjistit. Pokud během algoritmu přijdeme do vrcholu w (jiného než výchozího vrchol) a už z něj nemůžeme pokračovat dále (všechny hrany vedoucí z vrcholu už jsou označené), tak odpovíme, že w má lichý stupeň a tím pádem víme, že uzavřený Eulerovský tah neexistuje.

Poznámka: K tomu, abychom nakreslili jednotažku jedním tahem, aniž bychom zvedli tužku z papíru, nepotřebujeme uzavřený Eulerovský tah. Stačí otevřený Eulerovský tah. *Otevřený Eulerovský tah* je tah, který projde všechny hrany grafu. Neklademe si žádnou podmínku na to, aby tah skončil ve výchozím vrcholu. Může skončit jinde než začal. Platí věta, která říká, že v souvislém grafu existuje otevřený Eulerovský tah právě tehdy, když všechny vrcholy, až na dva, mají sudý stupeň. Eulerovský tah v jednom lichém vrcholu začne a ve druhém skončí.

Jak by vypadal algoritmus pro nalezení otevřeného Eulerovského tahu?

7.9.1 Pošťákův problém

Motivace: (Problém kropicího vozu) V jednom městě se vzorně starají o své občany. Každý den kropí a zametají všechny ulice, které ve městě mají. Protože je to vzorné město, tak se radní starají i o úsporný rozpočet města. Proto chtějí najít pro řidiče zametacího vozu takovou trasu, aby pokropil a zametl všechny ulice města, ale najezdil co nejméně kilometrů.
Nalezení optimální trasy by městu opravdu pomohlo, protože by ušetřili i při svozu odpadu, rozvozu informačních letáků či volebních lístů. Optimální trasu by ocenila i místní pošta.



Problém se v literatuře označuje jako *pošťákův problém*, protože se problém poprvé studoval v souvisloti s roznášením pošty.[7]

[7] Někdy se mu říká problém čínského pošťáka, protože ho prvně studoval čínský matematik Mei-Ku Kuan v roce 1962.

Úkol: (Pošťákův problém) Dostanete souvislý graf $G = (V, E)$. Nalezněte nejkratší sled, který obsahuje všechny hrany.

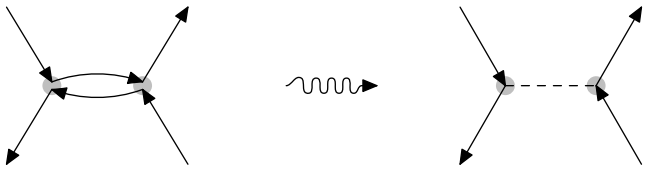
Připomeňme, že *sled* je posloupnost v_0, e_0, v_1, e_1,... v_n taková, že $e_i = \{v_i, v_{i+1}\}$ (neboli každé 2 po sobě jdoucí hrany mají společný vrchol). Sled, který je optimálním řešením úlohy nazveme *pošťákův sled.*

Pokud by v grafu G existoval uzavřený Eulerovský tah, tak je optimálním řešením úlohy. Eulerovský tah projde všechny hrany a každou hranu právě jednou. V takovém případě je řešení jednoduché.

Pokud v grafu neexistuje uzavřený Eulerovský tah, tak budeme muset projít některé hrany vícekrát. Souvislé grafy, které neobsahují uzavřený Eulerovský tah obsahují vrcholy lichého stupně. Nechť $T \subseteq V$ je množina vrcholů lichého stupně. $|T|$ je sudé, protože součet stupňů všech vrcholů je sudé číslo (každá hrana přispěje do tohoto součtu dvojkou).

Lemma 4 *Nechť S je sled, který je optimálním řešením úlohy. Každá hrana grafu je v S použita nejvýše dvakrát.*

Pokud by byla některá hrana uv sledu S použita aspoň třikrát, tak můžeme sled S zkrátit. Na následujícím obrázku je naznačeno zkrácení sledu v případě, kdy po hraně uv vedou 2 průchodu sledu S jdoucí proti sobě (hrana uv zůstane obsažena ve sledu S díky třetímu průchodu). Ještě si rozmyslete možnost, že by sled procházel hranu uv třikrát ve stejném směru.



Lemma 5 *Nechť $H \subseteq G$ je podgraf, který obsahuje právě ty hrany, které optimální sled S projde dvakrát. Vrcholy lichého stupně H jsou právě vrcholy T. Pograf H neobsahuje cykly.*

V multigrafu, který obsahuje každou hranu tolikkrát, kolikkrát ji projde sled S, má každý vrchol sudý stupeň (sled je uzavřený a proto pokaždé když vstoupí do vrcholu, tak z něj i odejde). Po vyhození hran $E(G)$ z multigrafu nám zůstane právě graf H. Lichý vrchol $v \in H$ mohl vzniknout pouze tak, že jsme vyhodili lichý počet hran vedoucích z v. To ukazuje, že liché vrcholy H jednoznačně odpovídají lichým vrcholům v G.

Pokud H obsahuje cyklus, tak ho z H vyhodíme a dostaneme graf H'. V multigrafu, který vznikne součtem G a H', už má každý vrchol sudý stupeň. Proto v něm existuje uzavřený Eulerovský tah, který je také řešením úlohy, a prochází méně hran než sled S. To je spor s tím, že S je optimální sled.

Důsledkem předchozího lemmatu už konečně dostaneme charakterizaci grafu H.

Lemma 6 *H se skládá z $|T|/2$ hranově disjunktních cest, jejichž konce leží v T.*

Teď už máme skoro všechna pozorování potřebná k tomu, abychom úlohu vyřešili. Poslední ingrediencí je algoritmus na maximální párování. Párování M v grafu G je množina vrcholově disjunktních hran (žádné dvě hrany z M nemohou sdílet vrchol).

Algoritmus na maximální párování si zde nevysvětlíme, protože jeho popis by vydal na samostatnou kapitolu. Časová složitost algoritmu je polynomiální. Popis

algoritmu na maximální párování v bipartitním grafu najdete u aplikací toků v sítích v sekci 12.5. Popis algoritmu pro obecné grafy čtenář nalezne například v [8].

Algoritmus řešící Pošťákův problém:

1. Nalezneme vrcholy T, které mají lichý stupeň v G. Chceme najít $|T|/2$ cest, které spárují vrcholy T a budou obsahovat co nejméně hran. Provedeme to následovně. Vytvoříme pomocný graf G' na vrcholech T. Každé dva vrcholy $u, v \in T$ spojíme hranou uv, kterou odhodnotíme délkou nejkratší cesty mezi u a v. V grafu G' nalezneme maximální párování minimální ceny. Nalezené párování odpovídá hledaným cestám. Ty tvoří optimální graf H.

2. V multigrafu $G + H$ najdeme uzavřený Eulerovský tah. Ten odpovídá sledu, který je řešením úlohy.

Poznámka: V reálných úlohách chceme skutečně najezdit co nejméně. Každá silnice má svojí délku $d(e)$. Chceme najít nejkratší sled, který projde všechny hrany. Délka sledu je součet délek hran, které prochází.

Algoritmus řešící Pošťákův problém v ohodnocených grafech vypadá skoro stejně jako pro neohodnocené grafy, jenom místo maximální párování hledáme maximální párování minimální ceny (cena párování M je součet cen všech hran patřících do M). Maximální párování minimální ceny se dá najít v polynomiálním čase a proto umíme ve stejném čase vyřešit i Pošťákův problém v ohodnocených grafech. Více informací čtenář najde například v [8].

Poznámka: Pošťákův problém je orientovaných grafech o něco jednoduší, protože ho můžeme vyřešit pomocí toků v sítích. Viz cvičení 12 na konci sekce 12.6.4.

7.10 BFS, hledání nejkratší cesty

Druhou implementací efektivního průchodu grafu je průchod do šířky (BFS, neboli Bread First Search). Jeho výhodou o proti DFS je, že kromě samotného řešení nalezne i nejkratší cestu, která k řešení vede.

Prohledávání začneme ve vrcholu s a projdeme celou komponentu souvislosti, do které s patří. Délku cesty měříme počtem hran na cestě. Proměnná H[v] bude obsahovat délku nejkratší cesty z s do v. Na začátku bude obsahovat hodnotu -1. V momentě, kdy se při průchodu grafu dostaneme do vrcholu v, nastavíme proměnnou H[v] na správnou hodnotu, což bude číslo větší nebo rovno nule. Proto podle hodnoty H[v] poznáme, jestli jsme už vrcholem v někdy prošli a nebo ještě ne.

```
BFS(s):
   ∀v ∈ V : H[v]:=−1
   H[s]:=0 a přidej s do fronty
   while fronta neprázdná do
      odeber vrchol v z fronty
      for all sousedy w vrcholu v do
         if H[w]=−1 then
            přidej w do fronty
            H[w]:=H[v]+1
```

Pozorování 8 *Nejprve zpracujeme všechny vrcholy se stejným číslem* H[v] *a pak teprve začneme zpracovávat vrcholy s číslem o jedna větším. V průběhu algoritmu jsou ve frontě jen vrcholy, které mají* H[v] *rovno d nebo $d+1$, pro nějaké d.*

Díky tomu, že zpracování vrcholů probíhá po vlnách podle čísla H[v], se algoritmu říká *algoritmus vlny.*

Pozorování 9 H[v] *obsahuje délku nejkratší cesty z s do v.*

Pozorování dokážeme indukcí podle hodnoty H[v]. Vrchol s má H[s]=0 a nejkratší cesta z s do s má rovněž délku nula. Předpokládejme, že pro všechny vrcholy s hodnotou H[v] $\leq k$ je H[v] délka nejkratší cesty z s do v. Potřebujeme ukázat indukční krok a to je, že každý vrchol v s H[v] $= k+1$ leží ve vzdálenosti $k+1$ od s. Pokud ne, tak existuje kratší cesta z s do v a nechť wv je poslední hrana na této cestě. H[w] $< k$. Potom se ale měl algoritmus při zpracovávání vrcholu w podívat na hranu wv a nastavit H[v] na H[w] $+1$. Hodnota H[w] $+1 < k+1$, což je spor.

Podobně jako v DFS tvoří stromové hrany DFS strom, tak i hrany, po kterých při BFS projdeme do nenavštíveného vrcholu, tvoří *BFS strom* (strom průchodu do šířky).

Jednotlivé „vlny“ nazveme *vrstvy* průchodu do šířky. V i-té vrstvě jsou obsaženy právě vrcholy ve vzdálenosti i od počátečního vrcholu s.

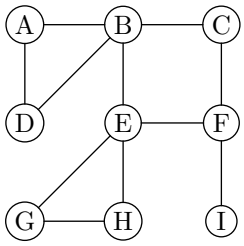
Příklad: Na kolik nejméně skoků doskáčeme šachovým koněm z A1 na A2? Úlohu vyřešíme pomocí průchodu do šířky (algoritmu vlny). Řešit ji můžeme přímo na papíře tak, že budeme do prázdných políček psát na kolik skoků se do nich dostaneme. Začneme na A1, kam napíšeme nulu. Do políček, kam doskočíme z A1 napíšeme jedničky. Ve druhé vlně napíšeme dvojku do všech políček, kam doskočíme z políček s jedničkou, atd. (do popsaných políček už nic nepíšeme). Skončíme ve chvíli, kdy se dostaneme do A2. Číslo napsané na políčku A2 udává nejmenší možný počet skoků.

	A	B	C	D	E	F	G	H
1	0	3	2	3	2	3		
2	3		1	2	3		3	
3	2	1		3	2	3		
4	3	2	3	2	3		3	
5	2	3	2	3		3		
6	3		3		3			
7		3		3				
8								

7.11 Příklady

7.11.1 Přímé procvičení vyložených algoritmů

1. Graf na následujícím obrázku projděte pomocí průchodu do hloubky (DFS). Pokud si v některém kroku můžete vybrat z několika vrcholů, tak si vždy vyberte ten abecedně nejmenší z nich.



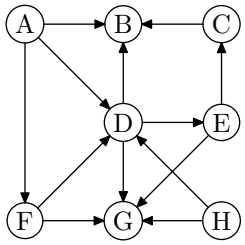
   (a) Rozdělte hrany na stromové a zpětné.

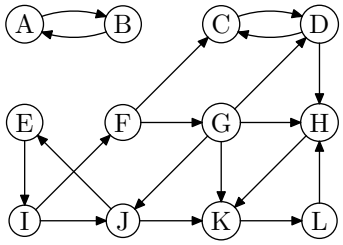
   (b) U každého vrcholu v spočítejte časy příchodu a odchodu – hodnoty `in`$[v]$ a `out`$[v]$.

   (c) Najděte 2-souvislé komponenty grafu.

   (d) Pro každý vrchol v spočítejte hodnotu `LOW`$[v]$.

2. Oba orientované grafy na následujících obrázcích projděte pomocí průchodu do hloubky (DFS). Pokud si v některém kroku můžete vybrat z několika vrcholů, tak si vždy vyberte ten abecedně nejmenší z nich.





   (a) Rozdělte hrany na stromové a zpětné, příčné a dopředné.

   (b) U každého vrcholu v spočítejte časy příchodu a odchodu – hodnoty `in`$[v]$ a `out`$[v]$.

   (c) Najděte topologické uspořádání každého z grafů.

   (d) Vyznačte, které vrcholy jsou zdroje a které stoky.

   (e) Najděte silně souvislé komponenty každého z grafů.

   (f) Vyznačte, které silně souvislé komponenty jsou zdrojové a které stokové.

7.11.2 Průchod grafu do šířky

1. (Cesta tunelem) Rodina tvořená tatínkem, maminkou, dcerou a synem chce projít tunelem. Tatínek projde tunelem za jednu minutu, maminka za dvě, syn za čtyři a dcera za pět. Problém je v tom, že v tunelu je hrozná tma a jejich svíčka vydrží hořet pouze dvanáct minut. Úzkým tunelem mohou procházet naráz nejvýše dva lidé a v žádném případě nesmí jít tunelem nikdo bez svíčky. Poradíte rodině, jak mají tunelem projít?

2. (Přelévání tekutin) Máme 3 nádoby o různých objemech. Žádná nádoba na sobě nemá stupnici a nádoby mají dokonce tak roztodivné tvary, že nám znemožňují odhadování množství vody v nich. Stále ale může naměřit i jiné množství tekutiny. Můžeme přelévat obsah jedné nádoby do druhé tak dlouho, dokud nepřelijeme všechno nebo dokud se druhá nádoba nezaplní. Jaký je nejmenší počet přelití, abychom vyřešili následující úlohy? Ve všech variantách je na počátku největší nádoba plná.

   (a) Máme 3 nádoby o objemech 7l, 5l a 3l. Chceme naměřit 1 litr.

   (b) Máme 3 nádoby o objemech 7l, 4l a 3l. Přeléváním se chceme dostat do stavu, kdy je v jedné nádobě 3l a ve zbylých dvou po 2 litrech.

   (c) Máme 3 nádoby o objemech 8l, 5l a 3l. Přeléváním se chceme dostat do stavu, kdy je počáteční množství rozděleno na dva stejné díly.

3. (Nejkratší cesta)

   (a) V neohodnoceném grafu G najděte nejkratší cestu z vrcholu s (start) do vrcholu c (cíl).

   (b) Všechny hrany v grafu G mají délku 1 nebo 2. Najděte nejkratší cestu z vrcholu s (start) do vrcholu c (cíl).

   (c) Dokázali byste řešení zobecnit i pro grafy s ohodnocením hran 1 až k? Zajímá nás lepší řešení, než pomocí Dijkstrova algoritmu. Chtěli bychom najít řešení v čase $\mathcal{O}(km + n)$.

4. (Cesta kulhavého koně) Náš šachový kůň kulhá. To znamená, že v sudých tazích provede krok jako šachový kůň a v lichých tazích provede krok jako šachový král. Zjistěte, na kolik nejméně kroků se dostane kulhavý kůň z políčka A1 na políčko H8. Až to zjistíte, tak úlohu vyřešte obecně pro libovolné počáteční a cílové políčko.

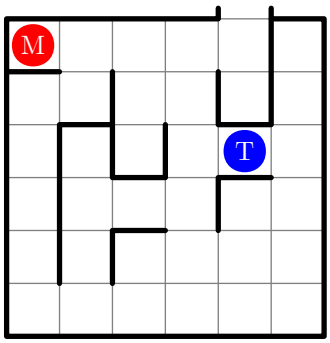
5. (Bludiště) Pomocí matice velikosti $n \times m$ máte zadáno bludiště. X značí zeď a nula volný prostor. Na okraji matice jsou všude zdi, takže jde o uzavřený systém místností a chodeb. V matici se vyskytují i znaky $=poklad a S=místo, kde se zrovna nacházíte. Zjistěte, zda se můžete dostat k pokladu. Předpokládejte, že nejste bílá paní, která by mohla procházet přes zdi.

X	X	X	X	X	X	X	X	X
X	S	0	0	X	0	0	0	X
X	0	X	0	X	0	X	0	X
X	0	0	0	0	0	X	0	X
X	0	0	0	X	0	X	$	X
X	X	X	X	X	X	X	X	X

6. (Bludiště se dveřmi) Máte zadané bludiště jako v předchozí úloze. Navíc se ale v bludišti vyskytují znaky D jako dveře. Dveře jsou ocelové a dají se otevřít pouze dynamitem. Nejste bílá paní, ale jste lupiči, co se chtějí vloupat do sejfu a vykrást banku. Podařilo se vám ukořistit plán sejfu. Pro hladký a bezpečný průběh akce se chcete dostat k penězům na co nejmenší počet kroků, ale hlavně tak, abyste museli odbouchnout co nejméně dveří. Jednou odbouchnutí dveří trvá nesrovnatelně déle než libovolný počet kroků v sejfu. Navíc nadělá ohromný hluk. Vaším úkolem je najít pro lupiče posloupnost pohybů vlevo, vpravo, nahoru, dolů tak, aby se dostali co nejrychleji k penězům. Lupiči už sami pochopí, že když mají projít dveřmi, tak je mají odprásknout.

7. (Průjezd městem) Máte zadaný plán města podobně jako v předchozích úlohách, tj. pomocí matice. Tentokrát jsou v něm pouze silnice široké jeden čtvereček a křižovatky. Vaším úkolem je najít cestu ze startu do cíle (čtverečky označené S a C). Zdá se vám to jednoduché? Tak zkuste najít cestu ze startu do cíle s dodržováním místních dopravních předpisů. V tomto městě je zakázáno na libovolné křižovatce odbočovat vpravo. Vpravo se ale můžete dostat například tak, že projedete rovně přes křižovatku a objedete jeden blok (třikrát zatočíte vlevo).

8. (Theseus a Minotaurus) Zahrajeme si hru ve dvourozměrném bludišti $n \times m$ polí. V bludišti se (samozřejmě kromě zdí, ty se vyskytují v každém pořádném bludišti) nachází Theseus a Minotaurus. Vy budete ovládat Thesea a budete se snažit dostat z bludiště ven, čili dostat se na hranici bludiště a následujícím krokem z bludiště utéct. Ovšem nikdy nesmíte narazit na Minotaura, sic bídně zhynete. Na Minotaura narazíte, pokud s ním sdílíte políčko.



Jeden tah probíhá následovně: nejprve se hýbe Minotaurus a táhne k-krát následujícím způsobem. Pokud nejsou postavy Minotaura a Thesea ve stejném sloupci, chce se Minotaurus pohnout o jedno políčko vlevo nebo vpravo, aby se Theseovi přiblížil. Pokud mu v tom nebrání zeď, skutečně se tam posune. Pokud nejsou obě postavy na stejném řádku bludiště, chce se Minotaurus pohnout nahoru či dolů opět směrem k Theseovi. Opět se na zvolené políčko Minotaurus přesune jen tehdy, není-li tam zeď. V jednom kroku provádí Minotaurus tyto pohyby v zadaném pořadí a může provést oba dva, čili se může dostat na jedno z okolních osmi políček. Po k takovýchto krocích Minotaura se hýbe Theseus, a to na jedno ze čtyř okolních volných polí. Takto se oba střídají na tahu, dokud buď Theseus neuteče z bludiště, nebo dokud Minotaurus nedohoní Thesea. Vaším úkolem bude najít pro Thesea nejkratší posloupnost pohybů vlevo, vpravo, nahoru, dolů, aby se dostal bezpečně z bludiště, případně říci, že to není možné.

9. (Loydova devítka) Napište program, který zjistí, jestli jde Loydova devítka poskládat a řekne vám, jak šoupat čtverečky tak, abyste ji poskládali na nejmenší počet tahů.

7.11.3 Průchod grafu do hloubky

1. (Obrácení hran) Orientovaný graf $G^R = (V, E^R)$ vznikne z orientovaného grafu $G = (V, E)$ obrácením hran. Tedy $E^R = \{xy \mid yx \in E\}$. Navrhněte algoritmus, který spočítá reprezentaci grafu G^R v lineárním čase $\mathcal{O}(n+m)$. Úlohu řešte pro reprezentaci seznamem sousedů. Jak by to bylo pro reprezentaci maticí incidence?

2. (Vstupní a výstupní stupně) Dostaneme orientovaný graf $G = (V, E)$ reprezentovaný seznamem sousedů. Stupeň vrcholu v (značíme $deg\, v$) je počet hran grafu G, které do vrcholu v vedou a nebo které z vrcholu v vychází. Vstupní stupeň vrcholu v (značíme $deg^+ v$) je počet hran grafu G, které do vrcholu v vstupují. Výstupní stupeň vrcholu v (značíme $deg^- v$) je počet hran grafu G, které z vrcholu v vychází. Platí $deg^+ v + deg^- v = deg\, v$.

   (a) Dokažte, že $\sum_{v \in V} deg\, v = 2|E|$.

(b) S využitím části (a) dokažte, že počet vrcholů s lichým stupněm musí být sudé číslo.

(c) Navrhněte algoritmus, který v lineárním čase spočítá vstupní a výstupní stupně všech vrcholů.

3. (Čtverce stupňů sousedů) Dostaneme neorientovaný graf $G = (V, E)$ reprezentovaný seznamem sousedů. Definujeme $\texttt{dvadeg}[v] := \sum_{u \in \texttt{Sousedi}[v]} (deg\, v)^2$. Navrhněte lineární algoritmus, který pro všechny vrcholy spočítá $\texttt{dvadeg}[\,\cdot\,]$ v čase $\mathcal{O}(n + m)$.

4. (Bipartitní graf) Graf $G = (V, E)$ je *bipartitní,* pokud jeho vrcholy můžeme rozdělit na dvě disjunktní množiny V_1, V_2 (tedy $V_1 \cup V_2 = V$ a $V_1 \cap V_2 = \emptyset$) tak, aby všechny hrany vedly z jedné množiny do druhé. (Graf nemůže obsahovat hranu s oběma konci ve stejné množině.)

   (a) Navrhněte lineární algoritmus, který dostane graf G a zjistí, jestli je bipartitní.

   (b) Navrhněte algoritmus, který v lineárním čase zjistí, jestli graf obsahuje lichý cyklus (cyklus liché délky).

   (c) Pokud graf neobsahuje lichý cyklus, dokážete ho obarvit 2 barvami? Obarvení grafu 2 barvami je obarvení jeho vrcholů 2 barvami takové, že každé dva vrcholy spojené hranou mají různou barvu. Je každý takový graf bipartitní?

5. (Vyhodnocení stromu) Dostanete binární zakořeněný strom, jehož každý vrchol v obsahuje číslo $\texttt{z}[v]$. Hodnota $\texttt{zmax}[v]$ je definovaná jako maximum z hodnot $\texttt{z}[w]$, pro všechny potomky w vrcholu v. Spočítejte celé pole $\texttt{zmax}[\,\cdot\,]$ v lineárním čase.

6. (Následník ve stromě) Dostaneme zakořeněný strom. Vrchol u je následníkem vrcholu v, pokud v leží na cestě z kořene do u. Postupně budete dostávat dvojice vrcholů x, y a chtěli bychom co nejrychleji odpovídat na otázku, jestli je x následníkem y?

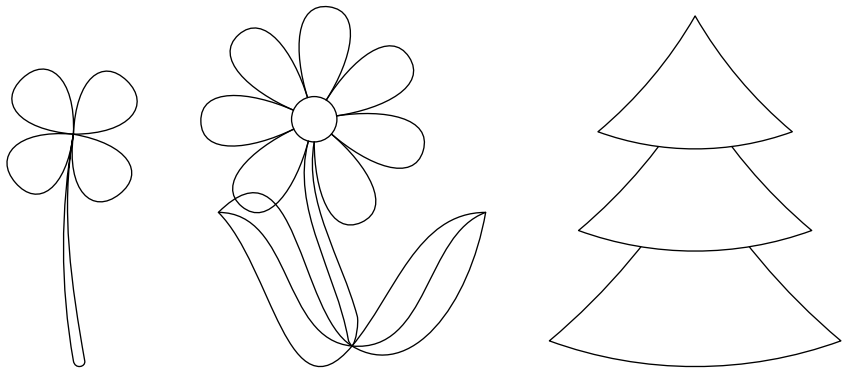
   Pokud si můžete dovolit předzpracování v čase $\mathcal{O}(n)$, zvládnete odpovídat v konstantním čase?

7. (Eulerovský tah) Tah v grafu G je posloupnost v_0, e_0, v_1, e_1,... v_n taková, že $e_i = \{v_i, v_{i+1}\}$ a $v_i \neq v_j$ pro všechna i, j (neboli každé 2 po sobě jdoucí hrany mají společný vrchol a vrcholy se v posloupnosti neopakují). *Uzavřený Eulerovký tah* je tah, který projde všechny hrany grafu a skončí ve stejném vrcholu, ve kterém začal. *Otevřený Eulerovský tah* projde všechny hrany grafu, ale může končit na jiném místě, než začal.

   (a) Navrhněte algoritmus, který v grafu G najde uzavřený Eulerovský tah a vypíše ho. Případně odpovězte, že takový tah neexistuje.

   (b) Navrhněte algoritmus, který v grafu G najde otevřený Eulerovský tah a vypíše ho. Případně odpovězte, že takový tah neexistuje.

   (c) Následující rostlinky nakreslete jedním tahem. Křížení čar považujte za vrcholy.



8. Navrhněte algoritmus, který v grafu G najde libovolnou kostru, případně odpovězte, že neexistuje. Zkuste vymyslet velice jednoduché řešení pracující v čase $\mathcal{O}(n + m)$.

9. Navrhněte algoritmus, který pro graf G vypíše jeho komponenty souvislosti.

10. Navrhněte algoritmus, který pro graf G vypíše jeho 2-souvislé komponenty.

11. Navrhněte algoritmus, který pro orientovaný graf G vypíše jeho silně souvislé komponenty.

12. (Topologické uspořádání) V poušti na Sahaře byly nalezeny zbytky knihovny nějaké dávné civilizace. Civilizace znala písmo a předpokládá se, že měla i abecedu (uspořádání písmenek). Vědci se domnívají, že nalezené spisy byly v knihovně seřazeny lexikograficky.[8] Dostanete názvy spisů, které se zachovaly a to v pořadí jak byly poskládány na poličce. Názvy spisů jsou přepsané tak, že si každý znak označíme jedním písmenkem naší abecedy.

    Například pro českou knihovnu byste dostali seznam: „Alenka v říši divů“, „Alexandr Veliký“, „Baron Prášil“, „Broučci“ „Malý princ“, „Medvídek Pů“. Ze seznamu můžete vyvodit, že v české abecedě je ‘A’ před ‘B’, ale také ‘n’ před ‘x’.

    Podle názvu spisů, které dostanete, zkuste potvrdit nebo vyvrátit teorii o tom, že saharská civilizace měla abecedu. Pokud teorii potvrdíte, tak vypište jednu možnost, jak mohla jejich abeceda vypadat.

13. (Poznámka k topologickému uspořádání) Navrhněte algoritmus, který dostane neorientovaný graf, a zjistí, jestli graf obsahuje kružnici. Zkuste vymyslet řešení, které běží v čase $\mathcal{O}(n)$ (tedy je nezávislé na počtu hran).

14. (Topologické uspořádání) Dokažte nebo vyvraťte: Pokud orientovaný graf obsahuje orientované cykly, tak procedura pro topologické uspořádání nalezne uspořádání vrcholů, které minimalizuje počet „špatných“ hran, vedoucích zprava do leva.

15. (Polosouvislé grafy) Orientovaný graf $G = (V, E)$ je polosouvislý, pokud pro každé dva vrcholy $u, v \in V$ existuje orientovaná cesta z u do v nebo orientovaná cesta z v do u. Navrhněte algoritmus, který zjistí, jestli je graf G polosouvislý. Dokažte jeho správnost a určete jeho časovou složitost.

16. (Jednoznačně souvislé grafy) Orientovaný graf $G = (V, E)$ je jednoznačně souvislý, pokud pro každé dva vrcholy $u, v \in V$ existuje právě jedna orientovaná

[8] To je tak, jak řadíme slova ve slovníku. Česky bychom místo lexikografické uspořádání mohli říci slovníkové uspořádání.

cesta z u do v a právě jedna orientovaná cesta z v do u. Navrhněte algoritmus, který zjistí, jestli je graf G jednoznačně souvislý. Dokažte jeho správnost a určete jeho časovou složitost.

17. (Cyklus obsahující hranu e) Dostanete neorientovaný graf G s vyznačenou hranou e. Navrhněte algoritmus pracující v lineárním čase, který zjistí, jestli v grafu G existuje cyklus obsahující hranu e.

18. Navrhněte efektivní algoritmus, který dostane orientovaný graf $G = (V, E)$ a zjistí, jestli v G existuje vrchol $v \in V$, ze kterého jsou všechny ostatní vrcholy dosažitelné.

19. Navrhněte efektivní algoritmus, který dostane orientovaný acyklický graf $G = (V, E)$ a dvojici vrcholů s, c a vrátí počet různých cest vedoucích z s do c.

20. Dostanete orientovaný acyklický graf. Navrhněte efektivní algoritmus, který zjistí, jestli v grafu existuje cesta, která navštíví každý vrchol právě jednou.

21. Dostanete orientovaný acyklický graf. Navrhněte efektivní algoritmus, který spočítá délku nejdelší orientované cesty.

22. Dostanete orientovaný graf $G = (V, E)$, a ke každému vrcholu $v \in V$ dostanete navíc jeho cenu $\texttt{c}[v]$. Definujeme $\texttt{mcena}[v]$:= cena nejlevnějšího vrcholu dosažitelného z v (včetně v). Naším cílem na vymyslet algoritmus pracující v čase $\mathcal{O}(n + m)$, který vyplní pole $\texttt{mcena}[\,\cdot\,]$.

    (a) Nejprve vymyslete algoritmus, který funguje pro orientované acyklické grafy.
    Nápověda: Zpracovávejte vrcholy v určitém pořadí.

    (b) Rozšiřte předchozí algoritmus tak, aby pracovat na všech orientovaných grafech.
    Nápověda: Připomeňte si strukturu orientovaných grafů (meta-graf, silně souvislé komponenty).

23. Dostanete neorientovaný graf $G = (V, E)$. Zorientujte jeho hrany tak, aby pro každý vrchol v byl $|deg^+ v - deg^- v| \leq 1$.

24. Dostanete neorientovaný graf $G = (V, E)$. Zorientujte jeho hrany tak, aby se po orientovaných cestách dalo dostat odkudkoliv kamkoliv. Případně řekněte, že taková orientace neexistuje.

25. Dostanete neorientovaný graf $G = (V, E)$. Přidejte k němu nejmenší možný počet hran, aby G byl 2-souvislý.

26. Dostanete orientovaný graf $\vec{G} = (V, E)$. Přidejte k němu nejmenší možný počet orientovaných hran tak, aby G byl silně souvislý souvislý.

27. Dostanete rovinný graf $G = (V, E)$. Obarvení grafu je přiřazení barev vrcholům takové, že vrcholy spojené hranou mají různou barvu. Zajímá nás, kolik nejméně barev je potřeba.

    (a) Navrhněte efektivní algoritmus, který obarví G co nejmenším počtem barev.

    (b) Navrhněte lineární algoritmus, který obarví G pomocí 6 barev.

    (c) Jak rychle byste dokázali obarvit G pomocí 5 barev?

    (d) Graf G je k-degenerovaný, pokud každý jeho podgraf (včetně G samotného) obsahuje vrchol stupně menšího nebo rovno k. Ukažte, jak obarvit k-degenerované grafy pomocí $k + 1$ barev. Jak rychle poběží Váš algoritmus?

7.11.4 Úlohy na DFS průchod stavovým prostorem

Tady je pár úloh, jejichž řešení můžete nalézt vyzkoušením všech možností (tak zvaný backtracking). Připomínáme, že zkoušení všech možností, není vždy nejrychlejším řešení, ale někdy nic lepšího neumíme.

Pozor, často se stává, že studenti napíší backtracking nevhodně a vyhodnocují některé stavy vícekrát (viz jak se neefektivně počítali Fibonacciho čísla v sekci 4.7).

1. (Proskákání šachovnice koněm) Dostanete šachovnici o rozměrech $n \times m$. Vaším úkolem je zjistit, jestli šachovnici můžeme proskákat šachovým koněm tak, abychom každé políčko navštívili právě jednou.

   Nápověda : Pro urychlení výpočtu zkuste použít následující heuristiku. Vždy když si můžete vybrat, kam skočíte, tak nejdřív zkuste skočit na to políčko, odkud je nejméně možností jak pokračovat.

2. (Rozmístění šachových figurek na šachovnici tak, aby se neohrožovali) Je dána šachovnice o rozměrech $n \times m$. Jedna figurka ohrožuje druhou, pokud ji může skočit jedním šachovým tahem. Vaším úkolem je rozmístit na šachovnici co nejvíce předepsaných figurek tak, aby se navzájem neohrožovali.

   (a) Šachové věže.

   (b) Šachové střelce.

   (c) Šachové koně.

   (d) Šachové dámy.

   (e) Maharadže. Maharádža si v každém tahu může vybrat, jestli bude skákat jako kůň a nebo jako dáma.

   (f) Sultány. Sultán si v každém tahu může vybrat, jestli bude skákat jako kůň a nebo jako střelec.

   Nápověda: Program lze zjednodušit malým trikem. Místo toho, abychom si pamatovali, jestli je určité políčko ohroženo nějakou už umístěnou figurkou, si budeme pamatovat kolika figurkami je dané políčko ohroženo. Zjednoduší nám to návrat do předchozího stavu (odebrání figurky). Trik lze dobře využít například pro šachového koně.

3. (Rozmístění šachových figurek na pneumatiku tak, aby se neohrožovali) Navážeme na předchozí úlohu. Jestli už umíte rozmístit co nejvíce předepsaných šachových figurek na normální šachovnici, tak je můžete zkusit rozmístit na šachovnici, které je nakreslená na pneumatice. Šachovnici na pneumatice dostaneme tak, že běžnou šachovnici nejprve stočíme do válce a válec pak ohneme tak, aby se děravé konce spojili. Předpokládejme, že se kraje původní šachovnice spojí tak, že už ani nepoznáme, kde byly.

   Pozor, diagonály na šachovnici na pneumatice se chovají úplně jinak než na normální šachovnici!

   (a) Pro které rozměry $n \times m$ má šachovnice jen jednu levou a jednu pravou diagonálu?

   (b) Jak poznáme, kolik levých a kolik pravých diagonál má šachovnice o rozměrech $n \times m$?

   (c) Pokud už víte, kolik diagonál má která šachovnice na pneumatice a jak jsou diagonály na šachovnici „namotané“, tak na takovou šachovnici můžete zkusit rozmístit co nejvíce dam tak, aby se vzájemně neohrožovali.

7.11.5 Související úložky z teorie grafů

1. (Struktura 2-souvislých komponent) Pro každou 2-souvislou komponentu vytvoříme zvláštní vrchol. Dva zvláštní vrcholy spojíme hranou právě tehdy, když jim odpovídající 2-souvislé komponenty mají společnou artikulaci. Ukažte, že takto vzniklý graf je strom.

   Nápověda: Ukažte, že vzniklý graf nemůže obsahovat kružnice.

2. Dvě cesty v grafu jsou hranově disjunktní, pokud nemají společnou hranu (ale mohou sdílet vrchol). Ukažte, že v libovolném neorientovaném grafu můžeme spárovat vrcholy lichého stupně a spojit každý pár cestou tak, aby všechny cesty byly hranově disjunktní.

3. (Relace ekvivalence) Nechť S je konečná množina. *Binární relace* R je podmnožina $S \times S$. Jinými slovy R je množina dvojic $(x, y) \in S \times S$. Například pokud by S byla množina lidí, tak $(x, y) \in R$ právě tehdy když „x zná y“.

   Relace je *relací ekvivalence*, pokud splňuje následující vlastnosti

   - (reflexivita): $(x, x) \in R$
   - (symetrie): pokud $(x, y) \in R$, potom i $(y, x) \in R$
   - (tranzitivita): pokud $(x, y) \in R$ a $(y, z) \in R$, potom $(x, z) \in R$

   Například relace „má ve stejný den narozeniny“ je relace ekvivalence a relace “je otcem” není relace ekvivalence.

   Dokažte, že relace ekvivalence rozkládá množinu S na disjunktní skupiny S_1, S_2, …, S_k takové, že

   - Každé dva prvky jedné skupiny jsou v relaci. To je $(x, y) \in R$ pro každé $x, y \in S_i$, pro každé i.
   - Žádné dva prvky z různých skupin nejsou v relaci. Jinými slovy pro každé $i \neq j$ a pro každé $x \in S_i$, $y \in S_j$ platí $(x, y) \notin R$.

   Rozkladem S na skupiny S_1, S_2, …, S_k myslíme, že $S = S_1 \cup S_2 \cup \cdots \cup S_k$ a $S_i \cap S_j = \emptyset$ pro každé $i \neq j$.

   Nápověda: Reprezentujte si relace pomocí orientovaných grafů.

4. (Kotouč) Máme veliký kotouč, který má na obvodu napsány nuly a jedničky. Celý kotouč je skryt v pouzdře, akorát na jedné straně má „okénko“, kterým je vidět k po sobě jdoucích čísel. Pootočením kotouče se nejlevější číslice v okénku schová, ale zprava se objeví nová číslice. Takovýmto pootočením dostaneme následující k-tici. Kolik nejvíc číslic (nul a jedniček) můžeme napsat na obvod kotouče tak, aby se žádná k-tice po sobě jdoucích čísel neopakovala? Umíte takové pořadí nul a jedniček najít?

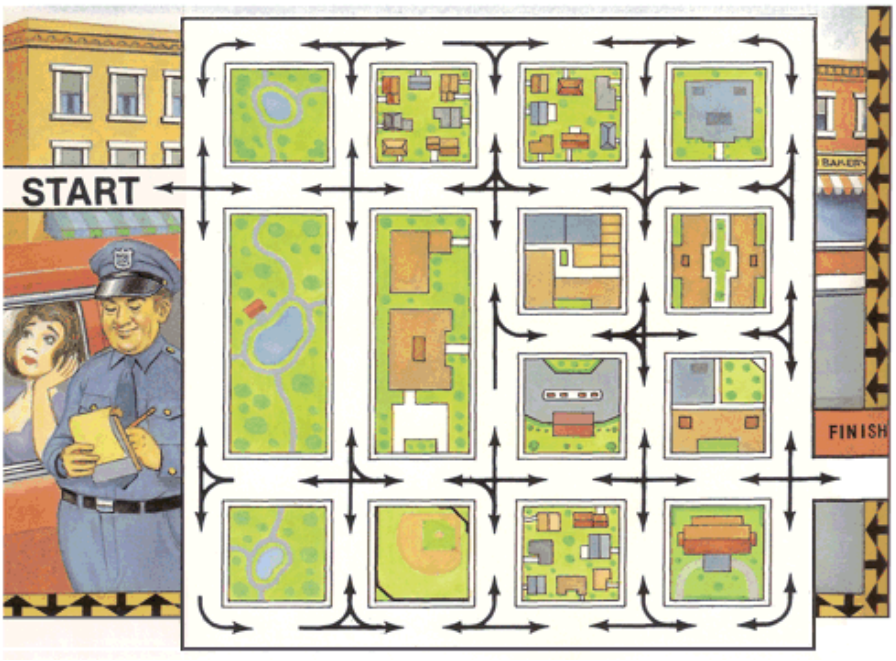
5. (Nekonečné grafy)

   (a) (Kráva před nekonečným plotem) Kráva stojí před nekonečně dlouhým a rovným plotem. Někde v plotu je díra. Kráva se může vydat hledat díru buď doleva nebo doprava. Zjistěte, jak má kráva postupovat, aby se nenachodila moc a zjistila, kde je díra.
   Zkuste najít algoritmus, podle kterého kráva nachodí nejvýše 2 krát tolik než je počáteční vzdálenost mezi krávou a dírou (plus malé epsilon). Takovýmto algoritmům se říká 2-kompetitivní, protože vydají nejvýše 2 krát větší odpověď, než kolik je optimum.

   (b) (Nekonečné bludiště) Stojíte uprostřed nekonečně velkého bludiště. Někde v bludišti je ukryt teleport, kterým se můžete dostat ven. Vymyslete, jak ho s jistotou najít.

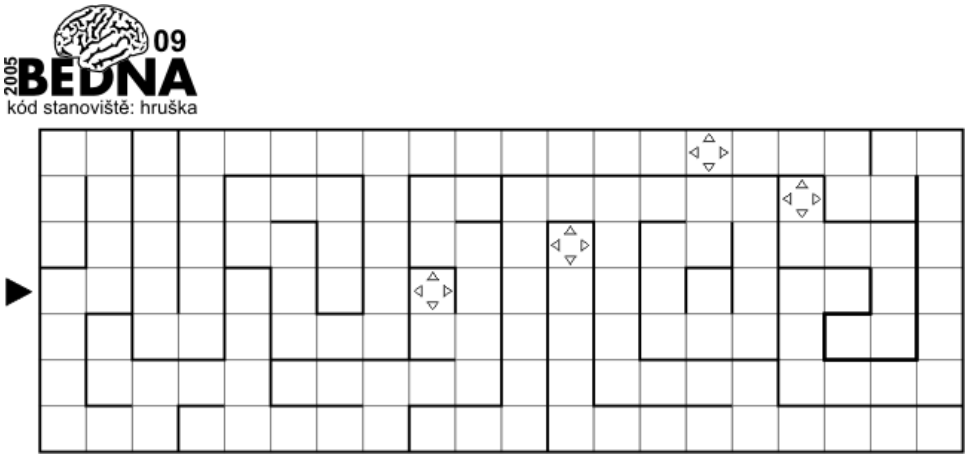
7.11.6 Hravá bludiště

1. (Farmář jede na trh) Následující bludiště pochází od Robert Abbotta a jmenuje se „Farmer goes to market“.[9] Vaším úkolem je poradit farmáři, jak má projet městem tak, aby se ze startu dostal do cíle (finish) a neporušil místní dopravní předpisy. Šipky na každé křižovatce znázorňují, odkud kam se dá křižovatka projet bez porušení dopravních předpisů.

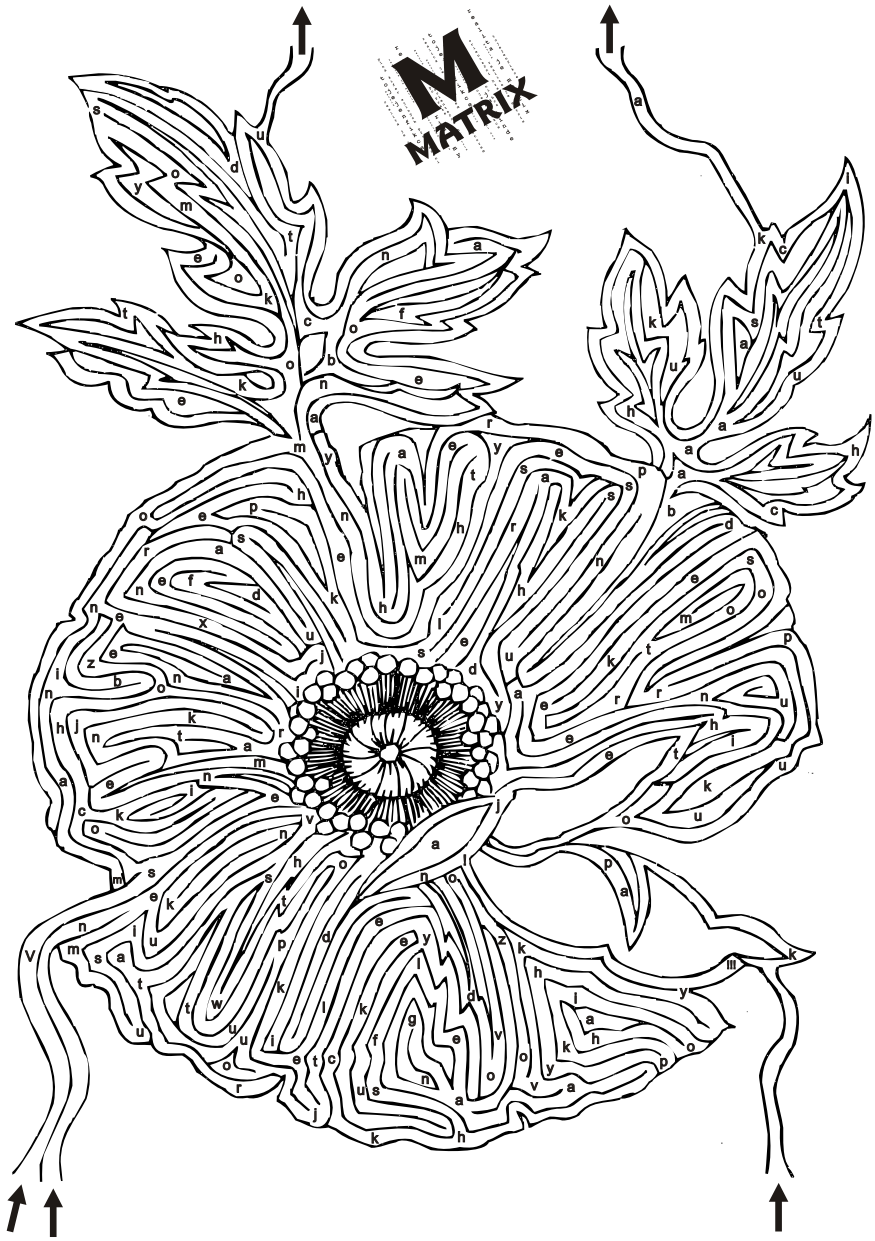


7.11.7 Šifry

Vyřešte následující šifry. První se objevila v šifrovací hře Bedna 2005 a druhá ve hře MATRIX 2005.



[9] Jde o složitější variantu bludiště od Martina Gardnera: Traffic Flow In Floyd's Knob.


M
MATRIX

Kapitola 8

Halda

8.1 Halda

Halda je často používaná datová struktura, která slouží k rychlému hledání minima. Dostaneme n prvků a chceme najít nejmenší z nich, tedy jejich minimum.

Pokud hledáme jen jeden nejmenší prvek, tak ho najdeme průchodem všech prvků. V každém kroku porovnáme procházený prvek s dosud nalezeným minimem. Pokud je procházený prvek menší, tak jsme nalezli nové dosavadní minimum. V podsekci 2.2 o časové složitosti jsme si ukázali, že je pro nalezení minima potřeba alespoň $n-1$ porovnání. Proto je hledání nejmenšího prvku pomocí průchodu nejlepším možným řešením.

Co když chceme najít i druhý nejmenší prvek? Nebo co když chceme najít k nejmenších prvků?

Jednoduchým řešením je nalézt nejmenší prvek, odebrat ho z množiny prvků[1] a ve zbývajících prvcích hledat minimum stejným způsobem. Tím dostaneme časovou složitost $\mathcal{O}(kn)$.[2] Můžeme najít následující nejmenší prvky rychleji?

Zajímavý nápad je rozdělit si zadaná čísla na dvě části o n_1 a n_2 prvcích. Na n_1-1 porovnání najdeme minimum v první části a na n_2-1 porovnání ve druhé. Porovnáním minim z obou částí najdeme minimum ze všech n čísel. Celkem jsme potřebovali $n_1-1+n_2-1+1 = n-1$ porovnání. Jak teď najít druhý nejmenší prvek? V jedné části minimum známe, ve druhé ho budeme muset znova spočítat. Pokud jsou části stejně velké, tak už druhé nejmenší číslo najdeme na nejvýše $\lceil n/2 \rceil + 1$ porovnání. Celkem jsme dvě nejmenší čísla nalezli na nejvýše $\lceil 3n/2 \rceil$ porovnání.

Můžete navrhnout, že celou myšlenku můžeme zopakovat i v každé části. Rozdělením obou částí dostaneme čtyři nové části. Ty můžeme zase dále dělit a to tak dlouho, dokud části nejsou jednoprvkové. Tím dostaneme datovou strukturu, které se říká halda. Při práci s haldou používáme i další užitečné operace: přidávání nových prvků, mazání prvků a podobně.

Ještě než přistoupíme k samotné definici, tak jeden problém pro vás. Dostanete n prvků. Jak rychle najdete $\sqrt{n}$ nejmenších z nich? Zvládli byste to v čase $\mathcal{O}(\sqrt{n})$? (viz. cvičení)

Definice: *Stromová datová struktura* je reprezentace stromu v počítači (viz definice na straně 55). V každém vrcholu v si pamatujeme hodnotu $x(v)$, které se říká klíč (key).

