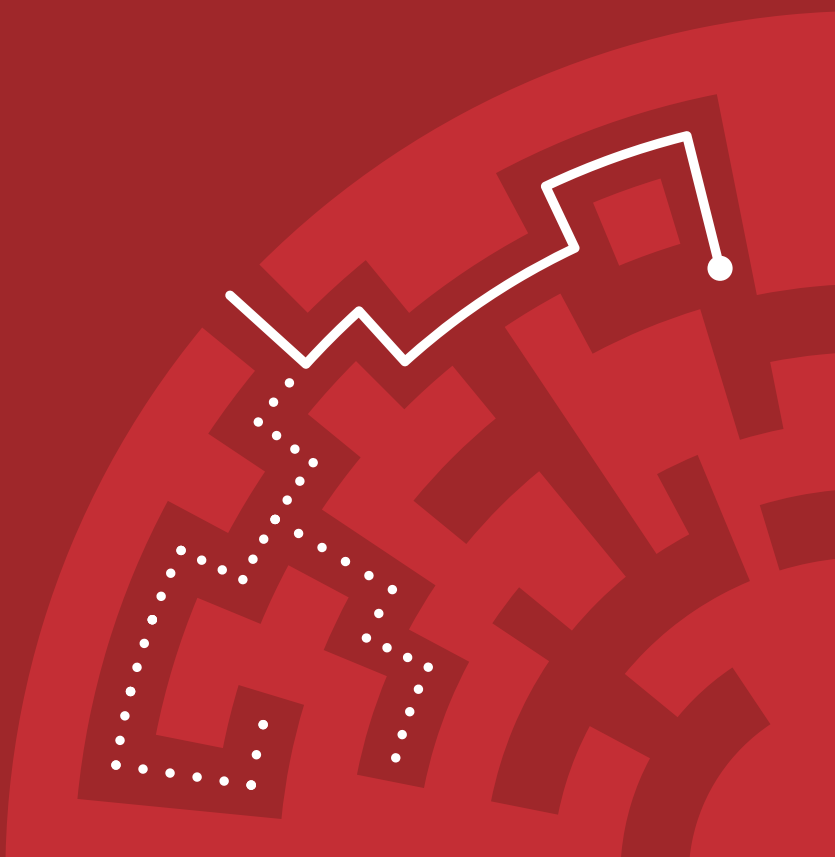
Martin Mareš, Tomáš Valla

Průvodce labyrintem algoritmů



Edice CZ.NIC

PRŮVODCE LABYRINTEM ALGORITMŮ
Martin Mareš, Tomáš Valla

Vydavatel:
CZ.NIC, z. s. p. o.
Milešovská 5, 130 00 Praha 3
Edice CZ.NIC
www.nic.cz

1. vydání, Praha 2017
Zapracována errata 2021-08-26.
Kniha vyšla jako 15. publikace v Edici CZ.NIC.
ISBN 978-80-88168-22-5

CZ.NIC | SPRÁVCE DOMÉNY CZ

ISBN 978-80-88168-22-5

— Martin Mareš, Tomáš Valla

Průvodce labyrintem algoritmů

— Edice CZ.NIC

Předmluva vydavatele

Vážení čtenáři,

snad mi jako vydavateli této skvělé knihy odpustíte, když hned úvodem poruším jedno její nepsané, leč velmi pedanticky dodržované pravidlo. Totiž to, že všechny zde vyřčené či spíše zapsané věty jsou podloženy precizními důkazy. Mé tvrzení se opírá pouze o mé vlastní subjektivní pozorování a pro jakýkoliv seriózní důkaz mi chybí dostatek píle a vlastně i znalostí.

Nicméně přesto věřím, že mnoho čtenářů pokýve hlavou, když prohlásím, že v současné době programuje kde kdo, ale skutečných programátorů je přibližně jako známého vzácného koření z čeledi kosatcovitých. Bez ohledu na to, jakým programovacím jazykem píšeme a o jaký typ softwaru jde, hlavním rozdělovníkem mezi programátorem a skutečným programátorem pro mě vždy byla schopnost pochopit a používat efektivní algoritmy a datové struktury.

Aniž bych chtěl nějak negativně ovlivnit živobytí výrobců hardwaru, přál jsem si vždy, aby skutečných programátorů bylo co nejvíce. Proto jsem ani na vteřinu neváhal s nabídkou, když za mnou přišel můj bývalý spolužák a také mimo jiné vynikající dlouholetý pedagog Martin Mareš s tím, že s Tomášem Vallou píší knihu o algoritmech a hledají, kdo by jim ji mohl vydat. Jen těžko najít v této zemi tak oddané propagátory tohoto umění jako právě je dva!

Přeji vám tedy příjemné čtení této poučné knihy. Martin i Tomáš vám v ní velmi podrobně vysvětlí, jak se labyrintem algoritmů prochází. Zdali v něm i v budoucnu vždy najdete správnou cestu, záleží jen na tom, jak podrobně budete číst.

Ondřej Filip, CZ.NIC
Praha, 9. června 2017

Předmluva autorů

Předmluva autorů

Každý, kdo se pokusí napsat složitější počítačový program, brzy zjistí, že více než na detailech konkrétního programovacího jazyka záleží na tom, jak řešení komplikované úlohy vyjádřit pomocí řady elementárních kroků srozumitelných počítači.

Tomuto vyjádření se obvykle říká *algoritmus* a právě tím, jak algoritmy navrhovat a analyzovat, se bude zabývat celá tato kniha. Napsali jsme ji pro každého, kdo už umí trochu programovat v jakémkoliv jazyce a chtěl by se naučit algoritmicky myslet. Hodit se může jak studentovi informatiky, tak zkušenému programátorovi z praxe.

Kniha vychází z mnoha let našich přednášek: Martinových na Matematicko-fyzikální fakultě Univerzity Karlovy a Tomášových na Fakultě informačních technologií Českého vysokého učení technického. Kniha pokrývá obsah obou přednášek, ale často se snaží vám čtenářům ukázat i něco navíc – naznačit, jak rozvyprávěný příběh pokračuje.

Jelikož se k analýze algoritmů obvykle používají matematické prostředky, předpokládáme, že čtenáři jsou zběhlí ve středoškolské matematice (logaritmy, exponenciály, jednoduchá kombinatorika) a základech vysokoškolské (jednoduchá lineární algebra a matematická analýza, teorie grafů). Většinou se snažíme vystačit si s co nejjednodušším aparátem, případně čtenáře odkázat na vhodný zdroj, ze kterého se lze aparát doučit. Upozorňujeme, že oproti středoškolským zvyklostem považujeme nulu za přirozené číslo a místo desetinné čárky píšeme tečku.

Algoritmy nezapisujeme v žádném konkrétním programovacím jazyce, nýbrž v takzvaném *pseudokódu* – abstraktním zápisu, který je příjemně srozumitelný člověku, ale také se dá s minimem úsilí převést do libovolného programovacího jazyka. Na začátku knihy píšeme pseudokódy velmi detailně, později se stávají abstraktnějšími, protože čtenář už dávno pochopil základní obraty a nemá smysl je znovu podrobně rozepisovat.

Nedílnou součástí výkladu jsou cvičení v závěru většiny oddílů. Ponoukají čtenáře k tomu, aby nad myšlenkami z daného oddílu uvažoval a pokusil se je použít k řešení dalších úloh. Pokud si nebudete vědět rady, na konci knihy najdete k některým cvičením nápovědu. Občas je na konci kapitoly ještě jeden oddíl s dalšími úlohami na procvičení látky z celé kapitoly.

Některá cvičení a oddíly knihy jsou označeny jednou nebo dvěma hvězdičkami. To znamená, že se v nich nachází pokročilejší a často také o něco obtížnější materiál. Při prvním čtení je doporučujeme přeskakovat, později si je užijete mnohem více. Mimo jiné proto, že se v nich mohou hodit znalosti z následujících kapitol.

Na konci knihy naleznete rejstřík, který také obsahuje přehled používaného matematického značení. Jednopísmenné značky jsou zařazeny pod příslušnými písmeny, ostatní symboly na začátku rejstříku.

Dodejme ještě, že do každé odborné knihy se přes všechnu snahu autorů vloudí pár chyb. Pokud na nějakou narazíte, dejte nám prosím vědět na adrese *pruvodce@ucw.cz,* ať ji můžeme v příštím vydání opravit. Seznam všech nalezených chyb budeme udržovat na webové stránce *http://pruvodce.ucw.cz/*. Tamtéž najdete elektronickou verzi celé knihy.

Doporučená literatura

Rádi bychom zmínili několik knih, jež nás při přednášení a psaní inspirovaly, a doporučili je všem čtenářům, kteří by se rádi o algoritmech dozvěděli více. Mnohá témata zmíněná v naší knize pokrývá monumentální dílo Introduction to Algorithms [2] od Cormena a spol. Učebnice Algorithms [3] od Dasgupty a spol. je méně encyklopedická, ale daleko bližší našemu způsobu uvažování – je psaná neformálně, a přitom přesně. Podobně kniha Algorithm Design [6] od Jona Kleinberga a Évy Tardos, která se výrazně více věnuje randomizovaným algoritmům a partiím na pomezí teorie složitosti. Zajímavé implementační triky se lze naučit z knihy Competitive Programmer's Handbook [7] od Antti Laaksonena.

Českých knih o algoritmech existuje pomálu. Pomineme-li překlady, jsou první a donedávna i poslední učebnicí algoritmizace Algoritmy a programovací techniky [11] od Töpfera. Oproti naší knize se daleko více věnují programátorskému řemeslu a výklad algoritmů ilustrují detailní implementací v Pascalu.

Kdo se chce věnovat řešení algoritmických úloh, najde rozsáhlý archiv na webových stránkách *http://ksp.mff.cuni.cz/* Korespondenčního semináře z programování MFF UK a *http://mo.mff.cuni.cz/* Matematické olympiády kategorie P. Dalším zajímavým zdrojem úloh je Skienův The Algorithm Design Manual [10].

Partie kombinatoriky a teorie grafů používané při analýze algoritmů pokrývají vynikající Kapitoly z diskrétní matematiky [9] od Matouška a Nešetřila. Aplikacemi grafů v informatice se zabývá také kniha Jiřího Demela [4]. Pokročilejší kombinatorice a asymptotické analýze se věnuje Concrete Mathematics [5] od Grahama, Knutha a Patashnika.

Poděkování

Rádi bychom poděkovali kolegům, kteří si obětavě přečetli mnoho pracovních verzí knihy a přispěli svými radami, připomínkami, opravami a cvičeními. Díky patří Tomáši Gavenčiakovi, Janu Hricovi, Radku Huškovi, Vladanu Majerechovi, Jiřímu Matouškovi, Janu Musílkovi, Ondřeji Suchému a Pavlu Töpferovi.

Uvažovat, přednášet a psát o algoritmech nás naučila především léta organizování Korespondenčního semináře z programování a semináře Introduction to problem solving. Z úloh KSP také pochází část našich cvičení. Za inspiraci děkujeme všem minulým i současným organizátorům KSP a IPS, zejména pak Karry Burešové, Meggy Calábkové, Zdeňku Dvořákovi, Ondrovi Hlavatému, Danovi Kráľovi, Martinu Krulišovi, Janu Matějkovi, Michalu

Pokornému, Jirkovi Setničkovi, Milanu Strakovi, Filipu Štědronskému, Michalu Vanerovi a Pavlu Veselému.

Díky patří též studentům, kteří pořizovali zápisy z našich prvních přednášek, sloužící jako inspirace k této knize. Byli to: Kateřina Böhmová, Lucia Banáková, Rudolf Barczi, Peter Bašista, Jakub Břečka, Roman Cinkais, Ján Černý, Michal Demín, Jiří Fajfr, Martin Franců, František Haško, Lukáš Hermann, Ondřej Hoferek, Tomáš Hubík, Josef Chludil, Martin Chytil, Jindřich Ivánek, Karel Jakubec, Petr Jankovský, František Kačmarik, Kamil Kaščák, Matej Klaučo, Pavel Klavík, Tereza Klimošová, Vojtěch Kolomičenko, Michal Kozák, Karel Král, Radoslav Krivák, Vincent Kríž, Vladimír Kudelas, Jiří Kunčar, Martin Kupec, Jiří Machálek, Luboš Magic, Bohdan Maslowski, Štěpán Masojídek, Jakub Melka, Jozef Menda, Petr Musil, Jan Návrat, Gábor Ocsovszky, Martin Petr, Oto Petřík, Martin Polák, Markéta Popelová, Daniel Remiš, Dušan Renát, Pavol Rohár, Miroslav Řezáč, Luděk Slinták, Michal Staša, Pavel Taufer, Ondrej Tichý, Radek Tupec, Vojtěch Tůma, Barbora Urbancová, Michal Vachna, Karel Vandas, Radim Vansa, Jan Volec a Jan Záloha.

Za mapu části Vinoře na obrázku 5.2 vděčíme projektu Openstreetmap. Pro vykreslení jsme použili experimentální mapový renderer Leo.

V neposlední řadě děkujeme našim rodinám, které se smířily s tím, že tátové tráví večery a noci nad knihou. A také Katedře aplikované matematiky MFF UK a Katedře teoretické informatiky FIT ČVUT za příjemné a velmi inspirující pracovní prostředí.

Přejeme vám příjemné čtení

Martin Mareš
Tomáš Valla

Obsah

Oddíly a cvičení označené hvězdičkami obsahují pokročilejší materiál. Při prvním čtení je doporučujeme přeskakovat.

1 Příklady na úvod

1 Příklady na úvod

Tématem této knihy má být návrh a analýza algoritmů. Měli bychom tedy nejdříve říci, co to algoritmus je. Formální definice je překvapivě obtížná. Nejspíš se shodneme na tom, že je to nějaký formální postup, jak něco provést, a že by měl být tak podrobný, aby byl srozumitelný i počítači. Jenže detaily už vůbec nejsou tak zřejmé. Proto s pořádným zavedením pojmů ještě kapitolu počkáme a zatím se podíváme na několik konkrétních příkladů algoritmů.

1.1 Úsek s největším součtem

Náš první příklad se bude týkat posloupností. Máme zadanou nějakou posloupnost celých čísel $x_1, \ldots, x_n$ a chceme v ní nalézt *úsek* (tím myslíme souvislou podposloupnost), jehož součet je největší možný. Takovému úseku budeme říkat *nejbohatší.* Jako výstup nám postačí hodnota součtu, nebude nutné ohlásit přesnou polohu úseku.

Nejprve si rozmyslíme triviální případy: Kdyby se na vstupu nevyskytovalo žádné záporné číslo, má evidentně maximální součet celá vstupní posloupnost. Pokud by naopak byla všechna x_i záporná, nejlepší je odpovědět prázdným úsekem, který má nulový součet; všechny ostatní úseky mají součet záporný.

Obecný případ bude komplikovanější: například v posloupnosti

$$1, -2, 4, 5, -1, -5, 2, 7$$

najdeme dva úseky kladných čísel se součtem 9 (totiž $4, 5$ a $2, 7$), ale dokonce se hodí spojit je přes záporná čísla $-1, -5$ do jediného úseku se součtem 12. Naopak hodnotu -2 se použít nevyplácí, jelikož přes ní je dosažitelná pouze počáteční jednička, takže bychom si o 1 pohoršili.

Nejpřímočařejší možný algoritmus by téměř doslovně kopíroval zadání: Vyzkoušel by všechny možnosti, kde může úsek začínat a končit, pro každou z nich by spočítal součet prvků v úseku a pak našel z těchto součtů maximum.

Algoritmus MaxSoučet1
Vstup: Posloupnost $X = x_1, \ldots, x_n$ uložená v poli

1. $m \leftarrow 0$ ◁ *zatím jsme potkali jen prázdný úsek*
2. Pro $i = 1, \ldots, n$ opakujeme: ◁ *i je začátek úseku*
3. Pro $j = i, \ldots, n$ opakujeme: ◁ *j je konec úseku*
4. $s \leftarrow 0$ ◁ *součet úseku*

5. Pro $k = i, \ldots, j$ opakujeme:
6. $\quad s \leftarrow s + x_k$
7. $m \leftarrow \max(m, s)$

Výstup: Součet m nejbohatšího úseku v X

Pojďme alespoň zhruba odhadnout, jak rychlý tento postup je. Prozkoumáme řádově n^2 dvojic (*začátek*, *konec*) a pro každou z nich strávíme řádově n kroků počítáním součtu. To dohromady dává řádově n^3 kroků, což už pro $n = 1\,000$ budou miliardy. Zkusme přijít na rychlejší způsob.

Podívejme se, čím náš první algoritmus tráví nejvíce času. Jistě počítáním součtů. Například sčítá jak úsek $x_i, \ldots, x_j$, tak $x_i, \ldots, x_{j+1}$, aniž by využil toho, že druhý součet je prostě o x_{j+1} vyšší než ten první. Nabízí se zvolit pevný začátek úseku i a pak zkoušet všechny možné konce j od nejlevějšího k nejpravějšímu. Každý další součet pak dovedeme triviálně spočítat z předchozího. Pro jedno i tedy provedeme řádově n kroků, celkově pak řádově n^2.

Algoritmus MaxSoučet2

Vstup: Posloupnost $X = x_1, \ldots, x_n$ uložená v poli

1. $m \leftarrow 0$ ◁ *zatím jsme potkali jen prázdný úsek*
2. Pro $i = 1, \ldots, n$ opakujeme: ◁ *i je začátek úseku*
3. $\quad s \leftarrow 0$ ◁ *součet úseku*
4. $\quad$ Pro $j = i, \ldots, n$ opakujeme: ◁ *j je konec úseku*
5. $\quad\quad s \leftarrow s + x_j$
6. $\quad\quad m \leftarrow \max(m, s)$

Výstup: Součet m nejbohatšího úseku v X

Myšlenka průběžného přepočítávání se ale dá využít i lépe, totiž na celou úlohu. Uvažujme, jak se změní výsledek, když ke vstupu $x_1, \ldots, x_n$ přidáme ještě x_{n+1}. Všechny úseky původního vstupu zůstanou zachovány a navíc k nim přibudou nové úseky tvaru $x_i, \ldots, x_{n+1}$. Stačí tedy ověřit, zda součet některého z nových úseků nepřekročil dosavadní maximum, čili porovnat maximum se součtem nejbohatšího *koncového* úseku nové posloupnosti.

Nejbohatší koncový úsek také neumíme najít v konstantním čase, ale pojďme tutéž myšlenku použít ještě jednou. Jak se změní koncové úseky po přidání x_n? Všem stávajícím koncovým úsekům stoupne součet o x_n a navíc vznikne nový koncový úsek obsahující samotné x_n. Maximální součet je proto roven buď předchozímu maximálnímu součtu plus x_n, nebo samotnému x_n – podle toho, co je větší.

Označíme-li si tedy k maximální součet koncového úseku, přidáním nového prvku se tato hodnota změní na $\max(k + x_n, x_n) = x_n + \max(k, 0)$. Jinými slovy: počítáme průběžné součty, ale pokud součet klesne pod nulu, tak ho vynulujeme. Hledaný maximální součet m je pak maximem ze všech průběžných součtů. Tímto principem se řídí náš třetí algoritmus:

Algoritmus MaxSoučet3
Vstup: Posloupnost $X = x_1, \ldots, x_n$ uložená v poli

1. $m \leftarrow 0$ ◁ *prázdný úsek je tu vždy*
2. $k \leftarrow 0$ ◁ *maximální součet koncového úseku*
3. Pro i od 1 do n opakujeme:
4. $\quad k \leftarrow \max(k, 0) + x_i$
5. $\quad m \leftarrow \max(m, k)$

Výstup: Součet m nejbohatšího úseku v X

V každém průchodu cyklem potřebujeme na přepočítání proměnných k a m pouze konstantně mnoho operací. Celkem jich tedy algoritmus provede řádově n, tedy lineárně s velikostí vstupu. Hodnoty ze vstupu navíc potřebuje jen jednou, takže je může číst postupně a vystačí si tudíž s konstantním množstvím paměti.

Dodejme ještě, že úvaha typu „jak se změní výstup, když na konec vstupu přidáme další prvek?“ je poměrně častá. Vysloužila si proto zvláštní jméno, algoritmům tohoto druhu se říká *inkrementální*. Ještě se s nimi několikrát potkáme.

Cvičení

1. Upravte algoritmus MaxSoučet3, aby oznámil nejen maximální součet, ale také polohu příslušného úseku.

2. Na vstupu je text složený z písmen české abecedy a mezer. Vymyslete algoritmus, který najde nejdelší úsek textu, v němž se žádné písmeno neopakuje.

3. Najděte v českém textu nejkratší úsek, který obsahuje všechna písmena abecedy. Malá a velká písmena nerozlišujte.

4.* Úsek posloupnosti je *k-hladký* (pro $k \geq 0$), pokud se každé dva jeho prvky liší nejvýše o k. Popište co nejefektivnější algoritmus pro hledání nejdelšího k-hladkého úseku.

5. Jak spočítat kombinační číslo $\binom{n}{k}$? Výpočtu přímo podle definice brání potenciálně obrovské mezivýsledky (až $n!$), které se nevejdou do celočíselné proměnné. Navrhněte algoritmus, který si vystačí s čísly omezenými n-násobkem výsledku.

1.2 Binární vyhledávání

Jak se hledá slovo ve slovníku? Jistě můžeme slovníkem listovat stránku po stránce a pečlivě zkoumat slovo po slovu. Jsme-li dostatečně trpěliví, hledané slovo nakonec najdeme, nebo slovník skončí a můžeme si být jistí, že slovo neobsahoval.

Listování slovníkem může být dobrá zábava na dlouhé zimní večery (nebo spíš na celou polární noc), ale obvykle hledáme jinak: otevřeme slovník někde uprostřed, podíváme se, jak blízko jsme k hledanému slovu, a na základě toho nadále aplikujeme stejný postup buďto v levé, anebo pravé části rozevřeného slovníku.

Nyní se tento postup pokusíme popsat precizně. Získáme tak algoritmus, kterému se říká *binární vyhledávání* nebo také *hledání půlením intervalu.* Na vstupu dostaneme nějakou uspořádanou posloupnost $x_1 \leq x_2 \leq \ldots \leq x_n$ a hledaný prvek y.

Postupujeme takto: pamatujeme si interval $x_\ell, \ldots, x_r$, ve kterém se prvek může nacházet. Na počátku je $\ell = 1$ a $r = n$. V každém kroku vybereme prvek ležící uprostřed (nebo přibližně uprostřed, pokud je prvků sudý počet). Ten bude sloužit jako mezník oddělující levou polovinu od pravé. Pokud se mezník rovná hledanému y, můžeme hned úspěšně skončit. Pokud je menší než y, znamená to, že y se může nacházet jen napravo od něj – všechny prvky nalevo jsou menší než mezník, tím pádem i menší než y. A pokud je naopak mezník větší než y, víme, že y se může nacházet pouze v levé polovině.

Postupně tedy interval $[\ell, r]$ zmenšujeme na polovinu, čtvrtinu, atd., až se dostaneme do stavu, kdy prohledávaný úsek pole má velikost jednoho prvku, nebo je dokonce prázdný. Pak už se snadno přesvědčíme, zda jsme hledaný prvek našli.

V pseudokódu náš algoritmus vypadá následovně.

Algoritmus BinSearch (hledání půlením intervalu)
Vstup: Uspořádaná posloupnost $x_1 \leq \ldots \leq x_n$, hledaný prvek y

1. $\ell \leftarrow 1$, $r \leftarrow n$ ◁ $x_\ell, \ldots, x_r$ *tvoří prohledávaný úsek pole*
2. Dokud je $\ell \leq r$:
3. $\quad s \leftarrow \lfloor (\ell + r)/2 \rfloor$ ◁ *střed prohledávaného úseku*
4. $\quad$ Pokud je $y = x_s$: vrátíme s a skončíme.
5. $\quad$ Pokud je $y > x_s$:
6. $\quad\quad \ell \leftarrow s + 1$ ◁ *přesouváme se napravo*
7. $\quad$ Jinak:
8. $\quad\quad r \leftarrow s - 1$ ◁ *přesouváme se nalevo*
9. Vrátíme 0. ◁ *nenašli jsme*

Výstup: Index hledaného prvku, případně 0, pokud prvek v poli není

Pokusme se poctivě dokázat, že algoritmus funguje. Především nahlédneme, že výpočet se vždy zastaví: v každém průchodu cyklem zmenšíme prohledávaný úsek alespoň o 1. Korektnost pak plyne z toho, že kdykoliv oblast zmenšíme, odstraníme z ní jen prvky, které jsou zaručeně různé od y. Jakmile tedy algoritmus skončí, buďto jsme y našli, nebo jsme naopak vyloučili všechny možnosti, kde by mohlo být.

Nyní ukážeme, že binární vyhledávání je mnohem rychlejší než probrání všech prvků.

Věta: Při hledání v posloupnosti délky n provede algoritmus BinSearch nejvýše $\log_2 n$ průchodů cyklem.

Důkaz: Stačí nahlédnout, že v každém průchodu cyklem se velikost prohledávaného úseku zmenší alespoň dvakrát. Proto po k průchodech úsek obsahuje nejvýše $n/2^k$ prvků, takže pro $k > \log_2 n$ je úsek nutně prázdný. □

Na závěr dodejme, že prvky naší posloupnosti vůbec nemusí být čísla: stačí, aby to byly libovolné objekty, které jsme schopni mezi sebou porovnávat. Třeba slova ve slovníku. V oddílu 3.3 navíc dokážeme, že logaritmický počet porovnání je nejlepší možný.

Dvojice se zadaným součtem

Podívejme se ještě na jeden příbuzný problém. Opět dostaneme na vstupu nějakou uspořádanou posloupnost $x_1 \leq x_2 \leq \ldots \leq x_n$ a číslo s. Tentokrát ovšem nehledáme jeden prvek, nýbrž dva (ne nutně různé), jejichž součet je s.

Řešení „hrubou silou“ by zkoušelo sečíst všechny dvojice $x_i + x_j$, ale těch je řádově n^2. Pokud ovšem zvolíme nějaké x_i, víme, že x_j musí být rovno $s - x_i$. Můžeme tedy vyzkoušet všechna x_i a pokaždé půlením intervalu hledat $s - x_i$. Každé vyhledávání spotřebuje řádově $\log_2 n$ kroků, celkově jich tedy bude řádově $n \cdot \log_2 n$.

To je mnohem lepší, ale ještě ne optimální. Představme si, že k x_1 hledáme $s - x_1$. Tentokrát ale nebudeme hledat binárně, nýbrž pěkně prvek po prvku od konce pole. Dokud jsou prvky větší, přeskakujeme je. Jakmile narazíme na prvek menší, víme, že už se můžeme zastavit, protože dál už budou jen samé menší.

Pozici, kde jsme skončili, si zapamatujeme. Máme tedy nějaké j takové, že $x_j < s - x_1 < x_{j+1}$. (Pokud protestujete, že x_{j+1} může ležet mimo posloupnost, představte si za koncem posloupnosti ještě $+\infty$.)

Nyní přejdeme na x_2 a hledáme $s - x_2$. Jelikož $x_2 \geq x_1$, musí být $s - x_2 \leq s - x_1$. Všechna čísla, která byla větší než $s - x_1$ jsou tedy také větší než $s - x_2$, takže v nich nemá smysl hledat znovu. Proto můžeme pokračovat od zapamatované pozice j dále doleva. Pak si zase zapamatujeme, kde jsme skončili, což se bude hodit pro x_3, a tak dále.

Existuje hezčí způsob, jak formulovat totéž. Říká se mu *metoda dvou jezdců*. Máme dva indexy: levý a pravý. Levý index i popisuje, které x_i zrovna zkoušíme jako první člen dvojice: začíná na pozici 1 a pohybuje se doprava. Pravý index j ukazuje na místo, kde jsme se zastavili při hledaní $s - x_i$: začíná na pozici n a postupuje doleva.

Kdykoliv je $x_j > s - x_i$, posuneme j doleva (pokračujeme v hledání $s - x_i$). Je-li naopak $x_j < s - x_i$, posuneme i doprava ($s - x_i$ se v posloupnosti určitě nenachází, zkoušíme další x_i). Takto pokračujeme, dokud buďto neobjevíme hledanou dvojici, nebo se jezdci nesetkají – tehdy dvojice zaručeně neexistuje.

Algoritmus DvojiceSeSoučtem
Vstup: Uspořádaná posloupnost $x_1 \leq \ldots \leq x_n$, hledaný součet s

1. $i \leftarrow 1$, $j \leftarrow n$
2. Dokud $i \leq j$:
3. Je-li $x_i + x_j = s$:
4. Vrátíme jako výsledek dvojici (i, j).
5. Jinak je-li $x_i + x_j < s$: ◁ *totéž jako* $x_i < s - x_j$
6. $i \leftarrow i + 1$
7. Jinak:
8. $j \leftarrow j - 1$
9. Ohlásíme neúspěch.

Výstup: Indexy i a j, pro něž je $x_i + x_j = s$, nebo neúspěch

Snadno nahlédneme, že cyklus proběhne nejvýše $2n$-krát. Pokaždé se totiž pohne jeden z jezdců, ale každý z nich může urazit nejvýše n kroků, než vyjede ven.

Překonali jsme tedy rychlost opakovaného binárního vyhledávání. Povedlo se nám to díky tomu, že mezi hledanými prvky existoval nějaký vztah: konkrétně každý prvek byl menší nebo roven předchozímu.

Cvičení

1. Rozmyslete si, jak se bude chovat algoritmus binárního vyhledávání, pokud se bude hledaný prvek v posloupnosti nacházet vícekrát. Algoritmus upravte, aby vždy vracel první výskyt hledaného prvku (ne jen libovolný).

2. Upravte binární vyhledávání, aby v případě, kdy hledaný prvek v posloupnosti není, nahlásilo nejbližší větší prvek.

3.* *Nekonečná verze:* Popište algoritmus, který v nekonečné posloupnosti $x_1 < x_2 < \ldots$ najde pozici i takovou, že $x_i \leq y < x_{i+1}$. Počet kroků hledání by neměl přesáhnout řádově $\log_2 i$.

4. *Lokální minimum:* Je dána posloupnost $+\infty = x_0, x_1, \ldots, x_n, x_{n+1} = +\infty$. O prvku x_i řekneme, že je lokálním minimem, pokud $x_{i-1} \geq x_i \leq x_{i+1}$. Navrhněte co nejrychlejší algoritmus, který nějaké lokální minimum najde.

5. *Součet úseku:* Je dána posloupnost $x_1, \ldots, x_n$ kladných čísel a číslo s. Hledáme i a j taková, že $x_i + \ldots + x_j = s$. Navrhněte co nejefektivnější algoritmus.

6.* Jak se změní úloha z předchozího cvičení, pokud povolíme i záporná čísla?

7. *Implicitní vstup:* Posloupnost, v níž binárně hledáme, nemusí být nutně celá uložená v paměti. Stačí, když se tak dokážeme dostatečně přesvědčivě tvářit: kdykoliv se algoritmus zeptá na hodnotu nějakého prvku, rychle ho vyrobíme. Zkuste tímto způsobem spočítat *celočíselnou odmocninu* z čísla x. To je největší y takové, že $y^2 \leq x$.

8. *První díra:* Na vstupu jsme dostali rostoucí posloupnost přirozených čísel. Chceme najít nejmenší přirozené číslo, které v ní chybí. Vymyslete, jak k tomu přesvědčit binární vyhledávání.

9.* Opět hledáme nejmenší chybějící číslo, ale tentokrát na vstupu dostaneme neuspořádanou posloupnost navzájem různých přirozených čísel. Posloupnost nesmíme měnit a kromě ní máme k dispozici jenom konstantně mnoho paměti.

10. *Monotónní predikáty:* Na předchozích několik cvičení se můžeme dívat trochu obecněji. Mějme nějakou vlastnost φ, kterou všechna přirozená čísla od 0 do nějaké hranice k mají a žádná větší nemají. Popište, jak binárním vyhledáváním zjistit, kde se nachází tato hranice.

11. *Rovnoměrná data:* Mějme pole délky n. Na každé pozici se může vyskytovat libovolné celé číslo z rozsahu 1 až k. Čísla vybíráme rovnoměrně náhodně (všechny hodnoty mají stejnou pravděpodobnost). Následně pole setřídíme a budeme v něm chtít vyhledávat. Zkuste upravit binární vyhledávání, aby pro tyto vstupy fungovalo v průměru rychleji.

12.* Kolik porovnání provede takový algoritmus v průměru?

13.* Může se stát, že výše uvedený algoritmus nedostane pěkná data. Můžeme mu nějak pomoci, aby nebyl ani v takovém případě o mnoho horší než binární vyhledávání?

1.3 Euklidův algoritmus

Pro další příklad se vypravíme do starověké Alexandrie. Tam žil ve 3. století před naším letopočtem filosof Euklides (Ευκλείδης) a stvořil jeden z nejstarších algoritmů.[1] Ten slouží k výpočtu největšího společného dělitele a používá se dodnes.

Značení: *Největšího společného dělitele* celých kladných čísel x a y budeme značit $\gcd(x, y)$ podle anglického Greatest Common Divisor.

Nejprve si všimneme několika zajímavých vlastností funkce gcd.

Lemma G: Pro všechna celá kladná čísla x a y platí:

1. $\gcd(x, x) = x$,
2. $\gcd(x, y) = \gcd(y, x)$,
3. $\gcd(x, y) = \gcd(x - y, y)$ pro $x > y$.

Důkaz: První dvě vlastnosti jsou zřejmé z definice. Třetí dokážeme v silnější podobě: ukážeme, že dvojice (x, y) a $(x - y, y)$ sdílejí množinu všech společných dělitelů, tedy i největšího z nich.

Pokud nějaké d je společným dělitelem čísel x a y, musí platit $x = dx'$ a $y = dy'$ pro vhodné x' a y'. Nyní stačí rozepsat $x - y = dx' - dy' = d(x' - y')$ a hned je jasné, že d dělí i $x - y$. Naopak pokud d dělí jak $x - y$, tak y, musí existovat čísla t' a y' taková, že $x - y = dt'$ a $y = dy'$. Zapíšeme tedy x jako $(x - y) + y$, což je rovno $dt' + dy' = d(t' + y')$, a to je dělitelné d. □

Díky lemmatu můžeme gcd počítat tak, že opakovaně odečítáme menší číslo od většího. Jakmile se obě čísla vyrovnají, jsou rovna největšímu společnému děliteli. Algoritmus nyní zapíšeme v pseudokódu.

Algoritmus OdčítacíEuklides
Vstup: Celá kladná čísla x a y

1. $a \leftarrow x$, $b \leftarrow y$
2. Dokud $a \neq b$, opakujeme:
3. Pokud $a > b$:
4. $a \leftarrow a - b$
5. Jinak:

[1] Tehdy se tomu ovšem tak neříkalo. Pojem *algoritmu* je novodobý, byl zaveden až začátkem 20. století při studiu „mechanické“ řešitelnosti matematických úloh. Název je poctou perskému matematikovi al-Chorézmímu, jenž žil cca 1100 let po Euklidovi a v pozdějších překladech jeho díla mu jméno polatinštili na Algoritmi.

6. $b \leftarrow b - a$

Výstup: Největší společný dělitel $a = \gcd(x, y)$

Nyní bychom měli dokázat, že algoritmus funguje. Důkaz rozdělíme na dvě části:

Lemma Z: Algoritmus se vždy zastaví.

Důkaz: Sledujme, jak se vyvíjí součet $a + b$. Na počátku výpočtu je roven $x + y$, každým průchodem cyklem se sníží alespoň o 1. Přitom zůstává stále nezáporný, takže průchodů nastane nejvýše $x + y$. □

Lemma S: Pokud se algoritmus zastaví, vydá správný výsledek.

Důkaz: Dokážeme následující *invariant,* neboli tvrzení, které platí po celou dobu výpočtu:

Invariant: $\gcd(a, b) = \gcd(x, y)$.

Důkaz: Obvyklý způsob důkazu invariantů je indukce podle počtu kroků výpočtu. Na počátku je $a = x$ a $b = y$, takže invariant jistě platí. V každém průchodu cyklem se pak díky vlastnostem 2 a 3 z lemmatu **G** platnost invariantu zachovává. □

Z invariantu plyne, že na konci výpočtu je $\gcd(a, a) = \gcd(x, y)$. Zároveň díky vlastnosti 1 z lemmatu **G** platí $\gcd(a, a) = a$. □

Víme tedy, že algoritmus je funkční. To bohužel neznamená, že je použitelný: například pro $x = 1\,000\,000$ a $y = 2$ vytrvale odčítá y od x, až po 499 999 krocích vítězoslavně ohlásí, že největší společný dělitel je roven 2.

Stačí si ale všimnout, že opakovaným odčítáním b od a dostaneme zbytek po dělení čísla a číslem b. Jen si musíme dát pozor, že pro a dělitelné b se zastavíme až na nule. To odpovídá tomu, že algoritmus provede ještě jedno odečtení navíc, takže skončí, až když se jedno z čísel vynuluje. Nahrazením odčítání za dělení se zbytkem získáme následující algoritmus. Když se v současnosti hovoří o Euklidově algoritmu, obvykle se myslí tento.

Algoritmus Euklides

Vstup: Celá kladná čísla x a y

1. $a \leftarrow x$, $b \leftarrow y$
2. Opakujeme:
3. Pokud $a < b$, prohodíme a s b.
4. Pokud $b = 0$, vyskočíme z cyklu.
5. $a \leftarrow a \bmod b$ ◁ *zbytek po dělení*

Výstup: Největší společný dělitel $a = \gcd(x, y)$

Správnost je zřejmá: výpočet nového algoritmu odpovídá výpočtu algoritmu předchozího, jen občas provedeme několik původních kroků najednou. Zajímavé ovšem je, že na první pohled nenápadnou úpravou jsme algoritmus podstatně zrychlili:

Lemma R: Euklidův algoritmus provede nejvýše $\log_2 x + \log_2 y + 1$ průchodů cyklem.

Důkaz: Vývoj výpočtu budeme sledovat prostřednictvím součinu ab:

> **Tvrzení:** Součin ab po každém průchodu cyklem klesne alespoň dvakrát.
>
> *Důkaz:* Kroky 3 a 4 součin ab nemění. Ve zbývajícím kroku 5 platí $a \geq b$ a b se evidentně nezmění. Ukážeme, že a klesne alespoň dvakrát, takže ab také. Rozebereme dva případy:
>
> - $b \leq a/2$. Tehdy platí $a \bmod b < b \leq a/2$.
> - $b > a/2$. Pak je $a \bmod b = a - b < a - (a/2) = a/2$. □

Na počátku výpočtu je $ab = xy$ a díky právě dokázanému tvrzení po k průchodech cyklem musí platit $ab \leq xy/2^k$. Kromě posledního neúplného průchodu cyklem ovšem ab nikdy neklesne pod 1, takže k může být nejvýše $\log_2 xy = \log_2 x + \log_2 y$. □

Shrnutím všeho, co jsme o algoritmu zjistili, získáme následující větu:

Věta: Euklidův algoritmus vypočte největšího společného dělitele čísel x a y. Provede přitom nejvýše $c \cdot (\log_2 x + \log_2 y + 1)$ aritmetických operací, kde c je konstanta.

Cvičení

1. Největšího společného dělitele bychom také mohli počítat pomocí prvočíselného rozkladu čísel x a y. Rozmyslete si, jak by se to dělalo a proč je to pro velká čísla velmi nepraktické.

2. V kroku 3 algoritmu EUKLIDES není potřeba porovnávat. Nahlédněte, že pokud bychom a s b prohodili pokaždé, vyjde také správný výsledek, jen nás to bude v nejhorším případě stát o jeden průchod cyklem navíc.

3. Dokažte, že počet průchodů cyklem je nejvýše $2 \log_2 \min(x, y) + 2$.

4. Pro každé x a y existují celá čísla α a β taková, že $\gcd(x, y) = \alpha x + \beta y$. Těmto číslům se říká *Bézoutovy koeficienty.* Upravte Euklidův algoritmus, aby je vypočetl.

5. Pomocí předchozího cvičení můžeme řešit *lineární kongruence.* Pro daná a a n chceme najít x, aby platilo $ax \bmod n = 1$. To znamená, že ax a 1 se liší o násobek n, tedy $ax + ny = 1$ pro nějaké y. Pokud je $\gcd(a, n) = 1$, pak x a y jsou Bézoutovy koeficienty, které to dosvědčí. Je-li $\gcd(a, n) \neq 1$, nemůže mít rovnice řešení, protože levá strana

je vždy dělitelná tímto gcd, zatímco pravá nikoliv. Jak najít řešení obecnější rovnice $ax \bmod n = b$?

6. Nabízí se otázka, není-li logaritmický odhad počtu operací z naší věty příliš velkorysý. Abyste na ni odpověděli, najděte funkci f, která roste nejvýše exponenciálně a při výpočtu $\gcd(f(n), f(n+1))$ nastane právě n průchodů cyklem.

7. Binární algoritmus na výpočet gcd funguje takto: Pokud x i y jsou sudá, pak $\gcd(x,y) = 2\gcd(x/2, y/2)$. Je-li x sudé a y liché, pak $\gcd(x,y) = \gcd(x/2, y)$. Jsou-li obě lichá, odečteme menší od většího. Zastavíme se, až bude $x = y$. Dokažte, že tento algoritmus funguje a že provede nejvýše $c \cdot (\log_2 x + \log_2 y)$ kroků pro vhodnou konstantu c.

8.* Mějme permutaci π na množině $\{1, \ldots, n\}$. Definujme její mocninu následovně: $\pi^0(x) = x$, $\pi^{i+1}(x) = \pi(\pi^i(x))$. Najděte nejmenší $k > 0$ takové, že $\pi^k = \pi^0$.

1.4 Fibonacciho čísla a rychlé umocňování

Dovolíme si ještě jednu historickou exkurzi, tentokrát do Pisy, kde na začátku 13. století žil jistý Leonardo řečený Fibonacci.[2] Příštím generacím zanechal zejména svou posloupnost.

Definice: *Fibonacciho posloupnost* $F_0, F_1, F_2, \ldots$ je definována následovně:

$$F_0 = 0, \quad F_1 = 1, \quad F_{n+2} = F_{n+1} + F_n.$$

Příklad: Prvních 11 Fibonacciho čísel zní 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55.

Pokud chceme spočítat F_n, můžeme samozřejmě vyjít z definice a postupně sestrojit prvních n členů posloupnosti. To nicméně vyžaduje řádově n operací, takže se nabízí otázka, zda to lze rychleji. V moudrých knihách nalezneme následující větu:

Věta (kouzelná formule): Pro každé $n \geq 0$ platí:

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

Důkaz: Laskavý, nicméně trpělivý čtenář jej provede indukcí podle n. Jak na kouzelnou formuli přijít, naznačíme v cvičení 3. □

Dobrá, ale jak nám formule pomůže, když pro výpočet n-té mocniny potřebujeme $n-1$ násobení? Inu nepotřebujeme, následující algoritmus to zvládne rychleji:

[2] Což je zkratka z „filius Bonaccii“, tedy „syn Bonacciho“.

Algoritmus MOCNINA
Vstup: Reálné číslo x, celé kladné n

1. Pokud $n = 0$, vrátíme výsledek 1.
2. $t \leftarrow \text{MOCNINA}(x, \lfloor n/2 \rfloor)$
3. Pokud n je sudé, vrátíme $t \cdot t$.
4. Jinak vrátíme $t \cdot t \cdot x$.

Výstup: x^n

Lemma: Algoritmus MOCNINA vypočte x^n pomocí nejvýše $2 \log_2 n + 2$ násobení.

Důkaz: Správnost je evidentní z toho, že $x^{2k} = (x^k)^2$ a $x^{2k+1} = x^{2k} \cdot x$. Co se počtu operací týče: Každé rekurzivní volání redukuje n alespoň dvakrát, takže po nejvýše $\log_2 n$ voláních musíme dostat jedničku a po jednom dalším nulu. Hloubka rekurze je tedy $\log_2 n + 1$ a na každé úrovni rekurze spotřebujeme nejvýše 2 násobení. □

To dává elegantní algoritmus pro výpočet F_n pomocí řádově $\log_2 n$ operací. Jen je bohužel pro praktické počítání nepoužitelný: Zlatý řez $(1 + \sqrt{5})/2 \doteq 1.618\,034$ je iracionální a pro vysoké hodnoty n bychom ho potřebovali znát velice přesně. To neumíme dostatečně rychle. Zkusíme to tedy menší oklikou.

Po Fibonacciho posloupnosti budeme posouvat okénkem, skrz které budou vidět právě dvě čísla. Pokud zrovna vidíme čísla F_n, F_{n+1}, v dalším kroku uvidíme $F_{n+1}, F_{n+2} = F_{n+1} + F_n$. To znamená, že posunutí provede s okénkem nějakou lineární transformaci a každá taková jde zapsat jako násobení maticí. Dostaneme:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_{n+2} \end{pmatrix}.$$

Levou matici označíme $\mathcal{F}$ a nahlédneme, že násobení okénka n-tou mocninou této matice musí okénko posouvat o n pozic. Tudíž platí:

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

Nyní stačí využít toho, že násobení matic je asociativní. Proto můžeme n-tou mocninu matice vypočítat obdobou algoritmu MOCNINA a vystačíme si s řádově $\log n$ maticovými násobeními. Jelikož pracujeme s maticemi konstantní velikosti, obnáší každé násobení matic jen konstantní počet operací s čísly. Všechny matice jsou příjemně celočíselné.

Proto platí:

Věta: n-té Fibonacciho číslo lze spočítat pomocí řádově $\log n$ celočíselných aritmetických operací.

Cvičení

1. Naprogramujte funkci Mocnina nerekurzivně. Může pomoci převést exponent do dvojkové soustavy.

2. Uvažujme obecnou *lineární rekurenci řádu k:* $A_0, \ldots, A_{k-1}$ jsou dána pevně, $A_{n+k} = \alpha_1 A_{n+k-1} + \alpha_2 A_{n+k-2} + \ldots + \alpha_k A_n$ pro konstanty $\alpha_1, \ldots, \alpha_k$. Vymyslete efektivní algoritmus na výpočet A_n, nejlépe pomocí řádově $\log_2 n$ operací.

3.* Jak odvodit kouzelnou formuli: Uvažujme množinu všech posloupností, které splňují rekurentní vztah $A_{n+2} = A_{n+1} + A_n$, ale mohou se lišit hodnotami A_0 a A_1. Tato množina tvoří vektorový prostor, přičemž posloupnosti sčítáme a násobíme skalárem po složkách a roli nulového vektoru hraje posloupnost samých nul. Ukažte, že tento prostor má dimenzi 2 a sestrojte jeho bázi v podobě exponenciálních posloupností tvaru $A_n = \alpha^n$. Fibonacciho posloupnost pak zapište jako lineární kombinaci prvků této báze.

4. Dokažte, že $F_{n+2} \geq \varphi^n$, kde $\varphi = (1 + \sqrt{5})/2 \doteq 1.618\,034$.

5.* Algoritmy založené na explicitní formuli pro F_n jsme odmítli, protože potřebovaly počítat s iracionálními čísly. To bylo poněkud ukvapené. Dokažte, že čísla tvaru $a + b\sqrt{5}$, kde $a, b \in \mathbb{Q}$ jsou uzavřená na sčítání, odčítání, násobení i dělení. K výpočtu formule si tedy vystačíme s racionálními čísly, dokonce pouze typu $p/2^q$, kde p a q jsou celá. Odvoďte z toho jiný logaritmický algoritmus.

2 Časová a prostorová složitost

2 Časová a prostorová složitost

V minulé kapitole jsme zkusili navrhnout algoritmy pro několik jednoduchých úloh. Zjistili jsme přitom, že pro každou úlohu existuje algoritmů více. Všechny fungují, ale jak poznat, který z nich je nejlepší? A co vlastně znamenají pojmy „lepší“ a „horší“? Kritérií kvality může být mnoho. Nás v této knize budou zajímat časové a paměťové nároky programu, tzn. rychlost výpočtu a velikost potřebné operační paměti počítače.

Jako první srovnávací metoda nás nejspíš napadne srovnávané algoritmy naprogramovat v nějakém programovacím jazyce, spustit je na větší množině testovacích dat a měřit se stopkami v ruce (nebo alespoň s těmi zabudovanými do operačního systému), který z nich je lepší. Takový postup se skutečně v praxi používá, z teoretického hlediska je však nevhodný. Kdybychom chtěli svým kolegům popsat vlastnosti určitého algoritmu, jen stěží nám postačí „na mém stroji doběhl do hodiny“. A jak bude fungovat na jiném stroji, s odlišnou architekturou, naprogramovaný v jiném jazyce, pod jiným operačním systémem, pro jinou sadu vstupních dat?

V této kapitole vybudujeme způsob, jak měřit dobu běhu algoritmu a jeho paměťové nároky nezávisle na technických podrobnostech – konkrétním stroji, jazyku, operačním systému. Těmto mírám budeme říkat *časová a prostorová složitost* algoritmu.

2.1 Jak fungují počítače uvnitř

Definice pojmu „počítač“ není samozřejmá. V současnosti i v historii bychom jistě našli spoustu strojů, kterým by se tak dalo říkat. Co mají společného? My se přidržíme všeobecně uznávané definice, kterou v roce 1946 vyslovil vynikající matematik John von Neumann.

Von neumannovský počítač se skládá z pěti funkčních jednotek:

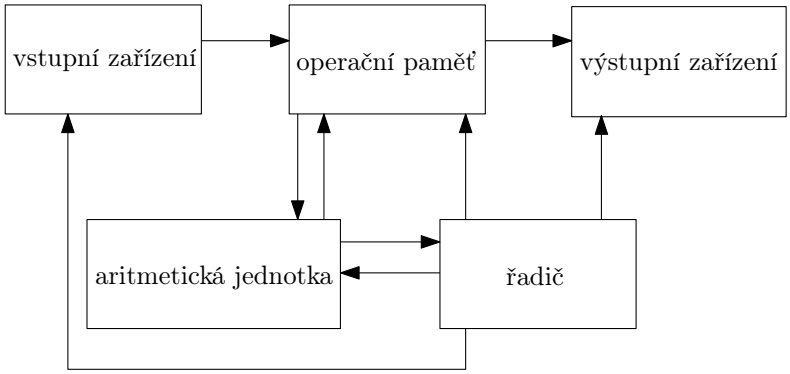
- *řídicí jednotka (řadič)* – koordinuje činnost ostatních jednotek a určuje, co mají v kterém okamžiku dělat,
- *aritmeticko-logická jednotka (ALU)* – provádí numerické výpočty, vyhodnocuje podmínky, ...,
- *operační paměť* – uchovává data a program,
- *vstupní zařízení* – zařízení, odkud se do počítače dostávají data ke zpracování,
- *výstupní zařízení* – do tohoto zařízení zapisuje počítač výsledky své činnosti.

Struktura počítače je nezávislá na zpracovávaných problémech, na řešení problému se musí zvenčí zavést návod na zpracování (program) a musí se uložit do paměti; bez tohoto programu není stroj schopen práce.

Programy, data, mezivýsledky a konečné výsledky se ukládají do téže paměti.[1] Paměť je rozdělená na stejně velké *buňky,* které jsou průběžně očíslované; pomocí čísla buňky (*adresy*) se dá přečíst nebo změnit obsah buňky. V každé buňce je uloženo číslo. Všechna data i instrukce programu kódujeme pomocí čísel. Kódování a dekódování zabezpečují vhodné logické obvody v řídicí jednotce.

Po sobě jdoucí instrukce programu se nacházejí v po sobě jdoucích paměťových buňkách. Instrukcemi skoku se dá odklonit od zpracování instrukcí v uloženém pořadí. Existují následující typy instrukcí:

- aritmetické instrukce (sčítání, násobení, ukládání konstant, ...)
- logické instrukce (porovnání, AND, OR, ...)
- instrukce přenosu (z paměti do ALU a opačně, na vstup a výstup)
- podmíněné a nepodmíněné skoky
- ostatní (čekání, zastavení, ...)



Obrázek 2.1: Schéma von Neumannova počítače (po šipkách tečou data a povely)

Přesné specifikaci počítače, tedy způsobu vzájemného propojení jednotek, jejich komunikace a programování, popisu instrukční sady, říkáme *architektura.* Nezabíhejme do detailů fungování běžných osobních počítačů, čili popisu jejich architektury. V každém z nich se však jednotky chovají tak, jak popsal von Neumann. Z našeho hlediska bude nejdůležitější podívat se co se děje, pokud na počítači vytvoříme a spustíme program.

[1] Tím se liší von Neumannova architektura od architektury zvané *harvardská,* jež program a data důsledně odděluje. Výhodou von Neumannova počítače je kromě jednoduchosti i to, že program může modifikovat sám sebe. Výhodou harvardské pak možnost přistupovat k programu i datům současně.

Algoritmus zapíšeme obvykle ve formě vyššího programovacího jazyka. Zde je příklad v jazyce C.

```
#include <stdio.h>

int main(void)
{
   static char s[] = "Hello world\n";
   int i, n = sizeof(s);
   for (i = 0; i < n; i++)
      putchar(s[i]);
   return 0;
}
```

Aby řídicí jednotka mohla program provést, musíme nejdříve spustit *kompilátor* neboli *překladač*. To je nějaký jiný program, který náš program z jazyka C přeloží do takzvaného *strojového kódu*. Tedy do posloupnosti jednoduchých instrukcí kódovaných pomocí čísel, jimž už počítač přímo rozumí. Na rozdíl od původního příkladu, který na všech počítačích s překladačem jazyka C bude vypadat stejně, strojový kód se bude lišit architekturu od architektury, operační systém od operačního systému, dokonce překladač od překladače.

Ukážeme příklad úseku strojového kódu, který vznikl po překladu našeho příkladu v operačním systému Linux na architektuře AMD64. Tato architektura kromě běžné paměti pracuje ještě s *registry* – těch jsou řádově jednotky, také se do nich ukládají čísla a jsou přístupné rychleji než operační paměť. Můžeme si představit, že jsou uložené uvnitř aritmeticko-logické jednotky.

Aby se lidem jednotlivé instrukce lépe četly, mají přiřazeny své symbolické názvy. Tomuto jazyku symbolických instrukcí se říká *assembler*. Kromě symbolických názvů instrukcí dovoluje assembler ještě pro pohodlí pojmenovat adresy a několik dalších užitečných věcí.

```
MAIN:   pushq   %rbx            # uschovej registr RBX na zásobník
        xorl    %ebx, %ebx      # vynuluj registr EBX
LOOP:   movsbl  str(%rbx), %edi # ulož do EDI RBX-tý znak řetězce
        incq    %rbx            # zvyš RBX o 1
        call    putchar         # zavolej funkci putchar z knihovny
        cmpq    $13, %rbx       # už máme v RBX napočítáno 13 znaků?
        jne     LOOP            # pokud ne, skoč na LOOP
        xorl    %eax, %eax      # vynuluj EAX: nastav návratový kód 0
        popq    %rbx            # vrať do RBX obsah ze zásobníku
        ret                     # vrať se z podprogramu
STR:    .string "Hello world\n"
```

Každá instrukce je zapsána posloupností několika bytů. Věříme, že čtenář si dokáže představit přechozí kód zapsaný v číslech, a ukázku vynecháme.

Programátor píšící programy v assembleru musí být perfektně seznámen s instrukční sadou procesoru, vlastnostmi architektury, technickými detaily služeb operačního systému a mnoha dalšími věcmi.

2.2 Rychlost konkrétního výpočtu

Dejme tomu, že chceme změřit dobu běhu našeho příkladu „Hello world“ z předchozího oddílu. Spustíme-li ho na svém počítači několikrát, nejspíš naměříme o něco rozdílné časy. Může za to aktivita ostatních procesů, stav operačního systému, obsahy nejrůznějších vyrovnávacích pamětí a desítky dalších věcí. A to ještě ukázkový program nečte žádná vstupní data. Co kdyby se doba jeho běhu odvíjela od nich?

Takový přístup se tedy hodí pouze pro testování kvality konkrétního programu na konkrétním hardwaru a konfiguraci. Nezatracujeme ho, velmi často se používá pro testování programů určených k nasazení v těch nejvypjatějších situacích. Ale naším cílem v této kapitole je vyvinout prostředek na měření doby běhu obecně popsaného algoritmu, bez nutnosti naprogramování v konkrétním programovacím jazyce a architektuře. Zatím předpokládejme, že program dostane nějaký konkrétní vstup.

Zapomeňme odteď na detaily překladu programu do strojového kódu, zapomeňme dokonce na detaily nějakého konkrétního programovacího jazyka. Algoritmy začneme popisovat *pseudokódem*. To znamená, že nebudeme v programech zabíhat do technických detailů konkrétních jazyků či architektury, nicméně s jejich znalostí bude už potom snadné pseudokód do programovacího jazyka přepsat. Operace budeme popisovat slovně, případně matematickou symbolikou.

Nyní spočítáme celkový počet provedených tzv. *elementárních operací*. Tímto pojmem rozumíme především operace sčítání, odčítání, násobení, porovnávání; také základní řídicí konstrukce, jako jsou třeba skoky a podmíněné skoky. Zkrátka to, co normální procesor zvládne jednou nebo nejvýše několika instrukcemi. Elementární operací rozhodně není například přesun paměťového bloku z místa na místo, byť ho zapíšeme jediným příkazem, třeba při práci s textovými řetězci.

Čas vykonání jedné elementární operace prohlásíme za jednotkový a zbavíme se tak jakýchkoli jednotek ve výsledné době běhu algoritmu. V zásadě je za elementární operace možné zvolit libovolnou rozumnou sadu – doba provádění programu se tak změní nejvýše konstanta-krát, na čemž, jak za chvíli uvidíme, zase tolik nezáleží.

Několik příkladů s hvězdičkami

Než pokročíme dále, zkusme určit počet provedených operací u několika jednoduchých algoritmů. Ty si nejprve na úvod přečtou ze vstupu přirozené číslo n a pak vypisují hvězdičky. Úkol si navíc zjednodušíme: místo počítání všech operací budeme počítat jen vypsané hvězdičky. Čtenář nechť zkusí nejdříve u každého algoritmu počet hvězdiček spočítat sám, a teprve potom se podívat na náš výpočet.

Algoritmus Hvězdičky1
Vstup: Číslo n
1. Pro $i = 1, \ldots, n$ opakujeme:
2. Pro $j = 1, \ldots, n$ opakujeme:
3. Vytiskneme $*$.

V algoritmu 1 vidíme, že vnější cyklus se provede n-krát, vnořený cyklus pokaždé také n-krát, dohromady tedy n^2 vytištěných hvězdiček.

Algoritmus Hvězdičky2
Vstup: Číslo n
1. Pro $i = 1, \ldots, n$ opakujeme:
2. Pro $j = 1, \ldots, i$ opakujeme:
3. Vytiskneme $*$.

Rozepišme, kolikrát se provede vnitřní cyklus v závislosti na i. Pro $i = 1$ se provede jedenkrát, pro $i = 2$ dvakrát, a tak dále, až pro $i = n$ se provede n-krát. Dohromady se vytiskne $1 + 2 + 3 + \ldots + n$ hvězdiček, což například pomocí vzorce na součet aritmetické řady sečteme na $n(n + 1)/2$.

Algoritmus Hvězdičky3
Vstup: Číslo n
1. Dokud $n \geq 1$, opakujeme:
2. Vytiskneme $*$.
3. $n \leftarrow \lfloor n/2 \rfloor$

V každé iteraci cyklu se n sníží na polovinu. Provedeme-li cyklus k-krát, sníží se hodnota n na $\lfloor n/2^k \rfloor$, neboli klesá exponenciálně rychle v závislosti na počtu iterací cyklu. Chceme-li určit počet iterací, vyřešíme rovnici $\lfloor n/2^\ell \rfloor = 1$ pro neznámou ℓ. Výsledkem je tedy zhruba dvojkový logaritmus n.

Algoritmus HVĚZDIČKY4
Vstup: Číslo n

1. Dokud je $n > 0$, opakujeme:
2. Je-li n liché:
3. Pro $i = 1, \ldots, n$ opakujeme:
4. Vytiskneme $*$.
5. $n \leftarrow \lfloor n/2 \rfloor$

Zde se již situace začíná komplikovat. V každé iteraci vnějšího cyklu se n sníží na polovinu a vnořený cyklus se provede pouze tehdy, bylo-li předtím n liché.

To, kolikrát se vnořený cyklus provede, tedy nepůjde úplně snadno vyjádřit pouze z velikosti čísla n. Spočítejme, jak vypadá nejdelší možný průběh algoritmu, kdy test na lichost n pokaždé uspěje. Tehdy se vytiskne $h = n + \lfloor n/2 \rfloor + \lfloor n/2^2 \rfloor + \ldots + \lfloor n/2^k \rfloor + \ldots + 1$ hvězdiček. Protože není na první pohled vidět, kolik h přepsané do tohoto jednoduchého vzorce vyjde, spokojíme se alespoň s *horním odhadem* hodnoty h.[2]

Označme symbolem s počet členů v součtu h. Pak můžeme tento součet upravovat takto:

$$h = \sum_{i=0}^{s} \left\lfloor \frac{n}{2^i} \right\rfloor \leq \sum_{i=0}^{s} \frac{n}{2^i} = n \cdot \sum_{i=0}^{s} \frac{1}{2^i} \leq n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i}.$$

Nejprve jsme využili toho, že $\lfloor x \rfloor \leq x$ pro každé x. Poté jsme vytkli n a přidali do řady další členy až do nekonečna, čímž součet určitě neklesl.

Jak víme z matematické analýzy, *geometrická řada* $\sum_{i=0}^{\infty} q^i$ pro jakékoliv $q \in (-1, 1)$ konverguje a má součet $1/(1-q)$. V našem případě je $q = 1/2$, takže součet řady je 2. Dostáváme, že počet vytištěných hvězdiček nebude vyšší než $2n$.

Protože se v této kapitole snažíme vybudovat míru doby běhu algoritmu a nikoli počtu vytištěných hvězdiček, ukážeme u našich příkladů, že z počtu vytištěných hvězdiček vyplývá i řádový počet všech provedených operací. V algoritmu 1 na vytištění jedné hvězdičky provedeme maximálně čtyři operace: změnu proměnné j, možná ještě změnu proměnné i a testy, zdali neskončil vnitřní či vnější cyklus. V algoritmech 2 a 3 je to velmi podobně – na vytištění jedné hvězdičky potřebujeme maximálně čtyři další operace.

Algoritmus 4 v případě, že všechny testy lichosti uspějí, pro tisk hvězdičky provede změnu proměnné i, maximálně jeden test lichosti, maximálně jednu aritmetickou operaci s n

[2] Nenechte se mýlit – ač na to nevypadá, *odhad* je naprosto exaktní pojem. V matematice to znamená libovolnou nerovnost, která nějakou neznámou veličinu omezuje shora (horní odhad), nebo zdola (odhad dolní).

a podmíněný skok. Co však je-li někdy v průběhu n sudé? Co když test na lichost uspěje pouze jednou nebo dokonce vůbec? (K rozmyšlení: kdy se to může stát?) Může se tedy přihodit, že se vytiskne jen velmi málo hvězdiček (třeba jedna), a algoritmus přesto vykoná velké množství operací. V tomto algoritmu tedy počet operací s počtem hvězdiček nekoresponduje. Čtenáře odkážeme na cvičení 2, aby zjistil přesně, na čem počet vytištěných hvězdiček závisí.

Který algoritmus je lepší?

Pojďme shrnout počty vykonaných kroků (nebo alespoň jejich horní odhady) našich čtyř algoritmů:

Hvězdičky1	$4n^2$
Hvězdičky2	$4n(n+1)/2 = 2n^2 + 2n$
Hvězdičky3	$4 \log_2 n$
Hvězdičky4	$8n$

Jakmile umíme počet kroků popsat dostatečně jednoduchou matematickou funkcí, můžeme předpovědět, jak se algoritmy budou chovat pro různá n, aniž bychom je skutečně spustili.

Představme si, že n je nějaké gigantické číslo, řekněme v řádu bilionů. Nejprve si všimněme, že algoritmus 3 bude nejrychlejší ze všech – i pro tak obrovské n se vykoná pouze několik málo kroků. Algoritmus 4 vykoná kroků řádově biliony. Zato algoritmy 1 a 2 budou mnohem, mnohem pomalejší.

Další postřeh se bude týkat algoritmů 1 a 2. Pro, řekněme, $n = 10^{10}$ vykoná první algoritmus $4 \cdot 10^{20}$ kroků a druhý algoritmus zhruba $2 \cdot 10^{20}$ kroků. Na první pohled se zdá, že je tedy první dvakrát pomalejší než druhý. Zdání ale klame: různé operace, které jsme považovali za elementární, mohou na skutečném počítači odpovídat různým kombinacím strojových instrukcí. A dokonce i jednotlivé strojové instrukce se mohou lišit v rychlosti.

V naší poněkud abstraktní představě o době výpočtu se takto jemné rozdíly ztrácejí. Algoritmy 1 a 2 prostě neumíme porovnat. Přesto můžeme jednoznačně říci, že oba jsou výrazně pomalejší než algoritmy 3 a 4.

Napříště tedy budeme multiplikativní konstanty v počtech operací velkoryse přehlížet. Beztak jsou strojově závislé a chování algoritmu pro velká n nijak zásadně neovlivňují. Podobně si můžeme všimnout, že ve výrazu $2n^2 + 2n$ je pro velká n člen $2n^2$ obrovský oproti $2n$, takže $2n$ můžeme klidně vynechat.

Doby výpočtu našich ukázkových algoritmů tedy můžeme popsat ještě jednodušími funkcemi n^2, n^2, $\log_2 n$ a n, aniž bychom přišli o cokoliv zásadního.

Cvičení

1. Určete počet vytištěných hvězdiček u algoritmu Hvězdičky3 naprosto přesně, jednoduchým vzorcem.

2.* Na čem u algoritmu Hvězdičky4 závisí počet vytištěných hvězdiček? Najděte přesný vzorec, případně co nejtěsnější dolní a horní odhad.

2.3 Složitost algoritmu

Časová složitost

Už jsme se naučili, jak stanovit dobu běhu algoritmu pro konkrétní vstup. Dokonce jsme v příkladech s hvězdičkami uměli dobu běhu vyjádřit jako funkci vstupu. Málokdy to půjde tak snadno: vstup bývá mnohem složitější než jedno jediné číslo. Přesto obvykle bude platit, že pro „větší“ vstupy program poběží pomaleji než pro ty „menší“.

Pořídíme si proto nějakou *míru velikosti vstupu* a čas budeme vyjadřovat v závislosti na ní. Pokud program pro různé vstupy téže velikosti běží různě dlouho, uvážíme ten nejpomalejší případ – vždy je dobré být připraveni na nejhorší. Tím dostaneme funkci, které se říká *časová složitost* algoritmu.

Zastavme se na chvíli u toho, co si představit pod velikostí vstupu. Pokud je vstupem posloupnost čísel, obvykle za velikost považujeme jejich počet. Podobně za velikost řetězce znaků prohlásíme počet znaků. Tento přístup ale selže třeba pro Euklidův algoritmus z oddílu 1.3 – ten na vstupu pokaždé dostává dvě čísla a běží různě dlouho v závislosti na jejich hodnotách. Tehdy je přirozené považovat za velikost vstupu maximum ze zadaných čísel.

Vyzbrojeni předchozími poznatky, popíšeme „kuchařku“, jak určit časovou složitost daného algoritmu. Ještě to nebude poctivá matematická definice, tu si necháme na příští oddíl, ale pro téměř všechny algoritmy z této knížky poslouží stejně dobře.

1. Ujasníme si, jak se měří velikost vstupu.
2. Určíme maximální možný počet $f(n)$ elementárních operací algoritmu provedených na vstupu o velikosti n. Pokud neumíme určit počet operací přesně, najdeme alespoň co nejlepší horní odhad.
3. Ve výsledné formuli $f(n)$, která je součtem několika členů, ponecháme pouze nejrychleji rostoucí člen a ty ostatní zanedbáme, tj. vypustíme.
4. Seškrtáme *multiplikativní konstanty* – tedy ty, kterými se zbytek funkce násobí. Ale jen ty! Nikoli ostatní čísla ve vzorci.

Jak bychom podle naší kuchařky postupovali u algoritmu Hvězdičky2? Už jsme spočetli, že se vykoná nejvýše $2n^2 + 2n$ elementárních operací. Škrtneme člen $2n$ a zbude nám $2n^2$. Na závěr škrtneme multiplikativní konstantu 2. Pozor, dvojka v exponentu není multiplikativní konstanta.

Funkci $g(n)$, která zbude, nazveme *asymptotickou* časovou složitostí algoritmu a tento fakt označíme výrokem „algoritmus má časovou složitost $\mathcal{O}(g(n))$“. Naše ukázkové algoritmy tudíž mají po řadě časovou složitost $\mathcal{O}(n^2)$, $\mathcal{O}(n^2)$, $\mathcal{O}(\log n)$ a $\mathcal{O}(n)$.

Zde nechť si čtenář povšimne, že jsem ve výrazu $\mathcal{O}(\log n)$ vynechali základ logaritmu. Jednak je v informatické literatuře zvykem, že log bez uvedení základu je dvojkový (takové jsou nejčastější). Mnohem důležitější ale je, že logaritmy o různých základech se liší pouze konstanta-krát a konstanty přeci zanedbáváme (viz cvičení 3).

Běžné složitostní funkce

Složitosti algoritmů mohou být velmi komplikované funkce. Nejčastěji se však setkáváme s algoritmy, které mají jednu z následujících složitostí. Složitosti $\mathcal{O}(n)$ říkáme *lineární,* $\mathcal{O}(n^2)$ *kvadratická,* $\mathcal{O}(n^3)$ *kubická,* $\mathcal{O}(\log n)$ *logaritmická,* $\mathcal{O}(2^n)$ *exponenciální* a $\mathcal{O}(1)$ *konstantní* (provede se pouze konstantně mnoho kroků).

Jak zásadní roli hraje časová složitost algoritmu, je poznat z následující tabulky růstu funkcí (obrázek 2.2).

Funkce	$n = 10$	$n = 100$	$n = 1000$	$n = 10\,000$
$\log n$	1	2	3	4
n	10	100	1000	10 000
$n \log n$	10	200	3000	40 000
n^2	100	10 000	10^6	10^8
n^3	1000	10^6	10^9	10^{12}
2^n	1024	$\approx 10^{31}$	$\approx 10^{310}$	$\approx 10^{3100}$

Obrázek 2.2: Chování různých funkcí

Zkusme do další tabulky (obrázek 2.3) zaznamenat odhad, jak dlouho by algoritmy s uvedenými časovými složitostmi běžely na současném počítači typu PC. Obvyklé PC (v roce 2017) vykoná okolo 10^9 instrukcí za sekundu. Algoritmus s logaritmickou složitostí doběhne v řádu nanosekund pro jakkoliv velký vstup. I ten s lineární složitostí je ještě příjemně rychlý a můžeme čekat, že bude použitelný i pro opravdu velké vstupy. Zato kubický algoritmus se pro $n = 10\,000$ řádně zadýchá a my se na výsledek načekáme přes čtvrt hodiny.

To ale nic není proti exponenciálnímu algoritmu: pro n rovné postupně 10, 20, 30, 40, 50 poběží cca 1 μs, 1 ms, 1 s, 18 min a 13 dní. Pro $n = 100$ se už výsledku nedočkáme – až Země zanikne a hvězdy vyhasnou, program bude stále počítat a počítat.

Složitost	$n = 10$	$n = 100$	$n = 1000$	$n = 10\,000$
$\log n$	1 ns	2 ns	3 ns	4 ns
n	10 ns	100 ns	1 μs	10 μs
$n \log n$	10 ns	200 ns	3 μs	40 μs
n^2	100 ns	10 μs	1 ms	0.1 s
n^3	1 μs	1 ms	1 s	16.7 min
2^n	1 μs	10^{24} let	10^{303} let	10^{3093} let

Obrázek 2.3: Přibližné doby běhu programů s různými složitostmi

Znalec trhu s hardwarem by samozřejmě mohl namítnout, že vývoj počítačů jde kupředu tak rychle, že každé dva roky se jejich výkon zdvojnásobí (tomu se říká Mooreův zákon). Jenže algoritmus se složitostí $\mathcal{O}(2^n)$ na dvakrát rychlejším počítači zpracuje ve stejném čase pouze o jedničku větší vstup. Budeme-li trpělivě čekat 20 let, získáme tisíckrát rychlejší počítač, takže zpracujeme o 10 větší vstup.

Proto se zpravidla snažíme exponenciálním algoritmům vyhýbat a uchylujeme se k nim, pouze pokud nemáme jinou možnost. Naproti tomu *polynomiální* algoritmy, tedy ty se složitostmi $\mathcal{O}(n^k)$ pro pevná konstantní k, můžeme chápat jako efektivní. I mezi nimi samozřejmě budeme rozlišovat a snažit se o co nejmenší stupeň polynomu.

Prostorová složitost

Velmi podobně jako časová složitost se dá zavést tzv. *prostorová složitost* (někdy též *paměťová složitost*), která měří paměťové nároky algoritmu. K tomu musíme spočítat, kolik nejvíce tzv. elementárních paměťových buněk bude v daném algoritmu v každém okamžiku použito. V běžných programovacích jazycích (jako jsou například C nebo Pascal) za elementární buňku můžeme považovat například proměnnou typu `integer`, `float`, `byte`, či ukazatel, naopak elementární velikost rozhodně nemají pole či textové řetězce.

Opět vyjádříme množství spotřebovaných paměťových buněk funkcí $f(n)$ v závislosti na velikosti vstupu n, pokud to neumíme přesně, tak alespoň co nejlepším horním odhadem, aplikujeme čtyřbodovou kuchařku a výsledek zapíšeme pomocí notace $\mathcal{O}(g(n))$. V našich čtyřech příkladech je tedy všude prostorová složitost $\mathcal{O}(1)$, neboť vždy používáme pouze konstantní množství celočíselných proměnných.

Průměrná složitost

Doposud jsme uvažovali takzvanou složitost v nejhorším případě: zajímalo nás, jak nejdéle může algoritmus počítat, dostane-li vstup dané velikosti. Někdy se ale stává, že výpočet obvykle doběhne rychle, pouze existuje několik málo anomálních vstupů, na nichž je pomalý. Tehdy může být praktičtější počítat *průměrnou složitost* (někdy se také říká složitost v průměrném případě). Funkce popisující tuto složitost je definována jako aritmetický průměr časových (prostorových) nároků algoritmů přes všechny vstupy dané velikosti.

Alternativně můžeme průměrnou složitost definovat pomocí teorie pravděpodobnosti. Představíme si, že budeme vstup volit náhodně ze všech vstupů dané velikosti. Potom střední hodnota časových (prostorových) nároků programu bude právě průměrná časová (prostorová) složitost.

Pravděpodobnostní analýzu algoritmů prozkoumáme v kapitole 11 a poskytne nám mnoho zajímavých výsledků.

Složitost problému

Vedle složitosti algoritmu (resp. programu) zavádíme také pojem *složitost problému.* Představme si, že pro daný problém P známe algoritmus, který ho řeší s časovou složitostí $s(n)$, a zároveň umíme dokázat, že neexistuje algoritmus, který by problém P řešil s lepší časovou složitostí než $s(n)$. Potom dává smysl říci, že *složitost problému* P je $s(n)$.

Stanovit složitost nějakého problému je obvykle velice obtížný úkol. Často se musíme spokojit pouze s horní mezí složitosti problému, odvozenou typicky popisem a analýzou vhodného algoritmu, a dolní mezí složitosti problému, odvozenou typicky nějakým matematickým argumentem.

Koncept je hezky vidět například na problému třídění prvků: dostaneme n prvků, které umíme pouze porovnávat a přesouvat, a máme je přerovnat do rostoucí posloupnosti. Tento problém je dobře prostudován: jeho složitost je řádově $n \log n$. To znamená, že existuje algoritmus schopný seřadit n prvků v čase $\mathcal{O}(n \log n)$ a zároveň neexistuje asymptoticky rychlejší algoritmus. Toto tvrzení precizně formulujeme a dokážeme v oddílu 3.3.

Cvičení

1. Jaká je složitost následujícího (pseudo)kódu vzhledem k n?

```
1. Opakujeme, dokud n > 0:
2.     Je-li n liché, položíme n ← n − 1.
3.     Jinak položíme n ← ⌊n/2⌋.
```

2. Stanovte časovou a prostorovou složitost všech algoritmů z kapitoly 1.

3. Dokažte, že $\log_a n$ a $\log_b n$ se liší pouze konstanta-krát, přičemž konstanta závisí na a a b, ale nikoliv na n.

2.4 Asymptotická notace

Matematicky založený čtenář jistě cítí, že popis „zjednodušování“ funkcí v naší čtyřbodové kuchařce je poněkud vágní a žádá si exaktní definice. Pojďme se do nich pustit.

Definice: Nechť $f, g : \mathbb{N} \to \mathbb{R}$ jsou dvě funkce. Řekneme, že funkce $f(n)$ *je třídy* $\mathcal{O}(g(n))$, jestliže existuje taková kladná reálná konstanta c, že pro *skoro všechna* n platí $f(n) \leq cg(n)$. Skoro všemi n se myslí, že nerovnost může selhat pro konečně mnoho výjimek, tedy že existuje nějaké přirozené n_0 takové, že nerovnost platí pro všechna $n \geq n_0$. Funkci $g(n)$ se pak říká *asymptotický horní odhad* funkce $f(n)$.⟨3⟩

Jinými slovy, dostatečně velký násobek funkce $g(n)$ shora omezuje funkci $f(n)$. Konečně mnoho výjimek se hodí tehdy, má-li funkce $g(n)$ několik počátečních funkčních hodnot nulových či dokonce záporných, takže je nemůžeme „přebít“ jakkoliv vysokou konstantou c.

Poněkud formálněji bychom se na zápis $\mathcal{O}(g)$ mohli dívat jako na množinu všech funkcí f, které splňují uvedenou definici. Pak můžeme místo „funkce f je třídy $\mathcal{O}(g)$“ psát prostě $f \in \mathcal{O}(g)$. Navíc nám to umožní elegantně zapisovat i různé vztahy typu $\mathcal{O}(n) \subseteq \mathcal{O}(n^2)$.

Ve většině informatické literatury se ovšem s $\mathcal{O}$-čkovou notací zachází mnohem nepořádněji: často se píše „f je $\mathcal{O}(g)$“, nebo dokonce $f = \mathcal{O}(g)$. I my si občas takové zjednodušení dovolíme. Stále ale mějme na paměti, že se nejedná o žádnou rovnost, nýbrž o nerovnost (horní odhad).

Zbývá nahlédnout, že instrukce naší čtyřbodové „kuchařky“ jsou důsledky právě vyslovené definice. Čtvrtý bod nás nabádá ke škrtání multiplikativních konstant, což definice $\mathcal{O}$ přímo dovoluje. Třetí bod můžeme formálně popsat takto:

Lemma: Nechť $f(n) = f_1(n) + f_2(n)$ a $f_1(n) \in \mathcal{O}(f_2(n))$. Pak $f(n) \in \mathcal{O}(f_2(n))$.

Důkaz: Z předpokladu víme, že $f_1(n) \leq cf_2(n)$ platí skoro všude pro vhodnou konstantu c. Proto je také skoro všude $f_1(n) + f_2(n) \leq (1 + c) \cdot f_2(n)$, což se jistě vejde do $\mathcal{O}(f_2(n))$. □

⟨3⟩ Proč se tomuto odhadu říká asymptotický? V matematické analýze se zkoumá asymptota funkce, což je přímka, jejíž vzdálenost od dané funkce se s rostoucím argumentem zmenšuje a v nekonečnu se dotýkají. Podobně my zde zkoumáme chování funkce pro n blížící se k nekonečnu.

Pozor na to, že vyjádření složitosti pomocí $\mathcal{O}$ může být příliš hrubé. Kvadratická funkce $2n^2 + 3n + 1$ je totiž třídy $\mathcal{O}(n^2)$, ale podle uvedené definice patří také do třídy $\mathcal{O}(n^3)$, $\mathcal{O}(n^4)$, atd. Proto se nám bude hodit také obdobné značení pro asymptotický dolní odhad a „asymptotickou rovnost“.

Definice: Mějme dvě funkce $f, g : \mathbb{N} \to \mathbb{R}$. Řekneme, že funkce $f(n)$ je třídy $\Omega(g(n))$, jestliže existuje taková kladná reálná konstanta c, že $f(n) \geq cg(n)$ pro skoro všechna n. Tomu se říká *asymptotický dolní odhad.*

Definice: Řekneme, že funkce $f(n)$ je třídy $\Theta(g(n))$, jestliže $f(n)$ je jak třídy $\mathcal{O}(g(n))$, tak třídy $\Omega(g(n))$.

Symboly Ω a Θ mohou opět značit i příslušné množiny funkcí. Pak jistě platí $\Theta(g) = \mathcal{O}(g) \cap \Omega(g)$.

Příklad: O našich ukázkových algoritmech 1, 2, 3, 4 můžeme říci, že mají složitosti po řadě $\Theta(n^2)$, $\Theta(n^2)$, $\Theta(\log n)$ a $\Theta(n)$.

Při skutečném srovnávání algoritmů by tedy bylo lepší zapisovat složitost pomocí Θ, nikoliv podle $\mathcal{O}$. To by zajisté poskytlo úplnější informaci o chování funkce. Ne vždy se nám to ale povede: analýzou algoritmu mnohdy dostáváme pouze horní odhad počtu provedených instrukcí nebo potřebných paměťových míst. Například se nám může stát, že v algoritmu je několik podmínek a nedovedeme určit, které z jejich možných kombinací mohou nastat současně. Raději tedy předpokládáme, že nastanou všechny, čímž dostaneme horní odhad.

Budeme proto nadále vyjadřovat složitost algoritmů převážně pomocí symbolu $\mathcal{O}$. Při tom však budeme usilovat o to, aby byl náš odhad asymptotické složitosti co nejlepší.

Cvičení

1. Nalezněte co nejvíce asymptotických vztahů mezi těmito funkcemi: n, $\log n$, $\log \log n$, $\sqrt{n}$, $n^{\log n}$, 2^n, $n^{3/2}$, $n!$, n^n.
2. Nahlédněte, že $f(n) \in \Theta(g(n))$ by se dalo ekvivalentně definovat tak, že pro vhodné konstanty $c_1, c_2 > 0$ platí $c_1 g(n) \leq f(n) \leq c_2 g(n)$ pro skoro všechna n.
3. Dokažte, že $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max(f(n), g(n)))$ pro $f, g \geq 0$.
4. Dokažte, že $n \log n \notin \mathcal{O}(n)$.
5. Dokažte, že $\log n \in \mathcal{O}(n^\varepsilon)$ pro každé $\varepsilon > 0$.
6. Najděte co nejlepší asymptotický odhad funkce $\log_n(n!)$.
7. Najděte funkce f a g takové, že neplatí ani $f = \mathcal{O}(g)$, ani $g = \mathcal{O}(f)$.

2.5 Výpočetní model RAM

Matematicky založený jedinec stále nemůže být plně spokojen. Doposud jsme totiž odbývali přesné určení toho, co můžeme v algoritmu považovat za elementární operace a elementární paměťové buňky. Naší snahou bude vyhnout se obtížně řešitelným otázkám u věcí jako například reprezentace reálných čísel a zacházení s nimi. Situaci vyřešíme šalamounsky – definujeme vlastní teoretický stroj, který bude mít přesně definované chování, přesně definovaný čas provádění instrukcí a přesně definovaný rozsah a vlastnosti paměťové buňky. Potom dává dobrý smysl měřit časovou a paměťovou náročnost naprogramovaného algoritmu naprosto přesně – nezdržují nás vedlejší efekty reálných počítačů a operačních systémů.

Jedním z mnoha teoretických modelů je tzv. *Random Access Machine*, neboli RAM.⟨4⟩ RAM není jeden pevný model, nýbrž spíše rodina podobných strojů, které sdílejí určité společné vlastnosti.

Paměť RAMu tvoří pole celočíselných buněk adresovatelné celými čísly. Každá buňka pojme jedno celé číslo. Bystrý čtenář se nyní otáže: „To jako neomezeně velké číslo?“ Problematiku omezení kapacity buňky rozebereme níže.

Program je konečná posloupnost sekvenčně prováděných instrukcí dvou typů: aritmetických a řídicích.

Aritmetické instrukce mají obvykle dva vstupní argumenty a jeden výstupní argument. Argumenty mohou být buďto přímé konstanty (s výjimkou výstupního argumentu), přímo adresovaná paměťová buňka (zadaná číslem) nebo nepřímo adresovaná paměťová buňka (její adresa je uložena v přímo adresované buňce).

Řídicí instrukce zahrnují skoky (na konkrétní instrukci programu), podmíněné skoky (například když se dva argumenty instrukce rovnají) a instrukci zastavení programu.

Na začátku výpočtu obsahuje paměť v určených buňkách vstup a obsah ostatních buněk je nedefinován. Potom je program sekvenčně prováděn, instrukci za instrukcí. Po zastavení programu je obsah smluvených míst v paměti interpretován jako výstup programu.

Zmiňme také, že existují „ještě teoretičtější“ výpočetní modely, jejichž zástupcem je tzv. Turingův stroj.

⟨4⟩ Název lze přeložit do češtiny jako „stroj s náhodným přístupem“. Méně otrocký a výstižnější překlad by mohl znít „stroj s přímým přístupem do paměti“, což je však zase příliš dlouhé a kostrbaté, stroji tedy budeme říkat prostě RAM. Pozor, hrozí zmatení zkratek s *Random Access Memory*, čili běžným názvem operační paměti počítače typu PC.

Konkrétní model RAMu

V našem popisu strojů z rodiny RAM jsme vynechali mnoho podstatných detailů. Například přesný čas vykonávání jednotlivých instrukcí, povolený rozsah čísel v jedné paměťové buňce, prostorovou složitost jedné buňky, přesné vymezení instrukční sady, zejména aritmetických operací.

V tomto oddílu přesně definujeme jeden konkrétní model RAMu. Popíšeme tedy paměť, zacházení s programem a výpočtem, instrukční sadu a chování stroje.

Procesor v každém kroku provede právě jednu instrukci. Typická instrukce přečte jeden nebo dva operandy z paměti, něco s nimi spočítá a výsledek opět uloží do paměti. Operandem instrukce může být:

- *literál* – konstanta zakódovaná přímo v instrukci. Literály zapisujeme jako čísla v desítkové soustavě.
- *přímo adresovaná buňka* – číslo uložené v paměťové buňce, jejíž adresa je zakódovaná v instrukci. Zapisujeme jako `[adresa]`, kde `adresa` je libovolné celé číslo.
- *nepřímo adresovaná buňka* – číslo uložené v buňce, jejíž adresa je v přímo adresované buňce. Píšeme `[[adresa]]`.

Operandy tedy mohou být například `42`, `[16]`, `[-3]` nebo `[[16]]`, ale nikoliv `[[[5]]]` ani `[3*[5]]`. Svůj výsledek může instrukce uložit do přímo či nepřímo adresované paměťové buňky.

Vstup a výstup stroj dostává a předává většinou v paměťových buňkách s nezápornými indexy, buňky se zápornými indexy se obvykle používají pro pomocná data a proměnné. Prvních 26 buněk se zápornými indexy, tj. `[-1]` až `[-26]` má pro snazší použití přiřazeny přezdívky `A`, `B`, až `Z` a říkáme jim *registry*. Jejich hodnoty lze libovolně číst a zapisovat a používat pro indexaci paměti, lze tedy psát např. `[A]`, ale nikoliv `[[A]]`. Registry lze použít například jako úložiště často užívaných pomocných proměnných.

Nyní vyjmenujeme instrukce stroje. X, Y a Z vždy představují některý z výše uvedených výrazů pro přístup do paměti či registrů, Y a Z mohou být navíc i konstanty.

- Aritmetické instrukce:
  - Přiřazení: X `:=` Y
  - Negace: X `:= -`Y
  - Sčítání: X `:=` Y `+` Z
  - Odčítání: X `:=` Y `-` Z
  - Násobení: X `:=` Y `*` Z

  - Celočíselné dělení: X `:=` Y `/` Z
    Číslo Z musí být nenulové. Zaokrouhlujeme vždy k nule, takže (-1) `/` $3 = 0 =$ `-`$(1$ `/` $3)$.
  - Zbytek po celočíselném dělení: X `:=` Y `%` Z
    Číslo Z musí být kladné. Dodržujeme, že $(Y$ `/` $Z)$ `*` Z `+` $(Y$ `%` $Z) = Y$, takže (-1) `%` $3 = -1$.

- Logické instrukce:
  - Bitová konjunkce (AND): X `:=` Y `&` Z
    i-tý bit výsledku je 1 právě tehdy, když jsou jedničkové i-té bity obou operandů. Například 12 `&` $5 = (1100)_2$ `&` $(0101)_2 = (0100)_2 = 4$.
  - Bitová disjunkce (OR): X `:=` Y `|` Z
    i-tý bit výsledku je 1, pokud je jedničkový i-tý bit aspoň jednoho operandu. Například 12 `|` $5 = (1100)_2$ `|` $(0101)_2 = (1101)_2 = 13$.
  - Bitová nonekvivalence (XOR): X `:=` Y `^` Z
    i-tý bit výsledku je 1, pokud je jedničkový i-tý bit právě jednoho operandu. Například 12 `^` $5 = (1100)_2$ `^` $(0101)_2 = (1001)_2 = 9$.
  - Bitový posun doleva: X `:=` Y `«` Z
    Doplnění Z nul na konec binárního zápisu čísla Y. Například 11`«`$3 = (1011)_2$`«`$3 = (1011000)_2 = 88$.
  - Bitový posun doprava: X `:=` Y `»` Z
    Smazání posledních Z bitů binárního zápisu čísla Y. Například 11`»`$2 = (1011)_2$`»` $2 = (10)_2 = 2$.

- Řídicí instrukce:
  - Ukončení výpočtu: `halt`
  - Nepodmíněný skok: `goto` *label*, kde *label* je návěští, které se definuje napsáním *label*`:` před instrukci.
  - Podmíněný příkaz: `if` *podmínka* `then` *instrukce*, přičemž *instrukce* je libovolná instrukce kromě podmíněného příkazu a *podmínka* je jeden z následujících logických výrazů:
    - Test rovnosti: Y `=` Z
    - Negace testu rovnosti: Y `<>` Z
    - Test ostré nerovnosti: Y `<` Z, případně Y `>` Z
    - Test neostré nerovnosti: Y `<=` Z, případně Y `>=` Z

Doba provádění podmíněného příkazu nezávisí na splnění jeho podmínky a je stejná jako doba provádění libovolné jiné instrukce. Doba běhu programu, kterou používáme v naší definici časové složitosti, je tedy rovna celkovému počtu provedených instrukcí.

S měřením spotřebované paměti musíme být trochu opatrnější, protože program by mohl využít malé množství buněk rozprostřených po obrovském prostoru. Budeme tedy měřit rozdíl mezi nejvyšším a nejnižším použitým indexem paměti.

Časovou a paměťovou složitost pak definujeme zavedeným způsobem jako maximum ze spotřeby času a paměti přes všechny vstupy dané velikosti. Roli velikosti vstupu obvykle hraje počet paměťových buněk obsahujících vstup.

Upozorňujeme, že do časové složitosti nepočítáme dobu potřebnou na načtení vstupu – podle naší konvence je vstup při zahájení výpočtu už přítomen v paměti. Můžeme tedy studovat i algoritmy s lepší než lineární časovou složitostí, například binární vyhledávání z oddílu 1.2. Pokud navíc program do vstupu nebude zapisovat, nebudeme paměť zabranou vstupem ani počítat do spotřebovaného prostoru.

Příklad programu pro RAM

Pro ilustraci přepíšeme algoritmus Hvězdičky2 z předchozích oddílů co nejvěrněji do programu pro náš RAM. Připomeňme tento algoritmus:

Algoritmus Hvězdičky2
Vstup: Číslo n
1. Pro $i = 1, \ldots, n$ opakujeme:
2. Pro $j = 1, \ldots, i$ opakujeme:
3. Vytiskneme `*`.

Zadání pro RAM formulujeme takto: V buňce `[0]` je uloženo číslo n. Výstup je tvořen posloupností buněk počínaje `[1]`, ve kterých je v každé zapsána jednička (namísto hvězdičky jako v původním programu).

```
          I := 1
          Z := 1
VNEJSI:   if I > [0] then halt
          J := 1
VNITRNI:  if J > I then goto DALSI
          [Z] := 1
          Z := Z + 1
          J := J + 1
          goto VNITRNI
DALSI:    I := I + 1
          goto VNEJSI
```

Registry `I` a `J` odpovídají stejnojmenným proměnným algoritmu, registr `Z` ukazuje na buňku paměti, kam zapíšeme příští hvězdičku.

Omezení kapacity paměťové buňky

Náš model má zatím jednu výrazně nereálnou vlastnost – neomezenou kapacitu paměťové buňky. Toho lze využít k nejrůznějším trikům. Ponechme například čtenáři k rozmyšlení, jak veškerá data programu uložit do konstantně mnoha paměťových buněk (cvičení 2 a 3) a pomocí této „komprese“ programy absurdně zrychlovat (cvičení 9).

Proto upravíme stroj tak, abychom na jednu stranu neomezili kapacitu buňky příliš, ale na druhou stranu kompenzovali nepřirozené výhody plynoucí z její neomezenosti. Možností je mnoho, ukážeme jich tedy několik, ke každé dodáme, jaké jsou její výhody a nevýhody, a na závěr zvolíme tu, kterou budeme používat v celé knize.

Přiblížení první. Omezíme kapacitu paměťové buňky pevnou konstantou, řekněme na 64 bitů. Tím jistě odpadnou problémy s neomezenou kapacitou, lze si také představit, že aritmetické instrukce pracující s 64-bitovými čísly lze hardwarově realizovat v jednotkovém čase. Aritmetiku čísel delších než 64 bitů lze řešit funkcemi na práci s dlouhými čísly rozloženými do více paměťových buněk. Zásadní nevýhoda však spočívá v tom, že jsme omezili i adresy paměťových buněk. Každý program proto může použít pouze konstantní množství paměti: 2^{64} buněk. Současné počítače typu PC to tak sice skutečně mají, nicméně z teoretického hlediska je takový stroj nevyhovující, protože umožňuje zpracovávat pouze konstantně velké vstupy.

Přiblížení druhé. Abychom mohli adresovat libovolně velký vstup, potřebujeme čísla o alespoň $\log n$ bitech, kde n je velikost vstupu. Omezíme tedy velikost čísla v jedné paměťové buňce na $k \cdot \log n$ bitů, kde k je libovolná konstanta. Hodnota čísla pak musí být menší než $2^{k \log n} = (2^{\log n})^k = n^k$. Jinými slovy, hodnoty čísel jsme omezili nějakým polynomem ve velikosti vstupu.

Tento model odstraňuje spoustu nevýhod předchozího: většina zákeřných triků využívajících kombinaci neomezené kapacity buňky a jednotkové ceny instrukce k nepřirozeně rychlému počítání na něm neuspěje. Model má jedno omezení – pokud je velikost čísla v buňce nejvýše polynomiální, znamená to, že nemůžeme použít exponenciálně či více paměťových buněk, protože jich tolik zkrátka nenaadresujeme. Nemůžeme tedy na tomto stroji používat algoritmy s exponenciální paměťovou složitostí. Ty jsou sice málokdy praktické, ale například v oddílu 19.5 se nám budou hodit.

Přiblížení třetí. Abychom neomezili množství paměti, potřebujeme povolit libovolně velká čísla. Vzdejme se tedy naopak předpokladu, že všechny instrukce trvají stejně dlouho. Zavedeme *logaritmickou cenu instrukce.* To znamená, že jedna instrukce potrvá tolik jednotek času, kolik je součet velikostí všech čísel, s nimiž pracuje, měřený v bitech. To zahrnuje operandy, výsledek i adresy použitých buněk paměti. Cena se nazývá logaritmická proto, že počet bitů čísla je úměrný logaritmu čísla. Tedy například instrukce `[5] := 3*8` bude mít cenu $2+4+5+3 = 14$ jednotek času, protože čísla 3 a 8 mají 2 a 4 bity, výsledek 24 zabere 5 bitů a ukládá se na adresu 5 zapsanou 3-bitovým číslem.

Je sice stále možné uložit veškerá data programu do konstantně mnoha buněk, ale instrukce s takovými čísly pracují výrazně pomaleji. V tom spočívá i nevýhoda modelu. I jednoduché algoritmy, které původně měly evidentně lineární časovou složitost, najednou poběží pomaleji – například vypsání n hvězdiček potrvá $\Theta(n \log n)$, jelikož i obyčejné zvýšení řídicí proměnné cyklu zabere čas $\Theta(\log n)$. To je poněkud nepohodlné a skutečné počítače se tak nechovají.

Přiblížení čtvrté. Pokusme se o kompromis mezi předchozími dvěma modely. Zavedeme *poměrnou logaritmickou cenu instrukce*. To bude logaritmická cena vydělená logaritmem velikosti vstupu a zaokrouhlená nahoru. Dokud tedy budeme pracovat s polynomiálně velkými čísly (tedy o řádově logaritmickém počtu bitů), poměr bude shora omezen konstantou a budeme si moci představovat, že všechny instrukce mají konstantní ceny. Jakmile začneme pracovat s většími čísly, cena instrukcí odpovídajícím způsobem poroste.

Poměrné logaritmické ceny se tedy chovají intuitivně, neomezují adresovatelný prostor a odstraňují paradoxy původního modelu. Všechny algoritmy v této knize tedy budeme analyzovat v tomto modelu. Navíc použijeme podobný model i pro měření spotřebované paměti: jedna buňka paměti zabere tolik prostoru, kolik je počet bitů potřebných na reprezentaci její adresy a hodnoty, relativně k logaritmu velikosti vstupu. Započítáme všechny buňky mezi minimální a maximální adresou, na níž program přistoupil.

Naše teoretická práce je nyní u konce. Máme přesnou definici teoretického stroje RAM, pro který je přesně definována časová a prostorová složitost programů na něm běžících. Můžeme se pustit do analýzy konkrétních algoritmů.

Cvičení

1. Naprogramujte na RAMu zbývající algoritmy z oddílu 2.2 a z úvodní kapitoly.

2. Mějme RAM s neomezenou velikostí čísel. Vymyslete, jak zakódovat libovolné množství celých čísel $c_1, \ldots, c_n$ do jednoho celého čísla C tak, aby se jednotlivá čísla c_i dala jednoznačně dekódovat.

3. Navrhněte postup, jak v případě neomezené kapacity paměťové buňky pozměnit libovolný program na RAMu tak, aby používal jen konstantně mnoho paměťových buněk. Program můžete libovolně zpomalit. Kolik nejméně buněk je potřeba?

4. Spočítejte přesně počet provedených instrukcí v algoritmu Hvězdičky2 naprogramovaném na RAMu. Vyjádřete jej jako funkci proměnné n.

5. Pokračujme v předchozím cvičení: jaká bude přesná doba běhu programu, zavedeme-li logaritmickou, případně poměrnou logaritmickou cenu instrukce?

6. Rozmyslete si, jak do instrukcí RAMu překládat konstrukce známé z vyšších programovacích jazyků: podmínky, cykly, volání podprogramů s lokálními proměnnými a rekurzí.

7. Vymyslete, jak na RAMu prohodit obsah dvou paměťových buněk, aniž byste použili jakoukoliv jinou buňku.

8.* Vymyslete, jak na RAMu v konstantním čase otestovat, zda je číslo mocninou dvojky.

9. Mějme RAM s jednotkovou cenou instrukce a neomezenou velikostí čísel. Ukažte, jak v čase $\mathcal{O}(n)$ zakódovat vektor n přirozených čísel tak, abyste z kódů uměli v konstantním čase vypočítat skalární součin dvou vektorů. Z toho odvoďte algoritmus pro násobení matic $n \times n$ v čase $\mathcal{O}(n^2)$.

10. *Interaktivní RAM:* Na naší verzi RAMu dostanou všechny programy hned na začátku celý vstup, pak nějakou dobu počítají a nakonec vydají celý výstup. Někdy se hodí dostávat vstup po částech a průběžně na něj reagovat. Navrhněte rozšíření RAMu, které to umožní.

11. *Registrový stroj* je ještě jednodušší model výpočtu. Disponuje konečným počtem registrů, každý je schopen pojmout jedno přirozené číslo. Má tři instrukce: `inc` pro zvýšení hodnoty registru o 1, `dec` pro snížení o 1 (snížením nuly vyjde opět nula) a `jmpeq` pro skok, pokud se hodnoty dvou registrů rovnají. Vymyslete, jak na registrovém stroji naprogramovat vynulování registru, zkopírování hodnoty z jednoho registru do druhého a vynásobení dvou registrů.

12.* Mějme program pro RAM, jehož vstupem a výstupem je konstantně mnoho čísel (z cvičení 3 víme, že libovolný vstup lze takto zakódovat). Ukažte, jak takový program přeložit na program pro registrový stroj, který počítá totéž. Časová složitost se překladem může libovolně zhoršit.

3 Třídění

3 Třídění

Vyhledávání ve velkém množství dat je každodenním chlebem programátora. Hledání bývá snazší, pokud si data vhodně uspořádáme. Seřadíme-li například pole n čísel podle velikosti, algoritmus binárního vyhledávání z oddílu 1.2 v nich dokáže hledat v čase $\Theta(\log n)$. V této kapitole prozkoumáme různé způsoby, jak data efektivně *seřadit* neboli *setřídit*.[1]

Obvykle budeme pracovat v takzvaném *porovnávacím modelu.* V paměti stroje RAM dostaneme posloupnost n prvků $a_1, \ldots, a_n$. Mimo to dostaneme *komparátor* – funkci, která pro libovolné dva prvky a_i a a_j odpoví, je-li $a_i < a_j$, $a_i > a_j$ nebo $a_i = a_j$. Úkolem algoritmu je přerovnat prvky v paměti do nějakého pořadí $b_1 \leq b_2 \leq \ldots \leq b_n$. S prvky nebudeme provádět nic jiného, než je porovnávat a přesouvat. Budeme předpokládat, že jedno porovnání i přesunutí stihneme provést v konstantním čase.

Čtenář znalý knihovních funkcí populárních programovacích jazyků si jistě vzpomene, že jedním z argumentů funkce pro třídění bývá typicky takovýto komparátor. My budeme pro názornost zápisu předpokládat, že třídíme čísla a komparátor se jmenuje prostě „<“. Stále se ale budeme hlídat, abychom nepoužili jiné operace než ty povolené.

U třídicích algoritmů nás kromě časové složitosti budou zajímat i následující vlastnosti:

Definice: Třídicí algoritmus je *stabilní,* pokud kdykoliv jsou si prvky p_i a p_j rovny, tak jejich vzájemné pořadí na výstupu se shoduje s jejich pořadím na vstupu. Tedy pokud je $p_i = p_j$ pro $i < j$, pak se p_i ve výstupu objeví před p_j.

Definice: Algoritmus třídí prvky *na místě,* pokud prvky neopouštějí paměťové buňky, v nichž byly zadány, s možnou výjimkou konstantně mnoha tzv. pracovních buněk. Kromě toho může algoritmus ukládat libovolné množství číselných proměnných (indexy prvků, parametry funkcí apod.), které prohlásíme za *pomocnou paměť* algoritmu.

Postupně ukážeme, že třídit lze v čase $\Theta(n \log n)$, a to na místě. Stabilní třídění zvládneme stejně rychle, ale už budeme potřebovat pomocnou paměť. Také dokážeme, že v porovnávacím modelu nemůže rychlejší třídicí algoritmus existovat. Ovšem pokud si dovolíme provádět s prvky i další operace, nalezneme efektivnější algoritmy.

3.1 Základní třídicí algoritmy

Nejjednodušší třídicí algoritmy patří do skupiny tzv. *přímých metod*. Všechny mají několik společných rysů: Jsou krátké, jednoduché a třídí na místě. Za tyto příjemné vlastnosti

[1] Striktně vzato, termín *řazení* je správnější. Data totiž nerozdělujeme do nějakých tříd, nýbrž je uspořádáváme, tedy řadíme, dle určitého kritéria. Ovšem pojem *třídění* je mezi českými informatiky natolik zažitý, že je bláhové chtít na tom cokoliv měnit.

zaplatíme kvadratickou časovou složitostí $\Theta(n^2)$. Přímé metody jsou proto použitelné jen tehdy, není-li tříděných dat příliš mnoho.

Selectsort – třídění výběrem

Třídění přímým výběrem (Selectsort) je založeno na opakovaném vybírání nejmenšího prvku. Pole rozdělíme na dvě části: V první budeme postupně stavět setříděnou posloupnost a v druhé nám budou zbývat dosud nesetříděné prvky. V každém kroku nalezneme nejmenší ze zbývajících prvků a přesuneme jej na začátek druhé (a tedy i na konec první) části. Následně zvětšíme setříděnou část o 1, čímž oficiálně potvrdíme členství právě nalezeného minima v konstruované posloupnosti a zajistíme, aby se při dalším hledání již s tímto prvkem nepočítalo.

Algoritmus SelectSort (třídění přímým výběrem)
Vstup: Pole $P[1 \ldots n]$

1. Pro $i = 1, \ldots, n-1$:
2. $\quad m \leftarrow i$ ◁ *m bude index nejmenšího dosud nalezeného prvku*
3. $\quad$ Pro $j = i+1, \ldots, n$:
4. $\quad\quad$ Pokud je $P[j] < P[m]$: $m \leftarrow j$
5. $\quad$ Prohodíme prvky $P[i]$ a $P[m]$. ◁ *pokud $i = m$, nic se nestane*

Výstup: Setříděné pole P

V i-tém průchodu vnějším cyklem hledáme minimum z $n-i+1$ čísel, na což potřebujeme čas $\Theta(n-i+1)$. Ve všech průchodech dohromady tedy spotřebujeme čas $\Theta(n+(n-1)+\ldots+3+2) = \Theta(n \cdot (n-1)/2) = \Theta(n^2)$.

Bubblesort – bublinkové třídění

Další z rodiny přímých algoritmů je *bublinkové třídění (Bubblesort).* Jeho základem je myšlenka nechat stoupat větší prvky v poli podobně, jako stoupají bublinky v limonádě.

V algoritmu budeme opakovaně procházet celé pole. Jeden průchod postupně porovná všechny dvojice sousedních prvků $P[i]$ a $P[i+1]$. Pokud dvojice není správně uspořádaná (tedy $P[i] > P[i+1]$), prvky prohodíme. V opačném případě necháme dvojici na pokoji. Menší prvky se nám tak posunou blíže k začátku pole, zatímco větší prvky „bublají“ na jeho konec. Pokaždé, když pole projdeme celé, začneme znovu od začátku. Tyto průchody opakujeme, dokud dochází k prohazování prvků. V okamžiku, kdy výměny ustanou, je pole setříděné.

Algoritmus BubbleSort (bublinkové třídění)
Vstup: Pole $P[1 \ldots n]$

1. *pokračuj* $\leftarrow 1$ ◁ *má proběhnout další průchod?*

2. Dokud je *pokračuj* = 1:
3. *pokračuj* ← 0
4. Pro $i = 1, \ldots, n-1$:
5. Pokud je $P[i] > P[i+1]$:
6. Prohodíme prvky $P[i]$ a $P[i+1]$.
7. *pokračuj* ← 1

Výstup: Setříděné pole P

Jeden průchod vnitřním cyklem (kroky 4 až 7) jde přes všechny prvky pole, takže má určitě složitost $\Theta(n)$. Není ovšem na první pohled zřejmé, kolik průchodů bude potřeba vykonat. To nahlédneme následovně:

Lemma: Po k-tém průchodu vnějším cyklem je na správných místech k největších prvků.

Důkaz: Indukcí podle k. V prvním průchodu se největší prvek dostane na samý konec pole. Na začátku k-tého průchodu je podle indukčního předpokladu na správných místech $k-1$ největších prvků. Během k-tého průchodu tyto prvky na svých místech zůstanou, takže průchod pracuje pouze s prvky na prvních $n-k$ pozicích. Mezi nimi správně najde maximum a přesune ho na konec. Toto maximum je právě k-tý největší prvek. Tím je indukční krok hotov. □

Cvičení

1. Nahlédněte, že v k-tém průchodu Bubblesortu stačí zkoumat prvky na pozicích $1, \ldots, n-k+1$. Změní se touto úpravou časová složitost?

2. Na jakých datech provede Bubblesort pouze jeden průchod? Na jakých právě dva průchody? Kolik přesně průchodů vykoná nad sestupně uspořádaným vstupem?

3. Všimněme si, že Bubblesort může provádět spoustu zbytečných porovnání. Například když bude první polovina pole setříděná a až druhá rozházená, Bubblesort bude stejně vždy procházet první polovinu, i když v ní nebude nic prohazovat. Navrhněte možná vylepšení, abyste eliminovali co nejvíce zbytečných porovnání.

4. Určete průměrnou časovou složitost Bubblesortu (v průměru přes všechny možné permutace prvků na vstupu).

5. *Insertsort* neboli třídění přímým vkládáním funguje takto: Udržujeme dvě části pole – na začátku leží setříděné prvky a v druhé části pak zbývající nesetříděné. V každém kroku vezmeme jeden prvek z nesetříděné části a vložíme jej na správné místo v části setříděné. Dopracujte detaily algoritmu a analyzujte jeho složitost.

6. Předpokládejme na chvíli, že by počítač, na kterém běží naše programy, uměl provést operaci posunutí celého úseku pole o 1 prvek na libovolnou stranu v konstantním

čase. Řekli bychom například, že chceme prvky na pozicích 42 až 54 posunout o 1 doprava (tj. na pozice 43 až 55), a počítač by to uměl provést v jednom kroku. Zkuste za těchto podmínek upravit Insertsort, aby pracoval s časovou složitostí $\Theta(n \log n)$.

7. Určete, jakou složitost bude mít Insertsort, pokud víme, že se setříděním každý prvek posune nejvýše o k pozic. Záleží na způsobu, jakým hledáme místo k zatřídění prvku?

8. Dokažte, že za stejných podmínek jako v předchozím cvičení provede Bubblesort nejvýše k průchodů. Pokud vám to dělá potíže, ukažte alespoň, že stačí $\mathcal{O}(k)$ průchodů.

9.* Navrhněte pro úlohu z cvičení 7 efektivnější algoritmus. Můžete se inspirovat třeba Heapsortem z oddílu 4.2.

10. Upravte třídicí algoritmy z tohoto oddílu, aby byly stabilní.

3.2 Třídění sléváním

Nyní představíme třídící algoritmus s časovou složitostí $\Theta(n \log n)$. Na vstupu je dána n-prvková posloupnost $a_0, \ldots, a_{n-1}$ v poli. Pro jednoduchost nejprve předpokládejme, že n je mocnina dvojky.

Základní myšlenka algoritmu je tato: Pole rozdělíme do tzv. *běhů* o délce mocniny dvou – souvislých úseků, které už jsou vzestupně setříděny. Na začátku budou všechny běhy jednoprvkové. Poté budeme dohromady slévat vždy dva sousední běhy do jediného setříděného běhu o dvojnásobné délce, který bude ležet na místě obou vstupních běhů. To znamená, že v i-té iteraci budou mít běhy délku 2^i prvků a jejich počet bude $n/2^i$. V poslední iteraci bude posloupnost sestávat z jediného běhu, a bude tudíž setříděná.

Algoritmus MergeSort1
Vstup: Posloupnost $a_0, \ldots, a_{n-1}$ k setřídění

```
1.  b ← 1                                          ◁ aktuální délka běhu
2.  Dokud b < n:                                   ◁ zbývají aspoň dva běhy
3.      Pro i = 0, 2b, 4b, 6b, ..., n − 2b:        ◁ začátky sudých běhů
4.          X ← (a_i, ..., a_{i+b−1})              ◁ sudý běh
5.          Y ← (a_{i+b}, ..., a_{i+2b−1})         ◁ následující lichý běh
6.          (a_i, ..., a_{i+2b−1}) ← Merge(X, Y)
7.      b ← 2b
```

Výstup: Setříděná posloupnost $a_0, \ldots, a_{n-1}$

Procedura MERGE se stará o samotné slévání. To zařídíme snadno: Pokud chceme slít posloupnosti $x_1 \leq x_2 \leq \ldots \leq x_m$ a $y_1 \leq y_2 \leq \ldots \leq y_n$, bude výsledná posloupnost začínat menším z prvků x_1 a y_1. Tento prvek z příslušné vstupní posloupnosti přesuneme na výstup a pokračujeme stejným způsobem. Pokud to byl (řekněme) prvek x_1, zbývá nám slít $x_2, \ldots, x_m$ s $y_1, \ldots, y_n$. Dalším prvkem výstupu tedy bude minimum z x_2 a y_1. To opět přesuneme a tak dále, než se buď x nebo y vyprázdní.

Procedura MERGE (slévání)
Vstup: Běhy $x_1, \ldots, x_m$ a $y_1, \ldots, y_n$

1. $i \leftarrow 1$, $j \leftarrow 1$ ◁ *zbývá slít* $x_i, \ldots, x_m$ *a* $y_j, \ldots, y_n$
2. $k \leftarrow 1$ ◁ *výsledek se objeví v* $z_k, \ldots, z_{m+n}$
3. Dokud $i \leq m$ a $j \leq n$, opakujeme:
4. Je-li $x_i \leq y_j$, přesuneme prvek z x: $z_k \leftarrow x_i$, $i \leftarrow i + 1$.
5. Jinak přesouváme z y: $z_k \leftarrow y_j$, $j \leftarrow j + 1$.
6. $k \leftarrow k + 1$
7. Je-li $i \leq m$, zkopírujeme zbylá x: $z_k, \ldots, z_{m+n} \leftarrow x_i, \ldots, x_m$.
8. Je-li $j \leq n$, zkopírujeme zbylá y: $z_k, \ldots, z_{m+n} \leftarrow y_j, \ldots, y_n$.

Výstup: Běh $z_1, \ldots, z_{m+n}$

Nyní odvodíme asymptotickou složitost algoritmu MERGESORT. Začneme funkcí MERGE: ta pouze přesouvá prvky a každý přesune právě jednou. Její časová složitost je tedy $\Theta(n + m)$, v i-té iteraci proto na slití dvou běhů spotřebuje čas $\Theta(2^i)$. V rámci jedné iterace se volá MERGE řádově $n/2^i$-krát, což dává celkem $\Theta(n)$ operací na iteraci. Protože se algoritmus zastaví, když $2^i = n$, počet iterací bude $\Theta(\log n)$, což dává celkovou časovou složitost $\Theta(n \log n)$.

Mergesort bohužel neumí třídit na místě: při slévání musí být zdrojové běhy uloženy jinde než cílový běh. Proto potřebujeme pomocnou paměť velikosti $\mathcal{O}(n)$, například v podobě pomocného pole stejné velikosti jako vstupní pole.

Nyní doplníme, co si počít, když počet prvků není mocninou dvojky. Pořád budeme velikost běhů zvyšovat po mocninách dvojky, ale připustíme, že poslední z běhů může být menší a že celkový počet běhů může být lichý. Proto bude platit, že v i-té iteraci činí počet běhů $\lceil n/2^i \rceil$, takže po $\Theta(\log n)$ iteracích se algoritmus zastaví. Implementace je jednoduchá, jak je ostatně vidět z následujícího pseudokódu.

Algoritmus MERGESORT (třídění sléváním)
Vstup: Posloupnost $a_1, \ldots, a_n$ k setřídění

1. $b \leftarrow 1$ ◁ *aktuální délka běhu*
2. Dokud $b < n$: ◁ *zbývají aspoň dva běhy*

3. $i \leftarrow 1$, $j \leftarrow b+1$ ◁ *začátek aktuálního sudého a lichého běhu*
4. Dokud $j \leq n$: ◁ *ještě zbývá nějaká dvojice běhů*
5. $k \leftarrow \min(j+b-1, n)$ ◁ *konec lichého běhu*
6. $X \leftarrow (a_i, \ldots, a_{j-1})$ ◁ *sudý běh*
7. $Y \leftarrow (a_j, \ldots, a_k)$ ◁ *lichý běh*
8. $(a_i, \ldots, a_k) \leftarrow \textsc{Merge}(X, Y)$
9. $i \leftarrow i+2b$, $j \leftarrow j+2b$ ◁ *posuneme se na další dvojici*
10. $b \leftarrow 2b$

Výstup: Setříděná posloupnost $a_1, \ldots, a_n$

Na závěr dodejme, že Mergesort také můžeme formulovat jako elegantní rekurzivní algoritmus. S tímto přístupem se setkáme v kapitole 10.

Cvičení

1. Navrhněte algoritmus pro efektivní třídění dat uložených v jednosměrném spojovém seznamu. Algoritmus smí používat pouze $\mathcal{O}(1)$ buněk pomocné paměti a jednotlivé položky seznamu smí pouze přepojovat, nikoliv kopírovat na jiné místo v paměti.

2. Je Mergesort stabilní? Pokud ne, uměli byste ho upravit, aby stabilní byl?

3.* *Knížky v knihovně:* Mějme posloupnost, která vznikla ze setříděné tím, že jsme přesunuli k prvků. Navrhněte algoritmus, který ji co nejrychleji dotřídí. Pozor na to, že k předem neznáme. Můžete nicméně předpokládat, že k je mnohem menší než délka posloupnosti.

4. *Pišvejcova čísla* říkejme číslům tvaru $2^i 3^j 5^k$. Vymyslete, jak co nejrychleji vygenerovat prvních n Pišvejcových čísel.

5.** *Mergesort na místě:* Na naší verzi Mergesortu je nešikovné, že potřebuje lineární množství pomocné paměti, neboť neumíme slévat na místě. Jde to i lépe: paměťové nároky procedury Merge lze srazit až na konstantu při zachování lineární časové složitosti. Je to ale docela složité (a v praxi se to kvůli vysokým konstantám nevyplatí), tak zkuste přijít na slévání v lineárním čase a pomocné paměti velikosti $\mathcal{O}(\sqrt{n})$. Existuje i verze, která slévá současně na místě a stabilně.

3.3 Dolní odhad složitosti třídění

Jak už jsme zmínili v úvodu kapitoly, nabízí se otázka, zda je možné třídit v čase rychlejším než $\Theta(n \log n)$. Nyní dokážeme, že odpověď je negativní, ale budeme k tomu potřebovat dva předpoklady:

- Algoritmus pracuje v porovnávacím modelu, smí tedy tříděné prvky pouze vzájemně porovnávat a přesouvat (přiřazovat).
- Algoritmus je deterministický – každý krok je jednoznačně určen výsledky kroků předchozích (algoritmus tedy nepoužívá žádný zdroj náhody).

Vyhledávání

Nejprve dokážeme jednodušší výsledek pro vyhledávání. Budeme se snažit ukázat, že binární vyhledávání je optimální (asymptoticky, tedy až na multiplikativní konstantu). Připomeňme, že jeho časová složitost je $\mathcal{O}(\log n)$. Zjistíme, že každý algoritmus potřebuje $\Omega(\log n)$ porovnání, tedy musí celkově provést $\Omega(\log n)$ operací.

Věta (o složitosti vyhledávání): Každý deterministický algoritmus v porovnávacím modelu, který nalezne zadaný prvek v n-prvkové uspořádané posloupnosti, použije v nejhorším případě $\Omega(\log n)$ porovnání.

Myšlenka důkazu: Porovná-li algoritmus nějaká dvě čísla x a y, dozví se jeden ze tří možných výsledků: $x < y$, $x > y$, $x = y$. Navíc nastane-li rovnost, výpočet skončí, takže všechna porovnání kromě posledního dávají jenom dva možné výsledky. Jedním porovnáním tedy získáme jeden bit informace. Žádná jiná operace nám o vztahu hledaného čísla s prvky posloupnosti nic neřekne.

Výstupem vyhledávacího algoritmu je pozice hledaného prvku v posloupnosti; to je číslo od 1 do n, na jehož určení je potřeba $\log_2 n$ bitů. Abychom ho určili, musíme tedy porovnat alespoň $(\log_2 n)$-krát. □

V této úvaze nicméně spoléháme na intuitivní představu o množství informace – poctivou teorii informace jsme nevybudovali. Větu proto dokážeme formálněji zkoumáním možných průběhů algoritmu.

Důkaz: Zvolíme pevné n a vstupní posloupnost $1, \ldots, n$. Budeme zkoumat, jak se výpočet algoritmu vyvíjí pro jednotlivá hledaná čísla $x = 1, \ldots, n$. Pokud ukážeme, že je potřeba provést $\Omega(\log n)$ porovnání pro vstupy tohoto speciálního typu, tím spíš to bude platit v nejhorším případě.

Spustíme algoritmus. Zpočátku jeho výpočet nezávisí na x (zatím jsme neprovedli jediné porovnání), takže první porovnání, které provede, bude vždy stejné. Pokud je to porovnání typu $a_i < a_j$, dopadne také vždy stejně. Až první porovnání typu $a_i < x$ může pro různá x dopadnout různě.

Pro každý z možných výsledků porovnání ale algoritmus pokračuje deterministicky, takže je opět jasné, jaké další porovnání provede. A tak dále, až se algoritmus rozhodne zastavit a vydat výsledek.

Možné průběhy výpočtu tedy můžeme popsat takzvaným *rozhodovacím stromem.* V každém vnitřním vrcholu tohoto stromu je jedno porovnání typu $x < a_i$. Vrchol má tři syny, kteří odpovídají možným výsledkům tohoto porovnání (menší, větší, rovno). Může se stát, že některé z výsledků nemohou nastat, protože by byly ve sporu s dříve provedenými porovnáními. V takovém případě příslušného syna vynecháme.

V listech rozhodovacího stromu jsou jednotlivé výsledky algoritmu: některé listy odpovídají nahlášení výskytu na nějaké pozici, v jiných odpovídáme, že prvek nebyl nalezen.

Rozhodovací strom je tedy ternární strom (vrcholy mají nejvýše 3 syny) s alespoň n listy. Použijeme následující lemma:

Lemma T: Ternární strom hloubky k má nejvýše 3^k listů.

Důkaz: Uvažme ternární strom hloubky k s maximálním počtem listů. V takovém stromu budou všechny listy určitě ležet na poslední hladině (kdyby neležely, můžeme pod některý list na vyšší hladině přidat další tři vrcholy a získat tak „listnatější“ strom stejné hloubky). Jelikož na i-té hladině je nejvýše 3^i vrcholů, všech listů je nejvýše 3^k. □

Strom má proto hloubku alespoň $\log_3 n$, takže v něm existuje cesta z kořene do listu, která obsahuje alespoň $\log_3 n$ vrcholů. Tudíž existuje vstup, na němž algoritmus provede alespoň logaritmicky mnoho porovnání. □

Třídění

Nyní použijeme podobnou metodu pro odhad složitosti třídění. Opět budeme předpokládat speciální typ vstupů, totiž permutace množiny $\{1, \ldots, n\}$. Různé permutace je přitom potřeba třídit různými posloupnostmi prohození, takže algoritmus musí správně rozpoznat, o kterou permutaci se jedná.

I zde funguje intuitivní úvaha o množství informace: jelikož jsou všechny prvky na vstupu různé, jedním porovnáním získáme nejvýše 1 bit informace. Všech permutací je $n!$, takže potřebujeme získat $\log_2(n!)$ bitů. Zbytek zařídí následující lemma:

Lemma F: $n! \geq n^{n/2}$.

Důkaz: Je $n! = \sqrt{(n!)^2} = \sqrt{1 \cdot n \cdot 2 \cdot (n-1) \cdot \ldots \cdot n \cdot 1}$, což můžeme také zapsat jako $\sqrt{1 \cdot n} \cdot \sqrt{2 \cdot (n-1)} \cdot \ldots \cdot \sqrt{n \cdot 1}$. Přitom pro každé $1 \leq k \leq n$ je $k(n+1-k) = kn+k-k^2 = n + (k-1)n + k(1-k) = n + (k-1)(n-k) \geq n$. Proto je každá z odmocnin větší nebo rovna $n^{1/2}$ a $n! \geq (n^{1/2})^n = n^{n/2}$. □

Počet potřebných bitů, a tím pádem i potřebných porovnání, tedy musí být $\log_2(n!) \geq \log_2(n^{n/2}) = (n/2) \cdot \log_2 n = \Omega(n \log n)$. Nyní totéž precizněji...

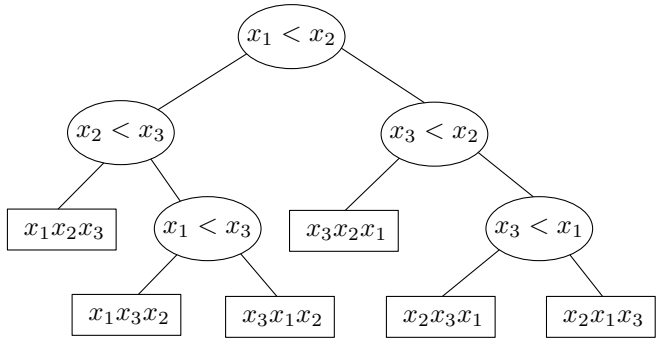
Věta (o složitosti třídění): Každý deterministický algoritmus v porovnávacím modelu, který třídí n-prvkovou posloupnost, použije v nejhorším případě $\Omega(n \log n)$ porovnání.

Důkaz: Jak už jsme naznačili, budeme uvažovat vstupy $a_1, \ldots, a_n$, které jsou permutacemi množiny $\{1, \ldots, n\}$. Stačí nám najít jeden „těžký“ vstup, pokud ho najdeme mezi permutacemi, úkol jsme splnili.

Mějme nějaký třídicí algoritmus. Upravíme ho tak, aby nejprve prováděl všechna porovnaní, a teprve pak prvky přesouval. Můžeme si například pro každou pozici v poli pamatovat, kolikátý z prvků a_i na ni zrovna je. Průběžné prohazování bude pouze měnit tyto pomocné údaje a skutečná prohození provedeme až na konci.

Nyní sestrojíme rozhodovací strom popisující všechny možné průběhy algoritmu. Ve vnitřních vrcholech budou porovnání typu $a_i < a_j$ se třemi možnými výsledky. Opět vynecháme výsledky, které jsou ve sporu s předchozími porovnáními. V listech stromu algoritmus provede nějaká prohození a zastaví se.

Pro různé vstupní permutace musí výpočet skončit v různých listech (dvě permutace nelze setřídit toutéž posloupností přesunů prvků). Listů je tedy alespoň $n!$, takže podle lemmatu **T** musí mít strom hloubku alespoň $\log_3(n!)$, což je podle lemmatu **F** $\Omega(n \log n)$. Proto musí existovat vstup, na němž se provede $\Omega(n \log n)$ porovnání. □



Obrázek 3.1: Příklad rozhodovacího stromu pro 3 prvky

Dokázali jsme tedy, že algoritmus Mergesort je optimální (až na multiplikativní konstantu). Brzy ale uvidíme, že oprostíme-li se od porovnávacího modelu, lze v některých případech třídit rychleji.

Cvičení

1. Jsou dány rovnoramenné váhy a 3 různě těžké kuličky. Ukažte, že je není možné uspořádat dle hmotnosti na méně než 3 vážení. Co se změní, pokud je cílem pouze najít nejtěžší kuličku?

2. Jsou dány rovnoramenné váhy a 12 kuliček, z nichž právě jedna je těžší než ostatní. Na misku lze dát i více kuliček naráz. Navrhněte, jak na 3 vážení najít těžší kuličku. Dokažte, že na 2 vážení to není možné.

3. Jsou dány rovnoramenné váhy a 12 kuliček, z nichž právě jedna je jiná než ostatní, nevíme však zda je lehčí nebo těžší. Na misku lze dát i více kuliček naráz. Navrhněte, jak na 3 vážení najít tuto jinou kuličku a zjistit, jestli je lehčí nebo těžší. Dokažte, že na 2 vážení to nejde.

4. Stejná úloha jako předchozí, avšak s 13 kuličkami. Dokažte, že stále stačí 3 vážení, pokud slevíme z požadavku zjistit, zda je odlišná kulička lehčí nebo těžší.

5. Řešte cvičení 2 obecně pro n kuliček a navrhněte algoritmus používající co nejmenší počet vážení. Dokažte, že tento počet je optimální.

6.* Řešte cvičení 3 obecně pro n kuliček. Uměli byste dokázat, že váš algoritmus je optimální?

7. Uvažme verzi komparátoru, který rozlišuje pouze $a_i \leq a_j$ a $a_i > a_j$. Projděte algoritmy a důkazy vět v této kapitole a modifikujte je pro tento model.

8. *Průměrná složitost:* Dokažte, že $\Omega(\log n)$ resp. $\Omega(n \log n)$ porovnání je potřeba nejen v nejhorším případě, ale i v průměru přes všechny možné vstupy. V případě vyhledávání průměrujeme přes všechna možná hledaná x, u třídění přes všechny permutace.

9. *Matice:* Mějme matici A tvaru $n \times n$, v níž jsou uložena celá čísla a navíc každý řádek i sloupec tvoří rostoucí posloupnost. Jak najít i, j takové, že $A_{i,j} = i + j$? Pokud existuje více řešení, stačí vypsat jedno. Čas na načtení matice do paměti nepočítáme.

10.* Dokažte, že vaše řešení předchozí úlohy je asymptoticky nejrychlejší možné.

3.4 Přihrádkové třídění

Dosud jsme se snažili navrhovat třídicí algoritmy tak, aby si poradily s libovolným typem dat. Proto jsme prvky posloupnosti byli ochotni pouze porovnávat. Nyní se zaměříme na konkrétní druhy dat, například celá kladná čísla z předem daného intervalu.

Counting sort – třídění počítáním

Představme si, že třídíme n celých čísel vybraných z množiny $\{1, \ldots, r\}$ pro nepříliš velké r. Jak vypadá setříděná posloupnost? Nejdřív v ní jsou nějaké jedničky, pak dvojky, atd. Stačí tedy zjistit, kolik má být kterých, čili spočítat, kolikrát se každé číslo od 1 do r na vstupu vyskytuje. Tomuto primitivnímu, ale překvapivě účinnému algoritmu se říká *třídění počítáním* neboli *Counting sort.*

Algoritmus CountingSort (třídění počítáním)
Vstup: Posloupnost $x_1, \ldots, x_n \in \{1, \ldots, r\}$

1. Pro $i = 1, \ldots, r$: ◁ *Inicializujeme počítadla*
2. $\quad p_i \leftarrow 0$
3. Pro $i = 1, \ldots, n$: ◁ p_j *bude počet výskytů čísla* j
4. $\quad p_{x_i} \leftarrow p_{x_i} + 1$
5. $j \leftarrow 1$ ◁ *pozice ve výstupu*
6. Pro $i = 1, \ldots, r$: ◁ *zapisujeme výstup*
7. $\quad$ Opakujeme p_i-krát:
8. $\quad\quad v_j \leftarrow i$
9. $\quad\quad j \leftarrow j + 1$

Výstup: Setříděná posloupnost $v_1, \ldots, v_n$

Inicializace počítadel v krocích 1 a 2 trvá $\Theta(r)$, počítání výskytů v dalších dvou krocích $\Theta(n)$. V krocích 6 až 9 znovu zapíšeme všech n prvků a navíc musíme jednou sáhnout na každou (i prázdnou) přihrádku. Časová složitost celého algoritmu tedy činí $\Theta(n + r)$.

Vstup a výstup mohou být uloženy v tomtéž poli, takže pracovní prostor algoritmu tvoří pouze r počítadel, tedy $\Theta(r)$ buněk paměti.

Bucketsort – přihrádkové třídění

Counting sort nám nepomůže, pokud místo celých čísel třídíme nějaké složitější záznamy, které kromě celočíselného *klíče,* podle nějž třídíme, obsahují navíc nějaká další data. Tehdy můžeme použít podobný algoritmus, kterému se říká *přihrádkové třídění (Bucketsort).*[2] Místo počítadel si pořídíme pole r přihrádek $P_1, \ldots, P_r$. V i-té přihrádce se bude nacházet seznam záznamů s klíčem i.

Algoritmus nejprve projde všechny záznamy a rozmístí je do přihrádek podle klíčů. Poté postupně projde přihrádky od P_1 do P_r a vypíše jejich obsah.

[2] *To bychom mohli přeložit jako „kbelíkové třídění“.*

Algoritmus BucketSort (přihrádkové třídění)
Vstup: Prvky $x_1, \ldots, x_n$ s klíči $k_1, \ldots, k_n \in \{1, \ldots, r\}$
1. Inicializujeme přihrádky: $P_1, \ldots, P_r \leftarrow \emptyset$
2. Pro $i = 1, \ldots, n$:
3. Vložíme x_i do P_{k_i}.
4. Vytvoříme prázdný seznam S.
5. Pro $j = 1, \ldots, r$:
6. Na konec seznamu S připojíme obsah P_j.

Výstup: Setříděný seznam S

Rozbor časové složitosti proběhne podobně jako u předchozího algoritmu: inicializace stojí $\Theta(r)$, rozmisťování do přihrádek $\Theta(n)$, procházení přihrádek $\Theta(r)$ a vypisování všech přihrádek dohromady $\Theta(n)$. Celkem tedy $\Theta(n + r)$.

V paměti máme uložené všechny přihrádky, jedna zabere konstantní prostor na hlavičku seznamu a pak konstantní na každý záznam. To celkem činí $\Theta(n + r)$ buněk paměti.

Bucketsort dokonce může třídit stabilně. K tomu stačí, abychom v každé přihrádce dodrželi vzájemné pořadí prvků. Nové záznamy tedy budeme přidávat na konec seznamu.

Lexikografický Bucketsort

Nyní uvažme případ, kdy klíče nejsou malá celá čísla, nýbrž uspořádané k-tice takových čísel. Úkolem je seřadit tyto k-tice *lexikograficky (slovníkově):* nejprve podle první souřadnice, v případě shody podle druhé, a tak dále.

Praktičtější je postupovat opačně: nejprve k-tice setřídit podle poslední souřadnice, pak je stabilně setřídit podle předposlední, … až nakonec podle první. Díky stabilitě získáme lexikografické pořadí. Stačí tedy k-krát aplikovat předchozí algoritmus přihrádkového třídění.

Algoritmus LexBucketSort
Vstup: Posloupnost k-tic $x_1, \ldots, x_n \in \{1, \ldots, r\}^k$
1. $S \leftarrow x_1, \ldots, x_n$
2. Pro $i = k, k-1, \ldots, 1$:
3. Setřídíme S BucketSortem podle i-té souřadnice.

Výstup: Lexikograficky setříděná posloupnost S

Nyní dokážeme korektnost tohoto algoritmu. Pro přehlednost budeme písmenem ℓ značit, v kolikátém průchodu cyklem jsme. Bude tedy $\ell = k - i + 1$.

Zadaná posloupnost:	173, 753, 273, 351, 171, 172, 069
Po 1. průchodu:	351, 171, 172, 173, 753, 273, 069
Po 2. průchodu:	351, 753, 069, 171, 172, 173, 273
Po 3. průchodu:	069, 171, 172, 173, 273, 351, 753

Obrázek 3.2: Příklad lexikografického přihrádkového třídění trojic

Lemma: Po ℓ-tém průchodu cyklem jsou prvky uspořádány lexikograficky podle i-té až k-té souřadnice.

Důkaz: Indukcí podle ℓ:

- Pro $\ell = 1$ jsou prvky uspořádány podle poslední souřadnice.
- Po ℓ průchodech již máme prvky setříděny lexikograficky podle i-té až k-té souřadnice. Spouštíme $(\ell+1)$-ní průchod, tj. budeme třídit podle $(i-1)$-ní souřadnice. Protože Bucketsort třídí stabilně, zůstanou prvky se stejnou $(i-1)$-ní souřadnicí vůči sobě seřazeny tak, jak byly seřazeny na vstupu. Z indukčního předpokladu tam však byly seřazeny lexikograficky podle i-té až k-té souřadnice. Tudíž po $(\ell+1)$-ním průchodu jsou prvky seřazeny podle $(i-1)$-ní až k-té souřadnice. □

Časová složitost je k-násobkem složitosti Bucketsortu, tedy $\Theta(k \cdot (n + r))$. Paměti spotřebujeme $\Theta(nk + r)$: první člen je paměť zabraná samotnými záznamy, druhý paměť potřebná na uložení pole přihrádek.

Radixsort

Přihrádkové třídění pro klíče z rozsahu $1, \ldots, r$ není efektivní, pokud r je řádově větší než počet záznamů. Tehdy totiž časové složitosti vévodí čas potřebný na inicializaci a procházení přihrádek. Můžeme si ale pomoci následovně.

Čísla zapíšeme v soustavě o vhodném základu z. Z každého čísla se tak stane k-tice cifer z rozsahu $0, \ldots, z-1$, kde $k = \lfloor \log_z r \rfloor + 1$. Tyto k-tice pak stačí setřídit lexikograficky, což zvládneme k-průchodovým přihrádkovým tříděním v čase $\Theta((\log_z r) \cdot (n + z)) = \Theta(\frac{\log r}{\log z} \cdot (n + z))$.

Jak zvolit základ z? Pokud bychom si vybrali konstantní (třeba pokaždé čísla zapisovali v desítkové soustavě), časová složitost by vyšla $\Theta(\log r \cdot n)$. To není moc zajímavé, jelikož pro navzájem různé klíče máme $r \geq n$, takže jsme nepřekonali složitost porovnávacích algoritmů.

Užitečnější je zvolit $z = \Theta(n)$. Pak dosáhneme složitosti $\Theta(\frac{\log r}{\log n} \cdot n)$. Pokud by čísla na vstupu byla polynomiálně velká vzhledem k n, tedy $r \leq n^\alpha$ pro nějaké pevné α, byl by

$\log r \leq \alpha \log n$, takže časová složitost by vyšla lineární. Polynomiálně velká celá čísla jde tedy třídit v lineárním čase (a také lineárním prostoru).

Tomuto algoritmu se říká *číslicové třídění* neboli *Radixsort.*⟨3⟩

Třídění řetězců

Myšlenku víceprůchodového přihrádkového třídění použijeme ještě k řazení řetězců znaků. Chceme je uspořádat lexikograficky – to definujeme stejně jako pro k-tice, jen navíc musíme říci, že pokud při porovnávání jeden řetězec skončí dřív než druhý, ten kratší bude menší.

Na vstupu jsme tedy dostali nějakých n řetězců $r_1, \ldots, r_n$ délek po řadě $\ell_1, \ldots, \ell_n$. Navíc označme $\ell = \max_i \ell_i$ délku nejdelšího řetězce a $s = \sum_i (\ell_i + 1)$ celkovou velikost vstupu (+1 kvůli uložení délky řetězce – musíme umět uložit i prázdný řetězec). Budeme předpokládat, že znaky abecedy máme očíslované od 1 do nějakého r.