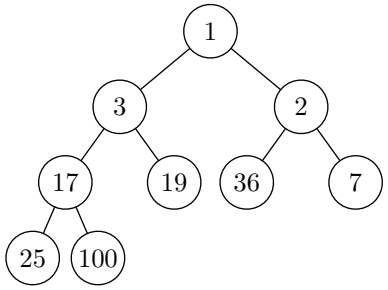
Halda je stromová datová struktura splňující vlastnost haldy. Zakořeněný strom

[1] Na místo smazaného prvku překopírujeme poslední prvek a zkrátíme pole o jedna.

[2] Někdo jiný by mohl navrhnout, ať nejprve všechny prvky setřídíme. Potom najdeme k nejmenších prvků v čase $\mathcal{O}(n \log n + k)$. To je ale opět velký čas v případech, kdy je k výrazně menší než $\log n$.

má *vlastnost haldy* právě tehdy, když pro každý uzel v a pro každého jeho syna w platí $x(v) \leq x(w)$. Díky této vlastnosti bude kořen stromu obsahovat nejmenší klíč z celé haldy.

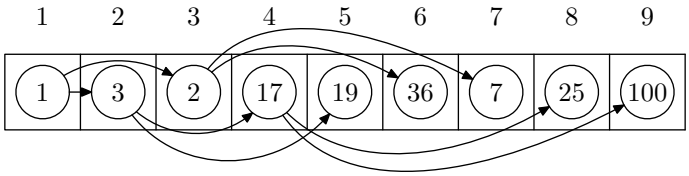
Binární halda je úplný binární strom s vlastností haldy. Strom je *binární*, pokud má každý vrchol nejvýše dva potomky. Binární strom je *úplný*, pokud jsou všechny jeho hladiny kromě poslední úplně zaplněny a v poslední hladině leží listy co nejvíce vlevo. Úplnost binárního stromu zaručuje hezký vyvážený tvar stromu a to nám garantuje výšku stromu nejvýše $\lceil \log n \rceil$. Příklad binární haldy je na obrázku vpravo.[3]



Operace: s haldou běžně provádíme následující operace:

- `MIN` – vrátí nejmenší klíč v haldě
- `DELETE_MIN` – vymaže uzel s nejmenším klíčem
- `INSERT`(h) – přidá nový uzel s hodnotou h
- `DELETE`(v) – smaže uzel v
- `DECREASE_KEY`(v, *okolik*) – uzlu v zmenší hodnotu klíče o *okolik*
- `INCRESE_KEY`(v, *okolik*) – uzlu v zvětší hodnotu klíče o *okolik*
- `MAKE` – dostane pole n prvků a vytvoří z nich haldu.

Binární haldu si můžeme snadno reprezentovat v poli $\mathtt{x}[\,\cdot\,]$. Využijeme při tom úplnosti binárního stromu. Uzly stromu očíslujeme po hladinách počínaje od jedničky. Těmito čísly budeme uzly označovat. Uzel i uložíme do $\mathtt{x}[i]$. Levý syn uzlu k bude uložen na pozici $2k$ a pravý syn uzlu k na pozici $2k+1$. Naopak otec vrcholu k se bude nacházet na pozici $\lfloor k/2 \rfloor$. Na následujícím obrázku jsme takto do pole poskládali binární strom z předchozího obrázku.



Aby se nám s haldou lépe pracovalo, zavedeme si dvě pomocné funkce `BUBBLE_UP`(v) a `BUBBLE_DOWN`(v). Bublání funguje stejně jako v třídění pomocí bublinkového algoritmu, akorát místo průchodu pole probubláváme podél cesty ve stromě vedoucí z uzlu v do kořene nebo do listu. `BUBBLE_UP` zajistí probublání lehkých prvků směrem nahoru ke kořeni. `BUBBLE_DOWN` naopak zajistí propad těžkých prvků dolů směrem k listům. Nebudeme probublávat podél celé cesty, ale jen dokud v aktuálním vrcholu

[3] **Pohádka:** (Jak si představit fungování haldy?) Zajdeme na kouzelnickou párty, kde si prohlédneme kouzelnickou šíšu. Kouzelnická šíša vypadá jako obrázek haldy. Ten znárodňuje kouzelné baňky, které jsou propojeny velmi úzkými trubičkami. Každá baňka je naplněna jiným plynem – podle toho, co který kouzelník donese. Vrchní baňka je opatřena hadičkou s ventilem, pomocí které lze plyn z horní baňky odpouštět a ochutnávat.

Lehčí plyny se snaží stoupat vzhůru a těžší naopak klesají. Podívejme se na dvě sousední baňky, které jsou spojeny trubičkou. Pokud spodní baňka obsahuje lehčí plyn než horní baňka, tak začne lehčí plyn probublávat do horní baňky. Těžší plyn začne naopak klesat a probublávat dolů. Jsou to kouzelná baňky. Pokud má baňka na výběr, tak kouzlo šíšové seance pohlídá, aby do baňky probublal jen ten lehčí ze dvou plynů.

Tato kouzelná šíša zajišťuje, aby se při kouzelnických dýcháncích mohli odpouštět plyny ve správném pořadí. Jinak by z toho kouzelními mohli mít střevní potíže (plyny by probublávali ve střevech stejně, jako to mohou dělat mezi baňkami).

není splněna vlastnost haldy. Pomocí `swap(`i`, `j`)` značíme prohození dvou prvků na pozicích i a j v poli `x`$[\,\cdot\,]$.

```
BUBBLE_UP(k):
   while k > 1 and x[⌊k/2⌋] > x[k] do
      swap(k, ⌊k/2⌋)
      k := ⌊k/2⌋

BUBBLE_DOWN(k):
   while k ≤ ⌊n/2⌋ do
      min := 2k {min bude pozice syna s menším klíčem }
      if 2k + 1 ≤ n and x[2k] > x[2k + 1] then
         min := 2k + 1
      if x[min] < x[k] then
         swap(k, min)
       else
         break
      k := min
```

Časová složitost obou probublávacích funkcí je nejvýše tolik, kolik je výška úplného binárního stromu a to je $\mathcal{O}(\log n)$. Pomocí pomocných funkcí už snadno naimplementuje ostatní operace haldy.

```
MIN:
   return x[1]

DELETE_MIN:
   x[1] := x[n]
   n := n − 1
   BUBBLE_DOWN(1)

INSERT(h):
   n := n + 1
   x[n] := h
   BUBBLE_UP(n)

DELETE(k):
   val := x[k]
   x[k] := x[n]
   n := n − 1
   if val ≤ x[k] then
      BUBBLE_DOWN(k)
   else
      BUBBLE_UP(k)

DECREASE_KEY(k,okolik):
   x[k] := x[k] − okolik
   BUBBLE_UP(k)

INCRESE_KEY(k,okolik):
   x[k] := x[k] + okolik
   BUBBLE_DOWN(k)
```

Časová složitost `MIN` je konstantní. Časová složitost ostatních výše uvedených operací je stejná jako časová složitost `BUBBLE_UP` nebo `BUBBLE_DOWN` a to je $\mathcal{O}(\log n)$.

Podívejme se na to, jak z n prvků na vstupu postavit haldu. Jednoduše bychom mohli začít s prázdnou haldou a n-krát zavolat operaci `INSERT`. To by mělo časovou složitost $\mathcal{O}(n \log n)$. My si ale ukážeme, jak postavit haldu z n prvků v poli v lineárním čase. Prvky necháme v poli $\mathtt{x}[\,\cdot\,]$ tak, jak jsme je dostali na vstupu a nad polem si představíme binární strom. Naším cílem je přeuspořádat prvky tak, aby splňovaly vlastnost haldy. Dosáhneme toho postupným voláním `BUBBLE_DOWN`.

```
MAKE:
  for i := ⌊n/2⌋ downto 1 do
    BUBBLE_DOWN(i)
```

Proč po skončení `MAKE` splňuje každý vrchol vlastnost haldy? Platí invariant, že v každém kroku algoritmu splňují všechny vrcholy j, pro $i \leq j \leq n$, vlastnost haldy. V následujícím kroku necháme klíč ve vrcholu $i-1$ probublat dolů, takže také vrchol $i-1$ bude splňovat vlastnost haldy. U vrcholů j, pro $i \leq j \leq n$, se tím vlastnost haldy neporuší.

Nyní dokážeme, že postavení haldy pomocí `MAKE` bude trvat jen $\mathcal{O}(n)$. Nechť $h = \lceil \log n \rceil$ je výška úplného binárního stromu na n vrcholech. Operace `BUBBLE_DOWN` zavolaná na uzel v k-té hladině od zdola bude trvat čas přímo úměrný k. Z vlastností binárního stromu víme, že v následující hladině stromu je dvojnásobek prvků. Proto je v k-té hladině od zdola 2^{h-k} prvků. Označíme-li časovou složitost operace `MAKE` pomocí X, dostaneme

$$X = \sum_{k=1}^{h} 2^{h-k} \cdot k.$$

Sumu lze spočítat fintou tak, že ji vynásobíme dvěma a odečteme od ní tu samou sumu. Koeficienty před stejnou mocninou dvojky se krásně odečtou a zbude nám jednoduchá suma. Konkrétně pro člen 2^{h-k} máme $2 \cdot k2^{h-k} - (k-1)2^{h-k+1} = 2^{h-k}$. Celkem dostaneme

$$X = 2X - X = 2^h + \sum_{k=1}^{h-1} 2^{h-k} - 2^0 h = \sum_{j=0}^{h} 2^j - 1 - h = 2n - \lceil \log n \rceil = \mathcal{O}(n).$$

Poznámka k implementaci: Operace `BUBBLE_UP`(k) a `BUBBLE_DOWN`(k) můžeme implementovat lépe. Místo prohazování dvou sousedních prvků na procházené cestě P ve stromě si zapamatujeme počáteční hodnotu $val := \mathtt{x}[k]$, na cestě P budeme procházené prvky posouvat o jedna a hodnotu val uložíme až na konečnou pozici.

Poznámka: Existují i jiné implementace haldy a haldových operací.

- (pole) Nejjednodušší realizace haldových operací je v poli. Prvky necháme v poli tak, jak jsme je dostali. Nalezení minima realizujeme průchodem celého pole v čase $\mathcal{O}(n)$. Všechny ostatní operace realizujeme přímým přístupem do pole v čase $\mathcal{O}(1)$. Přidávané prvky vložíme na konec pole a zvětšíme velikost pole o 1. Mazání prvku provedeme tak, že mazaný prvek nahradíme posledním prvkem a pole o jedna zkrátíme.

- (d-regulární halda) Zobecněním binární haldy je *d-regulární halda.* Od binární se liší pouze tím, že každý vrchol má nejvýše d synů. Podmínka na úplnost stromu (zaplněnost hladin) zůstává. Více se o d-regulární haldě dozvíte ve cvičeních.

- (Fibonacciho halda) Fibonacciho halda pochází od Fredmanna a Tarjana. Fibonacciho halda pro změnu slevuje z úplnosti stromu, ale stále vyžaduje rozumnou zaplněnost hladin ("košatost" binárních stromů). Fibonacciho halda je množina stromů splňujících vlastnost haldy. Stromy v haldě jsou různého stupně 0, 1, 2,..., k. Stupeň stromu zhruba odpovídá stupni grafu v kořeni. S rostoucí vzdáleností od kořene klesá i maximální povolený stupeň ve vrcholech stromu. Dá se ukázat, že strom stupně d obsahuje alespoň Φ^d vrcholů, kde $\Phi = (1+\sqrt{5})/2$.

  Nejhorší případ stromu stupně d, tj. nejmenší a tedy i nejméně košatý strom, jaký je povolen, se konstruuje složením nejmenších stromů stupně $d-1$ a $d-2$. Počet vrcholů v nejmenším stromu stupně d je $F_d = F_{d-1} + F_{d-2}$. Formulka je stejná jako při výpočtu Fibonacciho čísel, proto se této haldě říká Fibonacciho halda.

  Fibonacciho halda realizuje operace `MIN`, `INSERT`, `DECREASE_KEY`, `INCRESE_KEY` v amortizovaném čase $\mathcal{O}(1)$ a operace `DELETE_MIN` a `DELETE` v amortizovaném čase $\mathcal{O}(\log n)$. Implementace této haldy je podstatně složitější a konstanty před časovými složitostmi jednotlivých operací jsou poměrně vysoké. Na druhou stranu použitím Fibonacciho haldy můžeme dosáhnout podstatně lepších asymptotických časových složitostí.

Následující tabulka shrnuje časové složitosti jednotlivých operací při různých reprezentacích.

	`DELETE_MIN`	`DELETE`	`INSERT` `DECREASE_KEY`	`INCRESE_KEY`
pole	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
binární halda	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
d-regulární halda	$\mathcal{O}(\frac{d \log n}{\log d})$	$\mathcal{O}(\frac{d \log n}{\log d})$	$\mathcal{O}(\frac{\log n}{\log d})$	$\mathcal{O}(\frac{d \log n}{\log d})$
Fibonacciho halda	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Pozor, u Fibonacciho haldy je všude uvedena amortizovaná časová složitost.

Všimněme si, jak parametr d u d-regulární haldy interpoluje mezi polem a binární haldou (pro $d = 2$ dostaneme binární haldu a pro $d = n$ realizaci v poli). Optimální hodnota parametru d se hledá tak, aby celková časová složitost algoritmu, ve kterém haldu používáme, byla co nejmenší.

8.2 Prioritní fronta

Motivace: V počítači je fronta úloh čekajících na zpracování. Čekající úlohy „sedí" v paměti v dlouhé řadě židlí, které běžně říkáme pole. Pokud přibude nová úloha, tak se posadí na židli na konci fronty. Pan Procesor úlohy postupně zpracovává. Když řekne „další pán na holení", tak si vyzvedne úlohu na začátku fronty. Takhle funguje obyčejná fronta. Na její realizaci není nic složitého.[4]

Co když se ale přižene systémová úloha, která začne tvrdit, že je důležitější než ostatní úlohy, a začne předbíhat ve frontě? Pokud budeme chtít důležitější úlohy upřednostnit, tak každé úloze přiřadíme její prioritu. Představme si ji jako číselnou hodnotu. Úloha s vyšší prioritou může předběhnout všechny úlohy nižší prioritou.

[4]Pokud budeme mít dostatečně velké pole, tak nemusíme úlohy přesazovat. Vyzvednutí úlohy, která je na řadě, i posazení nově příchozí úlohy nám zabere jednotkový čas. Pokud bychom chtěli šetřit místem, tak můžeme řadu židlí stočit do kruhu. Díky tomu se budou uvolněné židle automaticky „recyklovat" tím, že se jakoby přesunou na konec řady. Ve skutečnosti nic nepřesunujeme, jen si posuneme ukazovátko určující začátek fronty. Díky tomu bude vyzvednutí úlohy, která je na řadě, i posazení nově příchozí úlohy trvat jednotkový čas. (Nikoho nemusíme přesazovat.)

Jak ale teď bude fungovat vyzvedávání úlohy, která je na řadě (má nejvyšší prioritu)? A kam posadíme nově příchozí úlohy? Frontu můžeme v poli udržovat setříděnou podle priorit. Odebrání úlohy, která je na řadě, se nebude lišit od obyčejné fronty. Odebereme první prvek pole. Pro přidání nové úlohy musíme nejprve přesadit úlohy s nižší prioritou o jedno místo zpátky a tím si vytvořit volnou židli pro nově příchozí úlohu. To ovšem vyžaduje až $\mathcal{O}(n)$ přesazování.

Navrhněte systém fungování prioritní fronty tak, aby přidání nové úlohy a i odebrání úlohy s nejvyšší prioritou, fungovalo co nejefektivněji. To je abychom museli přesadit co nejméně úloh. Není nutné, aby byly úlohy v poli seřazeny podle priority.

Úkol: Obyčejná fronta je seznam prvků seřazený podle času příchodu. Prioritní fronta je seznam prvků seřazený podle priorit. Každý prvek má svojí hodnotu, tzv. prioritu. Prvky s vyšší prioritou mohou ve frontě předběhnout prvky s nižší prioritou. Pokud mají dva prvky stejnou prioritu, tak jsou seřazeny podle času příchodu.

Pro práci s prioritní frontou potřebujeme umět odebrat prvek, který je na řadě (ten s nejvyšší prioritou), přidávat a mazat prvky a také u některých prvků měnit prioritu. A to vše co nejefektivněji.

Pro jednoduchost předpokládejme, že vyšší priorita odpovídá nižší číselné hodnotě. Prvky s nejvyšší prioritou jsou tedy ty s nejnižší číselnou hodnotou.[5]

Řešení pomocí pole: V tomto řešení je $\mathtt{x}[\,\cdot\,]$ neuspořádané pole. Nemusíme nic vytvářet. Operace nalezení minima spočívá v průchodu pole a trvá čas $\mathcal{O}(n)$. Ostatní operace lze realizovat v konstantním čase.

Použitím setříděného pole si moc nepomůžeme. Zkuste se zamyslet nad realizací a časovou složitostí jednotlivých operací. Dostaneme horší výsledek než při použití haldy.

Řešení pomocí haldy: Použijeme haldu, kde klíčem uzlů/úloh bude jejich priorita. Operace požadované po prioritní frontě přímo odpovídají operacím pro práci s haldou. Například odebrání prvku s nejvyšší prioritou zajistí funkce `MIN` a `DELETE_MIN`, změnu priority funkce `INCRESE_KEY` a `DECREASE_KEY` apod. Časová složitost operací je $\mathcal{O}(\log n)$.

8.3 Příklady

1. (nalezení prvních $\sqrt{n}$ nejmenších čísel) Dostaneme pole n prvků $\mathtt{A}[1,\ldots,n]$. Najděte algoritmus, který vypíše prvních $\sqrt{n}$ nejmenších čísel. Umíte to v čase $\mathcal{O}(n)$?

   Nápověda: čtverec.

2. (Max-heap) Vymyslete podobnou datovou strukturu jako je halda, ale tentokrát chceme odebírat prvek s největší hodnotou.

3. (Heapsort) Zkuste vymyslet jednoduchý třídící algoritmus využívající haldy. Jaká bude jeho časová složitost v nejhorším případě? Umíte ho realizovat tak, abychom potřebovali pouze to pole, ve kterém dostaneme vstup?

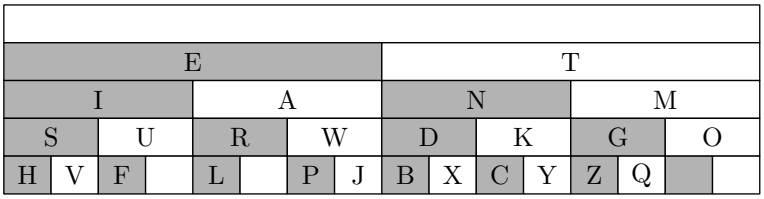
4. (d-regulární halda) Binární halda je halda, kde má každý vrchol nejvýše dva syny. V d-regulární haldě má každý vrchol nejvýše d synů. Zkuste zobecnit binární haldu na d-regulární haldu. Jak se d-regulární halda uloží do pole? Na které pozici budou synové vrcholu z pozice k? Na jaké pozici bude jeho otec? Jak se změní implementace haldových operací? Jaká bude jejich časová složitost?

[5] Čtenář by jistě zvládl prohodit maxima za minima, ale nechme to stejné jako v zavedení haldy.

5. (Kalendář událostí) Jak byste realizovali kalendář událostí? Kalendář událostí funguje podobně jako váš diář. Průběžně dostáváme úlohy, které mají přesně stanovený čas, kdy se mají vykonat. Můžeme je dostávat v libovolném pořadí, tedy ne nutně seřazené podle času. Úlohy zpracováváme v pořadí podle času. Při zpracovávání úlohy se dozvíme, jaké další úlohy přibyly. Občas naopak zjistíme, že máme nějakou naplánovanou úlohu zrušit, případně ji přeplánovat na jiný čas.

   Při realizaci se stačí zamyslet nad následujícími funkcemi: Odebrání úlohy, která je na řadě. Přidání nové úlohy, smazání a přeplánování konkrétní úlohy.

6. (Využití reprezentace binárního stromu v poli) Napište co nejrychlejší program, který čte ze vstupu morseovku a překládá ji do anglické abecedy. Využijte při tom vyhodnocovacího stromu z následujícího obrázku.[6] Binární strom na obrázku je netradičně nakreslený jako tabulka.



   Při vyhodnocování kódu písmene v morseovce (například `.-`) začneme v prvním řádku (v kořenu stromu). Pokud je první znak tečka, přesuneme se o řádek níže doleva. Pokud je první znak čárka, přesuneme se o řádek níže doprava. Dostali jsme se do dalšího vrcholu stromu. Tento postup opakujeme, dokud nezpracujeme všechny znaky aktuálního písmene v morseovce. Skončíme ve vrcholu označeném písmenem, které odpovídá Morseovu kódu.

   Binární vyhodnocovací strom si reprezentujte v poli, například jako řetězec "␣`ETIANMSURWDKGOHVF`␣`L`␣`PJBXCYZQ`␣␣", kde ␣ představuje mezeru. Předpokládejte, že vstup obsahuje pouze znaky '`.`', '`-`', '`|`' a že je celý vstup je ukončen znakem '`$`'. Tak "`....|---|-..|-.|.|...|-|.|...|-|..`".

   Nápověda: Jediný cyklus + stavový automat pamatující si pozici ve vyhodnocovacím stromě.

[6] Pro zjednodušení práce jsme použili anglickou verzi morseovky, bez písmene CH. To by nám lehce komplikovalo práci tím, že se skládá ze dvou znaků.

Kapitola 9

Nejkratší cesta v grafu

Motivace: Silniční síť si můžeme znázornit grafem. Křižovatky odpovídají vrcholům grafu a silnice mezi nimi hranám. Každý úsek silnice odpovídající hraně má svojí délku. Ta odpovídá ohodnocení hrany. Můžeme se ptát jaká je nejkratší cesta z Prahy do Ostravy? Kolik je to kilometrů a kudy vede? Maximální povolená rychlost na každém úseku silnice nám říká, jak rychle po ní můžeme jet. Z rychlosti a délky silnice se dá dopočítat, za jak dlouhou úsek silnice projedeme. Proto se můžeme ptát i na nejrychlejší cestu z Prahy do Ostravy. Nejrychlejší cesta nemusí být zrovna nejkratší.

Problém hledaní nejkratší cesty v grafu a jemu podobné problémy se vyskytují téměř všude: při hledání autobusového nebo vlakového spojení v online jízdních řádech, při hledání optimální cesty v navigacích do aut, při plánování pohybu robotů a nebo při routování paketů v internetu.

Definice: Co to je vzdálenost? Vzdálenost je matematicky popsána pojmem metrika.[1] My si v této kapitole vystačíme se vzdáleností v grafech, tj. s grafovou metrikou. Nechť G je graf s ohodnocením hran $c : E(G) \to \mathbb{R}$. Číslo $c(e)$ se nazývá cena hrany e, ale v kontextu hledání nejkratší cesty mu budeme říkat délka hrany e. *Délka cesty* $x = v_0e_1v_1e_2 \dots v_{k-1}e_kv_k = y$ se počítá jako $\sum_{i=1}^{k} c(e_i)$. *Vzdálenost dvou vrcholů* x a y v grafové metrice je délka nejkratší cesty mezi x a y.

Úkol 1: (Shortest path) Dostaneme graf G s ohodnocením hran $c : E \to \mathbb{R}$, počáteční vrchol s a cílový vrchol t. Chceme nalézt nejkratší cestu z s do t.

Všechna prakticky používaná řešení místo výše uvedeného problému řeší problém obecnější. Místo jedné cesty z s do t hledáme nejkratší cestu z s do všech ostatních vrcholů.

Úkol 2: (Single source shortest path) Dostaneme graf G s ohodnocením hran

[1] Nechť M je neprázdná množina. *Metrika* je zobrazení $\rho : M \times M \to \mathbb{R}_+$, které pro libovolné $x, y \in M$ splňuje axiomy:

1. (totožnost): $\rho(x, y) = 0$ právě tehdy když $x = y$
2. (symetrie): $\rho(x, y) = \rho(y, x)$
3. (trojúhelníková nerovnost): $\rho(x, z) \leq \rho(x, y) + \rho(y, z)$.

Z prvního a třetího axiomu vyplývá nezápornost metriky $\rho(x, y) \geq 0$. Číslu $\rho(x, y)$ říkáme vzdálenost bodů x a y. Nechť $p_1 = (x_1, y_1)$ a $p_2 = (x_2, y_2)$ jsou dva body v rovině. Mezi nejpoužívanější metriky v rovině patří:

- eukleidovská metrika – vzdálenost p_1 a p_2 je délka úsečky p_1p_2, neboli vzdálenost vzdušnou čarou.
- maximová metrika – vzdálenost p_1 a p_2 je maximum z $|x_1 - x_2|$ a $|y_1 - y_2|$.
- manhattanská metrika – po Manhattanu v New Yorku se chodí v síti pravoúhlých ulic, které jsou rovnoběžné s osami souřadného systému. Vzdálenost p_1 a p_2 je $|x_1 - x_2| + |y_1 - y_2|$.

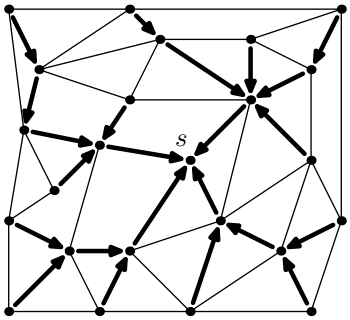
$c : E \to \mathbb{R}$, počáteční vrchol s. Chceme nalézt nejkratší cestu z s do všech ostatních vrcholů $v \in V(G)$.

Předchozí úkol můžeme ještě více zobecnit.

Úkol 3: (All pairs shortest path) Dostaneme graf G s ohodnocením hran $c : E \to \mathbb{R}$ a chceme nalézt nejkratší cestu mezi každou dvojicí vrcholů.

Nejkratší cesta splňuje princip optimality. Ten říká, že každá podcesta nejkratší cesty je také nejkratší cestou mezi svými koncovými vrcholy. Přesněji řečeno, je-li xPy nejkratší cesta z x do y a u,v jsou dva vrcholy na P, tak je i úsek uPv nejkratší cestou mezi u a v. Pokud mezi u a v existuje kratší cesta uQv, tak v původní cestě xPy nahradíme úsek uPv cestou uQv a dostaneme kratší cestu. To je spor.

Podívejme se na to, jak by měl vypadat výstup řešení druhého úkolu. V každém vrcholu v si budeme pamatovat předchůdce $\pi(v)$ na nejkratší cestě do s. Jednoduše řečeno, $\pi(v)$ je směrovka říkající kudy máme vrchol v opustit, abychom se dostali zpět do vrcholu s po nejkratší cestě. Je-li je graf G souvislý, tak z každého vrcholu v existuje cesta do s a každý vrchol kromě s má právě jednoho předchůdce $\pi(v)$ na nejkratší cestě.[2] Hrany $\{v, \pi(v)\}$ tvoří strom orientovaný do kořene s, který nazveme *strom nejkratší cesty.* Cílem hledání nejkratší cesty v grafu je najít strom nejkratší cesty.[3]



Protože ve městech kromě běžných silnic existují i jednosměrky, tak je zcela přirozené hledat nejkratší cestu i v orientovaných grafech. Vzdálenost dvou vrcholů s a t v orientovaném grafu je opět délka nejkratší orientované cesty z s do t.[4] Všechny algoritmy, které si v této kapitole uvedeme, fungují i pro orientované grafy. Není to nic překvapivého, vždyť i neorientovaný graf, ve kterém hledáme cestu, máme reprezentovaný jako orientovaný graf. Každou hranu si pamatujeme jako dvě orientované hrany/šipky vedoucí proti sobě.

9.1 Realizace grafu pomocí provázků a kuliček

Abychom lépe porozuměli tomu, jak vypadá nejkratší cesta z počátečního vrcholu s do ostatních vrcholů, tak si představme následující realizaci grafu. Místo vrcholů vezmeme kuličky a pokud mezi dvěma vrcholy vede hrana, spojíme odpovídající kuličky provázkem takové délky, kolik je ohodnocení hrany.

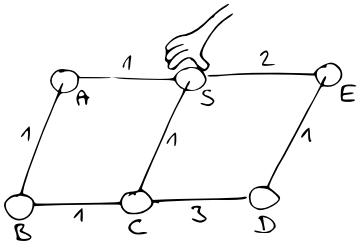
[2] Kdyby existovali dvě stejně dlouhé cesty, tak zvolíme libovolnou z nich.

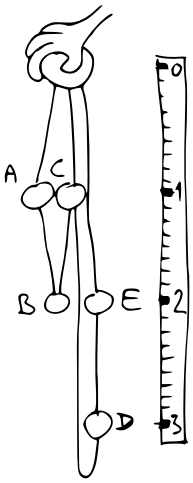
[3] Strom nejkratší cesty je spíše kostra, protože pokrývá všechny vrcholy. Ale pozor, tato kostra nemusí být minimální kostrou grafu. Zkuste najít protipříklad.

[4] Poznamenejme ale, že grafová vzdálenost v orientovaných grafech už nemusí být metrikou, protože nemusí splňovat symetrii.

Tento graf chytneme za kuličku, která odpovídá počátečnímu vrcholu s, a zvedneme ji do výšky. Ostatní kuličky zůstanou viset směrem dolů a to v takových vzdálenostech od s, které odpovídají délce nejkratší cesty.

Ke zjištění nejkratší cesty stačí vzít metr a změřit si vzdálenost s, t.





9.2 Neohodnocený graf

Délka cesty v neohodnoceném grafu je počet hran na cestě. Vzdálenost mezi vrcholy x a y je délka nejkratší cesty mezi x a y. Pokud neexistuje cesta mezi x a y, tak je vzdálenost nekonečná. Takto definovaná vzdálenost odpovídá vzdálenosti v ohodnoceném grafu, kde je každá hrana ohodnocená jedničkou.

Nejkratší cestu můžeme hledat pomocí průchodu do šířky, kterému se někdy říká *algoritmus vlny* (viz sekce 7.10). Nazýváme ho tak proto, že vrcholy procházíme ve vlnách, které se šíří z počátečního vrcholu jako vlny na vodní hladině. Jednotlivým vlnám se říká *vrstvy* průchodu do šířky. V *i-té vrstvě* jsou obsaženy právě vrcholy ve vzdálenosti i od počátečního vrcholu s.

V realizaci grafu pomocí kuliček a provázků jednotkové délky budou po zvednutí počátečního vrcholu ostatní kuličky viset přesně po vrstvách, které odpovídají průchodu do šířky.

9.3 Nezáporné ohodnocení hran

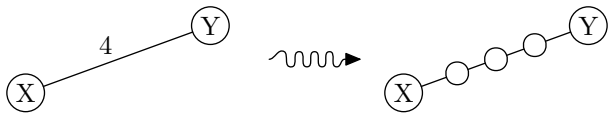
Ve většině aplikací mají hrany jinou než jednotkovou délku, protože délka hrany odpovídá například délce úseku silnice. Předpokládejme, že délky hran jsou nezáporná celá čísla. Jak najít nejkratší cestu teď?

Mohli byste navrhnout, abychom použili BFS úplně stejně jako v neohodnocených grafech. To by ale nefungovalo, protože přímá hrana z s do t může být mnohem delší než cesta vedoucí po dvou krátkých hranách.

Adaptace průchodu do šířky

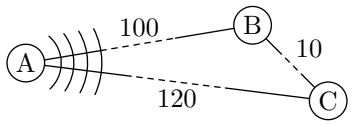
Nemohli bychom BFS upravit tak, aby už fungovalo? Ano to se dá. Zamysleme se nad podstatou BFS. Jednoduchá představa je, že vezmeme miliardu Číňanů, postavíme je do výchozího vrcholu s a necháme je procházet graf. Všichni Číňané chodí stejně rychle a pokud z vrcholu vede více cest, po kterých se mohou vydat, tak se rozdělí.

Průchod hrany délky dva jim bude trvat stejně dlouho jako průchod dvou hran jednotkové délky. Proto můžeme virtuálně každou hranu délky k rozdělit na k jednotkových úseků tím, že na ni přidáme $k-1$ pomocných vrcholů. Jinými slovy hranu délky k nahradíme cestou délky k. Touto úpravou dostaneme graf G'.



Graf G' už má jen hrany jednotkové délky a proto na nalezení nejkratší cesty můžeme použít BFS. Nejkratší cesta mezi původními vrcholy v G' odpovídá nejkratší cestě mezi stejnými vrcholy v G.

U grafů s vysokým ohodnocením hran je sledování průběhu algoritmu docela nudné, protože většinu času uvidíme jen to, jak Číňané postupují skrz hrany. Podívejme se například na graf K_3 s vrcholy A, B, C a s ohodnoceními hran $c(AB) = 100$, $c(AC) = 120$ a $c(BC) = 10$. Při zahájení průchodu z vrcholu A budeme čekat 100 časových jednotek, než Číňané dorazí do dalšího vrcholu. Jelikož dopředu známe délku hrany, po které se Číňané vydali, tak můžeme jít na chvíli spát a nastavit si budík na čas, kdy Číňané dorazí na druhý konec hrany.



Do každého vrcholu umístíme budík, který zazvoní v momentě, kdy do vrcholu dorazí Číňané. Budíky budou na začátku nastaveny na nekonečný čas. Pouze budík v počátečním vrcholu s bude nastaven tak, aby zazvonil okamžitě. Pokud v průběhu algoritmu zjistíme, že do vrcholu dorazíme dříve, než kolik je aktuálně nastavený čas na budíku, tak budík přeřídíme na dřívější čas.

Během algoritmu stačí být vzhůru jen v momentech krátce po té, co zazvoní budík u některého vrcholu. Zbytek času můžeme klidně prospat. Vždy, když nás probudí budík, tak se podíváme do kterých vrcholů dorazili Číňané, na které vstoupili hrany a pokud některá hrana bude zkratkou vedoucí do neprozkoumaného vrcholu, tak přeřídíme odpovídající budík na dřívější čas. Pak už zase můžeme jít spát.

Protože přeřizování budíků probíhá pouze když jsme vzhůru, tak můžeme těsně před usnutím přesně určit, který budík zazvoní jako první.

Zpracování vrcholů v pořadí, jak v nich zvoní budíky, můžeme snadno realizovat použitím prioritní fronty. Na začátku všechny budíky vložíme do prioritní fronty, kde priorita každého budíku je čas zvonění. Postupně budeme budíky z fronty odebírat (v pořadí jak mají zvonit) a zpracovávat události ve vrcholech. Tím jsme právě odvodili Dijkstrův algoritmus.

9.4 Dijkstrův algoritmus

Dijkstrův algoritmus[5] najde nejkratší cestu v orientovaném grafu s nezáporným ohodnocením hran. Už jsme si ho jednou odvodili v předchozí sekci. Nyní si ho ukážeme znova a nezávisle. Zdůrazněme raději ještě jednou, že ohodnocení hran grafu musí být nezáporné, jinak nebudeme moci zaručit správnost algoritmu.

Nejprve si vysvětlíme význam proměnných, které budeme používat. Množina $T \subseteq V$ je množina trvalých vrcholů, to je těch, do kterých známe nejkratší cestu (a do kterých už došli Číňané). Hodnota `d`$[v]$ délka nejkratší cesty z s do v vedoucí jen přes trvalé vrcholy a proto je horním odhadem na vzdálenost z s do v (víceméně odpovídá času, na který je nastaven budík). Do pole `odkud`$[v]$ si ukládáme předchůdce na nejkratší cestě z s do v (odkud jsme do v přišli po nejkratší cestě).

[5] Čte se „Dajkstrův“ algoritmus. Struktura algoritmu je téměř stejná jako v Jarníkově algoritmu pro hledání minimální kostry.

Začneme s prázdnou množinou T, vzdálenost do startu $\mathtt{d}[s]$ nastavíme na 0 a odhady vzdáleností do ostatních vrcholů na nekonečno. Postupně budeme odebírat netrvalé vrcholy s minimálním $\mathtt{d}[v]$ a prohlašovat je za trvalé. Během prohlašování musíme aktualizovat odhady $\mathtt{d}[w]$ u ostatních vrcholů w, protože jsme se do nich mohli dostat zkratkou z nově prohlášeného trvalého vrcholu v.

$T := \emptyset$
$\mathtt{d}[s] := 0$ a $\forall\, v \in V \setminus \{s\}$: $\mathtt{d}[v] := \infty$
$\forall\, v \in V$: $\mathtt{odkud}[v] := nil$
vytvoř prioritní frontu H z vrcholů V s prioritami $\mathtt{d}[v]$
`while` fronta H neprázdná `do`
  $v :=$ `DELETE_MIN`(H) {vyber netrvalý vrchol s nejmenším $\mathtt{d}[v]$}
  $T := T \cup \{v\}$
  `for each` $vw \in E$ `do`
    `if` $\mathtt{d}[w] > \mathtt{d}[v] + c(vw)$ `then`
      $\mathtt{d}[w] := \mathtt{d}[v] + c(vw)$
      $\mathtt{odkud}[w] := v$
      `DECREASE_KEY`(H, w) {aktualizuj haldu}

O realizaci operací `DELETE_MIN` a `DECREASE_KEY` se dočtete v kapitole 8 o haldě.[6] Algoritmus je konečný, protože v každé iteraci prohlásí jeden vrchol za trvalý. Vrcholů je jen n a z každého vede nejvýše n hran.

Pro každý vrchol x, do kterého neexistuje cesta ze startu s, zůstane po skončení algoritmu $\mathtt{d}[x] = \infty$. Proto můžeme už v průběhu algoritmu testovat, jestli je $\mathtt{d}[v]$ vybraného vrcholu v rovno nekonečnu a případně skončit (k žádným změnám už by stejně nedošlo). Podobně pokud hledáme pouze nejkratší cestu do vrcholu t a ne do všech vrcholů, tak můžeme skončit v momentě, kdy bude t prohlášen za trvalý.

Časová složitost Dijkstrova algoritmu odpovídá času potřebnému na provedení $n \times$ `DELETE_MIN` a $m \times$ `DECREASE_KEY`. Stačí tedy dosadit časové složitosti těchto haldových operací. Prioritní frontu můžeme implementovat v poli – pak dostaneme celkový čas $\mathcal{O}(n^2 + m)$, a nebo pomocí binární haldy – dostaneme celkový čas $\mathcal{O}((m + n) \log n)$. Kterou implementaci si máme vybrat? To záleží, jestli je graf hustý (má hodně hran) a nebo řídký (má málo hran). V každém grafu G je $m < n^2$. Pokud je počet hran $m = \Omega(n^2)$, tak je implementace v poli rychlejší. Když počet hran klesne pod $n^2/\log n$, tak už se vyplatí binární halda.

Můžete se ptát, jak to dopadne, pokud použijeme d-regulární haldu, která pro vhodná d zobecňuje obě předchozí řešení. Ale jak zvolit parametr d? Časová složitost Dijkstrova algoritmu s d-regulární haldou je $\mathcal{O}((nd + m)\frac{\log n}{\log d})$. Po chvilce počítání[7] se ukáže, že nejvýhodnější volba je $d \approx m/n$ (průměrný stupeň grafu). Pro velmi řídké grafy s $m = \mathcal{O}(n)$ dostaneme časovou složitost $\mathcal{O}(n \log n)$, stejně jako u řešení s binární haldou. Pro husté grafy s $m = \Omega(n^2)$ dostaneme lineární[8] časovou složitost $\mathcal{O}(n^2)$, stejně jako u řešení s polem. Pro grafy se střední hustotou $m = n^{1+\delta}$ dostaneme také lineární časovou složitost $\mathcal{O}(m/\delta) = \mathcal{O}(m)$.

Zbývá dokázat správnost Dijkstrova algoritmu.

Invariant: *Pro každý trvalý vrchol v je $\mathtt{d}[v]$ délka nejkratší cesty z s do v.*

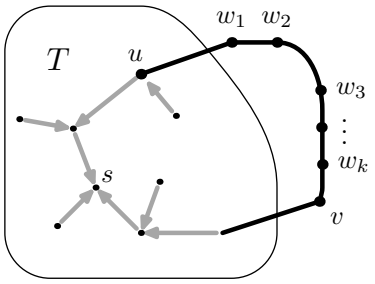
Invariant dokážeme indukcí podle prohlašování vrcholů za trvalé. Pro počáteční vrchol s to platí. Předpokládejme, že v momentě před prohlášením v za trvalý

[6] Pozorný čtenář si mohl všimnout, že jsme změnili parametry u operací pracujících s haldou. V Dijkstrově algoritmu navíc zdůrazňujeme, že pracujeme s haldou H, a přidáme H mezi parametry. Dále u `DECREASE_KEY`(H, w) neuvádíme *okolik* se zmenší klíč $\mathtt{d}[w]$, protože se zmenšení klíče provádí na předchozích řádcích algoritmu.

[7] Tady je vidět, k čemu se hodí matematická analýza – vyšetřování průběhu funkce a hledání minima. Profíci už to ale počítají z hlavy.

[8] Časová složitost je lineární ve velikosti vstupu, tj. velikosti grafu.

vrchol, invariant platí pro všechny vrcholy z množiny T. Ukažme indukční krok tj., že v ten moment je $\mathtt{d}[v]$ délka nejkratší cesty z s do v.

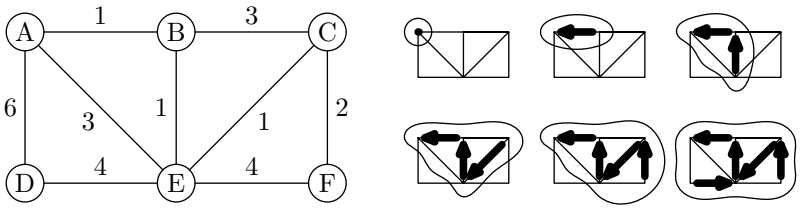


Předpokládejme pro spor, že existuje kratší cesta sPv. Nechť uw_1 je první hrana na cestě P vedoucí z trvalého do netrvalého vrcholu. Taková hrana určitě existuje, protože s je trvalý vrchol, ale v už trvalý není. Z volby vrcholu v jako netrvalého vrcholu s minimálním $\mathtt{d}[v]$ vyplývá $\mathtt{d}[w_1] \geq \mathtt{d}[v]$.

Z indukčního předpokladu je $\mathtt{d}[u]$ délka nejkratší cesty z s do u. Při prohlašování vrcholu u za trvalý jsme zkoušeli snížit hodnotu $\mathtt{d}[w_1]$ na $\mathtt{d}[u] + c(uw_1)$. To se buď povedlo a nebo už byla hodnota $\mathtt{d}[w_1]$ nižší. Proto je $\mathtt{d}[w_1] \leq \mathtt{d}[u] + c(uw_1)$.

Označme další vrcholy na cestě w_1Pv jako $w_2, w_3, \ldots, w_k$. Protože všechny hrany mají nezáporné ohodnocení, tak $c(sPv) = \mathtt{d}[u] + c(uw_1) + c(w_1w_2) + \cdots + c(w_kv) \geq \mathtt{d}[w_1] \geq \mathtt{d}[v]$. A to je spor s tím, že sPv je kratší cesta do v.

Příklad: V následujícím grafu pomocí Dijkstrova algoritmu najdeme nejkratší cestu z vrcholu A do všech ostatních vrcholů. Průběh Dijkstrova algoritmu je zachycen na obrázcích vpravo po řádcích. Množina trvalých vrcholů je zakroužkována. Šipky znázorňují strom nejkratší cesty na trvalých vrcholech.



9.5 Floyd-Warshallův algoritmus

Floyd-Warshallův algoritmus pracuje na orientovaném grafu, který neobsahuje záporné cykly[9] (to nám nezápornost hran bez problémů zaručí), a najde nejkratší orientované cesty mezi každou dvojicí vrcholů. Navíc ze všech cest stejné délky vybere tu s nejmenším počtem hran.

Pokud chceme spočítat vzdálenost každé dvojice vrcholů, tak můžeme n-krát použít Dijkstrův algoritmus (na každý vrchol). Lepší možností je použít Floyd-Warshallův algoritmus, který počítá všechny vzdálenosti přímo, proběhne rychleji než n-krát použitý Dijkstrův algoritmus a ještě se snadněji implementuje. Dokonce je tak jednoduchý, že pokud nám nebude záležet na časové složitosti, ale jen na rychlosti naprogramování, tak je lepší volbou než Dijkstrův algoritmus.

Vrcholy grafu očíslujeme čísly od jedničky do n. Vzdálenosti mezi každou dvojicí vrcholů si budeme ukládat do matice $n \times n$. Celý trik Floyd-Warshallova algoritmu spočívá v tom, že vzdálenosti nepočítáme přímo, ale v n iteracích.

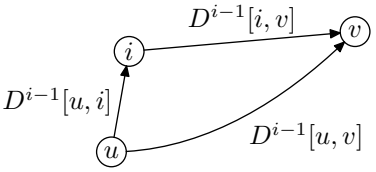
V i-té iteraci spočítáme matici D^i. Hodnota $D^i[u, v]$ je délka nejkratší cesty z u do v, která smí procházet pouze přes vrcholy $\{1, 2, \ldots, i\}$. Jinými slovy $D^i[u, v]$ je délka nejkratší cesty v podgrafu indukovaném vrcholy $\{1, 2, \ldots, i\}$.[10] V nulté iteraci začneme s maticí D^0. Hodnota $D^0[u, v]$ je délka hrany uv, pokud z u vede hrana do v, nula na diagonále a nekonečno jinak. Matice D^0 je tedy *matice vzdáleností* (upravená matice sousednosti, která místo jedniček obsahuje délky hran a místo

[9] Cyklus je záporný, pokud je součet ohodnocení hran podél cyklu záporný.

[10] Podgraf grafu G indukovaný množinou vrcholů $W \subseteq V$ dostaneme tak, že z grafu G vymažeme vrcholy $V \setminus W$ a všechny hrany z nich vedoucí (viz sekce 5.1 o grafových pojmech).

nul mimo diagonálu nekonečna). V poslední iteraci skončíme s maticí D^n, která už bude obsahovat hledané vzdálenosti, protože cesty mezi u a v smí procházet přes všechny vrcholy.

Pozorování 10 $D^i[u,v] = \min\{\, D^{i-1}[u,v]\,,\; D^{i-1}[u,i] + D^{i-1}[i,v]\,\}$



Nejkratší cesta mezi u a v, která smí procházet pouze přes vrcholy $\{1,2,\ldots,i\}$, buď projde přes vrchol i a nebo ne. Pokud nejkratší cesta neobsahuje i, tak je její délka $D^{i-1}[u,v]$. V opačném případě cestu rozložíme na dva úseky—před příchodem do i a po jeho opuštění. Ani jeden z úseků neobsahuje i a tak je délka této cesty $D^{i-1}[u,i] + D^{i-1}[i,v]$.

Mohli byste namítnout, že nemůžeme délky úseků jen tak sečíst, protože oba úseky mohou obsahovat stejný vrchol w. Složení úseků by nebyla cesta, ale jen tah. V tom případě můžeme část tahu mezi oběma výskyty w vypustit (cyklus z w do i a zpět) a dostaneme kratší tah, který už je cestou. Délka tahu se zkrátila, protože graf neobsahuje záporné cykly. Nově vzniklá cesta už neobsahuje i a byla uvažována v prvním případě.

Pozorování 11 *Hodnoty $D^{i-1}[*,i]$ a $D^{i-1}[i,*]$ se v i-té iteraci nezmění. Nebo-li $D^i[*,i] = D^{i-1}[*,i]$ a $D^i[i,*] = D^{i-1}[i,*]$ (hvězdička značí libovolný index).*

Pozorování plyne z předchozího pozorování a faktu, že $D^j[w,w]$ je rovno nule pro každé j a w. K výpočtu $D^i[u,v]$ potřebujeme znát jen hodnoty $D^{i-1}[u,v]$, $D^{i-1}[u,i]$, $D^{i-1}[i,v]$, ale poslední dvě hodnoty se během i-té iterace nezmění. Proto můžeme nové hodnoty $D^i[u,v]$ zapisovat do stejné matice jako předchozí iteraci. Přepsanou položku $D^{i-1}[u,v]$ už nebude během iterace potřebovat. V celém algoritmu si tedy vystačíme jen s jednou maticí $D[*,*]$, do které budeme zapisovat všechny iterace. Floyd-Warshallův algoritmus vypadá následovně:

```
D := D^0      { matice vzdáleností}
for i = 1 to n do
   for u = 1 to n do
   for v = 1 to n do
      if D[u,v] < D[u,i] + D[i,v] then
         D[u,v] := D[u,i] + D[i,v]
```

Časová složitost Floyd-Warshallova algoritmu je $\mathcal{O}(n^3)$.[11] Pokud si chceme zapamatovat kudy nejkratší cesty vedou, tak si v průběhu algoritmu budeme pro každou dvojici u,v pamatovat nejvyšší číslo vrcholu, přes který nejkratší cesta vede (poslední volbu k, která vedla ke zkrácení cesty). Z toho už se dá nejkratší cesta zrekonstruovat pomocí rekurze.

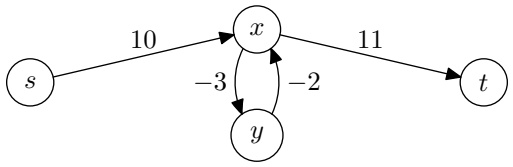
9.6 Obecné ohodnocení hran

V tomto případě už mohou být hrany grafu ohodnoceny čímkoliv, tedy i zápornými čísly. Záporná ohodnocení hran moc neodpovídají délkám hran. Ale ohodnocení si můžeme představit i jako cenu, kterou musíme zaplatit za průchod hranou. Záporná cena znamená, že naopak někdo zaplatí nám.

[11] Protože je struktura algoritmu hodně podobná algoritmu pro násobení matic, můžeme algoritmus urychlit stejným trikem kterým Strassenův algoritmus zrychluje násobení matic. Tím docílíme časové složitosti $\mathcal{O}(n^{\log_2 7}) = \mathcal{O}(n^{2.807})$.

Pokud jsou hrany ohodnoceny i zápornými čísly, tak nemůžeme použít Dijkstrův algoritmus. Je to z toho důvodu, že nejkratší cesta může vést nejdříve do vrcholu, který je dál, a pak se vrátit po záporné hraně.

Ba co hůř, graf může obsahovat záporné cykly. Průchodem po záporném cyklu vyděláme. Je výhodné po něm procházet pořád dokola a postupně snižovat aktuální cenu cesty. Po nekonečně mnoha průchodech ji snížíme až na mínus nekonečno (nekonečně vyděláme). Pokud graf obsahuje záporný cyklus, tak nejlevnější řešení neexistuje. Na druhou stranu nejkratší cesta zcela jistě existuje, protože v konečném grafu je jen konečně mnoho cest. Taková cesta ale nemusí být nejlevnějším řešením naší úlohy. V případě záporných cyklů nemůžeme použít ani Floyd-Warshallův algoritmus, protože by nám mohl vydat nesprávnou odpověď (nenašel by cestu, ale tah, který ani nebude optimální).



9.7 Bellman-Fordův algoritmus

Bellman-Fordův algoritmus najde nejkratší cestu v orientovaném grafu s libovolným ohodnocením hran.

Na chvíli předpokládejme, že graf neobsahuje záporný cyklus. Později si ukážeme, jak rozpoznat, zda ho graf obsahuje.

Budeme postupovat podobně jako v Dijkstrově algoritmu. Pro každý vrchol v si budeme udržovat hodnotu $\mathtt{d}[v]$, která odpovídá délce nějaké cesty vedoucí z s do v (ne nutně té nejkratší). Cena $\mathtt{d}[v]$ je vždy horním odhadem pro cenu nejkratší cesty do v. V průběhu algoritmu budeme tento odhad vylepšovat.

Na začátku u všech vrcholů kromě počátečního vrcholu s nastavíme $\mathtt{d}[v]$ na nekonečno a hodnotu $\mathtt{d}[s]$ na nulu.

Teď si vysvětlíme, co je to update hrany. Pokud víme, že se do vrcholu u umíme dostat za cenu $\mathtt{d}[u]$ a že uv je hrana, tak se umíme dostat do vrcholu v za cenu $\mathtt{d}[u] + c(uv)$ (cestu vedoucí do u prodloužíme o hranu uv). Hrana uv je *korektní,* pokud $\mathtt{d}[v] \leq \mathtt{d}[u] + c(uv)$. V opačném případě je hrana uv *nekorektní* a můžeme zlepšit odhad $\mathtt{d}[v]$. Kontrolu korektnosti hrany uv a případnou opravu zajistí procedura update:

```
procedure update(u,v)
    d[v] := min{ d[v], d[u] + c(u,v) }
```

Důležité vlastnosti updatování hrany jsou:

- Nastaví správnou hodnotu $\mathtt{d}[v]$ v případě, že u je předposlední vrchol na nejkratší cestě do v a hodnota $\mathtt{d}[u]$ už je správně nastavena.
- Nikdy nemůže zmenšit hodnotu $\mathtt{d}[v]$ pod cenu nejkratší cesty do v. V tomto ohledu je použití updatování hrany bezpečné.

Nechť $se_1v_1e_2v_2 \ldots v_{k-1}e_kv_k$ je nejkratší cesta z s do v_k. Podle první vlastnosti updatování hrany postupné zavolání procedury update na hrany e_1, e_2, …, e_k zajistí, že $\mathtt{d}[v]$ bude obsahovat cenu nejkratší cesty z s do v_k. Mezi updatováním jednotlivých hran cesty jsme mohli updatovat ještě i jiné hrany grafu, ale ty výslednou hodnotu $\mathtt{d}[v]$ neovlivní.

Na Dijkstrův algoritmus se můžeme nyní dívat jako na řadu volání procedury update ve správném pořadí. Abychom dostali správný výsledek i v případě záporných hran, musíme používat updatování hran o něco opatrněji.

V jakém pořadí updatovat jednotlivé hrany, abychom updatovali hrany na nejkratší cestě ve správném pořadí? Každá cesta má nejvýše $n-1$ hran. Stačí tedy v $(n-1)$ iteracích updatovat všechny hrany grafu. Jinými slovy posloupnost

$$\underbrace{e_1\, e_2\, \ldots\, e_m\, e_1\, e_2\, \ldots\, e_m\, e_1\, \ldots \qquad\qquad \ldots\, e_m}_{(n-1)\times}$$

má $(n-1)m$ hran a obsahuje každou posloupnost $n-1$ hran jako podposloupnost. Tím pádem updatujeme každou posloupnost $n-1$ hran ve správném pořadí. Celkem se provede $\mathcal{O}(nm)$ updatů. Tomuto algoritmu se říká Bellman-Fordův algoritmus.

$\forall\, v \in V(G)$: $\mathtt{d}[v] := \infty$
$\mathtt{d}[s] := 0$
`for` $i := 1$ `to` $n-1$ `do`
  `for all` $e \in E(G)$ `do`
    update(e)

Poznámka k implementaci: v celé řadě případů obsahuje nejkratší cesta mnohem méně než $n-1$ hran. Proto můžeme skončit po méně jak $n-1$ iteracích. Z toho důvodu se vyplatí v každé iteraci zjišťovat, jestli byla některá hrana nekorektní a tedy jestli vůbec proběhl update nějaké hrany. Pokud ne, tak jsou všechny hrany grafu korektní, algoritmus by v další iteraci také neprovedl žádný update a proto můžeme skončit.

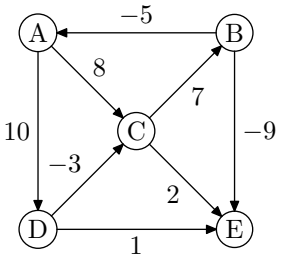
Detekce záporných cyklů

Na začátku jsme předpokládali, že graf neobsahuje záporný cyklus. Ale co když ano? Jak to poznáme? Ukázali jsme si, že když v grafu nebude záporný cyklus, tak Bellman-Fordův algoritmus najde nejkratší cestu a skončí v momentě, kdy jsou všechny hrany korektní.

Pokud graf obsahuje záporný cyklus, tak bude v grafu neustále existovat nekorektní hrana (podél cyklu můžeme odhady na cenu vylepšovat do nekonečna).

Proto stačí použít Bellman-Fordův algoritmus a po jeho skončení otestovat, jestli jsou všechny hrany grafu korektní. Pokud ano, tak algoritmus nalezl nejkratší cestu, pokud ne, tak nejkratší řešení neexistuje.

Příklad: V následujícím ohodnoceném grafu najdeme nejkratší cesty z vrcholu A do všech ostatních vrcholů. Hrany máme zadané v pořadí AC, AD, BA, BE, CB, CE, DC, DE. Průběh Bellman-Fordova algoritmu je částečně zachycen tabulkou, která pro každý vrchol v obsahuje hodnoty $\mathtt{d}[v]$ na konci každé iterace.



Vrchol	Iterace				
	0	1	2	3	4
A	0	0	0	0	0
B	∞	15	14	14	14
C	∞	7	7	7	7
D	∞	10	10	10	10
E	∞	10	6	5	5

9.8 Acyklické orientované grafy

Acyklické orientované grafy nám už z definice zaručují, že nebudou obsahovat záporný cyklus. Neobsahují totiž žádný cyklus. Proto v nich najdeme nejkratší cestu

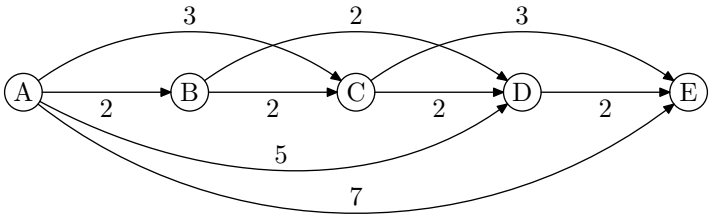
následujícím jednoduchým způsobem:

1. Nalezneme topologické uspořádání.
2. Procházíme vrcholy v nalezeném pořadí a updatujeme hrany z nich vedoucí.

První fázi vyřešíme pomocí průchodu do hloubky. Druhou jednoduchým průchodem vrcholů. Tím pádem zvládneme obě fáze v čase $\mathcal{O}(n+m)$ a celý algoritmus poběží v lineárním čase. Připomeňme, že hrany mohou být ohodnoceny libovolně. Tedy i zápornými čísly.

O Dijkstrově algoritmu se dá říci, že funguje podobně jako hledání nejkratší cesty v acyklických orientovaných grafech, akorát uspořádání vrcholů podle nejkratší cesty hledáme za chodu.

Poznamenejme ještě, že v acyklických orientovaných grafech můžeme podobným způsobem hledat i nejdelší cestu. Takovému algoritmu se říká *algoritmus kritické cesty*. Byl pojmenován kvůli následující aplikaci. Acyklický orientovaný graf popisuje závislosti mezi činnostmi projektu. Hrany odpovídají činnostem a ohodnocení hran době, jak dlouho bude daná činnost probíhat. Doba potřebná k dokončení projektu je délka nejdelší cesty. Nejdelší cestě se říká kritická, protože každé zpoždění činností na kritické cestě způsobí zpoždění celého projektu. **Příklad:** Na následujícím ohodnoceném grafu $G = (V, E)$ najděte nejkratší cestu z A do všech ostatních vrcholů.



Délky nejkratších cest z A do A, B, C, D, E jsou 0, 2, 3, 4, 6. Na papíře můžeme vzdálenosti počítat rovnou. Procházíme vrcholy zleva doprava (v topologickém pořadí) a pro každý vrchol v spočítáme podle hran do něj vedoucích jeho $\mathtt{d}[v] := \min\{\mathtt{d}[w] + c(wv) \mid wv \in E\}$.

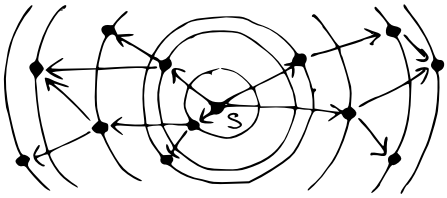
9.9 Potenciál

Ve fyzice je potenciál fyzikální veličina, která popisuje potencionální energii v poli. Například elektrický potenciál popisuje energii v elektrickém poli. Potenciálový rozdíl mezi místy A, B je množství práce, které musíme vykonat, abychom přenesli jednotkový náboj z místa A do místa B.[12] Nezáleží na cestě, kudy náboj přenášíme, ale jen na výchozím a cílovém bodě.

Potenciál a vzdálenost v grafu od počátečního vrcholu s mají hodně společného. Když jdeme po hraně směrem od s, tak vzdálenost roste. Když půjdeme na opačnou stranu, tak klesá. Když vyjdeme z vrcholu v na procházku, tak se vzdálenost bude měnit, ale po návratu do v bude stejná jako na začátku.

Označme délku nejkratší cesty z s do v pomocí y_v. Pro každou hranu vw je $y_v + c(vw) \geq y_w$, jinak můžeme cestu z s do w zkrátit (hrana vw musí být korektní). Tím se necháme inspirovat.

[12] Potenciálový rozdíl v elektrickém poli se nazývá napětí.



Definice: Je dán orientovaný graf $G = (V, E)$ s ohodnocením hran $c(e)$ a s počátečním vrcholem s. Vektor $y = (y_v : v \in V)$ je *přípustný potenciál* pro graf G, ohodnocení c a počátek s, pokud splňuje

$$\begin{array}{ll} (i) & y_v + c(vw) \geq y_w \quad \text{pro každou hranu } vw \in E \\ (ii) & y_s = 0. \end{array}$$

Hladinu nulového potenciálu si můžeme nastavit jak chceme, protože i po přičtením stejné konstanty ke každému y_v zůstane první podmínka platná. Druhá podmínka říká, že si hladinu nulového potenciálu nastavíme tak, aby byl v počátečním vrcholu nulový.[13]

Pro lepší představu o potenciálu si představíme následující gumičkovou realizaci. Místo vrcholů vezmeme kuličky a pokud jsou vrcholy spojeny hranou délky $c(vw)$, tak je spojíme gumičkou, která se natáhne nejvýše do délky $c(vw)$. Při roztažení do větší délky praskne. Vrcholy přišpendlíme na vysoký sloup do takové výšky, kolik je y_v. Potenciál je přípustný, pokud žádná gumička nepraskne ($y_w - y_v \leq c(vw)$) a vrchol s bude umístěn ve výšce 0.

Pozorování 12 *Přípustný potenciál je dolním odhadem na délku nejkratší cesty.*

Pro každou hranu vw platí $c(vw) \geq y_w - y_v$. Posčítáním těchto odhadů podél libovolné cesty $sPv = v_0 e_1 v_1 \ldots e_k v_k$, kde $s = v_0$ a $v = v_k$, dostaneme

$$c(sPv) = \sum_{i=1}^{k} c(e_i) \geq \sum_{i=1}^{k} y_{v_i} - y_{v_{i-1}} = y_{v_k} - y_{v_0} = y_v.$$

Dokonce jsme ukázali víc. Ukázali jsme, že potenciálový rozdíl $y_w - y_v$ je dolním odhadem na délku libovolné cesty z v do w.

Pozorování 13 *Graf má přípustný potenciál právě tehdy, když neobsahuje záporný cyklus.*

První implikace je jednoduchá. Předpokládejme, že graf má přípustný potenciál y a obsahuje záporný cyklus C. Podle předchozího pozorování je potenciálový rozdíl dolním odhadem na délku cesty. Z toho dostaneme odhad na délku cyklu $c(C) = \sum_{e \in C} c(e) \geq 0$. To je spor s tím, že má cyklus zápornou délku.

K důkazu druhé implikace můžeme použít Bellman-Fordův algoritmus. Ten nám najde hodnoty y_v (tj. délky nejkratších cest do všech vrcholů) právě tehdy, když graf neobsahuje záporný cyklus.

Úprava ohodnocení grafu pomocí potenciálu

Máme graf G, u kterého známe přípustný potenciál. Vytvoříme si nové ohodnocení grafu $c' : E \to \mathbb{R}$, které definujeme jako $c'(vw) = c(vw) + y_v - y_w$. Nové ohodnocení c' nezmění nejkratší cesty, protože pro každou cestu uPv platí $c'(uPv) = c(uPv) +$

[13] Zavedení potenciálu je zcela přirozené, protože hledání maximálního přípustného potenciálu je v lineárním programování duálním problémem k hledání nejkratší cesty.

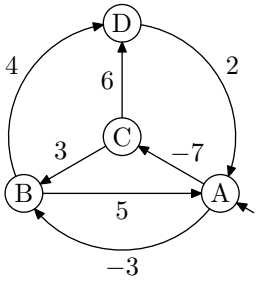
$y_u - y_v$ a potenciálový rozdíl mezi u a v je pro všechny cesty stejný. Pouze se změní ceny nejkratších cest.

Proto můžeme nejkratší cestu hledat i v grafu G s ohodnocením c' a nalezená nejkratší cesta bude stejná jako v grafu s původním ohodnocením.

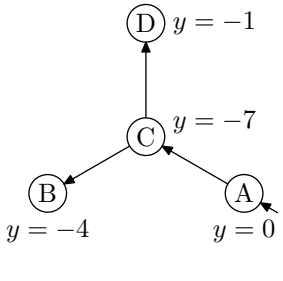
Důležitá poznámka: Pro tuto úvahu jsme používali pouze vlastnost (i) z definice přípustného potenciálu. Hladina nulového potenciálu mohla být nastavena libovolně.

Když známe přípustný potenciál, tak se můžeme zbavit záporných hran tím, že ke hranám přičteme vhodný násobek potenciálových rozdílů konců hran. Na graf s novým ohodnocením, tj. bez záporných hran, už můžeme použít Dijkstrův algoritmus.[14]

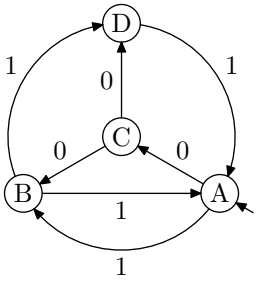
Příklad: V následujícím grafu chceme spočítat nejkratší cestu z A do všech ostatních vrcholů. V grafu se zápornými hranami (levý obrázek) spočítáme přípustný potenciál pomocí Bellman-Fordova algoritmu (vzdálenost vrcholu od A je přípustným potenciálem). Jako výstup dostaneme strom nejkratší cesty a přípustný potenciál (prostřední obrázek). Výše popsaným způsobem upravíme ohodnocení hran a dostaneme graf na pravém obrázku. Graf s novým ohodnocením má stejný strom nejkratší cesty jako graf s původním ohodnocením.



Původní graf se záporným ohodnocením.



Strom nejkratší cesty a přípustný potenciál.



Graf s upraveným ohodnocením bez záporných hran.

Jak najít přípustný potenciál? Můžeme si ho spočítat Bellman-Fordovým algoritmem. To už ale samo obnáší nalezení nejkratší cesty. Zkusme to ještě jinak.

Pokud náš graf odpovídá silniční síti, tak můžeme za přípustný potenciál ve vrcholech zvolit jejich vzdálenost od startu vzdušnou čarou.

A teď otázka pro vás. Můžeme za přípustný potenciál zvolit vzdálenost vzdušnou čarou od libovolného pevného vrcholu? Ano, můžeme. Bod (i) z definice přípustného potenciálu je splněn. Abychom splnili bod (ii), tak musíme nastavit hladinu nulového potenciálu tak, aby byla ve vrcholu s nulová. Toho docílíme přičtením vhodné konstanty (jaké?). Jak jsme si ale ukázali při úpravě ohodnocení grafu pomocí potenciálu, pro nalezení nejkratších cest není hladina nulového potenciálu podstatná. Můžeme si jí zvolit libovolně, jen už pak nemůžeme mluvit o přípustném potenciálu.

V následujících odstavcích si vysvětlíme, proč je nejlepší zvolit za potenciál vzdálenost od cíle t.

Heuristika pro Dijkstrův algoritmus

Dijkstrův algoritmus hledá nejkratší cestu rovnoměrně na všechny strany – prochází všechny vrcholy v pořadí určeném vzdáleností od počátku. Tedy i když hledáme cestu z Prahy do Brna, tak ve „vlně“ kolem 120 kilometrů spočítáme nejkratší

[14] Využití viz Johnsonův algoritmus na straně 115.

cestu do Jihlavy i do Karlových Varů. Přitom nám bude pouhým pohledem do mapy jasné, že je výpočet nejkratší cesty do Karlových Varů zbytečný, protože od Prahy leží na opačné straně než Brno. I kdybychom z Karlových pokračovali do Brna vzdušnou čarou, tak to bude mnohem více kilometrů než po nejkratší cestě z Prahy do Brna. Proto chceme Dijkstrovu algoritmu pomoci tak, aby nejdříve hledal nejkratší cestu do vrcholů správným směrem.

Co kdybychom v Dijkstrově algoritmu počítali místo $\mathtt{d}[v]$ se vzdáleností $\mathtt{d}'[v] := \mathtt{d}[v]$+vzdálenost z v do cíle vzdušnou čarou. Pak bychom nejkratší cestu do Jihlavy spočítali dříve než do Karlových Varů. Vrcholy poblíž cíle by byly zvýhodněny. Ale bude to fungovat? Ano, k $\mathtt{d}[v]$ jsme přičetli něco podobného přípustnému potenciálu, až na nastavení hladiny nulového potenciálu, ale to není pro hledání nejkratších cest potřeba.

Nyní celý algoritmus zopakujeme. Nechť G je graf, který odpovídá silniční síti. Vzdálenost vrcholů od cíle vzdušnou čarou je téměř přípustným potenciálem (až na nastavení hladiny nulového potenciálu). Nejkratší cestu budeme hledat v grafu G s ohodnocením c', které je upraveno podle potenciálu. To nám zajistí, že vlny, ve kterých Dijkstra prohledává vrcholy, budou trochu zdeformovány a protaženy správným směrem.

9.10 Dálniční hierarchie

Představme si, že graf G odpovídá silniční síti celé Evropy a má skoro milión vrcholů. Pokud chceme najít nejkratší cestu z Londýna do Budapešti, tak můžeme použít například Dijkstrův algoritmus. Ale co když se budeme na ptát na nejkratší cestu hodně často? Nemůžeme si něco předpočítat nebo nemůžeme použít nějakou fintu, abychom odpověď nalezli rychleji než Dijkstrovým algoritmem?

Co když chceme realizovat plánovač tras? To je internetový server, kterému zadáte odkud a kam jedete a on vám na mapě najde nejkratší cestu. Dokonce vám i vypíše instrukce, kam máte na které křižovatce odbočit. Takový server dostane během jediné vteřiny stovky dotazů a musí na ně stihnout odpovědět.

Prakticky se ukazuje, že když chcete jen někam hodně daleko, tak na začátku pojedete po lokálních silnicích. Pak najedete na dálnici a pojedete po ní skoro až do cíle.[15] Poblíž cíle sjedete z dálnice a do cíle dojedete po lokálních silnicích.

Nyní využijeme předchozího pozorování o dálnicích a vysvětlíme si hlavní myšlenku dálničních hierarchií. V České republice je mnoho silnic, ale jen málo hraničních přechodů. Pokud budeme přes Českou republiku jen projíždět, tak nemusíme znát všechny silnice, ale stačí znát nejkratší cesty mezi libovolnými hraničními přechody. Ty můžeme mít předpočítané.

Když budeme hledat nejkratší cestu z Londýna do Budapešti, tak ji nebudeme hledat v celé silniční síti Evropy (to je příliš velký graf), ale ve 3 fázích a v mnohem menších grafech. V první fázi najdeme nejkratší cesty z Londýna na hraniční přechody Velké Británie. Ve druhé fázi najdeme nejkratší cesty z hraničních přechodů Velké Británie do hraničních přechodů Maďarska. To hledáme ve zjednodušeném grafu evropské silniční sítě, který jako vrcholy obsahuje pouze hraniční přechody. Ve třetí fázi najdeme nejkratší cesty z hraničních přechodů Maďarska do Budapešti. Z výsledků všech 3 fázích poskládáme acyklický orientovaný graf, který obsahuje pouze Londýn, Budapešť a hraniční přechody Velké Británie a Maďarska. Ten už má jen pár vrcholů a navíc je acyklický, proto v něm nejkratší cestu můžeme najít v lineárním čase. Grafy ve všech fázích obsahují mnohem méně vrcholů, než silniční síť celé Evropy. Díky tomu dosáhneme zrychlení.

[15] Předpokládáme, že v Evropě existuje kvalitní dálniční síť. Pokud ne, tak za dálnici prohlásíme i významnou a hodně frekventovanou silnici.

Hierarchie může mít i více úrovní. Celou Evropu se můžeme rozdělit na oblasti podle států, státy na oblasti podle krajů... Oblasti vůbec nemusí odpovídat právnímu rozdělení. Jde jen o to, aby každá oblast měla málo vstupních míst, tak zvaných portů (obdoba hraničních přechodů). Čím méně jich bude mít, tím více se zjednoduší graf obsahující pouze porty a hrany mezi nimi.

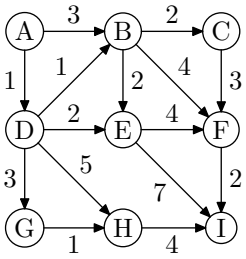
Problém dálničních hierarchií spočívá v tom, jak nalézt ty správné dálnice. Jak najít ty správné oblasti s malým počtem vstupních míst. To už je malinko složitější a je velké množství různých heuristik, které to řeší.[16]

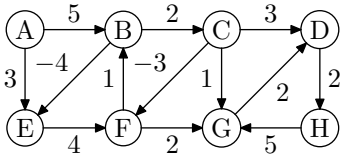
Pro ilustraci časových úspor uvedeme příklad. Na grafu evropské silniční sítě trvá Dijkstrův algoritmus řádově desítky minut. Stejný dotaz řešený pomocí dálničních hierarchií trvá méně jak milisekundu.

9.11 Příklady

9.11.1 Přímé procvičení vyložených algoritmů

1. V následujících grafech najděte nejkratší cestu z vrcholu A do všech ostatních vrcholů. Pokud byste během algoritmu měli na výběr z několika vrcholů, tak si vyberte abecedně menší vrchol. V každém kroku algoritmu (v každé iteraci) vypište hodnoty $\mathtt{d}[v]$ pro všechny vrcholy v. Nakreslete strom nejkratší cesty.





   (a) Použijte Dijkstrův algoritmus.

   (b) Použijte Bellman-Fordův algoritmus.

   (c) Využijte toho, že první graf jde topologicky uspořádat. V prvním grafu nalezněte nejkratší cestu pomocí algoritmu pro acyklické orientované grafy.

2. Máme orientovaný graf, jehož jediné záporně ohodnocené hrany jsou ty vedoucí z počátku. Ostatní hrany mají kladné ohodnocení. Bude v takovém grafu fungovat Dijkstrův algoritmus? Dokažte.

3. Profesor Všezlepšil navrhuje následující algoritmus na hledání nejkratší cesty z s do všech ostatních vrcholů v orientovaných grafech s obecným ohodnocením hran (tedy i záporným): Vezmeme dostatečně velkou konstantu a přičteme ji k ohodnocení každé hrany. Tím získáme nezáporné ohodnocení hran. Potom už můžeme použít Dijkstrův algoritmus. Nalezená nejkratší cesta bude stejná jako nejkratší cesta v grafu s původním ohodnocením.

   Dokažte, že navrhovaná metoda funguje, nebo nalezněte protipříklad.

[16] O silničních sítích je známo, že mají poměrně malou *treewidth (stromovou šířku)*. To si můžeme představit následovně. Pomocí jedné hranice s malým počtem portů rozdělíme graf na dvě poměrně velké části. Každou část můžeme rekurzivně dělit stejným způsobem, dokud nezbudou jen malinkaté části. Nepřesně řečeno, stromová šířka grafu je číslo omezující počet portů na každé hranici takové, aby šel graf ještě rekurzivně rozdělit.

4. (Floyd-Warshall a záporné cykly) Ukázali jsme si, že Floyd-Warshallův algoritmus nalezne nejkratší cestu mezi každou dvojicí vrcholů, pokud graf neobsahuje záporný cyklus. Co by se stalo, kdyby graf obsahoval záporný cyklus?

   Když dostaneme graf, o kterém nevíme, jestli obsahuje záporný cyklus. Nemůžeme pomocí Floyd-Warshallova algoritmu spočítat nejkratší cesty a nebo odpovědět, že graf určitě obsahuje záporný cyklus? Vymyslete, podle čeho to jednoduše poznat.

9.11.2 Varianty problému nejkratší cesty

1. Dostanete graf a ohodnocení jeho hran. Vaším úkolem je nalézt cestu z vrcholu s do vrcholu t takovou, aby největší ohodnocení hrany na cestě bylo co nejmenší. Tj. vzdálenost t od s po cestě sPt počítáme jako $\max_{e \in P}\{c(e)\}$.

2. Dostanete graf jehož hrany jsou ohodnoceny kladnými reálnými čísly. Délka cesty se počítá jako součin ohodnocení hran ležících na cestě, to je $\prod_{e \in sPt} c(e)$.

   (a) Najděte nejkratší cestu z vrcholu s do vrcholu t.
   Nápověda: Zkuste převést násobení na sčítání.

   (b) Profesor Všezlepšil povídá, že hledání nejkratší cesty není nic složitého. Navrhuje použít Dijkstrův algoritmus, kde nahradíme sčítání násobením (výpočet $\mathtt{d}[u] + c(uv)$ nahradíme výpočtem $\mathtt{d}[u] \cdot c(uv)$). Najděte panu profesorovi protipříklad demonstrující, že to tak jednoduše nejde.[17]

3. Hledání nejkratší cesty podle několika kritérií je zcela přirozené. Když jedeme někam autem, tak se tam chceme dostat co nejrychleji, ale také chceme najezdit co nejméně kilometrů, abychom zaplatili co nejméně za benzín.

   (a) Dostanete graf jehož hrany jsou ohodnoceny uspořádanou dvojicí (a, b). Najděte cestu z vrcholu s do vrcholu t takovou, aby byla nejkratší podle ohodnocení a. Pokud by takových cest existovalo více, tak z nich vyberte tu, která je kratší podle ohodnocení b. Délku cesty počítáme jako součet ohodnocení hran na cestě po složkách.

   (b) Co kdybychom neměli hrany ohodnocené jen dvojicí, ale trojicí hodnot? Hledáme cestu, která je nejkratší nejprve podle ohodnocení a, u cest se stejnou vzdáleností podle a vybereme tu s nejmenší vzdáleností podle b a ze všech cest se stejnou nejmenší vzdáleností podle a i b vybereme tu s nejmenší vzdáleností podle c.

4. Celé Norsko je podkopáno ohromným množstvím tunelů. Před každým tunelem je dopravní značka říkající, jak vysoké auto tunelem projede. Firma sídlící ve městě X rozváží celkem velké, ale hlavně vysoké zakázky, a má s tím pěkný problém. Dobře znají silniční síť a pro každou silnici vědí, jak vysoký náklad tudy provezou a jak je silniční úsek dlouhý.

   (a) Firma zná výšku nákladu naloženého na nákladňáku. Nejprve by je zajímalo, jestli vůbec mohou takto vysoký náklad dopravit do města Y. Pokud ano, tak by je zajímala nejkratší cesta ze skladu X do cílového místa Y.

   Samozřejmě mohou jezdit jen tak, aby projeli všemi tunely na nalezené cestě a žádný neponičili.

[17] Student se přihlásí a povídá: „Pane profesore, to lemma, co dokazujete, neplatí. Mám protipříklad.“ Profesor se tím nenechá rozhodit a odpoví: „To nevadí. Mám 2 důkazy.“

(b) Jaký nejvyšší náklad lze dopravit po silnicích z města X do města Y? Pochopte, že čím vyšší zakázku firma vyrobí, tím více vydělá. Na druhou stranu firma musí být schopna dopravit zakázku na místo určení.

5. Jedete na dovolenou do Norska, protože za vyřešení předchozí úlohy vám norská firma zaplatila dovolenou. Víte odkud vyjíždíte, víte kam jedete a znáte silniční síť. O každé silnici víte, jak je dlouhá a jak rychle po ní lze jet. Můžete si tedy spočítat, za jak dlouho ji projedete. Najděte takovou cestu do cílového místa, abyste tam dorazili co nejrychleji plus mínus 15min a za druhé, abyste najeli co nejkratší vzdálenost.

6. Mezi N městy označenými čísly 1 až N jezdí autobusové linky. Pro každou autobusovou linku dostanete údaje odkud a kam jezdí, a cenu jízdného.

   (a) Vypište všechna města, do kterých se lze dostat z města 1 na nejvýše 2 přestupy.

   (b) Zjistěte, jak se co nejlevněji dostat z města 1 do města N. Vypište cenu a návod jak přestupovat. Pokud by existovalo více nejlevnějších cest, tak vypište tu s nejmenším počtem přestupů.

   (c) Navrhněte program pro autobusovou informační kancelář. Do kanceláře volají jednotliví cestující a ptají se, jak se nejlevněji dostanou z města X do města Y. Chtěli bychom jim co nejrychleji odpovědět. Protože jsme slušná informační kancelář, tak v případě, kdy existuje více nejlevnějších cest, chceme zákazníkovi předložit tu s nejmenším počtem přestupů.

   (Jiná formulace problému je, že dostane d dotazů a máte na ně co nejrychleji odpovědět.)

7. (Nejspolehlivější cesta) Máme rozlehlou počítačovou síť, která je realizována rádiovým spojením. Rádiové spojení může být rušeno jiným vysíláním a tudíž není moc spolehlivé. Síť si reprezentujeme jako orientovaný graf $G = (V, E)$. Ke každé hraně $e \in E$ dostanete ještě číslo $0 \leq p(e) \leq 1$, které můžeme interpretovat jako spolehlivost hrany. Číslo $p(e)$ je pravděpodobnost, že informace poslaná z vrcholu x pro hraně $e = xy$ dorazí do y v pořádku (nedojde k chybě). Pokud budeme informace posílat po dvou zřetězených hranách $e, f \in E$, kde $e = xy$ a $f = yz$, tak je pravděpodobnost, že informace vyslané z x dorazí do z v pořádku, rovna $p(e) \cdot p(f)$. Pravděpodobnosti na cestě se násobí. Najděte v zadané síti nejspolehlivější cestu z vrcholu s do vrcholu t. Nejspolehlivější cesta je ta s nejmenší pravděpodobností chyby.

8. Vrcholy grafu $G = (V, E)$ reprezentují města a hrany reprezentují silnice jednoho podivného království. Silnice vedou vždy z města do města a jinde se nekříží. Pro každou silnici znáte její délku v kilometrech. Máte auto, které má nádrž na L litrů benzínu. Všechna auta v tomto království mají stejnou spotřebu σ litrů na 100km (okamžitá spotřeba je konstantní, ať jezdíte jak jezdíte). Takže s plnou nádrží ujedete nejvýše K kilometrů. Pak vám dojde benzín. Dopravu v tomto království komplikuje to, že jsou benzínové stanice pouze ve městech. Na silnicích žádné nejsou. Proto se při svých cestách musíte omezit na silnice kratší než Kkm.

   (a) Navrhněte lineární algoritmus, který zjistí, jestli se svým autem můžete dojet z města s do města t.

   (b) Bohužel jste zjistili, že se svým starým autem moc daleko nedojedete. Chcete si proto koupit nové auto. Jaká musí být jeho minimální velikost nádrže, abyste byli schopni dojet z s do t? Zkuste vymyslet algoritmus pracující v čase $\mathcal{O}((n + m) \log n)$.

9.11.3 Další algoritmy a speciální případy

1. Dostanete graf jehož hrany jsou ohodnoceny čísly 1, 2, ..., k. Najděte co nejrychleji nejkratší cestu v tomto grafu.

   (a) Dokážete to v čase $\mathcal{O}(n + km)$? Pro začátek můžete zkusit přemýšlet o případu, kde $k = 2$.

   (b) A zvládnete najít řešení pracující v čase $\mathcal{O}((n + m) \log k)$?
   Nápověda: Kolik různých odhadů vzdálenosti `d`$[v]$ může být v Dijkstrově algoritmu mezi netrvalými vrcholy?

2. Dostanete orientovaný graf s obecným ohodnocením hran (tedy i se záporným). Máte zaručeno, že nejkratší cesta mezi libovolnou dvojicí vrcholů obsahuje nejvýše k hran. Navrhněte algoritmus, který pro dva vrcholy u, v najde nejkratší cestu z u do v v čase $\mathcal{O}(km)$.

   Nápověda: Bellman-Ford.

3. (Yenovo vylepšení Bellman-Fordova algoritmu) Nechť v_1, ..., v_n je nějaké uspořádání vrcholů V takové, že $v_1 = s$. Hrany E rozdělíme do dvou skupin E_1 a E_2, kde $E_1 = \{v_i v_j \,|\, i < j\}$ a $E_2 = \{v_i v_j \,|\, i > j\}$ (hrany po směru a proti směru uspořádání). Ani jedna skupina z E_1, E_2 nemůže obsahovat orientovaný cyklus (rozmyslete si proč). Dá se tedy říci, že rozdělení hran do skupin odpovídá rozdělení grafu na dva acyklické podgrafy G_1, G_2.

   Nyní uspořádáme E_1 do posloupnosti $\mathcal{S}_1$ tak, aby hrana $v_i v_j$ předcházela hraně $v_k v_\ell$ pokud $i < k$. Podobně uspořádáme E_2 do posloupnosti $\mathcal{S}_2$ tak, aby hrana $v_i v_j$ předcházela hraně $v_k v_\ell$ pokud $i > k$. Hrany E_1 jsou v posloupnosti $\mathcal{S}_1$ v takovém pořadí, abychom postupným voláním procedury `update()` na hrany z $\mathcal{S}_1$ našli v podgrafu G_1 nejkratší cestu vedoucí z s do všech dostupných vrcholů.

   Nejkratší cesta v G ale může využívat zkratek přes hrany z E_2. Proto pro nalezení nejkratší cesty v celém grafu G použijeme Bellman-Fordův algoritmus s posloupností hran $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_1, \mathcal{S}_2, \ldots$ Co můžeme říci o potřebném počtu iterací? Jaká bude časová složitost vylepšeného algoritmu?

4. (Johnsonův algoritmus; využití přípustného potenciálu) Dostaneme orientovaný graf, jehož hrany jsou ohodnoceny reálnými čísly (i zápornými). Chtěli bychom pro každou dvojici vrcholů (x, y) najít nejkratší cestu z x do y. Dopředu nevíme, jestli graf obsahuje záporný cyklus.

   Mohli bychom n krát použít Bellman-Fordův algoritmus (celkový čas $\mathcal{O}(n^2 m)$). Nebo bychom mohli pomocí Bellman-Fordova algoritmu zjistit, jestli graf obsahuje záporný cyklus a pak použít Floyd-Warshallův algoritmus (celkový čas $\mathcal{O}(n^3)$). Třetí možností je následující řešení:

   (a) Pomocí Bellman-Fordova algoritmu najdeme nejkratší cestu z libovolného vrcholu do všech ostatních. Při tom spočítáme přípustný potenciál (nebo odpovíme, že neexistuje a skončíme).

   (b) Pomocí přípustného potenciálu upravíme ohodnocení hran tak, aby bylo všude nezáporné. (Podívejte se na příklad na straně 110.)

   (c) Nakonec použijeme $(n-1)$ krát Dijkstrův algoritmus na graf s upraveným ohodnocením.

   Rozmyslete si detaily, zdůvodněte správnost algoritmu a určete jeho celkovou časovou složitost.

5. (Gabow's scaling algorithm) Scaling algorithm se do češtiny překládá jako algoritmus měnící měřítko. Algoritmus se dá použít pouze pro grafy, jejichž hrany jsou ohodnoceny celým číslem (integerem). V první iteraci algoritmus zváží pouze nejvyšší bit na každé hraně a vyřeší tento zjednodušený problém (ohodnocení hran jsou jednobitové). Ve druhé iteraci přidá další bit. Za pomoci výsledků z předchozí fáze vyřeší zjednodušený problém, kde u každé hrany zvažuje pouze 2 nejvyšší bity. V každé další iteraci přidá další bit z ohodnocení na hranách, až se nakonec dostane ke všem bitům a tím spočítá řešení původní úlohy.

   Dostaneme orientovaný graf $G = (V, E)$ s nezáporným ohodnocením hran $w(e)$. Chceme najít délky nejkratších cest z počátečního vrcholu s do všech ostatních vrcholů v. Nechť W označuje největší hodnotu, kterou je ohodnocena nějaká hrana. Chceme vymyslet algoritmus, který bude pracovat v čase $\mathcal{O}(|E| \cdot \log W)$.

   Algoritmus začne s grafem G, který má hrany ohodnoceny pouze nejvyšším bitem z původních ohodnocení. Postupně v dalších iteracích přidává další bity. Konkrétně, nechť $k := \lceil \log(W+1) \rceil$ je počet bitů v binární reprezentaci každé hodnoty na hraně. V každé iteraci pro $i = 1, 2, \ldots, k$ algoritmus uvažuje ohodnocení hran $w_i(e) := \lfloor w(e)/2^{k-i} \rfloor$ pro každou $e \in E$. Hodnota $w_i(e)$ je původní hodnota $w(e)$, která má snížené měřítko a obsahuje pouze nejvyšších i bitů. Proto je $w_k(e) = w(e)$ pro každou hranu e. Například pokud $k = 5$ a $w(e) = 25$, což je v binární reprezentaci $[11001]_2$, tak $w_3(e) = [110]_2 = 6$.

   Definujme $\delta_i(u, v)$ jako délku nejkratší cesty z u do v v grafu G s ohodnocením w_i. Délka nejkratší cesty v G s ohodnocením w $\delta(u, v) = \delta_k(u, v)$ pro všechny $u, v \in V$. Pro zadaný počáteční vrchol s algoritmus nejprve spočítá $\delta_1(s, v)$ pro všechny $v \in V$. V dalších iteracích spočítá $\delta_2(s, v)$ pro všechny $v \in V$, dále $\delta_3(s, v)$ pro všechny $v \in V$,..., až se v poslední iteraci spočte $\delta_k(s, v)$ pro všechny $v \in V$. Celou dobu budeme předpokládat, že $|E| \geq |V| - 1$. Pokud bude výpočet δ_i z δ_{i-1} trvat čas $\mathcal{O}(|E|)$, tak celý algoritmus poběží v čase $\mathcal{O}(|E| \cdot \log W)$.

   Nyní si odvodíte, proč a jak algoritmus funguje. Pomohou vám úkoly a nápovědy v následujících bodech.

   (a) Předpokládejte, že $\delta(s, v) \leq |E|$ pro všechny $v \in V$. Ukažte, jak se dá spočítat $\delta(s, v)$ pro všechny $v \in V$ v čase $\mathcal{O}(|E|)$.

   (b) Ukažte, jak spočítat $\delta_1(s, v)$ pro všechny $v \in V$ v čase $\mathcal{O}(|E|)$. Tato iterace odpovídá hledání nejkratších cest v grafu bez ohodnocení hran.

   Nyní se zaměříme na výpočet δ_i z δ_{i-1}.

   (c) Dokažte, že pro $i = 1, 2, \ldots, k$, buď $w_i(e) = 2w_{i-1}(e)$ nebo $w_i(e) = 2w_{i-1}(e) + 1$. Potom dokažte, že pro všechna $v \in V$

   $$2\delta_{i-1}(s, v) \leq \delta_i(s, v) \leq 2\delta_{i-1}(s, v) + |V| - 1.$$

   (d) Pro všechna $i = 2, 3, \ldots, k$ a všechny hrany $uv \in E$ definujme

   $$\widehat{w}_i(uv) = w_i(uv) + 2\delta_{i-1}(s, u) - 2\delta_{i-1}(s, v).$$

   Všimněte si, jak je nové ohodnocení hran podobné úpravě ohodnocení podle přípustného potenciálu (viz sekce 9.9).

   Pro všechna $i = 2, 3, \ldots, k$ a všechny hrany $uv \in E$ dokažte, že upravená hodnota hrany $\widehat{w}_i(uv)$ je celočíselná a nezáporná.

(e) Teď definujme $\widehat{\delta}_i(s,v)$ jako délku nejkratší cesty z s do v v grafu G s ohodnocením $\widehat{w}_i$. Pro všechna $i = 2, 3, \ldots, k$ a všechny hrany $uv \in E$ dokažte, že

$$\delta_i(s,v) = \widehat{\delta}_i(s,v) + 2\delta_{i-1}(s,v).$$

Na základě toho dokažte, že $\widehat{\delta}_i(s,v) \leq |V| - 1 \leq |E|$.

(f) Nyní konečně ukažte, jak spočítat $\delta_i(s,v)$ z hodnoty $\delta_{i-1}(s,v)$ pro všechny $v \in V$ v čase $\mathcal{O}(|E|)$. Z toho vyvoďte, jak spočítat $\delta(s,v)$ pro všechny $v \in V$ v čase $\mathcal{O}(|E| \log W)$.

9.11.4 Úlohy na úpravu grafu

Následující úlohy můžeme snadno převést na obyčejný problém hledání nejkratší cesty. Stačí vhodně upravit graf a ohodnocení hran.

1. V Absurdistánu měli N měst spojených navzájem leteckými linkami S různých dopravních společností. Ale jak již to v tomto státě bývá, byrokracie se rozmohla a tak každá dopravní společnost měla různé tarify za překlad nákladu z letadel různých jiných společností. Přinesli Vám následující tabulky:

   - $p(s,i,j)$ - cena účtovaná společností s za přepravu vašeho nákladu z města i do města j. Nekonečno, pokud taková letecká linka neexistuje.
   - $m(i,r,s)$ - cena účtovaná za přeložení nákladu z letadla společnosti r do letadla společnosti s v městě i. Nekonečno, pokud tomu úřední předpisy brání.

   Máte vyřešit, jak svůj náklad co nejlevněji přepravit z města 1 do města N.

2. (Nejkratší cesta mezi množinami A a B) Dostaneme orientovaný graf $G = (V,E)$, ohodnocení $c \in \mathbb{R}^m$ a dvě disjunktní množiny $A, B \subseteq V$. Najděte nejkratší cestu, která začíná ve vrcholu množiny A a končí ve vrcholu množiny B.

   Nápověda: Zkuste problém převést na běžný problém nejkratší cesty.

3. (Nejkratší cesta s lichým počtem hran) Dostanete orientovaný graf s nezáporným ohodnocením hran. Chceme najít nejkratší cestu z vrcholu s do t, která obsahuje lichý počet hran. Až to vyřešíte, tak si rozmyslete, co by se změnilo, kdybychom chtěli najít nejkratší cestu obsahující sudý počet hran?

   Nápověda: nahraďte každý vrchol kromě s a t dvojicí vrcholů.

4. (Zobecněný problém nejkratší cesty) Při internetovém routování dochází k prodlevám na jednotlivých linkách, ale také v samotných routrech. To nás motivuje k zobecnění problému nejkratší cesty.

   Kromě cen na hranách (ohodnocení hran) máme i ceny ve vrcholech (ohodnocení vrcholů). Délka cesty v tomto novém modelu bude součet cen na hranách cesty plus součet cen ve vrcholech cesty (včetně koncových vrcholů).

   Dostanete orientovaný graf $G = (V,E)$ s kladným ohodnocením hran $c(e)$ a kladným ohodnocením vrcholů $c(v)$. Dále dostanete počáteční vrchol $s \in V$. Navrhněte efektivní algoritmus, který pro všechny vrcholy $v \in V$ vypíše cenu nejlevnější cesty z s do v.

   Poznámka: Cena nejlevnější cesty z s do s je $c(s)$.

9.11.5 Ostatní úlohy

1. (Počet nejkratších cest) Dostaneme neorientovaný neohodnocený graf a dva jeho vrcholy u, v. Mezi vrcholy u a v může existovat i více nejkratších cest. Navrhněte lineární algoritmus, který vypíše počet nejkratších cest mezi u a v.

2. (Jednoznačnost nejkratší cesty) Dostanete orientovaný graf s kladným ohodnocením hran a počáteční vrchol s. Potřebovali bychom vyplnit boolovské pole `unique`$[\,\cdot\,]$, kde `unique`$[w]$ = `true` právě tehdy když je nejkratší cesta z s do w jednoznačná (unikátní).

3. (Ověření korektnosti řešení) Dostanete orientovaný graf $G = (V, E)$, ohodnocení jeho hran c (může být i záporné), počáteční vrchol s a strom T, o kterém Vám někdo tvrdí, že je stromem nejkratší cesty. Navrhněte algoritmus, který ověří, že strom T je opravdu stromem nejkratší cesty pro graf G s ohodnocením c a počátečním vrcholem s. Váš algoritmus by měl běžet v lineárním čase.

4. (Hledání čtverců) Navrhněte algoritmus, který dostane neorientovaný graf a rozhodne, jestli graf obsahuje kružnici délky čtyři (čtverec). Časová složitost Vašeho algoritmu by neměla překročit $\mathcal{O}(n^3)$.

5. (Algoritmus na délku nejkratší kružnice) Podívejte se na návrh algoritmu, který najde délku nejkratší kružnice v grafu bez ohodnocení hran.

   Graf budeme procházet pomocí průchodu do hloubky (DFS). Každá zpětná hrana uv, na kterou narazíme, musí spolu se stromovými hranami vedoucími z v do u tvořit kružnici. Délka této kružnice je $level(u) - level(v) + 1$, kde $level(w)$ je vzdálenost vrcholu w od kořene ve stromě průchodu do hloubky. Tohle pozorování nás přivádí k následujícímu algoritmu. Budeme graf procházet do hloubky a u každého vrcholu si budeme pamatovat jeho $level$. Vždy když narazíme na zpětnou hranu, tak si spočítáme délku kružnice a porovnáme ji s dosud nejmenší nalezenou délkou kružnice.

   Ukažte, že tento algoritmus nemusí vždy fungovat. Nalezněte protipříklad a vysvětlete, proč je protipříkladem.

6. (Délka nejkratšího cyklu)

   (a) Dostanete obyčejný graf bez ohodnocení hran. Nalezněte algoritmus, který najde délku nejkratší kružnice v grafu nebo odpoví, že graf žádnou kružnici neobsahuje. Vaše řešení by mělo běžet v čase nejvýše $\mathcal{O}(nm)$.

   (b) Dostanete orientovaný graf s kladným ohodnocením hran. Nalezněte algoritmus, který spočítá délku nejkratšího cyklu. Pokud je graf acyklický, tak by to měl algoritmus zjistit. Vaše řešení by mělo běžet v čase nejvýše $\mathcal{O}(n^3)$.