Kdyby všechny řetězce byly stejně dlouhé, stačilo by je třídit jako k-tice. To by mělo složitost $\Theta(\ell n) = \Theta(s)$, tedy lineární v celkové velikosti vstupu.

S různě dlouhými řetězci bychom se mohli vypořádat tak, že bychom všechny doplnili mezerami na stejnou délku (mezerou myslíme nějaký znak, který je menší než všechny ostatní znaky). Složitost algoritmu bude nadále $\Theta(\ell n)$, ale to v některých případech může být i kvadratické ve velikosti vstupu. Uvažme třeba případ s t řetězci délky 1 a jedním délky t. To je celkem $n = t + 1$ řetězců o celkové délce $s = 2t - 1$. Třídění ovšem zabere čas $\Theta(\ell n) = \Theta(t^2) = \Theta(s^2)$.

Okamžitě vidíme, kde se většina času tráví: přidáváme ohromné množství mezer a pak ve většině průchodů většinu řetězců házíme do přihrádky odpovídající mezeře. Zkusme algoritmus vylepšit, aby mezery přidával jen pomyslně a řetězce, které jsou příliš krátké, v počátečních průchodech vůbec neuvažoval.

Začneme tím, že řetězce roztřídíme Bucketsortem do přihrádek (množin) P_j podle délky. V j-té přihrádce tedy skončí řetězce délky j. Při tom také spočítáme ℓ.

Pak budeme provádět ℓ průchodů přihrádkového třídění pro $i = \ell, \ell - 1, \ldots, 1$. Během nich budeme udržovat seznam Z tak, aby na konci průchodu pro konkrétní i obsahoval všechny řetězce s délkou alespoň i, a to seřazené podle i-tého až posledního znaku. Na konci posledního průchodu tedy budou v Z všechny řetězce seřazené lexikograficky podle všech znaků.

Zbývá popsat, jak jeden průchod pracuje. Použijeme přihrádky indexované znaky abecedy (Q_1 až Q_r) a budeme do nich rozhazovat řetězce podle jejich i-tého znaku. Nejprve roz-

⟨3⟩ *Radix* je latinsky kořen, ale zde se tím myslí základ poziční číselné soustavy.

házíme řetězce délky i z množiny P_i a pak přidáváme všechny, které zůstaly v seznamu Z z předchozího průchodu – kratší řetězce se tak dostaly před delší, jak má být. Nakonec všechny přihrádky vysbíráme a řetězce naskládáme do nového seznamu Z. Indukcí můžeme nahlédnout, že seznam Z splňuje požadované vlastnosti.

Ještě dodejme, že řetězce do přihrádek nebudeme kopírovat celé, znak po znaku. Postačí ukládat ukazatele a samotné řetězce nechávat netknuté ve vstupní paměti.

Algoritmus TříděníŘetězců
Vstup: Řetězce $r_1, \ldots, r_n$ délek $\ell_1, \ldots, \ell_n$ nad abecedou $\{1, \ldots, r\}$

1. $\ell \leftarrow \max(\ell_1, \ell_2, \ldots, \ell_n)$ ◁ *maximální délka*
2. Pro $i \leftarrow 1, \ldots, \ell$: ◁ *rozdělíme podle délek*
3. $\quad P_i \leftarrow \emptyset$
4. Pro $i \leftarrow 1, \ldots, n$ opakujeme:
5. $\quad$ Na konec P_{ℓ_i} přidáme řetězec r_i.
6. $Z \leftarrow \emptyset$ ◁ *výsledek předchozího průchodu*
7. Pro $i \leftarrow \ell, \ell - 1, \ldots, 1$: ◁ *průchody pro jednotlivé délky*
8. $\quad$ Pro $j \leftarrow 1, \ldots, r$: ◁ *inicializace přihrádek*
9. $\quad\quad Q_j \leftarrow \emptyset$
10. $\quad$ Pro řetězce u z přihrádky P_i: ◁ *nové, kratší řetězce*
11. $\quad\quad$ Na konec $Q_{u[i]}$ přidáme řetězec u.
12. $\quad$ Pro řetězce v ze seznamu Z: ◁ *řetězce z minulého průchodu*
13. $\quad\quad$ Na konec $Q_{v[i]}$ přidáme řetězec v.
14. $\quad Z \leftarrow \emptyset$ ◁ *vysbíráme přihrádky*
15. $\quad$ Pro $j \leftarrow 1, \ldots, r$:
16. $\quad\quad$ Pro řetězce w z přihrádky Q_j:
17. $\quad\quad\quad$ Na konec seznamu Z přidáme řetězec w.

Výstup: Setříděný seznam řetězců Z

Ještě stanovíme časovou složitost. Seřazení řetězců podle délky potrvá $\Theta(n + \ell)$. Průchod pro dané i spotřebuje r kroků na práci s přihrádkami a sáhne na ty řetězce, které mají délku alespoň i, tedy mají na pozici i nějaký znak.

V součtu přes všech ℓ průchodů tedy trvá práce s přihrádkami $\Theta(\ell r)$ a práce s řetězci $\Theta(\sum_i \ell_i) = \Theta(s)$ – každé sáhnutí na řetězec můžeme naúčtovat jednomu z jeho znaků.

Celý algoritmus proto běží v čase $\Theta(n+\ell r+s)$. Pokud je abeceda konstantně velká a vstup neobsahuje prázdné řetězce, můžeme tuto funkci zjednodušit na $\Theta(s)$. Algoritmus je tedy lineární ve velikosti vstupu.

Paměti spotřebujeme $\Theta(n+s)$ na řetězce a $\Theta(\ell+r)$ na přihrádky. To můžeme podobně zjednodušit na $\Theta(s)$.

Cvičení

1. *Rekurzivní Bucketsort:* Lexikografické třídění k-tic by se také dalo provést metodou Rozděl a panuj z kapitoly 10: nejprve záznamy rozdělit do přihrádek podle první souřadnice a pak každou přihrádku rekurzivně setřídit podle zbývajících $k-1$ souřadnic. Jakou časovou a prostorovou složitost by měl takový algoritmus?

2. *Bucketsort v poli:* V implementaci Bucketsortu může být nešikovné ukládat přihrádky jako spojové seznamy, protože spotřebujeme hodně paměti na ukazatele. Upravte Bucketsort, aby si vystačil se vstupním polem, výstupním polem a jedním nebo několika r-prvkovými poli.

3. Může se někdy v Radixsortu vyplatit zvolit základ soustavy řádově větší než počet čísel?

4. *Třídění floatů:* Uvažujme čísla typu *floating point* zadaná ve tvaru $m \cdot 2^e$, kde m je celočíselná *mantisa*, e celočíselný *exponent* a platí $2^{24} \leq m < 2^{25}$, $-128 \leq e < 128$. Ověřte, že tato čísla lze ukládat do 32 bitů paměti. Rozmyslete si, jaký rozsah a přesnost mají. Navrhněte co nejrychlejší algoritmus na jejich třídění.

5.* *Velké abecedy:* Algoritmus TŘÍDĚNÍŘETĚZCŮ není efektivní pro velké abecedy, protože tráví příliš mnoho času přeskakováním prázdných přihrádek. Pokuste se předem předpočítat, ve kterém průchodu budou potřeba které přihrádky. Dosáhněte složitosti $\mathcal{O}(r+s)$.

6.** *Dírou* v množině $\{x_1, \ldots, x_n\} \subset \mathbb{Z}$ nazveme dvojici (x_i, x_j) takovou, že $x_i < x_j$ a žádné jiné x_k neleží v intervalu $[x_i, x_j]$. Chceme nalézt největší z děr (s maximálním $x_j - x_i$). Setříděním množiny to jde snadno, ale existuje i lineární algoritmus založený na šikovném dělení do přihrádek. Zkuste na něj přijít.

3.5 Přehled třídicích algoritmů

Na závěr uvedeme přehlednou tabulku se souhrnem informací o jednotlivých třídicích algoritmech. Přidáme do ní i několik algoritmů, které představíme až v budoucích kapitolách. U všech algoritmů uvádíme číslo oddílu, kde jsou vyloženy.

Poznámky k tabulce:

- Quicksort má časovou složitost $\Theta(n \log n)$ pouze v průměru. Můžeme ale říci, že porovnáváme průměrné časové složitosti, protože u ostatních algoritmů vyjdou stejně jako jejich časové složitosti v nejhorším případě.

- Mergesort jde implementovat s konstantní pomocnou pamětí za cenu konstantního zpomalení, ovšem konstanta je neprakticky velká. Dále viz cvičení 3.2.5.
- Quicksort se dá naprogramovat stabilně, ale potřebuje lineárně pomocné paměti.
- Multiplikativní konstanta u Heapsortu není příliš příznivá a v běžných situacích tento algoritmus na celé čáře prohrává s efektivnějším Quicksortem.

algoritmus	*čas*	*pomocná paměť*	*stabilní*
Insertsort (3.1)	$\Theta(n^2)$	$\Theta(1)$	+
Bubblesort (3.1)	$\Theta(n^2)$	$\Theta(1)$	+
Mergesort (3.2, 10.2)	$\Theta(n \log n)$	$\Theta(n)$	+
Heapsort (4.2)	$\Theta(n \log n)$	$\Theta(1)$	−
Quicksort (10.7, 11.2)	$\Theta(n \log n)$	$\Theta(\log n)$	−
Bucketsort (3.4)	$\Theta(n + r)$	$\Theta(n + r)$	+
Bucketsort pro k-tice (3.4)	$\Theta(k \cdot (n + r))$	$\Theta(n + r)$	+
Radixsort (3.4)	$\Theta(n \log_n r)$	$\Theta(n)$	+
Bucketsort pro řetězce (3.4)	$\Theta(s)$	$\Theta(s)$	+

Obrázek 3.3: Přehled třídicích algoritmů
(n je počet prvků, r rozsah klíčů, s délka vstupu)

Cvičení

1. Navrhněte algoritmus na zjištění, jestli se v zadané n-prvkové posloupnosti opakují některé prvky.

2.* Dokažte, že problém z předchozí úlohy vyžaduje v porovnávacím modelu čas alespoň $\Theta(n \log n)$.

3. Je dána posloupnost čísel. Najděte nejdelší úsek, v němž se žádné číslo neopakuje. (To je podobné cvičení 1.1.2, ale zde není omezen rozsah prvků.)

4 Datové struktury

4 Datové struktury

V algoritmech potřebujeme zacházet s různými druhy dat – posloupnostmi, množinami, grafy, … Často se nabízí více způsobů, jak tato data uložit do paměti počítače. Jednotlivé způsoby se mohou lišit spotřebou paměti, ale také rychlostí různých operací s daty. Vhodný způsob tedy volíme podle toho, jaké operace využívá konkrétní algoritmus a jak často je provádí.

Otázky tohoto druhu se přitom opakují. Proto je zkoumáme obecně, což vede ke studiu datových struktur. V této kapitole se podíváme na ty nejběžnější z nich.

4.1 Rozhraní datových struktur

Nejprve si rozmyslíme, co od datové struktury očekáváme. Z pohledu programu má struktura jasné rozhraní: reprezentuje nějaký druh dat a umí s ním provádět určité operace. Uvnitř datové struktury pak volíme konkrétní uložení dat v paměti a algoritmy pro provádění jednotlivých operací. Z toho pak plyne prostorová složitost struktury a časová složitost operací.

Fronta a zásobník

Jednoduchým příkladem je *fronta.* Ta si pamatuje posloupnost prvků a umí s ní provádět tyto operace:

ENQUEUE(x)	přidá na konec fronty prvek x
DEQUEUE	odebere prvek ze začátku fronty, případně oznámí, že fronta je prázdná

Pokud frontu implementujeme jako spojový seznam, zvládneme obě operace v konstantním čase a vystačíme si s pamětí lineární v počtu prvků.

Nejbližším příbuzným fronty je *zásobník* – ten si také pamatuje posloupnost a dovede přidávat nové prvky na konec, ale odebírá je z téhož konce. Operaci přidání se obvykle říká PUSH, operaci odebrání POP.

Prioritní fronta

Zajímavější je „fronta s předbíháním“, zvaná též *prioritní fronta.* Každý prvek má přiřazenu číselnou *prioritu* a na řadu vždy přijde prvek s nejvyšší prioritou. Operace vypadají následovně:

ENQUEUE(x, p)	přidá do fronty prvek x s prioritou p
DEQUEUE	nalezne prvek s nejvyšší prioritou a odebere ho (pokud je takových prvků víc, vybere libovolný z nich)

Prioritní frontu lze reprezentovat polem nebo seznamem, ale nalezení maxima z priorit bude pomalé – v n-prvkové frontě $\Theta(n)$. V oddílu 4.2 zavedeme *haldu,* s níž dosáhneme časové složitosti $\mathcal{O}(\log n)$ u obou operací.

Množina a slovník

Množina obsahuje konečný počet prvků vybraných z nějakého *univerza*. Pod univerzem si můžete představit třeba celá čísla, ale nemusíme se omezovat jen na ně. Obecně budeme předpokládat, že prvky univerza je možné v konstantním čase přiřazovat a porovnávat na rovnost a „je menší než".

Množina nabízí následující operace:

MEMBER(x)	zjistí, zda x leží v množině (někdy též FIND(x))
INSERT(x)	vloží x do množiny (pokud tam už bylo, nestane se nic)
DELETE(x)	odebere x z množiny (pokud tam nebylo, nestane se nic)

Zobecněním množiny je *slovník.* Ten si pamatuje konečnou množinu *klíčů* a každému z nich přiřazuje *hodnotu* (to může být prvek nějakého dalšího univerza, nebo třeba ukazatel na jinou datovou strukturu). Slovník je tedy konečná množina dvojic (*klíč*, *hodnota*), v níž se neopakují klíče. Typické slovníkové operace jsou tyto:

GET(x)	zjistí, jaká hodnota je přiřazena klíči x (pokud nějaká)
SET(x, y)	přiřadí klíči x hodnotu y; pokud už nějaká dvojice s klíčem x existovala, tak ji nahradí
DELETE(x)	smaže dvojici s klíčem x (pokud existovala)

Někdy nás také zajímá vzájemné pořadí prvků – tehdy definujeme *uspořádanou množinu,* která má navíc tyto operace:

MIN	vrátí nejmenší hodnotu v množině
MAX	vrátí největší hodnotu v množině
PRED(x)	vrátí největší prvek menší než x, nebo řekne, že takový není
SUCC(x)	vrátí nejmenší prvek větší než x, nebo řekne, že takový není

Obdobně můžeme zavést uspořádané slovníky.

Množinu nebo slovník můžeme reprezentovat pomocí pole. Má to ale své nevýhody: Především potřebujeme dopředu znát horní mez počtu prvků množiny, případně si pořídit „nafukovací" pole (viz oddíl 9.1). Mimo to se s polem pracuje pomalu: množinové operace musí pokaždé projít všechny prvky, což trvá $\Theta(n)$.

Hledání můžeme zrychlit *uspořádáním* (setříděním) pole. Pak může MEMBER binárně vyhledávat v logaritmickém čase, ovšem vkládání i mazání zůstanou lineární.

Použijeme-li *spojový seznam,* všechny operace budou lineární. Uspořádáním seznamu si nepomůžeme, protože v seznamu nelze hledat binárně.

Můžeme trochu podvádět a upravit rozhraní. Kdybychom slíbili, že INSERT nikdy nezavoláme na prvek, který už v množině leží, mohli bychom nový prvek vždy přidat na konec pole či seznamu. Podobně kdyby DELETE dostal místo klíče ukazatel na už nalezený prvek, mohli bychom mazat v konstantním čase (cvičení 2).

Později vybudujeme vyhledávací stromy (kapitola 8) a hešovací tabulky (oddíl 11.3), které budou mnohem efektivnější. Abyste věděli, na co se těšit, prozradíme už teď složitosti jednotlivých operací:

	INSERT	DELETE	MEMBER	MIN	PRED
pole	$\Theta(1)^*$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
uspořádané pole	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$
spojový seznam	$\Theta(1)^*$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
uspořádaný seznam	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
vyhledávací strom	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
hešovací tabulka	$\Theta(1)^\dagger$	$\Theta(1)^\dagger$	$\Theta(1)^\dagger$	$\Theta(n)$	$\Theta(n)$

Operace MAX a SUCC jsou stejně rychlé jako MIN a PRED. Složitosti označené hvězdičkou platí jen tehdy, slíbíme-li, že se prvek v množině dosud nenachází; v opačném případě je potřeba předem provést MEMBER. Složitosti označené křížkem jsou průměrné hodnoty. U polí předpokládáme, že dopředu známe horní odhad velikosti množiny.

Cvičení

1. Navrhněte reprezentaci fronty v poli, která bude pracovat v konstantním čase. Můžete předpokládat, že předem znáte horní odhad počtu prvků ve frontě.

2. Při mazání z pole vznikne díra, kterou je potřeba zaplnit. Jak to udělat v konstantním čase?

3. Množiny čísel můžeme reprezentovat uspořádanými seznamy. Ukažte, jak v této reprezentaci počítat průnik a sjednocení množin v lineárním čase.

4.* Na uspořádané množině můžeme také definovat operaci INDEX(i), která najde i-tý nejmenší prvek. K ní inverzní je operace RANK(x), jež spočítá počet prvků množiny menších než x. Rozmyslete si, jakou složitost tyto dvě operace mají v různých reprezentacích množin.

4.2 Haldy

Jednou z nejjednodušších datových struktur je *halda* (anglicky *heap*), přesněji řečeno *minimová binární halda.* Co taková halda umí? Pamatuje si množinu prvků opatřených klíči a v nejjednodušší variantě nabízí tyto operace:

INSERT(x)	vloží prvek x do množiny
MIN(x)	najde prvek s nejmenším klíčem (pokud je takových víc, pak libovolný z nich)
EXTRACTMIN(x)	odebere prvek s nejmenším klíčem a vrátí ho jako výsledek

Klíč přiřazený prvku x budeme značit $k(x)$. Klíče si můžeme představovat jako celá čísla, ale obecně to mohou být prvky libovolného univerza. Jako obvykle budeme předpokládat, že klíče lze přiřazovat a porovnávat v konstantním čase.

Definice: Strom nazveme *binární*, pokud je zakořeněný a každý vrchol má nejvýše dva syny, u nichž rozlišujeme, který je levý a který pravý. Vrcholy rozdělíme podle vzdálenosti od kořene do *hladin:* v nulté hladině leží kořen, v první jeho synové atd.

Definice: *Minimová binární halda* je datová struktura tvaru binárního stromu, v jehož každém vrcholu je uložen jeden prvek. Přitom platí:

1. *Tvar haldy:* Všechny hladiny kromě poslední jsou plně obsazené. Poslední hladina je zaplněna zleva.
2. *Haldové uspořádání:* Je-li v vrchol a s jeho syn, platí $k(v) \leq k(s)$.

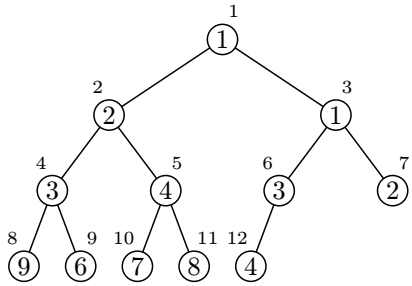
Pozorování: Vydáme-li se z kořene dolů po libovolné cestě, klíče nemohou klesat. Proto se v kořeni stromu musí nacházet jeden z minimálních prvků (těch s nejmenším klíčem; kdykoliv budeme mluvit o porovnávání prvků, myslíme tím podle klíčů).

Podotýkáme ještě, že haldové uspořádání popisuje pouze „svislé“ vztahy. Například o relaci mezi levým a pravým synem téhož vrcholu pranic neříká.

Lemma: Halda s n prvky má $\lfloor \log_2 n \rfloor + 1$ hladin.

Důkaz: Nejprve spočítáme, kolik vrcholů obsahuje binární strom o h úplně plných hladinách: $2^0 + 2^1 + 2^2 + \ldots + 2^{h-1} = 2^h - 1$. Pokud tedy do haldy přidáváme vrcholy po hladinách, nová hladina přibude pokaždé, když počet vrcholů dosáhne mocniny dvojky. □

Haldu jsme sice definovali jako strom, ale díky svému pravidelnému tvaru může být v paměti počítače uložena mnohem jednodušším způsobem. Vrcholy stromu očíslujeme *indexy* $1, \ldots, n$. Číslovat budeme po hladinách shora dolů, každou hladinu zleva doprava. V tomto pořadí můžeme vrcholy uložit do pole a pracovat s nimi jako se stromem. Platí totiž:



Obrázek 4.1: Halda a její očíslování

Pozorování: Má-li vrchol index i, pak jeho levý syn má index $2i$ a pravý syn $2i + 1$. Je-li $i > 1$, otec vrcholu i má index $\lfloor i/2 \rfloor$, přičemž $i \bmod 2$ nám řekne, zda je k otci připojen levou, nebo pravou hranou.

Zatím budeme předpokládat, že dopředu víme, kolik prvků budeme do haldy vkládat, a podle toho zvolíme velikost pole. Později (v oddílu 9.1) ukážeme, jak lze haldu podle potřeby zmenšovat a zvětšovat.

Dodejme ještě, že obdobně můžeme zavést *maximovou haldu,* která používá opačné uspořádání, takže namísto minima umí rychle najít maximum. Všechno, co v této kapitole ukážeme pro minimovou haldu, platí analogicky i pro tu maximovou.

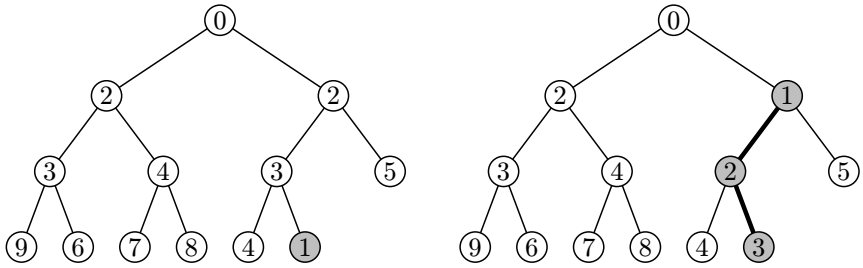
Vkládání

Prázdnou nebo jednoprvkovou haldu vytvoříme triviálně. Uvažujme nyní, jak do haldy přidat další prvek.

Podmínka na tvar haldy nám dovoluje přidat nový list na konec poslední hladiny. Pokud by tato hladina byla plná, můžeme založit novou s jediným listem úplně vlevo. Tím dostaneme strom správného tvaru, ale na hraně mezi novým listem ℓ a jeho otcem o jsme mohli porušit uspořádání, pokud $k(\ell) < k(o)$.

V takovém případě list s otcem prohodíme. Tím jsme chybu opravili, ale mohli jsme způsobit další chybu o hladinu výš. Tu vyřešíme dalším prohozením a tak dále. Nově přidaný prvek bude tedy „vybublávat“ nahoru, potenciálně až do kořene.

Zbývá se přesvědčit, že kdykoliv jsme prohodili otce se synem, nemohli jsme pokazit vztah mezi otcem a jeho druhým synem. To proto, že otec se prohozením zmenšil.



Obrázek 4.2: Vkládání do haldy: začátek a konec

Nyní vkládání zapíšeme v pseudokódu. Budeme předpokládat, že halda je uložena v poli, takže na všechny vrcholy se budeme odkazovat indexy. Prvek na indexu i označíme $p(i)$ a jeho klíč $k(i)$, v proměnné n si budeme pamatovat momentální velikost haldy.

Procedura HeapInsert (vkládání do haldy)
Vstup: Nový prvek p s klíčem k

1. $n \leftarrow n + 1$
2. $p(n) \leftarrow p$, $k(n) \leftarrow k$
3. BubbleUp(n)

Procedura BubbleUp(i)
Vstup: Index i vrcholu se změněným klíčem

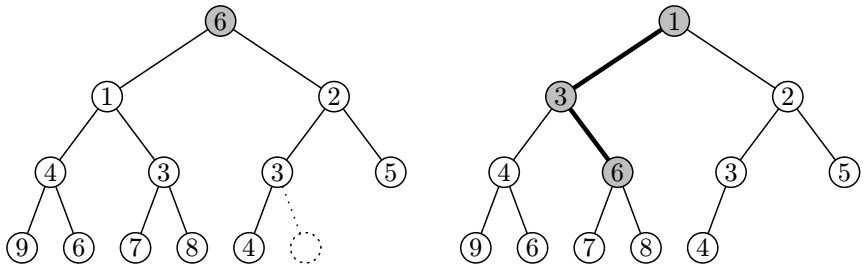
1. Dokud $i > 1$:
2. $\quad o \leftarrow \lfloor i/2 \rfloor$ ◁ *otec vrcholu* i
3. $\quad$ Je-li $k(o) \leq k(i)$, vyskočíme z cyklu.
4. $\quad$ Prohodíme $p(i)$ s $p(o)$ a $k(i)$ s $k(o)$.
5. $\quad i \leftarrow o$

Časovou složitost odhadneme snadno: na každé hladině stromu strávíme nejvýše konstantní čas a hladin je logaritmický počet. Operace Insert tedy trvá $\mathcal{O}(\log n)$.

Hledání a mazání minima

Operace nalezení minima (Min) je triviální, stačí se podívat do kořene haldy, tedy na index 1. Zajímavější bude, když se nám zachce provést ExtractMin čili minimum odebrat. Kořen stromu přímo odstranit nemůžeme. Axiom o tvaru haldy nám nicméně dovoluje beztrestně smazat nejpravější vrchol na nejnižší hladině. Smažeme tedy ten a prvek, který tam byl uložený, přesuneme do kořene.

Opět jsme v situaci, kdy strom má správný tvar, ale může mít pokažené uspořádání. Konkrétně se mohlo stát, že nový prvek v kořeni je větší než některý z jeho synů, možná dokonce než oba. V takovém případě kořen prohodíme s menším z obou synů. Tím jsme opravili vztahy mezi kořenem a jeho syny, ale mohli jsme způsobit obdobný problém o hladinu níže. Pokračujeme tedy stejným způsobem a „zabubláváme“ nově vložený prvek hlouběji, potenciálně až do listu.



Obrázek 4.3: Mazání z haldy: začátek a konec

Procedura HeapExtractMin (mazání minima z haldy)

1. $p \leftarrow p(1)$, $k \leftarrow k(1)$
2. $p(1) \leftarrow p(n)$, $k(1) \leftarrow k(n)$
3. $n \leftarrow n - 1$
4. BubbleDown(1)

Výstup: Prvek p s minimálním klíčem k

Procedura BubbleDown(i)

Vstup: Index i vrcholu se změněným klíčem

1. Dokud $2i \leq n$: ◁ *vrchol i má nějaké syny*
2. $s \leftarrow 2i$
3. Pokud $s + 1 \leq n$ a $k(s + 1) < k(s)$:
4. $s \leftarrow s + 1$
5. Pokud $k(i) < k(s)$, vyskočíme z cyklu.
6. Prohodíme $p(i)$ s $p(s)$ a $k(i)$ s $k(s)$.
7. $i \leftarrow s$

Opět trávíme čas $\mathcal{O}(1)$ na každé hladině, celkem tedy $\mathcal{O}(\log n)$.

Úprava klíče

Doplňme ještě jednu operaci, která se nám bude časem hodit. Budeme jí říkat Decrease a bude mít za úkol snížit klíč prvku, jenž v haldě už je.

Tvar kvůli tomu měnit nemusíme, ale co se stane s uspořádáním? Směrem dolů jsme ho pokazit nemohli, směrem nahoru ano. Jsme tedy ve stejné situaci jako při Insertu, takže stačí zavolat proceduru BubbleUp, aby uspořádání opravila. To stihneme v logaritmickém čase.

Je tu ale jeden zádrhel: musíme vědět, kde se prvek v haldě nachází. Podle klíče vyhledávat neumíme, ovšem můžeme haldu naučit, aby nás informovala, kdykoliv se změní poloha nějakého prvku.

Obdobně můžeme implementovat zvýšení klíče (Increase). Uspořádání se tentokrát bude kazit směrem dolů, takže ho budeme opravovat bubláním v tomto směru.

Všimněte si, že Insert můžeme také popsat jako přidání listu s hodnotou $+\infty$ a následný Decrease. Podobně ExtractMin odpovídá smazání listu a Increase kořene.

Složitost haldových operací shrneme následující větou:

Věta: V binární haldě o n prvcích trvají operace Insert, ExtractMin, Increase a Decrease čas $\mathcal{O}(\log n)$. Operace Min trvá $\mathcal{O}(1)$.

Konstrukce haldy

Pomocí haldy můžeme třídit: vytvoříme prázdnou haldu, do ní Insertujeme tříděné prvky a pak je pomocí ExtractMin vytahujeme od nejmenšího po největší. Jelikož provedeme $2n$ operací s nejvýše n-prvkovou haldou, má tento třídicí algoritmus časovou složitost $\mathcal{O}(n \log n)$.

Samotné vytvoření n-prvkové haldy lze dokonce stihnout v čase $\mathcal{O}(n)$. Provedeme to následovně: Nejprve prvky rozmístíme do vrcholů binárního stromu v libovolném pořadí – pokud máme strom uložený v poli, nemuseli jsme pro to udělat vůbec nic, prostě jenom začneme pozice v poli chápat jako indexy ve stromu.

Pak budeme opravovat uspořádání od nejnižší hladiny až k té nejvyšší, tedy v pořadí klesajících indexů. Kdykoliv zpracováváme nějaký vrchol, využijeme toho, že celý podstrom pod ním je už uspořádaný korektně, takže na opravu vztahů mezi novým vrcholem a jeho syny stačí provést bublání dolů. Pseudokód je mile jednoduchý:

Procedura MakeHeap (konstrukce haldy)
Vstup: Posloupnost prvků $x_1, \ldots, x_n$ s klíči $k_1, \ldots, k_n$
1. Prvky uložíme do pole tak, že $x(i) = x_i$ a $k(i) = k_i$.

2. Pro $i = \lfloor n/2 \rfloor, \ldots, 1$:
3. BubbleDown(i)

Výstup: Halda

Věta: Operace MakeHeap má časovou složitost $\mathcal{O}(n)$.

Důkaz: Nechť strom má h hladin očíslovaných od 0 (kořen) do $h-1$ (listy). Bez újmy na obecnosti budeme předpokládat, že všechny hladiny jsou úplně plné, takže $n = 2^h - 1$.

Zprvu se zdá, že provádíme n bublání, která trvají logaritmicky dlouho, takže jimi strávíme čas $\Theta(n \log n)$. Podíváme-li se pozorněji, všimneme si, že například na hladině $h-2$ leží přibližně $n/4$ prvků, ale každý z nich bubláme nejvýše o hladinu níže. Intuitivně většina vrcholů leží ve spodní části stromu, kde s nimi máme málo práce. Nyní to řekneme exaktněji.

Jedno BubbleDown na i-té hladině trvá $\mathcal{O}(h-1-i)$. Pokud to sečteme přes všech 2^i vrcholů hladiny a poté přes všechny hladiny, dostaneme (až na konstantu z $\mathcal{O}$):

$$\sum_{i=0}^{h-1} 2^i \cdot (h-1-i) = \sum_{j=0}^{h-1} 2^{h-1-j} \cdot j = \sum_{j=0}^{h-1} \frac{2^{h-1}}{2^j} \cdot j \leq n \cdot \sum_{j=0}^{h-1} \frac{j}{2^j} \leq n \cdot \sum_{j=0}^{\infty} \frac{j}{2^j}.$$

Podle podílového kritéria konvergence řad poslední suma konverguje, takže předposlední suma je shora omezena konstantou. □

Poznámka: Argument s konvergencí řady zaručuje existenci konstanty, ale její hodnota by mohla být absurdně vysoká. Pochybnosti zaplašíme sečtením řady. Jde to provést hezkým trikem – místo nekonečné řady budeme sčítat nekonečnou matici:

$$\begin{pmatrix} 1/2 & & & \\ 1/4 & 1/4 & & \\ 1/8 & 1/8 & 1/8 & \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

Sčítáme-li její prvky po řádcích, vyjde hledaná suma $\sum_j j/2^j$. Nyní budeme sčítat po sloupcích: první sloupec tvoří geometrickou řadu s kvocientem $1/2$, a tedy součtem 1 (to je mimochodem hezky vidět z binárního zápisu: $0.1 + 0.01 + 0.001 + \ldots = 0.111\ldots = 1$). Druhý sloupec má poloviční součet, třetí čtvrtinový, atd. Součet součtů sloupců je tudíž $1 + 1/2 + 1/4 + \ldots = 2$.

Třídění haldou – Heapsort

Již jsme přišli na to, že pomocí haldy lze třídit. Potřebovali jsme na to ovšem lineární pomocnou paměť na uložení haldy. Nyní ukážeme elegantnější a úspornější algoritmus, kterému se obvykle říká Heapsort.

Vstup dostaneme v poli. Z tohoto pole vytvoříme operací MakeHeap *maximovou* haldu. Pak budeme opakovaně mazat maximum. Halda se bude postupně zmenšovat a uvolněné místo v poli budeme zaplňovat setříděnými prvky.

Obecně po k-tém kroku bude na indexech $1, \ldots, n-k$ uložena halda a na $n-k+1, \ldots, n$ bude ležet posledních k prvků setříděné posloupnosti. V dalším kroku se tedy maximum haldy přesune na pozici $n-k$ a hranice mezi haldou a setříděnou posloupností se posune o 1 doleva.

Algoritmus HeapSort (třídění haldou)
Vstup: Pole $x_1, \ldots, x_n$

1. Pro $i = \lfloor n/2 \rfloor, \ldots, 1$: ◁ *vytvoříme z pole maximovou haldu*
2. HsBubbleDown(n, i)
3. Pro $i = n, \ldots, 2$:
4. Prohodíme x_1 s x_i. ◁ *maximum se dostane na správné místo*
5. HsBubbleDown$(i-1, 1)$ ◁ *opravíme haldu*

Výstup: Setříděné pole $x_1, \ldots, x_n$

Bublací procedura funguje podobně jako BubbleDown, jen používá opačné uspořádání a nerozlišuje prvky od jejich klíčů.

Procedura HsBubbleDown(m, i)
Vstup: Aktuální velikost haldy m, index vrcholu i

1. Dokud $2i \leq m$:
2. $s \leftarrow 2i$
3. Pokud $s+1 \leq m$ a $x_{s+1} > x_s$:
4. $s \leftarrow s+1$
5. Pokud $x_i > x_s$, vyskočíme z cyklu.
6. Prohodíme x_i a x_s.
7. $i \leftarrow s$

Věta: Algoritmus HeapSort setřídí n prvků v čase $\mathcal{O}(n \log n)$.

Důkaz: Celkem provedeme $\mathcal{O}(n)$ volání procedury HsBubbleDown. V každém okamžiku je v haldě nejvýše n prvků, takže jedno bublání trvá $\mathcal{O}(\log n)$. □

Z toho, že umíme pomocí haldy třídit, také plyne, že časová složitost haldových operací je nejlepší možná:

Věta: Mějme datovou strukturu s operacemi Insert a ExtractMin, která prvky pouze porovnává a přiřazuje. Pak má na n-prvkové množině alespoň jedna z těchto operací složitost $\Omega(\log n)$.

Důkaz: Pomocí n volání Insert a n volání ExtractMin lze setřídit n-prvkovou posloupnost. Z oddílu 3.3 ale víme, že každý třídicí algoritmus v porovnávacím modelu má složitost $\Omega(n \log n)$. □

Cvičení

1. Prioritní fronta se někdy definuje tak, že prvky se stejnou prioritou vrací v pořadí, v jakém byly do fronty vloženy. Ukažte, jak takovou frontu realizovat pomocí haldy. Dosáhněte časové složitosti $\mathcal{O}(\log n)$ na operaci.

2. Navrhněte operaci Delete, která z haldy smaže prvek zadaný jeho indexem.

3.* Dokažte, že vyhledávání prvku v haldě podle klíče vyžaduje čas $\Theta(n)$.

4. Definujme *d-regulární haldu* jako d-ární strom, který splňuje stejné axiomy o tvaru a uspořádání jako binární halda (binární tedy znamená totéž co 2-regulární). Ukažte, že d-ární strom má hloubku $\mathcal{O}(\log_d n)$ a lze ho také kódovat do pole. Dokažte, že haldové operace bublající nahoru trvají $\mathcal{O}(\log_d n)$ a ty bublající dolů $\mathcal{O}(d \log_d n)$. Zvýšením d tedy můžeme zrychlit Insert a Decrease za cenu zpomalení ExtractMin a Increase. To se bude hodit v Dijkstrově algoritmu, viz cvičení 6.2.3.

5. V rozboru operace MakeHeap jsme přehazovali pořadí sčítání v nekonečném součtu. To obecně nemusí být ekvivalentní úprava. Využijte poznatků z matematické analýzy, abyste dokázali, že v tomto případě se není čeho bát.

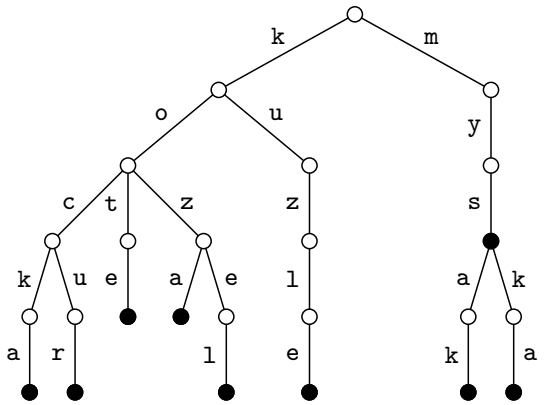
6.* Vymyslete algoritmus, který v haldě nalezne k-tý nejmenší prvek v čase $\mathcal{O}(k \log k)$.

4.3 Písmenkové stromy

Další jednoduchou datovou strukturou je *písmenkový strom* neboli *trie*.[1] Slouží k uložení *slovníku* nejen podle naší definice z oddílu 4.1, ale i v běžném smyslu tohoto slova. Pamatuje si množinu *slov* – řetězců složených ze znaků nějaké pevné konečné abecedy – a každému slovu může přiřadit nějakou hodnotu (třeba překlad slova do kočkovštiny).

[1] Zvláštní název, že? Vznikl zkřížením slov *tree* (strom) a *retrieval* (vyhledávání). Navzdory angličtině se u nás vyslovuje „trije“ a skloňuje podle vzoru růže.

Trie má tvar zakořeněného stromu. Z každého vrcholu vedou hrany označené navzájem různými znaky abecedy. V kořeni odpovídají prvnímu písmenu slova, o patro níž druhému, a tak dále.



Obrázek 4.4: Písmenkový strom pro slova `kocka`, `kocur`, `kote`, `koza`, `kozel`, `kuzle`, `mys`, `mysak`, `myska`

Vrcholům můžeme přiřadit řetězce tak, že přečteme všechny znaky na cestě z kořene do daného vrcholu. Kořen bude odpovídat prázdnému řetězci, čím hlouběji půjdeme, tím delší budou řetězce. Vrcholy odpovídající slovům slovníku označíme a uložíme do nich hodnoty přiřazené klíčům. Všimněte si, že označené mohou být i vnitřní vrcholy, je-li jeden klíč pokračováním jiného.

Každý vrchol si tedy bude pamatovat pole ukazatelů na syny (jako indexy slouží znaky abecedy), dále jednobitovou značku, zda se jedná o slovo ve slovníku, a případně hodnotu přiřazenou tomuto slovu. Je-li abeceda konstantně velká, celý vrchol zabere konstantní prostor. Pro větší abecedu můžeme pole nahradit některou z množinových datových struktur z příštích kapitol.

Vyhledávání (operace Member) bude probíhat takto: Začneme v kořeni a budeme následovat hrany podle jednotlivých písmen hledaného slova. Pokud budou všechny existovat, stačí se podívat, jestli vrchol, do kterého jsme došli, obsahuje značku. Chceme-li kdykoliv jít po neexistující hraně, ihned odpovíme, že se slovo se slovníku nenachází. Časová složitost hledání je lineární s délkou hledaného slova. Všimněte si, že na rozdíl od jiných datových struktur složitost nezávisí na tom, v jak velkém slovníku hledáme.

Při přidávání slova (operace Insert) se nové slovo pokusíme vyhledat. Kdykoliv při tom bude nějaká hrana chybět, vytvoříme ji a necháme ji ukazovat na nový list. Vrchol, do

kterého nakonec dojdeme, opatříme značkou. Časová složitost je zřejmě lineární s délkou přidávaného slova.

Při mazání (operace DELETE) bychom mohli slovo vyhledat a pouze z jeho koncového vrcholu odstranit značku. Tím by se nám ale mohly začít hromadit větve, které už nevedou do žádných označených vrcholů, a tedy jsou zbytečné. Proto mazání naprogramujeme rekurzivně: nejprve budeme procházet stromem dolů a hledat slovo, pak smažeme značku a budeme se vracet do kořene. Kdykoliv přitom narazíme na vrchol, který nemá ani značku, ani syny, smažeme ho. I zde vše stihneme v lineárním čase s délkou slova.

Vytvořili jsme tedy datovou strukturu pro reprezentaci slovníku řetězců, která zvládne operace MEMBER, INSERT a DELETE v lineárním čase s počtem znaků operandu. Jelikož stále platí, že všechny vrcholy stromu odpovídají prefixům (začátkům) slov ve slovníku, spotřebujeme prostor nejvýše lineární se součtem délek slovníkových slov.

Cvičení

1. Zkuste v písmenkovém stromu na obrázku vyhledat slova `kocka`, `kot` a `myval`. Pak přidejte `kot` a `kure` a nakonec smažte `myska`, `mysak` a `mys`.

2. Vymyslete, jak pomocí písmenkového stromu setřídit posloupnost řetězců v čase lineárním vzhledem k součtu jejich délek. Porovnejte s algoritmem přihrádkového třídění z oddílu 3.4.

3. Je dán text rozdělený na slova. Chceme vypsat frekvenční slovník, tedy tabulku všech slov setříděných podle počtu výskytů.

4. Vymyslete, jak pomocí písmenkového stromu reprezentovat množinu celých čísel z rozsahu 1 až ℓ. Jak bude složitost operací záviset na ℓ a na velikosti množiny?

5. Navrhněte datovou strukturu pro básníky, která si bude pamatovat slovník a bude umět hledat rýmy. Tedy pro libovolné zadané slovo najde jiné slovo ve slovníku, které má se zadaným co nejdelší společný suffix.

6.* Upravte datovou strukturu z předchozího cvičení, aby v případě, že nejlepších rýmů je více, vypsala lexikograficky nejmenší z nich.

7. Jak reprezentovat slovník, abyste uměli rychle vyhledávat všechny přesmyčky zadaného slova?

8. *Komprese trie:* Písmenkové stromy často obsahují dlouhé nevětvící se cesty. Tyto cesty můžeme komprimovat: nahradit jedinou hranou, která bude namísto jednoho písmene popsána celým řetězcem. Nadále bude platit, že všechny hrany vycházející z jednoho vrcholu se liší v prvních písmenech. Dokažte, že komprimovaná trie

pro množinu n slov má nejvýše $\mathcal{O}(n)$ vrcholů. Upravte operace MEMBER, INSERT a DELETE, aby fungovaly v komprimované trii.

4.4 Prefixové součty

Nyní se budeme zabývat datovými strukturami pro *intervalové operace.* Obecně se tím myslí struktury, které si pamatují nějakou posloupnost prvků $x_1, \ldots, x_n$ a dovedou efektivně zacházet se souvislými podposloupnostmi typu $x_i, x_{i+1}, \ldots, x_j$. Těm se obvykle říká *úseky* nebo také *intervaly* (anglicky *range*).

Začneme elementárním příkladem: Dostaneme posloupnost a chceme umět odpovídat na dotazy typu „Jaký je součet daného úseku?“. K tomu se hodí spočítat takzvané prefixové součty:

Definice: *Prefixové součty* pro posloupnost $x_1, \ldots, x_n$ tvoří posloupnost $p_1, \ldots, p_n$, kde $p_i = x_1 + \ldots + x_i$. Obvykle se hodí položit $p_0 = 0$ jako součet prázdného prefixu.

Všechny prefixové součty si dovedeme pořídit v čase $\Theta(n)$, jelikož $p_0 = 0$ a $p_{i+1} = p_i + x_{i+1}$. Jakmile je máme, hravě spočítáme součet obecného úseku $x_i + \ldots + x_j$: můžeme ho totiž vyjádřit jako rozdíl dvou prefixových součtů $p_j - p_{i-1}$.

Naše datová struktura tedy spotřebuje čas $\Theta(n)$ na inicializaci a pak dokáže v čase $\Theta(1)$ odpovídat na dotazy. Prvky posloupnosti nicméně neumí měnit – snadno si rozmyslíme, že změna prvku x_1 způsobí změnu všech prefixových součtů. Takovým strukturám se říká *statické,* na rozdíl od *dynamických*, jako je třeba halda.

Rozklad na bloky

Existuje řada technik, jimiž lze ze statické datové struktury vyrobit dynamickou. Jednu snadnou metodu si nyní ukážeme. V zájmu zjednodušení notace posuneme indexy tak, aby posloupnost začínala prvkem x_0.

Vstup rozdělíme na bloky velikosti b (konkrétní hodnotu b zvolíme později). První blok bude tvořen prvky $x_0, \ldots, x_{b-1}$, druhý prvky $x_b, \ldots, x_{2b-1}$, atd. Celkem tedy vznikne n/b bloků. Pakliže n není dělitelné b, doplníme posloupnost nulami na celý počet bloků.

Obrázek 4.5: Rozklad na bloky pro n = 30, b = 6 a dva dotazy

Libovolný zadaný úsek se buďto celý vejde do jednoho bloku, nebo ho můžeme rozdělit na konec (suffix) jednoho bloku, nějaký počet celých bloků a začátek (prefix) dalšího bloku. Libovolná z těchto částí přitom může být prázdná.

Pořídíme si tedy dva druhy struktur:

- *Lokální* struktury $L_1, \ldots, L_{n/b}$ budou vyřizovat dotazy uvnitř bloku. K tomu nám stačí spočítat pro každý blok prefixové součty.
- *Globální* struktura G bude naopak pracovat s celými bloky. Budou to prefixové součty pro posloupnost, která vznikne nahrazením každého bloku jediným číslem – jeho součtem.

Inicializaci těchto struktur zvládneme v lineárním čase: Každou z n/b lokálních struktur vytvoříme v čase $\Theta(b)$. Pak spočteme $\Theta(n/b)$ součtů bloků, každý v čase $\Theta(b)$ – nebo se na ně můžeme zeptat lokálních struktur. Nakonec vyrobíme globální strukturu v čase $\Theta(n/b)$. Všechno dohromady trvá $\Theta(n/b \cdot b) = \Theta(n)$.

Každý dotaz na součet úseku nyní můžeme přeložit na nejvýše dva dotazy na lokální struktury a nejvýše jeden dotaz na globální strukturu. Všechny struktury přitom vydají odpověď v konstantním čase.

Procedura SoučetÚseku(i, j)
Vstup: Začátek úseku i, konec úseku j

1. Pokud $j < i$, úsek je prázdný, takže položíme $s \leftarrow 0$ a skončíme.
2. $z \leftarrow \lfloor i/b \rfloor$, $k \leftarrow \lfloor j/b \rfloor$ ◁ *ve kterém bloku úsek začíná a kde končí*
3. Pokud $z = k$: ◁ *celý úsek leží v jednom bloku*
4. $\quad s \leftarrow$ LokálníDotaz$(L_z, i \bmod b, j \bmod b)$
5. Jinak:
6. $\quad s_1 \leftarrow$ LokálníDotaz$(L_z, i \bmod b, b - 1)$
7. $\quad s_2 \leftarrow$ GlobálníDotaz$(G, z + 1, k - 1)$
8. $\quad s_3 \leftarrow$ LokálníDotaz$(L_k, 0, j \bmod b)$
9. $\quad s \leftarrow s_1 + s_2 + s_3$

Výstup: Součet úseku s

Nyní se podívejme, co způsobí změna jednoho prvku posloupnosti. Především musíme aktualizovat příslušnou lokální strukturu, což trvá $\Theta(b)$. Pak změnit jeden součet bloku a přepočítat globální strukturu. To zabere čas $\Theta(n/b)$.

Celkem nás tedy změna prvku stojí $\Theta(b + n/b)$. Využijeme toho, že parametr b jsme si mohli zvolit jakkoliv, takže ho nastavíme tak, abychom výraz $b + n/b$ minimalizovali. S rostoucím b první člen roste a druhý klesá. Jelikož součet se asymptoticky chová stejně

jako maximum, výraz bude nejmenší, pokud se b a n/b vyrovnají. Zvolíme tedy $b = \lfloor\sqrt{n}\rfloor$ a dostaneme časovou složitost $\Theta(\sqrt{n})$.

Věta: Bloková struktura pro součty úseků se inicializuje v čase $\Theta(n)$, na dotazy odpovídá v čase $\Theta(1)$ a po změně jednoho prvku ji lze aktualizovat v čase $\Theta(\sqrt{n})$.

Odmocninový čas na změnu není optimální, ale princip rozkladu na bloky je užitečné znát a v příštím oddílu nás dovede k mnohem rychlejší struktuře.

Intervalová minima

Pokud se místo součtů budeme ptát na minima úseků, překvapivě dostaneme velmi odlišný problém. Pokusíme-li se použít osvědčený trik a předpočítat prefixová minima, tvrdě narazíme: minimum obecného úseku nelze získat z prefixových minim – například v posloupnosti $\{1, 9, 4, 7, 2\}$ jsou všechna prefixová minima rovna 1.

Opět nám pomůže rozklad na bloky. Lokální struktury si nebudou nic předpočítávat a dotazy budou vyřizovat otrockým projitím celého bloku v čase $\Theta(b)$. Globální struktura si bude pamatovat pouze n/b minim jednotlivých bloků, dotazy bude vyřizovat též otrocky v čase $\Theta(n/b)$.

Inicializaci struktury evidentně zvládneme v lineárním čase. Libovolný dotaz rozdělíme na konstantně mnoho dotazů na lokální a globální struktury, což dohromady potrvá $\Theta(b+n/b)$. Po změně prvku stačí přepočítat minimum jednoho bloku v čase $\Theta(b)$. Použijeme-li osvědčenou volbu $b = \sqrt{n}$, dosáhneme složitosti $\Theta(\sqrt{n})$ pro dotazy i modifikace.

Pro zvědavého čtenáře dodáváme, že existuje i statická struktura s lineárním časem na předvýpočet a konstantním na minimový dotaz. Je ovšem o něco obtížnější, takže zájemce odkazujeme na kapitolu o dekompozici stromů v knize Krajinou grafových algoritmů [8]. Jednu z technik, které se k tomu hodí, si můžete vyzkoušet v cvičení 12.

Cvičení

1. Vyřešte úlohu o úseku s maximálním součtem z úvodní kapitoly pomocí prefixových součtů.

2. Je dána posloupnost přirozených čísel a číslo s. Chceme zjistit, zda existuje úsek posloupnosti, jehož součet je přesně s. Jak se úloha změní, pokud dovolíme i záporná čísla?

3. Vymyslete algoritmus, který v posloupnosti celých čísel najde úsek se součtem co nejbližším danému číslu.

4. V posloupnosti celých čísel najděte nejdelší *vyvážený* úsek, tedy takový, v němž je stejně kladných čísel jako záporných.

5.* Mějme posloupnost červených, zelených a modrých prvků. Opět hledáme nejdelší vyvážený úsek, tedy takový, v němž jsou všechny barvy zastoupeny stejným počtem prvků. Co se změní, je-li barev více?

6. Navrhněte dvojrozměrnou analogii prefixových součtů: Pro matici $m \times n$ předpočítejte v čase $\mathcal{O}(mn)$ údaje, pomocí nichž půjde v konstantním čase vypočíst součet hodnot v libovolné souvislé obdélníkové podmatici.

7.* Jak by vypadaly prefixové součty pro d-rozměrnou matici?

8.* Pro ctitele algebry: Myšlenka skládání úseků z prefixů fungovala pro součty, ale selhala pro minima. Uvažujme tedy obecněji nějakou binární operaci $\oplus$, jíž chceme vyhodnocovat pro úseky: $x_i \oplus x_{i+1} \oplus \ldots \oplus x_j$. Co musí operace $\oplus$ splňovat, aby bylo možné použít prefixové součty? Jaké vlastnosti jsou potřeba pro blokovou strukturu?

9. K odmocninové časové složitosti aktualizací nám pomohlo zavedení dvojúrovňové struktury. Ukažte, jak pomocí tří úrovní dosáhnout času $\mathcal{O}(n^{1/3})$ na aktualizaci a $\mathcal{O}(1)$ na dotaz.

10.* Vyřešte předchozí cvičení pro obecný počet úrovní. Jaký počet je optimální?

11. Na kraji města stojí n-patrový panelák, jehož obyvatelé se baví házením vajíček na chodník před domem. Ideální vajíčko se při hodu z p-tého nebo vyššího patra rozbije; pokud ho hodíme z nižšího, zůstane v původním stavu. Jak na co nejméně pokusů zjistit, kolik je p, pokud máme jenom 2 vajíčka? Jak to dopadne pro neomezený počet vajíček? A jak pro 3?

12. Jak náročný předvýpočet je potřeba, abychom uměli minima úseků počítat v konstantním čase? V čase $\mathcal{O}(n^2)$ je to triviální, ukažte, že stačí $\mathcal{O}(n \log n)$. Hodí se přepočítat minima úseků tvaru $x_i, \ldots, x_{i+2^j-1}$ pro všechna i a j.

13.* V matici tvaru $R \times S$ najděte podmatici tvaru $r \times s$, jejíž medián je největší možný.

4.5 Intervalové stromy

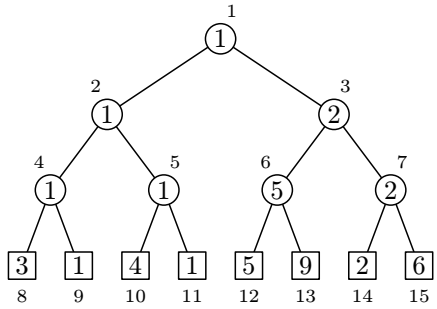
Rozklad posloupnosti na bloky, který jsme zavedli v minulém oddílu, lze elegantně zobecnit. Posloupnost budeme dělit na poloviny, ty zase na poloviny, a tak dále, až dojdeme k jednotlivým prvkům. Pro každou část tohoto rozkladu si přitom budeme udržovat něco předpočítaného.

Tato úvaha vede k takzvaným intervalovým stromům, které dovedou v logaritmickém čase jak vyhodnocovat intervalové dotazy, tak měnit prvky. Nadefinujeme je pro výpočet minim, ale pochopitelně by mohly počítat i součty či jiné operace.

Značení: V celém oddílu pracujeme s posloupností $x_0, \ldots, x_{n-1}$ celých čísel. Bez újmy na obecnosti budeme předpokládat, že n je mocnina dvojky. Interval $\langle i, j \rangle$ obsahuje prvky $x_i, \ldots, x_{j-1}$ (pozor, x_j už v intervalu neleží!). Pro $i \geq j$ to je prázdný interval.

Definice: *Intervalový strom* pro posloupnost $x_0, \ldots, x_{n-1}$ je binární strom s následujícími vlastnostmi:

1. Všechny listy leží na stejné hladině a obsahují zleva doprava prvky $x_0, \ldots, x_{n-1}$.
2. Každý vnitřní vrchol má dva syny a pamatuje si minimum ze všech listů ležících pod ním.



Obrázek 4.6: Intervalový strom a jeho očíslování

Pozorování: Stejně jako haldu, i intervalový strom můžeme uložit do pole po hladinách. Na indexech 1 až $n-1$ budou ležet vnitřní vrcholy, na indexech n až $2n-1$ listy s prvky x_0 až x_{n-1}. Strom budeme reprezentovat polem S, jehož prvky budou buď členy posloupnosti nebo minima podstromů.

Ještě si všimneme, že podstromy přirozeně odpovídají intervalům v posloupnosti. Pod kořenem leží celá posloupnost, pod syny kořene poloviny posloupnosti, pod jejich syny čtvrtiny, atd. Obecně očíslujeme-li hladiny od 0 (kořen) po $h = \log n$ (listy), bude na k-té hladině ležet 2^k vrcholů. Ty odpovídají *kanonickým intervalům* tvaru $\langle i, i + 2^{h-k} \rangle$ pro i dělitelné 2^{h-k}.

Statický intervalový strom můžeme vytvořit v lineárním čase: zadanou posloupnost zkopírujeme do listů a pak zdola nahoru přepočítáváme minima ve vnitřních vrcholech: každý vnitřní vrchol obdrží minimum z hodnot svých synů. Strávíme tím konstantní čas na vrchol, celkem tedy $\mathcal{O}(n + n/2 + n/4 + \ldots + 1) = \mathcal{O}(n)$.

Intervalový dotaz a jeho rozklad

Nyní se zabývejme vyhodnocováním dotazu na minimum intervalu. Zadaný interval rozdělíme na $\mathcal{O}(\log n)$ disjunktních kanonických intervalů. Jejich minima si strom pamatuje, takže stačí vydat jako výsledek minimum z těchto minim.

Příslušné kanonické intervaly můžeme najít třeba rekurzivním prohledáním stromu. Začneme v kořeni. Kdykoliv stojíme v nějakém vrcholu, podíváme se, v jakém vztahu je hledaný interval $\langle i, j\rangle$ s kanonickým intervalem $\langle a, b\rangle$ přiřazeným aktuálnímu vrcholu. Mohou nastat čtyři možnosti:

- $\langle i, j\rangle$ a $\langle a, b\rangle$ se shodují: $\langle i, j\rangle$ je kanonický, takže jsme hotovi.
- $\langle i, j\rangle$ leží celý v levé polovině $\langle a, b\rangle$: tehdy se rekurzivně zavoláme na levý podstrom a hledáme v něm stejný interval $\langle i, j\rangle$.
- $\langle i, j\rangle$ leží celý v pravé polovině $\langle a, b\rangle$: obdobně, ale jdeme doprava.
- $\langle i, j\rangle$ prochází přes střed s intervalu $\langle a, b\rangle$: dotaz $\langle i, j\rangle$ rozdělíme na $\langle i, s\rangle$ a $\langle s, j\rangle$. První z nich vyhodnotíme rekurzivně v levém podstromu, druhý v pravém.

Nyní tuto myšlenku zapíšeme v pseudokódu. Na vrcholy se budeme odkazovat pomocí jejich indexů v poli S. Chceme-li rozložit na kanonické intervaly daný interval $\langle i, j\rangle$, zavoláme IntCanon$(1, \langle 0, n\rangle, \langle i, j\rangle)$.

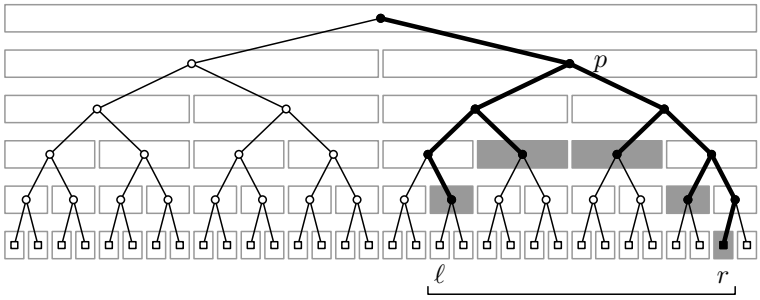
Procedura IntCanon$(v, \langle a, b\rangle, \langle i, j\rangle)$ (rozklad na kanonické intervaly)
Vstup: Index vrcholu v, který odpovídá intervalu $\langle a, b\rangle$; dotaz $\langle i, j\rangle$

1. Pokud $a = i$ a $b = j$, nahlásíme vrchol v a skončíme. ◁ *přesná shoda*
2. $s \leftarrow (a + b)/2$ ◁ *střed intervalu* $\langle a, b\rangle$
3. Pokud $j \leq s$, zavoláme IntCanon$(2v, \langle a, s\rangle, \langle i, j\rangle)$. ◁ *vlevo*
4. Jinak je-li $i \geq s$, zavoláme IntCanon$(2v + 1, \langle s, b\rangle, \langle i, j\rangle)$. ◁ *vpravo*
5. Ve všech ostatních případech: ◁ *dotaz přes střed*
6. Zavoláme IntCanon$(2v, \langle a, s\rangle, \langle i, s\rangle)$.
7. Zavoláme IntCanon$(2v + 1, \langle s, b\rangle, \langle s, j\rangle)$.

Výstup: Rozklad intervalu $\langle i, j\rangle$ na kanonické intervaly

Lemma: Procedura IntCanon rozloží dotaz na nejvýše $2\log_2 n$ disjunktních kanonických intervalů a stráví tím čas $\Theta(\log n)$.

Důkaz: Situaci sledujme na obrázku 4.7. Nechť dostaneme dotaz $\langle i, j\rangle$. Označme ℓ a r první a poslední list ležící v tomto intervalu (tyto listy odpovídají prvkům x_i a x_{j-1}). Nechť p je nejhlubší společný předek listů ℓ a r.



Obrázek 4.7: Rozklad dotazu na kanonické intervaly

Procedura prochází od kořene po cestě do p, protože do té doby platí, že dotaz leží buďto celý nalevo, nebo celý napravo. Ve vrcholu p se dotaz rozdělí na dva podintervaly.

Levý podinterval zpracováváme cestou z p do ℓ. Kdykoliv cesta odbočuje doleva, pravý syn leží celý uvnitř dotazu. Kdykoliv odbočuje doprava, levý syn leží celý venku. Takto dojdeme buďto až do ℓ, nebo dříve zjistíme, že podinterval je kanonický. Na každé z $\log n$ hladin různých od kořene přitom strávíme konstantní čas a vybereme nejvýše jeden kanonický interval.

Pravý podinterval zpracováváme analogicky cestou z p do r. Sečtením přes všechny hladiny získáme kýžené tvrzení. □

Rozklad zdola nahoru

Ukážeme ještě jeden způsob, jak dotaz rozkládat na kanonické intervaly. Tentokrát bude založen na procházení hladin stromu od nejnižší k nejvyšší. Dotaz přitom budeme postupně zmenšovat „ukusováním“ kanonických intervalů z jednoho či druhého okraje. V každém okamžiku výpočtu si budeme pamatovat souvislý úsek vrcholů $\langle a, b\rangle$ na aktuální hladině, které dohromady pokrývají aktuální dotaz.

Na počátku dostaneme dotaz $\langle i, j\rangle$ a přeložíme ho na úsek listů $\langle n + i, n + j\rangle$. Kdykoliv pak na nějaké hladině zpracováváme úsek $\langle a, b\rangle$, nejprve se podíváme, zda jsou a i b sudá. Pokud ano, úsek $\langle a, b\rangle$ pokrývá stejný interval, jako úsek $\langle a/2, b/2\rangle$ o hladinu výš, takže se můžeme na vyšší hladinu rovnou přesunout. Je-li a liché, ukousneme kanonický interval vrcholu a a zbude nám úsek $\langle a + 1, b\rangle$. Podobně je-li b liché, ukousneme interval vrcholu $b - 1$ a snížíme b o 1. Takto umíme všechny případy převést na sudé a i b, a tím pádem na úsek o hladinu výš.

Zastavíme se v okamžiku, kdy dostaneme prázdný úsek, což je nejpozději tehdy, když se pokusíme vystoupit z kořene nahoru.

Procedura IntCanon2(i, j) (rozklad na kanonické intervaly zdola nahoru)
Vstup: Dotaz $\langle i, j \rangle$

1. $a \leftarrow i + n$, $b \leftarrow j + n$ ◁ *indexy listů*
2. Dokud $a < b$:
3. Je-li a liché, nahlásíme vrchol a a položíme $a \leftarrow a + 1$.
4. Je-li b liché, položíme $b \leftarrow b - 1$ a nahlásíme vrchol b.
5. $a \leftarrow a/2$, $b \leftarrow b/2$

Výstup: Rozklad intervalu $\langle i, j \rangle$ na kanonické intervaly

Během výpočtu projdeme $\log_2 n + 1$ hladin, na každé nahlásíme nejvýše 2 vrcholy. Pokud si navíc uvědomíme, že nahlášení kořene vylučuje nahlášení kteréhokoliv jiného vrcholu, vyjde nám opět nejvýše $2 \log_2 n$ kanonických intervalů.

Aktualizace prvku

Od statického intervalového stromu je jen krůček k dynamickému. Co se stane, změníme-li nějaký prvek posloupnosti? Upravíme hodnotu v příslušném listu stromu a pak musíme přepočítat všechny kanonické intervaly, v nichž daný prvek leží. Ty odpovídají vrcholům ležícím na cestě z upraveného listu do kořene stromu.

Stačí tedy změnit list, vystoupat z něj do kořene a cestou všem vnitřním vrcholům přepočítat hodnotu jako minimum ze synů. To stihneme v čase $\Theta(\log n)$. Program je přímočarý:

Procedura IntUpdate(i, x) (aktualizace prvku v intervalovém stromu)
Vstup: Pozice i v posloupnosti, nová hodnota x

1. $a \leftarrow n + i$ ◁ *index listu*
2. $S[a] \leftarrow x$
3. Dokud $a > 1$:
4. $a \leftarrow \lfloor a/2 \rfloor$
5. $S[a] \leftarrow \min(S[2a], S[2a + 1])$

Aktualizace intervalu a líné vyhodnocování*

Nejen dotazy, ale i aktualizace mohou pracovat s intervalem. Naučíme náš strom pro výpočet minim operaci IncRange(i, j, δ), která ke všem prvkům v intervalu $\langle i, j \rangle$ přičte δ. Nemůžeme to samozřejmě udělat přímo – to by trvalo příliš dlouho. Použijeme proto trik, kterému se říká *líné vyhodnocování operací.*

Zadaný interval $\langle i, j \rangle$ nejprve rozložíme na kanonické intervaly. Pro každý z nich pak prostě zapíšeme do příslušného vrcholu stromu instrukci „někdy později zvyš všechny hodnoty v tomto podstromu o δ“.

Až později nějaká další operace na instrukci narazí, pokusí se ji vykonat. Udělá to ovšem líně: Místo aby pracně zvýšila všechny hodnoty v podstromu, jenom předá svou instrukci oběma svým synům, aby ji časem vykonali. Pokud budeme strom procházet vždy shora dolů, bude platit, že v části stromu, do níž jsme se dostali, jsou už všechny instrukce provedené. Zkratka a dobře, šikovný šéf všechnu práci předává svým podřízeným.

Budeme si proto pro každý vrchol v pamatovat nejen minimum $S[v]$, ale také nějaké číslo $\Delta[v]$, o které mají být zvětšené všechny hodnoty v podstromu: jak prvky v listech, tak všechna minima ve vnitřních vrcholech.

Proceduru pro operaci IncRange založíme na osvědčeném průchodu shora dolů v proceduře IntCanon. Místo hlášení kanonických intervalů do nich budeme rozmisťovat instrukce. Navíc potřebujeme aktualizovat minima ve vrcholech ležících mezi kořenem a těmito kanonickými intervaly. To snadno zařídíme při návratech z rekurze. Pro zvýšení intervalu $\langle i,j\rangle$ o δ budeme volat IntIncRange$(1,\langle 0,n\rangle,\langle i,j\rangle,\delta)$.

Procedura IntIncRange$(v,\langle a,b\rangle,\langle i,j\rangle,\delta)$ (aktualizace intervalu)
Vstup: Stojíme ve vrcholu v pro interval $\langle a,b\rangle$ a přičítáme δ k $\langle i,j\rangle$

1. Pokud $a=i$ a $b=j$: ◁ *už máme kanonický interval*
2. Položíme $\Delta[v]\leftarrow\Delta[v]+\delta$ a skončíme.
3. $s\leftarrow(a+b)/2$ ◁ *střed intervalu* $\langle a,b\rangle$
4. Pokud $j\le s$, zavoláme IntIncRange$(2v,\langle a,s\rangle,\langle i,j\rangle,\delta)$.
5. Jinak je-li $i\ge s$, zavoláme IntIncRange$(2v+1,\langle s,b\rangle,\langle i,j\rangle,\delta)$.
6. Ve všech ostatních případech: ◁ *dotaz přes střed*
7. Zavoláme IntIncRange$(2v,\langle a,s\rangle,\langle i,s\rangle,\delta)$.
8. Zavoláme IntIncRange$(2v+1,\langle s,b\rangle,\langle s,j\rangle,\delta)$.
9. $S[v]\leftarrow\min(S[2v]+\Delta[2v],S[2v+1]+\Delta[2v+1])$

Procedura běží v čase $\Theta(\log n)$, protože projde přesně tutéž část stromu jako procedura IntCanon a v každém vrcholu stráví konstantní čas.

Všechny ostatní operace odvodíme z procházení shora dolů a upravíme je tak, aby v každém navštíveném vrcholu volaly následující proceduru. Ta se postará o líné vyhodnocování instrukcí a zabezpečí, aby v navštívené části stromu žádné instrukce nezbývaly.

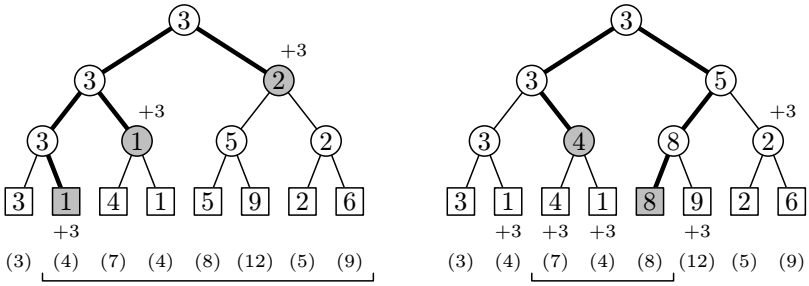
Procedura IntLazyEval(v) (líné vyhodnocování)
Vstup: Index vrcholu v

1. $\delta\leftarrow\Delta[v]$, $\Delta[v]\leftarrow 0$
2. $S[v]\leftarrow S[v]+\delta$
3. Pokud $v<n$: ◁ *předáváme synům*

4. $\Delta[2v] \leftarrow \Delta[2v] + \delta$
5. $\Delta[2v+1] \leftarrow \Delta[2v+1] + \delta$

Každou operaci jsme opět zpomalili konstanta-krát, takže složitost zůstává logaritmická.

Vše si můžete prohlédnout na obrázku 4.8. Začneme stromem z obrázku 4.6. Pak přičteme 3 k intervalu $\langle 1, 8\rangle$ a získáme levý strom (u vrcholů jsou napsané instrukce $\Delta[v]$, v závorkách pod listy skutečné hodnoty posloupnosti). Nakonec položíme dotaz $\langle 2, 5\rangle$, čímž se instrukce částečně vyhodnotí a vznikne pravý strom.



Obrázek 4.8: Líné vyhodnocování operací

Cvičení

1. Naučte intervalový strom zjistit *druhý nejmenší* prvek v zadaném intervalu.

2. Naučte intervalový strom zjistit nejbližší prvek, který leží napravo od zadaného listu a obsahuje větší hodnotu.

3. Upravte intervalový strom, aby hranice intervalů nebyla čísla $1, \ldots, n$, nýbrž prvky nějaké obecné posloupnosti $h_1 < \ldots < h_n$ zadané při inicializaci struktury.

4. Naprogramujte funkci IntCanon nerekurzivně. Nejprve se vydejte z kořene do společného předka p a pak paralelně procházejte levou i pravou cestu do krajů intervalu. Může se hodit, že bratr vrcholu v má index v xor 1.

5. Jeřáb se skládá z n ramen spojených klouby. Pro jednoduchost si ho představíme jako lomenou čáru v rovině. První úsečka je fixní, každá další je připojena kloubem na svou předchůdkyni. Koncový bod poslední úsečky hraje roli háku. Navrhněte datovou strukturu, která si bude pamatovat stav jeřábu a bude nabízet operace „otoč i-tým kloubem o úhel α“ a „zjisti aktuální pozici háku“.

6. Ukládáme-li intervalový strom do pole, potřebujeme předem vědět, jak velké pole si pořídit. Ukažte, jak se bez tohoto předpokladu obejít. Může se hodit technika nafukovacího pole z oddílu 9.1.

7.* Naučte intervalový strom operaci SetRange(i, j, x), která všechny prvky v intervalu $\langle i, j \rangle$ nastaví na x. Líným vyhodnocováním dosáhněte složitosti $\mathcal{O}(\log n)$.

8.* Vraťte se k cvičení 4.4.10 a všimněte si, že je-li n mocnina dvojky a zvolíte-li počet úrovní rovný $\log_2 n$, stane se z přihrádkové struktury intervalový strom.

9.* Navrhněte datovou strukturu, která si bude pamatovat posloupnost n levých a pravých závorek a bude umět v čase $\mathcal{O}(\log n)$ otočit jednu závorku a rozhodnout, zda je zrovna posloupnost správně uzávorkovaná.

5 Základní grafové algoritmy

5 Základní grafové algoritmy

Teorie grafů nám dává elegantní nástroj k popisu situací ze skutečného i matematického světa. Díky tomu dovedeme různé praktické problémy překládat na otázky týkající se grafů. To je zajímavé i samo o sobě, ale jak uvidíme v této kapitole, často díky tomu můžeme snadno přijít k rychlému algoritmu.

V celé kapitole budeme pro grafy používat následující značení:

Definice:

- G je graf, se kterým pracujeme (orientovaný nebo neorientovaný).
- V je množina vrcholů tohoto grafu, E množina jeho hran.
- n značí počet vrcholů, m počet hran.
- uv označuje hranu z vrcholu u do vrcholu v. Pokud pracujeme s orientovaným grafem, je to formálně uspořádaná dvojice (u, v); v neorientovaném grafu je to dvouprvková množina $\{u, v\}$.
- *Následníci* vrcholu v budou vrcholy, do kterých z v vede hrana. Analogicky z *předchůdců* vede hrana do v. V neorientovaném grafu tyto pojmy splývají. Předchůdcům a následníkům dohromady říkáme *sousedé* vrcholu v.
- *Stupeň* $\deg(v)$ vrcholu v udává počet jeho sousedů, *vstupní* stupeň $\deg^{\text{in}}(v)$ resp. *výstupní* stupeň $\deg^{\text{out}}(v)$ udává počet předchůdců resp. následníků vrcholu v.
- Někdy budeme uvažovat též *multigrafy,* v nichž mohou být dva vrcholy spojeny více paralelními hranami, případně mohou existovat *smyčky* spojující vrchol s ním samým.

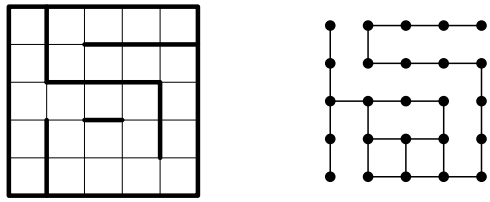
Čtenáře, kteří se dosud s teorií grafů nesetkali, odkazujeme na knihu Kapitoly z diskrétní matematiky [9].

5.1 Několik grafů úvodem

Pojďme se nejprve podívat, jak se dají praktické problémy modelovat pomocí grafů.

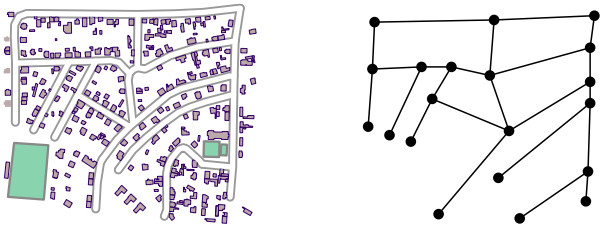
Bludiště na čtverečkovaném papíře: vrcholy jsou čtverečky, hranou jsou spojené sousední čtverečky, které nejsou oddělené zdí. Je přirozené ptát se na *komponenty souvislosti* bludiště, což jsou různé „místnosti“, mezi nimiž nelze přejít (alespoň bez prokopání nějaké zdi). Pokud jsou dva čtverečky v téže místnosti, chceme mezi nimi hledat *nejkratší cestu.*

Mapa města je podobná bludišti: vrcholy odpovídají křižovatkám, hrany ulicím mezi nimi. Hrany se hodí *ohodnotit* délkami ulic nebo časy potřebnými na průjezd; pak nás



Obrázek 5.1: Bludiště a jeho graf

opět zajímají nejkratší cesty. Když městečko zapadá sněhem, *minimální kostra* grafu nám řekne, které silnice chceme prohrnout jako první. *Mosty* a *artikulace* (hrany resp. vrcholy, po jejichž odebrání se graf rozpadne) mají také svůj přirozený význam. Pokud jsou ve městě jednosměrky, budeme uvažovat o orientovaném grafu.



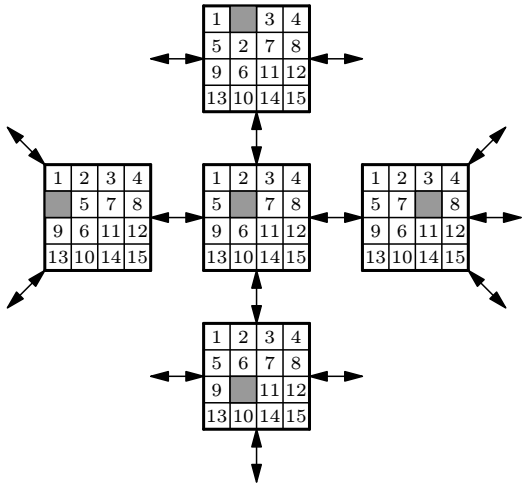
Obrázek 5.2: Mapa a její graf

Hlavolam „patnáctka“: v krabičce velikosti 4×4 je 15 očíslovaných jednotkových čtverečků a jedna jednotková díra. Jedním tahem smíme do díry přesunout libovolný čtvereček, který s ní sousedí; matematik by spíš řekl, že můžeme díru prohodit se sousedícím čtverečkem.

Opět se hodí graf: vrcholy jsou *konfigurace* (možná rozmístění čtverečků a díry v krabičce) a hrany popisují, mezi kterými konfiguracemi jde přejít jedním tahem. Tato konstrukce funguje i pro další hlavolamy a hry a obvykle se jí říká *stavový prostor* hry. Obecně vznikne orientovaný graf, ale zrovna u patnáctky ke každému možnému tahu existuje i tah opačný.

Šeherezádino číslo 1 001 je zajímavé například tím, že je nejmenším násobkem sedmi, jehož desítkový zápis sestává pouze z nul a jedniček. Co kdybychom obecně hledali nejmenší násobek nějakého čísla k tvořený jen nulami a jedničkami?

Zatím vyřešíme jednodušší otázku: spokojíme se s libovolným násobkem, ne tedy nutně nejmenším. Představme si, že takové číslo vytváříme postupným připisováním číslic.



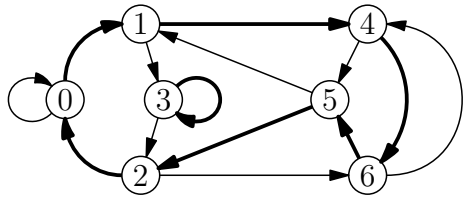
Obrázek 5.3: Část stavového grafu patnáctky

Začneme jedničkou. Kdykoliv pak máme nějaké číslo x, umíme z něj vytvořit čísla $x\mathbf{0} = 10x$ a $x\mathbf{1} = 10x + 1$. Všímejme si zbytků po dělení číslem k:

$$(x\mathbf{0}) \bmod k = (10x) \bmod k = (10 \cdot (x \bmod k)) \bmod k,$$
$$(x\mathbf{1}) \bmod k = (10x + 1) \bmod k = (10 \cdot (x \bmod k) + 1) \bmod k.$$

Ejhle: nový zbytek je jednoznačně určen předchozím zbytkem.

V řeči zbytků tedy začínáme s jedničkou a chceme ji pomocí uvedených dvou pravidel postupně přetransformovat na nulu. To je přeci grafový problém: vrcholy jsou zbytky 0 až $k - 1$, orientované hrany odpovídají našim pravidlům a hledáme cestu z vrcholu 1 do vrcholu 0.



Obrázek 5.4: Graf zbytků pro k = 7. Tenké čáry připisují 0, tučné 1.

Cvičení

1. Kolik nejvýše vrcholů a hran má graf, kterým jsme popsali bludiště z $n \times n$ čtverečků? A kolik nejméně?

2. Kolik vrcholů a hran má graf patnáctky? Vejde se do paměti vašeho počítače? Jak by to dopadlo pro menší verzi hlavolamu v krabičce 3×3?

3.* Kolik má graf patnáctky komponent souvislosti?

4. Kolik vrcholů a hran má graf Šeherezádina problému pro dané k?

5.* Dokažte, že Šeherezádin problém je pro každé $k > 0$ řešitelný.

6.* Jakému grafovému problému by odpovídalo hledání nejmenšího čísla z nul a jedniček dělitelného k? Pokud vás napadla nejkratší cesta, ještě chvíli přemýšlejte.

5.2 Prohledávání do šířky

Základním stavebním kamenem většiny grafových algoritmů je nějaký způsob *prohledávání grafu.* Tím myslíme postupné procházení grafu po hranách od určitého počátečního vrcholu. Možných způsobů prohledávání je víc, zatím ukážeme ten nejjednodušší: *prohledávání do šířky.* Často se mu říká zkratkou *BFS* z anglického *breadth-first search.*

Na vstupu dostaneme konečný orientovaný graf a počáteční vrchol v_0. Postupně nacházíme následníky vrcholu v_0, pak následníky těchto následníků, a tak dále, až objevíme všechny vrcholy, do nichž se dá z v_0 dojít po hranách. Obrazně řečeno, do grafu nalijeme vodu a sledujeme, jak postupuje vlna.

Během výpočtu rozlišujeme tři možné *stavy vrcholů:*

- *Nenalezené* – to jsou ty, které jsme na své cestě grafem dosud nepotkali.
- *Otevřené* – o těch už víme, ale ještě jsme neprozkoumali hrany, které z nich vedou.
- *Uzavřené* – už jsme prozkoumali i hrany, takže se takovými vrcholy nemusíme nadále zabývat.

Na počátku výpočtu tedy chceme prohlásit v_0 za otevřený a ostatní vrcholy za nenalezené. Pak v_0 uzavřeme a otevřeme všechny jeho následníky. Poté procházíme tyto následníky, uzavíráme je a otevíráme jejich dosud nenalezené následníky. A tak dále. Otevřené vrcholy si přitom pamatujeme ve *frontě,* takže pokaždé zavřeme ten z nich, který je otevřený nejdéle.

Následuje zápis algoritmu v pseudokódu. Pomocná pole D a P budou hrát svou roli později (v oddílu 5.5), zatím můžete všechny operace s nimi přeskočit – chod algoritmu evidentně neovlivňují.

Algoritmus BFS (prohledávání do šířky)
Vstup: Graf $G = (V, E)$ a počáteční vrchol $v_0 \in V$

1. Pro všechny vrcholy v:
2. $\quad$ $stav(v) \leftarrow$ *nenalezený*
3. $\quad$ $D(v) = \emptyset$, $P(v) \leftarrow \emptyset$
4. $stav(v_0) \leftarrow$ *otevřený*
5. $D(v_0) \leftarrow 0$
6. Založíme frontu Q a vložíme do ní vrchol v_0.
7. Dokud je fronta Q neprázdná:
8. $\quad$ Odebereme první vrchol z Q a označíme ho v.
9. $\quad$ Pro všechny následníky w vrcholu v:
10. $\quad\quad$ Je-li $stav(w) =$ *nenalezený*:
11. $\quad\quad\quad$ $stav(w) \leftarrow$ *otevřený*
12. $\quad\quad\quad$ $D(w) \leftarrow D(v) + 1$, $P(w) \leftarrow v$
13. $\quad\quad\quad$ Přidáme w do fronty Q.
14. $\quad$ $stav(v) \leftarrow$ *uzavřený*

Nyní dokážeme, že BFS skutečně dělá to, co jsme plánovali.

Lemma: Algoritmus BFS se vždy zastaví.

Důkaz: Vnitřní cyklus je evidentně konečný. V každém průchodu vnějším cyklem uzavřeme jeden otevřený vrchol. Jednou uzavřený vrchol už ale svůj stav nikdy nezmění, takže vnější cyklus proběhne nejvýše tolikrát, kolik je všech vrcholů. □

Definice: Vrchol v je *dosažitelný* z vrcholu u, pokud v grafu existuje cesta z u do v.

Poznámka: Cesta se obvykle definuje tak, že se na ní nesmí opakovat vrcholy ani hrany. Na dosažitelnosti se samozřejmě nic nezmění, pokud budeme místo cest uvažovat sledy, na nichž je opakování povoleno:

Lemma: Pokud z vrcholu u do vrcholu v vede sled, pak tam vede i cesta.

Důkaz: Ze všech sledů z u do v vybereme ten nejkratší (co do počtu hran) a nahlédneme, že se jedná o cestu. Vskutku: kdyby se v tomto sledu opakoval nějaký vrchol t, mohli bychom část sledu mezi první a poslední návštěvou t „vystřihnout“ a získat tak sled o ještě menším počtu hran. A pokud se neopakují vrcholy, nemohou se opakovat ani hrany. □

Lemma: Když algoritmus doběhne, vrcholy dosažitelné z v_0 jsou *uzavřené* a všechny ostatní vrcholy *nenalezené*.

Důkaz: Nejprve si uvědomíme, že každý vrchol buďto po celou dobu běhu algoritmu zůstane nenalezený, nebo se nejprve stane otevřeným a později uzavřeným. Formálně bychom to mohli dokázat indukcí podle počtu iterací vnějšího cyklu.

Dále nahlédneme, že kdykoliv nějaký vrchol w otevřeme, musí být dosažitelný z v_0. Opět indukcí podle počtu iterací: na počátku je otevřený pouze vrchol v_0 sám. Kdykoliv pak otevíráme nějaký vrchol w, stalo se tak proto, že do něj vedla hrana z právě uzavíraného vrcholu v. Přitom podle indukčního předpokladu existuje sled z v_0 do v. Prodloužíme-li tento sled o hranu vw, vznikne sled z v_0 do w, takže i w je dosažitelný.

Zbývá dokázat, že se nemohlo stát, že by algoritmus nějaký dosažitelný vrchol neobjevil. Pro spor předpokládejme, že takové „špatné“ vrcholy existují. Vybereme z nich vrchol s, který je k v_0 nejbližší, tedy do kterého vede z v_0 sled o nejmenším možném počtu hran. Jelikož sám v_0 není špatný, musí existovat vrchol p, který je na sledu předposlední (z p do s vede poslední hrana sledu).

Vrchol p také nemůže být špatný, protože jinak bychom si jej vybrali místo s. Tím pádem ho algoritmus nalezl, otevřel a časem i zavřel. Při tomto zavírání ovšem musel prozkoumat všechny sousedy vrcholu p, tedy i vrchol s. Není proto možné, aby s unikl otevření. □

5.3 Reprezentace grafů

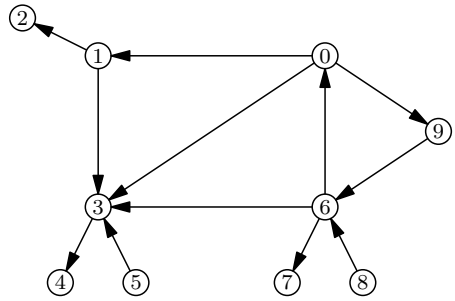
U algoritmu na prohledávání grafu do šířky jsme zatím nerozebrali časovou a paměťovou složitost. Není divu: algoritmus jsme popsali natolik abstraktně, že vůbec není jasné, jak dlouho trvá nalezení všech následníků vrcholu. Nyní tyto detaily doplníme.

Především se musíme rozhodnout, jak grafy reprezentovat v paměti počítače. Obvykle se používají následující způsoby:

Matice sousednosti. Vrcholy očíslujeme od 1 do n, hrany popíšeme maticí $n \times n$, která má na pozici i, j jedničku, je-li ij hrana, a jinak nulu. Jedničky v i-tém řádku tedy odpovídají následníkům vrcholu i, jedničky v j-tém sloupci předchůdcům vrcholu j. Pro neorientované grafy je matice symetrická.

Výhodou této reprezentace je, že dovedeme v konstantním čase zjistit, zda jsou dva vrcholy spojeny hranou. Vyjmenování hran vedoucích z daného vrcholu trvá $\Theta(n)$. Matice zabere prostor $\Theta(n^2)$.

Lepších parametrů obecně nemůžeme dosáhnout, protože sousedů může být až $n - 1$ a všech hran až řádově n^2. Často ale pracujeme s *řídkými grafy,* tedy takovými, které




```
  0123456789
0 0101000001
1 0011000000
2 0000000000
3 0000100000
4 0000000000
5 0001000000
6 1001000100
7 0000000000
8 0000001000
9 0000001000
```

Obrázek 5.5: „Prasátko“ a jeho matice sousednosti

mají méně než kvadraticky hran. Potkáváme je nečekaně často – řídké jsou například všechny rovinné grafy, čili i stromy. Pro ně se bude lépe hodit následující reprezentace.

Seznamy sousedů. Vrcholy opět očíslujeme od 1 do n. Pro každý vrchol uložíme seznam čísel jeho následníků. Přesněji řečeno, pořídíme si pole S, jehož i-tý prvek bude ukazovat na seznam následníků vrcholu i. V neorientovaném grafu zařadíme hranu ij do seznamů $S[i]$ i $S[j]$.

0: 1, 3, 9	1: 2, 3	2:	3: 4	4:
5: 3	6: 0, 3, 7	7:	8: 6	9: 6

Obrázek 5.6: Seznamy následníků pro prasátkový graf

Vyjmenování hran tentokrát stihneme lineárně s jejich počtem. Celá reprezentace zabírá prostor $\Theta(n + m)$. Ovšem test existence hrany ij se zpomalil: musíme projít všechny následníky vrcholu i.

Často se používají různá rozšíření této reprezentace. Například můžeme přidat seznamy předchůdců, abychom uměli vyjmenovávat i hrany vedoucí *do* daného vrcholu. Také můžeme čísla vrcholů nahradit ukazateli, pole seznamem a získat tak možnost graf za běhu libovolně upravovat. Pro neorientované grafy se pak hodí, aby byly oba výskyty téže hrany navzájem propojené.

Komprimované seznamy sousedů. Pokud chceme šetřit pamětí, může se hodit zkomprimovat seznamy následníků (či všech sousedů) do polí. Pořídíme si pole $S[1 \ldots m]$, ve kterém budou za sebou naskládaní nejdříve všichni následníci vrcholu 1, pak následníci dvojky, atd. Navíc založíme „rejstřík“ – pole $R[1 \ldots n]$, jehož i-tý prvek ukazuje na prvního následníka vrcholu i v poli S. Pokud navíc dodefinujeme $R[n+1] = m+1$, bude vždy platit, že následníci vrcholu i jsou v S na pozicích $R[i]$ až $R[i+1]-1$.

i	1	2	3	4	5	6	7	8	9	10	11	12
$S[i]$	1	3	9	2	3	4	3	0	3	7	6	6

i	0	1	2	3	4	5	6	7	8	9	10
$R[i]$	1	4	6	6	7	7	8	11	11	12	13

Obrázek 5.7: Komprimované seznamy následníků pro prasátkový graf

Tím jsme ušetřili ukazatele, což sice asymptotickou paměťovou složitost nezměnilo, ale přesto se to u velkých grafů může hodit. Základní operace jsou řádově stejně rychlé jako před kompresí, ovšem zkomplikovali jsme jakékoliv úpravy grafu.

Matice incidence. Často v literatuře narazíme na další maticovou reprezentaci grafů. Jedná se o matici tvaru $n \times m$, jejíž řádky jsou indexovány vrcholy a sloupce hranami. Sloupec, který popisuje hranu ij, má v i-tém řádku hodnotu -1, v j-tém řádku hodnotu 1 a všude jinde nuly. Pro neorientované grafy se znaménka buďto volí libovolně, nebo jsou obě kladná.

Tato matice hraje pozoruhodnou roli v důkazech různých vět na pomezí teorie grafů a lineární algebry (například Kirchhoffovy věty o počítání koster grafu pomocí determinantů). V algoritmech se ovšem nehodí – je obrovská a všechny základní grafové operace jsou s ní pomalé.

Vraťme se nyní k prohledávání do šířky. Pokud na vstupu dostane graf reprezentovaný seznamy sousedů, o jeho časové složitosti platí:

Lemma: BFS doběhne v čase $\mathcal{O}(n + m)$ a spotřebuje paměť $\Theta(n + m)$.

Důkaz: Inicializace algoritmu (kroky 1 až 6) trvá $\mathcal{O}(n)$. Jelikož každý vrchol uzavřeme nejvýše jednou, vnější cyklus proběhne nejvýše n-krát. Pokaždé spotřebuje konstantní čas na svou režii a navíc konstantní čas na každého nalezeného následníka. Celkem tedy $\mathcal{O}(n + \sum_i d_i)$, kde d_i je počet následníků vrcholu i. Tato suma je rovna počtu hran.

(Také si můžeme představovat, že algoritmus zkoumá vrcholy i hrany, obojí v konstantním čase. Každou hranu přitom prozkoumá v okamžiku, kdy uzavírá vrchol, z nějž tato hrana vede, čili právě jednou.)

Paměť jsme potřebovali na reprezentaci grafu, lineárně velkou frontu a lineárně velká pole. □

Cvičení

1. Jak reprezentovat multigrafy s násobnými hranami a smyčkami?

2. Jakou časovou složitost by BFS mělo, pokud bychom graf reprezentovali maticí sousednosti?

3. Navrhněte reprezentaci grafu, která bude efektivní pro řídké grafy, a přitom dokáže rychle testovat existenci hrany mezi zadanými vrcholy.

4. Je-li $\mathbf{A}$ matice sousednosti grafu, co popisuje matice $\mathbf{A}^2$? A co $\mathbf{A}^k$? (Mocniny matic definujeme takto: $\mathbf{A}^1 = \mathbf{A}$, $\mathbf{A}^{k+1} = \mathbf{A}^k\mathbf{A}$.)

5.* Na základě předchozího cvičení vytvořte algoritmus, který pomocí $\mathcal{O}(\log n)$ násobení matic spočítá *matici dosažitelnosti*. To je nula-jedničková matice $\mathbf{A}^*$, v níž $\mathbf{A}^*_{ij} = 1$ právě tehdy, když z i do j vede cesta. Ještě lepší časové složitosti můžete dosáhnout násobením matic pomocí Strassenova algoritmu z oddílu 10.5.

6. Je-li $\mathbf{I}$ matice incidence grafu, co popisují matice $\mathbf{I}^\mathrm{T}\mathbf{I}$ a $\mathbf{I}\mathbf{I}^\mathrm{T}$?

5.4 Komponenty souvislosti

Jakmile umíme prohledávat graf, hned dokážeme odpovídat na některé jednoduché otázky. Například umíme snadno zjistit, zda zadaný neorientovaný graf je souvislý: vybereme si libovolný vrchol a spustíme z něj BFS. Pokud jsme navštívili všechny vrcholy, graf je souvislý. V opačném případě jsme prošli celou jednu *komponentu souvislosti*. K nalezení ostatních komponent stačí opakovaně vybírat dosud nenavštívený vrchol a spouštět z něj prohledávání.

Následuje pseudokód algoritmu odvozený od BFS a mírně zjednodušený: zbytečně neinicializujeme vrcholy vícekrát a nerozlišujeme mezi otevřenými a uzavřenými vrcholy. Místo toho udržujeme pole C, které o navštívených vrcholech říká, do které komponenty patří; komponentu přitom identifikujeme nejmenším číslem vrcholu, který v ní leží.

Algoritmus KOMPONENTY
Vstup: Neorientovaný graf $G = (V, E)$

```
1. Pro všechny vrcholy v položíme C(v) ← nedefinováno.
2. Pro všechny vrcholy u postupně provádíme:
3.     Je-li C(u) nedefinováno:          ◁ nová komponenta, spustíme BFS
4.         C(u) ← u
5.         Založíme frontu Q a vložíme do ní vrchol u.
6.         Dokud Q není prázdná:
7.             Odebereme první vrchol z Q a označíme ho v.
8.             Pro všechny následníky w vrcholu v:
9.                 Pokud C(w) není definováno:
```

10. $C(w) \leftarrow u$
11. Přidáme w do fronty Q.

Výstup: Pole C přiřazující vrcholům komponenty

Korektnost algoritmu je zřejmá. Pro rozbor složitosti označme n_i a m_i počet vrcholů a hran v i-té nalezené komponentě. Prohledání této komponenty trvá $\Theta(n_i + m_i)$. Kromě toho algoritmus provádí inicializaci a hledá dosud neoznačené vrcholy, což se obojí týká každého vrcholu jen jednou. Celkově tedy spotřebuje čas $\Theta(n + \sum_i (n_i + m_i))$, což je rovno $\Theta(n+m)$, neboť každý vrchol i hrana leží v právě jedné komponentě. Paměti potřebujeme $\Theta(n)$ navíc k reprezentaci grafu.

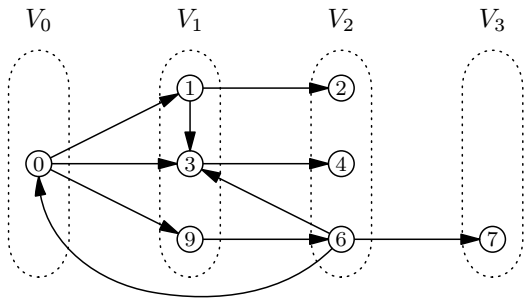
Cvičení

1. Navrhněte algoritmus, který v čase $\mathcal{O}(n + m)$ zjistí, zda zadaný graf je *bipartitní*. Tak se říká grafům, jejichž vrcholy lze rozdělit na dvě množiny tak, aby koncové vrcholy každé hrany patřily do různých množin.

5.5 Vrstvy a vzdálenosti

Z průběhu prohledávání do šířky lze zjistit spoustu dalších zajímavých informací o grafu. K tomu se budou hodit pomocná pole D a P, jež jsme si už v algoritmu přichystali.

Především můžeme vrcholy rozdělit do *vrstev* podle toho, v jakém pořadí je BFS prochází: ve vrstvě V_0 bude ležet vrchol v_0 a kdykoliv budeme zavírat vrchol z vrstvy V_k, jeho otevírané následníky umístíme do V_{k+1}. Algoritmus má tedy v každém okamžiku ve frontě vrcholy z nějaké vrstvy V_k, které postupně uzavírá, a za nimi přibývají vrcholy tvořící vrstvu V_{k+1}.



Obrázek 5.8: Vrstvy při prohledávání grafu z obr. 5.5 do šířky

Lemma: Je-li vrchol v dosažitelný, pak leží v nějaké vrstvě V_k a číslo $D(v)$ na konci výpočtu je rovno k. Navíc pokud $v \neq v_0$, pak $P(v)$ je nějaký předchůdce vrcholu v ležící ve vrstvě V_{k-1}. Tím pádem v, $P(v)$, $P(P(v))$, ..., v_0 tvoří cestu z v_0 do v (zapsanou pozpátku).

Důkaz: Vše provedeme indukcí podle počtu kroků algoritmu. Využijeme toho, že $D(v)$ a $P(v)$ se nastavují v okamžiku uzavření vrcholu v a pak už se nikdy nezmění. □

Čísla vrstev mají ovšem zásadnější význam:

Lemma: Je-li vrchol v dosažitelný z v_0, pak na konci výpočtu $D(v)$ udává jeho vzdálenost od v_0, měřenou počtem hran na nejkratší cestě.

Důkaz: Označme $d(v)$ skutečnou vzdálenost z v_0 do v. Z předchozího lemmatu víme, že z v_0 do v existuje cesta délky $D(v)$. Proto $D(v) \geq d(v)$.

Opačnou nerovnost dokážeme sporem: Nechť existují vrcholy, pro které je $D(v) > d(v)$. Takovým budeme opět říkat *špatné vrcholy* a vybereme z nich vrchol s, jehož skutečná vzdálenost $d(s)$ je nejmenší.

Uvážíme nejkratší cestu z v_0 do s a předposlední vrchol p na této cestě. Jelikož $d(p) = d(s) - 1$, musí p být dobrý (viz též cvičení 2), takže $D(p) = d(p)$. Nyní zaostřeme na okamžik, kdy algoritmus zavírá vrchol p. Tehdy musel objevit vrchol s jako následníka. Pokud byl v tomto okamžiku s dosud nenalezený, musel padnout do vrstvy $d(p)+1 = d(s)$, což je spor. Jenže pokud už byl otevřený nebo uzavřený, musel dokonce padnout do nějaké dřívější vrstvy, což je spor tím spíš. □

Strom prohledávání a klasifikace hran

Vrstvy vypovídají nejen o vrcholech, ale také o hranách grafu. Je-li ij hrana, rozlišíme následující možnosti:

- $D(j) < D(i)$ – hrana vede do některé z minulých vrstev. V okamžiku uzavírání i byl j už uzavřený. Takovým hranám budeme říkat *zpětné.*
- $D(j) = D(i)$ – hrana vede v rámci téže vrstvy. V okamžiku uzavírání i byl j buďto uzavřený, nebo ještě otevřený. Tyto hrany se nazývají *příčné.*
- $D(j) = D(i)+1$ – hrana vede do následující vrstvy (povšimněte si, že nemůže žádnou vrstvu přeskočit, protože by neplatila trojúhelníková nerovnost pro vzdálenost).
  - Pokud při uzavírání i byl j dosud nenalezený, tak jsme j právě otevřeli a nastavili $P(j) = i$. Tehdy budeme hraně ij říkat *stromová* a za chvíli prozradíme, proč.
  - V opačném případě byl j otevřený a hranu prohlásíme za *dopřednou.*

Lemma: Stromové hrany tvoří strom na všech dosažitelných vrcholech, orientovaný směrem od kořene, jímž je vrchol v_0. Cesta z libovolného vrcholu v do v_0 v tomto stromu je nejkratší cestou z v_0 do v v původním grafu. Proto se tomuto stromu říká *strom nejkratších cest.*

Důkaz: Graf stromových hran musí být strom s kořenem v_0, protože vzniká z vrcholu v_0 postupným přidáváním listů. Na každé hraně přitom roste číslo vrstvy o 1 a jak už víme, čísla vrstev odpovídají vzdálenostem, takže cesty ve stromu jsou nejkratší. □

Dodejme ještě, že stromů nejkratších cest může pro jeden graf existovat vícero (ani nejkratší cesty samotné nejsou jednoznačně určené, pouze vzdálenosti). Každý takový strom je ovšem kostrou prohledávaného grafu.

Pozorování: V neorientovaných grafech BFS potká každou dosažitelnou hranu dvakrát: buďto poprvé jako stromovou a podruhé jako zpětnou, nebo nejdříve jako dopřednou a pak jako zpětnou, anebo v obou případech jako příčnou. Tím pádem nemohou existovat zpětné hrany, které by se vracely o víc než jednu vrstvu.

Nyní pojďme vše, co jsme zjistili o algoritmu BFS, shrnout do následující věty:

Věta: Prohledávání do šířky doběhne v čase $\mathcal{O}(n+m)$ a spotřebuje prostor $\Theta(n+m)$. Po skončení výpočtu popisuje pole *stav* dosažitelnost z vrcholu v_0, pole D obsahuje vzdálenosti od vrcholu v_0 a pole P kóduje strom nejkratších cest.

Cvičení

1. Upravte BFS tak, aby pro každý dosažitelný vrchol zjistilo, kolik do něj vede nejkratších cest z počátečního vrcholu. Zachovejte lineární časovou složitost.

2. V důkazu lemmatu o vzdálenostech jsme považovali za samozřejmost, že usekneme-li nejkratší cestu z v_0 do s v nějakém vrcholu p, zbude z ní nejkratší cesta z v_0 do p. Jinými slovy, prefix nejkratší cesty je zase nejkratší cesta. Dokažte formálně, že je to pravda.

3. BFS v každém okamžiku zavírá nejstarší otevřený vrchol. Jak by se chovalo, kdybychom vybírali otevřený vrchol podle nějakého jiného kritéria? Která z dokázaných lemmat by stále platila a která ne?

4. Na jisté šachovnici žil kulhavý kůň. To je zvláštní šachová figurka, která v sudých tazích táhne jako jezdec, v lichých jako pěšec. Vymyslete algoritmus, který z jednoho zadaného políčka dokulhá na druhé na nejmenší možný počet tahů.

5. Mějme mapu Manhattanu: čtverečkovaný papír, křížení čar odpovídají křižovatkám, úsečky mezi nimi jednotlivým streets a avenues, z nichž některé jsou neprůjezdné kvůli dopravní zácpě. Zrovna se nám v jedné ulici porouchalo auto a nyní dovede

pouze jezdit rovně a odbočovat doprava. Nalezněte nejkratší cestu do servisu (na zadanou křižovatku).

6. Hrdina Théseus se vypravil do hlubin labyrintu a snaží se najít poklad. Chodbami labyrintu se ovšem pohybuje hladový Mínótauros a snaží se najít Théseu. Labyrint má tvar čtvercové sítě, jejíž každé políčko je buďto volné prostranství, anebo zeď. Známe mapu labyrintu a počáteční polohy Théseu, Mínótaura a pokladu. Théseus se v jednom tahu pohne na vybrané sousední políčko. Poté se vždy dvakrát pohne o políčko Mínótauros: pokaždé se pokusí zmenšit o 1 rozdíl své a Théseovy x-ové souřadnice, pokud to nejde, pak y-ové, pokud nejde ani to, stojí. Poraďte Théseovi, jak má dojít k pokladu a vyhnout se Mínótaurovi.

7. Koupili jste na inzerát dvojici skvělých robotů. Lacino, neboť jsou právě uvěznění v bludišti (čtvercová síť s některými políčky blokovanými). Znáte jejich polohy a můžete jim rádiem vysílat povely pro posun o políčko na sever, jih, východ či západ, abyste je dostali na okraj bludiště. Háček je ale v tom, že na každý povel reagují oba roboti. Vymyslete algoritmus, který najde nejkratší posloupnost povelů, jež vysvobodí oba roboty. Dodejme ještě, že robot ignoruje povel, který by způsobil okamžitý náraz do zdi nebo do druhého robota, a že jakmile se robot dostane na okraj, odchytíme ho a další povely neposlouchá.

   Poznamenejme, že úloha by byla řešitelná i tehdy, kdybychom počáteční polohy robotů neznali. To je ovšem mnohem obtížnější a potřebná „univerzální posloupnost povelů“ mnohem delší.

5.6 Prohledávání do hloubky

Dalším důležitým algoritmem k procházení grafů je *prohledávání do hloubky,* anglicky *depth-first search* čili *DFS.* Je založeno na podobném principu jako BFS, ale vrcholy zpracovává rekurzivně: kdykoliv narazí na dosud nenalezený vrchol, otevře ho, zavolá se rekurzivně na všechny jeho dosud nenalezené následníky, načež původní vrchol zavře a vrátí se z rekurze.

Algoritmus opět zapíšeme v pseudokódu a rovnou ho doplníme o pomocná pole *in* a *out*. Do nich zaznamenáme, v jakém pořadí jsme vrcholy otevírali a zavírali.

Algoritmus DFS (prohledávání do hloubky)
Vstup: Graf $G = (V, E)$ a počáteční vrchol $v_0 \in V$

1. Pro všechny vrcholy v:
2. $\quad stav(v) \leftarrow$ *nenalezený*
3. $\quad in(v), out(v) \leftarrow$ *nedefinováno*

4. $T \leftarrow 0$ ◁ *globální počítadlo kroků*
5. Zavoláme DFS2(v_0).

Procedura DFS2(v)

1. $stav(v) \leftarrow$ *otevřený*
2. $T \leftarrow T + 1$, $in(v) \leftarrow T$
3. Pro všechny následníky w vrcholu v:
4. Je-li $stav(w) =$ *nenalezený*, zavoláme DFS2(w).
5. $stav(v) \leftarrow$ *uzavřený*
6. $T \leftarrow T + 1$, $out(v) \leftarrow T$

Rozbor algoritmu povedeme podobně jako u BFS.

Lemma: DFS doběhne v čase $\mathcal{O}(n + m)$ a prostoru $\Theta(n + m)$.

Důkaz: Každý vrchol, který nalezneme, přejde nejprve do otevřeného stavu a posléze do uzavřeného, kde už setrvá. Každý vrchol proto uzavíráme nejvýše jednou a projdeme při tom hrany, které z něj vedou. Strávíme tak konstantní čas nad každým vrcholem a každou hranou.

Kromě reprezentace grafu algoritmus potřebuje lineárně velkou paměť na pomocná pole a na zásobník rekurze. □

Lemma: DFS navštíví právě ty vrcholy, které jsou z v_0 dosažitelné.

Důkaz: Nejprve indukcí podle běhu algoritmu nahlédneme, že každý navštívený vrchol je dosažitelný. Opačnou implikaci zase dokážeme sporem: ze špatných vrcholů (dosažitelných, ale nenavštívených) vybereme ten, který je k v_0 nejbližší, a zvolíme jeho předchůdce na nejkratší cestě. Ten nemůže být špatný, takže byl otevřen, a tím pádem musel být posléze otevřen i náš špatný vrchol. Spor. □

Prohledávání do hloubky nemá žádnou přímou souvislost se vzdálenostmi v grafu. Přesto pořadí, v němž navštěvuje vrcholy, skýtá mnoho pozoruhodných vlastností. Ty nyní prozkoumáme.

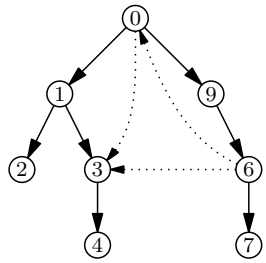
Chod algoritmu můžeme elegantně popsat pomocí řetězce závorek. Kdykoliv vstoupíme do vrcholu, připíšeme k řetězci levou závorku; až budeme tento vrchol opouštět, připíšeme pravou. Z průběhu rekurze je vidět, že jednotlivé páry závorek se nebudou křížit – dostali jsme tedy dobře uzávorkovaný řetězec s n levými a n pravými závorkami. Hodnoty $in(v)$ a $out(v)$ nám řeknou, kde v tomto řetězci leží levá a pravá závorka přiřazená vrcholu v.

Každé uzávorkování odpovídá nějakému stromu. V našem případě je to tak zvaný *DFS strom,* který je tvořen hranami z právě uzavíraného vrcholu do jeho nově objevených

následníků (těmi, po kterých jsme se rekurzivně volali). Je to tedy strom zakořeněný ve vrcholu v_0 a jeho hrany jsou orientované směrem od kořene. Budeme ho kreslit tak, že hrany vedoucí z každého vrcholu uspořádáme zleva doprava podle toho, jak je DFS postupně objevovalo.

Na průběh DFS se tedy také dá dívat jako na procházení DFS stromu. Vrcholy, které leží na cestě z kořene do aktuálního vrcholu, jsou přesně ty, které už jsme otevřeli, ale zatím nezavřeli. Rekurze je má na zásobníku a v závorkové reprezentaci odpovídají levým závorkám, jež jsme vypsali, ale dosud neuzavřeli pravými. Tyto vrcholy tvoří bájnou Ariadninu nit, po níž se vracíme směrem ke vchodu do bludiště.

Obrázek 5.9 ukazuje, jak vypadá průchod DFS stromem pro „prasátkový“ graf z obrázku 5.5. Začali jsme vrcholem 0 a pokaždé jsme probírali následníky od nejmenšího k největšímu.



$(_0(_1(_2)_2(_3(_4)_4)_3)_1(_9(_6(_7)_7)_6)_9)_0$

v	$in(v)$	$out(v)$
0	1	16
1	2	9
2	3	4
3	5	8
4	6	7
5	—	—
6	11	14
7	12	13
8	—	—
9	10	15

Obrázek 5.9: Průběh DFS na prasátkovém grafu

Pozorování: Hranám grafu můžeme přiřadit typy podle toho, v jakém vztahu jsou odpovídající závorky, čili v jaké poloze je hrana vůči DFS stromu. Tomuto přiřazení se obvykle říká *DFS klasifikace hran.* Pro hranu xy rozlišíme tyto možnosti:

- $(_x \ldots (_y \ldots)_y \ldots)_x$. Tehdy mohou nastat dva případy:
  - Vrchol y byl při uzavírání x nově objeven. Taková hrana leží v DFS stromu, a proto jí říkáme *stromová.*
  - Vrchol y jsme už znali, takže v DFS stromu leží v nějakém podstromu pod vrcholem x. Těmto hranám říkáme *dopředné.*
- $(_y \ldots (_x \ldots)_x \ldots)_y$ – vrchol y leží na cestě ve stromu z kořene do x a je dosud otevřený. Takové hrany se nazývají *zpětné.*

- $(_y \ldots)_y \ldots (_x \ldots)_x$ – vrchol y byl už uzavřen a rekurze se z něj vrátila. Ve stromu není ani předkem, ani potomkem vrcholu x, nýbrž leží v nějakém podstromu odpojujícím se doleva od cesty z kořene do x. Těmto hranám říkáme *příčné*.
- $(_x \ldots)_x \ldots (_y \ldots)_y$ – případ, kdy by vedla hrana z uzavřeného vrcholu x do vrcholu y, který bude otevřen teprve v budoucnosti, nemůže nastat. Před uzavřením x totiž prozkoumáme všechny hrany vedoucí z x a do případných nenalezených vrcholů se rovnou vydáme.

Na obrázku 5.9 jsou stromové hrany nakresleny plnými čarami a ostatní tečkovaně. Hrana $(6, 0)$ je zpětná, $(0, 3)$ dopředná a $(6, 3)$ příčná.

V neorientovaných grafech se situace zjednoduší. Každou hranu potkáme dvakrát: buďto poprvé jako stromovou a podruhé jako zpětnou, nebo poprvé jako zpětnou a podruhé jako dopřednou. Příčné hrany se neobjeví (k nim opačné by totiž byly onoho posledního druhu, který se nemůže vyskytnout).

K rozpoznání typu hrany vždy stačí porovnat hodnoty *in* a *out* a případně stavy obou krajních vrcholů. Zvládneme to tedy v konstantním čase.

Shrňme, co jsme v tomto oddílu zjistili:

Věta: Prohledávání do hloubky doběhne v čase $\mathcal{O}(n+m)$ a spotřebuje prostor $\Theta(n+m)$. Jeho výsledkem je dosažitelnost z počátečního vrcholu, DFS strom a klasifikace všech hran v dosažitelné části grafu.

Cvičení

1. Jakou časovou složitost by DFS mělo, pokud bychom graf reprezentovali maticí sousednosti?

2. Nabízí se svůdná myšlenka, že DFS získáme z BFS nahrazením fronty zásobníkem. To by například znamenalo, že si můžeme ušetřit většinu analýzy algoritmu a jen se odkázat na obecný prohledávací algoritmus z cvičení 5.5.3. Na čem tento přístup selže?

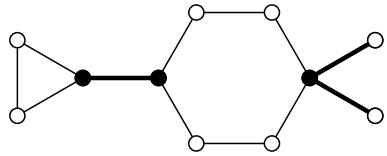
3. Zkontrolujte, že DFS klasifikace je kompletní, tedy že jsme probrali všechny možné polohy hrany vzhledem ke stromu.

4. Mějme souvislý orientovaný graf. Chceme mazat jeho vrcholy jeden po druhém tak, aby graf zůstal stále souvislý. Jak takové pořadí mazání najít?

5. Může být v orientovaném grafu kružnice složená ze samých zpětných hran? A v neorientovaném?

5.7 Mosty a artikulace

DFS klasifikaci hran lze elegantně použít pro hledání *mostů* a *artikulací* v souvislých neorientovaných grafech. Most se říká hraně, jejímž odstraněním se graf rozpadne na komponenty. Artikulace je vrchol s toutéž vlastností.



Obrázek 5.10: Mosty a artikulace grafu

Mosty

Začneme klasickou charakteristikou mostů:

Lemma: Hrana *není most* právě tehdy, když leží na alespoň jedné kružnici.

Důkaz: Pokud hrana xy není most, musí po jejím odebrání stále existovat nějaká cesta mezi vrcholy x a y. Tato cesta spolu s hranou xy tvoří kružnici v původním grafu.

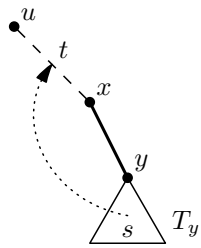
Naopak leží-li xy na nějaké kružnici C, nemůže se odebráním této hrany graf rozpadnout. V libovolném sledu, který používal hranu xy, totiž můžeme tuto hranu nahradit zbytkem kružnice C. □

Nyní si rozmysleme, které typy hran podle DFS klasifikace mohou být mosty. Jelikož každou hranu potkáme v obou směrech, bude nás zajímat její typ při prvním setkání:

- Stromové hrany mohou, ale nemusí být mosty.
- Zpětné hrany nejsou mosty, protože spolu s cestou ze stromových hran uzavírají kružnici.
- Dopředné ani příčné hrany v neorientovaných grafech nepotkáme.

Stačí tedy umět rozhodnout, zda daná stromová hrana leží na kružnici. Jak by tato kružnice mohla vypadat? Nazveme x a y krajní vrcholy stromové hrany, přičemž x je vyšší z nich (bližší ke kořeni). Označme T_y podstrom DFS stromu tvořený vrcholem y a všemi jeho potomky. Pokud kružnice projde z x do y, právě vstoupila do podstromu T_y a než se vrátí do x, zase musí tento podstrom opustit. To ale může pouze po zpětné hraně: po jediné stromové jsme vešli a dopředné ani příčné neexistují.

Chceme tedy zjistit, zda existuje zpětná hrana vedoucí z podstromu T_y ven, to znamená na stromovou cestu mezi kořenem a x.



Obrázek 5.11: Zpětná hrana st způsobuje, že xy není most

Pro konkrétní zpětnou hranu tuto podmínku ověříme snadno: Je-li st zpětná hrana a s leží v T_y, stačí otestovat, zda t je výše než y, nebo ekvivalentně zda $in(t) < in(y)$ – to je totéž, neboť in na každé stromové cestě shora dolů roste.

Abychom nemuseli pokaždé prohledávat všechny zpětné hrany z podstromu, provedeme jednoduchý předvýpočet: pro každý vrchol v spočítáme $low(v)$, což bude minimum z inů všech vrcholů, do nichž se lze dostat z T_v zpětnou hranou. Můžeme si představit, že to říká, jak vysoko lze z podstromu dosáhnout.

Pak už je testování mostů snadné: stromová hrana xy leží na kružnici právě tehdy, když $low(y) < in(y)$.

Předvýpočet hodnot $low(v)$ lze přitom snadno zabudovat do DFS: kdykoliv se z nějakého vrcholu v vracíme, spočítáme minimum z low jeho synů a z inů vrcholů, do nichž z v vedou zpětné hrany. Lépe je to vidět z následujícího zápisu algoritmu. DFS jsme mírně upravili, aby nepotřebovalo explicitně udržovat stavy vrcholů a vystačilo si s polem in.

Algoritmus MOSTY
Vstup: Souvislý neorientovaný graf $G = (V, E)$
1. $M \leftarrow \emptyset$ ◁ *seznam dosud nalezených mostů*
2. $T \leftarrow 0$ ◁ *počítadlo kroků*
3. Pro všechny vrcholy v nastavíme $in(v) \leftarrow$ *nedefinováno*.
4. Zvolíme libovolně vrchol $u \in V$.
5. Zavoláme MOSTY2(u, *nedefinováno*).

Výstup: Seznam mostů M

Procedura MOSTY2(v, p)
Vstup: Kořen podstromu v, jeho otec p
1. $T \leftarrow T + 1$, $in(v) \leftarrow T$

```
2.  low(v) ← +∞
3.  Pro všechny následníky w vrcholu v:
4.      Pokud in(w) není definován:                    ◁ hrana vw je stromová
5.          Zavoláme MOSTY2(w, v).
6.          Pokud low(w) ≥ in(w):                      ◁ vw je most
7.              Přidáme hranu vw do seznamu M.
8.          low(v) ← min(low(v), low(w))
9.      Jinak je-li w ≠ p a in(w) < in(v):             ◁ zpětná hrana
10.         low(v) ← min(low(v), in(w))
```

Snadno nahlédneme, že takto upravené DFS stále tráví konstantní čas nad každým vrcholem a hranou, takže běží v čase $\Theta(n+m)$. Paměti zabere $\Theta(n+m)$, neboť oproti DFS ukládá navíc pouze pole *low*.

Artikulace

I artikulace je možné charakterizovat pomocí kružnic a stromových/zpětných hran, jen je to maličko složitější.

Lemma A: Vrchol v *není artikulace* právě tehdy, když pro každé dva jeho různé sousedy x a y existuje kružnice, na níž leží hrany vx i vy.

Důkaz: Pokud v není artikulace, pak po odebrání vrcholu v (a tedy i hran vx a vy) musí mezi vrcholy x a y nadále existovat nějaká cesta. Doplněním hran xv a vy k této cestě dostaneme kýženou kružnici.

V opačném směru: Nechť každé dvě hrany incidentní s v leží na společné kružnici. Poté můžeme libovolnou cestu, která spojovala ostatní vrcholy a procházela při tom přes v, upravit na sled, který v nepoužije: pokud cesta do v vstoupila z nějakého vrcholu x a odchází do y, nahradíme hrany xv a vy opačným obloukem příslušné kružnice. Tím pádem graf zůstane po odebrání v souvislý a v není artikulace. □

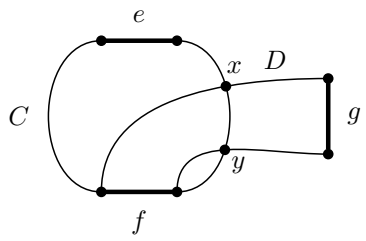
Definice: Zavedeme binární relaci $\approx$ na hranách grafu tak, že hrany e a f jsou v relaci právě tehdy, když $e = f$ nebo e a f leží na společné kružnici.

Lemma E: Relace $\approx$ je ekvivalence.

Důkaz: Reflexivita a symetrie jsou zřejmé z definice, ale potřebujeme ověřit tranzitivitu. Chceme tedy dokázat, že kdykoliv $e \approx f$ a $f \approx g$, pak také $e \approx g$. Víme, že existuje společná kružnice C pro e, f a společná kružnice D pro f a g. Potřebujeme najít kružnici, na níž leží současně e a g. Sledujme obrázek 5.12.

Vydáme se po kružnici D jedním směrem od hrany g, až narazíme na kružnici C (stát se to musí, protože hrana f leží na C i D). Vrchol, kde jsme se zastavili, označme x.

Podobně při cestě opačným směrem získáme vrchol y. Snadno ověříme, že vrcholy x a y musí být různé.



Obrázek 5.12: Situace v důkazu lemmatu **E**

Hledaná společná kružnice bude vypadat takto: začneme hranou g, pak se vydáme po kružnici D do vrcholu x, z něj po C směrem od vrcholu y ke hraně e, projdeme touto hranou, pokračujeme po C do vrcholu y a pak po D zpět ke hraně g. □

Ekvivalenčním třídám relace $\approx$ se říká *komponenty vrcholové 2-souvislosti* nebo také *bloky.* Pozor na to, že na rozdíl od komponent obyčejné souvislosti to jsou množiny hran, nikoliv vrcholů.

Pozorování: Vrchol v je artikulace právě tehdy, sousedí-li s hranami z alespoň dvou různých bloků.

Nyní povoláme na pomoc DFS klasifikaci. Uvážíme všechny hrany incidentní s nějakým vrcholem v. Nejprve si všimneme, že každá zpětná hrana je v bloku s některou ze stromových hran, takže stačí zkoumat pouze stromové hrany.

Dále nahlédneme, že pokud jsou dvě stromové hrany vedoucí z v dolů v témže bloku, pak musí v tomto bloku být i stromová hrana z v nahoru. Důvod je nasnadě: podstromy visící pod stromovými hranami nemohou být propojeny přímo (příčné hrany neexistují), takže je musíme nejprve opustit zpětnou hranou a pak se vrátit přes otce vrcholu v.

Zbývá tedy pro každou stromovou hranu mezi v a jeho synem zjistit, zda leží na kružnici se stromovou hranou z v nahoru. K tomu opět použijeme hodnoty *low*: je-li s syn vrcholu v, stačí otestovat, zda $low(s) < in(v)$.

Přímočarým důsledkem je, že má-li kořen DFS stromu více synů, pak je artikulací – hrany do jeho synů nemohou být propojeny ani přímo, ani přes vyšší patra stromu.

Stačí nám proto v algoritmu na hledání mostů vyměnit podmínku porovnávající *low* s *in* a hned hledá artikulace. Časová i paměťová složitost zůstávají lineární s velikostí grafu. Detailní zápis algoritmu ponechme jako cvičení.

Cvičení

1. Dokažte, že pokud se v souvislém grafu na alespoň třech vrcholech nachází most, pak je tam také artikulace. Ukažte, že opačná implikace neplatí.

2. Zapište v pseudokódu nebo naprogramujte algoritmus na hledání artikulací.

3. Definujme relaci $\sim$ na vrcholech tak, že $x \sim y$ právě tehdy, leží-li x a y na nějakém společném cyklu (uzavřeném sledu bez opakování hran). Dokažte, že tato relace je ekvivalence. Jejím ekvivalenčním třídám se říká *komponenty hranové 2-souvislosti,* jednotlivé třídy jsou navzájem pospojovány mosty. Upravte algoritmus na hledání mostů, aby graf rozložil na tyto komponenty.

4. *Blokový graf* $\mathcal{B}(G)$ grafu G popisuje vztahy mezi jeho bloky. Má dva typy vrcholů: jedny odpovídají artikulacím, druhé blokům (most považujeme za speciální případ bloku). Hranou spojíme vždy artikulaci se všemi incidentními bloky. Dokažte, že blokový graf je vždy strom, a rozšiřte algoritmus pro hledání artikulací, aby ho sestrojil.

5.* *Ušaté lemma* říká, že každý graf bez artikulací je možné postupně sestrojit tak, že vyjdeme z nějaké kružnice a postupně k ní přilepujeme „uší“ – cesty, jejichž krajní vrcholy už byly sestrojeny, a vnitřní vrcholy ještě ne. Vymyslete algoritmus, jenž pro zadaný graf najde příslušný postup přilepování uší. Jak vypadá analogické tvrzení a algoritmus pro grafy bez mostů?

6. *Náhrdelník* je graf sestávající z kružnic $C_1, \ldots, C_k$ libovolných délek, kde každé dvě sousední kružnice C_i a C_{i+1} mají společný právě jeden vrchol a nesousední kružnice jsou disjunktní. Navrhněte algoritmus, který na vstupu dostane graf a dva jeho vrcholy u, v a zjistí, zda u a v je možné propojit náhrdelníkem. Tedy zda v grafu existuje podgraf izomorfní nějakému náhrdelníku $C_1, \ldots, C_k$, v němž $u \in C_1$ a $v \in C_k$. Co kdybychom chtěli najít náhrdelník mezi u a v s nejmenším počtem kružnic?

5.8 Acyklické orientované grafy

Častým případem orientovaných grafů jsou *acyklické orientované grafy* neboli *DAGy* (z anglického *directed acyclic graph*). Pro ně umíme řadu problémů vyřešit efektivněji než pro obecné grafy. Mnohdy k tomu využíváme existenci topologického pořadí vrcholů, které zavedeme v tomto oddílu.

Detekce cyklů

Nejprve malá rozcvička: Jak poznáme, jestli zadaný orientovaný graf je DAG? K tomu použijeme DFS, které budeme opakovaně spouštět, než prozkoumáme celý graf (buď stejně, jako jsme to dělali při testování souvislosti v oddílu 5.4, nebo trikem z cvičení 1).

Lemma: V grafu existuje cyklus právě tehdy, najde-li DFS alespoň jednu zpětnou hranu.

Důkaz: Pakliže DFS najde nějakou zpětnou hranu xy, doplněním cesty po stromových hranách z y do x vznikne cyklus. Teď naopak dokážeme, že na každém cyklu leží alespoň jedna zpětná hrana.

Mějme nějaký cyklus a označme x jeho vrchol s nejnižším *out*em. Tím pádem na hraně vedoucí z x do následujícího vrcholu na cyklu roste *out*, což je podle klasifikace možné pouze na zpětné hraně. □

Topologické uspořádání

Důležitou vlastností DAGů je, že jejich vrcholy lze efektivně uspořádat tak, aby všechny hrany vedly po směru tohoto uspořádání. (Nabízí se představa nakreslení vrcholů na přímku tak, že hrany směřují výhradně zleva doprava.)

Definice: Lineární uspořádání $\prec$ na vrcholech grafu nazveme *topologickým uspořádáním vrcholů,* pokud pro každou hranu xy platí, že $x \prec y$.

Věta: Orientovaný graf má topologické uspořádání právě tehdy, je-li to DAG.

Důkaz: Existuje-li v grafu cyklus, brání v existenci topologického uspořádání: pro vrcholy na cyklu by totiž muselo platit $v_1 \prec v_2 \prec \ldots \prec v_k \prec v_1$.

Naopak v acyklickém grafu můžeme vždy topologické uspořádání sestrojit. K tomu se bude hodit následující pomocné tvrzení:

> **Lemma:** V každém neprázdném DAGu existuje *zdroj,* což je vrchol, do kterého nevede žádná hrana.
>
> *Důkaz:* Zvolíme libovolný vrchol v a půjdeme z něj proti směru hran, dokud nenarazíme na zdroj. Tento proces ovšem nemůže pokračovat do nekonečna, protože vrcholů je jen konečně mnoho a kdyby se nějaký zopakoval, našli jsme v DAGu cyklus. □

Pokud je náš DAG prázdný, topologické uspořádání je triviální. V opačném případě nalezneme zdroj, prohlásíme ho za první vrchol v uspořádání a odstraníme ho včetně všech hran, které z něj vedou. Tím jsme opět získali DAG a postup můžeme iterovat, dokud zbývají vrcholy. □

Důkaz věty nám rovnou dává algoritmus pro konstrukci topologického uspořádání, s trochou snahy lineární (cvičení 2). My si ovšem všimneme, že takové uspořádání lze přímo vykoukat z průběhu DFS:

Věta: Pořadí, v němž DFS opouští vrcholy, je opačné k topologickému.

Důkaz: Stačí dokázat, že pro každou hranu xy platí $out(x) > out(y)$. Z klasifikace hran víme, že je to pravda pro všechny typy hran kromě zpětných. Zpětné hrany se nicméně v DAGu nemohou vyskytovat. □

Stačí tedy do DFS doplnit, aby kdykoliv opouští vrchol, připojilo ho na začátek seznamu popisujícího uspořádání. Časová i paměťová složitost zůstávají lineární.

Topologická indukce

Ukažme alespoň jednu z mnoha aplikací topologického uspořádání. Dostaneme DAG a nějaký vrchol u a chceme spočítat pro všechny vrcholy, kolik do nich z u vede cest. Označme $c(v)$ hledaný počet cest z u do v.

Nechť $v_1, \ldots, v_n$ je topologické pořadí vrcholů a $u = v_k$ pro nějaké k. Tehdy $c(v_1) = c(v_2) = \ldots = c(v_{k-1}) = 0$, neboť do těchto vrcholů se z u nelze dostat. Také jistě platí $c(v_k) = 1$. Dále můžeme pokračovat indukcí:

Předpokládejme, že už známe $c(v_1)$ až $c(v_{\ell-1})$ a chceme zjistit $c(v_\ell)$. Jak vypadají cesty z u do v_ℓ? Musí se skládat z cesty z u do nějakého předchůdce w vrcholu v_ℓ, na níž je napojena hrana wv_ℓ. Všichni předchůdci ovšem leží v topologickém uspořádání před v_ℓ, takže pro ně známe počty cest z u. Hledané $c(v_\ell)$ je tedy součtem hodnot $c(w)$ přes všechny předchůdce w vrcholu v_ℓ.

Tento výpočet proběhne v čase $\Theta(n + m)$, neboť součty přes předchůdce dohromady projdou po každé hraně právě jednou.

Další aplikace topologické indukce naleznete v cvičeních.

Cvičení

1. Opakované spouštění DFS můžeme nahradit následujícím trikem: přidáme nový vrchol a hrany z tohoto vrcholu do všech ostatních. DFS spuštěné z tohoto „superzdroje“ projde na jedno zavolání celý graf. Nahlédněte, že jsme tím zachovali acykličnost grafu, a všimněte si, že chod tohoto algoritmu je stejný jako chod opakovaného DFS.

2. Ukažte, jak konstrukci topologického uspořádání postupným otrháváním zdrojů provést v čase $\mathcal{O}(n + m)$.

3. Příklad topologické indukce z tohoto oddílu by šel vyřešit i jednoduchou úpravou DFS, která by hodnoty $c(v)$ počítala rovnou při opouštění vrcholů. Ukažte jak.

4. Vymyslete, jak pomocí topologické indukce najít v lineárním čase délku nejkratší cesty mezi vrcholy u a v v DAGu s ohodnocenými hranami. Nejkratšími cestami v obecných grafech se budeme zabývat v příští kapitole.

5. Ukažte totéž pro nejdelší cestu, což je problém, který v obecných grafech zatím neumíme řešit v polynomiálním čase.

6. Jak spočítat, kolik mezi danými dvěma vrcholy neohodnoceného orientovaného grafu (ne nutně acyklického) vede nejkratších cest?

7. Kdy je topologické uspořádání grafu určeno jednoznačně?

8. *Sekvenční plánování:* Chceme provést řadu činností, které na sobě závisí – například sníst k večeři kančí guláš vyžaduje předem ho uvařit, pročež je potřeba kance někdy předtím ulovit. Situaci můžeme popsat grafem: vrcholy odpovídají činnostem, orientované hrany závislostem. Topologické uspořádání grafu pak vyjadřuje možné pořadí postupného provádění činností. Sestavte závislostní graf pro všechno, co potřebujete udělat, když ráno vstáváte, a najděte všechna topologická uspořádání.

9. *Paralelní plánování:* Stavíme dům a podobně jako v předchozím cvičení si sestavíme závislostní graf všech činností. Máme k dispozici dostatek pracovníků, takže zvládneme provádět libovolně mnoho činností současně. Stále ovšem musí být splněny závislosti: dřív, než se do nějaké činnosti pustíme, musí být hotové vše, na čem závisí. Vrcholy grafu ohodnotíme časem potřebným na vykonání činnosti. Spočítejte pro každou činnost, kdy se do ní máme pustit, abychom dům dostavěli co nejdříve.

10. *Kritické vrcholy:* O vrcholu grafu z předchozího cvičení řekneme, že je kritický, pokud by zpomalení příslušné činnosti způsobilo pozdější dokončení celého domu. Při řízení stavby si proto na takové vrcholy musíme dávat pozor. Vymyslete, jak je všechny najít.

5.9* Silná souvislost a její komponenty

Nyní se zamyslíme nad tím, jak rozšířit pojem souvislosti na orientované grafy. Intuitivně můžeme souvislost vnímat dvojím způsobem: Buďto tak, že kdykoliv graf rozdělíme na dvě části, vede mezi nimi alespoň jedna hrana. Anebo chceme, aby mezi každými dvěma vrcholy šlo přejít po cestě. Zatímco pro neorientované grafy tyto vlastnosti splývají, v orientovaných se liší, což vede ke dvěma různým definicím souvislosti:

Definice: Orientovaný graf je *slabě souvislý,* pokud zrušením orientace hran dostaneme souvislý neorientovaný graf.

Definice: Orientovaný graf je *silně souvislý,* jestliže pro každé dva vrcholy x a y existuje orientovaná cesta jak z x do y, tak opačně.

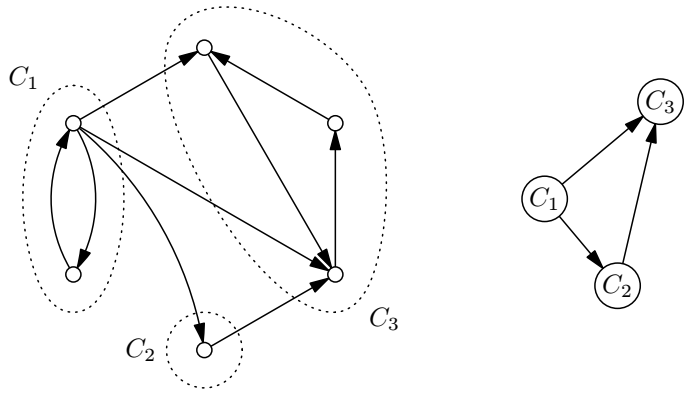
Slabá souvislost je algoritmicky triviální. V tomto oddílu ukážeme, jak lze rychle ověřovat silnou souvislost. Nejprve pomocí vhodné ekvivalence zavedeme její komponenty.

Definice: Buď $\leftrightarrow$ binární relace na vrcholech grafu definovaná tak, že $x \leftrightarrow y$ právě tehdy, existuje-li orientovaná cesta jak z x do y, tak z y do x.

Snadno nahlédneme, že relace $\leftrightarrow$ je ekvivalence (cvičení 1). Ekvivalenční třídy indukují podgrafy, kterým se říká *komponenty silné souvislosti* (v tomto oddílu říkejme prostě *komponenty*). Graf je tedy silně souvislý, pokud má právě jednu komponentu, čili pokud $u \leftrightarrow v$ pro každé dva vrcholy u a v. Vzájemné vztahy komponent můžeme popsat opět grafem:

Definice: *Graf komponent* $\mathcal{C}(G)$ má za vrcholy komponenty grafu G, z komponenty C_i vede hrana do C_j právě tehdy, když v původním grafu G existuje hrana z nějakého vrcholu $u \in C_i$ do nějakého $v \in C_j$.

Na graf $\mathcal{C}(G)$ se můžeme dívat i tak, že vznikl z G kontrakcí každé komponenty do jednoho vrcholu a odstraněním násobných hran. Proto se mu také někdy říká *kondenzace*.



Obrázek 5.13: Orientovaný graf a jeho graf komponent

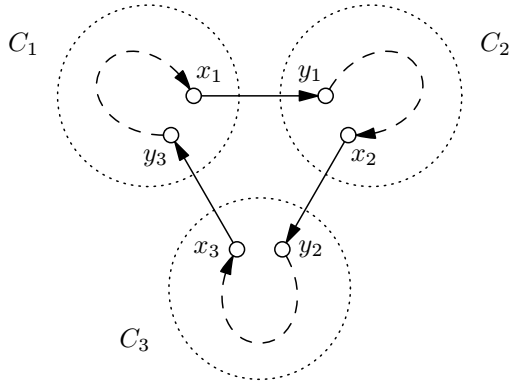
Lemma: Graf komponent $\mathcal{C}(G)$ každého grafu G je acyklický.