7. (Délka nejkratšího cyklu obsahujícího hranu e) Dostanete neorientovaný graf $G = (V, E)$ s délkami hran ℓ_e pro každou hranu $e \in E$. Dále dostanete jednu konkrétní hranu $f \in E$. Najděte v čase $\mathcal{O}(n^2)$ nejkratší kružnici obsahující hranu f.

8. (Hledání nejzápornějšího cyklu) Dostanete graf jehož hrany jsou ohodnoceny reálnými čísly, a to i zápornými. Najděte v tomto grafu záporný cyklus s co nejmenší hodnotou.

9. (Arbitráž) Proč hledat záporné cykly? Odpověď je jednoduchá. Abychom vydělali. Na světě funguje spousta směnáren. Kurz jedné měny vůči měně druhé

nám říká, kolik jednotek jedné měny dostaneme za měnu druhou. Například kurz $k_{Eur/Kc}$ udává, kolik Euro dostaneme za 1 korunu (kurzy jsou znormované). Pokud chceme vyměnit koruny za Eura a pak na výletě v Německu Eura za Dolary, tak se kurzy násobí. Počet dolarů, které dostaneme výměnou za 100 Kč, spočítáme jako $k_{\$/Eur} \cdot k_{Eur/Kc} \cdot 100$. Předpokládáme, že výměna ve směnárnách probíhá bez poplatků. Nakonec si můžeme v Americe vyměnit dolary zpátky na české koruny. Množství korun, které tak získáme, záleží na kurzech měn.

Jak provádět výměny v jednotlivých směnárnách, abychom vydělali? Pokud je součin kurzů podél cyklické výměny číslo větší než jedna, tak vyděláme. Pokud je to číslo menší než jedna, tak naopak proděláme.

Různé světové měny si můžete představit jako vrcholy grafu a kurzy měn jako ohodnocení hran. Navrhněte co nejrychlejší algoritmus, který v tomto grafu najde cykly, podél kterých se vyplatí vyměňovat peníze tak, abychom vydělali.

Nápověda: Stačí převést násobení hodnot na hranách cesty na jejich sčítání. Toho docílíme tak, že si vytvoříme kopii grafu, ve které hrany ohodnotíme mínus logaritmem původního ohodnocení. V novém grafu budeme hledat záporné cykly. Proč při záporném cyklu vyděláme a při kladném ne?

10. (Nejkratší cesty vedoucí přes vrchol v_0) Dostanete silně souvislý orientovaný graf $G = (V, E)$ s kladným ohodnocením hran. Graf je *silně souvislý* právě tehdy když z každého vrcholu existuje orientovaná cesta do libovolného jiného vrcholu. Dále dostanete jeden vrchol $v_0 \in V$. Navrhněte efektivní algoritmus, který najde nejkratší cesty mezi každou dvojicí vrcholů, ale s tou podmínkou, že každá cesta musí procházet přes vrchol v_0.

11. (Přidání hrany, která nejvíce zkrátí nejkratší cesty) Je dána silniční síť $G = (V, E)$ propojující města V. Pro každou silnici $e \in E$ známe její délku ℓ_e. Ministerstvo dopravy chce rozšířit silniční síť o jednu novou silnici. Zatím má dlouhý seznam dvojic měst E', které jsou kandidáty na propojení. Každá potenciální silnice $e' \in E'$ už je vyprojektovaná a víme, že bude mít délku $\ell_{e'}$. Vás poprosili, abyste jim pomohli vybrat tu nejvhodnější z uvažovaných silnic. Ministři nejčastěji jezdí z města s do města t a proto Vám položili následují otázku: Stavba které silnice povede k největšímu snížení vzdálenosti mezi městy s a t? Navrhněte efektivní algoritmus pro řešení této otázky.

Kapitola 10

Union-Find problém

Motivace: Po světě se toulá spousta agentů. Často se stává, že jeden agent má spoustu jmen/přezdívek, které používá například při rezervaci hotelu, restaurace, na návštěvě u zajímavých osobností. Tato jména si nevolí úplně náhodně, protože na každé jméno musí mít pravé doklady a ty je těžké sehnat. Každý agent má sadu různých jmen a dokladů na ně. Tajná služba postupně odhaluje ekvivalentní jména, tj. jména patřící stejnému agentovi. Tajná služba by ráda používala systém, do kterého si postupně bude ukládat nalezené dvojice ekvivalentních jmen a kterého by se mohla zeptat, jestli je určitá dvojice jmen ekvivalentní. Například by se chtěli zeptat: „Je agent 007 a James Bond ten samý člověk?“.

Úkol: (grafový pohled) Všechna jména agentů odpovídají vrcholům grafu $G = (V, E)$. Dvojice ekvivalentních jmen odpovídá hraně v grafu. Všechny přezdívky jednoho agenta tvoří komponentu souvislosti. Chceme se systému ptát: „Leží vrcholy u a v ve stejné komponentě souvislosti?“. Problému se také někdy říká dynamické udržování komponent souvislosti a nebo problém udržování ekvivalence.

Komponenta souvislosti určená vrcholem v je množina $C_v = \{u \in V \mid \exists \text{cesta z } u \text{ do } v\}$. V každé komponentě souvislosti vybereme jednoho reprezentanta. Pro jednoduchost budeme reprezentanta komponenty C_v značit r_v, takže pokud u a v leží ve stejné komponentě, tak $r_u = r_v$. Úkol budeme realizovat pomocí operací:

`FIND`$(v) = r_v$, operace vrátí reprezentanta komponenty souvislosti C_v.

`UNION`(u, v) provede sjednocení komponent souvislosti C_u a C_v. To odpovídá přidání hrany uv do grafu.

10.1 Triviální řešení

Předpokládejme, že vrcholy grafu jsou očíslované čísly 1 až n. Použijeme pole `R`$[1..n]$, kde `R`$[i] = r_i$, tj. číslo reprezentanta komponenty C_i. Operace `FIND` pouze vypíše `R`$[v]$ a tedy bude trvat $\mathcal{O}(1)$. K provedení `UNION`(u, v) najdeme reprezentanty $r_u =$ `FIND`(u), $r_v =$ `FIND`(v). Pokud jsou různí, tak projdeme celé pole `R`$[\cdot]$ a každý výskyt r_u přepíšeme na r_v. To nám zabere čas $\mathcal{O}(n)$.

10.2 Často dostačující řešení

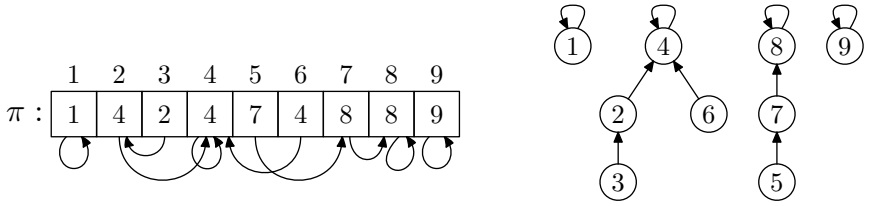
U každého reprezentanta r si pamatujeme ještě `size`$[r]$ = # prvků v komponentě C_r. A pro každou komponentu si pamatujeme spojový seznam jejich prvků. Seznam lze realizovat jako pole `next`$[\,\cdot\,]$ obsahující číslo dalšího prvku v seznamu. Seznam ukončíme nulou.

Předchozí řešení pozměníme tak, že v operaci `UNION` přepíšeme pouze prvky menší komponenty. Díky spojovému seznamu prvků menší komponenty to zvládneme v čase úměrném počtu prvků komponenty. Nakonec zřetězíme seznamy prvků obou komponent a sečteme velikosti komponent.

V nejhorším případě bude operace `UNION` trvat opět $\mathcal{O}(n)$. Ale všimněme si, že prvek i přepisujeme jenom tehdy, když komponentu C_i sjednotíme s druhou komponentou, která má alespoň tolik prvků jako C_i. Díku tomu může být každý prvek přepsán nejvýše $(\log n)$-krát. Proto bude n operací `UNION` dohromady trvat pouze $\mathcal{O}(n \log n)$. To dává amortizovaný čas pro `UNION` $\mathcal{O}(\log n)$.

10.3 Řešení s přepojováním stromečků

Další možnost, jak si pamatovat komponentu C_v je pomocí orientovaného stromečku pokrývajícího komponentu. Z každého vrcholu v povede orientovaná hrana (šipka) do jeho předchůdce $\pi(v)$ ve stromě. Z kořene r povede smyčka zpátky do kořene, tj. $\pi(r) = r$. K reprezentaci stromečků všech komponent nám tedy stačí jen pole $\pi[\cdot]$. Pro každý vrchol v si budeme navíc pamatovat délku nejdelší orientované cesty vedoucí do v a označíme ji $rank(v)$. Proto se tomuto řešení anglicky říká *union by rank*.



Začneme s grafem, který se skládá z izolovaných vrcholů a proto na začátku pro každý vrchol v nastavíme $rank(v) = 0$ a $\pi(v) = v$. V operaci `FIND`(v) najdeme reprezentanta komponenty C_v tak, že z v půjdeme po orientovaných hranách až do kořene. Kořen r_v je hledaný reprezentant komponenty C_v. Operace `UNION`(u, v) proběhne tak, že si nejprve najdeme reprezentanty r_u a r_v a pak natáhneme hranu z kořene stromu s menším rankem do kořene stromu s větším rankem. Tím vznikne nový strom reprezentující sjednocenou komponentu. Pokud budou mít oba stromy stejný rank, tak natažením hrany z r_u do r_v vznikne strom s o jedna větším rankem. Jinak se rank nezmění.

```
FIND(v):
    while v ≠ π(v) do
        v := π(v)
    return v


UNION(u, v):
    r_u := FIND(u)
    r_v := FIND(v)
    if r_u = r_v then return
    if rank(r_u) > rank(r_v) then
        π(r_v) := r_u
    else
        π(r_u) := r_v
        if rank(r_u) = rank(r_v) then
            rank(r_v) := rank(r_u) + 1
```

Časová složitost `FIND`(v) bude odpovídat výšce stromu, který procházíme, a to je $rank(r_v)$. Časová složitost `UNION` bude zhruba 2 krát tolik než časová složitost `FIND`.

Všimněme si, že pro každé v je $rank(v) < rank(\pi(v))$. Kořen s rankem k vznikl tak, že jsme spojili dva stromy s rankem $k-1$. Proto má každý strom s kořenem ranku k alespoň 2^k prvků. Graf má celkem n vrcholů, takže nejvyšší možný rank je nejvýše $\log n$. To je zároveň horním odhadem časové složitosti operací `UNION` a `FIND`.

10.4 Řešení s kompresí cestiček

Předchozí řešení můžeme vylepšit tak, že při každém zavolání `FIND`(v) a tedy při každém průchodu orientované cesty z v do kořene r_v, přepojíme všechny vrcholy na této cestě přímo do kořene r_v. Následující řešení využívá rekurze, ale můžete ho implementovat i bez ní (v prvním průchodu najdete kořen a ve druhém přesměrujete všechny vrcholy do kořene).

```
FIND(v):
  if v ≠ π(v) then
    π(v) := FIND(π(v))
  return π(v)
```

Tato heuristika zvýší čas operace `FIND` jen malinko a snadno se naprogramuje. Z dlouhodobého hlediska nám tato trocha práce navíc může hodně pomoci. Pokud znova zavoláme `FIND` na stejný vrchol, tak už kořen stromu najdeme na jeden krok. Heuristika s kompresí cestiček se vyplatí, pokud budeme operaci `FIND` volat častěji než `UNION`. To ale v grafových algoritmech nastává skoro vždy. Operaci `UNION` můžeme zavolat nejvýše n krát, protože pak už budou všechny vrcholy v jedné komponentě. Operaci `FIND` typicky voláme pro každou hranu grafu (například v algoritmech pro hledání minimální kostry). A hran může být řádově až n^2.

Dá se ukázat, že v implementaci s kompresí cestiček bude m operací `UNION` nebo `FIND` trvat $\mathcal{O}(m\alpha(n))$, kde $\alpha(n)$ je inverzní Ackermannova funkce[1] (pro prakticky použitelná čísla je to konstanta menší rovná 4, ale jinak pro hodně velká n roste až do nekonečna). Amortizovaný čas obou operací tedy je $\mathcal{O}(\alpha(n))$, což je prakticky konstantní čas $\mathcal{O}(1)$.

Poznámka: Prakticky se dobře chová i následující řešení, které při zavolání `FIND`(v) přepojí do kořene pouze vrchol v. Také se o něm dá ukázat, že časová složitost m operací `UNION` nebo `FIND` trvá $\mathcal{O}(m\alpha(n))$.

```
FIND(v):
  while π(v) ≠ π(π(v)) do
    π(v) := π(π(v))
  return π(v)
```

[1] Inverzní Ackermannova funkce $\alpha(n)$ je inverzní funkce k Ackermannově funkci $A(n)$. Ackermannova funkce je definovaná jako diagonála Ackermannovy hierarchie, tedy $A(n) := A(n,n)$. Funkce $A(n)$ roste mnohem rychleji než libovolný polynom, exponenciála či $n!$. Ackermannova hierarchie se počítá rekurzivně.

$$A(m,n) = \begin{cases} n+1 & \text{pro } m = 0, \\ A(m-1,1) & \text{pro } m > 0 \text{ a } n = 0, \\ A(m-1, A(m,n-1)) & \text{pro } m > 0 \text{ a } n > 0. \end{cases}$$

Funkce v jednotlivých řádcích jsou $A(1,n) = n+2$, $A(2,n) = 2n+3$, $A(3,n) = 2^{n+3} - 3$, $A(4,n) = \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{n+3} - 3$.

Hodnoty $A(n)$ pro $n = 0, 1, 2, \ldots$ jsou 1, 3, 8, 61, $2^{2^{2^{65533}}} - 3$, ...

10.4.1 Upočítání amortizovaného času $\mathcal{O}(\log^* n)$

V předchozích sekcích jsme si dokázali, že časová složitost operace `UNION` nebo `FIND` je v nejhorším případě $\mathcal{O}(\log n)$ a zmínili jsme se, že se dá pro kompresi cestiček ukázat amortizovaný čas jedné operace $\mathcal{O}(\alpha(n))$. Teď si ukážeme malinko slabší, ale prakticky dostačující, odhad amortizované časové složitosti.

Číslo $\log^* n$ je definováno jako počet po sobě aplikovaných operací log takový, abychom z čísla n dostali něco menšího nebo rovno 1. Například $\log^* 1000 = 4$ protože $\log\log\log\log 1000 \leq 1$. Pro všechna prakticky použitelná čísla x je $\log^* x \leq 5$. Aby byl iterovaný logaritmus $\log^* x > 5$, tak bychom potřebovali $x > 2^{65536}$. S tak velkým číslem se ale prakticky nikdy nesetkáte.

Věta 7 *Pokud začneme s prázdnou datovou strukturou obsahující n jednoprvkových komponent a provádíme posloupnost m operací* `UNION` *nebo* `FIND`*, tak je celkový čas provedení všech m operací $\mathcal{O}((m+n)\log^* n)$.*

Operace `UNION`(u,v) najde reprezentanty komponent (kořeny stromů) zavoláním `FIND`(u), `FIND`(v) a pak natáhne šipku mezi kořeny spojovaných komponent. Komprese cestiček na ní nemá žádný vliv. Natažení šipky mezi kořeny zabere jen konstantní čas. Proto při odhadu celkové časové složitosti budeme počítat jen čas operací `FIND` (místo času pro `UNION`(u,v) budeme počítat čas `FIND`(u), `FIND`(v)) a k nim přičteme $\mathcal{O}(\#\text{operací } \texttt{UNION})$.

Rank vrcholů mění pouze operace `UNION`. Rank kořene se ještě může zvýšit, ale jakmile vrchol přestane být kořenem, tak už se jeho rank nemění. Kompresí cestiček se $rank(v)$ nezmění. Na druhou stranu už ho nebudeme moci interpretovat jako délku nejdelší orientované cesty vedoucí do vrcholu v.

Pozorování 14

- *Pro každý vrchol v je $rank(v) < rank(\pi(v))$.*
- *Každý kořen s rankem k má alespoň 2^k potomků.*
- *Pokud máme celkem n vrcholů, tak vrcholů s rankem k je nejvýše $n/2^k$.*

Důkaz: První vlastnost platí, protože vrchol ranku k vznikl povýšením při spojování dvou kořenů ranku $(k-1)$ (a jejich podstromů). Z toho indukcí dokážeme i druhou vlastnost (zkuste to).

Druhou vlastnost můžeme rozšířit i na vrcholy, které už nejsou kořenem. Každý takový vrchol musel být jednou kořenem a tehdy pro něj vlastnost platila. Od té doby se jeho rank, ani jeho potomci nezměnili.

Protože jsou všechny podstromy určené vrcholy ranku k vzájemně disjunktní, a každý má 2^k vrcholů, tak je vrcholů ranku k nejvýše $n/2^k$. ∎

Pracujeme s n vrcholy a proto jejich rank může nabývat pouze hodnot od 0 do $\log n$ (dle předchozího pozorování). Tyto hodnoty si rozdělíme do následujících pečlivě vybraných intervalů (důvod vyplyne později)

$$\{1\},\ \{2\},\ \{3,4\},\ \{5,6,\ldots,16\},\ \{17,18,\ldots,2^{16}=65536\},\ \{65537,\ldots,2^{65536}\},\ldots$$

Každý interval je tvaru $\{k+1,k+2,\ldots,2^k\}$, kde k je mocnina dvojky. Počet těchto skupin je $\log^* n$. Prakticky si vystačíme s prvními pěti intervaly, protože jinak bychom museli mít více než 2^{65536} vrcholů. To se prakticky nikdy nestane.

V posloupnosti operací trvá každá operace `FIND` jinak dlouho. Pro výpočet amortizovaného času operace `FIND` použijeme účetní metodu (viz zavedení amortizované

časové složitosti na straně 13). Za každý krok práce budeme muset zaplatit jednou korunu.

Každý vrchol dostane na účet nějaké peníze a to tak, aby všechny vrcholy dohromady dostaly nejvýše $\mathcal{O}(n \log^* n)$ Kč. Každá operace `FIND`(v) dostane $\mathcal{O}(\log^* n)$ Kč. Z těchto peněz musí zaplatit všechnu svoji práci (průchod z v do kořene a kontrakce této cesty). Pokud by jí peníze nestačili, tak si může na platbu půjčit od vrcholů, které prochází. Celkový čas m operací `FIND` bude nejvýše počet peněz a to je $\mathcal{O}(m \log^* n) + \mathcal{O}(n \log^* n)$.

Nyní zbývá ukázat: (a) jak rozdělit $\mathcal{O}(n \log^* n)$ Kč mezi vrcholy a (b) jak budou probíhat platby za prováděnou práci, abychom na žádném účtu neklesly do mínusu.

Nejprve k rozdělení peněz. Každý vrchol dostane peníze v momentě, kdy přestane být kořenem. Od této chvíle už se jeho rank nemění. Pokud jeho rank leží v intervalu $\{k+1, k+2, \ldots, 2^k\}$, tak dostane 2^k Kč. Z pozorování 14 víme, že počet vrcholů s rankem větším než k je nejvýše

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \frac{n}{2^{k+3}} + \cdots \leq \frac{n}{2^k}$$

Proto vrcholům s rankem v jednom konkrétním intervalu zaplatíme nejvýše n korun. Různých intervalů je $\log^* n$ a tudíž celkem vrcholům rozdělíme $\leq n \log^* n$ Kč.

Teď se podíváme na to, jak budou probíhat platby. Čas jedné operace `FIND`(v) je počet skoků po šipkách z vrcholu v do kořene, v každém vrcholu na cestě musíme vykonat jeden krok práce. Rank vrcholů na této cestě roste. Každý vrchol x na této cestě padne jedné ze dvou kategorií: buď je $rank(\pi(x))$ ve vyšším intervalu než $rank(x)$, a nebo jsou oba ve stejném.

Vrcholů prvního typu je nejvýše $\mathcal{O}(\log^* n)$ a proto práce v nich zabere nejvýše čas $\mathcal{O}(\log^* n)$. Tuto práci platí operace `FIND`(v) ze svého účtu.

Vrcholy druhého typu musí práci zaplatit ze svého účtu. Za odměnu budou „převěšeni“ a budou si ukazovat na rodiče s vyšším rankem než byl rank jejich původního rodiče.

Pokaždé, když vrchol druhého typu platí ze svého účtu, tak je povýšen. Tedy po nejvýše tolika povýšeních, kolik dostal peněz (to je nejvýše počet různých hodnot v daném intervalu), dosáhne nirvány a bude si ukazovat na rodiče z vyššího intervalu (už nebude muset nikdy platit).

10.5 Přehled všech řešení

reprezentace	`FIND`	`UNION`
pole	$\mathcal{O}(1)$	$\mathcal{O}(n)$
pole (union by size)	$\mathcal{O}(1)$	$\mathcal{O}(n)$ amortiz $\mathcal{O}(\log n)$
stromečky (union by rank)	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
stromečky s kompresí	$\mathcal{O}(\log n)$ amortiz $\mathcal{O}(\alpha(n))$	$\mathcal{O}(\log n)$ amortiz $\mathcal{O}(\alpha(n))$

Co se týká implementace, tak jsou všechna řešení natolik jednoduchá, že můžeme vždy naprogramovat nejlepší řešení pomocí stromečků s kompresí cestiček. Dostaneme tak "téměř" konstantní časovou složitost každé operace.

Upočítání odhadů časové složitosti $\mathcal{O}(\alpha(n))$ pochází od Tarjana [29].

10.6 Příklady

1. (Tranzitivní uzávěr grafu) Dostanete graf $G = (V, E)$. Hrana uv vyjadřuje, že u a v jsou v relaci R (graf je v podstatě jen zápis relace R obrázkem). Relace R je určitě symetrická, protože máme neorientovaný graf. Relace R ale nemusí být tranzitivní (pokud aRb a bRc tak potom i aRc). Chtěli bychom ji rozšířit na relaci, která už tranzitivní je.

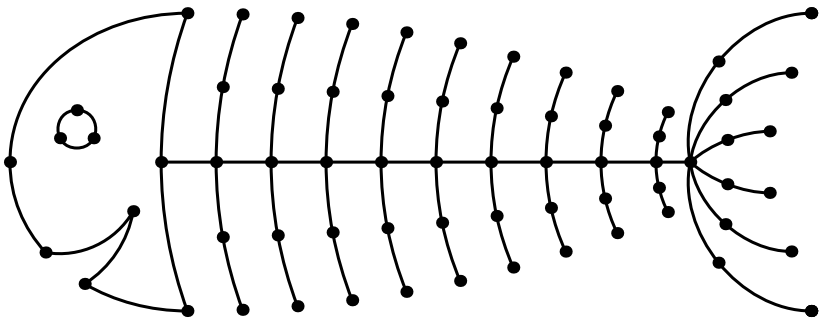
   Tranzitivní uzávěr grafu je nejmenší nadgraf G (doplnění hran), který už je tranzitivní. Jinými slovy, *tranzitivní uzávěr grafu* G je graf $G_T = (V, E_T)$ s původními vrcholy a vw je hrana tranzitivního uzávěru právě tehdy když v G existuje cesta mezi v a w.

   (a) Navrhněte efektivní algoritmus, který najde tranzitivní uzávěr neorientovaného grafu G.

   (b) Tranzitivní uzávěr můžeme definovat i pro orientované grafy (tj. pro relace, které nemusí být symetrické). *Tranzitivní uzávěr orientovaného grafu* G je orientovaný graf $G_T = (V, E_T)$ s původními vrcholy a vw je hrana tranzitivního uzávěru právě tehdy když v G existuje orientovaná cesta z v do w. Navrhněte efektivní algoritmus, který najde tranzitivní uzávěr orientovaného grafu G.

Kapitola 11

Minimální kostra



Motivace 1: (prohrnování silnic) V království je N měst a některá z nich jsou spojena přímou silnicí. Křižovatky jsou pouze ve městech. Mezi některými městy přímá silnice nevede, ale z každého města se dá po silnicích dostat do libovolného jiného. V noci se přihnala se pohádková vánice a zasněžila všechny silnice v celém království. Napadlo až metr sněhu. Odhrabovačů je málo a takovou sněhovou nadílku budou odklízet ještě hodně dlouho. Rozhodněte, které silnice se mají odhrabat jako první, aby mezi každýma dvěma městy vedla sjízdná silnice.

Motivace 2: (elektrifikace) Vysoko v tibetských horách se rozhodli začít využívat výhod moderního světa[1] a chtějí provést elektrifikaci všech N vesnic krásného údolí. Dráty se nesmí větvit jinde než ve vesnici. Znají vzdálenost mezi každýma dvěma vesnicemi, ale nevědí, jak natahat dráty elektrického vedení, aby jich spotřebovali co nejméně. Poradíte jim? Větší množství drátů než je nejmenší možné jim rada starších pro ochranu životního prostředí nepovolí.

Úkol: (grafový pohled) Je dán neorientovaný graf $G = (V, E)$ s ohodnocením hran $c : E \to \mathbb{R}$. Najděte kostru $T \subseteq G$ s nejmenším ohodnocením hran, tj. s nejmenším $\sum_{e \in T} c(e)$. Výstupem bude množina hran $M = E(T)$ tvořících kostru.

Místo ohodnocení hrany $c(e)$ často používáme pojem cena hrany. Nejdražší, respektive nejlevnější hrana je ta s největším, respektive nejmenším ohodnocením.[2]

Připomeňte si (viz kapitola 5 o grafech a stromech), že *strom* T je souvislý graf bez kružnic. Strom na všech vrcholech je také minimální souvislý podgraf (přidáním libovolné hrany k T vznikne kružnice). Mezi libovolnýma dvěma vrcholy stromu T

[1] Některé zdroje uvádějí, že k tomu byli donuceni Čínou.

[2] Často se také ohodnocení hrany nazývá váha hrany. Potom můžeme mluvit o nejtěžší, nejlehčí hraně.

vede jednoznačně určená cesta. Jednoznačnou cestu z vrcholu x do vrcholu y ve stromě T značíme xTy. *Kostra* grafu G je strom $T \subseteq G$ na všech n vrcholech. Také si připomeňte operace přidání a odebrání hrany z grafu G. Výsledky těchto operací značíme $G + e$, $G - e$.

Pro jednoduchost nebudeme v následujícím textu rozlišovat mezi množinou hran $M \subseteq E$ a podgrafem G obsahujícím právě hrany M. *Řez* určený množinou $A \subseteq V$ je množina hran $\delta(A) = \{uv \in E \,|\, u \in A \;\&\; v \notin A\}$. Každá cesta z $u \in A$ do $v \notin A$ prochází přes řez $\delta(A)$ (obsahuje hranu řezu). Platí, že graf je nesouvislý právě tehdy, když existuje řez $\delta(C) = \emptyset$ pro $\emptyset \neq C \subset V$.

Kostru v souvislém grafu najdeme jednoduše pomocí průchodu grafu do hloubky, protože DFS strom souvislého grafu je kostra. To dokazuje, že každý souvislý graf má alespoň jednu kostru. Taková kostra ovšem ještě nemusí být minimální, při hledání minimální kostry musíme postupovat malinko chytřeji. Pokud v zadání dostaneme nesouvislý graf, tak chceme najít řez $\delta(C) = \emptyset$ pro $C \neq \emptyset$, který je důkazem, že graf je nesouvislý a tedy že v grafu žádná kostra neexistuje. Stačí vzít řez určený jednou komponentou souvislosti.

Pozorování (o prohazování hran): Přidáním hrany e do kostry T vznikne právě jedna kružnice C. Vyhodíme-li z kružnice C libovolnou hranu f, dostaneme graf $T + e - f$, který je opět kostrou. Pokud $c(e) < c(f)$, tak bude mít nová kostra menší cenu.

Podobných pozorování využijeme v následujících algoritmech. Nejprve si ukážeme meta-algoritmus a z něj pak snadno odvodíme ostatní algoritmy pro hledání minimální kostry.

11.1 Základní meta-algoritmus

Množina hran $M \subseteq E$ je *rozšířitelná* do minimální kostry, pokud existuje minimální kostra T obsahující hrany M.

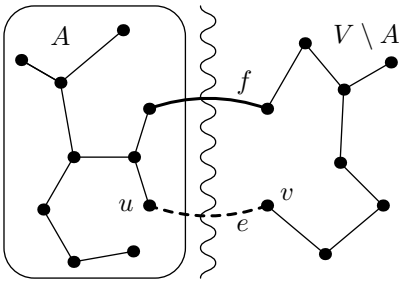
Meta-algoritmus: Postupně budeme konstruovat množinu M rozšířitelnou do minimální kostry. Začneme s prázdnou množinou M. Podle následujícího lemmatu 7 (o existenci řezu) najdeme řez neobsahující hranu M a podle lemmatu 8 (o nejlevnější hraně řezu) do kostry přidáme nejlevnější hranu tohoto řezu. Tento krok zopakujeme $(n-1)$-krát až skončíme s minimální kostrou.

Lemma 7 (o existenci řezu) *Nechť $M \subseteq E$ je množina rozšířitelná do minimální kostry, která ještě není minimální kostrou, pak existuje řez $\delta(A) \neq \emptyset$ takový, že $\delta(A) \cap M = \emptyset$.*

Důkaz: Zvolme si libovolný vrchol x a nechť A je množina vrcholů, do kterých z x vede cesta po hranách ležících v M. Jinými slovy A je komponenta souvislosti M obsahující x. Protože M není minimální kostrou, tak existuje vrchol $y \notin A$. Protože graf G je souvislý, tak existuje cesta xPy. Jelikož $x \in A$ a $y \notin A$, tak cesta xPy obsahuje hranu $e \in \delta(A)$. Ta dokazuje že $\delta(A) \neq \emptyset$. ∎

Následující lemma o nejlevnější hraně neříká nic složitějšího, než že nejlevnější hrana každého řezu patří do minimální kostry. Problémek ovšem nastane, pokud je těch nejlevnějších hran v řezu více. Potom do kostry můžeme dát kteroukoliv z nich, ale nejvýše jednu. Pojem rozšířitelnost do minimální kostry hlídá, jestli už jsme do M nepřidali jinou nejlevnější hranu téhož řezu (řez nesmí obsahovat žádnou hranu M).

Lemma 8 (o nejlevnější hraně řezu) *Nechť $M \subseteq E$ je množina rozšířitelná do minimální kostry a e je nejlevnější hrana řezu $\delta(A)$ splňujícího $\delta(A) \cap M = \emptyset$. Pak je i množina $M \cup \{e\}$ rozšířitelná do minimální kostry.*



Důkaz: M je rozšiřitelná do minimální kostry a proto existuje minimální kostra T obsahující hrany M. Pokud $e \in T$ tak jsme hotovi. Podívejme se na opačný případ $e \notin T$. Označme si konce hrany e pomocí u a v. Mezi u a v existuje v T jednoznačně určená cesta uTv. Jelikož $u \in A$ a $v \notin A$, tak na cestě uTv existuje hrana $f \in \delta(A)$. Cesta uTv spolu s e tvoří kružnici, proto i $T' = T - f + e$ tvoří kostru. Navíc $c(T') = c(T) - c(f) + c(e)$. Protože $c(e) \leq c(f)$, je i $c(T') \leq c(T)$ a T' je minimální kostra obsahující $M \cup \{e\}$. ∎

Uvedený meta-algoritmus je konečný, protože v každém kroku přidá do M jednu hranu a každá kostra má právě $n-1$ hran. Podle indukce a lemmatu 8 (o nejlevnější hraně řezu) bude množina M opravdu minimální kostrou.

Všechny používané algoritmy na nalezení minimální kostry se liší akorát tím, jakým způsobem procházejí vhodné řezy a také tím, jak v nich najdou nejlevnější hranu. Stačí uvažovat řezy určené jednou nebo více komponentami souvislosti M. Ostatní řezy obsahují hranu M a proto nevyhovují předpokladům lemmatu 8. Ukážeme si tři přístupy. Hladový (Kruskalův), Primův (Jarníkův) a Borůvkův algoritmus.

11.2 Kruskalův hladový algoritmus

Kruskalův algoritmus je nejjednoduší implementací meta-algoritmu. Zjenodušil si práci s výběrem nejlevnější hrany řezu tím, že prochází hrany v pořadí podle jejich velikosti. Každou procházenou hranu zkusí přidat do kostry. Pokud hrana $e = uv$ spojuje různé komponenty souvislosti množiny M, tak přidá hranu e do kostry. Aniž to algoritmus tuší, nalezl zároveň i vhodný řez pro meta-algoritmus. Je to řez určený komponentou souvislosti C_u nebo C_v, kde C_v značí komponentu souvislosti obsahující vrchol v. Hrana e je zaručeně nejlevnější hranou řezu $\delta(C_v)$, protože hrany procházíme podle velikosti a všechny menší hrany už leží uvnitř komponent souvislosti M. Správnost Kruskalova algoritmu plyne z předchozích lemmat meta-algoritmu.

Kruskalův hladový algoritmus:

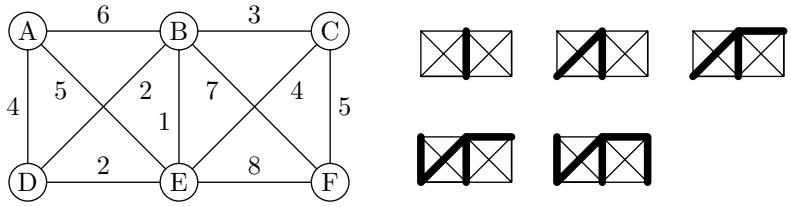
1. Seřaď hrany podle velikosti $e_1 \leq e_2 \leq \cdots \leq e_m$.
2. $M := \emptyset$. Procházej hrany v pořadí podle velikosti. Pokud hrana e_i spojuje různé komponenty souvislosti M, tak ji přidej do M a sjednoť příslušné komponenty souvislosti.
3. Na konci je M minimální kostra.

Algoritmus na začátku třídí m hran, což mu trvá $\mathcal{O}(m \log m)$. Zjišťování, jestli dva vrcholy patří do stejné komponenty souvislosti, případně sjednocení komponent, není nic jiného než Union-Find problém (viz kapitola 10). Proto v bodě 2 provede algoritmus m-krát operaci `FIND` a $(n-1)$-krát operaci `UNION`. To mu zabere čas $\mathcal{O}((m+n)\log n)$. Operace `UNION` a `FIND` umíme realizovat i tak, aby druhá fáze trvala jen čas $\mathcal{O}((m+n)\alpha(n))$, kde $\alpha(n)$ je inverzní Ackermanova funkce. Ale díky časové složitosti první fáze to není potřeba, celkovou časovou složitost algoritmu $\mathcal{O}(m \log n)$ to nezmenší.[3]

[3]Můžeme si dovolit být líní a implementovat Union-Find pomocí pole s řetízky. Amortizovaná časová složitost UNION bude $\mathcal{O}(\log n)$.

Pokud dostaneme hrany už v setříděném pořadí, tak je časová složitost jenom $\mathcal{O}((m+n)\alpha(n))$. Tedy prakticky lineální ve velikosti grafu.

Příklad: Na následujícím obrázku vlevo je ohodnocený graf, ve kterém chceme najít minimální kostru.



Hrany si seřadíme podle velikosti a dostaneme pořadí BE, BD, DE, BC, AD, CE, AE, CF, AB, BF, EF.[4] V tomto pořadí hrany probíráme a zkoušíme je přidat do kostry. Pokud by přidáním hrany vznikla kružnice, tak ji nepřidáme. Průběh algoritmu je zachycen na obrázcích vpravo.

11.3 Jarníkův, Primův algoritmus

Jarníkův algoritmus si udržuje jen jednu komponetu souvislosti a tu postupně rozšiřuje. Začne s komponentou souvislosti, která obsahuje jen počáteční vrchol r. V každém kroku tuto komponentu[5] $T = (R, M)$ rozšíří o nejlevnější hranu řezu určeného touto komponentou.

Jarníkův/Primův algoritmus:

1. $T := (R, M)$, $R := \{r\}$, $M := \emptyset$.
2. V každém kroku přidáme do M nejlevnější hranu e řezu $\delta(R)$ a do R přidáme druhý konec hrany e neležící v R.
3. Po $n-1$ krocích se zastavíme, máme minimální kostru T.

Správnost algoritmu opět plyne z lemmat o meta-algoritmu a z faktu, že přidáváme nejlevnější hranu řezu určeného jedinou komponentou souvislosti M.

Implementace

Při realizaci algoritmu nemusíme probírat všechny hrany řezu. Pro každý vrchol, který ještě není v R, si budeme pamatovat hodnotu $\texttt{d}[v]$ = nejmenší cena hrany vedoucí z v do množiny R, a také $\texttt{odkud}[v]$ ta hrana vede. Na začátku bude hodnota všech $\texttt{d}[v] = \infty$, jen počáteční vrchol r bude mít $\texttt{d}[r] = 0$. Místo probírání všech hran řezu $\delta(R)$ stačí projít všechny hodnoty $\texttt{d}[v]$ a vybrat z nich tu nejmenší. Abychom v průběhu algoritmu udržovali obě pole aktuální, tak po každém přidání nového vrcholu do R zkontrolujeme, jestli z něj nevede levnější hrana do vrcholů $V \setminus R$ a případně aktualizujeme hodnoty $\texttt{d}[v]$ a $\texttt{odkud}[v]$.

```
Jarník, Prim:
    d[r] := 0 a ∀ v ∈ V \ {r} : d[v] := ∞
    ∀ v ∈ V : odkud[v] := nil
    vytvoř prioritní frontu H z vrcholů V s prioritami d[v]
    while fronta H neprázdná do
        v := DELETE_MIN(H)
        přidej hranu {v, odkud[v]} do kostry
```

[4] Hrany stejné velikosti jsme seřadili abecedně.

[5] Komponenta T je stromem. Značíme ji jako podgraf $T = (R, M)$ s vrcholy R a hranami M.

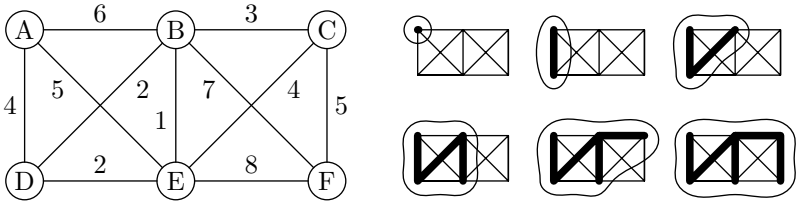
```
for each vw ∈ E do
   if d[w] > c(vw) then
      d[w] := c(vw)
      odkud[w] := v
      DECREASE_KEY(H, w)
```

Algoritmus je hodně podobný Dijkstrovu algoritmu pro hledání nejkratší cesty v grafu (viz. strana 102). Diskuse o možných realizacích prioritní fronty a jejich vhodnosti pro konkrétní grafy je naprosto stejná jako u Dijkstrova algoritmu. Proto zde uvedeme jen výsledky.

Časová složitost Jarníkova algoritmu při realizaci prioritní fronty v poli je $\mathcal{O}(n^2+m)$. Při realizaci binární haldou je $\mathcal{O}((n+m)\log n)$. Při použití d-regulární haldy s volbou $d \approx m/n$ (průměrný stupeň grafu) dostaneme pro velmi řídké grafy (tj. s $m = \mathcal{O}(n)$) časovou složitost $\mathcal{O}(n \log n)$ (stejně jako u binární haldy) a pro ostatní grafy lineární časovou složitost.

Příklad: Zkusme najít minimální kostru grafu, který je na následujícím obrázku vlevo, tentokráte ale podle Jarníkova algoritmu. Vrcholy i hrany grafu budeme procházet v abecedním pořadí. Jednotlivé kroky algoritmu jsou na obrázcích vpravo. Vrcholy patřící do množiny R jsou na obrázcích zakroužkovány.

Jako počáteční vrchol si vybereme A. V prvním kroku mají všechny vrcholy až na A hodnotu $\mathtt{d}[v] = \infty$. Proto si jako vrchol s nejmenší hodnotou $\mathtt{d}[\,\cdot\,]$ vybereme vrchol A a přidáme ho do R. Při aktualizaci $\mathtt{d}[\,\cdot\,]$ snížíme $\mathtt{d}[B] = 6$, $\mathtt{d}[D] = 4$, $\mathtt{d}[E] = 5$ a spolu s tím upravíme $\mathtt{odkud}[\,\cdot\,]$. Ve druhém kroku si za vrchol s nejnižším $\mathtt{d}[\,\cdot\,]$ vybereme vrchol D. Přidáme ho do R a hranu $AD = \{D, \mathtt{odkud}[D]\}$ přidáme do kostry. Dále postupujeme podobně, až dokud nedostaneme minimální kostru.



11.4 Jednoznačnost minimální kostry

Některé algoritmy pro hledání minimální kostry fungují pouze na grafech, ve kterých je minimální kostra určena jednoznačně. Nemůžeme se tohoto předpokladu nějak zbavit, aby dané algoritmy fungovaly na všech grafech? Ano můžeme. Vysvětlíme si jak.

Který krok není jednoznačný?

Podívejme se na průběh meta-algoritmu. Do budované kostry postupně přidáváme nejlevnější hranu každého řezu. Bohužel se ale může stát, že je těch nejlevnějších hran více (například v grafu, kde mají všechny hrany stejné ohodnocení). Máme možnost volby. Rozhodnutí se pro jednu nejlevnější hranu může zamezit přidání ostatních nejlevnějších hran. Pro každou volbu můžeme dostat jinou minimální kostru.

Příklad: (minimální kostra v kružnici C_n s konstantním ohodnocením hran) Pro libovolnou hranu $e \in C_n$ existuje minimální kostra, která e obsahuje, ale i minimální kostra, která e neobsahuje. Zkuste hledat minimální kostru C_n meta-algoritmem. Prozkoumejte, kolik hran z nalezených řezů bude nakonec patřit do minimální kostry.

Lemma 9 (o nejlevnější hraně řezu) *Nechť G je souvislý graf a $\delta(A)$ libovolný řez. Pokud je nejlevnější hrana e řezu $\delta(A)$ určena jednoznačně, tak e patří do minimální kostry.*

Důkaz: Důkaz tohoto lemmatu je téměř shodný s důkazem lemma 8.

Graf G je souvislý, takže má minimální kostru T. Pokud $e \in T$, tak jsme hotovi. Podívejme se na opačný případ a předpokládejme pro spor, že $e \notin T$. Nechť $e = uv$. Přidáním e do T vznikne kružnice, která je tvořena hranou uv a cestou uTv. Vrchol $u \in A$ a $v \notin A$ a proto na cestě uTv existuje hrana $f \in \delta(A)$. Protože e je nejlevnější hrana řezu $\delta(A)$, tak $c(e) < c(f)$. Graf $T' := T - f + e$ je opět kostra. Navíc $c(T') = c(T) - c(f) + c(e) < c(T)$. To je spor s minimalitou kostry T. ∎

Výhoda lemmatu 9 je, že nepotřebuje pojem rozšířitelnosti do minimální kostry.

Pokud mají všechny hrany v grafu různá ohodnocení, tak v každém řezu existuje právě jedna nejlevnější hrana. Potom je postup meta-algoritmu určen jednoznačně.

Lemma 10 (jednoznačnost minimální kostry) *Pokud mají všechny hrany v grafu G různá ohodnocení, tak je minimální kostra určena jednoznačně.*

Zkuste si toto lemma o jednoznačnosti minimální kostry dokázat. S využitím lemmatu 9 (o nejlevnější hraně řezu) je to lehké cvičení.

Jak si zajistit jednoznačnost?

Podle lemmatu 10 víme, že když mají všechny hrany v grafu různá ohodnocení, tak už je minimální kostra určena jednoznačně. Proto si uspořádání na hranách podle ohodnocení rozšíříme do lineálního uspořádání[6] tím, že každé hraně přidáme jednoznačné ohodnocení $c_2(e)$. Hrany budou ohodnoceny dvojicí $(c(e), c_2(e))$. Hrany budeme porovnávat lexikograficky—nejprve podle původního ohodnocení a pokud se hodnoty obou hran shodují, tak podle druhého pomocného uspořádání.

Jaké lineární uspořádání na hranách můžeme zvolit jako pomocné? První možností je jednoznačné očíslování hran (například indexem pole, ve kterém jsou uloženy). Druhou možností je, že máme vrcholy očíslovány čísly 1 až n. Potom porovnáme hrany $e = xy$, $f = uv$ lexikograficky. Předpokládejme, že jsme si hrany označili tak, že $x < y$ a $u < v$. Platí $e <_{LEX} f$ právě tehdy když $x < u$ nebo když $x = u$ a $y < v$.

11.5 Borůvkův algoritmus

Borůvkův algoritmus funguje správně pouze na ohodnocených grafech, ve kterých je minimální kostra určena jednoznačně. (Jak si v každém grafu zajistit jednoznačnost minimální kostry je popsáno v předchozí sekci 11.4). Výhodou Borůvkova algoritmu je, že lze dobře paralelizovat.

Borůvkův algoritmus si v průběhu udržuje les T (množinu hran rozšířitelnou do minimální kostry). Na začátku je les tvořen izolovanými vrcholy (neobsahuje žádnou hranu). V každé iteraci vybereme pro každý strom T_i (komponentu souvislosti) lesa T nejlevnější hranu řezu $\delta(T_i)$ a na závěr iterace ji přidáme do T. Proč ji nepřidáme rovnou? Musíme si dát pozor, abychom nějakou hranu nepřidali dvakrát. Mohlo by se stát, že jedna hrana bude spojovat dvě komponenty souvislosti a pro obě komponenty bude vybrána jako nejlevnější.

Proč algoritmus vyžaduje jednoznačnost minimální kostry? Jinak by se mohlo stát, že pro komponetu T_u vybereme nejlevnější hranu řezu e_u a pro komponentu T_v jinou nejlevnější hranu e_v téhož řezu. Po přidání obou hran do kostry by vznikla kružnice.

[6] V lineárním uspořádání umíme pro každé dvě hrany určit, která z nich je menší.

Borůvka:
  $T := (V, \emptyset)$
  `while` T není souvislý `do`
    Pro každou komponentu souvislosti T_i grafu T
      vyber nejlevnější hranu e_i řezu $\delta(T_i)$.
    Všechny hrany e_i přidej do T.

Lemma 11 *V každé iteraci Borůvkova algoritmu klesne počet komponent souvislosti aspoň na polovinu.*

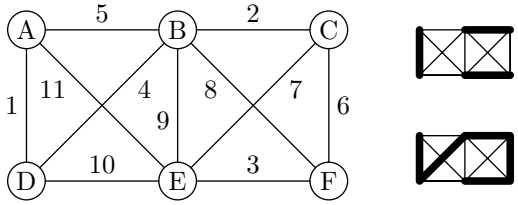
Lemma platí, protože se každá komponenta souvislosti T_i spojí s jinou komponentou souvislosti přes hranu e_i. Jako důsledek dostáváme, že Borůvkův algoritmus provede nejvýše $\log n$ iterací.

Lemma 12 *Borůvkův algoritmus najde minimální kostru.*

Důkaz: Borůvkův algoritmus opět funguje podle meta-algoritmu. Tentokrát ale s tou změnou, že používáme variantu lemmatu "o nejlevnější hraně řezu" pro grafy, jejich hrany mají vzájemně různé ohodnocení. Podle lemmatu 9 patří každá vybraná hrana e_i do jednoznačně určené minimální kostry. ∎

Zkuste si rozmyslet, proč během přidávání hran nevznikne kružnice.

Příklad: Najděte pomocí Borůvkova algoritmu minimální kostru grafu na následujícím obrázku. Jednotlivé iterace jsou na obrázcích vpravo. V první iteraci pro každý vrchol vybíráme nejlevnější hranu, která z něj vede. Pro vrchol A vybereme hranu AD, podobně pro vrchol D vybereme hranu AD. Dále pro vrcholy B a C vybereme hranu BC a pro vrcholy E a F hranu EF. V druhé iteraci podobně vybíráme nejlevnější hranu, která vede z každé „tlusté" komponenty souvislosti někam ven. Pro komponentu $\{A, D\}$ vybereme hranu DB, pro komponentu $\{B, C\}$ také hranu DB a pro komponentu $\{E, F\}$ hranu CF.



Implementace

Nechť T_v označuje komponentu souvislosti, ve které leží vrchol v. V každé iteraci chceme pro každou komponentu T_v nalézt nejlevnější hranu, která z ní vede ven. V každé komponentě souvislosti T_v zvolíme jednoho reprezentanta $\mathtt{r}[v] \in T_v$. Nejlevnější hranu vedoucí z komponenty T_v si zapamatujeme v položce $\mathtt{emin}[\,\cdot\,]$ u reprezentanta komponenty $\mathtt{r}[v]$. Iteraci provedeme následovně:

- Vytvoříme pomocný graf G_{pom} jehož vrcholy jsou reprezentanti komponent souvislosti. Postupně projdeme všechny hrany $e = xy \in E$ a zkusíme je přidat do pomocného grafu jako hrany $\mathtt{r}[x]\mathtt{r}[y]$. Při přidávání hran mohou vzniknout smyčky a násobné hrany. Smyčky rovnou vyhodíme a z násobných hran přidáme do pomocného grafu pouze tu nejlevnější.

  V Borůvkově algoritmu můžeme být velmi líní a nemusíme si z pomocného grafu skoro nic pamatovat. Zapamatujeme si pro každého reprezentanta komponenty pouze nejlevnější hranu $\mathtt{emin}[\mathtt{r}[v]]$, která z reprezentanta vede. Nejlevnější hrana vedoucí z reprezentanta $\mathtt{r}[v]$ odpovídá nejlevnější hraně řezu $\delta(C_v)$.

- Za každého reprezentanta $\mathtt{r}[v]$ přidáme do minimální kostry hranu $\mathtt{emin}[\mathtt{r}[v]]$. (Projdeme všechny vrcholy a najdeme reprezentanty.)
- Před další iterací musíme aktualizovat pole $\mathtt{r}[\,\cdot\,]$. Protože se nám mohlo sjednotit několik komponent souvislosti do jedné, je potřeba provést sjednocení komponent opatrněji než postupným sjednocováním dvojic komponent a přeznačováním $\mathtt{r}[u]$. Jinak by to mohlo trvat příliš dlouho. Pole $\mathtt{r}[\,\cdot\,]$ vyplníme úplně znova tak, že nalezneme komponenty souvislosti v grafu s hranami M (průchod do hloubky).[7]

Mohlo by vás napadnout, proč si neušetříme práci s aktualizací pole $\mathtt{r}[\,\cdot\,]$ a proč nepoužijeme řešení Union-Find problému pomocí přepojování stromečků s kompresí cestiček. Je pravda, že by se výrazně zjednodušila aktualizace pole $\mathtt{r}[\,\cdot\,]$, ale na druhou stranu by hledání nejlevnějších hran vycházejících z každé komponenty trvalo $\mathcal{O}(m\alpha(n))$. To je víc než $\mathcal{O}(m)$.

Výše popsaným postupem můžeme každou iteraci provést v čase $\mathcal{O}(m)$. Celkem bude Borůvkův algoritmus trvat čas $\mathcal{O}(m \log n)$. Výhodou Borůvkova algoritmu je, že v každé iteraci můžeme počítat nejlevnější hrany e_i paralelně.

11.6 Kontraktivní algoritmus

Kontraktivní algoritmus[8] vychází z Borůvkova algoritmu, malinko ho vylepšuje a díky tomu dosáhneme o něco lepší časové složitosti.

V každé iteraci Borůvkova algoritmu jsme konstruovali pomocný graf, ale u každého reprezentanta jsme si pamatovali jen nejlevnější hranu, která z něj vede. Tentokrát ten graf zkonstruujeme pořádně. Postupně projdeme všechny hrany $e = xy \in E$ a přidáme je do pomocného grafu jako hrany $\mathtt{r}[x]\mathtt{r}[y]$. Při přidávání hran mohou vzniknout smyčky a násobné hrany. Smyčky vyhodíme a z násobných hran si budeme pamatovat jen tu nejlevnější.[9] Tímto postupem dostaneme v k-té iteraci graf G_k. Jinými slovy, graf G_k vznikne z grafu G kontrakcí komponent souvislosti podgrafu $T \subseteq G$.

V k-té iteraci se pouze některé komponenty spojili do jedné. Proto nemusíme graf G_k konstruovat z G, ale můžeme ho konstruovat z menšího grafu G_{k-1}.

Nechť n_i označuje počet vrcholů grafu G_i a m_i jeho počet hran. Při postupném vytváření grafů $G_0, G_1, \ldots$ bude i-tá iterace Borůvkova algoritmu trvat čas $\mathcal{O}(m_i)$.

Lemma 13 *Na grafu s různým ohodnocením hran je časová složitost kontraktivního Borůvkova algoritmu* $\mathcal{O}(\min\{n^2, m \log n\})$.

Důkaz: Odhad $\mathcal{O}(m \log n)$ jsme si ukázali i pro nekontraktivní Borůvkův algoritmus. V kontraktivní variantě navíc ušetříme nějaký čas a tak odhad platí. Podívejme se na důkaz odhadu $\mathcal{O}(n^2)$. Podle lemmatu 11 klesne v každé iteraci počet komponent souvislosti aspoň na polovinu. Proto $n_i < n/2^i$. V každém grafu je $m_i < n_i^2$ a proto $m_i < n^2/4^i$. Každá iterace kontraktivního Borůvkova algoritmu trvá $\mathcal{O}(m_i)$ a tedy celkem algoritmus trvá čas $\mathcal{O}(\sum_i m_i) = \mathcal{O}(\sum_i n^2/4^i) = \mathcal{O}(n^2)$. ∎

Lemma 14 *Kontraktivní Borůvkův algoritmus má v rovinných grafech časovou složitost* $\mathcal{O}(n)$.

[7] Aktualizaci pole $\mathtt{r}[\,\cdot\,]$ můžeme udělat i rychleji. Nejprve nalezneme reprezentanty komponent souvislosti v pomocném grafu G_{pom}, který obsahuje pouze hrany přidávané do kostry. Pro každý vrchol $w \in V(G_{pom})$ pomocného grafu si spočítáme jeho reprezentanta do pole $\mathtt{rpom}[w]$. Průchodem do hloubky to zvládneme v lineárním čase v počtu reprezentantů. Samotná aktualizace proběhne tak, že pro všechny $v \in V$ provedeme $\mathtt{r}[v] := \mathtt{r}[\mathtt{rpom}[v]]$.

[8] Jestli je mi to dobře známo, tak pochází od Mareše [21].

[9] Rozmyslete si, jak to dělat, abychom to stihli v čase $\mathcal{O}(m)$.

Důkaz: Platí následující fakt. Kontrakcí a mazáním hran rovinného grafu vznikne rovinný graf. Víme, že $n_i \leq n/2^i$. V každém rovinném grafu $G_R = (V_R, E_R)$ platí $|E_R| \leq 3|V_R| - 6$. Proto $m_i \leq 3n_i \leq 3n/2^i$. Součtem přes všechny iterace dostaneme celkový čas algoritmu $\mathcal{O}(\sum_i m_i) = \mathcal{O}(3n \sum_i 1/2^i) = \mathcal{O}(n)$. ∎

11.7 Červenomodrý meta-algoritmus*

Základní meta-algoritmus můžeme ještě zobecnit o vymazávání hran, které do minimální kostry určitě nepatří. Pro jednoduchost předpokládejme, že všechny hrany mají různé ohodnocení a tím pádem že je minimální kostra určena jednoznačně. Vyhneme se tím pojmu "rozšířitelnost do minimální kostry" a výklad se zjednoduší.

Na začátku jsou všechny hrany grafu bezbarvé. Postupně je budeme barvit na červeno nebo na modro podle následujících lemmat. Barvíme tak dlouho, dokud není vše obarveno a dokud lze některé lemma aplikovat. Červené hrany $\check{C} \subseteq E$ jsou ty, které do kostry určitě nepatří. Klidně je místo barvení můžeme z grafu vymazávat. Minimální kostru budeme hledat v nečervených hranách. Modré hrany jsou ty, které do minimální kostry určitě patří.

Lemma 15 (červené lemma) *Je-li C kružnice v G, pak nejdražší hrana na kružnici e nepatří do minimální kostry. Hranu e obarvíme na červeno.*

Důkaz: Hrana e nepatří do žádné minimální kostry. Předpokládejme pro spor, že T je minimální kostra obsahující nejdražší hranu $e = xy$ kružnice C. Graf $T - e$ má dvě komponenty souvislosti. Protože x leží v jedné a y v druhé komponentě, tak na cestě $P = C \setminus e$ vedoucí z x do y existuje hrana f propojující obě komponenty. Proto je $T' = T + f - e$ kostra. Navíc $c(e) > c(f)$ a tedy $c(T') = c(T) + c(f) - c(e) < c(T)$. To je spor s minimalitou kostry T. ∎

Lemma 16 (modré lemma) *Nechť e je nejlevnější hrana řezu $\delta(A)$. Pak hrana e patří do minimální kostry. Hranu e obarvíme na modro.*

Důkaz: Důkaz tohoto lemmatu je shodný s důkazem lemma 9. (Předpokládejme pro spor, že existuje minimální kostra neobsahující e. Výměnou jedné hrany za e dostaneme levnější kostru a to je spor.) ∎

Lemma 17 (bezbarvé lemma) *Pokud by některá hrana zůstala neobarvená, tak lze ještě aplikovat červené nebo modré lemma.*

Důkaz: Označme neobarvenou hranu $e = xy$. Nechť W je množina vrcholů, do kterých se lze dostat z x po modrých hranách.

Buď $y \in W$, ale potom existuje cesta z modrých hran vedoucí z x do y a ta spolu s hranou $e = xy$ tvoří kružnici C. Každá kružnice obsahuje alespoň jednu červenou hranu, tu nejdražší. Všechny hrany kružnice C kromě e jsou modré a proto hrana e musí být nejdražší hranou kružnice C a můžeme ji podle červeného lemmatu obarvit na červeno.

Pokud $y \notin W$, tak je řez $\delta(W)$ určitě neprázdný (obsahuje hranu e) a neobsahuje žádnou modrou hranu. Proto na řez $\delta(W)$ můžeme aplikovat modré lemma. ∎

Červenomodrý algoritmus je konečný, protože v každém kroku obarví jednu hranu, hrany nepřebarvuje a všech hran je m. Podle lemmat najde minimální kostru, která bude tvořená modrými hranami.

11.8 Přehled algoritmů pro minimální kostru

Kruskalův hladový	$\mathcal{O}(m \log n)$
Kruskalův hladový (setříděné hrany)	$\mathcal{O}((n+m)\alpha(n))$
Jarníkův, Primův (pole)	$\mathcal{O}(n^2+m)$
Jarníkův, Primův (halda)	$\mathcal{O}((n+m)\log n)$
Borůvkův	$\mathcal{O}(m \log n)$

Asymptotické časové složitosti algoritmů pro nalezení minimální kostry se téměř neliší. Odhady můžeme zlepšit ve speciálních případech, kdy dostaneme dodatečná omezení na vstupní graf (více uvidíte v příkladech na konci kapitoly).

Nejrychlejší algoritmus na minimální kostru pochází od Chazelle [6], který s využitím SoftHeaps navrhnul algoritmus s časovou složitost $\mathcal{O}((n+m)\alpha(n))$. Pettie, Ramachandran [26] navrhli algoritmus, který už je optimální. Jen se dosud nepodařilo vyčíslit jeho časovou složitost. Je to někde mezi $\mathcal{O}(m)$ a $\mathcal{O}(m\alpha(n))$ (což už je prakticky jen multiplikativní konstanta).

Chong, Han, Lam [7] ukázali, že paralelní algoritmus s časovou složitostí $\mathcal{O}(\log n)$.

Hezky sepsaný přehled všech možných algoritmů týkajících se minimálních koster najdete v dizertační práci Martina Mareše [21].

11.9 Aplikace minimálních koster

Ukážeme si pár netriviálních aplikací, ve kterých se nám hodí to, že umíme najít minimální kostru grafu.

11.9.1 Steinerovy stromy

V motivačních problémech na začátku kapitoly o minimálních kostrách jsme si reálnou situaci poněkud zjednodušili. Při elektrifikaci Tibetu jsme zakázali, aby se dráty s elektrickým vedením větvily jinde než ve vesnicích. Pokud to povolíme, tak nám bude k propojení vesnic stačit mnohem méně drátu. Příklad 4 takových vesnic je na následujícím obrázku.

Jak určit vhodná místa pro větvení elektrického vedení? Zkuste se sami zamyslet nad tím, co taková místa musí splňovat. Kdy je takové místo optimální a kdy je naopak výhodnější ho posunout kousek vedle.

Předpokládejme, že už známe všechna možná místa větvení. V řadě úloh jsou tato místa známa. Například při prohrnování sněhu v reálné silniční síti chceme propojit všechna města. Kromě měst se silnice může větvit i na křižovatkách ležících mimo města. Graf silniční sítě má tedy dva druhy vrcholů, města která chceme vzájemně propojit a křižovatky, které odpovídají místům větvení. V optimálním řešení nám může po propojení všech měst zůstat spousta nedostupných křižovatek (cestu k nim není potřeba prohrnovat). To nás přivádí k následujícímu úkolu.

Úkol: (Steinerův strom) Je dán neorientovaný graf $G = (V, E)$ s nezáporným ohodnocením hran $c : E \to \mathbb{R}_+$. Vrcholy $V = R \cup S$ jsou dvou druhů, požadované R a Steinerovy S. Najděte strom $T \subseteq G$ s nejmenší cenou, který propojuje všechny požadované vrcholy. Cena stromu T se opět počítá jako $\sum_{e \in T} c(e)$.

Steinerův strom se od kostry liší tím, že nemusí použít všechny Steinerovy vrcholy. Nalezení optimálního Steinerova stromu patří mezi složité úlohy.[10] Není pro ně znám žádný polynomiální algoritmus.

Jediné, co umíme v polynomiálním čase spočítat, jsou přibližná řešení (viz. kapitola **??** o aproximačních algoritmech).

11.9.2 Aproximační algoritmus pro Steinerův strom

Následujcí aproximační algoritmus funguje pouze v grafech, jejichž ohodnocení hran splňuje *trojúhelníkovou nerovnost* (pro libovolné 3 hrany ab, bc, ac platí $c(ab) + c(bc) \geq c(ac)$).

Aproximační algoritmus:

1. Graf G obsahuje vrcholy $V = R \cup S$. Vytvoříme si pomocný úplný graf G', který obsahuje pouze požadované vrcholy R. Cena hrany $uv \in G'$ je cenou nejkratší cesty z u do v v grafu G. Grafu G' se říká *metrický uzávěr* grafu G. Graf G' už je úplným grafem (libovolná dvojice vrcholů určuje hranu). Cena Steinerova stromu v grafu G' je stejná nebo vyšší než cena Steinerova stromu v původním grafu G.

2. Najdeme minimální kostru T' v grafu G'. Ta je aproximací Steinerova stromu pro graf G'.

3. Některé hrany uv v kostře $T' \subseteq G'$ odpovídají v grafu G cestě spojující vrcholy u a v. Proto každou hranu $e \in T'$ nahradíme jí odpovídající cestou. Cesta má stejnou cenu, jako hrana, protože G' je metrickým uzávěrem G. Takto ale mohli vzniknout kružnice. Proto výsledný graf projdeme a přebytečné hrany odstraníme (opět nalezneme minimální kostru). Tím ještě snížíme cenu nalezeného řešení. Skončíme s kostrou $T \subseteq G$, která je aproximací Steinerova stromu v G.

Lemma 18 *Nechť OPT je cena optimálního Steinerova stromu T^* v grafu G. Cena stromu T nalezeného algoritmem je $OPT \leq c(T) \leq 2 \cdot OPT$.*

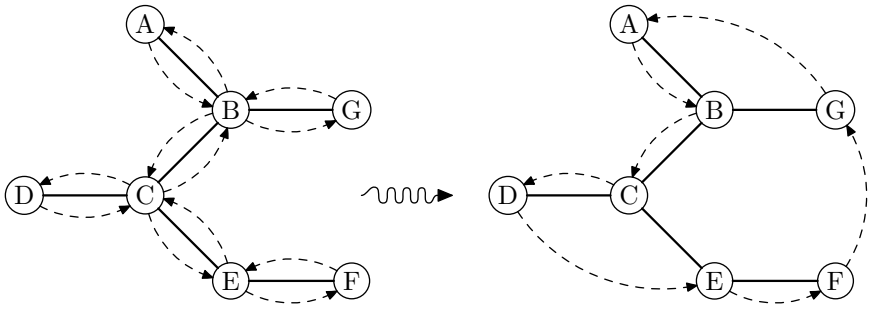
Důkaz: Algoritmem nalezený strom T je přípustným řešením úlohy. Proto $OPT \leq c(T)$.

Vezmeme optimální Steinerův strom T^* v grafu G. Zdvojíme hrany stromu T^* a nalezneme na nich uzavřený eulerovský tah W. Cena $c(W) = 2 \cdot OPT$.

Každý úsek tahu W, který prochází přes Steinerův vrchol u zkrátíme tak, že úsek $v \to u \to w$ procházející přes hrany vu, uw nahradíme zkratkou $v \to w$, která prochází pouze přes hranu vw. Protože hrany grafu splňují trojúhelníkovou nerovnost, dostaneme levnější tah. Zkracování tahu W budeme aplikovat tak dlouho, dokud nebudou všechny vrcholy tahu W patřit do požadovaných vrcholů R.

Potom tah projdeme ještě jednou a pomocí „zkratek“ vytvoříme Hamiltonovskou kružnici H. Uděláme to tak, že procházíme eulerovský tah. Pokud následující hrana tahu vede do vrcholu, který jsme už navštívili, tak půjdeme zkratkou do prvního z následujících vrcholů tahu, kde jsme ještě nebyli. Cena tahu mohla opět jen klesnout.

[10] Problém Steinerova stromu je NP-těžký a to i v případě, když se omezíme na grafy jejichž ceny hran splňují trojúhelníkovou nerovnost.

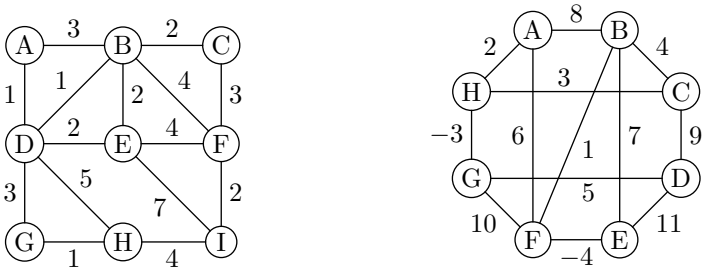


Vyhozením nejdražší hrany z kružnice H dostaneme strom T'^*. Cena stromu T'^* je $\leq 2 \cdot OPT$. Strom T'^* je kostrou v grafu G', ale nemusí být minimální kostrou. Cena minimální kostry je stejná a nebo menší. Protože T' je minimální kostra v G', dostáváme $c(T') \leq 2 \cdot OPT$. Ve 3. kroku algoritmu jsme z kostry $T' \subseteq G'$ vytvořili levnější kostru $T \subseteq G$. Proto je $c(T) \leq c(T') \leq 2 \cdot OPT$. ∎

11.10 Příklady

11.10.1 Přímé procvičení probraných algoritmů

1. V následujících grafech najděte minimální kostru:



   (a) Hladovým (Kruskalovým) algoritmem.

   (b) Jarníkovým (Primovým) algoritmem.

   (c) Borůvkovým algoritmem.

2. Najděte ohodnocený graf, který má jednoznačně určenou minimální kostru, ale jeho hrany nemají vzájemně různá ohodnocení.

3. Profesor Fishbone tvrdí, že pojem "rozšířitelnost do minimální kostry" není u základního meta-algoritmu vůbec potřeba a zbytečně meta-algoritmus komplikuje. Navrhuje meta-algoritmus, který postupně prochází libovolné řezy $\delta(W)$ a do minimální kostry přidá libovolnou nejlevnější hranu řezu $\delta(W)$. Meta-algoritmus skončí po přidání $n - 1$ hran, protože v ten moment bude mít minimální kostru.

   Rozhodněte, jestli má profesor Fishbone pravdu a své rozhodnutí dokažte.

4. (Test) Dostanete graf $G = (V, E)$ s cenami hran $c : E \to \mathbb{R}$. Následující tvrzení mohou, ale nemusí být pravdivé. Vždy buď dokažte, že je tvrzení pravdivé, nebo nalezněte protipříklad.

(a) Pokud má graf G více než $n-1$ hran, tak nejdražší hrana grafu určitě nepatří do minimální kostry.

(b) Jestli je e nejlevnější hranou v grafu G, tak určitě patří do minimální kostry.

(c) Pokud je nejlevnější hrana v grafu G určena jednoznačně (ostatní hrany jsou dražší), tak musí být obsažena v každé minimální kostře.

(d) Každá hrana, která je součástí minimální kostry, je nejlevnější hranou nějakého řezu.

(e) Pokud cyklus obsahuje pouze jednu nejlevnější hranu, tak tato hrana patří do minimální kostry.

(f) Nejkratší cesta mezi dvěma vrcholy určitě patří do minimální kostry.

(g) Strom nejkratší cesty obsahuje stejné hrany jako minimální kostra.

(h) Cesta P je r-levná, pokud je cena každé hrany na cestě P nejvýše r. Pokud graf G obsahuje nějakou r-levnou cestu mezi s a t, tak i každá minimální kostra T obsahuje r-levnou cestu sTt.

(i) Minimální kostra je souvislý podgraf takový, že součet cen jeho hran je nejmenší možný.

Správná odpověď je $114 = [011100010]_2$.

11.10.2 Na teorii

1. Je dáno n bodů v rovině. Definujme ohodnocení hran úplného grafu na těchto bodech tak, že ohodnocením hrany xy bude vzdálenost bodů x, y.

   (a) Ukažte, že maximální stupeň vrcholu v libovolné minimální kostře je nejvýše 6.

   (b) Ukažte, že existuje minimální kostra, která je rovinná (jejíž hrany se navzájem nekříží).

2. Dokažte, že neorientovaný graf G s n vrcholy a k komponentami souvislosti má alespoň $n-k$ hran.

3. Dostanete neorientovaný graf $G = (V, E)$, jehož hrany jsou ohodnoceny navzájem různými kladnými čísly. Dále dostanete vrchol $s \in V$. Je strom nejkratší cesty z s do všech ostatních vrcholů shodný s minimální kostrou? Pokud ano, tak uveďte důvod. Pokud ne, tak uveďte příklad grafu, kde se liší.

4. Nechť G je neorientovaný ohodnocený graf, $H \subseteq G$ jeho podgraf a T minimální kostra G. Ukažte, že $T \cap H$ je obsaženo v nějaké minimální kostře H.

11.10.3 Na algoritmy

1. (Maximální kostra) Najděte maximální kostru v grafu. Jaká je časová složitost vašeho algoritmu ve srovnání s algoritmy na nalezení minimální kostry?

2. (Varianty či nevarianty minimální kostry) Dostanete graf $G = (V, E)$ s ohodnocením hran $c : E \to \mathbb{R}$. Nalezněte kostru T s minimálním

   (a) $\max_{e \in T} c(e)$.

   (b) $\sum_{e \in T} c(e)$.

   (c) $\prod_{e \in T} c(e)$, kde $c(e) \geq 0$ pro všechny hrany e.

3. Dostanete graf, jehož hrany jsou ohodnoceny pouze čísly $1, 2, \ldots, k$. Jak rychle dovedete najít minimální kostru v tomto grafu?

   Dokážete to v čase $\mathcal{O}(kn + km)$? A v čase $\mathcal{O}(kn + m)$? No jestli dokážete i to, tak co v čase $\mathcal{O}((n + m)\alpha(n))$?

4. Dostanete souvislý neorientovaný graf G. Navrhněte algoritmus, který zjistí, jestli z G můžete smazat jednu hranu tak, aby graf zůstal souvislý. Nalezenou hranu vypište. Dovedete snížit časovou složitost vašeho algoritmu až na $\mathcal{O}(n)$?

5. Dostanete souvislý neorientovaný graf G s ohodnocením hran $c : E \to \mathbb{R}$. Navrhněte algoritmus, který najde nejlevnější množinu hran, které můžeme z grafu vymazat tak, aby graf zůstal souvislý, ale už neobsahoval žádnou kružnici.

6. Jedna velká železniční společnost propojuje n měst po celé Evropě. Některá města jsou propojeny přímým spojením. Fixní náklady na provoz přímého spojení mezi městy u a v jsou $c(uv)$ (bez ohledu na vytíženost spojení, jezdit se musí podle jízdního řádu). S několika přestupy existuje dopravní spojení mezi libovolnou dvojicí měst. Spočtěte, kolik nejvíc by železniční společnost mohla ušetřit, kdyby zrušina nadbytečná přímá spojení. (Stále musí být možné se dostat z libovolného města do libovolného jiného).

7. Firma Truhlík a syn má ve městě N budov a chce všechny svoje budovy propojit počítačovou sítí. Vedení firmy rozhodlo, ze pro K ($1 \leq K \leq N$) budov zakoupí vysokorychlostní připojení na Internet. Kromě toho mezi některými dvojicemi budov vybudují propojení optickým kabelem.

   Dvě budovy se nacházejí v téže komponentě sítě, pokud lze mezi nimi komunikovat pomocí optických kabelů (buď mají přímé spojení, nebo jsou spojeny nepřímo přes několik jiných budov). Aby bylo možné komunikovat mezi dvěma budovami ležícími v různých komponentách sítě, musí každá z těchto komponent obsahovat aspoň jeden počítač připojený na Internet.

   Soutěžní úloha: Na vstupu jsou dána čísla N a K a pro každou dvojici budov jedno kladné celé číslo $c(AB)$ = cena za propojení budov A a B optickým kabelem . Navrhněte efektivní algoritmus, jenž určí, kterých K budov se má připojit na Internet a které dvojice budov se mají propojit optickým kabelem tak, aby mezi každými dvěma budovami bylo možné komunikovat a přitom aby celková cena za položení optických kabelů byla co nejmenší.

8. Když chtěl Blátošlap konečně vyprat své ponožky tak, zjistil, že se mu na nich vyvinula celá kolonie neznámých živočichů. Protože to byl genetik, jal se hned tyto živočichy zkoumat a zjistil, ze i když se jedná o jediný druh, jeho zástupci jsou velmi rozmanití. DNA každého jedince má přesně 30 znaků, které ho charakterizují a které se mohou měnit pouze mutací. Dalším výzkumem zjistil, že v prádelníku má N různých poddruhů (poddruhy se navzájem liší alespoň v jednom znaku), i když jeden z nich převazuje.

   Ihned se dovtípil, že v jeho prádelníku došlo k evoluci. A protože ho zajímá její průběh, byli jste požádáni o vyřešení tohoto problému. Blátošlap Vám zadá N a dále DNA jednotlivých poddruhů. DNA každého druhu je zapsáno binárně jako číslo X, $0 \leq X \leq 2^{30} - 1$. i-tý bit tohoto čísla vyjadřuje, zda je i-tý z 30 znaků přítomen.

   Vaším úkolem je zjistit, jaký poddruh se vyvinul z jakého, a počet mutací, ke kterým muselo při tomto vývoji dojít. Předpokládejte, ze vývoj probíhal tak, že každý poddruh kromě toho, který Vám Blátošlap zadal jako první (to je ten nejpočetnější), se vyvinul právě z jednoho jiného poddruhu. Navíc

první zadaný poddruh je původním prapředkem všech poddruhů v prádelníku. Dále předpokládejte, že evoluce probíhala nejjednodušší možnou cestou, a tedy počet mutací v evoluci je nejmenší možný. Počet mutací je součet všech rozdílných znaků mezi každým poddruhem (kromě prvního zadaného) a jeho předkem. Navíc žádný poddruh v Blátošlapově prádelníku nevymřel, všechny evolucí vzniklé poddruhy přežily až do dnešní doby.

Příklad: Pro $N = 4$ a poddruhy 0, 3, 7, 12 je hledaná evoluce tato: poddruhy 3 a 12 se vyvinuly z poddruhu 0, poddruh 7 se vyvinul z poddruhu 3. Počet mutací, ke kterým muselo v evoluci dojít, je roven 5.

9. (Jednoznačnost minimální kostry) Dostanete souvislý ohodnocený graf G. Navrhněte co nejefektivnější algoritmus, který zjistí, jestli je minimální kostra grafu G určena jednoznačně.

10. (Počet koster) Jak je těžké spočítat, kolik má graf G koster? A jak je těžké spočítat, kolik má ohodnocený graf G minimálních koster? Navrhněte oba algoritmy a uveďte jejich časovou složitost.

11. (Bottleneck distance) Dostanete souvislý ohodnocený graf. *Úzké hrdlo* na cestě P je hrana $e \in P$, která má nejvyšší cenu. *Bottleneck distance* mezi vrcholy u a v je minimum z cen úzkého hrdla na všech cestách mezi u a v.

    (a) Navrhněte algoritmus, který pro každou dvojici vrcholů nalezne jejich bottleneck distance. Analyzujte časovou složitost vašeho řešení.

    (b) Co by se změnilo, kdyby úzkým hrdlem byla nejlevnější hrana na cestě? Navrhněte algoritmus, který pro každou dvojici vrcholů nalezne jejich bottleneck distance.

12. (Druhá nejmenší kostra). Když v ohodnoceném grafu G nalezneme minimální kostru T, tak z grafu vyhodíme všechny hrany kostry T a dostaneme graf G'. Minimální kostra v grafu G' je druhou nejmenší kostrou grafu G.

    (a) Navrhněte co nejrychlejší algoritmus, který najde druhou nejmenší kostru. Analyzujte jeho časovou složitost.

    (b) Co kdybychom chtěli nalézt k nejmenších koster? Jaká by byla časová složitost takového algoritmu?

13. (Hybridní algoritmus) Dostaneme graf G a chceme v něm nalézt minimální kostru. Podívejte se na algoritmus, který v první fázi provede k iterací Borůvkova algoritmu. Ve druhé fázi zkontrahuje nalezené komponenty do vrcholů (viz pomocný graf u Borůvkova algoritmu) a na tento graf pustí Jarníkův algoritmus (s Fibonacciho haldou).

    (a) Vyjádřete časovou složitost hybridního algoritmu pomocí n, m a k.

    (b) Pro jakou volbu parametru k bude časová složitost algoritmu nejmenší? Kolik bude časová složitost hybridního algoritmu pro vhodnou volbu k?

11.10.4 Aproximační algoritmy využívající minimální kostry

1. (Steinerovy stromy – dolní odhad pro aproximaci pomocí minimální kostry) Uvažujme pouze ohodnocené grafy takové, že ceny jejich hran splňují trojúhelníkovou nerovnost. Najděte graf $G = (V, E)$ a rozdělení jeho vrcholů V na požadované R a Steinerovy S tak, aby cena jeho minimální kostry byla $(2 - 1/n)$ krát větší než cena jeho optimálního Steinerova stromu ($n = |V|$, kde $V = R \cup S$).

2. (Problém obchodního cestujícího v grafech s trojúhelníkovou nerovností).[11] Obchodní cestující prodává svůj produkt v n městech. Chce si naplánovat takovou trasu, aby objel všechna města a najel co nejméně kilometrů.

   Dostaneme úplný graf, jehož hrany jsou ohodnoceny kladnými reálnými čísly. Chceme najít Hamiltonovskou kružnici[12] nejmenší ceny.

   Obecný problém obchodního cestujícího je NP-úplný, ale ve grafech s trojúhelníkovou nerovností lze aproximovat v polynomiálním čase s libovolnou pevnou přesností. V následujících příkladech budeme předpokládat, že ohodnocení hran grafu splňuje trojúhleníkovou nerovnost.

   (a) Navrhněte algoritmus, který nalezne řešení problému ochodního cestujícího, které je nevýše 2krát horší než optimální řešení.

   (b) Nalezněte příklad grafu na n vrcholech, ve kterém váš aproximační algoritmus nalezne 2krát horší řešení, než je to optimální.[13]

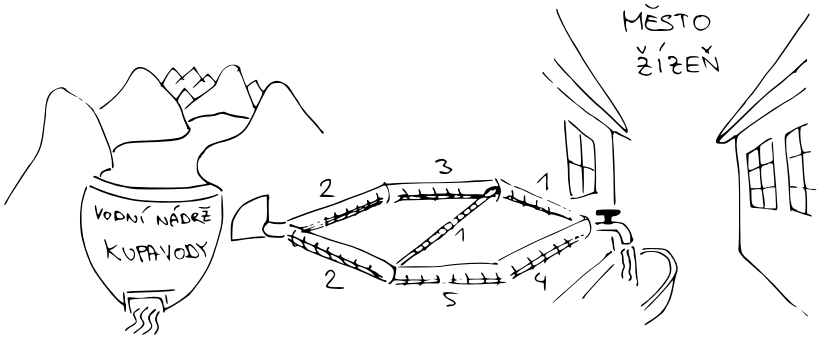
[11] Anglicky se označuje jako Traveling salesman problem (TSP).

[12] Hamiltonovská kružnice je kružnice, která obsahuje všechny vrcholy grafu.

[13] Tedy pokud jste postupovali podobně, jako algoritmus pro 2-aproximaci Steinerova stromu. Jinak ukažte horní i dolní odhad na velikost aproximačního faktoru vašeho algoritmu.

Kapitola 12

Toky v sítích



Motivace (pro toky): Ve městě Žížeň je velký nedostatek vody. Město je propojeno potrubní sítí s vodní nádrží Kupavody, kde je vody dostatek. Schéma vodovodní sítě je na obrázku. Každá trubka je jinak široká a proto je ve schématu u každé trubky napsán maximální počet litrů, který trubkou proteče za jednu minutu. Vaším úkolem je zjistit, kolik nejvíce litrů doteče z přehrady do města.[1]

Toky v sítích nejsou jen o vodě v potrubí. V analogických sítích může protékat elektrický proud, auta ve městě, telefonní hovory, peníze nebo pakety v počítačových sítích.[2]

Motivace (pro řezy): V Hádavém království jsou některé dvojice měst spojeny přímou silnicí. Po silniční síti se dá dostat z každého města do libovolného jiného. Jak už to tam bývá zvykem, dvě velká města Velezdroje a Hustostoky se pohádala o to, z které strany se loupe banán. Radní obou měst chtějí rozkopat některé silnice tak, aby už nebylo možné dojet po silnici z jednoho města do druhého. Prý tím zabrání šíření špatných názorů. Které silnice mají silničáři překopat, aby splnili úkol a zároveň překopali co nejméně silnic?

[1] Je zajímavé, že do skutečného potrubí stačí vodu pustit a ona už sama poteče tak, aby jí proteklo co nejvíce.

[2] Problém toků v sítích zkoumali už v 50. letech výzkumníci Air Force T. E. Harris a F. S. Ross. Studovali přesun materiálu po železniční síti ze Sovětského Svazu do satelitních zemí východní Evropy. Minimální řezy železniční sítě jsou strategická místa pro americké bombardéry. Jejich výzkum byl a přísně tajný a odtajněn byl až v roce 1999.

Protože je kapitola o tocích v sítích poměrně rozsáhlá, pojďme si stručně představit obsah hlavních sekcí.

- 12.1 Maximální tok a minimální řez – úvodní kapitola, ve které zavedeme důležité pojmy a vysvětlíme základní teorii.
- 12.2 Algoritmy vylepšující cesty – historicky starší algoritmy (Ford-Fulkersonův, Dinicův, Edmons-Karpův, 3 Indů), které pro nalezení maximálního toku využívají teorii o vylepšujících cestách.
- 12.3 Goldbergův Push-Relabel algoritmus – novější a prakticky rychlejší algoritmus, který není přímočarou aplikací teorie o tocích.
- 12.5 Aplikace toků v sítích – ukázka několika problémů, které se dají vyřešit převodem na hledání maximálního toku. Toky v sítích mají ohromné množství aplikací. S dalšími aplikacemi se seznámíme ve cvičeních na konci kapitoly.

12.1 Maximální tok a minimální řez

Definice: *Síť* (G, s, t, c) je orientovaný graf $G = (V, E)$, který má dva speciální vrcholy: *zdroj* s a *spotřebič* $t \in V$ (z anglického source a target), a každá hrana e má *kapacitu* $c(e)$, kde kapacita je funkce $c : E \to \mathbb{R}_+$. Spotřebič je někdy označován jako *stok*.

Tok f je funkce $f : E \to \mathbb{R}_+$, která splňuje

$$\begin{array}{lll} i) & 0 \le f(e) \le c(e) & \text{pro každou hranu } e \in E \\ ii) & \sum_{xv \in E} f(xv) = \sum_{vx \in E} f(vx) & \text{pro každý vrchol } v \in V \setminus \{s,t\} \end{array}$$

Číslu $f(e)$ říkáme tok po hraně e nebo také průtok hranou e. První podmínka říká, že průtok hranou je nezáporný a nemůže překročit kapacitu hrany. Druhá podmínka říká, že co do vrcholu přiteče, to z něj zase musí odtéci. Druhé podmínce se také říká zákon zachování toku, protože se tok ve vrcholech ani neztrácí[3], ani nepřibývá (kromě zdroje a spotřebiče). V analogii v elektrických obvodech se druhé podmínce říká *Kirchhoffův zákon.* Pokud chceme explicitně vyjádřit, že mluvíme o toku ze zdroje s do spotřebiče t, tak tok označíme jako (s, t)-tok.

Protože se v teorii o tocích vyskytuje rozdíl mezi přítokem do vrcholu v a odtokem z vrcholu v hodně často, tak si zavedeme zkrácené označení

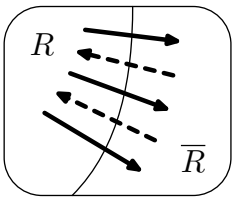
$$f(v) = \sum_{xv \in E} f(xv) - \sum_{vx \in E} f(vx).$$

Číslu $f(v)$ budeme říkat *bilance vrcholu* v nebo také *přebytek toku* ve vrcholu v. Při čtení následujícího textu budeme muset být malinko opatrní na to, co je argumentem $f()$, protože $f(e)$ nebo $f(uv)$ je průtok hranou $e = uv$, ale $f(v)$ je bilance vrcholu v. Druhou podmínku z definice toku můžeme jednoduše vyjádřit jako $f(v) = 0$.

Velikost toku $|f|$ je množství toku, které protéká ze zdroje do spotřebiče. Protože se tok ve vrcholech ani neztrácí ani nepřibývá, tak ho můžeme spočítat jako $|f| = f(t)$. Tedy jako tok, který přitéká do spotřebiče. Stejně bychom mohli velikost toku měřit jako $-f(s)$, což je velikost toku, který vytéká ze zdroje.

Doplněk množiny $R \subseteq V$ značíme $\overline{R}$. Tedy $\overline{R} = V \setminus R$. Množinu orientovaných hran $\delta(R) = \{vw \in E \mid v \in R \,\&\, w \in \overline{\mathcal{R}}\}$ nazveme *řezem* určeným množinou $R \subseteq V$. Řez je (s, t)-*řez,* pokud $s \in R$ a $t \notin R$. Tedy pokud řez odděluje zdroj od spotřebiče. Řez určený jediným vrcholem v budeme zjednodušeně značit $\delta(v)$ místo $\delta(\{v\})$. *Kapacita řezu* $c(\delta(R)) := \sum_{e \in \delta(R)} c(e)$.

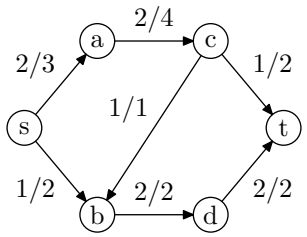
Pozor, je rozdíl mezi řezem v orientovaném grafu a řezem v neorientovaném grafu. Řez určený množinou R v orientovaném grafu obsahuje jen hrany, které z R vychází ven. Na proti tomu řez určený množinou R v neorientovaném grafu obsahuje hrany, které mají jeden konec v R.



Pokud chceme hledat maximální tok nebo minimální řez v neorientovaném grafu, tak nahradíme každou neorientovanou hranu dvojicí šipek jdoucích proti sobě. Kapacity obou šipek budou stejné jako kapacita původní hrany. Maximální tok a minimální řez v takto vytvořeném orientovaném grafu odpovídá toku a řezu v původním neorientovaném grafu. Pro jednoduchost budeme v celé kapitole o tocích pracovat pouze s orientovanými grafy.

[3] To už o pražské vodovodní síti říci nejde. Před lety se uvádělo, že se ztratí až pětina vody, která se do potrubí pustí.

Příklad: Na následujícím obrázku je síť. U každé hrany e jsou uvedeno „$f(e)/c(e)$“.



Velikost toku v síti na obrázku je 3. Velikost řezu určeného vrcholy $\{s, a, b\}$ je 6 a velikost opačného řezu, to je řezu určeného vrcholy $\{c, d, t\}$, je 1.

Platí věta, že v každém grafu existuje maximální tok. Přímý důkaz věty vyžaduje pokročilejší znalosti matematické analýzy a proto ho neuvedeme. Větu dokážeme nepřímo tím, že si ukážeme algoritmy, které maximální tok najdou.

Lemma 19 *Pro každý (s,t)-tok f a každý (s,t)-řez $\delta(R)$ platí*

$$f(\delta(R)) - f(\delta(\overline{R})) = f(t)$$

Důkaz: Spočítáme $X = \sum_{v \in \overline{R}} f(v)$ dvěma způsoby. Jednou přes příspěvky vrcholů, podruhé přes příspěvky hran.

Bilance všech vrcholů kromě zdroje a spotřebiče je rovna nule. Množina $\overline{R}$ obsahuje jen spotřebič t a proto $X = f(t)$.

Na druhou stranu se podívejme na příspěvky od jednotlivých hran. Hrana $e = vw$ s oběma konci v $\overline{R}$ přispívá do bilance vrcholu v hodnotou $-f(e)$ a do bilance vrcholu w hodnotou $f(e)$. Proto je její celkový příspěvek do X roven nule. Jediné hrany, které přispívají do X něčím nenulovým, jsou ty které mají jeden konec v R a druhý v $\overline{R}$. Můžeme je rozdělit na hrany $\delta(R)$ vedoucí z R ven a na hrany $\delta(\overline{R})$ vedoucí z venku do R. Jejich příspěvky do X jsou $f(\delta(R)) - f(\delta(\overline{R}))$. ∎

Důsledek 1 *Pro každý (s,t)-tok f a každý (s,t)-řez $\delta(R)$ platí*

$$f(t) \leq c(\delta(R))$$

Důkaz: Z lemma 19 dostáváme $f(t) \leq f(\delta(R)) \leq c(\delta(R))$. Druhá nerovnost platí, protože kapacita hrany je horním odhadem na průtok hranou. ∎

Velikost každého (s,t)-řezu je horním odhadem na velikost toku. Řez je zároveň jednoduchým a snadno ověřitelným certifikátem, jak dokázat, že v síti větší tok neexistuje. Dokonce platí následující věta, která byla objevena Fordem a Fulkersonem v roce 1956 a nezávisle Kotzigem[4] v témže roce.

Věta 8 (o maximálním toku a minimálním řezu) *Pokud existuje maximální (s,t)-tok, pak*

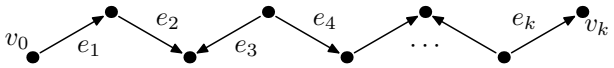
$$\max\{|f| : f \text{ je } (s,t)\text{-tok}\} = \min\{c(\delta(R)) : \delta(R) \text{ je } (s,t)\text{-řez}\}$$

Společně s důkazem věty si ukážeme i základní myšlenku, jak zvětšovat velikost toku. Předpokládejme, že už známe nějaký tok f. Pokud v grafu najdeme orientovanou cestu sPt takovou, že pro každou hranu e cesty P je $f(e) < c(e)$, tak zvětšíme průtok cestou P a tím zvětšíme i tok f. Tato myšlenka ale sama o sobě nestačí. Proč?

[4] Kotzig byl Slovák působící na bratislavské Vysoké škole ekonomické.

Pokud ze spotřebiče do zdroje vede hrana ts s průtokem $f(ts) > 0$, tak celkový tok zvětšíme tím, že z s do t pošleme tok o velikosti $f(ts)$ po hraně ts v protisměru. Ve skutečnosti nic v protisměru nepoteče. Průtoky hranou jdoucí proti sobě se odečtou. Místo toho, aby se z t posílal do s tok o velikosti $f(ts)$ (původní tok po hraně) a zároveň z s do t tok o velikosti $f(ts)$ (přidávaný tok po hraně), tak se vrcholy s a t dohodnou, že si každý nechá tok o velikosti $f(ts)$. Nebudou si nic vyměňovat, protože to vyjde na stejno.

Množství toku, které můžeme poslat po směru hrany e, nazveme *rezerva po směru hrany*. Její velikost je $c(e) - f(e)$. Množství toku, které můžeme poslat proti směru hrany e, nazveme *rezerva proti směru hrany*.[5] Její velikost je $f(e)$.



Cestou v následující definici vylepšující cesty myslíme cestu, u které ignorujeme orientaci hran. Hranu cesty zorientovanou směrem od zdroje do spotřebiče nazveme *dopřednou* a opačně orientovanou hranu nazveme *zpětnou*. Cesta $v_0e_1v_1e_2v_2 \dots e_kv_k$ je *vylepšující cesta pro tok* f, pokud pro každou dopřednou hranu e cesty platí $f(e) < c(e)$ a pro každou zpětnou hranu platí $f(e) > 0$. Cestu vedoucí ze zdroje s do spotřebiče t budeme zkráceně označovat jako (s,t)-cestu.

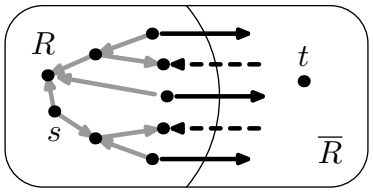
Lemma 20 *Pokud v síti existuje vylepšující cesta P pro tok f vedoucí ze zdroje do spotřebiče, tak tok f není maximální.*

Důkaz: Tok f můžeme vylepšit podél vylepšující cesty sPt. Nechť $\varepsilon = \min\{\varepsilon_1, \varepsilon_2\}$, kde $\varepsilon_1 = \min\{c(e) - f(e) \mid e \in P \text{ je dopředná}\}$ a $\varepsilon_2 = \min\{f(e) \mid e \in P \text{ je zpětná}\}$. Slovy se dá říci, že ε je největší tok, který se dá poslat ze zdroje do spotřebiče podél cesty P. Vylepšením toku f podél cesty P dostaneme tok f'.

$$f'(e) := \begin{cases} f(e) + \varepsilon, & e \in P \text{ je dopředná hrana,} \\ f(e) - \varepsilon, & e \in P \text{ je zpětná hrana,} \\ f(e), & e \notin P. \end{cases}$$

Funkce f' je opět tok, protože splňuje definici toku (ověřte). ∎

Důkaz: (Věty 8 o maximálním toku a minimálním řezu) Každý tok je menší nebo roven velikosti libovolného řezu (důsledek 1). To dokazuje první nerovnost.



Pro důkaz druhé nerovnosti vezmeme maximální tok f a budeme chtít najít řez stejné velikosti. V síti neexistuje vylepšující cesta ze zdroje do spotřebiče, protože jinak se dostaneme do sporu s lemmatem 20.

Nechť $R = \{v \in V$ do kterých vede vylepšující cesta z $s\}$. Množina $\delta(R)$ je (s,t)-řez, protože z s do s vede vylepšující cesta nulové délky, ale z s do t žádná vylepšující cesta nevede. Pro každou hranu řezu $\delta(R)$ je $f(e) = c(e)$ a pro každou hranu řezu $\delta(\overline{R})$ je $f(e) = 0$. Jinak by šla vylepšující cesta prodloužit do vrcholů mimo R. Podle lemmatu 19 je velikost toku $f(s) = f(\delta(R)) - f(\delta(\overline{R})) = c(\delta(R))$, což jsme chtěli ukázat. ∎

Důsledek 2 *Tok f je maximální $\iff$ neexistuje vylepšující cesta pro tok f.*

Předchozí důkaz nám dokonce ukazuje, jak najít minimální řez, který dosvědčí, že větší tok neexistuje.

[5] Později (u Dinicova algoritmu) rozšíříme graf G tak, aby ke každé hraně uv existovala opačná hrana vu, a díky tomu zavedeme rezervy trochu jednodušším způsobem.

12.2 Algoritmy vylepšující cesty

12.2.1 Ford-Fulkersonův algoritmus

Důkaz věty o maximálním toku a minimálním řezu je zároveň i návodem jak hledat maximální tok. Začneme s nulovým tokem a postupně budeme hledat vylepšující cesty, podél kterých zvětšíme aktuální tok. Když už nebude existovat vylepšující cesta, tak máme maximální tok. Dostáváme tak Ford-Fulkersonův algoritmus (z roku 1957).

```
Ford-Fulkerson:
    f := 0
    while existuje vylepšující cesta P z s do t do
        vylepši tok f podél cesty P
    return f
```

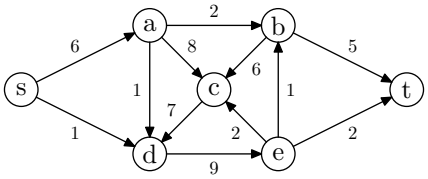
Jak hledat vylepšující cesty si vysvětlíme až v další sekci u Dinicova algoritmu.

Podívejme se, jak je to s konečností Ford-Fulkersonova algoritmu. Pokud má síť celočíselné kapacity, tak v každém kroku stoupne velikost toku alespoň o jedna. Součet všech kapacit, který je horním odhadem na velikost toku, je konečné číslo a proto se algoritmus po konečně mnoha krocích zastaví. Pokud má síť racionální kapacity, tak je můžeme přenásobit nejmenším společným jmenovatelem a tím úlohu převedeme na předchozí případ (průběh algoritmu se tím nezmění). Je zajímavé, že důkaz konečnosti nemůžeme rozšířit na sítě s iracionálními kapacitami. Dokonce existují sítě s iracionálními kapacitami, ve kterých se algoritmus zacyklí (viz třetí z následujících příkladů).

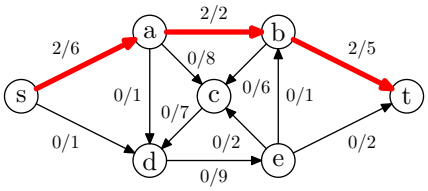
V celočíselné síti zvýšíme tok podél vylepšující cesty vždy o celé číslo. Proto dostáváme následující důsledek.