Důkaz: Sporem. Nechť $C_1, C_2, \ldots C_k$ tvoří cyklus v $\mathcal{C}(G)$. Podle definice grafu komponent musí existovat vrcholy $x_1, \ldots, x_k$ ($x_i \in C_i$) a $y_1, \ldots, y_k$ ($y_i \in C_{i+1}$, přičemž indexujeme cyklicky) takové, že $x_i y_i$ jsou hranami grafu G. Situaci sledujme na obrázku 5.14.

Jelikož každá komponenta C_i je silně souvislá, existuje cesta z y_{i-1} do x_i v C_i. Slepením těchto cest s hranami $x_i y_i$ vznikne cyklus v grafu G tvaru

$$x_1, y_1, \text{cesta v } C_2, x_2, y_2, \text{cesta v } C_3, x_3, \ldots, x_k, y_k, \text{cesta v } C_1, x_1.$$

To je ovšem spor s tím, že vrcholy x_i leží v různých komponentách. □



Obrázek 5.14: K důkazu lemmatu o acykličnosti $\mathcal{C}$(G)

Podle toho, co jsme o acyklických grafech zjistili v minulém oddílu, musí v $\mathcal{C}(G)$ existovat alespoň jeden zdroj (vrchol bez předchůdců) a stok (vrchol bez následníků). Proto vždy existují komponenty s následujícími vlastnostmi:

Definice: Komponenta je *zdrojová,* pokud do ní nevede žádná hrana, a *stoková,* pokud nevede žádná hrana z ní.

Představme si nyní, že jsme našli nějaký vrchol ležící ve stokové komponentě. Spustíme-li z tohoto vrcholu DFS, navštíví právě celou tuto komponentu (ven se dostat nemůže, hrany vedou v protisměru).

Jak ale vrchol ze stokové komponenty najít? Se zdrojovou by to bylo snazší: prohledáme-li graf do hloubky (opakovaně, nedostalo-li se na všechny vrcholy), vrchol s maximálním $out(v)$ musí ležet ve zdrojové komponentě (rozmyslete si, proč). Pomůžeme si proto následujícím trikem:

Pozorování: Nechť G^{T} je graf, který vznikne z G otočením orientace všech hran. Potom G^{T} má tytéž komponenty jako G a platí $\mathcal{C}(G^{\mathrm{T}}) = (\mathcal{C}(G))^{\mathrm{T}}$. Proto se prohodily zdrojové komponenty se stokovými.

Nabízí se tedy spustit DFS na graf G^{T}, vybrat v něm vrchol s maximálním *out*em a spustit z něj DFS v grafu G. Tím najdeme jednu stokovou komponentu. Tu můžeme odstranit a postup opakovat.

Je ale zbytečné stokovou komponentu hledat pokaždé znovu. Ukážeme, že postačí procházet vrcholy v pořadí klesajících *out*ů v G^{T}. Ty vrcholy, které jsme do nějaké komponenty

zařadili, budeme přeskakovat, z ostatních vrcholů budeme spouštět DFS v G a objevovat nové komponenty.

Procházení podle klesajících *out*ů už známe – to je osvědčený algoritmus pro topologické uspořádání. Zde ho ovšem používáme na graf, který není DAG. Výsledkem samozřejmě nemůže být topologické uspořádání, protože to pro grafy s cykly neexistuje. Přesto ale výstup algoritmu dává jistý smysl. Následující tvrzení zaručuje, že komponenty grafu budeme navštěvovat v opačném topologickém pořadí:

Lemma: Pokud v $\mathcal{C}(G)$ vede hrana z komponenty C_1 do C_2, pak

$$\max_{x \in C_1} out(x) > \max_{y \in C_2} out(y).$$

Důkaz: Nejprve rozeberme případ, kdy DFS vstoupí do C_1 dříve než do C_2. Začne tedy zkoumat C_1, během toho objeví hranu do C_2, po té projde, načež zpracuje celou komponentu C_2, než se opět vrátí do C_1. (Víme totiž, že z C_2 do C_1 nevede žádná orientovaná cesta, takže DFS může zpět přejít pouze návratem z rekurze.) V tomto případě tvrzení platí.

Nebo naopak vstoupí nejdříve do C_2. Odtamtud nemůže dojít po hranách do C_1, takže se nejprve vrátí z celé C_2, než do C_1 poprvé vstoupí. I tehdy tvrzení lemmatu platí. □

Nyní je vše připraveno a můžeme algoritmus zapsat:

Algoritmus KompSilnéSouvislosti
Vstup: Orientovaný graf G

1. Sestrojíme graf G^{T} s obrácenými hranami.
2. $Z \leftarrow$ *prázdný zásobník*
3. Pro všechny vrcholy v nastavíme $komp(v) \leftarrow$ *nedefinováno*.
4. Spouštíme DFS v G^{T} opakovaně, než prozkoumáme všechny vrcholy. Kdykoliv přitom opouštíme vrchol, vložíme ho do Z. Vrcholy v zásobníku jsou tedy setříděné podle $out(v)$.
5. Postupně odebíráme vrcholy ze zásobníku Z a pro každý vrchol v:
6. Pokud $komp(v) =$ *nedefinováno*:
7. Spustíme DFS(v) v G, přičemž vstupujeme pouze do vrcholů s nedefinovanou hodnotou $komp(\ldots)$ a tuto hodnotu přepisujeme na v.

Výstup: Pro každý vrchol v vrátíme identifikátor komponenty $komp(v)$

Věta: Algoritmus KompSilnéSouvislosti rozloží zadaný graf na komponenty silné souvislosti v čase $\Theta(n+m)$ a prostoru $\Theta(n+m)$.

Důkaz: Korektnost algoritmu vyplývá z toho, jak jsme jej odvodili. Všechna volání DFS dohromady navštíví každý vrchol a hranu právě dvakrát, práce se zásobníkem trvá také lineárně dlouho. Paměť kromě reprezentace grafu potřebujeme na pomocná pole a zásobníky (Z a zásobník rekurze), což je celkem lineárně velké. □

Dodejme ještě, že tento algoritmus objevil v roce 1978 Sambasiva Rao Kosaraju a nezávisle na něm v roce 1981 Micha Sharir.

Cvičení

1. Dokažte, že relace $\leftrightarrow$ z tohoto oddílu je opravdu ekvivalence.

2. Opakovanému spouštění DFS, dokud není celý graf prohledán, se dá i zde vyhnout přidáním „superzdroje“ jako v cvičení 5.8.1. Co se přitom stane s grafem komponent?

3. V orientovaném grafu jsou některé vrcholy obarvené zeleně. Jak zjistit, jestli existuje cyklus obsahující alespoň jeden zelený vrchol?

4.* O orientovaném grafu řekneme, že je *polosouvislý,* pokud mezi každými dvěma vrcholy vede orientovaná cesta alespoň jedním směrem. Navrhněte lineární algoritmus, který polosouvislost grafu rozhoduje.

5.10* Silná souvislost podruhé: Tarjanův algoritmus

Předvedeme ještě jeden lineární algoritmus na hledání komponent silné souvislosti. Je založen na několika hlubokých pozorováních o vztahu komponent s DFS stromem, jejichž odvození je pracnější. Samotný algoritmus je pak jednodušší a nepotřebuje konstrukci obráceného grafu. Objevil ho Robert Endre Tarjan v roce 1972.

Stejně jako v minulém oddílu budeme používat relaci $\leftrightarrow$ a komponentám silné souvislosti budeme říkat prostě komponenty.

Lemma: Každá komponenta indukuje v DFS stromu slabě souvislý podgraf.

Důkaz: Nechť x a y jsou vrcholy ležící v téže komponentě C. Rozebereme jejich možné polohy v DFS stromu a pokaždé ukážeme, že (neorientovaná) cesta P spojující ve stromu x s y leží celá uvnitř C.

Nejprve uvažme případ, kdy je x „nad“ y, čili z x do y lze dojít po směru stromových hran. Nechť t je libovolný vrchol cesty P. Jistě jde dojít z x do t – stačí následovat cestu P.

Ale také z t do x – můžeme dojít po cestě P do y, odkud se už do x dostaneme (x a y jsou přeci oba v C). Takže vrchol t musí také ležet v C.

Pokud y je nad x, postupujeme symetricky.

Zbývá případ, kdy x a y mají nějakého společného předka $p \neq x, y$. Kdyby tento předek ležel v C, máme vyhráno: p se totiž nachází nad x i nad y, takže podle předchozího argumentu leží v C i všechny ostatní vrcholy cesty P.

Pojďme dokázat, že p se nemůže nacházet mimo C. Pozastavíme DFS v okamžiku, kdy už se vrátilo z jednoho z vrcholů x a y (bez újmy na obecnosti x), stojí ve vrcholu p a právě se chystá odejít stromovou hranou směrem k y. Použijeme následující:

> **Pozorování:** Kdykoliv v průběhu DFS vedou z uzavřených vrcholů hrany pouze do uzavřených a otevřených.
>
> *Důkaz:* Přímo z klasifikace hran. □

Víme, že z x vede orientovaná cesta do y. Přitom x je už uzavřený a y dosud nenalezený. Podle pozorování tato cesta musí projít přes nějaký otevřený vrchol. Ten se ve stromu nutně nachází nad p (neostře), takže přes něj jde z x dojít do p. Ovšem z p lze dojít po stromových hranách do x, takže x a p leží v téže komponentě.

Je za tím jasná intuice: stromová cesta z kořene do p, na níž leží všechny otevřené vrcholy, tvoří přirozenou hranici mezi už uzavřenou částí grafu a dosud neprozkoumaným zbytkem. Cesta z x do y musí tuto hranici někde překročit. □

Víme tedy, že komponenty jsou v DFS stromu souvislé. Stačí umět poznat, které stromové hrany leží na rozhraní komponent. K tomu se hodí „chytit“ každou komponentu za její nejvyšší vrchol:

Definice: *Kořenem komponenty* nazveme vrchol, v němž do ní DFS poprvé vstoupilo. Tedy ten, jehož *in* je nejmenší.

Pokud odstraníme hrany, za které „visí“ kořeny komponent, DFS strom se rozpadne na jednotlivé komponenty. Ukážeme, jak v okamžiku opouštění libovolného vrcholu v poznat, zda je v kořenem své komponenty. Označíme T_v podstrom DFS stromu obsahující v a všechny jeho potomky.

Lemma: Pokud z T_v vede zpětná hrana ven, není v kořenem komponenty.

Důkaz: Zpětná hrana vede z T_v do nějakého vrcholu p, který leží nad v a má menší *in* než v. Přitom z v se jde dostat do p přes zpětnou hranu a zároveň z p do v po stromové cestě, takže p i v leží v téže komponentě. □

S příčnými hranami je to složitější, protože mohou vést i do jiných komponent. Zařídíme tedy, aby v okamžiku, kdy opouštíme kořen komponenty, byly již ke komponentě přiřazeny všechny její vrcholy. Pak můžeme použít:

Lemma: Pokud z T_v vede příčná hrana ven do dosud neopuštěné komponenty, pak v není kořenem komponenty.

Důkaz: Vrchol w, který je cílem příčné hrany, má nižší *in* než v a už byl uzavřen. Jeho komponenta ale dosud nebyla opuštěna, takže jejím kořenem musí být některý z otevřených vrcholů. Vrchol v je s touto komponentou obousměrně propojen, tedy v ní také leží, ovšem níže než kořen. □

Lemma: Pokud nenastane situace podle předchozích dvou lemmat, v je kořenem komponenty.

Důkaz: Kdyby nebyl kořenem, musel by skutečný kořen ležet někde nad v (komponenta je přeci ve stromu souvislá). Z v by do tohoto kořene musela vést cesta, která by někudy musela opustit podstrom T_v. To ale lze pouze zpětnou nebo příčnou hranou. □

Nyní máme vše připraveno k formulaci algoritmu. Graf budeme procházet do hloubky. Vrcholy, které jsme dosud nezařadili do žádné komponenty, budeme ukládat do pomocného zásobníku. Kdykoliv při návratu z vrcholu zjistíme, že je kořenem komponenty, odstraníme ze zásobníku všechny vrcholy, které leží v DFS stromu pod tímto kořenem, a zařadíme je do nové komponenty.

Pro rozhodování, zda z T_v vede zpětná nebo příčná hrana, budeme používat hodnoty $esc(v)$, které budou fungovat podobně jako $low(v)$ v algoritmu na hledání mostů.

Definice: $esc(v)$ udává minimum z *in*ů vrcholů, do nichž z podstromu T_v vede buď zpětná hrana, nebo příčná hrana do ještě neuzavřené komponenty.

Následuje zápis algoritmu. Pro zjednodušení implementace si vystačíme s polem *in* a neukládáme explicitně ani *out*, ani stav vrcholů. Při aktualizaci pole *esc* nebudeme rozlišovat mezi zpětnými, příčnými a dopřednými hranami – uvědomte si, že to nevadí.

Algoritmus KompSSTarjan
Vstup: Orientovaný graf G

1. Pro všechny vrcholy v nastavíme:
2. $\quad in(v) \leftarrow$ *nedefinováno*
3. $\quad komp(v) \leftarrow$ *nedefinováno*
4. $T \leftarrow 0$
5. $Z \leftarrow$ *prázdný zásobník*
6. Pro všechny vrcholy u:

7. Pokud $in(u)$ není definovaný:
8. Zavoláme KSST(u).

Výstup: Pro každý vrchol v vrátíme identifikátor komponenty $komp(v)$

Procedura KSST(v)

1. $T \leftarrow T + 1$, $in(v) \leftarrow T$
2. Do zásobníku Z přidáme vrchol v.
3. $esc(v) \leftarrow +\infty$
4. Pro všechny následníky w vrcholu v:
5. Pokud $in(w)$ není definován: ◁ *hrana vw je stromová*
6. Zavoláme KSST(w).
7. $esc(v) \leftarrow \min(esc(v), esc(w))$
8. Jinak: ◁ *zpětná, příčná nebo dopředná hrana*
9. Není-li $komp(w)$ definovaná:
10. $esc(v) \leftarrow \min(esc(v), in(w))$
11. Je-li $esc(v) \geq in(v)$: ◁ *v je kořen komponenty*
12. Opakujeme:
13. Odebereme vrchol ze zásobníku Z a označíme ho t.
14. $komp(t) \leftarrow v$
15. Cyklus ukončíme, pokud $t = v$.

Věta: Tarjanův algoritmus nalezne komponenty silné souvislosti v čase $\Theta(n+m)$ a prostoru $\Theta(n+m)$.

Důkaz: Správnost algoritmu plyne z jeho odvození. Časová složitost se od DFS liší pouze obsluhou zásobníku Z. Každý vrchol se do Z dostane právě jednou při svém otevření a pak ho jednou vyjmeme, takže celkem prací se zásobníkem strávíme čas $\mathcal{O}(n)$. Jediná paměť, kterou k DFS potřebujeme navíc, je na zásobník Z a pole esc, což je obojí velké $\mathcal{O}(n)$. □

5.11 Další cvičení

Najděte algoritmy pro tyto grafové problémy. Pokaždé existuje řešení pracující v lineárním čase vzhledem k velikosti grafu.

1. *Eulerovský tah:* Mějme souvislý neorientovaný graf. Chceme ho nakreslit jedním tahem, tedy nalézt posloupnost na sebe navazujících hran, která obsahuje každou hranu grafu právě jednou.

2. *Vyvážená orientace:* Hranám grafu chceme přiřadit orientace tak, aby z každého vrcholu vycházelo stejně hran, jako do něj vchází. Ukažte, že to lze, kdykoliv všechny vrcholy mají sudé stupně.

3. *Skoro vyvážená orientace:* Pokračujme v předchozím cvičení. Pokud graf obsahuje vrcholy lichého stupně, najděte takovou orientaci, v níž se vstupní a výstupní stupeň každého vrcholu liší nejvýše o 1.

4. *Šéf agentů:* Podařilo se vám sehnat schéma sítě tajných agentů. Má podobu orientovaného grafu, jehož vrcholy jsou agenti a hrana popisuje, že jeden agent velí druhému. Kdykoliv agent obdrží rozkaz, předá ho všem agentům, kterým velí. Šéfem sítě je libovolný agent, který vydá-li rozkaz, dostanou ho časem všichni ostatní agenti. Vymyslete algoritmus, jež najde šéfa sítě. Umíte najít všechny šéfy?

5. *Asfaltování:* Máme mapu městečka v podobě neorientovaného grafu. Parta asfaltérů umí za jednu směnu vyasfaltovat dvě na sebe navazující ulice. Jak vyasfaltovat každou ulici právě jednou? Jde to pokaždé, když je ulic sudý počet?

6. *Zjednodušení grafu:* Navrhněte algoritmus, který ze zadaného multigrafu odstraní všechny násobné hrany. Mezi každými dvěma vrcholy tedy zbude nejvýše jedna hrana.

7. *Vítěz:* n sportovců sehrálo zápasy každý s každým. Chceme zjistit, zda existuje někdo, kdo vyhrál nad všemi ostatními. Máme-li matici výsledků zápasů již načtenou v paměti, lze to spočítat v čase $\mathcal{O}(n)$.

8. *Slovní žebřík:* Je dán slovník. Sestrojte co nejdelší slovní žebřík, což je posloupnost slov ze slovníku taková, že $(i+1)$-ní slovo získáme z i-tého smazáním jednoho písmene. V češtině například `zuzavírání`, `uzavírání`, `zavírání`, `zvírání`, `zírání`, `zrání`, `zrní`.

9. *Strážníci:* Mapa městečka má tvar stromu. Na křižovatky chceme rozmístit co nejméně strážníků tak, aby každá ulice byla z alespoň jedné strany hlídaná.

10.* *Mafiáni:* Mapa městečka ve tvaru stromu, v některých vrcholech bydlí mafiáni. Chceme do vrcholů rozestavět co nejméně strážníků tak, aby se mafiáni nemohli nepozorovaně domlouvat, tedy aby mezi každými dvěma mafiány stál aspoň jeden strážník. Postavíme-li strážníka do vrcholu s mafiánem, zabráníme mu v kontaktu se všemi ostatními mafiány.

11. *Brtník:* V lese tvaru čtvercové sítě se nachází medvěd, brloh, překážky a několik brtí.[1] Medvěd si právě začal pochutnávat na medu, ale hned si toho všimly včely

[1] *Brť* je úl lesních včel v dutině stromu. Od toho medvěd brtník.

a začaly se na něj slétat ze všech brtí najednou. Chceme spočítat, jak dlouho ještě může medvěd mlsat, aby ho na cestě do brlohu nezastihly včely. V čase 0 je medvěd na zadaném místě a včely v brtích. Za každou další jednotku času se medvěd posune o jedno políčko a včely se také rozšíří o jedno políčko.

12. *Silně souvislá orientace:* V každém neorientovaném grafu bez mostů je možné hrany zorientovat tak, aby vznikl silně souvislý orientovaný graf. Vymyslete algoritmus, který takovou orientaci najde.

13.* *Jednosměrky:* Je dána mapa městečka v podobně neorientovaného grafu. Chceme z co nejvíce ulic udělat jednosměrky, ale stále musí být možné dojet autem odkudkoliv kamkoliv bez porušení předpisů.

14.* *Odolná síť:* Počítačovou síť popíšeme grafem: vrcholy odpovídají routerům, hrany kabelům mezi nimi. Přirozeně se nám nelíbí mosty: to jsou kabely, jejichž výpadek způsobí nedostupnost některých routerů. Navrhněte, jak do sítě přidat co nejméně kabelů, aby v ní žádné mosty nezbyly.

15. *Barvení 6 barvami:* Mějme *rovinný graf,* tedy takový, co se dá nakreslit do roviny bez křížení hran. Přiřaďte vrcholům čísla z množiny $\{1, \ldots, 6\}$ tak, aby žádné dva vrcholy spojené hranou nedostaly stejné číslo. Prozradíme ještě, že každý rovinný graf obsahuje vrchol stupně nejvýše 5.

16.* *Barvení 5 barvami:* Vyřešte předchozí cvičení pro 5 barev. To není v lineárním čase snadné, tak to zkuste v kvadratickém nebo lepším. (K obarvení každého rovinného grafu dokonce postačí 4 barvy, ale to je mnohem těžší a slavnější problém.)

17.* *Trojúhelníky:* Spočítejte v rovinném grafu trojúhelníky, tedy trojice vrcholů spojené každý s každým.

18.* *Zeměměřiči:* Jistý pozemek ve tvaru (ne nutně konvexního) n-úhelníku byl zeměměřiči kompletně triangulován (rozdělen na trojúhelníky nekřížícími se úhlopříčkami) a byly pořízeny záznamy o každé triangulační i obvodové úsečce: pro každou víme, která dva vrcholy n-úhelníka spojuje. Bohužel, některé záznamy o triangulačních úsečkách se ztratily, údaje o obvodových úsečkách zůstaly všechny, nicméně se všechny promíchaly dohromady. Na vstupu je n a seznam dvojic vrcholů, které byly propojeny úsečkou dle podmínek výše. Zjistěte, které úsečky jsou obvodové a které vnitřní.

6 Nejkratší cesty

6 Nejkratší cesty

Často potřebujeme hledat mezi dvěma vrcholy grafu cestu, která je v nějakém smyslu optimální – typicky nejkratší možná. Už víme, že prohledávání do šířky najde cestu s nejmenším počtem hran. Málokdy jsou ale všechny hrany rovnocenné: silnice mezi městy mají různé délky, po různých vedeních lze přenášet elektřinu s různými ztrátami a podobně.

V této kapitole proto zavedeme grafy s ohodnocenými hranami a odvodíme několik algoritmů pro hledání cesty s nejmenším součtem ohodnocení hran.

6.1 Ohodnocené grafy a vzdálenost

Mějme graf $G = (V, E)$. Pro větší obecnost budeme předpokládat, že je orientovaný. Algoritmy se tím nezkomplikují a neorientované grafy můžeme vždy převést na orientované zdvojením hran.

Každé hraně $e \in E$ přiřadíme její *ohodnocení* neboli *délku,* což bude nějaké reálné číslo $\ell(e)$. Tím vznikne *ohodnocený orientovaný graf* nebo též *síť.* Zatím budeme uvažovat pouze nezáporná ohodnocení, později se zamyslíme nad tím, co by způsobila záporná.

Délku můžeme přirozeně rozšířit i na cesty, či obecněji sledy. Délka $\ell(S)$ sledu S bude prostě součet ohodnocení jeho hran.

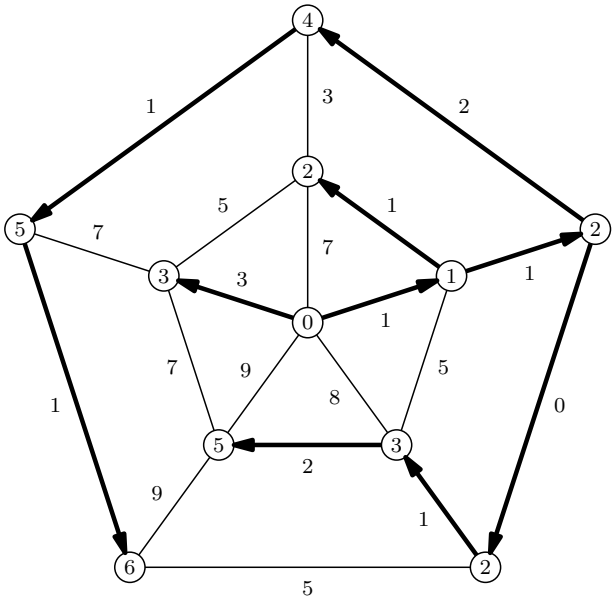
Pro libovolné dva vrcholy u, v definujeme jejich *vzdálenost* $d(u, v)$ jako minimum z délek všech uv-cest (cest z u do v), případně $+\infty$, pokud žádná uv-cesta neexistuje. Toto minimum je vždy dobře definované, protože cest v grafu existuje pouze konečně mnoho. Pozor na to, že v orientovaném grafu se $d(u, v)$ a $d(v, u)$ mohou lišit.

Libovolné uv-cestě, jejíž délka je rovná vzdálenosti $d(u, v)$ budeme říkat *nejkratší cesta.* Nejkratších cest může být obecně více.

Vzdálenost bychom také mohli zavést pomocí nejkratšího sledu. Podobně jako u dosažitelnosti by se nic podstatného nezměnilo, protože sled lze zjednodušit na cestu:

Lemma (o zjednodušování sledů): Pro každý uv-sled existuje uv-cesta stejné nebo menší délky.

Důkaz: Pokud sled není cestou, znamená to, že se v něm opakují vrcholy. Uvažme tedy nějaký vrchol, který se zopakoval. Část sledu mezi prvním a posledním výskytem tohoto vrcholu můžeme „vystřihnout“, čímž získáme uv-sled stejné nebo menší délky. Jelikož ubyla alespoň jedna hrana, opakováním tohoto postupu časem získáme cestu. □



Obrázek 6.1: Vzdálenosti od středového vrcholu (čísla uvnitř vrcholů) v ohodnoceném neorientovaném grafu. Zvýrazněné hrany tvoří strom nejkratších cest.

Důsledek: Nejkratší sled existuje a má stejnou délku jako nejkratší cesta.

Důkaz: Nejkratší cesta je jedním ze sledů. Pokud by tvrzení neplatilo, musel by existovat nějaký kratší sled. Ten by ovšem šlo zjednodušit na ještě kratší cestu. □

Důsledek: Pro vzdálenosti platí *trojúhelníková nerovnost:*

$$d(u,v) \leq d(u,w) + d(w,v).$$

Důkaz: Pokud je $d(u,w)$ nebo $d(w,v)$ nekonečná, nerovnost triviálně platí. V opačném případě uvažme spojení nejkratší uw-cesty s nejkratší wv-cestou. To je nějaký uv-sled a ten nemůže být kratší než nejkratší uv-cesta. □

Naše grafová vzdálenost se tedy chová tak, jak jsme u vzdáleností zvyklí.

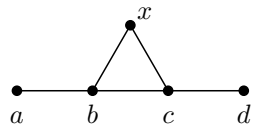
Při prohledávání grafů do šířky se osvědčilo popisovat nejkratší cesty z daného vrcholu v_0 do všech ostatních vrcholů pomocí stromu. Tento přístup funguje i v ohodnocených grafech a nyní ho zavedeme pořádně.

Definice: *Strom nejkratších cest* z vrcholu v_0 je podgraf zkoumaného grafu, který obsahuje všechny vrcholy dosažitelné z v_0. Má tvar stromu orientovaného z vrcholu v_0. Pro libovolný dosažitelný vrchol w platí, že cesta z v_0 do w ve stromu je jednou z nejkratších cest z v_0 do w ve zkoumaném grafu.

Stejně jako nejsou jednoznačně určeny nejkratší cesty, i stromů nejkratších cest může existovat více. Vždy ale existuje alespoň jeden (důkaz ponecháme jako cvičení 6).

Záporné hrany

Ještě si krátce rozmysleme, co by se stalo, kdybychom povolili hrany záporné délky. V následujícím grafu nastavíme všem hranám délku -1. Nejkratší cesta z a do d vede přes b, x a c a má délku -4. Sled $abxcbxcd$ je ovšem o 3 kratší, protože navíc obešel záporný cyklus $bxcb$. Pokud bychom záporný cyklus obkroužili vícekrát, získávali bychom kratší a kratší sledy. Tedy nejen že nejkratší sled neodpovídá nejkratší cestě, on ani neexistuje.



Podobně neplatí trojúhelníková nerovnost: $d(a,d) = -4$, ale $d(a,x) + d(x,d) = (-3) + (-3) = -6$.

Kdyby ovšem v grafu neexistoval záporný cyklus, sama existence záporných hran by problémy nepůsobila. Snadno ověříme, že lemma o zjednodušování sledů by nadále platilo, a tím pádem i trojúhelníková nerovnost. Grafy se zápornými hranami bez záporných cyklů jsou navíc užitečné (jak uvidíme ve cvičeních). Proto je budeme v některých částech této kapitoly připouštět, ale pokaždé na to výslovně upozorníme.

Cvičení

1. V neorientovaném grafu může roli záporného cyklu hrát dokonce i jediná hrana se záporným ohodnocením. Po ní totiž můžeme chodit střídavě tam a zpět. Nalezněte příklad takového grafu.

2. Ukažte, jak pro libovolné n sestrojit graf na nejvýše n vrcholech, v němž mezi nějakými dvěma vrcholy existuje $2^{\Omega(n)}$ nejkratších cest.

3. Připomeňte si definici metrického prostoru v matematické analýze. Kdy tvoří množina všech vrcholů spolu s funkcí $d(u,v)$ metrický prostor?

4. Navrhněte algoritmus pro výpočet vzdálenosti $d(u,v)$, který bude postupně počítat čísla $d_i(u,v)$ – nejmenší délka uv-sledu o nejvýše i hranách. Jaké časové složitosti jste dosáhli? Porovnejte ho s ostatními algoritmy z této kapitoly.

5. Dokažte, že pro nejkratší cesty platí takzvaná *prefixová vlastnost:* Nechť P je nějaká nejkratší cesta z v_0 do w a t nějaký vrchol na této cestě. Poté úsek (prefix) cesty P z v_0 do t je jednou z nejkratších cest z v_0 do t.

6. Dokažte, že pro libovolný ohodnocený graf a počáteční vrchol v_0 existuje strom nejkratších cest z v_0. Může se hodit předchozí cvičení.

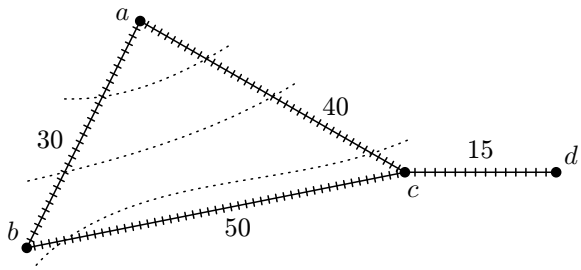
6.2 Dijkstrův algoritmus

Dnes asi nejpoužívanější algoritmus pro hledání nejkratších cest vymyslel v roce 1959 Edsger Dijkstra.[1] Funguje efektivně, kdykoliv jsou všechny hrany ohodnocené nezápornými čísly. Dovede nás k němu následující myšlenkový pokus.

Mějme orientovaný graf, jehož hrany jsou ohodnocené celými kladnými čísly. Každou hranu „podrozdělíme“ – nahradíme ji tolika jednotkovými hranami, kolik činila její délka. Tím vznikne neohodnocený graf, ve kterém můžeme nejkratší cestu nalézt prohledáváním do šířky. Tento algoritmus je funkční, ale sotva efektivní. Označíme-li L maximální délku hrany, podrozdělením vznikne $\mathcal{O}(Lm)$ nových vrcholů a hran.

Sledujme na obrázku 6.2, jak výpočet probíhá. Začneme ve vrcholu a. Vlna prvních 30 kroků prochází vnitřkem hran ab a ac. Pak dorazí do vrcholu b, načež se šíří dál zbytkem hrany ac a novou hranou bc. Za dalších 10 kroků dorazí do c hranou ac, takže pokračuje hranou cd a nyní již zbytečnou hranou bc.

Většinu času tedy trávíme dlouhými úseky výpočtu, uvnitř kterých se prokousáváme stále stejnými podrozdělenými hranami. Co kdybychom každý takový úsek zpracovali najednou?



Obrázek 6.2: Podrozdělený graf a poloha vlny v časech 10, 20 a 35

[1] Je to holandské jméno, takže ho čteme „dajkstra“.

Pro každý původní vrchol si pořídíme *„budík“*. Jakmile k vrcholu zamíří vlna, nastavíme jeho budík na čas, kdy do něj vlna má dorazit (pokud míří po více hranách najednou, zajímá nás, kdy dorazí poprvé).

Místo toho, abychom krok po kroku simulovali průchod vlny hranami, můžeme zjistit, kdy poprvé zazvoní budík. Tím přeskočíme nudnou část výpočtu a hned se dozvíme, že se vlna zarazila o vrchol. Podíváme se, jaké hrany z tohoto vrcholu vedou, a spočítáme si, kdy po nich vlna dorazí do dalších vrcholů. Podle toho případně přenastavíme další budíky. Opět počkáme, až zazvoní nějaký budík, a pokračujeme stejně.

Ještě by se mohlo stát, že vlna vstoupí do podrozdělené hrany z obou konců a obě části se „srazí“ uvnitř hrany. Tehdy můžeme nechat budíky zazvonit, jako by ke srážce nedošlo, a části zastavit až na konci hrany.

Zkusme tuto myšlenku zapsat v pseudokódu. Pro každý vrchol v si budeme pamatovat jeho *ohodnocení* $h(v)$, což bude buďto čas nastavený na příslušném budíku, nebo $+\infty$, pokud budík neběží. Podobně jako při prohledávání grafů do šířky budeme rozlišovat tři druhy vrcholů: *nenalezené, otevřené* (to jsou ty, které mají nastavené budíky) a *uzavřené* (jejich budíky už zazvonily). Také si budeme pamatovat předchůdce vrcholů $P(v)$, abychom uměli rekonstruovat nejkratší cesty.

Algoritmus Dijkstra (nejkratší cesty v ohodnoceném grafu)
Vstup: Graf G a počáteční vrchol v_0

1. Pro všechny vrcholy v:
2. $\quad$ *stav*$(v) \leftarrow$ *nenalezený*
3. $\quad$ $h(v) \leftarrow +\infty$
4. $\quad$ $P(v) \leftarrow$ *nedefinováno*
5. *stav*$(v_0) \leftarrow$ *otevřený*
6. $h(v_0) \leftarrow 0$
7. Dokud existují nějaké otevřené vrcholy:
8. $\quad$ Vybereme otevřený vrchol v, jehož $h(v)$ je nejmenší.
9. $\quad$ Pro všechny následníky w vrcholu v:
10. $\quad\quad$ Pokud $h(w) > h(v) + \ell(v,w)$:
11. $\quad\quad\quad$ $h(w) \leftarrow h(v) + \ell(v,w)$
12. $\quad\quad\quad$ *stav*$(w) \leftarrow$ *otevřený*
13. $\quad\quad\quad$ $P(w) \leftarrow v$
14. $\quad$ *stav*$(v) \leftarrow$ *uzavřený*

Výstup: Pole vzdáleností h, pole předchůdců P

Z úvah o podrozdělování hran plyne, že Dijkstrův algoritmus dává správné výsledky, kdykoliv jsou délky hran celé kladné. Důkaz správnosti pro obecné délky najdete v následujícím oddílu, teď se zaměříme především na časovou složitost.

Věta: Dijkstrův algoritmus spočte v grafu s nezáporně ohodnocenými hranami vzdálenosti od vrcholu v_0 v čase $\mathcal{O}(n^2)$.

Důkaz: Inicializace trvá $\mathcal{O}(n)$. Každý vrchol uzavřeme nejvýše jednou (to jsme zatím nahlédli z analogie s BFS, více viz příští oddíl), takže vnějším cyklem projdeme nejvýše n-krát. Pokaždé hledáme minimum z n ohodnocení vrcholů a procházíme až n následníků. □

Pokud bychom kromě vzdáleností chtěli vypsat i nejkratší cesty, můžeme je získat ze zapamatovaných předchůdců. Chceme-li získat nejkratší cestu z vrcholu v_0 do nějakého w, stačí se podívat na $P(w)$, $P(P(w))$ a tak dále až do v_0. Tak projdeme pozpátku nějakou cestu z v_0 do w, která má délku $h(w)$, a tedy je nejkratší. Dokonce platí, že předchůdci kódují hrany stromu nejkratších cest: kdykoliv $P(v) = u$, leží hrana uv ve stromu nejkratších cest. Důkaz těchto tvrzení ponecháváme jako cvičení 6.3.7.

Dijkstrův algoritmus s haldou

Složitost $\mathcal{O}(n^2)$ je příznivá, pokud máme co do činění s hustým grafem. V grafech s malým počtem hran je náš odhad zbytečně hrubý: v cyklu přes následníky trávíme čas lineární se stupněm vrcholu, za celou dobu běhu algoritmu tedy pouze $\mathcal{O}(m)$. Stále nás ale brzdí hledání minima v kroku 8.

Proto si pořídíme nějakou datovou strukturu, která umí rychle vyhledávat minimum. Nabízí se například binární halda z oddílu 4.2. Uložíme do ní všechny otevřené vrcholy, jako klíče budou sloužit ohodnocení $h(v)$. V každé iteraci Dijkstrova algoritmu nalezneme minimum pomocí operace ExtractMin. Když v kroku 11 měníme ohodnocení vrcholu, buďto tento vrchol nově otevíráme (takže provedeme Insert do haldy), nebo již je otevřený (což znamená Decrease prvku, který již v haldě je). Nesmíme zapomenout, že Decrease neumí prvek vyhledat, takže si u každého vrcholu budeme průběžně udržovat jeho pozici v haldě.

Věta: Dijkstrův algoritmus s haldou běží v čase $\mathcal{O}(n \cdot T_i + n \cdot T_x + m \cdot T_d)$, kde T_i, T_x a T_d jsou složitosti operací Insert, ExtractMin a Decrease.

Důkaz: Operaci Insert provádíme při otevírání vrcholu, ExtractMin při jeho zavírání. Obojí pro každý vrchol nastane nejvýše jednou. Decrease voláme, když při zavírání vrcholu v snižujeme hodnotu jeho souseda w. To se pro každou hranu vw stane za celou dobu běhu algoritmu nejvýše jednou. Zbývající operace algoritmu trvají $\mathcal{O}(n + m)$. □

Důsledek: Dijkstrův algoritmus s binární haldou běží v čase $\mathcal{O}((n + m) \cdot \log n)$.

Důkaz: V haldě není nikdy více než n prvků, takže všechny tři operace pracují v čase $\mathcal{O}(\log n)$. □

I tento čas lze ještě vylepšit. Některé z možných přístupů prozkoumáme v cvičeních 3 a 4. V kapitole 18 posléze vybudujeme Fibonacciho haldu, která umí DECREASE v konstantním čase, aniž by se asymptoticky zpomalily ostatní operace.

Důsledek: Dijkstrův algoritmus s Fibonacciho haldou běží v čase $\mathcal{O}(m + n \log n)$.

Cvičení

1. Odsimulujte chod Dijkstrova algoritmu na grafu z obrázku 6.1.

2. Uvažujte „spojité BFS“, které bude plynule procházet vnitřkem hran. Pokuste se o formální definici takového algoritmu. Nahlédněte, že diskrétní simulací tohoto spojitého procesu (která se bude zastavovat ve významných událostech, totiž ve vrcholech) získáme Dijkstrův algoritmus. Podobnou myšlenku potkáme v kapitole 16 o geometrických algoritmech.

3.* Dijkstrův algoritmus s haldou provede m operací DECREASE, ale pouze n operací INSERT a EXTRACTMIN. Hodila by se tedy halda, která má DECREASE rychlejší, byť za cenu zpomalení ostatních operací. Tuto vlastnost mají například d-regulární haldy z cvičení 4.2.4. Uvažte, jakou hodnotu d zvolit, aby se minimalizovala složitost celého algoritmu.

4. Nechť délky hran leží v množině $\{0, \ldots, L\}$. Navrhněte datovou strukturu založenou na přihrádkách, s níž Dijkstrův algoritmus poběží v čase $\mathcal{O}(nL + m)$. Pokuste se vystačit s pamětí $\mathcal{O}(n + m + L)$.

5. Na mapě města jsme přiřadili každé silnici čas na průjezd a každé křižovatce průměrnou dobu čekání na semaforech. Jak hledat nejrychlejší cestu?

6.3 Relaxační algoritmy

Na Dijkstrův algoritmus se můžeme dívat trochu obecněji. Získáme tak nejen důkaz jeho správnosti pro neceločíselné délky hran, ale též několik dalších zajímavých algoritmů. Ty budou fungovat i pro grafy se zápornými hranami (stále bez záporných cyklů), které v tomto oddílu dovolíme.

Esencí Dijkstrova algoritmu je, že přiřazuje vrcholům nějaká *ohodnocení* $h(v)$. To jsou nějaká reálná čísla, která popisují, jakým nejkratším sledem se zatím umíme do vrcholu dostat. Tato čísla postupně upravujeme, až se z nich stanou skutečné vzdálenosti od v_0. Pokusme se tento princip popsat obecně.

Na počátku výpočtu ohodnotíme vrchol v_0 nulou a všechny ostatní vrcholy nekonečnem. Pak opakujeme následující postup: vybereme nějaký vrchol v s konečným ohodnocením a pro všechny jeho následníky w otestujeme, zda se do nich přes v nedovedeme dostat lépe, než jsme zatím dovedli. Této operaci se obvykle říká *relaxace* a odpovídá krokům 9 až 13 Dijkstrova algoritmu.

Jeden vrchol můžeme obecně relaxovat vícekrát, ale nemá smysl dělat to znovu, pokud se jeho ohodnocení mezitím nezměnilo. To ohlídá stav vrcholu: *otevřené* vrcholy je potřeba znovu relaxovat, *uzavřené* jsou relaxované a zatím to znovu nepotřebují.

Z toho vychází obecný relaxační algoritmus. Ten pokaždé vybere jeden otevřený vrchol, uzavře ho a relaxuje a pokud se tím změní ohodnocení jiných vrcholů, tak je otevře. Na rozdíl od prohledávání do šířky ovšem můžeme jeden vrchol otevřít a uzavřít vícekrát.

Algoritmus Relaxace
Vstup: Graf G a počáteční vrchol v_0

1. Pro všechny vrcholy v:
2. $\quad$ *stav*$(v) \leftarrow$ *nenalezený*, $h(v) \leftarrow +\infty$, $P(v) \leftarrow$ *nedefinováno*
3. *stav*$(v_0) \leftarrow$ *otevřený*, $h(v_0) \leftarrow 0$
4. Dokud existují otevřené vrcholy:
5. $\quad$ $v \leftarrow$ nějaký otevřený vrchol
6. $\quad$ Pro všechny následníky w vrcholu v: $\quad \triangleleft$ *relaxujeme vrchol v*
7. $\quad\quad$ Pokud $h(w) > h(v) + \ell(v, w)$:
8. $\quad\quad\quad$ $h(w) \leftarrow h(v) + \ell(v, w)$
9. $\quad\quad\quad$ *stav*$(w) \leftarrow$ *otevřený*
10. $\quad\quad\quad$ $P(w) \leftarrow v$
11. $\quad$ *stav*$(v) \leftarrow$ *uzavřený*

Výstup: Ohodnocení vrcholů h a pole předchůdců P

Dijkstrův algoritmus je tedy speciálním případem relaxačního algoritmu, v němž vždy vybíráme otevřený vrchol s nejmenším ohodnocením. Prozkoumejme, co o relaxačním algoritmu platí obecně.

Invariant O (ohodnocení): Hodnota $h(v)$ nikdy neroste. Je-li konečná, rovná se délce nějakého sledu z v_0 do v.

Důkaz: Indukcí podle doby běhu algoritmu. Na počátku výpočtu tvrzení určitě platí, protože jediné konečné je $h(v_0) = 0$. Kdykoliv pak algoritmus změní $h(w)$, stane se tak relaxací nějakého vrcholu v, jehož $h(v)$ je konečné. Podle indukčního předpokladu tedy existuje v_0v-sled délky $h(v)$. Jeho rozšířením o hranu vw vznikne v_0w-sled délky $h(v) + \ell(v, w)$, což je přesně hodnota, na níž snižujeme $h(w)$. $\square$

Lemma D (dosažitelnost): Pokud se výpočet zastaví, uzavřené jsou právě vrcholy dosažitelné z v_0.

Důkaz: Dokážeme stejně jako obdobnou vlastnost BFS. Jediné, v čem se situace liší, je, že uzavřený vrchol je možné znovu otevřít. To se ovšem, pokud se výpočet zastavil, stane pouze konečně-krát, takže stačí uvážit situaci při posledním uzavření. □

Lemma V (vzdálenost): Pokud se výpočet zastaví, konečná ohodnocení vrcholů jsou rovna vzdálenostem od v_0.

Důkaz: Inspirujeme se důkazem obdobné vlastnosti BFS. Vrchol v označíme za špatný, pokud $h(v)$ není rovno $d(v_0, v)$. Jelikož $h(v)$ odpovídá délce nějakého v_0v-sledu, musí být $h(v) > d(v_0, v)$. Pro spor předpokládejme, že existují nějaké špatné vrcholy.

Mezi všemi nejkratšími cestami z v_0 do špatných vrcholů vybereme tu s nejmenším počtem hran. Označíme v poslední vrchol této cesty a p její předposlední vrchol (ten jistě existuje, neboť v_0 je dobrý). Vrchol p musí být dobrý, takže ohodnocení $h(p)$ je rovno $d(v_0, p)$. Podívejme se, kdy p získal tuto hodnotu. Tehdy musel p být otevřený. Později byl tudíž zavřen a relaxován, načež muselo platit $h(v) \leq d(v_0, p) + \ell(p, v) = d(v_0, v)$. Jelikož ohodnocení vrcholů nikdy nerostou, došlo ke sporu. □

Rozbor Dijkstrova algoritmu

Nyní se vraťme k Dijkstrovu algoritmu. Ukážeme, že nejsou-li v grafu záporné hrany, průběh výpočtu má následující zajímavé vlastnosti.

Invariant M (monotonie): V každém kroku výpočtu platí:

(1) Kdykoliv je z uzavřený vrchol a o otevřený, platí $h(z) \leq h(o)$.
(2) Ohodnocení uzavřených vrcholů se nemění.

Důkaz: Obě vlastnosti dokážeme dohromady indukcí podle délky výpočtu. Na počátku výpočtu obě triviálně platí, neboť neexistují žádné uzavřené vrcholy.

V každém dalším kroku vybereme otevřený vrchol v s nejmenším $h(v)$. Tehdy musí platit $h(z) \leq h(v) \leq h(o)$ pro libovolný z uzavřený a o otevřený. Nyní vrchol v relaxujeme: pro každou hranu vw se pokusíme snížit $h(w)$ na hodnotu $h(v) + \ell(v, w) \geq h(v)$.

- Pokud w byl uzavřený, nemůže se jeho hodnota změnit, neboť již před relaxací byla menší nebo rovna $h(v)$. Proto platí (2).
- Pokud w byl otevřený, jeho hodnota se sice může snížit, ale nikdy ne pod $h(v)$, takže ani pod $h(z)$ žádného uzavřeného z.

Nerovnost (1) v obou případech zůstává zachována a neporuší se ani přesunem vrcholu v mezi uzavřené. □

Věta: Dijkstrův algoritmus na grafu bez záporných hran uzavírá všechny dosažitelné vrcholy v pořadí podle rostoucí vzdálenosti od počátku (každý právě jednou). V okamžiku uzavření je ohodnocení rovno této vzdálenosti a dále se nezmění.

Důkaz: Především z invariantu **M** víme, že žádný vrchol neotevřeme vícekrát, takže ho ani nemůžeme vícekrát uzavřít. Algoritmus proto skončí a podle lemmat **D** a **V** jsou na konci uzavřeny všechny dosažitelné vrcholy a jejich ohodnocení odpovídají vzdálenostem.

Ohodnocení se přitom od okamžiku uzavření vrcholu nezměnilo (opět viz invariant **M**) a tehdy bylo větší nebo rovno ohodnocením předchozích uzavřených vrcholů. Pořadí uzavírání proto skutečně odpovídá vzdálenostem. □

Vidíme tedy, že naše analogie mezi BFS a Dijkstrovým algoritmem funguje i pro neceločíselné délky hran.

Bellmanův-Fordův algoritmus

Zkusme se ještě zamyslet nad výpočtem vzdáleností v grafech se zápornými hranami. Snadno nahlédneme, že Dijkstrův algoritmus na takových grafech může vrcholy otevírat opakovaně, ba dokonce může běžet exponenciálně dlouho (cvičení 1).

Relaxace se ovšem není potřeba vzdávat, postačí změnit podmínku pro výběr vrcholu: namísto haldy si pořídíme obyčejnou frontu. Jinými slovy, budeme uzavírat nejstarší z otevřených vrcholů. Tomuto algoritmu se podle jeho objevitelů Richarda Ernesta Bellmana a Lestera Forda Jr. říká Bellmanův-Fordův. V následujících odstavcích prozkoumáme, jak efektivní je.

Definice: Definujeme *fáze výpočtu* následovně: ve fázi F_0 otevřeme počáteční vrchol v_0, fáze F_{i+1} uzavírá vrcholy otevřené během fáze F_i.

Invariant F (fáze): Pro vrchol v na konci i-té fáze platí, že jeho ohodnocení je shora omezeno délkou nejkratšího v_0v-sledu o nejvýše i hranách.

Důkaz: Tvrzení dokážeme indukcí podle i. Pro $i = 0$ tvrzení platí – jediný vrchol dosažitelný z v_0 sledem nulové délky je v_0 sám; jeho ohodnocení je nulové, ohodnocení ostatních vrcholů nekonečné.

Nyní provedeme indukční krok. Podívejme se na nějaký vrchol v na konci i-té fáze ($i > 0$). Označme S nejkratší v_0v-sled o nejvýše i hranách.

Pokud sled S obsahuje méně než i hran, požadovaná nerovnost platila už na konci předchozí fáze a jelikož ohodnocení vrcholů nerostou, platí nadále.

Obsahuje-li S právě i hran, označme uv jeho poslední hranu a S' podsled z v_0 do u. Podle indukčního předpokladu je na začátku i-té fáze $h(u) \leq \ell(S')$. Na tuto hodnotu muselo

být $h(u)$ nastaveno nejpozději v $(i-1)$-ní fázi, čímž byl vrchol u otevřen. Nejpozději v i-té fázi proto musel být uzavřen a relaxován. Na začátku relaxace muselo stále platit $h(u) \leq \ell(S')$ – hodnota $h(u)$ se sice mohla změnit, ale ne zvýšit. Po relaxaci tedy muselo platit $h(v) \leq h(u) + \ell(u,v) \leq \ell(S') + \ell(u,v) = \ell(S)$. □

Důsledek: Pokud graf neobsahuje záporné cykly, po n-té fázi se algoritmus zastaví.

Důkaz: Po $(n-1)$-ní fázi jsou všechna ohodnocení shora omezena délkami nejkratších cest, takže se v n-té fázi už nemohou změnit a algoritmus se zastaví. □

Věta: V grafu bez záporných cyklů nalezne Bellmanův-Fordův algoritmus všechny vzdálenosti z vrcholu v_0 v čase $\mathcal{O}(nm)$.

Důkaz: Podle předchozího důsledku se po n fázích algoritmus zastaví a podle lemmat **D** a **V** vydá správný výsledek. Během jedné fáze přitom relaxuje každý vrchol nejvýše jednou, takže celá fáze dohromady trvá $\mathcal{O}(m)$. □

Cvičení

1. Ukažte příklad grafu s celočíselně ohodnocenými hranami, na kterém Dijkstrův algoritmus běží exponenciálně dlouho.

2. Upravte Bellmanův-Fordův algoritmus, aby uměl detekovat záporný cyklus dosažitelný z vrcholu v_0. Uměli byste tento cyklus vypsat?

3. Papeho algoritmus funguje podobně jako Bellmanův-Fordův, pouze místo fronty používá zásobník. Ukažte, že tento algoritmus v nejhorším případě běží exponenciálně dlouho.

4. Uvažujme následující algoritmus: provedeme n fází, v každé z nich postupně relaxujeme všechny vrcholy. Spočte tento algoritmus správné vzdálenosti? Jak si stojí v porovnání s Bellmanovým-Fordovým algoritmem?

5. První algoritmus vymyšlený Bellmanem relaxoval místo vrcholů hrany. Cyklicky procházel všechny hrany a pro každou hranu uv se pokusil snížit $h(v)$ na $h(u) + \ell(u,v)$. Dokažte, že tento algoritmus také spočítá správné vzdálenosti, a rozeberte jeho složitost.

6.* Dokažte, že v grafu bez záporných cyklů se obecný relaxační algoritmus zastaví, ať už vrchol k uzavření vybíráme libovolně.

7. Dokažte, že po zastavení obecného relaxačního algoritmu kódují předchůdci $P(v)$ hrany stromu nejkratších cest.

6.4 Matice vzdáleností a Floydův-Warshallův algoritmus

Někdy potřebujeme zjistit vzdálenosti mezi všemi dvojicemi vrcholů, tedy zkonstruovat *matici vzdáleností*. Pokud v grafu nejsou záporné hrany, mohli bychom spustit Dijkstrův algoritmus postupně ze všech vrcholů. To by trvalo $\mathcal{O}(n^3)$, nebo v implementaci s haldou $\mathcal{O}(n \cdot (n+m) \cdot \log n)$.

V této kapitole ukážeme jiný, daleko jednodušší algoritmus založený na dynamickém programování (tuto techniku později rozvineme v kapitole 12). Pochází z roku 1959 od Roberta Floyda a Stephena Warshalla. Matici vzdáleností spočítá v čase $\Theta(n^3)$, dokonce mu ani nevadí záporné hrany.

Definice: Označíme D^k_{ij} délku nejkratší cesty z vrcholu i do vrcholu j, jejíž vnitřní vrcholy leží v množině $\{1, \ldots, k\}$. Pokud žádná taková cesta neexistuje, položíme $D^k_{ij} = +\infty$.

Pozorování: Hodnoty D^k_{ij} mají následující vlastnosti:

- D^0_{ij} nedovoluje používat žádné vnitřní vrcholy, takže je to buďto délka hrany ij, nebo $+\infty$, pokud taková hrana neexistuje.
- D^n_{ij} už vnitřní vrcholy neomezuje, takže je to vzdálenost z i do j.

Algoritmus dostane na vstupu matici D^0 a postupně bude počítat matice D^1, D^2, …, až se dopočítá k D^n a vydá ji jako výstup.

Nechť tedy známe D^k_{ij} pro nějaké k a všechna i, j. Chceme spočítat D^{k+1}_{ij}, tedy délku nejkratší cesty z i do j přes $\{1, \ldots, k+1\}$. Jak může tato cesta vypadat?

(1) Buďto neobsahuje vrchol $k+1$, v tom případě je stejná, jako nejkratší cesta z i do j přes $\{1, \ldots, k\}$. Tehdy $D^{k+1}_{ij} = D^k_{ij}$.

(2) Anebo vrchol $k+1$ obsahuje. Tehdy se skládá z cesty z i do $k+1$ a cesty z $k+1$ do j. Obě dílčí cesty jdou přes vrcholy $\{1, \ldots, k\}$ a obě musí být nejkratší takové (jinak bychom je mohli vyměnit za kratší). Proto $D^{k+1}_{ij} = D^k_{i,k+1} + D^k_{k+1,j}$.

Za D^{k+1}_{ij} si proto zvolíme minimum z těchto dvou variant.

Musíme ale ošetřit jeden potenciální problém: v případě (2) spojujeme dvě cesty, což ovšem nemusí dát cestu, nýbrž sled: některým vrcholem z $\{1, \ldots, k\}$ bychom mohli projít vícekrát. Stačí si ale uvědomit, že kdykoliv by takový sled byl kratší než cesta z varianty (1), znamenalo by to, že se v grafu nachází záporný cyklus. V grafech bez záporných cyklů proto náš vzorec funguje.

Hotový algoritmus vypadá takto:

Algoritmus FloydWarshall
Vstup: Matice délek hran D^0

1. Pro $k = 0, \ldots, n-1$:
2. Pro $i = 1, \ldots, n$:
3. Pro $j = 1, \ldots, n$:
4. $D_{ij}^{k+1} \leftarrow \min(D_{ij}^k, D_{i,k+1}^k + D_{k+1,j}^k)$

Výstup: Matice vzdáleností D^n

Časová složitost evidentně činí $\Theta(n^3)$, paměťová bohužel také. Nabízí se využít toho, že vždy potřebujeme pouze matice D^k a D^{k+1}, takže by nám místo trojrozměrného pole stačila dvě dvojrozměrná.

Existuje ovšem elegantnější, byť poněkud drzý trik: použijeme jedinou matici a budeme hodnoty přepisovat na místě. Pak ovšem nerozlišíme, zda právě čteme D_{pq}^k, nebo na jeho místě už je zapsáno D_{pq}^{k+1}. Zajímavé je, že na tom nezáleží:

Lemma: Pro všechna i, j, k platí $D_{k+1,j}^{k+1} = D_{k+1,j}^k$ a $D_{i,k+1}^{k+1} = D_{i,k+1}^k$.

Důkaz: Podle definice se levá a pravá strana každé rovnosti liší jenom tím, zda jsme jako vnitřní vrchol cesty povolili použít vrchol $k+1$. Ten je ale už jednou použit jako počáteční, resp. koncový vrchol cesty, takže se uvnitř tak jako tak neobjeví. □

Věta: Floydův-Warshallův algoritmus s přepisováním na místě vypočte matici vzdálenosti grafu bez záporných cyklů v čase $\Theta(n^3)$ a prostoru $\Theta(n^2)$.

Cvičení

1. Jak z výsledku Floydova-Warshallova algoritmu zjistíme, kudy nejkratší cesta mezi nějakými dvěma vrcholy vede?

2. Upravte Floydův-Warshallův algoritmus, aby pro každý vrchol našel nejkratší kružnici, která jím prochází. Předpokládejte, že v grafu nejsou žádné záporné cykly.

3. Upravte Floydův-Warhsallův algoritmus, aby zjistil, zda v grafu existuje záporný cyklus.

6.5 Další cvičení

1. Lze se v algoritmech na hledání nejkratší cesty zbavit záporných hran tím, že ke všem ohodnocením hran přičteme nějaké velké číslo k?

2. Počítačovou síť popíšeme orientovaným grafem, jehož vrcholy odpovídají routerům a hrany linkám mezi nimi. Pro každou linku známe pravděpodobnost toho, že bude

funkční. Pravděpodobnost, že bude funkční nějaká cesta, je dána součinem pravděpodobností jejích hran. Jak pro zadané dva routery najít nejpravděpodobnější cestu mezi nimi?

3. Mějme mapu města ve tvaru orientovaného grafu. Každou hranu ohodnotíme podle toho, jaký nejvyšší kamion po dané ulici může projet. Po cestě tedy projede maximálně tak vysoký náklad, kolik je minimum z ohodnocení jejích hran. Jak pro zadané dva vrcholy najít cestu, po níž projede co nejvyšší náklad?

4. V Tramtárii jezdí po železnici samé rychlíky, které nikde po cestě nestaví. V jízdním řádu je pro každý rychlík uvedeno počáteční a cílové nádraží, čas odjezdu a čas příjezdu. Nyní stojíme v čase t na nádraží a a chceme se co nejrychleji dostat na nádraží b. Navrhněte algoritmus, který najde takové spojení.

5. Pokračujeme v předchozím cvičení: Mezi všemi nejrychlejšími spojeními chceme najít takové, v němž je nejméně přestupů.

6. Směnárna obchoduje s n měnami (měna číslo 1 je koruna) a vyhlašuje matici kurzů K. Kurz K_{ij} říká, kolik za jednu jednotku i-té měny dostaneme jednotek j-té měny. Vymyslete algoritmus, který zjistí, zda existuje posloupnost směn, která začne s jednou korunou a skončí s více korunami.

7. Vymyslete algoritmus, který nalezne všechny hrany, jež leží na alespoň jedné nejkratší cestě.

8. Kritická hrana budiž taková, která leží na všech nejkratších cestách. Tedy ta, jejíž prodloužení by ovlivnilo vzdálenost. Navrhněte algoritmus, který najde všechny kritické hrany.

9. Silnice v mapě máme ohodnocené dvěma čísly: délkou a mýtem (poplatkem za projetí). Jak najít nejlevnější z nejkratších cest?

10.* Sestrojte algoritmus pro řešení soustavy lineárních nerovnic tvaru $x_i - x_j \leq c_{ij}$, kde c_{ij} jsou reálné, ne nutně kladné konstanty.

7 Minimální kostry

7 Minimální kostry

Napadl sníh a přikryl peřinou celé městečko. Po ulicích lze sotva projít pěšky, natož projet autem. Které ulice prohrneme, aby šlo dojet odkudkoliv kamkoliv, a přitom nám házení sněhu dalo co nejméně práce?

Tato otázka vede na hledání minimální kostry grafu. To je slavný problém, jeden z těch, které stály u pomyslné kolébky teorie grafů. Navíc je pro jeho řešení známo hned několik zajímavých efektivních algoritmů. Jim věnujeme tuto kapitolu.

7.1 Od městečka ke kostře

Představme si mapu zasněženého městečka z našeho úvodního příkladu jako graf. Každou hranu ohodnotíme číslem – to bude vyjadřovat množství práce potřebné na prohrnutí ulice. Hledáme tedy podgraf na všech vrcholech, který bude souvislý a použije hrany o co nejmenším součtu ohodnocení.

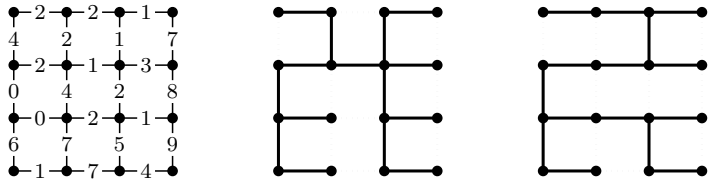
Takový podgraf jistě musí být strom: kdyby se v něm nacházel nějaký cyklus, smažeme libovolnou z hran cyklu. Tím neporušíme souvislost, protože konce hrany jsou nadále propojené zbytkem cyklu. Odstraněním hrany ovšem zlepšíme součet ohodnocení, takže původní podgraf nemohl být optimální. (Zde jsme použili, že odhrnutí sněhu nevyžaduje záporné množství práce, ale to snad není moc troufalé.)

Popišme nyní problém formálně.

Definice:

- Nechť $G = (V, E)$ je souvislý neorientovaný graf a $w : E \to \mathbb{R}$ váhová funkce, která přiřazuje hranám čísla – jejich *váhy.*
- n a m nechť jako obvykle značí počet vrcholů a hran grafu G.
- Váhovou funkci můžeme přirozeně rozšířit na podgrafy: Váha $w(H)$ podgrafu $H \subseteq G$ je součet vah jeho hran.
- *Kostra* grafu G je podgraf, který obsahuje všechny vrcholy a je to strom. Kostra je *minimální,* pokud má mezi všemi kostrami nejmenší váhu.

Jak je vidět z obrázku 7.1, jeden graf může mít více minimálních koster. Brzy dokážeme, že jsou-li váhy všech hran navzájem různé, minimální kostra už je určena jednoznačně. To značně zjednoduší situaci, takže ve zbytku kapitoly budeme unikátnost vah předpokládat.



Obrázek 7.1: Graf s vahami a dvě z jeho minimálních koster

Cvičení

1. Rozmyslete si, že předpoklad unikátních vah není na škodu obecnosti. Ukažte, jak pomocí algoritmu, který unikátnost předpokládá, nalézt jednu z minimálních koster grafu s neunikátními vahami.
2. Upravte definici kostry, aby dávala smysl i pro nesouvislé grafy.
3. Dokažte, že mosty v grafu jsou právě ty hrany, které leží v průniku všech koster.
4. Změníme-li váhu jedné hrany, jak se změní minimální kostra?

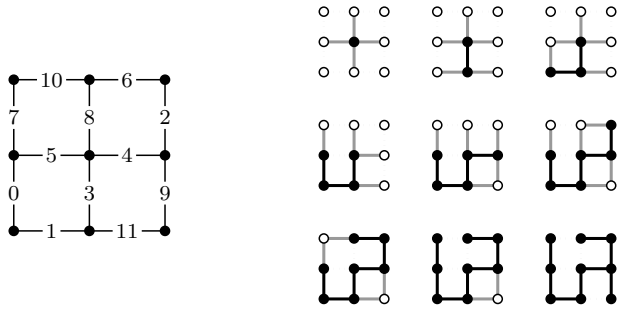
7.2 Jarníkův algoritmus a řezy

Vůbec nejjednodušší algoritmus pro hledání minimální kostry pochází z roku 1930, kdy ho vymyslel český matematik Vojtěch Jarník. Tehdy se o algoritmy málokdo zajímal, takže myšlenka zapadla a až později byla několikrát znovuobjevena – proto se algoritmu říká též Primův nebo Dijkstrův.

Kostru budeme „pěstovat“ z jednoho vrcholu. Začneme se stromem, který obsahuje libovolný jeden vrchol a žádné hrany. Pak vybereme nejlehčí hranu incidentní s tímto vrcholem. Přidáme ji do stromu a postup opakujeme: v každém dalším kroku přidáváme nejlehčí z hran, které vedou mezi vrcholy stromu a zbytkem grafu. Takto pokračujeme, dokud nevznikne celá kostra.

Algoritmus Jarník
Vstup: Souvislý graf s unikátními vahami
1. $v_0 \leftarrow$ libovolný vrchol grafu
2. $T \leftarrow$ strom obsahující vrchol v_0 a žádné hrany
3. Dokud existuje hrana uv taková, že $u \in V(T)$ a $v \notin V(T)$:
4. Nejlehčí takovou hranu přidáme do T.

Výstup: Minimální kostra T



Obrázek 7.2: Příklad výpočtu Jarníkova algoritmu. Černé vrcholy a hrany už byly přidány do kostry, mezi šedivými hranami hledáme tu nejlehčí.

Tento přístup je typickým příkladem takzvaného *hladového algoritmu* – v každém okamžiku vybíráme lokálně nejlepší hranu a neohlížíme se na budoucnost.[1] Hladové algoritmy málokdy naleznou optimální řešení, ale zrovna minimální kostra je jedním z řídkých případů, kdy tomu tak je. K důkazu se ovšem budeme muset propracovat.

Správnost

Lemma: Jarníkův algoritmus se po nejvýše n iteracích zastaví a vydá nějakou kostru zadaného grafu.

Důkaz: Graf pěstovaný algoritmem vzniká z jednoho vrcholu postupným přidáváním listů, takže je to v každém okamžiku výpočtu strom. Po nejvýše n iteracích dojdou vrcholy a algoritmus se musí zastavit.

Kdyby nalezený strom neobsahoval všechny vrcholy, musela by díky souvislosti existovat hrana mezi stromem a zbytkem grafu. Tehdy by se ale algoritmus ještě nezastavil. (Všimněte si, že tuto úvahu jsme už potkali v rozboru algoritmů na prohledávání grafu.) □

Minimalitu kostry bychom mohli dokazovat přímo, ale raději dokážeme trochu obecnější tvrzení o řezech, které se bude hodit i pro další algoritmy.

Definice: Nechť A je nějaká podmnožina vrcholů grafu a B její doplněk. Množině hran, které leží jedním vrcholem v A a druhým v B, budeme říkat *elementární řez* určený množinami A a B.

[1] Proto je možná výstižnější anglický název *greedy algorithm,* čili algoritmus chamtivý, nebo slovenský *pažravý algoritmus.*

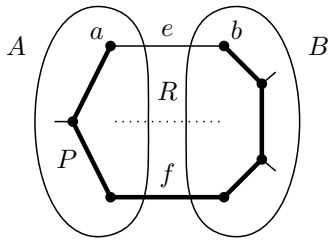
Lemma (řezové): Nechť G je graf opatřený unikátními vahami, R nějaký jeho elementární řez a e nejlehčí hrana tohoto řezu. Pak e leží v každé minimální kostře grafu G.

Důkaz: Dokážeme obměněnou implikaci: pokud nějaká kostra T neobsahuje hranu e, není minimální.

Sledujme situaci na obrázku 7.3. Označme A a B množiny vrcholů, kterými je určen řez R. Hrana e tudíž vede mezi nějakými vrcholy $a \in A$ a $b \in B$. Kostra T musí spojovat vrcholy a a b nějakou cestou P. Tato cesta začíná v množině A a končí v B, takže musí alespoň jednou překročit řez. Nechť f je libovolná hrana, kde se to stalo.

Nyní z kostry T odebereme hranu f. Tím se kostra rozpadne na dva stromy, z nichž jeden obsahuje a a druhý b. Přidáním hrany e stromy opět propojíme a tím získáme jinou kostru T'.

Spočítáme její váhu: $w(T') = w(T) - w(f) + w(e)$. Jelikož hrana e je nejlehčí v řezu, musí platit $w(f) \geq w(e)$. Nerovnost navíc musí být ostrá, neboť váhy jsou unikátní. Proto $w(T') < w(T)$ a T není minimální. □



Obrázek 7.3: Situace v důkazu řezového lemmatu

Každá hrana vybraná Jarníkovým algoritmem je přitom nejlehčí hranou elementárního řezu mezi vrcholy stromu T a zbytkem grafu. Z řezového lemmatu proto plyne, že kostra nalezená Jarníkovým algoritmem je podgrafem každé minimální kostry. Jelikož všechny kostry daného grafu mají stejný počet hran, znamená to, že nalezená kostra je všem minimálním kostrám rovna. Proto platí:

Věta (o minimální kostře): Souvislý graf s unikátními vahami má právě jednu minimální kostru a Jarníkův algoritmus tuto kostru najde.

Navíc víme, že Jarníkův algoritmus váhy pouze porovnává, takže ihned dostáváme:

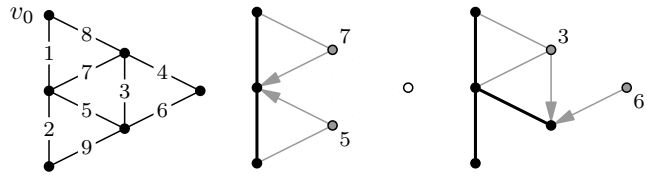
Důsledek: Minimální kostra je jednoznačně určena uspořádáním hran podle vah, na konkrétních hodnotách vah nezáleží.

Implementace

Zbývá rozebrat, jak rychle algoritmus poběží. Už víme, že proběhne nejvýše n iterací. Pokud budeme pokaždé zkoumat všechny hrany, jedna iterace potrvá $\mathcal{O}(m)$, takže celý algoritmus poběží v čase $\mathcal{O}(nm)$.

Opakované vybírání minima navádí k použití haldy. Mohli bychom v haldě uchovávat množinu všech hran řezu (viz cvičení 1), ale existuje elegantnější a rychlejší způsob.

Budeme udržovat *sousední* vrcholy – to jsou ty, které leží mimo strom, ale jsou s ním spojené alespoň jednou hranou. Každému sousedovi s přiřadíme *ohodnocení* $h(s)$. To bude udávat, jakou nejlehčí hranou je soused připojen ke stromu.



Obrázek 7.4: Jeden krok výpočtu v Jarníkově algoritmu s haldou

V každém kroku algoritmu vybereme souseda s nejnižším ohodnocením a připojíme ho ke stromu příslušnou nejlehčí hranou. To je přesně ta hrana, kterou si vybere původní Jarníkův algoritmus. Poté potřebujeme přepočítat sousedy a jejich ohodnocení.

Sledujme obrázek 7.4. Vlevo je nakreslen zadaný graf s vahami. Uprostřed vidíme situaci v průběhu výpočtu: tučné hrany už leží ve stromu, šedivé vrcholy jsou sousední (čísla udávají jejich ohodnocení), šipky ukazují, která hrana řezu je pro daného souseda nejlehčí. V tomto kroku tedy vybereme vrchol s ohodnocením 5, čímž přejdeme do situace nakreslené vpravo.

Pozorování: Pokud ke stromu připojujeme vrchol u, musíme přepočítat sousednost a ohodnocení ostatních vrcholů. Uvažme libovolný vrchol v a rozeberme možné situace:

- Pokud byl v součástí stromu, nemůže se stát sousedním, takže se o něj nemusíme starat.
- Pokud mezi u a v nevede hrana, v okolí vrcholu v se řez nezmění, takže ohodnocení $h(v)$ zůstává stejné.
- Jinak se hrana uv stane hranou řezu. Tehdy:
  - Pakliže v nebyl sousední, stane se sousedním a jeho ohodnocení nastavíme na váhu hrany uv.

- Pokud už sousední byl, bude se do jeho ohodnocení nově započítávat hrana uv, takže $h(v)$ může klesnout.

Stačí tedy projít všechny vrcholy v, do nichž vede z u hrana, a podle potřeby učinit v sousedem nebo snížit jeho ohodnocení.

Na této myšlence je založena následující varianta Jarníkova algoritmu. Kromě ohodnocení vrcholů si budeme pamatovat jejich *stav* (*uvnitř* stromu, *sousední,* případně úplně *mimo*) a u sousedních vrcholů příslušnou nejlehčí hranu. Při inicializaci algoritmu chvíli považujeme počáteční vrchol za souseda, což zjednoduší zápis.

Algoritmus JARNÍK2
Vstup: Souvislý graf s váhovou funkcí w

1. Pro všechny vrcholy v:
2. $\quad stav(v) \leftarrow mimo$
3. $\quad h(v) \leftarrow +\infty$
4. $\quad p(v) \leftarrow nedefinováno$ ◁ *druhý konec nejlehčí hrany*
5. $v_0 \leftarrow$ libovolný vrchol grafu
6. $T \leftarrow$ strom obsahující vrchol v_0 a žádné hrany
7. $stav(v_0) \leftarrow soused$
8. $h(v_0) \leftarrow 0$
9. Dokud existují nějaké sousední vrcholy:
10. $\quad$ Označme u sousední vrchol s nejmenším $h(u)$.
11. $\quad stav(u) \leftarrow uvnitř$
12. $\quad$ Přidáme do T hranu $\{u, p(u)\}$, pokud je $p(u)$ definováno.
13. $\quad$ Pro všechny hrany uv:
14. $\quad\quad$ Je-li $stav(v) \in \{soused, mimo\}$ a $h(v) > w(uv)$:
15. $\quad\quad\quad stav(v) \leftarrow soused$
16. $\quad\quad\quad h(v) \leftarrow w(uv)$
17. $\quad\quad\quad p(v) \leftarrow u$

Výstup: Minimální kostra T

Všimněte si, že takto upravený Jarníkův algoritmus je velice podobný Dijkstrovu algoritmu na hledání nejkratší cesty. Jediný podstatný rozdíl je ve výpočtu ohodnocení vrcholů.

Platí zde tedy vše, co jsme odvodili o složitosti Dijkstrova algoritmu v oddílu 6.2. Uložíme-li všechna ohodnocení do pole, algoritmus poběží v čase $\Theta(n^2)$. Pokud místo pole použijeme haldu, kostru najdeme v čase $\Theta(m \log n)$, případně s Fibonacciho haldou v $\Theta(m + n \log n)$.

Cvičení

1. V rozboru implementace jsme navrhovali uložit všechny hrany řezu do haldy. Rozmyslete si všechny detaily tak, aby váš algoritmus běžel v čase $\mathcal{O}(m \log n)$.

2. Dokažte správnost Jarníkova algoritmu přímo, bez použití řezového lemmatu.

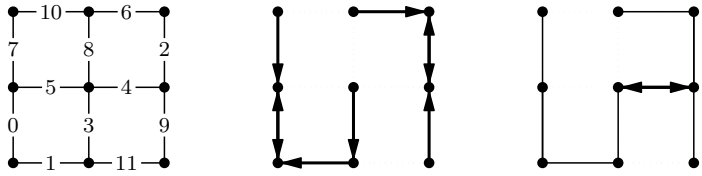
3. Dokažte, že Jarníkův algoritmus funguje i pro grafy, jejichž váhy nejsou unikátní. Jak by pro takové grafy vypadalo řezové lemma?

7.3 Borůvkův algoritmus

Inspirací Jarníkova algoritmu byl algoritmus ještě starší, objevený v roce 1926 Otakarem Borůvkou, pozdějším profesorem matematiky v Brně. Můžeme se na něj dívat jako na paralelní verzi Jarníkova algoritmu: namísto jednoho stromu jich pěstujeme více a v každé iteraci se každý strom sloučí s tím ze svých sousedů, do kterého vede nejlehčí hrana.

Algoritmus BORŮVKA
Vstup: Souvislý graf s unikátními vahami
1. $T \leftarrow (V, \emptyset)$ ◁ *začneme triviálním lesem izolovaných vrcholů*
2. Dokud T není souvislý:
3. Rozložíme T na komponenty souvislosti $T_1, \ldots, T_k$.
4. Pro každý strom T_i najdeme nejlehčí z hran mezi T_i a zbytkem grafu a označíme ji e_i.
5. Přidáme do T hrany $\{e_1, \ldots, e_k\}$.

Výstup: Minimální kostra T



Obrázek 7.5: Příklad výpočtu Borůvkova algoritmu.
Směr šipek ukazuje, který vrchol si vybral kterou hranu.

Správnost dokážeme podobně jako u Jarníkova algoritmu.

Věta: Borůvkův algoritmus se zastaví po nejvýše $\lfloor \log_2 n \rfloor$ iteracích a vydá minimální kostru.

Důkaz: Nejprve si všimneme, že po k iteracích má každý strom lesa T alespoň 2^k vrcholů. To dokážeme indukcí podle k: na počátku ($k = 0$) jsou všechny stromy jednovrcholové. V každé další iteraci se stromy slučují do větších, každý s alespoň jedním sousedním. Proto se velikosti stromů pokaždé minimálně zdvojnásobí.

Nejpozději po $\lfloor \log_2 n \rfloor$ iteracích už velikost stromů dosáhne počtu všech vrcholů, takže může existovat jen jediný strom a algoritmus se zastaví. (Zde jsme opět použili souvislost grafu, rozmyslete si, jak přesně.)

Zbývá nahlédnout, že nalezená kostra je minimální. Opět použijeme řezové lemma: každá hrana e_i, kterou jsme vybrali, je nejlehčí hranou elementárního řezu mezi stromem T_i a zbytkem grafu. Všechny vybrané hrany tedy leží v jednoznačně určené minimální kostře a je jich správný počet. (Zde jsme potřebovali unikátnost vah, viz cvičení 1.) □

Ještě si rozmyslíme implementaci. Ukážeme, že každou iteraci lze zvládnout v lineárním čase s velikostí grafu. Rozklad na komponenty provedeme například prohledáním do šířky. Poté projdeme všechny hrany, pro každou se podíváme, které komponenty spojuje, a započítáme ji do průběžného minima obou komponent. Nakonec vybrané hrany přidáme do kostry.

Důsledek: Borůvkův algoritmus nalezne minimální kostru v čase $\mathcal{O}(m \log n)$.

Cvičení

1. Unikátnost vah je u Borůvkova algoritmu důležitá, protože jinak by v kostře mohl vzniknout cyklus. Najděte příklad grafu, kde se to stane. Jak přesně pro takové grafy selže náš důkaz správnosti? Jak algoritmus opravit?

2. Borůvkův algoritmus můžeme upravit, aby každý strom lesa udržoval zkontrahovaný do jednoho vrcholu. Iterace pak vypadá tak, že si každý vrchol vybere nejlehčí incidentní hranu, tyto hrany zkontrahujeme a zapamatujeme si, že patří do minimální kostry. Ukažte, jak tento algoritmus implementovat tak, aby běžel v čase $\mathcal{O}(m \log n)$. Jak si poradit s násobnými hranami a smyčkami, které vznikají při kontrakci?

3. Sestrojte graf, na kterém algoritmus z předchozího cvičení potřebuje čas $\Omega(m \log n)$.

4.* Ukažte, že pokud algoritmus z cvičení 2 používáme pro rovinné grafy, běží v čase $\Theta(n)$. Opět je potřeba správně ošetřit násobné hrany.

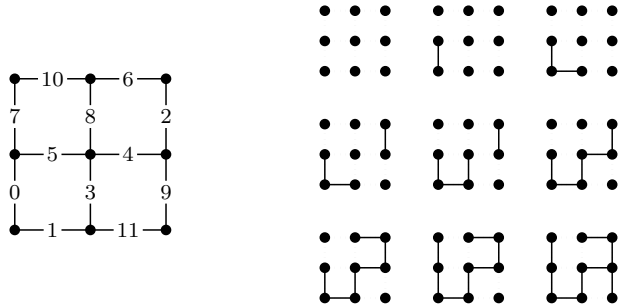
7.4 Kruskalův algoritmus a Union-Find

Třetí algoritmus na hledání minimální kostry popsal v roce 1956 Joseph Kruskal. Opět je založen na hladovém přístupu: zkouší přidávat hrany od nejlehčí po nejtěžší a zahazuje ty, které by vytvořily cyklus.

Algoritmus KRUSKAL
Vstup: Souvislý graf s unikátními vahami

1. Uspořádáme hrany podle vah: $w(e_1) < \ldots < w(e_m)$.
2. $T \leftarrow (V, \emptyset)$ ◁ *začneme lesem samých izolovaných vrcholů*
3. Pro $i = 1, \ldots, m$ opakujeme:
4. $\quad u, v \leftarrow$ krajní vrcholy hrany e_i
5. $\quad$ Pokud u a v leží v různých komponentách lesa T:
6. $\quad\quad T \leftarrow T + e_i$

Výstup: Minimální kostra T



Obrázek 7.6: Příklad výpočtu Kruskalova algoritmu.

Lemma: Kruskalův algoritmus se zastaví a vydá minimální kostru.

Důkaz: Konečnost je zřejmá z omezeného počtu průchodů hlavním cyklem. Nyní ukážeme, že hranu $e = uv$ algoritmus přidá do T právě tehdy, když e leží v minimální kostře.

Pokud algoritmus hranu přidá, stane se tak v okamžiku, kdy se vrcholy u a v nacházejí v nějakých dvou rozdílných stromech T_u a T_v lesa T. Hrana e přitom leží v elementárním řezu oddělujícím strom T_u od zbytku grafu. Navíc mezi hranami tohoto řezu musí být nejlehčí, neboť případnou lehčí hranu by algoritmus potkal dříve a už by stromy spojil. Nyní stačí použít řezové lemma.

Jestliže naopak algoritmus hranu e nepřidá, činí tak proto, že hrana uzavírá cyklus. Ostatní hrany tohoto cyklu algoritmus přidal, takže podle minulého odstavce tyto hrany leží v minimální kostře. Kostra ovšem žádné cykly neobsahuje, takže v ní e určitě neleží. □

Nyní se zamysleme nad implementací. Třídění hran potrvá $\mathcal{O}(m \log m) \subseteq \mathcal{O}(m \log n^2) = \mathcal{O}(m \log n)$. Zbytek algoritmu potřebuje opakovaně testovat, zda hrana spojuje dva různé

stromy. Jistě bychom mohli pokaždé prohledat les do šířky, ale to by trvalo $\mathcal{O}(n)$ na jeden test, celkově tedy $\mathcal{O}(nm)$.

Opakované prohledávání znamená spoustu zbytečné práce. Les se totiž mezi jednotlivými kroky algoritmu mění pouze nepatrně – buď zůstává stejný, nebo do něj přibude jedna hrana. Neuměli bychom komponenty průběžně přepočítávat? Na to by se hodila následující datová struktura:

Definice: *Struktura Union-Find* reprezentuje komponenty souvislosti grafu a umí na nich provádět následující operace:

- FIND(u, v) zjistí, zda vrcholy u a v leží v téže komponentě.
- UNION(u, v) přidá hranu uv, čili dvě komponenty spojí do jedné.

V kroku 5 Kruskalova algoritmu tedy provádíme operaci FIND a v kroku 6 UNION. Složitost celého algoritmu proto můžeme vyjádřit následovně:

Věta: Kruskalův algoritmus najde minimální kostru v čase $\mathcal{O}(m \log n + m \cdot T_f(n) + n \cdot T_u(n))$, kde $T_f(n)$ a $T_u(n)$ jsou časové složitosti operací FIND a UNION na grafech s n vrcholy.

Union-Find s polem

Hledejme nyní rychlou implementaci struktury Union-Find. Nejprve zkusíme, kam nás zavede triviální přístup: pořídíme si pole K, které každému vrcholu přiřadí číslo komponenty. Můžeme si ji představovat jako barvu vrcholu.

FIND se podívá na barvy vrcholů a v konstantním čase je porovná. Veškerou práci oddře UNION: při slučování komponent projde všechny vrcholy jedné komponenty a přebarví je.

Procedura FIND(u, v)

1. Odpovíme ANO právě tehdy, když $K(u) = K(v)$.

Procedura UNION(u, v)

1. Pro všechny vrcholy x:
2. Pokud $K(x) = K(u)$:
3. $K(x) \leftarrow K(v)$

FIND proběhne v čase $\mathcal{O}(1)$ a UNION v $\mathcal{O}(n)$, takže celý Kruskalův algoritmus potrvá $\mathcal{O}(m \log n + m + n^2) = \mathcal{O}(m \log n + n^2)$.

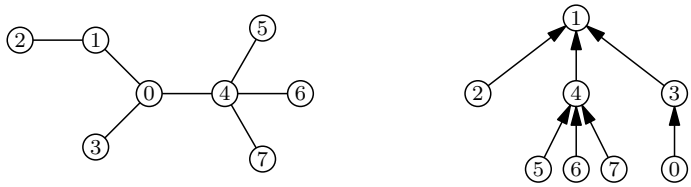
Kvadratická složitost nás sotva uspokojí. Můžeme se pokusit přečíslovávání komponent zrychlit (viz cvičení 3), ale místo toho raději změníme reprezentaci struktury.

Union-Find s keříky

Každou komponentu budeme reprezentovat stromem orientovaným směrem do kořene. Těmto stromům budeme říkat *keříky,* abychom je odlišili od stromů, s nimiž pracuje Kruskalův algoritmus.

Vrcholy každého keříku budou odpovídat vrcholům příslušné komponenty. Hrany nemusí odpovídat hranám původního grafu, jejich podoba záleží na historii operací s naší datovou strukturou.

Do paměti můžeme keříky ukládat přímočaře: každý vrchol v si bude pamatovat svého otce $P(v)$, případně nějakou speciální hodnotu $\emptyset$, pokud je kořenem.



Obrázek 7.7: Komponenta a její reprezentace keříkem

Operace FIND vystoupá z každého vrcholu do kořene keříku a porovná kořeny:

Procedura KOŘEN(x)

1. Dokud $P(x) \neq \emptyset$:
2. $\quad x \leftarrow P(x)$
3. Vrátíme kořen x.

Procedura FIND(u, v)

1. Vrátíme ANO právě tehdy, když KOŘEN(u) = KOŘEN(v).

Hledání kořene, a tím pádem i operace FIND trvají lineárně s hloubkou keříku.

Operace UNION sloučí komponenty tak, že mezi kořeny keříků natáhne novou hranu. Může si přitom vybrat, který kořen připojí pod který – obojí bude správně. Ukážeme, že vhodnou volbou udržíme keříky mělké a FIND rychlý.

Uděláme to takto: Do kořene každého keříku uložíme číslo $H(v)$, jež bude říkat, jak je tento keřík hluboký. Na počátku mají všechny keříky hloubku 0. Při slučování keříků připojíme mělčí keřík pod kořen toho hlubšího a hloubka se nezmění. Jsou-li oba stejně hluboké, rozhodneme se libovolně a keřík se prohloubí. UNION bude vypadat takto:

Procedura UNION(u, v):

1. $a \leftarrow$ KOŘEN(u), $b \leftarrow$ KOŘEN(v)
2. Je-li $a = b$, ihned skončíme.
3. Pokud $H(a) < H(b)$:
4. $\quad P(a) \leftarrow b$
5. Pokud $H(a) > H(b)$:
6. $\quad P(b) \leftarrow a$
7. Jinak:
8. $\quad P(b) \leftarrow a$
9. $\quad H(a) \leftarrow H(a) + 1$

Teď ukážeme, že naše slučovací pravidlo zaručí, že keříky jsou vždy mělké (a zaslouží si svůj název).

Invariant: Keřík hloubky h obsahuje alespoň 2^h vrcholů.

Důkaz: Budeme postupovat indukcí podle počtu operací UNION. Na počátku algoritmu mají všechny keříky hloubku 0 a $2^0 = 1$ vrchol.

Nechť nyní provádíme UNION(u, v) a hloubky obou keříků jsou různé. Připojením mělčího keříku pod kořen toho hlubšího se hloubka nezmění a počet vrcholů neklesne, takže nerovnost stále platí.

Pokud mají oba keříky tutéž hloubku h, víme z indukčního předpokladu, že každý z nich obsahuje minimálně 2^h vrcholů. Jejich sloučením tudíž vznikne keřík hloubky $h+1$ o alespoň $2 \cdot 2^h = 2^{h+1}$ vrcholech. Nerovnost je tedy opět splněna. □

Důsledek: Hloubky keříků nepřekročí $\log n$.

Důkaz: Strom větší hloubky by podle invariantu obsahoval více než n vrcholů. □

Věta: Časová složitost operací UNION a FIND v keříkové reprezentaci je $\mathcal{O}(\log n)$.

Důkaz: Hledání kořene keříku zabere čas lineární s jeho hloubkou, tedy $\mathcal{O}(\log n)$. Obě operace datové struktury provedou dvě hledání kořene a $\mathcal{O}(1)$ dalších operací. □

Důsledek: Kruskalův algoritmus s keříkovou strukturou pro Union-Find najde minimální kostru v čase $\mathcal{O}(m \log n)$.

Cvičení

1. Dokažte správnost Kruskalova algoritmu přímo, bez použití řezového lemmatu.

2. Fungoval by Kruskalův algoritmus pro neunikátní váhy hran?

3. Datová struktura pro Union-Find s polem by se dala zrychlit tím, že bychom pokaždé přečíslovávali tu menší z komponent. Dokažte, že pak je během života struktury každý vrchol přečíslován nejvýše $(\log n)$-krát. Co z toho plyne pro složitost operací? Nezapomeňte, že je potřeba efektivně zjistit, která z komponent je menší, a vyjmenovat její vrcholy.

4. Jaká posloupnost UNIONů odpovídá obrázku 7.7?

7.5* Komprese cest

Keříkovou datovou strukturu můžeme dále zrychlovat. Kruskalův algoritmus tím sice nezrychlíme, protože nás stejně brzdí třídění hran. Ale co kdybychom hrany dostali už setříděné, nebo jejich váhy byly celočíselné a šlo je třídit přihrádkově? Tehdy můžeme operace s keříky zrychlit ještě jedním trikem: *kompresí cest.*

Kdykoliv hledáme kořen nějakého keříku, trávíme tím čas lineární v délce cesty do kořene. Když už to děláme, zkusme při tom strukturu trochu vylepšit. Všechny vrcholy, přes které jsme prošli, převěsíme rovnou pod kořen. Tím si ušetříme práci v budoucnosti.

Procedura KOŘENSKOMPRESÍ(x)
1. $r \leftarrow$ KOŘEN(x)
2. Dokud $P(x) \neq r$:
3. $\quad t \leftarrow P(x)$
4. $\quad P(x) \leftarrow r$
5. $\quad x \leftarrow t$
6. Vrátíme kořen r.

Pozor na to, že převěšením vrcholů mohla klesnout hloubka keříku. Uložené hloubky, které používáme v UNIONech, tím pádem přestanou souhlasit se skutečností. Místo abychom je přepočítávali, necháme je být a přejmenujeme je. Budeme jim říkat *ranky* a budeme s nimi zacházet úplně stejně, jako jsme předtím zacházeli s hloubkami.[2]

Podobně jako u původní struktury bude platit následující invariant:

Invariant R: Keřík s kořenem ranku r má hloubku nejvýše r a obsahuje alespoň 2^r vrcholů.

Důkaz: Indukcí podle počtu operací UNION. Komprese cest nemění ani rank kořene, ani počet vrcholů, takže se jí nemusíme zabývat. □

[2] Anglický *rank* by se dal do češtiny přeložit jako *hodnost.* V lineární algebře se pojem hodnosti používá, ale při studiu datových struktur bývá zvykem používat původní anglický termín.

Ranky jsou tedy stejně jako hloubky nejvýše logaritmické, takže složitost operací v nejhorším případě zůstává $\mathcal{O}(\log n)$. Ukážeme, že průměrná složitost se výrazně snížila. (To je typický příklad takzvané amortizované složitosti, s níž se blíže setkáme v kapitole 9.)

Definice: *Věžovou funkci* $2\uparrow k$ definujeme následovně: $2\uparrow 0 = 1$, $2\uparrow(k+1) = 2^{2\uparrow k}$.

Definice: *Iterovaný logaritmus* $\log^* x$ je inverzí věžové funkce. Udává nejmenší k takové, že $2\uparrow k \geq x$.

Příklad: Funkce $2\uparrow k$ roste přímo závratně:

$$\begin{aligned} 2\uparrow 1 &= 2, \\ 2\uparrow 2 &= 2^2 = 4 \\ 2\uparrow 3 &= 2^4 = 16 \\ 2\uparrow 4 &= 2^{16} = 65\,536 \\ 2\uparrow 5 &= 2^{65\,536} \approx 10^{19\,728} \end{aligned}$$

Iterovaný logaritmus libovolného „rozumného“ čísla je tedy nejvýše 5.

Věta: Ve struktuře s kompresí cest na n vrcholech trvá provedení $n-1$ operací Union a m operací Find celkově $\mathcal{O}((n+m)\cdot \log^* n)$.

Ve zbytku tohoto oddílu větu dokážeme.

Pro potřeby důkazu budeme uvažovat ranky všech vrcholů, nejen kořenů – každý vrchol si ponese svůj rank z doby, kdy byl naposledy kořenem. Struktura sama se ovšem podle ranků vnitřních vrcholů neřídí a nemusí si je ani pamatovat. Dokažme dva invarianty o rankách vrcholů.

Invariant C: Na každé cestě z vrcholu do kořene příslušného keříku ranky ostře rostou. Jinými slovy rank vrcholu, který není kořen, je menší, než je rank jeho otce.

Důkaz: Pro jednovrcholové keříky tvrzení jistě platí. Dále se keříky mění dvojím způsobem:

Přidání hrany v operaci Union: Nechť připojíme vrchol b pod a. Cesty do kořene z vrcholů, které původně ležely pod a, zůstanou zachovány, pouze se vrcholu a mohl zvýšit rank. Cesty z vrcholů pod b se rozšíří o hranu ba, na které rank v každém případě roste.

Komprese cest nahrazuje otce vrcholu jeho vzdálenějším předkem, takže se rank otce může jedině zvýšit. □

Invariant P: Počet vrcholů ranku r nepřesáhne $n/2^r$.

Důkaz: Kdybychom nekomprimovali cesty, bylo by to snadné: vrchol ranku r by měl alespoň 2^r potomků (dokud je kořenem, plyne to z invariantu **R**; jakmile přestane být, potomci se už nikdy nezmění). Navíc díky invariantu **C** nemá žádný vrchol více předků ranku r, takže v keříku najdeme tolik disjunktních podkeříků velikosti alespoň 2^r, kolik je vrcholů ranku r.

Komprese cest nemůže invariant porušit, jelikož nemění ani ranky, ani rozhodnutí, jak proběhne který UNION. □

Nyní vrcholy ve struktuře rozdělíme do *skupin* podle ranků: k-tá skupina bude tvořena těmi vrcholy, jejichž rank je od $2\uparrow(k-1)+1$ do $2\uparrow k$. Vrcholy jsou tedy rozděleny do $1+\log^* \log n$ skupin (nezapomeňte, že ranky nepřesahují $\log n$). Odhadněme nyní shora počet vrcholů v k-té skupině.

Invariant S: V k-té skupině leží nejvýše $n/(2\uparrow k)$ vrcholů.

Důkaz: Sečteme odhad $n/2^r$ z invariantu **P** přes všechny ranky ve skupině:

$$\frac{n}{2^{2\uparrow(k-1)+1}} + \frac{n}{2^{2\uparrow(k-1)+2}} + \cdots + \frac{n}{2^{2\uparrow k}} \leq \frac{n}{2^{2\uparrow(k-1)}} \cdot \sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{n}{2^{2\uparrow(k-1)}} \cdot 1 = \frac{n}{2\uparrow k}.$$

□

Důkaz věty: Operace UNION a FIND potřebují nekonstantní čas pouze na vystoupání po cestě ze zadaného vrcholu do kořene keříku. Čas strávený na této cestě je přímo úměrný počtu hran cesty. Celá cesta je přitom rozpojena a všechny vrcholy ležící na ní jsou přepojeny přímo pod kořen keříku.

Hrany cesty, které spojují vrcholy z různých skupin (takových je $\mathcal{O}(\log^* n)$), naúčtujeme právě prováděné operaci. Celkem jimi tedy strávíme čas $\mathcal{O}((n+m)\cdot \log^* n)$. Zbylé hrany budeme počítat přes celou dobu běhu algoritmu a účtovat je vrcholům.

Uvažme vrchol v v k-té skupině, jehož rodič leží také v k-té skupině. Jelikož hrany na cestách do kořene ostře rostou, každým přepojením vrcholu v rank jeho rodiče vzroste. Proto po nejvýše $2\uparrow k$ přepojeních se bude rodič vrcholu v nacházet v některé z vyšších skupin. Jelikož rank vrcholu v se už nikdy nezmění, bude hrana z v do jeho otce již navždy hranou mezi skupinami. Každému vrcholu v k-té skupině tedy naúčtujeme nejvýše $2\uparrow k$ přepojení a jelikož, jak už víme, jeho skupina obsahuje nejvýše $n/(2\uparrow k)$ vrcholů, naúčtujeme celé skupině čas $\mathcal{O}(n)$ a všem skupinám dohromady $\mathcal{O}(n \log^* n)$. □

Dodejme, že komprese cest se ve skutečnosti chová ještě lépe, než jsme dokázali. Správnou funkcí, která popisuje rychlost operací, není iterovaný logaritmus, ale ještě mnohem pomaleji rostoucí *inverzní Ackermannova funkce.* Rozdíl se nicméně projeví až pro nerealisticky velké vstupy a důkaz příslušné věty je zcela mimo možnosti našeho úvodního textu.

7.6 Další cvičení

1. Vymyslete algoritmus na hledání kostry grafu, v němž jsou váhy hran přirozená čísla od 1 do 5.

2.* Rozmyslete si, jak v případě, kdy váhy nejsou unikátní, najít *všechny* minimální kostry. Jelikož koster může být mnoho (pro úplný graf s jednotkovými vahami jich je n^{n-2}), snažte se o co nejlepší složitost v závislosti na velikosti grafu a počtu minimálních koster.

3. Jak hledat minimální kostru za předpokladu, že se určené vrcholy musí stát jejími listy? Jako další listy můžete využívat i neoznačené vrcholy.

4.* *Rekonstrukce metriky:* Mějme strom na množině $\{1, \ldots, n\}$ s ohodnocenými hranami. Metrika stromu je matice, která na pozici i, j udává vzdálenost mezi vrcholy i a j. Vymyslete algoritmus, jenž sestrojí strom se zadanou metrikou, případně odpoví, že takový strom neexistuje.

5.* Známe-li minimální kostru, jak najít druhou nejmenší?

8 Vyhledávací stromy

8 Vyhledávací stromy

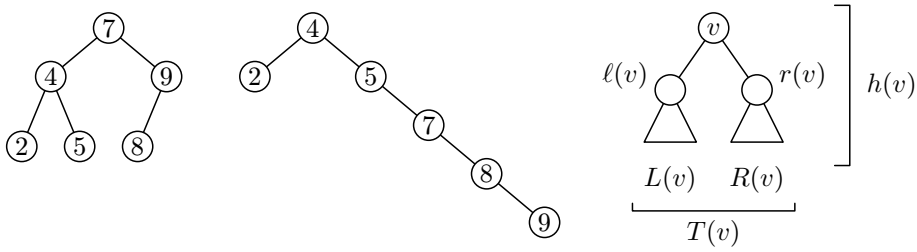
V kapitole 4 jsme se vydali po stopě datových struktur pro efektivní reprezentaci *množin* a *slovníků*. To nás nyní dovede k různým variantám vyhledávacích stromů. Začneme těmi binárními, ale později zjistíme, že se může hodit uvažovat o stromech obecněji.

8.1 Binární vyhledávací stromy

Množinu můžeme uložit do uspořádaného pole. V něm už umíme v logaritmickém čase vyhledávat, ovšem jakákoliv změna je pomalá. Pokusíme se proto od pole přejít k obecnější struktuře, která bude „pružnější“. Inspirujeme se popisem binárního vyhledávání pomocí rozhodovacích stromů. Ty už se nám hodily v oddílu 3.3: dokázali jsme pomocí nich, že binární vyhledávání je optimální.

Připomeňme si, jak takový rozhodovací strom sestrojit. Kořen popisuje první porovnání: leží v něm prostřední prvek pole a má dva syny – *levého* a *pravého*. Ti odpovídají dvěma možným výsledkům porovnání: pokud je hledaná hodnota menší, jdeme doleva; pokud větší, tak doprava. Následující vrchol nám řekne, jaké další porovnání máme provést, a tak dále až do doby, kdy buďto nastane rovnost (takže jsme našli), nebo se pokusíme přejít do neexistujícího vrcholu (takže hledaná hodnota v poli není).

Pro úspěšné vyhledávání přitom nepotřebujeme, abychom při konstrukci stromu pokaždé vybírali prostřední prvek intervalu. Pokud bychom volili jinak, dostaneme odlišný strom. Pomocí něj také půjde hledat, jen možná pomaleji – to je vidět na následujícím obrázku.



Obrázek 8.1: Dva binární vyhledávací stromy a jejich značení

Co všechno musí strom splňovat, aby se podle něj dalo hledat, přetavíme do následujících definic. Nejprve připomeneme definici binárního stromu z oddílu 4.2:

Definice: Strom nazveme *binární*, pokud je zakořeněný a každý vrchol má nejvýše dva syny, u nichž rozlišujeme, který je levý a který pravý.

Značení: Pro vrchol v binárního stromu T značíme:

- $T(v)$ – *podstrom* obsahující vrchol v a všechny jeho potomky
- $\ell(v)$ a $r(v)$ – levý a pravý syn vrcholu v
- $L(v)$ a $R(v)$ – levý a pravý podstrom vrcholu v, tedy $T(\ell(v))$ a $T(r(v))$
- $h(v)$ – *hloubka* stromu $T(v)$, čili maximum z délek cest z v do listů

Pokud vrchol nemá levého syna, položíme $\ell(v) = r(v) = \emptyset$. Pak se hodí dodefinovat, že $T(\emptyset)$ je prázdný strom a $h(\emptyset) = -1$.

Definice: *Binární vyhledávací strom* (BVS) je binární strom, jehož každému vrcholu v přiřadíme unikátní *klíč* $k(v)$ z univerza. Přitom musí pro každý vrchol v platit:

- Kdykoliv $a \in L(v)$, pak $k(a) < k(v)$.
- Kdykoliv $b \in R(v)$, pak $k(b) > k(v)$.

Jinak řečeno, vrchol v odděluje klíče v levém a pravém podstromu.

Základní operace

Pomocí vyhledávacích stromů můžeme přirozeně reprezentovat množiny: klíče uložené ve vrcholech budou odpovídat prvkům množiny. A kdybychom místo množiny chtěli slovník, přidáme do vrcholu hodnotu přiřazenou danému klíči.

Nyní ukážeme, jak provádět jednotlivé množinové operace. Jelikož stromy jsou definované rekurzivně, je přirozené zacházet s nimi rekurzivními funkcemi. Dobře je to vidět na následující funkci, která vypíše všechny prvky množiny.

Procedura BvsShow (uspořádaný výpis BVS)
Vstup: Kořen BVS v

1. Pokud $v = \emptyset$, jedná se o prázdný strom a hned skončíme.
2. Zavoláme BvsShow($\ell(v)$).
3. Vypíšeme klíč uložený ve vrcholu v.
4. Zavoláme BvsShow($r(v)$).

Pokaždé tedy projdeme levý podstrom, pak kořen, a nakonec pravý podstrom. To nám dává tak zvané *symetrické pořadí* vrcholů (někdy také *in-order*). Jejich klíče přitom vypisujeme od nejmenšího k největšímu.

Hledání vrcholu s daným klíčem x prochází stromem od kořene a každý vrchol v porovná s x. Pokud je $x < k(v)$, pak se podle definice nemůže x nacházet v pravém podstromu, takže zamíříme doleva. Je-li naopak $x > k(v)$, nic nepokazíme krokem doprava. Časem tedy x najdeme, anebo vyloučíme všechny možnosti, kde by se mohlo nacházet.

Algoritmus formulujeme jako rekurzivní funkci, kterou vždy voláme na kořen nějakého podstromu a vrátí nám nalezený vrchol.

Procedura BvsFind (hledání v BVS)
Vstup: Kořen BVS v, hledaný klíč x
1. Pokud $v = \emptyset$, vrátíme $\emptyset$.
2. Pokud $x = k(v)$, vrátíme v.
3. Pokud $x < k(v)$, vrátíme BvsFind$(\ell(v), x)$.
4. Pokud $x > k(v)$, vrátíme BvsFind$(r(v), x)$.

Výstup: Vrchol s klíčem x, anebo $\emptyset$

Minimum z prvků množiny spočteme snadno: půjdeme stále doleva, dokud je kam. Klíče menší než ten aktuální se totiž mohou nacházet pouze v levém podstromu.

Procedura BvsMin (minimum v BVS)
Vstup: Kořen BVS v
1. Pokud $\ell(v) = \emptyset$, vrátíme vrchol v.
2. Jinak vrátíme BvsMin$(\ell(v))$.

Výstup: Vrchol obsahující nejmenší klíč

Vkládání nového prvku funguje velmi podobně jako vyhledávání, pouze v okamžiku, kdy by vyhledávací algoritmus měl přejít do neexistujícího vrcholu, připojíme tam nový list. Rozmyslíme si, že to je jediné místo, kde podle definice nový prvek smí ležet. Operaci opět popíšeme rekurzivně. Funkce dostane na vstupu kořen (pod)stromu a vrátí nový kořen.

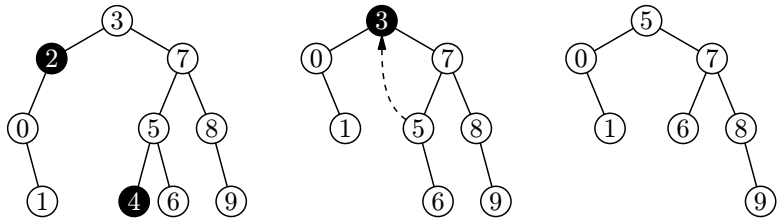
Procedura BvsInsert (vkládání do BVS)
Vstup: Kořen BVS v, vkládaný klíč x
1. Pokud $v = \emptyset$, vytvoříme nový vrchol v s klíčem x a skončíme.
2. Pokud $x < k(v)$, položíme $\ell(v) \leftarrow$ BvsInsert$(\ell(v), x)$.
3. Pokud $x > k(v)$, položíme $r(v) \leftarrow$ BvsInsert$(r(v), x)$.
4. Pokud $x = k(v)$, klíč x se ve stromu již nachází a není třeba nic měnit.

Výstup: Nový kořen v

Při *mazání* může nastat několik různých případů (viz obrázek 8.2). Nechť v je vrchol, který chceme smazat. Je-li v list, můžeme tento list prostě odstranit, čímž provedeme operaci opačnou k BvsInsertu. Má-li v právě jednoho syna, postačí v „vystřihnout“, tedy nahradit synem.

Ošemetný je případ se dvěma syny. Tehdy nemůžeme v jen tak smazat, neboť by syny nebylo kam připojit. Proto nalezneme následníka s vrcholu v, což je nejlevější vrchol

v pravém podstromu. Ten má nejvýše jednoho syna, takže smažeme s místo v a jeho hodnotu přesuneme do v.



Obrázek 8.2: Různé situace při mazání.
Nejprve mažeme vrcholy 2 a 4, poté 3.

Procedura BvsDelete (mazání z BVS)
Vstup: Kořen BVS v, mazaný klíč x

1. Pokud $v = \emptyset$, vrátíme $\emptyset$. ◁ *klíč x ve stromu nebyl*
2. Pokud $x < k(v)$, položíme $\ell(v) \leftarrow$ BvsDelete$(\ell(v), x)$.
3. Pokud $x > k(v)$, položíme $r(v) \leftarrow$ BvsDelete$(r(v), x)$.
4. Pokud $x = k(v)$: ◁ *chystáme se smazat vrchol v*
5. Pokud $\ell(v) = r(v) = \emptyset$, vrátíme $\emptyset$. ◁ *byl to list*
6. Pokud $\ell(v) = \emptyset$, vrátíme $r(v)$. ◁ *existoval jen pravý syn*
7. Pokud $r(v) = \emptyset$, vrátíme $\ell(v)$. ◁ *existoval jen levý syn*
8. $s \leftarrow$ BvsMin$(r(v))$ ◁ *máme oba syny: nahradíme následníka s*
9. $k(v) \leftarrow k(s)$
10. $r(v) \leftarrow$ BvsDelete$(r(v), s)$
11. Vrátíme v.

Výstup: Nový kořen v

Vyváženost stromů

Zamysleme se nad složitostí stromových operací pro strom na n vrcholech.

BvsShow projde všechny vrcholy a v každém stráví konstantní čas, takže běží v čase $\Theta(n)$. Ostatní operace projdou po nějaké cestě od kořene směrem k listům, a to buďto jednou tam a jednou zpět, nebo (v nejsložitějším případě BvsDelete) oběma směry dvakrát. Jejich časová složitost v nejhorším případě proto bude $\Theta(\textit{hloubka stromu})$.

Hloubka ovšem závisí na tom, jak moc je strom „košatý“. V příznivém případě vyjde sympatických $\mathcal{O}(\log n)$, jenže dalšími operacemi může strom degenerovat. Začneme-li třeba

s prázdným stromem a postupně vložíme klíče $1, \ldots, n$ v tomto pořadí, vznikne „liána“ hloubky $\Theta(n)$.

Budeme se proto snažit stromy *vyvažovat*, aby jejich hloubka příliš nerostla. Zkusme se opět držet paralely s binárním vyhledáváním.

Definice: Binární vyhledávací strom nazveme *dokonale vyvážený,* pokud pro každý jeho vrchol v platí

$$\big||L(v)| - |R(v)|\big| \leq 1.$$

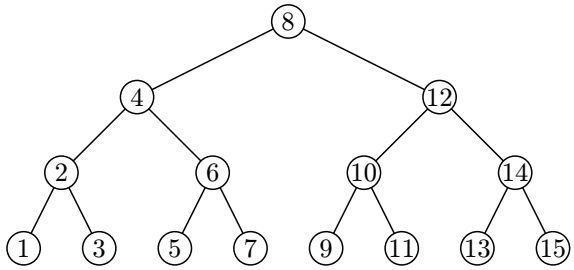
Jinými slovy počet vrcholů levého a pravého podstromu se smí lišit nejvýše o 1.

Pozorování: Dokonale vyvážený strom má hloubku $\lfloor \log_2 n \rfloor$, jelikož na kterékoliv cestě z kořene do listu klesá velikost podstromů s každým krokem alespoň dvakrát.

Dokonale vyvážený strom tedy zaručuje rychlé vyhledávání. Navíc pokud všechny prvky množiny známe předem, můžeme si takový strom snadno pořídit: z uspořádané posloupnosti ho vytvoříme v lineárním čase (cvičení 3). Tím bohužel dobré zprávy končí: ukážeme, že po vložení nebo smazání vrcholu nelze dokonalou vyváženost obnovit rychle.

Věta: Pro každou implementaci operací Insert a Delete udržujících strom dokonale vyvážený platí, že Insert nebo Delete trvá $\Omega(n)$ pro nekonečně mnoho různých n.

Důkaz: Nejprve si představíme, jak bude vypadat dokonale vyvážený BVS s klíči $1, \ldots, n$, kde $n = 2^k - 1$. Sledujme obrázek 8.3. Tvar stromu je určen jednoznačně: Kořenem musí být prostřední z klíčů (jinak by se levý a pravý podstrom kořene lišily o více než 1 vrchol). Levý a pravý podstrom proto mají právě $(n-1)/2 = 2^{k-1} - 1$ vrcholů, takže jejich kořeny jsou opět určeny jednoznačně a tak dále. Navíc si všimneme, že všechna lichá čísla jsou umístěna v listech stromu.



Obrázek 8.3: Dokonale vyvážený BVS

Nyní na tomto stromu provedeme následující posloupnost operací:

$$\textsc{Insert}(n+1),\ \textsc{Delete}(1),\ \textsc{Insert}(n+2),\ \textsc{Delete}(2),\ \ldots$$

Po provedení i-té dvojice operací bude strom obsahovat hodnoty $i+1,\ldots,i+n$. Podle toho, zda je i sudé nebo liché, se budou v listech nacházet buď všechna sudá, nebo všechna lichá čísla. Pokaždé se proto všem vrcholům změní, zda jsou listy, na což je potřeba upravit $\Omega(n)$ ukazatelů. Tedy aspoň jedna z operací Insert a Delete trvá $\Omega(n)$. □

Cvičení

1. Rekurze je pro operace s BVS přirozená, ale v některých programovacích jazycích je pomalejší než obyčejný cyklus. Navrhněte, jak operace s BVS naprogramovat nerekurzivně.

2. Místo vrcholu se dvěma syny jsme mazali jeho následníka. Samozřejmě bychom si místo toho mohli vybrat předchůdce. Jak by se algoritmus změnil?

3. Navrhněte algoritmus, který ze setříděného pole vyrobí v lineárním čase dokonale vyvážený BVS.

4. Navrhněte algoritmus, který v lineárním čase zadaný BVS dokonale vyváží.

5.** Vyřešte předchozí cvičení tak, aby vám kromě zadaného stromu stačilo konstantní množství paměti. Pokud nevíte, jak na to, zkuste to nejprve s logaritmickou pamětí.

6.* Ukázali jsme, že dokonale vyvážený strom o 2^k-1 vrcholech je *úplný* – všech k hladin obsahuje nejvyšší možný počet vrcholů. Dokažte, že ostatní dokonale vyvážené stromy mají podobnou strukturu, jen na poslední hladině mohou některé vrcholy chybět.

7. Zatím jsme předpokládali, že klíče je možné porovnávat v konstantním čase. Jak to dopadne, pokud klíče budou třeba řetězce o ℓ znacích? Stanovte složitost vyhledávání a srovnejte ji s hledáním v písmenkovém stromu z oddílu 4.3.

8. Navrhněte algoritmus, který dostane dva BVS T_1, T_2 a sloučí jejich obsah do jediného BVS. Algoritmus by měl pracovat v čase $\mathcal{O}(|T_1|+|T_2|)$.

9. Navrhněte operaci BvsSplit, která dostane BVS T a hodnotu s, a rozdělí strom na dva BVS T_1 a T_2 takové, že hodnoty v T_1 jsou menší než s a hodnoty v T_2 jsou větší než s.

10. Naše tvrzení o náročnosti operací Insert a Delete v dokonale vyváženém stromu lze ještě zesílit. Dokažte, že lineární musí být složitost *obou* operací.

11. Ukažte, jak zjistit *následníka* daného vrcholu, tedy vrchol s nejbližší větší hodnotou.

12. Dokažte, že projdeme-li celý strom opakovaným hledáním následníka, strávíme tím čas $\Theta(n)$.

13.* *Úsporné stromy:* Obvyklá reprezentace BVS v paměti potřebuje v každém vrcholu 3 ukazatele: na levého syna, na pravého syna a na otce. Ukažte, jak si vystačit se dvěma ukazateli. Původní 3 ukazatele by z těch vašich mělo jít spočítat v konstantním čase.

8.2 Hloubkové vyvážení: AVL stromy

Zjistili jsme, že dokonale vyvážené stromy nelze efektivně udržovat. Důvodem je, že jejich definice velmi striktně omezuje tvar stromu, takže i vložení jediného klíče může vynutit přebudování celého stromu. Zavedeme proto o trochu slabší podmínku.

Definice: Binární vyhledávací strom nazveme *hloubkově vyvážený*, pokud pro každý jeho vrchol v platí

$$\big|h(\ell(v)) - h(r(v))\big| \leq 1.$$

Jinými slovy, hloubka levého a pravého podstromu se vždy liší nejvýše o jedna.

Stromům s hloubkovým vyvážením se říká *AVL stromy,* neboť je vymysleli v roce 1962 ruští matematici Georgij Maximovič Adělson-Veľskij a Jevgenij Michailovič Landis. Nyní dokážeme, že AVL stromy mají logaritmickou hloubku.

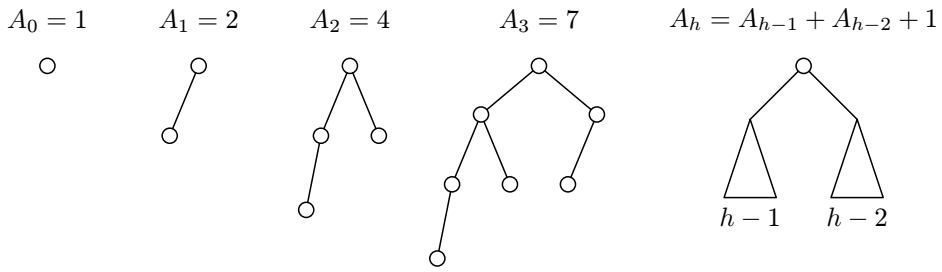
Tvrzení: AVL strom na n vrcholech má hloubku $\Theta(\log n)$.

Důkaz: Nejprve pro každé $h \geq 0$ stanovíme A_h, což bude minimální možný počet vrcholů v AVL stromu hloubky h, a dokážeme, že tento počet roste s hloubkou exponenciálně.

Pro malá h stačí rozebrat možné případy podle obrázku 8.4.

Pro větší h uvažujme, jak může minimální AVL strom o h hladinách vypadat. Jeho kořen musí mít dva podstromy, jeden z nich hloubky $h-1$ a druhý hloubky $h-2$ (kdyby měl také $h-1$, měl by zbytečně mnoho vrcholů). Oba tyto podstromy musí být minimální AVL stromy dané hloubky. Musí tedy platit $A_h = A_{h-1} + A_{h-2} + 1$.

Tato rekurence připomíná Fibonacciho posloupnost z oddílu 1.4. Vskutku: platí $A_h = F_{h+3} - 1$, kde F_k je k-té Fibonacciho číslo. Z toho bychom mohli získat explicitní vzorec pro A_h, ale pro důkaz našeho tvrzení postačí jednodušší asymptotický odhad.



Obrázek 8.4: Minimální AVL stromy
pro hloubky 0 až 3 a obecný případ

Dokážeme indukcí, že $A_h \geq 2^{h/2}$. Jistě je $A_0 = 1 \geq 2^{0/2} = 1$ a $A_1 = 2 \geq 2^{1/2} \doteq 1.414$. Indukční krok pak vypadá následovně:

$$A_h = 1 + A_{h-1} + A_{h-2} > 2^{\frac{h-1}{2}} + 2^{\frac{h-2}{2}} = 2^{\frac{h}{2}} \cdot (2^{-\frac{1}{2}} + 2^{-1}) \geq 2^{\frac{h}{2}} \cdot 1.2 > 2^{\frac{h}{2}}.$$

Tím jsme dokázali, že $A_h \geq c^h$ pro konstantu $c = \sqrt{2}$. Proto AVL strom o n vrcholech může mít nejvýše $\log_c n$ hladin – kdyby jich měl více, obsahoval by více než $c^{\log_c n} = n$ vrcholů.

Zbývá dokázat, že logaritmická hloubka je také nutná. K tomu dojdeme podobně: nahlédneme, že největší možný AVL strom hloubky h je úplný binární strom s $2^{h+1} - 1$ vrcholy. Tudíž minimální možná hloubka AVL stromu je $\Omega(\log n)$. □

Vyvažování rotacemi

Jak budou vypadat operace na AVL stromech? Find bude totožný. Operace Insert a Delete začnou stejně jako u obecného BVS, ale poté ještě ověří, zda strom zůstal hloubkově vyvážený, a případně zasáhnou, aby se vyváženost obnovila.

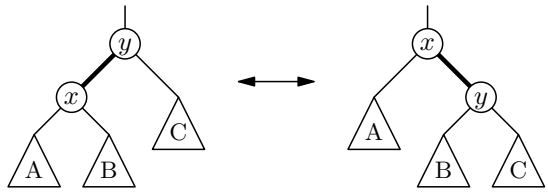
Abychom poznali, kdy je zásah potřeba, budeme v každém vrcholu v udržovat číslo $\delta(v) = h(r(v)) - h(\ell(v))$. To je takzvané *znaménko* vrcholu, které v korektním AVL stromu může nabývat jen těchto hodnot:

- $\delta(v) = 1$ (pravý podstrom je hlubší) – takový vrchol značíme $+$,
- $\delta(v) = -1$ (levý podstrom hlubší) – značíme $-$,
- $\delta(v) = 0$ (oba podstromy stejně hluboké) – značíme 0.

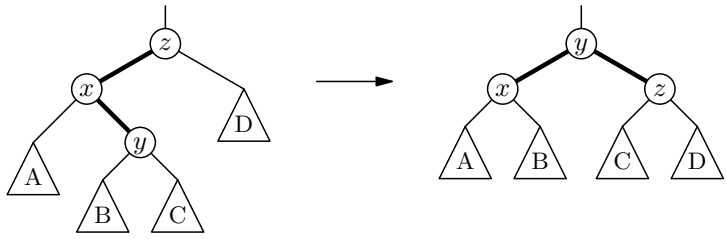
Jakmile narazíme na jiné $\delta(v)$, strom opravíme provedením jedné nebo více rotací.

Rotace je operace, která „otočí“ hranu mezi dvěma vrcholy a přepojí jejich podstromy tak, aby byli i nadále synové vzhledem k otcům správně uspořádáni. To lze provést jediným způsobem, který najdete na obrázku 8.5.

Často také potkáme *dvojitou rotaci* z obrázku 8.6. Tu lze složit ze dvou jednoduchých rotací, ale bývá přehlednější uvažovat o ní vcelku jako o „překořenění“ celé konfigurace za vrchol y.



Obrázek 8.5: Jednoduchá rotace



Obrázek 8.6: Dvojitá rotace

Vkládání do stromu

Nový prvek vložíme jako list se znaménkem 0. Tím se z prázdného podstromu hloubky -1 stal jednovrcholový podstrom hloubky 0, takže může být potřeba přepočítat znaménka na cestě ke kořeni.

Proto se budeme vracet do kořene a propagovat do vyšších pater informaci o tom, že se podstrom prohloubil. (To můžeme elegantně provést během návratu z rekurze v proceduře BvsInsert.)

Ukážeme, jak bude vypadat jeden krok. Nechť do nějakého vrcholu x přišla z jeho syna informace o prohloubení podstromu. Bez újmy na obecnosti se jednalo o levého syna; v opačném případě provedeme vše zrcadlově a prohodíme roli znamének $+$ a $-$. Rozlišíme několik případů.

Případ 1: Vrchol x měl znaménko $+$.

- Hloubka levého podstromu se právě vyrovnala s hloubkou pravého, čili znaménko x se změní na 0.
- Hloubka podstromu $T(x)$ se nezměnila, takže propagování informace ukončíme.

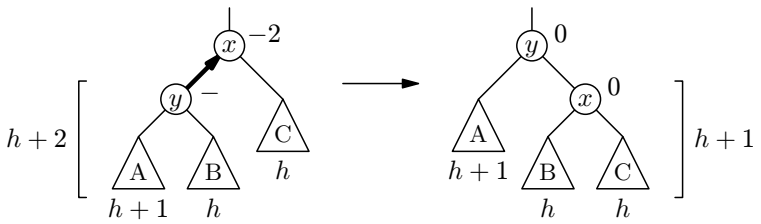
Případ 2: Vrchol x měl znaménko 0.

- Znaménko x se změní na $-$.
- Hloubka podstromu $T(x)$ vzrostla, takže v propagování musíme pokračovat.

Případ 3: Vrchol x měl znaménko $-$, tedy teď získá $\delta(v) = -2$. To definice AVL stromu nedovoluje, takže musíme strom vyvážit. Označme y vrchol, z nějž přišla informace o prohloubení, čili levého syna vrcholu x. Rozebereme případy podle jeho znaménka.

Případ 3a: Vrchol y má znaménko $-$. Situaci sledujme na obrázku.

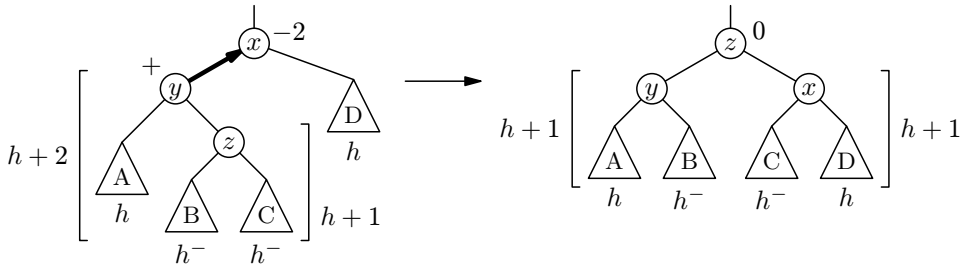
- Označíme-li h hloubku podstromu C, podstrom $T(y)$ má hloubku $h+2$, takže podstrom A má hloubku $h+1$ a podstrom B hloubku h.
- Provedeme rotaci hrany xy.
- Tím získá vrchol x znaménko 0, podstrom $T(x)$ hloubku $h+1$, vrchol y znaménko 0 a podstrom $T(y)$ hloubku $h+2$.
- Jelikož před započetím operace INSERT měl podstrom $T(x)$ hloubku $h+2$, z pohledu vyšších pater se nic nezměnilo. Propagování tedy zastavíme.



Případ 3b: Vrchol y má znaménko $+$. Sledujme opět obrázek.

- Označíme z pravého syna vrcholu y (uvědomte si, že musí existovat).
- Označíme jednotlivé podstromy tak jako na obrázku a spočítáme jejich hloubky. Referenční hloubku h zvolíme podle podstromu D. Hloubky h^- znamenají „buď h nebo $h-1$“.
- Provedeme dvojitou rotaci, která celou konfiguraci překoření za vrchol z.
- Přepočítáme hloubky a znaménka. Vrchol x bude mít znaménko buď $-$ nebo 0, vrchol y buď 0 nebo $+$, každopádně oba podstromy $T(x)$ a $T(y)$ získají hloubku $h+1$. Proto vrchol z získá znaménko 0.

- Před započtením INSERTu činila hloubka celé konfigurace $h+2$, nyní je také $h+2$, takže propagování zastavíme.



Případ 3c: Vrchol y má znaménko 0.

Tento případ je ze všech nejjednodušší – nemůže totiž nikdy nastat. Z vrcholu se znaménkem 0 se informace o prohloubení v žádném z předchozích případů nešíří.

Mazání ze stromu

Budeme postupovat obdobně jako u INSERTu: vrchol smažeme podle původního algoritmu BVSDELETE a po cestě zpět do kořene propagujeme informaci o snížení hloubky podstromu. Připomeňme, že pokaždé mažeme list nebo vrchol s jediným synem, takže stačí propagovat od místa smazaného vrcholu nahoru.

Opět popíšeme jeden krok propagování. Nechť do vrcholu x přišla informace o snížení hloubky podstromu, bez újmy na obecnosti z levého syna. Rozlišíme následující případy.

Případ 1: Vrchol x má znaménko $-$.

- Hloubka levého podstromu se právě vyrovnala s hloubkou pravého, znaménko x se mění na 0.
- Hloubka podstromu $T(x)$ se snížila, takže pokračujeme v propagování.

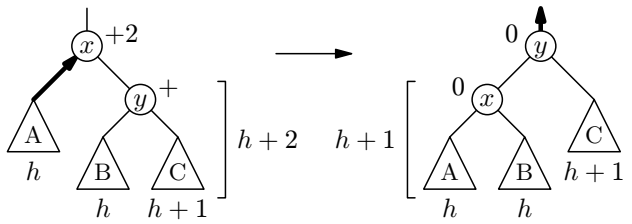
Případ 2: Vrchol x má znaménko 0.

- Znaménko x se změní na $+$.
- Hloubka podstromu $T(x)$ se nezměnila, takže propagování ukončíme.

Případ 3: Vrchol x má znaménko $+$. Tehdy se jeho znaménko změní na $+2$ a musíme vyvažovat. Rozebereme tři případy podle znaménka pravého syna y vrcholu x. (Všimněte si, že na rozdíl od vyvažování po INSERTu to musí být opačný syn než ten, ze kterého přišla informace o změně hloubky.)

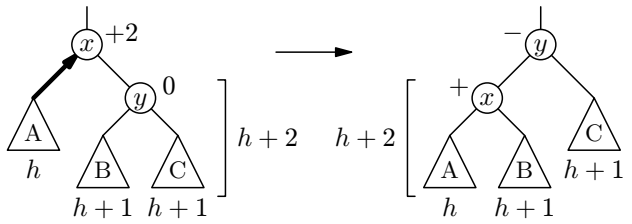
Případ 3a: Vrchol y má také znaménko $+$.

- Označíme-li h hloubku podstromu A, bude mít $T(y)$ hloubku $h+2$, takže C hloubku $h+1$ a B hloubku h.
- Provedeme rotaci hrany xy.
- Tím vrchol x získá znaménko 0, podstrom $T(x)$ hloubku $h+1$, takže vrchol y dostane také znaménko 0.
- Před započetím DELETE měl podstrom $T(x)$ hloubku $h+3$, nyní má $T(y)$ hloubku $h+2$, takže z pohledu vyšších hladin došlo ke snížení hloubky. Proto změnu propagujeme dál.



Případ 3b: Vrchol y má znaménko 0.

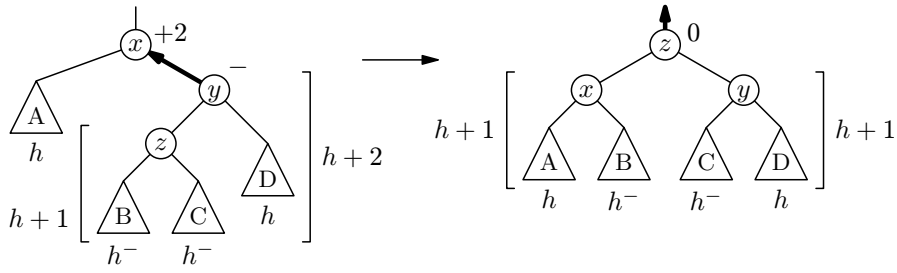
- Nechť h je hloubka podstromu A. Pak $T(y)$ má hloubku $h+2$ a B i C hloubku $h+1$.
- Provedeme rotaci hrany xy.
- Vrchol x získává znaménko $+$, podstrom $T(x)$ hloubku $h+2$, takže vrchol y obdrží znaménko $-$.
- Hloubka podstromu $T(x)$ před začátkem DELETE činila $h+3$, nyní má podstrom $T(y)$ hloubku také $h+3$, pročež propagování ukončíme.



Případ 3c: Vrchol y má znaménko $-$.

- Označíme z levého syna vrcholu y.
- Označíme podstromy podle obrázku a spočítáme jejich hloubky. Referenční hloubku h zvolíme opět podle A. Hloubky h^- znamenají „buď h nebo $h-1$“.

- Provedeme dvojitou rotaci, která celou konfiguraci překoření za vrchol z.
- Přepočítáme hloubky a znaménka. Vrchol x bude mít znaménko buď 0 nebo $-$, y buď 0 nebo $+$. Podstromy $T(y)$ a $T(x)$ budou každopádně hluboké $h+1$. Proto vrchol z obdrží znaménko 0.
- Původní hloubka podstromu $T(x)$ před začátkem DELETE činila $h+3$, nyní hloubka $T(z)$ činí $h+2$, takže propagujeme dál.



Složitost operací

Dokázali jsme, že hloubka AVL stromu je vždy $\Theta(\log n)$. Původní implementace operací BVSFIND, BVSINSERT a BVSDELETE tedy pracují v logaritmickém čase. Po BVSINSERT a BVSDELETE ještě musí následovat vyvážení, které se ovšem vždy vrací po cestě do kořene a v každém vrcholu provede $\Theta(1)$ operací, takže celkově také trvá $\Theta(\log n)$.

Cvičení

1. Dokažte, že pro minimální velikost A_k AVL stromu hloubky k platí vztah $A_k = F_{k+3} - 1$ (kde F_n je n-té Fibonacciho číslo). Z toho odvoďte přesný vzorec pro minimální a maximální možnou hloubku AVL stromu na n vrcholech. Může se hodit vztah z cvičení 1.4.4.

2. Při vyvažování po INSERTu jsme se nemuseli zabývat případem 3c proto, že z 0 se informace o prohloubení nikdy nešíří. Nemůžeme stejným způsobem dokázat, že případ 3b také nikdy nenastane? (Pozor, chyták!)

3. Upravte AVL stromy tak, aby dokázaly pro libovolné k najít k-tý nejmenší prvek. Pokud doplníte nějaké další informace do vrcholů stromu, nezapomeňte, že je musíte udržovat i při vyvažování.

4. Mějme AVL strom použitý jako slovník: v každém vrcholu sídlí klíč a nějaká celočíselná hodnota. Upravte strom, aby uměl rychle zjistit největší hodnotu přiřazenou nějakému klíči z intervalu $[a, b]$.

5.* Pokračujme v předchozím cvičení: Také chceme, aby strom uměl ve všech vrcholech s klíči v zadaném intervalu $[a, b]$ rychle zvýšit hodnoty o δ. Může se hodit princip líného vyhodnocování z oddílu 4.5.

6. AVL stromy si potřebují pamatovat v každém vrcholu znaménko. To může nabývat třech možných hodnot, takže na jeho uložení jsou potřeba dva bity. Ukažte, jak si vystačit s jediným bitem na vrchol.

8.3 Více klíčů ve vrcholech: (a,b)-stromy

Nyní prozkoumáme obecnější variantu vyhledávacích stromů, která připouští proměnlivý počet klíčů ve vrcholech. Tím si sice trochu zkomplikujeme úvahy o struktuře stromů, ale za odměnu získáme přímočařejší vyvažovací algoritmy bez složitého rozboru případů.

Definice: *Obecný vyhledávací strom* je zakořeněný strom s určeným pořadím synů každého vrcholu. Vrcholy dělíme na vnitřní a vnější, přičemž platí:

Vnitřní (interní) vrcholy obsahují libovolný nenulový počet klíčů. Pokud ve vrcholu leží klíče $x_1 < \ldots < x_k$, pak má $k + 1$ synů, které označíme $s_0, \ldots, s_k$. Klíče slouží jako oddělovače hodnot v podstromech, čili platí:

$$T(s_0) < x_1 < T(s_1) < x_2 < \ldots < x_{k-1} < T(s_{k-1}) < x_k < T(s_k),$$

kde $T(s_i)$ značí množinu všech klíčů z daného podstromu. Často se hodí dodefinovat $x_0 = -\infty$ a $x_{k+1} = +\infty$, aby nerovnost $x_i < T(s_i) < x_{i+1}$ platila i pro krajní syny.

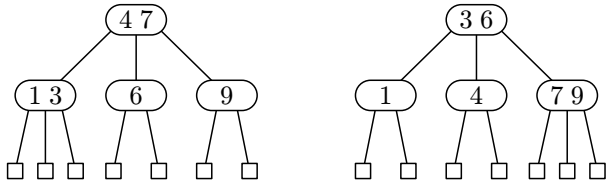
Vnější (externí) vrcholy neobsahují žádná data a nemají žádné potomky. Jsou to tedy listy stromu. Na obrázku je značíme jako malé čtverečky, v programu je můžeme reprezentovat nulovými ukazateli (`NULL` v jazyku C, `nil` v Pascalu).

Podobně jako BVS, i obecné vyhledávací stromy mohou degenerovat. Přidáme proto další podmínky pro zajištění vyváženosti.

Definice: *(a,b)-strom* pro parametry $a \geq 2$, $b \geq 2a - 1$ je obecný vyhledávací strom, pro který navíc platí:

1. Kořen má 2 až b synů, ostatní vnitřní vrcholy a až b synů.
2. Všechny vnější vrcholy jsou ve stejné hloubce.

Požadavky na a a b mohou vypadat tajemně, ale jsou snadno splnitelné a později vyplyne, proč jsme je potřebovali. Chcete-li konkrétní příklad, představujte si ten nejmenší možný: $(2, 3)$-strom. Vše ovšem budeme odvozovat obecně. Přitom budeme předpokládat, že a



Obrázek 8.7: Dva (2, 3)-stromy pro tutéž množinu klíčů

a b jsou konstanty, které se mohou „schovat do $\mathcal{O}$“. Později prozkoumáme, jaký vliv má volba těchto parametrů na vlastnosti struktury. Nyní začneme odhadem hloubky.

Lemma: (a, b)-strom s n klíči má hloubku $\Theta(\log n)$.

Důkaz: Půjdeme na to podobně jako u AVL stromů. Uvažujme, jak vypadá strom hloubky $h \geq 1$ s nejmenším možným počtem klíčů. Všechny jeho vrcholy musí mít minimální povolený počet synů (jinak by strom bylo ještě možné zmenšit). Vrcholy rozdělíme do hladin podle hloubky: na 0-té hladině je kořen se dvěma syny a jedním klíčem, úplně dole na h-té hladině leží vnější vrcholy bez klíčů. Na mezilehlých hladinách jsou všechny ostatní vnitřní vrcholy s a syny a $a - 1$ klíči.

Na i-té hladině pro $0 < i < h$ bude tedy ležet $2 \cdot a^{i-1}$ vrcholů a v nich celkem $2 \cdot a^{i-1} \cdot (a-1)$ klíčů. Sečtením přes hladiny získáme minimální možný počet klíčů m_h:

$$m_h = 1 + (a-1) \cdot \sum_{i=1}^{h-1} 2 \cdot a^{i-1} = 1 + 2 \cdot (a-1) \cdot \sum_{j=0}^{h-2} a^j.$$

Poslední sumu sečteme jako geometrickou řadu a dostaneme:

$$m_h = 1 + 2 \cdot (a-1) \cdot \frac{a^{h-1} - 1}{a-1} = 1 + 2 \cdot (a^{h-1} - 1) = 2a^{h-1} - 1.$$

Vidíme tedy, že minimální počet klíčů roste s hloubkou exponenciálně. Proto maximální hloubka musí s počtem klíčů růst nejvýše logaritmicky. (Srovnejte s výpočtem maximální hloubky AVL stromů.)

Podobně spočítáme, že maximální počet klíčů M_h roste také exponenciálně, takže minimální možná hloubka je také logaritmická. Tentokrát uvážíme strom, jehož všechny vnitřní vrcholy včetně kořene obsahují nejvyšší povolený počet $b - 1$ klíčů:

$$M_h = (b-1) \cdot \sum_{i=0}^{h-1} b^i = (b-1) \cdot \frac{b^h - 1}{b-1} = b^h - 1.$$

□

Hledání klíče

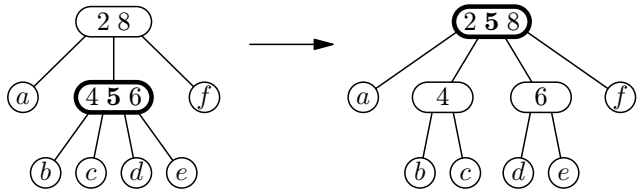
Hledání klíče v (a, b)-stromu probíhá podobně jako v BVS: začneme v kořeni a v každém vnitřním vrcholu se porovnáváním s jeho klíči rozhodneme, do kterého podstromu se vydat. Přitom buď narazíme na hledaný klíč, nebo dojdeme až do listu a tam skončíme s nepořízenou.