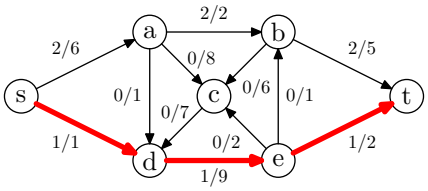
Důsledek 3 *V síti s celočíselnými kapacitami hran existuje maximální tok, který je celočíselný.*

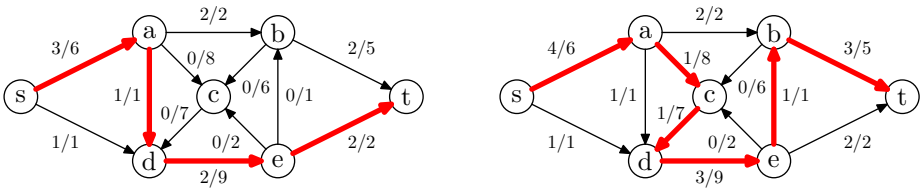
Příklad: (průběh algoritmu) Najděte maximální tok v síti na následujícím obrázku.



Budeme postupovat podle Ford-Fulkersonova algoritmu. Začneme s nulovým tokem a postupně budeme hledat vylepšující cesty. Průběh algoritmu je naznačen na následujících obrázcích (po řádkách). Každý obrázek zachycuje stav po vylepšení toku podél vyznačené vylepšující cesty.

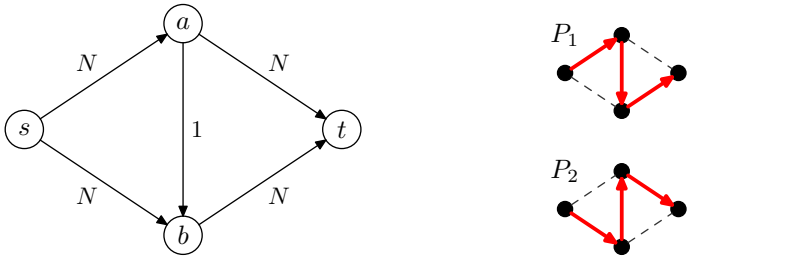






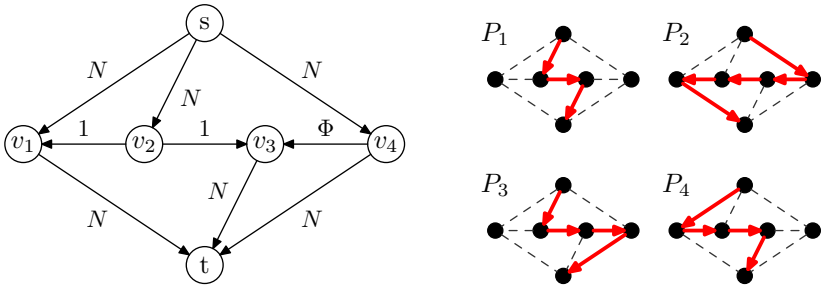
Po čtvrtém vylepšení už vylepšující cesta neexistuje. Proto je nalezený tok velikosti 5 maximální. Pokud bychom chtěli najít minimální řez, tak můžeme postupovat podle návodu z důkazu věty o maximálním toku a minimálním řezu. Vylepšující cesta ze zdroje vede jen do vrcholů $\{s, a, c, d, e\}$ a ty také určují minimální řez velikosti 5.

Příklad: (počet iterací závisí na ohodnocení) Doba běhu algoritmu může značně záviset na velikosti kapacit. Podívejme se na následující síť. Pokud budeme střídavě nacházet vylepšující cestu P_1 a vylepšující cestu P_2, tak budeme muset provést $2N$ vylepšení, než dostaneme maximální tok.



Na příkladu je také vidět, že stačilo vybrat dvě vhodné vylepšující cesty. Jednu vedoucí horem a druhou vedoucí spodem. To nás přivádí k vylepšení, které provedl Edmonds a Karp. Ukázali, že se je výhodné hledat vylepšující cestu s nejmenším počtem hran.

Příklad: (zacyklení, Uri Zwick) Pokud jsou kapacity hran reálná čísla, tak se algoritmus může zacyklit. Uvažme síť na následujícím obrázku. Dvě hrany sítě mají kapacitu jedna, šest hran má kapacitu N, kde N je dostatečně velké číslo, a jedna hrana má kapacitu $\Phi = (\sqrt{5}-1)/2 \approx 0.618$. Číslo Φ je zvoleno tak, aby $1 - \Phi = \Phi^2$. Přenásobením rovnice členem Φ^k dostaneme $\Phi^k - \Phi^{k+1} = \Phi^{k+2}$.



V průběhu Ford-Fulkersonova algoritmu budeme na vodorovných hranách sledovat rezervy po směru hrany a zapisovat je zleva doprava do uspořádané trojice. Připomeňme, že rezerva po směru hrany e je $c(e) - f(e)$.

Začneme s nulovým tokem. Vylepšením toku podél cesty P_1 dostaneme na vodorovných hranách rezervy $(1, 0, \Phi)$. Dále budeme pracovat jednotlivých iteracích. V každé iteraci postupně provedeme čtyři vylepšení podél cest P_2, P_3, P_2, P_4. Předpokládejme, že na začátku iterace jsou rezervy vodorovných hran $(\Phi^{k-1}, 0, \Phi^k)$.

V iteraci postupně zvětšíme tok o Φ^k, Φ^k, Φ^{k+1} a Φ^{k+1}. Rezervy na vodorovných hranách postupně budou

$$\xrightarrow{P_2} (\Phi^{k+1}, \Phi^k, 0) \xrightarrow{P_3} (\Phi^{k+1}, 0, \Phi^k) \xrightarrow{P_2} (0, \Phi^{k+1}, \Phi^{k+2}) \xrightarrow{P_4} (\Phi^{k+1}, 0, \Phi^{k+2})$$

Na konci n-té iterace (to je po $4n+1$ vylepšeních) budou rezervy vodorovných hran Φ^{2n-2}, 0, Φ^{2n-1}. S rostoucím počtem vylepšení konverguje velikost nalezeného toku k hodnotě

$$1 + 2\sum_{i=1}^{\infty} \Phi^i = \frac{2}{1-\Phi} - 1 = 2 + \sqrt{5} < 5.$$

Na druhou stranu je zřejmé, že velikost maximálního toku je $2N + 1$, kde N je libovolně velké číslo.

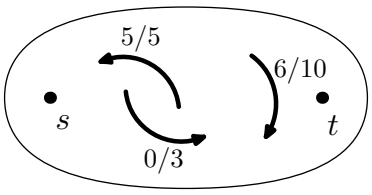
12.2.2 Dinicův/Edmonds-Karpův algoritmus

Pokud budeme ve Ford-Fulkersonově algoritmu volit nejkratší vylepšující cestu (s nejmenším počtem hran), tak se dramaticky zlepší časová složitost celého algoritmu.

Tento nápad uvedli ve své práci už Ford a Fulkerson, ale popsali ho jako heuristiku. Jako první provedl analýzu této heuristiky ruský matematik Dinits (často překládán jako Dinic) v roce 1970. Edmonds a Karp nezávisle publikovali slabší analýzu v roce 1972. Ale protože to byla první anglicky publikovaná analýza, tak se algoritmus často označuje jako Edmonds-Karpův. Dinic navíc přišel s vrstevnatou (čistou) sítí a blokujícím tokem a pomocí něj ukázal rychlejší implementaci algoritmu. Proto budeme algoritmus označovat jako Dinicův.

Ve skutečnosti je stejně těžké najít vylepšující cestu jako najít nejkratší vylepšující cestu. Oboje můžeme řešit průchodem do šířky. Proto je vylepšení algoritmu tak jednoduchou modifikací, že bychom ji ve Ford-Fulkersonově algoritmu použili, aniž bychom o tom věděli.

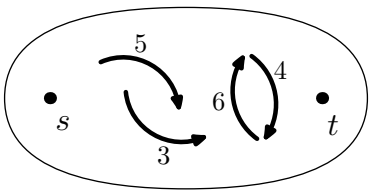
Nejprve si ukážeme, jak jednoduše hledat nejkratší vylepšující cestu.



Původní síť G

1) Máme síť s grafem G a tokem f (*původní síť*). Pro zjednodušení výkladu předpokládejme, že v G ke každé hraně uv existuje opačná hrana vu. Pokud ne, tak do G přidáme hranu vu s nulovou kapacitou. Toto rozšíření grafu G nijak nezmění aktuální, ani maximální tok, ale zjednoduší se zavedení a práci s rezervou hrany.

2) Chceme vytvořit pomocnou síť, která nám zjednoduší hledání vylepšující cesty. Nechceme se dívat na hrany v protisměru, ani nechceme, aby v síti existovaly násobné orientované hrany. Tuto síť vytvoříme na základě původní sítě a toku f. Nazveme ji *síť rezerv* G_f.

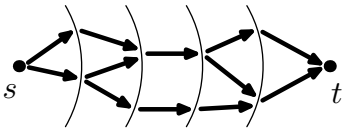


Síť rezerv G_f

Rezerva $r(uv)$ říká, jak velký tok protlačíme z u do v, a odpovídá součtu rezervy hrany uv po směru a rezervy hrany vu v protisměru. Spočítá se jako

$$r(uv) = (c(uv) - f(uv)) + f(vu).$$

Do sítě rezerv dáme jen ty hrany (rozšířené) původní sítě, které mají nenulovou rezervu, a ohodnotíme je rezervou. Každá orientovaná cesta v síti rezerv odpovídá vylepšující cestě v původní síti.



Čistá síť G_f^*

3) Na základě sítě rezerv vytvoříme *čistou síť* G_f^*. Do čisté sítě dáme jen ty hrany sítě rezerv, které leží na nejkratší cestě ze zdroje do spotřebiče. Můžeme ji zkonstruovat pomocí průchodu do šířky, který rozdělí vrcholy do vrstev podle vzdálenosti od zdroje. Proto se této síti někdy říká *vrstevnatá síť.*

Hrana e původní sítě je *nasycená* vzhledem k toku f, pokud $r(e) = 0$ (hranou $e = uv$ i opačnou hranou vu v protisměru protéká největší možný tok směrem z u do v). Orientovaná cesta je *nasycená*, pokud obsahuje nasycenou hranu. Nasycená cesta je tedy opakem vylepšující cesty.

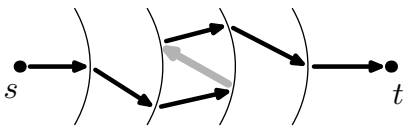
Analýza Dinicova/Edmonds-Karpova algoritmu

Délkou orientované cesty myslíme počet hran na cestě. Vzdálenost z vrcholu x do vrcholu y je délka nejkratší orientované cesty z x do y v síti rezerv G_f. Označíme ji $d_f(x, y)$. Pro cesty vedoucí ze zdroje s píšeme zkráceně $d_f(y)$ místo $d_f(s, y)$.

Klíčové lemma říká, že po vylepšení toku podél nejkratší vylepšující (s, t)-cesty neklesne v síti rezerv délka nejkratší cesty ze zdroje do spotřebiče.

Vylepšením toku podél vylepšující cesty se některé hrany nasytí. Jejich rezerva klesne na nulu a proto zmizí ze sítě rezerv. Tím délka nejkratší cesty v G_f určitě neklesne.

Na druhou stranu se v síti rezerv mohou objevit nové hrany. Jsou to hrany, které měly nulovou rezervu, ale při vylepšení toku jsme po opačné hraně poslali nenulový tok. Každá nová hrana vede z i-té vrstvy čisté sítě do $(i-1)$-ní (pro nějaké i). Každá (s, t)-cesta používající alespoň jednu novou hranu, musí alespoň jednou skočit o vrstvu zpět, ale nikdy nemůže skočit více než o jednu vrstvu dopředu. Proto je nová cesta alespoň o 2 delší, než byla délka cesty, podle které jsme vylepšovali tok.



Lemma zformulujeme malinko obecněji a ukážeme, že po vylepšení neklesne žádná vzdálenost ze zdroje s do libovolného vrcholu v.

Lemma 21 *Nechť f je tok a f' je tok, který vznikne vylepšením f podél nejkratší vylepšující cesty P. Potom pro každý vrchol $v \in V$ platí $d_{f'}(v) \geq d_f(v)$.*

Důkaz: Označme vrcholy na nejkratší vylepšující cestě P v síti G_f jako $s = v_0, v_1, \ldots, v_k = t$. Předpokládejme pro spor, že existuje vrchol v takový, že $d_{f'}(v) < d_f(v)$. Ze všech takových vrcholů si vybereme ten s nejmenším $d_{f'}(v)$. Určitě platí $v \neq s$. Nechť w je předposlední vrchol na nejkratší cestě do v v síti $G_{f'}$, potom $d_{f'}(v) = d_{f'}(w) + 1$. Z volby vrcholu v platí $d_f(w) \leq d_{f'}(w)$.

Hrana wv se musela v grafu objevit až po vylepšení, jinak bychom průchodem po hraně wv v síti G_f dostali $d_f(v) \leq d_f(w) + 1 \leq d_{f'}(w) + 1 \leq d_{f'}(v)$. Proto je wv opačnou hranou k hraně $v_{i-1}v_i$ na cestě P, pro nějaké i. Potom je $d_f(w) = i$ a $d_f(v) = i - 1$. Na druhou stranu je $d_{f'}(v) \geq d_{f'}(w) + 1 \geq i + 1$. To je spor. ∎

Následující lemma říká, že pokud po vylepšení toku podél nejkratší vylepšující (s, t)-cesty nevzroste délka nejkratší (s, t)-cesty, bude nová čistá síť podgrafem původní čisté sítě. Proto stačí aktualizovat původní čistou síť a nemusíme ji po každém vylepšení počítat znova.

Lemma 22 *Nechť f je tok a f' je tok, který vznikne vylepšením f podél nejkratší vylepšující cesty. Pokud $d_f(t) = d_{f'}(t)$, tak $G^*_{f'} \subseteq G^*_f$.*

Důkaz: Čistá síť obsahuje právě hrany ležící na nejkratší cestě ze zdroje do spotřebiče. Nechť $k := d_f(t)$.

Během vylepšování toku podél cesty se mohou objevit nové hrany. Z předchozího důkazu ale vyplývá, že každá cesta ze zdroje do spotřebiče používající novou hranu je alespoň o dva delší než k. Proto se žádná nová hrana nemůže objevit v čisté síti a tedy čistá síť $G^*_{f'}$ je podgrafem předchozí čisté sítě G^*_f. ∎

Lemma 23 *Dinicův/Edmonds-Karpův algoritmus provede vylepšení podél nejvýše mn vylepšujících cest.*

Důkaz: Délka nejkratší vylepšující cesty v průběhu algoritmu neklesá. Proto můžeme běh algoritmu rozdělit do fází podle délky nejkratší vylepšující cesty. Fází je nejvýše tolik, kolik je různých délek cest a to je nejvýše n. Podle lemma 22 do čisté sítě během fáze nepřibudou žádné hrany. V každé fázi provedeme nejvýše m vylepšení, protože se při každém vylepšení nasytí aspoň jedna hrana a zmizí z čisté sítě. ∎

Dinicův algoritmus

Na čistou síť G^*_f s rezervami se můžeme dívat jako na obyčejnou síť s kapacitami (kapacitou každé hrany je velikost rezervy) a můžeme v ní hledat tok. Graf čisté sítě je acyklický orientovaný graf. Tok ϕ v acyklické orientované síti je *blokující tok*, pokud je každá orientovaná (s,t)-cesta v G^*_f nasycená. Důležité je slovo orientovaná. Cesta obsahující hranu v protisměru není přípustná. Blokující tok nemusí být maximální tok, protože může existovat vylepšující cesta používající hrany v protisměru.

Blokující tok můžeme najít pomocí vylepšujících cest, které nevyužívají rezervy v protisměru. Blokující tok ϕ v čisté síti G^*_f je roven toku, o který zvětšíme f během jedné fáze Dinicova algoritmu, tj. při vylepšování toku f podél vylepšujících cest stejné délky.

```
1: Dinic:
2:     zvol počáteční tok, například f := 0
3:     repeat
4:         spočítej síť rezerv
5:         spočítej čistou síť
6:         nalezni blokující tok v čisté síti a přičti ho k f
7:     until spočítaná čistá síť obsahovala hrany
8:     return f
```

Když nově spočítaná čistá síť neobsahuje hrany, tak můžeme skončit, protože neexistuje cesta ze zdroje do spotřebiče v G_f, tedy ani žádná vylepšující cesta v G. V ten moment máme maximální tok.

Repeat-cyklus proběhne nejvýše n-krát, protože v každé iteraci se zvětší délka nejkratší vylepšující cesty. Provedení kroků 4 a 5 bude trvat čas $\mathcal{O}(n+m)$, protože oba kroky provedeme pomocí průchodu grafu. Jak se provede krok 6 si ukážeme za chvilku. Ukážeme, že krok 6 trvá čas $\mathcal{O}(nm)$. Dohromady dostaneme časovou složitost Dinicova algoritmu $\mathcal{O}(n^2m)$.

```
1: nalezení blokujícího toku v čisté síti:
```

```
2:  while čistá síť obsahuje hrany do
3:      najdi v čisté síti cestu ze zdroje do spotřebiče
4:      spočítej hodnotu nejmenší rezervy na cestě
5:      vylepši tok f podél cesty a uprav čistou síť
6:      dočisti čistou síť
```

Krok 3 provedeme hladově například průchodem do hloubky. Při návratu v průchodu do hloubky můžeme rovnou počítat krok 4. Proto budou oba kroky trvat čas $\mathcal{O}(n)$. Stejně tak vylepšení toku podél nalezené cesty.

Jak budeme upravovat a dočisťovat čistou síť? Pro každý vrchol si budeme pamatovat jeho vstupní a výstupní stupeň. Vylepšením toku klesne rezerva některých hran na nulu. Takové hrany musíme z čisté sítě vymazat. Musíme si ale dát pozor, aby nám vymazáním některých hran nevznikly slepé uličky. To jsou cesty vedoucí do vrcholů, ze kterých už nejde pokračovat dál. Proto při vymazávaní každé hrany vložíme její konce do fronty. Při dočišťování čisté sítě postupně probíráme vrcholy ve frontě a pokud mají vstupní nebo výstupní stupeň nula, tak vymažeme všechny hrany z nich vedoucí. Druhé konce mazaných hran vkládáme opět do fronty a při tom aktualizujeme vstupní a výstupní stupně těchto vrcholů. Po zpracování fronty dostaneme korektní čistou síť, ve které můžeme znova začít hledat vylepšující cestu.

Čas za zpracování fronty budeme účtovat jednotlivým hranám. Každý vrchol, který byl vložen do fronty, odpovídá jedné smazané hraně. Hrana mohla do fronty vložit nejvýše své dva koncové vrcholy. Z toho vyplývá, že časová složitost všech provedení kroků 6 je $\mathcal{O}(m)$. While-cyklus proběhne nejvýše m-krát, protože pokaždé vymažeme alespoň jednu hranu čisté sítě. Celková časová složitost nalezení blokujícího toku v čisté síti je $\mathcal{O}(mn)$.

Poznámky k implementaci

Ve skutečnosti nepotřebujeme rozlišovat mezi sítí rezerv a čistou sítí. V iteraci nám stačí jen jedna síť. Nejprve spočítáme síť rezerv. V té provedeme průchod do šířky, během kterého umazáváme hrany neležící na nejkratší cestě ze zdroje do spotřebiče. Nechť S je množina vrcholů ležících na nějaké nejkratší cestě ze zdroje do spotřebiče. Výpočet čisté sítě provedeme následovně. Na začátku je $S = \{t\}$. Síť rezerv procházíme do šířky a při každém návratu po hraně uv hranu smažeme, pokud $v \notin S$, jinak přidáme u do S a hranu necháme v čisté síti. Tím dostaneme čistou síť.

Také není potřeba provádět dokonalé dočišťování čisté sítě. Do vrcholů, do kterých nevede žádná cesta, se nemáme jak dostat. Proto takové vrcholy a cesty z nich vedoucí můžeme v síti nechat.

Odstraňování vrcholů na slepých uličkách, ze kterých nevede cesta dál, můžeme provést podobným trikem, jako při hledání čisté sítě. Čistou sít nebudeme dočišťovat v kroku 6, ale až v kroku 3 při dalším průběhu cyklu. V kroku 3 hledáme cestu ze zdroje do spotřebiče v čisté síti. Hledání provedeme pomocí průchodu do hloubky. Pokud bychom při průchodu do hloubky přešli po hraně uv takové, že z v nelze pokračovat dál, tak při návratu hranu uv smažeme. Mazání naúčtujeme mazaným hranám.

Ve speciálních sítích má Dinicův algoritmus ještě lepší časovou složitost. O tom se ale dozvíte více ve cvičeních. Zájemce také můžeme odkázat na Schrijvera [27] nebo Mareše [22].

12.2.3 Metoda tří Indů

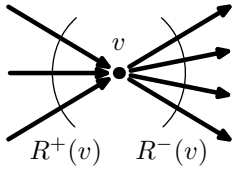
Indové Malhotra, Kumar a Maheshawari v roce 1978 vymysleli efektivnější algoritmus, jak nalézt blokující tok v čisté síti. Jejich metoda běží v čase $\mathcal{O}(n^2)$, což

zlepšuje čas Dinicova algoritmu na $\mathcal{O}(n^3)$.

Pro každý vrchol si spočítáme, jak velký tok může protékat skrz vrchol. Někdy místo „průtok skrz vrchol“ říkáme, jak velký tok jde protlačit skrz vrchol. Největší možný průtok přes vrchol v nazveme rezervou vrcholu v a označíme ho $R(v) := \min\{R^+(v), R^-(v)\}$, kde

$$R^+(v) = \sum_{xv \in E} r(xv),$$

$$R^-(v) = \sum_{vx \in E} r(vx).$$

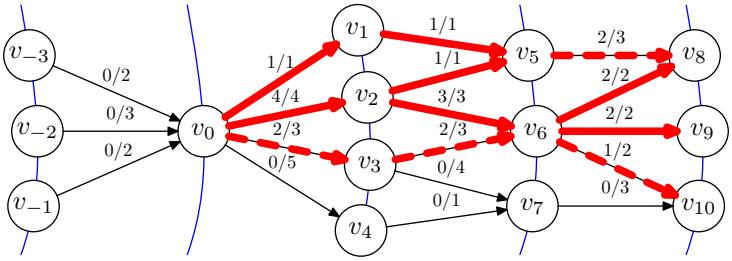


1: nalezení blokujícího toku v čisté síti (podle tří Indů):
2: spočítej $R(v)$ pro každý $v \in V$
3: **while** $V \neq \emptyset$ **do**
4: $v_0 :=$ vrchol $v \in V$ s minimálním $R(v)$
5: **if** $R(v_0) = 0$ **then**
6: $V := V \setminus \{v_0\}$
7: uprav $R(v)$ pro sousedy v_0
8: **else**
9: najdi tok velikosti $R(v_0)$ procházející vrcholem v_0
pomocí protlačení doleva a doprava
a uprav hodnoty $R(v)$

Kroky 5–7 odpovídají pročišťování čisté sítě. Z grafu vyloučíme vrchol v_0 i s hranami, které vedou z v_0 nebo do v_0. Kromě odstranění vrcholů zpracovaných v předchozím průběhu cyklu se takto odstraňují i slepé uličky (rozmyslete si, jak mohou slepé uličky vzniknout a jak je algoritmus odstraní).

Krok 2 bude trvat $\mathcal{O}(m)$. Cyklus proběhne n-krát, protože v každé iteraci vyhodíme z množiny vrcholů jeden vrchol. Ostatní kroky, kromě kroku 9 jsou proveditelné v čase $\mathcal{O}(n)$. Časovou složitost kroku 9 budeme počítat zvlášť a budeme ji účtovat hranám a vrcholům sítě.

Jak probíhá krok 9? Ukážeme si jen protlačování toku doprava. Protlačení toku doleva proběhne symetricky. Protlačování provádíme po vrstvách směrem od v_0.



Na začátku dáme do fronty jen v_0. Vrcholy z fronty postupně zpracováváme následujícím způsobem. Představme si, že už jsme ve vrcholu v, do kterého jsme dotlačili přebytek toku o velikosti K. Postupně probíráme hrany, které vedou z vrcholu v doprava, a snažíme se po nich poslat co největší tok. Pokud bude přebytek toku ve v větší, než rezerva probírané hrany, tak hranu nasytíme a postoupíme k další hraně. Z vrcholu vždy vede další hrana, po které můžeme tok poslat, protože $K \leq R(v_0) \leq R^-(v)$. Druhé konce hran, po kterých jsme poslali nějaký tok, vložíme do fronty.

Práci s hranami, které jsme nasytili, naúčtujeme hranám (nasycené hrany zmizí z čisté sítě). Práci s poslední hranou, po které jsme poslali nějaký tok, ale nemuseli jsme ji nasytit, naúčtujeme vrcholu v. Celkem jsme během algoritmu naúčtovali

každé hraně nejvýše jednu jednotku práce a každému vrcholu nejvýše n jednotek práce. Proto je celková časová složitost kroku 9 rovna $\mathcal{O}(n^2 + m)$.

Časová složitost celého algoritmu na nalezení blokujícího toku podle metody tří Indů je $\mathcal{O}(n^2)$. To dává časovou složitost nalezení maximálního toku $\mathcal{O}(n^3)$.

Poznámka k implementaci

Můžeme se vyhnout použití fronty při protlačování toku. Během kroku 2 si v čase $\mathcal{O}(m)$ spočítáme topologické uspořádání vrcholů čisté sítě (topologické uspořádání hledáme pro čistou síť pouze jednou). Během protlačování toku doprava postupně v topologickém pořadí probíráme vrcholy, které jsou topologicky větší než v_0 a pokud mají kladný přebytek toku, tak přebytek protlačíme do sousedních vrcholů. Podobně při protlačování toku doleva.

12.3 Goldbergův Push-Relabel algoritmus

V algoritmech vylepšující cesty se tok podél jedné hrany postupně nasčítává z toků podél vylepšujících cest. Těchto cest může být poměrně mnoho a nalezení zlepšující cesty může trvat až $\mathcal{O}(n)$. Proto se naskýtá myšlenka, jestli nemůžeme tok podél hrany poslat naráz.

Ukážeme si algoritmus, který je založen na daleko jednodušší myšlence než jsou vylepšující cesty. Algoritmus používá dvě základní operace: protlačení toku po hraně (push)[6] a zvýšení výšky vrcholu (relabel). Proto se algoritmus nazývá push-relabel. Algoritmus vymyslel Goldberg v roce 1985. Varianta, kterou si ukážeme je podle Goldberga a Tarjana z roku 1988.

Připomeňme, že G je orientovaný graf, jehož hrany jsou ohodnoceny kapacitami $c : E \to \mathbb{R}_+$. Graf G rozšíříme tak, aby ke každé hraně uv existovala opačná hrana vu. Přidávané hrany budou mít kapacitu nula, takže nijak neovlivní tok v síti (ve skutečnosti žádné hrany přidávat nemusíme, používáme je jen pro zjednodušení definice rezervy). Rozšířenému grafu budeme říkat původní síť.

Budeme pracovat s pomocným grafem G_f (síť rezerv) podobně jako v algoritmech pracujících s vylepšující cestou. Pro dvojici G a f dáme do grafu G_f každou hranu původního grafu s nenulovou rezervou. Připomeňme, že rezerva hrany uv je $r(uv) = (c(uv) - f(uv)) + f(vu)$. Rezerva $r(uv)$ znamená, že můžeme skrz hranu uv, případně hranu vu, *protlačit* $r(uv)$ jednotek toku z vrcholu u do vrcholu v. Aby v původním grafu po dvojici hran uv, vu neproudil tok tam i zpět, tak nejprve protlačíme co největší část toku v protisměru po vu a pak teprve zbytek toku po uv. Změnou toku se změní i pomocný graf G_f.

Dále připomeňme, že $f(v)$ značí *bilanci vrcholu* v, nebo také *přebytek* toku ve vrcholu v.

Myšlenka protlačování

Nejprve zavedeme jeden klíčový pojem. Funkce $f : E \to \mathbb{R}_+$ je *pratok*[7], pokud splňuje

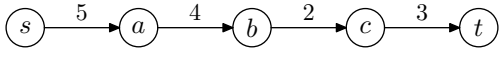
$$\begin{array}{lll} i) & 0 \le f(e) \le c(e) & \text{pro každou hranu } e \in E \\ ii) & f(v) \ge 0 & \text{pro každý vrchol } v \in V \setminus \{s,t\} \end{array}$$

Řekneme, že vrchol $v \in V \setminus \{s,t\}$ je *aktivní,* pokud $f(v) > 0$. Tedy pokud do vrcholu přitéká více, než z něj odtéká. Vrcholy s, t nejsou nikdy aktivní.

Jak se liší pratok od toku? V definici toku platí druhá podmínka s rovností ($f(v) = 0$ pro každý vrchol $v \in V \setminus \{s,t\}$).

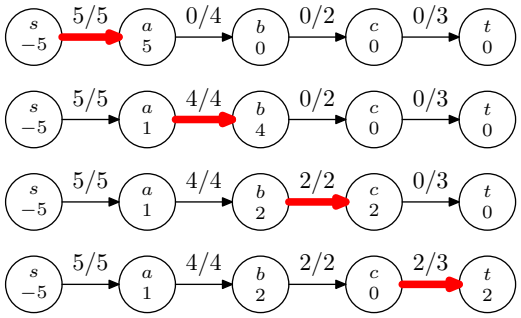
Podívejme se na základní myšlenku push-relabel algoritmu. Nejprve protlačíme ze zdroje co největší tok do sousedních vrcholů. Dále budeme probírat aktivní vrcholy a snažit se protlačit přebytek toku v nich do sousedních vrcholů směrem ke spotřebiči. Při protlačování toku nesmíme překročit kapacitu hran. Protlačování bude probíhat pouze po hranách s nenulovou rezervou. Postupně budeme chtít protlačit všechny přebytky toku až do spotřebiče. Když to nepůjde, tak je protlačíme zpátky do zdroje.

Podívejme se na příklad sítě na obrázku, ve které najdeme maximální tok pomocí protlačování toku po hranách.

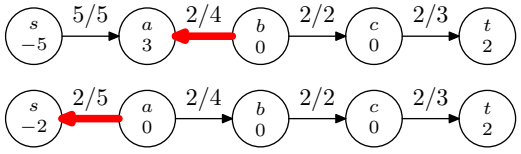


[6] Protlačování toku po hraně je podobné protlačování toku v metodě tří Indů.
[7] z anglického preflow

Průběh protlačování toku si můžeme představit jako vlnu, která se šíří ze zdroje do spotřebiče. Tam se odrazí a valí se zpátky do zdroje. Jednotlivé kroky protlačování jsou na následujících obrázcích. U každé hrany na obrázku je hodnota „$f(e)/c(e)$“. Ve vrcholech je uvedeno jméno vrcholu a aktuální přebytek toku. V první fázi postupně protlačujeme co nejvíce toku ze zdroje směrem ke spotřebiči. Ve zdroji s je nekonečně mnoho vody a tak po hraně sa protlačíme 5 jednotek toku do a. V dalším kroku protlačíme po hraně ab co největší část přebytku $f(a)$. Dále postupujeme podobně a to tak dlouho, dokud nedorazíme do spotřebiče.



Do spotřebiče jsme dotlačili největší možný tok, ale vrcholy a, b jsou stále aktivní. V grafu G_f nevede orientovaná cesta z aktivních vrcholů do spotřebiče t a proto nastane druhá fáze, ve které dotlačíme přebytek toku z aktivních vrcholů zpátky do zdroje.



Skončili jsme s maximálním tokem.

Na cestě je hledání toku jednoduché, ale v jakém pořadí provádět jednotlivá protlačení v obecném grafu?

Musíme si dát pozor, abychom se nezacyklili. Například by se mohlo stát, že budeme neustále protlačovat tok po jedné hraně tam a zpátky, tam a zpátky, ...

Rozhodování, podél kterých hran budeme tok protlačovat, provedeme na základě odhadu vzdáleností v G_f. Přebytky toku budeme protlačovat po nejkratších cestách do spotřebiče. Za chvíli si vysvětlíme, co je to výška každého vrcholu. Dolním odhadem vzdálenosti dvou vrcholů potom bude jejich výškový rozdíl. Samotný algoritmus nebude přemýšlet nad nejkratšími cestami do spotřebiče, ale bude tok posílat po libovolné hraně vedoucí z kopce dolů.

Platné označkování a výšky vrcholů

Vektor $d \in (\mathbb{N}_0 \cup \{\infty\})^n$ nazveme *platné označkování*[8] vrcholů vzhledem k toku f, pokud

$$\begin{aligned} &i) \quad d(s) = n, \quad d(t) = 0, \\ &ii) \quad d(v) \leq d(w) + 1 \qquad \text{pro každou hranu } vw \in E(G_f). \end{aligned}$$

Pod hodnotou $d(v)$ si budeme představovat *výšku,* ve které se vrchol v nachází. Protlačování toku budeme provádět podél hran vedoucích z kopce dolů. To je tak, jak voda přirozeně teče. Platné označkování říká, že zdroj bude vždy ve výšce n,

[8] z anglického valid labeling

spotřebič ve výšce 0. Neklade žádná omezení na stoupání, ale říká, že žádná hrana G_f nevede příliš strmě dolů.[9] Hrana může klesat nejvýše o jedna.

Z existence platného označkování vrcholů vyplývá důležitá vlastnost pratoku, která říká, že pratok „nasycuje" jistý řez. Řez $\delta(R)$ je *nasycený,* pokud pro každou hranu $e \in \delta(R)$ je $f(e) = c(e)$ a pro každou hranu $\delta(\overline{R})$ je $f(e) = 0$. Připomeňme, že tok je vždy menší roven velikosti řezu. Pokud pro tok f najdeme řez $\delta(R)$ nasycený tokem f, tak víme, že je tok maximální (platí $c(\delta(R)) = |f|$).

Lemma 24 *Nechť f je pratok a d je platné označkování pro f. Potom existuje nasycený (s,t)-řez $\delta(R)$.*

Důkaz: Protože má graf G_f jen n vrcholů, tak existuje hodnota k, $0 < k < n$, taková, že $d(v) \neq k$ pro všechny vrcholy $v \in V$. Položme $R = \{v \in V : d(v) > k\}$. Potom $s \in R$ a $t \notin R$, protože $d(s) = n$ a $d(t) = 0$. Z bodu *ii)* definice platného označkování plyne, že žádná hrana G_f nevede z R ven, protože nemůže klesnout o více než jedna. ∎

Důsledek 4 *Pokud existuje platné označkování pro tok f, tak je tok f maximální.*

Důsledek nám dává podmínku pro zastavení algoritmu. Push-relabel algoritmus si neustále udržuje platný pratok a platné označkování (tedy i nasycený řez). Skončí v momentě, kdy se z pratoku stane tok. V jistém smyslu je duální k algoritmům vylepšující cesty, protože ty si udržují platný tok a skončí, až se některý řez nasytí.

Připomeňme, že $d_f(v,w)$ je orientovaná vzdálenost v grafu G_f, tj. počet hran na nejkratší orientované cestě z v do w.

Ukážeme si, že rozdíl výšek dvou vrcholů je dolním odhadem jejich vzdálenosti v G_f.

Lemma 25 *Nechť f je pratok a d platné označkování. Potom pro každé dva vrcholy $v, w \in V$ platí $d_f(v,w) \geq d(v) - d(w)$.*

Důkaz: Pokud je $d_f(v,w) = \infty$, tak lemma platí. Předpokládejme tedy, že je $d_f(v,w)$ konečné. Uvažme nejkratší orientovanou (v,w)-cestu v G_f. Pro každou hranu pq orientované cesty z definice platného označkování platí $d(p) - d(q) \leq 1$. Sečtením těchto nerovností podél hran orientované cesty dostaneme výsledek.[10] ∎

Ve speciálním případě lemma říká, že

- $d(v)$ je dolním odhadem na $d_f(v,t)$ a že
- $d(v) - n$ je dolním odhadem na $d_f(v,s)$.

Poznamenejme, že $d(v) \geq n$ znamená, že $d_f(v,t) = \infty$. Tedy že v G_f nevede orientovaná cesta z v do spotřebiče t a proto by se měl v tomto případě posílat přebytek toku ve v zpátky do zdroje. Bez ohledu na velikost $d(v)$ budeme protlačovat tok z kopce dolů. Tedy z vrcholu v do vrcholů w s $d(w) < d(v)$, protože se tím podle odhadů dostane tok z v blíže k místu určení.

Poznámka: (o výpočtu platného označkování) Důkaz lemmatu nám naznačuje, jak si spočítat platné označkování, pokud bychom ho neznali. Nejprve se podívejme na speciální případ, kdy ze všech vrcholů kromě s existuje v síti rezerv G_f orientovaná cesta do t. Za výšky $d(v)$ zvolíme délku nejkratší cesty z v do t, jinými slovy $d(v) := d_f(v,t)$. Potom označkování vrcholů $d(v)$ splňuje vlastnosti platného

[9] Nasycené hrany zmizí ze sítě rezerv G_f. Proto se na ně nevstahuje omezení klesání.

[10] Z důkazu je vidět, odkud se vzala druhá podmínka v definici platného označkování. Potřebujeme, aby tohle lemma platilo.

označkování až na podmínku $d(s) = n$. Hodnotu $d(s)$ si můžeme zvolit jak chceme, protože v síti G_f nevede z s žádná hrana do ostatních vrcholů. Jinak by v G_f existovala i cesta z s do t. (Všechny hrany vedoucí z s jsou nasycené a tudíž nejsou v síti G_f).

V obecném případě zvolíme $d(v) := \min\{d_f(v,t),\ n + d_f(v,s)\}$. Důkaz, že takto dostaneme platné označkování, necháme jako cvičení. Poznamenejme jenom, že množina vrcholů, ze kterých v G_f vede orientovaná cesta do t, určuje nasycený řez.

Výpočtu platného označkování využijeme později v heuristice na straně 165.

Push-Relabel algoritmus

Inicializace: Začneme s počátečním pratokem f takovým, že $f(e) = c(e)$ pro hrany $e \in E$ vedoucí ze zdroje s a $f(e) = 0$ pro ostatní hrany. Položme $d(s) = n$ a $d(v) = 0$ pro všechny ostatní vrcholy v. Označkování d je platným označkováním pro pratok f, protože všechny hrany G_f mají oba konce ve výšce nula.

Hlavním úkolem ve zbytku algoritmu je likvidovat aktivní vrcholy, protože až v grafu nebude existovat aktivní vrchol, tak se pratok stane tokem. Při zpracování vrcholů nám pomohou následující operace.

Operace protlač: Operaci protlačení toku po hraně $vw \in E(G_f)$ nazveme *protlač*(vw) (anglicky se nazývá *push*). Jak už jsme řekli, přebytek toku budeme protlačovat pouze po hranách vedoucích z kopce dolů. Tedy při protlačení toku po hraně $vw \in E(G_f)$ je $d(w) < d(v)$. Protože hrany sítě rezerv nemohou klesat nejvíce o jedna, musí být $d(v) = d(w) + 1$. Abychom měli co protlačovat, tak musí být ve v kladný přebytek toku. To znamená, že v musí být aktivní vrchol. Protlačování proto může probíhat pouze po hranách, které vedou z aktivních vrcholů a klesají právě o jedna. Takové hrany nazveme *přípustné.*

Hrana uv se při protlačení buď nasytí a zmizí z G_f, nebo se nenasytí a zůstane v G_f. Do sítě rezerv přibude zpětná hrana wv, která vede do kopce. Jiné změny v G_f nejsou a proto po protlačení toky po vw zůstane označkování d platné.

Operace zvýšení: Předpokládejme, že v je aktivní, ale z v už v G_f nevede žádná přípustná hrana. Potom můžeme zvýšit $d(v)$ na $\min\{d(w) + 1 \mid vw \in E(G_f)\}$, aniž bychom porušili platnost označkování. Této operaci budeme říkat *zvýšení* vrcholu v (anglicky se označuje *relabel* nebo *lift*). Jinými slovy, aktivní vrchol v zvedáme o jedničku tak dlouho, dokud některá hrana z vrcholu vycházející nepovede z kopce.

Je spousta výsledků o tom, v jakém pořadí se mají operace provádět. My si předvedeme obecnější verzi algoritmu. Jakmile vybereme aktivní vrchol v, tak budeme provádět operaci protlač po přípustných hranách G_f tak dlouho, dokud se vrchol v nestane neaktivním a nebo dokud ho nezvýšíme. Tuto posloupnost operací označíme jako zpracování vrcholu.

```
Zpracuj(v):
    while v je aktivní a existuje přípustná hrana vw ∈ E(G_f) do
        Protlač co největší část přebytku ve v po hraně vw
    if v je aktivní then
        Zvyš vrchol v
```

Samotný algoritmus potom můžeme vyjádřit následovně.

```
Push-Relabel:
    Inicializuj f a d
    while f není tok do
        Vyber aktivní vrchol v
        Zpracuj v
```

Je několik různých pravidel pro výběr aktivního vrcholu.

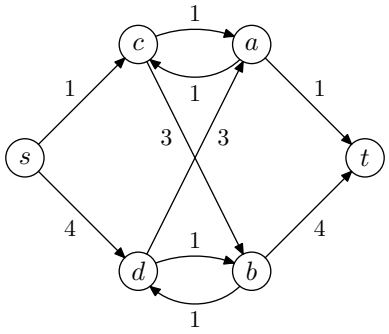
- Vybereme vrchol v s maximálním označením $d(v)$. Algoritmus s tímto pravidlem se označuje jako *maximum distance push-relabel.*[11] Toto pravidlo se používá nejčastěji, protože garantuje nejlepší časovou složitost algoritmu.

- Aktivní vrcholy vkládáme do fronty. Aktivní vrcholy zpracováváme v pořadí, jak jsou ve frontě. (Pokud vrchol zůstal po zpracování aktivní, tak se ocitne na konci fronty). Algoritmus s tímto pravidlem se označuje jako *FIFO push-relabel.*

Algoritmus si v průběhu udržuje platný pratok a také platné označkování. Při popisu operací jsme ověřili, že se provedením operace platnost označkování nezmění. Proto z důsledku 4 dostáváme, že pokud nebude existovat aktivní vrchol, tak se algoritmus zastaví a skončí s maximálním tokem.

Příklad

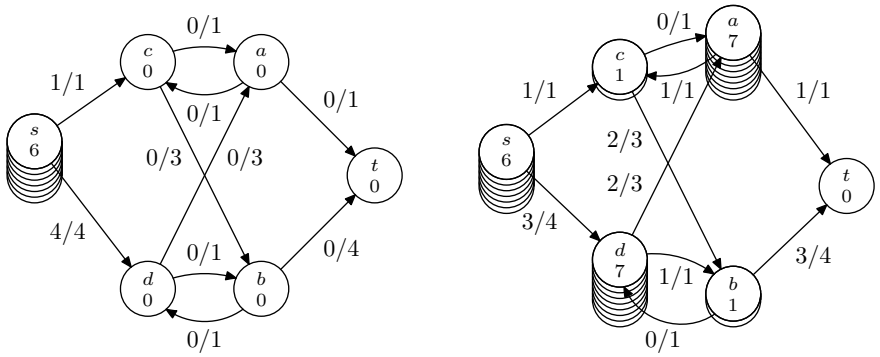
Na následujícím obrázku je graf, ve kterém chceme najít maximální tok pomocí algoritmu push-relabel. Použijeme pravidlo, které si vždy vybere aktivní vrchol s největším $d(v)$. Pokud mají dva vrcholy stejnou hodnotu $d(v)$, tak vybereme ten abecedně menší. V každém vrcholu v probíráme přípustné hrany vw v abecedním pořadí podle w.



Nejprve provedeme inicializaci a dostaneme ohodnocený graf na následujícím obrázku vlevo. Pozor, hodnota u vrcholu v není rezerva, ale výška $d(v)$. Po inicializaci jsou vrcholy c, d aktivní. Vybereme si c a zvýšíme $d(c)$ na 1. V dalším kroku protlačíme po hraně ca tok velikosti 1. Po tomto kroku už jsou všechny kroky algoritmu jasně určeny. Doporučujeme čtenáři, aby si odkrokoval zbytek algoritmu. Aktivní vrcholy zvolené algoritmem jsou c, a, d, d, a, a, d, a, d, a, d, a, d, c, b a postupně algoritmus protlačuje tok po hranách ca, at, db, da, ac, ad, da, ad, da, ad, ds, cb, bt (aktivní vrchol většinou zvedáme o 1, párkrát o 2 a v předposledním případě vůbec). Pro lepší pochopení si v každém kroku načtněte graf G_f, ať vidíte, které hrany jsou nasycené a vypadly. Algoritmus skončí s tokem a označkováním na následujícím obrázku vpravo.[12]

[11] Maximum distance proto, že hodnota $d(v)$ se označuje jako distance label.

[12] Všimněme si zdlouhavého protlačování toku mezi vrcholy a,d. Nejprve protlačíme přebytek toku z a do d, pak ten samý tok zpátky z d do a a tak několikrát dokola. Vypadá to jako „ping-pong“, který hrajeme tak dlouho, dokud výška jedno z vrcholů nepřekročí 6. V průběhu „ping-pongu“ v G_f neexistuje cesta do spotřebiče t. Proto kdybychom rovnou zvedli výšky vrcholů a, d na 6, tak bychom neporušili platnost označkování a ušetřili si „ping-pong“. Jak této myšlenky využít se dozvíte v heuristice na straně 165.



Výsledné označkování neobsahuje žádný vrchol s $d(v) = 5$ a proto množina $R = \{s, d, a\}$ určuje minimální řez (viz lemma 24).

Analýza Push-Relabel algoritmu

Naším cílem je dokázat následující věty.

Věta 9 *Algoritmus push-relabel provede $\mathcal{O}(n^2)$ zvýšení vrcholů a $\mathcal{O}(mn^2)$ protlačení po hraně.*

Věta 10 *Algoritmus maximum distance push-relabel provede $\mathcal{O}(n^2)$ zvýšení vrcholů a $\mathcal{O}(n^3)$ protlačení po hraně.*

První větu dokážeme řadou lemmat. Lemmata platí pro obecný push-relabel algoritmus. Akorát lemma 30 platí pouze pro maximum distance pust-relabel algoritmus. Jeho přidáním k předchozím lemmatům dokážeme větu 10.

Lemma 26 *Je-li f je pratok a w je aktivní vrchol, potom v G_f existuje orientovaná cesta z w do zdroje s.*

Důkaz: Sporem, nechť z w nevede orientovaná cesta do s. Označme jako R množinu vrcholů, ze kterých v G_f vede orientovaná cesta do zdroje s. Potom v G_f nevede žádná hrana z $\overline{R}$ do R a proto $f(\delta(R)) = 0$. Sečtěme nerovnosti $f(v) \geq 0$ pro všechny $v \in \overline{R}$ a dostaneme $X := \sum_{v \in \overline{R}} f(v) \geq 0$. Na druhou stranu, když sečteme příspěvky přebytků po hranách, tak dostaneme $X = f(\delta(R)) - f(\delta(\overline{R}))$ (každá hrana uv s oběma konci v $\overline{R}$ přispěje do $f(v)$ kladně a do $f(u)$ záporně a proto je její příspěvek do X roven nule). Protože $f(\delta(R)) = 0$, tak z nerovnosti $f(\delta(R)) - f(\delta(\overline{R})) \geq 0$ dostáváme $f(\delta(\overline{R})) = 0$. Součet nerovností platí s rovností a proto i každá nerovnost platí s rovností. To je spor s tím, že pro $w \in \overline{R}$ je $f(w) > 0$. ∎

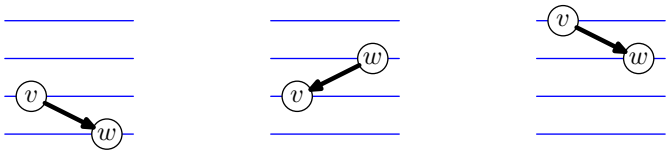
Lemma 27 *V každém kroku algoritmu pro každý vrchol $v \in V$ je $d(v) \leq 2n - 1$. Každý vrchol se zvýší nejvýše $2n - 1$ krát a tedy celkem proběhne nejvýše $\mathcal{O}(n^2)$ zvýšení.*

Důkaz: Každé zvýšení proběhne alespoň o jedna. Zvyšovány jsou pouze aktivní vrcholy a z těch vždy vede cesta do zdroje (lemma 26). Tedy $d_f(v, s) \leq n - 1$. Z lemmatu 25 víme, že $d_f(v, s) \geq d(v) - n$. Kombinací obou nerovností dostaneme $d(v) \leq 2n - 1$. ∎

Operace protlačení rozdělíme na dva typy podle toho, jestli se při ní hrana nasytila a nebo ne. Protlačení po hraně vw je *nasycující*, pokud byl přebytek ve v větší než rezerva hrany vw. Po nasycujícím protlačení zmizí hrana vw z G_f. V opačném případě je protlačení po hraně vw *nenasycující* a v tomto případě přestal být vrchol v aktivní.

Lemma 28 *Počet nasycujících protlačení během algoritmu je nejvýše* $2mn$.

Důkaz: Podívejme se na pevný pár (v, w) takový, že $vw \in E$ nebo $wv \in E$. Po nasycujícím protlačení po hraně vw hrana vw zmizí z G_f. Proto mezi dvěma nasycujícími protlačeními po hraně vw musí proběhnout protlačení po opačné hraně wv, aby se hrana vw opět objevila v G_f.



Protlačování probíhá pouze po přípustných hranách, vedoucích z kopce o jedna dolů. Hodnota $d(v)$ nikdy neklesá a proto musí být mezi dvěma nasycujícími protlačeními po vw alespoň jedna operace zvýšení vrcholu v. Zvýšení $d(v)$ bude alespoň o 2 a proto podle lemmatu 27 může proběhnout nejvýše $n-1$ krát. Celkem po hraně vw může proběhnout nejvýše n nasycujících protlačení.

Podobně po opačné hraně wv může proběhnout také nejvýše n nasycujících protlačení. Celkem tedy pro všechny hrany v původní síti G proběhne nejvýše $2mn$ nasycujících protlačení. ∎

Lemma 29 *Počet nenasycujících protlačení během algoritmu je nejvýše* $\mathcal{O}(mn^2)$.

Důkaz: Nechť A je množina aktivních vrcholů vzhledem k pratoku f a $D = \sum_{v \in A} d(v)$. Na začátku algoritmu je $D = 0$ a nikdy není záporné.

- Každé zvýšení vrcholu zvětšuje D.
- Nasycující protlačení po hraně vw může zvětšit D až o $2n - 1$, protože v může zůstat aktivní a vrchol w se může stát aktivním. Podle lemma 27 je $d(w) \leq 2n - 1$.
- Nenasycující protlačení po vw zmenší D. Vrchol v přestane být aktivní. Proto se D zmenší buď o $d(v)$ nebo o $d(v) - d(w) = 1$, pokud se zároveň w stane aktivním.

Všechny zvětšení D během algoritmu jsou kvůli zvýšení vrcholů a nebo kvůli nasycujícím protlačením. Podle lemma 27 a lemma 28 během celého algoritmu vzroste hodnota D o nejvýše $(n-2)(2n-1)+2mn(2n-1) = \mathcal{O}(mn^2)$. Každý nenasycující protlačení sníží tuto hodnotu alespoň o jedna a proto proběhne nejvýše $\mathcal{O}(mn^2)$ nenasycujících protlačení. ∎

Lemma 30 *Počet nenasycujících protlačení během maximum distance push-relabel algoritmu je nejvýše* $\mathcal{O}(n^3)$.

Důkaz: Každé nenasycující protlačení po vw deaktivuje vrchol v. Protože vždy zpracováváme vrchol v s největším $d(v)$, tak je $d(w) \leq d(v)$ pro všechny aktivní vrcholy w. Před tím, než se v stane znovu aktivním, musí proběhnout zvýšení nějakého souseda v (a protlačení toku z tohoto souseda do v).

Z toho dostáváme, že když proběhne n nenasycujících protlačení a žádné zvýšení vrcholu, tak žádný vrchol není aktivní a algoritmus skončí. Proto je počet nenasycujících protlačení nejvýše n krát větší než počet zvýšení a to je nejvýše $\mathcal{O}(n^3)$. ∎

Poznamenejme, že jsou i další pravidla pro výběr aktivních vrcholů a dávají také odhad $\mathcal{O}(n^3)$ na celkový počet protlačení (například FIFO push-relabel). My jsme si vybrali pravidlo maximum distance z toho důvodu, že pro něj Tunçel v roce 1994 dokázal lepší analýzu a ukázal odhad $\mathcal{O}(n^2\sqrt{m})$ na počet protlačení (viz lemma 31).

Tunçelův odhad na počet nenasycujících protlačení*

Lemma 31 *Počet nenasycujících protlačení během maximum distance push-relabel algoritmu je nejvýše* $\mathcal{O}(n^2\sqrt{m})$.

Důkaz: Připomeňme, že $d(v)$ nazýváme výškou vrcholu v. V průběhu algoritmu H označuje maximální výšku aktivního vrcholu.

Výpočet algoritmu rozdělíme do fází mezi změnami hodnoty H. Změna fáze nastane když dojde ke zvýšení vrcholu, který ležel ve výšce H, a nebo když se přebytky všech vrcholů ležících ve výšce H sníží na nulu.

Nyní ukážeme, že celkem proběhne nejvýše $4n^2$ fází. $H \geq 0$, roste pouze při zvýšení vrcholu a to o jedna. Celkový počet zvýšení H je nejvýše počet zvýšení vrcholů a to je nejvýše $2n^2$ (lemma 27). Počet poklesů H je nejvýše tolik, kolik celkem proběhne zvýšení H. Celkový počet změn H (zvýšení nebo poklesů) a tedy i počet fází je nejvýše $4n^2$.

Počet nenasycených protlačení spočítáme pomocí potenciálu. Pevně si zvolíme parametr $K \in \mathbb{N}$. Později ukážeme, že optimální volba je $K := \sqrt{m}$. Zvolme potenciál

$$\Psi := \sum_{f(v)>0} \frac{\vartheta(v)}{K},$$

kde $\vartheta(v) := |\{w \in V \mid d(w) \leq d(v)\}|$ je počet vrcholů ve výškách nejvýše $d(v)$. Fázi nazveme *levnou*, pokud během ní proběhne nejvýše K nenasycených protlačení, a *drahou* jinak.

Levných fází je nejvýše tolik, kolik je všech fází a to je nejvýše $4n^2$. Celkem během levných fází proběhne nejvýše $4Kn^2$ nenasycených protlačení.

Teď odhadneme počet nenasycujících protlačení během drahých fází. Podívejme se, co se děje s potenciálem Ψ při následujících operacích.

- Zvýšení vrcholu. Při zvýšení vrcholu v se $\vartheta(v)$ zvýší nejvýše o n. $\vartheta(w)$ ostatních vrcholů w může jen klesnout (vždy zvyšujeme vrchol v s maximálním $d(v)$). Proto se Ψ zvýší nejvýše o n/K.

- Nasycující protlačení po hraně uv. Protože neměníme výšky vrcholů, tak můžeme Ψ ovlivnit pouze tím, že přibude nebo ubude aktivní vrchol. Můžeme ubrat sčítanec $\vartheta(u)/K$ a přidat $\vartheta(v)/K \leq n/K$. Nasycující protlačení tedy zvýší Ψ nejvýše o n/K. Podle lemma 29 je počet nasycujících protlačení během celého algoritmu nejvýše $2mn$.

- Nenasycující protlačení po hraně uv. Po protlačení bude $f(u) = 0$ a tedy z Ψ ubyde $\vartheta(u)/K$. Dále může přibýt $\vartheta(v)/K$. Celkový úbytek Ψ bude nejvýše $(\vartheta(u) - \vartheta(v))/K$. Protože $H = d(u) = d(v) + 1$ (uv je přípustná hrana), tak $(\vartheta(u) - \vartheta(v))$ odpovídá počtu vrcholů ve výšce H. Počet vrcholů ve výšce H se během fáze nemění. Nemůže klesnout, protože pouze zvyšujeme vrcholy a povýšením některého vrcholu na výšku $H + 1$ ukončíme fázi. V průběhu fáze dojde nejvýše k tolika nenasycujícím protlačení, kolik je aktivních vrcholů ve výšce H. V drahé fázi proběhne alespoň K nenasycujících protlačení. Matematicky vyjádřeno $K \leq$ #nenasycujících protlačení $\leq$ #vrcholů ve výšce $H = \vartheta(u) - \vartheta(v)$. Proto je $(\vartheta(u) - \vartheta(v))/K \geq 1$. Tedy nenasycující protlačení sníží Ψ alespoň o 1.

Celkový součet přírůstků Ψ je nejvýše $(2n^2+2nm)n/K$. Po inicializaci algoritmu bylo Ψ nejvýše n^2/K. Protože Ψ je stále kladné a každé nenasycující protlačení sníží Ψ alespoň o 1, je počet nenasycujících protlačení v drahých fázích nejvýše

$$n^2/K + (2n^2 + 2nm)n/K \leq 5n^2m/K.$$

Při odhadu jsme použili nerovnost $n \leq m$. Dohromady je počet nenasycujících protlačení v levných i drahých fázích nejvýše

$$4n^2K + 5n^2m/K \leq 5n^2(K + m/K) \leq 10n^2\sqrt{m}$$

pro volbu $K = \sqrt{m}$. ∎

Implementace algoritmu Push-Relabel

Zatím jsme ukázali odhady na počet provedení operací zvýšení vrcholu a protlačení toku po hraně. Abychom mohli něco tvrdit o časové složitosti, tak ještě musíme upřesnit, jak proběhne nalezení přípustné hrany a jak poznáme, že je čas zvýšit vrchol. O maximum distance push-relabel algoritmu budeme muset ještě ukázat, jak najít aktivní vrchol v s maximálním $d(v)$.

Pozorování 15 *Nechť $v \in V$ je aktivní a hrana vw není přípustná. Před tím, než se hrana vw stane přípustnou, bude muset proběhnout zvýšení vrcholu v.*

Důkaz: Pokud hrana vw není přípustná, tak buď $d(v) \leq d(w)$ a nebo $r(vw) = 0$. Druhý případ se může změnit pouze protlačením toku po opačné hraně wv, ale potom bude $d(w) = d(v) + 1$. Proto bude v obou případech platit $d(v) \leq d(w)$ a jenom zvýšení vrcholu v to může změnit. ∎

Pro každý vrchol si pamatujeme seznam sousedů $Sousedi(v)$. Zpracování vrcholu v proběhne tak, že postupně projdeme $w \in Sousedi(v)$ a provedeme protlačení po přípustných hranách vw. Průchod seznamu sousedů skončí buď tak, že se v stane neaktivním, nebo tím, že dojdeme na konec seznamu $Sousedi(v)$.

V momentě, kdy dorazíme na konec seznamu, tak už z v nevede žádná přípustná hrana a proto zvýšíme v. Ke zvýšení vrcholu potřebujeme znát minimum z $d(w)$ pro všechny $w \in Sousedi(v)$. Toto minimum si můžeme počítat už během průchodu seznamu sousedů. Zvýšení vrcholu v dokonce proběhne právě tehdy, když se dostaneme na konec seznamu $Sousedi(v)$.

Když bude v znovu vybrán ke zpracování, tak podle pozorování nemohly od posledního zvýšení v přibýt nové přípustné hrany. Proto při zpracování vrcholu v nemusíme procházet seznam $Sousedi(v)$ od začátku, ale můžeme začít tam, kde jsme naposledy skončili (pro každý vrchol musíme pamatovat aktuální pozici v seznamu sousedů).

Protože každý vrchol můžeme zvýšit nejvýše $2n - 1$ krát, tak projdeme seznam sousedů každého vrcholu také nejvýše $2n - 1$ krát. Z toho důvodu je celkový čas strávený hledáním přípustných hran roven $\mathcal{O}(\sum_{v \in V} n \cdot |Sousedi(v)|) = \mathcal{O}(nm)$. Celkový čas strávený nad zvyšováním vrcholů je stejný.

Celkový čas strávený nad operacemi protlačení je $\mathcal{O}(N)$, kde N je počet všech protlačení. Čas nalezení dalšího aktivního vrcholu je konstantní, protože nám nezáleží na pořadí aktivních vrcholů a můžeme si je dávat do fronty. Aktivní vrchol hledáme nejvýše tolikrát, kolik je všech operací protlačení a zvýšení vrcholu. Celkem tedy hledání aktivních vrcholů zabere čas $\mathcal{O}(N + n^2)$ a to je podle věty 9 nejvýše $\mathcal{O}(n^2m)$.

Právě jsme si ukázali, že push-relabel algoritmus může být implementován tak, aby běžel v čase $\mathcal{O}(n^2m)$.

Podívejme se na případ maximum distance push-relabel algoritmu. Není jasné, jak implementovat výběr aktivního vrcholu s maximálním $d(v)$ tak, aby celkem

běžel v čase $\mathcal{O}(N)$. To v průměru odpovídá konstantnímu času na jednu operaci protlačení. Jednoduché řešení projde všechny vrcholy, ale to trvá čas $\mathcal{O}(n)$ na jedno nalezení aktivního vrcholu.

Provedeme to jinak. Všechny aktivními vrcholy s $d(v) = k$ si uložíme do fronty D_k. Frontu realizujeme jako obousměrný spojový seznam. Ten umožní provádět vkládání a mazání v konstantním čase. Navíc si pro každý vrchol budeme pamatovat ukazatel, který nám umožní přístup k položce ve správné frontě v konstantním čase (pro aktivní vrcholy).

Po provedení zvýšení vrcholu jednoduše přesuneme vrchol do jiné fronty. Pokud protlačení toku po hraně vw aktivuje w, tak ho vložíme do správné fronty. Pokud protlačení deaktivuje vrchol v, tak ho vyřadíme z fronty. Tyto operace proběhnou v konstantním čase.

Proč je tak jednoduché najít aktivní vrchol s maximálním $d(v)$? Po zvýšení v zůstane vrchol v aktivní a s maximálním $d(v)$. Dejme tomu, že $d(v) = k$. Pokud protlačení po vw deaktivuje v, tak se nejprve podíváme do fronty D_k. Když je prázdná, tak se podíváme do fronty D_{k-1}. Skoro vždy v ní najdeme aktivní vrchol, protože poslední protlačení po vw splňovalo $d(w) = k-1$. Vrchol w musí být aktivní, jinak je $w = s$ nebo $w = t$.

Případ $w = t$ je triviální, protože potom není žádný vrchol aktivní a algoritmus skončí.

Jediný případ, kdy neuspějeme s hledáním aktivního vrcholu, nastane, když zpracujeme vrchol z D_{n+1} a obě fronty D_{n+1}, D_n jsou prázdné. Než tato situace nastane znova, tak se bude muset nějaký vrchol dostat do fronty D_{n+1}. Neboli jeho $d(v)$ bude muset překročit n. To může nastat pro každý vrchol nejvýše jednou a proto tento špatný případ nastane nejvýše n krát. V tomto špatném případě budeme muset prohledat všechny fronty D_k s $k < n$. Špatné případy celkem přispějí do hledání aktivních vrcholů časem $\mathcal{O}(n^2)$. Celkem hledání aktivních vrcholů zabere čas $\mathcal{O}(N + n^2)$, kde N je počet všech protlačení.

Ukázali jsme si, že maximum distance push-relabel algoritmus může být implementován tak, aby běžel v čase $\mathcal{O}(n^3)$. Pokud bychom použili odhad $\mathcal{O}(n^2\sqrt{m})$ na počet protlačení, tak dokonce v čase $\mathcal{O}(n^2\sqrt{m})$.

Heuristika zrychlující Push-Relabel algoritmus

Ještě zmíníme heuristiku, které nemá vliv na odhad časové složitosti, ale v praxi podstatně zrychlí výpočet. Pravidelně, řekněme po $n/2$ zpracováních vrcholů, přepočítáme platné označkování. Ukazovali jsme si, že platné označkování $d(v)$ je dolním odhadem na $d_f(v,t)$ a že $d(v) - n$ je dolním odhadem $d_f(v,s)$. Zvolme proto nové označkování jako $d(v) := \min\{d_f(v,t),\ n + d_f(v,s)\}$. Ukažte, že toto označkování je platné a v jistém smyslu nejlepší možné. Také si rozmyslete, jak rychle ho můžeme spočítat.

Poznámka: Předvýpočet platného označkování se vyplatí už při inicializaci push-relabel algoritmu.

12.4 Srovnání algoritmů pro hledání maximálního toku

Přehled časových složitostí variant algoritmu vylepšující cesty.

Dinic	$\mathcal{O}(n^2m)$
3 indové	$\mathcal{O}(n^3)$
nejlepší známé	$\mathcal{O}(mn\log(n^2/m))$
jednotkové kapacity	$\mathcal{O}(\sqrt{m}\cdot m)$
jednotkové kapacity, prostý graf	$\mathcal{O}(n^{2/3}\cdot m)$
jednotkové kapacity, vstupní nebo výstupní stupeň ≤ 1	$\mathcal{O}(\sqrt{n}\cdot m)$
celočíselné kapacity	$\mathcal{O}(\lvert f\rvert \cdot n + nm)$
celočíselné kapacity $\leq U$	$\mathcal{O}(Un^2 + nm)$
scaling (celočíselné kapacity $\leq U$)	$\mathcal{O}(nm\log U)$

Prostým grafem myslíme graf bez násobných hran.[13] Ostatní algoritmy fungují i na multigrafech.

Golgberg a Tarjan [15] přišli s algoritmem na nalezní blokujícího toku v acyklickém orientovaném grafu v čase $\mathcal{O}(m\log(n^2/m))$. Důsledkem toho dostáváme algoritmus pro hledání maximálního toku v obecných orientovaných grafech v čase $\mathcal{O}(nm\log(n^2/m))$.

Většinu algoritmů pro speciální případy naleznete ve cvičeních. Podle navrhnutých základních myšlenek nebo kostry algoritmu si sami zkusíte domyslet detaily, sestavit algoritmus a dokázat, že funguje. Upočítání časových složitostí pro speciální případy naleznete v [22].

Přehled časových složitostí variant push-relabel algoritmu.

maximum distance push-relabel	$\mathcal{O}(n^2\sqrt{m})$
jednotkové kapacity	$\mathcal{O}(nm)$
celočíselné kapacity $\leq k$	$\mathcal{O}(knm)$
scaling přebytků	$\mathcal{O}(nm + n^2\log U)$

S některými variantami se opět seznámíte ve cvičeních.

Scaling přebytků je varianta push-relabel algoritmu, která protlačuje tok z vrcholů s dostatečně vysokým přebytkem do vrcholů s dostatečně nízkým přebytkem, přičemž nikdy nedovolíme, aby byl přebytek příliš velký. Myšlenka scalingu je podobná scalingu u algoritmů vylepšující cesty. Celkem hezky je to popsané v [2].

Poznamenejme, že push-relabel algoritmus se chová dobře i na speciálních grafech. Například na bipartitním grafu s n_1 a n_2 vrcholy doplněném o zdroj a spotřebič běží FIFO push-relabel algoritmus v čase $\mathcal{O}(n_1m + n_1^3)$ (podívejte se do [3]).

Nejlepší známé řešení pro toky v sítích používá nový přístup k problému. Je to algoritmus Golberg-Rao [13] s časovou složitostí $\mathcal{O}(m\sqrt{m}\log(n^2/m)\log U)$.[14]

Praktické chování.

Poznamenejme, že algoritmus push-relabel se v praxi chová velice dobře a je podstatně rychlejší než algoritmy vylepšující cesty. (Můžeme si to vysvětlit tím, že pomocí platného označkování snadněji „udržujeme“ vrstevnatou síť a nemusíme ji pokaždé přepočítávat.)

Push-relabel algoritmus můžeme ještě více zrychlit pomocí heuristiky popsané v podsekci 12.3.

[13] Tedy ne multigraf. Ikdyž pokud je násobnost hrany omezena pevnou konstantou, tak to nevadí.

[14] Dokonce se dá ukázat, že člen $\sqrt{m}$ v časové složitosti může být nahrazen členem $\min\{\sqrt{m}, n^{2/3}\}$.

12.5 Aplikace toků v sítích

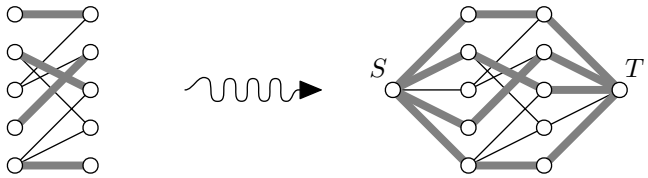
Pomocí toků v sítích se dá vyřešit nepřeberné množství problémů. Představíme si několik vzorových aplikací a ostaní si necháme jako cvičení.

12.5.1 Maximální párování v bipartitním grafu

Definice: Množina hran $M \subseteq E$ v grafu $G = (V, E)$ je *párování* pokud žádný vrchol $v \in V$ neleží ve dvou hranách M. Jinými slovy, párování je množina nezávislých hran. Párování M je *maximální*, pokud pro každé párování M' platí $|M| \geq |M'|$.

Věta 11 *Maximální párování v bipartitním grafu $G = (V_1 \cup V_2, E)$ můžeme spočítat pomocí toku v síti v čase $\mathcal{O}(n^2 m)$.*

Důkaz: Zkonstruujeme orientovaný graf $G' = (V_1 \cup V_2 \cup \{S, T\}, E')$ následujícím způsobem. K původnímu grafu G přidáme super-zdroj S a super-spotřebič T. Super-zdroj spojíme se všemi vrcholy $v_1 \in V_1$ hranou Sv_1. Každý vrchol $v_2 \in V_2$ spojíme se super-spotřebičem hranou v_2T. Původní hrany grafu orientujeme z množiny V_1 do V_2. Kapacity všech hran nastavíme na 1.



Tvrdíme, že v grafu G' existuje maximální tok velikosti k právě tehdy, když v bipartitním grafu G existuje maximální párování velikosti k.

Nejprve ukažme první implikaci. Protože jsou kapacity hran v grafu G' celočíselné, tak v G' existuje celočíselný maximální tok (důsledek 3). Tok po každé hraně je buď 0 nebo 1. Protože do každého vrcholu $v \in V_1$ může přitékat nejvýše tok velikosti 1 a z každého vrcholu $v \in V_2$ může odtékat nejvýše tok velikosti 1, tak je maximální celočíselný tok v G' sjednocením k vrcholově disjunktních[15] (S, T)-cest. Ty obsahují disjunktní hrany tvořící párování.

Na druhou stranu můžeme hrany tvořící párování v G rozšířit do vrcholově disjunktních (S, T)-cest grafu G'.

Pro nalezení maximálního párování v bipartiním grafu G tedy stačí najít maximální tok v pomocném grafu G'. ∎

Pomocný graf G' má speciální tvar a dá se ukázat, že v něm Dinicův algoritmus trvá jen čas $\mathcal{O}(n^{2/3}m)$ (viz cvičení).[16] Takže maximální párování ve skutečnosti najdeme rychleji.

12.5.2 Cirkulace s požadavky

Malinko zobecníme problém toků v sítích. Nechť $G = (V, E)$ je orientovaný graf. Každá hrana $e \in E$ má kapacitu $c(e)$, kde $c : E \to \mathbb{R}_+$. Každý vrchol $v \in V$ má požadavek $d(v) \in \mathbb{R}$. Chceme, aby ve vrcholu v zůstal přebytek toku $d(v)$. Vrchol v s $d(v) > 0$ se chová jako spotřebič, vrchol v s $d(v) < 0$ jako zdroj a vrchol v s $d(v) = 0$ jako normální vrchol.

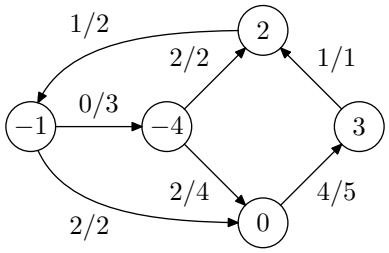
Definice: *Cirkulace s požadavky* $\{d(v)\}_{v \in V}$ je funkce $f : E \to \mathbb{R}_+$, která splňuje

[15] Dvě cesty jsou vrcholově disjunktní, pokud nemají společný vrchol – kromě počátku a konce, kde se to toleruje.

[16] Poznamenejme, že hledání maximálního párování v bipartitním grafu pomocí Dinicova algoritmu je ekvivalentní Hopcroft-Karpovu algoritmu využívajícího volné střídavé cesty.

$$\begin{array}{lll} i) & 0 \le f(e) \le c(e) & \text{pro každou hranu } e \in E \\ ii) & f(v) = d(v) & \text{pro každý vrchol } v \in V \end{array}$$

Problém: Existuje v síti G cirkulace splňující požadavky?



Na obrázku je příklad cirkulace f v grafu G. Hodnoty ve vrcholech označují požadavky $d(v)$ a hrany jsou označeny „$f(e)/c(e)$“.

Pozorování 16 *Pokud v síti G existuje cirkulace splňující požadavky $\{d(v)\}_{v\in V}$, tak je $\sum_{v\in V} d(v) = 0$.*

Důkaz: Použijeme počítání dvěma způsoby. Na jednu stranu je $X = \sum_{v\in V} d(v)$. To jsme sečetli příspěvky přebytků po vrcholech. Na druhou stranu můžeme příspěvky do X počítat po hranách.[17] Tok po každé hraně přispívá do X dvakrát (za každý konec hrany). Jednou s kladným a podruhé se záporným znaménkem. Proto je $X = 0$. ■

Pozorování zároveň říká, že pokud existuje cirkulace f, tak je

$$|f| = \sum_{v\in V, d(v)>0} d(v) = \sum_{v\in V, d(v)<0} -d(v).$$

Jak tedy zjistit, zda v G existuje cirkulace splňující požadavky? Nejprve zkontrolujeme, jestli $\sum_{v\in V,d(v)>0} d(v) = -\sum_{v\in V,d(v)<0} d(v)$. Pokud tato nutná podmínka platí, tak zkonstruujeme pomocný graf $G' = (V \cup \{S,T\}, E')$.

- Vytvoříme super-zdroj S a spojíme ho se všemi vrcholy v, které mají $d(v) < 0$ (chovají se jako „zdroje“). Kapacitu hrany Sv nastavíme na $-d(v)$.
- Vytvoříme super-spotřebič T a spojíme ho se všemi vrcholy v, které mají $d(v) > 0$ (chovají se jako „spotřebiče“). Kapacitu hrany vT nastavíme na $d(v)$.

Ve výsledné síti (která už má jen jeden zdroj a jeden spotřebič) nalezneme maximální tok. Pokud je jeho velikost rovna $\sum_{v\in V,d(v)>0} d(v)$, tak je restrikce[18] nalezeného toku na původní graf platnou cirkulací. Pokud je velikost maximálního toku menší, tak platná cirkulace neexistuje (zkuste si to dokázat).

Jako důsledek poznatků o tocích v sítích dostáváme, že když jsou všechny kapacity hran a pořadavky toku ve vrcholech celočíselné, tak existuje platná cirkulace, která je celočíselná.

[17] Připomeňme, že $d(v) = f(v) = \sum_{e=xv} f(e) - \sum_{e=vx} f(e)$.

[18] Restrikce je omezení se na určitou část. V tomto případě necháme ve funkci f pouze hrany původního grafu.

12.5.3 Cirkulace s limity na průtok hranou

V předchozí podsekci jsme si vysvětlili, co je to cirkulace v grafu $G = (V, E)$ s požadavky $\{d(v)\}_{v \in V}$. Každá hrana sítě $e \in E$ má svojí kapacitu $c(e)$, která je horním limitem na velikost toku po hraně. Tentokrát chceme velikost toku po hraně omezit i ze spoda. Hodnota $\ell(e)$ určuje minimální tok po hraně e.

Chceme nalézt tok f, který splňuje požadavky $d(v)$ ve vrcholech a navíc $\ell(e) \leq f(e) \leq c(e)$ pro každou hranu $e \in E$. Jak takový tok najít?

Zkusme následující, na první pohled naivní přístup. Na začátku po každé hraně e pošleme přesně $f_0(e) := \ell(e)$. Funkce f_0 splňuje omezení na průtok po hranách. Jediné, co brání funkci f_0 v tom, aby byla cirkulací s požadavky, jsou přebytky toku ve vrcholech. Přebytek toku ve vrcholu v je $f_0(v)$ a my potřebujeme, aby byl $d(v)$. Pokud $f_0(v) = d(v)$, tak je požadavek pro vrchol v splněn. V opačném případě musíme „tok“ f_0 upravit.

Provedeme to následovně. Nechť G' je graf G, ve kterém zvolíme nové požadavky $d'(v) := d(v) - f_0(v)$ pro každý vrchol $v \in V$ a nastavíme kapacity hran na $c'(e) := c(e) - l(e)$. Jinými slovy, od kapacit hran a požadavků ve vrcholech odečteme nutný minimální tok po hranách a dostaneme síť G'.

Všimněme si, že v G' „zmizely“ požadavky na minimální tok po hranách (požadavek $f(e) \geq l(e)$ se změnil na $f(e) \geq 0$). Pokud v G' najdeme cirkulaci f' splňující nové požadavky, tak tok $f := f' + f_0$ bude platnou cirkulací v G splňující požadavky ve vrcholech i limity na průtoky po hranách.

Tím jsme problém cirkulace s limity na průtok hranou převedli na předchozí případ „obyčejné“ cirkulace s požadavky, který umíme vyřešit pomocí hledání „klasického“ maximálního toku.

Pro cirkulace s limity na průtok hranou opět platí důsledek, že pokud jsou všechny požadavky ve vrcholech, dolní i horní limity na průtok hranou celočíselné, tak existuje cirkulace splňující požadavky a limity, která je celočíselná.

12.5.4 Rozvrhování letadel

Problém: Letecká společnost zajišťuje několik pravidelných linek mezi evropskými městy. Dostanete detailní informace o množině letů $\mathcal{L}$. Zajímalo by nás, kolik nejméně letadel je potřeba k zajištění všech letů $\mathcal{L}$. Příklad leteckého řádu je v následující tabulce.

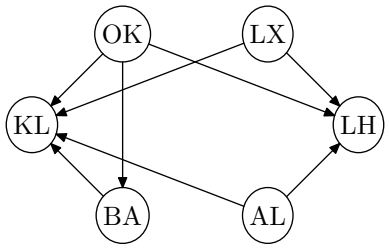
číslo letu	počátek	cíl
OK652	Praha (6am)	Londýn (8am)
LX2008	Milano (7am)	Vídeň (10am)
BA101	Londýn (9am)	Madrid (11am)
AL504	Milano (10am)	Franfurt (11am)
LH2451	Frankfurt (1pm)	Budapešť (3pm)
KL404	Barcelona (6pm)	Budapešť (9pm)

Lety i a j mohou být obslouženy stejným letadlem pokud je cílové letiště i stejné jako počáteční letiště j, a pokud je mezi oběma lety dostatek času na provedení údržby (úklid, doplnění paliva, apod). Další možností je, že letadlo přeletí z cílového letiště i na počáteční letiště j. To ovšem zabere více času a prostoj mezi oběma lety musí být výrazně delší. Přelety stojí výrazně méně než koupě dalšího letadla. Následující tabulka obsahuje příklad 3 letů, které mohou být obslouženy stejným letadlem.

číslo letu	počátek	cíl
OK652	Praha (6am)	Londýn (8am)
BA101	Londýn (9am)	Madrid (11am)
KL404	Barcelona (6pm)	Budapešť (9pm)

Modelování problému: Závislosti mezi jednotlivými lety budeme modelovat orientovaným grafem G. Lety budou tvořit vrcholy grafu a mezi dvěma vrcholy i a j povede orientovaná hrana, pokud letadlo obsluhující let i může obsloužit i let j. Orientované hrany dodržují časovou souslednost. Hrana ij mimo jiné znamená, že let i předchází letu j. Orientovaný graf G je acyklický, protože například časy odletu jednotlivých letů určují topologické uspořádání vrcholů.

Následující obrázek zachycuje modelový graf G pro množinu letů $\mathcal{L}$ z předchozího příkladu.



Otázka, jestli množina letů $\mathcal{L}$ jde obsloužit pomocí k letadel, odpovídá otázce, jestli v grafu G existuje k vrcholově disjunktních cest pokrývajících všechny vrcholy (cesta pokrývá vrchol, pokud přes něj vede).[19]

Řešení: Chceme zjistit, jestli v modelovém grafu G existuje nejvýše k vrcholově disjunktních cest, které pokryjí všechny vrcholy. Problém převedeme na výpočet cirkulace. Hlavní myšlenkou je, že podél každé cesty P_α odpovídající přeletům letadla α pošleme jednotkový tok. Pomocný graf H zkonstruujeme následovně:

- Pro každý let i vytvoříme dva vrcholy $u_i, v_i \in V(H)$. První odpovídá odletu a druhý příletu. Do grafu H ještě přidáme zdroj s a stok t. Požadavky vrcholů nastavíme na $d(s) = -k$, $d(t) = k$ a $d(x) = 0$ pro všechny ostatní vrcholy $x \in V(H)$.

- Každý let i musí být obsloužen. Proto mezi u_i a v_i přidáme hranu a nastavíme její horní i dolní limit na tok na 1. Tedy $l(u_iv_i) = 1$ a $c(u_iv_i) = 1$.

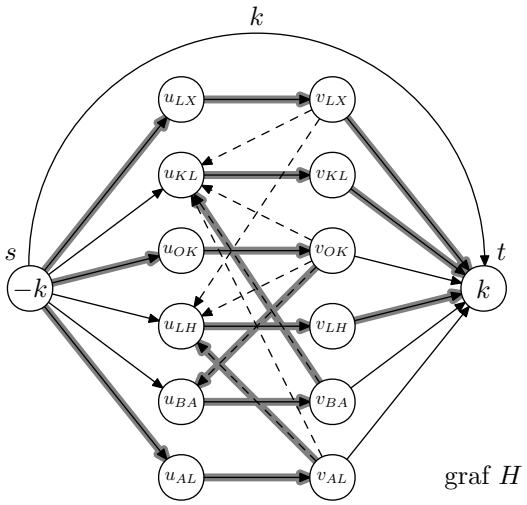
- Pokud může být let i a posléze i let j obsloužen stejným letadlem, tak do H přidáme hranu v_iu_j a nastvíme její horní limit na tok 1. Dolní limit necháme nenastaven. Tedy $c(v_iu_j) = 1$ a $l(v_iu_j) = 0$.

- Protože každé letadlo může zahájit letový den letem i, tak do H přidáme hranu su_i s horním limitem 1 a dolním 0.

- Podobně každé letadlo může skončit den letem j a proto do H přidáme hranu v_jt s horním limitem 1 a dolním 0.

- Může se stát, že nám na obsloužení všech letů bude stačit méně letadel. Proto přidáme hranu st s horním limitem k a dolním 0, po které přebytečná letadla přetečou z s do t.

[19] Porývání grafu cestami si můžeme představit tak, že se cesta potáhne asfaltem. Pak jsou všechny vrcholy/křižovatky na cestě doslova pokryty.



Lemma 32 *Lety $\mathcal{L}$ lze obsloužit pomocí k letadel právě tehdy pokud v pomocném grafu H existuje cirkulace.*

Důkaz: Předpokládejme, že lety $\mathcal{L}$ lze obsloužit pomocí $k' \leq k$ letadel. Nechť $\mathcal{L}(\alpha)$ jsou lety přiřazené letadlu α. Rozvrh letadla α odpovídá orientované cestě P_α v grafu H, která začíná v s, prochází hrany $u_i v_i$ obsluhovaných letů $i \in \mathcal{L}(\alpha)$ a končí v t, Podél této cesty P_α pošleme jednotkový tok. Pokud letadlu α nebyl přiřazen žádný let, tak pošleme jednotkový tok z s rovnou do t po hraně st. Požadavky ve vrcholech grafu H jsou splněny, protože máme k letadel a každé letadlo začíná v s a končí v t. Limity na tok po hranách jsou rovněž splněny, protože každý let byl obsloužen nějakým letadlem.

Pro důkaz druhé implikace uvažme přípustnou cirkulaci v grafu H. Protože všechny limity na hranách jsou celočíselné, tak existuje přípustná cirkulace, která je celočíselná. Tok po každé hraně kromě hrany st je buď 0 a nebo 1. Tok po hraně st může být až k. Jediný vrchol, který se chová jako zdroj, je s. Podobně jediný stokový vrchol je t. Z toho důvodu můžeme cirkulaci rozdělit na k jednotkových toků. Ty odpovídají k dijsjunktním (s,t)-cestám v H. Výjimkou jsou cesty po hraně st. Protože hrana $u_i v_i$ má horní i dolní limit 1, tak každá hrana $u_i v_i$ leží v právě jedné cestě. Každý jednotkový tok odpovídá jednomu letadlu a hrany $u_i v_i$ přes které tok prochází určují lety, které letadlu přiřadíme. ∎

Poznámka: O proti reálnýmu rozvrhování letadel jsme si problém docela zjednodušili. (i) Ve skutečnosti nestačí rozvrhovat pouze letadla, ale každému letadlu musíme přiřadit posádku. Posádka přináší další omezení, protože musí dodržovat pravidelný odpočinek (to letadla nemusí). (ii) Také bychom chtěli maximalizovat zisk. Proto bychom se chtěli vyhnout některým přeletům prázdných letadel. Každé hraně grafu H přiřadíme cenu přeletu a budeme hledat cirkulaci, která minimalizuje celkovou cenu hran, po kterých něco teče. To vede na problém toků minimální ceny (mincost flows), které se „naštěstí“ umí také dobře počítat (bližší informace čtenář najde v [8]).

Poznámka: Úloha se dá vyřešit i bez cirkulací. Můžeme ji přímo převést na toky v sítích – viz cvičení 11 v sekci 12.6.4.

12.6 Příklady

12.6.1 Toky a řezy

1. Nechť $G = (V, E)$ je síť a f_1, f_2 jsou toky v G (funkce z $E \to \mathbb{R}_+$). Rozhodněte jestli jsou následující funkce také tokem v G.

   - $f_1 + f_2$.
   - αf_1 pro $\alpha \geq 0$.

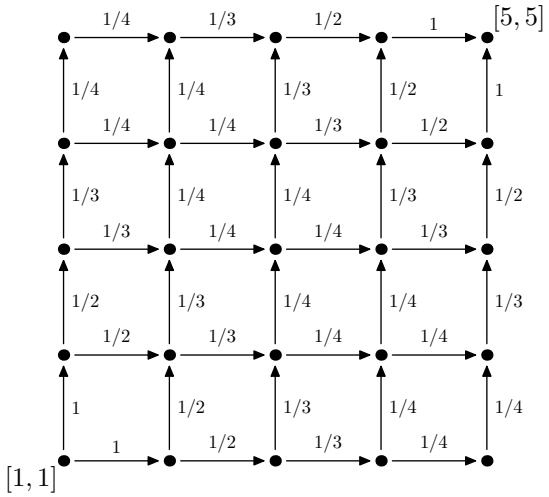
   Funkce $f_1 + f_2$ je definována jako $(f_1 + f_2)(e) := f_1(e) + f_2(e)$ pro každou hranu $e \in E$ (sčítání po složkách). Podobně $(\alpha f_1)(e) := \alpha f_1(e)$ pro každou hranu $e \in E$.

   Jaké vlastnosti (podmínky) z definice toku mohou být porušeny? Co musí funkce $f_1 + f_2$, respektive αf_1, splňovat, aby byla tokem?

2. Nechť $G = (V, E)$ je síť tvaru mřížky 5×5 s vrcholy $V = \{[x, y] : 1 \leq x, y \leq 5\}$ a orientovanými hranami $([x, y], [x + 1, y])$ a $([x, y], [x, y + 1])$ s kapacitami

$$c([x, y][u, v]) = \frac{1}{\min\{x + y - 1,\ 10 - x - y\}}.$$

   Určete maximální tok ze zdroje $[1, 1]$ do stoku $[5, 5]$.

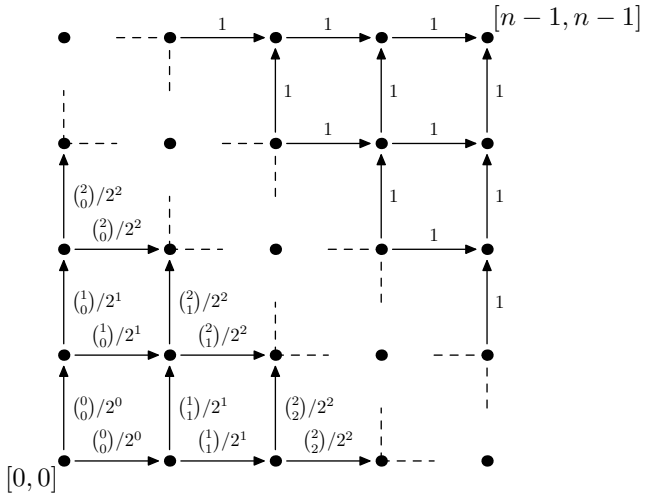


   Nápověda: $\frac{5}{3}$.

3. Nechť $G = (V, E)$ je síť tvaru mřížky $n \times n$ s vrcholy $V = \{[x, y] : 0 \leq x, y \leq n - 1\}$ a orientovanými hranami $([x, y], [x + 1, y])$ a $([x, y], [x, y + 1])$ s kapacitami

$$c([x, y][u, v]) = \begin{cases} \binom{x+y}{x}/2^{x+y} & \text{pro } x + y \leq n - 2 \\ 1 & \text{jinak.} \end{cases}$$

Určete maximální tok ze zdroje $[0,0]$ do stoku $[n-1,n-1]$.



Nápověda: 2.

4. Nechť Q_n je orientovaný graf n-dimenzionální krychle. Tj. graf jehož vrcholy jsou posloupnosti nul a jedniček délky n a dva vrcholy jsou spojeny hranou, pokud se příslušné posloupnosti liší právě v jedné souřadnici. Hranu orientujeme od posloupnosti s menším počtem jedniček do posloupnosti s větším počtem jedniček. Zvolme zdroj $z = (0,0,\ldots,0)$ a stok $s = (1,1,\ldots,1)$ a nastavme kapacity všech hran na jedničku.

   (a) Nalezněte maximální tok ze zdroje do spotřebiče.

   (b) Dokázali byste najít celočíselný tok?

   (c) Dokázali byste najít nikde nenulový tok? Tj. aby každou hranou něco teklo.

5. (Věta o minimálním toku a maximálním řezu) Nechť $G = (V,E)$ je orientovaný graf, $c : E \to \mathbb{R}_+$ je ohodnocení hran, z zdroj a s spotřebič.

   Hledáme minimální tok f ze zdroje do spotřebiče takový, aby $f(e) \geq c(e)$ pro každou hranu $e \in E$. Hodnota $c(e)$ je minimální přípustný průtok hranou. Podobně hledáme maximální řez $\delta(R)$ určený množinou $R \subseteq V$, který odděluje zdroj od spotřebiče a opačný řez $\delta(\overline{R})$ neobsahuje žádnou hranu (všechny hrany mezi R a $\overline{R}$ vedou směremem z R do $\overline{R}$).

   Dokažte, že platí

$$\max\{c(\delta(R)) : \delta(R) \text{ } (z,s)\text{-řez takový, že } \delta(\overline{R}) = \emptyset\} =$$

$$= \min\{|f| : f \text{ } (z,s)\text{-tok takový, že } \forall e \in E \text{ : } f(e) \geq c(e)\}$$

   Ukažme si příklad, proč je podmínka $\delta(\overline{R}) = \emptyset$ z věty nutná. Představme si síť obsahující jen hranu zs a opačnou hranu sz, obě s ohodnocením jedna. Nejmenší přípustný tok posílá jednotku toku po jedné hraně tam a po druhé hraně zpátky. Velikost takového toku je nula. Naproti tomu velikost jediného (z,s)-řezu je jedna. V tomto případě by „věta bez podmínky“ neplatila. Žádný řez nesplňuje podmínku $\delta(\overline{R}) = \emptyset$ a proto je velikost maximálního řezu splňujícího podmínku rovna nule.[20]

[20] Pro znalce lineálního programování: Podmínka souvisí s tím, že tok je vždy nezáporný.

Jako další jednoduchý příklad si rozmyslete tok a řez v síti, kde za z připojíme ještě jednu hranu zx ohodnocenou jedničkou a prohlásíme x za nový a jediný spotřebič.

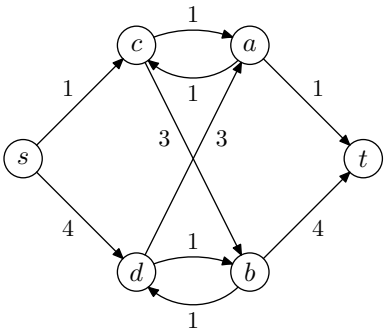
Nápověda: Postupujte podobně jako při důkazu věty o maximálním toku a minimálním řezu.

12.6.2 Algoritmy na toky v sítích

1. Nalezněte maximální tok v síti na obrázku pomocí

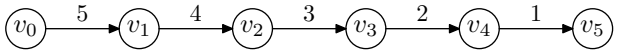
   (a) Ford-Fulkersonova algoritmu
   (b) Edmonds-Karpova algoritmu
   (c) Dinicova algoritmu
   (d) Dinicova algoritmu s metodou 3 Indů
   (e) Goldbergova Push-Relabel algoritmu.

   Který algoritmus se Vám nejjednodušeji provádí na papíře? Který algoritmus proběhne nejrychleji na počítači?



   Úlohu vyřeště ještě pro síť z příkladu 2 na straně 172. Čím je tato síť speciální?

2. Pomocí Goldbergova Push-Relabel algoritmu nalezněte maximální tok v následující síti se zdrojem v_0 a spotřebičem v_5.



   (a) Průběh celého algoritmu si odkrokujte. Kolik kroků algoritmus provede? Kolik kroků provede FIFO push-relabel algoritmus. A kolik kroků provede maximum distance push-relabel algoritmus? Výsledky porovnejte. Které pravidlo pro výběr aktivního vrcholu byste si vybrali?

   (b) Graf na obrázku je zužující cesta délky 5. Dokázali byste spočítat, kolik kroků provede která varianta push-relabel algoritmu na zužující se cestě délky n?

3. Profesor Protékal tvrdí, že ve Ford-Fulkersonově algoritmu není potřeba hledat vylepšující cesty používající hrany v protisměru. Nalezněte co nejjednoduší příklad, na kterém profesorovi dokážete, že jeho zjednodušení Ford-Fulkersonova algoritmu nenajde maximální tok.

4. (Kolik vylepšujících cest stačí?) Ukažte, že maximální tok v síti $G = (V, E)$ se dá vždy najít pomocí nejvýše $|E|$ vylepšujících cest.

   Nápověda: Vylepšující cesty určete až po nalezení maximálního toku.

5. (Kratší důkaz Edmonds-Karpova algoritmu) Dokažte, že v průběhu Edmonds-Karpova algoritmu může každá hrana zmizet ze sítě rezerv nejvýše $n/2$ krát. Z toho dále odvoďte, že Edmonds-Karpův algoritmus provede nejvýše $mn/2$ iterací (vylepšení podél nejkratší vylepšující cesty). Uměli byste ukázat ještě lepší odhad? Dokažte, že každá hrana $uv \in E$ může zmizet ze sítě rezerv nejvýše $n/4$ krát.

   Nápověda: Co víte o změnách $d_f(u)$ pro hranu $uv \in E$?

6. (Otázka k algoritmu 3 Indů) Proč musíme tok velikosti $R(v_0)$ protlačovat z vrcholu v_0 doprava a pak ještě doleva? Nemohli bychom začít ve zdroji s a protlačovat tok velikosti $R(v_0)$ pouze doprava?

7. (Jednotkové kapacity hran) Nechť $G = (V, E)$ je síť se zdrojem s, spotřebičem t a jednotkovými kapacitami hran. Předpokládejme, že G není multigraf a že ke každé hraně $uv \in E$ existuje opačná hrana $vu \in E$ (pokud ne, tak přidáme hranu s nulovou kapacitou).

   Budeme zkoumat, jak rychle poběží Dinicův algoritmus těchto sítích.

   (a) (Jednoduchý odhad) Při hledání toku v čisté síti mají všechny hrany jednotkovou rezervu. Proto při vylepšení toku podél vylepšující cesty odstraním z čisté sítě všechny hrany na této cestě. Na základě toho ukažte, že nalezení blokujícího toku v čisté síti (jedna iterace) bude trvat jen $\mathcal{O}(m)$. To potom dává časovou složitost celého algoritmu $\mathcal{O}(nm)$.

   (b) (Lepší odhad) Zastavme Dinicův algoritmus po k iteracích. Délka nejkratší cesty ze zdroje do spotřebiče je v tento moment $\ell > k$. Dosud jsme nalezli tok f_k a chtěli bychom nalézt maximální tok f. Zbývá nám tedy v síti rezerv najít tok $f_R := f - f_k$.[21] Každá vylepšující cesta zlepší tok alespoň o 1. Zbývá nám tedy najít nejvýše $|f_R|$ vylepšujících cest. Velikost toku lze ze shora odhadnout velikostí libovolného (s,t)-řezu.

   Jak najít vhodný (malý) (s,t)-řez? Zkoumejte řezy mezi jednotlivými vrstvami čisté sítě a ukažte, že existuje řez velikosti nejvýše m/k. Potom i $|f_R| \leq m/k$.

   Na základě toho odvoďte, že zbývá provédst nejvýše m/k iterací. Celkem algoritmus provede nejvýše $k + m/k$ iterací, což pro volbu $k := \sqrt{m}$ dává nejvýše $2\sqrt{m}$ iterací. Z bodu (a) víme, že jedna iterace trvá nejvýše $\mathcal{O}(m)$ a proto je časová složitost Dinicova algoritmu v této síti $\mathcal{O}(m^{3/2})$.

   (c) (Ještě lepší odhad) Myšlenka je podobná jako v předchozí části. Po k iteracích budeme chtít nalézt malý řez.

   Zkoumejte (s,t)-řezy mezi sousedními vrstvami čisté sítě a ukažte, že existuje řez velikosti nejvýše $(n/k)^2$.

   Z toho dostaneme, že algoritmus provede nejvýše $k + (k/n)^2$ iterací, což při volbě $k := n^{2/3}$ dává časovou složitost celého algoritmu $\mathcal{O}(n^{2/3}m)$.