Vkládání do stromu

Při vkládání nejprve zkusíme nový klíč vyhledat. Pokud ve stromu ještě není přítomen, skončíme v nějakém listu. Nabízí se změnit list na vnitřní vrchol přidáním jednoho klíče a dvou listů jako synů. Tím bychom ovšem porušili axiom o stejné hloubce listů.

Raději se proto zaměříme na otce nalezeného listu a vložíme klíč do něj. To nás donutí přidat mu syna, ale jelikož ostatní synové jsou listy, tento může být též list. Pokud jsme přidáním klíče vrchol nepřeplnili (má nadále nejvýš $b-1$ klíčů), jsme hotovi.

Pakliže jsme vrchol přeplnili, rozdělíme jeho klíče mezi dva nové vrcholy, přibližně napůl. K nadřazenému vrcholu ovšem musíme místo jednoho syna připojit dva nové, takže v nadřazeném vrcholu musí přibýt klíč. Proto přeplněný vrchol raději rozdělíme na tři části: prostřední klíč, který budeme vkládat o patro výš, a levou a pravou část, z nichž se stanou nové vrcholy.



Obrázek 8.8: Štěpení přeplněného vrcholu při vkládání do (2, 3)-stromu

Tím jsme vložení klíče do aktuálního vrcholu převedli na tutéž operaci o patro výš. Tam může opět dojít k přeplnění a následnému štěpení vrcholu a tak dále, možná až do kořene. Pokud rozštěpíme kořen, vytvoříme nový kořen s jediným klíčem a dvěma syny (zde se hodí, že jsme kořeni dovolili mít méně než a synů) a celý strom se o hladinu prohloubí.

Naše ukázková implementace má podobu rekurzivní funkce AbInsert2(v, x), která dostane za úkol vložit do podstromu s kořenem v klíč x. Jako výsledek vrátí trojici (p, x', q), pokud došlo k štěpení vrcholu v na vrcholy p a q oddělené klíčem x', anebo $\emptyset$, pokud v zůstalo kořenem podstromu. Hlavní procedura AbInsert navíc ošetřuje případ štěpení kořene.

Procedura AbInsert (vkládání do (a, b)-stromu)
Vstup: Kořen stromu r, vkládaný klíč x

1. $t \leftarrow$ AbInsert2(r, x)
2. Pokud t má tvar trojice (p, x', q):
3. $\quad r \leftarrow$ nový kořen s klíčem x' a syny p a q

Výstup: Nový kořen r

Procedura AbInsert2(v, x)
Vstup: Kořen podstromu v, vkládaný klíč x

1. Pokud v je list, skončíme a vrátíme trojici (ℓ_1, x, ℓ_2), kde ℓ_1 a ℓ_2 jsou nově vytvořené listy.
2. Označíme $x_1, \ldots, x_k$ klíče ve vrcholu v a $s_0, \ldots, s_k$ jeho syny.
3. Pokud $x = x_i$ pro nějaké i, skončíme a vrátíme $\emptyset$.
4. Najdeme i tak, aby platilo $x_i < x < x_{i+1}$ ($x_0 = -\infty$, $x_{k+1} = +\infty$).
5. $t \leftarrow$ AbInsert2(s_i, x)
6. Pokud $t = \emptyset$, skončíme a také vrátíme $\emptyset$.
7. Označíme (p, x', q) složky trojice t.
8. Mezi klíče x_i a x_{i+1} vložíme klíč x'.
9. Syna s_i zrušíme a nahradíme dvojicí synů p a q.
10. Pokud počet synů nepřekročil b, skončíme a vrátíme $\emptyset$.
11. $m \leftarrow \lfloor (b-1)/2 \rfloor + 1$ ◁ *štěpení, volíme prostřední z b klíčů*
12. Vytvoříme nový vrchol v_1 s klíči $x_1, \ldots, x_{m-1}$ a syny $s_0, \ldots, s_{m-1}$.
13. Vytvoříme nový vrchol v_2 s klíči $x_{m+1}, \ldots, x_b$ a syny $s_m, \ldots, s_b$.
14. Vrátíme trojici (v_1, x_m, v_2).

Zbývá dokázat, že vrcholy vzniklé štěpením mají dostatečný počet synů. Vrchol v jsme rozštěpili v okamžiku, kdy dosáhl právě $b + 1$ synů, a tedy obsahoval b klíčů. Jeden klíč posíláme o patro výš, takže novým vrcholům v_1 a v_2 přidělíme po řadě $\lfloor (b-1)/2 \rfloor$ a $\lceil (b-1)/2 \rceil$ klíčů. Kdyby některý z nich byl „podměrečný“, muselo by platit $(b-1)/2 < a-1$, a tedy $b-1 < 2a-2$, čili $b < 2a-1$. Ejhle, podmínka na b v definici (a, b)-stromu byla zvolena přesně tak, aby této situaci zabránila.

Mazání ze stromu

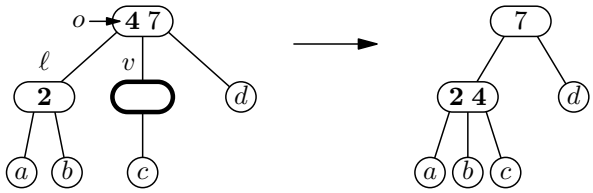
Chceme-li ze stromu smazat nějaký klíč, nejprve ho vyhledáme. Pokud se nachází na předposlední hladině (té, pod níž jsou už pouze listy), můžeme ho smazat přímo, jen musíme ošetřit případné podtečení vrcholu.

Klíče ležící na vyšších hladinách nemůžeme mazat jen tak, neboť smazáním klíče přicházíme i o místo pro připojení podstromu. To je situace podobná mazání vrcholu se dvěma syny v binárním stromu a vyřešíme ji také podobně. Mazaný klíč nahradíme jeho následníkem. To je nejnižší klíč z nejlevějšího vrcholu v pravém podstromu, který tudíž leží na předposlední hladině a může být smazán přímo.

Zbývá tedy vyřešit, co se má stát v případě, že vrchol v s a syny přijde o klíč, takže už je „pod míru“. Tehdy budeme postupovat opačně než při vkládání – pokusíme se vrchol sloučit s některým z jeho bratrů. To je ovšem možné provést pouze tehdy, když bratr také obsahuje málo klíčů; pokud jich naopak obsahuje hodně, nějaký klíč si od něj můžeme půjčit.

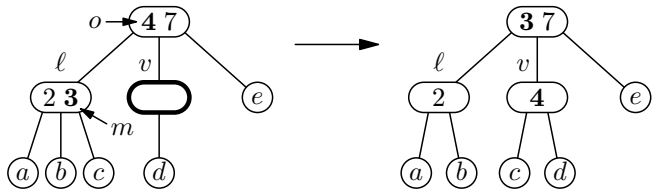
Nyní popíšeme, jak to přesně provést. Bez újmy na obecnosti předpokládejme, že vrchol v má levého bratra ℓ odděleného nějakým klíčem o v otci. Pokud by existoval pouze pravý bratr, vybereme toho a následující postup provedeme zrcadlově převráceně.

Pokud má bratr pouze a synů, sloučíme vrcholy v a ℓ do jediného vrcholu a přidáme do něj ještě klíč o z otce. Tím vznikne vrchol s $(a-2)+(a-1)+1 = 2a-2$ klíči, což není větší než $b-1$. Problém jsme tedy převedli na mazání klíče z otce, což je tentýž problém o hladinu výš.



Obrázek 8.9: Sloučení vrcholů při mazání z (2, 3)-stromu

Má-li naopak bratr více než a synů, odpojíme od něj jeho nejpravějšího syna c a největší klíč m. Poté klíč m přesuneme do otce a klíč o odtamtud přesuneme do v, kde se stane nejmenším klíčem, před který přepojíme syna c. Poté mají v i ℓ povolené počty synů a můžeme skončit. (Všimněte si, že tato operace je podobná rotaci hrany v binárním stromu.)



Obrázek 8.10: Doplnění vrcholu ve (2, 3)-stromu půjčkou od souseda

Nyní tento postup zapíšeme jako rekurzivní proceduru AbDelete2. Ta dostane kořen podstromu a klíč, který má smazat. Jako výsledek vrátí podstrom s tímtéž kořenem, ovšem možná podměrečným. Hlavní procedura AbDelete navíc ošetřuje případ, kdy z kořene zmizí všechny klíče, takže je potřeba kořen smazat a tím snížit celý strom o hladinu.

Procedura AbDelete (mazání z (a, b)-stromu)
Vstup: Kořen stromu r a mazaný klíč x

1. Zavoláme AbDelete2(r, x).
2. Pokud r má jediného syna s:
3. Zrušíme vrchol r.
4. $r \leftarrow s$

Výstup: Nový kořen r

Procedura AbDelete2
Vstup: Kořen podstromu v a mazaný klíč x

1. Označíme $x_1, \ldots, x_k$ klíče ve vrcholu v a $s_0, \ldots, s_k$ jeho syny.
2. Pokud $x = x_i$ pro nějaké i: ◁ *našli jsme*
3. Pokud s_i je list: ◁ *jsme na předposlední hladině*
4. Odstraníme z v klíč x_i a list s_i.
5. Skončíme.
6. Jinak: ◁ *jsme výš, musíme nahrazovat*
7. $m \leftarrow$ minimum podstromu s kořenem s_i
8. $x_i \leftarrow m$
9. Zavoláme AbDelete2(s_i, m).
10. Jinak: ◁ *mažeme z podstromu*
11. Najdeme i takové, aby $x_i < x < x_{i+1}$ ($x_0 = -\infty$, $x_{k+1} = +\infty$).
12. Pokud s_i je list, skončíme. ◁ *klíč ve stromu není*
13. Zavoláme AbDelete2(s_i, x).

14. ◁ *Vrátili jsme se z s_i a kontrolujeme, zda tento syn není pod míru.*
15. Pokud s_i má alespoň a synů, skončíme.
16. Je-li $i \geq 1$: ◁ *existuje levý bratr s_{i-1}*
17. Pokud má s_{i-1} alespoň $a+1$ synů: ◁ *půjčíme si klíč*
18. Odpojíme z s_{i-1} největší klíč m a nejpravějšího syna c.
19. K vrcholu s_i připojíme jako první klíč x_i a jako nejlevějšího syna c.
20. $x_i \leftarrow m$
21. Jinak: ◁ *slučujeme syny*
22. Vytvoříme nový vrchol s, který bude obsahovat všechny klíče a syny z vrcholů s_{i-1} a s_i a mezi nimi klíč x_i.
23. Z vrcholu v odstraníme klíč x_i a syny s_{i-1} a s_i. Tyto syny zrušíme a na jejich místo připojíme syna s.
24. Jinak provedeme kroky 17 až 23 zrcadlově pro pravého bratra s_{i+1} místo s_{i-1}.

Časová složitost

Pro rozbor časové složitosti předpokládáme, že parametry a a b jsou konstanty. Hledání, vkládání i mazání proto tráví na každé hladině stromu čas $\Theta(1)$ a jelikož můžeme počet hladin odhadnout jako $\Theta(\log n)$, celková časová složitost všech tří základních operací činí $\Theta(\log n)$.

Vraťme se nyní k volbě parametrů a, b. Především je známo, že se nevyplácí volit b výrazně větší než je dolní mez $2a-1$ (detaily viz cvičení 3). Proto se obvykle používají $(a, 2a-1)$-stromy, případně $(a, 2a)$-stromy. Volby $b = 2a-1$ a $b = 2a$ vedou na stejnou složitost operací v nejhorším případě, ale jak uvidíme ve cvičeních 9.3.6 a 9.3.7, vedou na úplně jiné dlouhodobé chování struktury.

Pokud chceme datovou strukturu udržovat v klasické paměti, vyplácí se volit a co nejnižší. Vhodné parametry jsou například $(2, 3)$ nebo $(2, 4)$.

Ukládáme-li data na disk, nabízí se využít toho, že je rozdělen na bloky. Přečíst celý blok je přitom zhruba stejně rychlé jako přečíst jediný byte, zatímco skok na jiný blok trvá dlouho. Proto nastavíme a tak, aby jeden vrchol stromu zabíral celý blok. Například pro disk s 4 KB bloky, 32-bitové klíče a 32-bitové ukazatele zvolíme $(256, 511)$-strom. Strom pak bude opravdu mělký: čtyři hladiny postačí pro uložení více než 33 milionů klíčů. Navíc na poslední hladině jsou pouze listy, takže při každém hledání přečteme pouhé tři bloky.

V dnešních počítačích často mezi procesorem a hlavní pamětí leží *cache* (rychlá vyrovnávací paměť), která má také blokovou strukturu s typickou velikostí bloku 64 B. Často se proto i u stromů v hlavní paměti vyplatí volit trochu větší vrcholy, aby odpovídaly

blokům cache. Pro 32-bitové klíče a 32-bitové ukazatele tedy použijeme $(4, 7)$-strom. Jen si musíme dávat pozor na správné zarovnání adres vrcholů na násobky 64 B.

Další varianty

Ve světě se lze setkat i s jinými definicemi (a, b)-stromů, než je ta naše. Často se například dělá to, že data jsou uložena pouze ve vrcholech na druhé nejnižší hladině, zatímco ostatní hladiny obsahují pouze pomocné klíče, typicky minima z podstromů. Tím si trochu zjednodušíme operace (viz cvičení 5), ale zaplatíme za to vyšší redundancí dat. Může to nicméně být šikovné, pokud potřebujeme implementovat slovník, který klíčům přiřazuje rozměrná data.

V teorii databází a souborových systémů se často hovoří o *B-stromech.* Pod tímto názvem se skrývají různé datové struktury, většinou $(a, 2a - 1)$-stromy nebo $(a, 2a)$-stromy, nezřídka v úpravě dle předchozího odstavce.

Cvičení

1. Dokažte, že procházíme-li obecný vyhledávací strom v symetrickém pořadí vrcholů, pravidelně se střídají vnitřní vrcholy s vnějšími. To znamená, že obsahují-li vnitřní vrcholy klíče $x_1, \ldots, x_n$, pak vnější vrcholy odpovídají intervalům $(-\infty, x_1)$, (x_1, x_2), (x_2, x_3), $\ldots$, $(x_n, +\infty)$.

2.* Využijte předchozí cvičení k sestrojení obecnější varianty intervalových stromů z oddílu 4.5. Hranice intervalů jsou tentokrát libovolná reálná čísla. Na počátku si strom pamatuje interval $(-\infty, +\infty)$, který pak umí v libovolném bodě podrozdělovat. Mimo to podporuje změny hodnot, intervalové dotazy a případně intervalové změny, stejně jako klasický intervalový strom.

3. Odhalte, jak závisí složitost operací s (a, b)-stromy na parametrech a a b. Z toho odvoďte, že se nikdy nevyplatí volit b výrazně větší než $2a$.

4.* Naprogramujte (a, b)-stromy a změřte, jak jsou na vašem počítači rychlé pro různé volby a a b. Projevuje se vliv cache tak, jak jsme naznačili?

5. Rozmyslete si, jak provádět operace Insert a Delete na variantě (a, b)-stromů, která ukládá užitečná data jen do nejnižších vnitřních vrcholů. Analyzujte časovou složitost a srovnejte s naší verzí struktury.

6. Ukažte, že pokud budeme do prázdného stromu postupně vkládat klíče $1, \ldots, n$, provedeme celkem $\Theta(n)$ operací. K tomu si potřebujeme pamatovat, ve kterém vrcholu skončil předchozí vložený klíč, abychom nemuseli pokaždé hledat znovu od kořene.

7. Navrhněte operaci Join(X, Y), která dostane dva (a, b)-stromy X a Y a sloučí je do jednoho. Může se přitom spolehnout na to, že všechny klíče z X jsou menší než všechny z Y. Zkuste dosáhnout složitosti $\mathcal{O}(\log |X| + \log |Y|)$.

8.* Navrhněte operaci Split(T, x), která zadaný (a, b)-strom T rozdělí na dva stromy. V jednom budou klíče menší než x, v druhém ty větší. Pokuste se o logaritmickou časovou složitost.

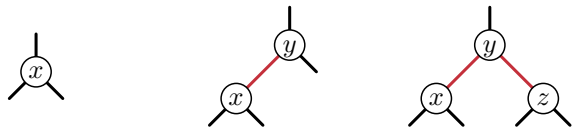
9. Nevýhodou (a, b)-stromů je, že plýtvají pamětí – může se stát, že vrcholy jsou zaplněné jen z poloviny. Navrhněte úpravu, která zaručí zaplnění z alespoň 2/3.

8.4* Červeno-černé stromy

Nyní se od obecných (a, b)-stromů vrátíme zpět ke stromům binárním. Ukážeme, jak překládat $(2, 4)$-stromy na binární stromy, čímž získáme další variantu BVS s logaritmickou hloubkou a poměrně jednoduchým vyvažováním. Říká se jí *červeno-černé stromy* (red-black trees, RB stromy). My si je předvedeme v trochu neobvyklé, ale příjemnější variantě navržené v roce 2008 Robertem Sedgewickem pod názvem left-leaning red-black trees (LLRB stromy).

Překlad bude fungovat tak, že každý vrchol $(2, 4)$-stromu nahradíme konfigurací jednoho nebo více binárních vrcholů. Aby bylo možné rekonstruovat původní $(2, 4)$-strom, rozlišíme dvě barvy hran: *červené hrany* budou spojovat vrcholy tvořící jednu konfiguraci, *černé hrany* povedou mezi konfiguracemi, čili to budou hrany původního $(2, 4)$-stromu. Barvu hrany si můžeme pamatovat například v jejím spodním vrcholu.

Strom přeložíme podle následujícího obrázku. Vrcholům $(2, 4)$-stromu budeme v závislosti na počtu synů říkat 2-vrcholy, 3-vrcholy a 4-vrcholy. 2-vrchol zůstane sám sebou. 3-vrchol nahradíme dvěma binárními vrcholy, přičemž červená hrana musí vždy vést doleva (to je ono LL v názvu LLRB stromů, obecné RB stromy nic takového nepožadují, což situaci později dost zkomplikuje). 4-vrchol nahradíme „třešničkou“ ze tří binárních vrcholů.



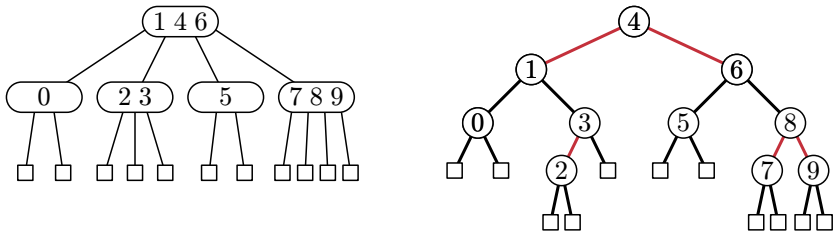
Pokud podle těchto pravidel transformujeme definici $(2, 4)$-stromu, vznikne následující definice LLRB stromu.

Definice: *LLRB strom* je binární vyhledávací strom s vnějšími vrcholy, jehož hrany jsou obarveny červeně a černě. Přitom platí následující axiomy:

1. Neexistují dvě červené hrany bezprostředně nad sebou.

2. Jestliže z vrcholu vede dolů jediná červená hrana, pak vede doleva.
3. Hrany do listů jsou vždy obarveny černě. (To se hodí, jelikož listy jsou pouze virtuální, takže do nich neumíme barvu hrany uložit.)
4. Na všech cestách z kořene do listu leží stejný počet černých hran.

Prvním dvěma axiomům budeme říkat červené, zbylým dvěma černé.



Obrázek 8.11: Překlad (2, 4)-stromu na LLRB strom

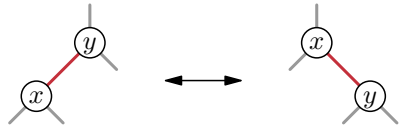
Pozorování: Z axiomů plyne, že každá konfigurace pospojovaná červenými hranami vypadá jedním z uvedených způsobů. Proto je každý LLRB strom překladem nějakého $(2, 4)$-stromu.

Důsledek: Hloubka LLRB stromu s n klíči je $\Theta(\log n)$.

Důkaz: Hloubka $(2, 4)$-stromu s n klíči činí $\Theta(\log n)$, překlad na LLRB strom počet hladin nesníží a nejvýše zdvojnásobí. □

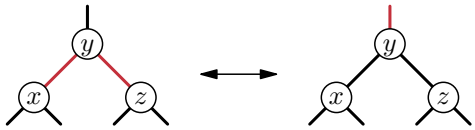
Vyvažovací operace

Operace s LLRB stromy se skládají ze dvou základních úprav. Tou první je opět rotace, ale používáme ji pouze pro červené hrany:



Rotace červené hrany zachovává nejen správné uspořádání klíčů ve vrcholech, ale i černé axiomy. Platnost červených axiomů záleží na barvách okolních hran, takže rotaci budeme muset používat opatrně. (Rotování černých hran se vyhýbáme, protože by navíc hrozilo porušení axiomu 4.)

Dále budeme používat ještě *přebarvení 4-vrcholu.* Dvojici červených hran tvořících 4-vrchol přebarvíme na černou, a naopak černou hranu vedoucí do 4-vrcholu shora přebarvíme na červenou:



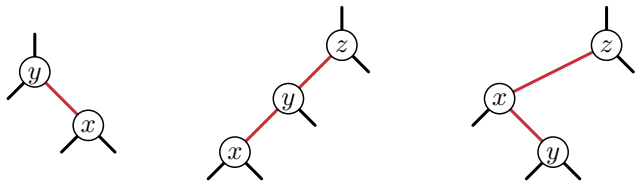
Tato úprava odpovídá rozštěpení 4-vrcholu na dva 2-vrcholy, přičemž prostřední klíč y přesouváme do nadřazeného k-vrcholu. Černé axiomy zůstanou zachovány, ale může dojít k porušení červených axiomů o patro výše.

Dodejme ještě, že přebarvení jde použít i v kořeni. Můžeme si představovat, že do kořene vede shora nějaká virtuální hrana, již můžeme bez porušení axiomů libovolně přebarvovat.

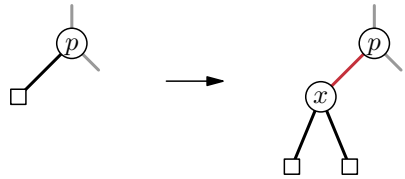
Vkládání štěpením shora dolů

Nyní popíšeme, jak se do LLRB stromu vkládá. Půjdeme na to asi takto: místo pro nový vrchol budeme hledat obvyklým způsobem, ale kdykoliv cestou potkáme 4-vrchol, rovnou ho rozštěpíme přebarvením. Až dorazíme do listu, připojíme místo něj nový vnitřní vrchol a hranu, po které jsme přišli, obarvíme červeně. Tím se nový klíč připojí k nadřazenému 2-vrcholu nebo 3-vrcholu. To zachovává černé axiomy, ale průběžně jsme porušovali ty červené, takže se budeme vracet zpět do kořene a rotacemi je opravovat.

Nyní podrobněji. Během hledání sestupujeme z kořene dolů a udržujeme invariant, že aktuální vrchol není 4-vrchol. Jakmile na nějaký 4-vrchol narazíme, přebarvíme ho. Tím se rozštěpí na dva 2-vrcholy a prostřední klíč se stane součástí nadřazeného k-vrcholu. Víme ovšem, že to nebyl 4-vrchol, takže se z něj nyní stane 3-vrchol nebo 4-vrchol. Jen možná bude nekorektně zakódovaný: 3-vrchol ve tvaru pravé odbočky nebo 4-vrchol se dvěma červenými hranami nad sebou:



Nakonec nás hledání nového klíče dovede do listu, což je místo, kam bychom klíč chtěli vložit. Nad námi leží 2-vrchol nebo 3-vrchol. List změníme na vnitřní vrchol s novým klíčem, pod něj pověsíme dva nové listy připojené černými hranami, hranu z otce přebarvíme na červenou:



Co se stane? Nový klíč leží na jediném místě, kde ležet může. Černé axiomy jsme neporušili, červené jsme opět mohli porušit vytvořením nekorektního 3-vrcholu nebo 4-vrcholu o patro výše.

Nyní se začneme vracet zpět do kořene a přitom opravovat všechna porušení červených axiomů tak, aby černé axiomy zůstaly zachovány.

Kdykoliv pod aktuálním vrcholem leží levá černá hrana a pravá červená, tak červenou hranu zrotujeme. Tím z nekorektního 3-vrcholu uděláme korektní a z nekorektního 4-vrcholu uděláme takový nekorektní, jehož obě hrany jsou levé.

Poté otestujeme, zda pod aktuálním vrcholem leží levá červená hrana do syna, který má také levou červenou hranu. Pokud ano, objevili jsme zbývající případ nekorektního 4-vrcholu, který rotací jeho horní červené hrany převedeme na korektní.

Až dojdeme do kořene, struktura opět splňuje všechny axiomy LLRB stromů.

Následuje implementace v pseudokódu. Externí vrcholy ukládáme jako konstantu $\emptyset$, barvu hran si pamatujeme v jejich spodním vrcholu.

Procedura LlrbInsert(v, x) (vkládání do LLRB stromu)
Vstup: Kořen stromu v, vkládaný klíč x

1. Pokud $v = \emptyset$, skončíme a vrátíme nově vytvořený červený vrchol v s klíčem x.
2. Pokud $x = k(v)$, skončíme (klíč x se ve stromu již nachází).
3. Jsou-li $\ell(v)$ i $r(v)$ červené, přebarvíme $\ell(v)$, $r(v)$ i v.
4. Pokud $x < k(v)$, položíme $\ell(v) \leftarrow$ LlrbInsert$(\ell(v), x)$.
5. Pokud $x > k(v)$, položíme $r(v) \leftarrow$ LlrbInsert$(r(v), x)$.
6. Je-li $\ell(v)$ černý a $r(v)$ červený, rotujeme hranu $(v, r(v))$ a do v uložíme původní $r(v)$.
7. Je-li $\ell(v)$ červený a $\ell(\ell(v))$ také červený, rotujeme hranu $(v, \ell(v))$ a do v uložíme původní $\ell(v)$.

Výstup: Nový kořen v

Vkládání štěpením zdola nahoru

Implementace vyšla překvapivě jednoduchá, ale to největší překvapení nás teprve čeká: Pokud v proceduře LlrbInsert přesuneme krok 3 za krok 7, dostaneme implementaci $(2,3)$-stromů.

Vskutku: pokud se před vkládáním prvku ve stromu nenacházel žádný 4-vrchol, nepotřebujeme štěpení 4-vrcholů cestou dolů. Nový list tedy přidáme k 2-vrcholu nebo 3-vrcholu. Pokud dočasně vznikne 4-vrchol, rozštěpíme ho cestou zpět do kořene. Tím mohou vznikat další 4-vrcholy, ale průběžně se jich zbavujeme.

Tím jsme získali kód velice podobný proceduře BvsInsert pro nevyvažované stromy, pouze si musíme dávat pozor, aby nově vzniklé vrcholy dostávaly červenou barvu a abychom před každým návratem z rekurze zavolali následující opravnou proceduru:

Procedura LlrbFixup(v)
Vstup: Kořen podstromu v

1. Je-li $\ell(v)$ černý a $r(v)$ červený, rotujeme hranu $(v, r(v))$ a do v uložíme původní $r(v)$.
2. Je-li $\ell(v)$ červený a $\ell(\ell(v))$ také červený, rotujeme hranu $(v, \ell(v))$ a do v uložíme původní $\ell(v)$.
3. Jsou-li $\ell(v)$ i $r(v)$ červené, přebarvíme $\ell(v)$, $r(v)$ i v.

Výstup: Nový kořen podstromu v

Mazání minima

Mazání bývá o trochu složitější než vkládání a LLRB stromy nejsou výjimkou. Proto si zjednodušíme práci, jak to jen půjde. Především využijeme toho, že se při vkládání umíme vyhnout 4-vrcholům, takže budeme předpokládat, že strom žádné neobsahuje. To speciálně znamená, že se nikde nevyskytuje pravá červená hrana.

Také nám situaci zjednoduší, že se voláním LlrbFixup při návratu z rekurze umíme zbavovat případných nekorektních 3-vrcholů a jakýchkoliv (potenciálně i nekorektních) 4-vrcholů. Proto nevadí, když během mazání nějaké vyrobíme.

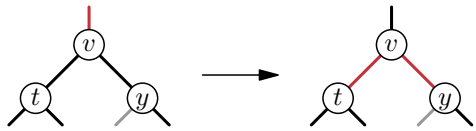
Než přikročíme k obecnému mazání, rozmyslíme si, jak smazat minimum. Najdeme ho tak, že z kořene půjdeme stále doleva, až narazíme na vrchol v, jehož levý syn je vnější. Všimněte si, že pravý syn musí být také vnější. Pokud by do v vedla shora červená hrana, mohli bychom v smazat a nahradit vnějším vrcholem. To odpovídá situací, kdy mažeme klíč z 3-vrcholu.

Horší je, jsou-li všechny hrany okolo v černé. Ve $(2,3)$-stromu jsme tedy potkali 2-vrchol, takže ho potřebujeme sloučit se sousedem, případně si od souseda půjčit klíč. Jak už se

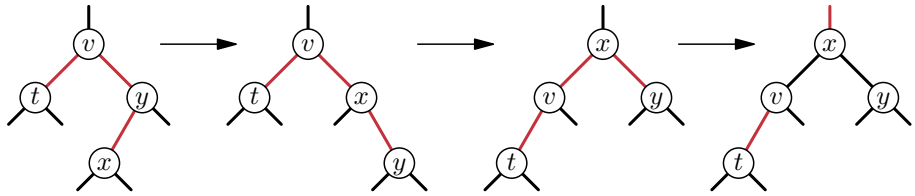
nám osvědčilo v první verzi vkládání, budeme to provádět preventivně při průchodu shora dolů, takže až opravdu dojde na mazání, žádný problém nenastane. Cestou proto budeme dodržovat:

Invariant L: Stojíme-li ve vrcholu v, pak vede červená hrana buďto shora do v, nebo z v do jeho levého syna. Výjimku dovolujeme pro kořen.

Jelikož z v pokaždé odcházíme doleva, jediný problém nastane, vede-li z v dolů levá černá hrana a pod ní je další taková. Co víme o hranách v okolí? Shora do v vede díky invariantu červená. Všechny pravé hrany jsou, jak už víme, černé. Situaci se pokusíme napravit přebarvením všech hran okolo v:



Invariant opět platí, ale pokud měl pravý syn levou červenou hranu, vyrobili jsme nekorektní 5-vrchol, navíc v místech, kudy se později nebudeme vracet. Poradíme si podle následujícího obrázku: rotací hrany yx, rotací hrany vx a nakonec přebarvením v okolí x.



Celou funkci pro nápravu invariantu můžeme napsat takto (opět předpokládáme barvy uložené ve spodních vrcholech hran):

Procedura MoveRedLeft(v)
Vstup: Kořen podstromu v

1. Přebarvíme v, $\ell(v)$ a $r(v)$.
2. Pokud je $\ell(r(v))$ červený:
3. Rotujeme hranu $(r(v), \ell(r(v)))$.
4. $x \leftarrow r(v)$
5. Rotujeme hranu (v, x).
6. Přebarvíme x, $\ell(x)$ a $r(x)$.
7. $v \leftarrow x$

Výstup: Nový kořen podstromu v

Jakmile umíme dodržet invariant, je už mazání minima snadné:

Procedura LlrbDeleteMin(v) (mazání minima z LLRB stromu)
Vstup: Kořen stromu v
1. Pokud $\ell(v) = \emptyset$, položíme $v \leftarrow \emptyset$ a skončíme.
2. Pokud $\ell(v)$ i $\ell(\ell(v))$ jsou černé:
3. $\quad v \leftarrow$ MoveRedLeft(v)
4. $\ell(v) \leftarrow$ LlrbDeleteMin($\ell(v)$)
5. $v \leftarrow$ LlrbFixup(v)

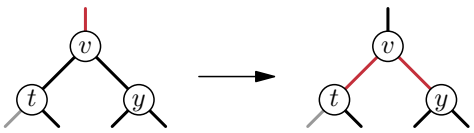
Výstup: Nový kořen v

Mazání maxima

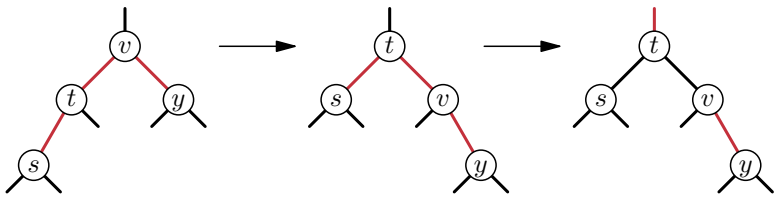
Nyní se naučíme mazat maximum. U obyčejných vyhledávacích stromů je to zrcadlová úloha k mazání minima, ne však u LLRB stromů, jejichž axiomy nejsou symetrické. Bude se každopádně hodit dodržovat stranově převrácenou obdobu předchozího invariantu:

Invariant R: Stojíme-li ve vrcholu v, pak vede červená hrana buďto shora do v, nebo z v do jeho pravého syna. Výjimku dovolujeme pro kořen.

Na cestě z kořene k maximu půjdeme stále doprava. Do pravého syna červená hrana sama od sebe nevede, ale pokud nějaká povede doleva, zrotujeme ji a tím invariant obnovíme. Problematická situace nastane, vedou-li z v dolů černé hrany a navíc z pravého syna vede doleva další černá hrana. Tehdy se inspirujeme mazáním minima a přebarvíme hrany v okolí v:



Pokud z t vede doleva černá hrana, je vše v pořádku. V opačném případě jsme nalevo vytvořili nekorektní 4-vrchol, který musíme opravit. Pomůže nám rotace hrany vt a přebarvení v okolí t:



Tato úvaha nás dovede k následující funkci pro opravu invariantu, na níž založíme celé mazání maxima.

Procedura MoveRedRight(v)
Vstup: Kořen podstromu v

1. Přebarvíme v, $\ell(v)$ a $r(v)$.
2. Pokud je $\ell(\ell(v))$ červený:
3. $\quad t \leftarrow \ell(v)$
4. $\quad$ Rotujeme hranu (v, t).
5. $\quad$ Přebarvíme t, $\ell(t)$ a $r(t)$.
6. $\quad v \leftarrow t$

Výstup: Nový kořen podstromu v

Procedura LlrbDeleteMax(v) (mazání maxima z LLRB stromu)
Vstup: Kořen stromu v

1. Pokud $\ell(v)$ je červený, rotujeme hranu $(v, \ell(v))$.
2. Pokud $r(v) = \emptyset$, položíme $v \leftarrow \emptyset$ a skončíme.
3. Pokud $r(v)$ i $\ell(r(v))$ jsou černé:
4. $\quad v \leftarrow$ MoveRedRight(v)
5. $r(v) \leftarrow$ LlrbDeleteMax($r(v)$)
6. $v \leftarrow$ LlrbFixup(v)

Výstup: Nový kořen v

Mazání obecně

Pro mazání obecného prvku nyní stačí vhodně zkombinovat myšlenky z mazání minima a maxima. Opět půjdeme shora dolů a budeme se vyhýbat tomu, abychom skončili ve 2-vrcholu. Pomůže nám k tomu tato kombinace invariantů **L** a **R**:

Invariant D: Stojíme-li ve vrcholu v, pak vede červená hrana buďto shora do v, nebo do syna, kterým se chystáme pokračovat. Výjimku dovolujeme pro kořen.

Pokud při procházení shora dolů chceme pokračovat po levé hraně, použijeme trik z mazání minima a pokud by pod námi byly dvě levé černé hrany, napravíme situaci pomocí MoveRedLeft. Naopak chceme-li odejít pravou hranou, chováme se jako při mazání maxima a v případě problémů povoláme na pomoc MoveRedRight.

Po čase najdeme vrchol, který chceme smazat. Má-li pouze vnější syny, můžeme ho přímo nahradit vnějším vrcholem. Jinak použijeme obvyklý obrat: vrchol nahradíme minimem z pravého podstromu, čímž problém převedeme na mazání minima, a to už umíme.

Procedura LlrbDelete(v, x) (mazání z LLRB stromu)
Vstup: Kořen stromu v, mazaný klíč x

1. Pokud $v = \emptyset$, vrátíme se. ◁ *klíč x ve stromu nebyl*
2. Pokud $k(v) > x$: ◁ *pokračujeme doleva jako při mazání minima*
3. Pokud $\ell(v)$ i $\ell(\ell(v))$ existují a jsou černé:
4. $v \leftarrow$ MoveRedLeft(v)
5. $\ell(v) \leftarrow$ LlrbDelete$(\ell(v), x)$
6. Jinak: ◁ *buďto hotovo, nebo doprava jako při mazání maxima*
7. Pokud $\ell(v)$ je červený, rotujeme hranu $(v, \ell(v))$.
8. Pokud $k(v) = x$ a $r(v) = \emptyset$:
9. $v \leftarrow \emptyset$ a skončíme.
10. Pokud $r(v)$ i $\ell(r(v))$ existují a jsou černé:
11. $v \leftarrow$ MoveRedRight(v)
12. Pokud $k(v) = x$:
13. Prohodíme $k(v)$ s minimem pravého podstromu $R(v)$.
14. $r(v) \leftarrow$ LlrbDeleteMin$(r(v))$
15. Jinak:
16. $r(v) \leftarrow$ LlrbDelete$(r(v), x)$
17. $v \leftarrow$ LlrbFixup(v)

Výstup: Nový kořen v

Časová složitost

Ukázali jsme tedy, jak pomocí binárních stromů kódovat $(2, 4)$-stromy, nebo dokonce $(2, 3)$-stromy. Časová složitost operací Find, Insert i Delete je zjevně lineární s hloubkou stromu a o té jsme již dokázali, že je $\Theta(\log n)$.

Dodejme na závěr, že existují i jiné varianty červeno-černých stromů, které jsou založeny na podobném překladu (a, b)-stromů na binární stromy. Některé z nich například zaručují, že při každé operaci nastane pouze $\mathcal{O}(1)$ rotací. Je to ovšem vykoupeno podstatně složitějším rozborem případů. Časová složitost samozřejmě zůstává logaritmická, protože je potřeba prvek nalézt a přebarvovat hrany.

Cvičení

1. Spočítejte přesně, jaká může být minimální a maximální hloubka LLRB stromu s n klíči.

2.* Navrhněte, jak z LLRB stromu mazat, aniž bychom museli při průchodu shora dolů rotovat. Všechny úpravy struktury provádějte až při návratu z rekurze podobně, jako se nám to podařilo při vkládání.

3. LLRB stromy jsou asymptoticky stejně rychlé jako AVL stromy. Zamyslete se nad jejich rozdíly při praktickém použití.

8.5 Další cvičení

1. Uspořádejme všechny permutace na množině $\{1, \ldots, n\}$ lexikograficky. Vymyslete algoritmus, který pro dané k sestrojí v pořadí k-tou permutaci v čase $\mathcal{O}(n \log n)$. Navrhněte též převod permutace na její pořadové číslo.

2. Vymyslete jiné uspořádání všech permutací, v němž půjde mezi permutací a jejím pořadovým číslem převádět v lineárním čase.

3. Dokažte, že budeme-li reprezentovat množiny binárními vyhledávacími stromy, nelze sjednocení provést rychleji než lineárně v nejhorším případě. Platí to dokonce i tehdy, máme-li na vstupu zaručený dokonale vyvážený strom a výstup může být jakkoliv nevyvážený.

4. *Okénkový medián:* Na vstupu postupně přicházejí čísla. Kdykoliv přijde další, vypište medián z posledních k čísel. Dosáhněte časové složitosti $\mathcal{O}(\log k)$ na operaci.

5. Dokažte, že u předchozího cvičení je čas $\Theta(\log k)$ nejlepší možný, pokud umíme čísla pouze porovnávat.

6. Sestrojte datovou strukturu pro uložení seznamu tak, abychom uměli rychle najít k-tý prvek a přesunout ho na začátek.

9 Amortizace

9 Amortizace

Při analýze datových struktur nás zatím zajímala složitost operací v nejhorším případě. Překvapivě často se ale stává, že operace s tou nejhorší časovou složitostí se vyskytují jen zřídka a většina je mnohem rychlejší. Například, dejme tomu, že jedna operace trvá $\Theta(n)$ v nejhorším případě, ale provedení libovolných m po sobě jdoucích operací se stihne za $\mathcal{O}(n+m)$. Pro m větší než n se tak operace v dlouhodobém měřítku chová, jako by měla konstantní složitost.

Tyto úvahy vedou k pojmu *amortizované časové složitosti,* který nejprve přiblížíme na několika příkladech a poté precizně nadefinujeme.

9.1 Nafukovací pole

Představme si, že nám přicházejí nějaké prvky a my je chceme postupně ukládat na konec pole: i-tý prvek na pozici i, a nevíme předem, kolik prvků má přijít. V okamžiku vytváření pole ovšem musíme vyhradit (alokovat) nějaký počet po sobě jdoucích paměťových buněk. A ať už pole vytvoříme jakkoliv velké, časem se může stát, že se do něj další prvky nevejdou.

Tehdy nám nezbude než pole zvětšit. Jenže paměťové buňky těsně za polem mohou obsahovat jiná data, takže musíme pořídit nový blok paměti, data do něj zkopírovat a starý blok paměti uvolnit. To se snadno provede,[1] ale není to zadarmo: kopírování musí sáhnout na každý prvek, tedy celkově potřebuje lineární čas. Pole proto nesmíme zvětšovat příliš často.

Osvědčený způsob je začít s jednoprvkovým polem (nebo o nějaké jiné konstantní velikosti) a kdykoliv dojde místo, zdvojnásobit velikost. Tím vznikne takzvané *nafukovací pole.*

Pseudokód pro přidání prvku bude vypadat následovně. Proměnná P bude ukazovat na adresu pole v paměti, m bude značit velikost pole (tomu budeme říkat *kapacita*), i aktuální počet prvků a x nově vkládaný prvek.

Procedura ArrayAppend(x) (přidání do nafukovacího pole)

1. Pokud $i = m$: ◁ *už na nový prvek nemáme místo*
2. $\quad m \leftarrow 2m$
3. $\quad$ Alokujeme paměť na pole P' o velikosti m.
4. $\quad$ Pro $j = 0, \ldots, i-1$: $P'[j] \leftarrow P[j]$

[1] V některých programovacích jazycích nicméně musíme dávat pozor, aby v jiných proměnných nezůstaly ukazatele na původní pozice prvků.

5. Dealokujeme paměť pole P.
6. $P \leftarrow P'$
7. $P[i] \leftarrow x$ ◁ *uložíme nový prvek*
8. $i \leftarrow i + 1$

Věta: Přidání n prvků do zpočátku prázdného nafukovacího pole trvá $\Theta(n)$.

Důkaz: Práce se strukturou sestává z vkládání jednotlivých prvků (každý v konstantním čase) proložených zvětšováním pole. Jedno zvětšování trvá čas $\Theta(i)$, celkový čas vkládání potom $\Theta(n)$. Ke zvětšování dochází právě tehdy, když je aktuální počet prvků mocnina dvojky. Všechna zvětšení dohromady tedy stojí $\Theta(2^0 + 2^1 + \ldots + 2^k)$, kde 2^k je nejvyšší mocnina dvojky menší než n. To je geometrická řada se součtem $2^{k+1} - 1 < 2n$. Celková časová složitost proto činí $\Theta(n)$. □

Ačkoliv je tedy složitost přidání jednoho prvku v nejhorším případě $\Theta(n)$, v posloupnosti operací se chová, jako kdyby byla konstantní. Budeme proto říkat, že je *amortizovaně konstantní.* Způsob, jakým jsme to spočítali, se nazývá *agregační metoda* – operace slučujeme neboli agregujeme do větších celků a pak zkoumáme chování těchto celků.

Zmenšování pole

Uvažujme nyní, že bychom mohli chtít prvky také odebírat. To se hodí třeba při implementaci zásobníku v poli. Tehdy se může stát, že nejprve přidáme spoustu prvků (čímž se pole nafoukne), načež většinu z nich zase smažeme a skončíme s obřím polem, v němž nejsou skoro žádné prvky. Mohlo by se proto hodit umět pole zase „vyfouknout“, abychom neplýtvali pamětí.

Nabízí se hlídat zaplnění pole a kdykoliv klesne pod polovinu, realokovat na poloviční velikost. To ale bude pomalé: představme si, že pole obsahovalo n prvků a mělo kapacitu $2n$. Pak jsme jeden prvek smazali, došlo k realokaci a pole nyní obsahuje $n-1$ prvků a má kapacitu n. Následně přidáme 2 prvky, ty se ale nevejdou, takže pole opět realokujeme, a teď má $n + 1$ prvků a kapacitu $2n$. Nyní 1 prvek smažeme a jsme tam, kde jsme byli. Můžeme tedy pořád dokola opakovat posloupnost 4 operací, která pokaždé vynucuje pomalou realokaci.

Problém nastal proto, že „skoro prázdné“ pole se po zmenšení okamžitě stalo „skoro plným“. Pomůže tedy oddálit od sebe meze pro zvětšování a zmenšování. Obvyklé pravidlo je *zvětšovat při přeplnění, zmenšovat při poklesu zaplnění pod čtvrtinu.* Počáteční kapacitu struktury nastavíme na 1 prvek (či jinou konstantu) a pod to ji nikdy nesnížíme.

Věta: Provedení libovolné posloupnosti n operací s nafukovacím polem, v níž se libovolně střídá přidávání a odebírání prvků, trvá $\mathcal{O}(n)$.

Důkaz: Posloupnost operací rozdělíme na bloky. Blok končí okamžikem realokace nebo koncem celé posloupnosti operací. Realokaci ještě k bloku počítáme, ale přidání prvku, které realokaci způsobilo, už patří do následujícího bloku.

V každém bloku tedy přidáváme a mažeme prvky, což stojí konstantní čas na prvek, a pak nejvýše jednou realokujeme. Dokážeme, že čas strávený realokací lze „rozúčtovat“ mezi operace zadané během bloku tak, aby na každou operaci připadl konstantní čas.

Některé bloky se chovají speciálně: V posledním bloku se vůbec nerealokuje. V prvním bloku a možná i některých dalších obsahuje pole nejvýše 1 prvek. Čas na realokaci v každém tomto bloku tedy můžeme omezit konstantou, což je též nanejvýš konstanta na operaci.

Zaměřme se nyní na některý ze zbývajících bloků. Označme p počet prvků v poli na začátku bloku. Předchozí blok skončil realokací a jelikož jak zmenšení, tak zvětšení pole ponechává přesně polovinu pole volnou, musí být aktuální kapacita pole přesně $2p$. K příští realokaci nás tedy donutí buďto nárůst počtu prvků na $2p$, anebo pokles na $p/2$. Aby se to stalo, musíme přidat alespoň p nebo ubrat alespoň $p/2$ prvků. Cenu $\Theta(p)$ za realokaci tedy můžeme rozpočítat mezi tyto operace tak, že každá přispěje konstantou. □

Tentokrát jsme použili takzvanou *účetní metodu.* Obecně spočívá v tom, že čas „přeúčtujeme“ mezi operacemi tak, aby celkový čas zůstal zachován a nová složitost každé operace vyšla nízká.

Předchozí důkaz lze také převyprávět v řeči *hustoty datové struktury.* Tak se říká podílu počtu prvků a kapacity struktury. Naše nafukovací a vyfukovací pole udržuje hustotu v intervalu $[1/4, 1]$.

Kdykoliv hustota klesne pod $1/4$, pole zmenšujeme; při nárůstu nad 1 zvětšujeme. Po každé realokaci přitom vychází hustota přesně $1/2$, takže mezi každými dvěma realokacemi se hustota změní alespoň o $1/4$. Konstantní změna hustoty přitom odpovídá lineární změně počtu prvků, takže lineární složitost realokace rozpočítáme mezi lineárně mnoho operací a amortizovaná složitost vyjde konstantní.

Zbývá drobný detail: kdykoliv je pole prázdné, třeba na začátku výpočtu, je hustota nulová, což leží mimo povolený interval. Při prázdné struktuře je ale kapacita nanejvýš 4 (maximální kapacita při 1 prvku) a tehdy mají operace konstantní složitost i v nejhorším případě. Hustotu prázdného pole tedy nemusíme uvažovat. To odpovídá výjimce pro prázdný blok v předchozím rozboru.

Další nafukovací datové struktury

Přístup zvětšování a zmenšování kapacity podle potřeby funguje i pro jiné datové struktury, jejichž kapacita musí být v okamžiku vytváření známá. Vyzkoušejme to třeba pro haldu z oddílu 4.2.

Když nám dojde kapacita haldy, pořídíme si novou, dvakrát větší haldu. Přesuneme do ní všechny prvky staré haldy a všimneme si, že jsme tím nepokazili haldové uspořádání. Stále tedy stačí, aby každý prvek na zvětšení struktury přispěl konstantním časem.

Podobně můžeme vytvářet nafukovací intervalové stromy (oddíl 4.5) nebo hešovací tabulky (11.3 a 11.4). Tam ale nestačí data zkopírovat, strukturu je potřeba znovu vybudovat. Jelikož však budování stihneme provést v lineárním čase, zamortizuje se úplně stejně jako pouhé kopírování.

Cvičení

1. Uvažujme, že bychom pole namísto zdvojnásobování zvětšovali o konstantní počet prvků. Dokažte, se tím pokazí časová složitost.

2. Jak by to dopadlo, kdybychom m-prvkové pole rovnou zvětšovali na m^2-prvkové? Počáteční velikost musíme samozřejmě zvýšit na konstantu větší než 1.

3. K dispozici jsou dva zásobníky, které podporují pouze operace PUSH (přidej na vrchol zásobníku) a POP (odeber z vrcholu zásobníku). Navrhněte algoritmus, který bude pomocí těchto dvou zásobníků simulovat frontu s operacemi ENQUEUE (přidej na konec fronty) a DEQUEUE (odeber z počátku fronty). Kromě zásobníků máte k dispozici pouze konstantní množství paměti. Ukažte, že operace s frontou budou mít amortizovaně konstantní časovou složitost.

9.2 Binární počítadlo

Další příklad, na kterém vyzkoušíme amortizovanou analýzu, je *binární počítadlo.* To si pamatuje číslo zapsané ve dvojkové soustavě a umí s ním provádět jednu jedinou operaci: INC – zvýšení o jedničku.

0
1
10
1**1**
100
10**1**
1**10**
11**1**
1000

Obrázek 9.1: Prvních 8 kroků dvojkového počítadla (změněné bity tučně)

Průběh počítání můžeme sledovat na obrázku 9.1. Pokud číslo končí nulou, Inc ji přepíše na jedničku a skončí. Končí-li jedničkami, dochází k přenosu přes tyto jedničky, takže jedničky se změní na nuly a nejbližší nula vlevo od nich na jedničku.

V pseudokódu to můžeme zapsat následovně. Číslice počítadla si budeme pamatovat v poli P, nejnižší řád bude uložený v $P[0]$, druhý nejnižší v $P[1]$, atd. Za nejvyšší jedničkou bude následovat dostatečně mnoho nul.

Algoritmus Inc (zvýšení binárního počítadla o 1)

1. $i \leftarrow 0$
2. Dokud $P[i] = 1$:
3. $\quad P[i] \leftarrow 0$
4. $\quad i \leftarrow i + 1$
5. $P[i] \leftarrow 1$

Po provedení n operací bude mít počítadlo $\ell = \lfloor \log n \rfloor$ bitů. Složitost operace Inc je lineární v počtu změněných bitů. Může tedy dosáhnout $\Theta(\ell)$, třeba pokud přecházíme z `0111...1` na `1000...0`. Překvapivě ale vyjde, že v amortizovaném smyslu je tato složitost konstantní.

Můžeme to nahlédnout agregací: nejnižší řád se mění pokaždé, druhý nejmenší v každé druhé operaci, další v každé čtvrté operaci, atd., takže celá posloupnost n operací trvá

$$\sum_{i=0}^{\ell} \left\lfloor \frac{n}{2^i} \right\rfloor \leq \sum_{i=0}^{\ell} \frac{n}{2^i} \leq n \cdot \sum_{i=0}^{\ell} \frac{1}{2^i} \leq n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n.$$

Existuje ale elegantnější a „ekonomičtější“ způsob analýzy.

Věta: Provedení n operací Inc na zpočátku nulovém počítadle trvá $\mathcal{O}(n)$.

Důkaz: Pro potřeby analýzy si představíme, že za zavolání jedné operace Inc zaplatíme dvě mince a každá z nich reprezentuje jednotkové množství času. Některé mince spotřebujeme ihned, jiné uložíme do zásoby a použijeme později.

Konkrétně budeme udržovat invariant, že ke každému jedničkovému bitu počítadla patří jeden penízek v zásobě. Třeba takto ($\star$ symbolizuje jeden penízek):

$$\begin{array}{c} \star\star\ \ \star\star\star\qquad\ \ \star\star\star\star \\ 1\,1\,0\,1\,1\,1\,0\,0\,0\,0\,1\,1\,1\,1 \end{array}$$

Nyní provedeme Inc. Dokud přepisuje jedničky na nuly, platí za to penízky uloženými u těchto jedniček. Nakonec přepíše nulu na jedničku, za což jeden ze svých penízků rovnou utratí a druhý uloží k této jedničce. Tím jakoby předplatí její budoucí vynulování.

Výsledek vypadá takto:

```
⋆⋆  ⋆⋆⋆     ⋆
1 1 0 1 1 1 0 0 0 1 0 0 0 0
```

Provedeme-li tedy n operací Inc, zaplatíme $2n$ penízků. Pomocí nich zaplatíme všechny provedené operace, ty tedy trvají $\mathcal{O}(n)$. Na konci nám nějaké penízky zbudou v zásobě, ale to jistě nevadí. □

Tomuto typu úvah se říká *penízková metoda.* Obecně ji můžeme popsat takto: Slíbíme nějakou amortizovanou časovou složitost operace vyjádřenou určitým počtem penízků. Některé penízky utratíme rovnou, jiné uložíme „na horší časy". Pozdější operace mohou tento naspořený čas využít, aby mohly běžet déle, než jsme slíbili.

Cvičení

1. Spočítejte, jak dlouho bude trvat posloupnosti n operací Inc, která začne s nenulovým stavem počítadla.
2. Rozmyslete si, že kdyby mělo binární počítadlo podporovat zároveň operace Inc a Dec (tedy zvyšení a snížení o 1), operace rozhodně nebudou mít konstantní amortizovanou složitost.
3.* Navrhněte jinou reprezentaci čísel, v níž bude možné provádět operace Inc, Dec a TestZero (zjisti, zda číslo je nulové) v amortizovaně konstantním čase.
4. Dokažte, že počítadlo v soustavě o základu k, kde $k \geq 3$ je nějaká pevná konstanta, také provádí Inc v amortizovaně konstantním čase.
5. Uvažujme místo Inc operaci Add(k), která k počítadlu přičte číslo k. Dokažte, že amortizovaná složitost této operace je $\mathcal{O}(\log k)$.
6. Použijte penízkovou metodu k analýze nafukovacího pole z minulého oddílu.

9.3 Potenciálová metoda

Potkali jsme několik příkladů, kdy amortizovaná složitost datové struktury byla mnohem lepší než její složitost v nejhorším případě. Důkazy těchto tvrzení byly založené na nějakém přerozdělování času mezi operacemi: někdy explicitně (účetní metoda), jindy tak, že jsme čas odkládali a využili později (penízková metoda). Pojďme se nyní na tento princip podívat obecněji.

Mějme nějakou datovou strukturu podporující různé operace. Uvažme libovolnou posloupnost operací $O_1, \ldots, O_m$ na této datové struktuře. *Skutečnou cenou* C_i operace O_i

nazveme, jak dlouho tato operace doopravdy trvala. Cena závisí na aktuálním stavu struktury, ale často ji postačí odhadnout shora časovou složitostí operace v nejhorším případě. Jednotky, ve kterých cenu počítáme, si přitom můžeme zvolit tak, aby nám výpočty vyšly hezky. Jen musíme zachovat, že časová složitost operace je lineární v její ceně.

Dále každé operaci stanovíme její *amortizovanou cenu* A_i. Ta vyjadřuje naše mínění o tom, jak bude tato operace přispívat k celkovému času všech operací. Smíme ji zvolit libovolně, ale součet všech amortizovaných cen musí být větší nebo roven součtu cen skutečných. Vnějšímu pozorovateli, který nevidí dovnitř výpočtu a sleduje pouze celkový čas, můžeme tvrdit, že i-tá operace trvá A_i, a on to nemůže jakkoliv vyvrátit.

Na provedení prvních i operací jsme tedy potřebovali čas $C_1 + \ldots + C_i$, ale tvrdili jsme, že to bude trvat $A_1 + \ldots + A_i$. Rozdíl těchto dvou hodnot si můžeme představit jako stav jakéhosi „bankovního účtu“ na spoření času. Obvykle se mu říká *potenciál* datové struktury a značí se $\Phi_i := (\sum_{j=1}^{i} A_j) - (\sum_{j=1}^{i} C_j)$. Před provedením první operace je přirozeně $\Phi_0 = 0$.

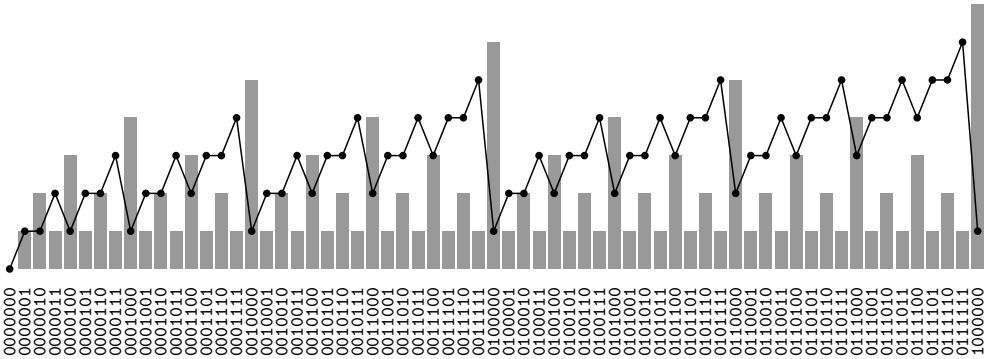
Potenciálový rozdíl $\Delta\Phi_i := \Phi_i - \Phi_{i-1}$ je pak roven $A_i - C_i$ a poznáme z něj, jestli i-tá operace čas ukládá ($\Delta\Phi_i > 0$), nebo naopak čas naspořený v minulosti spotřebovává ($\Delta\Phi_i < 0$).

Příklady:

- V úloze s binárním počítadlem odpovídá potenciál celkovému počtu naspořených penízků. Amortizovaná cena každé operace činí vždy 2 penízky, skutečnou cenu stanovíme jako $1 + j$, kde j je počet jedniček na konci čísla. Skutečná časová složitost operace je jistě lineární v této ceně. Potenciálový rozdíl vyjde $2 - (1 + j) = 1 - j$, což odpovídá tomu, že jedna jednička přibyla a j jich zmizelo. Sledujme obrázek 9.2: černá křivka zobrazuje vývoj potenciálu, šedivé sloupce skutečnou cenu jednotlivých operací.

- Při analýze nafukovacího pole se jako potenciál chová počet operací s polem od poslední realokace. Všem operacím opět přiřadíme amortizovanou cenu 2, z čehož jedničku spotřebujeme a jedničku uložíme do potenciálu. Provádíme-li realokaci, tak „rozbijeme prasátko“, vybereme z potenciálu všechen naspořený čas a nahlédneme, že postačí na provedení realokace.

V obou případech existuje přímá souvislost mezi hodnotou potenciálu a stavem či historií datové struktury. Nabízí se tedy postupovat opačně: nejprve zavést nějaký potenciál podle stavu struktury a pak podle něj vypočíst amortizovanou složitost operací. Vztah $A_i - C_i = \Delta\Phi_i = \Phi_i - \Phi_{i-1}$ totiž můžeme „obrátit naruby“: $A_i = C_i + \Delta\Phi_i$. To říká, že *amortizovaná cena je rovna součtu skutečné ceny a rozdílu potenciálů.*



Obrázek 9.2: Potenciál a skutečná cena operací u binárního počítadla

Pro součet všech amortizovaných cen pak platí:

$$\sum_{i=1}^{m} A_i = \sum_{i=1}^{m} (C_i + \Phi_i - \Phi_{i-1}) = \left(\sum_{i=1}^{m} C_i \right) + \Phi_m - \Phi_0.$$

Druhé sumě se říká *teleskopická:* každé Φ_i kromě prvního a posledního se jednou přičte a jednou odečte.[2]

Kdykoliv je tedy $\Phi_m \geq \Phi_0$, součet skutečných cen je shora omezen součtem amortizovaných cen a amortizace funguje.

Shrneme, jak se potenciálová metoda používá:

- Definujeme vhodnou potenciálovou funkci Φ v závislosti na stavu datové struktury. Na to neexistuje žádný univerzální návod, ale obvykle chceme, aby potenciál byl tím větší, čím víc se blížíme k operaci, která bude trvat dlouho.
- Ukážeme, že $\Phi_0 \leq \Phi_m$ (aneb nezůstali jsme „dlužit čas“). Často náš potenciál vyjadřuje počet něčeho ve struktuře, takže je přirozeně na počátku nulový a pak vždy nezáporný. Tehdy požadovaná nerovnost triviálně platí.
- Vypočteme amortizovanou cenu operací ze skutečné ceny a potenciálového rozdílu: $A_i = C_i + \Phi_i - \Phi_{i-1}$.

[2] Říká se jí tak podle starých dalekohledů, které se skládaly z do sebe zasunutých kovových válců. Má-li i-tý válec vnitřní poloměr r_i a vnější R_i, můžeme celkovou tloušťku válců spočítat jako $\sum_i (R_i - r_i)$, ale to je evidentně také rovno $R_n - r_1$.

- Pokud neumíme skutečnou cenu vyjádřit přesně, spokojíme se s horním odhadem. Také se může hodit upravit multiplikativní konstantu u skutečné ceny (tedy zvolit vhodnou jednotku času), aby cena lépe odpovídala zvolenému potenciálu.

Amortizovaná analýza je užitečná v případech, kdy datovou strukturu používáme uvnitř nějakého algoritmu. Tehdy nás nezajímají konkrétní časy operací, nýbrž to, jak datová struktura ovlivňuje časovou složitost celého algoritmu. Mohou se tudíž hodit i struktury, které mají špatnou časovou složitost v nejhorším případě (říká se také *worst-case složitost*), ale dobrou amortizovanou.

Jsou ovšem případy, kdy to nestačí: programujeme-li automatické řízení letadla, musíme na události reagovat okamžitě. Kdybychom reakci odložili, protože zrovna uklízíme datové struktury, program by mohl doslova spadnout.

Dodejme ještě, že bychom si neměli plést amortizovanou a průměrnou složitost. Průměr počítaný přes všechny možné vstupy nebo přes všechny možné průběhy randomizovaného algoritmu (blíže viz kapitola 11) obvykle nic neslibuje o konkrétním vstupu. Naproti tomu amortizovaná složitost nám dá spolehlivý horní odhad času pro libovolnou posloupnost operací, jen neprozradí, jak bude tento čas rozdělen mezi jednotlivé operace.

Cvičení

1. Analyzujte potenciálovou metodou amortizovanou složitost nafukovacího pole, které místo na dvojnásobek realokuje na k-násobek pro nějaké pevné $k > 1$.

2. *Okénková minima:* Na vstupu postupně přicházejí čísla. Kdykoliv přijde další, vypište minimum z posledních k čísel. Na rozdíl od cvičení 8.5.4 existuje i řešení pracující v amortizovaně konstantním čase na operaci.

3.** Vyřešte předchozí cvičení ve worst-case konstantním čase.

4. *Minimový strom* pro posloupnost $x_1, \ldots, x_n$ navzájem různých prvků je definován takto: v kořeni leží prvek x_j s nejmenší hodnotou, levý podstrom je minimovým stromem pro $x_1, \ldots, x_{j-1}$, pravý podstrom pro $x_{j+1}, \ldots, x_n$. Navrhněte algoritmus, který sestrojí minimový strom v čase $\mathcal{O}(n)$.

5.* V binárním vyhledávacím stromu budeme provádět operace Find (nalezení prvku se zadaným klíčem) a Succ (nalezení následníka prvku, který nám vrátila předchozí operace Find nebo Succ). Najděte potenciál, vůči kterému vyjde amortizovaná složitost Find $\mathcal{O}(\log n)$ a Succ $\mathcal{O}(1)$.

6. $(2,3)$-*stromy* z oddílu 8.3: Operaci Insert rozdělíme na hledání ve stromu a *strukturální změny* stromu: tím myslíme úpravy klíčů a ukazatelů uložených ve vrcholech. Ukažte, že pokud na zprvu prázdný $(2,3)$-strom aplikujeme jakoukoliv posloupnost

INSERTů, každý z nich provede amortizovaně konstantní počet strukturálních změn. Zobecněte pro libovolné (a, b)-stromy.

7.* Podobně jako v předchozím cvičení počítejme strukturální změny v (a, b)-stromu, ale tentokrát pro libovolnou kombinaci operací INSERT a DELETE. Dokažte, že ve $(2, 4)$-stromu každá taková operace provede amortizovaně $\mathcal{O}(1)$ změn, zatímco ve $(2, 3)$-stromech je jich lineárně mnoho. Výsledek pak zobecněte na $(a, 2a)$-stromy a $(a, 2a - 1)$-stromy.

9.4 Líné vyvažování stromů

Při ukládání dat do binárního vyhledávacího stromu potřebujeme strom vyvažovat, abychom udrželi logaritmickou hloubku. Potkali jsme několik vyvažovacích technik pracujících v logaritmickém čase, nicméně všechny byly dost pracné. Nyní předvedeme mnohem jednodušší variantu stromů. Složitost vyvažování sice budeme mít v nejhorším případě lineární, ale amortizovanou stále logaritmickou.

Vzpomeňme na definici dokonalé vyváženosti: ta požaduje, aby pro každý vrchol platilo, že velikosti jeho podstromů se liší nejvýše o 1. Tedy že poměr těchto velikostí je skoro přesně 1 : 1. Ukázalo se, že je to příliš přísná podmínka, takže ji nelze efektivně udržovat. Tedy ji trochu uvolníme: poměr velikostí podstromů bude ležet někde mezi 1 : 2 a 2 : 1. Totéž můžeme formulovat pomocí poměru mezi velikostí podstromů otce a synů:⟨3⟩

Definice: V binárním stromu zavedeme *mohutnost vrcholu* $m(v)$ jako počet vrcholů v podstromu zakořeněném pod v (list má tedy mohutnost 1). Strom je *v rovnováze,* pokud pro každý vrchol v a jeho syna s platí $m(s) \leq 2/3 \cdot m(v)$.

Lemma: Strom o n vrcholech, který je v rovnováze, má hloubku $\mathcal{O}(\log n)$.

Důkaz: Sledujme, jak se mění mohutnosti vrcholů na libovolné cestě z kořene do listu. Kořen má mohutnost n, každý další vrchol má mohutnost nejvýše $2/3$ předchozího, až dojdeme do listu s mohutností 1. Cesta tedy může být dlouhá nejvýše $\log_{2/3}(1/n) = \log_{3/2} n = \mathcal{O}(\log n)$. □

Nyní si rozmyslíme, jak strom udržovat v rovnováze. Abychom mohli rovnováhu kontrolovat, zapamatujeme si v každém vrcholu jeho mohutnost.

Budeme předpokládat, že do stromu pouze vkládáme nové prvky; mazání ponecháme jako cvičení. Vyjdeme z algoritmu BVSINSERT pro obyčejný vyhledávací strom. Ten se

⟨3⟩ Proč jsme nezůstali u původní definice pomocí poměru velikostí podstromů? To proto, že je-li některý z podstromů prázdný, dělili bychom nulou.

pokusí nový prvek najít a když se mu to nepovede, přidá nový list. Navíc se pak budeme vracet z přidaného listu zpět do kořene a všem vrcholům po cestě zvyšovat mohutnost o 1 (ostatním vrcholům se mohutnost evidentně nezmění).

Kdekoliv změníme mohutnost, zkontrolujeme, zda je stále splněna podmínka rovnováhy. Pokud všude je, jsme hotovi. V opačném případě nalezneme nejvyšší vrchol, v němž je porušena, a celý podstrom pod tímto vrcholem rozebereme a přebudujeme na dokonale vyvážený strom. Rozebrání a přebudování stihneme v lineárním čase (blíže viz cvičení 8.1.4).

Insert do našeho stromu se tedy nezatěžuje lokálními nepravidelnostmi, pouze udržuje rovnováhu. To je takový „líný“ přístup – dokud situace není opravdu vážná, tváříme se, jako by nic. Rovnováha nám zaručuje logaritmickou hloubku, tím pádem i logaritmickou složitost vyhledávání. A jakmile nevyváženost překročí kritickou mez, uklidíme ve stromu pořádně, což sice potrvá dlouho, ale pak zase dlouho nebudeme muset nic dělat.

Věta: Amortizovaná složitost operace Insert s líným vyvažováním je $\mathcal{O}(\log n)$.

Důkaz: Zavedeme šikovný potenciál. Měl by vyjadřovat, jak daleko jsme od dokonale vyváženého stromu: vkládáním by měl postupně růst a jakmile nějaký podstrom vyvedeme z rovnováhy, potenciál by měl být dostatečně vysoký na to, abychom z něj zaplatili přebudování podstromu.

Potenciál proto definujeme jako součet příspěvků jednotlivých vrcholů, přičemž každý vrchol přispěje rozdílem mohutností svého levého a pravého syna (chybějícím synům přisoudíme nulovou mohutnost). Budeme ovšem potřebovat, aby dokonale vyvážený podstrom měl potenciál nulový, takže přidáme výjimku: pokud se mohutnosti liší přesně o 1, příspěvek bude nula.

$$\Psi := \sum_v \psi(v), \quad \text{kde}$$
$$\psi(v) := \begin{cases} |m(\ell(v)) - m(r(v))| & \text{pokud je to alespoň 2,} \\ 0 & \text{jinak.} \end{cases}$$

Přidáme-li nový list, vrcholům na cestě mezi ním a kořenem se zvýší mohutnost o 1, tím pádem příspěvky těchto vrcholů k potenciálu se změní nejvýše o 2 (obvykle o 1, ale pokud je zrovna rozdíl vah jedničkový, příspěvek skočí z 0 rovnou na 2 či opačně).

Pokud nedošlo k žádnému přebudování, strávili jsme $\mathcal{O}(\log n)$ času průchodem cesty tam a zpět a změnili potenciál taktéž o $\mathcal{O}(\log n)$. To dává amortizovanou složitost $\mathcal{O}(\log n)$.

Nechť tedy nastane přebudování. V nějakém vrcholu v platí, že jeden ze synů, bez újmy na obecnosti $\ell(v)$, má příliš velkou mohutnost relativně k otci: $m(\ell(v)) > 2/3 \cdot m(v)$. Opačný

podstrom tedy musí mít naopak malou mohutnost, $m(r(v)) < 1/3 \cdot m(v)$. Příspěvek $\psi(v)$ proto činí aspoň $1/3 \cdot m(v)$. Tento příspěvek se přebudováním vynuluje, stejně tak příspěvky všech vrcholů ležících pod v; ostatním vrcholům se příspěvky nezmění. Potenciál tedy celkově klesne alespoň o $1/3 \cdot m(v)$.

Skutečná cena přebudování činí $\Theta(m(v))$, takže pokles potenciálu o řádově $m(v)$ ji vyrovná a amortizovaná cena přebudování vyjde nulová. (Kdybychom chtěli být přesní, vynásobili bychom potenciál vhodnou konstantou, aby pokles potenciálu přebil i konstantu z Θ.) □

Na závěr zmíníme, že myšlenku vyvažování pomocí mohutností podstromů popsal již v roce 1972 Edward Reingold. Jeho BB-α stromy byly ovšem o něco složitější a vyvažovaly se pomocí rotací.

Cvičení

1. Doplňte operaci Delete. Pro analýzu použijte tentýž potenciál.

2.* Co by se pokazilo, kdybychom v definici $\psi(v)$ neudělali výjimku pro rozdíl 1?

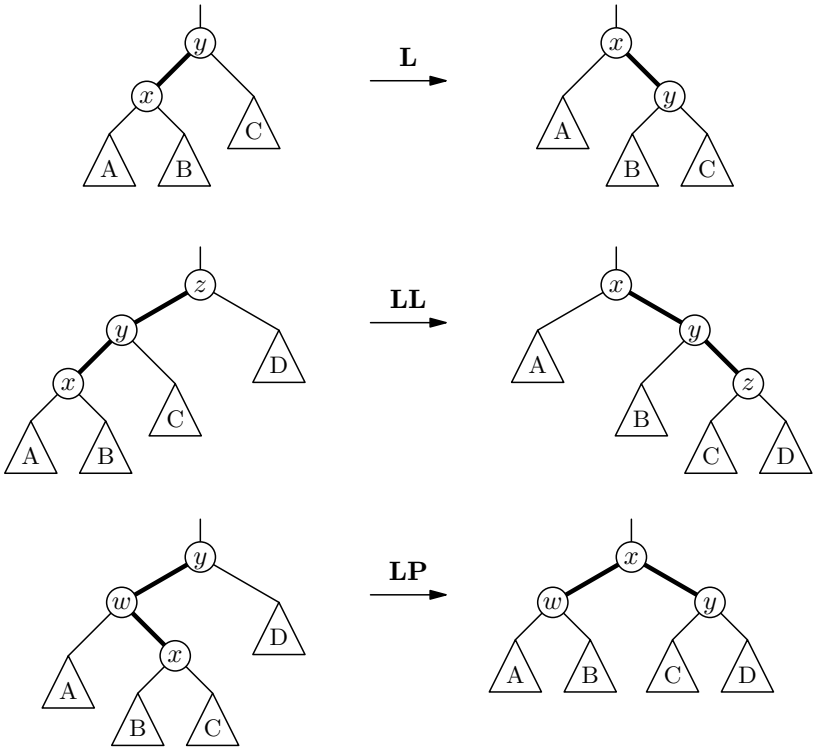
9.5* Splay stromy

Ukážeme ještě jeden pozoruhodný přístup k vyhledávacím stromům, který vede na logaritmickou amortizovanou složitost. Objevili ho v roce 1983 Daniel Sleator a Robert Tarjan. Je založený na prosté myšlence: kdykoliv chceme pracovat s nějakým vrcholem, postupnými rotacemi ho „vytáhneme“ až do kořene stromu. Této operaci budeme říkat Splay a budeme mluvit o *splayování*⟨4⟩ vrcholu.

Rotace na cestě od vybraného prvku do kořene ovšem můžeme volit více způsoby a většina z nich nevede k dobré složitosti (viz cvičení 1). Kouzlo spočívá v tom, že budeme preferovat dvojité rotace. Možné situace při splayování vrcholu x vidíme na obrázku 9.3: Je-li x levým synem levého syna, provedeme krok typu LL. Podobně krok LP pro pravého syna levého syna. Kroky PP a PL jsou zrcadlovými variantami LL a LP. Konečně pokud už x je synem kořene, provedeme jednoduchou rotaci, tedy krok L nebo jeho zrcadlovou variantu P.

Jak se z těchto kroků složí celé splayování, můžeme pozorovat na obrázku 9.4: Nejprve provedeme krok PP, pak opět PP, a nakonec P. Všimněte si, že splayování má tendenci přetvářet dlouhé cesty na rozvětvenější stromy. To nám dává naději, že nahodile vzniklé degenerované části nás nebudou dlouho brzdit.

⟨4⟩ Anglické *splay* znamená zešikmení či rozprostření. Nám však průběh operace nic takového nepřipomíná, takže raději strpíme neelegantní anglicismus.



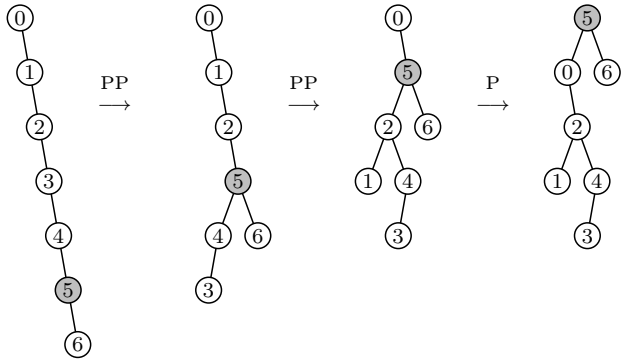
Obrázek 9.3: Splayovací kroky L, LL a LP

Pojďme se pustit do amortizované analýzy. Základem je následující potenciál. Vypadá poněkud magicky – Sleator s Tarjanem ho vytáhli jako králíka z kouzelnického klobouku, aniž by za ním byla vidět jasná intuice. Jakmile známe potenciál, zbytek už bude snadný.

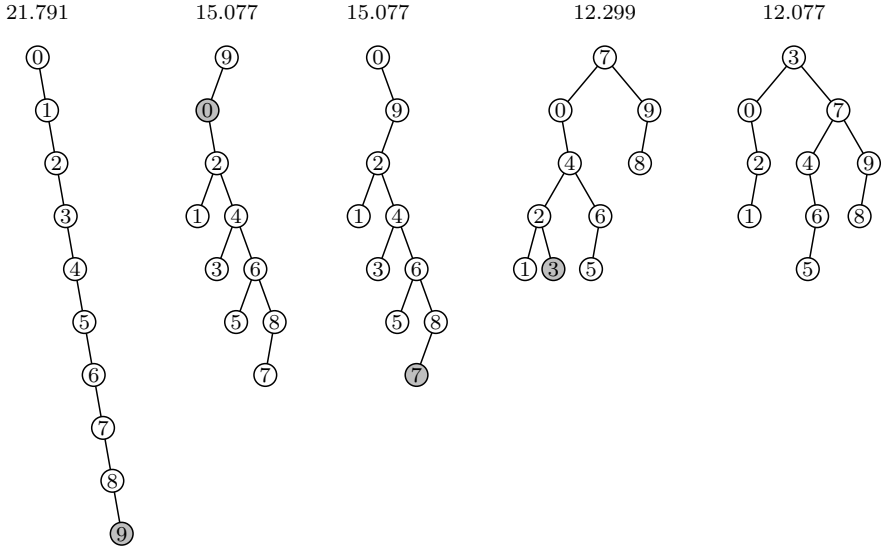
Definice:

- $T(v)$ označíme podstrom zakořeněný ve vrcholu v,
- *mohutnost vrcholu* $m(v)$ je počet vrcholů v podstromu $T(v)$,
- *rank vrcholu* $r(v)$ je dvojkový logaritmus mohutnosti $m(v)$,
- *potenciál* splay stromu je součet ranků všech vrcholů.

Obrázek 9.5 naznačuje, že vyšší potenciály opravdu odpovídají méně vyváženým stromům – cesta se postupným splayováním proměňuje na košatý strom a potenciál při tom vytrvale klesá, tím víc, čím víc práce nám splayování dalo.



Obrázek 9.4: Postup splayování vrcholu 5



Obrázek 9.5: Vývoj potenciálu během splayování hodnot 9, 0, 7, 3

Dokážeme, že tomu tak je obecně. Cenu operace Splay budeme měřit počtem provedených rotací (takže dvojitá rotace se počítá za dvě); skutečná časová složitost je zjevně lineární v této ceně. Platí následující věta:

Věta: Amortizovaná cena operace Splay(x) je nejvýše $3 \cdot (r'(x) - r(x)) + 1$, kde $r(x)$ je rank vrcholu x před provedením operace a $r'(x)$ po něm.

Nyní větu dokážeme. Čtenáři, kteří se zajímají především o další operace se splay stromy, mohou přeskočit na stranu 228 a pak se případně k důkazu vrátit.

Důkaz: Amortizovaná cena operace Splay je součtem amortizovaných cen jednotlivých kroků. Označme $r_1(x), \ldots, r_t(x)$ ranky vrcholu x po jednotlivých krocích splayování a dále $r_0(x)$ rank před prvním krokem.

V následujících lemmatech dokážeme, že cena každého kroku je shora omezena $3r_i(x) - 3r_{i-1}(x)$. Jedinou výjimku tvoří kroky L a P, které mohou být o jedničku dražší. Jelikož preferujeme dvojrotace, nastane takový krok nejvýše jednou. Pro celkovou cenu tedy dostáváme:

$$A \leq \sum_{i=1}^{t} \big(3r_i(x) - 3r_{i-1}(x)\big) + 1.$$

To je teleskopická suma: kromě r_0 a r_t se každý rank jednou přičte a jednou odečte, takže pravá strana je rovna $3r_t(x) - 3r_0(x) + 1$. To dává tvrzení věty. □

Nyní doplníme výpočty ceny jednotlivých typů kroků. Vždy budeme splayovat vrchol x, nečárkované proměnné budou odpovídat stavu před provedením kroku a čárkované stavu po něm. Nejprve ovšem dokážeme obecnou nerovnost o logaritmech.

Lemma (o průměru logaritmů): Pro každá dvě kladná reálná čísla α, β platí

$$\log \frac{\alpha + \beta}{2} \geq \frac{\log \alpha + \log \beta}{2}.$$

Důkaz: Požadovaná nerovnost platí kromě logaritmu pro libovolnou *konkávní* funkci f. Tak se říká funkcím, pro jejichž graf platí, že úsečka spojující libovolné dva body na grafu leží celá pod grafem (případně se může grafu dotýkat). Konkávnost se pozná podle záporné druhé derivace.

Pro logaritmus to jde snadno: první derivace přirozeného logaritmu $\ln x$ je $1/x$, druhá $-1/x^2$, což je záporné pro každé $x > 0$. Dvojkový logaritmus je $(\ln 2)$-násobkem přirozeného logaritmu, takže je také konkávní.

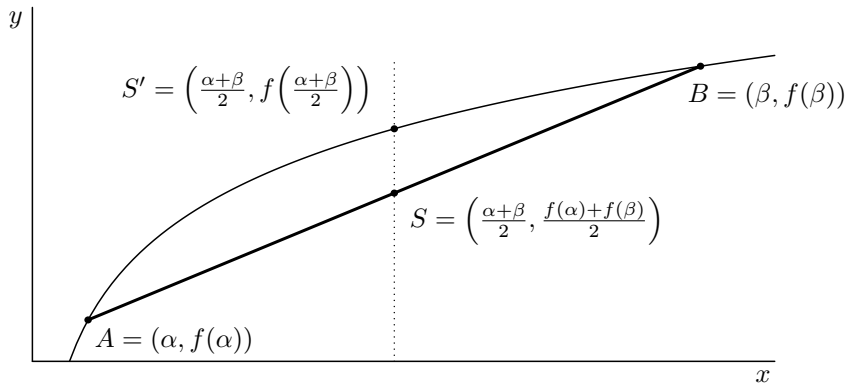
Uvažujme nyní graf nějaké konkávní funkce f na obrázku 9.6. Vyznačíme v něm body $A = (\alpha, f(\alpha))$ a $B = (\beta, f(\beta))$. Najdeme střed S úsečky AB. Jeho souřadnice jsou průměrem souřadnic krajních bodů, tedy

$$S = \left(\frac{\alpha + \beta}{2}, \frac{f(\alpha) + f(\beta)}{2}\right).$$

Díky konkávnosti musí bod S ležet pod grafem funkce, tedy speciálně pod bodem

$$S' = \left(\frac{\alpha+\beta}{2}, f\left(\frac{\alpha+\beta}{2}\right)\right).$$

Porovnáním y-ových souřadnic bodů S a S' získáme požadovanou nerovnost. □



Obrázek 9.6: Průměrová nerovnost pro konkávní funkci f

Důsledek: Jelikož $\log \frac{\alpha+\beta}{2} = \log(\alpha+\beta) - 1$, můžeme také psát $\log\alpha + \log\beta \leq 2\log(\alpha+\beta) - 2$.

Lemma LP: Amortizovaná cena kroku typu LP je nejvýše $3r'(x) - 3r(x)$.

Důkaz: Sledujme obrázek 9.3 a uvažujme, jak se změní potenciál. Jediné vrcholy, jejichž rank se může změnit, jsou w, x a y. Potenciál tedy vzroste o $(r'(w) - r(w)) + (r'(x) - r(x)) + (r'(y) - r(y))$. Skutečná cena operace činí 2 jednotky, takže pro amortizovanou cenu A platí:

$$A = 2 + r'(w) + r'(x) + r'(y) - r(w) - r(x) - r(y).$$

Chceme ukázat, že $A \leq 3r'(x) - 3r(x)$. Potřebujeme proto ranky ostatních vrcholů nějak odhadnout pomocí $r(x)$ a $r'(x)$.

Na součet $r'(w) + r'(y)$ využijeme lemma o průměru logaritmů:

$$\begin{aligned} r'(w) + r'(y) &= \log m'(w) + \log m'(y) \\ &\leq 2\log(m'(w) + m'(y)) - 2. \end{aligned}$$

Protože podstromy $T'(w)$ a $T'(y)$ jsou disjunktní a oba leží pod x, musí platit $\log(m'(w)+m'(y)) \leq \log m'(x) = r'(x)$. Celkem tedy dostáváme:

$$r'(w) + r'(y) \leq 2r'(x) - 2.$$

To dosadíme do nerovnosti pro A a získáme:

$$A \leq 3r'(x) - r(w) - r(x) - r(y).$$

Zbývající ranky můžeme odhadnout triviálně:

$$\begin{array}{ll} r(w) \geq r(x) & \text{protože } T(w) \supseteq T(x), \\ r(y) \geq r(x) & \text{protože } T(y) \supseteq T(x). \end{array}$$

Dostáváme tvrzení lemmatu. □

Lemma LL: Amortizovaná cena kroku typu LL je nejvýše $3r'(x) - 3r(x)$.

Důkaz: Budeme postupovat podobně jako u kroku LP. Skutečná cena je opět 2, ranky se mohou změnit pouze vrcholům x, y a z, takže amortizovaná cena činí

$$A = 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z).$$

Chceme se zbavit všech členů kromě $r(x)$ a $r'(x)$. Zase by se nám hodilo použít průměrové lemma na nějaké dva podstromy, které jsou disjunktní a dohromady obsahují skoro všechny vrcholy. Tentokrát se nabízí $T(x)$ a $T'(z)$:

$$\begin{aligned} r(x) + r'(z) &= \log m(x) + \log m'(z) \\ &\leq 2\log(m(x) + m'(z)) - 2 \\ &\leq 2\log m'(x) - 2 = 2r'(x) - 2. \end{aligned}$$

To je ekvivalentní s nerovností $r'(z) \leq 2r'(x) - r(x) - 2$. Tím pádem:

$$A \leq 3r'(x) + r'(y) - 2r(x) - r(y) - r(z).$$

Zbylé nežádoucí členy odhadneme elementárně:

$$\begin{array}{ll} r(z) = r'(x) & \text{protože } T(z) = T'(x), \\ r(y) \geq r(x) & \text{protože } T(y) \supseteq T(x), \\ r'(y) \leq r'(x) & \text{protože } T'(y) \subseteq T'(x). \end{array}$$

Z toho plyne požadovaná nerovnost $A \leq 3r'(x) - 3r(x)$. □

Lemma L: Amortizovaná cena kroku typu L je nejvýše $3r'(x) - 3r(x) + 1$.

Důkaz: Skutečna cena je 1, ranky se mohou měnit jen vrcholům x a y, takže amortizovaná cena vyjde:

$$A = 1 + r'(x) + r'(y) - r(x) - r(y).$$

Z inkluze podstromů plyne, že $r'(y) \leq r'(x)$ a $r(y) \geq r(x)$, takže:

$$A \leq 1 + 2r'(x) - 2r(x).$$

Z inkluze ale také víme, že $r'(x) - r(x)$ nemůže být záporné, takže tím spíš platí i $A \leq 1 + 3r'(x) - 3r(x)$, což jsme chtěli. □

Důsledek: Jelikož definice ranku je symetrická vzhledem k prohození stran, kroky typů PP, PL a P mají stejné amortizované ceny jako LL, LP a L.

Hledání podle klíče

Dokázali jsme, že amortizovaná složitost operace Splay(x) je $\mathcal{O}(r'(x) - r(x) + 1)$, kde $r(x)$ a $r'(x)$ jsou ranky vrcholu x před operací a po ní. Ranky jakožto logaritmy mohutností nikdy nepřekročí $\log n$, takže složitost evidentně leží v $\mathcal{O}(\log n)$. Nyní ukážeme, jak pomocí splayování provádět běžné operace s vyhledávacími stromy.

Operaci Find, tedy vyhledání prvku podle klíče, provedeme stejně jako v obyčejném vyhledávacím stromu a nakonec nalezený prvek vysplayujeme do kořene. Kdybychom klíč nenalezli, vysplayujeme poslední navštívený vrchol. Samotné hledání trvá lineárně s hloubkou posledního navštíveného vrcholu. Práci lineární s hloubkou ovšem vykoná i splayování, takže vychází-li amortizovaná složitost splayování $\mathcal{O}(\log n)$, musí totéž vyjít i pro hledání.

Můžeme si to představit také tak, že operaci Splay naúčtujeme i čas spotřebovaný hledáním. Tím se splayování zpomalí nejvýše konstanta-krát, takže stále bude platit amortizovaný odhad $\mathcal{O}(\log n)$.

Vkládání prvků

Operaci Insert implementujeme také obvyklým způsobem, takže nový prvek se stane listem stromu. Ten posléze vysplayujeme do kořene. Opět můžeme složitost hledání správného místa pro nový list naúčtovat provedenému splayi. Je tu ovšem drobný háček: připojení listu zvýší ranky vrcholů ležících mezi ním a kořenem, takže musíme do složitosti Insertu započítat i zvýšení potenciálu. Naštěstí je pouze logaritmické.

Lemma: Přidání listu zvýší potenciál o $\mathcal{O}(\log n)$.

Důkaz: Označíme $v_1, \ldots, v_t$ vrcholy na cestě z kořene do nového listu ℓ. Nečárkované proměnné budou jako obvykle popisovat stav před připojením listu, čárkované stav po něm. Potenciálový rozdíl činí:

$$\Delta\Phi = \sum_{i=1}^{t} \big(r'(v_i) - r(v_i)\big) + r'(\ell).$$

Jelikož ℓ je list, má jednotkovou mohutnost a nulový rank. Nový rank v_i je $r'(v_i) = \log m'(v_i) = \log(m(v_i) + 1)$, jelikož v podstromu $T(v_i)$ přibyl právě list ℓ.

Logaritmus výrazu $m(v_i) + 1$ vzdoruje pokusům o úpravu, ale můžeme ho trochu nečekaně odhadnout shora pomocí $m(v_{i-1})$. Vskutku, podstrom $T(v_{i-1})$ obsahuje vše, co leží v $T(v_i)$, a ještě navíc vrchol v_{i-1}. Proto musí platit $m(v_{i-1}) \geq m(v_i) + 1 = m'(v_i)$, a tím pádem také $r(v_{i-1}) \geq r'(v_i)$.

Tuto nerovnost dosadíme do vztahu pro potenciálový rozdíl (případ $i = 1$ jsme museli ponechat zvlášť):

$$\Delta\Phi \leq \big(r'(v_1) - r(v_1)\big) + \sum_{i=2}^{t} \big(r(v_{i-1}) - r(v_i)\big).$$

Tato suma je teleskopická – většina ranků se jednou přičte a jednou odečte. Získáme

$$\Delta\Phi \leq r'(v_1) - r(v_t),$$

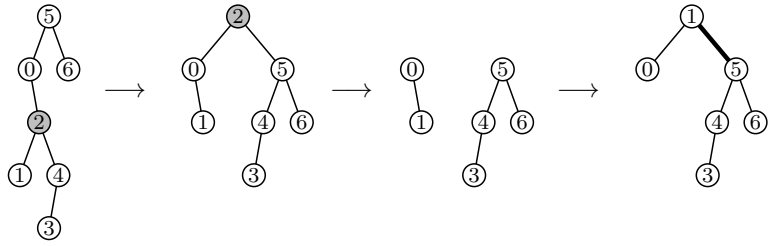
a to je jistě nejvýše logaritmické. □

Mazání prvků

Z klasických operací zbývá DELETE. Ten provedeme netradičně: nalezneme zadaný vrchol a vysplayujeme ho do kořene. Pak ho odebereme, čímž se strom rozpadne na levý a pravý podstrom. Nyní nalezneme maximum levého podstromu a opět ho vysplayujeme. Tím jsme levý strom dostali do stavu, kdy jeho maximum leží v kořeni a nemá pravého syna. Jako pravého syna mu tedy můžeme připojit kořen pravého podstromu, čímž podstromy opět spojíme.

Amortizovaná analýza se provede podobně, jen musíme pracovat s více stromy najednou. To je snadné: potenciály stromů sečteme do jednoho celkového potenciálu.

Hledání mazaného prvku a minima v podstromu naúčtujeme následujícímu splayování. Odebrání kořene potenciál sníží o rank kořene, ostatní vrcholy přispívají stále stejně.



Obrázek 9.7: Postup mazání ze splay-stromu

Nakonec spojíme dva stromy do jednoho, čímž vzroste pouze rank nového kořene, nejvýše o $\log n$.

Dokázali jsme tedy, že amortizovaná časová složitost operací Find, Insert a Delete ve splay stromu je $\mathcal{O}(\log n)$. Algoritmus mazání navíc můžeme rozdělit na operace Split (rozdělení stromu okolo daného prvku na dva) a Join (slepení dvou stromů, kde všechny prvky jednoho jsou menší než všechny prvky druhého, do jednoho stromu), obě rovněž amortizovaně logaritmické.

Zmíníme ještě jednu pozoruhodnou vlastnost splay stromů: často používané prvky mají tendenci hromadit se blízko kořene, takže přístup k nim je rychlejší než k prvkům, na které saháme zřídka. Blíže o tom ve cvičení 5.

Cvičení

1. *Naivní splayování:* Ukažte, že kdybychom splayovali pouze jednoduchými rotacemi, tedy kroky typu L a P, amortizovaná složitost operace Splay by byla lineární. Dobře je to vidět na stromech ve tvaru cesty.

2. *Splayování shora dolů:* Nevýhodou splay stromů je, že se po nalezení prvku musíme po stejné cestě ještě vrátit zpět, abychom prvek vysplayovali. Ukažte, jak splayovat už během hledání prvku, tedy shora dolů.

3.* *Sekvenční průchod:* Dokažte, že pokud budeme postupně splayovat vrcholy stromu od nejmenšího po největší, potrvá to celkem $\mathcal{O}(n)$.

4.* *Vážená analýza:* Vrcholům splay stromu přiřadíme *váhy*, což budou nějaká kladná reálná čísla. Algoritmus samotný o vahách nic neví, ale použijeme je při analýze. Mohutnost vrcholu předefinujeme na součet vah všech vrcholů v podstromu. Rank a potenciál ponecháme definovaný stejně. Dokažte, že věta o složitosti splayování stále platí. Operace Find a Delete se budou chovat podobně. Suma v rozboru Insertu ovšem přestane být teleskopická, takže vkládání implementujte také pomocí

rozdělování a spojování stromů. Pozor na to, že ranky už nemusí být logaritmické, takže složitosti budou záviset na vahách. A také pozor, že pro váhy v intervalu $(0, 1)$ mohou ranky vycházet záporné.

5.* *Náhodné přístupy:* Uvažujme splay strom, který si pamatuje prvky $x_1, \ldots, x_n$ a přicházejí na ně dotazy náhodně s pravděpodobnostmi $p_1, \ldots, p_n$, přičemž $\sum_i p_i = 1$. Využijte výsledku předchozího cvičení a dokažte, že amortizovaná složitost přístupu k x_i je $\mathcal{O}(\log(1/p_i))$. Pokuste se o srovnání se statickými optimálními vyhledávacími stromy z oddílu 12.4.

10 Rozděl a panuj

10 Rozděl a panuj

Potkáme-li spletitý problém, často pomáhá rozdělit ho na jednodušší části a s těmi se pak vypořádat postupně. Jak říkali staří Římané: *rozděl a panuj.*⟨1⟩ Tato zásada se pak osvědčila nejen ve starořímské politice, ale také o dvě tisíciletí později při návrhu algoritmů.

Nás v této kapitole bude přirozeně zajímat zejména algoritmická stránka věci. Naším cílem bude rozkládat zadaný problém na menší podproblémy a z jejich výsledků pak skládat řešení celého problému. S jednotlivými podproblémy potom naložíme stejně – opět je rozložíme na ještě menší a tak budeme pokračovat, než se dostaneme k tak jednoduchým vstupům, že je už umíme vyřešit přímo.

Myšlenka je to trochu bláznivá, ale často vede k překvapivě jednoduchému, rychlému a obvykle rekurzivnímu algoritmu. Postupně ji použijeme na třídění posloupností, násobení čísel i matic a hledání k-tého nejmenšího ze zadaných prvků.

Nejprve si tuto techniku ovšem vyzkoušíme na jednoduchém hlavolamu známém pod názvem Hanojské věže.

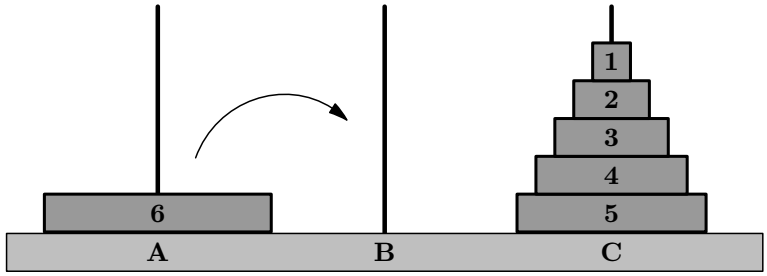
10.1 Hanojské věže

Legenda vypráví, že v daleké Hanoji stojí starobylý klášter. V jeho sklepení se skrývá rozlehlá jeskyně, v níž stojí tři sloupy. Na nich je navlečeno celkem 64 zlatých disků různých velikostí. Za úsvitu věků byly všechny disky srovnané podle velikosti na prvním sloupu: největší disk dole, nejmenší nahoře. Od té doby mniši každý den za hlaholu zvonů obřadně přenesou nejvyšší disk z některého sloupu na jiný sloup. Tradice jim přitom zakazuje položit větší disk na menší a také zopakovat již jednou použité rozmístění disků. Říká se, že až se všechny disky opět sejdou na jednom sloupu, nastane konec světa.

Nabízí se samozřejmě otázka, za jak dlouho se mnichům může podařit splnit svůj úkol a celou „věž“ z disků přenést. Zamysleme se nad tím rovnou pro obecný počet disků a očíslujme si je podle velikosti od 1 (nejmenší disk) do n (největší). Také si označme sloupy: na sloupu A budou disky na počátku, na sloup B je chceme přemístit a sloup C můžeme používat jako pomocný.

Ať už zvolíme jakýkoliv postup, někdy musíme přemístit největší disk na sloup B. V tomto okamžiku musí být na jiném sloupu samotný největší disk a všechny ostatní disky na zbývajícím sloupu (viz obrázek). Nabízí se tedy nejprve přemístit disky $1, \ldots, n-1$ z A na C, pak přesunout disk n z A na B a konečně přestěhovat disky $1, \ldots, n-1$ z C na B. Tím jsme tedy problém přesunu věže výšky n převedli na dva problémy s věží výšky $n-1$.

⟨1⟩ Oni to říkali spíš latinsky: *divide et impera*.

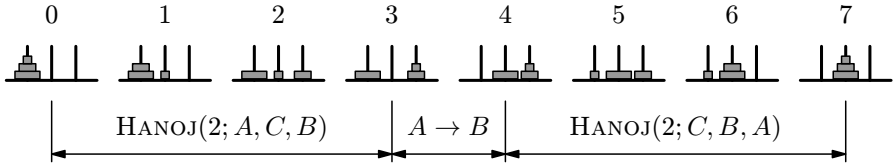


Obrázek 10.1: Stav hry při přenášení největšího disku (n = 5)

Ty ovšem můžeme vyřešit stejně, rekurzivním zavoláním téhož algoritmu. Zastavíme se až u věže výšky 1, kterou zvládneme přemístit jedním tahem. Algoritmus bude vypadat takto:

Algoritmus HANOJ
Vstup: Výška věže n; sloupy A (zdrojový), B (cílový), C (pomocný)

1. Pokud je $n = 1$, přesuneme disk 1 z A na B.
2. Jinak:
3. Zavoláme HANOJ$(n-1; A, C, B)$.
4. Přesuneme disk n z A na B.
5. Zavoláme HANOJ$(n-1; C, B, A)$.



Obrázek 10.2: Průběh algoritmu HANOJ pro n = 3

Ujistěme se, že náš algoritmus při přenášení věží neporuší pravidla. Když v kroku 3 přesouváme disk z A na B, o všech menších discích víme, že jsou na věži C, takže na ně určitě nic nepoložíme. Taktéž nikdy nepoužijeme žádnou konfiguraci dvakrát. K tomu si stačí uvědomit, že se konfigurace navštívené během obou rekurzivních volání liší polohou n-tého disku.

Spočítejme, kolik tahů náš algoritmus spotřebuje. Pokud si označíme $T(n)$ počet tahů použitý pro věž výšky n, bude platit:

$$T(1) = 1,$$
$$T(n) = 2 \cdot T(n-1) + 1.$$

Z tohoto vztahu okamžitě zjistíme, že $T(2) = 3$, $T(3) = 7$ a $T(4) = 15$. Nabízí se, že by mohlo platit $T(n) = 2^n - 1$. To snadno ověříme indukcí: Pro $n = 1$ je tvrzení pravdivé. Pokud platí pro $N - 1$, dostaneme:

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

Časová složitost algoritmu je tedy exponenciální. Ve cvičení 1 ale snadno ukážeme, že exponenciální počet tahů je nejlepší možný. Pro $n = 64$ proto mniši budou pracovat minimálně $2^{64} \approx 1.84 \cdot 10^{19}$ dní, takže konce světa se alespoň po nejbližších pár biliard let nemusíme obávat.

Cvičení

1. Dokažte, že algoritmus Hanoj je nejlepší možný, čili že $2^n - 1$ tahů je opravdu potřeba.

2. Přidejme k regulím hanojských mnichů ještě jedno pravidlo: je zakázáno přenášet disky přímo ze sloupu A na B nebo opačně (každý přesun se tedy musí uskutečnit přes sloup C). I nyní je problém řešitelný. Jak a s jakou časovou složitostí?

3. Dokažte, že algoritmus z předchozího cvičení navštíví každé korektní rozmístění disků na sloupy (tj. takové, v němž nikde neleží větší disk na menším) právě jednou.

4. Vymyslete algoritmus, který pro zadané rozmístění disků na sloupy co nejrychleji přemístí všechny disky na libovolný jeden sloup.

5.* Navrhněte takové řešení Hanojských věží, které místo rekurze bude umět z pořadového čísla tahu rovnou určit, který disk přesunout a kam.

10.2 Třídění sléváním – Mergesort

Zopakujme si, jakým způsobem jsme vyřešili úlohu z minulé kapitoly. Nejprve jsme ji rozložili na dvě úlohy menší (věže výšky $n-1$), ty jsme vyřešili rekurzivně, a pak jsme z jejich výsledků přidáním jednoho tahu utvořili výsledek úlohy původní. Podívejme se nyní, jak se podobný přístup osvědčí na problému třídění posloupnosti. Ukážeme rekurzivní verzi *třídění sléváním* – algoritmu Mergesort z oddílu 3.2.

Dostaneme-li posloupnost n prvků, jistě ji můžeme rozdělit na dvě části poloviční délky (řekněme prvních $\lfloor n/2 \rfloor$ a zbývajících $\lceil n/2 \rceil$ prvků). Ty setřídíme rekurzivním zavoláním téhož algoritmu. Setříděné poloviny posléze *slijeme* dohromady do jedné setříděné posloupnosti a máme výsledek. Když ještě ošetříme triviální případ $n = 1$, aby se nám rekurze zastavila (na to není radno zapomínat), dostaneme následující algoritmus.

Algoritmus MergeSort (rekurzivní třídění sléváním)
Vstup: Posloupnost $a_1, \ldots, a_n$ k setřídění
1. Pokud $n = 1$, vrátíme jako výsledek $b_1 = a_1$ a skončíme.
2. $x_1, \ldots, x_{\lfloor n/2 \rfloor} \leftarrow \text{MergeSort}(a_1, \ldots, a_{\lfloor n/2 \rfloor})$
3. $y_1, \ldots, y_{\lceil n/2 \rceil} \leftarrow \text{MergeSort}(a_{\lfloor n/2 \rfloor + 1}, \ldots, a_n)$
4. $b_1, \ldots, b_n \leftarrow \text{Merge}(x_1, \ldots, x_{\lfloor n/2 \rfloor}; y_1, \ldots, y_{\lceil n/2 \rceil})$

Výstup: Setříděná posloupnost $b_1, \ldots, b_n$

Procedura Merge má na vstupu dva vzestupně setříděné sousedící úseky prvků v poli a provádí samotné jejich slévání do jediného setříděného úseku. Byla popsána v oddílu 3.2 a připomeneme jen, že má lineární časovou složitost vzhledem k délce slévaných úseků a vyžaduje lineárně velkou pomocnou paměť ve formě pomocného pole.

Rozbor složitosti

Spočítejme, kolik času tříděním strávíme. Slévání ve funkci Merge má lineární časovou složitost. Složitost samotného třídění můžeme popsat takto:

$$\begin{aligned} T(1) &= 1, \\ T(n) &= 2 \cdot T(n/2) + cn. \end{aligned}$$

První rovnost nám popisuje, co se stane, když už v posloupnosti zbývá jediný prvek. Dobu trvání této operace jsme si přitom zvolili za jednotku času. Druhá rovnost pak odpovídá „zajímavé“ části algoritmu. Čas cn potřebujeme na rozdělení posloupnosti a slití setříděných kusů. Mimo to voláme dvakrát sebe sama na vstup velikosti $n/2$, což pokaždé trvá $T(n/2)$. (Zde se dopouštíme malého podvůdku a předpokládáme, že n je mocnina dvojky, takže se nemusíme starat o zaokrouhlování. V oddílu 10.4 uvidíme, že to opravdu neuškodí.)

Jak tuto rekurentní rovnici vyřešíme? Zkusme v druhém vztahu za $T(n/2)$ dosadit podle téže rovnice:

$$\begin{aligned} T(n) &= 2 \cdot (2 \cdot T(n/4) + cn/2) + cn = \\ &= 4 \cdot T(n/4) + 2cn. \end{aligned}$$

To můžeme dále rozepsat na:

$$T(n) = 4 \cdot (2 \cdot T(n/8) + cn/4) + 2cn = 8 \cdot T(n/8) + 3cn.$$

Vida, pravá strana se chová poměrně pravidelně. Můžeme obecně popsat, že po k rozepsáních dostaneme:

$$T(n) = 2^k \cdot T(n/2^k) + kcn.$$

Nyní zvolme k tak, aby $n/2^k$ bylo rovno jedné, čili $k = \log_2 n$. Dostaneme:

$$\begin{aligned} T(n) &= 2^{\log_2 n} \cdot T(1) + \log_2 n \cdot cn = \\ &= n + cn \log_2 n. \end{aligned}$$

Časová složitost Mergesortu je tedy $\Theta(n \log n)$, stejně jako u nerekurzivní verze. Jaké má paměťové nároky? Nerekurzivní Mergesort vyžadoval lineárně velké pomocné pole na slévání. Pojďme dokázat, že nám lineární množství *pomocné paměti* (to je paměť, do které nepočítáme velikost vstupu a výstupu) také úplně stačí.

Zavoláme-li funkci MergeSort na vstup velikosti n, potřebujeme si pamatovat lokální proměnné této funkce (poloviny vstupu a jejich setříděné verze – dohromady $\Theta(n)$ paměti) a pak také, kam se z funkce máme vrátit (na to potřebujeme konstantní množství paměti). Mimo to nějakou paměť spotřebují obě rekurzivní volání, ale jelikož vždy běží nejvýše jedno z nich, stačí ji započítat jen jednou. Opět dostaneme jednoduchou rekurentní rovnici:

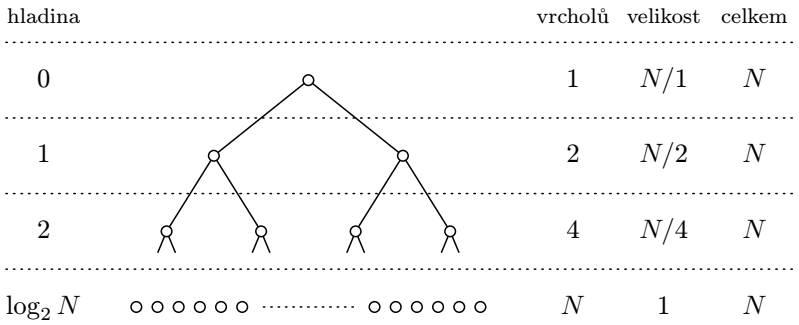
$$\begin{aligned} M(1) &= 1, \\ M(n) &= dn + M(n/2) \end{aligned}$$

pro nějakou kladnou konstantu d. To nám pro $M(n)$ dává geometrickou řadu $dn + dn/2 + dn/4 + \ldots$, která má součet $\Theta(n)$. Prostorová složitost je tedy opravdu lineární.

Stromy rekurze

Někdy je jednodušší místo počítání s rekurencemi odhadnout složitost úvahou o *stromu rekurzivních volání*. Nakreslíme strom, jehož vrcholy budou odpovídat jednotlivým podúlohám, které řešíme. Kořen je původní úloha velikosti n, jeho dva synové podúlohy velikosti $n/2$. Pak následují 4 podúlohy velikosti $n/4$, a tak dále až k listům, což jsou podúlohy o jednom prvku. Obecně i-tá hladina bude mít 2^i vrcholů pro podúlohy velikosti $n/2^i$, takže hladin bude celkem $\log_2 n$.

Rozmysleme si nyní, kolik času kde trávíme. Rozdělování i slévání jsou lineární, takže jeden vrchol na i-té hladině spotřebuje čas $\Theta(n/2^i)$. Celá i-tá hladina přispěje časem



Obrázek 10.3: Strom rekurze algoritmu MergeSort

$2^i \cdot \Theta(n/2^i) = \Theta(n)$. Když tento čas sečteme přes všechny hladiny, dostaneme celkovou časovou složitost $\Theta(n \log n)$.

Všimněte si, že tento „stromový důkaz“ docela věrně odpovídá tomu, jak jsme předtím rozepisovali rekurenci. Situace po k-tém rozepsání totiž popisuje řez stromem rekurze na k-té hladině. Vyšší hladiny jsme již sečetli, nižší nás teprve čekají.

I prostorové nároky algoritmu můžeme vyčíst ze stromu. V každém vrcholu potřebujeme paměť lineární s velikostí podúlohy, ve vrcholu na i-té hladině tedy $\Theta(n/2^i)$. V paměti je vždy vrchol, který právě zpracováváme, a všichni jeho předci. Maximálně tedy nějaká cesta z kořene do listu. Sečteme-li prostor zabraný vrcholy na takové cestě, dostaneme $\Theta(n) + \Theta(n/2) + \Theta(n/4) + \ldots + \Theta(1) = \Theta(n)$.

Cvičení

1. Naprogramujte třídění seznamu pomocí Mergesortu. Jde to snáze rekurzivně, nebo cyklem?

2. Popište třídicí algoritmus, který bude vstup rozkládat na více než dvě části a ty pak rekurzivně třídit. Může být rychlejší než náš Mergesort?

10.3 Násobení čísel – Karacubův algoritmus

Při třídění metodou Rozděl a panuj jsme získali algoritmus, který byl sice elegantnější než předchozí třídicí algoritmy, ale měl stejnou časovou složitost. Pojďme se nyní podívat na příklad, kdy nám tato metoda pomůže k efektivnějšímu algoritmu. Půjde o násobení dlouhých čísel.

Mějme n-ciferná čísla X a Y, která chceme vynásobit. Rozdělíme je na horních $n/2$ a dolních $n/2$ cifer (pro jednoduchost opět předpokládejme, že n je mocnina dvojky). Platí tedy:

$$X = A \cdot 10^{n/2} + B,$$
$$Y = C \cdot 10^{n/2} + D$$

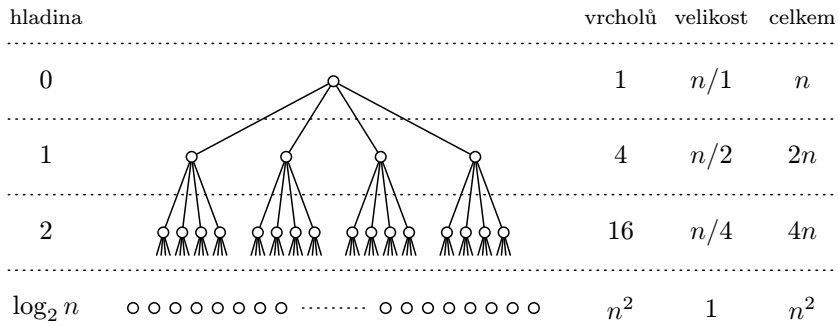
pro nějaká $(n/2)$-ciferná čísla A, B, C, D. Hledaný součin XY můžeme zapsat takto:

$$XY = AC \cdot 10^n + (AD + BC) \cdot 10^{n/2} + BD.$$

Nabízí se spočítat rekurzivně součiny AC, AD, BC a BD a pak z nich složit výsledek. Skládání obnáší několik $2n$-ciferných sčítání a několik násobení mocninou desítky, to druhé ovšem není nic jiného, než doplňování nul na konec čísla. Řešíme tedy čtyři podproblémy poloviční velikosti a k tomu spotřebujeme lineární čas. Pro časovou složitost proto platí:

$$T(1) = 1,$$
$$T(n) = 4 \cdot T(n/2) + \Theta(n).$$

Podobně jako minule, i zde k vyřešení rovnice stačí rozmyslet si, jak vypadá strom rekurzivních volání. Na jeho i-té hladině se nachází 4^i vrcholů s podproblémy o $n/2^i$ cifrách. V každém vrcholu tedy trávíme čas $\Theta(n/2^i)$ a na celé hladině $4^i \cdot \Theta(n/2^i) = \Theta(2^i \cdot n)$. Jelikož hladin je opět $\log_2 n$, strávíme jenom na poslední hladině čas $\Theta(2^{\log_2 n} \cdot n) = \Theta(n^2)$. Oproti běžnému „školnímu“ násobení jsme si tedy vůbec nepomohli.



Obrázek 10.4: První pokus o násobení rekurzí

Po počátečním neúspěchu se svého plánu nevzdáme, nýbrž nahlédneme, že ze zmíněných čtyř násobení poloviční velikosti můžeme jedno ušetřit. Když vynásobíme $(A+B)\cdot(C+D)$, dostaneme $AC + AD + BC + BD$. To se od závorky $(AD + BC)$, kterou potřebujeme, liší jen o $AC + BD$. Tyto dva členy nicméně známe, takže je můžeme odečíst. Získáme následující formulku pro XY:

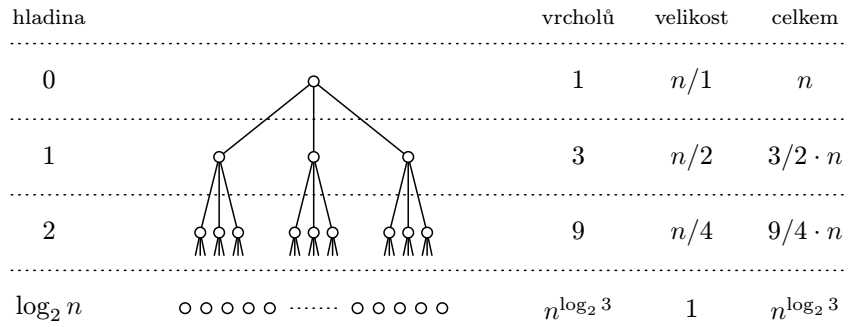
$$XY = AC \cdot 10^n + ((A + B)(C + D) - AC - BD) \cdot 10^{n/2} + BD.$$

Časová složitost se touto úpravou změní následovně:

$$T(n) = 3 \cdot T(n/2) + \Theta(n).$$

Sledujme, jak se změnil strom: na i-té hladině nalezneme 3^i vrcholů s $(n/2^i)$-ciferným problémy a jeho hloubka bude nadále činit $\log_2 n$. Na i-té hladině nyní dohromady trávíme čas $\Theta(n \cdot (3/2)^i)$, v součtu přes všechny hladiny dostaneme:

$$T(n) = \Theta(n \cdot [(3/2)^0 + (3/2)^1 + \ldots (3/2)^{\log_2 n}]).$$



Obrázek 10.5: Strom rekurze algoritmu NÁSOB

Výraz v hranatých závorkách je geometrická řada s kvocientem 3/2. Tu můžeme sečíst obvyklým způsobem na

$$\frac{(3/2)^{1+\log_2 n} - 1}{3/2 - 1}.$$

Když zanedbáme konstanty, obdržíme $(3/2)^{\log_2 n}$. To dále upravíme na

$$(2^{\log_2(3/2)})^{\log_2 n} = 2^{\log_2(3/2)\cdot\log_2 n} = (2^{\log_2 n})^{\log_2(3/2)} = n^{\log_2(3/2)} = n^{\log_2 3 - 1}.$$

Časová složitost našeho algoritmu tedy činí $\Theta(n \cdot n^{\log_2 3-1}) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$. To je již podstatně lepší než obvyklý kvadratický algoritmus. Paměti nám bude stačit lineárně mnoho (viz cvičení 3).

Algoritmus NÁSOB (násobení čísel – Karacubův algoritmus)
Vstup: n-ciferná čísla X a Y

1. Pokud $n \leq 1$, vrátíme $Z = XY$ a skončíme.
2. $k = \lfloor n/2 \rfloor$
3. $A \leftarrow \lfloor X/10^k \rfloor$, $B \leftarrow X \bmod 10^k$
4. $C \leftarrow \lfloor Y/10^k \rfloor$, $D \leftarrow Y \bmod 10^k$
5. $P \leftarrow$ NÁSOB(A, C)
6. $Q \leftarrow$ NÁSOB(B, D)
7. $R \leftarrow$ NÁSOB$(A + B, C + D)$
8. $Z \leftarrow P \cdot 10^{2k} + (R - P - Q) \cdot 10^k + Q$

Výstup: Součin $Z = XY$

Dodejme ještě, že tento princip není žádná novinka: objevil ho už v roce 1960 Anatolij Alexejevič Karacuba. Dnes jsou však známé i rychlejší metody. Ty jednodušší z nich využívají podobný princip rozkladu na podproblémy, ovšem s více částmi (cvičení 5–6). Pokročilejší algoritmy jsou pak často založeny na Fourierově transformaci, s níž se potkáme v kapitole 17. Arnold Schönhage v roce 1979 ukázal, že obdobným způsobem lze dokonce dosáhnout lineární časové složitosti. Násobení je tedy, alespoň teoreticky, stejně těžké jako sčítání a odčítání. Ve cvičeních 7–9 navíc odvodíme, že dělit lze stejně rychle jako násobit.

Cvičení

1. Pozornému čtenáři jistě neuniklo, že se v našem rozboru časové složitosti skrývá drobná chybička: čísla $A + B$ a $C + D$ mohou mít více než $n/2$ cifer, konkrétně $\lceil n/2 \rceil + 1$. Ukažte, že to časovou složitost algoritmu neovlivní.

2. Problému z předchozího cvičení se lze také vyhnout jednoduchou úpravou algoritmu. Místo $(A + B)(C + D)$ počítejte $(A - B)(C - D)$.

3. Dokažte, že funkce NÁSOB má lineární prostorovou složitost. (Podobnou úvahou jako u Mergesortu.)

4. Algoritmus NÁSOB je sice pro velká n rychlejší než školní násobení, ale pro malé vstupy se ho nevyplatí použít, protože režie na rekurzi a spojování mezivýsledků bude daleko větší než čas spotřebovaný kvadratickým algoritmem. Často proto pomůže „zkřížit“ chytrý rekurzivní algoritmus s nějakým primitivním. Pokud velikosti vstupu klesne pod vhodně zvolenou konstantu n_0, rekurzi zastavíme a použijeme hrubou sílu.

Zkuste si takový hybridní algoritmus pro násobení naprogramovat a experimentálně zjistit nejvýhodnější hodnotu hranice n_0.

5.* Zrychlete algoritmus NÁSOB tím, že budete číslo dělit na tři části a rekurzivně počítat pět součinů. Nazveme-li části čísla X po řadě X_2, X_1, X_0 a analogicky pro Y, budeme počítat tyto součiny:

$$\begin{aligned}
W_0 &= X_0 Y_0, \\
W_1 &= (X_2 + X_1 + X_0)(Y_2 + Y_1 + Y_0), \\
W_2 &= (X_2 - X_1 + X_0)(Y_2 - Y_1 + Y_0), \\
W_3 &= (4X_2 + 2X_1 + X_0)(4Y_2 + 2Y_1 + Y_0), \\
W_4 &= (4X_2 - 2X_1 + X_0)(4Y_2 - 2Y_1 + Y_0).
\end{aligned}$$

Ukažte, že součin XY lze zapsat jako lineární kombinaci těchto mezivýsledků. Jakou bude mít tento algoritmus časovou složitost?

6.** Pomocí nápovědy k předchozímu cvičení ukažte, jak pro libovolné $r \geq 1$ čísla dělit na $r + 1$ částí a rekurzivně počítat $2r + 1$ součinů. Co z toho plyne pro časovou složitost násobení? Může se hodit Kuchařková věta z následujícího oddílu.

7.* Z rychlého násobení můžeme odvodit i efektivní algoritmus pro dělení. Hodí se k tomu Newtonova iterační metoda řešení rovnic, zvaná též metoda tečen. Vyzkoušíme si ji na výpočtu n cifer podílu $1/a$ pro $2^{n-1} \leq a < 2^n$. Uvážíme funkci $f(x) = 1/x - a$. Tato funkce nabývá nulové hodnoty pro $x = 1/a$ a její derivace je $f'(x) = -1/x^2$. Budeme vytvářet posloupnost aproximací kořene této funkce. Za počáteční aproximaci x_0 zvolíme 2^{-n}, hodnotu x_{i+1} získáme z x_i tak, že sestrojíme tečnu ke grafu funkce f v bodě $(x_i, f(x_i))$ a vezmeme si x-ovou souřadnici průsečíku této tečny s osou x. Pro tuto souřadnici platí $x_{i+1} = x_i - f(x_i)/f'(x_i) = 2x_i - ax_i^2$. Posloupnost $x_0, x_1, x_2, \ldots$ velmi rychle konverguje ke kořeni $x = 1/a$ a k jejímu výpočtu stačí pouze sčítání, odčítání a násobení čísel. Rozmyslete si, jak tímto způsobem dělit libovolné číslo libovolným.

8.** Dokažte, že newtonovské dělení z minulého cvičení nalezne podíl po $\mathcal{O}(\log n)$ iteracích. Pracuje tedy v čase $\mathcal{O}(M(n) \log n)$, kde $M(n)$ je čas potřebný na vynásobení dvou n-ciferných čísel.

9.** Logaritmu v předchozím odhadu se lze zbavit, pokud funkce $M(n)$ roste alespoň lineárně, čili platí $M(cn) = \mathcal{O}(cM(n))$ pro každé $c \geq 1$. Stačí pak v k-té iteraci algoritmu počítat pouze s $2^{\Theta(k)}$-ciferými čísly, čímž složitost klesne na $\mathcal{O}(M(n) + M(n/2) + M(n/4) + \ldots) = \mathcal{O}(M(n) \cdot (1 + 1/2 + 1/4 + \ldots)) = \mathcal{O}(M(n))$. Dělení je tedy stejně těžké jako násobení.

10. *Převod mezi soustavami:* Máme n-ciferné číslo v soustavě o základu z a chceme ho převést do soustavy o jiném základu. Ukažte, jak to metodou Rozděl a panuj zvládnout v čase $\mathcal{O}(M(n))$, kde $M(n)$ je čas potřebný na násobení n-ciferných čísel v soustavě o novém základu.

10.4 Kuchařková věta o složitosti rekurzivních algoritmů

U předchozích algoritmů založených na principu Rozděl a panuj jsme pozorovali několik různých časových složitostí: $\Theta(2^n)$ u Hanojských věží, $\Theta(n \log n)$ u Mergesortu a $\Theta(n^{1.59})$ u násobení čísel. Hned se nabízí otázka, jestli v těchto složitostech lze nalézt nějaký řád. Pojďme to zkusit: Uvažme rekurzivní algoritmus, který vstup rozloží na a podproblémů velikosti n/b a z jejich výsledků složí celkovou odpověď v čase $\Theta(n^c)$.[2] Dovolíme-li si zamést zase pod rohožku případné zaokrouhlování předpokladem, že n je mocninou čísla b, bude příslušná rekurence vypadat takto:

$$\begin{aligned} T(1) &= 1, \\ T(n) &= a \cdot T(n/b) + \Theta(n^c). \end{aligned}$$

Použijeme osvědčenou metodu založenou na stromu rekurze. Jak tento strom vypadá? Každý vnitřní vrchol stromu má přesně a synů, takže na i-té hladině se nachází a^i vrcholů. Velikost problému se zmenšuje b-krát, proto na i-té hladině leží podproblémy velikosti n/b^i. Po $\log_b n$ hladinách se tudíž rekurze zastaví.

Nyní počítejme, kolik času kde strávíme. V jednom vrcholu i-té hladiny je to $\Theta((n/b^i)^c)$, na celé hladině pak $\Theta(a^i \cdot (n/b^i)^c)$. Tento výraz snadno upravíme na $\Theta(n^c \cdot (a/b^c)^i)$, což v součtu přes všechny hladiny dá:

$$\Theta(n^c \cdot [(a/b^c)^0 + (a/b^c)^1 + \ldots + (a/b^c)^{\log_b n}]).$$

Výraz v hranatých závorkách je opět nějaká geometrická řada, tentokrát s kvocientem $q = a/b^c$. Její chování bude proto záviset na tom, jak velký je kvocient:

- $q = 1$: Všechny členy řady jsou rovny jedné, takže se řada sečte na $\log_b n + 1$. Tomu odpovídá časová složitost $T(n) = \Theta(n^c \log n)$. Tak se chová například Mergesort – na všech hladinách stromu se vykonává stejné množství práce.
- $q < 1$: I kdyby řada byla nekonečná, bude mít součet nejvýše $1/(1-q)$, a to je konstanta. Dostaneme tedy $T(n) = \Theta(n^c)$. To znamená, že podstatnou část času

[2] Do tohoto schématu nám nezapadají Hanojské věže, ale ty jsou neobvyklé i tím, že jejich podproblémy jsou jen o jedničku menší než původní problém. Mimo to algoritmy s exponenciální složitostí jsou poněkud nepraktické.

trávíme v kořeni stromu a zbytek je zanedbatelný. Algoritmus tohoto typu jsme ještě nepotkali.

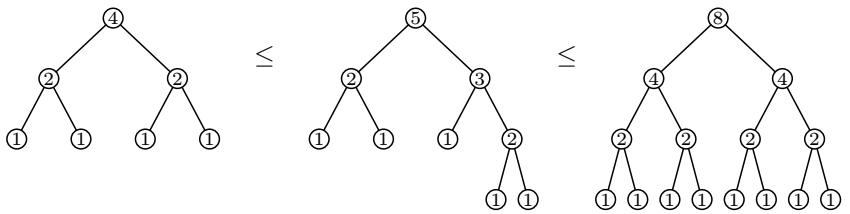
- $q > 1$: Řadu sečteme na $(q^{1+\log_b n} - 1)/(q-1) = \Theta(q^{\log_b n})$, dominantní je tentokrát čas trávený v listech. To jsme už viděli u algoritmu na násobení čísel, zkusme tedy výraz upravit obdobně:

$$q^{\log_b n} = \left(\frac{a}{b^c}\right)^{\log_b n} = \frac{a^{\log_b n}}{(b^c)^{\log_b n}} = \frac{b^{\log_b a \cdot \log_b n}}{b^{c \cdot \log_b n}} =$$

$$= \frac{(b^{\log_b n})^{\log_b a}}{(b^{\log_b n})^c} = \frac{n^{\log_b a}}{n^c}.$$

  Vyjde nám $T(n) = \Theta(n^c \cdot q^{\log_b n}) = \Theta(n^{\log_b a})$.

Zbývá maličkost: vymést zpod rohožky případ, kdy n není mocninou čísla b. Tehdy dělení na podproblémy nebude úplně rovnoměrné – některé budou mít velikost $\lfloor n/b \rfloor$, jiné $\lceil n/b \rceil$. My se ale komplikovanému počítání vyhneme následující úvahou: označme si n^- nejbližší nižší mocninu b a n^+ nejbližší vyšší. Jelikož časová složitost s rostoucím n jistě neklesá, leží $T(n)$ mezi $T(n^-)$ a $T(n^+)$. Jenže n^- a n^+ se liší jen b-krát, což se do $T(\ldots)$ ve všech třech typech chování promítne pouze konstantou. Proto jsou $T(n^-)$ i $T(n^+)$ asymptoticky stejné a taková musí být i $T(n)$.[3]



Obrázek 10.6: Obecné n jsme sevřeli mezi n^- a n^+

Zjistili jsme tedy, že hledaná funkce $T(n)$ se vždy chová jedním ze tří popsaných způsobů. To můžeme shrnout do následující „kuchařkové“ věty, známé také pod anglickým názvem *Master theorem.*

[3] To trochu připomíná „Větu o policajtech“ z matematické analýzy. Vlastně říkáme, že pokud $f(n) \leq g(n) \leq h(n)$ a existuje nějaká funkce $z(n)$ taková, že $f(n) = \Theta(z(n))$ a $h(n) = \Theta(z(n))$, pak také platí $g(n) = \Theta(z(n))$.

Věta (kuchařka na řešení rekurencí): Rekurentní rovnice $T(n) = a \cdot T(n/b) + \Theta(n^c)$, $T(1) = 1$ má pro konstanty $a \geq 1, b > 1, c \geq 0$ řešení:

- $T(n) = \Theta(n^c \log n)$, pokud $a/b^c = 1$;
- $T(n) = \Theta(n^c)$, pokud $a/b^c < 1$;
- $T(n) = \Theta(n^{\log_b a})$, pokud $a/b^c > 1$.

Cvičení

1. Nalezněte nějaký algoritmus, který odpovídá druhému typu chování ($q < 1$).

2.* Vylepšete kuchařkovou větu, aby pokrývala i případy, v nichž se velikosti podproblémů liší až o nějakou konstantu. To by se hodilo například u násobení čísel.

3.** *Kuchařka pro různě hladové jedlíky:* Jak by věta vypadala, kdybychom problém dělili na nestejně velké části? Tedy kdyby rekurence měla tvar $T(n) = T(\beta_1 n) + T(\beta_2 n) + \ldots + T(\beta_a n) + \Theta(n^c)$.

4. Řešte „nekuchařkovou“ rekurenci $T(n) = 2T(n/2) + \Theta(n \log n)$, $T(1) = 1$.

5. Jiná „nekuchařková“ rekurence: $T(n) = n^{1/2} \cdot T(n^{1/2}) + \Theta(n)$, $T(1) = 1$.

10.5 Násobení matic – Strassenův algoritmus

Nejen násobením čísel živ jest matematik. Často je potřeba násobit i složitější matematické objekty, zejména pak čtvercové matice. Pokud počítáme součin dvou matic tvaru $n \times n$ přesně podle definice, potřebujeme $\Theta(n^3)$ kroků. Jak v roce 1969 ukázal Volker Strassen, i zde dělení na menší podproblémy přináší ovoce v podobě rychlejšího algoritmu.

Nejprve si rozmyslíme, že stačí umět násobit matice, jejichž velikost je mocnina dvojky. Jinak stačí matice doplnit vpravo a dole nulami a nahlédnout, že vynásobením takto orámovaných matic získáme stejným způsobem orámovaný součin původních matic. Navíc orámované matice obsahují nejvýše čtyřikrát tolik prvků, takže se nemusíme obávat, že bychom tím algoritmus podstatně zpomalili.

Mějme tedy matice X a Y, obě tvaru $n \times n$ pro $n = 2^k$. Rozdělíme je na čtvrtiny – *bloky* tvaru $n/2 \times n/2$. Bloky matice X označíme A až D, bloky matice Y nazveme P až S. Pomocí těchto bloků můžeme snadno zapsat jednotlivé bloky součinu $X \cdot Y$:

$$X \cdot Y = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} P & Q \\ R & S \end{pmatrix} = \begin{pmatrix} AP + BR & AQ + BS \\ CP + DR & CQ + DS \end{pmatrix}.$$

Tento vztah vlastně vypadá úplně stejně jako klasická definice násobení matic, jen zde jednotlivá písmena nezastupují čísla, nýbrž bloky.

Jedno násobení matic $n \times n$ jsme tedy převedli na 8 násobení matic poloviční velikosti a režii $\Theta(n^2)$. Letmým nahlédnutím do kuchařkové věty z minulé kapitoly zjistíme, že vznikne opět kubický algoritmus. (Pozor, obvyklá terminologie je tu poněkud zavádějící – n zde neznačí velikost vstupu, nýbrž počet řádků matice; vstup je tedy velký n^2.)

Stejně jako u násobení čísel nás zachrání, že dovedeme jedno násobení ušetřit. Jen příslušné formule jsou daleko komplikovanější a připomínají králíka vytaženého z klobouku. Neprozradíme vám, jak kouzelník pan Strassen svůj trik vymyslel (sami neznáme žádný systematický postup, jak na to přijít), ale když už vzorce známe, není těžké ověřit, že opravdu fungují (viz cvičení). Formulky vypadají takto:

$$X \cdot Y = \begin{pmatrix} T_1 + T_4 - T_5 + T_7 & T_3 + T_5 \\ T_2 + T_4 & T_1 - T_2 + T_3 + T_6 \end{pmatrix},$$

kde:

$$\begin{aligned} T_1 &= (A + D) \cdot (P + S) & T_5 &= (A + B) \cdot S \\ T_2 &= (C + D) \cdot P & T_6 &= (C - A) \cdot (P + Q) \\ T_3 &= A \cdot (Q - S) & T_7 &= (B - D) \cdot (R + S) \\ T_4 &= D \cdot (R - P) \end{aligned}$$

Stačí nám tedy provést 7 násobení menších matic a 18 maticových součtů a rozdílů. Součty a rozdíly umíme počítat v čase $\Theta(n^2)$, takže časovou složitost celého algoritmu bude popisovat rekurence $T(n) = 7T(n/2)+\Theta(n^2)$. Podle kuchařkové věty je jejím řešením $T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$.

Pro úplnost dodejme, že jsou známy i efektivnější algoritmy, které jsou ovšem mnohem složitější a vyplatí se je používat až pro opravdu obří matice. Nejrychlejší z nich (Le Gallův z roku 2014) dosahuje složitosti cca $\Theta(n^{2.373})$ a obecně se soudí, že k $\Theta(n^2)$ se lze libovolně přiblížit.

Cvičení

1. Dokažte Strassenovy vzorce. Návod:

$$T_1 = \begin{bmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{bmatrix} \qquad T_4 = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & + & \cdot \end{bmatrix} \qquad T_5 = \begin{bmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix} \qquad T_7 = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & - & - \end{bmatrix}$$

$$T_1 + T_4 - T_5 + T_7 = \begin{bmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix} = AP + BR.$$

2. *Tranzitivní uzávěr* orientovaného grafu s vrcholy $\{1, \ldots, n\}$ je nula-jedničková matice T tvaru $n \times n$, kde $T_{uv} = 1$ pravě tehdy, když v grafu existuje cesta z vrcholu u

do vrcholu v. Ukažte, že umíme-li násobit matice $n \times n$ v čase $\mathcal{O}(n^\omega)$, můžeme vypočítat tranzitivní uzávěr v čase $\mathcal{O}(n^\omega \log n)$. Inspirujte se cvičením 5.3.4 z kapitoly o grafech.

10.6 Hledání k-tého nejmenšího prvku – Quickselect

Při použití metody Rozděl a panuj se někdy ukáže, že některé z částí, na které jsme vstup rozdělili, nemusíme vůbec zpracovávat. Typickým příkladem je následující algoritmus na hledání k-tého nejmenšího prvku posloupnosti.

Dostaneme-li na vstupu nějakou posloupnost prvků, jeden z nich si vybereme a budeme mu říkat *pivot*. Zadané prvky poté „rozhrneme“ na tři části: *doleva* půjdou prvky menší než pivot, *doprava* prvky větší než pivot a *uprostřed* zůstanou ty, které se pivotovi rovnají. Tyto části budeme značit po řadě L, P a S.

Kdybychom posloupnost setřídili, musí v ní vystupovat nejdříve všechny prvky z levé části, pak prvky z části střední a konečně ty z pravé. Pokud je tedy $k \leq |L|$, musí se hledaný prvek nalézat nalevo a musí tam být k-tý nejmenší (žádný prvek z jiné části ho nemohl předběhnout). Podobně je-li $|L| < k \leq |L| + |S|$, padne hledaný prvek v setříděné posloupnosti tam, kde leží S, a tedy je roven pivotovi. A konečně pro $k > |L| + |S|$ se musí nacházet v pravé části a musí tam být $(k - |L| - |S|)$-tý nejmenší.

Ze tří částí vstupu jsme si tedy vybrali jednu a v ní opět hledáme několikátý nejmenší prvek, na což samozřejmě použijeme rekurzi. Vznikne následující algoritmus, obvykle známý pod názvem Quickselect:

Algoritmus QuickSelect (hledání k-tého nejmenšího prvku)
Vstup: Posloupnost prvků $X = x_1, \ldots, x_n$ a číslo k $(1 \leq k \leq n)$

1. Pokud $n = 1$, vrátíme $y = x_1$ a skončíme.
2. $p \leftarrow$ některý z prvků $x_1, \ldots, x_n$ (pivot)
3. $L \leftarrow$ prvky v X, které jsou menší než p
4. $P \leftarrow$ prvky v X, které jsou větší než p
5. $S \leftarrow$ prvky v X, které jsou rovny p
6. Pokud $k \leq |L|$, pak $y \leftarrow$ QuickSelect(L, k).
7. Jinak je-li $k \leq |L| + |S|$, nastavíme $y \leftarrow p$.
8. Jinak $y \leftarrow$ QuickSelect$(P, k - |L| - |S|)$.