   Nápověda: Označme s_i počet vrcholů v i-té vrstvě. Ukažte, že existuje i takové, že $s_i + s_{i+1} \leq 2n/k$. Velikost řezu mezi i-tou a $(i+1)$-ní vrstvou je rovna počtu hran mezi s_i a s_{i+1} a to je nevýše $s_i \cdot s_{i+1}$.

8. (Jednotkové kapacity a každý vrchol má vstupní nebo výstupní stupeň 1) Nechť $G = (V, E)$ je síť se zdrojem s, spotřebičem t a jednotkovými kapacitami hran. Každý vrchol má vstupní a nebo výstupní stupeň roven jedné.

   Postupujeme stejně jako v předchozím cvičení. Zastavme Dinicův algoritmus po k iteracích. Ukažte, že existuje vrstva čisté sítě, která má nejvýše n/k

[21] Pozor, tok f_R je tokem v síti rezerv a ne tokem v původní síti.

vrcholů. Z toho vyvoďte, že zbývá provést nejvýše n/k iterací a že časová složitost Dinicova algoritmu v této síti je $\mathcal{O}(\sqrt{n}m)$.

9. („Bipartitní graf“) Dostanete bipartitní graf $G = (A \cup B, E)$. Ke grafu přidáme zdroj a spojíme ho hranou se všemi vrcholy $v \in A$. Podobně přidáme spotřebič a spojíme ho hranou se všemi vrcholy $v \in B$. Všem hranám nastavíme jednotkovou kapacitu. Těmito úpravami jsme dostali síť G'. Jaká je časová složitost Dinicova algoritmu puštěného na síť G'?

   Poznámka: Jak jste se dozvěděli v sekci 12.5 o aplikacích toků v sítích, hledání maximálního toku v síti G' odpovídá hledání maximálního párování v bipartitním grafu.

   Nápověda: $\mathcal{O}(\sqrt{n}m)$.

10. (Celočíselné kapacity hran) Nechť $G = (V, E)$ je síť se zdrojem s, spotřebičem t a celočíselnými kapacitami hran $c(e) \in \{0, 1, 2, \ldots, U\}$ pro každé $e \in E$. Označme f maximální tok v síti G.

    Ukažte, že algoritmus provede nejvýše $|f|$ vylepšení podél vylepšujících cest.

    Jednu zlepšující cestu najdeme v čase $\mathcal{O}(n)$. Během celého algoritmu zabere hledání vylepšujících cest nejvýše čas $\mathcal{O}(|f|n)$. Kromě hledání vylepšujících cest ještě provádíme pročišťování čisté sítě. To během jedné iterace zabere čas $\mathcal{O}(m)$ a celkem během algoritmu zabere čas $\mathcal{O}(nm)$.

    Ukažte, že v síti G existuje (s, t)-řez velikosti nejvýše Un. Velikost libovolného toku je menší než velikost libovolného řezu. Proto je celková časová složitost Dinicova algoritmu pro sítě s celočíselnými kapacitami hran $\mathcal{O}(Un^2 + nm)$.

11. (Scaling algoritmus) Nechť $G = (V, E)$ je síť se zdrojem s, spotřebičem t a celočíselnými kapacitami hran $c(e) \in \{0, 1, 2, \ldots, U\}$ pro každé $e \in E$. Nechť $k := \lfloor \log_2 U \rfloor$, neboli k je počet bitů potřebných k zápisu největší kapacity nějaké hrany.

    (a) Dokažte, že velikost minimálního (s, t)-řezu v G je nejvýše Um. (Dokonce můžete najít (s, t)-řez velikosti nejvýše Un.)

    (b) Dostanete pevné číslo K. Ukažte, že vylepšující cesta s kapacitou alespoň K se dá v síti G nalézt v čase $\mathcal{O}(m)$, tedy pokud taková cesta existuje.

    Následující modifikace Ford-Fulkersonova algoritmu se dá použít pro hledání maximálního toku v G.

```
1: Scaling algoritmus pro maximální tok:
2:     U := max{c(e) | e ∈ E}, k := ⌊log2 U⌋
3:     f := 0 (inicializace toku na nulový tok)
4:     K := 2^k
5:     while K ≥ 1 do
6:         while existuje vylepšující cesta P kapacity alespoň K do
7:             vylepši tok f podél cesty P
8:         K := K/2
9:     return f
```

    (c) Dokažte, že předchozí scaling algoritmus vrátí maximální tok.

    (d) Ukažte, že pokaždé, když probíhá krok 5, tak je kapacita rezerv na hranách minimálního řezu G nejvýše $2Km$

    (e) Dokažte, že vnitřní cyklus na řádcích 6–7 může pro každou hodnotu K proběhnout nejvýše $\mathcal{O}(m)$ krát.

(f) Z výše dokázaných pozorování vyvoďte, že časová složitost scaling algoritmu pro hledání maximálního toku je $\mathcal{O}(m^2 \log U)$.

(g) S využitím cvičení 10 ukažte, že časová složitost scaling algoritmu je dokonce jen $\mathcal{O}(nm \log U)$.

12. (Updatování maximálního toku) Nechť $G = (V, E)$ je síť se zdrojem s, spotřebičem t a celočíselnými kapacitami hran. Předpokládejme, že už známe maximální tok v G.

(a) Kapacita jedné hrany $uv \in E$ se zvýší o 1. Navrhněte algoritmus, který v čase $\mathcal{O}(n + m)$ přepočítá maximální tok.

(b) Kapacita jedné hrany $uv \in E$ se sníží o 1. Navrhněte algoritmus, který v čase $\mathcal{O}(n + m)$ přepočítá maximální tok.

13. (Minimální řez pomocí Goldbergova Push-Relabel algoritmu) Předpokládejme, že už jste pomocí Goldbergova Push-Relabel algoritmu nalezli maximální tok v síti $G = (V, E)$. Navrhněte rychlý algoritmus, který nalezne minimální řez. Jak rychle poběží?

14. (Speciální případy Push-Relabel algoritmu) Analyzujte časovou složitost Golbergova Push-Relabel algoritmu v sítích s následujícími kapacitami hran. Časovou složitost vyjádřete vzhledem k n, m a k.

(a) Všechny hrany sítě mají kapacitu 1.

(b) Kapacity všech hran sítě leží v množině $\{1, 2, \ldots, k\}$.

Nápověda: Kolikrát proběhne nenasycující protlačení po hraně e před tím, než se hrana e nasytí?

15. (Goldbergův Push-Relabel algoritmus) Jak by se změnila analýza obecného Golbergova push-relabel algoritmu, kdybychom vrcholy v nezvyšovali až na $\min\{d(w) \mid vw \in E(G_f)\}$ ale vždy jen o 1?

12.6.3 Modifikace sítě

V následujících úlohách vymyslete, jak upravit zadanou síť tak, abychom pro vyřešení úlohy mohli použít hledání klasického toku v síti.

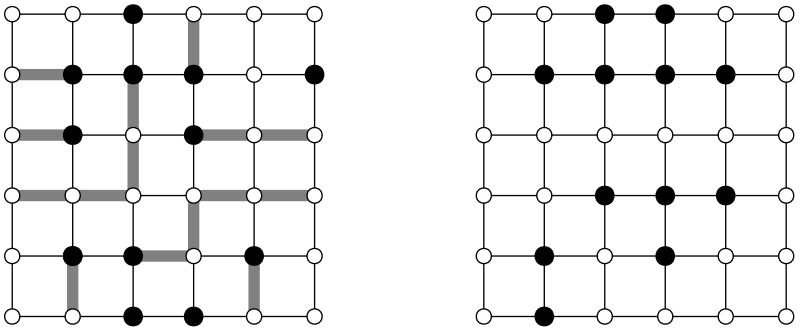
1. Co kdybychom hledali maximální tok v síti s neomezenými kapacitami hran, ale s omezením průtoku skrz vrcholy? Vyslovte a dokažte analogii Ford-Fulkersonovy věty.

2. A co kdyby byly kapacity na hranách a i ve vrcholech (omezní průtoku skrz vrchol)?

3. Jak najít maximální tok v síti, která má několik zdrojů a několik spotřebičů?

12.6.4 Aplikace toků v sítích

1. (Útěk z mřížky) Mřížka $n \times n$ je neorientovaný graf s vrcholy $[i, j]$ pro $1 \leq i, j \leq n$. Vrcholy, které se liší v právě jedné souřadnici a to o jedna, jsou spojeny hranou. Hranice mřížky obsahuje vrcholy pro které je $i = 1$, $i = n$, $j = 1$ nebo $j = n$.



Dostanete $m \leq n^2$ startovních bodů (x_1, y_1), (x_2, y_2),..., (x_m, y_m) ležících v bodech mřížky (černé body na obrázku). Problémem útěku je rozhodnout, jestli ze startovních bodů vede m vrcholově disjunktních cest na hranici mřížky (do m různých bodů na hranici mřížky). Mřížka na obrázku vlevo má řešení, mřížka na obrázku vpravo nemá.

Navrhněte efektivní algoritmus pro rozhodnutí problému útěku. Analyzujte jeho časovou složitost.

Nápověda: Síť s m zdroji a $2n - 2$ spotřebiči, kde hrany i vrcholy mají kapacitu 1. Jak tuto síť upravit, abychom mohli použít algoritmus pro hledání maximálního toku?

2. (Mengerova věta) Pomocí toků v sítích najděte:

   (a) Maximální množinu hranově disjunktních cest mezi danou dvojicí vrcholů. Jak se dá snadno ukázat, že víc cest neexistuje?

   (b) Maximální množinu vrcholově disjunktních cest mezi danou dvojicí vrcholů. Jak se dá snadno ukázat, že víc cest neexistuje?

   (c) Dokažte Mengerovu větu.
   Graf G je hranově (vrcholově) k-souvislý právě tehdy když mezi každou dvojicí vrcholů existuje k hranově (vrcholově) disjunktních cest.

3. (Königova věta) Dostenete bipartitní graf $G = (A \cup B, E)$. Pomocí toků v sítích najděte:

   (a) Maximální párování v bipartitním grafu G. Jak se dá snadno ukázat, že větší párování neexistuje? Párování $M \subseteq E$ je množina disjunktních hran (žádné dvě hrany M nemají společný vrchol).

   (b) Minimální vrcholové pokrytí v bipartitním grafu G. Vrcholové pokrytí $S \subseteq V$ je množina vrcholů taková, že pro každou hranu $e \in E$ je $S \cap e \neq \emptyset$.

   (c) Dokažte Königovu větu:
   Nechť G je bipartitní graf. Velikost maximálního párování v G je rovna velikosti minimálního vrcholového pokrytí v G.

4. (Rozvrhování letadel) V podsekci 12.5.4 jsme řešili rozvrhování letadel pomocí cirkulací. Zkuste úlohu vyřešit přímo převodem na maximální párování v bipartitním grafu.

5. (Pokrytí šachovnice dominem) Dostanete šachovnici, na které už stojí některé figurky. Rozhodněte, jestli lze všechna prázdná políčka pokrýt kostičkami 1×2? Při pokrytí se kostičky nesmí překrývat. Pokud ne, tak kolik nejvíce políček můžete pokrýt?

6. (Rozmístění věží, aby se neohrožovali) Dostanete šachovnici, která má místo některých políček díry. Kolik nejvíc věží můžete na šachovnici rozmístit tak, aby se navzájem neohrožovali? Věž se nesmí položit na díru, ale může přes ní útočit.

7. (Hallova věta) Pomocí věty o minimálním řezu a maximálním toku dokažte:

   Nechť Q je množina a $(S_1, S_2, \ldots S_k)$ je systém jejích podmnožin. Systém různých reprezentantů (SRR) je množina různých prvků $\{q_1, q_2, \ldots, q_k\}$ taková, že $q_i \in S_i$ pro $1 \leq i \leq k$. Hallova věta říká, že systém podmnožin má SRR právě tehdy když pro každou podmnožinu $I \subseteq \{1, 2, \ldots, k\}$ platí $|\cup_{i \in I} S_i| \geq |I|$ (Hallova podmínka).

8. (Dopravní problém) Máme množinu l obchodů O a množinu k továren P. Za určité časové období je továrna i schopna vyrobit nejvýše a_i kusů produktu. Obchod j prodá za stejné časové období b_j kusů produktu. Dostaneme bipartitní graf $G = (O \cup P, E)$, kde $ij \in E$ pokud továrna i může zásobovat obchod j. Rozhodněte, jestli za zadaných podmínek dokáží továrny dostatečně zásobovat všechny obchody. Případně pro každou továrnu spočítejte, kolik kusů produktu má vyrobit a do kterých obchodů se mají produkty rozvézt.

9. (Maximalizace zisku z projektů) Jako ředitel firmy plánujete projekty na příští rok. Můžete začít realizovat projekty $P = \{p_1, p_2, \ldots, p_k\}$. K realizaci některých projektů budete potřebovat některé ze zdrojů $Z = \{z_1, z_2, \ldots, z_l\}$.

   Na realizaci projektu p_i vyděláte částku r_i, ale na druhou stranu k realizaci každého projektu p_i potřebujete množinu zdrojů $S_i \subseteq Z$. Každý zdroj z_j pro $1 \leq j \leq l$ je spojen s náklady c_j. Ale jakmile už zdroj z_j zakoupíme, tak ho můžeme použít ve všech projektech, které ho vyžadují.

   Navrhněte algoritmus, který zjistí, které projekty realizovat, aby byl celkový zisk co největší.

10. (Dilworthova věta) Nechť G je acyklický orientovaný graf. Dva vrcholy jsou nezávislé, pokud neexistuje orientovaná cesta z jedno vrcholu do druhého. Pokrytí grafu řetězci je množina orientovaných cest $\{P_1, P_2, \ldots, P_k\}$ taková, že každý vrchol grafu leží na některé cestě P_i. Dokažte Dilworthovu věta, která říká, že velikost maximální nezávislé množiny je rovna minimálnímu počtu pokrývajících řetězců.

    Nápověda: použijte větu o minimálním toku a maximálním řezu (viz cvičení 5 v podsekci 12.6.1).

11. (Minimální pokrytí cestami) *Cestové pokrytí*[22] orientovaného grafu $G = (V, E)$ je množina $\mathcal{P}$ vrcholově disjunktních cest v G takových, že každý vrchol je obsažen právě v jedné cestě z $\mathcal{P}$. Někdy jednoduše říkáme, že každý vrchol je „pokryt“ právě jednou cestou z $\mathcal{P}$. Cesty mohou začínat a končit v libovolných vrcholech a mohou mít libovolnou délku (včetně délky 0). Minimální cestové pokrytí je cestové pokrytí nejmenším možným počtem cest.

    (a) Vymyslete efektivní algoritmus, který dostane orientovaný acyklický graf $G = (V, E)$ a najde jeho minimální cestové pokrytí.

    Nápověda: Nechť $V = \{1, 2, 3, \ldots, n\}$. Zkonstruujeme orientovaný graf $G' = (V', E')$, kde $V' = \{x_0, x_1, x_2, \ldots, x_n\} \cup \{y_0, y_1, y_2, \ldots, y_n\}$, $E' = \{(x_0, x_i) \mid i \in V\} \cup \{(y_i, y_0) \mid i \in V\} \cup \{(x_i, y_j) \mid (i, j) \in E\}$ a pustíme na něj algoritmus pro hledání maximálního toku.

[22] z anglického „path cover“

(b) Funguje váš algoritmus i pro obecné orientované grafy? To je, nebude vadit, když G bude obsahovat orientované cykly?

12. (Orientovaný Eulerovský tah) Dostanete orientovaný graf $G = (V, E)$. *Orientovaný Eulerovský tah* je tah, který prochází každou hranu grafu právě jednou a to po směru hrany. Tah navíc skončí ve steném vrcholu, ve kterém začal (jde o orientovaný „cyklus“, který může projít některé vrcholy vícekrát).

(a) Ukažte, že graf G obsahuje orientovaný Euleroský tah právě tehdy, když je graf G souvislý a pro každý vrchol $v \in V$ platí $deg^+(v) = deg^-(v)$ (počet šipek vstupujících do vrcholu se rovná počtu šipek vycházejících z vrcholu).

(b) (Pošťákův problém v orientovaných grafech) Pošťák roznáší poštu ve městě, kde jsou samé jednosměrky. Mapa města odpovídá silně souvislému orientovanému grafu $G = (V, E)$ (odevšud se dá dostat kamkoliv). Pošťák potřebuje projít všechny ulice města a roznést poštu. Navrhněte mu takovou trasu, aby se nachodil co nejméně (tj. minimalizujte počet ulic, které bude muset projít vícekrát).

Pokud v grafu nebude existovat orientovaný Eulerovský tah, tak bude pošťák muset projít některé hrany dvakrát. Navrhněte algoritmus, který rozhodne, kolik nejméně a které hrany se mají v grafu zdvojit (dostaneme tak multigraf), aby už v grafu existoval orientovaný Euleroský tah.

12.7 Doporučená literatura

Zvídavého čtenáře můžeme odkázat na přehledový článek Goldberg, Tardos a Tarjan [14] nebo knihu Ahuja, Magnanti a Orlin: *Network Flows* [1]. Mezi hezké lecture notes patří Har-Peled [16]. Z česky psaných textů můžeme doporučit Mareš: *Krajinou grafových algoritmů* [22] a Matoušek, Valla: *Kombinatorika a grafy I.* [25].

Příloha A

Jak se učit

A.1 Jak se učit?

Když chcete s něčím pracovat efektivně, tak si prostudujete návod, abyste věděli, jak to funguje. Ale četli jste už návod na použití vlastního mozku? Mohu vřele doporučit následující knížky.

Buzan: *Mentální mapování.* [5]
Howard: *Příručka uživatele mozku.* [17]
Buzan: *Use both sides of your brain.* [4]

Při psaní této knihy jsem vycházel z moderních poznatků o učení a fungování mozku. Rád bych vám o nich něco řekl.

- **Schémata.** Lidské učení funguje na základě schémat. Když slyšíme nebo vidíme něco nového, tak v hlavě upravujeme starší a zažitá schémata. Srovnáváme nové schéma s tím, co už známe. Pamatujeme si jen to, čím se nové schéma odlišuje od starého. Například když už umíte anglicky, tak se německy naučíte daleko snadněji (podobně to platí pro jazyky programovací). Nebo když už se naučíte jezdit na kole, tak to rychle zvládnete i na malé motorce. Z tohoto důvodu se v této knize snažím používat analogie a motivovat některé úlohy „pohádkou“ ze života.

- **Paměť.** Jak funguje paměť? Každý jsme individualita, ale obecně se vypozorovaly následující poznatky. Člověk si špatně pamatuje dlouhé seznamy. Jsme schopni si dobře zapamatovat nejvýše 7 položek. Proto je v pohádkách sedmero řek, proto je 7 divů světa. U delších seznamů se nám často stane, že něco zapomeneme. Pamatování si delších seznamů ale můžeme obelstít tím, že některé položky sdružíme do skupin. Pak si budeme pamatovat 7 skupin a v každé skupině si budeme pamatovat několik položek. Je to zcela přirozené. Například si pamatujeme různá témata (nejkratší cesta v grafu, minimální kostra, toky v sítích) a v každé skupině si už snadněji vybavíme konkrétní algoritmy na daný problém.

  Lépe si pamatujeme citově zabarvené události, neboli události spojené s nějakou emocí. Emoce se projevují v okamžicích, které zásadně ovlivňují náš život. Například když nám jde o život.[1] Emoce sídlí v části mozku zvané amygdala. V krizových situacích emoce přeberou rozhodování místo velkého mozku (levé a pravé hemisféry), protože jsou rychlejší. Bohužel jsou i primitivnější. Proto

[1] Moderní člověk západního typu má fyzické projevy emocí poměrně dost potlačené. Stává se emočně nečitelným. Často je maskuje tak dobře, že se v sobě nevyzná ani on sám. Ani si pořádně neužije radost. Rozdíl je vidět při srovnání lidmí na ulici v Evropě a v zemích třetího světa, například v Indii.

se lidé, kteří jsou v emoci chovají tak, jak se chovají. V daný moment vůbec nevyužívají velký mozek. Využívají jen vývojově starší části mozku, takže se dostanou na mentální úroveň ještěrky.

Mozek si krizovou situaci zapamatuje lépe, aby se poučil a příště se podobného ohrožení vyvaroval. Pro pamatování si emocí máme speciální paměťové centrum. Je to jiná paměť než ve velkém mozku. Toho můžeme využít tak, že si některé věci spojíme s emocí a tím si je lépe zapamatujeme. Můžeme využít i pozitivní emoce.[2]

Emoce jsou jedním z důvodů, proč tak dobře funguje učení zážitkem (formou hry, simulace apod). Při učení zážitkem provádíme danou činnost, takže se učíme tím, že to děláme. Činnost si navíc spojíme s emocí, takže si ji lépe zapamatujeme. Zaroveň se aktivně snažíme a nejsme pouze ti, kdo přijímají informace.

Emoce jsou nakažlivé. Člověk, který je v emoci, ji svým projevem přenáší na ostatní. Proto si studenti lépe zapamatují „emotivně“ podanou přednášku, než přednášku pronesenou monotónním projevem uspávače hadů.

Také si lépe pamatujeme věci, které jsou unikátní a vyčnívají z řady. Proto si studenti lépe zapamatují blbost, o které se jen zmíníme, než delší (a často i monotónní) kus výkladu.

- **Motivace.** Rychleji se naučíme věci, které se chceme naučit. Proto se nebojte v knize rovnou přeskočit na to, co vás zajímá. V nejhorším se vrátíte kousek zpátky.

- **Mít v tom řád.** Měli bychom neustále vědět, co děláme, znát souvislosti a neztratit se v detailech. Proto věnujme více času analýze toho, co děláme. Připravte si plán, co se chcete naučit, co si dnes přečtete a pak se teprve pusťte do učení či čtení knihy.[3]

- **Pozitivní přístup.** Pokud jsme ve stresu, tak se nám snižuje schopnost se učit a i schopnost si vybavovat (lidově se tomu říká, že máte „okno“). Pokud si vnitřně namluvíme, že je něco těžké, tak to bude těžké. Pokud si namluvíme, že je to lehké, tak to bude lehké. Bohužel, často toto rozhodnutí dělá mozek podvědomě za nás. Když před sebou máme něco neznámého, tak nevíme, co nás čeká. Když ještě neznáme řešení, tak nevíme, jak to bude složité. Mozek už si ale za nás udělá představu. Říká se tomu očekávání a obavy.[4]

  Fungování Vašeho mozku tato kniha nezmění. Jediné, co se dá dělat, je prezentovat výklad tak, aby Váš mozek hned na začátku nabyl dojem, že to bude lehké. Vždy dopředu v pár větách vysvětlíme, co budeme dělat. Případně uvedeme příklad ze života nebo analogii. Také se snažíme, aby bylo čtení této knihy příjemné. Proto se snažíme text dostatečně členit, prokládat „hutné“ kusy textu velkým množstvím obrázků a příkladů, které zvýší srozumitelnost.

[2] Velmi dobře si pamatujeme i události spojené se sexualitou.

[3] Dnes už si nemusíte dělat poznámky formou lineárního textu. Můžete využít *myšlenkových map* nebo grafických schémat s obrázky (mám pohled na celou věc; vidím, kam co patří a souvislosti).

[4] Často nám někdo říká, co je dobré a co špatné, co je lehké a co těžké, kdy budeme šťastni,... My to přijímáme jako skutečnost a automaticky podle toho žijeme. Prohlédnutí toho faktu a získání nadhledu je základem většiny velkých učení (Buddhismus, kniha Čtyři dohody nebo Covey: 7 návyků). Buddhismus říká, že je potřeba rozlišovat realitu a vnímání reality. Realitu nezměníme, ale vnímání reality můžeme změnit vytrénováním mysli. Stejnou myšlenku znáte i z filmu Matrix. Jen tam neřekli, že nepotřebujeme žádné agenty, že je to náš mozek, kdo nás podvědomně ovládá aniž si to uvědomujeme. Jakmile to pochopíte, už není cesty zpět.

- **Znalosti vs. dovednosti.** To, co se učíme, můžeme rozdělit na 3 věci – znalosti, dovednosti a postoje. Znalosti jsou fakta, údaje, data, apod, která se naučíme z knížek nebo přednášek. Znalosti jsou o tom, abychom věděli CO dělat. Dovednosti jsou o tom, JAK to udělat. Jestli dovedeme znalosti využít, jaké máme myšlení či manuální schopnosti. Dovednosti se učíme pouze tím, že danou činnost děláme. Postoje jsou o tom, PROČ to vůbec dělat. Jsou o motivaci a o vůli to udělat.

  Někteří učitelé se při výuce zaměřují pouze na znalosti. Aby se to žáci naučili dobře a mělo to pro ně smysl, tak je potřeba rozvíjet nejen znalosti, ale i dovednosti a postoje.

- **Praxe. Vyzkoušejte si to, hrajte si s tím.** K dobrému porozumění nestačí jen číst tuto knihu, ale je potřeba si nabyté znalosti osvojit. Proto je na konci každé kapitoly několik příkladů. Příklady vyzkouší, jestli jste se pouze naučili nějaký postup jako „kuchařku“, a nebo jestli dovedete o problémech sami přemýšlet. Rozvíjejte svojí kreativitu a myšlení.[5]

  Nevíte, jestli jste vše pochopili? Chcete si otestovat své znalosti? Zavřete knihu a zkuste si probrané algoritmy naprogramovat. Ano, i testem se učíme.

  „Vše co se učíme, se učíme tím, že to děláme.“ Říká se tomu zkušenostní učení nebo anglicky „learning by doing.“ V programování toto heslo platí dvojnásob. Z knížek se programovat nenaučíte. Je to hlavně o tom, kolik času strávíte programováním. Programujte, programujte a programujte. Můžeme vřele doporučit úlohy, které najdete na webových stránkách Korespondenčního semináře z programování (KSP) nebo na stránkách ACM programming contest (tam si své řešení můžete nechat zkontrolovat online, stačí odeslat zdrojový kód).

- **Zapomínání. Pravidelně si vše zopakujte.** Zapomínání je přirozený proces. Krátce po prvním přečtení jedné kapitoly si běžný člověk pamatuje zhruba 75% obsahu. Množství informací, které si pamatuje, s časem klesá a po měsíci se dostane až na nějakých 25%. Abychom zabránili této ztrátě informací, tak je potřeba si vše opakovat, opakovat a opakovat. Opakujeme si to i tím, že to používáme. Doporučuje se si vše zopakovat za 1 hodinu, za 1 den, za 1 týden, za 1 měsíc a za 1 rok. Po několika opakováních se křivka zapomínání stane méně strmou a po pár měsících zůstane množství informací, které si pamatujeme, nad 50%.[6]

- **Únava, udržení pozornosti. Dělejte si přestávky.** Pokud se nepřetržitě soustředíme, tak naše pozornost s časem klesá. Když víme, že bude učení trvat předem známý, pevný čas (například přednáška), tak je naše pozornost největší na začátku, pak pomalu klesá, nejnižší je zhruba uprostřed a s blížícím se koncem se zvyšuje. Proto je dobré dělat přestávky a rozdělit učení do více časových úseků. Na grafu má pozornost tvar "misky". Pokud učení rozdělíme do více menších "misek" (na časové ose jsou vedle sebe), tak jejich dna neklesnou tak hluboko. Paradoxně se toho s přestávkami naučíme více než za stejný čas v kuse.

- **Správné dýchání a posez.** Zní to divně, ale více jak polovina moderní populace neumí dýchat. Plochý dech vede k menšímu prokrvení celého oraganismu. V důsledku pak máme méně energie a dříve se unavíme. Výuka správného dýchání je základem řady dovedností – od správně posazeného hlasu až třeba po jógu.

[5] Jinak vás nahradí Wikipedie.

[6] Někdo mi říkal, že pro něj nemá smysl vstoupit do kurzu a naučit se další jazyk, protože nebude mít čas si ho opakovat a procvičovat a za pár let by ho stejně zapomněl.

Většina lidí špatně sedí na židli nebo v křesle. Ani si neuvědomují, jak si tím brání v dýchání. Pokud se vyvalíme, sedíme zkrouceně a máme propadlé břicho, tak můžeme dýchat pouze hrudníkem. Brániční dýchání do břicha v ten moment funguje jen minimálně. Dýchání hrudníkem není dostatečné.

Lepšímu učení prospívá i pravidelný pohyb. Když si o přestávce zacvičíme, rozproudíme krev a okyslíčíme tělo, tak se toho posléze více naučíme.

- **Klíčová slova a kontext.** Je spousta slov, která nenesou žádnou informační hodnotu (lidově jim říkáme „omáčka“). Na druhou stranu jsou slova, která v nás vyvolávají jisté asociace (těm říkáme „klíčová slova“). Stejné slovo může vyvolávat různé asociace podle toho v jakém kontextu je použité a jaké má daný člověk zkušenosti. Při psaní poznámek je dobré se omezit pouze na klíčová slova. Psaním omáčky akorát ztrácíte čas (jak při psaní, tak při čtení) a vaše zápisky se stanou méně přehledné. Na druhou stranu každý máme svoje vlastní asociace, takže vaše poznámky mohou být nepřenosné mezi ostatní lidi. To je základní rys lidské komunikace: Jeden člověk si něco myslí. Nějak to říká. Druhý to nějak slyší a něco svého si o tom myslí.

  Každý máme jiný kontext, který je tvořen našimi znalostmi, zkušenostmi, náladou, kulturou,... Když slyšíme něco nového, tak to srovnáváme se schématy, která už známe. Proto se může stát, že si nové informace vyložíme po svém.

  Pokud chcemě někomu něco vysvětlit, musíme věc zaobalit dostatečně velkým kontextem, aby si druhá strana vyložila informace správně.[7] Viktor Frankl, jeden z velkých filosofů, o přednáškách říká: „Není takový problém, když studenti něco nepochopí. Problémem je, když to pochopí špatně.“

  Mě se stalo už několikrát, že se mi líbil citát od jednoho filosofa. Myslel jsem si, že mu rozumím, ale teprve za pár let jsem zjistil, co tím filosof opravdu myslel. Podobně je to s významem vět v matematice, ale i s komunikací v běžných partnerských vztazích.

[7] Proto se při vyjednávání, ve smlouvách apod. věnuje tolik času vysvětlení si základních pojmů.

A.2 Proslov ke studentům

U nohou vám leží svět plný nepřeberného množství možností.
Záleží jen na vás, kam chcete dojít a kterým směrem se vydáte.

V dnešní době neustále rostou nároky na technologie. Chceme po počítačích stále víc. Zrychlovat můžeme jak hardware, tak i software. Když použijeme přirovnání, tak hardware je jako nůž a software říká, jak ho budeme používat. Software určuje, kterou stranou čepele s ním budeme krájet – ostrou nebo tupou? Hardware se pomalu zlepšuje, ale těch největších zlepšení dosáhneme právě softwarem.[8] Abychom zvládli spočítat víc, tak musíme rozvíjet především software. Proto je potřeba znát a vyvíjet efektivní algoritmy.

Dva programátoři: Podívejme se na dva extrémní případy, jak může vypadat programátor.

Je spousta lidí, kteří se z návodů na internetu naučili snadno a rychle dělat webové stránky a programovat. Jejich styl lze vystihnout hesly „cut&paste“ a „nějak to zbastlit, aby to fungovalo“. Nic lepšího neumí, a to je škoda. Jejich kód je pro ostatní nečitelný a proto se téměř nedá upravovat. Jejich programy jsou pomalé a proto neprogramují nic náročného. Raději se drží ve vodách webových skriptů, kde to tolik nevadí. Neradi přemýšlí, raději používají zaběhané a osvědčené metody.

Co dělá dobrý programátor? Dobrý programátor si nejprve udělá plán. Promyslí si účel a požadavky (rychlost, množství paměti, čas na naprogramování, velikost dat, které se budou zpracovávat). Podle toho zvolí vhodný programovací jazyk, vhodný algoritmus a vybere knihovny případně další nástroje, které použije. Pak se pustí do práce. Je zvyklý psát bez chyb a pro jistotu si každý modul pořádně otestuje. Čas vložený do testování se mu bohatě vrátí. Laděním chyb jinak ztratí většinu svého času. Jeho programy jsou efektivní, rychlé. Jeho kód je čitelný a jiný programátor se v něm při provádění úprav rychle zorientuje. Má radost, když může programovat něco, co je výzvou. Něco „zajímavějšího“. Má radost, když může něco vymyslet.

Můžete si vybrat, kterým programátorem chcete být. Je to jen ve vašich rukou.

Pasivním chováním se nic neučíte. Knížky ani přednášky vás skoro nic nenaučí, pokud se nebudete snažit. Sice si uděláte čárku, že jste něco absolvovali, ale k čemu vám to bude? Učitel či knížka je jen průvodce na vaší cestě za vzděláním. Po této cestě musíte kráčet sami.[9] Musíte přemýšlet, sami se ptát na souvislosti, zkoušet to na příkladech, hrát si s tím, programovat. Lidstvo se posunulo dopředu právě díky lidské touze, poznat něco nového.

Proč se říká, že se někdo „vypracoval“ na nějakou pozici? Od slova pasivně sedět to není... Edison říká, že úspěch je 1% geniality a 99% potu.

[8] Když lidé vymyslí něco nového, tak to nejprve použijí jako software. Teprve časem, když se to osvědčí, podle toho postaví hardware. Říkáme, že se určité funkce “zadrátovaly” do hardwaru.

[9] Škoda, že ve vzdělání neexisují jezdíci schody. Že vám znalosti nepředepíše doktor v tabletách. Nebo snad bohudík?

A.3 Proslov k učitelům

Vážení učitelé. Vám všechna čest, že se snažíte vychovávat český národ. Že vytváříte budoucí českou inteligenci. Je to hlavně ve vašich rukou.

Několik let jsem učil na vysokých školách, a za tu dobu jsem leccos objevil. Učení mě baví. V následujících odstavcích bych chtěl předat pár zkušeností z výuky na vysoké škole. Mohlo by se to hodit doktorandům či mladým učitelům, kteří si hledají vlastní koncepci výuky.

Snažil jsem se nadchnout studenty pro danou věc a **vypěstovat jejich vztah k předmětu.** Vždyť první kontakt s každým předmětem určuje, jak na něj studenti budou v budoucnu nahlížet. Pokud je to nadchne, tak se o tom budou chtít dovědět více. Pokud ne, tak už se k tomu nevrátí.[10]

Mrzí mě, jak se někteří studenti chovají[11] a také jak to na některých školách funguje. Trochu je to dáno naší kulturou a taky dnešní dobou. Studenti jsou děsně pasivní. Mají představu, že si sednou do lavice, zasunou si do hlavy trychtýř a učitelé jim tam vše nalijí. Tento přístup vůbec nerozvíjí kreativitu a myšlení. Studenti se akorát naučí papouškovat to, co slyší, a používat návody z kuchařky. To není dobře. Je to podobné jako s učením cizích jazyků. Jedna věc je pasivně rozumět a druhá věc je umět mluvit. Přeci nechcete, vážení učitelé, aby studenti pasivně rozuměli vašemu výkladu, ale neuměli myslet. Učitel by se měl snažit s touto pasivitou bojovat. Někteří učitelé ji bohužel tvrdě podporují. Studenti, kteří přemýšlí, jim totiž komplikují život hloupými dotazy.

Nemá smysl nadávat na systém školství v České Republice. Vždyť při výuce se setká jen učitel a jeho žáci. Nikdo další. Záleží hlavně na učitelích, jakým způsobem bude probíhat výuka.

Učitel by se měl neustále vzdělávat, prahnout po vědění. Neměl by dělat věci jistým způsobem jen proto, že ho to tak někdo naučil. Měl by vědět, co dělá a proč. Krásně to popisuje následující vtip.[12]

Pochutnalovi budou mít k nedělnímu obědu husu. Při té příležitosti se manžel ptá ženy: „Mařko, proč vždycky uříznеš té huse stehna a upečeš tu husu bez nich? Já mám stehna ze všeho nejraději." „Já ti ani nevím, ale moje maminka to tak vždycky dělala." „Tak se jí zeptej." Při návštěvě maminky se jí na to ptají a ona jim odpoví: „Já ti nevím, ale moje maminka to tak vždycky dělala." Tak se jdou zeptat prababičky a ta jim odpoví: „No já nevím, proč to holky děláte vy, ale za nás dělali děsně malý trouby."

Jak bojuji s pasivitou studentů já? Osvědčilo se mi přidat do výuky zážitkovou formu. (Už jste zkoušeli řadit studenty do řady podle velikosti předem zvoleným třídícím algoritmem?) Abych studenty více zaujmul, tak některé úlohy zadávám jako problém ze života (něco, co studenty zajímá[13]). Občas udělám soutěž o to, kdo vymyslí lepší řešení, nebo která skupina ho vymyslí rychleji. Samozřejmě s vtipnou odměnou pro vítěze (například v soutěži o nejefektivnější algoritmus, ve kterém se hází vajíčka z mrakodrapu, je odměnou velké čokoládové vejce).

[10] Někteří lidé si celý život myslí, že neumí zpívat, protože jim to řekla paní učitelka na základní škole. Takový nesmysl. Vždyť je to stejné, jak když vám tělocvikář řekne, že neumíte běhat.

[11] A to vůbec nemluvím o jejich nezodpovědnosti, o pozdních příchodech. Moc se mi líbí přístup jednoho nejmenovaného profesora z MFF UK, který se začátkem hodiny zamkne posluchárnu a odemkne ji až po půl hodině.

[12] Realita v životě už tak vtipná není. Potkávám se s tím celkem často.

[13] Ale zjistit, co studenty zajímá, může být problém. V první řadě je potřeba trocha empatie a vůbec si udělat čas nad takovou věcí přemýšlet. Řadu věcí zjistíme během neformální diskuse se studenty před nebo po vyučovací hodině

Rád nechávám studenty, aby sami navrhli, jak se má postupovat, a pomocí nápověd je směřuji k cíli. Sice to trvá déle, ale má to pro studenty úplně jiný význam. Místo pasivního přijímání informací zkouší aktivně přemýšlet. Pokud si na to přijdou sami, tak si to daleko lépe zapamatují.

Ve cvičeních se snažím, aby každý zkoušel pracovat na svém vlastním řešení a ne aby se všichni koukali na toho „chudáka“ u tabule. Když už se něco řeší u tabule, tak by to mělo být efektivní. Když všichni řeší úlohu po svém do sešitu, tak jsou všichni aktivní na 100%. Když stejnou úlohu řeší někdo u tabule, tak je aktivní 1 člověk (ten co je u tabule) a ostatní jsou pasivními posluchači nebo opisovači. Na druhou stranu, občas je potřeba u tabule zopakovat teorii nebo ukázat vzorové řešení. To by mělo být připravené a efektivní.

Proti tomu, aby studenti prezentovali své řešení u tabule, jde fakt, že se většina studentů neumí vyjadřovat. Když jsou u tabule, tak to z nich leze v nesrozumitelné podobě a nebo jako z chlupaté deky. Přesto jsem se rozhodl poskytovat studentům omezený prostor, aby si zkusili se vyjádřit. Kde jinde se to mají naučit, když ne tady? Pokud se studenti naučí vyjadřovat se, tak dostanou do života mnohem více, než ze samotné náplně cvičení. Ocení to vyučující ve vyšších ročnících, budoucí zaměstnavatel i všichni kolegové z týmu, ve kterém bude student pracovat.[14]

Moje běžné cvičení vypadá tak, že si donesu vytištěné příklady, které postupně řešíme. Zadám příklady a pak studenty obcházím a ptám se, jak jsou na tom a na co už přišli. Jako správný průvodce jim řeknu: „Tudy cesta nevede. Podívej se na tenhle protipříklad.“ a nebo je naopak povzbudím či nasměruji správným směrem: „Paráda, už to máš skoro vyřešené. Zkus ještě vymyslet, jak to udělat, aby.... “. Díky tomu každý dostane skoro individuální přístup. Na závěr poprosím nejlepšího řešitele dané úlohy, aby své řešení prezentoval na tabuli.

A co když jsou studenti stále pasivní a nechtějí nic dělat? Tak nic, vy jste zkusili všechno možné a přeci nebudete „házet hrách na stěnu“.

Jaký učitel, takoví studenti. Na studenty se ohromě přenáší to, v jakém stavu je učitel. Hodně dělá **pozitivní očekávání.** Pokud studentům řeknu, že jim věřím, že to zvládnou, že je to jednoduché, tak to tak vezmou a budou šikovní. Pokud jim řeknu, že to stejně nepochopí, tak se ani nebudou snažit.

Pozor na **projekci vlastního já.** Někdy si myslím, že jsou studenti unavení, ale jsem to jen já, kdo je unavený. Podobně si mohu myslet, že jsou studenti nadšení, ale jsem to jen já, kdo je skutečně nadšený.

[14] Vyjadřovací schopnosti se musí trénovat jako jakákoliv jiná činnost. Bohužel se to na řadě škol nedělá, nebo se tiše předpokládá, že se to studenti měli naučit už na střední škole. Ale co s tím? Můžeme je začít trénovat. Nejlepší forma je pomocí referátu. Student si vše připraví doma v psané formě (vše sepíše na papír a má přitom čas v klidu přemýšlet), pak je připuštěn k referátu před ostatními studenty. Tam si natrénuje prezentační dovednosti. Ostatní studenti mu mohou poskytnout zpětnou vazbu (čemu neporozuměli, co se jim líbilo).

A.4 Nápad na projekt

Často známe pro jeden problém celou řadu řešení. Jak poznat, které řešení je v praxi lepší? Ano, nezbývá než to naprogramovat, spustit, a porovnat výsledky.

Líbilo by se mi, když někdo udělal univerzální systém na porovnávání algoritmů. Systém by obsahoval sadu problémů, které se dají řešit. Například nejkratší cesta v grafu, minimální kostra, maximální tok v síti. Každý problém by vyžadoval určitý formát vstupu a výstupu. Každý student by si mohl naprogramovat své řešení, poslat ho do systému a porovnat ho s ostatními programy. V systému by se pak dalo snadno dohledat nejrychlejší řešení daného problému.

Systém by pomocí grafu zobrazoval časovou složitost algoritmů na vstupech různé velikosti. Umožňoval by srovnávat různé metody (například Dijkstrův a Floyd-Warshallův algoritmus, nebo různé algoritmy pro toky v sítích). Také by se dali porovnávat různé implementace, různé programovací jazyky.

Studenti by tak získali lepší povědomí o tom, jak je která metoda či algoritmus dobrý. Ba co víc, také by poznali rozdíly mezi implementacemi v různých programovacích jazycích – ve kterém programovacím jazyce bude zdrojový kód nejkratší? Ve kterém programovacím jazyce poběží program nejrychleji? Díky zobrazování zdrojových kódů by i věděli, jak je pracné daný algoritmus naprogramovat. Studium ukázkových zdrojových kódů by pro ně bylo vzorem toho, jak se může programovat.

Příloha B

Značení

B.1 Matika

$\mathbb{N}$ ..... množina přirozených čísel
$\mathbb{Z}$ ..... množina celých čísel
$\mathbb{R}$ ..... množina reálných čísel
$\mathbb{R}_+$ ..... množina reálných nezáporných čísel
$[n]$ ..... množina čísel $\{1, 2, \ldots, n\}$
#objektů .... „počet“ objektů
$\log n$ ..... dvojkový logaritmus n
$\alpha(n)$ ..... inverzní Ackermannova funkce pro n
A $\iff$ B ..... ekvivalence obou tvrzení, A platí právě tehdy když B
$\mathcal{O}(g)$ ..... asymptotické $\mathcal{O}$: $f = \mathcal{O}(g) \iff \exists\, c > 0\, \forall\, n \geq n_0 : f(n) \leq c \cdot g(n)$
$\Omega(g)$ ..... asymptotické Ω: $f = \Omega(g) \iff \exists\, c > 0\, \forall\, n \geq n_0 : f(n) \geq c \cdot g(n)$
$\Theta(g)$ ..... asymptotické Θ: $f = \Theta(g) \iff f = \mathcal{O}(g)\ \&\ f = \Omega(g)$

Permutace π.
Úvod do exponenciál
Úvod do logaritmu + Harmonické číslo
Číslo v pozičních soustavách o základu a.

B.2 Grafy

$G = (V, E)$ ... graf s vrcholy V a hranami E
V ..... vrcholy grafu
$V(G)$ ..... vrcholy grafu G
V ..... vrcholy grafu
$E(G)$ ..... hrany grafu G
n ..... počet vrcholů grafu
m ..... počet hran grafu
$H \subseteq G$ ..... H je podgrafem G
$G[w]$ ..... podgraf grafu G indukovaný množinou vrcholů $W \subset V$
$deg\, v$ ..... stupeň vrcholu v
$G + e$ ..... přidání hrany e do grafu G
$G - e$ ..... smazání hrany e z grafu G
$G - v$ ..... smazání vrcholu v z grafu G
$G.e$ ..... graf po kontrakci hrany e
$G.W$ ..... graf po kontrakci množiny vrcholů W
uPv ..... jednoznačně určený úsek cesty P mezi vrcholy u a v

uTv ..... jednoznačná cesta ve stromě T mezi vrcholy u a v
$c(e)$ ..... ohodnocení hrany e, cena hrany
$c(E)$ ..... cena hran $e \in E$, tj. $\sum_{e \in E} c(e)$
$w(v)$ ..... váha vrcholu v
$w(V)$ ..... váha vrcholů $v \in V$, tj. $\sum_{v \in V} w(v)$
C_v ..... komponenta souvislosti obsahující v
$\delta(A)$ ..... řez grafu určený množinou vrcholů A

B.3 Algoritmy

`A`$[\cdot]$ ..... pole A (tečka jako argument zastupuje všechny položky)
`A`$[2..5]$ ..... podúsek pole A mezi indexy 2 a 5 včetně

Vysvětlení pseudokódu, ve kterém píšeme algoritmy.

Rejstřík

Literatura

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[2] R. K. Ahuja and J. B. Orlin. A fast and simple algorithm for the maximum flow problem. *Operations Research*, 37(5):748–759, 1989.

[3] R. K. Ahuja, J. B. Orlin, and C. Stein. Improved algorithms for bipartite network flow. *SIAM J. Comput.*, 23(5):906–933, 1994.

[4] T. Buzan. *Use Both Sides of Your Brain*. Plume; 3rd edition, 1991.

[5] T. Buzan. *Mentální mapování*. Portál, 2007.

[6] B. Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *J. ACM*, 47:1028–1047, November 2000.

[7] K. W. Chong, Y. Han, and T. W. Lam. Concurrent threads and optimal parallel minimum spanning trees algorithm. *J. ACM*, 48:297–323, March 2001.

[8] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver. *Combinatorial optimization*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[9] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[10] S. Dasgupta, C. Papadimitriou, and U. Vazirani. *Algorithms*. McGraw-Hill Higher Education, 2007. `http://www.cs.berkeley.edu/~vazirani/algorithms.html`.

[11] R. Diestel. *Graph Theory*, volume 173 of *Graduate texts in mathematics*. Springer, Berlin, 3rd edition, 2005.

[12] J. Erickson. Algorithms course materials, 2009. `http://www.cs.uiuc.edu/~jeffe/teaching/algorithms`.

[13] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *J. ACM*, 45(5):783–797, 1998.

[14] A. V. Goldberg, E. Tardos, and R. E. Tarjan. Network flow algorithms. In *Paths, Flows and VLSI-Design (eds. B. Korte, L. Lovasz, H.J. Proemel, and A. Schrijver)*, pages 101–164. Springer Verlag, 1990.

[15] A. V. Goldberg and R. E. Tarjan. Solving minimum-cost flow problems by successive approximation. In *STOC*, pages 7–18. ACM, 1987.

[16] S. Har-Peled. Cs473g - graduate algorithms, 2007. `http://www.cs.uiuc.edu/class/fa07/cs473g/lectures.html`.

[17] P. Howard. *Příručka uživatele mozku.* Portál, s.r.o, 2005.

[18] D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition.* Addison-Wesley, 1973.

[19] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms.* Addison-Wesley, 1973.

[20] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, 1973.

[21] M. Mareš. *Graph Algorithms (The Saga of Minimum Spanning Trees).* PhD thesis, Charles University, Prague, Czech Republic, 2008. `http://mj.ucw.cz/papers/saga/`.

[22] M. Mareš. *Krajinou grafových algoritmů.* ITI Series, Prague, 2008. `http://mj.ucw.cz/vyuka/ga/`.

[23] J. Matoušek and J. Nešetřil. *Kapitoly z diskrétní matematiky.* MatfyzPress, 1996.

[24] J. Matoušek and J. Nešetřil. *Invitation to Discrete Mathematics.* Oxford University Press, 1998.

[25] J. Matoušek and T. Valla. Kombinatorika a grafy i., 2005. `http://kam.mff.cuni.cz/~valla/kg.html`.

[26] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *J. ACM*, 49:16–34, January 2002.

[27] A. Schrijver. *Combinatorial Optimization - Polyhedra and Efficiency.* Springer-Verlag, Berlin, 2008.

[28] A. Schrijver. Lecture notes, 2008. `http://homepages.cwi.nl/~lex/`.

[29] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22:215–225, April 1975.

[30] P. Töpfer. *Algoritmy a programovací techniky.* Prometheus, 1995.