Výstup: $y = k$-tý nejmenší prvek v X

Správnost algoritmů je evidentní, ale jak to bude s časovou složitostí? Pokaždé strávíme lineární čas rozdělováním posloupnosti a pak se zavoláme rekurzivně na menší vstup. O kolik menší bude, to závisí zejména na volbě pivota. Jestliže si ho budeme vybírat nešikovně, například jako největší prvek vstupu, skončí $n-1$ prvků nalevo. Pokud navíc bude $k=1$, budeme se rekurzivně volat vždy na tuto obří levou část. Ta se pak opět zmenší pouze o jedničku a tak dále, takže celková časová složitost vyjde $\Theta(n)+\Theta(n-1)+\ldots+\Theta(1)=\Theta(n^2)$.

Obecněji pokud rozdělujeme vstup nerovnoměrně, hrozí nám, že nepřítel zvolí k tak, aby nás vždy vehnal do té větší části. Ideální obranou by tedy pochopitelně bylo volit za pivota *medián* posloupnosti.⟨4⟩ Tehdy bude nalevo i napravo nejvýše $n/2$ prvků (alespoň jeden je uprostřed) a ať už si během rekurze vybereme levou nebo pravou část, n bude exponenciálně klesat. Algoritmus pak doběhne v čase $\Theta(n)+\Theta(n/2)+\Theta(n/4)+\ldots+\Theta(1)=\Theta(n)$.

Medián ovšem není jediným pivotem, pro kterého algoritmus poběží lineárně. Zkusme za pivota zvolit *„skoromedián“* – tak budeme říkat prvku, který leží v prostředních dvou čtvrtinách setříděné posloupnosti. Tehdy bude nalevo i napravo nejvýše $3/4\cdot n$ prvků a velikost vstupu bude opět exponenciálně klesat, byť o chlup pomaleji: $\Theta(n)+\Theta(3/4\cdot n)+\Theta((3/4)^2\cdot n)+\ldots+\Theta(1)$. To je opět geometrická řada se součtem $\Theta(n)$.

Ani medián, ani skoromedián bohužel neumíme rychle najít. Jakého pivota tedy v algoritmu používat? Ukazuje se, že na tom příliš nezáleží – můžeme zvolit třeba prvek $x_{\lfloor n/2\rfloor}$ nebo si hodit kostkou (totiž pseudonáhodným generátorem) a vybrat ze všech x_i náhodně. Algoritmus pak bude mít průměrně lineární časovou složitost. Co to přesně znamená a jak to dokázat, odložíme do kapitoly 11. V praxi tento přístup každopádně funguje výtečně.

Prozatím se spokojíme s intuitivním vysvětlením: Alespoň polovina všech prvků jsou skoromediány, takže pokud se budeme trefovat náhodně (nebo pevně, ale vstup bude „dobře zamíchaný“), často se strefíme do skoromediánu a algoritmus bude „postupovat kupředu“ dostatečně rychle.

Cvičení

1. Proč navrhujeme volit za pivota prvek $x_{\lfloor n/2\rfloor}$, a ne třeba x_1 nebo x_n?

2. Student Šťoura si místo skoromediánů za pivoty volí „skoroskoromediány“, které leží v prostředních šesti osminách vstupu. Jaké dosahuje časové složitosti?

3. Jak by dopadlo, kdybychom na vstupu dostali posloupnost reálných čísel a jako pivota používali aritmetický průměr?

⟨4⟩ *Medián* je prvek, pro který platí, že nejvýše polovina prvků je menší než on a nejvýše polovina větší; tuto vlastnost má $\lfloor(n+1)/2\rfloor$-tý a $\lceil(n+1)/2\rceil$-tý nejmenší prvek.

4. Uvědomte si, že binární vyhledávání je také algoritmus typu Rozděl a panuj, v němž velikost vstupu exponenciálně klesá. Spočítejte jeho časovou složitost metodami z této kapitoly. Čím se liší od Quickselectu?

10.7 Ještě jednou třídění – Quicksort

Rozdělování vstupu podle pivota, které se osvědčilo v minulé kapitole, můžeme použít i ke třídění dat. Připomeňme, že rozdělíme-li vstup na levou, pravou a střední část, budou v setříděné posloupnosti vystupovat nejdříve prvky z levé části, pak ty z prostřední a nakonec prvky z části pravé. Můžeme tedy rekurzivně setřídit levou a pravou část (prostřední je sama od sebe setříděná), pak části poskládat ve správném pořadí a získat setříděnou posloupnost. Tomuto třídicímu algoritmu se říká *Quicksort.*⟨5⟩

Algoritmus QuickSort
Vstup: Posloupnost prvků $X = x_1, \ldots, x_n$ k setřídění
1. Pokud $n \leq 1$, vrátíme $Y = X$ a skončíme.
2. $p \leftarrow$ některý z prvků $x_1, \ldots, x_n$ (pivot)
3. $L \leftarrow$ prvky v X, které jsou menší než p
4. $P \leftarrow$ prvky v X, které jsou větší než p
5. $S \leftarrow$ prvky v X, které jsou rovny p
6. Rekurzivně setřídíme části:
7. $\quad L \leftarrow \text{QuickSort}(L)$
8. $\quad P \leftarrow \text{QuickSort}(P)$
9. Slepíme části za sebe: $Y \leftarrow L, S, P$.

Výstup: Setříděná posloupnost Y

Dobrou představu o rychlosti algoritmu nám jako obvykle dá strom rekurzivních volání. V kořeni máme celý vstup, na první hladině jeho levou a pravou část, na druhé hladině levé a pravé části těchto částí, a tak dále, až v listech triviální posloupnosti délky 1. Rekurzivní volání na vstup nulové délky do stromu kreslit nebudeme a rovnou je zabudujeme do jejich otců.

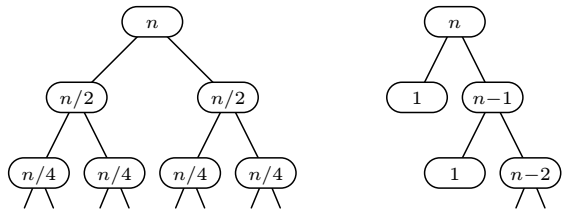
Jelikož rozkládání vstupu i skládání výsledku jistě pokaždé stihneme v lineárním čase, trávíme v každém vrcholu čas přímo úměrný velikosti příslušného podproblému. Pro libovolnou hladinu navíc platí, že podproblémy, které na ní leží, mají dohromady nejvýše n

⟨5⟩ Za jménem se často skrývá příběh. Quicksort (což znamená „Rychlotřidič“) přišel ke svému jménu tak, že v roce 1961, kdy vznikl, byl prvním třídicím algoritmem se složitostí $\mathcal{O}(n \log n)$ aspoň v průměru. Dodejme ještě, že jeho objevitel Anthony Hoare byl později anglickou královnou povýšen na rytíře mimo jiné za zásluhy o informatiku.

prvků – vznikly totiž rozdělením vstupu na disjunktní části a ještě se nám při tom některé prvky (pivoti) poztrácely. Na jedné hladině proto trávíme čas $\mathcal{O}(n)$.

Tvar stromu a s ním i časová složitost samozřejmě opět stojí a padají s volbou pivota. Pokud za pivoty volíme mediány nebo alespoň skoromediány, klesají velikosti podproblémů exponenciálně (na i-té hladině $\mathcal{O}((3/4)^i \cdot n)$), takže strom je vyvážený a má hloubku $\mathcal{O}(\log n)$. V součtu přes všechny hladiny proto časová složitost činí $\mathcal{O}(n \log n)$.

Jestliže naopak volíme pivoty nešťastně jako (řekněme) největší prvky vstupu, oddělí se na každé hladině od vstupu jen úsek o jednom prvku a hladin bude $\Theta(n)$. To povede na kvadratickou časovou složitost. Horší případ již nenastane, neboť na každé hladině přijdeme alespoň o prvek, který se stal pivotem.



Obrázek 10.7: Quicksort při dobré a špatné volbě pivota

Podobně jako u Quickselectu, i zde je mnoho „dobrých“ pivotů, se kterými se algoritmus chová efektivně (alespoň polovina prvků jsou skoromediány). V praxi proto opět funguje spoléhat na náhodný generátor nebo dobře zamíchaný vstup. V kapitole 11.2 pak vypočteme, že Quicksort s náhodnou volbou pivota má časovou složitost $\mathcal{O}(n \log n)$ v průměru.

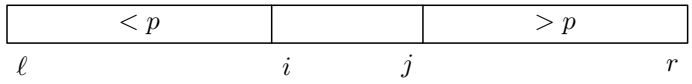
Quicksort v praxi

Závěrem si dovolme krátkou poznámku o praktických implementacích Quicksortu. Ačkoliv tento algoritmus mezi ostatními třídicími algoritmy na první pohled ničím nevyniká, u většiny překladačů se v roli standardní funkce pro třídění setkáte právě s ním. Důvodem této nezvyklé popularity není móda, nýbrž praktické zkušenosti. Dobře vyladěná implementace Quicksortu totiž na reálném počítači běží výrazně rychleji než jiné třídicí algoritmy.

Cesta od našeho poměrně obecně formulovaného algoritmu k takto propracovanému programu je samozřejmě složitá a vyžaduje mimo mistrného zvládnutí programátorského řemesla i detailní znalost konkrétního počítače. My si ukážeme alespoň první kroky této cesty.

Především Quicksort upravíme tak, aby prvky zbytečně nekopíroval. Vstup dostane jako ostatní třídicí algoritmy v poli a pak bude pouze prohazovat prvky uvnitř tohoto pole. Rekurzivně tedy budeme třídit různé úseky společného pole. Kterým úsekem se máme právě zabývat, vymezíme snadno indexy krajních prvků. Levý okraj úseku (ten blíže k začátku pole) budeme značit ℓ, pravý pak r.

Rozdělování se zjednoduší, budeme-li vstup dělit jen na dvě části namísto tří – prvky rovné pivotovi mohou bez újmy na korektnosti přijít jak nalevo, tak napravo. Budeme postupovat následovně: Použijeme dva indexy i a j. První z nich bude procházet tříděným úsekem zleva doprava a přeskakovat prvky, které mají zůstat nalevo; druhý index půjde zprava doleva a bude přeskakovat prvky patřící do pravé části. Levý index se tudíž zastaví na prvním prvku, který je vlevo, ale patří doprava; podobně pravý index se zastaví na nejbližším prvku vpravo, který patří doleva. Stačí pak tyto dva prvky prohodit a pokračovat stejným způsobem dál, až se indexy setkají.



Obrázek 10.8: Quicksort s rozdělováním na místě

Nyní máme obě části uložené v souvislých úsecích pole, takže je můžeme rekurzivně setřídit. Navíc se tyto úseky vyskytují přesně tam, kde mají ležet v setříděné posloupnosti, takže „slepovací“ krok 9 původního Quicksortu můžeme zcela vynechat.

Algoritmus QuickSort2
Vstup: Pole $P[1 \ldots n]$, indexy ℓ a r úseku, který třídíme

1. Pokud $\ell \geq r$, ihned skončíme.
2. $p \leftarrow$ některý z prvků $P[\ell], \ldots, P[r]$ (pivot)
3. $i = \ell$, $j = r$
4. Dokud $i \leq j$, opakujeme:
5. Dokud $P[i] < p$, zvyšujeme i o 1.
6. Dokud $P[j] > p$, snižujeme j o 1.
7. Je-li $i < j$, prohodíme $P[i]$ a $P[j]$.
8. Je-li $i \leq j$, pak $i \leftarrow i + 1$, $j \leftarrow j - 1$.
9. Rekurzivně setřídíme části:
10. QuickSort2(P, ℓ, j)
11. QuickSort2(P, i, r)

Výstup: Úsek $P[\ell \ldots r]$ je setříděn

Popsanými úpravami jsme jistě nezhoršili časovou složitost: V krocích 2–8 zpracujeme každý prvek úseku nejvýše jednou, celkově tedy rozdělováním trávíme čas lineární s délkou úseku, s čímž naše analýza časové složitosti počítala. Naopak jsme se zbavili zbytečného kopírování prvků do pomocné paměti. Další možná vylepšení ponecháváme čtenáři jako cvičení s nápovědou.

Cvičení

1. Rozmyslete si, že procedura QuickSort2 je korektní. Zejména si uvědomte, co se stane, když si jako pivota vybereme nejmenší nebo největší prvek úseku nebo když dokonce budou všechny prvky v úseku stejné. Ani tehdy během kroků 4–8 nemohou indexy i, j opustit tříděný úsek a každá z částí, na které se rekurzivně zavoláme, bude ostře menší než původní vstup.

2. *Vlastní zásobník:* Abyste netrávili tolik času rekurzivním voláním a předáváním parametrů, nahraďte rekurzi svým vlastním zásobníkem, na kterém si budete pamatovat začátky a konce úseků, které ještě zbývá setřídit.

3. *Šetříme pamětí:* Vylepšete postup z minulého cvičení tak, že dvojici rekurzivních volání v krocích 7 a 8 nahradíte uložením *většího* úseku na zásobník a pokračováním tříděním *menšího* úseku. Dokažte, že po této úpravě může být v libovolný okamžik na zásobníku jen $\mathcal{O}(\log n)$ úseků. Tím množství potřebné pomocné paměti kleslo z lineárního na logaritmické.

4. *Včas se zastavíme:* Podobně jako při násobení čísel (cvičení 10.3.4) se u Quicksortu hodí zastavit rekurzi předčasně (pro n menší než vhodná konstanta n_0) a přepnout na některý z kvadratických třídicích algoritmů. Vyzkoušejte si pro svůj konkrétní program najít hodnotu n_0, se kterou poběží nejrychleji.

5. *Medián ze tří:* Oblíbený trik na výběr pivota je spočítat medián z prvního, prostředního a posledního prvku úseku. Předpokládáme-li, že na vstupu dostaneme náhodnou permutaci čísel $1, \ldots, n$, jaká je pravděpodobnost, že takový pivot bude skoromediánem? S jakou pravděpodobností bude minimem?

10.8 k-tý nejmenší prvek v lineárním čase

Algoritmus Quickselect pro hledání k-tého nejmenšího prvku, který jsme potkali v kapitole 10.6, pracuje v lineárním čase pouze průměrně. Nyní ukážeme, jak ho upravit, aby tuto časovou složitost měl i v nejhorším případě. Jediné, co změníme, bude volba pivota.

Prvky si nejprve seskupíme do pětic (není-li poslední pětice úplná, doplníme ji „nekonečně velkými“ hodnotami). Poté nalezneme medián každé pětice a z těchto mediánů

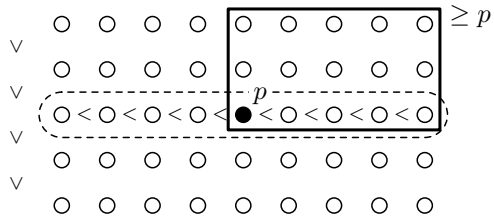
rekurzivním zavoláním našeho algoritmu spočítáme opět medián. Ten pak použijeme jako pivota k rozdělení vstupu na levou, střední a pravou část a pokračujeme jako v původním Quickselectu. Celý algoritmus bude vypadat takto:

Algoritmus LinearSelect (k-tý nejmenší prvek v lineárním čase)
Vstup: Posloupnost prvků $X = x_1, \ldots, x_n$ a číslo k $(1 \leq k \leq n)$

1. Pokud $n \leq 5$, úlohu vyřešíme triviálním algoritmem.
2. Prvky rozdělíme na pětice $P_1, \ldots, P_{\lceil n/5 \rceil}$.
3. Spočítáme mediány pětic: $m_i \leftarrow$ medián P_i.
4. Najdeme pivota: $p \leftarrow \textsc{LinearSelect}(m_1, \ldots, m_{\lceil n/5 \rceil}; \lceil n/10 \rceil)$.
5. $L, P, S \leftarrow$ prvky z X, které jsou menší než p, větší než p, rovny p.
6. Pokud $k \leq |L|$, pak $y \leftarrow \textsc{LinearSelect}(L, k)$.
7. Jinak je-li $k \leq |L| + |S|$, nastavíme $y \leftarrow p$.
8. Jinak $y \leftarrow \textsc{LinearSelect}(P, k - |L| - |S|)$.

Výstup: $y = k$-tý nejmenší prvek v X

Abychom chování algoritmu pochopili, uvědomíme si nejdříve, že vybraný pivot není příliš daleko od mediánu celé posloupnosti X. K tomu nám pomůže obrázek.



Obrázek 10.9: Pětice a jejich mediány

Překreslíme do něj vstup a každou pětici uspořádáme zdola nahoru. Mediány pětic tedy budou ležet v prostředním řádku. Pětice ještě přeházíme tak, aby jejich mediány rostly zleva doprava. (Pozor, algoritmus nic takového nedělá, pouze my při jeho analýze!) Navíc budeme pro jednoduchost předpokládat, že pětic je lichý počet a že všechny prvky jsou navzájem různé.

Náš pivot (medián mediánů pětic) se tedy na obrázku nachází přesně uprostřed. Mediány všech pětic, které leží napravo od něj, jsou proto větší než pivot. Všechny prvky umístěné nad nimi jsou ještě větší, takže celý obdélník, jehož levým dolním rohem je pivot, padne v našem algoritmu do části P nebo S. Počítejme, kolik obsahuje prvků: Všech pětic je $n/5$, polovina z nich ($\lceil n/10 \rceil$ pětic) zasahuje do našeho obdélníku, a to třemi prvky. To

celkem dává alespoň $3/10 \cdot n$ prvků, o kterých s jistotou víme, že se neobjeví v L. Levá část proto měří nejvýše $7/10 \cdot n$.

Podobně nahlédneme, že napravo je také nejvýše $7/10 \cdot n$ prvků – stačí uvážit obdélník, který se rozprostírá od pivota doleva dolů. Všechny jeho prvky leží v L nebo S a opět jich je minimálně $3/10 \cdot n$.

Tato úvaha nám pomůže v odhadu časové složitosti:

- Rozdělovaní na pětice a počítání jejich mediánů je lineární – pětice jsou konstantně velké, takže medián jedné spočítáme sebehloupějším algoritmem za konstantní čas.
- Dělení posloupnosti na části L, P a S a rozhodování, do které z částí se vydat, trvá také $\Theta(n)$.
- Poprvé zavoláme LinearSelect rekurzivně ve 4. kroku na $n/5$ prvků.
- Podruhé ho zavoláme v kroku 6 nebo 8, a to na levou nebo pravou část vstupu. Jak už víme, každá z nich měří nejvýše $7/10 \cdot n$.

Pro časovou složitost v nejhorším případě proto dostaneme následující rekurentní rovnici (konstant jsme se zbavili vhodnou volbou jednotky času):

$$\begin{aligned} T(1) &= \mathcal{O}(1), \\ T(n) &= T(n/5) + T(7/10 \cdot n) + n. \end{aligned}$$

Metody z předchozích kapitol jsou na vyřešení této rekurence krátké (s výjimkou obecného postupu z cvičení 10.4.3). Pomůže nám válečná lest: uhodneme, že $T(n) = cn$, a ověříme dosazením, že existuje taková konstanta c, pro kterou tato funkce naši rekurenci splňuje:

$$\begin{aligned} cn &= 1/5 \cdot cn + 7/10 \cdot cn + n = \\ &= 9/10 \cdot cn + n. \end{aligned}$$

Tato rovnost platí pro $c = 10$. Náš algoritmus tedy opravdu hledá k-tý nejmenší prvek v lineárním čase.

Nyní bychom mohli upravit Quicksort, aby jako pivota používal medián spočítaný tímto algoritmem. Pak by třídil v čase $\Theta(n \log n)$ i v nejhorším případě. Příliš praktický takový algoritmus ale není. Jak asi tušíte, naše dvojitě rekurzivní hledání mediánu je sice asymptoticky lineární, ale konstanty, které v jeho složitosti vystupují, nejsou zrovna malé. Bývá proto užitečnější volit pivota náhodně a smířit se s tím, že občas promarníme jeden průchod kvůli nešikovnému pivotovi, než si třídění stále brzdit důmyslným vybíráním kvalitních pivotů.

Cvičení

1. Rozmyslete si, že našemu algoritmu nevadí, když prvky na vstupu nebudou navzájem různé.

2. Upravte funkci LinearSelect tak, aby si vystačila s konstantně velkou pomocnou pamětí. Prvky ve vstupním poli můžete libovolně přeskupovat.

3. Jak bude vypadat strom rekurzivních volání funkce LinearSelect? Kolik bude mít listů? Jak dlouhá bude nejkratší a nejdelší větev?

4. Proč při vybírání k-tého nejmenšího prvku používáme zrovna pětice? Fungoval by algoritmus s trojicemi? Nebo se sedmicemi? Byl by pak stále lineární?

5.* Na medián se můžeme dívat také tak, že je to „patník“ na půli cesty od minima k maximu. Jinými slovy, mezi minimem a mediánem leží přibližně stejně prvků jako mezi mediánem a maximem. Co kdybychom chtěli mezi minimum a maximum co nejrovnoměrněji rozmístit více patníků?

   Přesněji: pro n-prvkovou množinu prvků X a číslo ε ($0 < \varepsilon < 1$) definujeme *ε-síť* jako posloupnost $\min X = x_0 < x_1 < \ldots < x_{\lceil 1/\varepsilon \rceil} = \max X$ prvků vybraných z X tak, aby se mezi x_i a x_{i+1} vždy nacházelo nejvýše εn prvků z X. Pro $\varepsilon = 1/2$ tedy počítáme minimum, medián a maximum, pro $\varepsilon = 1/4$ přidáme prvky ve čtvrtinách, …, a při $\varepsilon = 1/n$ už třídíme.

   Složitost hledaní ε-sítě se tedy v závislosti na hodnotě ε bude pohybovat mezi $\mathcal{O}(n)$ a $\mathcal{O}(n \log n)$. Najděte algoritmus s časovou složitostí $\mathcal{O}(n \log(1/\varepsilon))$.

6. Je dáno n-prvkové pole, ve kterém jsou za sebou dvě vzestupně setříděné posloupnosti (ne nutně stejně dlouhé). Navrhněte algoritmus, který najde medián sjednocení obou posloupností v sublineárním čase. (Lze řešit v čase $\mathcal{O}(\log n)$.)

10.9 Další cvičení

1. *Spletitý kabel:* Mějme dlouhý kabel, z jehož obou konců vystupuje po n drátech. Každý drát na levém konci je propojen s právě jedním na konci druhém a my chceme zjistit, který s kterým. K tomu můžeme používat následující operace: (1) přivést napětí na daný drát na levém konci, (2) odpojit napětí z daného drátu na levém konci, (3) změřit napětí na daném drátu na pravém konci. Navrhněte algoritmus, který pomocí těchto operací zjistí, co je s čím propojeno. Snažte se počet operací minimalizovat.

2.* Nalezněte *neadaptivní* řešení předchozího cvičení, tedy takové, v němž položené dotazy nezávisí na výsledcích předchozích dotazů.

3. Dokažte, že v předchozích dvou cvičeních je potřeba $\Omega(n \log n)$ dotazů. Pokud nevíte, jak na to, dokažte to alespoň pro neadaptivní algoritmy.

4. *Inverze matice:* Navrhněte algoritmus typu Rozděl a panuj na výpočet inverze trojúhelníkové matice $n \times n$ v čase lepším než $\Omega(n^3)$. Jako podprogram se může hodit Strassenovo násobení matic z oddílu 10.5. Můžete předpokládat, že n je mocnina dvojky.

5. *Inverze v posloupnosti* $x_1, \ldots, x_n$ říkáme každé dvojici (i, j) takové, že $i < j$ a současně $x_i > x_j$. Vymyslete algoritmus, který spočítá, kolik daná posloupnost obsahuje inverzí. To může sloužit jako míra neuspořádanosti.

6.* *Nejbližší body:* Máme n bodů v rovině a chceme najít dvojici s nejmenší vzdáleností. Nabízí se rozdělit body vodorovnou přímkou podle mediánu y-ových souřadnic, rekurzivně spočítat nejmenší vzdálenosti ε_1 a ε_2 v obou polorovinách a pak dopočítat, co se děje v pásu o šíři $2\min(\varepsilon_1, \varepsilon_2)$ podél dělící přímky. Dokažte, že probíráme-li body pásu zleva doprava, stačí každý bod porovnat s $\mathcal{O}(1)$ sousedy. To vede na algoritmus o složitosti $\Theta(n \log n)$.

7.* *Šroubky a matičky:* Na stole leží n různých šroubků a n matiček. Každá matička pasuje na právě jeden šroub a my chceme zjistit, která na který. Umíme ale pouze porovnávat šroub s matičkou – tím získáme jeden ze tří možných výsledků: matička je příliš velká, příliš malá, nebo správně velká. Nalezněte co nejefektivnější algoritmus.

11 Randomizace

11 Randomizace

Výhodou algoritmů je, že se na ně můžeme spolehnout. Jsou to dokonale deterministické předpisy, které pro zadaný vstup pokaždé vypočítají tentýž výstup. Přesto může být zajímavé vpustit do nich pečlivě odměřené množství náhody. Získáme tím takzvané pravděpodobnostní neboli randomizované algoritmy. Ty mnohdy dovedou dospět k výsledku daleko rychleji než jejich klasičtí příbuzní. Přitom budeme stále schopní o jejich chování ledacos dokázat.

Randomizace nám pomůže například s výběrem pivota v algoritmech Quicksort a Quickselect z předchozí kapitoly. Také zavedeme hešovací tabulky – velice rychlé datové struktury založené na náhodném rozmisťování hodnot do přihrádek.

11.1 Pravděpodobnostní algoritmy

Nejprve rozšíříme definici algoritmu z kapitoly o složitosti, aby umožňovala náhodný výběr. Zařídíme to tak, že do výpočetního modelu RAM doplníme novou instrukci X `:= random(`Y`,`Z`)`, kde X je odkaz do paměti a Y a Z buďto odkazy do paměti, nebo konstanty.

Kdykoliv stroj při provádění programu narazí na tuto instrukci, vygeneruje náhodné celé číslo v rozsahu od Y do Z a uloží ho do paměťové buňky X. Všechny hodnoty z tohoto rozsahu budou stejně pravděpodobné a volba bude nezávislá na předchozích voláních instrukce `random`. Bude-li $Y > Z$, program skončí běhovou chybou podobně, jako kdyby dělil nulou.

Ponechme stranou, zda je možné počítač s náhodným generátorem skutečně sestrojit. Ostatně, samu existenci náhody v našem vesmíru nejspíš nemůžeme nijak prokázat. Je to tedy spíš otázka víry. Teoretické informatice na odpovědi nezáleží – prostě předpokládá výpočetní model s ideálním náhodným generátorem, který se řídí pravidly teorie pravděpodobnosti. V praxi si pak pomůžeme generátorem *pseudonáhodným,* který generuje prakticky nepředvídatelná čísla. Dodejme ještě, že existují i jiné modely náhodných generátorů, než je naše funkce `random`, ale ty ponechme do cvičení.

Algoritmům využívajícím náhodná čísla se obvykle říká *pravděpodobnostní* nebo *randomizované.* Výpočet takového algoritmu pro konkrétní vstup může v závislosti na náhodě trvat různě dlouho a dokonce může dospět k různým výsledkům. Doba běhu, případně výsledek pak nejsou konkrétní čísla, ale náhodné veličiny. U těch se obvykle budeme ptát na střední hodnotu, případně na pravděpodobnost, že překročí určitou mez.

Připomenutí teorie pravděpodobnosti

Pro analýzu randomizovaných algoritmů budeme používat aparát matematické teorie pravděpodobnosti. Zopakujme si její základní pojmy a tvrzení. Pokud jste se s nimi dosud nesetkali, naleznete je například v Kapitolách z diskrétní matematiky [9].

Budeme pracovat s *diskrétním pravděpodobnostním prostorem* (Ω, P). Ten je tvořen nejvýše spočetnou množinou Ω *elementárních jevů* a funkcí $P : \Omega \to [0, 1]$, která elementárním jevům přiřazuje jejich *pravděpodobnosti.* Součet pravděpodobností všech elementárních jevů je roven 1. *Jev* je obecně nějaká množina elementárních jevů $A \subseteq \Omega$. Funkci P můžeme přirozeně rozšířit na všechny jevy: $P(A) = \sum_{e \in A} P(e)$. Pravděpodobnosti můžeme také připisovat výrokům: $\Pr[\varphi(x)]$ je pravděpodobnost jevu daného množinou všech elementárních jevů x, pro které platí výrok $\varphi(x)$.

Pro každé dva jevy A a B platí $P(A \cup B) = P(A) + P(B) - P(A \cap B)$. Pokud $P(A \cap B) = P(A) \cdot P(B)$, řekneme, že A a B jsou *nezávislé.* Pro více jevů $A_1, \ldots, A_k$ rozlišujeme *nezávislost po dvou* (každé dva jevy A_i a A_j jsou nezávislé) a *plnou nezávislost* (pro každou podmnožinu indexů $\emptyset \neq I \subseteq \{1, \ldots, k\}$ platí $P(\cap_{i \in I} A_i) = \prod_{i \in I} P(A_i)$).

Náhodné veličiny (proměnné) jsou funkce z Ω do reálných čísel, přiřazují tedy reálné hodnoty možným výsledkům pokusu. Můžeme se ptát na pravděpodobnost, že veličina má nějakou vlastnost, například $\Pr[X > 5]$. *Střední hodnota* $\mathbb{E}[X]$ náhodné veličiny X je průměr všech možných hodnot vážený jejich pravděpodobnostmi, tedy

$$\sum_{x \in \mathbb{R}} x \cdot \Pr[X = x] = \sum_{e \in \Omega} X(e) \cdot P(e).$$

Často budeme používat následující vlastnost střední hodnoty:

Fakt (linearita střední hodnoty): Nechť X a Y jsou náhodné veličiny a α a β reálná čísla. Potom $\mathbb{E}[\alpha X + \beta Y] = \alpha \mathbb{E}[X] + \beta \mathbb{E}[Y]$.

Tak dlouho se chodí se džbánem ...

Než se pustíme do pravděpodobnostní analýzy algoritmů, začneme jednoduchým příkladem: budeme chodit se džbánem pro vodu tak dlouho, než se utrhne ucho. To při každém pokusu nastane náhodně s pravděpodobností p $(0 < p < 1)$, nezávisle na výsledcích předchozích pokusů. Kolik pokusů v průměru podnikneme?

Označme T počet kroků, po kterém dojde k utržení ucha. To je nějaká náhodná veličina a nás bude zajímat její střední hodnota $\mathbb{E}[T]$. Podle definice střední hodnoty platí:

$$\mathbb{E}[T] = \sum_i \left(i \cdot \Pr[\text{ucho se utrhne při } i\text{-tém pokusu}]\right) = \sum_i \left(i \cdot (1-p)^{i-1} \cdot p\right).$$

U každé nekonečné řady se sluší nejprve zeptat, zda vůbec konverguje. Na to nám kladně odpoví třeba podílové kritérium. Nyní bychom mohli řadu poctivě sečíst, ale místo toho použijeme jednoduchý trik založený na linearitě střední hodnoty.

V každém případě provedeme první pokus. Pokud se ucho utrhne (což nastane s pravděpodobností p), hra končí. Pokud se neutrhne (pravděpodobnost $1-p$), dostaneme se do přesně stejné situace, jako předtím – náš ideální džbán totiž nemá žádnou paměť. Z toho vyjde následující rovnice pro $\mathbb{E}[T]$:

$$\mathbb{E}[T] = 1 + p \cdot 0 + (1-p) \cdot \mathbb{E}[T].$$

Vyřešíme-li ji, dostaneme $\mathbb{E}[T] = 1/p$. (Zde jsme nicméně potřebovali vědět, že střední hodnota existuje a je konečná, takže úvahy o nekonečných řadách byly nezbytné.)

Tento výsledek se nám bude často hodit, formulujme ho proto jako lemma:

Lemma (o džbánu): Čekáme-li na náhodný jev, který nastane s pravděpodobností p, dočkáme se ve střední hodnotě po $1/p$ pokusech.

Cvičení

1. *Ideální mince:* Mějme počítač, jehož náhodným generátorem je ideální mince. Jinými slovy, máme instrukci `random_bit`, ze které na každé zavolání vypadne jeden rovnoměrně náhodný bit vygenerovaný nezávisle na předchozích bitech. Jak pomocí takové funkce generovat rovnoměrně náhodná přirozená čísla od 1 do n? Minimalizujte průměrný počet hodů mincí.

2. Ukažte, že v předchozím cvičení nelze počet hodů mincí v nejhorším případě nijak omezit, leda kdyby n bylo mocninou dvojky.

3. *Míchání karet:* Popište algoritmus, který v lineárním čase vygeneruje náhodnou permutaci množiny $\{1, 2, \ldots, n\}$.

4. V mnoha programovacích jazycích je k dispozici funkci `random`, která nám vrátí rovnoměrně (pseudo)náhodné číslo z pevně daného intervalu. Lidé ji často používají pro generování čísel v rozsahu od 0 do $n-1$ tak, že spočtou `random` mod n. Jaký se v tom skrývá háček? Jak ho obejít?

5. *Náhodná k-tice:* Máme-li obrovský soubor a chceme o něm získat alespoň hrubou představu, hodí se prozkoumat náhodnou k-tici řádků. Vymyslete algoritmus, který ji vybere tak, aby všechny k-tice měly stejnou pravděpodobnost. Vstup se celý nevejde do paměti a jeho velikost ani předem neznáme; k-tice se do paměti spolehlivě vejde. Zvládnete to na jediný průchod daty?

6.* *Míchání podruhé:* Vasil Vasiljevič míchá karty takto: připraví si n prázdných přihrádek. Pak postupně umisťuje čísla $1, \ldots, n$ do přihrádek tak, že vždy vybere rovnoměrně náhodně přihrádku a pokud v ní již něco je, vybírá znovu. Kolik pokusů bude v průměru potřebovat?

7. Lemma o džbánu můžeme dokázat i sečtením uvedené nekonečné řady. Ta je ostatně podobná řadě, již jsme zkoumali při rozboru konstrukce haldy v oddílu 4.2. Zkuste to.

8. Představte si, že hodíme 10 hracími kostkami a počty ok sečteme. V jakém pravděpodobnostním prostoru se tento pokus odehrává? O jakou náhodnou veličinu jde? Jak stanovit její střední hodnotu?

9. V jakém pravděpodobnostním prostoru se odehrává lemma o džbánu?

11.2 Náhodný výběr pivota

U algoritmů založených na výběru pivota (Quickselect a Quicksort) jsme spoléhali na to, že pokud budeme pivota volit náhodně, bude se algoritmus „chovat dobře.“ Nyní nastal čas říci, co to přesně znamená, a přednést důkaz.

Mediány, skoromediány a Quickselect

Úvaha o džbánu nám dává jednoduchý algoritmus, pomocí kterého umíme najít skoromedián posloupnosti n prvků: Vybereme si *rovnoměrně náhodně* jeden z prvků posloupnosti; tím *rovnoměrně* myslíme tak, aby všechny prvky měly stejnou pravděpodobnost. Pak ověříme, jestli vybraný prvek je skoromedián. Pokud ne, na vše zapomeneme a postup opakujeme.

Kolik pokusů budeme potřebovat, než algoritmus skončí? Skoromediány tvoří alespoň polovinu prvků, tedy pravděpodobnost, že se do nějakého strefíme, je minimálně $1/2$. Podle lemmatu o džbánu tedy střední hodnota počtu pokusů bude nejvýše 2. (Počet pokusů v nejhorším případě ovšem neumíme omezit nijak – při dostatečné smůle můžeme stále vybírat nejmenší prvek. To ovšem nastane s nulovou pravděpodobností.)

Nyní už snadno spočítáme časovou složitost našeho algoritmu. Jeden pokus trvá $\Theta(n)$, střední hodnota počtu pokusů je $\Theta(1)$, takže střední hodnota časové složitosti je $\Theta(n)$. Obvykle budeme zkráceně mluvit o *průměrné* časové složitosti.

Pokud tento výpočet skoromediánu použijeme v Quickselectu, získáme algoritmus pro hledání k-tého nejmenšího prvku s průměrnou složitostí $\Theta(n)$. Dobrá, tím jsme se ale šalamounsky vyhnuli otázce, jakou průměrnou složitost má původní Quickselect s rovnoměrně náhodnou volbou pivota.

Tu odhadneme snadno: Rozdělíme běh algoritmu na *fáze.* Fáze bude končit v okamžiku, kdy za pivota zvolíme skoromedián. Fáze se skládá z *kroků* spočívajících v náhodné volbě pivota, lineárně dlouhém výpočtu a zahození části vstupu. Už víme, že skoromedián se průměrně podaří najít za dva kroky, tudíž jedna fáze trvá v průměru lineárně dlouho. Navíc si všimneme, že během každé fáze se vstup zmenší alespoň o čtvrtinu. K tomu totiž stačil samotný poslední krok, ostatní kroky mohou situaci jedině zlepšit. Průměrnou složitost celého algoritmu pak vyjádříme jako součet průměrných složitostí jednotlivých fází: $\Theta(n) + \Theta((3/4) \cdot n) + \Theta((3/4)^2 \cdot n) + \ldots = \Theta(n)$.

Indikátory a Quicksort

Podíváme-li se na výpočet v minulém odstavci s odstupem, všimneme si, že se opírá zejména o linearitu střední hodnoty. Časovou složitost celého algoritmu jsme vyjádřili jako součet složitostí fází. Přitom fázi jsme nadefinovali tak, aby se již chovala dostatečně průhledně.

Podobně můžeme analyzovat i Quicksort, jen se nám bude hodit složitost rozložit na daleko více veličin. Mimo to si všimneme, že Quicksort na každé porovnání provede jen $\mathcal{O}(1)$ dalších operací, takže postačí odhadnout počet provedených porovnání.

Očíslujeme si prvky podle pořadí v setříděné posloupnosti $y_1, \ldots, y_n$. Zavedeme náhodné veličiny C_{ij} pro $1 \leq i < j \leq n$ tak, aby platilo $C_{ij} = 1$ právě tehdy, když během výpočtu došlo k porovnání y_i s y_j. V opačném případě je $C_{ij} = 0$. Proměnným, které nabývají hodnoty 0 nebo 1 podle toho, zda nějaká událost nastala, se obvykle říká *indikátory.* Počet všech porovnání je tudíž roven součtu všech indikátorů C_{ij}. (Zde využíváme toho, že Quicksort tutéž dvojici neporovná vícekrát.)

Zamysleme se nyní nad tím, kdy může být $C_{ij} = 1$. Algoritmus porovnává pouze s pivotem, takže jedna z hodnot y_i, y_j se těsně předtím musí stát pivotem. Navíc všechny hodnoty $y_{i+1}, \ldots, y_{j-1}$ se ještě pivoty stát nesměly, jelikož jinak by y_i a y_j už byly rozděleny v různých částech posloupnosti. Jinými slovy, C_{ij} je rovno jedné právě tehdy, když se z hodnot $y_i, y_{i+1}, \ldots, y_j$ stane jako první pivotem buď y_i nebo y_j. A poněvadž pivota vybíráme rovnoměrně náhodně, má každý z prvků $y_i, \ldots, y_j$ stejnou pravděpodobnost, že se stane pivotem jako první, totiž $1/(j-i+1)$. Proto $C_{ij} = 1$ nastane s pravděpodobností $2/(j-i+1)$.

Nyní si stačí uvědomit, že když indikátory nabývají pouze hodnot 0 a 1, je jejich střední hodnota rovna právě pravděpodobnosti jedničky, tedy také $2/(j-i+1)$. Sečtením přes všechny dvojice (i, j) pak dostaneme pro počet všech porovnání:

$$\mathbb{E}[C] = \sum_{1 \leq i < j \leq n} \frac{2}{j-i+1} \leq 2n \cdot \sum_{2 \leq d \leq n} \frac{1}{d}.$$

Nerovnost na pravé straně platí díky tomu, že rozdíly $j - i + 1$ se nacházejí v intervalu $[2, n]$ a každým rozdílem přispěje nejvýše n různých dvojic (i, j). Poslední suma je tzv. harmonická suma, která sečte na $\Theta(\log n)$. Jelikož se s ní při analýze algoritmů potkáváme často, vyslovíme o ní samostatné lemma.

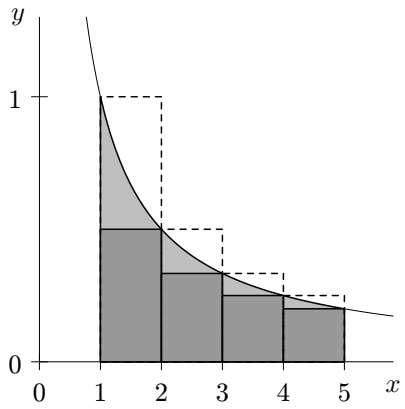
Lemma (o harmonických číslech): Pro součet harmonické řady $H_n = 1/1 + 1/2 + \ldots + 1/n$ platí $\ln n \leq H_n \leq \ln n + 1$.

Důkaz: Sumu odhadneme pomocí integrálu

$$I(n) = \int_1^n 1/x \, \mathrm{d}x = [\ln x]_1^n = \ln n - \ln 1 = \ln n.$$

Sledujme obrázek 11.1. Funkce $I(n)$ vyjadřuje obsah plochy mezi křivkou $y = f(x)$, osou x a svislými přímkami $x = 1$ a $x = n$. Součástí této plochy je tmavé „schodiště“, jehož obsah je $1 \cdot (1/2) + 1 \cdot (1/3) + \ldots + 1 \cdot (1/n) = H_n - 1$. Proto musí platit $H_n - 1 \leq I(n) = \ln n$, což je horní odhad z tvrzení lemmatu.

Dolní odhad dostaneme pomocí čárkovaného schodiště. Jeho obsah je $1 \cdot (1/1) + 1 \cdot (1/2) + \ldots + 1 \cdot (1/(n-1)) = H_n - 1/n$. Plocha měřená integrálem je součástí tohoto schodiště, pročež $\ln n = I(n) \leq H_n - (1/n) \leq H_n$. □



Obrázek 11.1: K důkazu lemmatu o harmonických číslech

Důsledek: Střední hodnota časové složitosti Quicksortu s rovnoměrně náhodnou volbou pivota je $\mathcal{O}(n \log n)$.

Chování na náhodném vstupu

Když jsme poprvé přemýšleli o tom, jak volit pivota, všimli jsme si, že pokud volíme pivota pevně, náš algoritmus není odolný proti zlomyslnému uživateli. Takový uživatel může na vstupu zadat vychytrale sestrojenou posloupnost, která algoritmus donutí vybrat v každém kroku pivota nešikovně, takže poběží kvadraticky dlouho. Tomu jsme se přirozeně vyhnuli náhodnou volbou pivota – pro sebezlotřilejší vstup doběhneme v průměru rychle. Hodí se ale také vědět, že i pro pevnou volbu pivota je špatných vstupů málo.

Zavedeme si proto ještě jeden druh složitosti algoritmů, tentokrát opět deterministických (bez náhodného generátoru). Bude to *složitost v průměru přes vstupy.* Jinými slovy algoritmus bude mít pevný průběh, ale budeme mu dávat náhodný vstup a počítat, jak dlouho v průměru poběží.

Co to ale takový náhodný vstup je? U našich dvou problémů to docela dobře vystihuje *náhodná permutace* – vybereme si rovnoměrně náhodně jednu z $n!$ permutací množiny $\{1, 2, \ldots, n\}$.

Jak Quicksort, tak Quickselect se pak budou chovat velmi podobně, jako když měly pevný vstup, ale volily náhodně pivota. Pokud je na vstupu rovnoměrně náhodná permutace, je její prostřední prvek rovnoměrně náhodně vybrané číslo z množiny $\{1, 2, \ldots, n\}$. Vybereme-li ho za pivota a rozdělíme vstup na levou a pravou část, obě části budou opět náhodné permutace, takže se na nich algoritmus bude opět chovat tímto způsobem. Můžeme tedy analýzu z této kapitoly použít i na tento druh průměru se stejným výsledkem.

Cvičení

1. Průměrnou časovou složitost Quicksortu lze spočítat i podobnou úvahou, jakou jsme použili u Quickselectu. Uvažujte pro každý prvek, kolika porovnání se účastní a jak se mění velikosti úseků, v nichž se nachází.

2.* Ještě jeden způsob, jak analyzovat průměrnou složitost Quicksortu, je použitím podobné úvahy jako v důkazu Lemmatu o džbánu. Sestavíme rekurenci pro průměrný počet porovnání: $R(n) = n + \frac{1}{n} \sum_{i=1}^{n} (R(i-1) + R(n-i))$, $R(0) = R(1) = 0$. Dokažte indukcí, že $R(n) \leq 4n \ln n$.

3. *Náhodné stromy:* Uvažujme binární vyhledávací strom, který vznikl postupným vkládáním hodnot $1, \ldots, n$ v náhodném pořadí, bez jakéhokoliv vyvažování. Dokažte, že střední hodnota průměrné hloubky vrcholu je $\mathcal{O}(\log n)$. (Pro jistotu: průměr je obyčejný aritmetický, nijak v něm nefiguruje náhoda; z těchto průměrů pak počítáme střední hodnotu přes všechny možné průběhy algoritmu.) Napovíme, že náhodné stromy souvisí s možnými průběhy Quicksortu.

4. *Úsporný medián:* Nalezněte k-tý nejmenší z n prvků, máte-li k dispozici pouze paměť asymptoticky menší než k. Pokuste se dosáhnout lepší časové složitosti než $\Theta(kN)$.

5. *Eratosthenovo síto* je pradávný algoritmus na hledání prvočísel. Začne se seznamem čísel $2, \ldots, n$. V i-tém kroku zkontroluje číslo i: pokud není škrtnuté, nahlásí ho jako prvočíslo a vyškrtá všechny jeho násobky. Časová složitost tohoto algoritmu je poněkud překvapivě $\mathcal{O}(n \log \log n)$. Zkuste dokázat alespoň slabší odhad $\mathcal{O}(n \log n)$.

11.3 Hešování s přihrádkami

Lidé už dávno zjistili, že práci s velkým množstvím věcí si lze usnadnit tím, že je rozdělíme do několika menších skupin a každou zpracujeme zvlášť. Příklady najdeme všude kolem sebe: Slovník spisovného jazyka českého má díly A až M, N až Q, R až U a V až Ž. Katastrální úřady mají svou působnost vymezenu územím na mapě. Padne-li v Paříži smog, smí v některé dny do centra jezdit jenom auta se sudými registračními čísly, v jiné dny ta s lichými.

Informatici si tuto myšlenku také oblíbili a pod názvem *hešování* ji často používají k uchovávání dat. (Hešuje se i v kryptografii, ale trochu jinak a o tom zde nebude řeč.)

Mějme univerzum $\mathcal{U}$ možných hodnot, konečnou množinu přihrádek $\mathcal{P} = \{0, \ldots, m-1\}$ a *hešovací funkci,* což bude nějaká funkce $h : \mathcal{U} \to \mathcal{P}$, která každému prvku univerza přidělí jednu přihrádku. Chceme-li uložit množinu prvků $X \subseteq \mathcal{U}$, rozstrkáme její prvky do přihrádek: prvek $x \in X$ umístíme do přihrádky $h(x)$. Budeme-li pak hledat nějaký prvek $u \in \mathcal{U}$, víme, že nemůže být jinde než v přihrádce $h(u)$.

Podívejme se na příklad: Univerzum všech celých čísel budeme rozdělovat do 10 přihrádek podle poslední číslice. Jako hešovací funkci tedy použijeme $h(x) = x \bmod 10$. Zkusíme uložit několik slavných letopočtů naší historie: 1212, 935, 1918, 1948, 1968, 1989:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
		1212			935			1918	1989
								1948	
								1968	

Hledáme-li rok 2015, víme, že se musí nacházet v přihrádce 5. Tam je ovšem pouze 935, takže hned odpovíme zamítavě. Hledání roku 2016 je dokonce ještě rychlejší: přihrádka 6 je prázdná. Zato hledáme-li rok 1618, musíme prozkoumat hned 3 hodnoty.

Uvažujme obecněji: kdykoliv máme nějakou hešovací funkci, můžeme si pořídit pole p přihrádek, v každé pak „řetízek“ – spojový seznam hodnot. Tato jednoduchá datová struktura je jednou z možných forem *hešovací tabulky.*

Jakou má hešovací tabulka časovou složitost? Hledání, vkládání i mazání sestává z výpočtu hešovací funkce a projití řetízku v příslušné přihrádce. Pokud bychom uvažovali „ideální hešovací funkci“, kterou lze spočítat v konstantním čase a která zadanou n-prvkovou množinu rozprostře mezi m přihrádek dokonale rovnoměrně, budou mít všechny řetízky n/m prvků. Zvolíme-li navíc počet přihrádek $m = \Theta(n)$, vyjde konstantní délka řetízku, a tím pádem i časová složitost operací.

Praktické hešovací funkce

Ideální hešovací funkce patří spíše do kraje mýtů, podobně jako třeba ideální plyn. Přesto nám to nebrání hešování v praxi používat – ostřílení programátoři znají řadu funkcí, které se pro reálná data chovají „prakticky náhodně“. Autorům této knihy se osvědčily například tyto funkce:

- *Lineární kongruence:* $x \mapsto ax \bmod m$
  Zde m je typicky prvočíslo a a je nějaká dostatečně velká konstanta nesoudělná s m. Často se a nastavuje blízko $0.618m$ (další nečekaná aplikace zlatého řezu z oddílu 1.4).

- *Vyšší bity součinu:* $x \mapsto \lfloor (ax \bmod 2^w)/2^{w-\ell} \rfloor$
  Pokud hešujeme w-bitová čísla do $m = 2^\ell$ přihrádek, vybereme w-bitovou lichou konstantu a. Pak pro každé x spočítáme ax, ořízneme ho na w bitů a z nich vezmeme nejvyšších ℓ. Vzhledem k tomu, že přetečení ve většině programovacích jazyků automaticky ořezává výsledek, stačí k vyhodnocení funkce jedno násobení a bitový posun.

- *Skalární součin:* $x_0, \ldots, x_{d-1} \mapsto (\sum_i a_i x_i) \bmod m$
  Chceme-li hešovat posloupnosti, nabízí se zahešovat každý prvek zvlášť a výsledky sečíst (nebo vyxorovat). Pokud prvky hešujeme lineární kongruencí, je heš celé posloupnosti její skalární součin s vektorem konstant, to vše modulo m. Pozor, nefunguje používat pro všechny prvky tutéž konstantu: pak by výsledek nezávisel na pořadí prvků.

- *Polynom:* $x_0, \ldots, x_{d-1} \mapsto \left(\sum_i a^i x_i\right) \bmod m$
  Tentokrát zvolíme jenom jednu konstantu a a počítáme skalární součin zadané posloupnosti s vektorem $(a^0, a^1, \ldots, a^{d-1})$. Tento typ funkcí bude hrát důležitou roli v Rabinově-Karpově algoritmu na vyhledávání v textu v oddílu 13.4.

U všech čtyř funkcí je experimentálně ověřeno, že dobře hešují nejrůznější druhy dat. Nemusíme se ale spoléhat jen na pokusy: v oddílu 11.5 vybudujeme teorii, pomocí které o chování některých hešovacích funkcí budeme schopni vyslovit exaktní tvrzení.

Pokud chceme hešovat objekty nějakého jiného typu, nejprve je zakódujeme do čísel nebo posloupností čísel. U floating-point čísel se například může hodit hešovat jejich interní

reprezentaci (což je nějaká posloupnost bytů, kterou můžeme považovat za jedno celé číslo).

Přehešovávání

Hešovací tabulky dovedou vyhledávat s průměrně konstantní časovou složitostí, pokud použijeme $\Omega(n)$ přihrádek. Jaký počet přihrádek ale zvolit, když n předem neznáme? Pomůže nám technika amortizovaného nafukování pole z oddílu 9.1.

Na počátku založíme prázdnou hešovací tabulku s nějakým konstantním počtem přihrádek. Kdykoliv pak vkládáme prvek, zkontrolujeme poměr $\alpha = n/m$ – tomu se říká *faktor naplnění* neboli *hustota* tabulky a chceme ho udržet shora omezený konstantou, třeba $\alpha \leq 1$. Pokud už je tabulka příliš plná, zdvojnásobíme m a všechny prvky přehešujeme. Jedno přehešování trvá $\Theta(n)$ a jelikož mezi každými dvěma přehešováními vložíme řádově n prvků, stačí, když každý prvek přispěje konstantním časem. Při mazání prvků můžeme tabulku zmenšovat, ale obvykle nevadí ponechat ji málo zaplněnou.

Sestrojili jsme tedy datovou strukturu pro reprezentaci množiny, která dokáže vyhledávat, vkládat i mazat v průměrně konstantním čase. Pokud předem neumíme odhadnout počet prvků množiny, je v případě vkládání a mazání tento čas pouze amortizovaný.

Cvičení

1. Mějme množinu přirozených čísel a číslo x. Chceme zjistit, zda množina obsahuje dvojici prvků se součtem x.

2. Chceme hešovat řetězce 8-bitových znaků. Výpočet můžeme zrychlit tak, že čtveřice znaků prohlásíme za 32-bitová čísla a zahešujeme posloupnost čtvrtinové délky. Naprogramujte takovou hešovací funkci a nezapomeňte, že délka řetězce nemusí být dělitelná čtyřmi.

3.* *Bloomův filtr* je datová struktura pro přibližnou reprezentaci množiny. Skládá se z pole bitů $B[1 \ldots m]$ a hešovací funkce h, která prvkům univerza přiřazuje indexy v poli. INSERT(x) nastaví $B[h(x)] = 1$. MEMBER(x) otestuje, zda $B[h(x)] = 1$. Vložme nyní do filtru nějakou n-prvkovou množinu M. Pokud $x \in M$, MEMBER(x) vždy odpoví správně. Pokud se ale zeptáme na $x \notin M$, může se stát, že $h(x) = h(y)$ pro nějaké $y \in M$, a MEMBER(x) odpoví špatně. Spočítejte, s jakou pravděpodobností se to pro dané m a n stane.

4.* Spolehlivost Bloomova filtru můžeme zvýšit tak, že si pořídíme k filtrů s různými hešovacími funkcemi. INSERT bude vkládat do všech, MEMBER se zeptá všech a odpoví ANO pouze tehdy, když se na tom všechny filtry shodnou. Je-li pravděpodobnost chyby jednoho filtru p, pak kombinace k filtrů chybuje s pravděpodobností pouhých p^k. Vymyslete, jak nastavit m a k pro případ, kdy chceme ukládat 10^6 prvků s pravděpodobností chyby nejvýše 10^{-9}. Minimalizujte spotřebu paměti.

11.4 Hešování s otevřenou adresací

Ještě ukážeme jeden způsob hešování, který je prostorově úspornější a za příznivých okolností může být i rychlejší. Za tyto výhody zaplatíme složitějším chováním a pracnější analýzou. Tomuto druhu hešování se říká *otevřená adresace.*

Opět si pořídíme pole s m přihrádkami $A[0], \ldots, A[m-1]$, jenže tentokrát se do každé přihrádky vejde jen jeden prvek. Pokud bychom tam potřebovali uložit další, použijeme náhradní přihrádku, bude-li také plná, zkusíme další, a tak dále.

Můžeme si to představit tak, že hešovací funkce každému prvku $x \in \mathcal{U}$ přiřadí jeho *vyhledávací posloupnost* $h(x,0), h(x,1), \ldots, h(x,m-1)$. Ta určuje pořadí přihrádek, do kterých se budeme snažit x vložit. Budeme předpokládat, že posloupnost obsahuje všechna čísla přihrádek v dokonale náhodném pořadí (všechny permutace přihrádek budou stejně pravděpodobné). Vkládání do tabulky bude vypadat následovně:

Algoritmus OpenInsert (vkládání s otevřenou adresací)
Vstup: Prvek $x \in \mathcal{U}$

1. Pro $i = 0, \ldots, m-1$:
2. $\quad j \leftarrow h(x,i)$ ◁ *číslo přihrádky, kterou právě zkoušíme*
3. $\quad$ Pokud $A[j] = \emptyset$:
4. $\quad\quad$ Položíme $A[j] \leftarrow x$ a skončíme.
5. Ohlásíme, že tabulka je už plná.

Při vyhledávání budeme procházet přihrádky $h(x,0)$, $h(x,1)$ a tak dále. Zastavíme se, jakmile narazíme buď na x, nebo na prázdnou přihrádku.

Algoritmus OpenFind (hledání s otevřenou adresací)
Vstup: Prvek $x \in \mathcal{U}$

1. Pro $i = 0, \ldots, m-1$:
2. $\quad j \leftarrow h(x,i)$ ◁ *číslo přihrádky, kterou právě zkoušíme*
3. $\quad$ Pokud $A[j] = x$, ohlásíme, že jsme x našli, a skončíme.
4. $\quad$ Pokud $A[j] = \emptyset$, ohlásíme neúspěch a skončíme.
5. Ohlásíme neúspěch.

Mazání je problematické: kdybychom libovolný prvek odstranili z tabulky, mohli bychom způsobit, že vyhledávání nějakého jiného prvku skončí předčasně, protože narazí na přihrádku, která v okamžiku vkládání byla plná, ale nyní už není. Proto budeme prvky pouze označovat za smazané a až jich bude mnoho (třeba $m/4$), celou strukturu přebudujeme.

Podobně jako u zvětšování tabulky je vidět, že toto přebudovávání nás stojí amortizovaně konstantní čas na smazaný prvek.

Odvodíme, kolik kroků průměrně provedeme při neúspěšném vyhledávání, což je současně počet kroků potřebných na vložení prvku do tabulky. Úspěšné hledání nebude pomalejší.

Věta: Pokud jsou vyhledávací posloupnosti nezávislé náhodné permutace, průměrný počet přihrádek navštívených při neúspěšném hledání činí nejvýše $1/(1-\alpha)$, kde $\alpha = n/m$ je faktor naplnění.

Důkaz: Nechť x je hledaný prvek a $h_1, h_2, \ldots, h_m$ jeho vyhledávací posloupnost. Označme p_i pravděpodobnost toho, že během hledání projdeme alespoň i přihrádek.

Do přihrádky h_1 se jistě podíváme, proto $p_1 = 1$. Jelikož je to náhodně vybraná přihrádka, s pravděpodobností n/m v ní je nějaký prvek (různý od x, neboť vyhledávání má skončit neúspěchem), a tehdy pokračujeme přihrádkou h_2. Proto $p_2 = n/m = \alpha$.

Nyní obecněji: pakliže přihrádky $h_1, \ldots, h_i$ byly obsazené, zbývá $n - i$ prvků a $m - i$ přihrádek. Přihrádka h_{i+1} je tedy obsazena s pravděpodobností $(n-i)/(m-i) \leq n/m$ (to platí, jelikož $n \leq m$). Proto $p_{i+1} \leq p_i \cdot \alpha$. Indukcí dostaneme $p_i \leq \alpha^{i-1}$.

Nyní počítejme střední hodnotu S počtu navštívených přihrádek. Ta je rovna $\sum_i i \cdot q_i$, kde q_i udává pravděpodobnost, že jsme navštívili právě i přihrádek. Jelikož $q_i = p_i - p_{i+1}$, platí

$$S = \sum_{i \geq 1} i \cdot (p_i - p_{i+1}) = \sum_{i \geq 1} p_i \cdot (i - (i-1)) = \sum_{i \geq 1} p_i \leq \sum_{i \geq 1} \alpha^{i-1} = \sum_{i \geq 0} \alpha^i.$$

To je ovšem geometrická řada se součtem $1/(1-\alpha)$. □

Pokud se tedy faktor naplnění přiblíží k jedničce, začne se hledání drasticky zpomalovat. Pokud ale ponecháme alespoň čtvrtinu přihrádek prázdnou, navštívíme během hledání průměrně nanejvýš 4 přihrádky. Opět můžeme použít přehešovávání, abychom tento stav udrželi. Tak dostaneme vyhledávání, vkládání i mazání v amortizovaně konstantním průměrném čase.

Zbývá vymyslet, jak volit prohledávací posloupnosti. V praxi se často používají tyto možnosti:

- *Lineární přidávání:* $h(x,i) = (f(x) + i) \bmod m$,
  kde $f(x)$ je „obyčejná“ hešovací funkce. Využíváme tedy po sobě jdoucí přihrádky. Výhodou je sekvenční přístup do paměti (který je na dnešních počítačích rychlejší), nevýhodou to, že jakmile se vytvoří souvislé bloky obsazených přihrádek, další vkládání se do nich často strefí a bloky stále porostou (viz obrázek 11.2).

Bez důkazu uvádíme, že pro neúspěšné hledání platí pouze slabší odhad průměrného počtu navštívených přihrádek $1/(1-\alpha)^2$, a to pouze, jestliže je f dokonale náhodná. Není-li, chování struktury obvykle degraduje.

- *Dvojité hešování:* $h(x,i) = (f(x) + i \cdot g(x)) \bmod m$,
kde $f : \mathcal{U} \to \{0,\ldots,m-1\}$ a $g : \mathcal{U} \to \{1,\ldots,m-1\}$ jsou dvě různé hešovací funkce a m je prvočíslo. Díky tomu je $g(x)$ vždy nesoudělné s m a posloupnost navštíví každou přihrádku právě jednou. Je známo, že pro dokonale náhodné funkce f a g se dvojité hešování chová stejně dobře, jako při použití plně náhodných vyhledávacích posloupností. Dokonce stačí vybírat f a g ze silně univerzálního systému (viz příští oddíl). Tato tvrzení též ponecháváme bez důkazu.

		42		14	75	36	24	95	17
0	1	2	3	4	5	6	7	8	9

Obrázek 11.2: Hešování podle poslední číslice s lineárním přidáváním. Stav po vložení čísel 75, 36, 14, 42, 24, 95, 17.

Cvičení

1.* Úspěšné hledání v hešovací tabulce s otevřenou adresací je o něco rychlejší než neúspěšné. Spočítejte, kolik přihrádek průměrně navštíví. Předpokládejte, že hledáme náhodně vybraný prvek tabulky.

2. Uvažujme hešování řízené obecnější lineární posloupností $h(x,i) = (f(x) + c \cdot i) \bmod m$, kde c je konstanta nesoudělná s m. Srovnejte jeho chování s obyčejným lineárním přidáváním.

11.5* Univerzální hešování

Zatím jsme hešování používali tak, že jsme si vybrali nějakou pevnou hešovací funkci a „zadrátovali“ ji do programu. Ať už je to jakákoliv funkce, nikdy není těžké najít n čísel, která *zkolidují* v téže přihrádce, takže jejich vkládáním strávíme čas $\Theta(n^2)$. Můžeme spoléhat na to, že vstup takhle nešikovně vypadat nebude, ale co když nám vstupy dodává zvědavý a potenciálně velmi škodolibý uživatel?

Raději při každém spuštění programu zvolíme hešovací funkci náhodně – nepřítel o této funkci nic neví, takže se mu sotva povede vygenerovat dostatečně ošklivý vstup. Nemůžeme ale náhodně vybírat ze všech možných funkcí z $\mathcal{U}$ do $\mathcal{P}$, protože na popis jedné takové funkce bychom potřebovali $\Theta(|\mathcal{U}|)$ čísel. Místo toho se omezíme na nějaký menší systém

funkcí a náhodně vybereme z něj. Aby to fungovalo, musí tento systém být dostatečně bohatý, což zachycuje následující definice.

Značení: Často budeme mluvit o různých množinách přirozených čísel. Proto zavedeme zkratku $[k] = \{0, \ldots, k-1\}$. Množina přihrádek bude typicky $\mathcal{P} = [m]$.

Definice: Systém $\mathcal{H}$ funkcí z univerza $\mathcal{U}$ do $[m]$ nazveme *c-univerzální* pro konstantu $c \geq 1$, pokud pro každé dva různé prvky $x, y \in \mathcal{U}$ platí $\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq c/m$.

Co si pod tím představit? Kdybychom funkci h rovnoměrně náhodně vybírali z úplně všech funkcí z $\mathcal{U}$ do $[m]$, kolidovaly by prvky x a y s pravděpodobností $1/m$. Nezávisle na tom, kolik vyšlo $h(x)$, by totiž bylo všech m možností pro $h(y)$ stejně pravděpodobných. A pokud místo ze všech funkcí vybíráme h z c-univerzálního systému, smí x a y kolidovat nejvýše c-krát častěji.

Navíc budeme chtít, aby šlo funkci $h \in \mathcal{H}$ určit malým množstvím parametrů a na základě těchto parametrů ji pro konkrétní vstup vyhodnotit v konstantním čase. Například můžeme říci, že $\mathcal{H}$ bude systém lineárních funkcí tvaru $x \mapsto ax \bmod m$ pro $\mathcal{U} = [U]$ a $a \in [U]$. Každá taková funkce je jednoznačně určena parametrem a, takže náhodně vybrat funkci je totéž jako náhodně zvolit $a \in [U]$. To zvládneme v konstantním čase, stejně jako vyhodnotit funkci pro dané a a x.

Za chvíli ukážeme, jak nějaký c-univerzální systém sestrojit. Předtím ale dokážeme, že splní naše očekávání.

Lemma: Buď h hešovací funkce náhodně vybraná z nějakého c-univerzálního systému. Nechť $x_1, \ldots, x_n$ jsou navzájem různé prvky univerza vložené do hešovací tabulky a x je libovolný prvek univerza. Potom pro střední počet prvků ležících v téže přihrádce jako x platí:

$$\mathbb{E}[\#i : h(x) = h(x_i)] \leq cn/m + 1.$$

Důkaz: Pro dané x definujeme indikátorové náhodné proměnné:

$$Z_i = \begin{cases} 1 & \text{když } h(x) = h(x_i), \\ 0 & \text{jindy.} \end{cases}$$

Jinými slovy Z_i říká, kolikrát padl prvek x_i do přihrádky $h(x)$, což je buď 0, nebo 1. Celkový počet kolidujících prvků je $Z = \sum_i Z_i$ a díky linearitě střední hodnoty je hledaná hodnota $\mathbb{E}[Z]$ rovna $\sum_i \mathbb{E}[Z_i]$. Přitom $\mathbb{E}[Z_i] = \Pr[Z_i = 1]$, což je podle definice c-univerzálního systému nejvýše c/m, pokud $x_i \neq x$. Pokud $x_i = x$, pak $\mathbb{E}[Z_i] = 1$, což ale může nastat pro nejvýše jedno i. Tedy $\mathbb{E}[Z]$ je nejvýše $cn/m + 1$. □

Důsledek: Nechť k hešování použijeme funkci vybranou rovnoměrně náhodně z nějakého c-univerzálního systému. Pokud už hešovací tabulka obsahuje n prvků, bude příští operace nahlížet do přihrádky s průměrně nanejvýš $cn/m + 1$ prvky. Udržíme-li $m = \Omega(n)$, bude tedy průměrná velikost přihrádky omezena konstantou, takže průměrná časová složitost operace bude také konstantní.

Mějme na paměti, že neprůměrujeme přes možná vstupní data, nýbrž přes možné volby hešovací funkce, takže tvrzení platí pro libovolně škodolibý vstup. Také upozorňujeme, že tyto úvahy vyžadují oddělené přihrádky a nefungují pro otevřenou adresaci.

Ve zbytku tohoto oddílu budeme předpokládat, že čtenář zná základy lineární algebry (vektorové prostory a skalární součin) a teorie čísel (dělitelnost, počítání s kongruencemi a s konečnými tělesy).

Konstrukce ze skalárního součinu

Postupně ukážeme, jak upravit praktické hešovací funkce z oddílu 11.3, aby tvořily univerzální systém. Nejsnáze to půjde se skalárními součiny.

Zvolíme nějaké konečné těleso $\mathbb{Z}_p$ pro prvočíselné p. Pořídíme si $m = p$ přihrádek a očíslujeme je jednotlivými prvky tělesa. Univerzem bude vektorový prostor $\mathbb{Z}_p^d$ všech d-složkových vektorů nad tímto tělesem. Hešovací funkce bude mít tvar skalárního součinu s nějakým pevně zvoleným vektorem $\mathbf{t} \in \mathbb{Z}_p^d$.

Věta: Systém funkcí $\mathcal{S} = \{h_\mathbf{t} \mid \mathbf{t} \in \mathbb{Z}_p^d\}$, kde $h_\mathbf{t}(\mathbf{x}) = \mathbf{t} \cdot \mathbf{x}$, je 1-univerzální.

Důkaz: Mějme nějaké dva různé vektory $\mathbf{x}, \mathbf{y} \in \mathbb{Z}_p^d$. Nechť k je nějaká souřadnice, v níž je $x_k \neq y_k$. Jelikož skalární součin nezáleží na pořadí složek, můžeme složky přečíslovat tak, aby bylo $k = d$.

Nyní volíme $\mathbf{t}$ náhodně po složkách a počítáme pravděpodobnost kolize (rovnost modulo p značíme $\equiv$):

$$\begin{aligned}\Pr_{\mathbf{t}\in\mathbb{Z}_p^d}[h_\mathbf{t}(\mathbf{x}) \equiv h_\mathbf{t}(\mathbf{y})] = \Pr[\mathbf{x}\cdot\mathbf{t} \equiv \mathbf{y}\cdot\mathbf{t}] = \Pr[(\mathbf{x}-\mathbf{y})\cdot\mathbf{t} \equiv 0] = \\ = \Pr\left[\sum_{i=1}^{d}(x_i - y_i)t_i \equiv 0\right] = \Pr\left[(x_d - y_d)t_d \equiv -\sum_{i=1}^{d-1}(x_i - y_i)t_i\right].\end{aligned}$$

Pokud už jsme $t_1, \ldots, t_{d-1}$ zvolili a nyní náhodně volíme t_d, nastane kolize pro právě jednu volbu: Poslední výraz je lineární rovnice tvaru $az = b$ pro nenulové a a ta má v libovolném tělese právě jedno řešení z. Pravděpodobnost kolize je tedy nejvýše $1/p = 1/m$, jak požaduje 1-univerzalita. □

Intuitivně náš důkaz funguje takto: Pro nenulové $x \in \mathbb{Z}_p$ a rovnoměrně náhodně zvolené $a \in \mathbb{Z}_p$ nabývá výraz ax všech hodnot ze $\mathbb{Z}_p$ se stejnou pravděpodobností. Proto se d-tý sčítanec skalárního součinu chová rovnoměrně náhodně. Ať už má zbytek skalárního součinu jakoukoliv hodnotu, přičtením d-tého členu se z něj stane také rovnoměrně rozložené náhodné číslo.

Příklad: Kdybychom chtěli hešovat 32-bitová čísla do cca 250 přihrádek, nabízí se zvolit $p = 257$ a každé číslo rozdělit na 4 části po 8 bitech. Jelikož $2^8 = 256$, můžeme si tyto části vyložit jako 4-složkový vektor nad $\mathbb{Z}_{257}$. Například číslu $123\,456\,789 = 7 \cdot 2^{24} + 91 \cdot 2^{16} + 205 \cdot 2^8 + 21$ odpovídá vektor $\mathbf{x} = (7, 91, 205, 21)$. Pro $\mathbf{t} = (1, 2, 3, 4)$ se tento vektor zahešuje na $\mathbf{x} \cdot \mathbf{t} \equiv 7 \cdot 1 + 91 \cdot 2 + 205 \cdot 3 + 21 \cdot 4 \equiv 7 + 182 + 101 + 84 \equiv 117$.

Poznámka: Jistou nevýhodou této konstrukce je, že počet přihrádek musí být prvočíselný. To by nám mohlo vadit při přehešovávání do dvojnásobně velké tabulky. Zachrání nás ovšem *Bertrandův postulát,* který říká, že mezi m a $2m$ vždy leží alespoň jedno prvočíslo. Pokud budeme zaokrouhlovat počet přihrádek na nejbližší vyšší prvočíslo, máme zaručeno, že pokaždé tabulku nejvýše zčtyřnásobíme, což se stále uamortizuje. Teoreticky nás může brzdit hledání vhodných prvočísel, v praxi si na 64-bitovém počítači pořídíme tabulku 64 prvočísel velikostí přibližně mocnin dvojky.

Konstrukce z lineární kongruence

Nyní se inspirujeme lineární kongruencí $x \mapsto ax \bmod m$. Pro prvočíselné m by stačilo náhodně volit a a získali bychom 1-univerzální systém – jednorozměrnou obdobu předchozího systému se skalárním součinem. My ovšem dáme přednost trochu komplikovanějším funkcím, které zato budou fungovat pro libovolné m.

Budeme se pohybovat v univerzu $\mathcal{U} = [U]$. Pořídíme si nějaké prvočíslo $p \geq U$ a počet přihrádek $m < U$. Budeme počítat lineární funkce tvaru $ax + b$ v tělese $\mathbb{Z}_p$ a výsledek dodatečně modulit číslem m.

Věta: Nechť $h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$. Potom systém funkcí $\mathcal{L} = \{h_{a,b} \mid a, b \in [p], a \neq 0\}$ je 1-univerzální.

Důkaz: Mějme dvě různá čísla $x, y \in [U]$. Nejprve rozmýšlejme, jak se chovají lineární funkce modulo p bez dodatečného modulení číslem m a bez omezení $a \neq 0$. Pro libovolnou dvojici parametrů $(a, b) \in [p]^2$ označme:

$$r = (ax + b) \bmod p,$$
$$s = (ay + b) \bmod p.$$

Každé dvojici $(a, b) \in [p]^2$ tedy přiřadíme nějakou dvojici $(r, s) \in [p]^2$. Naopak každá dvojice (r, s) vznikne z právě jedné dvojice (a, b): podmínky pro a a b dávají soustavu

dvou nezávislých lineárních rovnic o dvou neznámých, která musí mít v libovolném tělese právě jedno řešení. (Explicitněji: Odečtením rovnic dostaneme $r - s \equiv a(x - y)$, což dává jednoznačné a. Dosazením do libovolné rovnice pak získáme jednoznačné b.)

Máme tedy bijekci mezi všemi dvojicemi (a, b) a (r, s). Nezapomínejme ale, že jsme zakázali $a = 0$, což odpovídá zákazu $r = s$.

Nyní vraťme do hry modulení číslem m a počítejme *špatné* dvojice (a, b), pro něž nastane $h_{a,b}(x) = h_{a,b}(y)$. Ty odpovídají dvojicím (r, s) splňujícím $r \equiv s$ modulo m. Pro každé r spočítáme, kolik možných s s ním je kongruentních. Pokud množinu $[p]$ rozdělíme na m-tice, dostaneme $\lceil p/m \rceil$ m-tic, z nichž poslední je neúplná. V každé úplné m-tici leží právě jedno číslo kongruentní s r, v té jedné neúplné nejvýše jedno. Navíc ovšem víme, že $r \neq s$, takže možných s je o jedničku méně.

Celkem tedy pro každé r existuje nanejvýš $\lceil p/m \rceil - 1$ kongruentních s. To shora odhadneme výrazem $(p + m - 1)/m - 1 = (p - 1)/m$. Jelikož možností, jak zvolit r, je přesně p, dostáváme maximálně $p(p - 1)/m$ špatných dvojic (r, s). Mezi dvojicemi (a, b) a (r, s) vede bijekce, takže špatných dvojic (a, b) je stejný počet. Jelikož možných dvojic (a, b) je $p(p-1)$, pravděpodobnost, že vybereme nějakou špatnou, je nejvýše $1/m$. Systém je tedy 1-univerzální. □

Konstrukce z vyšších bitů součinu

Nakonec ukážeme univerzalitu hešovacích funkcí založených na vyšších bitech součinu. Univerzum budou tvořit všechna w-bitová čísla, tedy $\mathcal{U} = [2^w]$. Hešovat budeme do $m = 2^\ell$ přihrádek. Hešovací funkce pro klíč x vypočte součin ax a „vykousne“ z něj bity na pozicích $w - \ell$ až $w - 1$. (Bity číslujeme obvyklým způsobem od nuly, tedy i-tý bit má váhu 2^i.)

Věta: Nechť $h_a(x) = \lfloor (ax \bmod 2^w)/2^{w-\ell} \rfloor$. Potom systém funkcí $\mathcal{M} = \{h_a \mid a \in [2^w],$ a liché$\}$ je 2-univerzální.

Důkaz: Mějme nějaké dva různé klíče x a y. Bez újmy na obecnosti předpokládejme, že $x < y$. Označme i pozici nejnižšího bitu, v němž se x od y liší. Platí tedy $y - x = z \cdot 2^i$, kde z je nějaké liché číslo.

Chceme počítat pravděpodobnost, že $h_a(x) = h_a(y)$ pro rovnoměrně náhodné a. Jelikož a je liché, můžeme ho zapsat jako $a = 2b + 1$, kde $b \in [2^{w-1}]$, a volit rovnoměrně náhodné b.

Prozkoumejme, jak se chová výraz $a(y - x)$. Můžeme ho zapsat takto:

$$a(y - x) = (2b + 1)(z \cdot 2^i) = bz \cdot 2^{i+1} + z \cdot 2^i.$$

Podívejme se na binární zápis:

- Člen $z \cdot 2^i$ má v bitech 0 až $i-1$ nuly, v bitu i jedničku a vyšší bity mohou vypadat jakkoliv, ale nezávisí na b.
- Člen $bz \cdot 2^{i+1}$ má bity 0 až i nulové. V bitech $i+1$ až $w+i$ leží $bz \bmod 2^w$, které nabývá všech hodnot z $[2^w]$ se stejnou pravděpodobností: Jelikož z je liché, a tedy nesoudělné s 2^w, má kongruence $bz \equiv d \pmod{2^w}$ pro každé d právě jedno řešení b (viz cvičení 1.3.5). Všechna b nastávají se stejnou pravděpodobností, takže všechna d také.
- Součet těchto dvou členů tedy musí mít bity 0 až $i-1$ nulové, v bitu i jedničku a v bitech $i+1$ až $w+i$ rovnoměrně náhodné číslo z $[2^w]$ (sečtením rovnoměrně náhodného čísla s čímkoliv nezávislým vznikne modulo 2^w opět rovnoměrně náhodné číslo). O vyšších bitech nic neříkáme.

Vraťme se k rovnosti $h_a(x) = h_b(y)$. Ta nastane, pokud se čísla ax a ay shodují v bitech $w-\ell$ až $w-1$. Využijeme toho, že $ay = ax + a(y-x)$, takže ax a ay se určitě shodují v bitech 0 až $i-1$ a neshodují v bitu i. Rozlišíme dva případy:

- Pokud $i \geq w-\ell$, bit i patří mezi bity vybrané do výsledku hešovací funkce, takže $h_a(x)$ a $h_a(y)$ se určitě liší.
- Je-li $i < w-\ell$, pak jsou všechny vybrané bity v $a(y-x)$ rovnoměrně náhodné. Kolize x s y nastane, pokud tyto bity budou všechny nulové, nebo když budou jedničkové a navíc v součtu $ax + a(y-x)$ nastane přenos z nižšího řádu. Obojí má pravděpodobnost nejvýš $2^{-\ell}$, takže kolize nastane s pravděpodobností nejvýš $2^{1-\ell} = 2/m$. □

Vzorkování a silná univerzalita*

Hešovací funkce se hodí i pro jiné věci, než je reprezentace množin. Představte si počítačovou síť, v níž putují pakety přes velké množství routerů. Chtěli byste sledovat, co se v síti děje, třeba tak, že necháte každý router zaznamenávat, jaké pakety přes něj projdou. Jenže na to je paketů příliš mnoho. Nabízí se pakety *navzorkovat,* tedy vybrat si z nich jen malou část a tu sledovat.

Vzorek ale nemůžeme vybírat náhodně: kdyby si každý router hodil korunou, zda daný paket zaznamená, málokdy u jednoho paketu budeme znát celou jeho cestu. Raději si pořídíme hešovací funkci h, která paketu p přiřadí nějaké číslo $h(p) \in [m]$. Kdykoliv router přijme paket, zahešuje ho a pokud vyjde méně než nějaký parametr t, paket zaznamená. To nastane s pravděpodobností t/m a shodnou se na tom všechny routery po cestě.

Nabízí se zvolit funkci h náhodně z nějakého c-univerzálního systému. Jenže pak nebudeme vzorkovat spravedlivě: například v našem systému $\mathcal{S}$ odvozeném ze skalárního součinu

padne nulový vektor vždy do přihrádky 0, takže ho pro každé t vybereme do vzorku. Aby vzorkování fungovalo, budeme potřebovat silnější definici univerzality.

Definice: Systém $\mathcal{H}$ funkcí z univerza $\mathcal{U}$ do $[m]$ nazveme *silně c-univerzální* pro konstantu $c \geq 1$, pokud pro každé dva různé prvky $x, y \in \mathcal{U}$ a každé dvě přihrádky $a, b \in [m]$ (ne nutně různé) platí $\Pr_{h\in\mathcal{H}}[h(x) = a \wedge h(y) = b] \leq c/m^2$.

Podobně jako u obyčejné (neboli slabé) univerzality, i zde vlastně říkáme, že funkce náhodně vybraná z daného systému je nejvýše c-krát horší než úplně náhodná funkce: ta by s pravděpodobností $1/m$ zahešovala x do přihrádky a a nezávisle na tom y do přihrádky b.

Důsledek: Pro prvek x a konkrétní přihrádku a je $\Pr_h[h(x) = a] \leq c/m$. Proto náš způsob vzorkování každý paket zaznamená s pravděpodobností nejvýše c-krát větší, než by odpovídalo rovnoměrně náhodnému výběru.

Navíc ukážeme, že malými úpravami již popsaných systémů funkcí z nich můžeme udělat silně univerzální. Pro systém $\mathcal{L}$ to vzápětí dokážeme, ostatní nechme jako cvičení 7 a 8.

Věta: Definujme $h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$. Potom systém funkcí $\mathcal{L}' = \{h_{a,b} \mid a, b \in [p]\}$ je silně 4-univerzální.

Důkaz: Z důkazu 1-univerzality systému $\mathcal{L}$ víme, že pro pevně zvolené x a y existuje bijekce mezi dvojicemi parametrů $(a, b) \in \mathbb{Z}_p^2$ a dvojicemi $(r, s) = ((ax+b) \bmod p, (ay+b) \bmod p)$. Pokud volíme parametry rovnoměrné náhodně, dostáváme i (r, s) rovnoměrně náhodně. Zbývá ukázat, že závěrečným modulením m se rovnoměrnost příliš nepokazí.

Potřebujeme, aby pro každé $i, j \in [m]$ platilo ($\equiv$ značí kongruenci modulo m):

$$\Pr_{r,s}[r \equiv i \wedge s \equiv j] \leq \frac{4}{m^2}.$$

Jelikož $r \equiv i$ a $s \equiv j$ jsou nezávislé jevy, navíc se stejnou pravděpodobností, stačí ověřit, že $\Pr_r[r \equiv i] \leq 2/m$.

Čísel $r \in [p]$ kongruentních s i může být nejvýše $\lceil p/m \rceil = \lfloor (p + m - 1)/m \rfloor \leq (p + m - 1)/m = (p - 1)/m + 1$. Využijeme-li navíc toho, že $m \leq p$, získáme:

$$\Pr_r[r \equiv i] = \frac{\#r : r \equiv i}{p} \leq \frac{p-1}{p \cdot m} + \frac{1}{p} \leq \frac{1}{m} + \frac{1}{m} = \frac{2}{m}.$$

Tím jsme větu dokázali. □

Cvičení

1. Dostali jste hešovací funkci $h : [U] \to [m]$. Pokud o této funkci nic dalšího nevíte, kolik vyhodnocení funkce potřebujete, abyste našli k-tici prvků, které se všechny zobrazí do téže přihrádky?

2. Ukažte, že pokud bychom v „lineárním“ systému $\mathcal{L}$ zafixovali parametr b na nulu, už by nebyl 1-univerzální, ale pouze 2-univerzální. Totéž by se stalo, pokud bychom připustili nulové a.

3. Ukažte, že pokud bychom v „součinovém“ systému $\mathcal{M}$ připustili i sudá a, už by nebyl c-univerzální pro žádné c.

4. Studujme chování polynomiálního hešování z minulého oddílu. Uvažujme funkce $h_a : \mathbb{Z}_p^d \to \mathbb{Z}_p$, přičemž $h_a(x_0, \ldots, x_{d-1}) = \sum_i x_i a^i \bmod p$. Dokažte, že systém $\mathcal{P} = \{h_a \mid a \in \mathbb{Z}_p\}$ je d-univerzální. Mohou se hodit vlastnosti polynomů z oddílu 17.1.

5. Dokažte, že je-li nějaký systém funkcí silně c-univerzální, pak je také (slabě) c-univerzální.

6.* O systému $\mathcal{L}'$ jsme dokázali, že je silně c-univerzální pro $c = 4$. Rozmyslete si, že pro žádné menší c to neplatí.

7.* Ukažte, že systém $\mathcal{S}$ odvozený ze skalárního součinu není silně c-univerzální pro žádné c. Ovšem pokud ho rozšíříme na $\mathcal{S}' = \{h_{\mathbf{t},r} \mid \mathbf{t} \in \mathbb{Z}_p^d, r \in \mathbb{Z}_p\}$, kde $h_{\mathbf{t},r}(\mathbf{x}) = \mathbf{t} \cdot \mathbf{x} + r$, už bude silně 1-univerzální.

8.** Podobně systém $\mathcal{M}$ není silně c-univerzální pro žádné c, ale jde to jednoduchou úpravou zachránit. Definujeme $\mathcal{M}' = \{h_{a,b} \mid a, b \in [2^{w+\ell}]\}$, přičemž $h_{a,b}(x) = \lfloor ((ax + b) \bmod 2^{w+\ell})/2^w \rfloor$. Jinak řečeno výsledkem hešovací funkce jsou bity w až $w + \ell - 1$ hodnoty $ax + b$. Dokažte, že systém $\mathcal{M}'$ je silně 2-univerzální.

9. Nechť $\mathcal{H}$ je nějaký systém funkcí z $\mathcal{U}$ do $[m]$. Pro $m' \leq m$ definujme $\mathcal{H}^* = \{x \mapsto h(x) \bmod m' \mid h \in \mathcal{H}\}$. Dokažte, že je-li $\mathcal{H}$ silně c-univerzální, pak $\mathcal{H}^*$ je $2c$-univerzální a silně $4c$-univerzální. Jakou roli hraje tento fakt v rozboru systémů $\mathcal{L}$ a $\mathcal{L}'$? Lze dosáhnout lepších konstant než $2c$ a $4c$?

12 Dynamické programování

12 Dynamické programování

V této kapitole prozkoumáme ještě jednu techniku návrhu algoritmů, která je založená na rekurzivním rozkladu problému na podproblémy. V tom je podobná metodě Rozděl a panuj, ovšem umí využít toho, že se podproblémy během rekurze opakují. Proto v mnoha případech vede na mnohem rychlejší algoritmy. Říká se jí poněkud tajemně *dynamické programování.*⟨1⟩

12.1 Fibonacciho čísla podruhé

Princip dynamického programování si nejprve vyzkoušíme na triviálním příkladu. Budeme počítat Fibonacciho čísla, se kterými jsme se už potkali v úvodní kapitole. Začneme přímočarým rekurzivním algoritmem na výpočet n-tého Fibonacciho čísla F_n, který postupuje přesně podle definice $F_n = F_{n-1} + F_{n-2}$.

Algoritmus FIB(n)
1. Pokud $n \leq 1$, vrátíme n.
2. Jinak vrátíme FIB($n-1$) + FIB($n-2$).

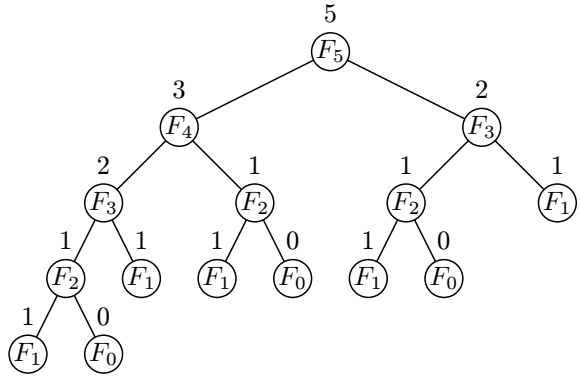
Zkusme zjistit, jakou časovou složitost tento algoritmus má. Sledujme strom rekurze na následujícím obrázku. V jeho kořeni počítáme F_n, v listech F_0 a F_1, vnitřní vrcholy odpovídají výpočtům čísel F_k pro $2 \leq k < n$.

Libovolný vnitřní vrchol přitom vrací součet hodnot ze svých synů. Pokud tento argument zopakujeme, dostaneme, že hodnota vnitřního vrcholu je rovna součtu hodnot všech listů ležících pod ním. Speciálně tedy F_n v kořeni musí být rovno součtu všech listů. Z každého listu přitom vracíme buďto 0 nebo 1, takže abychom nasčítali F_n, musí se ve stromu celkově nacházet alespoň F_n listů. Z cvičení 1.4.4 víme, že $F_n \approx 1.618^n$, takže strom rekurze má přinejmenším exponenciálně mnoho listů a celý algoritmus se plouží exponenciálně pomalu.

Nyní si všimněme, že funkci FIB voláme pouze pro argumenty z rozsahu 0 až n. Jediné možné vysvětlení exponenciální časové složitosti tedy je, že si necháváme mnohokrát spočítat totéž. To je vidět i na obrázku: F_2 vyhodnocujeme dvakrát a F_1 dokonce čtyřikrát.

Zkusme tomu zabránit. Pořídíme si tabulku T a budeme do ní vyplňovat, která Fibonacciho čísla jsme už spočítali a jak vyšly jejich hodnoty. Při každém volání rekurzivní

⟨1⟩ Legenda říká, že s tímto názvem přišel Richard Bellman, když v 50. letech pracoval v americkém armádním výzkumu a potřeboval nadřízeným vysvětlit, čím se vlastně zabývá. Programováním se tehdy mínilo zejména plánování (třeba postupu výroby) a Bellman zkoumal vícekrokové plánování, v němž optimální volba každého kroku záleží na předchozích krocích – proto dynamické.



Obrázek 12.1: Rekurzivní výpočet Fibonacciho čísel

funkce se pak podíváme do tabulky. Pokud již výsledek známe, rovnou ho vrátíme; v opačném případě ho poctivě spočítáme a hned uložíme do tabulky. Upravený algoritmus bude vypadat následovně.

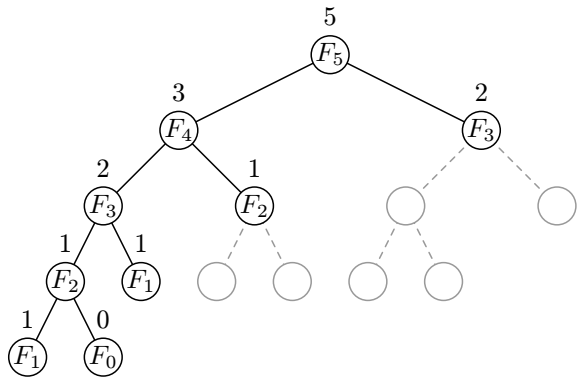
Algoritmus FIB2(n)

1. Je-li $T[n]$ definováno, vrátíme $T[n]$.
2. Pokud $n \leq 1$, položíme $T[n] \leftarrow n$.
3. Jinak položíme $T[n] \leftarrow \text{FIB2}(n-1) + \text{FIB2}(n-2)$.
4. Vrátíme $T[n]$.

Jak se změnila časová složitost? K rekurzi nyní dojde jedině tehdy, vyplňujeme-li políčko tabulky, v němž dosud nic nebylo. To se může stát nejvýše $(n+1)$-krát, z toho dvakrát triviálně (pro F_0 a F_1), takže strom rekurze má nejvýše n vnitřních vrcholů. Pod každým z nich leží nejvýše 2 listy, takže celkově má strom nanejvýš $3n$ vrcholů. V každém z nich trávíme konstantní čas, celkově běží funkce FIB2 v čase $\mathcal{O}(n)$.

Ve stromu rekurze jsme tedy prořezali opakující se větve, až zbylo $\mathcal{O}(n)$ vrcholů. Jak to dopadne pro F_5, vidíme na obrázku 12.2.

Nakonec si uvědomíme, že tabulku mezivýsledků T nemusíme vyplňovat rekurzivně. Jelikož k výpočtu $T[k]$ potřebujeme pouze $T[k-1]$ a $T[k-2]$, stačí ji plnit v pořadí $T[0], T[1], T[2], \ldots$ a vždy budeme mít k dispozici všechny hodnoty, které v daném okamžiku potřebujeme. Dostaneme následující nerekurzivní algoritmus.



Obrázek 12.2: Prořezaný strom rekurze po zavedení tabulky

Algoritmus FIB3(n)

1. $T[0] \leftarrow 0$, $T[1] \leftarrow 1$
2. Pro $k = 2, \ldots, n$:
3. $\quad T[k] \leftarrow T[k-1] + T[k-2]$
4. Vrátíme $T[n]$.

Funkci FIB3 jsme pochopitelně mohli vymyslet přímo, bez úvah o rekurzi. Postup, který jsme si předvedli, ovšem funguje i v méně přímočarých případech. Zkusme proto shrnout, co jsme udělali.

Princip dynamického programování:

- Začneme s rekurzivním algoritmem, který je exponenciálně pomalý.
- Odhalíme opakované výpočty stejných podproblémů.
- Pořídíme si tabulku a budeme si v ní pamatovat, které podproblémy jsme už vyřešili. Tím prořežeme strom rekurze a vznikne rychlejší algoritmus. Tomuto přístupu se často říká *kešování* a tabulce *keš* (anglicky cache).[2]
- Uvědomíme si, že keš lze vyplňovat bez rekurze, zvolíme-li vhodné pořadí podproblémů. Tím získáme stejně rychlý, ale jednodušší algoritmus.

[2] Cache v angličtině znamená obecně skrýš, třeba tu, kam si veverka schovává oříšky. V informatice se tak říká různým druhům paměti na často používaná data. Je-li řeč o zrychlování rekurze, používá se též poněkud krkolomný termín *memoizace* – *memo* je zkrácenina z latinského *memorandum* a dnes značí libovolnou poznámku.

Cvičení

1. Spočítejte, kolik přesně vrcholů má strom rekurze funkce Fib a dokažte, že časová složitost této funkce činí $\Theta(\tau^n)$, kde $\tau = (1 + \sqrt{5})/2$ je zlatý řez.

12.2 Vybrané podposloupnosti

Metodu dynamického programování nyní předvedeme na méně triviálním příkladu. Dostaneme posloupnost $x_1, \ldots, x_n$ celých čísel a chceme z ní škrtnout co nejméně prvků tak, aby zbývající prvky tvořily rostoucí posloupnost. Jinak řečeno, chceme najít *nejdelší rostoucí podposloupnost (NRP)*. Tu můžeme formálně popsat jako co nejdelší posloupnost indexů $i_1, \ldots, i_k$ takovou, že $1 \leq i_1 < \ldots < i_k \leq n$ a $x_{i_1} < \ldots < x_{i_k}$.

Například v následující posloupnosti je jedna z NRP vyznačena tučně.

i	1	2	3	4	5	6	7	8	9	10	11	12	13
x_i	**3**	**14**	**15**	92	65	**35**	89	79	32	**38**	46	26	**43**

Rekurzivní řešení

Nabízí se použít hladový algoritmus: začneme prvním prvkem posloupnosti a pokaždé budeme přidávat nejbližší další prvek, který je větší. Pro naši ukázkovou posloupnost bychom tedy začali $3, 14, 15, 92$ a dál bychom už nemohli přidat žádný další prvek. Optimální řešení je ale delší.

Problém byl v tom, že jsme z možných pokračování podposloupnosti (tedy čísel větších než poslední přidané a ležících napravo od něj) zvolili hladově to nejbližší. Pokud místo toho budeme zkoušet všechna možná pokračování, dostaneme rekurzivní algoritmus, který bude korektní, byť pomalý.

Jeho jádrem bude rekurzivní funkce Nrp(i). Ta pro dané i spočítá maximální délku rostoucí podposloupnosti začínající prvkem x_i. Udělá to tak, že vyzkouší všechna x_j navazující na x_i (tedy $j > i$ a $x_j > x_i$) a pro každé z nich se zavolá rekurzivně. Z možných pokračování si pak vybere to, které dá celkově nejlepší výsledek.

Všimněme si, že rekurze se přirozeně zastaví pro $i = n$: tehdy totiž cyklus neproběhne ani jednou a funkce se ihned vrátí s výsledkem 1.

Řešení původní úlohy získáme tak, že zavoláme Nrp(i) postupně pro $i = 1, \ldots, n$ a vypočteme maximum z výsledků. Probereme tedy všechny možnosti, kterým prvkem může optimální řešení začínat. Elegantnější ovšem je dodefinovat $x_0 = -\infty$. Tím získáme prvek, který se v optimálním řešení zaručeně vyskytuje, takže postačí zavolat Nrp(0).

Algoritmus NRP(i) (nejdelší rostoucí podposloupnost rekurzivně)
Vstup: Posloupnost $x_i, \ldots, x_n$

1. $d \leftarrow 1$
2. Pro $j = i+1, \ldots, n$:
3. Je-li $x_j > x_i$:
4. $d \leftarrow \max(d, 1 + \text{NRP}(j))$

Výstup: Délka d nejdelší rostoucí podposloupnosti v $x_i, \ldots, x_n$

Tento algoritmus je korektní, nicméně má exponenciální časovou složitost: pokud je vstupní posloupnost sama o sobě rostoucí, projdeme během výpočtu úplně všechny podposloupnosti a těch je 2^n. Pro každý prvek si totiž nezávisle na ostatních můžeme vybrat, zda v podposloupnosti leží.

Podobně jako u příkladu s Fibonacciho čísly nás zachrání, když si budeme pamatovat, co jsme už spočítali, a nebudeme to počítat znovu. Funkci NRP totiž můžeme zavolat pouze pro $n+1$ různých argumentů. Pokaždé v ní strávíme čas $\mathcal{O}(n)$, takže celý algoritmus poběží v příjemném čase $\mathcal{O}(n^2)$.

Iterativní řešení

Sledujme dále osvědčený postup. Rekurze se můžeme zbavit a tabulku vyplňovat postupně od největšího i k nejmenšímu. Budeme tedy počítat $T[i]$, což bude délka té nejdelší ze všech rostoucích podposloupností začínajících prvkem x_i.

Algoritmus NRP2 (nejdelší rostoucí podposloupnost iterativně)
Vstup: Posloupnost $x_1, \ldots, x_n$

1. $x_0 \leftarrow -\infty$
2. Pro $i = n, n-1, \ldots, 0$: ◁ *všechny možné začátky NRP*
3. $T[i] \leftarrow 1$
4. $P[i] \leftarrow 0$ ◁ *bude se později hodit pro výpis řešení*
5. Pro $j = i+1, \ldots, n$: ◁ *všechna možná pokračování*
6. Pokud $x_i < x_j$ a $T[i] < 1 + T[j]$: ◁ *máme lepší řešení*
7. $T[i] \leftarrow 1 + T[j]$
8. $P[i] \leftarrow j$

Výstup: Délka $T[0]$ nejdelší rostoucí podposloupnosti

Tento algoritmus běží také v kvadratickém čase. Jeho průběh na naší ukázkové posloupnosti ilustruje obrázek 12.3. Všimněte si, že algoritmus našel jiné optimální řešení, než jakého jsme si prve všimli my.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
x_i	$-\infty$	**3**	**14**	**15**	92	65	**35**	89	79	32	**38**	**46**	26	43
$T[i]$	7	6	5	4	1	2	3	1	1	3	2	1	2	1
$P[i]$	1	2	3	6	0	7	10	0	0	10	11	0	13	0

Obrázek 12.3: Průběh výpočtu algoritmu NRP2

Korektnost algoritmu můžeme dokázat zpětnou indukcí podle i. K tomu se nám hodí nahlédnout, že začíná-li optimální řešení pro vstup $x_i, \ldots, x_n$ dvojicí x_i, x_j, pak z něj odebráním x_i vznikne optimální řešení pro kratší vstup $x_j, \ldots, x_n$ začínající x_j. Kdyby totiž existovalo lepší řešení pro kratší vstup, mohli bychom ho rozšířit o x_i a získat lepší řešení pro původní vstup. Této vlastnosti se říká *optimální substruktura* a už jsme ji potkali například u nejkratších cest v grafech.

Zbývá domyslet, jak kromě délky NRP nalézt i posloupnost samu. K tomu nám pomůže, že kdykoliv jsme spočítali $T[i]$, uložili jsme do $P[i]$ index druhého prvku příslušné optimální podposloupnosti (prvním prvkem je vždy x_i). Proto $P[0]$ říká, jaký prvek je v optimálním řešení celé úlohy první, $P[P[0]]$ udává druhý a tak dále. Opět to funguje analogicky s hledáním nejkratší cesty třeba prohledáváním do šířky: tam jsme si pamatovali předchůdce každého vrcholu a pak zpětným průchodem rekonstruovali cestu.

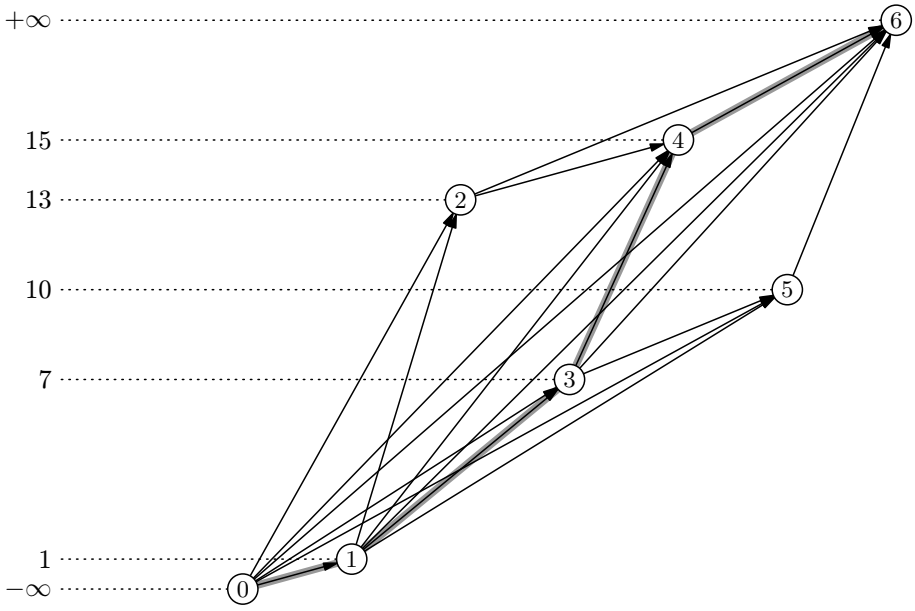
Grafový pohled

Podobnost s hledáním cest v grafech není náhodná – celou naši úlohu totiž můžeme přeformulovat do řeči grafů. Sestrojíme orientovaný graf, jehož vrcholy budou prvky $x_0, \ldots, x_{n+1}$, přičemž dodefinujeme $x_0 = -\infty$ a $x_{n+1} = +\infty$. Hrana povede z x_i do x_j tehdy, mohou-li x_i a x_j sousedit v rostoucí podposloupnosti, čili pokud $i < j$ a současně $x_i < x_j$. (Na obrázku tyto hrany vedou „doprava nahoru“.)

Každá rostoucí podposloupnost pak odpovídá nějaké cestě v tomto grafu. Chceme proto nalézt nejdelší cestu. Ta bez újmy na obecnosti začíná v x_0 a končí v x_{n+1}.

Náš graf má $\Theta(n)$ vrcholů a $\mathcal{O}(n^2)$ hran. Navíc je acyklický a pořadí vrcholů $x_0, \ldots, x_{n+1}$ je topologické. Nejdelší cestu tedy můžeme najít v čase $\mathcal{O}(n^2)$ indukcí podle topologického uspořádání (podrobněji viz oddíl 5.8).

Výsledný algoritmus je přitom náramně podobný našemu NRP2: vnější cyklus prochází pozpátku topologickým pořadím, vnitřní cyklus zkoumá hrany z vrcholu x_i a $T[i]$ můžeme interpretovat jako délku nejdelší cesty z x_i do x_{n+1}. To je poměrně typické: dynamické programování je často ekvivalentní s hledáním cesty ve vhodném grafu. Někdy je jednodušší nalézt tento graf, jindy zase k algoritmu dojít „převrácením“ rekurze.



Obrázek 12.4: Graf reprezentující posloupnost
$-\infty, 1, 13, 7, 15, 10, +\infty$ a jedna z nejdelších cest

Rychlejší algoritmus

Kvadratické řešení je jistě lepší než exponenciální, ale můžeme ho ještě zrychlit. Výběr maxima hrubou silou totiž můžeme nahradit použitím šikovné datové struktury. Ta si bude pro všechna zpracovaná x_i pamatovat dvojice $(x_i, T[i])$, přičemž x_i slouží jako klíč a $T[i]$ jako hodnota přiřazená tomuto klíči.

Algoritmus Nrp2 v každém průchodu vnějšího cyklu uvažuje jedno x_i. Výpočet vnitřního cyklu odpovídá tomu, že v datové struktuře hledáme největší z hodnot přiřazených klíčům z intervalu $(x_i, +\infty)$. Následně vložíme novou dvojici s klíčem x_i. Jednoduchá modifikace vyvážených vyhledávacích stromů (cvičení 8.2.4) zvládne obě tyto operace v čase $\Theta(\log n)$. (Technický detail: Naše klíče se mohou opakovat. Tehdy stačí zapamatovat si největší z hodnot.)

Jeden průchod vnějšího cyklu pak zvládneme v čase $\Theta(\log n)$, takže celý algoritmus poběží v $\Theta(n \log n)$. Jiný stejně rychlý algoritmus odvodíme ve cvičení 4.

Cvičení

1. *Kopcem* nazveme podposloupnost, která nejprve roste a pak klesá. Vymyslete algoritmus, který v zadané posloupnosti nalezne nejdelší kopec.

2. NRP nemusí být jednoznačně určena. Jak spočítat, kolik různých NRP obsahuje zadaná posloupnost?

3. Pokud existuje více NRP, jakou význačnou vlastnost má ta, kterou najde algoritmus Nrp2?

4. Prozkoumejme jiný přístup ke hledání nejdelší rostoucí podposloupnosti. Zadanou posloupnost budeme procházet zleva doprava. Pro již zpracovanou část si budeme udržovat čísla $K[i]$ udávající, jakou nejmenší hodnotou může končit rostoucí podposloupnost délky i. Nahlédněte, že $K[i] < K[i+1]$. Ukažte, že rozšíříme-li vstup o další prvek x, změní se $\mathcal{O}(1)$ hodnot $K[i]$ a k jejich nalezení stačí nalézt binárním vyhledáváním, kam do posloupnosti K patří x. Z toho získejte algoritmus o složitosti $\Theta(n \log n)$.

5. Mějme posloupnost n knih. Každá kniha má nějakou šířku s_i a výšku v_i. Knihy chceme naskládat do knihovny s nějakým počtem polic tak, abychom dodrželi abecední pořadí. Prvních několik knih tedy půjde na první polici, další část na druhou polici, a tak dále. Máme zadanou šířku knihovny S a chceme rozmístit police tak, aby se do nich vešly všechny knihy a celkově byla knihovna co nejnižší. Tloušťku polic a horní a spodní desky přitom zanedbáváme.

6. Podobně jako v předchozím cvičení chceme navrhnout knihovnu, jež pojme dané knihy. Tentokrát ovšem máme zadanou maximální výšku knihovny a chceme najít minimální možnou šířku. Pokud vám to pomůže, předpokládejte, že všechny knihy mají jednotkovou šířku.

7. Dešifrovali jsme tajnou depeši, ale chybí v ní mezery. Známe však slovník všech slov, která se v depeši mohou vyskytnout. Chceme tedy rozdělit depeši na co nejméně slov ze slovníku.

8. Grafový pohled na dynamické programování funguje i pro Fibonacciho čísla. Ukažte, jak pro dané n sestrojit graf na $\mathcal{O}(n)$ vrcholech, v němž bude existovat právě F_n cest mezi startem a cílem. Jak tento graf souvisí se stromem rekurze algoritmu Fib?

12.3 Editační vzdálenost

Pokud ve slově `koule` uděláme překlep, vznikne `boule`, nebo třeba `kdoule`. Kolik překlepů je potřeba, aby z `poutník`a vznikl `potemník`? Podobné otázky vedou ke zkoumání editační vzdálenosti řetězců nebo obecně posloupností.

Definice: *Editační operací* na řetězci nazveme vložení, smazání nebo změnu jednoho znaku. *Editační vzdálenost*[3] řetězců $x = x_1 \dots x_n$ a $y = y_1 \dots y_m$ udává, kolik nejméně editačních operací je potřeba, abychom z prvního řetězce vytvořili druhý. Budeme ji značit $L(x, y)$.

V nejkratší posloupnosti operací se každého znaku týká nejvýše jedna editační operace, takže operace lze vždy uspořádat „zleva doprava“. Můžeme si tedy představit, že procházíme řetězcem x od začátku do konce a postupně ho přetváříme na řetězec y.

Rekurzivní řešení

Zkusme rozlišit případy podle toho, jaká operace nastane v optimální posloupnosti na samém začátku řetězce:

- Pokud $x_1 = y_1$, můžeme první znak ponechat beze změny.
  Tehdy $L(x, y) = L(x_2 \dots x_n, y_2 \dots y_m)$.
- Znak x_1 změníme na y_1. Pak $L(x, y) = 1 + L(x_2 \dots x_n, y_2 \dots y_m)$.
- Znak x_1 smažeme. Tehdy $L(x, y) = 1 + L(x_2 \dots x_n, y_1 \dots y_m)$.
- Na začátek vložíme y_1. Tehdy $L(x, y) = 1 + L(x_1 \dots x_n, y_2 \dots y_m)$.

Pokaždé tedy $L(x, y)$ závisí na vzdálenosti nějakých *suffixů* řetězců x a y. Kdybychom tyto vzdálenosti znali, mohli bychom snadno rozpoznat, která z uvedených čtyř možností nastala – byla by to ta, z níž vyjde nejmenší $L(x, y)$.

Pokud vzdálenosti suffixů neznáme, vypočítáme je rekurzivně. Zastavíme se v případech, kdy už je jeden z řetězců prázdný – tehdy je evidentně vzdálenost rovna délce druhého řetězce.

Z toho vychází následující algoritmus. Pro výpočet $L(x, y)$ postačí zavolat $\textsc{Edit}(1, 1)$.

Algoritmus $\textsc{Edit}(i, j)$ (editační vzdálenost řetězců rekurzivně)
Vstup: Řetězce $x_i \dots x_n$ a $y_j \dots y_m$

1. Pokud $i > n$, vrátíme $m - j + 1$. ◁ *jeden z řetězců už skončil*
2. Pokud $j > m$, vrátíme $n - i + 1$.
3. $\ell_z \leftarrow \textsc{Edit}(i + 1, j + 1)$ ◁ *ponechání či změna znaku*
4. Pokud $x_i \neq y_j$: $\ell_z \leftarrow \ell_z + 1$.
5. $\ell_s \leftarrow \textsc{Edit}(i + 1, j) + 1$ ◁ *smazání znaku*
6. $\ell_v \leftarrow \textsc{Edit}(i, j + 1) + 1$ ◁ *vložení znaku*
7. Vrátíme $\min(\ell_z, \ell_s, \ell_v)$

Výstup: Editační vzdálenost $L(x_i \dots x_n, y_j \dots y_m)$

[3] Někdy též *Levenštejnova vzdálenost* podle Vladimira Josifoviče Levenštejna, který ji zkoumal okolo roku 1965. Odtud značení $L(x, y)$.

Algoritmus je zjevně korektní, nicméně může běžet exponenciálně dlouho (třeba pro $x = y = \texttt{aaa}\ldots\texttt{a}$). Opět nás zachrání, že funkci Edit můžeme zavolat jen s $(n+1)(m+1)$ různými argumenty. Budeme si tedy kešovat, pro které argumenty už známe výsledek, a známé hodnoty nebudeme počítat znovu. Funkce pak poběží jen $\mathcal{O}(nm)$-krát a pokaždé spotřebuje konstantní čas.

Iterativní řešení

Pokračujme podobně jako v minulém oddílu. Otočíme směr výpočtu a tabulku T s výsledky podproblémů budeme vyplňovat bez použití rekurze. Představíme-li si ji jako matici, každý prvek závisí pouze na těch, které leží napravo a dolů od něj. Tabulku proto můžeme vyplňovat po řádcích zdola nahoru, zprava doleva.

Tím získáme následující jednodušší algoritmus, který zjevně běží v čase $\Theta(nm)$. Příklad výpočtu naleznete na obrázku 12.5.

Algoritmus Edit2 (editační vzdálenost řetězců iterativně)
Vstup: Řetězce $x_1 \ldots x_n$ a $y_1 \ldots y_m$

1. Pro $i = 1, \ldots, n+1$ položíme $T[i, m+1] \leftarrow n - i + 1$.
2. Pro $j = 1, \ldots, m+1$ položíme $T[n+1, j] \leftarrow m - j + 1$.
3. Pro $i = n, \ldots, 1$:
4. Pro $j = m, \ldots, 1$:
5. Je-li $x_i = y_j$: $\delta \leftarrow 0$, jinak $\delta \leftarrow 1$
6. $T[i, j] \leftarrow \min(\delta + T[i+1, j+1], 1 + T[i+1, j], 1 + T[i, j+1])$

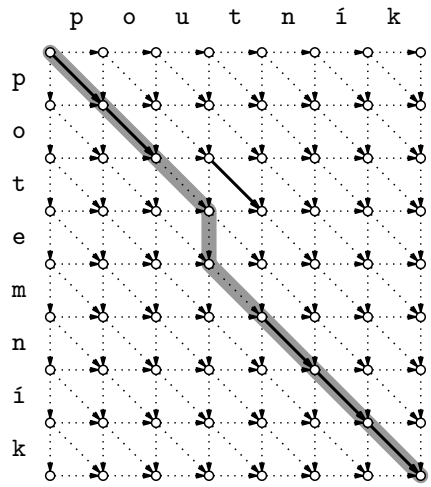
Výstup: Editační vzdálenost $L(x_1 \ldots x_n, y_1 \ldots y_m) = T[1, 1]$

	p	o	t	e	m	n	í	k	
p	3	4	4	4	4	4	5	6	7
o	4	3	3	3	3	3	4	5	6
u	4	3	3	2	2	2	3	4	5
t	4	3	2	2	1	1	2	3	4
n	5	4	3	2	1	0	1	2	3
í	6	5	4	3	2	1	0	1	2
k	7	6	5	4	3	2	1	0	1
	8	7	6	5	4	3	2	1	0

Obrázek 12.5: Tabulka T pro slova `poutník` a `potemník`

Grafové řešení

Editační vzdálenost můžeme také popsat pomocí vhodného orientovaného grafu (obrázek 12.6). Vrcholy budou odpovídat možným pozicím v obou řetězcích. Budou to tedy dvojice (i, j), kde $1 \leq i \leq n+1$ a $1 \leq j \leq m+1$. Hrany budou popisovat možné operace: z vrcholu (i, j) povede hrana do $(i+1, j)$, $(i, j+1)$ a $(i+1, j+1)$. Tyto hrany odpovídají po řadě smazání znaku, vložení znaku a ponechání/záměně znaku. Všechny budou mít jednotkovou délku, pouze v případě ponechání nezměněného písmene ($x_i = y_j$) bude délka nulová.



Obrázek 12.6: Graf k výpočtu editační vzdálenosti.
Plné hrany mají délku 0, čárkované 1.

Každá cesta z vrcholu $(1, 1)$ do $(n+1, m+1)$ proto odpovídá jedné posloupnosti operací uspořádané zleva doprava, která z řetězce x vyrobí y. Jelikož graf je acyklický a má $\Theta(nm)$ vrcholů a $\Theta(nm)$ hran, můžeme v něm nalézt nejkratší cestu indukcí podle topologického uspořádání v čase $\Theta(nm)$.

Přesně to ostatně dělá náš algoritmus Edit2. Indukcí můžeme dokázat, že $T[i, j]$ je rovno délce nejkratší cesty z vrcholu (i, j) do $(n+1, m+1)$.

Cvičení

1. Dokažte, že editační vzdálenost $L(x, y)$ se chová jako *metrika:* je vždy nezáporná, nulová pouze pro $x = y$, symetrická ($L(x, y) = L(y, x)$) a splňuje trojúhelníkovou nerovnost $L(x, z) \leq L(x, y) + L(y, z)$.

2. Algoritmus EDIT2 zabere $\Theta(nm)$ buněk paměti na uložení tabulky T. Ukažte, jak spotřebu paměti snížit na $\Theta(n+m)$.

3. Kromě editační vzdálenosti můžeme chtít spočítat i příslušnou nejkratší posloupnost editačních operací. V grafové interpretaci našeho algoritmu je to triviální – prostě vypíšeme nalezenou nejkratší cestu. Ukažte, jak to udělat bez explicitního sestrojení grafu, třeba přímou úpravou algoritmu EDIT2.

4.* I v předchozím cvičení si lze vystačit s pamětí $\Theta(n+m)$. Existuje pěkný trik založený na metodě Rozděl a panuj. Budeme hledat nejkratší cestu v grafu z obrázku 12.6. Graf postačí procházet po řádcích (to je topologické pořadí), ale obvyklé grafové algoritmy by si pro vypisování cesty musely zapamatovat předchůdce každého vrcholu. My si místo toho uložíme jen to, ve kterém sloupci proťala nejkratší cesta z počátku do daného vrcholu $(n/2)$-tý řádek. To si postačí pamatovat jen v okolí aktuálního řádku. Na konci výpočtu zjistíme, jaký je prostřední vrchol na nejkratší cestě, takže umíme problém rozdělit na dva poloviční podproblémy. Doplňte detaily a dokažte, že tomuto algoritmu postačí lineární paměť, a přesto celkově poběží v čase $\Theta(nm)$.

5. Na první pohled se zdá, že čím podobnější řetězce dostaneme, tím by mělo být jednodušší zjistit jejich editační vzdálenost. Náš algoritmus ovšem pokaždé vyplňuje celou tabulku. Ukažte, jak ho zrychlit, aby počítal v čase $\mathcal{O}((n+m)(L(x,y)+1))$.

6.* Jak by se výpočet editační vzdálenosti změnil, kdybychom mezi editační operace řadili i *prohození* dvou sousedních písmen?

7. Navrhněte algoritmus pro nalezení *nejdelší společné podposloupnosti* daných posloupností $x_1, \ldots, x_n$ a $y_1, \ldots, y_m$. Jak tento problém souvisí s editační vzdáleností a s grafem z obrázku 12.6?

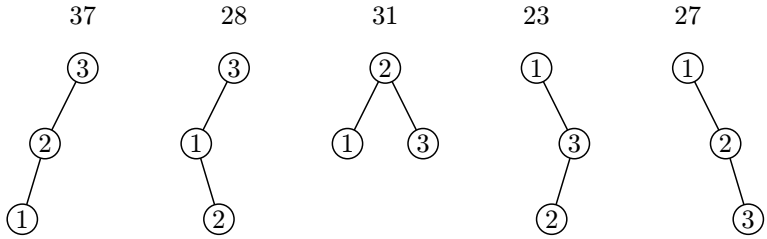
8. Jak naopak najít *nejkratší společnou nadposloupnost* dvou posloupností A a B? Tím se myslí nejkratší posloupnost, která obsahuje jako podposloupnosti jak A, tak B.

12.4 Optimální vyhledávací stromy

Když jsme vymýšleli binární vyhledávací stromy (BVS), uměli jsme zařídit, aby žádný prvek neležel příliš hluboko. Hned několik způsobů vyvažování nám zaručilo logaritmickou hloubku stromu. Co kdybychom ale věděli, že se na některé prvky budeme ptát mnohem častěji než na jiné? Nevyplatilo by se potom umístit tyto „oblíbené“ prvky blízko ke kořeni, byť by to znamenalo další prvky posunout níže?

Vyzkoušejme si to se třemi prvky. Na prvek 1 se budeme ptát celkem 10krát, na 2 jen jednou, na 3 celkem 5krát. Obrázek 12.7 ukazuje možné tvary vyhledávacího stromu

a jejich *ceny* – počty vrcholů navštívených během všech 16 vyhledávání. Například pro prostřední, dokonale vyvážený strom nahlédneme při hledání prvku 1 do 2 vrcholů, při hledání 2 do 1 vrcholu a při hledání 3 opět do 2 vrcholů. Celková cena tedy činí $10 \cdot 2 + 1 \cdot 1 + 5 \cdot 2 = 31$. Následující strom ovšem dosahuje nižší ceny 23, protože často používaná 1 leží v kořeni.



Obrázek 12.7: Cena hledání v různých vyhledávacích stromech

Pojďme se na tento problém podívat obecněji. Máme n prvků s klíči $x_1 < \ldots < x_n$ a kladnými vahami $w_1, \ldots, w_n$. Každému binárnímu vyhledávacímu stromu pro tuto množinu klíčů přidělíme *cenu* $C = \sum_i w_i \cdot h_i$, kde h_i je hloubka klíče x_i (hloubky tentokrát počítáme od jedničky). Chceme najít *optimální vyhledávací strom*, tedy ten s nejnižší cenou.

Rekurzivní řešení

Představme si, že nám někdo napověděl, jaký prvek x_i se nachází v kořeni optimálního stromu. Hned víme, že levý podstrom obsahuje klíče $x_1, \ldots, x_{i-1}$ a pravý podstrom klíče $x_{i+1}, \ldots, x_n$. Navíc oba tyto podstromy musí být optimální – jinak bychom je mohli vyměnit za optimální a tím celý strom zlepšit.

Pokud nám prvek v kořeni nikdo nenapoví, vystačíme si sami: vyzkoušíme všechny možnosti a vybereme tu, která povede na minimální cenu. Levý a pravý podstrom pokaždé sestrojíme rekurzivním zavoláním téhož algoritmu. Původní problém tedy postupně rozkládáme na podproblémy. V každém z nich hledáme optimální strom pro nějaký souvislý úsek klíčů $x_i, \ldots, x_j$. Zatím se spokojíme s tím, že spočítáme cenu tohoto stromu. Tím vznikne funkce OptStrom(i, j) popsaná níže.

Funkce vyzkouší všechny možné kořeny, pro každý z nich rekurzivně spočítá optimální cenu c_ℓ levého podstromu a c_p pravého. Zbývá domyslet, jak z těchto cen spočítat cenu celého stromu. Všem prvkům v levém podstromu jsme zvýšili hloubku o 1, takže cena podstromu vzrostla o součet vah těchto prvků. Podobně to bude v pravém podstromu. Navíc přibyly dotazy na kořen, který má hloubku 1, takže přispívají k ceně přesně vahou kořene. Váhu každého prvku jsme tedy přičetli právě jednou, takže celková cena stromu činí $c_\ell + c_p + (w_i + \ldots + w_j)$.

Algoritmus OptStrom(i, j) (cena optimálního BVS rekurzivně)
Vstup: Klíče $x_i, \ldots, x_j$ s vahami $w_i, \ldots, w_j$

1. Pokud $i > j$, vrátíme 0. ◁ *prázdný úsek dává prázdný strom*
2. $W \leftarrow w_i + \ldots + w_j$ ◁ *celková váha prvků*
3. $C \leftarrow +\infty$ ◁ *zatím nejlepší cena stromu*
4. Pro $k = i, \ldots, j$: ◁ *různé volby kořene*
5. $\quad c_\ell \leftarrow$ OptStrom$(i, k-1)$ ◁ *levý podstrom*
6. $\quad c_p \leftarrow$ OptStrom$(k+1, j)$ ◁ *pravý podstrom*
7. $\quad C \leftarrow \min(C, c_\ell + c_p + W)$ ◁ *cena celého stromu*

Výstup: Cena C optimálního vyhledávacího stromu

Jako obvykle jsme napoprvé získali exponenciální řešení, které půjde zrychlit kešováním spočítaných mezivýsledků. Budeme-li si pamatovat hodnoty $T[i, j] =$ OptStrom(i, j), spočítáme celkově $\mathcal{O}(n^2)$ políček tabulky a každým strávíme čas $\mathcal{O}(n)$. Celkem tedy algoritmus poběží v čase $\mathcal{O}(n^3)$.

Iterativní řešení

Nyní obrátíme směr výpočtu. Využijeme toho, že odpověď pro daný úsek závisí pouze na odpovědích pro kratší úseky. Proto můžeme tabulku mezivýsledků vyplňovat od nejkratších úseků k nejdelším. Tím vznikne následující iterativní algoritmus. Oproti předchozímu řešení si navíc budeme pro každý úsek pamatovat optimální kořen, což nám za chvíli usnadní rekonstrukci optimálního stromu.

Algoritmus OptStrom2 (cena optimálního BVS iterativně)
Vstup: Klíče $x_1, \ldots, x_n$ s vahami $w_1, \ldots, w_n$

1. Pro $i = 1, \ldots, n+1$: $T[i, i-1] \leftarrow 0$ ◁ *prázdné stromy nic nestojí*
2. Pro $\ell = 1, \ldots, n$: ◁ *délky úseků*
3. $\quad$ Pro $i = 1, \ldots, n - \ell + 1$: ◁ *začátky úseků*
4. $\quad\quad j \leftarrow i + \ell - 1$ ◁ *konec aktuálního úseku*
5. $\quad\quad W \leftarrow w_i + \ldots + w_j$ ◁ *celková váha úseku*
6. $\quad\quad T[i, j] \leftarrow +\infty$
7. $\quad\quad$ Pro $k = i, \ldots, j$: ◁ *možné kořeny*
8. $\quad\quad\quad C \leftarrow T[i, k-1] + T[k+1, j] + W$ ◁ *cena stromu*
9. $\quad\quad\quad$ Pokud $C < T[i, j]$: ◁ *průběžné minimum*
10. $\quad\quad\quad\quad T[i, j] \leftarrow C$
11. $\quad\quad\quad\quad K[i, j] \leftarrow k$

Výstup: Cena $T[1, n]$ optimálního stromu, pole K s optimálními kořeny

Spočítejme časovou složitost. Vnitřní cyklus (kroky 4 až 11) běží v čase $\mathcal{O}(n)$ a spouští se $\mathcal{O}(n^2)$-krát. To celkem dává $\mathcal{O}(n^3)$.

Odvodit ze zapamatovaných kořenů skutečnou podobu optimálního stromu už bude hračka. Kořenem je prvek s indexem $r = K[1, n]$. Jeho levým synem bude kořen optimálního stromu pro úsek $1, \ldots, r-1$, což je prvek s indexem $K[1, r-1]$, a tak dále. Z této úvahy ihned plyne následující rekurzivní algoritmus. Zavoláme-li OptStromReko$(1, n)$, vrátí nám celý optimální strom.

Algoritmus OptStromReko(i, j) (konstrukce optimálního BVS)
Vstup: Klíče $x_i, \ldots, x_j$, pole K spočítané algoritmem OptStrom2

1. Pokud $i > j$, vrátíme prázdný strom.
2. $r \leftarrow K[i, j]$ ◁ *kolikátý prvek je v kořeni?*
3. Vytvoříme nový vrchol v s klíčem x_r.
4. Jako levého syna nastavíme OptStromReko$(i, r-1)$.
5. Jako pravého syna nastavíme OptStromReko$(r+1, j)$.

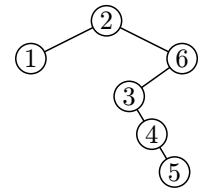
Výstup: Optimální vyhledávací strom s kořenem v

Samotná rekonstrukce stráví v každém rekurzivním volání konstantní čas a vyrobí přitom jeden vrchol stromu. Jelikož celkem vytvoříme n vrcholů, stihneme to v čase $\mathcal{O}(n)$. Celkem tedy hledáním optimálního stromu strávíme čas $\mathcal{O}(n^3)$. Dodejme, že existuje i kvadratický algoritmus (cvičení 7).

$w_1 = 1$
$w_2 = 10$
$w_3 = 3$
$w_4 = 2$
$w_5 = 1$
$w_6 = 9$

T	0	1	2	3	4	5	6
1	0	1	12	18	24	28	52
2	–	0	10	16	22	26	50
3	–	–	0	3	7	10	25
4	–	–	–	0	2	4	16
5	–	–	–	–	0	1	11
6	–	–	–	–	–	0	9
7	–	–	–	–	–	–	0

K	1	2	3	4	5	6
1	1	2	2	2	2	2
2	–	2	2	2	2	2
3	–	–	3	3	3	6
4	–	–	–	4	4	6
5	–	–	–	–	5	6
6	–	–	–	–	–	6



Obrázek 12.8: Ukázka výpočtu algoritmu OptStrom2 a nalezený optimální strom

Abstraktní pohled na dynamické programování

Na závěr se zkusme zamyslet nad tím, co mají jednotlivé aplikace dynamického programování v této kapitole společného – tedy kromě toho, že jsme je odvodili z rekurzivních algoritmů zavedením kešování.

Pokaždé umíme najít vhodný systém podproblémů – těm se často říká *stavy* dynamického programování. Závislosti mezi těmito podproblémy tvoří acyklický orientovaný graf. Díky tomu můžeme všechny stavy procházet v topologickém uspořádání a vždy mít připraveny všechny mezivýsledky potřebné k výpočtu aktuálního stavu.

Aby tento přístup fungoval, nesmí být stavů příliš mnoho: v našich případech jich bylo lineárně nebo kvadraticky. Každý stav jsme pak uměli spočítat v nejhůře lineárním čase, takže jsme dostali samé příjemně polynomiální algoritmy.

Někdy může být dynamické programování zajímavé i s exponenciálně mnoha stavy. Sice pak dostaneme algoritmus o exponenciální složitosti, ale i ten může být rychlejší než jiná možná řešení. Příklady tohoto typu najdete v oddílu 19.5.

Cvičení

1. Optimální vyhledávací strom můžeme také zavést pomocí pravděpodobností. Nechť se na jednotlivé klíče ptáme náhodně, přičemž s pravděpodobností p_i se zeptáme na klíč x_i. Počet vrcholů navštívených při hledání se pak chová jako náhodná veličina se střední hodnotou $\sum_i p_i h_i$ (h_i je opět hloubka i-tého vrcholu). Zkuste formulovat podobný algoritmus v řeči těchto středních hodnot. Jak bude fungovat argument se skládáním stromů z podstromů?

2. Navrhněte, jak rovnou při výpočtu vah konstruovat strom. Využijte toho, že se více vrcholů může odkazovat na tytéž podstromy.

3. Rozmyslete si, že nastavíme-li všem prvkům stejnou váhu, vyjde dokonale vyvážený strom.

4. Jak se algoritmus změní, pokud budeme uvažovat i neúspěšné dotazy? Nejjednodušší je představit si, že váhy přidělujeme i externím vrcholům stromu, jež odpovídají intervalům (x_i, x_{i+1}) mezi klíči.

5. Co jsou v případě optimálních stromů stavy dynamického programování a jak vypadá graf jejich závislostí?

6.** *Knuthova nerovnost:* Nechť $K[i,j]$ je kořen spočítaný algoritmem OptStrom2 pro úsek $x_i, \ldots, x_j$ (je to tedy nejlevější z optimálních kořenů). Donald Knuth dokázal, že platí $K[i,j-1] \leq K[i,j] \leq K[i+1,j]$. Zkuste to dokázat i vy.

7.* *Rychlejší algoritmus:* Vymyslete jak pomocí nerovnosti z předchozího cvičení zrychlit algoritmus OptStrom2 na $\mathcal{O}(n^2)$.

8. *Součin matic:* Násobíme-li matice $X \in \mathbb{R}^{a \times b}$ a $Y \in \mathbb{R}^{b \times c}$ podle definice, počítáme $a \cdot b \cdot c$ součinů čísel. Pokud chceme spočítat maticový součin $X_1 \times \ldots \times X_n$, výsledek

nezávisí na uzávorkování, ale časová složitost (měřená pro jednoduchost počtem součinů čísel) ano. Vymyslete algoritmus, který stanoví, jak výraz uzávorkovat, abychom složitost minimalizovali.

9. *Minimální triangulace:* Konvexní mnohoúhelník můžeme triangulovat, tedy rozřezat neprotínajícími se úhlopříčkami na trojúhelníky. Nalezněte takovou triangulaci, aby součet délek řezů byl nejmenší možný.

10.* *Optimalizace na stromech:* Ukažte, že předchozí dvě cvičení lze formulovat jako hledání optimálního binárního stromu vzhledem k nějaké cenové funkci. Rozšiřte algoritmy z tohoto oddílu, aby uměly pracovat s obecnými cenovými funkcemi a plynulo z nich automaticky i řešení minulých cvičení.

13 Vyhledávání v textu

13 Vyhledávání v textu

V této kapitole se budeme věnovat příslovečnému hledání jehly v kupce sena. *Seno* bude představovat nějaký text σ délky S. Budeme v něm chtít najít všechny výskyty *jehly* – podřetězce ι délky J.

Kupříkladu v seně `bananas` se jehla `ana` vyskytuje hned dvakrát, přičemž výskyty se překrývají. V seně `anna` se tatáž jehla nevyskytuje vůbec, protože hledáme souvislé podřetězce, a nikoliv vybrané podposloupnosti.

Senem přitom nemusí být jenom obyčejný text. Podobné problémy potkáváme třeba v bioinformatice při zkoumání genetického kódu, nebo v matematice, kde pomocí řetězců kódujeme grafy a jiné kombinatorické struktury.

13.1 Řetězce a abecedy

Aby se nám o řetězcových algoritmech lépe vyprávělo, uděláme si nejprve pořádek v terminologii okolo řetězců.

Definice:

- *Abeceda* Σ je nějaká konečná množina, jejím prvkům budeme říkat *znaky* (někdy též *písmena*).
- Σ^* je množina všech *slov* neboli *řetězců* nad abecedou Σ, což jsou konečné posloupnosti znaků ze Σ.

Příklady: Abeceda může být tvořena třeba písmeny `a` až `z`, bity `0` a `1` nebo nukleotidy `C`, `T`, `A`, `G`. Potkáme ovšem i rozlehlejší abecedy: například mezinárodní znaková sada Unicode má $2^{16} = 65\,536$ znaků, v novějších verzích dokonce 1 114 112 znaků. Ještě extrémnějším způsobem používají řetězce lingvisté: na český text se někdy dívají jako na řetězec nad abecedou, jejíž znaky jsou česká slova.

Velikost abecedy se obvykle považuje za konstantu. My budeme navíc předpokládat, že abeceda je dostatečně malá, abychom si mohli dovolit ukládat do paměti pole indexovaná znakem. Později se tohoto předpokladu zbavíme.

Značení:

- *Slova* budeme značit malými písmenky řecké abecedy α, β, ...
- *Znaky* abecedy označíme malými písmeny latinky x, y, ... Konkrétní znaky budeme psát `psacím strojem`. Znak budeme používat i ve smyslu jednoznakového řetězce.
- *Délka slova* $|\alpha|$ udává, kolika znaky je slovo tvořeno.

- *Prázdné slovo* značíme písmenem ε, je to jediné slovo délky 0.
- *Zřetězení* $\alpha\beta$ vznikne zapsáním slov α a β za sebe. Platí $|\alpha\beta| = |\alpha|+|\beta|$, $\alpha\varepsilon = \varepsilon\alpha = \alpha$.
- $\alpha[k]$ je k-tý znak slova α, indexujeme od 0 do $|\alpha| - 1$.
- $\alpha[k : \ell]$ je *podslovo* začínající k-tým znakem a končící těsně před ℓ-tým. Tedy $\alpha[k : \ell] = \alpha[k]\alpha[k + 1] \ldots \alpha[\ell - 1]$. Pokud $k \geq \ell$, je podslovo prázdné. Pokud některou z mezí vynecháme, míní se $k = 0$ nebo $\ell = |\alpha|$.
- $\alpha[: \ell]$ je *prefix* (předpona) tvořený prvními ℓ znaky řetězce.
- $\alpha[k :]$ je *suffix* (přípona) od k-tého znaku do konce řetězce.
- $\alpha[:] = \alpha$.

Dodejme ještě, že každé slovo je podslovem sebe sama a prázdné slovo je podslovem každého slova. Pokud budeme hovořit o *vlastním* podslovu, budeme tím myslet podslovo různé od celého slova. Analogicky pro prefixy a suffixy.

13.2 Knuthův-Morrisův-Prattův algoritmus

Vraťme se nyní zpět k původnímu problému hledání podřetězců. Na vstupu jsme dostali seno σ a jehlu ι. Na výstupu chceme oznámit všechny výskyty; snadno je popíšeme například množinou všech indexů k takových, že $\sigma[k : k + |\iota|] = \iota$.

Kdybychom postupovali podle definice, zkoušeli bychom všechny možné pozice v seně a pro každou z nich otestovali, zda tam nezačíná nějaký výskyt jehly. To je funkční, nicméně pomalé: možných začátků je řádově S, pro každý z nich porovnáváme až J znaků jehly. Celková časová složitost je tedy $\Theta(JS)$.

Zkusme jiný přístup: nalezneme v seně první znak jehly a od tohoto místa budeme porovnávat další znaky. Pokud se přestanou shodovat, přepneme opět na hledání prvního znaku. Jenže odkud? Pokud od místa, kde nastala neshoda, selže to třeba při hledání jehly `kokos` v seně `clanekokokosu` – neshoda nastane za `koko` a zbylý `kos` nás neuspokojí. Nebo se můžeme vrátit až k výskytu prvního znaku a pokračovat těsně za ním, ale to zase trvá $\Theta(JS)$.

Nyní ukážeme algoritmus, který je o trochu složitější, ale nalezne všechny výskyty v čase $\Theta(J + S)$. Později ho zobecníme, aby uměl hledat více různých jehel najednou.

Inkrementální algoritmus

Na hledání podřetězce půjdeme *inkrementálně*. Tím se obecně myslí, že chceme postupně rozšiřovat vstup a přepočítávat, jak se změní výstup. V našem případě vždy přidáme další znak na konec sena a započítáme případný nový výskyt jehly, který končí tímto znakem.

Abychom toho dosáhli, budeme si průběžně udržovat informaci o tom, jakým nejdelším prefixem jehly končí zatím přečtená část sena. Tomu budeme říkat *stav algoritmu.* A jakmile bude tento prefix roven celé jehle, ohlásíme výskyt.

V našem „kokosovém“ příkladě se tedy po přečtení sena `clanekoko` nacházíme ve stavu `koko`, následují stavy `kok`, `koko` a `kokos`.

Představme si nyní obecně, že jsme přečetli řetězec σ, který končil stavem α. Pak vstup rozšíříme o znak x na σx. V jakém stavu se teď máme nacházet? Pokud to nebude prázdný řetězec, musí končit na x, tedy ho můžeme napsat ve tvaru $\alpha' x$.

Všimneme si, že α' *musí být suffixem slova* α*:* Jelikož $\alpha' x$ je prefix jehly, je α' také prefix jehly. A protože $\alpha' x$ je suffixem σx, musí α' být suffixem σ. Tedy jak α, tak α' jsou suffixy slova σ, které jsou současně prefixy jehly. Ovšem stav α jsme vybrali jako nejdelší slovo s touto vlastností, takže α' musí být nejvýše tak dlouhé, a tedy je suffixem α.

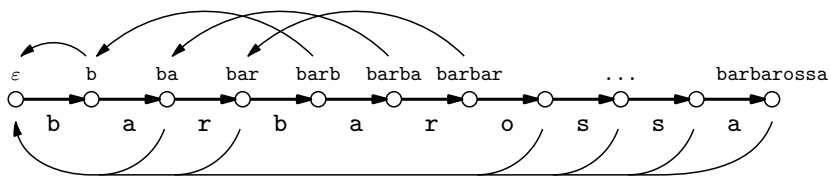
Stačilo by proto probrat všechny suffixy slova α, které jsou prefixem jehly, a vybrat z nich nejdelší, který po rozšíření o znak x stále je prefixem jehly.

Abychom ale nemuseli suffixy procházet všechny, předpočítáme si *zpětnou funkci* z. Ta nám pro každý prefix jehly řekne, jaký je jeho nejdelší vlastní suffix, který je opět prefixem jehly. To nám umožní procházet rovnou kandidáty na nový stav: probereme řetězce α, $z(\alpha)$, $z(z(\alpha))$, ... a použijeme první z nich, který lze rozšířit o znak x. Pokud nepůjde rozšířit ani jeden z těchto kandidátů, novým stavem bude prázdný řetězec.

Na této myšlence je založen následující algoritmus, objevený v roce 1974 Donaldem Knuthem, Jamesem Morrisem a Vaughanem Prattem.

Knuthův-Morrisův-Prattův algoritmus

Algoritmus se opírá o *vyhledávací automat.* To je orientovaný graf, jehož vrcholy (*stavy* automatu) odpovídají prefixům jehly. Vrcholy jsou spojeny hranami dvou druhů: *dopředné* popisují rozšíření prefixu přidáním jednoho písmene, *zpětné* vedou podle zpětné funkce, čili z každého stavu do jeho nejdelšího vlastního suffixu, který je opět stavem.



Obrázek 13.1: Vyhledávací automat pro slovo `barbarossa`

Reprezentace automatu bude přímočará: stavy očíslujeme od 0 do J, dopředná hrana povede vždy ze stavu s do $s+1$ a bude odpovídat rozšíření prefixu o příslušný znak jehly, tedy o $\iota[s]$. Zpětné hrany si zapamatujeme v poli Z: prvek $Z[s]$ bude říkat číslo stavu, do nějž vede zpětná hrana ze stavu s, případně bude nedefinované, pokud taková hrana neexistuje.

Kdybychom takový automat měli, mohli bychom pomocí něj inkrementální algoritmus z předchozího oddílu popsat následovně:

Procedura KmpKrok (jeden krok automatu)
Vstup: Jsme ve stavu s, přečetli jsme znak x
1. Dokud $\iota[s] \neq x \ \wedge \ s \neq 0 : s \leftarrow Z[s]$. ◁ *zpětné hrany*
2. Pokud $\iota[s] = x$, pak $s \leftarrow s+1$. ◁ *dopředná hrana*

Výstup: Nový stav s

Algoritmus KmpHledej (spuštění automatu na řetězec σ)
Vstup: Seno σ, zkonstruovaný automat
1. $s \leftarrow 0$
2. Pro znaky $x \in \sigma$ postupně provádíme:
3. $\quad s \leftarrow \text{KmpKrok}(s, x)$
4. $\quad$ Pokud $s = J$, ohlásíme výskyt.

Invariant: Stav algoritmu s v každém okamžiku říká, jaký nejdelší prefix jehly je suffixem zatím přečtené části sena. (To už víme z úvah o inkrementálním algoritmu.)

Důsledek: Algoritmus ohlásí všechny výskyty. Pokud jsme právě přečetli poslední znak nějakého výskytu, je celá jehla suffixem zatím přečtené části sena, takže se musíme nacházet v posledním stavu.

Jen musíme opravit drobnou chybu – těsně poté, co ohlásíme výskyt, se algoritmus zeptá na dopřednou hranu z posledního stavu. Ta přeci neexistuje! Napravíme to jednoduše: přidáme fiktivní dopřednou hranu, na níž je napsán znak odlišný od všech skutečných znaků. Tím zajistíme, že se po této hraně nikdy nevydáme. Stačí tedy vhodně dodefinovat $\iota[J]$.[1]

Lemma: Funkce KmpHledej běží v čase $\Theta(S)$.

Důkaz: Výpočet funkce můžeme rozdělit na průchody dopřednými a zpětnými hranami. S dopřednými je to snadné – pro každý z S znaků sena projdeme po nejvýše jedné dopředné hraně. To o zpětných hranách neplatí, ale pomůže nám, že každá dopředná hrana

[1] V jazyce C můžeme zneužít toho, že každý řetězec je ukončen znakem s nulovým kódem.

vede o právě 1 stav doprava a každá zpětná o aspoň 1 stav doleva. Proto je všech průchodů po zpětných hranách nejvýše tolik, kolik jsme prošli dopředných hran, takže také nejvýše S. □

Mimochodem, předchozí lemma nám vlastně říká, že jeden krok automatu má konstantní amortizovanou složitost. A důkaz v sobě skrývá přímočaré použití potenciálové metody z oddílu 9.3: roli potenciálu zde hraje číslo stavu.

Konstrukce automatu

Hledání tedy pracuje v lineárním čase, zbývá domyslet, jak v lineárním čase sestrojit automat. Stavy a dopředné hrany získáme triviálně, se zpětnými budeme mít trochu práce.

Podnikneme myšlenkový pokus: Představme si, že automat už máme hotový, ale nevidíme, jak vypadá uvnitř. Chtěli bychom zjistit, jak v něm vedou zpětné hrany, ovšem jediné, co umíme, je spustit automat na nějaký řetězec a zjistit, v jakém stavu skončil.

Tvrdíme, že pro zjištění zpětné hrany ze stavu α stačí automatu předložit řetězec $\alpha[1:]$. Definice zpětné funkce je totiž nápadně podobná invariantu, který jsme dokázali o funkci KmpHledej. Obojí hovoří o nejdelším suffixu daného slova, který je prefixem jehly. Jediný rozdíl je v tom, že v případě zpětné funkce uvažujeme pouze vlastní suffixy, zatímco invariant připouští i ty nevlastní. To ovšem snadno vyřešíme „ukousnutím“ prvního znaku jména stavu.

Pokud bychom chtěli objevit všechny zpětné hrany, stačilo by automat spouštět postupně na řetězce $\iota[1:1]$, $\iota[1:2]$, $\iota[1:3]$, atd. Jelikož funkce KmpHledej je lineární, stálo by nás to dohromady $\mathcal{O}(J^2)$. Pokud si ale všimneme, že každý ze zmíněných řetězců je prefixem toho následujícího, je jasné, že stačí spustit automat jen jednou na řetězec $\iota[1:]$ a jen zaznamenávat, kterými stavy jsme prošli.

To je zajímavé pozorování, řeknete si, ale jak nám pomůže ke konstrukci automatu, když samo hotový automat potřebuje? Pomůže pěkný trik: pokud hledáme zpětnou hranu z i-tého stavu, spouštíme automat na slovo délky $i-1$, takže se můžeme dostat pouze do prvních $i-1$ stavů a vůbec nám nevadí, že v tom i-tém ještě není zpětná hrana hotova.[2]

Při konstrukci automatu tedy nejdříve sestrojíme dopředné hrany, načež rozpracovaný automat spustíme na řetězec $\iota[1:]$ a podle toho, jakými stavy bude procházet, doplníme

[2] Konstruovat nějaký objekt pomocí téhož objektu je osvědčený postup, který si už vysloužil svůj vlastní název. V angličtině se mu říká *bootstrapping* a z toho také vzniklo bootování počítačů, protože při něm operační systém zavádí do paměti sám sebe. Kde se toto slovo vzalo? Bootstrap znamená česky *štruple* – to je takové to očko na patě boty, které usnadňuje nazouvání. A v jednom z příběhů o baronu Prášilovi slyšíme barona vyprávět, jak se uvíznuv v bažině zachránil tím, že se vytáhl za štruple. Krásný popis bootování, není-liž pravda?

zpětné hrany. Jak už víme, vyhledávání má lineární složitost, takže celá konstrukce potrvá $\Theta(J)$.

Hotový algoritmus pro konstrukci automatu můžeme zapsat následovně:

Algoritmus KmpKonstrukce
Vstup: Jehla ι délky J
1. $Z[0] \leftarrow$ *nedefinováno*, $Z[1] \leftarrow 0$
2. $s \leftarrow 0$
3. Pro $i = 2, \ldots, J$:
4. $\quad s \leftarrow \textsc{KmpKrok}(s, \iota[i-1])$
5. $\quad Z[i] \leftarrow s$

Výstup: Pole zpětných hran Z

Výsledky můžeme shrnout do následující věty:

Věta: Algoritmus KMP najde všechny výskyty v čase $\Theta(J + S)$.

Důkaz: Lineární čas s délkou jehly potřebujeme na postavení automatu, lineární čas s délkou sena pak na samotné vyhledání. □

Cvičení

1. Naivní algoritmus, který zkouší všechny možné začátky jehly v seně a vždy porovnává řetězce, má časovou složitost $\mathcal{O}(JS)$. Může být opravdu tak pomalý, uvážíme-li, že porovnávání řetězců skončí, jakmile najde první neshodu? Sestrojte vstup, na kterém algoritmus poběží $\Theta(JS)$ kroků, přestože nic nenajde.

2. *Rotací* řetězce α o K pozic nazýváme řetězec $\alpha[K :]\alpha[: K]$. Jak o dvou řetězcích zjistit, zda je jeden rotací druhého?

3. Jak v lineárním čase zrotovat řetězec, dostačuje-li paměť počítače jen na uložení jednoho řetězce a $\mathcal{O}(1)$ pomocných proměnných?

4.* Navrhněte algoritmus, který v lineárním čase nalezne tu z rotací zadaného řetězce, jež je lexikograficky minimální.

5. Je dáno slovo. Chceme nalézt jeho nejdelší vlastní prefix, který je současně suffixem.

13.3 Více řetězců najednou: algoritmus Aho-Corasicková

Nyní si zahrajeme tutéž hru v trochu složitějších kulisách. Tentokrát bude jehel vícero: $\iota_1, \ldots, \iota_N$, jejich délky označíme $J_i = |\iota_i|$. Dostaneme nějaké seno σ délky S a chceme nalézt všechny výskyty jehel v seně.

Opět si nejdřív musíme ujasnit, co má být výstupem. Dokud byla jehla jedna jediná, bylo to zřejmé – chtěli jsme nalézt množinu všech pozic v seně, na kterých začínaly výskyty jehly. Jak tomu bude zde? Chceme se dozvědět, která jehla se vyskytuje na které pozici. Jinými slovy vypsat všechny dvojice (k, i) takové, že $\sigma[k : k + J_i] = \iota_i$.

Těchto dvojic může být poměrně hodně. Pokud je totiž jedna jehla suffixem druhé, na jedné pozici v seně mohou končit výskyty obou. Celková velikost výstupu tak může být větší než lineární v délce vstupu (viz cvičení 1). Budeme proto hledat algoritmus, který bude lineární v délce vstupu plus délce výstupu, což je evidentně to nejlepší, čeho můžeme dosáhnout.

Algoritmus, který si nyní ukážeme, objevili v roce 1975 Alfred Aho a Margaret Corasicková. Je elegantním zobecněním Knuthova-Morrisova-Prattova algoritmu pro více řetězců.

Opět se budeme snažit sestrojit *vyhledávací automat,* jehož stavy budou odpovídat prefixům jehel a dopředné hrany budou popisovat rozšiřování prefixů o jeden znak. Hrany tedy budou tvořit strom orientovaný směrem od kořene (písmenkový strom pro daný slovník, který už jsme potkali v oddílu 4.3).

Každý list stromu bude odpovídat některé z jehel, ale jak je vidět na obrázku, některé jehly se mohou vyskytovat i ve vnitřních vrcholech (pokud je jedna jehla prefixem jiné). Výskyty jehel ve stromu si tedy nějak označíme, příslušným stavům budeme říkat *koncové.*

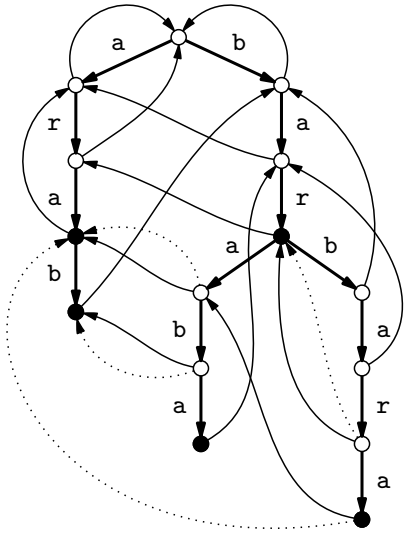
Dále potřebujeme zpětné hrany (na obrázku tenké šipky). Jejich definice bude úplně stejná jako u automatu KMP. Z každého stavu půjde zpětná hrana do jeho nejdelšího vlastního suffixu, který je také stavem. Čili se budeme snažit jméno stavu zkracovat zleva tak dlouho, než dostaneme jméno dalšího stavu. Z kořene – prázdného stavu – pak evidentně žádná zpětná hrana nepovede.

Funkce pro hledání v seně bude vypadat stejně jako u KMP: začne v počátečním stavu (to je kořen stromu) a postupně bude rozšiřovat seno o další písmenka. Pokaždé zkusí jít dopřednou hranou a pokud to nepůjde, bude se vracet po zpětných hranách. Přitom se buďto dostane do vrcholu, kde vhodná dopředná hrana existuje, nebo se nový znak nehodí ani v kořeni, a tehdy je zahozen.

Stejně jako u KMP nahlédneme, že procházení sena trvá $\Theta(S)$ a že platí analogický invariant: v každém okamžiku se nacházíme ve stavu, který odpovídá nejdelšímu suffixu zatím přečteného sena, který je prefixem některé jehly.

Hlášení výskytů

Kdy ohlásíme výskyt jehly? U KMP to bylo snadné: kdykoliv jsme dospěli do posledního stavu, znamenalo to nalezení jehly. Nabízí se hlásit výskyt, kdykoliv dojdeme do stavu označeného jako koncový. To ale nefunguje: pokud náš ukázkový automat přečte seno



Obrázek 13.2: Vyhledávací automat
pro slova `ara`, `bar`, `arab`, `baraba`, `barbara`

`bara`, skončí ve stavu `bara`, který není koncový, a přitom by zde měl ohlásit výskyt jehly `ara`. Stejně tak přečteme-li `barbara`, nevšimneme si, že na témže místě končí i `ara`.

Platí ale, že všechna slova, která bychom měli v daném stavu ohlásit, jsou suffixy jména tohoto stavu. Mohli bychom se tedy vydat po zpětných hranách až do kořene a kdykoliv projdeme přes koncový vrchol, ohlásit výskyt. To ovšem trvá příliš dlouho – jistě by se stávalo, že bychom podnikli dlouhou cestu do kořene a nenašli na ní vůbec nic.

Další, co se nabízí, je předpočítat si pro každý stav β množinu slov $M(\beta)$, jejichž výskyty máme v tomto stavu hlásit. To by fungovalo, ale existují množiny jehel, pro které bude celková velikost množin $M(\beta)$ superlineární (viz cvičení 3). Museli bychom se tedy vzdát lákavé možnosti stavby automatu v lineárním čase.

Jak to tedy vyřešíme? Zavedeme zkratky (na obrázku vyznačeny tečkovaně):

Definice: *Zkratková hrana* ze stavu α vede do nejbližšího koncového stavu $\zeta(\alpha)$ dosažitelného z α po zpětných hranách (a různého od α).

Jinými slovy, zkratka $\zeta(\alpha)$ nám řekne, jaký je nejdelší vlastní suffix slova α, který je jehlou. Pokud takový suffix neexistuje, žádná zkratková hrana ze stavu α nepovede. Pomocí zkratkových hran můžeme snadno vyjmenovat všechny výskyty. Budeme postupovat

stejně, jako bychom procházeli po všech zpětných hranách, jen budeme dlouhé úseky zpětných hran, na nichž není nic k hlášení, přeskakovat v konstantním čase.

Reprezentace automatu

Vyhledávací automat sestává ze stromu dopředných hran, ze zpětných hran a ze zkratkových hran. Rozmysleme si, jak vše uložit do paměti. Stavy očíslujeme, třeba podle toho jak vznikaly, a pro každý stav s si budeme pamatovat:

- *Zpět*(s) – číslo stavu, kam vede zpětná hrana (nebo $\emptyset$, pokud ze stavu s žádná nevede),
- *Zkratka*(s) – kam vede zkratková hrana (obdobně),
- *Slovo*(s) – zda tu končí nějaké slovo (a pokud ano, tak které),
- *Dopředu*(s, x) – kam vede dopředná hrana označená písmenem x (pro malé abecedy si to můžeme pamatovat v poli, pro velké viz cvičení 5).

Celý algoritmus pro zpracování sena automatem pak bude vypadat takto:

Procedura AcKrok (jeden krok automatu)
Vstup: Jsme ve stavu s, přečetli jsme znak x

1. Dokud *Dopředu*$(s, x) = \emptyset \ \wedge \ s \neq$ *kořen*: $s \leftarrow$ *Zpět*(s).
2. Pokud *Dopředu*$(s, x) \neq \emptyset$: $s \leftarrow$ *Dopředu*(s, x).

Výstup: Nový stav s

Algoritmus AcHledej (spuštění automatu na daný řetězec)
Vstup: Seno σ, zkonstruovaný automat

1. $s \leftarrow$ *kořen*
2. Pro znaky $x \in \sigma$ postupně provádíme:
3. $\quad s \leftarrow$ AcKrok(s, x)
4. $\quad j \leftarrow s$
5. $\quad$ Dokud $j \neq \emptyset$:
6. $\quad\quad$ Je-li *Slovo*$(j) \neq \emptyset$:
7. $\quad\quad\quad$ Ohlásíme *Slovo*(j).
8. $\quad\quad j \leftarrow$ *Zkratka*(j)

Stejným argumentem jako u KMP zdůvodníme, že všechny kroky automatu dohromady trvají $\Theta(S)$. Mimo to ještě hlásíme výskyty, což trvá $\Theta($*počet výskytů*$)$. Zbývá ukázat, jak automat sestrojit.

Konstrukce automatu

Opět se inspirujeme algoritmem KMP a nahlédneme, že zpětná hrana ze stavu β vede tam, kam by se automat dostal při hledání ve slově β bez prvního znaku. Chtěli bychom tedy začít sestrojením dopředných hran a pak spouštěním ještě nehotového automatu na jednotlivé jehly doplňovat zpětné hrany, doufajíce, že si vystačíme s už sestrojenou částí automatu.

Kdybychom však automat spouštěli na jednu jehlu po druhé, dostali bychom se do úzkých, protože zpětné hrany mohou vést křížem mezi jednotlivými větvemi stromu. Mohlo by se nám tedy stát, že bychom při hledání potřebovali zpětnou hranu, která dosud nebyla vytvořena.

Budeme tedy zpětné hrany raději konstruovat po hladinách. Každá taková hrana vede alespoň o jednu hladinu výš, takže se při hledání vždy budeme pohybovat po té části stromu, která už je bezpečně hotová. Můžeme si představit, že paralelně spustíme vyhledávání ve všech slovech bez prvních písmenek a vždy uděláme jeden krok každého z těchto hledání, což nám dá zpětné hrany v dalším patře stromu.

Navíc kdykoliv vytvoříme zpětnou hranu, sestrojíme také zkratkovou hranu z téhož vrcholu: Pokud vede zpětná hrana ze stavu s do stavu z a $\mathit{Slovo}(z)$ je definováno, musí vést zkratka z s také do z. Pokud v z žádné slovo nekončí, musí zkratka z s vést do téhož vrcholu, kam vede zkratka ze z.

Algoritmus AcKonstrukce
Vstup: Slova $\iota_1, \ldots, \iota_n$

1. Založíme strom, který obsahuje pouze kořen r.
2. Vložíme do stromu slova $\iota_1 \ldots \iota_n$, nastavíme *Slovo* ve všech stavech.
3. $\mathit{Zpět}(r) \leftarrow \emptyset$, $\mathit{Zkratka}(r) \leftarrow \emptyset$
4. Založíme frontu F a vložíme do ní syny kořene.
5. Pro všechny syny s kořene: $\mathit{Zpět}(s) \leftarrow r$, $\mathit{Zkratka}(s) \leftarrow \emptyset$.
6. Dokud $F \neq \emptyset$:
7. Vybereme i z fronty F.
8. Pro všechny syny s vrcholu i:
9. $z \leftarrow$ AcKrok($\mathit{Zpět}(i)$, písmeno na hraně is)
10. $\mathit{Zpět}(s) \leftarrow z$
11. Pokud $\mathit{Slovo}(z) \neq \emptyset$: $\mathit{Zkratka}(s) \leftarrow z$.
12. Jinak $\mathit{Zkratka}(s) \leftarrow \mathit{Zkratka}(z)$.
13. Vložíme s do fronty F.

Výstup: Strom, pole *Slovo*, *Zpět* a *Zkratka*

Pro rozbor časové složitosti si uvědomíme, že konstrukce zpětných hran hledá všechny jehly, jen kroky jednotlivých hledání vhodným způsobem střídá (jakoby je prováděla paralelně). Časovou složitost tedy můžeme shora omezit součtem složitostí hledání jehel, což, jak už víme, je lineární v délce jehel.

Chování celého algoritmu shrneme do následující věty:

Věta: Algoritmus Aho-Corasicková najde všechny výskyty v čase $\Theta\left(\sum_i J_i + S + V\right)$, kde $J_1, \ldots, J_n$ jsou délky jednotlivých jehel, S je délka sena a V počet výskytů.

Cvičení

1. Nalezněte příklad jehel a sena, v němž je asymptoticky více než lineární počet výskytů. Přesněji řečeno ukažte, že pro každé n existuje vstup, v němž je součet délek jehel a sena $\Theta(n)$ a počet výskytů není $\mathcal{O}(n)$.

2. Uvažujme zjednodušený algoritmus AC, který nepoužívá zkratkové hrany a vždy projde po zpětných hranách až do kořene. Ukažte vhodnými příklady vstupů, že tento algoritmus je asymptoticky pomalejší.

3. Jednoduchý způsob, jak si poradit s hlášením výskytů, je předpočítat si pro každý stav s množinu $M(s)$ slov k ohlášení. Dokažte, že tyto množiny není možné sestrojit v lineárním čase s velikostí slovníku, protože součet jejich velikostí může být pro některé vstupy superlineární.

4. Rozmyslete si, že množiny $M(s)$ z předchozího příkladu by bylo možné reprezentovat jako srůstající spojové seznamy – tedy takové, kde si každý prvek pamatuje ukazatel na svého následníka, který ovšem může ležet v jiném seznamu. Přesvědčte se, že námi zavedené zkratkové hrany lze interpretovat jako ukazatele ve srůstajících seznamech.

5. Upravte algoritmy z této kapitoly, aby si poradily s velkými abecedami.

6. Co kdybychom chtěli pro každou pozici v seně hlásit jenom jeden výskyt jehly? Mohl by to být třeba ten nejdelší, který na dané pozici končí. Ukažte, jak to zařídit bez vyjmenování všech výskytů. Jak by se situace změnila, kdybychom místo nejdelšího hledali nejkratší?

7. Mějme seno a jehly. Popište algoritmus, který v lineárním čase pro každou jehlu spočítá, kolikrát se v seně vyskytuje. Časová složitost by neměla záviset na počtu výskytů – ten, jak už víme, může být superlineární.

8. Cenzor dostane množinu zakázaných podřetězců a text. Vždy najde nejlevější výskyt zakázaného podřetězce v textu (s nejlevějším koncem; pokud jich je více, tak nejdelší takový), vystřihne ho a postup opakuje. Ukažte, jak text cenzurovat v lineárním čase. Chování algoritmu si vyzkoušejte na textu $\texttt{a}^n\texttt{b}^n$ a zakázaných slovech $\texttt{a}^{n+1}$, $\texttt{b}$.

13.4 Rabinův-Karpův algoritmus

Na závěr ukážeme ještě jeden přístup k hledání jehly v seně, založený na hešování. Časová složitost v nejhorším případě sice bude srovnatelná s hledáním hrubou silou, ale v průměru bude lineární a v praxi tento algoritmus často překoná KMP.

Představme si, že máme seno délky S a jehlu délky J. Pořídíme si nějakou hešovací funkci H, která J-ticím znaků přiřazuje čísla z množiny $\{0, \ldots, N-1\}$ pro nějaké dost velké N. Budeme posouvat okénko délky J po seně, pro každou jeho polohu si spočteme heš znaků uvnitř okénka, porovnáme s hešem jehly a pokud se rovnají, porovnáme okénko s jehlou znak po znaku.

Pokud je hešovací funkce „kvalitní“, málokdy se stane, že by se heše rovnaly, takže místo času $\Theta(J)$ na porovnávání řetězců si vystačíme s porovnáním hešů v konstantním čase. Jenže ouha, čas $\Theta(J)$ potřebujeme i na vypočtení heše pro každou polohu okénka. Jak z toho ven?

Pořídíme si hešovací funkci, kterou lze při posunutí okénka o pozici doprava v konstantním čase přepočítat. Tyto požadavky splňuje třeba polynom

$$H(x_1, \ldots, x_J) = (x_1 P^{J-1} + x_2 P^{J-2} + \ldots + x_{J-1} P^1 + x_J P^0) \bmod N,$$

přičemž písmena považujeme za přirozená čísla a P je nějaká vhodná konstanta – potřebujeme, aby byla nesoudělná s N a aby P^J bylo řádově větší než N. Posuneme-li nyní okénko z $x_1, \ldots, x_J$ na $x_2, \ldots, x_{J+1}$, heš se změní takto:

$$\begin{aligned} H(x_2, \ldots, x_{J+1}) &= (x_2 P^{J-1} + x_3 P^{J-2} + \ldots + x_J P^1 + x_{J+1} P^0) \bmod N \\ &= (P \cdot H(x_1, \ldots, x_J) - x_1 P^J + x_{J+1}) \bmod N. \end{aligned}$$

Pokud si mocninu P^J předpočítáme, proběhne aktualizace heše v konstantním čase. Celý algoritmus pak bude vypadat následovně:

Algoritmus RabinKarp
Vstup: Jehla ι délky J, seno σ délky S

1. Zvolíme P a N a předpočítáme $P^J \bmod N$.
2. $j \leftarrow H(\iota)$ ◁ *heš jehly*
3. $h \leftarrow H(\sigma[:J])$ ◁ *heš první pozice okénka*
4. Pro i od 0 do $S - J$: ◁ *možné pozice okénka*
5. Je-li $h = j$:
6. Pokud $\sigma[i : i+J] = \iota$, ohlásíme výskyt na pozici i.
7. Pokud $i < S - J$: ◁ *přepočítáme heš*
8. $h \leftarrow (P \cdot h - \sigma[i] \cdot P^J + \sigma[i+J]) \bmod N$

Pojďme prozkoumat složitost algoritmu. Inicializace algoritmu a počítání hešů okének trvají celkem $\mathcal{O}(J + S)$. Pro každou polohu okénka ovšem můžeme strávit čas $\mathcal{O}(J)$ porovnáváním řetězců. To může celkem trvat až $\mathcal{O}(JS)$. Abychom ukázali, že průměr je lepší, odhadneme pravděpodobnost porovnání.

Pokud nastane výskyt, určitě porovnáváme. Nenastane-li, heš jehly se shoduje s hešem okénka s pravděpodobností $1/N$ (za předpokladu dokonale náhodného chování hešovací funkce, což jsme o té naší nedokázali; blíže viz cvičení 1).

V průměru tedy spotřebujeme čas $\mathcal{O}(J + S + VJ + S/N \cdot J)$, kde V je počet nalezených výskytů. Pokud nám bude stačit najít první výskyt a zvolíme $N > SJ$, algoritmus poběží v průměrném čase $\mathcal{O}(J + S)$.

Dodejme, že tento algoritmus objevili v roce 1987 Richard Karp a Michael Rabin. Později se podobná myšlenka stala základem metod na detekci podobnosti souborů, které můžete ochutnat ve cvičení 2.

Cvičení

1. Polynomiální hešovací funkce nejsou dokonale náhodné, ale kdybychom zvolili prvočíselné N a náhodné P, mohli bychom využít poznatků o univerzálním hešování z oddílu 11.5. Spočítejte pomocí cvičení 11.5.4, kolik v průměru nastane kolizí, a pomocí toho stanovte průměrnou časovou složitost vyhledávání.

2. Bob a Bobek si povídají po telefonu a pojali podezření, že každý z nich používá trochu jinou verzi softwaru pro kouzelný klobouk. Bob navrhuje rozdělit soubor s programem na 32 KB bloky, každý z nich zahešovat do 64-bitového čísla a výsledky si říci. Bobek oponuje, že tak by snadno poznali pár změněných bytů, ale vložení jediného bytu by mohlo změnit všechny heše. Poradíme jim, aby soubor prošli „okénkovou“ hešovací funkcí a kdykoliv je nejnižších B bitů výsledku nulových, začali nový blok. Rozmyslete si, že toto dělení je odolné i proti vkládání a mazání bytů. Jak zvolit B a parametry hešovací funkce, aby průměrná velikost bloku zůstala 32 KB?

13.5 Další cvičení

1. Jak zjistit, zda je zadané slovo α periodické? Tím myslíme zda existuje slovo β a číslo $k > 1$ takové, že $\alpha = \beta^k$ (zřetězení k kopií řetězce β).

2.* Navrhněte datovou strukturu pro dynamické vyhledávání v textu. Jehla je pevná, v seně lze průběžně měnit jednotlivé znaky a struktura odpovídá, zda se v seně právě vyskytuje jehla.

3. *Pestrý* budeme říkat takovému řetězci, jehož všechny rotace jsou navzájem různé. Kolik existuje pestrých řetězců v Σ^n pro konečnou abecedu Σ a prvočíslo n?

4.** Vyřešte předchozí cvičení pro obecné n.

5.* Substituční šifra funguje tak, že zpermutujeme znaky abecedy: například permutací abecedy `abcdeo` na `dacebo` zašifrujeme slovo `abadcode` na `dadecoeb`. Zašifrovaný text je méně srozumitelný, ale například vyzradí, kde v originálu byly stejné znaky a kde různé. Buď dáno seno zašifrované substituční šifrou a nezašifrovaná jehla. Najděte všechny možné výskyty jehly v originálním seně (tedy takové pozice v seně, pro něž existuje permutace abecedy, která přeloží jehlu na příslušný kousek sena).

6. Definujme Fibonacciho slova takto: $F_0 = \texttt{a}$, $F_1 = \texttt{b}$, $F_{n+2} = F_n F_{n+1}$. Jak v zadaném řetězci nad abecedou $\{\texttt{a}, \texttt{b}\}$ najít nejdelší Fibonacciho podslovo?

7.* Pokračujme v předchozím cvičení. Dostaneme řetězec nad nějakou obecnou abecedou, chceme nalézt jeho nejdelší podřetězec, který je isomorfní s nějakým Fibonacciho slovem (liší se pouze substitucí jiných znaků za `a` a `b`).

8.* Je dán text a číslo K. Jak zjistit, který podřetězec délky K se v textu vyskytuje nejčastěji?

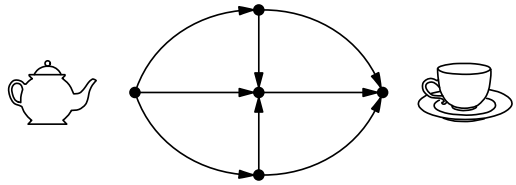
9.* Opět je dán text, tentokrát hledáme nejdelší podřetězec, který se vyskytuje alespoň dvakrát.

10.* Ukažte, jak pro dané dva řetězce najít jejich nejdelší společný podřetězec.

14 Toky v sítích

14 Toky v sítích

Posluchači jisté univerzity měli rádi čaj. Tak si řekli, že by do každé přednáškové místnosti mohli zavést čajovod. Nemuselo by to být komplikované: ve sklepě velikánská čajová konvice, všude po budově trubky. Tlustší by vedly od konvice do jednotlivých pater, pak by pokračovaly tenčí do jednotlivých poslucháren. Jak ale ověřit, že potrubí má dostatečnou kapacitu na uspokojení požadavků všech čajechtivých studentů?



Obrázek 14.1: Čajovod

Podívejme se na to obecněji: Máme síť trubek přepravujících nějakou tekutinu. Popíšeme ji orientovaným grafem. Jeden význačný vrchol funguje jako zdroj tekutiny, jiný jako její spotřebič. Hrany představují jednotlivé trubky s určenou kapacitou, ve vrcholech se trubky setkávají a větví. Máme na výběr, kolik tekutiny pošleme kterou trubkou, a přirozeně chceme ze zdroje do spotřebiče přepravit co nejvíce.

K podobné otázce dojdeme při studiu přenosu dat v počítačových sítích. Roli trubek zde hrají přenosové linky, kapacita říká, kolik dat linka přenese za sekundu. Linky jsou spojené pomocí routerů a opět chceme dopravit co nejvíce dat z jednoho místa v síti na druhé. Data sice na rozdíl od čaje nejsou spojitá (přenášíme je po bytech, nebo rovnou po paketech), ale při dnešních rychlostech přenosu je za spojitá můžeme považovat.

V této kapitole ukážeme, jak sítě a toky formálně popsat a předvedeme několik algoritmů na nalezení největšího možného toku. Také ukážeme, jak pomoci toků řešit jiné, zdánlivě nesouvisející úlohy.

14.1 Definice toku

Definice: *Síť* je uspořádaná pětice (V, E, z, s, c), kde:

- (V, E) je orientovaný graf,
- $c : E \to \mathbb{R}_0^+$ je funkce přiřazující hranám jejich *kapacity,*
- $z, s \in V$ jsou dva různé vrcholy grafu, kterým říkáme *zdroj* a *stok* (neboli *spotřebič*).

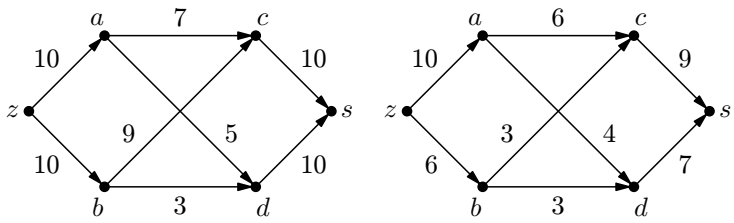
Podobně jako v předchozích kapitolách budeme počet vrcholů grafu značit n a počet hran m.

Mimo to budeme často předpokládat, že graf neobsahuje izolované vrcholy a je symetrický: je-li uv hranou grafu, je jí i vu. Činíme tak bez újmy na obecnosti: kdyby některá z opačných hran chyběla, můžeme ji přidat a přiřadit jí nulovou kapacitu.

Definice: *Tok* v síti je funkce $f : E \to \mathbb{R}_0^+$, pro níž platí:

1. Tok po každé hraně je omezen její kapacitou: $\forall e \in E : f(e) \leq c(e)$.
2. *Kirchhoffův zákon:* Do každého vrcholu přiteče stejně, jako z něj odteče („síť těsní“). Výjimku může tvořit pouze zdroj a spotřebič. Formálně:

$$\forall v \in V \setminus \{z, s\} : \sum_{u:uv \in E} f(uv) = \sum_{u:vu \in E} f(vu).$$



Obrázek 14.2: Nalevo síť, napravo tok v ní o velikosti 16

Sumy podobné těm v Kirchhoffově zákoně budeme psát často, tak pro ně zavedeme šikovné značení:

Definice: Pro libovolnou funkci $f : E \to \mathbb{R}$ definujeme:

- $f^+(v) := \sum_{u:uv \in E} f(uv)$ (celkový *přítok* do vrcholu)
- $f^-(v) := \sum_{u:vu \in E} f(vu)$ (celkový *odtok* z vrcholu)
- $f^\Delta(v) := f^+(v) - f^-(v)$ (*přebytek* ve vrcholu)

Kirchhoffův zákon pak říká prostě to, že $f^\Delta(v) = 0$ pro všechna $v \neq z, s$.

Definice: *Velikost toku* f označíme $|f|$ a bude rovna přebytku spotřebiče $f^\Delta(s)$. Říká nám tedy, kolik tekutiny přiteče do spotřebiče a nevrátí se zpět do sítě.

Jelikož síť těsní, mělo by být jedno, zda velikost toku měříme u spotřebiče, nebo u zdroje. Vskutku – přebytky zdroje a spotřebiče se liší pouze znaménkem:

Lemma: $f^\Delta(z) = -f^\Delta(s)$.

Důkaz: Uvážíme součet přebytků všech vrcholů

$$S = \sum_v f^\Delta(v).$$

Podle Kirchhoffova zákona může přebytek nenulový pouze ve zdroji a spotřebiči, takže tato suma musí být rovna $f^\Delta(z) + f^\Delta(s)$. Současně ale musí být nulová: je to totiž součet nějakých kladných a záporných toků po hranách, přičemž každá hrana přispěje jednou kladně (ve vrcholu, do kterého vede) a jednou záporně (ve vrcholu, odkud vede). □

Poznámka: Když vyslovíme nějakou definici, měli bychom se ujistit, že definovaný objekt existuje. S tokem jako takovým je to snadné: definici toku splňuje v libovolné síti všude nulová funkce. Maximální tok je ale ošidnější: i v jednoduché síti můžeme najít nekonečně mnoho různých toků. Není tedy a priori jasné, že nějaký z nich musí být největší. Zde by pomohla matematická analýza (cvičení 2), ale my na to raději půjdeme konstruktivně – předvedeme algoritmus, jenž maximální tok najde. Nejprve se nám to podaří pro racionální kapacity, později pro libovolné reálné.

Cvičení

1. Naše definice toku v síti úplně nepostihuje „čajový“ příklad z úvodu kapitoly: v něm totiž bylo více spotřebičů. Ukažte, jak takový příklad pomocí našeho modelu toků popsat.

2. Doplňte detaily do následujícího důkazu existence maximálního toku: Uvažme množinu všech toků coby podprostor metrického prostoru $\mathbb{R}^m$. Tato množina je omezená a uzavřená, tedy je kompaktní. Velikost toku je spojitá funkce z této množiny do $\mathbb{R}$, pročež musí nabývat minima i maxima.

14.2 Fordův-Fulkersonův algoritmus

Nejjednodušší z algoritmů na hledání maximálního toku je založen na prosté myšlence: začneme s nulovým tokem a postupně ho vylepšujeme, až dostaneme maximální tok.

Uvažujme, jak by vylepšování mohlo probíhat. Nechť existuje cesta P ze z do s taková, že po všech jejích hranách teče méně, než dovolují kapacity. Takové cestě budeme říkat *zlepšující,* protože po ní můžeme tok zvětšit. Zvolíme

$$\varepsilon := \min_{e \in P} (c(e) - f(e))$$

a tok po každé hraně cesty zvýšíme o ε. Přesněji řečeno, definujeme nový tok f' takto:

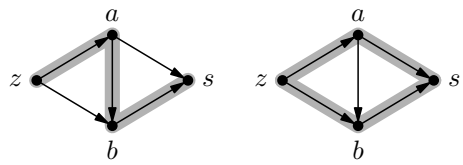
$$f'(e) := \begin{cases} f(e) + \varepsilon & \text{pro } e \in P \\ f(e) & \text{pro } e \notin P \end{cases}$$

Ověříme, že f' je opět korektní tok: Kapacity nepřekročíme, neboť ε jsme zvolili největší, pro něž se to ještě nestane. Kirchhoffovy zákony zůstanou neporušeny, jelikož zdroj a stok nijak neomezují a každému jinému vrcholu na cestě P se zvětší o ε jak přítok $f^+(v)$, tak odtok $f^-(v)$. Ostatním vrcholům se přebytek nezměnil. Také si všimneme, že velikost toku stoupla o ε.

Například v toku na obrázku 14.2 můžeme využít cestu $zbcs$ a poslat po ní 1 jednotku.

Tento postup můžeme opakovat, dokud existují nějaké zlepšující cesty, a získávat čím dál větší toky.

Až zlepšující cesty dojdou (pomiňme na chvíli, jestli se to skutečně stane), bude tok maximální? Překvapivě ne vždy. Uvažujme například síť s jednotkovými kapacitami nakreslenou na obrázku 14.3. Najdeme-li nejdříve cestu $zabs$, zlepšíme po ní tok o 1. Tím dostaneme tok z levého obrázku, ve kterém už žádná další zlepšující cesta není. Jenže jak ukazuje pravý obrázek, maximální tok má velikost 2.



Obrázek 14.3: Algoritmus v úzkých (všude c = 1)

Tuto prekérní situaci by zachránilo, kdybychom mohli poslat tok velikosti 1 proti směru hrany ab. Pak bychom tok z levého obrázku zlepšili po cestě $zbas$ a získali bychom maximální tok z pravého obrázku. Posílat proti směru hrany ve skutečnosti nemůžeme, ale stejný efekt bude mít odečtení jedničky od toku po směru hrany.

Rozšíříme tedy náš algoritmus, aby byl ochoten nejen přičítat po směru hran, ale také odčítat proti směru. Pokud chceme zvýšit tok z u do v, můžeme přičíst k $f(uv)$ nejvýše $c(uv) - f(uv)$, abychom nepřekročili kapacitu. Podobně odečíst od $f(vu)$ smíme nejvýše $f(vu)$, abychom nevytvořili záporný tok. Součtu těchto dvou možných zlepšení se říká rezerva:

Definice: *Rezerva hrany uv* je číslo $r(uv) := c(uv) - f(uv) + f(vu)$. Hraně s nulovou rezervou budeme říkat *nasycená,* hraně s kladnou rezervou *nenasycená.* O cestě řekneme,

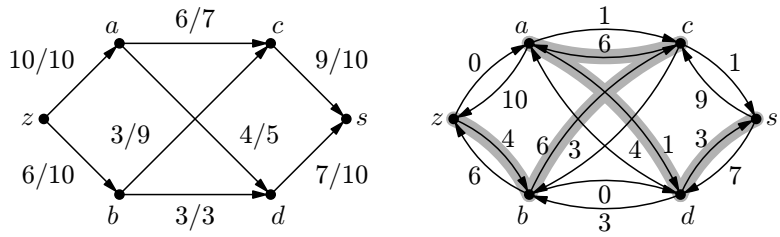
že je *nasycená,* pokud je nasycená alespoň jedna její hrana; jinak mají všechny hrany kladné rezervy a cesta je *nenasycená.*

Roli zlepšujících cest tedy budou hrát nenasycené cesty. Budeme je opakovaně hledat a tok po nich zlepšovat. Tím dostaneme algoritmus, který objevili v roce 1954 Lester Ford a Delbert Fulkerson.

Algoritmus FordFulkerson
Vstup: Síť (V, E, z, s, c)

1. $f \leftarrow$ libovolný tok, např. všude nulový
2. Dokud existuje nenasycená cesta P ze z do s, opakujeme:
3. $\quad \varepsilon \leftarrow \min\{r(e) \mid e \in P\}$ ◁ *spočítáme rezervu celé cesty*
4. $\quad$ Pro všechny hrany $uv \in P$:
5. $\quad\quad \delta \leftarrow \min\{f(vu), \varepsilon\}$ ◁ *kolik můžeme odečíst v protisměru*
6. $\quad\quad f(vu) \leftarrow f(vu) - \delta$
7. $\quad\quad f(uv) \leftarrow f(uv) + \varepsilon - \delta$ ◁ *zbytek přičteme po směru*

Výstup: Maximální tok f



Obrázek 14.4: Fordův-Fulkersonův algoritmus v řeči rezerv: vlevo tok/kapacita, vpravo rezervy a nenasycená cesta

Rozbor algoritmu

Abychom dokázali, že algoritmus vydá maximální tok, nejprve si musíme ujasnit, že se vždy zastaví. Nemohlo by se třeba stát, že bude tok růst donekonečna o stále menší a menší hodnoty?

- Pakliže jsou všechny kapacity celá čísla, vypočtené toky budou také celočíselné. Velikost toku se proto v každém kroku zvětší alespoň o 1 a algoritmus se zastaví po nejvýše tolika krocích, kolik je nějaká horní mez pro velikost maximálního toku – např. součet kapacit všech hran vedoucích do stoku. (O moc rychlejší ale být nemusí, jak uvidíme ve cvičení 1.)

- Pro racionální kapacity využijeme jednoduchý trik. Nechť M je nejmenší společný násobek jmenovatelů všech kapacit. Spustíme-li algoritmus na síť s kapacitami $c'(e) = c(e) \cdot M$, bude se rozhodovat stejně jako v původní síti, protože bude stále platit $f'(e) = f(e) \cdot M$. Nová síť je přitom celočíselná, takže se algoritmus jistě zastaví.
- Na síti s iracionálními kapacitami se algoritmus může chovat divoce: nemusí se zastavit, ba ani nemusí konvergovat ke správnému výsledku (cvičení 2).

Pro celočíselné a racionální kapacity se tedy algoritmus zastaví a vydá jako výsledek nějaký tok f. Abychom dokázali, že tento tok je maximální, povoláme na pomoc řezy. Definujeme je podobně jako v kapitole o minimálních kostrách, ale tentokrát pro orientované grafy se zdrojem a stokem.

Definice: Pro libovolné dvě množiny vrcholů A a B označíme $E(A,B)$ množinu hran vedoucích z A do B, tedy $E(A,B) := E \cap (A \times B)$. Je-li dále f nějaká funkce přiřazující hranám čísla, označíme:

- $f(A,B) := \sum_{e \in E(A,B)} f(e)$ (tok z A do B)
- $f^\Delta(A,B) := f(A,B) - f(B,A)$ (čistý tok z A do B)

Definice: *Řez* (přesněji řečeno *elementární řez*, viz cvičení 5) je množina hran, kterou lze zapsat jako $E(A,B)$ pro nějaké dvě množiny vrcholů A a B. Tyto množiny musí být disjunktní, musí dohromady obsahovat všechny vrcholy a navíc v A musí ležet zdroj a v B stok. Množině A budeme říkat *levá množina řezu,* množině B *pravá. Kapacitu řezu* definujeme jako součet kapacit hran zleva doprava, tedy $c(A,B)$.

Lemma: Pro každý řez $E(A,B)$ a každý tok f platí $f^\Delta(A,B) = |f|$.

Důkaz: Opět šikovně sečteme přebytky vrcholů:

$$f^\Delta(A,B) = \sum_{v \in B} f^\Delta(v) = f^\Delta(s).$$

První rovnost získáme počítáním přes hrany: každá hrana vedoucí z vrcholu v B do jiného vrcholu v B k sumě přispěje jednou kladně a jednou záporně; hrany ležící zcela mimo B nepřispějí vůbec; hrany s jedním koncem v B a druhým mimo přispějí jednou, přičemž znaménko se bude lišit podle toho, který konec je v B. Druhá rovnost je snadná: všechny vrcholy v B kromě spotřebiče mají podle Kirchhoffova zákona nulový přebytek a zdroj v B neleží. □

Poznámka: Původní definice velikosti toku coby přebytku spotřebiče je speciálním případem předchozího lemmatu – měří tok přes řez $E(V \setminus \{s\}, \{s\})$.

Důsledek: Pro každý tok f a každý řez $E(A,B)$ platí $|f| \leq c(A,B)$. Jinak řečeno, velikost každého toku je shora omezena kapacitou každého řezu.

Důkaz: $|f| = f^{\Delta}(A,B) = f(A,B) - f(B,A) \leq f(A,B) \leq c(A,B)$. □

Důsledek: Pokud $|f| = c(A,B)$, pak je tok f maximální a řez $E(A,B)$ minimální (tedy s nejmenší možnou kapacitou). Velikost toku f totiž nelze zvětšit nad kapacitu řezu $E(A,B)$, zatímco řez $E(A,B)$ nejde zmenšit pod velikost toku f.

Takže pokud najdeme k nějakému toku stejně velký řez, můžeme řez použít jako certifikát maximality toku a tok jako certifikát minimality řezu. Následující lemma nám zaručí, že takovou dvojici toku s řezem vždy najdeme.

Lemma: Pokud se Fordův-Fulkersonův algoritmus zastaví, vydá maximální tok.

Důkaz: Nechť se algoritmus zastaví. Uvažme množiny vrcholů

$$A := \{v \in V \mid \text{existuje nenasycená cesta ze } z \text{ do } v\} \quad \text{a} \quad B := V \setminus A.$$

Situaci sledujme na obrázku 14.5. Všimneme si, že množina $E(A,B)$ je řez: Zdroj z leží v A, protože ze z do z existuje cesta nulové délky, která je tím pádem nenasycená. Spotřebič musí ležet v B, neboť jinak by existovala nenasycená cesta ze z do s, tudíž by algoritmus ještě neskončil.

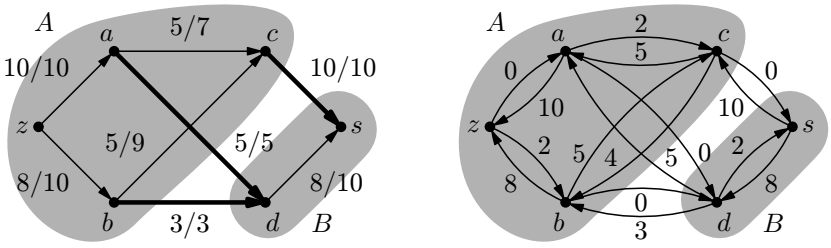
Dále víme, že všechny hrany řezu mají nulovou rezervu: kdyby totiž pro nějaké $u \in A$ a $v \in B$ měla hrana uv rezervu nenulovou (nebyla nasycená), spojením nenasycené cesty ze zdroje do u s touto hranou by vznikla nenasycená cesta ze zdroje do v, takže vrchol v by také musel ležet v A, a nikoliv v B.

Proto po všech hranách řezu vedoucích z A do B teče tok rovný kapacitě hran a po hranách z B do A neteče nic. Nalezli jsme tedy řez $E(A,B)$, pro nějž $f^{\Delta}(A,B) = c(A,B)$. To znamená, že tento řez je minimální a tok f maximální. □

Poznámka: Kdyby přišel kouzelník a rovnou ze svého klobouku vytáhl maximální tok, jen těžko by skeptické publikum přesvědčoval o jeho maximalitě. Fordův-Fulkersonův algoritmus to má snazší: k toku vydá i *certifikát* jeho maximality, totiž příslušný minimální řez. To, že tok i řez jsou korektní a že jejich velikosti se rovnají, může publikum ověřit v lineárním čase.

Nyní konečně můžeme vyslovit větu o správnosti Fordova-Fulkersonova algoritmu:

Věta: Pro každou síť s racionálními kapacitami se Fordův-Fulkersonův algoritmus zastaví a vydá maximální tok a minimální řez.



Obrázek 14.5: Situace po zastavení F.-F. algoritmu. Nalevo tok/kapacita, napravo rezervy, v obou obrázcích vyznačen minimální řez E(A, B).

Důsledek: Síť s celočíselnými kapacitami má aspoň jeden z maximálních toků celočíselný a Fordův-Fulkersonův algoritmus takový tok najde.

Důkaz: Když dostane Fordův-Fulkersonův algoritmus celočíselnou síť, najde v ní maximální tok. Tento tok bude jistě celočíselný, protože algoritmus čísla pouze sčítá, odečítá a porovnává, takže nemůže nikdy z celých čísel vytvořit necelá. □

To, že umíme najít celočíselné řešení, není vůbec samozřejmé. U mnoha problémů je racionální varianta snadná, zatímco celočíselná velmi obtížná (viz třeba celočíselné lineární rovnice v kapitole 19.3). Teď si ale chvíli užívejme, že toky se v tomto ohledu chovají pěkně.

Cvičení

1. Najděte příklad sítě s nejvýše 10 vrcholy a 10 hranami, na níž Fordův-Fulkersonův algoritmus provede více než milion iterací.

2.** Najděte síť s reálnými kapacitami, na níž Fordův-Fulkersonův algoritmus nedoběhne. Lze dokonce zařídit, aby k maximálnímu toku ani nekonvergoval.

3. Navrhněte algoritmus, který pro zadaný orientovaný graf a jeho vrcholy u a v nalezne největší možný systém hranově disjunktních cest z u do v.

4. Upravte algoritmus z předchozího cvičení, aby nalezené cesty byly dokonce vrcholově disjunktní (až na krajní vrcholy).

5. Obecná definice řezu říká, že řez je množina hran grafu, po jejímž odebrání se graf rozpadne na více komponent (případně máme-li určený zdroj a stok, skončí oba v různých komponentách). Srovnejte tuto definici s naší definicí elementárního řezu. Ukažte, že existují i neelementární řezy. Také ukažte, že jsou-li kapacity všech hran kladné, pak každý minimální řez je elementární.

6. Profesor Forderson si přečetl začátek tohoto oddílu a pokusil se chybný algoritmus (který zlepšuje tok pouze po směru hran) zachránit tím, že bude vybírat *nejkratší* zlepšující cesty. Ukažte, že zlepšovák pana profesora nefunguje!

7. Pokračujme ve vynálezech profesora Fordersona z předchozího cvičení. Existuje vůbec nějaká posloupnost zlepšujících cest po směru hran, která vede k maximálnímu toku? Pokud ano, platí to i pro posloupnost nejkratších zlepšujících cest?

8.* Pro daný neorientovaný graf nalezněte co největší k takové, že graf je hranově k-souvislý. (To znamená, že je souvislý i po odebrání nejvýše $k-1$ hran.)

9.** Přímočará implementace Fordova-Fulkersonova algoritmu bude nejspíš graf prohledávat do šířky, takže vždy najde *nejkratší* nenasycenou cestu. Pak překvapivě platí, že algoritmus zlepší tok jen $\mathcal{O}(nm)$-krát, než se zastaví. Návod k důkazu: Nechť $\ell(u)$ je vzdálenost ze zdroje do vrcholu u po nenasycených hranách. Nejprve si rozmyslete, že $\ell(u)$ během výpočtu nikdy neklesá. Pak dokažte, že mezi dvěma nasyceními libovolné hrany uv se musí $\ell(u)$ zvýšit. Proto každou hranu nasytíme nejvýše $\mathcal{O}(n)$-krát.

14.3 Největší párování v bipartitních grafech

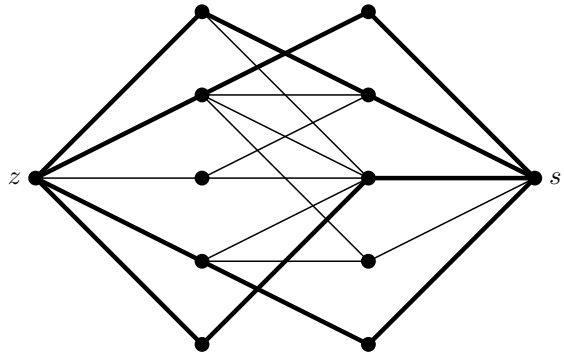
Problém maximálního toku je zajímavý nejen sám o sobě, ale také tím, že na něj můžeme elegantně převádět jiné problémy. Jeden takový si ukážeme a rovnou při tom využijeme celočíselnost.

Definice: Množina hran $F \subseteq E$ se nazývá *párování*, jestliže žádné dvě hrany této množiny nemají společný vrchol. *Velikostí* párování myslíme počet jeho hran.

Párování v bipartitních grafech má zjevné aplikace: Jednu partitu může tvořit třeba množina mlsounů, druhou množina zákusků a hrany říkají, kdo má na co chuť. Párování pak odpovídá tomu, že mlsounům rozdáme jejich oblíbené zákusky tak, aby žádný mlsoun nedostal dva a žádný zákusek nebyl sněden vícekrát. Přirozeně chceme nalézt párování o co nejvíce hranách.

Mějme tedy nějaký bipartitní graf (V, E). Přetvoříme ho na síť (V', E', z, s, c) následovně:

- Nalezneme partity grafu, budeme jim říkat *levá* a *pravá*.
- Všechny hrany zorientujeme zleva doprava.
- Přidáme zdroj z a vedeme z něj hrany do všech vrcholů levé partity.
- Přidáme spotřebič s a vedeme do něj hrany ze všech vrcholů pravé partity.
- Všem hranám nastavíme jednotkovou kapacitu.



Obrázek 14.6: Hledání největšího párování v bipartitním grafu. Hrany jsou orientované zleva doprava a mají kapacitu 1.

Nyní v této síti najdeme maximální celočíselný tok. Jelikož všechny hrany mají kapacitu 1, musí po každé hraně téci buď 0 nebo 1. Do výsledného párování vložíme právě ty hrany původního grafu, po kterých teče 1.

Dostaneme opravdu párování? Kdybychom nedostali, znamenalo by to, že nějaké dvě vybrané hrany mají společný vrchol. Pokud by to byl vrchol pravé partity, pak do tohoto vrcholu přitekly alespoň 2 jednotky toku, jenže ty nemají kudy odtéci. Analogicky pokud by se hrany setkaly nalevo, musely by z vrcholu odtéci alespoň 2 jednotky, které se tam nemají jak dostat.

Zbývá nahlédnout, že nalezené párování je největší možné. K tomu si stačí všimnout, že z toku vytvoříme párování o tolika hranách, kolik je velikost toku, a naopak z každého párování umíme vytvořit celočíselný tok odpovídající velikosti. Nalezli jsme bijekci mezi množinou všech celočíselných toků a množinou všech párování a tato bijekce zachovává velikost. Největší tok tudíž musí odpovídat největšímu párování.

Navíc dokážeme, že Fordův-Fulkersonův algoritmus na sítích tohoto druhu pracuje překvapivě rychle:

Věta: Pro síť, jejíž všechny kapacity jsou jednotkové, nalezne Fordův-Fulkersonův algoritmus maximální tok v čase $\mathcal{O}(nm)$.

Důkaz: Jedna iterace algoritmu běží v čase $\mathcal{O}(m)$: nenasycenou cestu najdeme prohledáním grafu do šířky, samotné zlepšení toku zvládneme v čase lineárním s délkou cesty. Jelikož každá iterace zlepší tok alespoň o 1, počet iterací je omezen velikostí maximálního toku, což je nejvýše n (uvažte řez okolo zdroje). □

Důsledek: Největší párování v bipartitním grafu lze nalézt v čase $\mathcal{O}(nm)$.

Důkaz: Předvedená konstrukce vytvoří z grafu síť o $n' = n + 2$ vrcholech a $m' = m + 2n$ hranách a spotřebuje na to čas $\mathcal{O}(m' + n')$. Pak nalezneme maximální celočíselný tok Fordovým-Fulkersonovým algoritmem, což trvá $\mathcal{O}(n'm')$. Nakonec tok v lineárním čase přeložíme na párování. Vše dohromady trvá $\mathcal{O}(n'm') = \mathcal{O}(nm)$. □

Cvičení

1. V rozboru Fordova-Fulkersonova algoritmu v sítích s jednotkovými kapacitami jsme použili, že tok se pokaždé zvětší alespoň o 1. Může se stát, že se zvětší víc?

2. Mějme šachovnici $r \times s$, z níž políčkožrout sežral některá políčka. Chceme na ni rozestavět co nejvíce šachových věží tak, aby se navzájem neohrožovaly. Věž můžeme postavit na libovolné nesežrané políčko a ohrožuje všechny věže v témže řádku i sloupci. Navrhněte efektivní algoritmus, který takové rozestavění najde.

3. Situace stejná jako v minulém cvičení, ale dvě věže se neohrožují přes sežraná políčka.

4. Opět šachovnice po zásahu políčkožrouta. Chceme na nesežraná políčka rozmístit kostky velikosti 1×2 políčka tak, aby každé nesežrané políčko bylo pokryto právě jednou kostkou. Kostky je povoleno otáčet.

5.* Hledání největšího párování jsme převedli na hledání maximálního toku v jisté síti. Přeložte chod Fordova-Fulkersonova algoritmu v této síti zpět do řeči párování v původním grafu. Čemu odpovídá zlepšující cesta?

6.* Podobně jako v minulém cvičení přeformulujte řešení úlohy 4, aby pracovalo přímo s kostkami na šachovnici.

14.4 Dinicův algoritmus

V kapitole 14.2 jsme ukázali, jak nalézt maximální tok Fordovým-Fulkersonovým algoritmem. Začali jsme s tokem nulovým a postupně jsme ho zvětšovali. Pokaždé jsme v síti našli *nenasycenou cestu,* tedy takovou, na níž mají všechny hrany kladnou rezervu. Podél cesty jsme pak tok zlepšili.

Nepříjemné je, že může trvat velice dlouho, než se tímto způsobem dobereme k maximálnímu toku. Pro obecné reálné kapacity se to dokonce nemusí stát vůbec. Proto odvodíme o něco složitější, ale výrazně rychlejší algoritmus objevený v roce 1970 Jefimem Dinicem. Jeho základní myšlenkou je nezlepšovat toky pomocí cest, ale rovnou pomocí toků ...

Síť rezerv

Nejprve přeformulujeme definici toku, aby se nám s ní lépe pracovalo. Už několikrát se nám totiž osvědčilo simulovat zvýšení průtoku nějakou hranou pomocí snížení průtoku opačnou hranou. To je přirozené, neboť přenesení x jednotek toku po hraně vu se chová stejně jako přenesení $-x$ jednotek po hraně uv. To vede k následujícímu popisu toků.

Definice: Každé hraně uv přiřadíme její *průtok* $f^*(uv) = f(uv) - f(vu)$.

Pozorování: Průtoky mají následující vlastnosti:

(1) $f^*(uv) = -f^*(vu)$,
(2) $f^*(uv) \leq c(uv)$,
(3) $f^*(uv) \geq -c(vu)$,
(4) pro všechny vrcholy $v \neq z, s$ platí $\sum_{u:uv\in E} f^*(uv) = 0$.

Podmínka (3) přitom plyne z (1) a (2). Suma ve (4) není nic jiného než vztah pro přebytek $f^\Delta(v)$ přepsaný pomocí (1).

Lemma P (o průtoku): Nechť funkce $f^* : E \to \mathbb{R}$ splňuje podmínky (1), (2) a (4). Potom existuje tok f, jehož průtokem je f^*.

Důkaz: Tok f určíme pro každou dvojici hran uv a vu zvlášť. Předpokládejme, že $f^*(uv) \geq 0$; v opačném případě využijeme (1) a u prohodíme s v. Nyní stačí položit $f(uv) := f^*(uv)$ a $f(vu) := 0$. Díky vlastnosti (2) funkce f nepřekračuje kapacity, díky (4) pro ni platí Kirchhoffův zákon. □

Důsledek: Místo toků tedy stačí uvažovat průtoky hranami. Tím se ledacos formálně zjednoduší: přebytek $f^\Delta(v)$ je prostým součtem průtoků hranami vedoucími do v, rezervu $r(uv)$ můžeme zapsat jako $c(uv) - f^*(uv)$. To nám pomůže k zobecnění zlepšujících cest z Fordova-Fulkersonova algoritmu.

Definice: *Síť rezerv* k toku f v síti $S = (V, E, z, s, c)$ je síť $R(S, f) := (V, E, z, s, r)$, kde $r(e)$ je rezerva hrany e při toku f.

Lemma Z (o zlepšování toků): Pro libovolný tok f v síti S a libovolný tok g v síti $R(S, f)$ lze v čase $\mathcal{O}(m)$ nalézt tok h v síti S takový, že $|h| = |f| + |g|$.

Důkaz: Toky přímo sčítat nemůžeme, ale průtoky po jednotlivých hranách už ano. Pro každou hranu e položíme $h^*(e) := f^*(e) + g^*(e)$. Nahlédněme, že funkce h^* má všechny vlastnosti vyžadované lemmatem **P**.

(1) Jelikož první podmínka platí pro f^* i g^*, platí i pro jejich součet.
(2) Víme, že $g^*(uv) \leq r(uv) = c(uv) - f^*(uv)$, takže $h^*(uv) = f^*(uv) + g^*(uv) \leq c(uv)$.
(4) Když se sečtou průtoky, sečtou se i přebytky.

Zbývá dokázat, že se správně sečetly velikosti toků. K tomu si stačí uvědomit, že velikost toku je přebytkem spotřebiče a přebytky se sečetly. □

Poznámka: Zlepšení po nenasycené cestě je speciálním případem tohoto postupu – odpovídá toku v síti rezerv, který je konstantní na jedné cestě a všude jinde nulový.

Dinicův algoritmus

Dinicův algoritmus začne s nulovým tokem a bude ho vylepšovat pomocí nějakých pomocných toků v síti rezerv, až se dostane k maximálnímu toku. Počet potřebných iterací přitom bude záviset na tom, jak „vydatné“ pomocné toky seženeme – na jednu stranu bychom chtěli, aby byly podobné maximálnímu toku, na druhou stranu jejich výpočtem nechceme trávit příliš mnoho času. Vhodným kompromisem jsou tzv. blokující toky:

Definice: Tok je *blokující*, jestliže na každé orientované cestě ze zdroje do spotřebiče existuje alespoň jedna hrana, na níž je tok roven kapacitě.

Blokující tok ale nebudeme hledat v celé síti rezerv, nýbrž jen v podsíti tvořené nejkratšími cestami ze zdroje do spotřebiče. Můžeme si představit, že provádíme najednou mnoho iterací Fordova-Fulkersonova algoritmu pro všechny nejkratší cesty.

Definice: Síť je *vrstevnatá (pročištěná)*, pokud všechny její vrcholy a hrany leží na nejkratších cestách ze z do s. (Abychom vyhověli naší definici sítě, musíme ke každé takové hraně přidat hranu opačnou s nulovou kapacitou, ale ty algoritmus nebude používat a ani udržovat v paměti.)

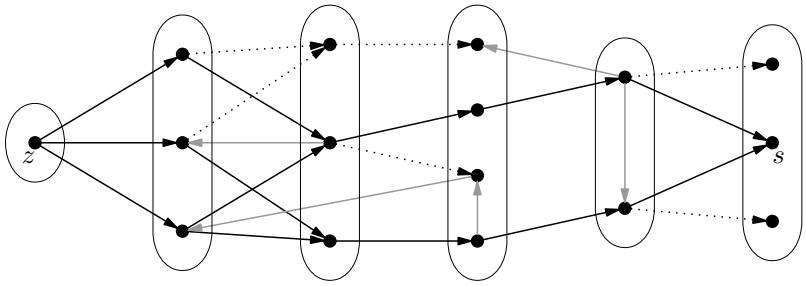
Základ Dinicova algoritmu vypadá takto:

Algoritmus DINIC
Vstup: Síť (V, E, z, s, c)
1. $f \leftarrow$ nulový tok
2. Opakujeme:
3. Sestrojíme síť rezerv R a smažeme hrany s nulovou rezervou.
4. $\ell \leftarrow$ délka nejkratší cesty ze z do s v R
5. Pokud žádná taková cesta neexistuje, zastavíme se a vrátíme tok f.
6. Pročistíme síť R.
7. $g \leftarrow$ blokující tok v R
8. Zlepšíme tok f pomocí g.

Výstup: Maximální tok f

Nyní je potřeba domyslet *čištění sítě.* Situaci můžeme sledovat na obrázku 14.7. Síť rozdělíme na vrstvy podle vzdálenosti od zdroje. Hrany vedoucí uvnitř vrstvy nebo do minulých vrstev (na obrázku šedivé) určitě neleží na nejkratších cestách. Ostatní hrany vedou

o právě jednu vrstvu dopředu, ale některé z nich vedou do „slepé uličky“ (na obrázku tečkované), takže je také musíme odstranit.



Obrázek 14.7: Síť rozdělená na vrstvy. Šedivé a tečkované hrany během čištění zmizí, plné zůstanou.

Procedura ČIŠTĚNÍSÍTĚ

1. Rozdělíme vrcholy do vrstev podle vzdálenosti od z.
2. Odstraníme vrstvy za s (tedy vrcholy ve vzdálenosti větší než ℓ).
3. Odstraníme hrany do předchozích vrstev a hrany uvnitř vrstev.
4. Odstraníme „slepé uličky“, tedy vrcholy s $\deg^{\text{out}}(v) = 0$:
5. $F \leftarrow \{v \neq z, s \mid \deg^{\text{out}}(v) = 0\}$ ◁ *fronta vrcholů ke smazání*
6. Dokud $F \neq \emptyset$, opakujeme:
7. Odebereme vrchol v z F.
8. Smažeme ze sítě vrchol v i všechny hrany, které do něj vedou.
9. Pokud nějakému vrcholu klesl $\deg^{\text{out}}$ na 0, přidáme ho do F.

Nakonec doplníme *hledání blokujícího toku.* Začneme s nulovým tokem g a budeme ho postupně zlepšovat. Pokaždé najdeme nějakou orientovanou cestu ze zdroje do stoku – to se ve vrstevnaté síti dělá snadno: stačí vyrazit ze zdroje a pak vždy následovat libovolnou hranu. Až cestu najdeme, tok g podél ní zlepšíme, jak nejvíce to půjde.

Pokud nyní tok na nějakých hranách dosáhl jejich rezervy, tyto hrany smažeme. Tím jsme mohli porušit pročištěnost – pakliže nějaký vrchol přišel o poslední odchozí nebo poslední příchozí hranu. Takových vrcholů se opět pomocí fronty zbavíme a síť dočistíme. Pokračujeme zlepšováním po dalších cestách, dokud nějaké existují.

Procedura BLOKUJÍCÍTOK

Vstup: Vrstevnatá síť R s rezervami r

1. $g \leftarrow$ nulový tok

2. Dokud v R existuje orientovaná cesta P ze z do s, opakujeme:
3. $\quad\varepsilon \leftarrow \min_{e\in P}(r(e) - g(e))$
4. $\quad$Pro všechny $e \in P : g(e) \leftarrow g(e) + \varepsilon$.
5. $\quad$Pokud pro kteroukoliv e nastalo $g(e) = r(e)$, smažeme e z R.
6. $\quad$Dočistíme síť pomocí fronty.

Výstup: Blokující tok g

Analýza Dinicova algoritmu

Lemma K (o korektnosti): Pokud se algoritmus zastaví, vydá maximální tok.

Důkaz: Z lemmatu o zlepšování toků plyne, že f je stále korektní tok. Algoritmus se zastaví tehdy, když už neexistuje cesta ze z do s po hranách s kladnou rezervou. Tehdy by se zastavil i Fordův-Fulkersonův algoritmus a ten, jak už víme, je korektní. □

Nyní rozebereme časovou složitost. Rozdělíme si k tomu účelu algoritmus na *fáze* – tak budeme říkat jednotlivým průchodům vnějším cyklem. Také budeme předpokládat, že síť na vstupu neobsahuje izolované vrcholy, takže $\mathcal{O}(n + m) = \mathcal{O}(m)$.

Lemma S (o složitosti fází): Každá fáze trvá $\mathcal{O}(nm)$.

Důkaz: Sestrojení sítě rezerv, mazání hran s nulovou rezervou, hledání nejkratší cesty i konečné zlepšování toku trvají $\mathcal{O}(m)$.

Čištění sítě (i se všemi dočišťováními během hledání blokujícího toku) pracuje taktéž v $\mathcal{O}(m)$: Smazání hrany trvá konstantní čas, smazání vrcholu po smazání všech incidentních hran taktéž. Každý vrchol i hrana jsou smazány nejvýše jednou za fázi.

Hledání blokujícího toku projde nejvýše m cest, protože pokaždé ze sítě vypadne alespoň jedna hrana (ta, na níž se v kroku 3 nabývalo minimum) a už se tam nevrátí. Jelikož síť je vrstevnatá, nalézt jednu cestu stihneme v $\mathcal{O}(n)$. Celkem tedy spotřebujeme čas $\mathcal{O}(nm)$ plus čištění, které jsme ale už započítali.

Celá jedna fáze proto doběhne v čase $\mathcal{O}(m + m + nm) = \mathcal{O}(nm)$. □

Zbývá určit, kolik proběhne fází. K tomu se bude hodit následující lemma:

Lemma C (o délce cest): Délka ℓ nejkratší cesty ze z do s vypočtená v kroku 4 Dinicova algoritmu vzroste po každé fázi alespoň o 1.

Důkaz: Označme R_i síť rezerv v i-té fázi poté, co jsme z ní smazali hrany s nulovou rezervou, ale ještě před pročištěním. Nechť nejkratší cesta ze z do s v R_i je dlouhá ℓ.

Jak se liší R_{i+1} od R_i? Především jsme z každé cesty délky ℓ smazali alespoň jednu hranu: každá taková cesta totiž byla blokujícím tokem zablokována, takže alespoň jedné

její hraně klesla rezerva na nulu a hrana vypadla. Žádná z původních cest délky ℓ tedy již v R_{i+1} neexistuje.

To ovšem nestačí – hrany mohou také přibývat. Pokud nějaká hrana měla nulovou rezervu a během fáze jsme zvýšili tok v protisměru, rezerva se zvětšila a hrana se v R_{i+1} najednou objevila. Ukážeme ale, že všechny cesty, které tím nově vznikly, jsou dostatečně dlouhé.

Rozdělme vrcholy grafu do vrstev podle vzdáleností od zdroje v R_i. Tok jsme zvyšovali pouze na hranách vedoucích o jednu vrstvu dopředu, takže jediné hrany, které se mohou v R_{i+1} objevit, vedou o jednu vrstvu zpět. Jenže každá cesta ze zdroje do spotřebiče, která se alespoň jednou vrátí o vrstvu zpět, musí mít délku alespoň $\ell + 2$ (spotřebič je v ℓ-té vrstvě a neexistují hrany, které by vedly o více než 1 vrstvu dopředu). □

Důsledek: Proběhne maximálně n fází.

Důkaz: Cesta ze z do s obsahuje nejvýše n hran, takže k prodloužení cesty dojde nejvýše n-krát. □

Věta: Dinicův algoritmus najde maximální tok v čase $\mathcal{O}(n^2m)$.

Důkaz: Podle právě vysloveného důsledku proběhne nejvýše n fází. Každá z nich podle lemmatu **S** trvá $\mathcal{O}(nm)$, což dává celkovou složitost $\mathcal{O}(n^2m)$. Speciálně se tedy algoritmus vždy zastaví, takže podle lemmatu **K** vydá maximální tok. □

Poznámka: Na rozdíl od Fordova-Fulkersonova algoritmu jsme tentokrát nikde nevyžadovali racionálnost kapacit – odhad časové složitosti se o kapacity vůbec neopírá. Nezávisle jsme tedy dokázali, že i v sítích s iracionálními kapacitami vždy existuje alespoň jeden maximální tok.

V sítích s malými celočíselnými kapacitami se navíc algoritmus chová daleko lépe, než říká náš odhad. Snadno se dá dokázat, že pro jednotkové kapacity doběhne v čase $\mathcal{O}(nm)$ (stejně jako Fordův-Fulkersonův). Uveďme bez důkazu ještě jeden silnější výsledek: v síti vzniklé při hledání největšího párování algoritmem z minulé kapitoly Dinicův algoritmus pracuje v čase $\mathcal{O}(\sqrt{n} \cdot m)$.

Cvičení

1. Všimněte si, že algoritmus skončí tím, že smaže všechny vrcholy i hrany. Také si všimněte, že vrcholy s nulovým vstupním stupněm jsme ani nemuseli mazat, protože se do nich algoritmus při hledání cest nikdy nedostane.

2. Odsimululte běh Dinicova algoritmu na svém řešení cvičení 14.2.1.

3. Dokažte, že pro jednotkové kapacity Dinicův algoritmus doběhne v čase $\mathcal{O}(nm)$.

4. Dokažte totéž pro celočíselné kapacity omezené konstantou.

5. Blokující tok lze také sestrojit pomocí prohledávání do hloubky. Pokaždé, když projdeme hranou, přepočítáme průběžné minimum. Pokud najdeme stok, vracíme se do kořene a upravujeme tok na hranách. Pokud narazíme na slepou uličku, vrátíme se o krok zpět a smažeme hranu, po níž jsme přišli. Doplňte detaily tak, aby zůstala zachovaná časová složitost $\mathcal{O}(n^2m)$.

6. Sestrojte vrstevnatou síť, v níž hledání blokujícího toku trvá $\Omega(nm)$.

7. Sestrojte síť, na níž Dinicův algoritmus provede $\Omega(n)$ fází.

8. Zkombinujte předchozí dvě cvičení a vytvořte síť, na níž Dinicův algoritmus běží v čase $\Omega(n^2m)$.

9.** *Algoritmus tří Indů:* Blokující tok ve vrstevnaté síti lze nalézt chytřejším způsobem v čase $\mathcal{O}(n^2)$, čímž zrychlíme celý Dinicův algoritmus na $\mathcal{O}(n^3)$. Následuje stručný popis, doplňte k němu detaily.

   Pro každý vrchol v definujme $r^+(v)$ jako součet rezerv na všech hranách vedoucích do v. Nechť dále $r^-(v)$ je totéž přes hrany vedoucí z v a $r(v) = \min(r^+(v), r^-(v))$ „rezerva vrcholu“. Pokud je $r(v)$ všude 0, tok už je blokující.

   V opačném případě opakovaně vybíráme nejmenší $r(v)$ a snažíme se ho vynulovat. Potřebujeme tedy dopravit $r(v)$ jednotek toku ze zdroje do v a totéž množství z v do stoku. Popišme dopravu do stoku (ze zdroje postupujeme symetricky): ve vrcholech udržujeme plán $p(w)$, který říká, kolik potřebujeme z w dopravit do stoku. Na začátku je $p(v) = r(v)$ a všechna ostatní $p(w) = 0$. Procházíme po vrstvách od v ke stoku a pokaždé plán převedeme po hranách s kladnou rezervou do vrcholů v další vrstvě. Jelikož $r(v) \leq r(w)$ pro všechna w, vždy nám to vyjde. Průběžně čistíme slepé uličky.

14.5 Goldbergův algoritmus

Představíme si ještě jeden algoritmus pro hledání maximálního toku v síti, navržený Andrewem Goldbergem. Bude daleko jednodušší než Dinicův algoritmus z předchozí kapitoly a po pár snadných úpravách bude mít stejnou, nebo dokonce lepší časovou složitost. Jednoduchost algoritmu bude ale vykoupena trochu složitějším rozborem jeho správnosti a efektivity.

Vlny, přebytky a výšky

Předchozí algoritmy začínaly s nulovým tokem a postupně ho zlepšovaly, až se stal maximálním. Goldbergův algoritmus naproti tomu začne s ohodnocením hran, které ani nemusí

být tokem, a postupně ho upravuje a zmenšuje, až se z něj stane tok, a to dokonce tok maximální.

Definice: Funkce $f : E \to \mathbb{R}_0^+$ je *vlna* v síti (V, E, z, s, c), splňuje-li obě následující podmínky:

- $\forall e \in E : f(e) \leq c(e)$ (vlna nepřekročí kapacity hran),
- $\forall v \in V \setminus \{z, s\} : f^\Delta(v) \geq 0$ (přebytek ve vrcholech je nezáporný).

Každý tok je tedy vlnou, ale opačně tomu tak být nemusí. V průběhu výpočtu se tedy potřebujeme postupně zbavit nenulových přebytků ve všech vrcholech kromě zdroje a spotřebiče. K tomu bude sloužit následující operace:

Definice: *Převedení přebytku* po hraně uv můžeme provést, pokud $f^\Delta(u) > 0$ a $r(uv) > 0$. Proběhne tak, že po hraně uv pošleme $\delta = \min(f^\Delta(u), r(uv))$ jednotek toku, podobně jako v předchozích algoritmech buď přičtením po směru nebo odečtením proti směru.

Pozorování: Převedení změní přebytky a rezervy následovně:

$$\begin{aligned} f'^\Delta(u) &= f^\Delta(u) - \delta \\ f'^\Delta(v) &= f^\Delta(v) + \delta \\ r'(uv) &= r(uv) - \delta \\ r'(vu) &= r(vu) + \delta \end{aligned}$$

Rádi bychom postupným převáděním všechny přebytky přepravili do spotřebiče, nebo je naopak přelili zpět do zdroje. Chceme se ovšem vyhnout přelévání přebytků tam a zase zpět, takže vrcholům přiřadíme *výšky* – to budou nějaká přirozená čísla $h(v)$.

Přebytek pak budeme ochotni převádět pouze z vyššího vrcholu do nižšího. Pokud se stane, že nalezneme vrchol s přebytkem, ze kterého nevede žádná nenasycená hrana směrem dolů, budeme tento vrchol *zvedat* – tedy zvyšovat mu výšku po jedné, než se dostane dostatečně vysoko, aby z něj přebytek mohl odtéci.

Získáme tak následující algoritmus:

Algoritmus GOLDBERG
Vstup: Síť (V, E, z, s, c)

1. Nastavíme počáteční výšky: ◁ *zdroj ve výšce n, ostatní ve výšce 0*
2. $h(z) \leftarrow n$
3. $h(v) \leftarrow 0$ pro všechny $v \neq z$
4. Vytvoříme počáteční vlnu: ◁ *všechny hrany ze z na maximum*
5. $f \leftarrow$ všude nulová funkce

6. $f(zv) \leftarrow c(zv)$, kdykoliv $zv \in E$
7. Dokud existuje vrchol $u \neq z, s$ takový, že $f^{\Delta}(u) > 0$:
8. Pokud existuje hrana uv s $r(uv) > 0$ a $h(u) > h(v)$, *převedeme přebytek* po hraně uv.
9. V opačném případě *zvedneme* u: $h(u) \leftarrow h(u) + 1$.

Výstup: Maximální tok f

Analýza algoritmu*

Algoritmus je jednoduchý, ale na první pohled není vidět ani to, že se vždy zastaví, natož že by měl vydat maximální tok. Postupně dokážeme několik invariantů a lemmat a pomocí nich se dobereme důkazu správnosti a časové složitosti.

Invariant A (základní): V každém kroku algoritmu platí:

1. Funkce f je vlna.
2. Výška $h(v)$ žádného vrcholu v nikdy neklesá.
3. $h(z) = n$ a $h(s) = 0$.
4. $f^{\Delta}(v) \geq 0$ v každém vrcholu $v \neq z$.

Důkaz: Indukcí dle počtu průchodů cyklem (7. – 9. krok algoritmu):

- Po inicializaci algoritmu je vše v pořádku: přebytky všech vrcholů mimo zdroj jsou nezáporné, výšky souhlasí.
- Při převedení přebytku: Z definice převedení přímo plyne, že neporušuje kapacity a nevytváří záporné přebytky. Výšky se nemění.
- Při zvednutí vrcholu: Tehdy se naopak mění jen výšky, ale pouze u vrcholů různých od zdroje a stoku. Výšky navíc pouze rostou. □

Invariant S (o spádu): Neexistuje hrana uv, která by měla kladnou rezervu a spád $h(u) - h(v)$ větší než 1.

Důkaz: Indukcí dle běhu algoritmu. Na začátku mají všechny hrany ze zdroje rezervu nulovou a všechny ostatní vedou mezi vrcholy s výškou 0 nebo do kopce. V průběhu výpočtu by se tento invariant mohl pokazit pouze dvěma způsoby:

- Zvednutím vrcholu u, ze kterého vede hrana uv s kladnou rezervou a spádem 1. Tento případ nemůže nastat, neboť algoritmus by dal přednost převedení přebytku po této hraně před zvednutím.
- Zvětšením rezervy hrany se spádem větším než 1. Toto také nemůže nastat, neboť rezervu bychom mohli zvětšit jedině tak, že bychom poslali něco v protisměru – a to nesmíme, jelikož bychom převáděli přebytek z nižšího vrcholu do vyššího. □

Lemma K (o korektnosti): Když se algoritmus zastaví, f je maximální tok.

Důkaz: Nejprve ukážeme, že f *je tok:* Omezení na kapacity splňuje tok stejně jako vlna, takže postačí dokázat, že platí Kirchhoffův zákon. Ten požaduje, aby přebytky ve všech vrcholech kromě zdroje a spotřebiče byly nulové. To ovšem musí být, protože nenulový přebytek by musel být kladný a algoritmus by se dosud nezastavil.

Zbývá zdůvodnit, že f *je maximální:* Pro spor předpokládejme, že tomu tak není. Ze správnosti Fordova-Fulkersonova algoritmu plyne, že tehdy musí existovat nenasycená cesta ze zdroje do stoku. Uvažme libovolnou takovou cestu. Zdroj je stále ve výšce n a stok ve výšce 0 (viz invariant **A**). Tato cesta tedy překonává spád n, ale může mít nejvýše $n-1$ hran. Proto se v ní nachází alespoň jedna hrana se spádem alespoň 2. Jelikož je tato hrana součástí nenasycené cesty, musí být sama nenasycená, což je spor s invariantem **S**. Tok je tedy maximální. □

Invariant C (cesta do zdroje): Mějme vrchol v, jehož přebytek $f^\Delta(v)$ je kladný. Pak existuje nenasycená cesta z tohoto vrcholu do zdroje.

Důkaz: Buď v vrchol s kladným přebytkem. Uvažme množinu

$$A := \{u \in V \mid \text{existuje nenasycená cesta z } v \text{ do } u\}.$$

Ukážeme, že tato množina obsahuje zdroj.

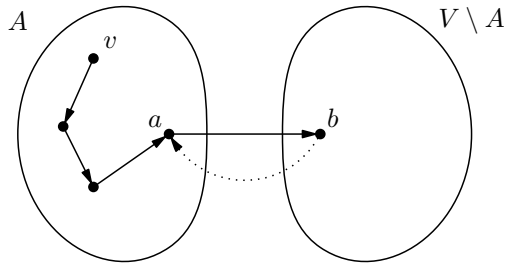
Použijeme už mírně okoukaný trik: sečteme přebytky ve všech vrcholech množiny A. Všechny hrany ležící celé uvnitř A nebo celé venku přispějí dohromady nulou. Stačí tedy započítat pouze hrany vedoucí ven z A, nebo naopak zvenku dovnitř. Získáme:

$$\sum_{u\in A} f^\Delta(u) = \underbrace{\sum_{ba\in E(V\setminus A,A)} f(ba)}_{=0} - \underbrace{\sum_{ab\in E(A,V\setminus A)} f(ab)}_{\geq 0} \leq 0.$$

Ukážeme, že první svorka je rovna nule (sledujme obrázek 14.8). Mějme hranu ab ($a \in A$, $b \in V \setminus A$). Ta musí mít nulovou rezervu – jinak by totiž i vrchol b patřil do A. Proto po hraně ba nemůže nic téci.

Druhá svorka je evidentně nezáporná, protože je to součet nezáporných ohodnocení hran.

Proto součet přebytků přes množinu A je menší nebo roven nule. Zároveň však v A leží aspoň jeden vrchol s kladným přebytkem, totiž v, tudíž v A musí být také nějaký vrchol se záporným přebytkem – a jediný takový je zdroj. Tím je dokázáno, že z leží v A, tedy že vede nenasycená cesta z vrcholu v do zdroje. □



Obrázek 14.8: Situace v důkazu invariantu **C**

Invariant V (o výšce): Pro každý vrchol v je $h(v) \leq 2n$.

Důkaz: Kdyby existoval vrchol v s výškou $h(v) > 2n$, mohl se do této výšky dostat pouze zvednutím z výšky alespoň $2n$. Vrchol přitom zvedáme jen tehdy, má-li kladný přebytek. Dle invariantu **C** musela v tomto okamžiku existovat nenasycená cesta z v do zdroje. Ta nicméně překonávala spád alespoň n, ale mohla mít nejvýše $n-1$ hran. Tudíž musela obsahovat nenasycenou hranu se spádem alespoň 2 a máme spor s invariantem **S**. □

Lemma Z (počet zvednutí): Během výpočtu nastane nejvýše $2n^2$ zvednutí.

Důkaz: Z předchozího invariantu plyne, že každý z n vrcholů mohl být zvednut nejvýše $2n$-krát. □

Teď nám ještě zbývá určit počet provedených převedení. Bude se nám hodit, když převedení rozdělíme na dva druhy:

Definice: Řekneme, že převedení po hraně uv je *nasycené*, pokud po převodu rezerva $r(uv)$ klesla na nulu. V opačném případě je *nenasycené*, a tehdy určitě klesne přebytek $f^{\Delta}(u)$ na nulu (to se nicméně může stát i při nasyceném převedení).

Lemma S (nasycená převedení): Nastane nejvýše nm nasycených převedení.

Důkaz: Zvolíme hranu uv a spočítáme, kolikrát jsme po ní mohli nasyceně převést.

Po prvním nasyceném převedení z u do v se vynulovala rezerva hrany uv. V tomto okamžiku muselo být u výše než v, a dokonce víme, že bylo výše přesně o 1 (invariant **S**). Než nastane další převedení po této hraně, hrana musí opět získat nenulovou rezervu. Jediný způsob, jak k tomu může dojít, je převedením části přebytku z v zpátky do u. Na to se musí v dostat (alespoň o 1) výše než u. A abychom provedli nasycené převedení znovu ve směru z u do v, musíme u dostat (alespoň o 1) výše než v. Proto musíme u alespoň o 2 zvednout – nejprve na úroveň v a pak ještě o 1 výše.

Ukázali jsme tedy, že mezi každými dvěma nasycenými převedeními po hraně uv musel být vrchol u alespoň dvakrát zvednut. Podle lemmatu **V** k tomu ale mohlo dojít nejvýše n-krát za celý výpočet, takže všech nasycených převedení po hraně uv je nejvýše n a po všech hranách dohromady nejvýše nm. □

Předchozí dvě lemmata jsme dokazovali „lokálním“ způsobem – zvednutí jsme počítali pro každý vrchol zvlášť a nasycená převedení pro každou hranu. Tento přístup pro nenasycená převedení nefunguje, jelikož jich lokálně může být velmi mnoho. Podaří se nám však omezit jejich celkový počet.

Použijeme potenciálovou metodu, která se již osvědčila při amortizované analýze v oddílu 9.3. Pořídíme si *potenciál,* což bude nějaká *nezáporná* funkce, která popisuje stav výpočtu. Pro každou operaci pak stanovíme, jaký vliv má na hodnotu potenciálu. Z toho odvodíme, že operací, které potenciál snižují, nemůže být výrazně více než těch, které ho zvyšují. Jinak by totiž potenciál musel někdy během výpočtu klesnout pod nulu.

Někdy bývá potenciál přímočará funkce, ale v následujícím lemmatu bude trochu složitější. Zvolíme ho tak, aby operace, jejichž počty už známe (zvednutí, nasycené převedení), přispívaly nanejvýš malými kladnými čísly, a nenasycená převedení potenciál vždy snižovala.

Lemma N (nenasycená převedení): Počet všech nenasycených převedení je $\mathcal{O}(n^2m)$.

Důkaz: Uvažujme následující potenciál:

$$\Phi := \sum_{\substack{v \neq z,s \\ f^\Delta(v)>0}} h(v).$$

Každý vrchol s kladným přebytkem tedy přispívá svou výškou. Sledujme, jak se tento potenciál během výpočtu vyvíjí:

- Na počátku je $\Phi = 0$.
- Během celého algoritmu je $\Phi \geq 0$, neboť potenciál je součtem nezáporných členů.
- Zvednutí vrcholu zvýší Φ o jedničku. (Aby byl vrchol zvednut, musel mít kladný přebytek, takže vrchol do sumy již přispíval. Teď jen přispěje číslem o 1 vyšším.) Již víme, že za celý průběh algoritmu nastane maximálně $2n^2$ zvednutí, pročež zvedáním vrcholů zvýšíme potenciál dohromady nejvýše o $2n^2$.
- Nasycené převedení zvýší Φ nejvýše o $2n$: Pokud vrchol v měl původně nulový přebytek, bude mít teď kladný a do sumy začne přispívat; tím se suma zvýší o $h(v) \leq 2n$.

Vrchol u již přispíval a buďto bude přispívat nadále, nebo se suma sníží. Podle lemmatu **S** nastane nejvýše nm nasycených převedení a ta celkově potenciál zvýší maximálně o $2n^2m$.

- Konečně když převádíme po hraně uv nenasyceně, tak od potenciálu určitě odečteme výšku vrcholu u (neboť se vynuluje přebytek ve vrcholu u) a možná přičteme výšku vrcholu v (nevíme, zda tento vrchol předtím měl přebytek). Jenže $h(v) = h(u) - 1$, a proto nenasycené převedení potenciál vždy sníží alespoň o jedna.

Potenciál celkově stoupne o nejvyše $2n^2+2n^2m = \mathcal{O}(n^2m)$ a při nenasycených převedeních klesá, pokaždé alespoň o 1. Proto je všech nenasycených převedení $\mathcal{O}(n^2m)$. □

Implementace

Zbývá vyřešit, jak síť a výšky reprezentovat, abychom dokázali rychle hledat vrcholy s přebytkem a nenasycené hrany vedoucí s kopce.

Budeme si pamatovat seznam P všech vrcholů s kladným přebytkem. Když měníme přebytek nějakého vrcholu, můžeme tento seznam v konstantním čase aktualizovat – buďto vrchol do seznamu přidat, nebo ho naopak odebrat. (K tomu se hodí, aby si vrcholy pamatovaly ukazatel na svou polohu v seznamu P). V konstantním čase také umíme odpovědět, zda existuje nějaký vrchol s přebytkem.

Dále si pro každý vrchol u budeme udržovat seznam $L(u)$. Ten bude uchovávat všechny nenasycené hrany, které vedou z u dolů (mají spád alespoň 1). Opět při změnách rezerv můžeme tyto seznamy v konstantním čase upravit.

Jednotlivé operace budou mít tyto složitosti:

- *Inicializace* algoritmu – triviálně $\mathcal{O}(m)$.
- *Výběr vrcholu* s kladným přebytkem a nalezení nenasycené hrany vedoucí dolů – $\mathcal{O}(1)$ (stačí se podívat na počátky příslušných seznamů).
- *Převedení přebytku* po hraně uv – změny rezerv $r(uv)$ a $r(vu)$ způsobí přepočítání seznamů $L(u)$ a $L(v)$, změny přebytků $f^\Delta(u)$ a $f^\Delta(v)$ mohou způsobit změnu v seznamu P. Vše v čase $\mathcal{O}(1)$.
- *Zvednutí vrcholu* u může způsobit, že nějaká hrana s kladnou rezervou, která původně vedla po rovině, začne vést z u dolů. Nebo se naopak může stát, že hrana, která původně vedla s kopce do u, najednou vede po rovině. Musíme proto obejít všechny hrany do u a z u, kterých je nejvýše $2n$, porovnat výšky a případně tyto hrany uv odebrat ze seznamu $L(v)$, resp. přidat do $L(u)$. To trvá $\mathcal{O}(n)$.

Vidíme, že zvednutí je sice drahé, ale zase je jich poměrně málo. Naopak převádění přebytků je častá operace, takže je výhodné, že trvá konstantní čas.

Věta: Goldbergův algoritmus najde maximální tok v čase $\mathcal{O}(n^2m)$.

Důkaz: Inicializace algoritmu trvá $\mathcal{O}(m)$. Pak algoritmus provede nejvýše $2n^2$ zvednutí (viz lemma **Z**), nejvýše nm nasycených převedení (lemma **S**) a $\mathcal{O}(n^2m)$ nenasycených převedení (lemma **N**). Vynásobením složitostmi jednotlivých operací dostaneme čas $\mathcal{O}(n^3 + nm + n^2m) = \mathcal{O}(n^2m)$. Jakmile se algoritmus zastaví, podle lemmatu **K** vydá maximální tok. □

Cvičení

1. Rozeberte chování Goldbergova algoritmu na sítích s jednotkovými kapacitami. Bude rychlejší než ostatní algoritmy?

2. Co by se stalo, kdybychom v inicializaci algoritmu umístili zdroj do výšky $n-1$, $n-2$, anebo $n-3$?

14.6* Vylepšení Goldbergova algoritmu

Základní verze Goldbergova algoritmu dosáhla stejné složitosti jako Dinicův algoritmus. Nyní ukážeme, že drobnou úpravou lze Goldbergův algoritmus ještě zrychlit. Postačí ze všech vrcholů s přebytkem pokaždé vybírat ten nejvyšší.

Při rozboru časové složitosti původního algoritmu hrál nejvýznamnější roli člen $\mathcal{O}(n^2m)$ za nenasycená převedení. Ukážeme, že ve vylepšeném algoritmu jich nastane řádově méně.

Lemma N': Goldbergův algoritmus s volbou nejvyššího vrcholu provede $\mathcal{O}(n^3)$ nenasycených převedení.

Důkaz: Dokazovat budeme opět pomocí potenciálové metody. Vrcholy rozdělíme do hladin podle výšky. Speciálně nás bude zajímat *nejvyšší hladina s přebytkem*:

$$H := \max\{h(v) \mid v \neq z, s \ \wedge \ f^{\Delta}(v) > 0\}.$$

Rozdělíme běh algoritmu na *fáze.* Každá fáze končí tím, že se H změní. Buďto se H zvýší, což znamená, že nějaký vrchol s přebytkem v nejvyšší hladině byl o 1 zvednut, anebo se H sníží. Už víme, že v průběhu výpočtu nastane $\mathcal{O}(n^2)$ zvednutí, což shora omezuje počet zvýšení H. Zároveň si můžeme uvědomit, že H je nezáporný potenciál a snižuje se i zvyšuje přesně o 1. Počet snížení bude proto omezen počtem zvýšení. Tím pádem nastane všeho všudy $\mathcal{O}(n^2)$ fází.

Během jedné fáze přitom provedeme nejvýše jedno nenasycené převedení z každého vrcholu. Po každém nenasyceném převedení po hraně uv se totiž vynuluje přebytek v u a aby se provedlo další nenasycené převedení z vrcholu u, muselo by nejdříve být co převádět. Muselo by tedy do u něco přitéci. My ale víme, že převádíme pouze shora dolů a u je

v nejvyšší hladině (to zajistí právě ono vylepšení algoritmu), tedy nejdříve by musel být nějaký jiný vrchol zvednut. Tím by se ale změnilo H a skončila by tato fáze.

Proto počet všech nenasycených převedení během jedné fáze je nejvýše n. A již jsme dokázali, že fází je $\mathcal{O}(n^2)$. Tedy počet všech nenasycených převedení je $\mathcal{O}(n^3)$. □

Ve skutečnosti je i tento odhad trochu nadhodnocený. Trochu složitějším argumentem lze dokázat těsnější odhad, který se hodí zvláště u řídkých grafů.

Lemma N'': Počet nenasycených převedení je $\mathcal{O}(n^2\sqrt{m})$.

Důkaz: Zavedeme fáze stejně jako v důkazu předchozí verze lemmatu a rozdělíme je na dva druhy. Pro každý druh pak odhadneme celkový počet převedení jiným způsobem.

Nechť k je nějaké kladné číslo, jehož hodnotu určíme později. *Laciné* nazveme ty fáze, během nichž se provede nejvýše k nenasycených převedení. *Drahé* fáze budou všechny ostatní.

Nejprve rozebereme chování laciných fází. Jejich počet shora odhadneme počtem všech fází, tedy $\mathcal{O}(n^2)$. Nenasycených převedení se během jedné laciné fáze provede nejvíce k, za všechny laciné fáze dohromady to činí $\mathcal{O}(n^2k)$.

Pro počet nenasycených převedení v drahých fázích zavedeme nový potenciál:

$$\Psi := \sum_{\substack{v \neq z,s \\ f^\Delta(v)\neq 0}} p(v),$$

kde $p(v)$ je počet vrcholů u, které nejsou výše než v. Jelikož $p(v)$ je nezáporné a nikdy nepřesáhne počet všech vrcholů, potenciál Ψ bude také vždy nezáporný a nepřekročí n^2. Rozmysleme si, jak bude potenciál ovlivňován operacemi algoritmu:

- *Inicializace:* Počáteční potenciál je nejvýše n^2.
- *Zvednutí vrcholu v*: Hodnota $p(v)$ se zvýší nejvýše o n a všechna ostatní $p(w)$ se buďto nezmění, nebo klesnou o 1. Bez ohledu na přebytky vrcholů se tedy potenciál zvýší nejvýše o n.
- *Nasycené převedení* po hraně uv: Hodnoty $p(\ldots)$ se nezmění, ale mění se přebytky – vrcholu u se snižuje, vrcholu v zvyšuje. Z potenciálu proto může zmizet člen $p(u)$ a naopak přibýt $p(v)$. Potenciál Ψ tedy vzroste nejvýše o n.
- *Nenasycené převedení* po hraně uv: Hodnoty $p(\ldots)$ se opět nemění. Přebytek v u se vynuluje, což sníží Ψ o $p(u)$. Přebytek v se naopak zvýší, takže pokud byl předtím nulový, Ψ se zvýší o $p(v)$. Celkově tedy Ψ klesne alespoň o $p(u) - p(v)$.

Teď využijeme toho, že pokud převádíme po hraně uv, má tato hrana spád 1. Výraz $p(u) - p(v)$ tedy udává počet vrcholů na hladině $h(u)$, což je nejvyšší hladina s přebytkem. Z předchozího důkazu víme, že těchto vrcholů je alespoň tolik, kolik je nenasycených převedení během dané fáze.

Z toho plyne, že nenasycené převedení provedené během drahé fáze sníží potenciál alespoň o k. Převedení v laciných fázích ho nesnižuje tak výrazně, ale důležité je, že ho určitě nezvýší.

Potenciál Ψ se tedy může zvětšit pouze při operacích inicializace, zvednutí a nasyceného převedení. Inicializace přispěje n^2. Všech zvednutí se provede celkem $\mathcal{O}(n^2)$ a každé zvýší potenciál nejvýše o n. Nasycených převedení se provede celkem $\mathcal{O}(nm)$ a každé zvýší potenciál taktéž nejvýše o n. Celkem se tedy Ψ zvýší nejvýše o:

$$n^2 + n \cdot \mathcal{O}(n^2) + n \cdot \mathcal{O}(nm) = \mathcal{O}(n^3 + n^2 m).$$

Teď využijeme toho, že Ψ je nezáporný potenciál, tedy když ho každé nenasycené převedení v drahé fázi sníží Ψ alespoň o k, může takových převedení nastat nejvýše $\mathcal{O}(n^3/k + n^2 m/k)$. To nyní sečteme s odhadem pro laciné fáze a dostaneme, že všech nenasycených převedení proběhne

$$\mathcal{O}\left(n^2 k + \frac{n^3}{k} + \frac{n^2 m}{k}\right) = \mathcal{O}\left(n^2 k + \frac{n^2 m}{k}\right)$$

(využili jsme toho, že v grafech bez izolovaných vrcholů je $n = \mathcal{O}(m)$, a tedy $n^3 = \mathcal{O}(n^2 m)$).

Tento odhad ovšem platí pro libovolnou volbu k. Proto zvolíme takové k, aby byl co nejnižší. Jelikož první člen s rostoucím k roste a druhý klesá, asymptotické minimum nastane tam, kde se tyto členy vyrovnají, tedy když $n^2 k = n^2 m/k$.

Nastavíme tedy $k = \sqrt{m}$ a získáme kýžený odhad $\mathcal{O}(n^2\sqrt{m})$. □

Cvičení

1. Navrhněte implementaci vylepšeného Goldbergova algoritmu se zvedáním nejvyššího vrcholu s přebytkem. Snažte se dosáhnout časové složitosti $\mathcal{O}(n^2\sqrt{m})$.

14.7 Další cvičení

1. *Svišti:* Na louce je n svišťů a m děr v zemi (obojí je zadáno jako body v rovině nebo raději body v nepříliš velké celočíselné mřížce). Když se objeví orel, zvládne

svišť uběhnout pouze d metrů, než bude uloven. Kolik maximálně svišťů se může zachránit útěkem do díry, když jedna díra pojme nejvýše jednoho sviště? A co když pojme k svišťů?

2. *Parlamentní kluby:* V parlamentu s n poslanci je m různých klubů. Jeden poslanec může být členem mnoha různých klubů. Každý klub nyní potřebuje zvolit svého předsedu a tajemníka tak, aby všichni předsedové a tajemníci byli navzájem různé osoby (tedy aby nikdo „neseděl na více křeslech“). Navrhněte algoritmus, který zvolí všechny předsedy a tajemníky, případně oznámí, že řešení neexistuje. Mimochodem, za jakých podmínek je existence řešení garantována?

3. *Dopravní problém:* Uvažujme továrny $T_1, \ldots, T_p$ a obchody $O_1, \ldots, O_q$. Všichni vyrábějí a prodávají tentýž druh zboží. Továrna T_i ho denně vyprodukuje t_i kusů, obchod O_j denně spotřebuje o_j kusů. Navíc známe bipartitní graf určující, která továrna může dodávat zboží kterému obchodu. Najděte efektivní algoritmus, který zjistí, zda je požadavky obchodů možné splnit, aniž by se překročily výrobní kapacity továren, a pokud je to možné, vypíše, ze které továrny se má přepravit kolik zboží do kterého obchodu.

4.* *Průchod šachovnicí:* Je dána šachovnice $n \times n$, kde některá políčka jsou nepřístupná. Celý dolní řádek je obsazen figurkami, které se mohou hýbat o jedno pole dopředu, šikmo vlevo dopředu, či šikmo vpravo dopředu. V jednom tahu se všechny figurky naráz pohnou (mohou i zůstat stát na místě), na jednom políčku se však musí vyskytovat nejvýše jedna figurka. Ocitne-li se figurka na některém políčku horního řádku šachovnice, zmizí. Navrhněte algoritmus, který najde minimální počet tahů takový, že z šachovnice dokážeme odstranit všechny figurky, případně oznámí, že řešení neexistuje.

5.* *Minimální izolace:* Je dán nepříliš velký celočíselný kvádr (dejme tomu s objemem nejvýše 20 000) a v něm k nebezpečných jednotkových kostiček. Navrhněte algoritmus, který najde podmnožinu M kostiček kvádru takovou, že každá nebezpečná kostička leží v M a zároveň M má minimální možný povrch (povrch měříme jako počet takových stěn kostiček z M, které nesousedí s jinou kostičkou z M).

6.* *Doly a továrny:* Uvažujeme o vybudování dolů $D_1, \ldots, D_p$ a továren $T_1, \ldots, T_q$. Vybudování dolu D_i stojí cenu d_i a od té doby důl zadarmo produkuje neomezené množství i-té suroviny. Továrna T_j potřebuje ke své činnosti zadanou množinu surovin a pokud jsou v provozu všechny doly produkující tyto suroviny, vyděláme na továrně zisk t_j. Vymyslete algoritmus, jenž pro zadané ceny dolů, zisky továren a bipartitní graf závislostí továren na surovinách stanoví, které doly postavit, abychom vydělali co nejvíce.

7.* *Permanent* matice $n \times n$ je definovaný podobně jako determinant, jen bez alternace znamének. Nahlédněte, že na permanent se dá dívat jako na součet přes všechna rozestavení n neohrožujících se věží na políčka matice, přičemž sčítáme součiny políček pod věžemi. Jakou vlastnost bipartitního grafu vyjadřuje permanent bipartitní matice sousednosti? ($A_{ij} = 1$, pokud vede hrana mezi i-tým vrcholem nalevo a j-tým napravo.) Radost nám kazí pouze to, že na rozdíl od determinantů neumíme permanenty počítat v polynomiálním čase.

15 Paralelní algoritmy

15 Paralelní algoritmy

Pomocí počítačů řešíme stále složitější a rozsáhlejší úlohy a potřebujeme k tomu čím dál víc výpočetního výkonu. Rychlost a kapacita počítačů zatím rostla exponenciálně, takže se zdá, že stačí chvíli počkat. Jenže podobně rostou i velikosti problémů, které chceme řešit. Navíc exponenciální růst výkonu se určitě někdy zastaví – nečekáme třeba, že by bylo možné vyrábět transistory menší než jeden atom.

Jak si poradíme? Jedna z lákavých možností je zapřáhnout do jednoho výpočtu více procesorů najednou. Ostatně, vícejádrové procesory, které dneska najdeme ve svých stolních počítačích, nejsou nic jiného než miniaturní víceprocesorové systémy na jednom čipu.

Nabízí se tedy obtížnou úlohu rozdělit na několik částí, nechat každý procesor (či jádro) spočítat jednu z částí a nakonec jejich výsledky spojit dohromady. To se snadno řekne, ale s výjimkou triviálních úloh už obtížněji provede.

Pojďme se podívat na několik zajímavých paralelních algoritmů. Abychom se nemuseli zabývat detaily hardwaru konkrétního víceprocesorového počítače, zavedeme poměrně abstraktní výpočetní model, totiž hradlové sítě. Tento model je daleko paralelnější než skutečný počítač, ale přesto se techniky používané pro hradlové sítě hodí i prakticky. Konec konců sama vnitřní struktura procesorů se našemu modelu velmi podobá.

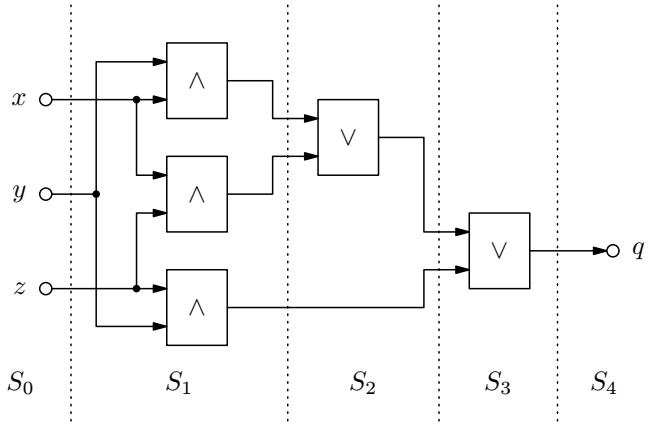
15.1 Hradlové sítě

Hradlové sítě jsou tvořeny navzájem propojenými *hradly.* Každé hradlo přitom počítá nějakou (obecně libovolnou) funkci $\Sigma^k \to \Sigma$. Množina Σ je konečná abeceda, stejná pro celou síť. Přirozené číslo k udává počet vstupů hradla, jinak též jeho *aritu.*

Příklad: Často studujeme hradla *booleovská* pracující nad abecedou $\Sigma = \{0, 1\}$. Ta počítají jednotlivé logické funkce, například:

- *nulární funkce:* to jsou konstanty (FALSE $= 0$, TRUE $= 1$),
- *unární funkce:* identita a negace (NOT, $\neg$),
- *binární funkce:* logický součin (AND, & či $\wedge$), součet (OR, $\vee$), aritmetický součet modulo 2 (XOR, $\oplus$), ...

Propojením hradel pak vznikne *hradlová síť.* Než vyřkneme formální definici, pojďme se podívat na příklad jedné takové sítě na obr. 15.1. Síť má tři vstupy, pět booleovských hradel a jeden výstup. Na výstupu je přitom jednička právě tehdy, jsou-li jedničky přítomny na alespoň dvou vstupech. Vrací tedy hodnotu, která na vstupech převažuje, neboli *majoritu.*



Obrázek 15.1: Hradlová síť pro majoritu ze tří vstupů

Obecně každá hradlová síť má nějaké vstupy, hradla a výstupy. Hradla dostávají data ze vstupů sítě a výstupů ostatních hradel. Výstupy hradel mohou být připojeny na libovolně mnoho vstupů dalších hradel, případně na výstupy sítě. Jediné omezení je, že v propojení nesmíme vytvářet cykly.

Nyní totéž formálněji:

Definice: *Hradlová síť* je určena:

- *Abecedou* Σ, což je nějaká konečná množina symbolů.
- Po dvou disjunktními konečnými množinami I (*vstupy*), O (*výstupy*) a H (*hradla*).
- Acyklickým orientovaným multigrafem (V, E) s množinou vrcholů $V = I \cup O \cup H$ (*multi*graf potřebujeme proto, abychom uměli výstup jednoho hradla připojit současně na více různých vstupů jiného hradla).
- Zobrazením F, které každému hradlu $h \in H$ přiřadí nějakou funkci $F(h) : \Sigma^{a(h)} \to \Sigma$, což je funkce, kterou toto hradlo vykonává. Číslu $a(h)$ říkáme *arita* hradla h.
- Zobrazením $z : E \to \mathbb{N}$, jež o hranách vedoucích do hradel říká, kolikátému argumentu funkce odpovídají. (Na hranách vedoucích do výstupů necháváme hodnotu této funkce nevyužitu.)

Přitom jsou splněny následující podmínky:

- Do vstupů nevedou žádné hrany.
- Z výstupů nevedou žádné hrany. Do každého výstupu vede právě jedna hrana.
- Do každého hradla vede tolik hran, kolik je jeho arita. Z každého hradla vede alespoň jedna hrana.
- Všechny vstupy hradel jsou zapojeny. Tedy pro každé hradlo h a každý jeho vstup $j \in \{1, \ldots, a(h)\}$ existuje právě jedna hrana e, která vede do hradla h a $z(e) = j$.

Na obrázcích většinou sítě kreslíme podobně jako elektrotechnická schémata: místo více hran z jednoho hradla raději nakreslíme jednu, která se cestou rozvětví. V místech křížení hran tečkou rozlišujeme, zda jsou hrany propojeny či nikoliv.

Poznámka: Někdy se hradlovým sítím také říká *kombinační obvody* a pokud pracují nad abecedou $\Sigma = \{0, 1\}$, tak *booleovské obvody*.

Definice: *Výpočet sítě* postupně přiřazuje hodnoty z abecedy Σ vrcholům grafu. Výpočet probíhá po *taktech*. V nultém taktu jsou definovány pouze hodnoty na vstupech sítě a v hradlech arity 0 (konstantách). V každém dalším taktu pak ohodnotíme vrcholy, jejichž všechny vstupní hrany vedou z vrcholů s již definovanou hodnotou.

Hodnotu hradla h přitom spočteme funkcí $F(h)$ z hodnot na jeho vstupech uspořádaných podle funkce z. Výstup sítě pouze zkopíruje hodnotu, která do něj po hraně přišla.

Jakmile budou po nějakém počtu taktů definované hodnoty všech vrcholů, výpočet se zastaví a síť vydá výsledek – ohodnocení výstupů.

Podle průběhu výpočtu můžeme vrcholy sítě rozdělit do vrstev (na obrázku 15.1 jsou naznačeny tečkovaně).

Definice: *i-tá vrstva S_i* obsahuje ty vrcholy, které vydají výsledek poprvé v i-tém taktu výpočtu.

Lemma (o průběhu výpočtu): Každý vrchol vydá v konečném čase výsledek (tedy patří do nějaké vrstvy) a tento výsledek se už nikdy nezmění.

Důkaz: Jelikož síť je acyklická, můžeme postupovat indukcí podle topologického pořadí vrcholů.

Pokud do vrcholu v nevede žádná hrana, vydá výsledek v 0. taktu. V opačném případě do v vedou hrany z nějakých vrcholů $u_1, \ldots, u_k$, které leží v topologickém pořadí před v, takže už víme, že vydaly výsledek v taktech $t_1, \ldots, t_k$. Vrchol v tedy musí vydat výsledek v taktu $\max_i t_i + 1$. A jelikož výsledky vrcholů $u_1, \ldots, u_k$ se nikdy nezmění, výsledek vrcholu v také ne. □

Každý výpočet se tedy zastaví, takže můžeme definovat časovou a prostorou složitost očekávaným způsobem.

Definice: *Časovou složitost* definujeme jako *hloubku sítě,* tedy počet vrstev obsahujících aspoň jeden vrchol. *Prostorová složitost* bude rovna počtu hradel v síti. Všimněte si, že čas ani prostor nezávisí na konkrétním vstupu, pouze na jeho délce.

Poznámka (o aritě hradel): Kdybychom připustili hradla s libovolně vysokým počtem vstupů, mohli bychom jakýkoliv problém se vstupem délky n a výstupem délky ℓ vyřešit v jedné vrstvě pomocí ℓ kusů n-vstupových hradel. Každému hradlu bychom prostě přiřadili funkci, která počítá příslušný bit výsledku ze všech bitů vstupu.

To není ani realistické, ani pěkné. Jak z toho ven? Omezíme arity všech hradel nějakou pevnou konstantou, třeba dvojkou. Budeme tedy používat výhradně nulární, unární a binární hradla. (Kdybychom zvolili jinou konstantu, dopadlo by to podobně, viz cvičení 7.)

Poznámka (o uniformitě): Od běžných výpočetních modelů, jako je třeba RAM, se hradlové sítě liší jednou podstatnou vlastností – každá síť zpracovává výhradně vstupy jedné konkrétní velikosti. Řešením úlohy tedy typicky není jedna síť, ale posloupnost sítí pro jednotlivé velikosti vstupu. Všechny sítě přitom používají stejné typy hradel a stejnou abecedu. Takovým výpočetním modelům se říká *neuniformní.*

Obvykle budeme chtít, aby existoval algoritmus (klasický, neparalelní), který pro danou velikost vstupu sestrojí příslušnou síť. Tento algoritmus by měl běžet v polynomiálním čase – kdybychom dovolili i pomalejší algoritmy, mohli bychom během konstrukce provádět nějaký náročný předvýpočet a jeho výsledek zabudovat do struktury sítě. To je málokdy žádoucí.

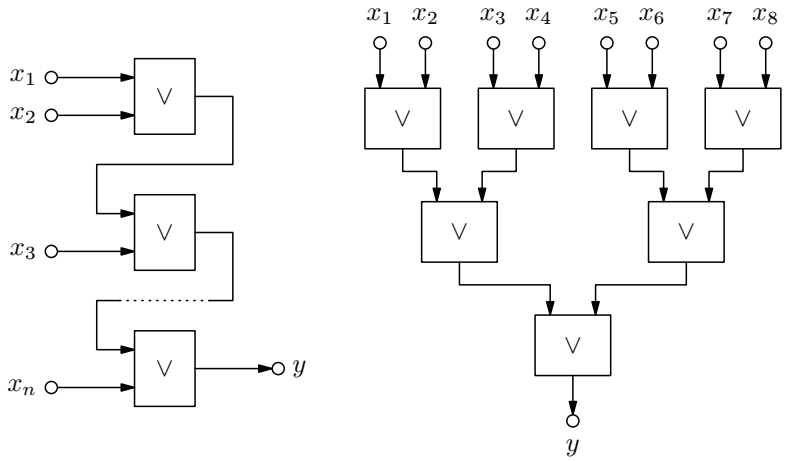
Hledá se jednička

Abychom si nový výpočetní model osahali, zkusme nejprve sestrojit booleovský obvod, který zjistí, zda se mezi jeho n vstupy vyskytuje alespoň jedna jednička. To znamená, že počítá n-vstupovou funkci OR.

První řešení (obrázek 15.2 vlevo): Spočítáme OR prvních dvou vstupů, pak OR výsledku s třetím vstupem, pak se čtvrtým, a tak dále. Každé hradlo závisí na výsledcích všech předchozích, takže výpočet běží striktně sekvenčně. Časová i prostorová složitost činí $\Theta(n)$.

Druhé řešení (obrázek 15.2 vpravo): Hradla budeme spojovat do dvojic, výsledky těchto dvojic opět do dvojic, a tak dále. Síť se tentokrát skládá z $\Theta(\log n)$ vrstev, které celkem obsahují $n/2 + n/4 + \ldots + 1 = \Theta(n)$ hradel.

Logaritmická časová složitost je pro paralelní algoritmy typická a budeme se jí snažit dosáhnout i u dalších problémů.



Obrázek 15.2: Dvě hradlové sítě pro n-bitový OR

Cvičení

1. Jak vypadá všech 16 booleovských funkcí dvou proměnných?

2. Dokažte, že každou booleovskou funkci dvou proměnných lze vyjádřit pomocí hradel AND, OR a NOT. Proto lze každý booleovský obvod s nejvýše dvouvstupovými hradly upravit tak, aby používal pouze tyto tři typy hradel. Jeho hloubka přitom vzroste pouze konstanta-krát.

3. Pokračujme v předchozím cvičení: dokažte, že stačí jediný typ hradla, a to NAND (negovaný AND). Podobně by stačil NOR (negovaný OR). Existuje nějaká další funkce s touto vlastností?

4. Sestavte hradlovou síť ze čtyř hradel NAND (negovaný AND), která počítá XOR dvou bitů.

5. Dokažte, že n-bitový OR nelze spočítat v menší než logaritmické hloubce.

6. Sestrojte hradlovou síť pro majoritu ze 4 vstupů.

7. Ukažte, že libovolnou booleovskou funkci s k vstupy lze spočítat booleovským obvodem hloubky $\mathcal{O}(k)$ s $\mathcal{O}(2^k)$ hradly. To speciálně znamená, že pro pevné k lze booleovské obvody s nejvýše k-vstupovými hradly překládat na obvody s 2-vstupovými hradly. Hloubka přitom vzroste pouze konstanta-krát.

8.* Exponenciální velikost obvodu z minulého cvičení je nepříjemná, ale bohužel nevyhnutelná: Dokažte, že pro žádné k neplatí, že všechny n-vstupové booleovské funkce lze spočítat obvody s $\mathcal{O}(n^k)$ hradly.

9. Ukažte, jak hradlovou síť s libovolnou abecedou přeložit na ekvivalentní booleovský obvod s nejvýše konstantním zpomalením. Abecedu zakódujte binárně, hradla simulujte booleovskými obvody.

10. Definujeme *výhybku* – to je analogie operátoru `?:` v jazyce C, tedy ternární booleovské hradlo se vstupy x_0, x_1 a p, jehož výsledkem je x_p. Ukažte, že libovolnou k-vstupovou booleovskou funkci lze spočítat obvodem složeným pouze z výhybek a konstant. Srovnejte s cvičením 7. Jak by se naopak skládala výhybka z binárních hradel?

11. Dokažte, že každou booleovskou formuli lze přeložit na booleovský obvod. Velikost obvodu i jeho hloubka přitom budou lineární v délce formule.

12.** Ukažte, že obvody z minulého cvičení si vystačí s logaritmickou hloubkou v délce formule.

13. Místo omezení arity hradel bychom mohli omezit typy funkcí, řekněme na AND, OR a NOT, a požadovat polynomiální počet hradel. Tím by také vznikl realistický model, byť s trochu jinými vlastnostmi. Dokažte, že síť tohoto druhu s n vstupy lze přeložit na síť s omezenou aritou hradel, která bude pouze $\mathcal{O}(\log n)$-krát hlubší. K čemu bylo nutné omezení počtu hradel?

15.2 Sčítání a násobení binárních čísel

Nalezli jsme rychlý paralelní algoritmus pro n-bitový OR. Zajímavější úlohou, jejíž paralelizace už nebude tak triviální, bude sčítání dvojkových čísel. Mějme dvě čísla x a y zapsané ve dvojkové soustavě. Jejich číslice označme $x_{n-1} \ldots x_0$ a $y_{n-1} \ldots y_0$, přičemž i-tý řád má váhu 2^i. Chceme spočítat dvojkový zápis $z_n \ldots z_0$ čísla $z = x + y$.

Školní algoritmus

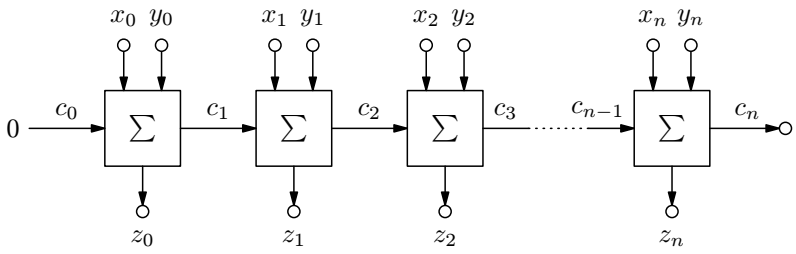
Ihned se nabízí použít starý dobrý „školní algoritmus sčítání pod sebou“. Ten funguje ve dvojkové soustavě stejně dobře jako v desítkové. Sčítáme čísla zprava doleva, vždy sečteme x_i s y_i a přičteme přenos z nižšího řádu. Tím dostaneme jednu číslici výsledku a přenos do vyššího řádu. Formálně bychom to mohli zapsat třeba takto:

$$z_i = x_i \oplus y_i \oplus c_i,$$

kde z_i je i-tá číslice součtu, $\oplus$ značí operaci XOR (součet modulo 2) a c_i je *přenos* z $(i-1)$-ního řádu do i-tého. Přenos do vyššího řádu nastane tehdy, pokud se nám potkají dvě jedničky pod sebou, nebo když se vyskytne alespoň jedna jednička a k tomu přenos z nižšího řádu. Čili tehdy, jsou-li mezi třemi xorovanými číslicemi alespoň dvě jedničky – k tomu se nám hodí již známý obvod pro majoritu:

$$\begin{aligned} c_0 &= 0, \\ c_{i+1} &= (x_i \wedge y_i) \vee (x_i \wedge c_i) \vee (y_i \wedge c_i). \end{aligned}$$

O tomto předpisu snadno dokážeme, že funguje (zkuste to), nicméně pokud podle něj postavíme hradlovou síť, bude poměrně pomalá. Můžeme si ji představit tak, že je složena z nějakých podsítí („krabiček“), které budou mít na vstupu x_i, y_i a c_i a jejich výstupem bude z_i a c_{i+1}. To je hezky vidět na obrázku 15.3.



Obrázek 15.3: Sčítání školním algoritmem

Každá krabička má sama o sobě konstantní hloubku, ovšem k výpočtu potřebuje přenos vypočítaný předcházející krabičkou. Jednotlivé krabičky proto musí ležet v různých vrstvách sítě. Časová i prostorová složitost sítě jsou tedy lineární, stejně jako sčítáme-li po bitech na RAMu.

Bloky a jejich chování

To, co nás při sčítání brzdí, je evidentně čekání na přenosy z nižších řádů. Jakmile je zjistíme, máme vyhráno – součet už získáme jednoduchým xorováním, které zvládneme paralelně v čase $\Theta(1)$. Uvažujme tedy nad způsobem, jak přenosy spočítat paralelně.

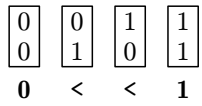
Podívejme se na libovolný *blok* výpočtu školního algoritmu. Tak budeme říkat části sítě, která počítá součet bitů $x_j \ldots x_i$ a $y_j \ldots y_i$ v nějakém intervalu indexů $[i, j]$. Přenos c_{j+1} vystupující z tohoto bloku závisí kromě hodnot sčítanců už pouze na přenosu c_i, který do bloku vstupuje.

Pro konkrétní sčítance se tedy můžeme na blok dívat jako na nějakou funkci, která dostane jednobitový vstup (přenos zespoda) a vydá jednobitový výstup (přenos nahoru). To je milé, neboť takové funkce existují pouze čtyři:

$f(x) = 0$	konstantní **0**, blok *pohlcuje* přenos
$f(x) = 1$	konstantní **1**, blok *vytváří* přenos
$f(x) = x$	identita (značíme <), blok *kopíruje* přenos
$f(x) = \neg x$	negace; ukážeme, že u žádného bloku nenastane

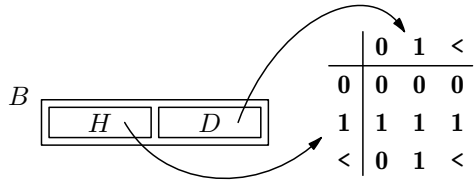
Této funkci budeme říkat *chování bloku.*

Jednobitové bloky se chovají velice jednoduše:



Blok prvního druhu vždy předává nulový přenos, ať už do něj vstoupí jakýkoliv – přenos tedy pohlcuje. Poslední blok naopak sám o sobě přenos vytváří, ať dostane cokoliv. Prostřední dva bloky se chovají tak, že samy o sobě žádný přenos nevytvoří, ale pokud do nich nějaký přijde, tak také odejde.

Větší bloky můžeme rozdělit na části a podle chování částí určit, jak se chová celek. Mějme blok B složený ze dvou menších podbloků H (horní část) a D (dolní). Chování celku závisí na chování částí takto:



Pokud vyšší blok přenos pohlcuje, pak ať se nižší blok chová jakkoli, složení obou bloků musí vždy pohlcovat. V prvním řádku tabulky jsou tudíž nuly. Analogicky pokud vyšší blok generuje přenos, tak ten nižší na tom nic nezmění. V druhém řádku tabulky jsou tedy samé jedničky. Zajímavější případ nastává, pokud vyšší blok kopíruje – tehdy záleží čistě na chování nižšího bloku.

Všimněme si, že skládání chování bloků je vlastně úplně obyčejné skládání funkcí. Nyní bychom mohli prohlásit, že budeme počítat nad tříprvkovou abecedou a že celou tabulku dokážeme spočítat jedním jediným hradlem. Pojďme si přeci jen rozmyslet, jak bychom takovou operaci popsali čistě binárně.

Tři stavy můžeme zakódovat pomocí dvou bitů, říkejme jim třeba p a q. Dvojice (p, q) přitom může nabývat hned čtyř možných hodnot, my dvěma z nich přiřadíme stejný význam:

$$(1, *) = \mathbf{<} \qquad (0, 0) = \mathbf{0} \qquad (0, 1) = \mathbf{1}.$$

Kdykoliv $p = 1$, blok kopíruje přenos. Naopak $p = 0$ odpovídá tomu, že blok posílá do vyššího řádu konstantní přenos, a q pak určuje, jaký. Kombinování bloků (skládání funkcí) pak můžeme popsat následovně:

$$\begin{aligned} p_B &= p_H \wedge p_D, \\ q_B &= (\neg p_H \wedge q_H) \vee (p_H \wedge q_D). \end{aligned}$$

Průchod přenosu blokem (dosazení do funkce) bude vypadat takto:

$$c_{i+1} = (p \wedge c_i) \vee (\neg p \wedge q).$$

Rozmyslete si, že tyto formule odpovídají výše uvedené tabulce. (Mimochodem, totéž by se mnohem přímočařeji formulovalo pomocí výhybek z cvičení 15.1.10.)

Paralelní sčítání

Od popisu chování bloků je už jenom krůček k paralelnímu předpovídání přenosů, a tím i k paralelní sčítačce. Bez újmy na obecnosti budeme předpokládat, že počet bitů vstupních čísel n je mocnina dvojky; jinak vstup doplníme zleva nulami.

Algoritmus bude rozdělen na dvě části:

(1) Spočítáme chování všech *kanonických bloků* – tak budeme říkat blokům, jejichž velikost je mocnina dvojky a pozice je dělitelná velikostí (bloky téže velikosti se tedy nepřekrývají). Nejprve v konstantním čase stanovíme chování bloků velikosti 1, ty pak spojíme do dvojic, dvojice zase do dvojic atd., obecně v i-tém kroku spočteme chování všech kanonických bloků velikosti 2^i.

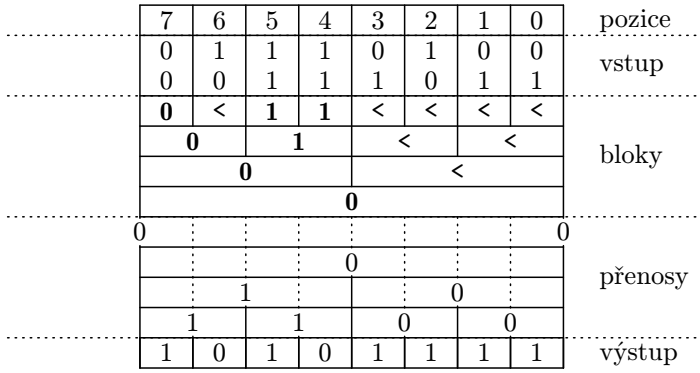
(2) Dopočítáme přenosy, a to tak, aby v i-tém kroku byly známy přenosy do řádů dělitelných $2^{\log n - i}$. V nultém kroku známe pouze $c_0 = 0$ a c_n, který spočítáme z c_0 pomocí chování bloku $[0, n]$. V prvním kroku pomocí bloku $[0, n/2]$ dopočítáme $c_{n/2}$, v druhém pomocí $[0, n/4]$ spočítáme $c_{n/4}$ a pomocí $[n/2, 3/4 \cdot n]$ dostaneme $c_{3/4 \cdot n}$, atd. Obecně v i-tém kroku používáme chování bloků velikosti $2^{\log n - i}$. Každý krok přitom zabere konstantní čas.

Celkově bude sčítací síť vypadat takto (viz obr. 15.4):

- $\Theta(1)$ hladin výpočtu chování bloků velikosti 1,

- $\Theta(\log n)$ hladin počítajících chování všech kanonických bloků,
- $\Theta(\log n)$ hladin dopočítávajících přenosy „zahušťováním“,
- $\Theta(1)$ hladin na samotné sečtení: $z_i = x_i \oplus y_i \oplus c_i$ pro všechna i.

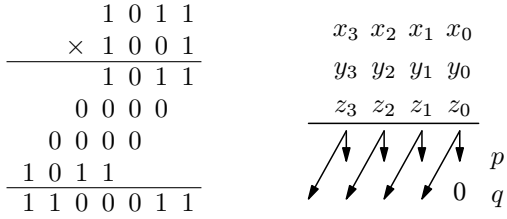
Algoritmus tedy pracuje v čase $\Theta(\log n)$. Využívá k tomu lineárně mnoho hradel: při výpočtu chování bloků na jednotlivých hladinách počet hradel exponenciálně klesá od n k 1, během zahušťování přenosů naopak exponenciálně stoupá od 1 k n. Obě geometrické řady se sečtou na $\Theta(n)$.



Obrázek 15.4: Průběh paralelního sčítání pro n = 8

Paralelní násobení

Ještě si rozmyslíme, jak rychle je možné čísla násobit. Opět se inspirujeme školním algoritmem: pokud násobíme dvě n-ciferná čísla x a y, uvážíme všech n posunutí čísla x, každé z nich vynásobíme příslušnou číslicí v y a výsledky posčítáme.



Obrázek 15.5: Školní násobení a kompresor

Ve dvojkové soustavě je to ještě jednodušší: násobení jednou číslicí je prostý AND. Paralelně tedy vytvoříme všechna posunutí a spočítáme všechny ANDy. To vše stihneme za 1 takt výpočtu.

Zbývá sečíst n čísel, z nichž každé má $\Theta(n)$ bitů. Mohli bychom opět sáhnout po osvědčeném triku: sčítat dvojice čísel, pak dvojice těchto součtů, atd. Taková síť by měla tvar binárního stromu hloubky $\log n$, jehož každý vrchol by obsahoval jednu sčítačku, a na tu, jak víme, postačí $\Theta(\log n)$ hladin. Celý výpočet by tedy běžel v čase $\Theta(\log^2 n)$.

Jde to ale rychleji, použijeme-li jednoduchý, téměř kouzelnický trik. Sestrojíme *kompresor* – to bude obvod konstantní hloubky, jenž na vstupu dostane tři čísla a vypočte z nich dvě čísla mající stejný součet jako zadaná trojice.

K čemu je to dobré? Máme-li sečíst n čísel, v konstantním čase dokážeme tento úkol převést na sečtení $2/3 \cdot n$ čísel (vhodně zaokrouhleno), to pak opět v konstantním čase na sečtení $(2/3)^2 \cdot n$ čísel atd., až nám po $\log_{3/2} n = \Theta(\log n)$ krocích zbudou dvě čísla a ta sečteme klasickou sčítačkou. Zbývá vymyslet kompresor.

Konstrukce kompresoru: Označme vstupy kompresoru x, y a z a výstupy p a q. Pro každý řád i spočteme součet $x_i + y_i + z_i$. To je nějaké dvoubitové číslo, takže můžeme jeho nižší bit prohlásit za p_i a vyšší za q_{i+1}.

Jinými slovy všechna tři čísla jsme normálně sečetli, ale místo abychom přenosy posílali do vyššího řádu, vytvořili jsme z nich další číslo, které má být k výsledku časem přičteno. To je vidět na obrázku 15.5.

Naše síť pro paralelní násobení nyní pracuje v čase $\Theta(\log n)$ – nejdříve v konstantním čase vytvoříme mezivýsledky, pak použijeme $\Theta(\log n)$ hladin kompresorů konstantní hloubky a nakonec jednu sčítačku hloubky $\Theta(\log n)$.

Jistou vadou na kráse ovšem je, že spotřebujeme $\Theta(n^2)$ hradel. Proto se v praxi používají spíš násobicí sítě odvozené od rychlé Fourierovy transformace, s níž se potkáme v kapitole 17.

Cvičení

1. Modifikujte sčítací síť, aby odčítala.

2. Sestrojte hradlovou síť hloubky $\mathcal{O}(\log n)$, která porovná dvě n-bitová čísla x a y a vydá jedničku, pokud $x < y$.

3. Ukažte, jak v logaritmické hloubce otestovat, zda je n-bitové dvojkové číslo dělitelné jedenácti.

4.** Pro ctitele teorie automatů: Dokažte, že každý regulární jazyk lze rozpoznávat hradlovou sítí logaritmické hloubky. Ukažte, jak pomocí toho vyřešit všechna předchozí cvičení.

5.* Sestrojte hradlovou síť logaritmické hloubky, která dostane matici sousednosti neorientovaného grafu a rozhodne, zda je graf souvislý.

6. Sestrojte hradlovou síť, která pro zadané dvojkové číslo $x_{n-1} \ldots x_0$ spočítá dolní celou část z jeho dvojkového logaritmu, čili nejvyšší i takové, že $x_i = 1$.

15.3 Třídicí sítě

Ještě zkusíme paralelizovat jeden klasický problém, totiž třídění. Budeme k tomu používat *komparátorovou síť* – to je hradlová síť složená z *komparátorů.*

Jeden komparátor umí porovnat dvě hodnoty a rozhodnout, která z nich je větší a která menší. Nevrací však booleovský výsledek jako běžné hradlo, ale má dva výstupy: na jednom z nich vrací menší ze vstupních hodnot a na druhém tu větší.

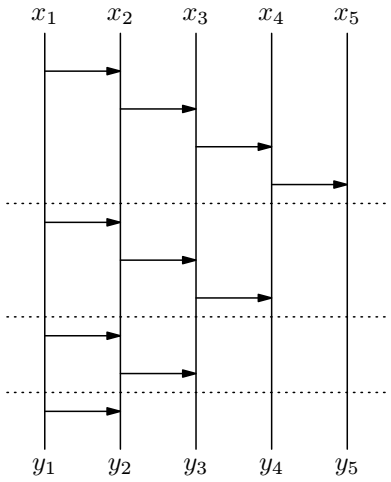
V našem formalismu hradlových sítí bychom mohli komparátor reprezentovat dvojicí hradel: jedno z nich by počítalo minimum, druhé maximum. Hodnoty, které třídíme, bychom považovali za prvky abecedy. Komparátorovou síť můžeme také snadno přeložit na booleovský obvod, viz cvičení 4.

Ještě se dohodneme, že výstupy komparátorů se nikdy nebudou větvit. Každý výstup přivedeme na vstup dalšího komparátoru, nebo na výstup sítě. Větvení by nám ostatně k ničemu nebylo, protože na výstupu potřebujeme vydat stejný počet hodnot, jako byl na vstupu. Nemáme přitom žádné hradlo, kterým bychom mohli hodnoty slučovat, a definice hradlové sítě nám nedovoluje výstup hradla „zahodit“.

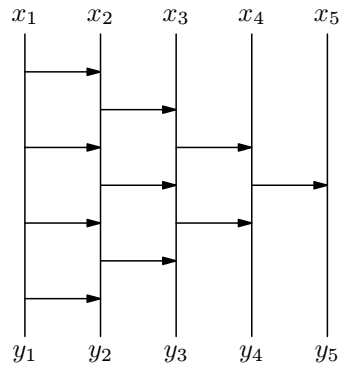
Důsledkem je, že výstup každé vrstvy, a tedy i celé sítě, je nějaká permutace prvků ze vstupu.

Jako rozcvičku zkusíme do řeči komparátorových sítí přeložit *bublinkové třídění.* Z něj získáme obvod na obrázku 15.6. Komparátory kreslíme jako šipky: shora do šipky vedou vstupy, zdola z ní vycházejí výstupy, větší výstup leží ve směru šipky.

Toto nakreslení ovšem poněkud klame – pokud síť necháme počítat, mnohá porovnání budou probíhat paralelně. Skutečný průběh výpočtu znázorňuje obrázek 15.7, na němž jsme všechny operace prováděné současně znázornili vedle sebe. Ihned vidíme, že paralelní bublinkové třídění pracuje v čase $\Theta(n)$ a potřebuje kvadratický počet komparátorů.



Obrázek 15.6: Bublinkové třídění



Obrázek 15.7: Skutečný průběh výpočtu

Bitonické třídění

Nyní vybudujeme rychlejší třídicí algoritmus. Půjdeme na něj menší oklikou. Nejdříve vymyslíme síť, která bude umět třídit jenom něco – totiž bitonické posloupnosti. Z ní pak odvodíme obecné třídění. Bez újmy na obecnosti přitom budeme předpokládat, že každé dva prvky na vstupu jsou navzájem různé a že velikost vstupu je mocnina dvojky.

Definice: Posloupnost $x_0, \ldots, x_{n-1}$ je *čistě bitonická,* pokud ji můžeme rozdělit na nějaké pozici k na rostoucí posloupnost $x_0, \ldots, x_k$ a klesající posloupnost $x_k, \ldots, x_{n-1}$. Jak rostoucí, tak klesající část mohou být prázdné či jednoprvkové.

Definice: Posloupnost $x_0, \ldots, x_{n-1}$ je *bitonická*, jestliže ji lze získat rotací (cyklickým posunutím) nějaké čistě bitonické posloupnosti. Tedy pokud existuje číslo j takové, že posloupnost $x_j, x_{(j+1) \bmod n}, \ldots, x_{(j+n-1) \bmod n}$ je čistě bitonická.

Definice: *Separátor řádu n* je komparátorová síť S_n se vstupy $x_0, \ldots, x_{n-1}$ a výstupy $y_0, \ldots, y_{n-1}$. Dostane-li na vstupu bitonickou posloupnost, vydá na výstup její permutaci s následujícími vlastnostmi:

- $y_0, \ldots, y_{n/2-1}$ a $y_{n/2}, \ldots, y_{n-1}$ jsou bitonické posloupnosti;
- $y_i < y_j$, kdykoliv $0 \leq i < n/2 \leq j < n$.

Jinak řečeno, separátor rozdělí bitonickou posloupnost na dvě poloviční a navíc jsou všechny prvky v první polovině menší než všechny v té druhé.

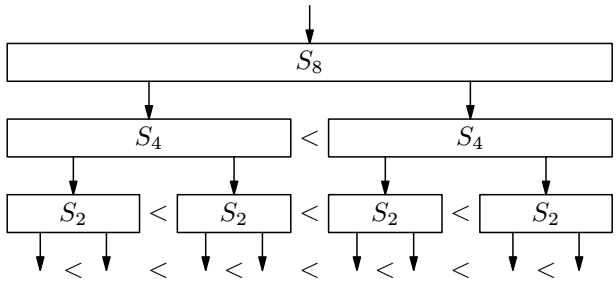
Lemma: Pro každé sudé n existuje separátor S_n konstantní hloubky, složený z $\Theta(n)$ komparátorů.

Důkaz tohoto lemmatu si necháme na konec. Nejprve předvedeme, k čemu jsou separátory dobré.

Definice: *Bitonická třídička řádu n* je komparátorová síť B_n s n vstupy a n výstupy. Dostane-li na vstupu bitonickou posloupnost, vydá ji setříděnou.

Lemma: Pro libovolné $n = 2^k$ existuje bitonická třídička B_n hloubky $\Theta(\log n)$ s $\Theta(n \log n)$ komparátory.

Důkaz: Konstrukce bitonické třídičky je snadná: nejprve separátorem S_n zadanou bitonickou posloupnost rozdělíme na dvě bitonické posloupnosti délky $n/2$, každou z nich pak separátorem $S_{n/2}$ na dvě části délky $n/4$, atd., až získáme jednoprvkové posloupnosti ve správném pořadí. Celkem použijeme $\log n$ hladin, každá hladina má konstantní hloubku a leží na ní $n/2$ komparátorů. □



Obrázek 15.8: Bitonická třídička B_8

Bitonické třídičky nám nyní pomohou ke konstrukci třídičky pro obecné posloupnosti. Ta bude založena na třídění sléváním – nejprve se tedy musíme naučit slít dvě rostoucí posloupnosti do jedné.

Definice: *Slévačka řádu n* je komparátorová síť M_n s $2 \times n$ vstupy a $2n$ výstupy. Dostane-li dvě setříděné posloupnosti délky n, vydá setříděnou posloupnost vzniklou jejich slitím.

Lemma: Pro $n = 2^k$ existuje slévačka M_n hloubky $\Theta(\log n)$ s $\Theta(n \log n)$ komparátory.

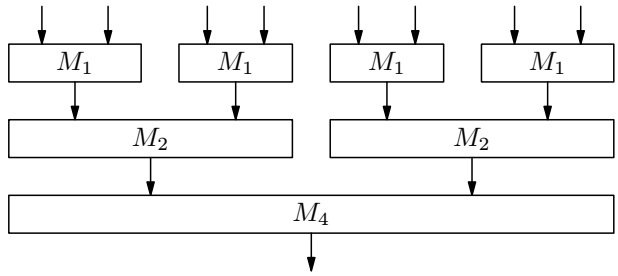
Důkaz: Stačí jednu vstupní posloupnost obrátit a „přilepit“ za tu druhou. Tím vznikne bitonická posloupnost, již setřídíme bitonickou třídičkou B_{2n}. □

Definice: *Třídicí síť řádu n* je komparátorová síť T_n s n vstupy a n výstupy, která pro každý vstup vydá jeho setříděnou permutaci.

Věta: Pro $n = 2^k$ existuje třídicí síť T_n hloubky $\Theta(\log^2 n)$ složená z $\Theta(n \log^2 n)$ komparátorů.

Důkaz: Síť bude třídit sléváním, podobně jako algoritmus Mergesort z oddílu 3.2. Vstup rozdělíme na n jednoprvkových posloupností. Ty jsou jistě setříděné, takže je slévačkami M_1 můžeme slít do dvouprvkových setříděných posloupností. Na ty pak aplikujeme slévačky M_2, M_4, …, $M_{n/2}$, až všechny části slijeme do jedné, setříděné.

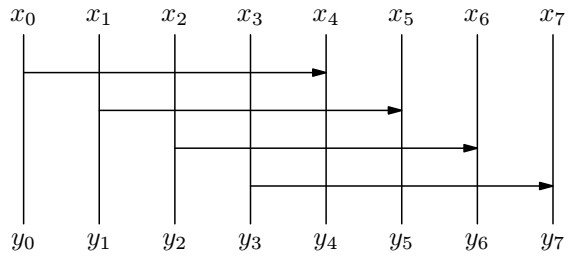
Celkem provedeme $\log n$ kroků slévání, i-tý z nich obsahuje slévačky $M_{2^{i-1}}$ a ty, jak už víme, mají hloubku $\Theta(i)$. Celkový počet vrstev tedy činí $\Theta(1+2+3+\ldots+\log n) = \Theta(\log^2 n)$. Každý krok přitom potřebuje $\Theta(n \log n)$ komparátorů, což dává celkem $\Theta(n \log^2 n)$ komparátorů. □



Obrázek 15.9: Třídička T_8

Konstrukce separátoru

Zbývá dokázat, že existují slíbené separátory konstantní hloubky. Vypadají překvapivě jednoduše: pro $i = 0, \ldots, n/2 - 1$ zapojíme komparátor se vstupy x_i, $x_{i+n/2}$, jehož minimum přivedeme na y_i a maximum na $y_{i+n/2}$.



Obrázek 15.10: Separátor S_8

Proč separátor separuje? Nejprve předpokládejme, že vstupem je čistě bitonická posloupnost. Označme m polohu maxima této posloupnosti; maximum bez újmy na obecnosti leží v první polovině (jinak celý důkaz provedeme „zrcadlově“). Označme dále k nejmenší index, pro který komparátor zapojený mezi x_k a $x_{n/2+k}$ hodnoty prohodí, tedy $k = \min\{i \mid x_i > x_{n/2+i}\}$.

Jelikož maximum je jedinečné, musí platit $x_m > x_{n/2+m}$, takže k existuje a navíc platí $0 \leq k \leq m < n/2$. Situace tedy odpovídá obrázku 15.11.

Nyní nahlédneme, že pro $i = k, \ldots, n/2 - 1$ už komparátory vždy prohazují: Platí $x_i > x_{n/2+k}$ (pro $i \geq m$ je to vidět přímo, pro $i < m$ je $x_i \geq x_k > x_{n/2+k}$). Ovšem $x_{n/2+k} \geq x_{n/2+i}$, protože zbytek posloupnosti je klesající.

Separátor se tedy chová velice přímočaře: levá polovina výstupu vznikne slepením rostoucího úseku $x_0, \ldots, x_{k-1}$ s klesajícím úsekem $x_{n/2+k}, \ldots, x_{n-1}$; pravou polovinu tvoří spojení klesajícího úseku $x_{n/2}, \ldots, x_{n/2+k-1}$, rostoucího úseku $x_k, \ldots, x_{m-1}$ a klesajícího úseku $x_m, \ldots, x_{n/2-1}$.

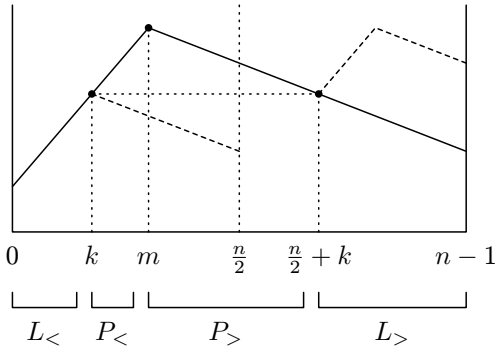
Snadno ověříme, že obě poloviny jsou bitonické: ta první je dokonce čistě bitonická, druhou lze na čistě bitonickou zrotovat díky tomu, že $x_{n/2-1} > x_{n/2}$.

Zbývá dokázat, že levá polovina je menší než pravá. Zdá se to být zřejmé z obrázku: křivku rozkrojíme vodorovnou tečkovanou linkou a části přeskládáme. Jenže nesmíme zapomínat, že x_k a $x_{n/2+k}$ jsou různé prvky, takže tečkovaná linka není ve skutečnosti vodorovná.

Proveďme podobnou úvahu precizně: Levou polovinu rozdělíme na rostoucí část $L_< = x_0, \ldots, x_{k-1}$ a klesající část $L_> = x_{n/2+k}, \ldots, x_{n-1}$; podobně rozdělíme pravou na $P_< = x_k, \ldots, x_{m-1}$ a $P_> = x_m, \ldots, x_{n/2+k-1}$ (ve výstupu prvky leží v jiném pořadí, ale to teď nevadí). Tyto části nyní porovnáme:

- $L_< < P_<$: obě části původně tvořily jeden společný rostoucí úsek;
- $L_< < P_>$: $\max L_< = x_{k-1} < x_{n/2+k-1} = \min P_>$ (kdyby neplatila prostřední nerovnost, mohli bychom snížit k);
- $L_> < P_<$: $\max L_> = x_{n/2+k} < x_k = \min P_<$;
- $L_> < P_>$: obě části původně tvořily jeden společný klesající úsek.

Doplňme, co se stane, pokud vstup není čistě bitonický. Zde využijeme toho, že separátor je symetrický, tudíž zrotujeme-li jeho vstup o p pozic, dostaneme o p pozic zrotované i obě poloviny výstupu. Podle definice ovšem pro každou bitonickou posloupnost existuje její rotace, která je čistě bitonická, a pro níž, jak už víme, separátor funguje. Takže pro nečistou bitonickou posloupnost musí vydat výsledek pouze zrotovaný, což na jeho správnosti nic nemění. □



Obrázek 15.11: Ilustrace činnosti separátoru

Shrnutí

Nalezli jsme paralelní třídicí algoritmus o časové složitosti $\Theta(\log^2 n)$, který se skládá z $\Theta(n \log^2 n)$ komparátorů. Dodejme, že jsou známé i třídicí sítě hloubky $\Theta(\log n)$, ale jejich konstrukce je mnohem komplikovanější a dává obrovské multiplikativní konstanty, jež brání praktickému použití.

Z dolního odhadu složitosti třídění v oddílu 3.3 navíc plyne, že logaritmický počet hladin je nejnižší možný. Máme-li totiž libovolnou třídicí síť hloubky h, můžeme ji simulovat po hladinách a získat tak sekvenční třídicí algoritmus. Jelikož na každé hladině může ležet nejvýše $n/2$ komparátorů, náš algoritmus provede maximálně $hn/2$ porovnání. Už jsme nicméně dokázali, že pro každý třídicí algoritmus existují vstupy, na kterých porovná $\Omega(n \log n)$-krát. Proto $h = \Omega(\log n)$.

Cvičení

1. Jak by vypadala komparátorová síť pro třídění vkládáním? Jak se bude její průběh výpočtu lišit od paralelního bublinkového třídění?

2. Navrhněte komparátorovou síť pro hledání maxima: dostane-li n prvků, vydá takovou permutaci, v níž bude poslední hodnota největší.

3. Navrhněte komparátorovou síť pro zatřídění prvku do setříděné posloupnosti: dostane $(n-1)$-prvkovou setříděnou posloupnost a jeden prvek navíc, vydá setříděnou permutaci.

4. Ukažte, jak komparátorovou síť přeložit na booleovský obvod. Každý prvek abecedy Σ reprezentujte číslem o $b = \lceil \log_2 |\Sigma| \rceil$ bitech a pomocí cvičení 15.2.2 sestrojte komparátory o $\mathcal{O}(\log b)$ hladinách.

5. Upravte algoritmus bitonického třídění, aby fungoval i pro vstupy, jejichž délka není mocninou dvojky.

6. Dokažte *nula-jedničkový princip:* pro ověření, že komparátorová síť třídí všechny vstupy, ji postačí otestovat na všech posloupnostech nul a jedniček.

7.* *Batcherovo třídění:* Stejné složitosti paralelního třídění lze také dosáhnout následujícím rekurzivním algoritmem pro slévání setříděných posloupností:

Procedura BMerge
Vstup: Setříděné posloupnosti $(x_0, \ldots, x_{n-1})$ a $(y_0, \ldots, y_{n-1})$
1. Je-li $n \leq 1$, vyřešíme triviálně.
2. $(a_0, \ldots, a_{n-1}) \leftarrow \text{BMerge}((x_0, x_2, \ldots, x_{n-2}), (y_0, y_2, \ldots, y_{n-2}))$
3. $(b_0, \ldots, b_{n-1}) \leftarrow \text{BMerge}((x_1, x_3, \ldots, x_{n-1}), (y_1, y_3, \ldots, y_{n-1}))$

Výstup: $(a_0, \min(a_1, b_0), \max(a_1, b_0), \min(a_2, b_1), \max(a_2, b_1), \ldots, b_{n-1})$

Pomocí předchozího cvičení dokažte, že tato procedura funguje. Zapište tento algoritmus ve formě třídicí sítě.

16 Geometrické algoritmy

16 Geometrické algoritmy

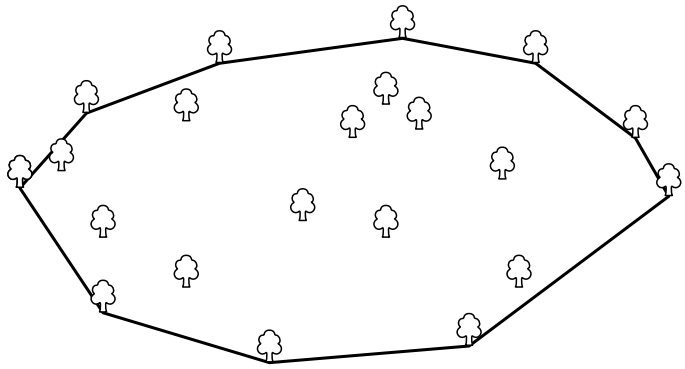
Mnoho praktických problémů má geometrickou povahu: můžeme chtít oplotit jabloňový sad nejkratším možným plotem, nalézt k dané adrese nejbližší poštovní úřadovnu, nebo třeba naplánovat trasu robota trojrozměrnou budovou.

V této kapitole ukážeme několik základních způsobů, jak geometrické algoritmy navrhovat. Také uvidíme, jak je pak volbou vhodné datové struktury výrazně zrychlit. Soustředíme se přitom na problémy v rovině: ty jednorozměrné bývají triviální, vícerozměrné naopak mnohem náročnější.

16.1 Konvexní obal

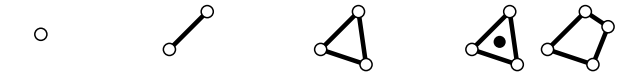
Byl jest jednou jeden jabloňový sad. Každý podzim v něm dozrávala kulaťoučká červeňoučká jablíčka, tak dobrá, že je za noci chodili otrhávat všichni tuláci z okolí. Aby alespoň část úrody vydržela do sklizně, nabízí se sad oplotit. Chceme postavit plot, který obklopí všechny jabloně a spotřebujeme na něj co nejméně pletiva.

Méně poeticky řečeno: Dostali jsme nějakou množinu n bodů v euklidovské rovině a chceme nalézt co nejkratší uzavřenou křivku, uvnitř níž leží všechny body. Geometrická intuice nám napovídá, že hledaná křivka bude uzavírat konvexní mnohoúhelník, v jehož vrcholech budou některé ze zadaných bodů. Ostatní body budou ležet uvnitř mnohoúhelníka, případně na jeho hranách. Tomu se obvykle říká *konvexní obal* zadaných bodů. (Pokud se nechcete odvolávat na intuici, trochu formálnější pohled najdete ve cvičeních 4 a 5.)



Obrázek 16.1: Ohrazený jabloňový sad

Pro malé počty bodů bude konvexní obal vypadat následovně:



$n = 1$ $n = 2$ $n = 3$ $n = 4$

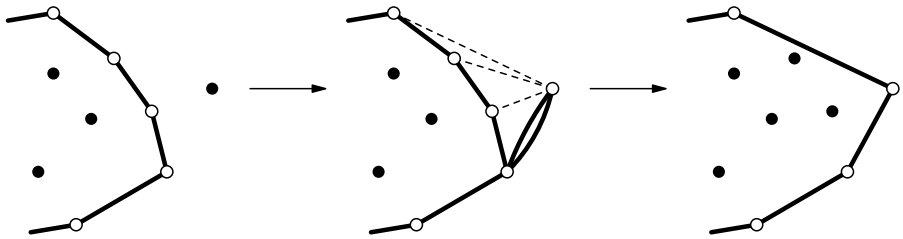
Naším úkolem tedy bude najít konvexní obal a vypsat na výstup jeho vrcholy tak, jak leží na hranici (buď po směru hodinových ručiček, nebo proti němu). Pro jednoduchost budeme konvexní obal říkat přímo tomuto seznamu vrcholů.

Prozatím budeme předpokládat, že všechny body mají různé x-ové souřadnice. Existuje tedy jednoznačně určený nejlevější a nejpravější bod a ty musí oba ležet na konvexním obalu. (Obecně se hodí geometrické problémy řešit nejdříve pro body, které jsou v nějakém vhodném smyslu *v obecné poloze,* a teprve pak se starat o speciální případy.)

Použijeme princip, kterému se obvykle říká *zametání roviny.* Budeme procházet rovinu zleva doprava („zametat ji přímkou“) a udržovat si konvexní obal těch bodů, které jsme už prošli.

Na počátku máme konvexní obal jednobodové množiny, což je samotný bod. Nechť tedy už známe konvexní obal prvních $k - 1$ bodů a chceme přidat k-tý bod. Ten určitě na novém konvexním obalu bude ležet (je nejpravější), ale jeho přidání k minulému obalu může způsobit, že hranice přestane být konvexní. To lze snadno napravit – stačí z hranice odebírat body po směru a proti směru hodinových ručiček, než opět bude konvexní.

Například na následujícím obrázku nemusíme po směru hodinových ručiček odebrat ani jeden bod, obal je v pořádku. Naopak proti směru ručiček musíme odstranit dokonce tři body.



Obrázek 16.2: Přidání bodu do konvexního obalu

Podle tohoto principu už snadno vytvoříme algoritmus. Aby se lépe popisoval, rozdělíme konvexní obal na *horní obálku* a *dolní obálku* – to jsou části, které vedou od nejlevějšího bodu k nejpravějšímu „horem“ a „spodem“.

Obě obálky jsou lomené čáry, navíc horní obálka pořád zatáčí doprava a dolní naopak doleva. Pro udržování bodů v obálkách stačí dva zásobníky. V k-tém kroku algoritmu přidáme k-tý bod zvlášť do horní i dolní obálky. Přidáním k-tého bodu se však může porušit směr, ve kterém obálka zatáčí. Proto budeme nejprve body z obálky odebírat a k-tý bod přidáme až ve chvíli, kdy jeho přidání směr zatáčení neporuší.

Algoritmus KonvexníObal

1. Setřídíme body podle x-ové souřadnice, označíme je $b_1, \ldots, b_n$.
2. Vložíme do horní a dolní obálky bod b_1: $H \leftarrow D \leftarrow (b_1)$.
3. Pro každý další bod $b = b_2, \ldots, b_n$:
4. Přepočítáme horní obálku:
5. Dokud $|H| \geq 2$, $H = (\ldots, h_{k-1}, h_k)$ a úhel $h_{k-1}h_kb$ je orientovaný doleva:
6. Odebereme poslední bod h_k z obálky H.
7. Přidáme bod b na konec obálky H.
8. Symetricky přepočteme dolní obálku (s orientací doprava).
9. Výsledný obal je tvořen body v obálkách H a D.

Rozebereme časovou složitost algoritmu. Setřídit body podle x-ové souřadnice dokážeme v čase $\mathcal{O}(n \log n)$. Přidání dalšího bodu do obálek trvá lineárně vzhledem k počtu odebraných bodů. Zde využijeme obvyklý argument: Každý bod je odebrán nejvýše jednou, a tedy všechna odebrání trvají dohromady $\mathcal{O}(n)$. Konvexní obal dokážeme sestrojit v čase $\mathcal{O}(n \log n)$ a pokud bychom měli seznam bodů již utříděný, zvládneme to dokonce v $\mathcal{O}(n)$.

Zbývá dořešit případy, kdy body nejsou v obecné poloze. Pokud se to stane, představíme si, že všemi body nepatrně pootočíme. Tím se nezmění, které body leží na konvexním obalu, a x-ové souřadnice se již budou lišit. Pořadí otočených bodů podle x-ové souřadnice přitom odpovídá lexikografickému pořadí původních bodů (nejprve podle x, pak podle y). Takže stačí v našem algoritmu vyměnit třídění podle x za lexikografické.

Orientace úhlu a determinanty

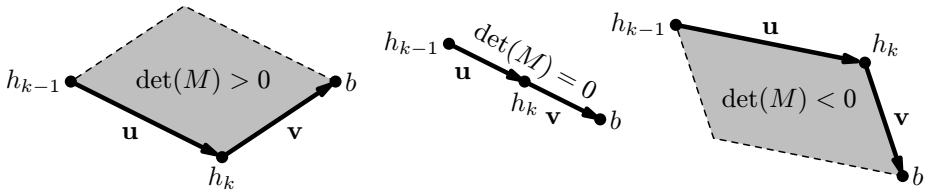
Při přepočítávání obálek jsme potřebovali testovat, zda je nějaký úhel orientovaný doleva nebo doprava. Jak na to? Ukážeme jednoduchý způsob založený na lineární algebře. Budou se k tomu hodit vlastnosti determinantu. Absolutní hodnota determinantu je objem rovnoběžnostěnu určeného řádkovými vektory matice. Důležitější však je, že znaménko determinantu určuje *orientaci* vektorů – zda je levotočivá či pravotočivá. Protože náš problém je rovinný, budeme používat determinanty matic 2×2.

Uvažme souřadnicový systém v rovině, jehož x-ová souřadnice roste směrem doprava a y-ová směrem nahoru. Chceme zjistit orientaci úhlu $h_{k-1}h_kb$. Označme $\mathbf{u} = (x_1, y_1)$ rozdíl souřadnic bodů h_k a h_{k-1} a podobně $\mathbf{v} = (x_2, y_2)$ rozdíl souřadnic bodů b a h_k. Matici M definujeme následovně:

$$M = \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \end{pmatrix} = \begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \end{pmatrix}.$$

Úhel $h_{k-1}h_kb$ je orientován doleva, právě když $\det M = x_1y_2 - x_2y_1$ je nezáporný. Možné situace jsou nakresleny na obrázku 16.3.

Determinant přitom zvládneme spočítat v konstantním čase a pokud jsou souřadnice bodů celočíselné, vystačí si i tento výpočet s celými čísly. Poznamenejme, že k podobnému vzorci se lze také dostat přes vektorový součin vektorů $\mathbf{u}$ a $\mathbf{v}$.



Obrázek 16.3: Jak vypadají determinanty různých znamének v rovině

Cvičení

1. Vyskytnou-li se na vstupu tři body na společné přímce, může náš algoritmus vydat konvexní obal, jehož některé vnitřní úhly jsou rovny 180 °. Definice obalu to připouští, ale někdy to může být nepraktické. Upravte algoritmus, aby takové vrcholy z obalu vynechával.

2. V rovině je dána množina červených a množina zelených bodů. Sestrojte přímku, která obě množiny oddělí. Na jedné její straně tedy budou ležet všechny červené body, zatímco na druhé všechny zelené. Navrhněte algoritmus, který takovou přímku nalezne.

3. Všimněte si, že pokud bychom netrvali na tom, aby bylo našich n jabloní oploceno jediným plotem, mohli bychom ušetřit pletivo. Sestrojte dva uzavřené ploty tak, aby každá jabloň byla oplocena a celkově jste spotřebovali nejméně pletiva.

4. Naznačíme, jak konvexní obal zavést formálně. Pamatujete si na lineární obaly ve vektorových prostorech? *Lineární obal* $\mathcal{L}(X)$ množiny vektorů X je průnik všech

vektorových podprostorů, které tuto množinu obsahují. Ekvivalentně je to množina všech *lineárních kombinací* vektorů z X, tedy všech součtů tvaru $\sum_i \alpha_i x_i$, kde $x_i \in X$ a $\alpha_i \in \mathbb{R}$.

Podobně můžeme definovat *konvexní obal* $\mathcal{C}(X)$ jako průnik všech konvexních množin, které obsahují X. Konvexní je přitom taková množina, která pro každé dva body obsahuje i celou úsečku mezi nimi. Nyní uvažujme množinu všech *konvexních kombinací,* což jsou součty tvaru $\sum_i \alpha_i x_i$, kde $x_i \in X$, $\alpha_i \in [0,1]$ a $\sum_i \alpha_i = 1$.

Jak vypadají konvexní kombinace pro 2-bodovou a 3-bodovou množinu X? Dokažte, že obecně je množina všech konvexních kombinací vždy konvexní a že je rovna $\mathcal{C}(X)$. Pro konečnou X má navíc tvar konvexního mnohoúhelníku, dokonce se to někdy používá jako jeho definice.

5.* Hledejme mezi všemi mnohoúhelníky, které obsahují danou konečnou množinu bodů, ten, který má nejmenší obvod. Dokažte, že každý takový mnohoúhelník musí být konvexní a navíc rovný konvexnímu obalu množiny. (Fyzikální analogie: do bodů zatlučeme hřebíky a natáhneme kolem nich gumičku. Ta zaujme stav o nejnižší energii, tedy nejkratší křivku. My zde nechceme zabíhat do matematické analýzy, takže se omezíme na lomené čáry.)

6. Může jít sestrojit konvexní obal rychleji než v $\Theta(n \log n)$? Nikoliv, alespoň pokud chceme body na konvexním obalu vypisovat v pořadí, v jakém se na jeho hranici nacházejí. Ukažte, že v takovém případě můžeme pomocí konstrukce konvexního obalu třídit reálná čísla. Náš dolní odhad složitosti třídění z oddílu 3.3 sice na tuto situaci nelze přímo použít, ale existuje silnější (a těžší) věta, z níž plyne, že i na třídění n reálných čísel je potřeba $\Omega(n \log n)$ operací. Dále viz oddíl 16.5.

16.2 Průsečíky úseček

Nyní se zaměříme na další geometrický problém. Dostaneme n úseček a zajímá nás, které z nich se protínají a kde. Na první pohled na tom není nic zajímavého: n úseček může mít až $\Theta(n^2)$ průsečíků, takže i triviální algoritmus, který zkusí protnout každou úsečku s každou, bude optimální. V reálných situacích nicméně počet průsečíků bývá mnohem menší. Podobnou situaci jsme už potkali při vyhledávání v textu. Opět budeme hledat algoritmus, který má příznivou složitost nejen vzhledem k počtu bodů n, ale také k počtu průsečíků p.

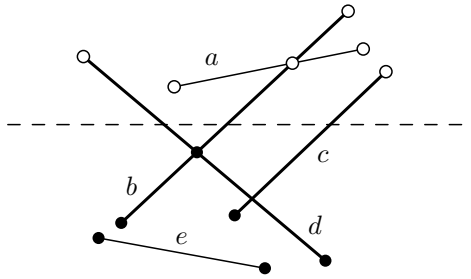
Pro začátek zase předpokládejme, že úsečky leží v obecné poloze. To tentokrát znamená, že žádné tři úsečky se neprotínají v jednom bodě, průnikem každých dvou úseček je nejvýše jeden bod, krajní bod žádné úsečky neleží na jiné úsečce, a konečně také neexistují vodorovné úsečky.

Podobně jako u hledání konvexního obalu, i zde využijeme myšlenku zametání roviny. Budeme posouvat vodorovnou přímku odshora dolů, všímat si, jaké úsečky zrovna protínají zametací přímku a jaké mezi sebou mají průsečíky.

Namísto spojitého posouvání budeme přímkou skákat po *událostech,* což budou místa, kde se něco zajímavého děje: *začátky úseček, konce úseček* a *průsečíky úseček.* Pozice začátků a konců úseček známe předem, průsečíkové události budeme objevovat průběžně.

V každém kroku výpočtu si pamatujeme *průřez* P – posloupnost úseček zrovna protatých zametací přímkou. Tyto úsečky máme utříděné zleva doprava. Navíc si udržujeme *kalendář* K budoucích událostí.

V kalendáři jsou naplánovány všechny začátky a konce ležící pod zametací přímkou. Navíc se pro každou dvojici sousedních úseček v průřezu podíváme, zda se pod zametací přímkou protnou, a pokud ano, tak takový průsečík také naplánujeme. Všimněme si, že těsně předtím, než se dvě úsečky protnou, musí v průřezu sousedit, takže na žádný průsečík nezapomeneme. Jen pozor na to, že naplánované průsečíky musíme občas z kalendáře zase vymazat – mezi dvojici sousedních úseček se může dočasně vtěsnat třetí.



Obrázek 16.4: Průřez a události v kalendáři

Jak to vypadá, můžeme sledovat na obrázku 16.4: pro čárkovanou polohu zametací přímky leží v průřezu tučné úsečky. Kroužky odpovídají událostem: plné kroužky jsou naplánované, prázdné už nastaly. O průsečíku úseček c a d dosud nevíme, neboť se ještě nestaly sousedními.

Celý algoritmus bude vypadat následovně:

Algoritmus Průsečíky (průsečíky úseček)

1. Inicializujeme průřez P na $\emptyset$.
2. Do kalendáře K vložíme začátky a konce všech úseček.
3. Dokud K není prázdný:

4. Odebereme nejvyšší událost.
5. Pokud je to začátek úsečky: zatřídíme novou úsečku do P.
6. Pokud je to konec úsečky: odebereme úsečku z P.
7. Pokud je to průsečík: nahlásíme ho a prohodíme úsečky v P.
8. Přepočítáme naplánované průsečíkové události v okolí změny v P (nejvýše dvě odebereme a dvě nové přidáme).

Zbývá rozebrat, jaké datové struktury použijeme pro reprezentaci průřezu a kalendáře. S kalendářem je to snadné, ten můžeme uložit například do haldy nebo do vyhledávacího stromu. V každém okamžiku se v kalendáři nachází nejvýše $3n$ událostí: n začátků, n konců a n průsečíků. Proto operace s kalendářem stojí $\mathcal{O}(\log n)$.

Co potřebujeme dělat s průřezem? Vkládat a odebírat úsečky a při plánování průsečíkových událostí také hledat nejbližší další úsečku vlevo či vpravo od aktuální. Nabízí se využít vyhledávací strom. Jenže jako klíče v něm nemohou vystupovat přímo x-ové souřadnice úseček, respektive jejich průsečíků se zametací přímkou. Ty se totiž při každém posunutí našeho „koštěte“ mohou všechny změnit.

Uložíme raději do vrcholů místo souřadnic jen odkazy na úsečky. Ty se nemění a mezi událostmi se nemění ani jejich pořadí. Kdykoliv pak operace se stromem navštíví nějaký vrchol, dopočítáme aktuální souřadnici úsečky a podle toho se rozhodneme, zda se vydat doleva, nebo doprava. Jelikož průřez vždy obsahuje nejvýše n úseček, operace se stromem budou trvat $\mathcal{O}(\log n)$.

Při vyhodnocování každé události provedeme $\mathcal{O}(1)$ operací s datovými strukturami, takže jednu událost zpracujeme v čase $\mathcal{O}(\log n)$. Všech $\mathcal{O}(n+p)$ událostí zpracujeme v čase $\mathcal{O}((n+p)\log n)$, což je také časová složitost celého algoritmu.

Na závěr poznamenejme, že existuje efektivnější, byť daleko komplikovanější, algoritmus od Bernarda Chazella dosahující časové složitosti $\mathcal{O}(n \log n + p)$.

Cvičení

1. Tvrdili jsme, že n úseček může mít $\Theta(n^2)$ průsečíků. Zkuste takový systém úseček najít.

2. Popište, jak algoritmus upravit, aby nepotřeboval předpoklad obecné polohy úseček. Především je potřeba v některých případech domyslet, co vůbec má být výstupem algoritmu.

3. Navrhněte algoritmus, který nalezne nejdelší vodorovnou úsečku ležící uvnitř daného (ne nutně konvexního) mnohoúhelníku.

4. Je dána množina obdélníků, jejichž strany jsou rovnoběžné s osami souřadnic. Spočítejte obsah jejich sjednocení.

5. Jak zjistit, zda dva konvexní mnohoúhelníky jsou disjunktní? Mnohoúhelníky uvažujeme včetně vnitřku. Prozradíme, že to jde v lineárním čase.

6.* Pro dané dva mnohoúhelníky vypočtěte jejich průnik (to je obecně nějaká množina mnohoúhelníků). Nebo alespoň zjistěte, zda průnik je neprázdný.

7. Mějme množinu parabol tvaru $y = ax^2 + bx + c$, kde $a > 0$. Nalezněte všechny jejich průsečíky.

8.* Co když v předchozím cvičení dovolíme i $a < 0$?

16.3 Voroného diagramy

V daleké Arktidě bydlí Eskymáci a lední medvědi.[1] A navzdory obecnému mínění se spolu přátelí. Představte si medvěda putujícího nezměrnou polární pustinou na cestě za nejbližším iglú, kam by mohl zajít na kus řeči a pár ryb. Proto se medvědovi hodí mít po ruce Voroného diagram Arktidy.

Definice: *Voroného diagram*[2] pro množinu bodů neboli *míst* $x_1, \ldots, x_n \in \mathbb{R}^2$ je systém *oblastí* $B_1, \ldots, B_n \subseteq \mathbb{R}^2$, kde B_i obsahuje ty body roviny, jejichž vzdálenost od x_i je menší nebo rovna vzdálenostem od všech ostatních x_j.

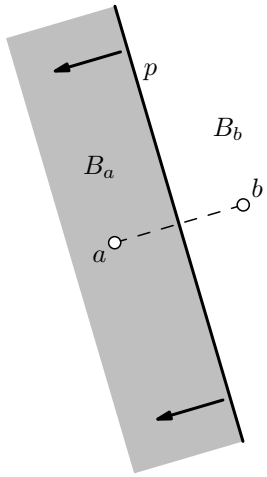
Nahlédneme, že Voroného diagram má překvapivě jednoduchou strukturu. Nejprve uvažme, jak budou vypadat oblasti B_a a B_b pro dva body a a b (viz obrázek 16.5). Všechny body stejně vzdálené od a i b leží na přímce p – ose úsečky ab. Oblasti B_a a B_b jsou tedy tvořeny polorovinami ohraničenými osou p. Osa sama leží v obou oblastech.

Nyní obecněji: Oblast B_i má obsahovat body, které mají k x_i blíže než k ostatním bodům. Musí být tedy tvořena průnikem $n-1$ polorovin, takže je to (možná neomezený) konvexní mnohoúhelník. Příklad Voroného diagramu najdete na obrázku 16.6: zadaná místa jsou označena prázdnými kroužky, hranice oblastí B_i jsou vyznačeny plnými čarami.

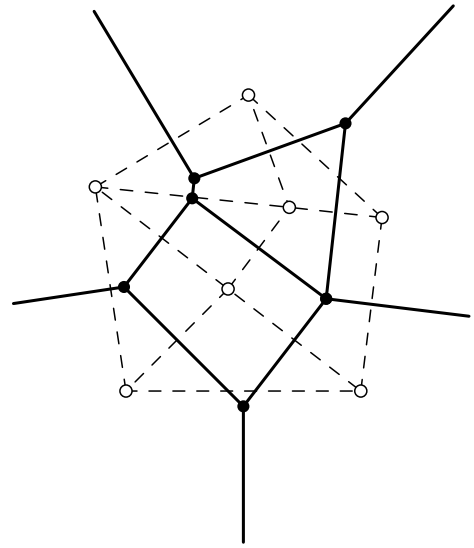
Voroného diagram připomíná rovinný graf. Jeho vrcholy jsou body, které jsou stejně vzdálené od alespoň tří zadaných míst. Jeho stěny jsou oblasti B_i. Hrany jsou tvořeny částmi hranice mezi dvěma oblastmi – těmi body, které mají obě oblasti společné (to může

[1] Ostatně, Arktida se podle medvědů (řecky ἄρκτος) přímo jmenuje. Jen ne podle těch ledních, nýbrž nebeských: daleko na severu se souhvězdí Velké medvědice vyjímá přímo v nadhlavníku.

[2] Diagramy tohoto druhu zkoumal začátkem 20. století ruský matematik Georgij Voronoj. Dvojrozměrnou verzi nicméně znal už René Descartes v 17. století.



Obrázek 16.5: Body bližší k a než b



Obrázek 16.6: Voroného diagram

být úsečka, polopřímka nebo přímka). Oproti rovinnému grafu nemusí stěny být omezené, ale pokud nám to vadí, můžeme celý diagram uzavřít do dostatečně velkého obdélníku.

Můžeme také sestrojit duální graf: jeho vrcholy budou odpovídat oblastem (nakreslíme je do jednotlivých míst), stěny vrcholům diagramu a hrany budou úsečky spojující místa v sousedních oblastech (přerušované čáry na obrázku).

Lemma: Voroného diagram má lineární kombinatorickou složitost. Tím myslíme, že diagram pro n míst obsahuje $\mathcal{O}(n)$ vrcholů, hran i stěn.

Důkaz: Využijeme následující standardní tvrzení o rovinných grafech [9]:

> **Fakt:** Mějme souvislý rovinný graf bez násobných hran. Označme $v \geq 3$ počet jeho vrcholů, e počet hran a f počet stěn. Pak platí:
>
> - $e \leq 3v - 6$
> - $v + f = e + 2$ (Eulerova formule)

Diagram pro n míst má n oblastí, takže po „zavření do krabičky“ vznikne rovinný graf o $f = n + 1$ stěnách (z toho jedna vnější). Jeho duál má $v' = f$ vrcholů a nejsou v něm násobné hrany (rozmyslete si, proč). Proto pro jeho počet hran musí platit $e' \leq 3v' - 6$. Hrany duálu nicméně odpovídají hranám původního grafu, kde tedy platí $e \leq 3f - 6 =$

$3n - 3$. Počet vrcholů odhadneme dosazením do Eulerovy formule: $v = e + 2 - f \leq (3n - 3) + 2 - (n + 1) = 2n - 2$. □

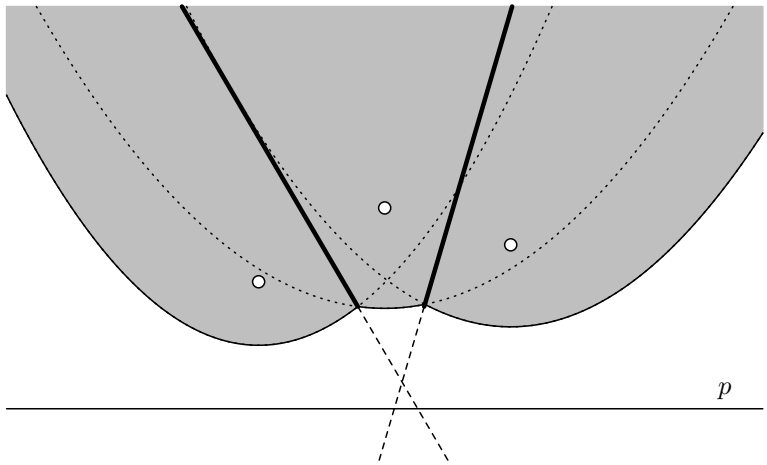
Voroného diagram pro n zadaných míst je tedy velký $\mathcal{O}(n)$. Nyní ukážeme, jak ho zkonstruovat v čase $\mathcal{O}(n \log n)$.

Fortunův algoritmus*

Situaci si zjednodušíme předpokladem obecné polohy: budeme očekávat, že žádné čtyři body neleží na společné kružnici. Vrcholy diagramu proto budou mít stupeň nejvýše 3.

Použijeme osvědčenou strategii zametání roviny přímkou shora dolů. Obvyklá představa, že nad přímkou už máme vše hotové, ovšem selže: Pokud přímka narazí na nové místo, hotová část diagramu nad přímkou se může poměrně složitě změnit. Pomůžeme si tak, že nebudeme považovat za hotovou celou oblast nad zametací přímkou, nýbrž jen tu její část, která má blíž k některému z míst nad přímkou než ke přímce. V této části se už to, co jsme sestrojili, nemůže přidáváním dalších bodů změnit.

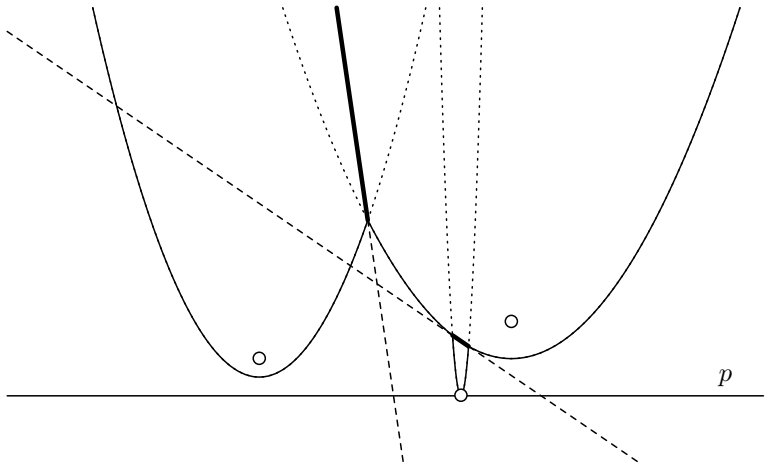
Jak vypadá hranice hotové části? Body mající stejnou vzdálenost od bodu (ohniska) jako od řídicí přímky tvoří parabolu. Hranice tudíž musí být tvořena posloupností parabolických oblouků. Krajní dva oblouky jdou do nekonečna, ostatní jsou konečné. Vzhledem k charakteristickému tvaru budeme hranici říkat *pobřeží.*



Obrázek 16.7: Linie pobřeží ohraničuje šedou oblast, v níž je diagram hotov

Posouváme-li zametací přímkou, pobřeží se mění a průsečíky oblouků vykreslují hrany diagramu. Pro každý průsečík totiž platí, že je vzdálený od zametací přímky stejně jako od dvou různých míst. Tím pádem leží na hraně diagramu oddělující tato dvě místa.

Kdykoliv zametací přímka narazí na nějaké další místo, vznikne nová parabola, zprvu degenerovaná do polopřímky kolmé na zametací přímku. Této situaci říkáme *místní událost* a vidíme ji na obrázku 16.8. Pokračujeme-li v zametání, nová parabola se začne rozevírat a její průsečíky s původním pobřežím vykreslují novou hranu diagramu. Hrana se přitom rozšiřuje na obě strany a teprve časem se propojí s ostatními hranami.

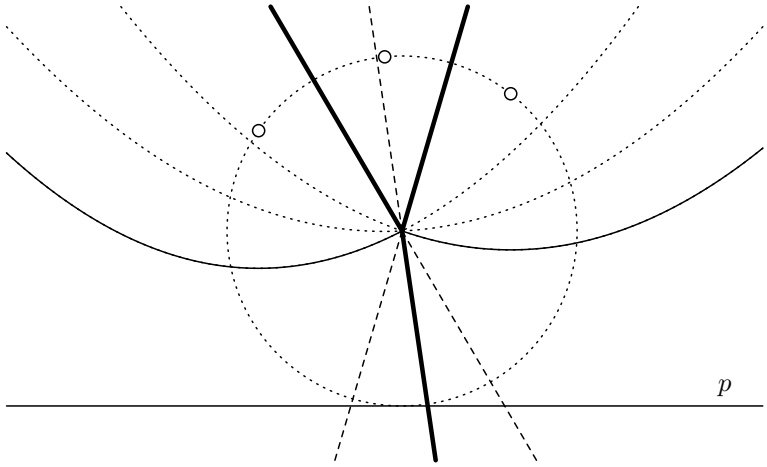


Obrázek 16.8: Krátce po místní události: přibyla nová parabola, její průsečíky kreslí tutéž hranu diagramu do obou stran

Mimo to se může stát, že nějaká parabola se rozevře natolik, že pohltí jiné a ty zmizí z pobřežní linie. Situaci sledujme na obrázku 16.9. Mějme nějaké tři paraboly jdoucí v pobřeží po sobě. Prostřední z nich je pohlcena v okamžiku, kdy se hrany vykreslované průsečíky parabol setkají v jednom bodě. Tento bod musí být stejně daleko od všech třech ohnisek, takže je středem kružnice opsané trojici ohnisek.

Kde je v tomto okamžiku zametací přímka? Musí být v takové poloze, aby střed kružnice právě vykoukl zpoza pobřeží. Jinými slovy musí být stejně daleko od středu, jako jsou ohniska, čili se kružnice dotýkat zespodu. Této situaci říkáme *kružnicová událost*.

Algoritmus proto bude udržovat nějaký kalendář událostí a vždy skákat zametací přímkou na následující událost. Místní události můžeme všechny naplánovat dopředu, kružnicové budeme plánovat (a přeplánovávat) průběžně, kdykoliv se pobřeží změní.



Obrázek 16.9: Kružnicová událost: parabola se schovává pod dvě sousední, dvě hrany zanikají a jedna nová vzniká

To je podobné algoritmu na průsečíky úseček a stejně tak budou podobné i datové struktury: kalendář si budeme uchovávat v haldě nebo vyhledávacím stromu, pobřeží ve vyhledávacím stromu s implicitními klíči: v každém vrcholu si uložíme ohniska dvou parabol, jejichž průsečíkem má vrchol být.

Poslední datovou strukturou bude samotný diagram, reprezentovaný grafem se souřadnicemi a vazbami hran na průsečíky v pobřeží.

Algoritmus FORTUNE

1. Vytvoříme kalendář K a vložíme do něj všechny místní události.
2. Založíme prázdnou pobřežní linii P.
3. Dokud kalendář není prázdný:
4. Odebereme další událost.
5. Je-li to místní událost:
6. Najdeme v P parabolu podle x-ové souřadnice místa.
7. Rozdělíme ji a mezi její části vložíme novou parabolu.
8. Do diagramu zaznamenáme novou hranu, která zatím není nikam připojena.
9. Je-li to kružnicová událost:
10. Smažeme parabolu z P.

11. Do diagramu zaznamenáme vrchol, v němž dvě hrany končí a jedna začíná.
12. Po změně pobřeží přepočítáme kružnicové události ($\mathcal{O}(1)$ jich zanikne, $\mathcal{O}(1)$ vznikne).

Věta: Fortunův algoritmus pracuje v čase $\mathcal{O}(n \log n)$ a prostoru $\mathcal{O}(n)$.

Důkaz: Celkově nastane n místních událostí (na každé místo narazíme právě jednou) a n kružnicových (kružnicová událost smaže jednu parabolu z pobřeží a ty přibývají pouze při místních událostech). Z toho plyne, že kalendář i pobřežní linie jsou velké $\mathcal{O}(n)$, takže pracují v čase $\mathcal{O}(\log n)$ na operaci. Jednu událost proto naplánujeme i obsloužíme v čase $\mathcal{O}(\log n)$, což celkem dává $\mathcal{O}(n \log n)$. □

Cvičení

1. Dokažte, že sestrojíme-li konvexní obal množiny míst, prochází každou jeho hranou právě jedna nekonečná hrana Voroného diagramu.

2. Vymyslete, jak algoritmus upravit, aby nepotřeboval předpoklad obecné polohy.

3.* *Navigace robota:* Mějme kruhového robota, který se pohybuje mezi bodovými překážkami v rovině. Jak zjistit, zda se robot může dostat z jednoho místa na druhé? Rozmyslete si, že stačí uvažovat cesty po hranách Voroného diagramu. Jen je potřeba dořešit, jak se z počátečního bodu dostat na diagram a na konci zase zpět.

4.* *Delaunayova triangulace* (popsal ji Boris Děloné, ale jeho jméno obvykle potkáváme v pofrancouzštěné podobě) je duálním grafem Voroného diagramu, na obrázku 16.6 je vyznačena čárkovaně. Vznikne tak, že spojíme úsečkami místa, jejichž oblasti ve Voroného diagramu sousedí. Dokažte, ze žádné dvě vybrané úsečky se nekříží a že pokud žádná čtyři místa neleží na společné kružnici, jedná se o rozklad vnitřku konvexního obalu míst na trojúhelníky. Může se hodit, že úsečka mezi místy a a b je vybraná právě tehdy, když kružnice s průměrem ab neobsahuje žádná další místa.

5.* *Euklidovská minimální kostra:* Představte si, že chceme pospojovat zadaná místa systémem úseček tak, aby se dalo dostat odkudkoliv kamkoliv a celková délka úseček byla nejmenší možná. Hledáme tedy minimální kostru úplného grafu, jehož vrcholy jsou místa a délky hran odpovídají euklidovským vzdálenostem. Můžeme použít libovolný algoritmus z kapitoly 7, ale musíme zpracovat kvadraticky mnoho hran. Dokažte, že hledaná kostra je podgrafem Delaunayovy triangulace, takže stačí zkoumat lineárně mnoho hran.

6.** *Obecnější Voroného diagram:* Jak by to dopadlo, kdyby místa nebyla jen bodová, ale mohly by to být i úsečky? Dokažte, že v takovém případě diagram opět tvoří

rovinný graf, ovšem jeho hrany jsou kromě úseček tvořeny i parabolickými oblouky. Dokázali byste upravit Fortunův algoritmus, aby fungoval i pro tyto diagramy?

16.4 Lokalizace bodu

Pokračujme v problému z minulého oddílu. Máme nějakou množinu *míst* v rovině a chceme umět pro libovolný bod nalézt nejbližší místo (pokud jich je víc, stačí libovolné jedno). To už umíme převést na nalezení oblasti ve Voroného diagramu, do které zadaný bod padne.

Chceme tedy pro nějaký rozklad roviny na mnohoúhelníkové oblasti vybudovat datovou strukturu, která pro libovolný bod rychle odpoví, do jaké oblasti patří. Tomuto problému se říká *lokalizace bodu.*

Začneme primitivním řešením bez předzpracování. Rovinu zametáme shora dolů vodorovnou přímkou, podobně jako při hledání průsečíků úseček. Udržujeme si průřez hranic oblastí zametací přímkou. Tento průřez se mění jenom ve vrcholech mnohoúhelníků. Ve chvíli, kdy narazíme na hledaný bod, podíváme se, do kterého intervalu mezi hranicemi v průřezu patří. Tento interval odpovídá jedné oblasti, kterou nahlásíme. Kalendář událostí i průřez opět ukládáme do vyhledávacích stromů, jednu událost obsloužíme v $\mathcal{O}(\log n)$ a celý algoritmus běží v $\mathcal{O}(n \log n)$.

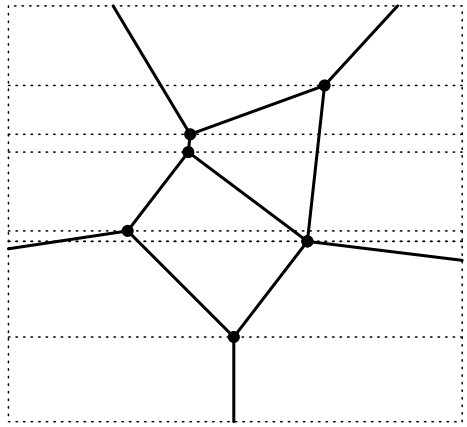
To je obludně pomalé, dokonce pomalejší než pokaždé projít všechny oblasti a pro každou zjistit, zda v ní zadaný bod leží. Ale i z ošklivé housenky se může vyklubat krásný motýl ...

Zavedeme předzpracování. Zametání oblastí necháme běžet „naprázdno“, aniž bychom hledali konkrétní bod. Rovinu rozřežeme polohami zametací přímky při jednotlivých událostech na pásy. Pro každý pás si zapamatujeme kopii průřezu (ten se uvnitř pásu nemění) a navíc si uložíme y-ové souřadnice hranic pásů. Nyní na vyhodnocení dotazu stačí najít podle y-ové souřadnice správný pás (což jistě zvládneme v logaritmickém čase) a poté položit dotaz na zapamatovaný průřez pro tento pás.

Dotaz dokážeme zodpovědět v čase $\mathcal{O}(\log n)$, ovšem předvýpočet vyžaduje čas $\Theta(n^2)$ na zkopírování všech n stromů a spotřebuje na to stejné množství prostoru.

Persistentní vyhledávací stromy

Složitost předvýpočtu zachráníme tím, že si pořídíme *persistentní vyhledávací strom.* Ten si pamatuje historii všech svých změn a umí vyhledávat nejen v aktuálním stavu, ale i ve všech stavech z minulosti. Přesněji řečeno, po každé operaci, která mění stav stromu, vznikne nová *verze* stromu a operace pro dotazy dostanou jako další parametr identifikátor verze, ve které mají hledat. Upravovat lze vždy jen nejnovější verzi.



Obrázek 16.10: Oblasti rozřezané na pásy

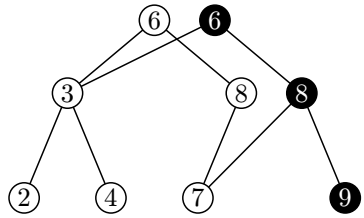
Předvýpočet tedy bude udržovat průřez v persistentním stromu a místo aby ho v každém pásu zkopíroval, jen si zapamatuje identifikátor verze, která k pásu patří.

Popíšeme jednu z možných konstrukcí persistentního stromu. Uvažujme obyčejný vyhledávací strom, řekněme AVL strom. Rozhodneme se ale, že jeho vrcholy nikdy nebudeme měnit, abychom neporušili zaznamenanou historii. Místo toho si pořídíme kopii vrcholu a tu změníme. Musíme ovšem změnit ukazatel na daný vrchol, aby ukazoval na kopii. Proto zkopírujeme i otce a upravíme v něm ukazatel. Tím pádem musíme upravit i ukazatel na otce, atd., až se dostaneme do kořene. Kopie kořene se pak stane identifikátorem nové verze.

Strom nové verze tedy obsahuje novou cestu mezi kořenem a upravovaným vrcholem. Tato cesta se odkazuje na podstromy z minulé verze. Uchování jedné verze nás proto strojí čas $\mathcal{O}(\log n)$ a prostor taktéž $\mathcal{O}(\log n)$. Ještě nesmíme zapomenout, že po každé operaci následuje vyvážení stromu. To ovšem upravuje pouze vrcholy, které leží v konstantní vzdálenosti od cesty mezi místem úpravy a kořenem, takže jejich zkopírováním časovou ani prostorou složitost nezhoršíme.

Na předzpracování Voroného diagramu a vytvoření persistentního stromu tedy spotřebujeme čas $\mathcal{O}(n \log n)$. Strom spotřebuje paměť $\mathcal{O}(n \log n)$. Dotazy vyřizujeme v čase $\mathcal{O}(\log n)$, neboť nejprve vyhledáme správný pás a poté položíme dotaz na příslušnou verzi stromu.

Poznámka: Persistence datových struktur je přirozená pro striktní funkcionální programovací jazyky (například Haskell). V nich neexistují vedlejší efekty příkazů, takže jednou



Obrázek 16.11: Vložení prvku 9 do persistentního stromu

sestrojená data již nelze modifikovat, pouze vyrobit novou verzi datové struktury s provedenou změnou.

Persistence v konstantním prostoru na verzi*

Spotřeba paměti $\Theta(\log n)$ na uložení jedné verze je zbytečně vysoká. Existuje o něco chytřejší konstrukce persistentního stromu, které stačí konstantní paměť, tedy alespoň amortizovaně. Nastíníme, jak funguje.

Nejprve si pořídíme vyhledávací strom, který při každém vložení nebo smazání prvku provede jen amortizovaně konstantní počet *strukturálních změn* (to jsou změny hodnot a ukazatelů, zkrátka všeho, podle čeho se řídí vyhledávání, a co je tudíž potřeba verzovat; změna znaménka uloženého ve vrcholu AVL-stromu tedy strukturální není). Tuto vlastnost mají třeba (2,4)-stromy (viz cvičení 9.3.7) nebo některé varianty červeno-černých stromů.

Nyní ukážeme, jak jednu strukturální změnu zaznamenat v amortizovaně konstantním prostoru. Každý vrchol stromu si tentokrát bude pamatovat až dvě své verze (spolu s časy jejich vzniku). Při průchodu od kořene porovnáme čas vzniku těchto verzí s aktuálním časem a vybereme si správnou verzi. Pokud potřebujeme zaznamenat novou verzi vrcholu, buďto na ni ve vrcholu ještě je místo, nebo není a v takovém případě vrchol zkopírujeme, což vynutí změnu ukazatele v rodiči, a tedy i vytvoření nové verze rodiče, atd. až případně do kořene. Identifikátorem verze celé datové struktury bude ukazatel na aktuální kopii kořene spolu s časem vzniku verze.

Chod struktury si můžeme představovat tak, že stejně jako v předchozí verzi persistentních stromů propagujeme změny směrem ke kořeni, ale tentokrát se tempo propagování exponenciálně zmenšuje: Změna vrcholu způsobí zkopírování, a tím pádem změnu otce, průměrně v každém druhém případě. Počet všech změn tedy tvoří geometrickou řadu s konstantním součtem. Exaktněji řečeno:

Věta: Uchování jedné strukturální změny stojí amortizovaně konstantní čas i prostor.

Důkaz: Každé vytvoření verze vrcholu stojí konstantní čas a prostor. Jedna operace může v nejhorším případě způsobit vznik nových verzí všech vrcholů až do kořene, ale jednoduchým potenciálovým argumentem lze dokázat, že počet všech vzniklých verzí bude amortizovaně konstantní.

Potenciál struktury definujeme jako počet verzí uchovaných ve všech vrcholech dosažitelných z aktuálního kořene v aktuálním čase. V klidovém stavu struktury jsou ve vrcholu nejvýš dvě verze, během aktualizace dočasně připustíme tři verze.

Strukturální změna způsobí zaznamenání nové verze jednoho vrcholu, což potenciál zvýší o 1, ale možná tím vznikne „tříverzový“ vrchol. Zbytek algoritmu se tříverzových vrcholů snaží zbavit: pokaždé vezme vrchol se 3 verzemi, vytvoří jeho kopii s 1 verzí a upraví ukazatel v otci, čímž přibude nová verze otce. Originálnímu vrcholu zůstanou 2 verze, ale přestane být dosažitelný, takže se už do potenciálu nepočítá.

Potenciál tím klesne o 3 (za odpojený originál), zvýší se o 1 (za nově vytvořenou kopii s jednou verzí) a poté ještě o 1 (za novou verzi otce). Celkově tedy klesne o 1. Proto veškeré kopírování vrcholů zaplatíme z konstantního příspěvku od každé strukturální změny. □

Důsledek: Existuje persistentní vyhledávací strom s časem amortizovaně $\mathcal{O}(\log n)$ na operaci a prostorem amortizovaně $\mathcal{O}(1)$ na uložení jedné verze. Pomocí něj lze v čase $\mathcal{O}(n \log n)$ vybudovat datovou strukturu pro lokalizaci bodu, která odpovídá na dotazy v čase $\mathcal{O}(\log n)$ a zabere prostor $\mathcal{O}(n)$.

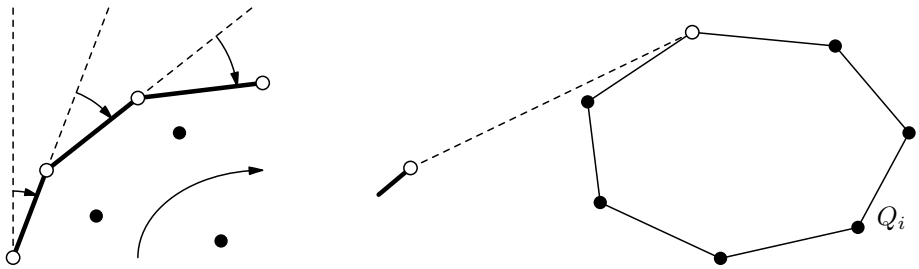
16.5* Rychlejší algoritmus na konvexní obal

Konvexním obalem naše putování po geometrických algoritmech začalo a také jím skončí. Našli jsme algoritmus pro výpočet konvexního obalu n bodů v čase $\Theta(n \log n)$. Ve cvičení 16.1.6 jsme dokonce dokázali, že tato časová složitost je optimální. Přesto předvedeme ještě rychlejší algoritmus objevený v roce 1996 Timothym Chanem. S naším důkazem optimality je nicméně všechno v pořádku: časová složitost Chanova algoritmu dosahuje $\mathcal{O}(n \log h)$, kde h značí počet bodů ležících na konvexním obalu.

Předpokládejme, že bychom znali velikost konvexního obalu h. Body libovolně rozdělíme do $\lceil n/h \rceil$ množin $Q_1, \ldots, Q_k$ tak, aby v každé množině bylo nejvýše h bodů. Pro každou z těchto množin nalezneme konvexní obal pomocí obvyklého algoritmu. To dokážeme pro jednu množinu v čase $\mathcal{O}(h \log h)$ a pro všechny v $\mathcal{O}(n \log h)$. Poté tyto předpočítané obaly slepíme do jednoho pomocí takzvaného *provázkového algoritmu.* Ten se opírá o následující pozorování:

Pozorování: Úsečka spojující dva body a a b leží na konvexním obalu, právě když všechny ostatní body leží na téže straně přímky proložené touto úsečkou.

Algoritmu se říká provázkový, protože svou činností připomíná namotávání provázku podél konvexního obalu. Začneme bodem, který na konvexním obalu určitě leží – třeba tím nejlevějším. V každém dalším kroku nalezneme následující bod po obvodu konvexního obalu. Například tak, že projdeme všechny body a vybereme ten, který svírá nejmenší úhel s předchozí stranou konvexního obalu. Nově přidaná úsečka vyhovuje pozorování, a tudíž do konvexního obalu patří. Po h krocích se dostaneme zpět k nejlevějšímu bodu a výpočet ukončíme. V každém kroku potřebujeme projít všechny body a vybrat následníka, což dokážeme v čase $\mathcal{O}(n)$. Celková složitost algoritmu je tedy $\mathcal{O}(nh)$.



Obrázek 16.12: Provázkový algoritmus a jeho použití v předpočítaném obalu

Provázkový algoritmus funguje, ale je ukrutně pomalý. Kýženého zrychlení dosáhneme, pokud použijeme předpočítané konvexní obaly. Ty umožní rychleji hledat následníka. Pro každou z množin Q_i najdeme zvlášť kandidáta a poté z nich vybereme toho nejlepšího. Možný kandidát vždy leží na konvexním obalu množiny Q_i. Využijeme toho, že body obalu jsou „uspořádané“, i když trochu netypicky do kruhu. Kandidáta můžeme hledat metodou půlení intervalu, jen detaily jsou maličko složitější, než je obvyklé. Jak půlit, zjistíme podle směru zatáčení konvexního obalu. Zbytek ponechme jako cvičení.

Časová složitost půlení je $\mathcal{O}(\log h)$ pro jednu množinu. Množin je $\mathcal{O}(n/h)$, tedy následující bod konvexního obalu nalezneme v čase $\mathcal{O}(n/h \cdot \log h)$. Všech h bodů obalu nalezneme ve slibovaném čase $\mathcal{O}(n \log h)$.

Popsanému algoritmu schází jedna důležitá věc: Ve skutečnosti málokdy známe velikost h. Budeme proto algoritmus iterovat s rostoucí hodnotou h, dokud konvexní obal nesestrojíme. Pokud při slepování konvexních obalů zjistíme, že konvexní obal je větší než h, výpočet ukončíme. Zbývá ještě zvolit, jak rychle má h růst. Pokud by rostlo moc pomalu, budeme počítat zbytečně mnoho fází, naopak při rychlém růstu by nás poslední fáze mohla stát příliš mnoho.

V k-té fázi položíme $h = 2^{2^k}$. Dostáváme celkovou složitost algoritmu:

$$\sum_{m=0}^{\lceil \log \log h \rceil} \mathcal{O}(n \log 2^{2^m}) = \sum_{m=0}^{\lceil \log \log h \rceil} \mathcal{O}(n \cdot 2^m) = \mathcal{O}(n \log h),$$

kde poslední rovnost dostaneme jako součet prvních $\lceil \log \log h \rceil$ členů geometrické řady $\sum_m 2^m$.

Cvičení

1. Domyslete detaily hledání kandidáta „kruhovým“ půlením intervalu.

16.6 Další cvičení

1. Navrhněte algoritmus pro výpočet obsahu konvexního mnohoúhelníku. Mnohoúhelník je zadán seznamem souřadnic vrcholů tak, jak jdou po obvodu ve směru hodinových ručiček.

2.* Navrhněte algoritmus pro výpočet obsahu nekonvexního mnohoúhelníku. Prozradíme, že to jde v lineárním čase.

3. Jak o množině bodů v rovině zjistit, zda je středově symetrická?

4.* Je dána množina bodů v rovině. Rozložte ji na dvě disjunktní středově symetrické množiny, je-li to možné.

5. Jak k dané množině bodů v rovině najít obdélník s nejmenším možným obvodem, který obsahuje všechny dané body? Obdélník nemusí mít strany rovnoběžné s osami.

6. Vymyslete datovou strukturu, která bude udržovat konvexní obal množiny bodů a bude ho umět rychle přepočítat po přidání bodu do množiny.

7. Je dána množina obdélníků, jejichž strany jsou rovnoběžné s osami souřadnic. Vybudujte datovou strukturu, která bude umět rychle odpovídat na dotazy typu „v kolika obdélnících leží zadaný bod?“.

17 Fourierova transformace

17 Fourierova transformace

Co má společného násobení polynomů s kompresí zvuku? Nebo třeba s rozpoznáváním obrazu? V této kapitole ukážeme, že na pozadí všech těchto otázek je společná algebraická struktura, kterou matematici znají pod názvem *diskrétní Fourierova transformace.* Odvodíme efektivní algoritmus pro výpočet této transformace a ukážeme některé jeho zajímavé důsledky.

17.1 Polynomy a jejich násobení

Nejprve stručně připomeňme, jak se pracuje s polynomy.

Definice: *Polynom* je výraz typu

$$P(x) = \sum_{i=0}^{n-1} p_i \cdot x^i,$$

kde x je proměnná a p_0 až p_{n-1} jsou čísla, kterým říkáme *koeficienty* polynomu. Zde budeme značit polynomy velkými písmeny a jejich koeficienty příslušnými malými písmeny s indexy. Zatím budeme předpokládat, že všechna čísla jsou reálná, v obecnosti by to mohly být prvky libovolného komutativního okruhu.

V algoritmech obvykle polynomy reprezentujeme *vektorem koeficientů* $(p_0, \ldots, p_{n-1})$; oproti zvyklostem lineární algebry budeme složky vektorů v celé této kapitole indexovat od 0. Počtu koeficientů n budeme říkat *velikost polynomu* $|P|$. Časovou složitost algoritmu budeme vyjadřovat vzhledem k velikostem polynomů zadaných na vstupu. Budeme předpokládat, že s reálnými čísly umíme pracovat v konstantním čase na operaci.

Pokud přidáme nový koeficient $p_n = 0$, hodnota polynomu se pro žádné x nezmění. Stejně tak je-li nejvyšší koeficient p_{n-1} nulový, můžeme ho vynechat. Takto můžeme každý polynom zmenšit na *normální tvar,* v němž má buďto nenulový nejvyšší koeficient, nebo nemá vůbec žádné koeficienty – to je takzvaný *nulový polynom,* který pro každé x roven nule. Nejvyšší mocnině s nenulovým koeficientem se říká *stupeň polynomu* $\deg P$, nulovému polynomu přiřazujeme stupeň -1.

S polynomy zacházíme jako s výrazy. Sčítání a odečítání je přímočaré, ale podívejme se, co se děje při *násobení:*

$$P(x) \cdot Q(x) = \left(\sum_{i=0}^{n-1} p_i \cdot x^i\right) \cdot \left(\sum_{j=0}^{m-1} q_j \cdot x^j\right) = \sum_{i,j} p_i q_j x^{i+j}.$$

Tento součin můžeme zapsat jako polynom $R(x)$, jehož koeficient u x^k je roven $r_k = p_0q_k + p_1q_{k-1} + \ldots + p_kq_0$. Nahlédneme, že polynom R má stupeň $\deg P + \deg Q$ a velikost $|P| + |Q| - 1$.

Algoritmus, který počítá součin dvou polynomů velikosti n přímo podle definice, proto spotřebuje čas $\Theta(n)$ na výpočet každého koeficientu, takže celkem $\Theta(n^2)$. Podobně jako u násobení čísel, i zde se budeme snažit najít efektivnější způsob.

Grafy polynomů

Odbočme na chvíli a uvažujme, kdy dva polynomy považujeme za stejné. Na to se dá nahlížet více způsoby. Buďto se na polynomy můžeme dívat jako na výrazy a porovnávat jejich symbolické zápisy. Pak jsou si dva polynomy rovny právě tehdy, mají-li po normalizaci stejné vektory koeficientů. Tehdy říkáme, že jsou *identické* a obvykle to značíme $P \equiv Q$.

Nebo můžeme porovnávat polynomy jako reálné funkce. Polynomy P a Q jsou si rovny ($P = Q$) právě tehdy, je-li $P(x) = Q(x)$ pro všechna $x \in \mathbb{R}$. Identicky rovné polynomy si jsou rovny i jako funkce, ale musí to platit i naopak? Následující věta ukáže, že ano, a že dokonce stačí rovnost pro konečný počet x.

Věta: Buďte P a Q polynomy stupně nejvýše d. Pokud platí $P(x_i) = Q(x_i)$ pro navzájem různá čísla $x_0, \ldots, x_d$, pak P a Q jsou identické.

Důkaz: Připomeňme nejprve následující standardní lemma o kořenech polynomů:

> **Lemma:** Polynom R stupně $t \geq 0$ má nejvýše t kořenů (čísel α, pro něž je $R(\alpha) = 0$).
>
> *Důkaz:* Pokud vydělíme polynom R polynomem $x - \alpha$ (viz cvičení 1), dostaneme $R(x) \equiv (x - \alpha) \cdot R'(x) + \beta$, kde β je konstanta. Je-li α kořenem R, musí být $\beta = 0$. Navíc polynom R' má stupeň $t - 1$ a stejné kořeny, jako měl polynom R, s možnou výjimkou kořene α.
>
> Budeme-li tento postup opakovat t-krát, buďto nám v průběhu dojdou kořeny (a pak lemma jistě platí), nebo dostaneme rovnost $R(x) \equiv (x - \alpha_1) \cdot \ldots \cdot (x - \alpha_t) \cdot R''(x)$, kde R'' je polynom nulového stupně. Takový polynom ovšem nemůže mít žádný kořen, a tím pádem nemůže mít žádné další kořeny ani R. □

Abychom dokázali větu, stačí uvážit polynom $R(x) \equiv P(x) - Q(x)$. Tento polynom má stupeň nejvýše d, ovšem každé z čísel $x_0, \ldots, x_d$ je jeho kořenem. Podle lemmatu musí tedy být identicky nulový, a proto $P \equiv Q$. □

Díky předchozí větě můžeme polynomy reprezentovat nejen vektorem koeficientů, ale také *vektorem funkčních hodnot* v nějakých smluvených bodech – tomuto vektoru budeme

říkat *graf polynomu.* Pokud zvolíme dostatečně mnoho bodů, je polynom svým grafem jednoznačně určen.

V této reprezentaci je násobení polynomů triviální: Součin polynomů P a Q má v bodě x hodnotu $P(x) \cdot Q(x)$. Stačí tedy grafy vynásobit po složkách, což zvládneme v lineárním čase. Jen je potřeba dát pozor na to, že součin má vyšší stupeň než jednotliví činitelé, takže musíme polynomy vyhodnocovat ve dvojnásobném počtu bodů.

Algoritmus NásobeníPolynomů

1. Jsou dány polynomy P a Q velikosti n, určené svými koeficienty. Bez újmy na obecnosti předpokládejme, že horních $n/2$ koeficientů je u obou polynomů nulových, takže součin $R \equiv P \cdot Q$ bude také polynom velikosti n.
2. Zvolíme navzájem různá čísla $x_0, \ldots, x_{n-1}$.
3. Spočítáme grafy polynomů P a Q, čili vektory $(P(x_0), \ldots, P(x_{n-1}))$ a $(Q(x_0), \ldots, Q(x_{n-1}))$.
4. Z toho vypočteme graf součinu R vynásobením po složkách: $R(x_i) = P(x_i) \cdot Q(x_i)$.
5. Nalezneme koeficienty polynomu R tak, aby odpovídaly grafu.

Krok 4 trvá $\Theta(n)$, takže rychlost celého algoritmu stojí a padá s efektivitou převodů mezi koeficientovou a hodnotovou reprezentací polynomů. To obecně neumíme v lepším než kvadratickém čase, ale zde máme možnost volby bodů $x_0, \ldots, x_{n-1}$, takže si je zvolíme tak šikovně, aby převod šel provést rychle.

Vyhodnocení polynomu metodou Rozděl a panuj

Nyní se pokusíme sestrojit algoritmus pro vyhodnocení polynomu založený na metodě Rozděl a panuj. Sice tento pokus nakonec selže, ale bude poučné podívat se, proč a jak selhal.

Uvažujme polynom P velikosti n, který chceme vyhodnotit v n bodech. Body si zvolíme tak, aby byly *spárované,* tedy aby tvořily dvojice lišící se pouze znaménkem:

$$\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1}.$$

Polynom P můžeme rozložit na členy se sudými exponenty a na ty s lichými:

$$P(x) = (p_0 x^0 + p_2 x^2 + \ldots + p_{n-2} x^{n-2}) + (p_1 x^1 + p_3 x^3 + \ldots + p_{n-1} x^{n-1}).$$

Navíc můžeme z druhé závorky vytknout x:

$$P(x) = (p_0 x^0 + p_2 x^2 + \ldots + p_{n-2} x^{n-2}) + x \cdot (p_1 x^0 + p_3 x^2 + \ldots + p_{n-1} x^{n-2}).$$

V obou závorkách se nyní vyskytují pouze sudé mocniny x. Proto můžeme každou závorku považovat za vyhodnocení nějakého polynomu velikosti $n/2$ v bodě x^2, tedy:

$$P(x) = P_s(x^2) + x \cdot P_\ell(x^2),$$

kde:

$$P_s(t) = p_0 t^0 + p_2 t^1 + \ldots + p_{n-2} t^{\frac{n-2}{2}},$$
$$P_\ell(t) = p_1 t^0 + p_3 t^1 + \ldots + p_{n-1} t^{\frac{n-2}{2}}.$$

Navíc pokud podobným způsobem dosadíme do P hodnotu $-x$, dostaneme:

$$P(-x) = P_s(x^2) - x \cdot P_\ell(x^2).$$

Vyhodnocení polynomu P v bodech $\pm x_0, \ldots, \pm x_{n/2-1}$ tedy můžeme převést na vyhodnocení polynomů P_s a P_ℓ poloviční velikosti v bodech $x_0^2, \ldots, x_{n/2-1}^2$.

To naznačuje algoritmus s časovou složitostí $T(n) = 2T(n/2) + \Theta(n)$ a z Kuchařkové věty víme, že taková rekurence má řešení $T(n) = \Theta(n \log n)$. Jenže ouvej, tento algoritmus nefunguje: druhé mocniny, které předáme rekurzivnímu volání, jsou vždy nezáporné, takže už nemohou být správně spárované. Tedy … alespoň dokud počítáme s reálnými čísly.

Ukážeme, že v oboru komplexních čísel už můžeme zvolit body, které budou správně spárované i po několikerém umocnění na druhou.

Cvičení

1. Odvoďte dělení polynomů se zbytkem: Jsou-li P a Q polynomy a $\deg Q > 0$, pak existují polynomy R a S takové, že $P \equiv QR + S$ a $\deg S < \deg Q$. Zkuste pro toto dělení nalézt co nejefektivnější algoritmus.

2. Převod grafu na polynom v obecném případě: Hledáme polynom stupně nejvýše n, který prochází body $(x_0, y_0), \ldots, (x_n, y_n)$ pro x_i navzájem různá. Pomůže *Lagrangeova interpolace:* definujme polynomy

$$A_j(x) = \prod_{k \neq j} (x - x_k), \quad B_j(x) = \frac{A_j(x)}{\prod_{k \neq j}(x_j - x_k)}, \quad P(x) = \sum_j y_j B_j(x).$$

   Dokažte, že $\deg P \leq n$ a $P(x_j) = y_j$ pro všechna j. K tomu pomůže rozmyslet si, jak vyjde $A_j(x_k)$ a $B_j(x_k)$.

3. Sestrojte co nejrychlejší algoritmus pro Lagrangeovu interpolaci z předchozího cvičení.

4. Jiný pohled na interpolaci polynomů: Hledáme-li polynom $P(x) = \sum_{k=0}^{n} p_k x^k$ procházející body $(x_0, y_0), \ldots, (x_n, y_n)$, řešíme vlastně soustavu rovnic tvaru $\sum_k p_k x_j^k = y_j$ pro $j = 0, \ldots, n$. Rovnice jsou lineární v neznámých $p_0, \ldots, p_n$, takže hledáme vektor $\mathbf{p}$ splňující $\mathbf{Vp} = \mathbf{y}$, kde $\mathbf{V}$ je takzvaná *Vandermondova matice* s $\mathbf{V}_{jk} = x_j^k$. Dokažte, že pro x_j navzájem různá je matice $\mathbf{V}$ regulární, takže soustava rovnic má právě jedno řešení.

5.* Pro odpálení jaderné bomby je potřeba, aby se na tom shodlo alespoň k z celkového počtu n generálů. Vymyslete, jak z odpalovacího kódu odvodit n klíčů pro generály tak, aby libovolná skupina k generálů uměla ze svých klíčů kód vypočítat, ale žádná menší skupina nemohla o kódu zjistit nic než jeho délku.

17.2 Intermezzo o komplexních číslech

V této kapitole budeme počítat s komplexními čísly. Zopakujme si proto, jak se s nimi zachází.

Základní operace

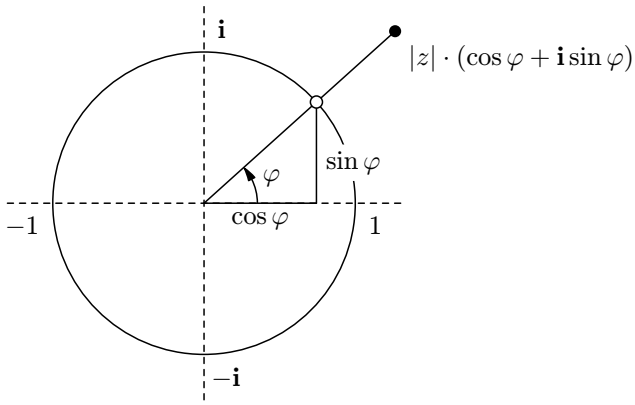
- Definice: $\mathbb{C} = \{a + b\mathbf{i} \mid a, b \in \mathbb{R}\}$, $\mathbf{i}^2 = -1$.
- Sčítání a odčítání: $(a + b\mathbf{i}) \pm (p + q\mathbf{i}) = (a \pm p) + (b \pm q)\mathbf{i}$.
- Násobení: $(a + b\mathbf{i})(p + q\mathbf{i}) = ap + aq\mathbf{i} + bp\mathbf{i} + bq\mathbf{i}^2 = (ap - bq) + (aq + bp)\mathbf{i}$. Pro $\alpha \in \mathbb{R}$ je $\alpha(a + b\mathbf{i}) = \alpha a + \alpha b\mathbf{i}$.
- Komplexní sdružení: $\overline{a + b\mathbf{i}} = a - b\mathbf{i}$.
  $\overline{\overline{x}} = x$, $\overline{x \pm y} = \overline{x} \pm \overline{y}$, $\overline{x \cdot y} = \overline{x} \cdot \overline{y}$, $x \cdot \overline{x} = (a + b\mathbf{i})(a - b\mathbf{i}) = a^2 + b^2 \in \mathbb{R}$.
- Absolutní hodnota: $|x| = \sqrt{x \cdot \overline{x}}$, takže $|a + b\mathbf{i}| = \sqrt{a^2 + b^2}$.
  Také $|\alpha x| = |\alpha| \cdot |x|$ pro $\alpha \in \mathbb{R}$ (později uvidíme, že to platí i pro α komplexní).
- Dělení: Podíl x/y rozšíříme číslem $\overline{y}$ na $(x \cdot \overline{y})/(y \cdot \overline{y})$. Nyní je jmenovatel reálný, takže můžeme vydělit každou složku čitatele zvlášť.

Gaussova rovina a goniometrický tvar[1]

- Komplexním číslům přiřadíme body v $\mathbb{R}^2$: $a + b\mathbf{i} \leftrightarrow (a, b)$.
- $|x|$ vyjadřuje vzdálenost od bodu $(0, 0)$.

[1] Říká se jí podle slavného německého matematika Carla Friedricha Gauße, jenž je známý spíš jako Carolus Fridericus Gauss – svá díla totiž stejně jako většina vědců 18. století psal latinsky.

- $|x| = 1$ pro čísla ležící na jednotkové kružnici (*komplexní jednotky*).
  Pak platí $x = \cos\varphi + \mathbf{i}\sin\varphi$ pro nějaké $\varphi \in [0, 2\pi)$.
- Pro libovolné $x \in \mathbb{C}$: $x = |x| \cdot (\cos\varphi(x) + \mathbf{i}\sin\varphi(x))$.
  Číslu $\varphi(x)$ říkáme *argument* čísla x a někdy ho také značíme $\arg x$. Argumenty jsou periodické s periodou 2π, často se normalizují do intervalu $[0, 2\pi)$.
- Navíc $\varphi(\overline{x}) = -\varphi(x)$, uvažujeme-li argumenty modulo 2π.



Obrázek 17.1: Goniometrický tvar komplexního čísla

Exponenciální tvar

- Eulerova formule: $\mathrm{e}^{\mathbf{i}\varphi} = \cos\varphi + \mathbf{i}\sin\varphi$.
- Každé $x \in \mathbb{C}$ lze tedy zapsat jako $|x| \cdot \mathrm{e}^{\mathbf{i}\varphi(x)}$.
- Násobení: $xy = \left(|x| \cdot \mathrm{e}^{\mathbf{i}\varphi(x)}\right) \cdot \left(|y| \cdot \mathrm{e}^{\mathbf{i}\varphi(y)}\right) = |x| \cdot |y| \cdot \mathrm{e}^{\mathbf{i}(\varphi(x)+\varphi(y))}$
  (absolutní hodnoty se násobí, argumenty sčítají).
- Umocňování: Pro $\alpha \in \mathbb{R}$ je $x^\alpha = \left(|x| \cdot \mathrm{e}^{\mathbf{i}\varphi(x)}\right)^\alpha = |x|^\alpha \cdot \mathrm{e}^{\mathbf{i}\alpha\varphi(x)}$.

Odmocniny z jedničky

Odmocňování v komplexních číslech není obecně jednoznačné: jestliže třeba budeme hledat čtvrtou odmocninu z jedničky, totiž řešit rovnici $x^4 = 1$, nalezneme hned čtyři řešení: 1, -1, $\mathbf{i}$ a $-\mathbf{i}$.

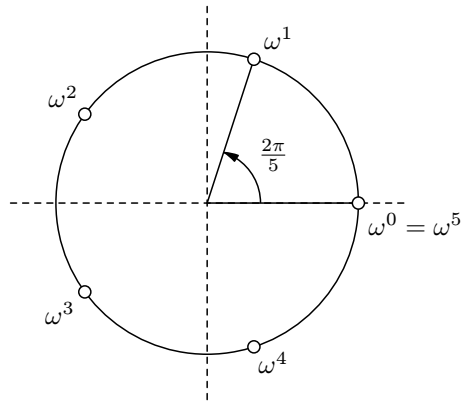
Prozkoumejme nyní obecněji, jak se chovají n-té odmocniny z jedničky, tedy komplexní kořeny rovnice $x^n = 1$:

- Jelikož $|x^n| = |x|^n$, musí být $|x| = 1$. Proto $x = \mathrm{e}^{\mathbf{i}\varphi}$ pro nějaké φ.
- Má platit $1 = x^n = \mathrm{e}^{\mathbf{i}\varphi n} = \cos\varphi n + \mathbf{i}\sin\varphi n$. To nastane, kdykoliv $\varphi n = 2k\pi$ pro nějaké $k \in \mathbb{Z}$.

Dostáváme tedy n různých n-tých odmocnin z 1, totiž $\mathrm{e}^{2k\pi\mathbf{i}/n}$ pro $k = 0, \ldots, n-1$. Některé z těchto odmocnin jsou ovšem speciální:

Definice: Komplexní číslo x je *primitivní* n-tá odmocnina z 1, pokud $x^n = 1$ a žádné z čísel $x^1, x^2, \ldots, x^{n-1}$ není rovno 1.

Příklad: Ze čtyř zmíněných čtvrtých odmocnin z 1 jsou $\mathbf{i}$ a $-\mathbf{i}$ primitivní a druhé dvě nikoliv (ověřte sami dosazením). Pro obecné $n > 2$ vždy existují alespoň dvě primitivní odmocniny, totiž čísla $\omega = \mathrm{e}^{2\pi\mathbf{i}/n}$ a $\overline{\omega} = \mathrm{e}^{-2\pi\mathbf{i}/n}$. Platí totiž, že $\omega^j = \mathrm{e}^{2\pi\mathbf{i}j/n}$, a to je rovno 1 právě tehdy, je-li j násobkem n (jednotlivé mocniny čísla ω postupně obíhají jednotkovou kružnici). Analogicky pro $\overline{\omega}$.



Obrázek 17.2: Primitivní pátá odmocnina z jedničky ω a její mocniny

Pozorování: Pro sudé n a libovolné číslo ω, které je primitivní n-tou odmocninou z jedničky, platí:

- $\omega^j \neq \omega^k$, kdykoliv $0 \leq j < k < n$. Stačí se podívat na podíl $\omega^k/\omega^j = \omega^{k-j}$. Ten nemůže být roven jedné, protože $0 < k - j < n$ a ω je primitivní.
- Pro sudé n je $\omega^{n/2} = -1$. Platí totiž $(\omega^{n/2})^2 = \omega^n = 1$, takže $\omega^{n/2}$ je druhá odmocnina z 1. Takové odmocniny jsou jenom dvě: 1 a -1, ovšem 1 to být nemůže, protože ω je primitivní.

Cvičení

1. Dokažte, že pro každou primitivní n-tou odmocninu z jedničky α platí, že $\alpha = \omega^t$, kde $\omega = \mathrm{e}^{2\pi\mathbf{i}/n}$ a t je přirozené číslo nesoudělné s n. Kolik primitivních n-tých odmocnin tedy existuje?

17.3 Rychlá Fourierova transformace

Ukážeme, že primitivních odmocnin lze využít k záchraně párovacího algoritmu na vyhodnocování polynomů z oddílu 17.1.

Nejprve polynomy doplníme nulami tak, aby jejich velikost n byla mocninou dvojky. Poté zvolíme nějakou primitivní n-tou odmocninu z jedničky ω a budeme polynom vyhodnocovat v bodech $\omega^0, \omega^1, \ldots, \omega^{n-1}$. To jsou navzájem různá komplexní čísla, která jsou správně spárovaná: hodnoty $\omega^{n/2}, \ldots, \omega^{n-1}$ se od $\omega^0, \ldots, \omega^{n/2-1}$ liší pouze znaménkem. To snadno ověříme: pro $0 \leq j < n/2$ je $\omega^{n/2+j} = \omega^{n/2}\omega^j = -\omega^j$. Navíc ω^2 je primitivní $(n/2)$-tá odmocnina z jedničky, takže se rekurzivně voláme na problém téhož druhu, který je správně spárovaný.

Náš plán použít metodu Rozděl a panuj tedy nakonec vyšel: opravdu máme algoritmus o složitosti $\Theta(n \log n)$ pro vyhodnocení polynomu. Ještě ho upravíme tak, aby místo s polynomy pracoval s vektory jejich koeficientů či hodnot. Tomuto algoritmu se říká FFT, vzápětí prozradíme, proč.

Algoritmus FFT (rychlá Fourierova transformace)
Vstup: Číslo $n = 2^k$, primitivní n-tá odmocnina z jedničky ω,
vektor $(p_0, \ldots, p_{n-1})$ koeficientů polynomu P

1. Pokud $n = 1$, položíme $y_0 \leftarrow p_0$ a skončíme.
2. Jinak se rekurzivně zavoláme na sudou a lichou část koeficientů:
3. $(s_0, \ldots, s_{n/2-1}) \leftarrow \mathrm{FFT}(n/2, \omega^2, (p_0, p_2, p_4, \ldots, p_{n-2}))$.
4. $(\ell_0, \ldots, \ell_{n/2-1}) \leftarrow \mathrm{FFT}(n/2, \omega^2, (p_1, p_3, p_5, \ldots, p_{n-1}))$.
5. Z grafů obou částí poskládáme graf celého polynomu:
6. Pro $j = 0, \ldots, n/2 - 1$:
7. $y_j \leftarrow s_j + \omega^j \cdot \ell_j$ ◁ *mocninu ω^j průběžně přepočítáváme*
8. $y_{j+n/2} \leftarrow s_j - \omega^j \cdot \ell_j$

Výstup: Graf polynomu P, tedy vektor $(y_0, \ldots, y_{n-1})$, kde $y_j = P(\omega^j)$

Vyhodnotit polynom v mocninách čísla ω umíme, ale ještě nejsme v cíli. Potřebujeme umět provést dostatečně rychle i opačný převod – z hodnot na koeficienty. K tomu nám pomůže podívat se na vyhodnocování polynomu trochu abstraktněji jako na zobrazení,

které jednomu vektoru komplexních čísel přiřadí jiný vektor. Toto zobrazení matematici potkávají v mnoha různých kontextech už po staletí a nazývají ho Fourierovou transformací.

Definice: *Diskrétní Fourierova transformace (DFT)* je zobrazení $\mathcal{F} : \mathbb{C}^n \to \mathbb{C}^n$, které vektoru $\mathbf{x}$ přiřadí vektor $\mathbf{y}$ daný přepisem

$$y_j = \sum_{k=0}^{n-1} x_k \cdot \omega^{jk},$$

kde ω je nějaká pevně zvolená primitivní n-tá odmocnina z jedné. Vektor $\mathbf{y}$ se nazývá *Fourierův obraz* vektoru $\mathbf{x}$.

Jak to souvisí s naším algoritmem? Pokud označíme $\mathbf{p}$ vektor koeficientů polynomu P, pak jeho Fourierova transformace $\mathcal{F}(\mathbf{p})$ není nic jiného než graf tohoto polynomu v bodech $\omega^0, \ldots, \omega^{n-1}$. To ověříme snadno dosazením do definice. Algoritmus tedy počítá diskrétní Fourierovu transformaci v čase $\Theta(n \log n)$. Proto se mu říká FFT – Fast Fourier Transform.

Také si všimněme, že DFT je lineární zobrazení. Jde proto zapsat jako násobení nějakou maticí $\mathbf{\Omega}$, kde $\mathbf{\Omega}_{jk} = \omega^{jk}$. Pro převod grafu na koeficienty potřebujeme najít inverzní zobrazení určené inverzní maticí $\mathbf{\Omega}^{-1}$.

Jelikož $\omega^{-1} = \overline{\omega}$, pojďme zkusit, zda hledanou inverzní maticí není $\overline{\mathbf{\Omega}}$.

Lemma: $\mathbf{\Omega} \cdot \overline{\mathbf{\Omega}} = n \cdot \mathbf{E}$, kde $\mathbf{E}$ je jednotková matice.

Důkaz: Dosazením do definice a elementárními úpravami:

$$\begin{aligned}(\mathbf{\Omega} \cdot \overline{\mathbf{\Omega}})_{jk} &= \sum_{\ell=0}^{n-1} \mathbf{\Omega}_{j\ell} \cdot \overline{\mathbf{\Omega}}_{\ell k} = \sum_{\ell=0}^{n-1} \omega^{j\ell} \cdot \overline{\omega^{\ell k}} = \sum_{\ell=0}^{n-1} \omega^{j\ell} \cdot \overline{\omega}^{\ell k} \\ &= \sum_{\ell=0}^{n-1} \omega^{j\ell} \cdot (\omega^{-1})^{\ell k} = \sum_{\ell=0}^{n-1} \omega^{j\ell} \cdot \omega^{-\ell k} = \sum_{\ell=0}^{n-1} \omega^{(j-k)\ell}.\end{aligned}$$

Poslední suma je ovšem geometrická řada. Pokud $j = k$, jsou všechny členy řady jedničky, takže se sečtou na n. Pro $j \neq k$ použijeme známý vztah pro součet geometrické řady s kvocientem $q = \omega^{j-k}$:

$$\sum_{\ell=0}^{n-1} q^\ell = \frac{q^n - 1}{q - 1} = \frac{\omega^{(j-k)n} - 1}{\omega^{j-k} - 1} = 0.$$

Poslední rovnost platí díky tomu, že $\omega^{(j-k)n} = (\omega^n)^{j-k} = 1^{j-k} = 1$, takže čitatel zlomku je nulový; naopak jmenovatel určitě nulový není, jelikož ω je primitivní a $0 < |j-k| < n$. □

Důsledek: $\boldsymbol{\Omega}^{-1} = (1/n) \cdot \overline{\boldsymbol{\Omega}}$.

Matice $\boldsymbol{\Omega}$ tedy je regulární a její inverze se kromě vydělení n liší pouze komplexním sdružením. Navíc číslo $\overline{\omega} = \omega^{-1}$ je také primitivní n-tou odmocninou z jedničky, takže až na faktor $1/n$ se jedná opět o Fourierovu transformaci a můžeme ji spočítat stejným algoritmem FFT. Shrňme, co jsme zjistili, do následujících vět:

Věta: Je-li n mocnina dvojky, lze v čase $\Theta(n \log n)$ spočítat diskrétní Fourierovu transformaci v $\mathbb{C}^n$ i její inverzi.

Věta: Polynomy velikosti n nad tělesem $\mathbb{C}$ lze násobit v čase $\Theta(n \log n)$.

Důkaz: Nejprve vektory koeficientů doplníme nulami, tak aby jejich délka byla současně mocnina dvojky a alespoň $2n$. Pak pomocí DFT v čase $\Theta(n \log n)$ převedeme oba polynomy na grafy, v $\Theta(n)$ vynásobíme grafy po složkách a výsledný graf pomocí inverzní DFT v čase $\Theta(n \log n)$ převedeme zpět na koeficienty polynomu. □

Fourierova transformace se kromě násobení polynomů hodí i na ledacos jiného. Své uplatnění nachází nejen v dalších algebraických algoritmech, ale také ve fyzikálních aplikacích – odpovídá totiž spektrálnímu rozkladu signálu na siny a cosiny o různých frekvencích. Na tom jsou založeny například algoritmy pro filtrování zvuku, pro kompresi zvuku a obrazu (MP3, JPEG), nebo třeba rozpoznávání řeči. Něco z toho naznačíme ve zbytku této kapitoly.

Cvičení

1. O jakých vlastnostech vektoru vypovídá nultý a $(n/2)$-tý koeficient jeho Fourierova obrazu?

2. Spočítejte Fourierovy obrazy následujících vektorů z $\mathbb{C}^n$:

   - $(x, \ldots, x)$
   - $(1, -1, 1, -1, \ldots, 1, -1)$
   - $(1, 0, 1, 0, 1, 0, 1, 0)$
   - $(\omega^0, \omega^1, \omega^2, \ldots, \omega^{n-1})$
   - $(\omega^0, \omega^2, \omega^4, \ldots, \omega^{2n-2})$

3. Inspirujte se předchozím cvičením a najděte pro každé j vektor, jehož Fourierův obraz má na j-tém místě jedničku a všude jinde nuly. Jak z toho přímo sestrojit inverzní transformaci?

4.* Co vypovídá o vektoru $(n/4)$-tý koeficient jeho Fourierova obrazu?

5. Mějme vektor $\mathbf{y}$, který vznikl rotací vektoru $\mathbf{x}$ o k pozic ($\mathbf{y}_j = \mathbf{x}_{(j+k) \bmod n}$). Jak spolu souvisí $\mathcal{F}(\mathbf{x})$ a $\mathcal{F}(\mathbf{y})$?

6. *Fourierova báze:* Uvažujme systém vektorů $\mathbf{b}^0, \ldots, \mathbf{b}^{n-1}$ se složkami $\mathbf{b}^j_k = \overline{\omega}^{jk}/\sqrt{n}$. Dokažte, že tyto vektory tvoří ortonormální bázi prostoru $\mathbb{C}^n$, pokud použijeme standardní skalární součin nad $\mathbb{C}$: $\langle \mathbf{x}, \mathbf{y} \rangle = \sum_j \mathbf{x}_j \overline{\mathbf{y}_j}$. Složky vektoru $\mathbf{x}$ vzhledem k této bázi pak jsou $\langle \mathbf{x}, \mathbf{b}^0 \rangle, \ldots, \langle \mathbf{x}, \mathbf{b}^{n-1} \rangle$ (to platí pro libovolnou ortonormální bázi). Uvědomte si, že tyto skalární součiny odpovídají definici DFT, tedy až na konstantu $1/\sqrt{n}$.

7.* *Volba ω:* Ve Fourierově transformaci máme volnost v tom, jakou primitivní odmocninu ω si vybereme. Ukažte, že Fourierovy obrazy pro různé volby ω se liší pouze pořadím složek.

17.4* Spektrální rozklad

Ukážeme, jak FFT souvisí s digitálním zpracováním signálu – pro jednoduchost jednorozměrného, tedy třeba zvuku.

Uvažujme reálnou funkci f definovanou na intervalu $[0, 1)$. Pokud její hodnoty *navzorkujeme* v n pravidelně rozmístěných bodech, získáme vektor $\mathbf{f} \in \mathbb{R}^n$ o složkách $\mathbf{f}_j = f(j/n)$. Co o funkci f vypovídá Fourierův obraz vektoru $\mathbf{f}$?

Lemma R (DFT reálného vektoru): Je-li $\mathbf{x}$ reálný vektor z $\mathbb{R}^n$, jeho Fourierův obraz $\mathbf{y} = \mathcal{F}(\mathbf{x})$ je *antisymetrický:* $\mathbf{y}_j = \overline{\mathbf{y}_{n-j}}$ pro všechna j. (Připomínáme, že vektory v této kapitole indexujeme modulo n, takže $\mathbf{y}_n = \mathbf{y}_0$.)

Důkaz: Z definice DFT víme, že

$$\mathbf{y}_{n-j} = \sum_k \mathbf{x}_k \omega^{(n-j)k} = \sum_k \mathbf{x}_k \omega^{nk-jk} = \sum_k \mathbf{x}_k \omega^{-jk} = \sum_k \mathbf{x}_k \overline{\omega}^{jk}.$$

Jelikož komplexní sdružení lze distribuovat přes aritmetické operace, můžeme psát $\overline{\mathbf{y}_{n-j}} = \sum_k \overline{\mathbf{x}_k} \cdot \omega^{jk}$, což je pro reálné $\mathbf{x}$ rovno $\sum_k \mathbf{x}_k \omega^{jk} = \mathbf{y}_j$. □

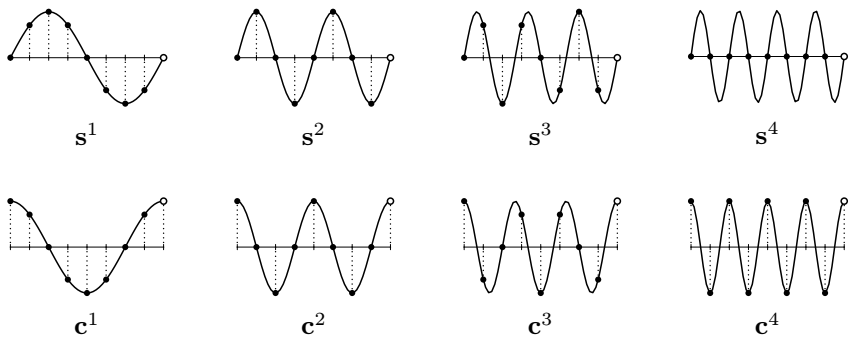
Důsledek: Speciálně $\mathbf{y}_0 = \overline{\mathbf{y}_0}$ a $\mathbf{y}_{n/2} = \overline{\mathbf{y}_{n/2}}$, takže obě tyto hodnoty jsou reálné.

Lemma A (o antisymetrických vektorech): Antisymetrické vektory v $\mathbb{C}^n$ tvoří vektorový prostor dimenze n nad tělesem reálných čísel.

Důkaz: Ověříme axiomy vektorového prostoru. (To, že prostor budujeme nad $\mathbb{R}$, a nikoliv nad $\mathbb{C}$, je důležité: násobení vektoru komplexním skalárem obecně nezachovává antisymetrii.)

Co se dimenze týče: V antisymetrickém vektoru $\mathbf{y}$ jsou složky $\mathbf{y}_0$ a $\mathbf{y}_{n/2}$ reálné, u složek $\mathbf{y}_1, \ldots, \mathbf{y}_{n/2-1}$ můžeme volit jak reálnou, tak imaginární část. Ostatní složky tím jsou už jednoznačně dány. Vektor je tedy určen n nezávislými reálnými parametry. □

Definice: V dalším textu zvolme pevné n a $\omega = \mathrm{e}^{2\pi\mathbf{i}/n}$. Označíme $\mathbf{e}^k$, $\mathbf{s}^k$ a $\mathbf{c}^k$ vektory získané navzorkováním funkcí $\mathrm{e}^{2k\pi\mathbf{i}x}$, $\sin 2k\pi x$ a $\cos 2k\pi x$ (komplexní exponenciála, sinus a cosinus s frekvencí k) v n bodech intervalu $[0, 1)$.



Obrázek 17.3: Vektory $\mathbf{s}^k$ a $\mathbf{c}^k$ při vzorkování v 8 bodech

Lemma V (o vzorkování funkcí): Fourierův obraz vektorů $\mathbf{e}^k$, $\mathbf{s}^k$ a $\mathbf{c}^k$ vypadá pro $0 < k < n/2$ následovně:

$$\begin{aligned}
\mathcal{F}(\mathbf{e}^k) &= (0, \ldots, 0, n, 0, \ldots, 0),\\
\mathcal{F}(\mathbf{s}^k) &= (0, \ldots, 0, n/2\mathbf{i}, 0, \ldots, 0, -n/2\mathbf{i}, 0, \ldots, 0),\\
\mathcal{F}(\mathbf{c}^k) &= (0, \ldots, 0, n/2, 0, \ldots, 0, n/2, 0, \ldots, 0),
\end{aligned}$$

přičemž první vektor má nenulu na pozici $n - k$, další dva na pozicích k a $n - k$.

Zatímco vztah pro $\mathcal{F}(\mathbf{e}^k)$ funguje i s $k = 0$ a $k = n/2$, siny a cosiny se chovají odlišně: $\mathbf{s}^0$ i $\mathbf{s}^{n/2}$ jsou nulové vektory, takže $\mathcal{F}(\mathbf{s}^0)$ a $\mathcal{F}(\mathbf{s}^{n/2})$ jsou také nulové; $\mathbf{c}^0$ je vektor samých jedniček s $\mathcal{F}(\mathbf{c}^0) = (n, 0, \ldots, 0)$ a $\mathbf{c}^{n/2} = (1, -1, \ldots, 1, -1)$ s $\mathcal{F}(\mathbf{c}^{n/2}) = (0, \ldots, 0, n, 0, \ldots 0)$ s n na pozici $n/2$.

Důkaz: Pro $\mathbf{e}^k$ si stačí všimnout, že $\mathbf{e}^k_j = \mathrm{e}^{2k\pi\mathbf{i}\cdot j/n} = \mathrm{e}^{jk\cdot 2\pi\mathbf{i}/n} = \omega^{jk}$. Proto t-tá složka Fourierova obrazu vyjde $\sum_j \omega^{jk}\omega^{jt} = \sum_j \omega^{j(k+t)}$. To je opět geometrická řada, podobně jako u odvození inverzní FT. Pro $t = n - k$ se sečte na n, všude jinde na 0. Vektory $\mathbf{s}^k$ a $\mathbf{c}^k$ necháváme jako cvičení. □

Všimněme si nyní, že reálnou lineární kombinací vektorů $\mathcal{F}(\mathbf{s}^1)$, …, $\mathcal{F}(\mathbf{s}^{n/2-1})$ a $\mathcal{F}(\mathbf{c}^0)$, …, $\mathcal{F}(\mathbf{c}^{n/2})$ můžeme získat libovolný antisymetrický vektor. Jelikož DFT je lineární, plyne z toho, že lineární kombinací $\mathbf{s}^1, \ldots, \mathbf{s}^{n/2-1}$ a $\mathbf{c}^0, \ldots, \mathbf{c}^{n/2}$ lze získat libovolný reálný vektor. Přesněji to říká následující věta:

Věta: Pro každý vektor $\mathbf{x} \in \mathbb{R}^n$ existují reálné koeficienty $\alpha_0, \ldots, \alpha_{n/2}$ a $\beta_0, \ldots, \beta_{n/2}$ takové, že:

$$\mathbf{x} = \sum_{k=0}^{n/2} \left(\alpha_k \mathbf{c}^k + \beta_k \mathbf{s}^k\right). \qquad (*)$$

Tyto koeficienty jdou navíc vypočíst z Fourierova obrazu

$$\mathbf{y} = \mathcal{F}(\mathbf{x}) = (a_0 + b_0\mathbf{i}, \ldots, a_{n-1} + b_{n-1}\mathbf{i})$$

takto:

$$\begin{aligned}
&\alpha_0 = a_0/n,\\
&\alpha_j = 2a_j/n \quad \text{pro } j = 1, \ldots, n/2,\\
&\beta_0 = \beta_{n/2} = 0,\\
&\beta_j = -2b_j/n \quad \text{pro } j = 1, \ldots, n/2-1.
\end{aligned}$$

Důkaz: Jelikož DFT má inverzi, můžeme bez obav fourierovat obě strany rovnice $(*)$. Tedy chceme, aby platilo $\mathbf{y} = \mathcal{F}(\sum_k \alpha_k \mathbf{s}^k + \beta_k \mathbf{c}^k)$. Suma na pravé straně je přitom díky linearitě $\mathcal{F}$ rovna $\sum_k (\alpha_k \mathcal{F}(\mathbf{s}^k) + \beta_k \mathcal{F}(\mathbf{c}^k))$. Označme tento vektor $\mathbf{z}$ a za vydatné pomoci lemmatu **V** vypočítejme jeho složky:

- K $\mathbf{z}_0$ přispívá pouze $\mathbf{c}^0$ (ostatní $\mathbf{s}^k$ a $\mathbf{c}^k$ mají nultou složku nulovou). Takže $\mathbf{z}_0 = \alpha_0 \mathbf{c}^0_0 = (a_0/n) \cdot n = a_0$.
- K $\mathbf{z}_j$ pro $j = 1, \ldots, n/2-1$ přispívají pouze $\mathbf{c}^j$ a $\mathbf{s}^j$: $\mathbf{z}_j = \alpha_j \mathbf{c}^j_j + \beta_j \mathbf{s}^j_j = 2a_j/n \cdot n/2 - 2b_j/n \cdot n/2\mathbf{i} = a_j + b_j\mathbf{i}$.
- K $\mathbf{z}_{n/2}$ přispívá pouze $\mathbf{c}^{n/2}$, takže analogicky vyjde $\mathbf{z}_{n/2} = 2a_{n/2}/n \cdot n/2 = a_{n/2}$.

Vektory $\mathbf{z}$ a $\mathbf{y}$ se tedy shodují v prvních $n/2+1$ složkách (nezapomeňte, že $b_0 = b_{n/2} = 0$). Jelikož jsou oba antisymetrické, musí se shodovat i ve zbývajících složkách. □

Důsledek: Pro libovolnou reálnou funkci f na intervalu $[0, 1)$ existuje lineární kombinace funkcí $\sin 2k\pi x$ a $\cos 2k\pi x$ pro $k = 0, \ldots, n/2$, která není při vzorkování v n bodech od dané funkce f rozlišitelná.

To je diskrétní ekvivalent známého tvrzení o spojité Fourierově transformaci, podle nějž každou „dostatečně hladkou“ periodickou funkci lze lineárně nakombinovat ze sinů a cosinů o celočíselných frekvencích.

To se hodí například při zpracování zvuku: jelikož $\alpha\cos x + \beta\sin x = A\sin(x+\varphi)$ pro vhodné A a φ, můžeme kterýkoliv zvuk rozložit na sinusové tóny o různých frekvencích. U každého tónu získáme jeho amplitudu A a fázový posun φ, což je vlastně (až na nějaký násobek n) absolutní hodnota a argument původního komplexního Fourierova koeficientu. Tomu se říká *spektrální rozklad* signálu a díky FFT ho můžeme z navzorkovaného signálu spočítat velmi rychle.

Cvičení

1. Dokažte „inverzní“ lemma **R**: DFT antisymetrického vektoru je vždy reálná.

2. Dokažte zbytek lemmatu **V**: Jak vypadá $\mathcal{F}(\mathbf{s}^k)$ a $\mathcal{F}(\mathbf{c}^k)$?

3.* Analogií DFT pro reálné vektory je *diskrétní cosinová transformace (DCT)*. Z DFT v $\mathbb{C}^n$ odvodíme DCT v $\mathbb{R}^{n/2+1}$. Vektor $(x_0,\ldots,x_{n/2})$ doplníme jeho zrcadlovou kopií na $\mathbf{x} = (x_0, x_1, \ldots, x_{n/2}, x_{n/2-1}, \ldots, x_1)$. To je reálný a antisymetrický vektor, takže jeho Fourierův obraz $\mathbf{y} = \mathcal{F}(\mathbf{x})$ musí být podle lemmatu **R** a cvičení 1 také reálný a antisymetrický: $\mathbf{y} = (y_0, y_1, \ldots, y_{n/2}, y_{n/2-1}, \ldots, y_1)$. Vektor $(y_0,\ldots,y_{n/2})$ prohlásíme za výsledek DCT.

   Rozepsáním $\mathcal{F}^{-1}(\mathbf{y})$ podle definice dokažte, že tento výsledek popisuje, jak zapsat vektor $\mathbf{x}$ jako lineární kombinaci cosinových vektorů $\mathbf{c}^0,\ldots,\mathbf{c}^{n/2}$. Oproti DFT tedy používáme pouze cosiny, zato však o dvojnásobném rozsahu frekvencí.

4. *Konvoluce* vektorů $\mathbf{x}$ a $\mathbf{y}$ je vektor $\mathbf{z} = \mathbf{x} * \mathbf{y}$ takový, že $\mathbf{z}_j = \sum_k \mathbf{x}_k \mathbf{y}_{j-k}$, přičemž indexujeme modulo n. Tuto sumu si můžeme představit jako skalární součin vektoru $\mathbf{x}$ s vektorem $\mathbf{y}$ napsaným pozpátku a zrotovaným o j pozic. Konvoluce nám tedy řekne, jak tyto „přetočené skalární součiny“ vypadají pro všechna j. Dokažte následující vlastnosti:

   a) $\mathbf{x} * \mathbf{y} = \mathbf{y} * \mathbf{x}$ *(komutativita)*
   b) $\mathbf{x} * (\mathbf{y} * \mathbf{z}) = (\mathbf{x} * \mathbf{y}) * \mathbf{z}$ *(asociativita)*
   c) $\mathbf{x} * (\alpha\mathbf{y} + \beta\mathbf{z}) = \alpha(\mathbf{x} * \mathbf{y}) + \beta(\mathbf{x} * \mathbf{z})$ *(bilinearita)*
   d) $\mathcal{F}(\mathbf{x} * \mathbf{y}) = \mathcal{F}(\mathbf{x}) \odot \mathcal{F}(\mathbf{y})$, kde $\odot$ je součin vektorů po složkách. To nám dává algoritmus pro výpočet konvoluce v čase $\Theta(n \log n)$.

5. *Vyhlazování signálu:* Mějme vektor $\mathbf{x}$ naměřených dat. Obvyklý způsob, jak je vyčistit od šumu, je transformace typu $\mathbf{y}_j = \frac{1}{4}\mathbf{x}_{j-1} + \frac{1}{2}\mathbf{x}_j + \frac{1}{4}\mathbf{x}_{j+1}$. Ta „obrousí špičky“ tím, že každou hodnotu zprůměruje s okolními. Pokud budeme $\mathbf{x}$ indexovat cyklicky, jedná se o konvoluci $\mathbf{x} * \mathbf{z}$, kde $\mathbf{z}$ je maska tvaru $(\frac{1}{2}, \frac{1}{4}, 0, \ldots, 0, \frac{1}{4})$.

Fourierův obraz $\mathcal{F}(\mathbf{z})$ nám říká, jak vyhlazování ovlivňuje spektrum. Například pro $n = 8$ vyjde $\mathcal{F}(\mathbf{z}) \approx (1, 0.854, 0.5, 0.146, 0, 0.146, 0.5, 0.854)$, takže stejnosměrná složka signálu $\mathbf{c}^0$ zůstane nezměněna, naopak nejvyšší frekvence $\mathbf{c}^4$ zcela zmizí a pro ostatní $\mathbf{c}^k$ a $\mathbf{s}^k$ platí, že čím vyšší frekvence, tím víc je tlumena. Tomu se říká *dolní propust* – nízké frekvence propustí, vysoké omezuje. Jak vypadá propust, která vynuluje $\mathbf{c}^4$ a ostatní frekvence propustí beze změny?

6. Zpět k polynomům: Uvědomte si, že to, co se děje při násobení polynomů s jejich koeficienty, je také konvoluce. Jen musíme doplnit vektory nulami, aby se neprojevila cykličnost indexování. Takže vztah $\mathcal{F}(\mathbf{x} * \mathbf{y}) = \mathcal{F}(\mathbf{x}) \odot \mathcal{F}(\mathbf{y})$ z cvičení 4 je jenom jiný zápis našeho algoritmu na rychlé násobení polynomů.

7.** *Diagonalizace:* Mějme vektorový prostor dimenze n a lineární zobrazení f v něm. Zvolíme-li nějakou bázi, můžeme vektory zapisovat jako n-tice čísel a zobrazení f popsat jako násobení n-tice vhodnou maticí $\mathbf{A}$ tvaru $n \times n$. Někdy se povede najít bázi z vlastních vektorů, vzhledem k níž je matice $\mathbf{A}$ *diagonální* – má nenulová čísla pouze na diagonále. Tehdy umíme součin $\mathbf{Ax}$ spočítat v čase $\Theta(n)$. Pro jeden součin se to málokdy vyplatí, protože převod mezi bázemi bývá pomalý, ale pokud jich chceme počítat hodně, pomůže to.

   Rozmyslete si, že podobně se můžeme dívat na DFT. Konvoluce je bilineární funkce na $\mathbb{C}^n$, což znamená, že je lineární v každém parametru zvlášť. Zvolíme-li bázi, můžeme každou bilineární funkci popsat trojrozměrnou tabulkou $n \times n \times n$ čísel (to už není matice, ale *tenzor třetího řádu*). Vztah $\mathcal{F}(\mathbf{x} * \mathbf{y}) = \mathcal{F}(\mathbf{x}) \odot \mathcal{F}(\mathbf{y})$ pak můžeme vyložit takto: $\mathcal{F}$ převádí vektory z kanonické báze do *Fourierovy báze* (viz cvičení 17.3.6), vzhledem k níž je tenzor konvoluce diagonální (má jedničky na „tělesové úhlopříčce“ a všude jinde nuly). Pomocí FFT pak můžeme mezi bázemi převádět v čase $\Theta(n \log n)$.

8.* Při zpracování obrazu se hodí *dvojrozměrná DFT,* která matici $\mathbf{X} \in \mathbb{C}^{n \times n}$ přiřadí matici $\mathbf{Y} \in \mathbb{C}^{n \times n}$ takto (ω je opět primitivní n-tá odmocnina z jedné):

$$\mathbf{Y}_{jk} = \sum_{u,v} \mathbf{X}_{uv} \omega^{ju+kv}.$$

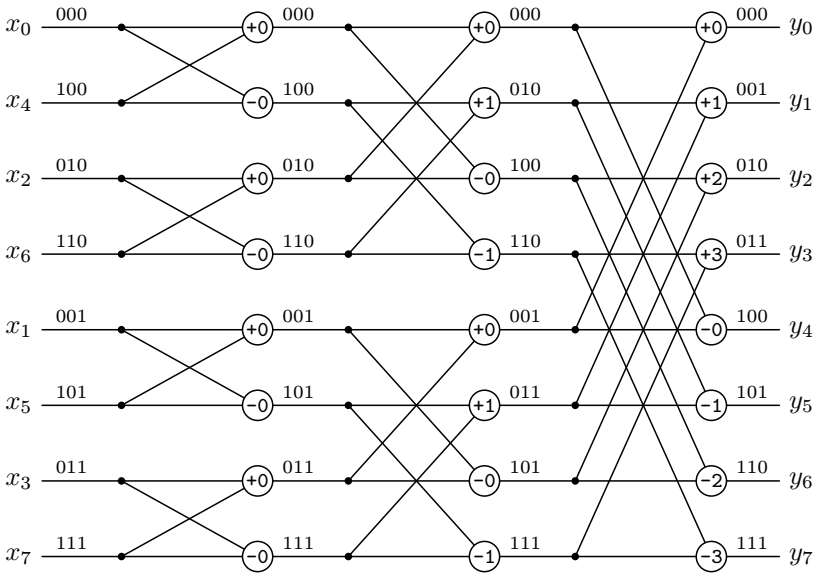
   Ověřte, že i tato transformace je bijekce, a odvoďte algoritmus na její efektivní výpočet pomocí jednorozměrné FFT. Fyzikální interpretace je podobná: Fourierův obraz popisuje rozklad matice na „prostorové frekvence“. Také lze odvodit dvojrozměrnou cosinovou transformaci, na níž je založený například kompresní algoritmus JPEG.

17.5* Další varianty FFT

FFT jako hradlová síť

Zkusme průběh algoritmu FFT znázornit graficky. Na levé straně obrázku 17.4 se nachází vstupní vektor $x_0, \ldots, x_{n-1}$ (v nějakém pořadí), na pravé straně pak výstupní vektor $y_0, \ldots, y_{n-1}$. Sledujme chod algoritmu pozpátku: Výstup spočítáme z výsledků „poloviční“ transformací vektorů $x_0, x_2, \ldots, x_{n-2}$ a $x_1, x_3, \ldots, x_{n-1}$. Kroužky přitom odpovídají výpočtu lineární kombinace $a + \omega^k b$, kde a, b jsou vstupy kroužku (a přichází rovně, b šikmo) a k číslo uvnitř kroužku. Každá z poloviční transformací se počítá analogicky z výsledků transformace velikosti $n/4$ atd. Celkově výpočet probíhá v $\log_2 n$ vrstvách po $\Theta(n)$ operacích.

Čísla nad čarami prozrazují, jak jsme došli k permutaci vstupních hodnot nalevo. Ke každému podproblému jsme napsali ve dvojkové soustavě indexy prvků vstupu, ze kterých podproblém počítáme. V posledním sloupci je to celý vstup. V předposledním nejprve indexy končící 0, pak ty končící 1. O sloupec vlevo jsme každou podrozdělili podle předposlední číslice atd. Až v prvním sloupci jsou čísla uspořádaná podle dvojkových zápisů čtených pozpátku. (Rozmyslete si, proč tomu tak je.)



Obrázek 17.4: Průběh FFT pro vstup velikosti 8

Na obrázek se také můžeme dívat jako na schéma hradlové sítě pro výpočet DFT. Kroužky jsou přitom „hradla“ pracující s komplexními čísly. Všechny operace v jedné vrstvě jsou na sobě nezávislé, takže je síť počítá paralelně. Síť tedy pracuje v čase $\Theta(\log n)$ a prostoru $\Theta(n \log n)$.

Nerekurzivní FFT

Obvod z obrázku 17.4 můžeme vyhodnocovat po hladinách zleva doprava. Tím získáme elegantní nerekurzivní algoritmus pro výpočet FFT. Pracuje v čase $\Theta(n \log n)$ a prostoru $\Theta(n)$:

Algoritmus FFT2 (rychlá Fourierova transformace nerekurzivně)
Vstup: Komplexní čísla $x_0, \ldots, x_{n-1}$, primitivní n-tá odmocnina z jedné ω

1. Předpočítáme tabulku hodnot $\omega^0, \omega^1, \ldots, \omega^{n-1}$.
2. Pro $k = 0, \ldots, n-1$ položíme $y_k \leftarrow x_{r(k)}$, kde r je funkce bitového zrcadlení.
3. $b \leftarrow 1$ ◁ *velikost bloku*
4. Dokud $b < n$, opakujeme:
5. Pro $j = 0, \ldots, n-1$ s krokem $2b$ opakujeme: ◁ *začátek bloku*
6. Pro $k = 0, \ldots, b-1$ opakujeme: ◁ *pozice v bloku*
7. $\alpha \leftarrow \omega^{(nk/2b) \bmod n}$
8. $(y_{j+k}, y_{j+k+b}) \leftarrow (y_{j+k} + \alpha \cdot y_{j+k+b}, y_{j+k} - \alpha \cdot y_{j+k+b})$
9. $b \leftarrow 2b$

Výstup: $y_0, \ldots, y_{n-1}$

FFT v konečných tělesech

Nakonec dodejme, že Fourierovu transformaci lze zavést nejen nad tělesem komplexních čísel, ale i v některých konečných tělesech, pokud zaručíme existenci primitivní n-té odmocniny z jedničky. Například v tělese $\mathbb{Z}_p$ pro prvočíslo p tvaru $2^k + 1$ platí $2^k = -1$. Proto $2^{2k} = 1$ a $2^0, 2^1, \ldots, 2^{2k-1}$ jsou navzájem různé. Číslo 2 je tedy primitivní $2k$-tá odmocnina z jedné. To se nám ovšem nehodí pro algoritmus FFT, neboť $2k$ bude málokdy mocnina dvojky.

Zachrání nás ovšem algebraická věta, která říká, že multiplikativní grupa[⟨2⟩] libovolného konečného tělesa $\mathbb{Z}_p$ je cyklická, tedy že všech $p-1$ nenulových prvků tělesa lze zapsat jako mocniny nějakého čísla g (generátoru grupy). Jelikož mezi čísly $g^0, g^1, g^2, \ldots, g^{p-2}$ se každý nenulový prvek tělesa vyskytne právě jednou, je g primitivní $(p-1)$-ní odmocninou z jedničky.

[⟨2⟩] To je množina všech nenulových prvků tělesa s operací násobení.

V praxi se hodí například tyto hodnoty:

- $p = 2^{16} + 1 = 65\,537$, $g = 3$, takže funguje $\omega = 3$ pro $n = 2^{16}$ (analogicky $\omega = 3^2$ pro $n = 2^{15}$ atd.),
- $p = 15 \cdot 2^{27} + 1 = 2\,013\,265\,921$, $g = 31$, takže pro $n = 2^{27}$ dostaneme $\omega = g^{15} \bmod p = 440\,564\,289$.
- $p = 3 \cdot 2^{30} + 1 = 3\,221\,225\,473$, $g = 5$, takže pro $n = 2^{30}$ vyjde $\omega = g^3 \bmod p = 125$.

Bližší průzkum našich úvah o FFT dokonce odhalí, že není ani potřeba těleso. Postačí libovolný komutativní okruh, ve kterém existuje příslušná primitivní odmocnina z jedničky, její multiplikativní inverze (ta ovšem existuje vždy, protože $\omega^{-1} = \omega^{n-1}$) a multiplikativní inverze čísla n. To nám poskytuje ještě daleko více volnosti než tělesa, ale není snadné takové okruhy hledat.

Výhodou těchto podob Fourierovy transformace je, že na rozdíl od té klasické komplexní nejsou zatíženy zaokrouhlovacími chybami (komplexní odmocniny z jedničky mají obě složky iracionální). To se hodí například v algoritmech na násobení velkých čísel – viz cvičení 2.

Cvičení

1. Navrhněte, jak počítat bitové zrcadlení v algoritmu FFT2.

2.* Pomocí FFT lze rychle násobit čísla. Každé n-bitové číslo x můžeme rozložit na k-bitové bloky $x_0, \ldots, x_{m-1}$ (kde $m = \lceil n/k \rceil$). To je totéž, jako kdybychom ho zapsali v soustavě o základu $B = 2^k$: $x = \sum_j x_j B^j$. Pokud k číslu přiřadíme polynom $X(t) = \sum_j x_j t^j$, bude $X(B) = x$.

   To nám umožňuje převést násobení čísel na násobení polynomů: Chceme-li vynásobit čísla x a y, sestrojíme polynomy X a Y, pomocí FFT vypočteme jejich součin Z a pak do něj dosadíme B. Dostaneme $Z(B) = X(B) \cdot Y(B) = xy$.

   Na RAMu přitom můžeme zvolit $k = \Theta(\log n)$, takže s čísly polynomiálně velkými vzhledem k B zvládneme počítat v konstantním čase. Proto FFT ve vhodném konečném tělese poběží v čase $\mathcal{O}(m \log m) = \mathcal{O}(n/\log n \cdot \log(n/\log n)) \subseteq \mathcal{O}(n/\log n \cdot \log n) = \mathcal{O}(n)$. Zbývá domyslet, jak vyhodnotit $Z(B)$. Spočítejte, jak velké jsou koeficienty polynomu Z, a ukažte, že při vyhodnocování od nejnižších řádů jsou přenosy dostatečně malé na to, abychom výpočet $Z(B)$ stihli v čase $\Theta(n)$.

   Tím jsme získali algoritmus na násobení n-bitových čísel v čase $\Theta(n)$.

18 Pokročilé haldy

18 Pokročilé haldy

Při analýze Dijkstrova algoritmu v oddílu 6.2 jsme zatoužili po haldě, která by některé operace uměla rychleji než obyčejná binární halda. V této kapitole postupně odvodíme tři datové struktury: binomiální haldu, línou binomiální haldu a Fibonacciho haldu. Poslední z nich bude mít vytoužené vlastnosti.

18.1 Binomiální haldy

Základní funkce binomiální haldy jsou podobné binární haldě, nicméně jich dosahuje jinými metodami. Navíc podporuje operaci Merge, která umí rychle sloučit dvě binomiální haldy do jedné.

Shrňme na začátek podporované operace spolu s jejich worst-case časovými složitostmi (tedy složitostmi v nejhorším případě). Číslo n udává počet prvků v haldě a haldu zde chápeme jako minimovou.

operace	*složitost*	*činnost*
Insert	$\Theta(\log n)$	vloží nový prvek
Min	$\Theta(1)$	vrátí minimum množiny
ExtractMin	$\Theta(\log n)$	vrátí a odstraní minimum množiny
Merge	$\Theta(\log n)$	sloučí dvě haldy do jedné
Build	$\Theta(n)$	postaví z n prvků haldu
Decrease	$\Theta(\log n)$	sníží hodnotu klíče prvku
Increase	$\Theta(\log n)$	zvýší hodnotu klíče prvku
Delete	$\Theta(\log n)$	smaže prvek

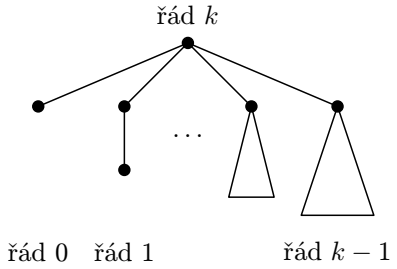
Nadále platí, že podle klíče neumíme vyhledávat, takže operace Decrease, Increase a Delete musí dostat ukazatel na prvek v haldě, nikoliv jeho klíč.

Nyní binomiální haldu definujeme. Na rozdíl od binární haldy nebude mít tvar stromu, nýbrž souboru více tzv. binomiálních stromů.

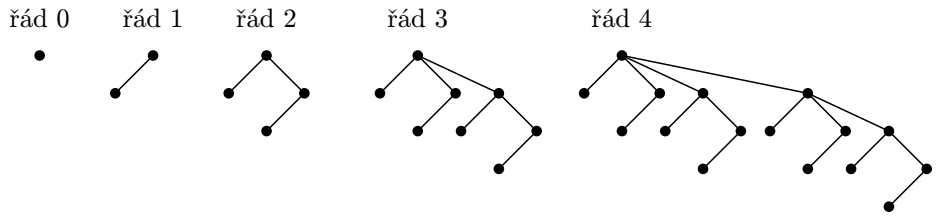
Binomiální stromy

Definice: Řekneme, že zakořeněný strom T je *binomiálním stromem řádu k*, pokud splňuje následující pravidla:

1. Pokud je řád k roven nule, pak T obsahuje pouze kořen.
2. Pokud je řád k nenulový, pak T má kořen s právě k syny. Tito synové jsou kořeny podstromů, které jsou po řadě binomiálními stromy řádů $0, \ldots, k-1$.



Obrázek 18.1: Binomiální strom řádu k

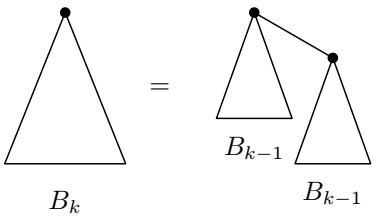


Obrázek 18.2: Příklady binomiálních stromů

Náhled na strukturu binomiálního stromu získáme z obrázku 18.1. Také se podívejme na obrázek 18.2, jak budou vypadat některé nejmenší binomiální stromy.

Podáme nyní tzv. rekurzivní definici binomiálních stromů, pro níž následně ukážeme ekvivalenci s předchozí definicí.

Definice: Zakořeněné stromy B_k jsou definovány takto: B_0 obsahuje pouze kořen, B_k pro $k > 0$ se skládá ze stromu B_{k-1}, pod jehož kořenem je napojený další strom B_{k-1}.



Obrázek 18.3: Rekurzivní definice binomiálního stromu

Lemma R (o rekurzivní definici): Strom B_k je binomiální strom řádu k. Každý strom T, který je binomiální strom řádu k, je izomorfní stromu B_k.

Důkaz: Postupujme matematickou indukcí. Pro $k = 0$ tvrzení zjevně platí. Nechť nyní $k > 0$. Pod kořenem stromu T jsou dle definice zavěšeny binomiální stromy řádů $0, \ldots, k-1$. Odtržením posledního binomiálního podstromu S řádu $k-1$ a použitím indukčního předpokladu dostáváme z T binomiální strom řádu $k-1$, tedy B_{k-1}. Opětovným připojením S zpět dostáváme přesně strom B_k.

Naopak, uvážíme-li strom B_k, z indukce vyplývá, že B_{k-1} je binomiální strom řádu $k-1$, pod jehož kořen jsou dle definice napojeny binomiální stromy řádů $0, \ldots, k-2$. Pod kořen B_k jsou tudíž napojeny binomiální stromy řádů $0, \ldots, k-1$, tedy strom B_k je binomiální strom řádu k. □

Lemma V (vlastnosti binomiálních stromů): Binomiální strom T řádu k má 2^k vrcholů, které jsou rozděleny do $k+1$ hladin. Kořen stromu má právě k synů.

Důkaz: Počet synů kořene plyne přímo z definice binomiálního stromu, zbytek dokážeme matematickou indukcí. Pro $k = 0$ má T jistě 1 hladinu a $2^0 = 1$ vrchol. Uvažme $k > 0$. Z indukčního předpokladu vyplývá, že binomiální strom řádu $k-1$ má k hladin a 2^{k-1} vrcholů. Užitím lemmatu **R** dostáváme, že strom T je složený ze dvou stromů B_{k-1}, z nichž jeden je o hladinu níže než druhý, což dává hloubku $k+1$ stromu T. Složením dvou stromů B_{k-1} dostáváme $2 \cdot 2^{k-1} = 2^k$ vrcholů. □

Důsledek: Binomiální strom s n vrcholy má hloubku $\mathcal{O}(\log n)$ a počet synů kořene je taktéž $\mathcal{O}(\log n)$.

Od stromu k haldě

Z binomiálních stromů nyní zkonstruujeme binomiální haldu.

Definice: *Binomiální halda* pro danou množinu prvků se skládá ze souboru binomiálních stromů $\mathcal{T} = T_1, \ldots, T_\ell$, kde:

1. Každý strom T_i je binomiální strom.
2. Uchovávané prvky jsou uloženy ve vrcholech stromů T_i. Klíč uložený ve vrcholu $v \in V(T_i)$ značíme $k(v)$.
3. Pro každý strom $T_i \in \mathcal{T}$ platí *haldové uspořádání,* neboli pro každý vrchol $v \in V(T_i)$ a libovolného jeho syna s je $k(v) \leq k(s)$.
4. V souboru $\mathcal{T}$ se žádný řád stromu nevyskytuje více než jednou.
5. Soubor stromů $\mathcal{T}$ je uspořádán vzestupně podle řádu stromu.

Jako vhodný způsob uložení souboru stromů $\mathcal{T}$ tedy poslouží například spojový seznam. Každý vrchol stromu si též bude pamatovat spojový seznam svých synů a řád podstromu, jehož je kořenem.

Lemma D (o dvojkovém zápisu): Binomiální strom řádu k se vyskytuje v souboru stromů n-prvkové binomiální haldy právě tehdy, když je ve dvojkovém zápisu čísla n k-tý nejnižší bit roven 1.

Důkaz: Z definice binomiální haldy vyplývá, že binomiální stromy dohromady obsahují $\sum_{i=0}^{k} b_i 2^i = n$ vrcholů, kde k je maximální řád stromu v $\mathcal{T}$ a $b_i \in \{0, 1\}$. Číslo $b_k b_{k-1} \ldots b_0$ tedy tvoří zápis čísla n v dvojkové soustavě. Z vlastností zápisu čísla ve dvojkové soustavě plyne, že pro dané n jsou čísla b_i (a tím i řády binomiálních stromů v $\mathcal{T}$) určena jednoznačně. □

Z lemmatu **V** nebo **D** plyne následující důležitá vlastnost:

Důsledek: Binomiální halda s n prvky sestává z nejvýše $\lfloor \log_2 n \rfloor + 1$ binomiálních stromů.

Cvičení

1. Binomiální stromy vděčí za svůj název následující vlastnosti: Počet prvků na i-té hladině (číslujeme od 0) binomiálního stromu řádu k je roven kombinačnímu číslu neboli binomiálnímu koeficientu $\binom{k}{i}$. Dokažte.

2. Kolik má n-prvková binomiální halda listů?

3. Dokažte, že libovolné přirozené číslo x lze zapsat jako konečný součet mocnin dvojky $2^{k_1} + 2^{k_2} + \ldots$ tak, že $k_i \neq k_j$ pro různá i, j. Ukažte, že v tomto součtu figuruje nejvýše $\lfloor \log_2 x \rfloor + 1$ sčítanců.

4. Na binomiální strom řádu k lze pohlížet jako na speciální kostru grafu k-rozměrné hyperkrychle. A to ne ledajakou, ale dokonce je to strom nejkratších cest. Dokažte například užitím cvičení 1.

18.2 Operace s binomiální haldou

Nalezení minima

Jak jsme již ukázali u binární haldy, pokud strom splňuje haldovou podmínku, prvek s nejmenším klíčem se nachází v kořeni stromu. Minimum celé binomiální haldy se proto musí nacházet v kořeni jednoho ze stromů v $\mathcal{T}$. Operaci MIN tedy postačí projít seznam $\mathcal{T}$, což potrvá čas $\Theta(\log n)$. Pokud bychom tuto operaci chtěli volat často, můžeme ji urychlit na $\Theta(1)$ tím, že budeme během všech operací udržovat ukazatel na globální minimum. Tuto údržbu v ostatních algoritmech explicitně nepopisujeme, nýbrž ji přenecháváme čtenáři do cvičení 3.

Slévání

Operaci MERGE poněkud netypicky popíšeme jako jednu z prvních, protože ji budeme nadále používat jako podproceduru ostatních operací. Algoritmus slévání vezme dvě binomiální haldy H_1 a H_2 a vytvoří z nich jedinou binomiální haldu H, jež obsahuje prvky obou hald.

Nejprve popíšeme spojení dvou binomiálních stromů stejného řádu. Při něm je potřeba napojit kořen jednoho stromu jako posledního syna kořene druhého stromu. Přitom musíme dát pozor, aby zůstalo zachováno haldové uspořádání, takže vždy zapojujeme kořen s větším prvkem pod kořen s menším prvkem. Vznikne binomiální strom o jedna vyššího řádu.

Procedura MERGEBINOMTREES (spojení binomiálních stromů)
Vstup: Stromy B_1, B_2 z binomiální haldy ($\check{r}\acute{a}d(B_1) = \check{r}\acute{a}d(B_2)$)

1. Pokud $k(kořen(B_1)) \leq k(kořen(B_2))$:
2. Připojíme $kořen(B_2)$ jako posledního syna pod $kořen(B_1)$.
3. $B \leftarrow B_1$
4. Jinak:
5. Připojíme $kořen(B_1)$ jako posledního syna pod $kořen(B_2)$.
6. $B \leftarrow B_2$
7. $\check{r}\acute{a}d(B) \leftarrow \check{r}\acute{a}d(B) + 1$

Výstup: Výsledný strom B

Algoritmus BHMERGE svým průběhem bude připomínat algoritmus „školního“ sčítání čísel pod sebou, byť ve dvojkové soustavě.

Při sčítání dvojkových čísel procházíme obě čísla současně od nejnižšího řádu k nejvyššímu. Pokud je v daném řádu v obou číslech 0, píšeme do výsledku 0. Pokud je v jednom z čísel 0 a ve druhém 1, píšeme 1. A pokud se setkají dvě 1, píšeme 0 a přenášíme 1 do vyššího řádu. Díky přenosu se pak ve vyšším řádu mohou setkat až tři 1, a tehdy píšeme 1 a posíláme přenos 1.

Podobně pracuje slévání binomiálních hald: binomiální strom řádu i se chová jako číslice 1 na i-tém řádu čísla. Procházíme tedy oběma haldami od nejnižšího řádu k nejvyššímu a kdykoliv se setkají dva stromy téhož řádu, sloučíme je pomocí MERGEBINOMTREES, což vytvoří strom o jedna vyššího řádu, čili přenos.

Protože udržujeme soubory stromů hald uspořádané dle řádu binomiálních stromů, lze algoritmus realizovat průchodem dvěma ukazateli po těchto seznamech, jako když sléváme setříděné posloupnosti. Řády stromů nepřítomné v haldě při tom přirozeně přeskakujeme.

Procedura BhMerge (slévání binomiálních hald)
Vstup: Binomiální haldy A, B

1. Založíme prázdnou haldu C.
2. $p \leftarrow$ *nedefinováno* ◁ *přenos do vyššího řádu*
3. Dokud A i B jsou neprázdné nebo je p definováno:
4. $\quad r_a \leftarrow$ nejnižší řád stromu v A, nebo $+\infty$ pro $A = \emptyset$
5. $\quad r_b \leftarrow$ nejnižší řád stromu v B, nebo $+\infty$ pro $B = \emptyset$
6. $\quad r_p \leftarrow$ řád stromu p (nebo ∞, pokud p není definováno)
7. $\quad r \leftarrow \min(r_a, r_b, r_p)$
8. $\quad$ Pokud $r = +\infty$, skončíme. ◁ *už není co slévat*
9. $\quad S \leftarrow \emptyset$ ◁ *sem uložíme stromy, které budeme spojovat*
10. $\quad$ Pokud $r_a = r$, přesuneme strom nejnižšího řádu z A do S.
11. $\quad$ Pokud $r_b = r$, přesuneme strom nejnižšího řádu z B do S.
12. $\quad$ Pokud p je definováno, přidáme p do S. ◁ *musí být* $r_p = r$
13. $\quad$ Pokud $|S| \geq 2$:
14. $\quad\quad$ Odebereme dva stromy z S a označíme je x a y.
15. $\quad\quad p \leftarrow$ MergeBinomTrees(x, y)
16. $\quad$ Pokud v S zbývá nějaký strom, přesuneme ho na konec haldy C.
17. Pokud je A nebo B neprázdná, připojíme ji na konec haldy C.

Výstup: Binomiální halda C poskládaná z prvků A a B

Pozorování: Algoritmus BhMerge je korektní a jeho časová složitost je $\Theta(\log n)$.

Vkládání prvků a postavení haldy

Operaci Insert vyřešíme snadno. Vytvoříme novou binomiální haldu obsahující pouze vkládaný prvek a následně zavoláme slévání hald.

Snadno nahlédneme, že pouhé přeuspořádání stromů při přidání nového prvku tak, aby v seznamu nebyly dva stromy stejného řádu, může v nejhorším případě vyžadovat čas $\Theta(\log n)$.

Procedura BhInsert (vkládání do binomiální haldy)
Vstup: Binomiální halda H, vkládaný prvek x

1. Vytvoříme binomiální haldu H' s jediným prvkem x.
2. $H \leftarrow$ BhMerge(H, H')

Výstup: Binomiální halda H s vloženým prvkem x

Tvrzení: Operace Insert má časovou složitost $\Theta(\log n)$. Pro zpočátku prázdnou binomiální haldu trvá libovolná posloupnost k volání operace Insert čas $\Theta(k)$.

Důkaz: Jednoprvkovou haldu lze vytvořit v konstantním čase, takže těžiště práce bude ve slévání hald. Slévání zvládneme v čase $\Theta(\log n)$, tedy i vkládání prvku bude mít v nejhorším případě logaritmickou časovou složitost.

Zde je však jedna ze slévaných hald jednoprvková. V nejhorším případě se samozřejmě může stát, že původní halda obsahuje všechny stromy od B_0 až po $B_{\lceil \log_2 n \rceil - 1}$, takže při slévání dojde k řetězové reakci a postupně se všechny stromy sloučí do jediného, což si vyžádá $\Theta(\log n)$ operací. Ukážeme ovšem, že se to nemůže dít často.

Využijeme skutečností dokázaných pro binární počítadlo v kapitole 9. Připomeňme, že provedeme-li posloupnost k inkrementů na počítadle, které bylo zpočátku nulové, strávíme celkově čas $\mathcal{O}(k)$. Opakované volání BhInsert je ekvivalentní opakovanému inkrementu binárního počítadla. Operace součtu dvou jedničkových bitů potom odpovídá operaci slití dvou binomiálních stromů. Při použití procedury BhMerge stačí odhadnout pouze maximální počet volání MergeBinomTrees, z čehož vyplývá celková časová složitost $\Theta(k)$ pro k volání procedury BhInsert. □

Analýza operace Insert dává návod na realizaci rychlé operace Build pro postavení binomiální haldy: opakovaně voláme na zpočátku prázdnou binomiální haldu operaci Insert.

Důsledek: Časová složitost operace Build pro n prvků je $\Theta(n)$.

Na rozdíl od binární haldy, jejíž rychlá stavba vyžadovala speciální postup, zde pro rychlé postavení binomiální haldy stačilo pouze lépe analyzovat časovou složitost InsertTu.

Odstranění minima

Při odstraňování minima z binomiální haldy H opět využijeme operaci Merge. Nejprve průchodem souboru stromů najdeme binomiální strom M, jehož kořen je minimem haldy H, a tento strom z H odpojíme. Následně ze stromu M odtrhneme kořen a všechny jeho syny (včetně jejich podstromů) vložíme do nové binomiální haldy H'. Tato operace je poměrně jednoduchá, neboť se mezi syny dle definice binomiálního stromu nevyskytují dva stromy stejného řádu. Nakonec slijeme H s H', čímž se odtržené prvky začlení zpět.

Procedura BhExtractMin (odebrání minima z binomiální haldy)
Vstup: Binomiální halda H

1. $M \leftarrow$ strom s nejmenším kořenem v haldě H
2. $m \leftarrow k(kořen(M))$
3. Odebereme M z H.
4. Vytvoříme prázdnou binomiální haldu H'.
5. Pro každého syna s kořene stromu M:
6. Odtrhneme podstrom s kořenem s a vložíme jej do H'.
7. Zrušíme M. ◁ *zbude jen kořen, který zrušíme*

8. $H \leftarrow$ BhMerge(H, H')

Výstup: Binomiální halda H s odstraněným minimem m

Tvrzení: Časová složitost operace BhExtractMin v n-prvkové binomiální haldě činí $\Theta(\log n)$. Libovolná korektní implementace operace BhExtractMin má časovou složitost $\Omega(\log n)$.

Důkaz: Nalezení minima trvá čas $\mathcal{O}(\log n)$, protože v souboru stromů je jich jen logaritmicky mnoho. Vytvoření haldy H' pro podstromy odstraňovaného kořene zabere nejvýše tolik času, kolik podstromů do ni vkládáme, tedy $\mathcal{O}(\log n)$. Slévání hald má také logaritmickou složitost, takže celková složitost algoritmu v nejhorším případě je $\Theta(\log n)$.

Dolní odhad složitosti odstraňování minima získáme z dolního odhadu složitosti třídění velmi podobně, jako jsme podobnou skutečnost dokázali u binární haldy (viz kapitola 3). Kdyby existoval algoritmus na odstranění minima s časovou složitostí lepší než $\Theta(\log n)$, zkonstruovali bychom třídicí algoritmus takto: Vložili bychom n tříděných prvků operací Build do haldy a následně n-násobným odstraněním minima vypsali uspořádanou posloupnost. Tento algoritmus by však měl časovou složitost lepší než $\Theta(n \log n)$, což je spor s dolním odhadem složitosti třídění. □

V úvodu této kapitoly jsme zmínili ještě operace Decrease, Delete a Increase, které dostanou ukazatel na binomiální haldu a ukazatel na prvek v ní a provedou po řadě snížení klíče prvku, smazání prvku a zvýšení klíče. Tyto operace přenecháme čtenáři jako cvičení 4, 5 a 6.

Cvičení

1. Přeformulujte všechny definice a operace pro maximovou binomiální haldu.
2. Promyslete detaily reprezentace binomiální haldy ve vašem oblíbeném programovacím jazyce tak, aby byla zachována časová složitost všech operací.
3. U binomiální haldy lze minimum (přesněji referenci na kořen obsahující minimum) udržovat stranou, abychom k němu mohli přistupovat v konstantním čase. Upravte všechny operace tak, aby zároveň udržovaly odkaz na minimum a nezhoršily se jejich časové složitosti.
4. Navrhněte operaci Decrease s časovou složitostí $\Theta(\log n)$. Nezapomeňte, že bude třeba do reprezentace haldy v paměti doplnit další ukazatele a ty udržovat.
5. Navrhněte operaci Delete s časovou složitostí $\Theta(\log n)$.
6. Navrhněte operaci Increase s časovou složitostí $\Theta(\log n)$.

7.* Pokuste se definovat *trinomiální haldu,* jejíž stromy budou mít velikost mocnin trojky. Od každého řádu se přitom v haldě budou smět vyskytovat až dva stromy. Domyslete operace s touto haldou a srovnejte jejich složitosti s haldou binomiální.

18.3 Líná binomiální halda

Alternativou k „pilné“ binomiální haldě je tzv. *líná (lazy) binomiální halda.* Její princip spočívá v odložení některých úkonů při vkládání prvků a odstraňování minima, dokud nejsou opravdu potřeba. Ukážeme, že tento postup sice zhorší časovou složitost v nejhorším případě, ale amortizovaně bude odebírání minima nadále logaritmické, a některé další operace dokonce konstantní.

Definice líné binomiální haldy se téměř neliší od pilné. Pouze povolíme, že se v souboru stromů může vyskytovat více stromů stejného řádu. Reprezentace struktury v paměti bude stejná jako u pilné haldy, tedy pomocí spojových seznamů. Navíc se bude hodit, aby seznamy byly obousměrné a kruhové.

Operace slití dvou hald, kterou využívá vkládání prvku i vypuštění minima, se značně zjednoduší. Vzhledem k tomu, že se řády stromů mohou v haldě opakovat, slití realizujeme spojením seznamů stromů obou hald, což jistě zvládneme v konstantním čase.

Procedura LazyBhMerge (slévání líných binomiálních hald)
Vstup: Líné binomiální haldy H_1, H_2
1. Založíme novou haldu H.
2. Seznam stromů v $H \leftarrow$ spojení seznamů stromů v H_1 a H_2.

Výstup: Líná binomiální halda H poskládaná z prvků H_1 a H_2

Operace Insert pro vložení nového prvku do haldy je opět realizována jako slití haldy s novou, jednoprvkovou haldou.

Aby halda nezdegenerovala v obyčejný spojový seznam, musíme čas od času provést tzv. *konsolidaci* a stromy sloučit tak, aby jich bylo v seznamu co nejméně. Nejvhodnější čas na tento úklid je při mazání minima, které provedeme podobně jako u pilné binomiální haldy: Odtrhneme minimální kořen, z jeho synů uděláme haldu a tu následně výše popsaným způsobem slijeme s původní haldou. Na závěr provedeme následující konsolidaci.

Všechny stromy rozdělíme do $\lceil \log n \rceil + 1$ přihrádek (číslovaných od 0) tak, že v i-té přihrádce se budou nacházet všechny stromy řádu i. Již však nemůžeme činit žádné předpoklady o počtech stromů v jednotlivých přihrádkách.

Proto budeme procházet přihrádky od nejnižšího řádu k nejvyššímu a kdykoliv v některé objevíme více stromů, budeme je odebírat po dvojicích a slučovat. Sloučené stromy budou

mít o jedna vyšší řád, takže je přehodíme o přihrádku výše a vyřešíme v následujícím kroku. Nakonec tedy v každé přihrádce zbude nejvýše jeden strom.

Procedura LazyBhConsolidation (konsolidace líné binomiální haldy)
Vstup: Líná binomiální halda H o n prvcích

1. Připravíme pole $P[0 \dots \lceil \log n \rceil]$ spojových seznamů.
2. Pro každý strom T v H:
3. Odtrhneme T z H.
4. Vložíme T do $P[\check{r}\acute{a}d(T)]$.
5. Pro všechna $i = 0, \dots, \lceil \log n \rceil$:
6. Opakujeme, dokud jsou v $P[i]$ alespoň dva stromy:
7. Odtrhneme dva stromy B_1, B_2 z $P[i]$.
8. $B \leftarrow$ MergeBinomTrees(B_1, B_2)
9. Vložíme B do $P[i+1]$.
10. Pokud v $P[i]$ zbyl strom T:
11. Přesuneme T z $P[i]$ do H.

Výstup: Zkonsolidovaná halda H

Nyní můžeme přesněji popsat mazání minima.

Procedura LazyBhExtractMin (odebrání minima z líné binomiální haldy)
Vstup: Líná binomiální halda H

1. $M \leftarrow$ strom s nejmenším kořenem v haldě H
2. $m \leftarrow k(kořen(M))$
3. Odebereme M z H.
4. Odtrhneme seznam S podstromů kořene stromu M.
5. Připojíme S do seznamu stromů H.
6. Zrušíme M. ◁ *zbude jen kořen, který zrušíme*
7. $H \leftarrow$ LazyBhConsolidation(H)

Výstup: Líná binomiální halda H s odstraněným minimem m

Analýza líné binomiální haldy

Pro účely amortizované analýzy zvolíme potenciál $\Phi = c_\Phi \cdot t$, kde t je celkový počet stromů ve všech haldách, kterých se naše operace týkají, a c_Φ je vhodná konstanta, kterou určíme později. Pro ilustraci a analýzu penízkovou metodou si můžeme představit, že na každém stromu leží položeno c_Φ mincí. Označíme Φ_i hodnotu potenciálu po provedení i-té operace. Zjevně platí, že $\Phi_0 \leq \Phi_k$, kde k je počet provedených operací, neboť na počátku jsou všechny struktury prázdné a na konci je ve strukturách nezáporný počet stromů.

Lemma: Uvažme volání procedury LazyBhConsolidation(H) pro haldu H s nejvýše n prvky. Jeho skutečná cena je $\Theta(\log n + \textit{počet stromů } H)$ a jeho amortizovaná cena je $\mathcal{O}(\log n)$ vzhledem k potenciálu Φ.

Důkaz: Označme t počet stromů H před provedením konsolidace, A amortizovanou cenu konsolidace a C skutečnou cenu konsolidace. Inicializace pole P zabere čas $\Theta(\log n)$, stejně tak průchod přihrádkami. Vkládání stromů do P trvá lineárně s počtem stromů v H. Stejně tak jejich následné spojování, neboť každým spojením ubude jeden strom. Je tedy $C \leq c_1(\log n + t)$ pro jistou konstantu c_1.

Abychom ukázali, že amortizovaná cena A konsolidace je logaritmická, stačí ověřit $A = C + \Delta\Phi = \mathcal{O}(\log n)$, kde $\Delta\Phi$ je změna potenciálu. Označme t' počet stromů v H po provedení konsolidace; zřejmě $t' \leq \lceil \log_2 n \rceil$. Tedy platí $A \leq c_1(\log n + t) + c_\Phi(t' - t) \leq c_\Phi(\log n + t) + c_\Phi(t' - t) = \mathcal{O}(\log n)$, za předpokladu že $c_\Phi \geq c_1$. □

Lemma: Uvažme volání procedur LazyBhMerge(H_1, H_2) a LazyBhInsert(H, x). Jejich skutečná cena je $\Theta(1)$ a jejich amortizovaná cena je $\Theta(1)$ vzhledem k potenciálu Φ.

Důkaz: Během slévání dvou hald se provede pouze konstantně mnoho operací, tedy skutečná cena je konstantní. Změna potenciálu je nulová, protože celkový počet stromů se nezmění, a tedy amortizovaná cena je též konstantní.

Vkládání do haldy nejprve založí jednoprvkový strom, což má konstantní skutečnou cenu a zvýší potenciál o c_Φ, takže amortizovaná cena je též konstantní. Poté provedeme slévání, čímž skutečnou ani amortizovanou cenu nezhoršíme. □

Lemma: Uvažme volání procedury LazyBhExtractMin(H), kde H má nejvýše n prvků. Jeho skutečná cena je $\Theta(\log n + \textit{počet stromů } H)$ a jeho amortizovaná cena je $\mathcal{O}(\log n)$ vzhledem k potenciálu Φ.

Důkaz: Označme nejprve s počet synů kořene stromu M, které procedura přepojuje. Jelikož řády stromů jsou nejvýše logaritmické, je s také $\mathcal{O}(\log n)$.

Volbu minimálního stromu M si necháme na konec důkazu. Odtržení M z H, připojení synů kořene M do H a zrušení M lze realizovat za konstantní skutečnou cenu. Potenciál se při tom zvýší o $s - 1$. Amortizovaná cena tedy vyjde $\mathcal{O}(\log n)$.

Označme t počet stromů H a t' počet stromů po provedení konsolidace; je tedy $t' = \mathcal{O}(\log n)$. Následná konsolidace má podle předchozích lemmat skutečnou cenu $\Theta(\log n + t + s) = \Theta(\log n + t) \leq c_3(\log n + t)$ pro nějakou konstantu c_3. Konečně započítáme také volbu minimálního stromu. Tu lze realizovat ve skutečném čase $\Theta(t)$, její skutečná cena je tedy nejvýše $c_2 t$ pro nějakou konstantu c_2. Stejně jako u konsolidace se tato cena sečte s poklesem potenciálu a vyjde amortizovaná cena $\mathcal{O}(\log n)$. Přesněji, pro amortizovanou

cenu A platí $A \leq c_2 t + c_3(\log n + t) + c_\Phi(t' - t) \leq c_\Phi(\log n + t) + c_\Phi(t' - t) = \mathcal{O}(\log n)$, za předpokladu $c_\Phi \geq c_2 + c_3$. □

Nyní zbývá stanovit, jaká bude konstanta c_Φ v definici potenciálu Φ: stačí zvolit $c_\Phi = \max(c_1, c_2 + c_3)$ z předchozích důkazů. Poznamenejme ještě, že počet stromů líné binomiální haldy může být až $\Theta(n)$, takže složitost operace LazyBhExtractMin v nejhorším případě je taktéž lineární.

Shrňme na závěr složitosti zmiňovaných operací líné binomiální haldy. Z nich zbývá ukázat ještě dolní odhad amortizované složitosti ExtractMin, který přenecháme do cvičení 3.

operace	*worst-case čas*	*amortizovaný čas*
Insert	$\Theta(1)$	$\Theta(1)$
ExtractMin	$\Theta(n)$	$\Theta(\log n)$
Merge	$\Theta(1)$	$\Theta(1)$

Význam líné binomiální haldy vzroste především v dalším oddílu, kde slouží jako předstupeň pro návrh tzv. Fibonacciho haldy.

Cvičení

1. Zjistěte, jak by se změnily složitosti jednotlivých operací, kdybychom v implementaci používali místo obousměrného kruhového seznamu pouze jednosměrný lineární a udržovali zároveň ukazatel na poslední prvek seznamu.

2. Navrhněte operace Min, Decrease, Delete a Increase pro línou binomiální haldu. Jaká bude jejich skutečná a amortizovaná cena?

3. Ukažte, že složitost ExtractMin musí nutně být $\Omega(\log n)$, a to jak worst-case, tak amortizovaně.

4. *Konsolidace párováním:* Konsolidaci se můžeme pokusit zjednodušit tak, že po spojení dvou stromů stejného řádu výsledný strom rovnou umístíme do výsledné haldy. Může se tedy stát, že ve zkonsolidované haldě se budou řády stromů opakovat. Dokažte, že to neuškodí amortizované složitosti operací vůči potenciálu Φ.

18.4 Fibonacciho haldy

Budeme pokračovat v myšlence dalšího „zlíňování“ již tak dost líné binomiální haldy. Hlavním prostředkem bude zvolnění požadavku na strukturu stromů, ze kterých halda sestává.

Definice: *Fibonacciho halda* pro danou množinu prvků se skládá ze souboru stromů $\mathcal{T} = T_1, \ldots, T_\ell$, kde:

1. Uchovávané prvky jsou uloženy ve vrcholech stromů T_i. Klíč uložený ve vrcholu $v \in V(T_i)$ značíme $k(v)$.
2. Pro každý strom $T_i \in \mathcal{T}$ platí *haldové uspořádání,* neboli pro každý vrchol $v \in V(T_i)$ a libovolného jeho syna s je $k(v) \leq k(s)$.
3. Pro práci se strukturou se používají výhradně operace popsané níže.

Omezení na tvar stromů tentokrát neplynou přímo z definice, nýbrž z chování jednotlivých operací.

Soubor stromů a interní reprezentaci stromů budeme opět udržovat v kruhových spojových seznamech. Kromě toho si každý vrchol bude pamatovat svůj *řád,* tentokrát definovaný přímo jako počet synů vrcholu. *Řádem stromu* se rozumí řád jeho kořene.

Operace

Fibonacciho halda podporuje následující operace. Uvádíme u nich časové složitosti v nejhorším případě i amortizované, vše vzhledem k aktuálnímu počtu prvků n.

operace	*nejhůře*	*amortizovaně*	*činnost*
Insert	$\Theta(1)$	$\Theta(1)$	vloží nový prvek
Min	$\Theta(1)$	$\Theta(1)$	vrátí minimum množiny
ExtractMin	$\Theta(n)$	$\Theta(\log n)$	vrátí a odstraní minimum
Merge	$\Theta(1)$	$\Theta(1)$	sloučí dvě haldy do jedné
Build	$\Theta(n)$	$\Theta(n)$	postaví z n prvků haldu
Decrease	$\Theta(n)$	$\Theta(1)$	sníží hodnotu klíče prvku
Delete	$\Theta(n)$	$\Theta(\log n)$	smaže prvek

Základní operace Fibonacciho haldy se téměř neliší od haldy binomiální.

- Min bude vracet ukazatel na nejmenší z kořenů stromů tvořících haldu. Tento ukazatel budeme průběžně přepočítávat. V algoritmech jeho přepočet nezmiňujeme, přenecháváme ho čtenáři k rozmyšlení jako cvičení 2.
- Merge pouze zřetězí spojové seznamy obou hald.
- Insert vyrobí jednovrcholovou haldu s novým prvkem a tuto novou haldu spojí voláním Merge s haldou původní. Všechny předchozí operace mají zjevně worst-case časovou složitost $\Theta(1)$.
- Operaci Build provedeme jako n-násobné volání Insert s celkovou složitostí $\Theta(n)$.

Operaci ExtractMin provedeme takřka stejně jako u líné binomiální haldy. Najdeme a odtrhneme kořen s minimálním klíčem, seznam jeho podstromů prohlásíme za novou Fibonacciho haldu a tu připojíme k haldě původní. Potom provedeme konsolidaci: setřídíme přihrádkově stromy dle jejich řádů a následně pospojujeme stromy stejných řádů tak, aby od každého řádu zbyl nejvýše jeden strom. Spojení stromů funguje analogicky ke spojování binomiálních stromů, tedy pod kořen s menším prvkem přivěsíme druhý strom, čímž řád vzroste o 1.

Hlavním rozdílem oproti binomiální haldě a zároveň motivací pro použití Fibonacciho haldy je zcela jinak fungující operace Decrease pro snížení hodnoty klíče. Pokud dojde ke snížení klíče, může vzniknout porucha v uspořádání haldy mezi modifikovaným prvkem a jeho otcem. U binární nebo binomiální haldy bychom tuto poruchu vyřešili vybubláním sníženého klíče nahoru. Zde však podstrom zakořeněný v prvku se sníženým klíčem odtrhneme a vložíme do spojového seznamu mezi ostatní stromy haldy. Tomuto odtržení a přesunutí budeme říkat operace Cut.

Všimněme si, že pokud bychom používali pouze operace vkládání prvku a odstraňování minima, vznikaly by v souboru stromů haldy pouze binomiální stromy. Opakované snižování klíčů v jednom stromu by však mohlo způsobit, že by strom degeneroval na strom s nevhodnými vlastnostmi, protože by vrcholy mohly mít odtrženo příliš mnoho svých synů.

Každého otce, jemuž byl odtržen jeden syn, proto *označíme.* Pokud byl odtržen syn vrcholu v, který byl již předtím označený, zavoláme Cut i na v. To může vyústit v kaskádovité odtrhávání podstromů, dokud nenarazíme na neoznačený vrchol, případně kořen.

Co se kořenů stromů týče, algoritmus udržuje invariant, že kořeny nikdy nejsou označené. Operace ExtractMin s tímto invariantem musí počítat a musíme ji tedy ještě upravit: při zařazení odtržených podstromů do haldy z jejich kořenů odstraníme případné označení. Podobně u Insertu je nově vytvořený prvek neoznačený.

Procedura FhDecrease (snížení klíče prvku ve Fibonacciho haldě)
Vstup: Fibonacciho halda H, vrchol x, nový klíč t

1. $k(x) \leftarrow t$
2. Pokud je x kořen nebo $k(otec(x)) \leq k(x)$, skončíme.
3. Zavoláme FhCut(x).

Výstup: Upravená halda H

Procedura FhCut (odseknutí podstromu Fibonacciho haldy)
Vstup: Fibonacciho halda H, kořen x podstromu k odseknutí

1. $o \leftarrow otec(x)$

2. Odtrhneme x i s podstromem, odstraníme případné označení x a vložíme x do H.
3. Pokud o je označený, zavoláme FhCut(o). ◁ *o jistě není kořen*
4. Jinak není-li o kořen, označíme ho.

Výstup: Upravená halda H

Konečně, operaci Delete realizujeme snížením klíče mazaného prvku na $-\infty$ voláním Decrease a následným voláním ExtractMin.

Analýza Fibonacciho haldy

V analýze haldy budeme využívat Fibonacciho čísla F_n zavedená v oddílu 1.4. Bude se nám hodit jejich následující vlastnost, jejíž důkaz přenecháme do cvičení 1.

Lemma: Pro Fibonacciho čísla platí:

$$1 + \sum_{i=0}^{d} F_i = F_{d+2}.$$

Značení: Označme T_v podstrom zakořeněný ve vrcholu v a $|T_v|$ počet jeho vrcholů.

Tvrzení: Po každé provedené operaci splňuje halda $\mathcal{T}$ definici Fibonacciho haldy a platí, že je-li strom $T \in \mathcal{T}$ řádu k, potom $|T| \geq F_{k+2}$.

Důkaz: Výsledná halda po operacích Insert, Min, Merge splňuje definici Fibonacciho haldy, protože tyto operace nijak nemění strukturu stromů. Ze stejného důvodu je po předchozích operacích zachován vztah pro velikost stromu. Uvažme nyní operace ExtractMin a Decrease.

Zvolme vrchol v z libovolného stromu haldy. Indukcí podle hloubky T_v ukážeme, že $|T_v| \geq F_{k+2}$, kde k je řád v. Jestliže T_v má hloubku 0, je $|T_v| = 1 = F_2$. Předpokládejme dále, že T_v má kladnou hloubku a řád $k > 0$. Označme $x_1, \ldots, x_k$ syny vrcholu v v pořadí, v jakém byly vrcholy připojeny pod v (x_1 první, x_k poslední), a označme $r_1, \ldots, r_k$ jejich řády.

Dokážeme, že $r_i \geq i-2$ pro každé $2 \leq i \leq k$. Než byl x_i připojen pod v, byly $x_1, \ldots, x_{i-1}$ už syny vrcholu v a tedy v měl řád alespoň $i-1$. Stromy jsou spojovány jen tehdy, mají-li stejný řád, musel tedy také x_i mít v okamžiku připojení pod v řád alespoň $i-1$. Od té doby mohl x_i ztratit pouze jednoho syna, což zaručuje mechanismus označování vrcholů, a tedy $r_i \geq i-2$.

Jelikož hloubky všech T_{x_i} jsou ostře menší než hloubka T_v, z indukčního předpokladu plyne $|T_{x_i}| \geq F_{r_i+2} \geq F_{(i-2)+2} = F_i$. Vrcholy v a x_1 přispívají do $|T_v|$ každý alespoň 1.

Dostáváme tak

$$|T_v| \geq 2 + \sum_{i=2}^{k} |T_{x_i}| \geq 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=0}^{k} F_i = F_{k+2}.$$

Korektnost operace DELETE plyne jednoduše z korektnosti předchozích operací. □

Z předchozího tvrzení a toho, že Fibonacciho čísla rostou alespoň tak rychle jako funkce 1.618^n (viz cvičení 1.4.4), plyne následující důsledek.

Důsledek: Řád každého stromu ve Fibonacciho haldě je nejvýše $\lceil \log_{1.618} n \rceil$.

Nyní přistoupíme k amortizované analýze. Jako potenciál zvolíme $\Phi = c_\Phi(t + 2m)$, kde t je celkový počet stromů ve všech haldách, kterých se operace týkají, m je celkový počet označených vrcholů v nich a c_Φ je konstanta, kterou určíme později. Pro představu analýzou pomocí penízkové metody: na každém stromu bude položeno c_Φ mincí a na každém označeném vrcholu $2c_\Phi$ mincí.

Lemma: Uvažme volání procedur FHMIN, FHMERGE a FHINSERT. Jejich skutečná cena i amortizovaná cena jsou $\mathcal{O}(1)$ vzhledem k potenciálu Φ.

Důkaz: FHMIN pouze vrací zapamatované minimum a potenciál nemění. Během slévání hald se provede jen konstantně mnoho operací, takže skutečná cena je konstantní. Amortizovaná cena také, neboť potenciál se nemění (počet stromů se nemění, nic se neoznačuje). Konečně FHINSERT kromě slévání vytvoří jednoprvkovou haldu, což stojí konstantní skutečný čas a zvýší potenciál o konstantu. Amortizovaná cena je proto opět konstantní. □

Z předchozí analýzy FHINSERT snadno vyplyne analýza FHBUILD.

Důsledek: Uvažme volání procedury FHBUILD pro n prvků. Jeho skutečná i amortizovaná cena je $\mathcal{O}(n)$.

Lemma: Uvažme volání procedury FHEXTRACTMIN(H), kde H má n prvků. Potom jeho skutečná cena je $\Theta(\log n + \textit{počet stromů } H)$ a jeho amortizovaná cena je $\mathcal{O}(\log n)$ vzhledem k potenciálu Φ.

Důkaz: Označme t počet stromů H před odstraněním minima, A amortizovanou cenu a C skutečnou cenu. Volbu stromu M s nejmenším kořenem lze realizovat v čase nejvýše $c_1 t$ pro vhodnou konstantu c_1. Odtržení M z H, připojení seznamu synů M do H a zrušení M lze realizovat za konstantní skutečnou cenu. Odznačení synů M trvá čas lineární k jejich počtu, což je $\mathcal{O}(\log n)$. Při konsolidaci zabere inicializace pole přihrádek čas $\Theta(\log n)$, stejně tak průchod přihrádkami, jelikož maximální řád stromu je $\mathcal{O}(\log n)$. Zbývajících operací, tedy vkládání stromů do přihrádek a jejich následné spojování, je $\Theta(\log n + t)$. Je tedy $C \leq c_2(\log n + t)$ pro nějakou konstantu c_2.

Abychom ukázali, že amortizovaná cena A je logaritmická, stačí ověřit $A = C + \Delta\Phi = \mathcal{O}(\log n)$, kde $\Delta\Phi$ je změna potenciálu. Označme t' počet stromů v H a m' počet označených vrcholů po provedení konsolidace. Nový počet stromů $t' \leq c_3 \log n$ pro vhodnou konstantu c_3 a $m' \leq m$, jelikož se neoznačují žádné další vrcholy. Tedy za předpokladu $c_1 + c_2 + c_3 \leq c_\Phi$ platí $A \leq c_1 t + c_2(\log n + t) + c_3 t' + 2m' - c_\Phi(t + 2m) \leq c_\Phi(\log n + t) + c_\Phi(t' + 2m') - c_\Phi(t + 2m) = \mathcal{O}(\log n)$. □

Z předchozího důkazu vidíme, že v definici potenciálu Φ stačí zvolit konstantu $c_\Phi \geq c_1 + c_2 + c_3$. Zřejmě v nejhorším případě může v n-prvkové haldě FhExtractMin trvat čas $\Theta(n)$.

Lemma: Uvažme volání procedury FhCut(v), kde v je odsekávaný vrchol. Označme ℓ počet nových stromů, které operace při svém běhu vytvoří. Potom skutečná cena operace je $\mathcal{O}(\ell)$ a její amortizovaná cena je $\mathcal{O}(1)$ vzhledem k potenciálu Φ.

Důkaz: Odtržení jednoho podstromu a jeho vložení do haldy trvá konstantní čas. Skutečná cena C je tedy lineární s počtem nových stromů ℓ. FhCut udržuje invariant, že kořen stromu nikdy není označený. Každý z ℓ nových stromů možná s výjimkou prvního byl už před zahájením operace označený. Poté, co z kořenů podstromů vznikly kořeny stromů v haldě, byly odznačeny a nově označen mohl být jen jeden vrchol. Počet označených vrcholů m' po konci FhCut tedy bude $m' = m - (\ell - 1) + 1 = m - \ell + 2$. Změna potenciálu je tudíž $\Delta\Phi = c_\Phi(\ell + 2(m' - m)) = c_\Phi(\ell - 2\ell + 4) = c_\Phi(-\ell + 4)$. Skutečná cena je $C = \mathcal{O}(\ell)$, a tedy amortizovaná cena $A = \mathcal{O}(\ell) + \Delta\Phi = \mathcal{O}(1)$ za předpokladu dostatečně velké konstanty c_Φ v definici potenciálu Φ. □

Důsledek: Složitost procedury FhDecrease je stejná jako u procedury FhCut.

Důkaz: Snížení klíče obnáší kromě případného odseknutí podstromu pouze konstantní počet operací, aniž by se změnil potenciál. □

Poznámka: Po provedení FhDecrease může být v n-prvkové haldě až $\Theta(n)$ nových stromů. Tuto skutečnost přenecháváme k ověření čtenáři do cvičení 4.

Z analýzy FhDecrease a FhExtractMin také plyne analýza složitosti FhDelete.

Důsledek: Uvažme volání procedury FhDelete. Jeho skutečná cena je $\Theta(\log n + t + \ell)$, kde t je počet stromů haldy a ℓ počet stromů vzniklých při mazání vrcholu. Amortizovaná cena činí $\mathcal{O}(\log n)$ vzhledem k potenciálu Φ.

Na worst-case složitost FhDelete případě se vztahují tytéž odhady jako u FhDecrease. Pro složitost některých operací jsme zatím ukázali pouze horní asymptotický odhad. Dolní odhady worst-case složitosti FhDecrease, a tedy také FhDelete, přenecháme do cvičení 4, dolní odhady amortizované složitosti FhExtractMin, a tedy také FhDelete,

přenecháme do cvičení 5. Tím bude analýza všech operací zmíněných v úvodu tohoto oddílu kompletní.

Cvičení

1. Dokažte, že $1 + \sum_{i=0}^{d} F_i = F_{d+2}$.

2. Navrhněte operaci FhMin s konstantní časovou složitostí pomocí udržování ukazatele na nejmenší prvek v ostatních operacích. Bude třeba též znovu provést jejich (amortizovanou) časovou analýzu.

3. O zakořeněném stromu T řekneme, že má Fibonacciho vlastnost, pokud pro každý $v \in V(T)$ řádu k je $|T_v| \geq F_{k+2}$. Dokažte, že pro každé n existuje strom na n vrcholech, který má Fibonacciho vlastnost.

4. Ukažte, že existuje posloupnost $\mathcal{O}(n)$ operací na Fibonacciho haldě taková, že všech n prvků v haldě je uloženo v jediném stromu, všechny vrcholy až na jeden mají řád 1 a všechny vrcholy až na kořen jsou označené.

5. Dokažte, že při zachování ostatních operací nelze implementovat FhExtractMin v lepším než logaritmickém amortizovaném čase.

6. Bylo by možné ve Fibonacciho haldě také realizovat operaci zvýšení klíče (pro minimovou haldu) tak, aby fungovala v lepším než logaritmickém čase?

19 Těžké problémy

19 Těžké problémy

Ohlédněme se za předchozími kapitolami: pokaždé, když jsme potkali nějakou úlohu, dovedli jsme ji vyřešit algoritmem s polynomiální časovou složitostí, tedy $\mathcal{O}(n^k)$ pro pevné k. V prvním přiblížení můžeme říci, že polynomialita docela dobře vystihuje praktickou použitelnost algoritmu.[1]

Existují tedy polynomiální algoritmy pro všechny úlohy? Zajisté ne – jsou dokonce i takové úlohy, jež nelze vyřešit žádným algoritmem. Ale i mezi těmi algoritmicky řešitelnými se běžně stává, že nejlepší známé algoritmy jsou exponenciální, nebo dokonce horší.

Pojďme si pár takových příkladů předvést. Navíc uvidíme, že ačkoliv je neumíme efektivně řešit, jde mezi nimi nalézt zajímavé vztahy a pomocí nich obtížnost problémů vzájemně porovnávat. Z těchto úvah vyrůstá i skutečná teorie složitosti se svými hierarchiemi složitostních tříd. Následující kapitolu tedy můžete považovat za malou ochutnávku toho, jak se teorie složitosti buduje. Čtenářům toužícím po hlubším porozumění doporučujeme knihu Arory a Baraka [1].

19.1 Problémy a převody

Aby se nám teorie příliš nerozkošatila, omezíme své úvahy na rozhodovací problémy. To jsou úlohy, jejichž výstupem je jediný bit – máme rozhodnout, zda vstup má či nemá určitou vlastnost. Vstup přitom budeme reprezentovat řetězcem nul a jedniček – libovolnou jinou „rozumnou“ reprezentaci dokážeme na binární řetězce převést v polynomiálním čase. Za velikost vstupu budeme vždy považovat délku řetězce.

Nyní tuto představu přetavíme do exaktní definice:

Definice: *Rozhodovací problém* (zkráceně *problém*) je funkce z množiny $\{0,1\}^*$ všech řetězců nad binární abecedou do množiny $\{0,1\}$.

Ekvivalentně bychom se na problém mohli také dívat jako na nějakou množinu $A \subseteq \{0,1\}^*$ vstupů, na něž je odpověď 1. Tento přístup mají rádi v teorii automatů.

Příklad problému: *Bipartitní párování* – je dán bipartitní graf a číslo $k \in \mathbb{N}$. Máme odpovědět, zda v zadaném grafu existuje párování, které obsahuje alespoň k hran. (Je jedno, zda se ptáme na párování o alespoň k hranách nebo o právě k, protože podmnožina párování je zase párování.)

[1] Jistě vás napadne spousta protipříkladů, jako třeba algoritmus se složitostí $\mathcal{O}(1.001^n)$, který nejspíš je použitelný, ačkoliv není polynomiální, a jiný se složitostí $\mathcal{O}(n^{100})$, u kterého je tomu naopak. Ukazuje se, že tyto případy jsou velmi řídké, takže u většiny problémů náš zjednodušený pohled funguje překvapivě dobře.

Abychom vyhověli definici, musíme určit, jak celý vstup problému zapsat jedním řetězcem bitů. Nabízí se očíslovat vrcholy grafu od 1 do n, hrany popsat maticí sousednosti a požadovanou velikost množiny k zapsat dvojkově. Musíme to ale udělat opatrně, abychom poznali, kde která část kódu začíná. Třeba takto:

$$\underbrace{\langle \texttt{11}\ldots\texttt{10}\rangle}_{t}\,\underbrace{\langle n \text{ dvojkově}\rangle}_{t}\,\underbrace{\langle k \text{ dvojkově}\rangle}_{t}\,\underbrace{\langle \text{matice sousednosti}\rangle}_{n^2}$$

Počet jedniček na začátku kódu nám řekne, v kolika bitech je uložené n a k a zbytek kódu přečteme jako matici sousednosti.

Rozhodovací problém ovšem musí odpovědět na všechny řetězce bitů, nejen na ty ve správném tvaru. Dohodněme se tedy, že syntaxi vstupu budeme kontrolovat a na všechny chybně utvořené vstupy odpovíme nulou.

Jak párovací problém vyřešit? Jako správní matematici ho převedeme na nějaký, který už vyřešit umíme. To už jsme ostatně ukázali – umíme ho převést na toky v sítích. Pokaždé, když se ptáme na existenci párování velikosti alespoň k v nějakém bipartitním grafu, dovedeme sestrojit určitou síť a zeptat se, zda v této síti existuje tok velikosti alespoň k. Překládáme tedy v polynomiálním čase vstup jednoho problému na vstup jiného problému, přičemž odpověď zůstane stejná.

Podobné převody mezi problémy můžeme definovat i obecněji:

Definice: Jsou-li A, B rozhodovací problémy, říkáme, že A lze *převést* na B (píšeme $A \to B$) právě tehdy, když existuje funkce $f : \{0,1\}^* \to \{0,1\}^*$ taková, že pro všechna $x \in \{0,1\}^*$ platí $A(x) = B(f(x))$, a navíc lze funkci f spočítat v čase polynomiálním vzhledem k $|x|$. Funkci f říkáme *převod* nebo také *redukce.*

Pozorování: $A \to B$ také znamená, že problém B je alespoň tak těžký jako problém A (mnemotechnická pomůcka: obtížnost teče ve směru šipky). Tím myslíme, že kdykoliv umíme vyřešit B, je vyřešit A nanejvýš polynomiálně obtížnější. Speciálně platí:

Lemma: Pokud $A \to B$ a B lze řešit v polynomiálním čase, pak i A lze řešit v polynomiálním čase.

Důkaz: Nechť existuje algoritmus řešící problém B v čase $\mathcal{O}(b^k)$, kde b je délka vstupu tohoto problému a k konstanta. Mějme dále funkci f převádějící A na B v čase $\mathcal{O}(a^\ell)$ pro vstup délky a. Chceme-li nyní spočítat $A(x)$ pro nějaký vstup x délky a, spočítáme nejprve $f(x)$. To bude trvat $\mathcal{O}(a^\ell)$ a vyjde výstup délky taktéž $\mathcal{O}(a^\ell)$ – delší bychom v daném čase ani nestihli vypsat. Tento vstup pak předáme algoritmu pro problém B, který nad ním stráví čas $\mathcal{O}((a^\ell)^k) = \mathcal{O}(a^{k\ell})$. Celkový čas výpočtu proto činí $\mathcal{O}(a^\ell + a^{k\ell})$, což je polynom v délce původního vstupu. □

Relace převoditelnosti tedy jistým způsobem porovnává problémy podle obtížnosti. Nabízí se představa, že se jedná o uspořádání na množině všech problémů. Je tomu doopravdy tak?

Pozorování: O relaci „$\to$“ platí:

- *Je reflexivní* ($A \to A$) – úlohu můžeme převést na tutéž identickým zobrazením.
- *Je tranzitivní* ($A \to B \wedge B \to C \Rightarrow A \to C$) – pokud funkce f převádí A na B a funkce g převádí B na C, pak funkce $g \circ f$ převádí A na C. Složení dvou polynomiálně vyčíslitelných funkcí je zase polynomiálně vyčíslitelná funkce, jak už jsme zpozorovali v důkazu předchozího lemmatu.
- *Není antisymetrická* – například problémy „na vstupu je řetězec začínající nulou“ a „na vstupu je řetězec končící nulou“ lze mezi sebou převádět oběma směry.
- Existují *navzájem nepřevoditelné problémy* – třeba mezi problémy „na každý vstup odpověz 0“ a „na každý vstup odpověz 1“ nemůže existovat převod ani jedním směrem.

Relacím, které jsou reflexivní a tranzitivní, ale obecně nesplňují antisymetrii, se říká *kvaziuspořádání*. Převoditelnost je tedy částečné kvaziuspořádání na množině všech problémů.

Cvičení

1. Nahlédněte, že množina všech polynomů je nejmenší množina funkcí z $\mathbb{R}$ do $\mathbb{R}$, která obsahuje všechny konstantní funkce, identitu a je uzavřená na sčítání, násobení a skládání funkcí. Pokud tedy prohlásíme za efektivní právě polynomiální algoritmy, platí, že složením efektivních algoritmů (v mnoha možných smyslech) je zase efektivní algoritmus. To je velice příjemná vlastnost.

2.* Při kódování vstupu řetězcem bitů se často hodí umět zapsat číslo předem neznámé velikosti *instantním kódem,* tj. takovým, při jehož čtení poznáme, kdy skončil. Dvojkový zápis čísla x zabere $\lfloor \log_2 x \rfloor + 1$ bitů, ale není instantní. Kódování použité v problému párování je instantní a spotřebuje $2\lfloor \log_2 x \rfloor + \mathcal{O}(1)$ bitů. Navrhněte instantní kód, kterému stačí $\lfloor \log_2 x \rfloor + o(\log x)$ bitů. (Připomeňme, že $f \in o(g)$, pokud $\lim_{n\to\infty} f/g = 0$.)

3.* Převoditelnost je pouze kvaziuspořádání, ale můžeme z ní snadno vyrobit skutečné uspořádání: Definujeme relaci $A \sim B \equiv (A \to B) \wedge (B \to A)$. Dokažte, že je to ekvivalence, a relaci převoditelnosti zaveďte na třídách této ekvivalence. Taková převoditelnost už bude slabě antisymetrická. To je v matematice dost běžný trik, říká se mu *faktorizace* kvaziuspořádání. Vyzkoušejte si ho na relaci dělitelnosti na množině celých čísel.

19.2 Příklady převodů

Nyní se podíváme na příklady několika problémů, které se obecně považují za těžké. Uvidíme, že každý z nich je možné převést na všechny ostatní, takže z našeho „polynomiálního“ pohledu jsou stejně obtížné.

Problém SAT – splnitelnost (satisfiability) logických formulí v CNF

Mějme nějakou logickou formuli s proměnnými a logickými spojkami. Zajímá nás, je-li tato formule *splnitelná,* tedy zda lze za proměnné dosadit 0 a 1 tak, aby formule dala výsledek 1 (byla *splněna*).

Zaměříme se na formule ve speciálním tvaru, v takzvané *konjunktivní normální formě (CNF):*

- *formule* je složena z jednotlivých *klauzulí* oddělených spojkou $\wedge$,
- každá *klauzule* je složená z *literálů* oddělených $\vee$,
- každý *literál* je buďto proměnná, nebo její negace.

Vstup problému: Formule ψ v konjunktivní normální formě.

Výstup problému: Existuje-li dosazení 0 a 1 za proměnné tak, aby $\psi(\ldots) = 1$.

Příklad: Formule $(x \vee y \vee z) \wedge (\neg x \vee y \vee z) \wedge (x \vee \neg y \vee z) \wedge (x \vee y \vee \neg z)$ je splnitelná, stačí nastavit například $x = y = z = 1$ (jaká jsou ostatní splňující ohodnocení?). Naproti tomu formule $(x \vee y) \wedge (x \vee \neg y) \wedge \neg x$ splnitelná není, což snadno ověříme třeba vyzkoušením všech čtyř možných ohodnocení.

Poznámka: Co kdybychom chtěli zjistit, zda je splnitelná nějaká formule, která není v CNF? V logice se dokazuje, že ke každé formuli lze najít ekvivalentní formuli v CNF, ale při tom se bohužel formule může až exponenciálně prodloužit. Později ukážeme, že pro každou formuli χ existuje nějaká formule χ' v CNF, která je splnitelná právě tehdy, když je χ splnitelná. Formule χ' přitom bude dlouhá $\mathcal{O}(|\chi|)$, ale budou v ní nějaké nové proměnné.

Problém 3-SAT – splnitelnost formulí s krátkými klauzulemi

Pro SAT zatím není známý žádný polynomiální algoritmus. Co kdybychom zkusili problém trochu zjednodušit a uvažovat pouze formule ve speciálním tvaru?

Povolíme tedy na vstupu pouze takové formule v CNF, jejichž každá klauzule obsahuje nejvýše tři literály. Ukážeme, že tento problém je stejně těžký jako původní SAT.

Převod 3-SAT $\rightarrow$ SAT: Jelikož 3-SAT je speciálním případem SATu, poslouží tu jako převodní funkce identita. (Implicitně předpokládáme, že oba problémy používají stejné kódování formulí do řetězců bitů.)

Převod SAT → 3-SAT: Nechť se ve formuli vyskytuje nějaká „dlouhá“ klauzule o $k > 3$ literálech. Můžeme ji zapsat ve tvaru $(\alpha \vee \beta)$, kde α obsahuje 2 literály a β $k-2$ literálů. Pořídíme si novou proměnnou x a klauzuli nahradíme dvěma novými $(\alpha \vee x)$ a $(\beta \vee \neg x)$. První z nich obsahuje 3 literály, tedy je krátká. Druhá má $k-1$ literálů, takže může být stále dlouhá, nicméně postup můžeme opakovat.

Takto postupně nahradíme všechny dlouhé klauzule krátkými, což bude trvat nejvýše polynomiálně dlouho, neboť klauzuli délky k rozebereme po $k-3$ krocích.

Zbývá ukázat, že nová formule je splnitelná právě tehdy, byla-li splnitelná formule původní. K tomu stačí ukázat, že každý jednotlivý krok převodu splnitelnost zachovává.

Pokud původní formule byla splnitelná, uvažme nějaké splňující ohodnocení proměnných. Ukážeme, že vždy můžeme novou proměnnou x nastavit tak, aby vzniklo splňující ohodnocení nové formule. Víme, že klauzule $(\alpha \vee \beta)$ byla splněna. Proto v daném ohodnocení:

- Buďto $\alpha = 1$. Pak položíme $x = 0$, takže $(\alpha \vee x)$ bude splněna díky α a $(\beta \vee \neg x)$ díky x.
- Anebo $\alpha = 0$, a tedy $\beta = 1$. Pak položíme $x = 1$, čímž bude $(\alpha \vee x)$ splněna díky x, zatímco $(\beta \vee \neg x)$ díky β.

Ostatní klauzule budou stále splněny.

V opačném směru: pokud dostaneme splňující ohodnocení nové formule, umíme z něj získat splňující ohodnocení formule původní. Ukážeme, že stačí zapomenout proměnnou x. Všechny klauzule, kterých se naše transformace netýká, jsou nadále splněné. Co klauzule $(\alpha \vee \beta)$?

- Buďto $x = 0$, pak musí být $(\alpha \vee x)$ splněna díky α, takže $(\alpha \vee \beta)$ je také splněna díky α.
- Anebo $x = 1$, pak musí být $(\beta \vee \neg x)$ splněna díky β, takže i $(\alpha \vee \beta)$ je splněna.

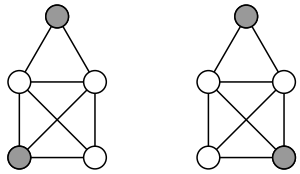
Tím je převod hotov. SAT a 3-SAT jsou tedy stejně těžké.

Problém NzMna – nezávislá množina vrcholů v grafu

Definice: Množina vrcholů grafu je *nezávislá,* pokud žádné dva vrcholy ležící v této množině nejsou spojeny hranou. (Jinými slovy nezávislá množina indukuje podgraf bez hran.)

Na samotnou existenci nezávislé množiny se nemá smysl ptát – prázdná množina či libovolný jeden vrchol jsou vždy nezávislé. Zajímavé ale je, jestli graf obsahuje dostatečně velkou nezávislou množinu.

Vstup problému: Neorientovaný graf G a číslo $k \in \mathbb{N}$.



Obrázek 19.1: Největší nezávislé množiny

Výstup problému: Zda existuje nezávislá množina $A \subseteq V(G)$ velikosti alespoň k.

Převod 3-SAT $\rightarrow$ NzMna: Dostaneme formuli a máme vytvořit graf, v němž se bude nezávislá množina určené velikosti nacházet právě tehdy, je-li formule splnitelná. Myšlenka převodu bude jednoduchá: z každé klauzule budeme chtít vybrat jeden literál, jehož nastavením klauzuli splníme. Samozřejmě si musíme dát pozor, abychom v různých klauzulích nevybírali konfliktně, tj. jednou x a podruhé $\neg x$.

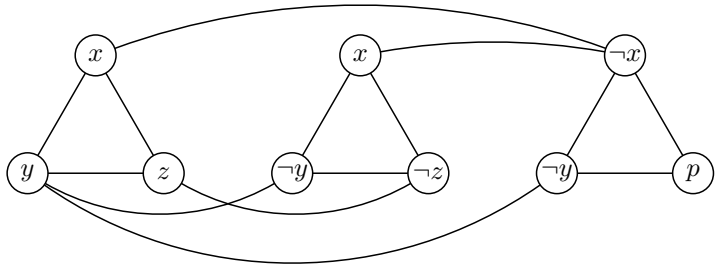
Jak to přesně zařídit: pro každou z k klauzulí zadané formule vytvoříme trojúhelník a jeho vrcholům přiřadíme literály klauzule. (Pokud by klauzule obsahovala méně literálů, prostě některé vrcholy trojúhelníka smažeme.) Navíc spojíme hranami všechny dvojice konfliktních literálů (x a $\neg x$) z různých trojúhelníků.

V tomto grafu se budeme ptát po nezávislé množině velikosti alespoň k. Jelikož z každého trojúhelníka můžeme do nezávislé množiny vybrat nejvýše jeden vrchol, jediná možnost, jak dosáhnout požadované velikosti, je vybrat z každého právě jeden vrchol. Ukážeme, že taková nezávislá množina existuje právě tehdy, je-li formule splnitelná.

Máme-li splňující ohodnocení formule, můžeme z každé klauzule vybrat jeden pravdivý literál. Do nezávislé množiny umístíme vrcholy odpovídající těmto literálům. Je jich právě k. Jelikož každé dva vybrané vrcholy leží v různých trojúhelnících a nikdy nemůže být pravdivý současně literál a jeho negace, množina je opravdu nezávislá.

A opačně: Kdykoliv dostaneme nezávislou množinu velikosti k, vybereme literály odpovídající vybraným vrcholům a příslušné proměnné nastavíme tak, abychom tyto literály splnili. Díky hranám mezi konfliktními literály se nikdy nestane, že bychom potřebovali proměnnou nastavit současně na 0 a na 1. Zbývající proměnné ohodnotíme libovolně. Jelikož jsme v každé klauzuli splnili alespoň jeden literál, jsou splněny všechny klauzule, a tedy i celá formule.

Převod je tedy korektní, zbývá rozmyslet si, že běží v polynomiálním čase: Počet vrcholů grafu odpovídá počtu literálů ve formuli, počet hran je maximálně kvadratický. Každý vrchol i hranu přitom sestrojíme v polynomiálním čase, takže celý převod je také polynomiální.



Obrázek 19.2: Graf pro formuli (x ∨ y ∨ z) ∧ (x ∨ ¬y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ p)

Převod NzMna → SAT: Dostaneme graf a číslo k, chceme vytvořit formuli, která je splnitelná právě tehdy, pokud se v grafu nachází nezávislá množina o alespoň k vrcholech. Tuto formuli sestrojíme následovně.

Vrcholy grafu očíslujeme od 1 do n a pořídíme si pro ně proměnné $v_1, \ldots, v_n$, které budou indikovat, zda byl příslušný vrchol vybrán do nezávislé množiny (příslušné ohodnocení proměnných tedy bude odpovídat charakteristické funkci nezávislé množiny).

Aby množina byla opravdu nezávislá, pro každou hranu $ij \in E(G)$ přidáme klauzuli $(\neg v_i \vee \neg v_j)$.

Ještě potřebujeme zkontrolovat, že množina je dostatečně velká. To neumíme provést přímo, ale použijeme lest: vyrobíme matici proměnných $\mathbf{X}$ tvaru $k \times n$, která bude popisovat očíslování vrcholů nezávislé množiny čísly od 1 do k. Konkrétně $x_{i,j}$ bude říkat, že v pořadí i-tý prvek nezávislé množiny je vrchol j. K tomu potřebujeme zařídit:

- Aby v každém sloupci byla nejvýše jedna jednička. Na to si pořídíme klauzule $(x_{i,j} \Rightarrow \neg x_{i',j})$ pro $i' \neq i$. (Jsou to implikace, ale můžeme je zapsat i jako disjunkce, protože $a \Rightarrow b$ je totéž jako $\neg a \vee b$.)
- Aby v každém řádku ležela právě jedna jednička. Nejprve zajistíme nejvýše jednu klauzulemi $(x_{i,j} \Rightarrow \neg x_{i,j'})$ pro $j' \neq j$. Pak přidáme klauzule $(x_{i,1} \vee x_{i,2} \vee \ldots \vee x_{i,n})$, které požadují alespoň jednu jedničku v řádku.
- Vztah mezi očíslováním a nezávislou množinou: přidáme klauzule $x_{i,j} \Rightarrow v_j$. (Všimněte si, že nezávislá množina může obsahovat i neočíslované prvky, ale to nám nevadí. Důležité je, aby jich měla k očíslovaných.)

Správnost převodu je zřejmá, ověřme ještě, že probíhá v polynomiálním čase. To plyne z toho, že vytvoříme polynomiálně mnoho klauzulí a každou z nich stihneme vypsat v lineárním čase.

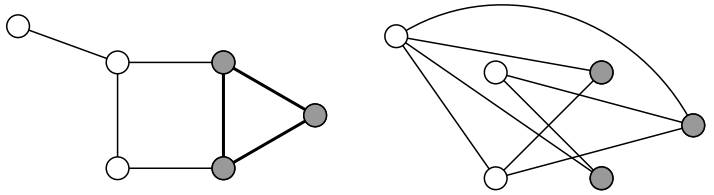
Dokázali jsme tedy, že testování existence nezávislé množiny je stejně těžké jako testování splnitelnosti formule. Pojďme se podívat na další problémy.

Problém Klika – úplný podgraf

Podobně jako nezávislou množinu můžeme v grafu hledat i *kliku* – úplný podgraf dané velikosti.

Vstup problému: Graf G a číslo $k \in N$.

Výstup problému: Existuje-li úplný podgraf grafu G na alespoň k vrcholech.



Obrázek 19.3: Klika v grafu a nezávislá množina v jeho doplňku

Tento problém je ekvivalentní s hledáním nezávislé množiny. Pokud v grafu prohodíme hrany a nehrany, stane se z každé kliky nezávislá množina a naopak. Převodní funkce tedy zneguje hrany a ponechá číslo k.

Problém 3,3-SAT – splnitelnost s malým počtem výskytů

Než se pustíme do dalšího kombinatorického problému, předvedeme ještě jednu speciální variantu SATu, se kterou se nám bude pracovat příjemněji.

Již jsme ukázali, že SAT zůstane stejně těžký, omezíme-li se na formule s klauzulemi délky nejvýše 3. Teď budeme navíc požadovat, aby se každá proměnná vyskytovala v maximálně třech literálech. Tomuto problému se říká 3,3-SAT.

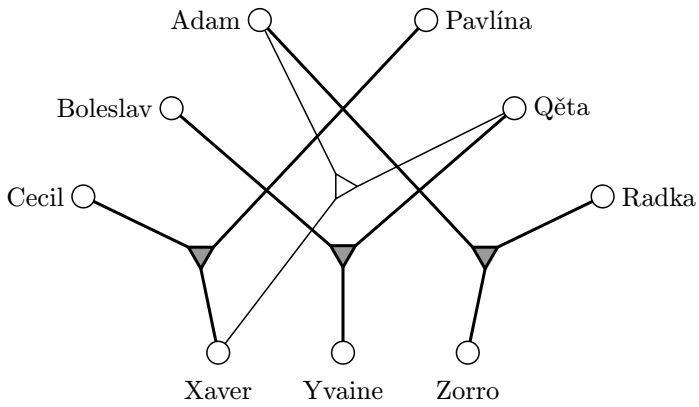
Převod 3-SAT → 3,3-SAT: Pokud se proměnná x vyskytuje v $k > 3$ literálech, nahradíme její výskyty novými proměnnými $x_1, \ldots, x_k$ a přidáme klauzule, které zabezpečí, že tyto proměnné budou pokaždé ohodnoceny stejně: $(x_1 \Rightarrow x_2)$, $(x_2 \Rightarrow x_3)$, $(x_3 \Rightarrow x_4)$, $\ldots$, $(x_{k-1} \Rightarrow x_k)$, $(x_k \Rightarrow x_1)$.

Zesílení: Můžeme dokonce zařídit, aby se každý literál vyskytoval nejvýše dvakrát (tedy že každá proměnná se vyskytuje alespoň jednou pozitivně a alespoň jednou negativně). Pokud by se nějaká proměnná objevila ve třech stejných literálech, můžeme na ni také použít náš trik a nahradit ji třemi proměnnými. V nových klauzulích se pak bude vyskytovat jak pozitivně, tak negativně (opět připomínáme, že $a \Rightarrow b$ je jen zkratka za $\neg a \vee b$).

Problém 3D-párování

Vstup problému: Tři množiny, např. K (kluci), H (holky), Z (zvířátka) a množina $T \subseteq K \times H \times Z$ kompatibilních trojic (těch, kteří se spolu snesou).

Výstup problému: Zda existuje perfektní podmnožina trojic, tedy taková, v níž se každý prvek množin K, H a Z účastní právě jedné trojice.



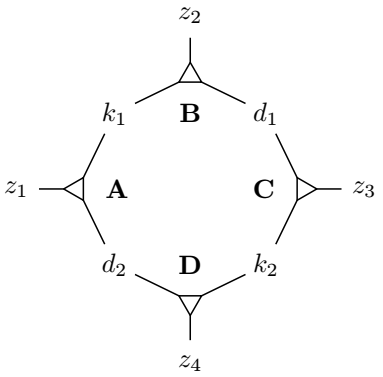
Obrázek 19.4: 3D-párování

Převod 3,3-SAT → 3D-párování: Uvažujme trochu obecněji. Pokud chceme ukázat, že se na nějaký problém dá převést SAT, potřebujeme obvykle dvě věci: Jednak konstrukci, která bude simulovat proměnné, tedy něco, co nabývá dvou stavů 0/1. Poté potřebujeme cosi, co umí zařídit, aby každá klauzule byla splněna alespoň jednou proměnnou. Jak to provést u 3D-párování?

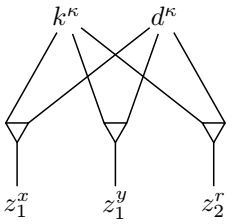
Uvažujme konfiguraci z obrázku 19.5. V ní se nacházejí 4 zvířátka (z_1 až z_4), 2 kluci (k_1 a k_2), 2 dívky (d_1 a d_2) a 4 trojice (A, B, C a D). Zatímco zvířátka se budou moci účastnit i jiných trojic, kluky a děvčata nikam jinam nezapojíme.

Všimneme si, že existují právě dvě možnosti, jak tuto konfiguraci spárovat. Abychom spárovali kluka k_1, tak musíme vybrat buď trojici A nebo B. Pokud si vybereme A, k_1 i d_2 už jsou spárovaní, takže si nesmíme vybrat B ani D. Pak jediná možnost, jak spárovat d_1 a k_2, je použít C. Naopak začneme-li trojicí B, vyloučíme A a C a použijeme D (situace je symetrická).

Vždy si tedy musíme vybrat dvě protější trojice v obrázku a druhé dvě nechat nevyužité. Tyto možnosti budeme používat k reprezentaci proměnných. Pro každou proměnnou si pořídíme jednu kopii obrázku. Volba $A + C$ bude odpovídat nule a nespáruje zvířátka



Obrázek 19.5: Konfigurace pro proměnnou



Obrázek 19.6: Konfigurace pro klauzuli

z_2 a z_4. Volba $B + D$ reprezentuje jedničku a nespáruje z_1 a z_3. Přes tato nespárovaná zvířátka můžeme předávat informaci o hodnotě proměnné do klauzulí.

Zbývá vymyslet, jak reprezentovat klauzule. Mějme klauzuli tvaru řekněme $(x \vee y \vee \neg r)$. Potřebujeme zajistit, aby x bylo nastavené na 1 nebo y bylo nastavené na 1 nebo r na 0.

Pro takovouto klauzuli přidáme konfiguraci z obrázku 19.6. Pořídíme si novou dvojici kluk a dívka, kteří budou figurovat ve třech trojicích se třemi různými zvířátky, což budou volná zvířátka z obrázků pro příslušné proměnné. Zvolíme je tak, aby se uvolnila při správném nastavení proměnné. Žádné zvířátko přitom nebude vystupovat ve více klauzulích, což můžeme splnit díky tomu, že každý literál se v 3,3-SATu vyskytuje nejvýše dvakrát a máme pro něj dvě volná zvířátka.

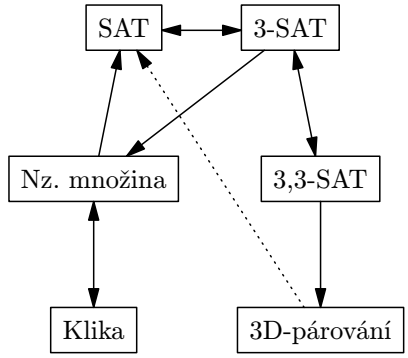
Ještě nám určitě zbude $4p - 3k$ zvířátek, kde p je počet proměnných a k počet klauzulí. Každá proměnná totiž dodá 4 volná zvířátka a každá klauzule použije 3 z nich. Přidáme proto ještě $4p - 3k$ párů lidí, kteří milují úplně všechna zvířátka; ti vytvoří zbývající trojice. Pokud je nějaká klauzule kratší než 3 literály, $3k$ se příslušně sníží.

Snadno ověříme, že celý převod pracuje v polynomiálním čase. Rozmysleme si ještě, že je korektní. Pokud formule byla splnitelná, z každého splňujícího ohodnocení můžeme vyrobit párování v naší konstrukci. Obrázek pro každou proměnnou spárujeme podle ohodnocení (buď $A + C$ nebo $B + D$). Pro každou klauzuli si vybereme trojici, která odpovídá některému z literálů, jimiž je klauzule splněna.

A opačně: Když nám někdo dá párování v naší konstrukci, dokážeme z něj vyrobit splňující ohodnocení dané formule. Podíváme se, v jakém stavu je proměnná, a to je všechno. Z toho, že jsou správně spárované klauzule, už okamžitě víme, že jsou všechny splněné.

Ukázali jsme tedy, že na 3D-párování lze převést 3,3-SAT, a tedy i obecný SAT. Převod v opačném směru ponecháme jako cvičení 2.

Jak je vidět ze schématu (obr. 19.7), všechny problémy z tohoto oddílu jsou navzájem převoditelné.



Obrázek 19.7: Problémy a převody mezi nimi

Cvičení

1. Domyslete detaily kódování vstupu pro libovolný z problémů z tohoto oddílu.

2. Převeďte 3D-párování na SAT. Jde to podobně, jako když jsme na SAT převáděli nezávislou množinu.

3. Co jsme ztratili omezením na rozhodovací problémy? Dokažte pro libovolný problém z tohoto oddílu, že pokud bychom ho dokázali v polynomiálním čase vyřešit, uměli bychom polynomiálně řešit i „zjišťovací“ verzi (najít konkrétní párování, splňující ohodnocení, kliku apod.).

   Jak na to, ukážeme na párování: Chceme v grafu G najít párování velikosti k. Zvolíme libovolnou hranu e a otestujeme (zavoláním rozhodovací verze problému), zda i v grafu $G - e$ existuje párování velikosti k. Pokud ano, můžeme hranu e smazat a pokračovat dál. Pokud ne, znamená to, že hrana e leží ve všech párováních velikosti k, takže si ji zapamatujeme, smažeme z grafu včetně krajních vrcholů a všech incidentních hran, načež snížíme k o 1. Tak postupně získáme všech k hran párování.

4. *Vrcholové pokrytí* grafu je množina vrcholů, která obsahuje alespoň jeden vrchol z každé hrany. (Chceme na křižovatky rozmístit strážníky tak, aby každou ulici alespoň jeden hlídal.) Ukažte vzájemné převody mezi problémem nezávislé množiny a problémem „Existuje vrcholové pokrytí velikosti nejvýše k?“.

5. Zesilte náš převod SATu na nezávislou množinu tak, aby vytvářel grafy, jejichž všechny vrcholy mají stupeň nejvýše 4.

19.3 NP-úplné problémy

Všechny problémy, které jsme zatím zkoumali, měly jednu společnou vlastnost. Šlo v nich o to, zda existuje nějaký objekt. Například splňující ohodnocení formule nebo klika v grafu. Kdykoliv nám přitom někdo takový objekt ukáže, umíme snadno ověřit, že má požadovanou vlastnost. Ovšem najít ho už tak snadné není. Podobně se chovají i mnohé další „vyhledávací problémy“, zkusme je tedy popsat obecněji.

Definice: P je třída[2] rozhodovacích problémů, které jsou řešitelné v polynomiálním čase. Jinak řečeno, problém L leží v P právě tehdy, když existuje nějaký algoritmus A a polynom f, přičemž pro každý vstup x algoritmus A doběhne v čase nejvýše $f(|x|)$ a vydá výsledek $A(x) = L(x)$.

Třída P tedy zachycuje naši představu o efektivně řešitelných problémech. Nyní definujeme třídu NP, která bude odpovídat naší představě vyhledávacích problémů.

Definice: NP je třída rozhodovacích problémů, v níž problém L leží právě tehdy, pokud existuje nějaký problém $K \in \mathsf{P}$ a polynom g, přičemž pro každý vstup x je $L(x) = 1$ právě tehdy, pokud pro nějaký řetězec y délky nejvýše $g(|x|)$ platí $K(x, y) = 1$.[3]

Co to znamená? Algoritmus K řeší problém L, ale kromě vstupu x má k dispozici ještě polynomiálně dlouhou *nápovědu* y. Přitom má platit, že je-li $L(x) = 1$, musí existovat alespoň jedna nápověda, kterou algoritmus K schválí. Pokud ovšem $L(x) = 0$, nesmí ho přesvědčit žádná nápověda.

Jinými slovy y je jakýsi *certifikát,* který stvrzuje kladnou odpověď, a problém K má za úkol certifikáty kontrolovat. Pro kladnou odpověď musí existovat alespoň jeden schválený certifikát, pro zápornou musí být všechny certifikáty odmítnuty.

To je podobné dokazování v matematice: k pravdivému tvrzení by měl existovat důkaz, u nepravdivého nás jakýkoliv „důkaz“ nepřesvědčí.

Příklad: Splnitelnost logických formulí je v NP. Stačí si totiž nechat napovědět, jak ohodnotit jednotlivé proměnné, a pak ověřit, je-li formule splněna. Nápověda je polynomiálně velká (dokonce lineárně), splnění zkontrolujeme také v lineárním čase. Podobně to lze dokázat i o ostatních rozhodovacích problémech, se kterými jsme v minulém oddílu potkali.

[2] Formálně vzato je to množina, ale v teorii složitosti se pro množiny problémů vžil název *třídy.*

[3] Rozhodovací problémy mají na vstupu řetězec bitů. Tak jaképak x, y? Máme samozřejmě na mysli nějaké binární kódování této dvojice.

Pozorování: Třída P leží uvnitř NP. Pokud totiž problém umíme řešit v polynomiálním čase bez nápovědy, tak to zvládneme v polynomiálním čase i s nápovědou. Algoritmus K tedy bude ignorovat nápovědy a odpověď spočítá přímo ze vstupu.

Nevíme ale, zda jsou třídy P a NP skutečně různé. Na to se teoretičtí informatici snaží přijít už od 70. let minulého století a postupně se z toho stal nejspíš nejslavnější otevřený problém informatiky.

Například pro žádný problém z předchozího oddílu nevíme, zda leží v P. Povede se nám ale dokázat, že tyto problémy jsou v jistém smyslu ty nejtěžší v NP.

Definice: Problém L nazveme NP-*těžký*, je-li na něj převoditelný každý problém z NP. Pokud navíc L leží v NP, budeme říkat, že L je NP-*úplný*.

Lemma: Pokud nějaký NP-těžký problém L leží v P, pak $\mathsf{P} = \mathsf{NP}$.

Důkaz: Již víme, že $\mathsf{P} \subseteq \mathsf{NP}$, takže stačí dokázat opačnou inkluzi. Vezměme libovolný problém $A \in \mathsf{NP}$. Z NP-těžkosti problému L plyne $A \to L$. Už jsme ale dříve dokázali (lemma na straně 432), že pokud $L \in \mathsf{P}$ a $A \to L$, pak také $A \in \mathsf{P}$. □

Existují ale vůbec nějaké NP-úplné problémy? Na první pohled zní nepravděpodobně, že by na nějaký problém z NP mohly jít převést všechny ostatní. Stephen Cook ale v roce 1971 dokázal následující překvapivou větu:

Věta (Cookova): SAT je NP-úplný.

Důkaz této věty je značně technický a alespoň v hrubých rysech ho předvedeme v příštím oddílu. Teď především ukážeme, že jakmile známe jeden NP-úplný problém, můžeme pomocí převoditelnosti dokazovat i NP-úplnost dalších.

Lemma: Mějme dva problémy $L, M \in \mathsf{NP}$. Pokud L je NP-úplný a $L \to M$, pak M je také NP-úplný. (Intuitivně: Pokud L je nejtěžší v NP a $M \in \mathsf{NP}$ je alespoň tak těžký jako L, pak M je také nejtěžší v NP.)

Důkaz: Jelikož M leží v NP, stačí o něm dokázat, že je NP-těžký, tedy že na něj lze převést libovolný problém z NP. Uvažme tedy nějaký problém $Q \in \mathsf{NP}$. Jelikož L je NP-úplný, musí platit $Q \to L$. Převoditelnost je ovšem tranzitivní, takže z $Q \to L$ a $L \to M$ plyne $Q \to M$. □

Důsledek: Všechny problémy z minulého oddílu jsou NP-úplné.

Poznámka (o dvou možných světech): Jestli je $\mathsf{P} = \mathsf{NP}$, to nevíme a nejspíš ještě dlouho vědět nebudeme. Nechme se ale na chvíli unášet fantazií a zkusme si představit, jak by vypadaly světy, v nichž platí jedna nebo druhá možnost.

- $\mathsf{P} = \mathsf{NP}$ – to je na první pohled idylický svět, v němž jde každý vyhledávací problém vyřešit v polynomiálním čase, nejspíš tedy i prakticky efektivně. Má to i své stinné stránky: například jsme přišli o veškeré efektivní šifrování – rozmyslete si, že pokud umíme vypočítat nějakou funkci v polynomiálním čase, umíme efektivně spočítat i její inverzi.
- $\mathsf{P} \neq \mathsf{NP}$ – tehdy jsou P a NP-úplné dvě disjunktní třídy. SAT a ostatní NP-úplné problémy nejsou řešitelné v polynomiálním čase. Je ale stále možné, že aspoň na některé z nich existují prakticky použitelné algoritmy, třeba o složitosti $\Theta((1+\varepsilon)^n)$ nebo $\Theta(n^{\log n/100})$. Také platí (tomu se říká Ladnerova věta [1]), že třída NP obsahuje i problémy, které svou obtížností leží někde mezi P a NP-úplnými.

Katalog NP-úplných problémů

Pokud se setkáme s problémem, který neumíme zařadit do P, hodí se vyzkoušet, zda je NP-úplný. K tomu se hodí mít alespoň základní zásobu „učebnicových“ NP-úplných problémů, abychom si mohli vybrat, z čeho převádět. U některých jsme už NP-úplnost dokázali, u ostatních alespoň naznačíme, jak na to.

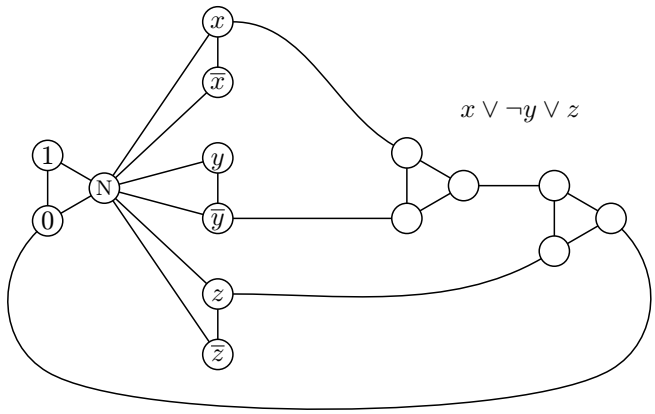
- *Logické problémy:*
  - *SAT:* splnitelnost logických formulí v CNF
  - *3-SAT:* každá klauzule obsahuje max. 3 literály
  - *3,3-SAT:* navíc se každá proměnná vyskytuje nejvýše třikrát
  - *SAT pro obecné formule:* nejen v CNF (viz oddíl 19.4)
  - *Obvodový SAT:* splnitelnost booleovského obvodu (viz oddíl 19.4)
- *Grafové problémy:*
  - *Nezávislá množina:* existuje množina alespoň k vrcholů, mezi nimiž nevede žádná hrana?
  - *Klika:* existuje úplný podgraf na k vrcholech?
  - *Barvení grafu:* lze obarvit vrcholy k barvami (přidělit každému vrcholu číslo od 1 do k) tak, aby vrcholy stejné barvy nebyly nikdy spojeny hranou)? To je NP-úplné už pro $k = 3$, viz cvičení 2 a 3.
  - *Hamiltonovská cesta:* existuje cesta obsahující všechny vrcholy? (cvičení 4 a 5)
  - *Hamiltonovská kružnice:* existuje kružnice obsahující všechny vrcholy? (cvičení 4)
  - *3D-párování:* máme tři množiny se zadanými trojicemi; zjistěte, zda existuje taková množina disjunktních trojic, ve které jsou všechny prvky právě jednou? (Striktně vzato, není to grafový problém, ale hypergrafový – hrany nejsou páry, ale trojice.)

- *Číselné problémy:*
  - *Součet podmnožiny:* má daná množina přirozených čísel podmnožinu s daným součtem? (cvičení 7)
  - *Batoh:* jsou dány předměty s váhami a cenami a kapacita batohu, chceme najít co nejdražší podmnožinu předmětů, jejíž váha nepřesáhne kapacitu batohu. Aby se jednalo o rozhodovací problém, ptáme se, zda existuje podmnožina s cenou větší nebo rovnou zadanému číslu. (cvičení 8)
  - *Dva loupežníci:* lze rozdělit danou množinu čísel na dvě podmnožiny se stejným součtem? (cvičení 6)
  - $\mathbf{Ax} = \mathbf{1}$ *(soustava nula-jedničkových lineárních rovnic):* je dána matice $\mathbf{A} \in \{0,1\}^{m \times n}$. Existuje vektor $\mathbf{x} \in \{0,1\}^n$ takový, že $\mathbf{Ax}$ je rovno vektoru samých jedniček? (cvičení 1)

Cvičení

1. Dokažte NP-úplnost problému $\mathbf{Ax} = \mathbf{1}$.

2.* Dokažte NP-úplnost problému barvení grafu třemi barvami převodem z 3-SATu. Inspirujte se obrázkem 19.8.



Obrázek 19.8: K důkazu NP-úplnosti barvení grafu

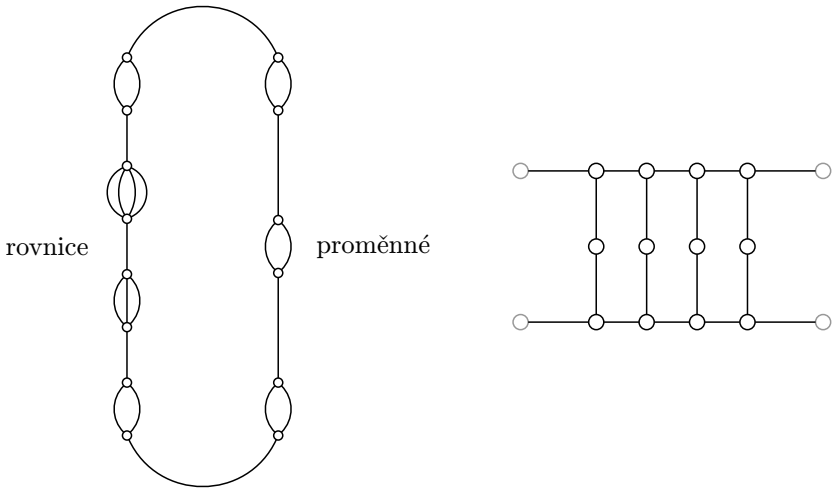
3. Ukažte, že barvení grafu jednou nebo dvěma barvami leží v P.

4.* Dokažte NP-úplnost problému hamiltonovské kružnice podle následujícího návodu. Nejprve budeme převádět problém $\mathbf{Ax} = \mathbf{1}$ na speciální variantu hamiltonovské

kružnice. V té budou povoleny paralelní hrany a bude možné dvojice hran *párovat* – určit, že hledaná kružnice projde právě jednou hranou z dvojice.

Pro danou soustavu rovnic sestrojíme graf podle obrázku 19.9 vlevo. Pro každou proměnnou pořídíme dvě paralelní hrany, které budou odpovídat jejím možným hodnotám. Pro každou rovnici pořídíme paralelní hrany odpovídající jednotlivým výskytům proměnných (s nenulovým koeficientem). Nakonec spárujeme každý výskyt proměnné v rovnici s nulovým stavem příslušné proměnné.

Dokažte, že v tomto grafu existuje párovací hamiltonovská kružnice právě tehdy, má-li soustava rovnic řešení. Vymyslete, jak se zbavit paralelních hran a požadavků na párování. K tomu se může hodit rozmyslet si, jak může hamiltonovská kružnice procházet podgrafem z obrázku 19.9 vpravo. A nakonec převod upravte pro hamiltonovskou cestu.



Obrázek 19.9: K důkazu NP-úplnosti hamiltonovské kružnice

5. Uvažujme variantu problému hamiltonovské cesty, v níž máme pevně určené krajní vrcholy cesty. Ukažte, jak tento problém převést na hamiltonovskou kružnici. Ukažte též opačný převod.

6. Převeďte součet podmnožiny na dva loupežníky a opačně.

7. Dokažte NP-úplnost problému součtu podmnožiny.

8. Dokažte NP-úplnost problému batohu.

9. Pokud bychom definovali P-úplnost analogicky k NP-úplnosti, které problémy z P by byly P-úplné?

10. Převeďte libovolný problém z katalogu na SAT, aniž byste použili Cookovu větu.

11. *Kvadratické rovnice:* Převeďte SAT na řešitelnost soustavy kvadratických rovnic více proměnných, tedy rovnic tvaru

$$\sum_i \alpha_i x_i^2 + \sum_{i,j} \beta_{ij} x_i x_j + \sum_i \gamma_i x_i + \delta = 0,$$

kde $x_1, \ldots, x_n$ jsou reálné neznámé a řecká písmena značí celočíselné konstanty. (Všimněte si, že vůbec není jasné, zda tento problém leží v NP.)

19.4* Důkaz Cookovy věty

Zbývá dokázat Cookovu větu. Potřebujeme ukázat, že SAT je NP-úplný, a to přímo z definice NP-úplnosti. Nejprve se nám to povede pro jiný problém, pro takzvaný *obvodový SAT*. V něm máme na vstupu booleovský obvod (hradlovou síť) s jedním výstupem a ptáme se, zda můžeme přivést na vstupy obvodu takové hodnoty, aby vydal výsledek 1. To je obecnější než SAT pro formule (dokonce i neomezíme-li formule na CNF), protože každou formuli můžeme přeložit na lineárně velký obvod (cvičení 15.1.11).

Nejprve tedy dokážeme NP-úplnost obvodového SATu a pak ho převedeme na obyčejný SAT v CNF. Tím bude důkaz Cookovy věty hotov. Začněme lemmatem, v němž bude koncentrováno vše technické.

Budeme se snažit ukázat, že pro každý problém v P existuje polynomiálně velká hradlová síť, která ho řeší. Jenom si musíme dát pozor na to, že pro různé velikosti vstupu potřebujeme různé hradlové sítě, které navíc musíme umět efektivně generovat.

Lemma: Nechť L je problém ležící v P. Potom existuje polynom p a algoritmus, který pro každé n sestrojí v čase $p(n)$ hradlovou síť B_n s n vstupy a jedním výstupem, která řeší L. Tedy pro všechny řetězce $x \in \{0,1\}^n$ musí platit $B_n(x) = L(x)$.

Náznak důkazu: Vyjdeme z intuice o tom, že počítače jsou jakési složité booleovské obvody, jejichž stav se mění v čase. (Formálněji bychom konstruovali booleovský obvod simulující výpočetní model RAM.)

Uvažme tedy nějaký problém $L \in \mathsf{P}$ a polynomiální algoritmus, který ho řeší. Pro vstup velikosti n algoritmus doběhne v čase T polynomiálním v n a spotřebuje $\mathcal{O}(T)$ buněk paměti. Stačí nám tedy „počítač s pamětí velkou $\mathcal{O}(T)$“, což je nějaký booleovský obvod

velikosti polynomiální v T, a tedy i v n. Vývoj v čase ošetříme tak, že sestrojíme T kopií tohoto obvodu, každá z nichž bude odpovídat jednomu kroku výpočtu a bude propojena s „minulou“ a „budoucí“ kopií. Tím sestrojíme booleovský obvod, který bude řešit problém L pro vstupy velikosti n a bude polynomiálně velký vzhledem k n. □

Úprava definice NP: Pro důkaz následující věty si dovolíme drobnou úpravu v definici třídy NP. Budeme chtít, aby nápověda měla pevnou velikost, závislou pouze na velikosti vstupu (tedy: $|y| = g(|x|)$ namísto $|y| \leq g(|x|)$). Proč je taková úprava bez újmy na obecnosti? Stačí původní nápovědu doplnit na požadovanou délku nějakými „mezerami“, které budeme při ověřování nápovědy ignorovat. Podobně můžeme zaokrouhlit koeficienty polynomu g na celá čísla, aby ho bylo možné vyhodnotit v konstantním čase.

Věta (téměř Cookova): Obvodový SAT je NP-úplný.

Důkaz: Obvodový SAT evidentně leží v NP – stačí si nechat poradit vstup, síť topologicky setřídit a v tomto pořadí počítat hodnoty hradel.

Mějme nyní nějaký problém L z NP, o němž chceme dokázat, že se dá převést na obvodový SAT. Když nám někdo předloží nějaký vstup x délky n, spočítáme velikost nápovědy $g(n)$. Víme, že algoritmus K, který kontroluje, zda nápověda je správně, leží v P. Využijeme předchozí lemma, abychom získali obvod, který pro konkrétní velikost vstupu n počítá to, co kontrolní algoritmus K. Vstupem tohoto obvodu bude x (vstup problému L) a nápověda y. Na výstupu se dozvíme, zda je nápověda správná. Velikost tohoto obvodu bude činit $p(g(n))$, což je také polynom.

V tomto obvodu zafixujeme vstup x (na místa vstupu dosadíme konkrétní hodnoty z x). Tím získáme obvod, jehož vstup je jen y, a chceme zjistit, zda za y můžeme dosadit nějaké hodnoty tak, aby na výstupu byla 1. Jinými slovy, ptáme se, zda je tento obvod splnitelný.

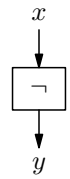
Ukázali jsme tedy, že pro libovolný problém z NP dokážeme sestrojit funkci, která pro každý vstup x v polynomiálním čase vytvoří obvod, jenž je splnitelný pravě tehdy, když odpověď tohoto problému na vstup x má být kladná. To je přesně převod z daného problému na obvodový SAT. □

Lemma: Obvodový SAT se dá převést na 3-SAT.

Důkaz: Budeme postupně budovat formuli v konjunktivní normální formě. Každý booleovský obvod se dá v polynomiálním čase převést na ekvivalentní obvod, ve kterém se vyskytují jen hradla AND a NOT (cvičení 15.1.3), takže stačí najít klauzule odpovídající těmto hradlům. Pro každé hradlo v obvodu zavedeme novou proměnnou popisující jeho výstup. Přidáme klauzule, které nám kontrolují, že toto hradlo máme ohodnocené konzistentně.

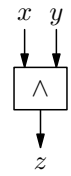
Převod hradla NOT: Na vstupu hradla budeme mít nějakou proměnnou x (která přišla buďto přímo ze vstupu celého obvodu, nebo je to výstup nějakého jiného hradla) a na výstupu proměnnou y. Přidáme klauzule, které nám zaručí, že jedna proměnná bude negací té druhé:

$$(x \vee y)$$
$$(\neg x \vee \neg y)$$



Převod hradla AND: Hradlo má vstupy x, y a výstup z. Potřebujeme přidat klauzule, které nám popisují, jak se má hradlo AND chovat. Tyto vztahy přepíšeme do konjunktivní normální formy:

$$\left.\begin{array}{c} x \,\&\, y \Rightarrow z \\ \neg x \Rightarrow \neg z \\ \neg y \Rightarrow \neg z \end{array}\right\} \quad \begin{array}{c} (z \vee \neg x \vee \neg y) \\ (\neg z \vee x) \\ (\neg z \vee y) \end{array}$$



Tím v polynomiálním čase vytvoříme formuli, která je splnitelná právě tehdy, je-li splnitelný zadaný obvod. Ve splňujícím ohodnocení formule bude obsaženo jak splňující ohodnocení obvodu, tak výstupy všech hradel obvodu. □

Poznámka: Tím jsme také odpověděli na otázku, kterou jsme si kladli při zavádění SATu: tedy zda omezením na CNF o něco přijdeme. Teď už víme, že nepřijdeme – libovolná booleovská formule se dá přímočaře převést na obvod a ten zase na formuli v CNF. Zavádíme sice nové proměnné, ale nová formule je splnitelná právě tehdy, kdy ta původní.

Cvičení

1.* Dokažte lemma o vztahu mezi problémy z P a hradlovými sítěmi pomocí výpočetního modelu RAM.

19.5 Co si počít s těžkým problémem

NP-úplné problémy jsou obtížné, nicméně v životě velmi běžné. Přesněji řečeno spíš než s rozhodovacím problémem se potkáme s problémem *optimalizačním,* ve kterém jde o nalezení *nejlepšího* objektu s danou vlastností. To může být třeba největší nezávislá množina

v grafu nebo obarvení grafu nejmenším možným počtem barev. Kdybychom uměli efektivně řešit optimalizační problém, umíme samozřejmě řešit i příslušný rozhodovací, takže pokud $\mathsf{P} \neq \mathsf{NP}$, jsou i optimalizační problémy těžké.

Ale co naplat, svět nám takové úlohy předkládá a my je potřebujeme vyřešit. Naštěstí situace není zase tak beznadějná. Nabízejí se tyto možnosti, co si počít:

1. *Spokojit se s málem.* Nejsou vstupy, pro které problém potřebujeme řešit, dostatečně malé, abychom si mohli dovolit použít algoritmus s exponenciální složitostí? Zvlášť když takový algoritmus vylepšíme prořezáváním neperspektivních větví výpočtu a třeba ho i paralelizujeme.

2. *Vyřešit speciální případ.* Nemají naše vstupy nějaký speciální tvar, kterého bychom mohli využít? Grafové problémy jsou často v P třeba pro stromy nebo i obecněji pro bipartitní grafy. U číselných problémů zase někdy pomůže, jsou-li čísla na vstupu dostatečně malá.

3. *Řešení aproximovat.* Opravdu potřebujeme optimální řešení? Nestačilo by nám o kousíček horší? Často existuje polynomiální algoritmus, který nalezne nejhůře c-krát horší řešení, než je optimum, přičemž c je konstanta.

4. *Použít heuristiku.* Neumíme-li nic lepšího, můžeme sáhnout po některé z mnoha heuristických technik, které sice nic nezaručují, ale obvykle nějaké uspokojivé řešení najdou. Může pomoci třeba hladový algoritmus nebo evoluční algoritmy. Často platí, že čím déle heuristiku necháme běžet, tím lepší řešení najde.

5. *Kombinace přístupů.* Mnohdy lze předchozí přístupy kombinovat: například použít aproximační algoritmus a poté jeho výsledek ještě heuristicky vylepšovat. Tak získáme řešení, které od optima zaručeně není moc daleko, a pokud budeme mít štěstí, bude se od něj lišit jen velmi málo.

Nyní si některé z těchto technik předvedeme na konkrétních příkladech.

Největší nezávislá množina ve stromu

Ukážeme, že hledání největší nezávislé množiny je snadné, pokud graf je strom, nebo dokonce les.

Lemma: Buď T les a ℓ jeho libovolný list. Pak alespoň jedna z největších nezávislých množin obsahuje ℓ.

Důkaz: Mějme největší nezávislou množinu M, která list ℓ neobsahuje. Podívejme se na souseda p listu ℓ. Leží p v M? Pokud ne, mohli bychom do M přidat list ℓ a dostali bychom větší nezávislou množinu. V opačném případě odebereme z M souseda p a nahradíme ho listem ℓ, čímž dostaneme stejně velkou nezávislou množinu obsahující ℓ. □

Algoritmus bude přímočaře používat toto lemma. Dostane na vstupu les a najde v něm libovolný list. Tento list umístí do nezávislé množiny a jeho souseda z lesa smaže, protože se nemůže v nezávislé množině vyskytovat. Toto budeme opakovat, dokud nějaké listy zbývají. Zbylé izolované vrcholy také přidáme do nezávislé množiny.

Tento algoritmus jistě pracuje v polynomiálním čase. Šikovnou implementací můžeme složitost snížit až na lineární, například tak, že budeme udržovat seznam listů. My zde ukážeme jinou lineární implementaci založenou na prohledávání do hloubky. Bude pracovat s polem značek M, v němž na počátku bude všude *false* a postupně obdrží *true* všechny prvky hledané nezávislé množiny.

Algoritmus NzMnaVeStromu
Vstup: Strom T s kořenem v, pole značek M

1. $M[v] \leftarrow$ *true*.
2. Pokud je v list, skončíme.
3. Pro všechny syny w vrcholu v:
4. Zavoláme se rekurzivně na podstrom s kořenem w.
5. Pokud $M[w] =$ *true*, položíme $M[v] \leftarrow$ *false*.

Výstup: Pole M indikující nezávislou množinu

Barvení intervalového grafu

Mějme n přednášek s určenými časy začátku a konce. Chceme je rozvrhnout do co nejmenšího počtu poslucháren tak, aby nikdy neprobíhaly dvě přednášky naráz v jedné místnosti.

Chceme tedy obarvit co nejmenším počtem barev graf, jehož vrcholy jsou časové intervaly a dvojice intervalů je spojena hranou, pokud má neprázdný průnik. Takovým grafům se říká *intervalové* a pro jejich barvení existuje pěkný polynomiální algoritmus.

Podobně jako jsme geometrické problémy řešili zametáním roviny, zde budeme „zametat přímku bodem“, tedy procházet ji zleva doprava, a všímat si událostí, což budou začátky a konce intervalů. Pro jednoduchost předpokládejme, že všechny souřadnice začátků a konců jsou navzájem různé.

Kdykoliv interval začne, přidělíme mu barvu. Až skončí, o barvě si poznamenáme, že je momentálně volná. Dalším intervalům budeme přednostně přidělovat volné barvy. Řečeno v pseudokódu:

Algoritmus BarveníIntervalů
Vstup: Intervaly $[x_1, y_1], \ldots, [x_n, y_n]$

1. $b \leftarrow 0$ ◁ *počet zatím použitých barev*

2. $B \leftarrow \emptyset$ ◁ *které barvy jsou momentálně volné*
3. Setřídíme množinu všech x_i a y_i.
4. Procházíme všechna x_i a y_i ve vzestupném pořadí:
5. Narazíme-li na x_i:
6. Je-li $B \neq \emptyset$, odebereme jednu barvu z B a uložíme ji do c_i.
7. Jinak $b \leftarrow b + 1$ a $c_i \leftarrow b$.
8. Narazíme-li na y_i:
9. Vrátíme barvu c_i do B.

Výstup: Obarvení $c_1, \ldots, c_n$

Tento algoritmus má časovou složitost $\mathcal{O}(n \log n)$ kvůli třídění souřadnic. Samotné obarvování je lineární.

Ještě ovšem potřebujeme dokázat, že jsme použili minimální možný počet barev. Uvažujme okamžik, kdy proměnná b naposledy vzrostla. Tehdy začal interval a množina B byla prázdná, což znamená, že jsme $b - 1$ předchozích barev museli přidělit intervalům, jež začaly a dosud neskončily. Existuje tedy b různých intervalů, které mají společný bod (v grafu tvoří kliku), takže každé obarvení potřebuje alespoň b barev.

Problém batohu s malými čísly

Připomeňme si *problém batohu.* Jeho optimalizační verze vypadá takto: Je dána množina n předmětů s hmotnostmi $h_1, \ldots, h_n$ a cenami $c_1, \ldots, c_n$ a nosnost batohu H. Hledáme podmnožinu předmětů $P \subseteq \{1, \ldots, n\}$, která se vejde do batohu (tedy $h(P) \leq H$, kde $h(P) := \sum_{i \in P} h_i$) a její cena $c(P) := \sum_{i \in P} c_i$ je největší možná.

Ukážeme algoritmus, jehož časová složitost bude polynomiální v počtu předmětů n a součtu všech cen $C = \sum_i c_i$.

Použijeme dynamické programování. Představme si problém omezený na prvních k předmětů. Označme $A_k(c)$ (kde $0 \leq c \leq C$) minimum z hmotností těch podmnožin, jejichž cena je právě c; pokud žádná taková podmnožina neexistuje, položíme $A_k(c) = \infty$.

Tato A_k spočteme indukcí podle k: Pro $k = 0$ je určitě $A_0(0) = 0$ a $A_0(1) = A_0(2) = \ldots = A_0(C) = \infty$. Pokud již známe A_{k-1}, spočítáme A_k následovně: $A_k(c)$ odpovídá nějaké podmnožině předmětů z $1, \ldots, k$. V této podmnožině jsme buďto k-tý předmět nepoužili, a pak je $A_k(c) = A_{k-1}(c)$, nebo použili, a tehdy bude $A_k(c) = A_{k-1}(c - c_k) + h_k$ (to samozřejmě jen pokud $c \geq c_k$). Z těchto dvou možností si vybereme tu, která dává množinu s menší hmotností:

$$A_k(c) = \min(A_{k-1}(c), A_{k-1}(c - c_k) + h_k).$$

Přechod od A_{k-1} k A_k tedy trvá $\mathcal{O}(C)$, od A_1 až k A_n se dopočítáme v čase $\mathcal{O}(Cn)$.

Jakmile získáme A_n, známe pro každou cenu příslušnou nejlehčí podmnožinu. Maximální cena množiny, která se vejde do batohu, je tedy největší c^*, pro něž je $A_n(c^*) \leq H$. Jeho nalezení nás stojí čas $\mathcal{O}(C)$.

Zbývá zjistit, které předměty do nalezené množiny patří. Upravíme algoritmus, aby si pro každé $A_k(c)$ pamatoval ještě $B_k(c)$, což bude index posledního předmětu, který jsme do příslušné množiny přidali. Pro nalezené c^* tedy bude $i = B_n(c^*)$ poslední předmět v nalezené množině, $i' = B_{i-1}(c^* - c_i)$ ten předposlední a tak dále. Takto v čase $\mathcal{O}(n)$ rekonstruujeme celou množinu od posledního prvku k prvnímu.

Máme tedy algoritmus, který vyřeší problém batohu v čase $\mathcal{O}(nC)$. Tato funkce ovšem není polynomem ve velikosti vstupu: reprezentujeme-li vstup binárně, C může být až exponenciálně velké vzhledem k délce jeho zápisu. To je pěkný příklad tzv. *pseudopolynomiálního* algoritmu, tedy algoritmu, jehož složitost je polynomem v počtu čísel na vstupu a jejich velikosti. Pro některé NP-úplné problémy takové algoritmy existují, pro jiné (např. pro nezávislou množinu) by z jejich existence plynulo P = NP.

Problém batohu bez cen

Někdy se uvažuje též zjednodušená verze problému batohu, v níž nerozlišujeme mezi hmotnostmi a cenami. Chceme tedy do batohu naskládat nejtěžší podmnožinu, která se tam ještě vejde. Tento problém zvládneme pro malá čísla vyřešit i jiným algoritmem, opět založeným na dynamickém programování.

Indukcí podle k vytváříme množiny Z_k obsahující všechny hmotnosti menší než H, kterých nabývá nějaká podmnožina prvních k prvků. Jistě je $Z_0 = \{0\}$. Podobnou úvahou jako v předchozím algoritmu dostaneme, že každou další Z_k můžeme zapsat jako sjednocení Z_{k-1} s kopií Z_{k-1} posunutou o h_k, ignorujíce hodnoty větší než H. Nakonec ze Z_n vyčteme výsledek.

Všechny množiny přitom mají nejvýše $H + 1$ prvků, takže pokud si je budeme udržovat jako setříděné seznamy, spočítáme sjednocení sléváním v čase $\mathcal{O}(H)$ a celý algoritmus doběhne v čase $\mathcal{O}(Hn)$.

Cvičení

1. Popište polynomiální algoritmus pro hledání nejmenšího vrcholového pokrytí stromu. (To je množina vrcholů, která obsahuje alespoň jeden vrchol z každé hrany.)

2.* Nalezněte polynomiální algoritmus pro hledání nejmenšího vrcholového pokrytí bipartitního grafu.

3. Vážená verze nezávislé množiny: Vrcholy stromu mají celočíselné *váhy,* hledáme nezávislou množinu s maximálním součtem vah.

4. Ukažte, jak v polynomiálním čase najít největší nezávislou množinu v intervalovém grafu.

5.* Vyřešte v polynomiálním čase 2-SAT, tedy splnitelnost formulí zadaných v CNF, jejichž klauzule obsahují nejvýše 2 literály.

6. Problém E3,E3-SAT je dalším zesílením 3,3-SATu. Chceme zjistit splnitelnost formule v CNF, jejíž každá klauzule obsahuje právě tři různé proměnné a každá proměnná se nachází v právě třech klauzulích. Ukažte, že tento problém lze řešit efektivně z toho prostého důvodu, že každá taková formule je splnitelná.

7. Pokusíme se řešit problém dvou loupežníků hladovým algoritmem. Probíráme předměty od nejdražšího k nejlevnějšímu a každý dáme tomu loupežníkovi, který má zrovna méně. Je nalezené řešení optimální?

8. Problém tří loupežníků: Je dána množina předmětů s cenami, chceme ji rozdělit na 3 části o stejné ceně. Navrhněte pseudopolynomiální algoritmus.

19.6 Aproximační algoritmy

Neumíme-li najít přesné řešení problému, ještě není vše ztraceno: můžeme ho *aproximovat.* Co to znamená?

Optimalizační problémy obvykle vypadají tak, že mají nějakou množinu *přípustných řešení,* každé z nich ohodnoceno nějakou *cenou* $c(x)$. Mezi nimi hledáme *optimální řešení* s minimální cenou c^*. Zde si vystačíme s jeho α-*aproximací,* čili s přípustným řešením s cenou $c' \leq \alpha c^*$ pro nějakou konstantu $\alpha > 1$. To je totéž jako říci, že relativní chyba $(c' - c^*)/c^*$ nepřekročí $\alpha - 1$.

Analogicky bychom mohli studovat maximalizační problémy a chtít alespoň α-násobek optima pro $0 < \alpha < 1$.

Aproximace problému obchodního cestujícího

V *problému obchodního cestujícího* je zadán neorientovaný graf G, jehož hrany jsou ohodnoceny délkami $\ell(e) \geq 0$. Chceme nalézt nejkratší z hamiltonovských kružnic, tedy těch, které navštíví všechny vrcholy. (Obchodní cestující chce navštívit všechna města na mapě a najezdit co nejméně.)

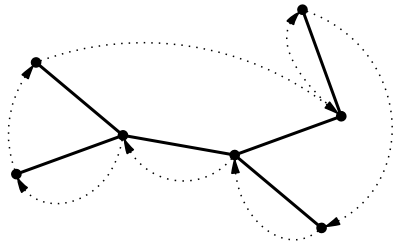
Není překvapivé, že tento problém je těžký – už sama existence hamiltonovské kružnice je NP-úplná. Nyní ukážeme, že pokud je graf úplný a platí v něm trojúhelníková nerovnost (tj. $\ell(x,z) \leq \ell(x,y) + \ell(y,z)$ pro všechny trojice vrcholů x, y, z), můžeme problém

obchodního cestujícího 2-aproximovat. To znamená najít v polynomiálním čase kružnici, která je přinejhorším dvakrát delší než ta optimální.

Grafy s trojúhelníkovou nerovností přitom nejsou nijak neobvyklé – odpovídají konečným metrickým prostorům.

Algoritmus bude snadný: Najdeme nejmenší kostru a obchodnímu cestujícímu poradíme, ať ji obejde. To můžeme popsat například tak, že kostru zakořeníme, prohledáme ji do hloubky a zaznamenáme, jak jsme procházeli hranami. Každou hranou kostry přitom projdeme dvakrát – jednou dolů, podruhé nahoru. Tím však nedostaneme kružnici, nýbrž jen nějaký uzavřený sled, protože vrcholy navštěvujeme vícekrát.

Sled tedy upravíme tak, že kdykoliv se dostává do již navštíveného vrcholu, přeskočí ho a přesune se až do nejbližšího dalšího nenavštíveného. Tak ze sledu vytvoříme hamiltonovskou kružnici a jelikož v grafu platí trojúhelníková nerovnost, celková délka nevzrostla. (Pořadí vrcholů na kružnici můžeme získat také tak, že během prohledávání budeme vypisovat vrcholy při jejich první návštěvě. Rozmyslete si, že je to totéž.)



Obrázek 19.10: Obchodní cestující obchází kostru

Věta: Nalezená kružnice není delší než dvojnásobek optima.

Důkaz: Označme T délku minimální kostry, A délku kružnice vydané naším algoritmem a O (optimum) délku nejkratší hamiltonovské kružnice. Z toho, jak jsme kružnici vytvořili, víme, že $A \leq 2T$. Platí ovšem také $T \leq O$, jelikož z každé hamiltonovské kružnice vznikne vynecháním hrany kostra a ta nemůže být kratší než minimální kostra. Složením obou nerovností získáme $A \leq 2T \leq 2O$. □

Sestrojili jsme 2-aproximační algoritmus pro problém obchodního cestujícího. Dodejme ještě, že trochu složitějším trikem lze tento problém 1.5-aproximovat a že v některých metrických prostorech (třeba v euklidovské rovině) lze v polynomiálním čase najít $(1+\varepsilon)$--aproximaci pro libovolné $\varepsilon > 0$. Ovšem čím menší je ε, tím déle algoritmus poběží.

Trojúhelníková nerovnost ovšem byla pro tento algoritmus klíčová. To není náhoda – hned dokážeme, že bez tohoto předpokladu je libovolná aproximace stejně těžká jako přesné řešení.

Věta: Pokud pro nějaké reálné $t \geq 1$ existuje polynomiální t-aproximační algoritmus pro problém obchodního cestujícího v úplném grafu (bez požadavku trojúhelníkové nerovnosti), pak je $\mathsf{P} = \mathsf{NP}$.

Důkaz: Ukážeme, že pomocí takového aproximačního algoritmu dokážeme v polynomiálním čase zjistit, zda v libovolném grafu existuje hamiltonovská kružnice, což je NP-úplný problém.

Dostali jsme graf G, ve kterém hledáme hamiltonovskou kružnici (zkráceně HK). Doplníme G na úplný graf G'. Všem původním hranám nastavíme délku na 1, těm novým na nějaké dost velké číslo c. Kolik to bude, určíme za chvíli.

Graf G' je úplný, takže v něm určitě nějaké HK existují. Ty, které se vyskytují i v původním grafu G, mají délku přesně n. Jakmile ale použijeme jedinou hranu, která z G nepochází, vzroste délka kružnice alespoň na $n - 1 + c$.

Podle délky nejkratší HK v G' tedy dokážeme rozpoznat, zda existuje HK v G. Potřebujeme to ovšem zjistit i přes zkreslení způsobené aproximací. Musí tedy platit $tn < n - 1 + c$. To snadno zajistíme volbou hodnoty c větší než $(t - 1)n + 1$.

Naše konstrukce přidala polynomiálně mnoho hran s polynomiálně velkým ohodnocením, takže graf G' je polynomiálně velký vzhledem ke G. Rozhodujeme tedy existenci HK v polynomiálním čase a $\mathsf{P} = \mathsf{NP}$. □

Podobně můžeme dokázat, že pokud $\mathsf{P} \neq \mathsf{NP}$, neexistuje pro problém obchodního cestujícího ani pseudopolynomiální algoritmus. Stačí původním hranám přiřadit délku 1 a novým délku 2.

Aproximační schéma pro problém batohu

Již víme, jak optimalizační verzi problému batohu vyřešit v čase $\mathcal{O}(nC)$, pokud jsou hmotnosti i ceny na vstupu přirozená čísla a C je součet všech cen. Jak si poradit, pokud je C obrovské? Kdybychom měli štěstí a všechny ceny byly násobky nějakého čísla p, mohli bychom je tímto číslem vydělit. Tak bychom dostali zadání s menšími čísly, jehož řešením by byla stejná množina předmětů jako u zadání původního.

Když nám štěstí přát nebude, můžeme přesto zkusit ceny vydělit a výsledky nějak zaokrouhlit. Optimální řešení nové úlohy pak sice nemusí odpovídat optimálnímu řešení té původní, ale když nastavíme parametry správně, bude alespoň jeho dobrou aproximací. Budeme se snažit relativní chybu omezit libovolným $\varepsilon > 0$.

Základní myšlenka: Označíme $c_{\max}$ maximum z cen c_i. Zvolíme nějaké přirozené číslo $M < c_{\max}$ a zobrazíme interval cen $[0, c_{\max}]$ na $\{0, \ldots, M\}$ (tedy každou cenu znásobíme poměrem $M/c_{\max}$ a zaokrouhlíme). Jak jsme tím zkreslili výsledek? Všimněme si, že efekt je stejný, jako kdybychom jednotlivé ceny zaokrouhlili na násobky čísla $c_{\max}/M$ (prvky z intervalu $[i \cdot c_{\max}/M, (i+1) \cdot c_{\max}/M)$ se zobrazí na stejný prvek). Každé c_i jsme tím tedy změnili o nejvýše $c_{\max}/M$, celkovou cenu libovolné podmnožiny předmětů pak nejvýše o $n \cdot c_{\max}/M$. Navíc odstraníme-li ze vstupu předměty, které se samy nevejdou do batohu, má optimální řešení původní úlohy cenu $c^* \geq c_{\max}$, takže chyba naší aproximace nepřesáhne $n \cdot c^*/M$. Má-li tato chyba být shora omezena $\varepsilon \cdot c^*$, musíme zvolit $M \geq n/\varepsilon$.

Na této myšlence „kvantování cen“ je založen následující algoritmus.

Algoritmus AproximaceBatohu

1. Odstraníme ze vstupu všechny předměty těžší než H.
2. Spočítáme $c_{\max} = \max_i c_i$ a zvolíme $M = \lceil n/\varepsilon \rceil$.
3. Kvantujeme ceny: Pro $i = 1, \ldots, n$ položíme $\hat{c}_i \leftarrow \lfloor c_i \cdot M/c_{\max} \rfloor$.
4. Vyřešíme dynamickým programováním problém batohu pro upravené ceny $\hat{c}_1, \ldots, \hat{c}_n$ a původní hmotnosti i kapacitu batohu.
5. Vybereme stejné předměty, jaké použilo optimální řešení kvantovaného zadání.

Rozbor algoritmu: Kroky 1–3 a 5 jistě zvládneme v čase $\mathcal{O}(n)$. Krok 4 řeší problém batohu se součtem cen $\hat{C} \leq nM = \mathcal{O}(n^2/\varepsilon)$, což stihne v čase $\mathcal{O}(n\hat{C}) = \mathcal{O}(n^3/\varepsilon)$. Zbývá dokázat, že výsledek našeho algoritmu má opravdu relativní chybu nejvýše ε.

Označme P množinu předmětů použitých v optimálním řešení původní úlohy a $c(P)$ cenu tohoto řešení. Podobně Q bude množina předmětů v optimálním řešení nakvantované úlohy a $\hat{c}(Q)$ jeho hodnota v nakvantovaných cenách. Potřebujeme odhadnout ohodnocení množiny Q v původních cenách, tedy $c(Q)$, a srovnat ho s $c(P)$.

Nejprve ukážeme, jakou cenu má optimální řešení P původní úlohy v nakvantovaných cenách:

$$\hat{c}(P) = \sum_{i \in P} \hat{c}_i = \sum_{i \in P} \left\lfloor c_i \cdot \frac{M}{c_{\max}} \right\rfloor \geq \sum_{i \in P} \left(c_i \cdot \frac{M}{c_{\max}} - 1 \right) \geq$$
$$\geq \left(\sum_{i \in P} c_i \cdot \frac{M}{c_{\max}} \right) - n = c(P) \cdot \frac{M}{c_{\max}} - n.$$

Nyní naopak spočítejme, jak dopadne optimální řešení Q nakvantovaného problému při přepočtu na původní ceny (to je výsledek našeho algoritmu):

$$c(Q) = \sum_{i \in Q} c_i \geq \sum_{i \in Q} \hat{c}_i \cdot \frac{c_{\max}}{M} = \left(\sum_i \hat{c}_i \right) \cdot \frac{c_{\max}}{M} = \hat{c}(Q) \cdot \frac{c_{\max}}{M} \geq \hat{c}(P) \cdot \frac{c_{\max}}{M}.$$

Poslední nerovnost platí proto, že $\hat{c}(Q)$ je optimální řešení kvantované úlohy, zatímco $\hat{c}(P)$ je nějaké další řešení téže úlohy, které nemůže být lepší.[4] Teď už stačí složit obě nerovnosti a dosadit za M:

$$\begin{aligned} c(Q) &\geq \left(\frac{c(P) \cdot M}{c_{\max}} - n \right) \cdot \frac{c_{\max}}{M} \geq c(P) - \frac{n \cdot c_{\max}}{n/\varepsilon} \geq c(P) - \varepsilon c_{\max} \geq \\ &\geq c(P) - \varepsilon c(P) = (1 - \varepsilon) \cdot c(P). \end{aligned}$$

Na přechodu mezi řádky jsme využili toho, že každý předmět se vejde do batohu, takže optimum musí být alespoň tak cenné jako nejcennější z předmětů.

Shrňme, co jsme dokázali:

Věta: Existuje algoritmus, který pro každé $\varepsilon > 0$ nalezne $(1 - \varepsilon)$-*aproximaci* problému batohu s n předměty v čase $\mathcal{O}(n^3/\varepsilon)$.

Dodejme ještě, že algoritmům, které dovedou pro každé $\varepsilon > 0$ najít v polynomiálním čase $(1 - \varepsilon)$-aproximaci optimálního řešení, říkáme *polynomiální aproximační schémata* (PTAS – Polynomial-Time Approximation Scheme). V našem případě je dokonce složitost polynomiální i v závislosti na $1/\varepsilon$, takže schéma je *plně polynomiální* (FPTAS – Fully Polynomial-Time Approximation Scheme).

Cvičení

1. *Problém MaxCut:* vrcholy zadaného grafu chceme rozdělit do dvou množin tak, aby mezi množinami vedlo co nejvíce hran. Jinými slovy chceme nalézt bipartitní podgraf s co nejvíce hranami. Rozhodovací verze tohoto problému je NP-úplná, optimalizační verzi zkuste v polynomiálním čase 2-aproximovat.

2.* V problému *MaxE3-SAT* dostaneme formuli v CNF, jejíž každá klauzule obsahuje právě 3 různé proměnné, a chceme nalézt ohodnocení proměnných, při němž je splněno co nejvíce klauzulí. Rozhodovací verze je NP-úplná. Ukažte, že náhodné ohodnocení proměnných splní v průměru 7/8 klauzulí. Z toho odvoďte deterministickou 7/8-aproximaci v polynomiálním čase.

[4] Zde nás zachraňuje, že ačkoliv u obou úloh leží optimum obecně jinde, obě mají stejnou množinu *přípustných řešení,* tedy těch, která se vejdou do batohu. Kdybychom místo cen kvantovali hmotnosti, nebyla by to pravda a algoritmus by nefungoval.

3. Hledejme vrcholové pokrytí následujícím hladovým algoritmem. V každém kroku vybereme vrchol nejvyššího stupně, přidáme ho do pokrytí a odstraníme ho z grafu i se všemi již pokrytými hranami. Je nalezené pokrytí nejmenší? Nebo alespoň $\mathcal{O}(1)$-aproximace nejmenšího?

4.* Uvažujme následující algoritmus pro nejmenší vrcholové pokrytí grafu. Graf projdeme do hloubky, do výstupu vložíme všechny vrcholy vzniklého DFS stromu kromě listů. Dokažte, že vznikne vrcholové pokrytí a že 2-aproximuje to nejmenší.

5.* V daném orientovaném grafu hledáme acyklický podgraf s co nejvíce hranami. Navrhněte polynomiální 2-aproximační algoritmus.

6.* Řešení problému obchodního cestujícího hrubou silou by prohledávalo graf do hloubky a zkoušelo všechny hamiltonovské kružnice. To může v grafu na n vrcholech trvat až $n!$ kroků. Pokuste se najít rychlejší algoritmus. Dynamickým programováním lze dosáhnout složitosti $\mathcal{O}(2^n \cdot n^k)$ pro konstantní k. To je sice exponenciální, ale stále mnohem lepší než faktoriál.

7. *Konvexní obchodní cestující* chce navštívit všech n vrcholů zadaného konvexního n-úhelníku v rovině. Vzdálenosti měříme v běžném euklidovském smyslu. Najděte polynomiální algoritmus.

Nápovědy k cvičením

Nápovědy k cvičením

1.1.5. Využijte toho, že $\binom{n}{k} = \binom{n}{k-1} \cdot \frac{n-k+1}{k}$.

1.2.5. Použijte metodu dvou jezdců.

1.2.9. Vymyslete, jak pro dané k ověřit, že posloupnost obsahuje všechna čísla od 1 do k. Stačí na to řádově n operací.

1.2.11. Zkuste nedělit pole na poloviny, ale podle rovnoměrného rozložení čísel odhadnout lepší místo pro dělení.

1.2.13. Zkuste algoritmus zkřížit s klasickým binárním vyhledáváním.

1.3.6. Mohou se hodit Fibonacciho čísla z oddílu 1.4.

3.2.5. Máme pole, v němž jsou za sebou dvě setříděné posloupnosti a chceme je slít do jedné, uložené v tomtéž poli. Pole rozdělte na $\Theta(\sqrt{n})$ bloků velkých $\Theta(\sqrt{n})$ a pořiďte si odkladiště na $\mathcal{O}(1)$ bloků. Pak slévejte do odkladiště a kdykoliv z jedné ze vstupních posloupností přečtěte celý blok, nahraďte ho hotovým blokem z odkladiště. Výstup tedy ukládáme napřeskáčku do bloků vstupu, tak nakonec bloky setřídíme podle jejich prvních prvků.

3.3.9. Podívejte se na levý dolní prvek matice. Co plyne z $A_{i,j} < i+j$ a co z $A_{i,j} > i+j$?

3.4.2. Spočítejte četnost jednotlivých klíčů a podle toho si naplánujte, kde ve výstupním poli bude která přihrádka.

3.4.3. Funkce $r/\log r$ je rostoucí.

3.4.4. Třiďte uspořádané dvojice (e, m) lexikograficky.

3.4.5. Sestavte množinu dvojic (i, z), které říkají, že na i-té pozici nějakého řetězce se vyskytuje znak z. Setřiďte ji lexikograficky.

3.4.6. Nalezněte minimum m a maximum M, interval $[m, M]$ rozdělte na $n+1$ přihrádek a prvky do nich rozházejte. Všimněte si, že alespoň jedna přihrádka zůstane prázdná.

4.2.3. Pokud prvních $m = 2^k - 1$ indexů obsadíme čísly 1 až m v tomto pořadí, může na indexech $m+1$ až $2m+1$ ležet libovolná permutace čísel $\{m+1, \ldots, 2m+1\}$.

4.4.1. Počítejte prefixové součty a pamatujte si jejich průběžné minimum.

4.4.6. Předpočítejte součty všech podmatic s levým horním rohem $(1, 1)$.

4.4.13. Maximální medián hledejte půlením intervalu. V každém kroku testujte, zda existuje podmatice s mediánem větším nebo rovným středu intervalu, pomocí dvojrozměrných prefixových součtů.

5.3.5. Spočítejte $(\mathbf{A}+\mathbf{E})^n$, kde $\mathbf{A}$ je matice sousednosti a $\mathbf{E}$ jednotková matice téže velikosti (tou jsme efektivně do grafu přidali smyčky). Použijte algoritmus na rychlé umocňování z oddílu 1.4.

5.8.2. Udržujte si vstupní stupně vrcholů a frontu všech vrcholů, kterým už vstupní stupeň klesl na nulu.

5.8.3. Hodí se obrátit hrany grafu a počítat naopak cesty vedoucí do u.

5.8.6. Provedeme-li BFS, graf tvořený stromovými a dopřednými hranami je DAG.

5.9.1. Opět se hodí, že každý sled je možné zjednodušit na cestu.

5.9.4. Jak u polosouvislého grafu vypadá graf komponent silné souvislosti?

5.11.3. Doplňte hrany mezi vrcholy lichého stupně.

5.11.17. Hodí se postupně odtrhávat vrcholy stupně nejvýše 5 a počítat trojúhelníky, kterých se tyto vrcholy účastní.

5.11.18. Máme co do činění s *vnějškově rovinným grafem* – to je rovinný graf, jehož všechny vrcholy leží na vnější stěně. Každý takový graf musí obsahovat alespoň jeden vrchol stupně nejvýš 2.

6.1.6. Dokud existuje nějaký dosažitelný vrchol x mimo strom, hledejte nejkratší cestu z v_0 do x a pokoušejte se ji do stromu přidat.

6.3.5. Použijte analogii invariantu **F**.

6.3.6. Dokažte, že $h(v)$ je rovno délce nějaké v_0v-cesty, a využijte toho, že takových cest je pouze konečně mnoho.

6.5.8. Graf tvořený hranami z cvičení 6.5.7 je acyklický.

6.5.10. Úlohu převeďte na hledání nejkratší cesty v grafu, jehož vrcholy odpovídají proměnným a hrany nerovnicím. Všimněte si, že pokud jsou z vrcholu v dosažitelné všechny ostatní vrcholy, pak pro každou hranu xy platí $d(v,y)-d(v,x) \leq \ell(x,y)$.

7.2.2. Zvolte nějakou minimální kostru a vyčkejte na první okamžik, kdy se od ní algoritmus odchýlí.

7.4.1. Nechť T je kostra nalezená algoritmem a T' nějaká lehčí. Uspořádejte hrany obou koster podle vah a najděte první místo, kde se liší. Co algoritmus udělal, když tuto hranu potkal?

8.1.5. Nejprve pomocí rotací doprava strom přeskládejte na cestu. Potom ukažte, jak cestu délky 2^k-1 rotacemi přetvarovat na dokonale vyvážený strom (v tomto případě úplný binární). Nakonec domyslete, co si počít pro obecnou délku cesty.

8.1.13. Vytvořte cyklus z ukazatelů přes vrchol a jeho dva syny.

8.3.8. Odřízněte všechny podstromy ležící vpravo od cesty z kořene do x a pak je pospojujte operací Join z předchozího cvičení.

9.2.3. Rozšiřte dvojkovou soustavu o číslici -1.

9.2.5. Rozdělte přičtení čísla k na přímé zpracování jeho $\log k$ číslic, které zaplatíme přímo, a případné další přenosy, na které postačí uložené penízky.

9.3.2. Pamatujte si minimum, náhradu za minimum, náhradu za tuto náhradu, a tak dále.

9.3.3. Rozdělte vstup na bloky velikosti $k/2$ a počítejte pro ně prefixové a suffixové součty. Výpočty těchto součtů vhodně rozdělte mezi jednotlivé operace.

9.3.4. Postupně přidávejte prvky a pamatujte si, který vrchol stromu odpovídá zatím poslednímu prvku.

9.3.7. Napovíme potenciál pro $(2,4)$-stromy: je to součet příspěvků vrcholů. Vrcholy s 0 až 4 klíči přispějí po řadě 2, 1, 0, 2, 4. Přitom 0 a 4 klíče se objevují pouze dočasně během štěpení a slučování.

10.1.1. Libovolný algoritmus, který by největší disk přenesl vícekrát, je nutně pomalejší než ten náš.

10.1.2. Opět uvažte, jak se pohybuje největší disk.

10.1.3. Kolik je korektních rozmístění?

10.1.5. Použijeme stejnou posloupnost tahů jako u rekurzivního algoritmu. Pokud pořadové číslo tahu zapíšeme ve dvojkové soustavě, počet nul na konci čísla nám prozradí, který disk se má pohnout. Když si sloupy uspořádáme cyklicky, bude se každý disk pohybovat buďto vždy po směru hodinových ručiček, nebo naopak vždy proti směru.

10.3.5. Lineární kombinaci není těžké najít zkusmo, ale existuje i obecný postup. Bude se nám hodit *Lagrangeova interpolační formule* z cvičení 17.1.2. Nyní se stačí na zadaná čísla podívat jako na polynomy: pokud označíme $f(t) = X_2t^2+X_1t^1+X_0$, $g(t) = Y_2t^2+Y_1t+Y_0$, bude platit $X = f(10^n)$, $Y = g(10^n)$, a tedy $XY = h(10^n)$, kde $h = f \cdot g$. Naše mezivýsledky W_i ovšem nejsou ničím jiným než hodnotami $h(0)$, $h(1)$, $h(-1)$, $h(2)$, $h(-2)$. Interpolační formule pak ukazuje, jak $h(10^n)$ spočítat jako lineární kombinaci těchto hodnot.

10.3.10. Zadané číslo rozdělte na horních a dolních $n/2$ cifer, každé převeďte zvlášť a potom násobte číslem $z^{n/2}$ zapsaným v nové soustavě a sčítejte.

10.4.2. Nechť $T(n) = a \cdot T(n/b+k) + \Theta(n^c)$. Podproblémy na i-té hladině budou velké maximálně $n/b^i+k+k/b+k/b^2+\ldots = n/b^i+kb/(b-1)$. Rekurzi ovšem musíme zastavit už pro $n = \lceil kb/(b-1)\rceil$, jinak by se nám algoritmus mohl zacyklit.

10.4.3. Uvažte číslo q, které je řešením rovnice $\beta_1^q + \ldots + \beta_a^q = 1$, a dokažte, že strom rekurze má řádově n^q listů.

10.7.3. Úsek, který nám zůstal v ruce při odkládání i-té nejstarší položky na zásobníku, má nejvýše $n/2^i$ prvků.

10.8.2. Pětice nechť jsou pouze myšlené, do i-té pětice patří prvky $i, k+i, 2k+i, 3k+i, 4k+i$, kde $k = \lfloor n/5 \rfloor$. Medián pětice vždy prohoďte s jejím prvním prvkem, takže mediány pětic budou tvořit souvislý úsek. Zbytek je podobný Quicksortu na místě.

10.9.4. Inverzní matice je opět trojúhelníková. Bloky, rekurze.

11.1.5. Jak z k-tice vybrané rovnoměrně náhodně z $n-1$ hodnot získat k-tici vybranou rovnoměrně náhodně z n hodnot?

11.1.6. Průběh algoritmu rozdělte na fáze, i-tá fáze končí umístěním čísla i. Jaký je střední počet pokusů v i-té fázi?

11.2.2. Využijte toho, že $\sum_{i=1}^{n} i \ln i \leq \int_1^{n+1} x \ln x \, \mathrm{d}x$.

11.2.5. i-tý krok trvá $\mathcal{O}(n/i)$.

11.3.3. Využijte toho, že $1 + \alpha \leq \mathrm{e}^{\alpha}$ pro každé $\alpha \in \mathbb{R}$.

11.4.1. Úspěšné hledání prvku projde tytéž přihrádky, jaké jsme prošli při jeho vkládání. Průměrujte přes všechny hledané prvky a použije lemma o harmonických číslech ze strany 266.

12.2.6. Použijte binární vyhledávání a předchozí cvičení.

12.3.2. Stačí si vždy pamatovat dva sousední řádky tabulky.

13.3.5. Hrany vedoucí z vrcholu si místo v poli pamatujte v nějaké slovníkové datové struktuře.

14.2.2. Najděte číslo $\tau > 0$, pro které posloupnost $a_n = \tau^n$ splňuje rekurenci $a_{n+2} = a_n - a_{n+1}$. Donuťte Fordův-Fulkersonův algoritmus k tomu, aby se rezervy na vybraných hranách vyvíjely podle této posloupnosti. Jelikož všechny prvky posloupnosti jsou nenulové, algoritmus se nikdy nezastaví.

15.1.5. Spočítejte, z kolika hradel může do výstupu sítě vést cesta délky k.

15.1.8. Shora odhadněte počet všech booleovských obvodů s $\mathcal{O}(n^k)$ hradly a ukažte, že pro dost velké n je to méně než počet n-vstupových booleovských funkcí.

15.1.12. Formule má stromovou strukturu a v každém stromu se nachází vrchol, jehož odebráním vzniknou komponenty nejvýše poloviční velikosti. Použijte myšlenku skládání bloků z oddílu 15.2.

15.2.4. Podobně jako jsme u sčítačky předpovídali přenosy, zde můžeme předpovídat stav automatu pro jednotlivé pozice ve vstupu.

15.2.5. Jak vypadají mocniny matice sousednosti?

16.1.6. Hledejte konvexní obal bodů ležících na parabole $y = x^2$.

16.2.4. Ke kvadratickému řešení postačí přímočaré zametání, pro zrychlení na $\mathcal{O}(n \log n)$ se inspirujte cvičeními 8.2.5 a 8.3.1 z kapitoly o stromech.

17.1.4. Dokažte, že $\det \mathbf{V} = \prod_{0 \le i < j \le n} (x_j - x_i)$.

17.1.5. Nechť odpalovací kód K je číslo z nějakého konečného tělesa $\mathbb{Z}_p$. Pro $n = 2$ zvolíme náhodné $x \in \mathbb{Z}_p$ a položíme $y = K - x$. Pro $n > 2$ náhodně zvolíme vhodný polynom nad $\mathbb{Z}_p$.

17.3.7. Použijte cvičení 17.2.1.

17.4.2. Využijte toho, že $\mathrm{e}^{\mathbf{i}x} = \cos x + \mathbf{i} \sin x$ a $\mathrm{e}^{-x} = \cos x - \mathbf{i} \sin x$.

17.4.3. Využijte toho, že $\omega^k + \omega^{-k} = 2 \cos(2k\pi/n)$.

17.4.4. Rozepište $\mathcal{F}^{-1}(\mathcal{F}(\mathbf{x}) \odot \mathcal{F}(\mathbf{y}))$ podle definice.

17.4.8. Transformujte nejdříve řádky a pak sloupce.

19.3.1. Například převodem z 3D-párování.

19.3.7. Převodem z $\mathbf{A}\mathbf{x} = \mathbf{1}$.

19.3.8. Převodem ze součtu podmnožiny.

19.5.2. Vzpomeňte si na síť z algoritmu na největší párování. Jak v ní vypadají řezy?

19.5.3. Pro každý podstrom spočítejte dvě maximální váhy nezávislé množiny: jednu pro případ, kdy kořen v množině leží, druhou když neleží.

19.5.5. Na každou klauzuli se můžeme podívat jako na implikaci.

19.5.6. Použijte Hallovu větu.

19.6.1. Použijte hladový algoritmus.

19.6.2. Linearita střední hodnoty.

19.6.4. Najděte v G párování obsahující alespoň tolik hran, kolik je polovina počtu vrcholů vráceného pokrytí. Jak velikost párování souvisí s velikostí nejmenšího vrcholového pokrytí?

19.6.5. Libovolné očíslování vrcholů rozdělí hrany na „dopředné“ a „zpětné“.

19.6.6. Stavy dynamického programování budou odpovídat trojicím (U, x, y), kde U je množina vrcholů a x a y její prvky. Pro každý z nich spočítáme, jaká je nejkratší cesta z x do y, která navštíví všechny vrcholy v U.

19.6.7. Použijte dynamické programování.

Rejstřík

Rejstřík

Symboly

Číslice

A

Literatura

Literatura

[1] Arora, Sanjeev a Barak, Boaz. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009. ISBN 978-0521424264.

[2] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L. a Stein, Clifford. *Introduction to Algorithms*. MIT Press, 3. vydání, 2009. ISBN 978-0262033848.

[3] Dasgupta, Sanjoy, Papadimitriou, Christos a Vazirani, Umesh. *Algorithms*. McGraw-Hill Education, 2006. ISBN 978-0073523408.

[4] Demel, Jiří. *Grafy a jejich aplikace*. Vydáno vlastním nákladem, 2. vydání, 2015. ISBN 978-80-260-7684-1.

[5] Graham, Ronald L., Knuth, Donald E. a Patashnik, Oren. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Professional, 2. vydání, 1994. ISBN 978-0201558029.

[6] Kleinberg, Jon a Tardos, Éva. *Algorithm Design*. Pearson, 2005. ISBN 978-0321295354.

[7] Laaksonen, Antti. *Competitive Programmer's Handbook*. Preprint, 2017. Dostupné online na *https://cses.fi/book.html*.

[8] Mareš, Martin. *Krajinou grafových algoritmů*, *ITI Series*, svazek 330. Institut teoretické informatiky MFF UK Praha, 2007. ISBN 978-80-239-9049-2. Dostupné online na *http://mj.ucw.cz/vyuka/ga/*.

[9] Matoušek, Jiří a Nešetřil, Jaroslav. *Kapitoly z diskrétní matematiky*. Karolinum, 2010. ISBN 978-80-246-1740-4.

[10] Skiena, Steven S. *The Algorithm Design Manual*. Springer, 2. vydání, 2008. ISBN 978-1848000698.

[11] Töpfer, Pavel. *Algoritmy a programovací techniky*. Prometheus, 1995. ISBN 80-85849-83-6.

PRŮVODCE LABYRINTEM ALGORITMŮ
Martin Mareš, Tomáš Valla

Vydavatel:
CZ.NIC, z. s. p. o.
Milešovská 5, 130 00 Praha 3
Edice CZ.NIC
www.nic.cz

1. vydání, Praha 2017
Zapracována errata 2021-08-26.
Kniha vyšla jako 15. publikace v Edici CZ.NIC.

ISBN 978-80-88168-19-5 (tištěná verze)
ISBN 978-80-88168-20-1 (ve formátu EPUB)
ISBN 978-80-88168-21-8 (ve formátu MOBI)
ISBN 978-80-88168-22-5 (ve formátu PDF)

cz.nic | SPRÁVCE DOMÉNY CZ

Edice CZ.NIC

O knize Chceme-li napsat počítačový program, obvykle začínáme algoritmem – popisem řešení úlohy pomocí řady elementárních kroků srozumitelných počítači. Často bývá nalezení vhodného algoritmu důležitější než detaily programu. Tato kniha vypráví o tom, jak algoritmy navrhovat a jak jejich chování zkoumat. Mimo to obsahuje mnoho příkladů algoritmů a datových struktur s aplikacemi a cvičeními. Je určena každému, kdo už umí trochu programovat v jakémkoliv jazyce a chtěl by se naučit algoritmicky myslet. Hodit se může jak studentovi informatiky, tak zkušenému programátorovi z praxe.

O autorech **Martin Mareš** se dlouhodobě pohybuje po křivolaké hranici mezi teoretickou a praktickou informatikou. Na Matematicko-fyzikální fakultě Univerzity Karlovy se zabývá návrhem a teoretickou analýzou algoritmů a také jejich chováním na reálných počítačích. Především ovšem o algoritmech rád vypráví. **Tomáš Valla** se zabývá teoretickou informatikou, diskrétní matematikou a teorií her. To však neznamená, že také rád neprogramuje. Působí na Fakultě informačních technologií ČVUT, kde přednáší a pracuje ve výše zmíněných oblastech. Do práce na Průvodci se pustil proto, že má rád elegantní poznatky a jejich elegantní výklad.

O edici Edice CZ.NIC je jedním z osvětových projektů správce české domény nejvyšší úrovně. Cílem tohoto projektu je vydávat odborné, ale i populární publikace spojené s Internetem a jeho technologiemi. Kromě tištěných verzí vychází v této edici současně i elektronická podoba knih. Ty je možné najít na stránkách knihy.nic.cz.

knihy.nic.cz

ISBN 978-80-88168-22-5

9 788088 168225 >